# SCALABLE PARALLEL PROCESSING OF MULTI-OBJECTIVE OPTIMIZED DNA SEQUENCE ASSEMBLY

by

MUNIB AHMED

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
December 2011

ACKNOWLEDGEMENTS

First of all, I want to thank my parents, Mian Nisar Ahmed and Abida Begum, for providing me with the opportunity and assistance to continue with my quest for higher learning. My father, in particular, always encouraged me and has been a source of inspiration and a role model from the very beginning. Also, I very much appreciate the understanding and support extended by my wife Saira and children, Adil, Nawal, Hasan, and Zara, who had to sacrifice family time and numerous fun activities while I studied hard for the examinations or the research work.

On the academic front, I owe many thanks to my advisor Dr. Ishfaq Ahmad who has been very instrumental in steering me in the right direction, ensuring that I stayed focused and strived for the best quality of work. I would not have been able to reach this point without his kind guidance.

I would also like to thank my committee members, Dr. Weems, Dr. Elmasri, and Dr. Huang for their time and advice they offered to me. Dr. Weems and Dr. Elmasri also mentored and provided an opportunity for me to learn and research during my Master's degree at UTA.

Finally, I thank my sister and brothers along with all those sincere friends who have wished, inspired and helped me in different ways to stay on course during this arduous yet rewarding journey.

November 10, 2011

ABSTRACT

SCALABLE PARALLEL PROCESSING OF MULTI-OBJECTIVE

OPTIMIZED DNA SEQUENCE ASSEMBLY

Munib Ahmed, PhD

The University of Texas at Arlington, 2011

Supervising Professor: Ishfaq Ahmad

Bioinformatics is an emerging branch of science where issues pertaining to molecular biology are evaluated and resolved by leveraging the techniques and algorithms devised in the field of computer science. Most of these issues are due to the enormous amount of data and the computational complexity involved in generating expeditious and qualitatively viable solutions. This poses a challenge to the algorithm developers who must strive to achieve multiple conflicting objectives of processing very large dataset with the highest accuracy possible while keeping the execution time to a minimum. Genome assembly is one such problem in bioinformatics where a DNA sequence is reconstructed using millions of small fragments of DNA that are produced in the laboratory as a result of sequencing process. When examined purely as data, these fragments are small in size ($< 10^3$ characters long) but large in numbers, have repetitive regions which exacerbates the complexity of the reconstruction algorithms, and contain erroneous data due to imperfect laboratory procedures. This dissertation takes a holistic approach to resolve these issues by first presenting a comprehensive study of contemporary work, highlighting its strengths and weaknesses while proposing improvements wherever needed, followed by the design and implementation of a new parallel framework. With the extra processing power available in a parallel computing

environment, this framework enhances accuracy of the solution by correcting errors in the low quality data regions and improves the speedup by dynamically balancing the load among multiple processors and by utilizing innovative data structures along with a hashing technique that require lesser memory compared to other contemporary programs. One of the chief objectives of this work is to carve out an important and sizeable piece of the DNA sequence assembly process, device a new parallel algorithm, and provide its modular implementation in order to facilitate the scalability analysis and parametric study of various characteristics and interdependencies of multiple conflicting objectives such as speedup, accuracy, and data size. A comparison between experimental and theoretical statistics of the system explains similarities or deviations and their causes and effects. This research work and the underlying approach can be easily extended to other related areas of bioinformatics, including multiple sequence alignment and phylogenetics, using parallel computing.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

# CHAPTER 1

# INTRODUCTION

"It is the mark of an educated mind to be able to
entertain a thought without accepting it."
Aristotle

The field of bioinformatics has seen phenomenal advances since the early 1950s when the structural composition of DNA was first laid out [97]. Several hundred gigabytes of genomic data has been sequenced, deciphered, annotated, and stored. Genome assembly has recently been an area of active research and gathered more steam after the Human Genome project was announced [57], [69], [94]. In a perfect world, the sequencing process would have read and deciphered the genetic code as a one-step process. However, the limit imposed by the sequencing technology, which allows reading only a few hundreds of DNA bases at a time, renders the reconstruction of full genome very difficult. The objective is to assemble the DNA structure from a set of numerous overlapping, repeating, and somewhat inaccurate fragments. These fragments are usually denoted by strings of characters from an alphabet set $\partial$ = {A, T, C, G} representing four bases Adenine, Thymine, Cytosine, and Guanine, respectively. The large volumes of data and the inaccuracies of the sequencing process demand intensive computations and complex algorithms. Often, the genome assembly of a medium to large size DNA takes days, if not weeks, to complete a multi-phase process in which small fragments are filtered, sorted, joined, aligned, and oriented to yield much longer pieces of genome that can be mapped, validated and annotated using information from other laboratory sources. Such lengthy process does not lend itself very well to a quick what-if analysis that is often required to study

the specie under evaluation. Therefore, the need for the high performance computing arises requiring a new set of parallel algorithms that can speed up the entire process. Also, distributed environments provide pool of resources that allow processing of very large datasets which would otherwise be infeasible.

There have been a large number of genomes assembled and many more are still in the pipeline. Consequently, there have been a number of full scale assemblers as well as different modules reported that aid in the process. As the volume of data involved in this process is extraordinarily large, requiring significantly long runtimes, some of the reported work has leveraged parallel computing to achieve faster speeds. Genome assembly in the realm of parallel and distributed computing is a logical area of interdisciplinary study and a focal point of this dissertation. Furthermore, from a bioinformatician's perspective, the whole genome shotgun sequencing (WGSS), a specific methodology to sequence and reconstruct the DNA, offers a unique opportunity to device parallel algorithms that can provide fast and efficient means to reconstruct the DNA.

In response to the special needs of genome assembly process as stated above, many researchers have designed and implemented sophisticated parallel algorithms that can be used both academically as well as commercially enabling a faster reconstruction and analysis of the genome being studied. Although many of such programs, including assemblers and individual modules to perform various tasks, e.g., pairwise alignment, multiple sequence alignment, and repeat finding etc., have been analyzed and documented, there has not been a study conducted to take a holistic view of the entire WGSS process and the associated algorithms in the context of high performance computing (HPC). It is also very important to evaluate such algorithms and analyze their scalability to allow their selection based on different characteristics that should be prioritized by the users.

This dissertation is organized as follows. In chapter 2, we will take a deeper look at the entire assembly process from a computer scientist's perspective while providing enough

background of different laboratory procedures to help the readership understand and appreciate the complexities of one of the most important subjects in the field of bioinformatics. We will survey the work that has been done, point out areas of weaknesses and strengths of contemporary work, propose several enhancements, and provide insight of the upcoming trends and methodologies in this area. Our focus will remain on the high performance computing, parallel and distributed paradigms in particular.

Chapter 3 provides a detailed discussion of two most widely used approaches for Whole Genomes Shotgun Sequencing. A detailed scalability analysis of both approaches is presented and the key performance measures are evaluated and compared in the context of parallel and distributed computing.

In chapter 4, we present a new parallel algorithm aimed at improving the speedup and enhancing the accuracy of some of the key stages of the assembly process. We provide a detailed account of our implementation and discuss the results. We formulate a mathematical model, comprising a set of objective functions to be optimized, to be used for a detailed scalability analysis of the new algorithm. This sheds light on various aspects of the system and the interrelationships among multiple, and often conflicting, objectives. Such analysis is very useful as it allows bioinformaticians to select one system over others based upon their priorities of different decision variables. Also, our approach can be easily extended on to many other related areas in bioinformatics for similar analysis.

## 1.1 The Big Picture

Figure 1.1 illustrates various stages of the assembly process including the flow of the control and data. The detailed discussion of each of these stages is provided in Chapter 2 though subsequent chapters may also reference this figure. These stages can be categorized in three broader groups. The first, comprising of module 1 through 4, as shown in the figure, is a series of procedures mostly performed in the laboratories that, with the help from some

computer programs, provide the essential data to be processed. The second group consists of module 5 through 13 that are primarily software-based and process the input data producing the reconstructed sequences, which are as close to the original genome as possible. Finally, the third category, called Finishing, involves certain degree of manual validation as well as different laboratory-based steps performed to validate the results before the final sequences are officially released. In the next chapter, we will take a closer look at each of these stages in the realm of high performance computing.

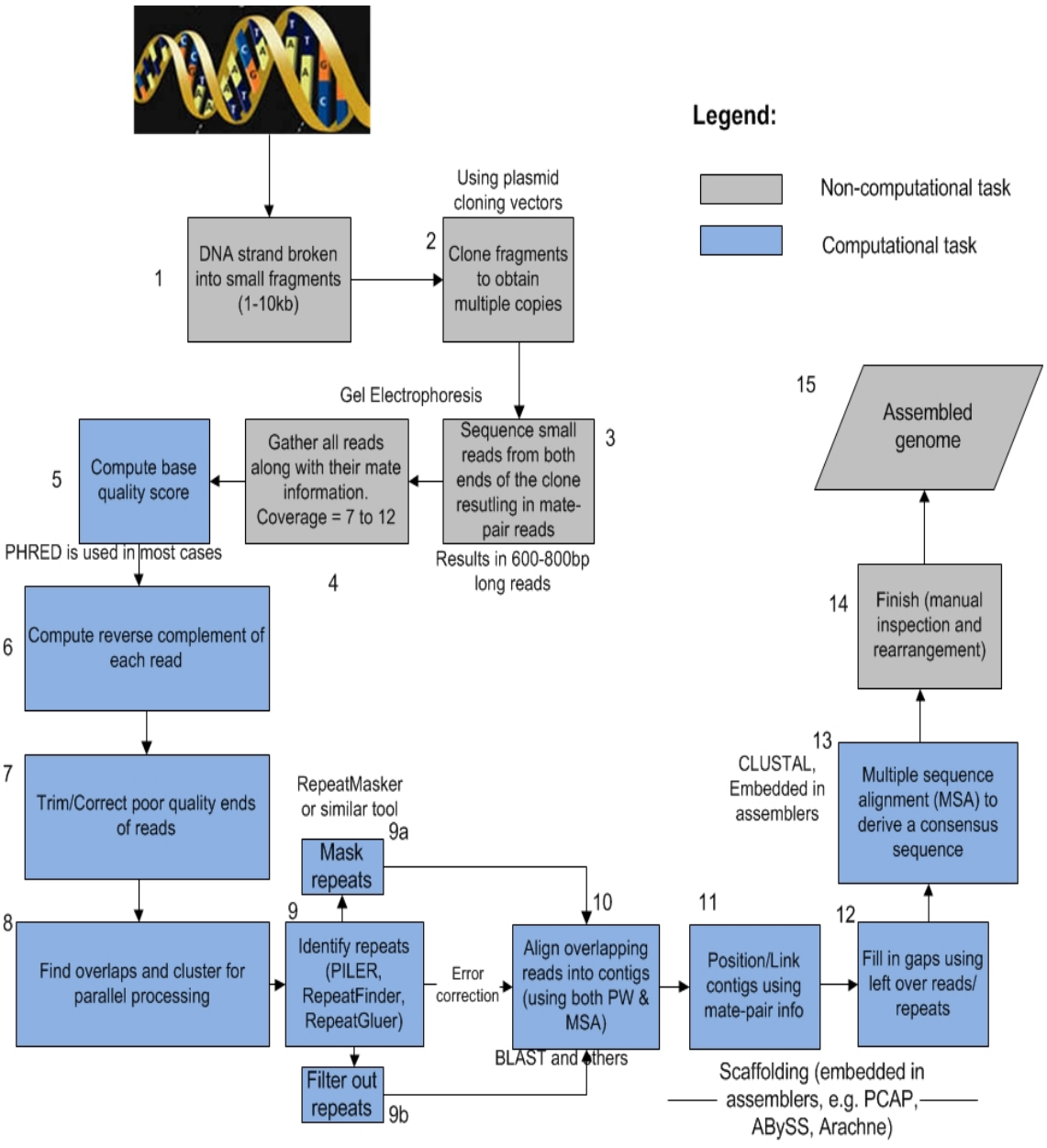


Figure 1.1 A block diagram illustrating a typical WGSS assembly process.

# CHAPTER 2

# A SURVEY OF SEQUENCE ASSEMBLY TECHNIQUES AND ALGORITHMS USING HIGH PERFORMANCE COMPUTING

"Believe you can and you are halfway there."

Theodore Roosevelt

This part of the dissertation summarizes our study of contemporary work in the area of DNA sequence assembly using parallel and distributed computing. We streamline all the individual tasks related to WGSS into different consecutive stages in order to present it as a coherent, though loosely coupled, process where one stage consumes the output of previous stage and passes its results to the next one to be processed. While we walk our readership through the whole streamlined process, we also discuss, analyze, and, in some areas, propose enhancements to the algorithms and techniques that have been designed and implemented for each of these stages.

This chapter is organized as follows. The sections 1 and 2 delve into the biologist's world to provide some background information on laboratory procedures with sufficient details for the readers to understand the overall process and several key issues associated with it along with the type of data obtained in the laboratories. Section 3 discusses the need and ways to detect overlaps among data. In section 4, a detailed account of repetitive regions, called *repeats*, and the methods to identify and process them are discussed. Once the overlaps have been detected and repeats have been identified, the next phase of the assembly process is to

find pairwise alignments, which is explained in section 5. As the pairwise alignments result in longer and contiguous segments compared to the original input fragments, the next step is to align multiple sequences at the same time to obtain even more completeness and correctness with reference to the original genomic sequence. It is complicated and an active area of research in genomic studies, as discussed in section 6 of this chapter. The section 7 provides the methods and issues involved in scaffolding. In section 8, a brief introduction to the final steps of finishing is provided as it is outside the scope of this work. Finally, section 9 discusses various measures and criteria to assess the quality and accuracy of the reconstructed genome to validate the results. At the end of the chapter, we also discuss some emerging technologies in this area and study their pros and cons and the effects on the existing methodologies.

## 2.1 Sequence Extraction

This section describes various stages of DNA sequencing in detail, from the cloning of the DNA to obtaining the sequences and the corresponding base quality scores. The readings and results collected during these stages provide the necessary data required for the assembly process. Most of these tasks involve laboratory procedures and therefore do not require an in-depth analysis for the purpose of this survey. Also, it should be noted that this study is focused mainly on the genome assembly using shotgun sequencing methodology which is described below.

### *2.1.1 Shotgun Sequencing*

The whole genome shotgun sequencing (WGSS) is a process of breaking up a DNA molecule into smaller pieces so it can be read. The WGSS process typically results in millions of fragments, each consisting of $10^3$ or fewer bases, referred to as “reads”. These reads carry forward very little detail on how to join them back together to create the complete DNA blueprint of the specie being studied. These procedures are less than perfect and prone to errors,

resulting in regions of poor quality within reads, especially at the both ends, further adding to the difficulty in assembling these fragments. There are special programs, e.g. Phred [26], which evaluate the reads and assign a quality score to each base to quantify its accuracy or reliability in that position. We will describe these laboratory based processes at a high level in the following sections.

### *2.1.2 Identification And Preparation of Clone*

The first and foremost step in the assembly process is to identify and obtain a copy of the genome to be studied. The DNA strands are broken into smaller pieces for a special replication process termed as “cloning”. Here, a piece of sequence is embedded into a host DNA cell which, through a natural process of cell division called mitosis, produces large quantities of the subject DNA segment resulting in multiple copies of the clone. Since the subsequent process is prone to errors, this multiplicity, often referred to as coverage, provides a redundancy that is necessary to join all the segments together exploiting the overlaps among them.

### *2.1.3 Purifying DNA Clones*

The clone is separated from the host sequence by extracting small segments called “reads”, preferably in opposite directions noting the orientation and the distance between each pair of reads. This distance is called a “mate-pair distance” and provides an important piece of information used in the later stages of assembly. The computer aided assembly uses this information to correctly place a given segment with reference to its corresponding mate segment. After extracting these pieces, a special process called Gel electrophoresis is used to read the actual sequencing of bases in the laboratory [89] as illustrated in Figure 2.1.

*2.1.4 Electrophoresis*

One of the most extensively used methods to detect and isolate DNA fragments is known as gel electrophoresis. A mixture of DNA fragments is placed into an agarose gel, which is an uncharged polysaccharide that allows freedom of movement of molecules within it. The fragments are placed next to the cathode of an electric field. Once the field is activated, the fragments travel towards the anode of the field. Distinct fragments of different sizes migrate at different speeds, and thus separate into bands. Using a stain, such as Ethidium bromide, the bands fluoresce under UV light. The fragments can then be handily isolated and scanned by computer equipment. This process is used in Sanger's method of sequencing [83]. However, in the past decade, capillary electrophoresis began to replace even automated versions of gel electrophoresis equipment. The human genome was sequenced entirely by high-throughput sequencers based on arrays of capillary electrophoresis [20]. This design of electrophoresis has vastly improved the speed and efficiency of sequencing DNA fragments. The time needed to operate capillary electrophoresis equipment is also reduced significantly. Due to modern advances in such instrumentation, sequencing a prokaryote only takes weeks and perhaps less than a year for a vertebrate animal.

The capillaries range from 150-350 micrometers in diameter and about 30 centimeters in length. They are composed of fused silica, similar to what is used for fiber optics. One end of the capillary is dipped into a solution, a brief current is applied, and DNA fragments migrate to the tip of the capillary. The electric field is then reactivated so that the DNA fragments travel through the capillary. A fluorescence detector records DNA signatures in four different spectral channels. This equipment has also been specialized with the use of lasers to excite fluorescence signals and optical upgrades around the capillaries to improve clarity. With automated capillary array electrophoresis machines, fragments can be detected continuously over twenty-four hours without intervention. Readouts are collected with colored peaks represent different sizes of the DNA fragments based on their detected fluorescence [35]. A

genome center with 200 capillary machines can output 150,000 reads within a day, i.e. a sequence of a mammal, around 3 billion base pairs, can be assembled in about two years [35].

*2.1.5 Assigning Base Quality Scores*

The extracted sequences must be validated after they have been read and translated into strings of characters representing bases from the set $\partial$ = {A, T, C, G}. As the laboratory procedure reaches its limits to how quickly and efficiently such validation can be performed, the computer algorithms provide the best available option to scan the sequences and, based on chromatogram and related data points collected during electrophoresis, assign a quality score to each base. This quality score represents the accuracy of a given base in a particular position. The most commonly used program to compute quality scores is called Phred [26]. Phred expresses the quality score using a logarithmic relationship with the base-calling error probabilities. This function can be formulated as $P = 10^{-Q/10}$ where P represents probability of error and Q is the quality score for the base. For example, a quality score of 10 would represent a 1 in 10 probability of error, i.e. an accuracy of 90%, and a score of 20 translates into a 1 in 100 probability of error, a 99% accurate score and so on. Normally, the scores assessed for sequences are in the range of 10 to 50 with 10 treated as a low quality score and 50 as the best quality. Phred uses a multi-stage approach by scanning the traces and analyzing the curves representing bases and their strengths in their respective positions. There are other software tools [1] that have been used for computing the quality scores but very little work is reported [54] employing high performance computing and even that is related to the next generation sequencing, which will be discussed towards the end of this survey, and is not directly related to the shotgun sequencing. The fact that traces must be scanned and analyzed in phases and that the process does not involve large and multiple copies of the data means that base-calling does not lend itself very well to a distributed computing paradigm. Moreover, this survey provides a

high level review of laboratory and preparatory work that feeds the software based assemblers and focuses more on the assembly algorithms as presented in the later sections.

## 2.2 Data Preparation

In this section, we discuss the data preparation before the software based assembly begins. This data set includes the sequenced reads, their corresponding quality scores, mate-pair distance, and orientation information as explained in the next section. Figure 2.1 illustrates the process flow and the data produced by some procedures that are carried out in the laboratory prior to the assembly process. All such data is fed to the computer programs, called assemblers, for further processing.

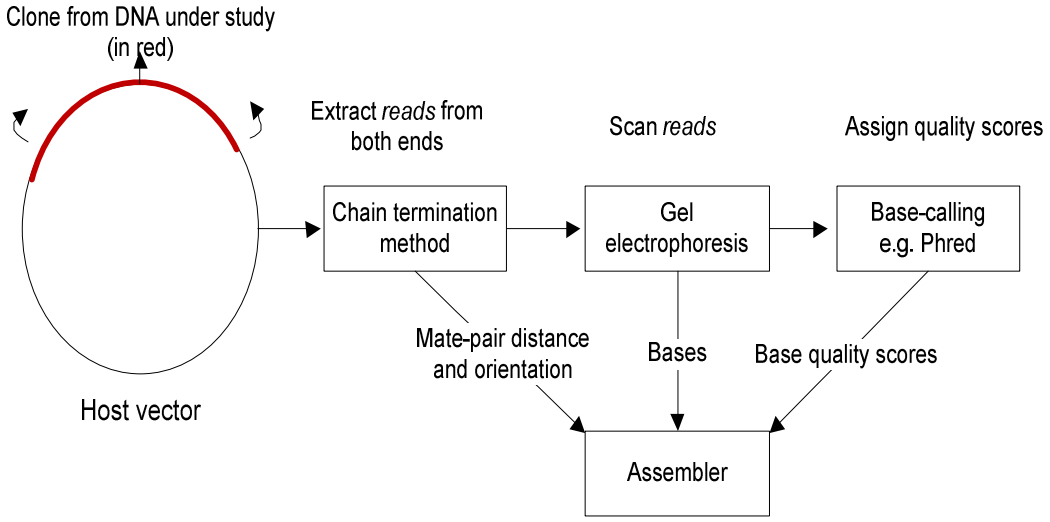


Figure 2.1 Different phases of sequence extraction performed in the laboratory to produce the data for computer-assisted assembly.

### *2.2.1 Computing Reverse Complements*

A DNA molecule consists of two equal but complementary polymers made of four types of nucleotides represented by A, C, T, & G. Each of these two polymers, also called strands,

has a foundation of sugar and phosphate and a unique direction that runs anti-parallel to each other. The following example shows how two strands can be attached to each other.

Strand 1: A A T C C G G T G T C A → 5'-3'

| | | | | | | | | | | |

Strand 2: T T A G G C C A C A G T ← 3'-5'

Due to the chemical structure, one strand is called 5-prime or, in terms of orientation, 5" to 3" whereas the other and opposite one is denoted as 3-prime or 3" to 5". Each base in a given strand is joined with an opposite base on the other strand. An *A* always pairs with *T* and *C* is always connected to a *G*. Hence a reverse complement S' of a sequence S can be shown by the following example.

S = A C T T T G A C C C

S' = G G G T C A A A G T

As the example shows, a reverse complement of a sequence is like its mirror image. It is written in reverse direction and A is replaced with T, C with G, and vice versa. In the context of WGSS, it is difficult to ascertain for a given sequence whether it came from a 5 or a 3 prime strand. Some assemblers, e.g. [74], assume that the given set of sequences cover both directions while others calculate reverse complements at run time [10], [42]. It should be noted that a calculated reverse complement also inherits a mirror image of quality scores of the original sequence. Adding these to the mix, the problem size grows two-fold and there may be some redundancy, which can be identified along with other repeated regions in the input data.

### *2.2.2 Handling Weak Bases*

The sequencing process is inherently error prone and the reads obtained using laboratory procedures [83] differ in quality. A more detailed account can be found elsewhere [89]. As explained earlier, each base of the sequenced read can be given a quality score using special programs, e.g. Phred, and as expected, the end regions of the read are usually of lower

quality compared to the rest of the read. It raises a question as to whether the low quality region of a read should be used and processed during the assembly or filtered out to avoid an erroneous assembly. In this section, we will review how various implementations treat these "weak" or low quality regions and how that impacts the overall efficiency and accuracy of the assembly algorithms.

Most assemblers, including Arachne [10], CAP3 [42], and PCAP [42], avoid processing the weak regions of the reads and use a threshold value, 9 in case of PCAP, to trim the bases with quality score less than the threshold from either end of the read. Usually, a criterion such as a predefined number X is also used such that X consecutive low quality bases determine the beginning of a weak region. There are, however, some programs like Phusion [68] that leave the low quality bases in the mix and rely upon some value provided by processing full length reads compared with trimmed reads. It is argued that trimming part of the original read may render it non-alignable and therefore it is sometimes better to use the read in its entirety. In a distributed computing environment, processing and correcting such low quality region is even more appealing as it also increases the utilization of each processor [4]. There are times during parallel processing when a processor has to wait and synchronize with other processors before proceeding with the next steps. A well designed algorithm can effectively use such idle time which can not only provide extra functionality, but also enhance the utilization of the resources resulting in an improved efficiency of the entire system.

#### 2.2.2.1 A Simple Approach

Arachne uses a simple yet effective scheme of trimming the poor quality ends of a read. First, it identifies a high quality section, i.e. a maximum contiguous segment in a read, such that the total error rate of that segment is less than 5%. It further trims that segment, if needed, such that no base of quality score less than 10 is within 12 bases of either end. Finally, if the trimmed segment size is less than 50 then the read is fully discarded. This process is illustrated in the

following example. Many other assemblers use similar or even simpler approaches where poor quality ends are trimmed off whenever the number of bases with quality having less than a certain threshold value exceeds a predefined limit.

*Example*

ACCT**GGTGTGAACCAACGAATTTATCATCGTCTTCCAAGTCGTA***GTGTTAACCGGG*$\underline{G^{9}}$*GG*AA AGTG

The above example illustrates a sample read with high quality section showing in boldface. The 12-letters following the high quality section would have been part of it but were trimmed off because of the underlined G which had a quality score = 9 < 10 ($G^{9}$ means a base G with a quality score of 9)

2.2.2.2 A More Complex Approach

PCAP [42] uses a rather elaborate method to determine the poor quality regions that should be trimmed off. There are several differences between this method and the one used by Arachne and other assemblers. Here, trimming of a read *f* takes place after overlaps have been determined since the trimming method requires information about other reads that overlap with *f*. A high-quality region of *f* is defined as a region with the maximum quality score which is the sum of differences between each base quality score within such region and a cutoff value, which is typically set to 12. Furthermore, if there are any other reads overlapping the read *f* then a clipping position is determined by locating the first occurrence of an optimal overlap where an optimal overlap is defined as the minimum of a predefined number denoted by cdep, with default value of 3, and the maximum number of overlaps possible within the high-quality region. This approach is illustrated in Figure 2.2. As noted previously, the trimming has to be delayed to find the overlaps first and hence the poor quality ends of the reads are part of the data when overlaps are calculated. That may adversely affect the specificity of the overlaps due to possibly erroneous data that is mostly found in these poor quality regions. Although part of the overall

parallel computing algorithm, it does not provide any details specific to that aspect nor discusses any potential benefits of performing this in parallel vs. sequential.

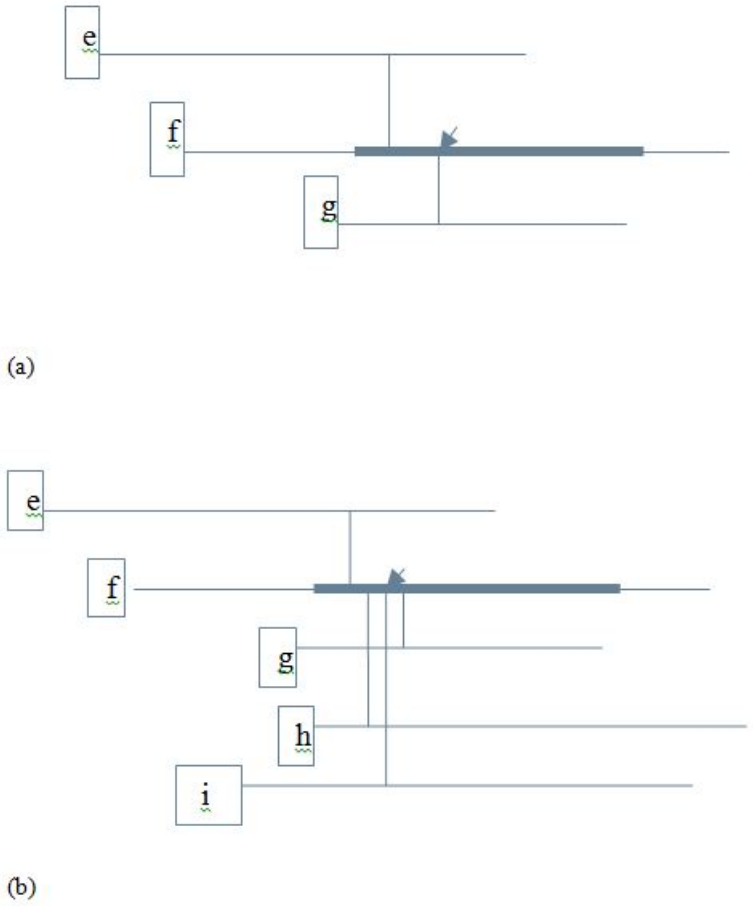


Figure 2.2 Read *f* is shown with thick line representing the high-quality region. The vertical lines mark the beginning of the overlap between two reads. The arrow shows the clipping position when cdep is set to 3. (a) Maximum overlap is 2 and therefore clipping position is where 2 overlaps are first encountered. (b) Maximum overlap is 4 but cdep parameter is 3; therefore, clipping occurs where first overlap involving three reads is found.

A different approach presented by [4] uses a distributed computing model and attempts to correct assembly errors induced by the poor quality of end regions. In this approach, each processor keeps track of “good” and “weak” regions and the occurrence of each base in a specific position. In the process of detecting potential overlaps, it assigns a reliability score to each base value in a given position within the weak regions and chooses the value with the

highest score. The overall algorithm is efficient and yields high processor utilization; however, it does not guarantee a perfect error correction.

## 2.3 Overlap Detection

After all the data needed to assemble the genome has been collected, the process begins with a tedious task of comparing segments with each other to find overlaps among them. Since the dataset includes several copies of the segments, whose starting and ending sequences are usually different, it is imperative that the algorithm is capable of identifying these overlaps. Once these overlapping segments are positioned and lined up, longer contiguous segments, sometimes known as contigs, are formed. The rest of the assembly process builds upon this phase and therefore it wouldn't be incorrect to say that overlap detection is the foundation of the assembly process. Any errors introduced in this phase will skew the final results. It is also a candidate for distributed computing as the work can be divided among multiple processors. Following is a comparative analysis of different parallel algorithms and techniques that have been implemented to detect overlaps.

### *2.3.1 Generic Approach*

In general, most schemes divide each input read into smaller sub-reads or words, sometimes called k-mers, which are *k* bases long followed by a comparison of all such k-mers with each other. Each input sequence $S_i$ is broken into a set $K_i$ of smaller subsequences of length *k*, called k-mers, such that $K_i = \{S_i[x][x+k-1]\} \forall\ x,\ 1 \leq x < |S_i|-k+1$. All k-mers along with their origin information are stored in a sorted array as illustrated in Figure 2.3. An overlap between a suffix of one and a prefix of another sequence is noted and can be readily observed in such a sorted array. Unique overlaps are further verified using a local alignment algorithm like Smith-Waterman [88] or one of its several variations. For *n* sequences yielding *m* k-mers per sequence, a sequential algorithm to generate and sort k-mers and then align promising

sequences takes O($nm$ + $nm$log($nm$) + α$n$) where α denotes the product of genome coverage $c$ (how many copies of a sequence are possibly present in dataset). and the square of average length of a sequence, to account for potential alignments for each of $n$ sequences. A parallel implementation on $p$ processors of the same should take $pT_s$ + $nT_t$ + $nm/p$ + ($nm/p$) log($nm/p$) + α$n/p$ where $T_s$ is the start time and $T_t$ is the transfer time of a sequence between two processors [4]. It should be noted that $T_s$ and $T_t$ are very small compared to actual processing times.

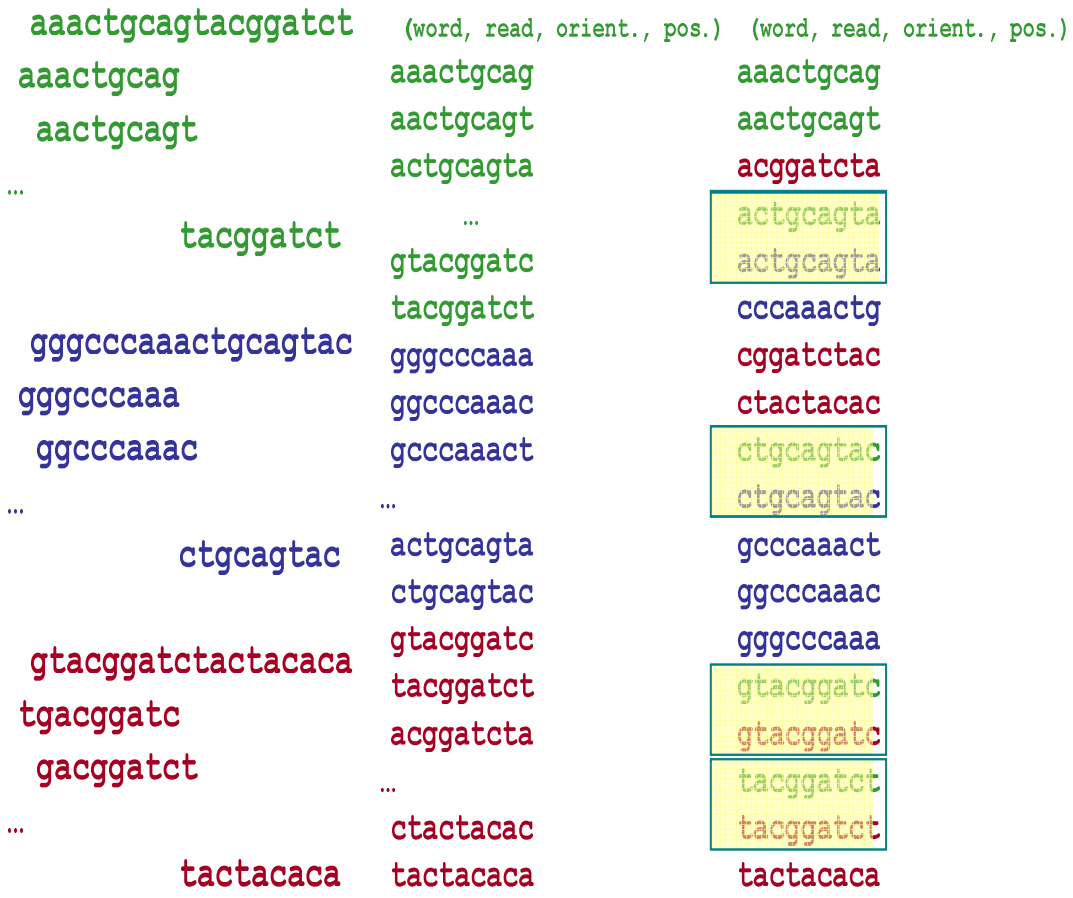


Figure 2.3 A sorted array of all the $k$ long words extracted from the original reads. The shaded portion shows common words among 2 or more reads indicating a possible overlap.

*2.3.2 More Complex Methods*

Huang et al. [42] designed a parallel assembler called PCAP that uses an approach to divide the input dataset *S*, using scattered partitioning, into *t* subsets where each subset $S_i$ is compared with the complete set *S* by a processor. It also adds the reverse complements of each read into the mix, i.e. for every read $f_x$, it computes a reverse complement $r_x$ and adds it to the input dataset. As it is possible to have a read $f_x$ compared with another read $f_y$ twice, PCAP avoids such redundant comparisons by implementing a logic which ensures that the comparison occurs only in one order, i.e. x<y and that $f_x$ is part of *S* whereas $f_y$ is part of $S_i$. These comparisons provide additional statistics, e.g. the occurrence count which shows how many times a base appears in a specific position in the input dataset. Such information helps identify the repeat regions as discussed in the following sections. PCAP detects overlaps and, using the occurrence count maintained in a special array structures called coverage arrays (see Table 2.1), identifies and filters out the repeat regions before making further comparisons for overlap detection.

The process goes back and forth between overlap detection and repeat identification/masking which further helps reduce the processing time as unnecessary comparisons are avoided by filtering out the repeats that would have to be filtered out sooner or later. As for finding and confirming the overlaps, the PCAP uses a variation of the generic approach presented above. It builds an ordered list of k-mers called “words”. The possible overlaps are detected using a similar model as explained in the previous section. Once a list of potentially overlapping reads has been prepared, the program uses a variety of algorithms to extend word matches [6] and to perform string concatenations and matching [42] using banded Smith-Waterman algorithm [16].

Table 2.1 A Coverage Array For Three Sample Reads R1, R2, and R3

| $R_1$ | $R_2$ | $R_3$ | Count |
|---|---|---|---|
| A | C | T | 0 |
| A | A | T | 2 |
| T | T | T | 3 |
| C | A | A | 2 |
| A | A | A | 3 |
| … | … | … | … |

*2.3.3 Graph Theoretical Solutions*

The two most widely used assembly approaches are “Overlap-layout-consensus” [10], [42] and “Euler Superpath” [74]. The aforementioned approaches tend to reduce the problem into graphs such that a consensus sequence can be inferred by traversing either all of the nodes or all of the edges exactly once, resulting in a Hamiltonian or an Euler path, respectively. The subsequent text provides an in-depth discussion of the two approaches and their parallel implementations. It should be noted that a few other approaches, e.g. Genetic algorithms, e.g. [101], have also been devised but we will leave those out of this discussion.

2.3.3.1 General Approach

In most cases the layout is a directed graph *G* with sequences laid out as vertices and the overlaps between two sequences as the edges as depicted in Figure 2.4. Once a directed graph has been constructed, the redundant edges are removed using transitive property. That is, an edge $u \rightarrow v$ between *u* and *v* is removed if two other edges $u \rightarrow z$ and $z \rightarrow v$ are found and the sum of non-overlapping prefix lengths of *u* and *z* in $u \rightarrow z$ and $z \rightarrow v$, respectively, is equal to the non-overlapping prefix of *u* in $u \rightarrow v$. The reduced graph is to be traversed to find a Hamiltonian path, which is an NP-Complete problem. In practice, it is unlikely to find such a path from the perspective of both data and computation but the heuristic algorithms result in sub-paths that are recorded as contigs. In terms of time complexity, assuming a heuristic algorithm is employed, a scan of *nm* entries constructs a basic graph and then removing the redundant

arcs can be done in asymptotically $O(n^2m^2)$ time whereas a parallel implementation can achieve the same in asymptotically $O(n^2m^2/p + \log p)$ time [3] when ignoring latency during merge operations among processors.

A parallel algorithm based on the above approach, designed and implemented by Blazewicz et al. [14], introduces a notion of overlapped length as well as a few more parameters, e.g. shift identifying the number of bases after which the suffix of first read begins that matches with the prefix of the second read. Below is a summary of their approach.

1) Identify overlaps & create an overlap graph
2) Define a window, of size *k*, to break sequences into k-mers
3) Define a hash function to compute a non-negative "characteristic number" for each window.
4) Store hashed value along with sequence id, orientation, position
5) Define a minimum overlap, "mo", required to assume an overlap

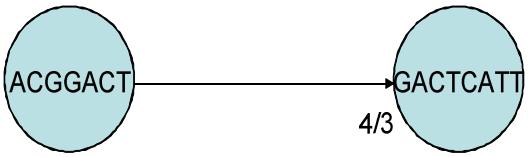


Figure 2.4 An overlap between two reads ACGGACT and GACTCATT shown as a subgraph with two nodes representing the reads and the arrow representing the overlap. The suffix of first read overlaps, with mo=4, with the prefix of the second read.

6) For all windows of all sequences, find out which sequences overlap
7) Two sequences *U* and *V* overlap directionally when one or more consecutive windows of *U* starting at position "i" match with first and more windows of *V*
8) The overlap length "ol" is equal or greater than "mo" (minimum overlap). The length of prefix of U before overlap begins is called "shift". For example, ACGGACT and GACTCATT overlap with a shift *s*=3.

9) Construct an edge u→ol/sv between vertices u and v when such an overlap is found between sequences u and v with a shift value of *s*.

10) Drop any subsequences (fully contained in other sequences) found in the process. Those already created may be deleted at the end of this stage.

11) Reduces the unnecessary edges, which can be built indirectly using other edges as follows (see Figure 2.5):

12) Delete an edge u→sw iff there exist two (for simplicity use two only) other edges v →pu and u →qw such that s=p+q. This is a transitive edge.

13) Assign a reliability score (score of deleted edge) to the remaining two edges as the transitive edge confirmed that v→pu and u →qw were not just accidental.

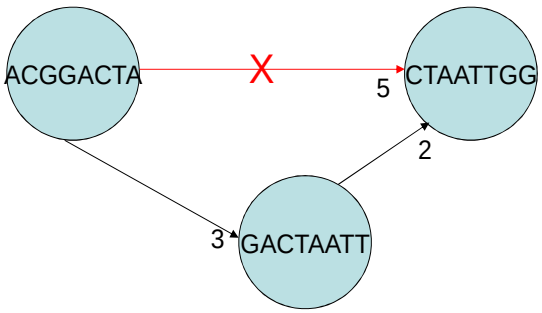


Figure 2.5 Reducing unnecessary (transitive) edges. The edges should be considered for deletion in the decreasing order of shifts to avoid disposal of those that could help deleting other edges.

2.3.3.2 Euler Subpath

The Euler path based approach was originally proposed in [74]. This approach reduces the genome assembly problem into an Eulerian path problem that lends itself to a polynomial computational complexity compared to the NP-completeness of the Hamiltonian path seen in the overlap-layout-consensus approach. The central idea of this algorithm is based on sequencing by hybridization where smaller reads are obtained leveraging a faster and cheaper

sequencing process. It must be noted that the same concept could also be applied to the relatively larger sized reads obtained through Sanger sequencing [83]. For example, a *k*-mer can be further split into *l*-mers such that suffix of $l_i$ overlaps with the prefix of $l_{i+1}$ for a length of (*l*-1) $\forall$ *i*, *l*: $0<i\leq k-l+1$, $l<k$. Using all such *l*-mers, a directed deBruijn graph [77] is constructed with the suffix and prefix as vertices and the *l*-mer as the edge connecting the two vertices. Ideally, a path traversing all such edges, called Eulerian path, would include all *l*-mers yielding a fully reconstructed genome. However, it is a set of paths which is usually obtained representing multiple contigs due to the presence of repeats and inaccurate data. The algorithm also attempts to identify and remove repeats by finding directed paths where the indegree of the starting vertex and the outdegree of the ending vertex are both greater than 1 whereas all intermediate vertices have both indegree and outdegree equal to 1. Using the relation and position of all of the *l*-mers in such path, within the original reads, the algorithm distinguishes the false overlaps, i.e. repeats from the true overlaps.

A parallel implementation of the algorithm has been reported in [86]. Another implementation of a parallel assembler [85] is based upon the similar concept, i.e. sequencing by hybridization, and builds a distributed deBruijn graph to find overlaps and to build contigs. Following are various stages of such a generic algorithm employing graph theory based on ideas presented by [74].

*(a) Generating k-mers*

Considering each *k*-mer generation as a single operation, the time taken to generate all *k*-mers is proportional to the product of *n* and *m*. The parallel implementation will asymptotically take O(*nm*/*p*) time.

*(b) Distributing k-mers*

This is an extra step in the parallel implementation. However, using a hashing scheme, the procedure takes near linear time, i.e. O(*nm*). In the parallel version, each processor keeps

some *k*-mers to itself and distributes the others based on the hash value. This may result in some communication overhead but the time complexity in this stage is asymptotically equal to O(*nm*/*p*).

*(c) Preparing Data*

The algorithm requires the multiplicities of *k*-mers to be computed for later use as a boundary condition when traversing paths. In the sequential algorithm, which is achieved by using suffix arrays to store and use *k*-mers, it takes O(*nm*log(*nm*)) time. In parallel environment, the multiplicity of each *k*-mer can be calculated as a side step when hashing and distributing the *k*-mers incurring insignificant time. If the coverage *c* is not available, then it takes asymptotically O(*nm*) to compute the coverage. This can be achieved using some fast string matching algorithms, such as the Gusfield's Z-algorithm presented in [38].

*(d) Building deBruijn Graph*

This task requires the bulk of computing power and time. Hashing may help find the adjacent *k*-mers faster; however, quadratic time is required to compare all *k*-mers with each other. A slight improvement would be to use the fact that each one of the *n* sequences was broken into a total of *m* k-mers. Therefore, those *nm* edges could be constructed without even comparing with each other. That results in *nm*(*nm*-1) or asymptotically O($n^2m^2$) runtime complexity. Once the vertices have been designated and edges drawn, the resulting graph (or sub-graphs) provide a complete list of overlaps that can be used to merge and/or align segments. As we will learn later on, this particular approach, using Euler method, offers a unique way to align sequences by traversing the graph like an Eulerian path. Therefore, this method is less computationally intensive compared to the overlap-layout-consensus approach discussed earlier which uses Hamiltonian path to align sequences.

2.3.3.3 PaCE: Parallel Clustering of ESTs

Expressed sequence tags (ESTs) are small pieces of DNA sequence (usually 200 to 500 nucleotides long) that are generated by sequencing either one or both ends of an expressed gene, thereby filtering out what is sometimes referred to as “junk DNA”. Kalyanaraman et al. [52] presented a time and space efficient algorithm and its implementation called “PaCE” that identifies a set of disjoint clusters of reads which overlap directly or transitively only with others reads within same cluster. Such clusters are then processed independently by a sequential assembler CAP3 [42] thereby exploiting parallelism due to independent and disjoint datasets. Here we will take a brief look at what ESTs are in order to better understand the algorithm used in PaCE which could also be generalized and used for sequence assembly.

***Example***

Original DNA

ACTGATGG***CCAATTA***ACCGTGCT***GACCGTA***AACGT……..

After splicing, the intron regions (in *italics*) are filtered out

ACTGATGG ACCGTGCT AACGT……..

The actual segments are usually as long as few thousand base pairs (kbp) in size. These filtered segments are then sequenced to get small reads which are typically 400 to 500 long due to the limits imposed by the chemical processes like gel electrophoresis which can only handle a few hundreds of bases at a time. These small reads are called ESTs.

ACTGA……TGG…… ACCGTGCT….. AACGT…….

The assembly of all the ESTs helps lay out the expressed genes for a given genome.

We will now summarize the general logic, proposed and leveraged by PaCE for overlap detection. The readership is encouraged to reference the original paper for more detailed discussion. The input data is distributed among available processors. Each processor scans through the input segments and, using a window of length *w*, builds buckets of segments where

prefixes of length *w* are identical. For example, for *w*=4, each processor builds one or more sorted buckets from a set of fragments (see Figure 2.6 and Figure 2.7).

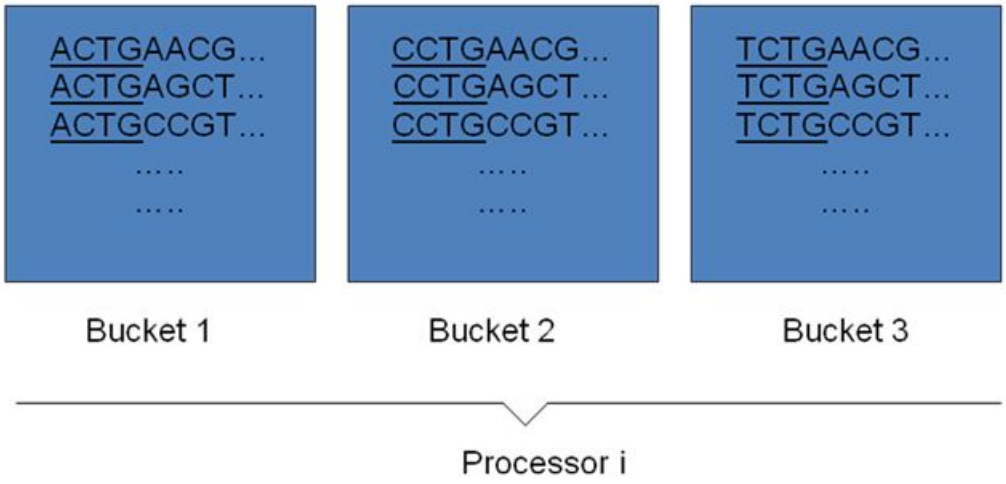


Figure 2.6 Partitioning of segments into buckets of length *w.*

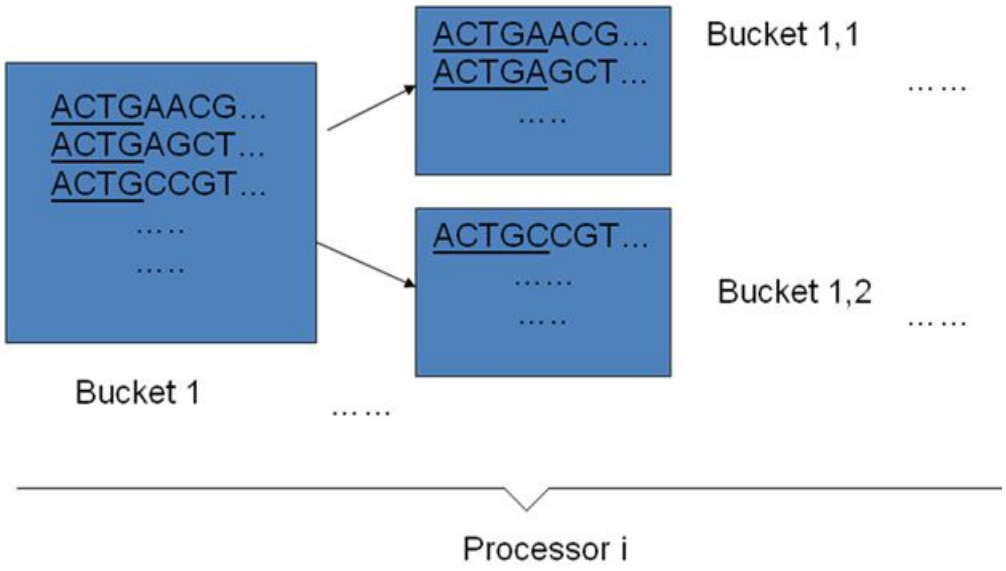


Figure 2.7 Further partitioning of buckets based on (*w*+1)th position.

1. This process is used recursively resulting in a suffix tree where the nodes are sorted in a decreasing order of path label length. The tree is then traversed in that order and the following processing takes place at each node *u*.
2. If two suffixes can be left-extended, i.e. there overlap could be extended to the left of the suffix then all such fragments having these suffixes on and under the node “u” are put into one LSet.
3. Generate only those pairs of fragments that are in two different LSets.
4. Being in same LSet signifies that the maximal length condition has yet not been met. The maximal length is defined as the length of match where it cannot be further extended to left or right.
5. Once the distributed suffix trees for all fragments have been generated, each processor generates promising pairs from its local data.
6. These pairs are reported to Master processor, which then assigns alignment task to the available processors. The result of alignment may require merging two clusters.

## 2.4 Repeats Identification

In this section, we will discuss various techniques used to detect repetitive regions within genome. Although most of these techniques/algorithms and the programs based upon them are sequential, some employ logic that can be easily parallelized or optimized to run on a high performance computing environment. First, we will explain what repeats are and why they pose issues that can be termed as the most challenging problems in the realm of sequence assembly.

### *2.4.1 The Repeats*

Consider a genome, a very long DNA sequence over the alphabet $\partial$ = {A, C, T, G}. Genomic repeats are repetitive regions in the DNA sequence that range from a simple 3 to 5

bases long repeats to more complex “interspersed repeats”. For example, a short section of a DNA sequence, e.g. ACTGCAGCAGACAGCAG, contains four repetitions of “CAG” in it. The purpose or function of repeats is not fully understood though many studies have been conducted showing some diseases’ link to a particular repetitive sequence in DNA, e.g. the above example of CAG subsequence when repeated in excess of certain number in a particular region of DNA, gene “IT15”, leads to Huntington disease. The repetitive regions may exist anywhere in the DNA and make up a large part of most of the large size genomes. For example, about 50% of the 3 billion bases long human genome consists of repeats. Repeats come in different sizes, patterns, and lengths. Some repeats are contiguous (tandem) while others may be interspersed. Some repeats may even overlap with other repeats. Some are transposable, which can even move to a different position within the genome.

*2.4.2 Problems Posed By The Repeats*

In order to appreciate the issue of repeats, it is important to take a look at the genome sequencing process at a high level. As we have discussed above, the genome sequencing includes breaking up the DNA strand in small size segments as the existing technology is capable of reading only 600-800 bases long segments of the DNA at a time. The sequencing process does not carry much detail of the whereabouts of such segments, i.e. location within DNA that they were extracted from. The number of segments may run in tens of millions. Once all segments have been read, the work begins to assemble them back to reconstruct the full DNA code. It is similar to having a very large jigsaw puzzle with millions of pieces that need to be put back together.

The challenge stems from the fact that not much detail of the individual segments is available to help assemble the genome. One way to resolve this issue is to sequence multiple copies of the DNA at the same time resulting in multiple copies of each segment with somewhat different start and end positions. This allows devising algorithms to find overlaps among

segments to enable reconstruction of the whole genome sequence. However, presence of repeats in the segments complicates the issue. The repeat regions, especially the ones longer than the average length of segments, can be mistaken as overlaps resulting in incorrect assembly. This can be related to the example of jigsaw puzzle again, except that now there are millions of pieces that have only blue sky background.

Repetitive sequences make up a significant fraction of almost every genome and an important and still open question in bioinformatics is how to represent all repeats in DNA sequences. In a pioneering paper [9], Bao and Eddy wrote, "The problem of automated repeat sequence family classification is inherently messy and ill-defined and does not appear to be amenable to a clean algorithmic attack." One of the difficulties in repeat classification is that many repeats represent mosaics of sub-repeats. Different combinations of sub-repeats form different repeat copies making it difficult to delineate the boundaries of sub-repeats (repeat boundary problem) and to represent the overall repeat structure. It is therefore very important to understand the characteristics of repeats and categorize them based on their size, pattern, and/or location in the genome.

### *2.4.3 Methods For Finding Repeats*

There are several repeat finding programs available for both academic and commercial use. For example, RepeatFinder [96], RepeatMasker [87], PILER [24] and others [75] employ sophisticated logic to detect and list different types of repetitive regions within given sequences. Here, we will take a closer look at different algorithms devised and implemented for repeat identification in the realm of high performance computing. In most cases, these algorithms are part of overall genome assembly process especially when detecting overlaps and computing alignments as discussed below.

#### 2.4.3.1 Using Coverage Arrays

Table 2.1 showed a coverage array which is essentially a table that stores a count of each base position in which it is repeated in a given set of sequences. PCAP [42] uses such arrays to find overlaps. However, as a by-product of the algorithm, repeats are identified wherever the count exceeds a threshold value set by the program. The coverage array of a read is an integer array of the same length in which the value at a position of the array is the number of overlaps between the read and other reads that cover the position. A region of a read is repetitive if the values at every position of the corresponding region of the coverage array are greater than a repeat coverage cutoff threshold. PCAP filters all such regions out and exclude them from further processing that involves more overlap detection and alignments.

As explained in earlier sections, PCAP divides the data into subsets that are then distributed evenly to all the available processors. Each subset is compared against the full set which is read from the secondary storage due to its very large size. During this comparison, the coverage arrays are created. Since each such comparison involves the full set of data including the subsets that were distributed to other processors, there are no partial repetitive regions identified by any given processor and hence there is no need to merge the results or share among processors for this particular phase of assembly.

#### 2.4.3.2 Graph theoretic algorithms

Although not a parallel approach in its original form, Pevzner et al. presented a different algorithm for sequence assembly which deviated from long established “overlap-layout-consensus” approach and used a special type of graph called de Bruijn graph that allows connecting fixed-length substrings of a given set of strings as discussed in previous section [74]. The graph is then traversed for validating overlaps. Similar to the concept of coverage arrays where coverage of a specific position exceeding a threshold value suggests a repeat, a set of edges participating in multiple sub-graphs indicate a potential repetitive region. The

algorithm formalizes the logic to determine repeats as follows. For a de Bruijn sub-graph with *n* vertices, a path $v_1 \ldots v_n$ is called a repeat if indegree ($v_1$) > 1, outdegree($v_n$) > 1, and indegree ($v_1$) = outdegree($v_i$) = 1 for $1 \leq i \leq n - 1$. Edges entering the vertex $v_1$ are called entrances into a repeat, whereas edges leaving the vertex $v_n$ are called exits from a repeat. An Eulerian path visits a repeat a few times, and every such visit defines a pairing between an entrance and an exit. Determining repetitive subsequences using this rule can be parallelized by finding all such sub-graphs in distributed fashion and merging those together later on. It should be noted, however, that the time complexity of graph based solutions in parallel computing tends to be a bit higher [49], i.e. $O(\log^2 N)$.

2.4.3.3 Parallelization of PILER

Some repeat finding algorithms lend themselves well to a distributed computing model compared to others. In this section, we propose a parallel implementation of one of the latest repeat finding algorithms called PILER [24]. Following is a high level scheme without many details of data structures or processing that would be required for an implementation. Please note that in order to avoid any confusion, $P_i$ is used here as $i^{th}$ processor and $Pile_i$ is used for $i^{th}$ pile..

*(a) Creation of Piles*

PILER assumes that another program PALS or BLAST (or a similar program to calculate all possible pair-wise local alignments between various regions of the input sequence) has already produced a list of local alignments $A_1$, $A_2$, $A_3$, …$A_N$ where *N* is the number of such alignments calculated. Each $A_i$ is obtained by aligning two regions Q and T, also called partner images of each other, of the input sequence. PILER calls each such alignment $A_i$ a “hit” and stores the start and end position of each of the two regions Q & T that were alignment to create

a given hit. It should be noted here that regions *Q* and *T* are in order, i.e. region *Q* is followed (not necessarily contiguously) by region *T*.

The main algorithm centers around building piles where each pile is a list of local alignments between pairs of regions in the input sequence such that all alignments in any given pile overlap with each other directly or transitively. Figure 2.8 illustrates how the alignments $A_1$ thru $A_4$ are in one pile $Pile_1$ while $A_5$ and $A_6$ are stored in another pile $Pile_2$. At present, PILER uses a sequential algorithm to scan through all the alignments and the constituent regions $Q_i$ , for $1 \leq i < 2N$, and as long as it overlaps with the previous alignment it goes into the current pile. For example, in Figure 2.8, all the regions $Q_i$ that contributed in alignments $A_1$ thru $A_4$ go with pile $P_1$. This sequential algorithm runs in O(*N*) time.

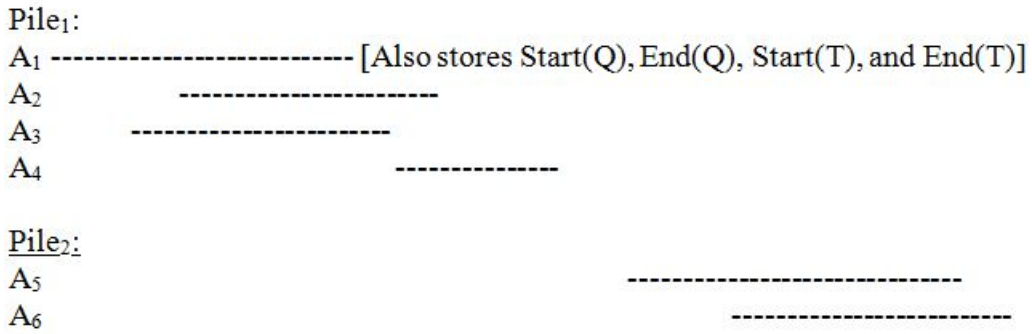


Figure 2.8 Two disjoint piles.

*(b) Parallel Creation of Piles Using p Processors*

Once all the piles have been created, different methods can be designed to derive relationships among regions that also represent repeats in the input sequence. Let us first suggest how to parallelize the algorithm for creating piles. Since all *N* hits (alignments) and their constituent regions are scanned in order of their positions within the sequence, the hits can be allocated to *p* processors such that each processor $P_i$ is assigned *N*/*p* hits using block decomposition, i.e. from *i***N*/*p* to (*i*+1)**N*/*p*. Each processor finds the overlapping hits from

among those assigned to it and stores them into local pile(s). It is quite possible for a processor $P_i$ to detect multiple piles. Each processor maintains such multiple piles along with two pointers F(i) and L(i) pointing to the first and the last piles stored locally. Once the piles have been created in parallel, each processor $P_{i>0}$ requests the preceding processors $P_{i-1}$ for the L(i-1), i.e. pointer to the last pile of $P_{i-1}$. This is needed to check if there is an overlap between the last pile of $P_{i-1}$ and the first pile of $P_i$. If so, $P_i$ receives and merges that last pile from $P_{i-1}$ into its first pile and $P_{i-1}$ removes its last pile. This can be further optimized by detecting which pile is smaller than the other and then sending that over to be merged with the larger pile. Finally, the master processor receives all the piles (pointers only in a shared memory PRAM environment or sequence positions for all regions in the piles in case of message passing architecture).

*(c) Performance*

In terms of measuring the performance, like any parallel algorithm, the proposed logic should run in O(*N*/*p*) time. It promises to be highly scalable due to the linear time complexity.

*(d) Parallelizing PILER-DF Search Method*

Next, we will show how to use the above approach to further improve PILER by parallelizing one of its four main search methods PILER-DF. This method is designed to find intact and isolated members of a dispersed repeat family. These members are identified as sets of three or more globally alignable piles.

This method works through the list of piles built in the main algorithm, as detailed in previous section, and finds out the partner images *T* of region *Q* in each hit (a local alignment of regions *Q* & *T*) in a given pile. It then checks if regions *Q* and *T* both cover a predefined fraction of bases in their corresponding piles, i.e. if *Q* overlaps with another region *Q'* to a certain extent (set as a user specified criteria) in the same pile where *Q* and *Q'* are two overlapping regions. Similarly, the region *T* is evaluated for such coverage in its corresponding pile. If both *Q* and *T*

for a given hit pass the test then their piles are connected by creating an edge in a graph G(V,E) such that *E* represents all such connections between various piles and *V* is the set of piles. Once the graph *G* has been constructed, finding dispersed repeat families is as easy as finding connected components (of order > 2) of *G*. Edgar [24] provides more detail on the method and the characteristics of the dispersed repeat family but that is outside the scope of this assignment.

Continuing with our parallelization of PILER, once all the processors have created local piles, it is trivial to parallelize the search method DF (dispersed family). Each processor builds a partial graph *G*' using its local piles and querying other processors (in case of message passing environment) when necessary to check coverage of the partner *T* (possibly remote) of every local *Q* analyzed. Once all such *G*'s have been constructed, master processor can merge them. In case a caching mechanism is required to support multiple runs of the search method such that the piles do not have to be created again every time, the master processor can assign the piles equally to all the worker processors and have each of them construct the partial graph as explained above.

Finally, finding the connected components can also be parallelized. Such algorithms have been presented in [49] and elsewhere. However, the complexity and the low speedup of such algorithms along with requirement for a large number of processors may be prohibitive of such approach for this part of the algorithm.

In summary, a significant portion of PILER can be parallelized in a way that will achieve a significant speedup over the sequential program.

## 2.5 Pairwise Sequence Alignment

One of the key tasks that are performed frequently in sequence assembly is aligning two sequences. This involves comparing the two sequences base by base and deriving an alignment which takes into account any mutations, insertions or deletions in either sequence

with the underlying assumption that the two sequences either overlap, be it partial or full, or can be merged together into one extended sequence. An alignment “score” is calculated by rewarding the matches and penalizing the mismatches between the corresponding bases of the two sequences.

The method of aligning two sequences is to write one on top of the other, and break them into smaller pieces by inserting spaces in one or the other, as needed, so that identical subsequences are eventually aligned in a one-to-one correspondence. In the end, the sequences end up with the same size.

```
A = ACAAGACAG-CGT
    |  ||  |  ||  |||
B = AGAACA_AGGCGT
```

Figure 2.9 A Pair-wise Alignment Example.

The objective is to match identical subsequences as far as possible. In the example, Figure 2.9, nine matches are highlighted with vertical bars. Two mismatches can be identified as: a “C” of sequence *A* aligned with a “G” of sequence *B*, and a “G” of sequence *A* aligned with a “C” of sequence *B*. The insertion of spaces produced gaps in the sequences. The gaps were important to allow a good alignment between the last three characters of both sequences.

An alignment can be seen as a way of transforming one sequence into the other. From this point of view, a mismatch is regarded as a substitution of characters. A gap in the first sequence is considered an insertion of a character from the second sequence into the first one, whereas a gap in the second sequence is considered a deletion of a character of the first sequence.

Once the alignment is produced, a score can be assigned to each pair of aligned letters according to a chosen scoring scheme. The matches are usually rewarded and the mismatches

and gaps are penalized. The overall score of the alignment can then be computed by adding up the score of each pair of letters. For instance, using a scoring scheme that gives a +1 value to matches and -1 to mismatches and gaps, the alignment of above example scores:

$9 \cdot (1) + 2 \cdot (-1) + 2 \cdot (-1) = 5$

The similarity of two sequences can be defined as the best score among all possible alignments between them.

### *2.5.1 Sequential Algorithms For Pairwise Alignment*

The automatic methods to perform sequence alignment belong to a class of programming methods called Dynamic Programming (DP), which is a general algorithm for solving optimization problems. In the following sections, we briefly describe several most popular Pair-wise Alignment algorithms.

#### 2.5.1.1 Needleman And Wunsch Algorithm

The Needleman and Wunsch algorithm [70] sets up a matrix where each sequence is placed along the sides of the matrix. Each element in the matrix represents the two residues of the sequences being aligned at that position. To calculate the score in every position [i,j], one looks at the alignment that has already been made up to that point and finds the best way to continue. Having gone through the entire matrix in this way, one can go back and trace through which way the matrix gives the best alignment. The key observation for the alignment problem is that the score between sequences A[1:n] and B[1:m] can be computed by taking the maximum of the three following values:

the score of A[1:n -1] and B[1:m -1] plus the score of substituting A[n] for B[m]

the score of A[1:n -1] and B[1:m] plus the score of deleting aligning A[n]

the score of A[1..n] and B[1..m .1] plus the score of inserting B[m]

From this observation, the following recurrence in Figure 2.10 can be derived:

$$Score(A[i,\ j], B[i,\ j]) = MAX \begin{cases} Score(A[1, i-1], B[1,\ j-1]) + sub(A[i], B[\ j]) \\ Score(A[1, i-1], B[1,\ j]) + del(A[i]) \\ Score(A[1, i], B[1,\ j-1]) + ins(A[i], B[\ j]) \end{cases}$$

Figure 2.10 Equation 1 to compute similarity score for Needleman-Wunsch algorithm.

where

Score (A, B) is a function that gives the similarity of two sequences A and B

sub (a, b) = scoring functions gives the score of substitution of a character a for b

del (c) = scoring functions gives the score of deletion of a character c

ins (c) = scoring functions gives the score of insertion of a character c

This recurrence is complete with the following base case:

**Score (A[0], B[0]) = 0**

Where A[0] and B[0] are defined as empty strings.

To solve the problem with this recurrence, the algorithm builds an ($n$ +1) × ($m$ +1) matrix $M$, where each M[i, j] represents the similarity between sequences A[1..i] and B[1..j]. The first row and the first column represent alignments of one sequence with spaces. M[0, 0] represents the alignment of two empty strings, and is set to zero. All other entries are computed with the following formula shown in Figure 2.11.

$$M[i,\ j] = MAX \begin{cases} M[i-1,\ j-1] + sub(A[i], B[\ j]) \\ M(i-1,\ j] + del(A[i]) \\ M[i,\ j-1] + ins(B[\ j]) \end{cases}$$

Figure 2.11 Equation 2 for Needleman-Wunsch algorithm.

After computing every element, M[n, m] will contain the similarity score of the two sequences. Since there are (m+1) · (n+1) positions to compute and each takes a constant

amount of work, this algorithm has time complexity of O(*mn*). Clearly, it has also quadratic space complexity since it needs to keep the entire matrix in memory.

2.5.1.2 Smith And Waterman Algorithm

The Needleman and Wunsch algorithm uses a constant gap penalty. That is, it gives the same penalty for each gap regardless of its length. In reality, it is often desired to set a higher penalty for a longer gap than for a shorter one. The Needleman and Wunsch algorithm was later generalized by Smith and Waterman to include this criterion [88]. In addition, the Smith and Waterman algorithm can be used for three different types of comparisons (see Figure 2.12).

The gap penalty function is defined as:

$$W(k) = u + v * k$$

Where

*k* = the gap length

*u* = the penalty for initiating a gap

*v* = the penalty for extending a gap

$$M[i, j] = MAX \begin{cases} M[i-1, j-1] + sub(A[i], B[j]) \\ M(i-k, j] + W(k) \\ M[i, j-k] + W(k) \end{cases}$$

Figure 2.12 Equation for Smith-Waterman algorithm.

The Smith and Waterman algorithm has run-time complexity of

$$\mathrm{O}(m * n)$$

The space required is

$$\mathrm{O}(m * n)$$

The Smith and Waterman algorithm gives the maximum sensitivity for comparing two sequences. Sensitivity is a measure of how well a method can detect the actual similarity between two sequences. There are a number of other algorithms, mostly based upon heuristics, which are popular among bioinformaticians. These include BLAST [6], FASTA [60] and many others which are all originally sequential but some parallel implementations have been reported elsewhere [18], [66].

### *2.5.2 Parallel Algorithms For Pairwise Alignment*

A number of parallel algorithms have been proposed for the classic dynamic programming case. This method has also been used in optimal and global pair-wise alignment of two biological sequences [70] later on extended by others [88] to perform local alignments as well.

This method of finding the longest common subsequence between two given sequences *X* of size *m* and *Y* of size *n* can be parallelized in many different ways. The original algorithm requires that an (m+1) x (n+1) similarity matrix *T* be built where each cell T[i,j] ∀ i, j where $0 < i \leq m$, $0 < j \leq n$ can be computed when three other cells T[i-1, j-1], T[i-1, j], and T[i, j-1] have been computed. First row and first column are initialized to 0. Without the loss of generality, we can assume that $m \leq n$. A sequential dynamic program would take $\Theta(mn)$. Below are three different approaches to a parallel solution to this problem.

#### 2.5.2.1 Minor-diagonal Approach

Any parallel solution to this problem must consider the above mentioned dependency upon three cells for computing any given cell in the matrix (except for 1st row or 1st column) and hence a parallel approach of simple row-wise or column-wise data partitioning is infeasible. One solution to this problem suggests building the similarity matrix anti-diagonally (or minor-diagonally) as shown in Figure 2.13. This approach lends itself to a simple parallelization where

each one of the *p* processors computes *n*/*p* columns using column-oriented data decomposition and assuming $n > p$. The arrows show the computational direction where one processor may be computing more than one cell to complete a given minor-diagonal (also referred to as diagonal here).

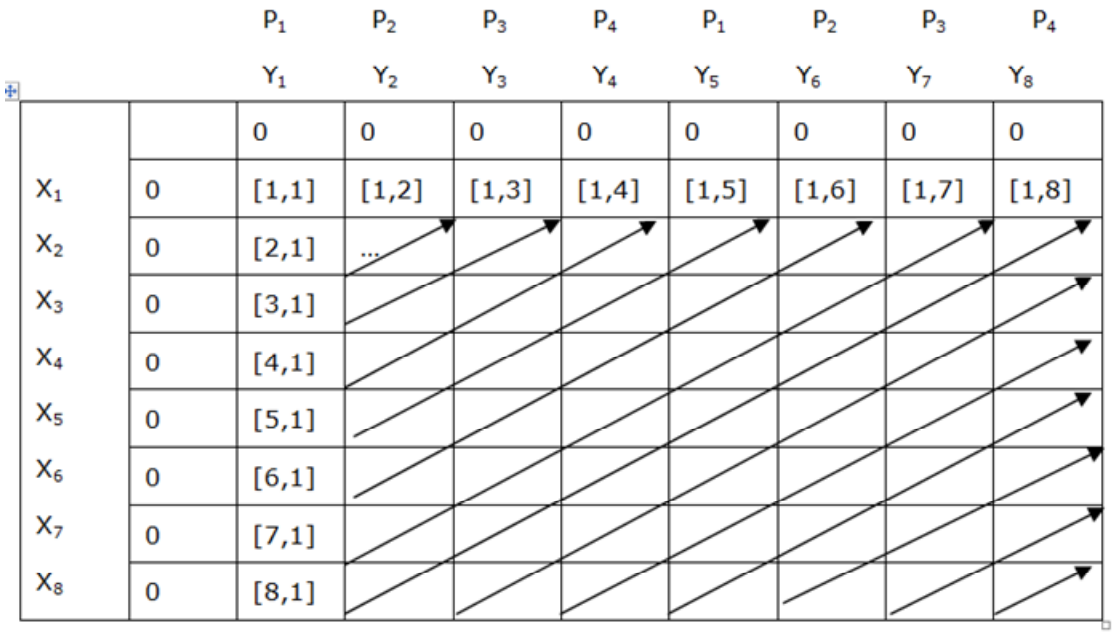


Figure 2.13 A parallel computation of the similarity matrix of two sequences X and Y, where |X|=|Y|=8, using 4 processors. For simplicity and clarity, only a couple of rows and columns' indices are shown inside the matrix cells.

The back trace should also be built during the processing and stored locally to be finally merged by the coordinator processor thereby resulting in the longest common subsequence (LCS) that can also be used to build alignment. In case of shared memory, an additional matrix can be maintained with each cell storing a trace back pointer.

Using PRAM model such that all processors have equal access time to all memory locations, i.e. Uniform Memory Access (UMA), a shared memory based implementation may be well suited for this approach in a concurrent-read-exclusive-write (CREW) environment since each processor has its own designated cells to write into. Once a processor has computed

cell(s) assigned to it, it must wait for other processors to finish before next diagonal can be computed. The disadvantage of using a shared memory based solution is that for very large sequences, which are typical in computational biology, the matrix size tends to be huge requiring memory in tens of gigabyte that may not always be available.

In a message passing environment, this approach requires that all processors broadcast their results once a diagonal has been fully computed. All the calculations along any given diagonal require values from previous two diagonals and for any given processor $P_i$ that translates into receiving two values from the preceding processor $P_{i-1}$ and one from within its own local memory.

For large sequences, this method should run in O(*mn*/*p*). All processors will broadcast and receive at least once after every diagonal is calculated, i.e. O(*m*+*n*) broadcasts for a processor. The communication-to-computation ratio (CCR) is fairly high in this approach. However, the benefit of having a large pool of distributed memory of individual processors outweighs the communication overhead and is more scalable than the shared memory model.

Regardless of the memory architecture used, a limitation of this approach is that the sizes of the minor diagonals vary and often can be too short to be computed using all processors resulting in idling of some processors. This implementation can use at most **min**(*m*, *n*) processors and is not very efficient when one of the sequences is short resulting in low utilization of processors.

2.5.2.2. Block Division Approach

Another approach that should reduce the inter-process communication has also been suggested [64]. This approach is based on dividing the similarity matrix into multiple rectangular blocks that are then assigned to different processors. Figure 2.14 shows a similarity matrix divided into 32 rectangular blocks, which are assigned to four processors $P_1$, $P_2$, $P_3$, and $P_4$. In this example, block 1 is computed first followed by computation of blocks 2 & 5 followed by 3, 6,

and 9 which are followed by 4, 7, 10, and 13 and so on. A better utilization can be achieved by assigning multiple rows of blocks to each processor, e.g. row 1 and 5 of blocks is assigned to processor $P_1$. Several of the blocks can be computed in parallel after some initial computation.

Finding LCS is essentially similar to the minor-diagonal approach using the [*m*,*n*] value of the matrix and tracing back. Such partitioning of the matrix decreases the maximum achievable parallelism as compared to the previous approach discussed above, but for very large sequences and a reasonable number of processors, that may not be a limiting factor.

| | | | | |
|---|---|---|---|---|
| $P_1$ | 1 | 2 | 3 | 4 |
| $P_2$ | 5 | 6 | 7 | 8 |
| $P_3$ | 9 | 10 | 11 | 12 |
| $P_4$ | 13 | 14 | 15 | 16 |
| $P_1$ | 17 | 18 | 19 | 20 |
| $P_2$ | 21 | 22 | 23 | 24 |
| $P_3$ | 25 | 26 | 27 | 28 |
| $P_4$ | 29 | 30 | 31 | 32 |

Figure 2.14 Task assignments in a block division approach.

2.5.2.3 A Row-wise Approach Using Parallel Prefix Computation

Considering the limitations and idling issues in the above two approaches, another way to calculate the similarity matrix for two given sequences in parallel has been proposed by Aluru et al. [7]. Their work pertains to the sequence alignment that also includes considering single or extended gaps in the sequences that carry penalty when computing similarity score. Following is a method to parallelize the matrix computation either row-wise or column-wise based on their referenced work.

The central idea in this approach is to calculate three matrices instead of only one. These three matrices correspond to the three sources for computing any given cell T[i,j] of the matrix, i.e. the cell directly above (T[i-1, j]), the cell diagonally above (T[i-1, j-1]), and the cell directly to the left (T[i, j-1]). Calculating only one matrix requires that all three predecessor cells' values be available before computing a given cell. However, by breaking this into three matrices with following conditions can be done in a quicker and parallel way as described below.

Let A = $a_1$, $a_2$, …..$a_m$ and B = $b_1$, $b_2$, ….$b_n$ be two input sequences with assumption that $m \leq n$. It can be safely assumed as the algorithm holds for symmetric case.

We can calculate three matrices T1, T2, and T3 where in T1 entries correspond to when a*i* = b*j*. T2 holds values for when a*i* ≠ b*j* and T[i, j-1] > T[i-1, j], i.e. when top cell is to be chosen over the left cell. T3 holds values for when a*i* ≠ b*j* and T[i-1, j] > T[i, j-1], i.e. when left cell is to be chosen over the top cell.

T1[i,j] = **max** { T1[i-1,j-1], T2[i-1,j-1], T3[i-1,j-1] } + 1 (2.1)

T2[i,j] = **max** { T1[i,j-1] , T2[i,j-1], T3[i,j-1] } (2.2)

T3[i,j] = **max** { T1[i-1,j], T2[i-1,j] T3[i-1,j] } (2.3)

Consider computing $i^{th}$ row of each table T1, T2, and T3. The $i^{th}$ rows of T1 and T3 depend upon corresponding $(i-1)^{th}$ rows and therefore can be computed directly. After that, $i^{th}$ row of T2 can be computed as follows. Two out of three values required to calculate the **max** for T2[i,j] are available from the other tables T1 and T3. The remaining term T2[i, j-1] can be computed using parallel prefix as below.

Let w[j] = **max**{T1[i,j-1], T3[i,j-1] } (2.4)

Substituting (4) in (2) we get

T2[i,j] = **max** { w[j], T2[i,j-1] } (2.5)

Let x[j]= T2[i,j] = **max** { w[j], T2[i,j-1] } (2.6)

x[j] = **max** {w[j], x[j-1] } (2.7)

Since “w[j]” is known for all j, x[j]’s can be computed using parallel prefix with “Max” as the binary associative operator. Then, T2[i,j] (1≤j≤n) can be derived using (2.6) as T2[i,j] = x[j] . Hence we can compute all three matrices concurrently using multiple processors. Given *p* processors, each computes *n*/*p* columns of each of the matrices concurrently except for when a communication is needed (for the 1st column for computing T1[i,j] and w[j].). Message passing is used to broadcast the score to next processor as soon as available. Processor *i* is responsible for computing the columns *i*(*n*/*p*)+1 through (*i*+1)*n*/*p* of the matrices T1, T2, and T3.

As this approach uses two additional matrices of size (*m*+1)*(*n*+1), it is much better suited to the distributed memory environment using message passing compared to the shared memory environment that would require extra ordinarily large memory space for longer sequences.

*Finding LCS*

The length of the longest common subsequence is given by **max**{T1[*m*,*n*], T2[*m*,*n*], T3[*m*,*n*]}, i.e. the maximum of the [*m*,*n*] entries in all three matrices. The LCS itself can be built using a trace back procedure starting from that entry in O(*m*+*n*) time. This can be accommodated without much effect on parallel runtime that is asymptotically O(*mn*/*p*). The space required at each processor is also O(*mn*/*p*). Aluru et al. [7] has worked out more details on space saving and parallel trace back algorithms.

## 2.6 Multiple Sequence Alignment

The multiple sequence alignment, sometimes referred to as MSA, aims to align *n* sequences $S_1$, $S_2$,…, $S_n$, *n*>2, by inserting gaps in one or more sequences such that all sequences have the maximum possible alignment and the same length. It is a technique that is used in the computational biology for various reasons. From phylogenetic research [73] to finding a consensus sequence for genome assembly, there are numerous applications of the

MSA algorithms. Although MSA is commonly applied to the analysis of protein, RNA, and DNA sequences, our focus for this work remains DNA centric. In this section, we will first explain the technique followed by a detailed discussion of the high performance implementations available.

### *2.6.1 The Technique*

The primary objective of MSA is to compare a set of three or more sequences to locate maximum overlapping regions. As discussed in earlier section on the pair-wise alignment, two sequences are aligned by positioning one on top of the other and leaving the matches intact while either leaving or replacing a mismatch with a gap, based on the overall matching score accumulated up to that point. At the end of the process, both sequences are of same length with gaps inserted in one or both to achieve best possible alignment. Although the same concept is valid for MSA, the number of possible alignments grows with the number of sequences in the input data set raising question of how to choose the best possible alignment. Following are three sequences to illustrate the process.

S1 = CGCAAATTAG

S2 = TCGAAG

S3 = GCATGTG

There are several options available to align these sequences. Table 2.2 shows a few possible alignments.

Table 2.2 Multiple Sequence Alignment

| Alignment 1 | Alignment 2 | Alignment 3 |
|---|---|---|
| CGCAAATTAG<br>TCGAAG<br>GCATGTG | CGCAAATTAG<br>TCGAAG<br>GCATGTG | CGCAAATTAG<br>-TCGAA---G<br>-GC-ATGT-G |

There are different methods to align multiple sequences including dynamic programming, progressive and hierarchical methods, iterative methods, and less commonly used non-deterministic methods, e.g. genetic [46], [100] and HMM based algorithms [15]. Following is a brief introduction to each of the more widely used approaches followed by a discussion of their high performance implementations.

*2.6.2 Dynamic Programming*

The dynamic programming (DP) approach used in pair-wise alignment can be extended to align *n* sequences by building an *n*-dimensional matrix. Although guaranteed to produce an optimal alignment, the time and space requirements for solving solutions for more than a few sequences exceed the capability of normally available computational resources as the space complexity of the algorithm can be given as $O(l^n)$ where *l* is the average length of each of the *n* sequences [70]. This limitation renders the DP approach practically infeasible for MSA. The parallel algorithms devised for the pair-wise alignments can also be used here; however, not much improvement can be made due to NP-hard nature of the problem.

The primary objective of dynamic programming is to produce an optimal alignment and it follows from our discussion of optimal pairwise alignment that for *n* input sequences, an *n*-dimensional matrix needs to be built. Assuming an average length *l* for all the sequences, the space complexity of multidimensional dynamic programming is $O(l^n)$. The time complexity is

$O(2^n l^n)$. It is obvious that computing such an optimal alignment for even a few sequences becomes very computationally intensive; therefore, this approach is not used for any reasonable number of sequences. There are many heuristic methods and variations of this approach none of which guarantees a mathematically optimal alignment. There have been some implementations of dynamic programming [60] to align multiple sequences but those programs are not scalable beyond limited number of sequences with small lengths.

*2.6.3 Progressive Alignment*

The most commonly used method to align multiple sequences solves the problem progressively by comparing each possible pair of input sequences for overlaps, i.e., $\binom{n}{2}$ pair-wise comparisons are made. Such comparisons yield a distance matrix based on the degree of overlap between each possible pair. The algorithm builds a phylogenetic tree using the matrix and aligns the sequences in a hierarchical order dictated by the tree. The primary issue with this approach is that an incorrect pairing in the initial stages may lead to a poor alignment which is never revisited and therefore affects the final outcome of the process. CLUSTAL [39] was one of the first implementations of this approach and has been widely used by researchers in its various forms reported by [40], [48], [56], and others including a parallel implementation by [58] discussed in the following sections.

Figure 2.15 shows a typical alignment of four sequences using progressive method. There are three main stages of computation in this approach. For a set *S* of *n* input sequences, the first and foremost task is to perform pairwise alignment between sequences $S_i$ and $S_j$ for all *i*, *j* such that $0<i<n$ and $i<j<=n$ yielding $n(n-1)/2$ pairwise alignments. This stage is an ideal candidate for a parallel implementation as each of these alignments can be made independent of others. A master processor can distribute a balanced load to all processors and then receive the resulting alignments information after all processors have completed their tasks. An *n*x*n*

matrix is built using the alignment information which, depending upon a particular implementation of the algorithm, includes a distance or a similarity score between each pair of the sequences.

This distance matrix will be used in the later stages. Several parallel implementations, e.g. [58] and [61] have applied a similar approach to CLUSTAL, which is the most commonly used tool based upon progressive alignment methodology.

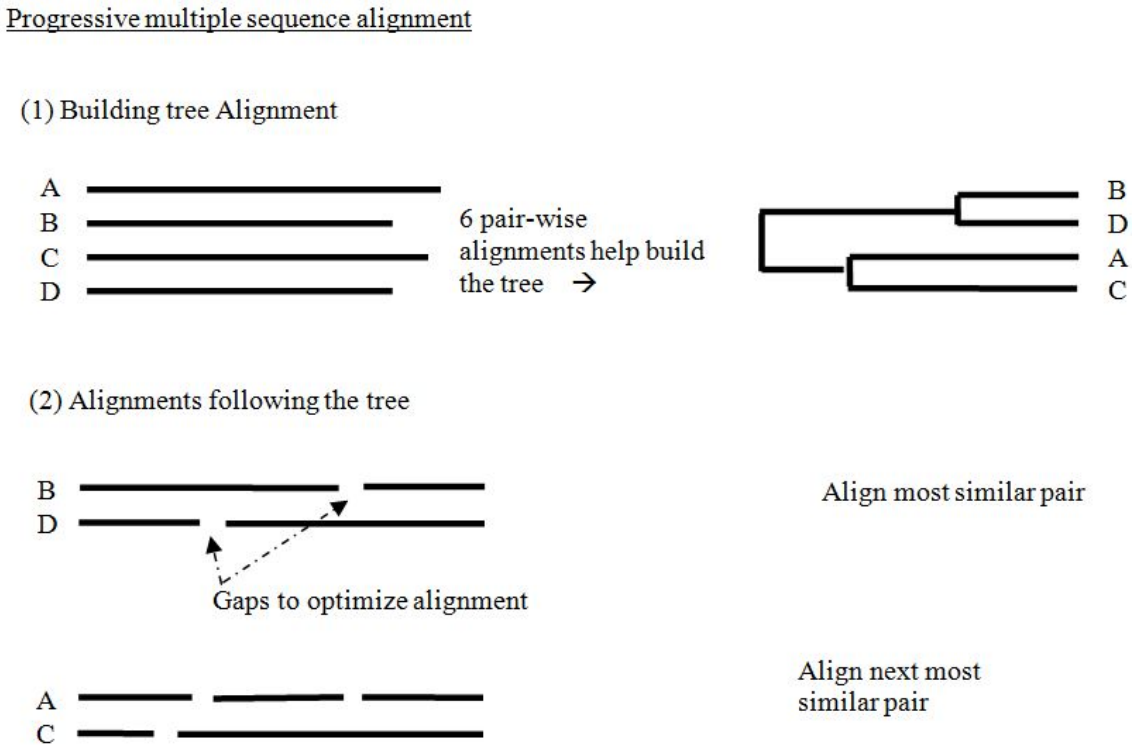


Figure 2.15 An example of progressive alignment.

MUSCLE [23] and T-Coffee [71] both spend upfront processing time to build initial lists of similarity scores for each possible pair of alignments which can easily leverage an embarrassingly parallel and distributed computing where each processor has access to the entire data set but works only on its share of sequences. Parallel implementations of both algorithms have been reported [19], [102]. The next stage does not lend itself to an

embarrassingly parallel implementation that was achieved during the construction of the distance matrix. Here, a phylogenetic tree, also referred to as a guide tree, has to be built using the distances computed in the previous stage. There are various approaches to constructing the guide tree. Most commonly, the neighbor-joining (NJ) method is used [82]. There is also another method called “Unweighted Pair Group Method with Arithmetic mean” or simply, the UPGMA. One of the main differences between the two approaches is that NJ, unlike UPGMA, does not assume that all of the sequences evolved at the same rate and is therefore much better suited to a diverse data set. Also, from a parallel implementation perspective, it is more efficient and a near linear speedup has been reported [21].

In the third and the final stage, the guide tree is traversed starting from the outermost (leaf) nodes representing the most closely related pairs of sequences. An alignment is computed between every two nodes and the result is used as input to the next alignment. The efficiency and scalability of parallelization in this stage depends upon the structure of the guide tree and is therefore best accomplished using dynamic scheduling [61], which is explained in the following section. Li [58] has used a mixed approach and employs coarse-grained parallel distribution for the leaf nodes whereas inner nodes are aligned sequentially for the most part. An optimal speedup of $n/\log n$ is reported when the tree is well balanced.

#### 2.6.3.1 Dynamic Scheduling

Many parallel implementations of the third stage of progressive alignment distribute workload to the available processors based on the topology of the guide tree assuming a uniform communication cost and synchronization overhead [58], [17]. However, a very important aspect of task scheduling is the ability to assign tasks at runtime based upon dynamic utilization of each processor such that the workload can be distributed real-time in a more balanced way, i.e. the scheduling decisions are made at run time because the execution time of an alignment task and communication overhead depend upon the outputs of the preceding tasks and are not

available at compile time. Therefore, dynamic scheduling approach is required. Luo et al. [61] implemented a dynamic scheduling technique designed for distributed memory parallel machines as follows. The guide tree is mapped onto a “task” tree using directed acyclic graph (DAG). The master processor acts as a scheduler and maintains a list of ready tasks with all the children nodes already processed. A priority list is also maintained for each task where the priority is the length of the longest path from the task to the root task. Once a task has completed, the scheduler updates the estimated execution time and communication cost of its ancestors. The ready task list and the priorities of the ready tasks are updated as well. Finally, the task with the highest priority is assigned to an idle processor.

### *2.6.4 Iterative Alignment*

In a classic trade-off between accuracy and efficiency, the iterative methods attempt to address the key deficiency of the progressive methods by revisiting an alignment in an attempt to improve it in the later stages based upon the deductions learned in later steps. While it helps enhance the accuracy of the alignment in many cases, the improvement comes at the expense of the overall longer execution time. A detailed account of such algorithms can be found in [12] and [41]. The Burger-Munson [12] iteration method applies randomized techniques with optimization functions to iteratively improve the multiple sequence alignment. It contains three steps as discussed below.

The *n* input sequences are partitioned into two groups randomly. One group has two sequences. This has been proven as the best partition strategy resulting in a total number of partitions to be *n**(*n*-1)/2. One partition is aligned as if they were two sequences (freezing the alignment of sequences within each group). The new gap positions are saved for step 3.

A decision flag is set to true if the new resulting alignment is accepted, otherwise, it is set to false. A new alignment is accepted if its align-score is higher than the best score. If the flag is set to true, the gap positions determined in step 1 are used to modify the current

sequence pool and the best score is updated. The modified and unmodified alignment is used as the input for the next iteration.

Termination condition: after *q* consecutive iterations of rejection. $q = n*(n-1)/2$.

The Berger-Munson algorithm is expressed as the following:

```
1. best_score = initial_score();
2. While (stop criteria is not met)
3. {
4. current_score = calculate(seq, gap_positions);
5. flag = decide(current_score, best_score);
6. seq = modify(seq, flag, gap_positions);
7. }
```

#### 2.6.4.1 High Performance

The original algorithm is sequential and each step in iteration depends upon the previous one. A number of attempts were made to parallelize this algorithm in order to attain a significant speedup. Here, we look into an improved approach presented by Yap *et al*. [98] who studied other similar sequential algorithms that were parallelized using “speculative computation”. This approach led to a higher speedup and a more scalable implementation in addition to guaranteeing the same results as achieved by original algorithm proposed by Burger-Munson.

The original Burger-Munson algorithm does not lend itself well to a parallel programming environment due to the high degree of dependence among different steps. The algorithm divides the sequences into two groups. It then compares sequences from one group to the second group and attempts to align the sequences by inserting gaps. A score is calculated for every alignment, as described earlier. The alignment with higher score is accepted and kept and the one with lower score is rejected. The process is repeated with the

new aligned sequence as input. This is an iterative process and the final outcome is the closest possible match with the highest score.

Yap et al. [98] successfully parallelized the above algorithm using speculative computation that is described in this section. In the original algorithm, the rate of acceptance of alignments is higher in the beginning as there are not many scores available to compare with. However, in the later iterations, more alignments are rejected as depicted below where *A* denotes an "Accept" and *R* is a rejection.

**Sequential Iteration number**

1 2 3 4 5 6 7 . . .

**Decision sequence**

A A A A A R A A A R R A R R R A R R R A R ……

Considering the above observation and the heuristic data that supports it, they speculate that 'x' number of iterations will be rejected after one has been accepted. When an alignment *m* is accepted, its score is higher than that of the previously accepted alignment. It does not depend upon the preceding alignments that were rejected because their scores were lower. Therefore, the $i^{th}$ iteration depends on $(i-1)^{th}$ iteration only if the $(i-1)^{th}$ iteration had accepted the alignment; otherwise it only depends upon the last accepted iteration that could be (i-j) for 1<j<i. Hence, it was proposed that the iterations be partitioned and assigned to multiple processors. The recommended number of processors depends upon the average and speculative number of rejections between two acceptances. This would allow parallel alignments with *p* processor such that (*p*-1) processors would supposedly handle the rejections only and one processor will be successful in alignment that would be accepted. In other words, if a 4-processor machine is used based on the speculation that 3 consecutive iterations would be rejected before one is accepted, then the iterations by processor $p_0$ would result in acceptance whereas the iterations by processors $p_1$, $p_2$, and $p_3$ would be rejected. This

proposition is quite speculative; therefore, it must also handle the exceptions as discussed in the referenced report [98] in detail.

Using the above-mentioned approach, the decision sequence, as given above, can be assigned to 4 processors that would result in 13 parallel steps compared to 28 sequential. In this scenario, a total of 3 (i.e. *p*-1) iterations are speculated at each parallel step. The processors $p_3$, $p_2$, and $p_1$ all speculate that their preceding processor will reject its iteration. The processor $p_0$, however, does not speculate. It is claimed that the parallel alignment is guaranteed to be the same as the sequential one and that this algorithm may provide significant speedup when matching a large number of sequences.

It is assumed that inter-process data transfer is much faster than the I/O; therefore, only processor $p_0$ is assigned to read input sequences which then broadcasts that to other processors. Although the initial iterations are usually accepted because of high probability of finding higher score each time, the rejection rate increases after a few iterations. Therefore, the new algorithm starts speculating from $4^{th}$ iteration onward. Every processor evaluates the same partition by initializing the same random seed. This strategy avoids the communication cost associated with the parallel speculative computation. The iteration number is used as the random seed that helps backtracking in the case of an incorrect speculation.

After a rejection is encountered, the speculation begins until all possible partition rejections have been encountered. A random partition is selected only if it has not already been selected since the last accepted partition. That is, no partition is selected more than once by any processor or by different processors simultaneously. In order to ensure that no partition is selected more than once, each processor must know (1) the partitions that have already been selected and (2) the partitions that are currently being selected by other processors. The algorithm attempts to avoid inter-process communication required to know that information by using an array of *q* bits which correspond to the *q* possible partitions. When iteration *i* is selected, $i^{th}$ bit is set as an indicator to all processors that it has been selected once so the

processor may select another one. All *q* bits are cleared every time a new partition is accepted. To determine the partitions that are being selected by other processors, each processor generates *p* random selectable partitions instead of just one and then selects the $i^{th}$ one (the other processors are selecting the remaining partitions). All *p* bits that are corresponding to the *p* selectable partitions are set. The global operation is performed after each processor makes its decision. The accepted alignment with the smallest iteration number is selected as the input for the next iteration since the alignments with higher iteration numbers are invalid as those were based on incorrect speculations. When a new partition is accepted, the contents of local variables are copied from the accepted processor to the other processors. The number of sequential iterations, which were correctly speculated for skipping, is determined. If there is a global accepted partition (iteration) among the *p* partitions evaluated, only the iterations smaller than or equal to the accepted iteration are skipped. Otherwise, all *p* iterations are skipped.

Although iterative methods attempt to correct any erroneous alignments by repeatedly examining and reprocessing the partial results, the progressive MSA approach is most widely used due to its faster processing time that allows quick analysis.

## 2.7 Scaffolding

After the longer contigs have been obtained using pairwise and multiple sequence alignments, the process of ordering and lining them together begins by utilizing the mate-pair information that was recorded in the laboratory and passed on as a data file to the assemblers earlier in the assembly process. In general, the input data for scaffolding is of the following types.

1. the contigs created in previous stage.
2. mate-pair constraints which specify relationship among different reads in terms of distance and orientation noted at the time of sequencing.
3. reads which were unused due to lack of any overlaps with other reads.

4. reads which were thought to be repeats and were therefore filtered out during earlier stages.

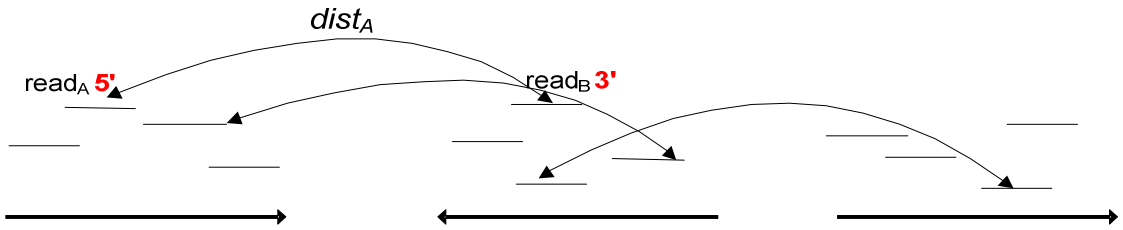


Figure 2.16 A typical scaffolding.

Figure 2.16 illustrates that, when sequenced during chain termination process, the mate-pair distance $dist_A$ in number of bases between readA and readB, along with the orientation. i.e. 5' vs. 3', is known and used later on during scaffolding to align and orient the contigs containing these reads along with other such pairs. A high level process flow can be stated as below.

1. At first, all the relationships among the given set of contigs is determined by using the corresponding mate-pair distances between each of the reads that constitute the contigs. The reads within the same contig are not matched in this fashion for obvious reasons.
2. A set of scaffolds is created along with marked start and end positions, relative to other contigs within the scaffold, of each read and contig.
3. Once such a frame is constructed, the gaps are to be filled using the available pool of reads as explained above.

Although step 1 and 2 can be easily parallelized, matching each contig and the reads within it, to every other read in other contigs would generate a very large number of messages among processors when implemented in a distributed computing environment. This will result in a high communication to computation ratio. Therefore some parallel assemblers, e.g. PCAP [42], implement this part of the algorithm as sequential, i.e. on the master node. Once all the

contigs have been grouped into scaffolds, the data is distributed to all available processors. Often, data load is balanced by sorting the scaffolds in descending order of size and assigning them to the available *p* processors in a round robin fashion. In other words, after the sorting is completed, scaffold *i* gets assigned to processor *j* where *j=i* mod *p*. Each processor works on a group of scaffolds in parallel as there is only minimal communication needed after this point. Each processor scans the scaffolds and attempts to fill the gaps between the contigs using the available pool of reads *R* as follows.

A 3' end region of the contig before the gap is located such that the end region is longer than any overlap. Let U3 denote the set of unique reads that end in the region. In a similar way, another set U5 is identified that contains unique reads that start in a proper 5' end region of the contig after the gap. Using the three sets, i.e. *R*, U3 and U5, overlaps are computed and used to fill in the gap resulting in a new and longer contig. If no overlaps are found then the two contigs cannot be combined together. Once all such gaps have been considered, the program attempts to perform multiple alignments of reads and consensus sequences are constructed for the contigs in the group. The details of such algorithms used in assemblers have been reported [42].

## 2.8 Finishing

The computer aided assembly is an example of a very challenging problem in bioinformatics where scientists have used the well-established concepts of computer science and application of both sequential and parallel algorithms. Although it helps tremendously in processing large sets of data allowing the researchers to fine tune different parameters to yield an array of good quality sequences, an optimal or even a near-perfect solution is well beyond the capabilities of this automated process. Therefore, a manual intervention is required to complete the sequencing process by validating the scaffold sequences, their orientation and the

filled gaps, using other data, e.g. chromosomal maps and markers etc., available from the laboratory.

There are many reasons why such manual effort is necessary. At the time of this writing, the procedures performed in the laboratory are less than perfect. The length of the reads is limited to less than $10^3$ and the quality, especially near the ends, is still not optimal. Additionally, the repeats continue to be an issue, especially when the repeat size is greater than the length of the read itself. As described earlier, the standards related to the error rate and contiguity require careful verification and adjustments. Hence the final phase of the assembly needs a human checking and is termed as *finishing*.

There are some programs which have been developed to help in finishing with a primary focus on providing a visual aid to determine and correct the assembled sequences. Tools like Consed [37] and ClustalX [48] provide user friendly graphical interfaces to assist in visual editing of the final sequences.

## 2.9 Quality And Performance Evaluation

From computer science perspective, solving bioinformatics problems require the design and development of intelligent algorithms, which can provide speedy, accurate, and scalable solutions that are economically viable to implement. Since there are many different algorithms and various implementations of each of them, it becomes necessary to evaluate such techniques and the corresponding implementations using a high level frame of reference and common set of metrics to allow a comparative analysis. Furthermore, for various algorithms in the realm of parallel and distributed computing, it is essential to have the ability to compare them both quantitatively and qualitatively, using the well-established metrics. Table 2.3 shows some of the metrics that should be used to distinguish one algorithm from another.

Table 2.3 Key Measures To Evaluate Algorithms And Their Implementations.

| **Metric / Measure** | **Algorithmic (A), Implementation (I), or Both (B)** |
|---|---|
| Time complexity | A |
| Processing time | I |
| Ease of use | B |
| Accuracy | B |
| Data Size | I |
| Speedup | I |
| Efficiency | I |
| Isoefficiency (to measure scalability) | B |
| Space requirements | B |

It should be noted that it is not possible to declare one approach to be the best of all as different problems require their unique criteria to be met and hence a “one solution solves all” approach does not exist. There is generally a trade-off between performance and quality where performance can be defined as the faster processing of the data and the quality can be assessed using certain standards to measure accuracy and reliability of the solution. Depending upon the need at hand, a faster, easy to use, and more economically viable program may be more desirable than a slow and complex yet more accurate implementation and vice versa.

### *2.9.1 Standards And Limitations*

We have discussed that there are certain limitations that prevent the computer scientists from developing the optimal solutions, e.g. NP-completeness of optimal alignment algorithms, which if deployed, could take non-deterministic time to produce an optimal solution for even a small data set. It is, therefore, infeasible to calculate 100% accurate results. However, there are some standards developed by the umbrella research organizations like NHGRI that, if met, validate the viability of a solution. For example, during the Human genome project (HGP), it was decided that there should be no more than 1 erroneous base for every 10 thousand bases in the finished sequence, i.e. an error ratio of 1:10,000. This is referred to as

Bermuda quality standard due to the location where the conference took place. Also agreed upon standard is the elimination of all gaps within the sequences. This is a rather ambitious target and is normally set to encourage further accuracy in the sequencing techniques. Another important standard is that the length of the finished contig is required to be greater than 30kb [27]. It should also be noted here that fewer contigs of larger size generally indicate a better assembly compared to a larger number of small size contigs.

In this section, we analyze different performance and quality related characteristics of techniques and implementation commonly used in assembly process. Although most assemblers use various phases of the assembly process as illustrated by Figure 1.1, there are some techniques that are used outside of the regular assemblers when only partial function of the assembly process is needed. For example, there are times when a multiple alignment of a set of sequences is needed for developing a phylogenetic tree or when sequences are analyzed to check the existence of certain types of repetitive regions. In these scenarios, a full-fledged assembler is not required and special purpose programs are used.

Following sections summarize the capabilities and key performance indicators related to a few commonly used assemblers as well as other special purpose individual programs that are widely used by the researchers.

### *2.9.2 Assemblers*

Table 2.4 lists some key assemblers of interest and some important performance related characteristics. It is provided purely for reference purposes. As stated earlier, it is not feasible to compare assemblers as each program has used a different set of reads, quality scores, and mate-pair constraints. Some have used real data while others created data set artificially. However, some of the performance measures can be used to broadly compare different assemblers. For example, N50 contig length is an important metric indicating the

**max**($|C_i|$) where $1 \leq i < N$, in a sorted array *C* of all *N* contigs in ascending order of length such that

$$\sum_{j=i}^{N} |C_j| \geq 50\% \text{ of genome size} \tag{2.8}$$

i.e., size of the smallest contig such that 50% of the length of the genome is contained in contigs of size N50 or greater. However, this metric can only be used for comparison when two or more assemblers used same genomic data. We encourage the reader to find similarities, differences, weaknesses, and strengths of different assembly programs using the following statistics which represent a high level performance and quality assessment of such programs including Arachne [10], PCAP [42], Phusion [68], and Celera [69]. There are also other assemblers, e.g. Phrap [34], CAP3 [42], and TIGR [90], which have all proven to be quite efficient and accurate but remain outside of the scope of this work due to their purely sequential algorithms that do not utilize any significant high performance computing (HPC).

Table 2.4 A High Level Statistical Summary of Various Assembly Programs Employing HPC

| **Assembler** | **Genome type & data size** | **Reported** | **Coverage** | **N50 contig length** | **Error frequency** | **Platform** | **Processing time** |
|---|---|---|---|---|---|---|---|
| Arachne | D. Melanogaster (120Mb) | 2002 | 10x | 350kb | 1:1Mb | Single processor Compaq Alpha (8GB RAM) | < 1 day |
| PCAP | Human chromosome 20 (60Mb) | 2003 | 12x | 41.3kb | 8:1Mb (includes both mis-joins & mis-links) | 8-processors (2GB RAM each) | Several days |
| ABySS | Human genome | 2009 | 42x | 1.5kb | 2:30Mb (98.8% accurate) | 8-processors (144GB total) | 3 days |
| Phusion | Mouse (2.5Gb) | 2002 | 7.5x | 41kb | 12:1Mb | Single processor Compaq Alpha (32GB RAM). Also used 400-node cluster for an intermediate grouping of reads | Several days (adjusting for single processor time) |
| Celera | Human genome | 2004 | 5.11x | 82kb | <100:1Mb (accurate details not available) | 10 4-processors SMPs (4Gb RAM per cluster)<br><br>and<br><br>8-processors Compaq Alpha (64Gb shared RAM) | Several weeks |

### *2.9.3 Repeat Finding Tools*

At the time of this writing, only RepeatMasker has an option to run the program in parallel mode. However, other programs also lend themselves to distributed computing to varying extents. Table 2.5 provides some key characteristics of widely repeat finding programs available in the public domain that were studied for this survey.

### *2.9.4 Multiple Sequence Alignment Programs*

Multiple sequence alignment has always been an area of interest for bioinformaticians as there are numerous reasons for aligning multiple sequences. Most of the algorithms developed and implemented by the research community were originally written to aid in the area of homology and have also been extensively used for deriving consensus sequences during genome assembly. Due to the high order time complexities, the algorithms have been of interest and therefore studied to leverage the parallel and distributed computing capabilities made readily available by today's cost effective cluster based architectures. Table 2.6 lists some of the high performance algorithms and implementations pertinent to the scope of our work. A more detailed comparison of sequential MSA programs and the benchmarks used for their evaluation has been documented in [92], [25] and [13] among others.

Table 2.5 Various Repeat Finding Tools And Their Characteristics

| **Program** | **Reported in** | **Types of repeats** | **Dependencies** | **Strengths/Weaknesses** | **Capable of high performance computing** |
|---|---|---|---|---|---|
| PILER | 2005 | Elementary repeats: Terminal repeats, Tandem (contiguous) arrays, Pseudo-satellite (clustered but not too close like Tandem), Dispersed (isolated & scattered) | Any tool that can first build alignments, e.g. RepeatMatch, REPuter or BLAST) | *Pros:*<br>- High specificity in finding the dispersed family for mostly mobile elements though it may vary by genome<br>- Modular<br>- No external reference sequence(s) needed<br><br>*Cons:*<br>- Sensitivity varies by genome<br>- Requires significant computer programming skills to use | High potential for parallel implementation as described in this work |
| RepeatGluer | 2004 | Shows mosaic structure of repeats but does not identify families like PILER | A tool to first build alignments, e.g. BLAST, REPuter, or PatternHunter | *Pros:*<br>- Complements SBH and enables Euler path construction<br>- Supports A-Bruijn graphs to yield order independent consensus repeat segments<br><br>*Cons:*<br>- Not ideal for identifying specific repeat families<br>- Minimal documentation | Possible, however, a relatively large CCR due to distributed graphs can limit the merits of parallelization |

Table 2.5 - continued

| Program | Reported in | Types of repeats | Dependencies | Strengths/Weaknesses | Capable of high performance computing |
|---|---|---|---|---|---|
| RepeatFinder | 2001 | Repeats are groups and classified based on the type of constituent repeats. Classes are combined when repeats from different families are merged | REPuter to first build alignments as input and WUBLAST as a search engine at a later stage | *Pros:*<br>- Faster<br>- Makes no prior assumption of repeat structures<br><br>*Cons:*<br>- Uses larger memory due to suffix tree implementation<br>-Does not differentiate among major families of repeats | N/A |
| Repeat Masker | 1996 | Alu and interspersed repeat types | Repbase: A database of previously found repeat regions<br><br>Also, WUBLAST or Cross_Match as search engines | *Pros:*<br>- Highly sensitive<br>- A user friendly browser based interface<br>- Faster because of using existing repeat sequences of same or similar specie<br>- Can be run in parallel mode<br><br>*Cons:*<br>- Dependent upon external reference (Repbase) | Can be run with two processors as an option |

Table 2.6 Various Multiple Sequence Alignment Tools And Their Characteristics

| **Program** | **Reported in** | **Used in** | **Methodology** | **Performance**<br><br>*(N=Number of sequences, L=avg. length of a sequence, S=Speedup, E=Efficiency)* |
|---|---|---|---|---|
| Clustal | 1988 | Various | Progressive | Single MS-DOS based computer |
| Parallel ClustalW with dynamic scheduling | 2005 | Stand alone | Progressive | N=80, L=340 on 8-processor Linux 2GB RAM each<br><br>*S=6, E=0.75* |
| MUSCLE | 2004 | Stand alone | Progressive | N=1000, L=282 on a single 2.5GHz processor |
| Parallel MUSCLE | 2006 | Stand alone | Progressive | N=150, L=330 on a Unisys 16-processors system<br><br>*S=15, E=0.93* |
| T-Coffee | 2000 | Stand alone | Progressive | Single PII-330MHz processor. More accurate at the cost of speed |
| Parallel T-Coffee | 2007 | Stand alone | Progressive | N=1048, L=520 (max) on 80- processor system<br><br>*S=4 (relative to a 16-processors base), E=N/A* |
| Parallel Burger-Munsun | 1998 | Stand alone | Iterative | N=324, L=64 on a 64-processor system<br><br>*S=50, E=0.78* |

## 2.10 Techniques To Enhance Accuracy

Following is a brief discussion of some contemporary approaches to ensure correctness of the genome assembly. Most of these approaches apply to both hierarchical ([B|P|Y]AC based) and whole-genome shotgun sequencing (WGSS) though the latter requires comparatively more error checking and manual intervention during the "finishing" stage due to the size of data and the number of repeats to handle correctly.

It is not easy to attain the required degree of correctness, usually governed by the Bermuda standard for HGP that allowed no more than 1 error in 10,000 base pairs, by any single measure. A multi-pronged approach is needed that facilitates quality assurance from the start of the sequencing process, e.g. redundancy (coverage) of *reads* data. Some of the generic and most widely used techniques to ensure that the constructed genome represents the actual genome sequence are mentioned below in somewhat procedural order.

### *2.10.1 Redundancy/Coverage*

Cloning techniques allow over-sampling and hence theoretically guarantee the inclusion of each base in the data set at least once. A high coverage also yields better probability of getting high quality base calls and therefore more accurate overlaps during the assembly process. However, extra coverage also increases computational complexity, processing time, and memory requirements. Additionally, an underlying assumption here is the attainability of uniformly random-sampled inserts from the genome which is practically not feasible. Such cloning bias leads to a non-uniform random sampling that causes some areas of genome to be less covered than others, affecting accuracy of the overall assembly.

### *2.10.2 Mate-pair Constraints*

As discussed in previous sections, the mate-pair constraint, which is the distance between two reads recorded at the time of sequencing, is helpful in achieving high accuracy of

assembled sequence. Usually, after the contigs have been merged together as a result of overlap detection and subsequent alignment, they can be scaffold together using the mate-pair information between reads that belong to the two contigs. However, this technique has a shortcoming due to the fact that some repetitive regions (repeats) may be longer than the clone-insert size, especially in the WGSS paradigm, thereby rendering the mate-pair information less useful.

*2.10.3 Chromosomal Maps*

In most genome sequencing projects, the physical and genetic maps are either available or can be made available using either fluorescence in situ hybridization (FISH) mapping, mapping of sequenced tag sites [STS] on YAC clones, and/or radiation hybrid (RH) mapping before the new genome is fully sequenced. These maps provide markers that are useful in ensuring that certain sequence checkpoints are met during assembly process. However, these maps & markers are at a high level and are not sufficient alone for ensuring accuracy of the entire assembly.

*2.10.4 Comparative Mapping*

For a genome of specie that is considered to be closely related to species already sequenced, a comparative alignment can show large conserved chromosome segments. Many studies have shown that large regions of DNA are conserved in mammalian species as divergent as human and fin whales and gene order is also conserved to significant extent. Although this may help in verifying an assembled sequence, especially in the areas of conservation and ordering of genes, it is not very powerful technique to rely upon as some closely related species (e.g. Mice) may have a fairly high mutation rate.

*2.10.5 Genome Specific Measures*

Some genomes are polymorphic in nature resulting in multiple haplotypes. Depending upon the degree of heterozygosity, i.e. presence of different alleles at genes' loci for each of the homologous chromosome, the conventional algorithms and procedures for genome assembly may not work well as that may result in a superfluous sequence, lower N50 sizes (for both contigs and scaffolds), and mishandling of repeats thus breaking scaffolds incorrectly at later stages of assembly.

Vinson et al. [95] proposed some measures to ensure that a polymorphic genome is assembled correctly. Using a regular genome assembler Arachne, they added a constraint called "the splitting rule" that skips overlaps between the reads belonging to the opposite haplotypes. That results in two separate haplotype assemblies that are then combined by merging their haplotype scaffolds using their long-range inter-correspondences. In terms of accuracy, it claims to have achieved 96% alignment with the finished sequence of 14 Fosmid clones totaling 542kb out of 190 Mb genome of Ciona savignyi. Also, others [50] have used Phrap assembler to tackle this issue in somewhat similar way. Their work is, however, not generic in terms of the number of haplotypes and is limited to a diploid genome sequence, e.g. Ciona savignyi, i.e. it cannot be directly used for assembling other polyploid genomes (having more than two haplotype sequences) that are fairly common in plants.

Additionally, several techniques are used that indirectly help achieve high accuracy by keeping the error rate low such as filtering out incorrect data, trimming low quality ends, choosing consensus data, i.e. choosing most frequently occurring base in a given position when comparing multiple copies of a read, and detecting and handling the repeats. Finally, the number and the sizes of the contigs (or supercontigs) is used to indicate high or low quality of the assembled sequence.

## 2.11 Conclusion

The primary contribution of this chapter is to streamline and present an in-depth survey of the entire process of shotgun sequence assembly in the realm of high performance computing. It should be noted that there are many different methodologies and variations of processes and algorithms that we discussed and it is not feasible to encompass all that in this dissertation. It is obvious that there is a huge potential for enhancing both the speed and quality of genome assembly by employing parallel and distributed computing throughout the process. In this chapter, we have commented and pointed out specific areas with such potential. For certain stages, e.g. pairwise alignment and repeat finding, we laid the foundation for further research work by proposing high level parallel algorithms to enhance performance of the corresponding modules. As hardware cost decreases and processors and memory modules become cheaper, many of the currently sequential algorithms can easily be parallelized to improve speed. This in turn will allow for more complex algorithms to be implemented resulting in an overall improved quality of the genome assembly.

# CHAPTER 3

# A COMPARATIVE ANALYSIS OF MOST WIDELY USED PARALLEL GENOME ASSEMBLY TECHNIQUES

"I have never let my schooling interfere with my education."

Mark Twain

In this chapter, we will first revisit the shotgun sequencing technique and identify the tasks and the data they produce to feed the assembly programs. We will then explain the need for employing parallel computing and the criteria used to compare different parallel algorithms. Following that, we will define a frame of reference that outlines a generic sequence of stages of the assembly process. Next, the two commonly used approaches to genome assembly, Overlap Layout Consensus and Euler Superpath, and their parallel implementations will be discussed. Finally, we will present the scalability analysis of both approaches and results of an experiment using the Overlap Layout Consensus based implementation. This chapter is based upon our recently published research work [3] on this topic.

## 3.1 Shotgun Sequencing

As discussed in the previous chapter, the whole genome shotgun sequencing (WGSS) is a process of breaking up a DNA molecule into smaller pieces so it can be interpreted. The WGSS process typically results in millions of fragments, each consisting of 1000 or fewer bases, referred to as "reads". These reads carry very little detail on how to join them back

together to create the complete DNA blueprint of the specie being studied. The reads are usually obtained by embedding a piece of the DNA sequence called “clone” into a special host molecule, generally termed as a “vector”, which can be replicated in the laboratory to produce multiple copies of the clone. The clone is then separated from the vector extracting reads preferably in opposite directions noting the orientation and the distance between the two reads. This distance is called a “mate-pair distance” and provides an important piece of information used in the later stages of assembly. The extracted pieces are decoded in the laboratory using special processes, e.g. gel electrophoresis [89] as explained in previous chapter. The entire process is all but perfect and yields many poor quality reads, further adding to the difficulty in assembling these fragments. There are special programs, e.g. Phred [26], which evaluate the reads and assign a quality score to each base to quantify its accuracy in that position. The end regions of reads are usually of relatively poor quality. From a computer science perspective, the process of sequencing a DNA involves various steps many of which can be performed with special multi-phase programs called “assemblers”. A typical assembler: (a) reads a set of input reads along with the corresponding base quality scores, (b) detects the overlaps among reads, and (c) aligns the pairs with the most suitable overlap. The process is repeated until no more overlaps are possible, yielding a set of aligned sequences called “contigs”. These contigs are then oriented and arranged using multiple sequence alignment techniques yielding a “consensus” sequence. Finally, the relative position of each element in the array is validated using two important pieces of information, i.e. orientation and the mate-pair distance, both recorded during extraction of the reads earlier in the process. We will discuss the process in more detail in the following sections.

## 3.2 The Role of Parallel Computing

To compensate for the possible reading errors, multiple copies of the DNA strands are sequenced at the same time to allow for multiple overlaps. This is called “coverage” or

"redundancy factor", which is typically in a range of 5 to 12. The coverage or redundancy factor can be derived as (*n***l*)/*s*, where *n*, *l*, and *s* are the number of reads, average length of a read, and size of the DNA, respectively. Although the coverage improves the chances of getting better overlaps and hence more accurate results, it increases the data size by many-fold rendering the process computationally more intensive. This makes the sequencing an excellent candidate for parallel or distributed computing where multiple processors can work on intelligently divided subsets of the data concurrently. Each processor may share and utilize dedicated resources, e.g. memory and communication network. Processors may often communicate intermediate results among each other. As with any parallel program, the usual yardstick to measure the performance of the algorithm is "speedup". A speedup is defined as the ratio of sequential program execution time to the parallel program execution time. It should be noted that a parallel implementation (usually) incurs an overhead because of the communication among processors. A communication to computation ratio, denoted as CCR, provides a measurable characteristic of a parallel program. Moreover, because the parallel program employs multiple processors, it is important to evaluate the utilization of the extra resources, e.g. processors and memory, to justify the cost associated with using such a high performance computing environment. A key measure providing such a cost benefit analysis is the efficiency that is calculated as the ratio of speedup to the number of processors used in the underlying parallel computation. Finally, a critical factor for the success of a parallel algorithm is scalability, i.e. the capability to exhibit a consistent performance when the data size grows provided that an increase in the number of processors is also permitted. In other words, there needs to be a balanced relationship between the number of available processors and the size of the data required to efficiently utilize the available resources.

*3.2.1 Parameters Of Analysis And Measurement*

In this section, we discuss the metrics that will be used to compare the two parallel algorithms for genome assembly. As explained earlier, speedup, efficiency, and scalability are usually the three key metrics to measure the viability of a parallel algorithm. Grama, Gupta, and Kumar [33] proposed the term “Isoefficiency” to relate the number of processors to the input data size while maintaining efficiency, which allows measuring the scalability of an algorithm on a parallel processing system. Isoefficiency helps determine how much data should be increased to maintain efficiency within an acceptable range in response to an increased number of processors. Firstly, the overhead time can be derived in terms of sequential and parallel times as

$$T_0 = pT_p - T_1 \quad (3.1)$$

where $T_p$ is the parallel time taken by $p$ processors to process the input data and $T_1$ is the sequential time (using a single processor). Secondly, the input data, denoted by $W$ is directly proportional to the overhead time $T_0$.That is, the overhead time increases with the number of processors because the input data must be increased proportionally to maintain the efficiency. Using an efficiency based constant, denoted by $K$,

$$W = KT_0 \quad (3.2)$$

It can be observed that a smaller isoefficiency function means that the input data size can be increased in small increments for efficiently using a growing number of processors resulting in a highly scalable system whereas a larger isoefficiency function is interpreted as a poorly scalable system which requires disproportionately large increments of data for maintaining the efficiency when more processors are added to the system.

*3.2.2 Frame Of Reference*

For the purpose of comparative analysis of parallel algorithms presented in this dissertation, we will assume the underlying sequencing process to be based on WGSS as

explained before. The two most widely used assembly approaches are “Overlap-layout-consensus” [10], [42] and “Euler Superpath” [74].

The aforementioned approaches tend to reduce the problem into graphs such that a consensus sequence can be inferred by traversing either all of the nodes or all of the edges exactly once, resulting in a Hamiltonian or an Euler path, respectively. The subsequent text provides an in-depth discussion of the two approaches and their parallel implementations. It should be noted that a few other approaches, e.g. Genetic algorithms, e.g. [101], have also been devised but we will leave those out of this discussion. Moreover, we may also use the terms “approach” and “algorithm” interchangeably. The laboratory procedures are outside of the scope of this study.

It is important to understand that even though the objective of each of the algorithms presented here is the same, the logical flow and the number of stages in each of the algorithms are quite different. Therefore, it is not a straightforward one-to-one comparison of common set of capabilities. As a result, before we can compare two such approaches with respect to their parallel implementations, it is important to lay out a frame of reference using common standards and comparison criteria.

There are multiple variations of both approaches. However, for the purpose of our analysis, we will describe a more generic version that has all of the functional capabilities of the corresponding approach. We assume a common data set used as input with a total of $n$ input reads, each of an average length of $l$, with a base coverage of $c$. For simplicity, we will assume that the reverse complements of the reads also have been added to the mix and are included in $n$.

Almost all of the assemblers assume that the reads have been extracted and the corresponding base quality scores, mostly optional, have been calculated as prerequisites. Following is a list of high level tasks that are generally performed by an assembler. Figure 3.1 illustrates the process flow.

1. Iterate through the input reads and corresponding quality scores.
2. Detect overlaps by comparing each read $r_i$ with $r_j \forall\ i, j$: $0<i<n$, $i<j \leq n$. A comparison is two way, i.e. suffix of $r_i$ is compared with prefix of $r_j$ and also suffix of $r_j$ is compared with prefix of $r_i$.
3. Identify repeats consisting of either full or partial reads.
4. Align the pairs with most promising overlaps, resulting in aligned sequences.
5. Reiterate the above steps until no more overlaps are feasible, producing a set of long and contiguous segments called contigs that are disconnected from each other.
6. Lay out the contigs in order using the orientation and mate-pair distance information.

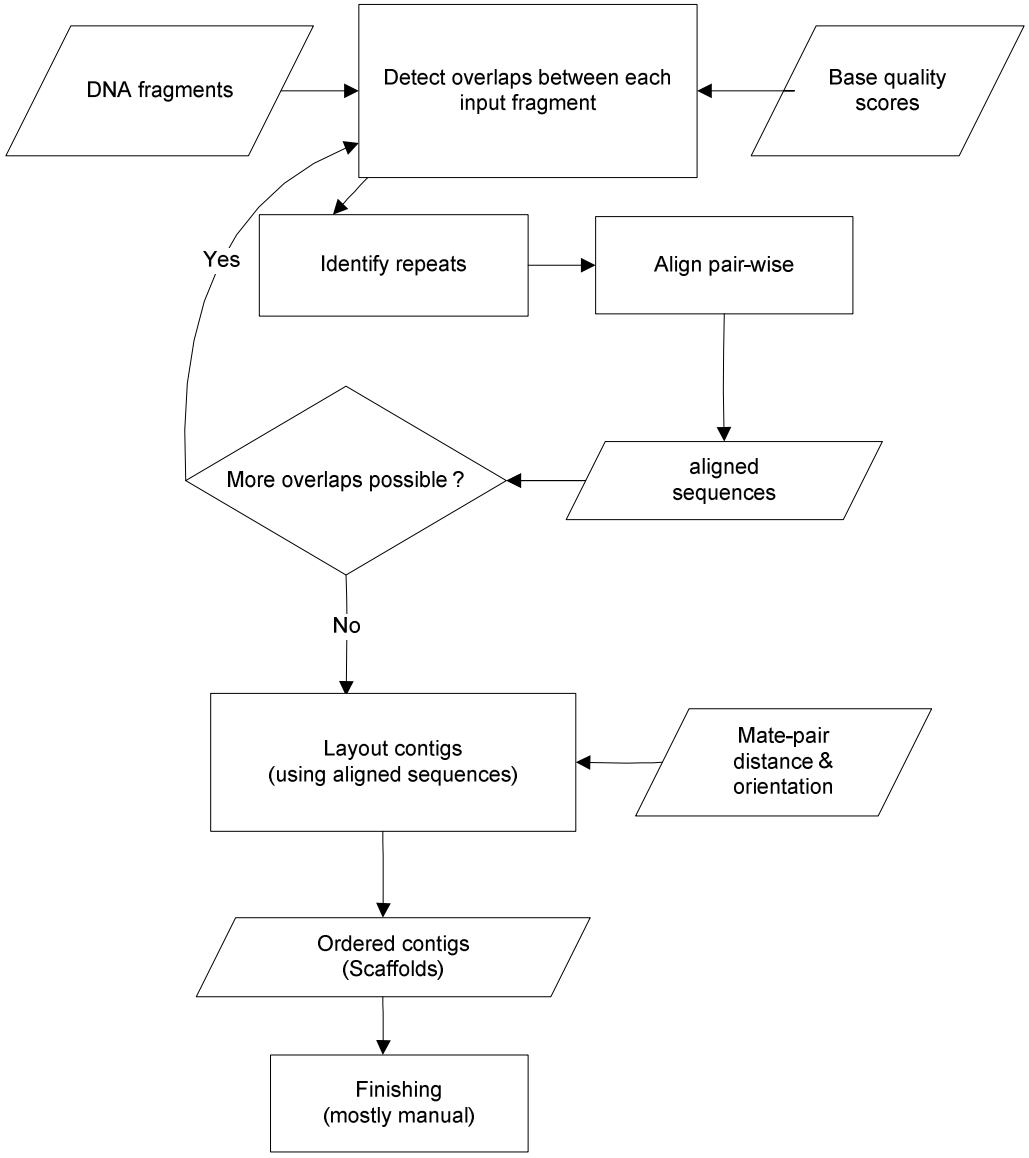


Figure 3.1 A generic process flow of assembly tasks showing some data and decision points.

In the subsequent text, we will discuss each approach with both sequential and parallel version of the corresponding implementation.

## 3.3 Overlap-Layout-Consensus

This is the most commonly used approach and a large number of assemblers have been implemented based on one or more variations of the overlap-layout-consensus. This approach, as evident from the name, comprises three high level stages. In the first stage, repeated regions and overlaps among the reads are detected. The repeats are either filtered out or masked by replacing each base value within repeat region with a special character using special purpose programs, such as RepeatMasker [87]. The process is reiterated until no further overlaps are detected. The outcome of this process is a set of contiguous segments, contigs that are finally arranged using the orientation and mate-pair distance information obtained during the earlier stages of genomic assembly.

We will now discuss the key tasks in both sequential and parallel implementations of this approach with a focus on deriving time complexity that will be utilized in measuring the isoefficiency. A few parallel versions of genomic sequencers have been published, e.g. [14], but we will look at a more generic implementation. Following are the various stages of such a generic algorithm employing graph theory.

### *3.3.1 Finding Overlaps*

Each input sequence $S_i$ is broken into a set $K_i$ of smaller subsequences of length $k$, called $k$-mers, such that $K_i = \{S_i[x][x+k-1]\} \ \forall \ x$, $1 \leq x < |S_i|-k+1$. All $k$-mers along with their origin information are stored in a sorted array. An overlap between a suffix of one and a prefix of another sequence is noted and can be readily observed as all such overlapping $k$-mers can be found contiguous in the sorted array. Unique overlaps are further verified using a local alignment algorithm, such as [88] or one of its several variants. For $n$ sequences yielding $m$ $k$-mers per sequence, a sequential algorithm to generate and sort $k$-mers and then align promising sequences takes asymptotically $O(nm + nm\log(nm) + \alpha n)$ time, where $\alpha$ is a product of genome coverage and the square of average length of a sequence to account for potential

alignments for each of $n$ sequences. A parallel implementation on $p$ processors of the same must asymptotically take $O(pT_s + nT_t + nm/p + (nm/p)\log(nm/p) + \alpha n/p)$, where $T_s$ is the start time and $T_t$ is the transfer time of a sequence between two processors. It must be noted that $T_s$ and $T_t$ are relatively smaller and insignificant compared to the actual processing times.

*3.3.2 Building Layout*

In most cases the layout is a directed graph $G$ with sequences laid out as vertices and the overlaps between two sequences as the edges. Once a directed graph has been constructed, the redundant edges are removed using transitive property. That is, an edge $u \rightarrow v$ between $u$ and $v$ is removed if two other edges $u \rightarrow z$ and $z \rightarrow v$ are found and the sum of non-overlapping prefix lengths of $u$ and $z$ in $u \rightarrow z$ and $z \rightarrow v$, respectively, is equal to the non-overlapping prefix of $u$ in $u \rightarrow v$. The reduced graph is to be traversed to find a Hamiltonian path, which is an NP-Complete problem. However, in practice, it is unlikely to find such a path from the perspective of both data and computation but the effort using heuristic algorithms results in subpaths that are recorded as contigs. In terms of time complexity, assuming a heuristic algorithm is employed, a scan of $nm$ entries constructs a basic graph and then removing the redundant arcs can be done in asymptotically $O(n^2m^2)$ time whereas a parallel implementation can achieve the same in asymptotically $O(n^2m^2/p + \log p)$ time, when ignoring latency during merge operations among processors.

*3.3.3 Finding Consensus*

Using the contigs produced above, a multiple alignment is performed to derive a consensus sequence. A recently proposed sequential algorithm by Edgar [23] would require asymptotically $O(nm\log n)$ time to align $n$ sequences assuming $m << n$. However, a parallel version would be able to achieve the same in asymptotically $O(nm\log n)/p$ time.

## 3.4 Euler Path Method

The Euler path based approach [74] reduces the genome assembly problem into an Eulerian path problem that lends itself to a polynomial computational complexity compared to the NP-completeness of the Hamiltonian path seen in the overlap-layout-consensus approach. The central idea of this algorithm is based on sequencing by hybridization where smaller reads are obtained leveraging a faster and cheaper sequencing process. It must be noted that the same concept could also be applied to the relatively larger sized reads obtained through Sanger sequencing [83]. For example, a *k*-mer can be split into *l*-mers such that suffix of $l_i$ overlaps with the prefix of $l_{i+1}$ for a length of (*l*-1) $\forall$ i, *l*: $0<i\leq k-l+1$, *l*<*k*. Using all such *l*-mers, a directed deBruijn graph [77] is constructed with suffix and prefix as vertices and the *l*-mer as the edge connecting suffix to the prefix. Ideally, a path traversing all such edges, called Eulerian path, would include all *l*-mers yielding a fully reconstructed genome. However, practically, a set of paths is obtained representing multiple contigs due to the presence of repeats and inaccurate data. The algorithm also attempts to identify and remove repeats by finding directed paths where the indegree of the starting vertex and the outdegree of the ending vertex are both greater than 1 whereas all intermediate vertices have both indegree and outdegree equal to 1. Using the relation and position of all of the *l*-mers in such path, within the original reads, the algorithm distinguishes the repeats from overlaps.

A parallel implementation of the algorithm has been reported by Shi and Zhou [86]. We will now discuss the key tasks in both sequential and a generic version of parallel implementation of this approach with a focus on deriving time complexity that will be utilized in measuring the isoefficiency to help analyze the scalability of the system. Following are various stages of such a generic algorithm employing graph theory based on ideas originally presented by Pevzner [74].

### *3.4.1 Generating K-mers*

Considering each *k*-mer generation as a single operation, the time taken to generate all *k*-mers is proportional to the product of *n* and *m*. A parallel implementation will asymptotically take O(*nm*/*p*) time.

### *3.4.2 Distributing K-mers*

This is an extra step in the parallel implementation. However, using a hashing scheme, the procedure takes near linear time, i.e. O(*nm*). In the parallel version, each processor keeps some *k*-mers to itself and distributes the others based on the hash value. This may result in some communication overhead but the time complexity in this stage is asymptotically equal to O(*nm*/*p*).

### *3.4.3 Preparing Data*

The algorithm requires the multiplicities of *k*-mers to be computed for later use as a boundary condition when traversing paths. In the sequential algorithm, which is achieved by using suffix arrays to store and use *k*-mers, it takes O(*nm*log(*nm*)) time. In a parallel environment, the multiplicity of each *k*-mer can be calculated as a side step when hashing and distributing the *k*-mers, incurring insignificant time. If the coverage *c* is not available, then it takes asymptotically O(*nm*) to compute the coverage. This can be achieved using some fast string matching algorithms, such as the Gusfield's Z-algorithm [38].

### *3.4.4 Building deBruijn Graph*

This task requires the bulk of computing power and time. Hashing may help find the adjacent *k*-mers faster; however, quadratic time is required to compare all *k*-mers with each other. A slight improvement would be to use the fact that each one of the *n* sequences was broken into a total of *m* k-mers. Therefore, those *nm* edges could be constructed without even

comparing with each other. That results in $nm(nm\text{-}1)$ or asymptotically $O(n^2m^2)$ runtime complexity.

### *3.4.5 Traversing The Euler Path*

An Euler path can be identified in $O(E)$, where $nm < E < n^2m^2$. A parallelized version should take $n^2m^2/p + pT_s + nT_t$, i.e. asymptotically $O(n^2m^2/p)$ time. The Euler traversal of the graph provides the solution, i.e. sub-paths that can be recorded as contigs. The remaining work that includes scaffolding and finishing is generic in nature, which mostly is performed sequentially. Therefore, we consider that out of our scope of work.

## 3.5 Analyzing Scalability

We will use the time complexities we noted above along with other criteria for evaluating the scalability of these approaches in the following sections.

### *3.5.1 Isoefficiency Comparison*

An algorithm is usually characterized by its time complexity that represents a relationship between runtime and the input data. In case of a parallel algorithm, the speedup, a runtime comparison of the execution speed of the parallel algorithm to its sequential version, is normally used as a metric for measuring performance. The efficiency or the speedup per processor, measures the fraction of time a processor spends in useful computation and therefore provides a gauge of utilization that characterizes a parallel algorithm. Assuming a constant data size $W$, the efficiency decreases with the increasing number of processors. This is due to the fact that more processors are available to do the same work that was previously achieved by fewer processors therefore causing lower utilization. It also results in more communication overhead and start-up times, denoted together by $T_0$ as correlated with input data in equation 2 above. A parallel algorithm is considered scalable if increasing resources,

processors in most cases, requires only a proportional increase in the input data to keep the efficiency from decreasing significantly.

The sequential time complexities for different phases of overlap-layout-consensus approach can be summed up as:

$$T_1 = nm + nmlog(nm) + \alpha n + n^2m^2 + nmlog(n) \quad (3.3)$$

whereas a parallel version would result in

$$T_p = pT_s + nT_t + \frac{nm}{p} + \frac{nm}{p}log\left(\frac{nm}{p}\right) + \alpha\frac{n}{p} + \frac{n^2m^2}{p} + log(p) + \frac{nmlog(n)}{p} \quad (3.4)$$

Using equations (3.1) and (3.2), assuming p>>1, and ignoring any lower asymptotic factors, we simplify to obtain

$$W = K(p^2T_s + pnT_t + plogp) \quad (3.5)$$

A similar workout for the Euler path approach yields

$$W = K(p^2T_s + pnT_t + nmp) \quad (3.6)$$

It should be noted that all factors of the $T_0$ side of the equation contribute to the isoefficiency computation. However, we are more interested in the component that causes the data size to grow at the fastest rate with respect to *p* [33] and it follows that, asymptotically, both equations (3.5) and (3.6) have a quadratic isoefficiency in terms of the number of processors. Hence, we can deduce that an increase in the number of processors from *p* to $p_{new}$ would require input data to be increased by a factor of $p_{new}^2/p^2$. It can be observed that these parallel implementations help decrease execution time and are reasonably scalable considering a small to mid-range parallel computing environment.

## 3.6 Experiment and Results

To test and validate the time complexity analysis discussed above, we ran some tests using a modified version of a parallel implementation based upon the Overlap layout consensus

approach using C programming language in MPICH environment [36], an implementation of Message Passing Interface (MPI) framework, using up to 32 nodes in a cluster environment. The cluster runs RH-Linux with an MPICH version 1.2 with a 2GB memory per node. The raw traces of *Drosophila Yakuba* were obtained from ENSEMBL [44] along with the corresponding quality scores. A multi-phase program was run and total runtime was used to compute efficiency at different data points (see Figure 3.2). Each data point represents number of processors from a range of 2 to 32 and the size of the dataset from 13Mb to 200Mb. Most of the data points validate the Isoefficiency derived in equation 3 in the previous section. For example, doubling the number of processors in this case requires almost four fold increase in the data size to maintain the efficiency. At few points, a slight deviation is observed that can be attributed to the communication factor. It should be noted that a similar parallel implementation of Euler path method was not available at the time of this.

### *3.6.1 Communication-to-Computation Ratio (CCR)*

A closer look at the above relations reveals that the communication factor is significant in both approaches. Both employ graph theoretical solutions that tend to be communication intensive due to complex and interrelated paths where vertices and edges are spread across multiple processors. The parallel version of Euler path approach has to process a relatively larger number of vertices and edges due to the nature of deBruijn graph, which is centric to the whole approach, resulting in a large CCR compared to the overlap-layout-consensus approach. It must be noted here that a mere comparison of the CCR between the two approaches does not necessarily favor one or the other algorithm; however, it affects the scalability as the number of processors grow larger.

*3.6.2 Space Comparison*

Although most of the metrics used in the analysis of parallel algorithms are time-centric, i.e. focus on time derivatives such as speedup and efficiency, the space complexity is also a critical factor in evaluating scalability. A general discussion on the subject can be found elsewhere [78]. For the purpose of this high level discussion, we can observe that both approaches can effectively utilize space using a good hashing scheme; however, a larger number of vertices and edges in Euler approach generally require more memory space.

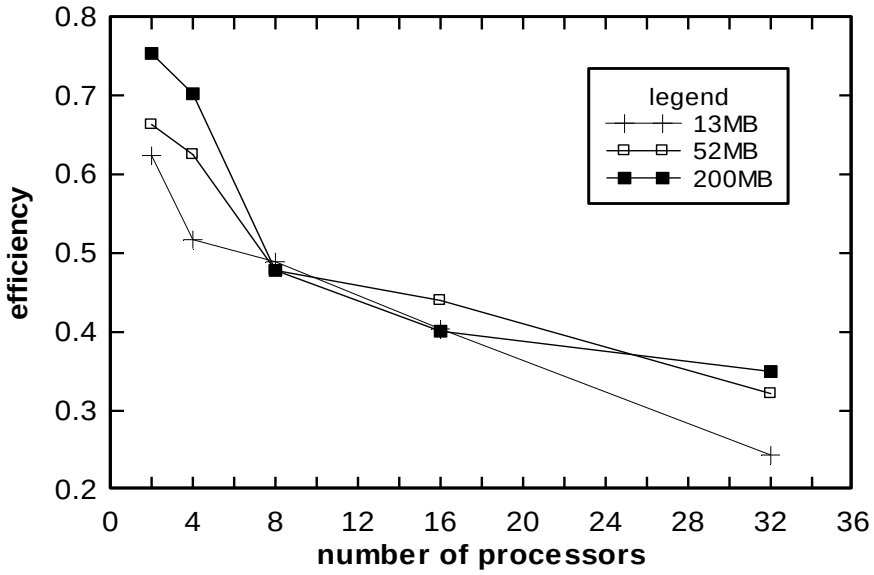


Figure 3.2 Efficiencies for different datasets using parallel Overlap Layout Consensus based implementation.

## 3.7 Conclusion

Most published work has analyzed parallel software programs based upon various assembly algorithms by comparing runtime, lengths of the contigs, and error rate. There is a need to provide some additional evaluation of the algorithms, such as their efficiency and

scalability for varying data size and complexity. In this chapter, we presented a high level approach to analyze and compare two different methodologies used in genome assembly process to establish the importance of, and to provide an insight of, the relationship between the size of the input data and the number of processors available using generic parallel algorithms for the corresponding methodologies. Although a detailed comparison of different implementations is not feasible due the fact that each assembler uses unique components to improve speedup, accuracy, or handling of large data sets, having such insight of the overall approach should enable bioinformaticians to select a class of algorithms based on the type and size of the data they are working with and to study the cost effectiveness of a proposed parallel system.

# CHAPTER 4

# A PARALLEL FRAMEWORK FOR SCALABILITY ANALYSIS AND MULTI-OBJECTIVE OPTIMIZATION IN WHOLE GENOME SEQUENCE ASSEMBLY

"The width of life is more important than the length of life."

Ibne Sina

As we have discussed in the previous chapters, parallel computing can be leveraged to help process large volumes of data expeditiously. We have also presented scalability analysis of two widely used approaches to DNA sequence assembly. In this chapter, we propose a holistic approach to simultaneously achieving better speedups and improving the accuracy using error correction logic in a high performance computing environment. By leveraging the extra processing power available in a parallel computing environment, we propose an approach to correct errors in the weaker end regions as opposed to trimming them off thereby enhancing the accuracy of the solution. The speedup is improved by dynamically balancing the load among multiple processors and utilizing innovative data structures along with a hashing technique that require lesser memory compared to some other contemporary programs. This allows us to carve out an important and sizeable piece of assembly process to design and implement a distributed framework that lends itself to scalability analysis while providing a platform to study different characteristics of a bioinformatics problem to help understand various trade-offs and interdependencies among multiple and conflicting objectives such as processing speedup and accuracy. Finally, it aims to provide a foundation for more research work in related

areas of bioinformatics, e.g. multiple sequence alignment and phylogenetics, especially in the realms of high performance computing.

## 4.1 Problem Statement

Based on our earlier discussion of the shotgun sequencing and the need for employing parallelism in the area of DNA sequence assembly, we can begin with stating the multi-faceted problem, which will help define the essence of our research work as follows. Given a large number of small size DNA segments and some preliminary information, e.g. base pair distances, base quality scores, and segments' orientation, the objective is to *reconstruct the sequences as few in numbers and as large in size as possible while minimizing the errors and the processing time*. In this work, we focus on one of the critical phases of sequence assembly where overlaps among the input segments must be detected. While improving speed to process a large set of data, our algorithm attempts to enhance accuracy of the overall solution by correcting errors in the low quality regions of the input segments. This aligns very well with our overall research goal of optimizing inter conflicting objectives, i.e. speedup and accuracy in relation to the changing data size. In the following sections, we will lay out the details of our algorithm and the results of the experiments based on that approach.

## 4.2 Key Issues

In order to understand the importance of the subject of research, it is useful to identify the key issues faced by the researchers in this field. Following is a brief overview of some of these issues.

### *4.2.1 Time Consuming*

Due to large volumes of data and the complex alignment and merging techniques required, the algorithms, which are generally used to assemble sequences, take significant time

to process input/output (I/O) and perform the computation. For large sized genomes, it is not uncommon to see an execution time of several days and even weeks, discouraging a quick "what-if" analysis of the problem at hand.

### *4.2.2 Erroneous Data*

The sequencing procedures [83], used widely in the laboratories, have limitations that result in less than perfect data. The reads obtained at the end of these procedures are usually of good quality in the center region but the end regions are unreliable and are therefore assigned a low quality score by quality assessment programs, e.g. Phred. Most assemblers trim such end regions to ensure that the data used by assembly algorithms is mostly free of errors. Others attempt to use whole reads and employ error correction schemes that are less than perfect and require comparatively more processing time. These errors may cause a mis-assembly if left unaddressed and result in a larger number of small contigs.

### *4.2.3 Error Prone Process*

In addition to the erroneous data, the process itself is prone to different errors that are introduced while sequencing, quality score assignment, overlap detection, pairwise and multiple alignment and many other phases of the assembly process. For example, aligning *n* sequences optimally requires that an *n*-dimensional matrix be built and traced, which is a very time and memory consuming process. Therefore, heuristic algorithms are used that enable faster processing and produce quick results. However, the possible errors introduced during various phases of the process often lead to an incorrect solution. Furthermore, many genomes are repeat-heavy, i.e. part of their segments repeat themselves many times throughout the genome and may be present in different reads. The assembly program may mistakenly treat them as copies of same segments and merge unnecessarily. All such issues affect the end result and therefore need to be addressed efficiently as early as possible in the process.

*4.2.4 Conflicting Objectives*

It can be easily inferred from the above discussion that sequence assembly is a complex computational problem and there is room for improvement in various dimensions. For example, a short execution time, ability to handle very large data sets, enhanced accuracy, and the ability to scale up the system are some of the most critical needs that should be addressed. Figure 4.1 illustrates the most important yet often conflicting objectives.

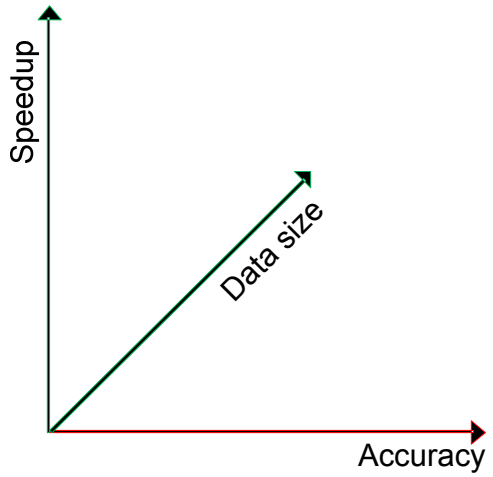


Figure 4.1 Objectives that often conflict with each other.

These are the primary objectives that call for parallelization to achieve faster execution, handling of large data sets and improved accuracy. However, optimizing one often impacts the others adversely because most of these objectives depend upon the same set of decision variables with one objective directly proportional to a variable while another is inversely proportional to the same variable. Analyzing scalability in the context of these complex inter-relationships is the primary focus of our research.

## 4.3 Our Approach and Algorithm

Now that we have discussed different objectives and their inter-dependencies, we can state the goals and scope of this research as follows. We first define a set of tasks that can be

carved out of the WGSS assembly process to be designed and implemented as a new parallel system. For our newly developed parallel system, we analyze the algorithm and its time complexities; derive functions for speedup, efficiency and Isoefficiency followed by experiments where we run several sets of genomic data through our program and compile the runtime statistics to be compared with the theoretical analysis. This allows us to study the behavior of a parallel bioinformatics algorithm. The results and conclusions drawn from the study can be used as an example for other similar and related areas as explained later in this chapter. In this section, we will first describe the scope of our work followed by a detailed account of the algorithm to enhance accuracy using an innovative error correction scheme and optimize overlap detection by leveraging distributed computing. As we will be using parallel performance metrics like speedup, efficiency and Isoefficiency [33] for the scalability analysis, we will use the concepts and equations we referenced in previous chapter as below. Firstly, the overhead time can be derived in terms of sequential and parallel times as

$$T_0 = pT_p - T_1 \quad (4.1)$$

where $T_p$ is the parallel time taken by $p$ processors to process the input data and $T_1$ is the sequential time (using a single processor). Secondly, the input data, denoted by $W$ is directly proportional to the overhead time $T_0$.That is, the overhead time increases with the number of processors because the input data must be increased proportionally to maintain the efficiency. Using efficiency based constant, denoted by $K$,

$$W = KT_0 \quad (4.2)$$

*4.3.1 Scope of The Proposed Work*

Sequence assembly is a multi-stage process in which reads, along with their quality scores, pairing and orientation information, are used to find overlaps to help align a large number of reads into a small number of contiguous segments called “contigs”. At a more detailed level, repeats are also detected to avoid an incorrect assembly. Moreover, low quality

regions of the reads are either trimmed off or an attempt is made to identify and correct errors within these regions. The overall process consists of very complex set of computation intensive tasks. Our research is focused primarily on the overlap detection and error handling logic which helps enhance accuracy and achieve faster processing thus allowing larger datasets to be processed more frequently. Figure 4.2 highlights the scope by showing a sample of reads and the effects of error correction logic and clustering in the context of a larger assembly process.

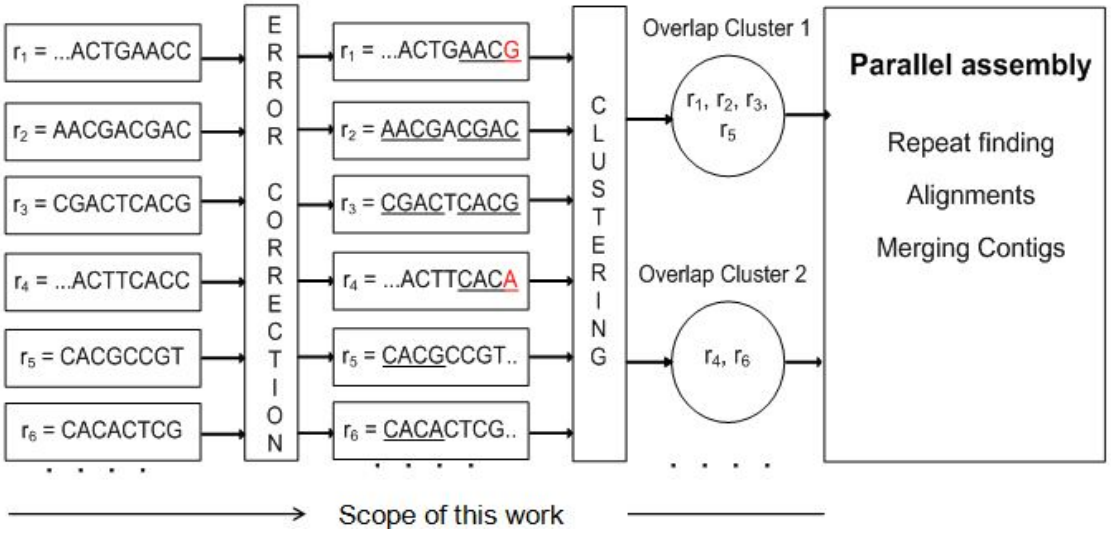


Figure 4.2 A sample of 6 partial reads shown here to illustrate effect of error correction and clustering using a minimum overlap length of 4 bases. The reads r1 & r4 have been corrected using error correction scheme without which both reads could have been left stand alone or clustered with incorrect groups.

*4.3.2 The Algorithm*

This section provides a high level description of the algorithm that fragments the reads into small k-long tokens called k-mers, where *k* is a small number ranging from 10-25, corrects errors, detects overlaps and finally clusters the input reads to facilitate parallel processing of the subsequent tasks that are shown in Figure 4.2. Following are some high level tasks accomplished by this algorithm.

- Compute reverse complements (optional).

- Tokenize the reads into small *k* long fragments called k-mers.
- Identify weak regions of each read
- Enhance accuracy by correcting errors in the weak regions
- Find overlaps among reads
- Cluster the reads that overlap directly or transitively

The main contributions of our work include an innovative error correction scheme, a fast clustering logic using smart data structures, and dynamic load balancing in a distributed computing environment. More importantly, we present an approach to analyze scalability and optimize multiple objectives of a parallel system. The following sections explain the main tasks of this algorithm, i.e. identification of weak regions, error correction scheme and clustering logic, followed by a detailed scalability analysis of multi-objective optimization.

#### 4.3.2.1 Identifying The Weak Regions And Their Partner Images

It is important to provide some concepts and definitions used here including the notion of a *weak region* and its *partner image*. In the following section, we define these terms in more detail showing some examples.

*(a) Weak Region*

As explained previously, the end regions of the reads are inherently of low quality due to the imperfection of the sequencing methods, i.e. a read of length *l* may have *X* number of bases in the beginning and *Y* number of bases at the end that can be considered weak or unreliable. Let $T_q$ be the minimum threshold value of quality scores under which a base is considered weak, $T_w$ be the minimum number of weak bases having a quality score of $T_q$ or less within a window that is $T_y$ bases long. In other words, a weak region is identified when a certain

number of bases ($T_w$) with quality scores less than the threshold ($T_q$) are found in a span of a fixed length ($T_y$), $\forall$ $T_q$, $T_w$, $T_y$: $0 < T_q \leq 50$, $T_w \leq T_y \leq l$. An optimal quality score for a base is 50, i.e. an accuracy of 99.999% or 1 error in 100,000 bases. Figure 4.3 shows a read, its k-mers and a weak region *r* at the end of the read.

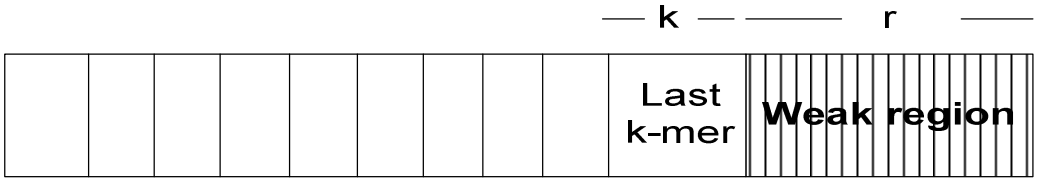


Figure 4.3 A typical read, k-mers, and a weak region *r* at the end of the read.

*(b) Partner Images*

A partner image within a sequence *f* is a subsequence $S_i$ of length *i* that flanks another subsequence $S_k$ of length *k* such that there exists another k-mer within a sequence *g*, which precedes a weak region, and can be aligned with $S_k$ by extending both k-mers by a user defined parameter *d*. The extension is used to enhance reliability of a partner image. Figure 4.4 shows an example of how a partner image can be found.

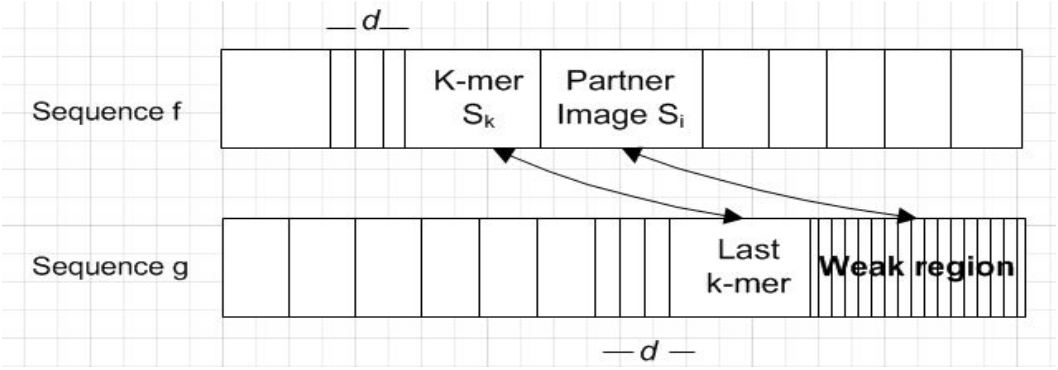


Figure 4.4 Two sequences *f* and *g* and the relationship between weak regions and their partner images.

The weak regions are identified at the same time when a given sequence is being k-merized, i.e. broken into k-mers and stored in a local hash table. The k-mers that are

immediately followed by a weak region are flagged. This process is described in more detail in the following section.

4.3.2.2 Creating, Hashing and Storing the k-mers

Breaking the reads into k-mers is a widely used method to simplify and expedite the overlap detection. It has been used by different assemblers and an optimal value of *k* has been proven to be in the range of 10 to 24 [10], [68]. Our algorithm provisions a set of user defined parameters, including the value of k, to allow benchmarking and fine tuning of the solution.

The master processor evenly distributes the input segments to all the available processors to generate the k-mers. In an environment where each processor can concurrently read the input data from same location, each processor works on the data according to its rank. For example, processor *i* reads each segment *n* such that $n\%p = i$ where *i* is the processor rank, % denotes a mod operation, *p* is the total number of processors and *n* is the sequence number of a given segment. For a given segment *s* of length *l*, k-mers are generated by tokenizing the string into $l-k+1$ tokens with the first one being s[1] through s[k], the second one containing s[2] through s[k+1], and so on. The last k-mer is composed of the last *k* characters of the segment. This generally yields $l-k+1$ token. Mathematically, given a set *S* of sequences, each input sequence $S_i$ is broken into a set $K_i$ of smaller subsequences of length *k*, called k-mers, such that $K_i = \{S_{i[x][x+k-1]}\}$ for all *x* where $1 \leq x < |S_i| - k$.

The k-mers can be optionally generated for the reverse complements of the input segments, if not already part of the data. The set of reverse segments is derived at runtime. For example, a reverse complement of ACTG is CAGT where the string is not only reversed but also A replaces T, G replaces C and vice versa. All processors generate k-mers as above and, using a hash function, either place them into a local hash table or send it to another processor determined by the hash value. Each processor is assigned a range of hash values so that all the overlapping k-mers are stored in one processor. For example, each k-mer yielding a hash

value of 100 is stored in $P_1$ assuming that $P_1$ has a range of 1 through 100 or more. The entry to be stored does not contain the k-mer string itself. The attributes required are (1) start position in the segment, (2) the segment identifier, and (3) the orientation, i.e., forward or reverse complement. Also, to resolve the collisions, the entry contains a pointer to a linked list of nodes having similar structure so more than one k-mers hashing onto same index can be stored. The pointer initially points to null.

The advantage of using a k-mer approach compared to simple matching of the suffices with prefixes of all segments is that the k-mers are generated by a sliding action, i.e., each position of the segment is scanned one at a time and a k-mer is generated which can inherently account for the Inserts and Deletes, also referred to as *InDels* in sequences, as the two segments can likely share a combination of k-mers even if one or both had some insertions or/and deletions. That is a major advantage of k-mer approach over comparing fixed length prefixes vs. suffices as it reduces the computations needed by checking various patterns of a k-mer with gaps in every possible position. Also, overlapping segments can be easily found by simply sorting the k-mers resulting in consecutive k-mers from the overlapping regions of the segments. The time taken to read and then generate k-mers for all *n* segments by *p* processors for a segment of length *l* can be given in terms of time complexity as

$$O(n\frac{l}{p}) \tag{4.3}$$

*Hash function and table allocation*

The hash table plays a very important role in storing and evenly distributing the k-mers to processors as described below. The hash function needs to be such that it can produce a unique hash value for a given k-mer and provide for a space and time efficient hashing and resolution to any collisions. We will use the following hashing algorithm which is suitable for small k, i.e., 12-15. It should be noted that the following discussion assumes dataset to contain

DNA sequences though it can also be applied to Protein sequences with slight modification in the hash function.

Let *s* be a k-long segment to be hashed and f(x) be a mapping function that maps a member *x* of the alphabet $\partial$ = {A, T, C, G} onto a number such that f(A) = 0, f(T) = 1, f(C) = 2, and f(G) = 3. The hash index for a given k-mer is calculated as

$$\text{Hash (k-mer)} = \sum_{i=0}^{k} (4^i * f(s_{k-i}))$$

The time complexity of hashing a k-mer is constant or simply in $O(k)$ (4.4)

Each letter of k-mer is translated into a number based on its position and the letter itself. For example, *A* is translated into 0 no matter where in k-mer it appears. The *T* gets translated into a 1, 4, 16, 64, and onward that is always $4^i$ where *i* is the position of base *T* in the k-mer from right and ranges between 0 to k-1 for a k-mer. Similarly, *C* and *G* are translated to $4^i * 2$ and $4^i * 3$ respectively. For example, a k-mer (k=5) ATAAT is translated into 0+64+0+0+1 = 65. Similarly, TAACG is translated into 256+0+0+8+3=267. All such numbers are added together to result in a number ranging from 0 to $4^k$-1. Therefore the hash array size needs to be at least $4^k$. For k=12, the required array size is 16,777,216. Depending upon the structure of the hash array elements, this number may be feasible for small size computing environment. However, the actual number of k-mers resulting from the input segments may be quite large resulting in several identical k-mers competing for the same element index of the hash array. This collision can be resolved by maintaining a linked list of all such subsequent k-mers pointed by a k-mer hashed onto a given index. Hence the hash array elements structure should support pointers to the similar type of structures.

```
if(iLocalIndex != myproc) // Need to send it to processor number iLocalIndex */
{
        RecToSend.hash_index=lHashVal;
        RecToSend.segment_id = iSegId;
        MPI_Send(&RecToSend, 1, comm_rec, iLocalIndex, myproc, MPI_COMM_WORLD);
}
else   /* This kmer belongs to local proc so it will be stored locally */
        InsertHash(lHashVal, iSegId);

/* Now receive from others (if any) */
ReceiveAnyMessages();
```

Figure 4.5 Inserting hash table record locally or sending it to another processor

A typical structure of the linked list node should contain a segment identifier and a pointer to the next node. Since the algorithm does not depend upon the order of the nodes within the linked list, we always insert a new linked node at the head of the linked list improving the performance of this phase. The structure of the array element, which also stores a segment identifier segment_id and a pointer to the first node of linked list, or an initial value of null, should additionally have a special field called “parent_id” which stores the largest of all the segment_ids hashed and stored in that particular hash index and its corresponding linked list. In case of a single element, the segment_id is same as parent_id as initially set at the time of first k-mer’s hashing onto that index. This parent_id allows the hash array to be sorted in decreasing order such that all array records that have a non-null data are on the top so we can ignore all null record indices in later processing thereby saving time. Additionally, the parent_id is used in implementing the Union-Find structure [5] to help cluster the overlapping segments as explained in the following section. We also need to store segment_id so that at the end of this hashing we can scan the hash array and cluster all the segments that are linked by the link list. Initially, the pointers will be set to null.

As we find 2$^{nd}$ and subsequent k-mers hashing onto an index that has been previously used, a new node will be added pointed by the pointer of that previous k-mer element of the array. Figure 4.6 illustrates a typical entry in the hash array (array element shown in gray and linked list shown in red) with 2$^{nd}$ element storing two identical k-mers, which may or may not be from same segments, a reason why we always store corresponding segment identifier. The 1st

& 3rd elements are shown to store only one k-mer. The number of k-mers can be large for a medium to large size genome sequences and can be calculated as *l*-*k*+1. This hashing scheme which is based upon the positions of bases allows us to group repetitive k-mers to allow a quick detection of repeats.

| Parent Id | Segment Id | Start pos | Orient | Weak Follower | Next |
|---|---|---|---|---|---|
| 101 | 101 | 511 | F | Y | Null |
| 101 | 99 | 231 | F | N | -> |
| 88 | 88 | 150 | F | N | Null |

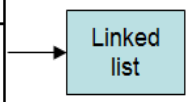


Figure 4.6 Three sample records in the hash table. Some other fields, e.g. partner images, also dynamically allocated, are not shown here for simplicity.

For example, a 700 bases long sequence results in 700-12+1=689 k-mers for k=12. A small genome like Ciona savignyi with approximately 271,000 sequences can yield about 190 million k-mers. Additionally, coverage of 5 to 7 times of a genome can result in quite a large number of sequences to work with. Finally, that large number is doubled when reverse complements are added into the mix. A cluster of processors can be helpful in handling large dataset and in balancing the workload. Hence, a reasonable size of the required memory for assembly of a small genome like Ciona savignyi would be of the order of several gigabytes. Such large size memory requirement is not uncommon in genome assembly programs. The time taken for hashing and storing a k-mer is proportional to its size k which is constant and is therefore *O*(1). Each processor works on a set of n/*p* segments and, assuming an averagely uniform data set, stores its share, i.e. $\frac{1}{p}(\frac{nl}{p})$ k-mers, in its local hash table. The complexity of time for building each hash table can be inferred as O($\frac{nl}{p}$) for *n* sequences and *l*>>*k*. The time to

append a k-mer to the linked list in case of a conflict is also constant as we always place the new record as the head of the list shifting the previous head to the second position in the list. The time complexity for k-mer creation and storage can be expressed as

$$O(n\frac{l}{p}) \quad (4.5)$$

4.3.2.3 Error Correction Scheme

Once the data has been tokenized into k-mers, the weak regions have been identified, and the hash table has been built, the algorithm processes all such k-mers that are flanked by weak regions. For simplicity, we will only describe the process for the weak region at the end of the read. A similar process with reverse orientation handles the weak region at the beginning of the read. Following are the key steps described at a high level.

- For every input segment in the data set, traverse through the hash table as described in the following steps. This time, each processor scans all segments as the local hash table of a given processor may include k-mers from any segments.
- For every $k_i$ that has a weak follower found in $i^{th}$ position (in main cell or linked list)
    - Using all k-mers stored in $i^{th}$ index (including linked list), find the right flanks called "partner images".
    - Validate partner images by extending k-mers by a delta, i.e. *d* positions to the left, like FASTA [60], to ensure a true overlap where *d* is an optional input parameter.
    - Using the valid partner images, compare the bases in the weak region and find the best possible value for each position.
    - Sum up the quality scores of each base in a given position for all values.
- After all segments have been re-processed and all possible partner images have been located and used to compare the bases in the weak region, the base value with the highest sum is used as the corrected value. See Figure 4.7.

- More k-mers (including corrected positions) are generated from weak regions and added to the mix.

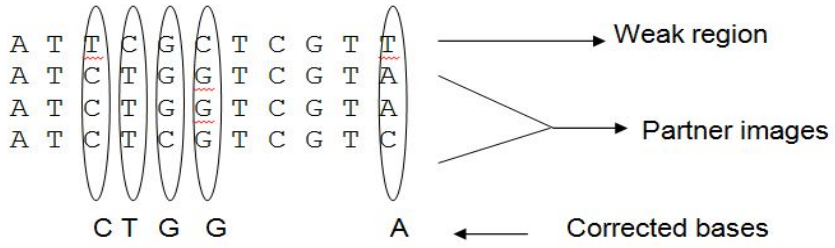


Figure 4.7 Error correction using partner images of the weak region.

As discussed in the above section, a uniformly distributed dataset may yield an average of nl/p records stored in the local hash table of a given processor. The error correction logic phase scans a significant part of local hash table for each of the *n* segments and therefore results in a quadratic processing time, i.e.

Time complexity of error correction logic can be estimated as $O(n^2 \frac{l}{p})$ (4.6)

4.3.2.4 Clustering of Overlapping Reads

Each processor iterates through the list of k-mers stored locally and groups the corresponding segments of the overlapping k-mers into clusters. A cluster is a set of segment identifiers which have one or more overlapping k-mers directly or transitively. For example, if a k-mer from segment $S_1$ overlaps with one or more k-mers of $S_2$ and furthermore some k-mer(s) of $S_2$ overlap with those of $S_3$ then $S_1$, $S_2$, and $S_3$ all belong to the same cluster. This step involves finding all such overlaps and grouping the segments together into clusters, e.g. $C_1$ = {$S_1$, $S_2$, $S_5$, $S_8$} and $C_2$ = {$S_3$, $S_4$, $S_6$, $S_7$} as in Figure 4.7 and so on.

*(a) Enhanced Union-Find implementation*

In order to group the segment identifier (segment_ids), a processor iterates through its hash array. For each array element that has at least one extra node in the linked list, it proceeds as follows. For those without any linked list, it ignores them as such entries do not help in any way in clustering and are therefore referred to as “stand-alone” elements. It is, however, quite possible that the same segment_id appears in another hash array location along with some other segments due to another k-mer of such segment.

Let us define a hash array element. An element of the array is either the record that is stored in the hash array at a given index or it is one of the many possible nodes that are dynamically created to resolve hash collision, i.e., they all hashed to same index and could belong to different segments. Therefore, an element is any of the segment_id that is stored in a given hash index location whether in the array position or the corresponding linked list. Each hash array element has a parent_id that identifies the largest segment_id stored in that hash index.

The array is sorted by the parent_id in a descending order. The ordering of the array is critical to the algorithm as it ensures that a search and union operations are performed only on the current and a preceding index and not on any following indices. This helps reduce the processing time significantly as the Union-Find operations are very fast for lower indices and so the average time to find a match and then perform a union remains low compared to an unordered dataset. It is also important to note that each array index with a non-null data has a parent_id designated to it. For a given element of a hash index including the segment_id stored in the hash record itself as well as those stored in the corresponding linked list (as in Figure 4.6 above), iterate through the hash array as an inner loop and find a matching element in the preceding indices of the hash array. If a match is found then the current parent_id of the outer loop is replaced with the newly found parent of the matched segment in preceding index. If the two parents happened to be same then that index is ignored for that particular pass, i.e., none

of the elements stored at that index are considered for matching. Once a parent_id is replaced, it is important to update all indices of the hash array, which have the same parent_id that was just replaced, with the new value. This ensures that all such indices share the same parent_id and can be subsequently retrieved as part of the same cluster.

After all the above processing is done, the hash array is sorted once again by the parent_id in a descending order. The QSORT operation used in the program sorts only the non-null elements of the array as dictated by a variable that is set dynamically at runtime. This variable keeps the total number of non-null locations in the hash array that are on the top of the array due to the sorting that we performed earlier. All null records, where no k-mer was hashed to, stack up below the non-null records. This helps in data processing and speeds up the program significantly. A statistical analysis shows that the number of array elements to be processed in some cases can be reduced by up to 95% using this technique, which leads to a significant saving in subsequent processing. In one instance, for example, an array of 145971 records was reduced to 1852 records which is about 1.3% of the size of the initial array size.

Each processor then traverses the hash array in order and retrieves the clusters by simply collecting all segment_ids that share a parent_id as the array is sorted by parent_id. The processor then sends all clustering information to the Master processor that will perform the final clustering on the incoming data as described in the following section. The above approach can be compared to approaches taken by other assemblers where whole segments are read, stored and compared. Our approach is equally fast in terms of time complexity but uses considerably smaller amount of space as it does not store the whole segments.

```
for(lHashCtr=0; lHashCtr<lProcHashSize; lHashCtr++)
{
  iClusterId=hash_array[lHashCtr].parent_id;  /* Init */
  if(iClusterId != iPrevClusterId) /* A new cluster starts. Send the previous one to Master*/
  {
    /* Now send the CommArray that may have one or many records containing one full cluster */
    MPI_Send(CommArray, comm_index, comm_cluster_rec, MASTERPROC, myproc, MPI_COMM_WORLD);
    MPI_Wait (&Request, &status);
    /* Now reset values so we start working on a new cluster */
    iPrevClusterId = iClusterId; /* Set current & prev IDs equal as new cluster starts here */
    comm_index=0; /* Reset cluster size counter */
  }
```

Figure 4.8 A worker processor sending clusters to the master processor by passing a vector *CommArray* of type *comm_cluster_rec*

A worker processor packs a cluster in a dynamically allocated structure that is a set of records each containing the parent_id (or cluster_id) and the segment_id. While this cluster is being packed to be communicated to the Master processor, it is ensured that a segment_id is not repeated. This is done because multiple instances of a specific segment_id should be sharing the same parent_id due to the logic in the clustering algorithm that ensures that two instances of a segment always share the same parent_id. This extra checking, before the worker-to-master communication takes place, saves considerable amount of processing at the Master node as we pass much fewer records reducing the communication cost as well. Typically, the packet size transmitted in one MPI_SEND is large enough to significantly reduce the communication latencies compared to another approach of sending each array element by itself yet it is small enough to benefit from a parallel send-receive while the sender is busy packing the next packet and the receiver (master) is busy processing the received packet.

```
/* Declare the structure that will be passed among processors to send/receive a
   hashed kmer index, segment ID, etc */
typedef struct {
        long hash_index;
        int segment_id;
        short start_pos;
        short orient_flag;
        short weak_flag;
        } COMM_REC;

/* Declare another structure that will be passed from workers to master processor
   to send/receive a cluster record*/
typedef struct {
        int parent_id;
        int segment_id;
        } COMM_CLUSTER_REC;

COMM_CLUSTER_REC *CommArray;
.....................
.....................
CommArray = (COMM_CLUSTER_REC *)malloc(sizeof(COMM_CLUSTER_REC)*COMM_ARRAY_SIZE);
```

Figure 4.9 Declaration of communication structures COMM_REC and COMM_CLUSTER_REC followed by definition of an array to package a cluster for transmissions.

The time complexity of this phase of the program is based on the Quicksort as well as the hash table traversal to perform union-find. Although the hash table size is of the order of $4^k$, the number of non-null elements can be estimated as $2n(l-k+1)/p$ for $n$ segments distributed among $p$ processors with an average length $l$ of a segment to generate k-mers of size $k$. Assuming $l>>k$, the time complexity runtime of this phase is given by

$$O\left(\frac{nl}{p}\log\frac{nl}{p}\right) \tag{4.7}$$

*(b) Merge Clusters Using Transitive Property*

The master processor works through the local clusters sent by the worker processors and merges them into larger clusters. Two clusters are merged into one when a common member segment is found in both. All clusters are processed in the order of their receipt at master node until no further mergers can take place or all are merged into one. This is implemented as a sequential process and takes $O(n^2)$. The final clustering may sometimes yield groups of clusters of uneven sizes, i.e., one cluster may be disproportionately larger than the

other. When the workload is assigned to parallel processing units, it may result in an unbalanced workload causing some processors to work more than the others. This should be addressed at a later stage using dynamic scheduling in the multiple-alignment algorithm [61]. Time complexity of this final sequential clustering phase is

$$O(n^2) \tag{4.8}$$

Our implementation is not optimized to identify the repeat regions. However, a depth count can be calculated for each k-mer that signifies the number of times a particular k-mer appears in the hash table. Typically, a depth count close to the value of coverage *c* is expected but a larger value indicates a potential repeat region. If required, it can be inferred from the hash table at any given time. The flowchart in Figure 4.10 illustrates the process as described above.

### *4.3.3 Calculation Of Communication Overhead*

The hash function described above has a constant time complexity and the collision resolution process is quick. However, one limitation of this function is that the distribution of work among processors can become data specific. For example, a data set with a heavy concentration of one or two types of bases may lead to a disproportionate assignment to one or more processors due to a more common range of indices of the hash table calculated by the function. It is obvious that the level of communication, i.e. passing messages among processors, increases with the number of processors.

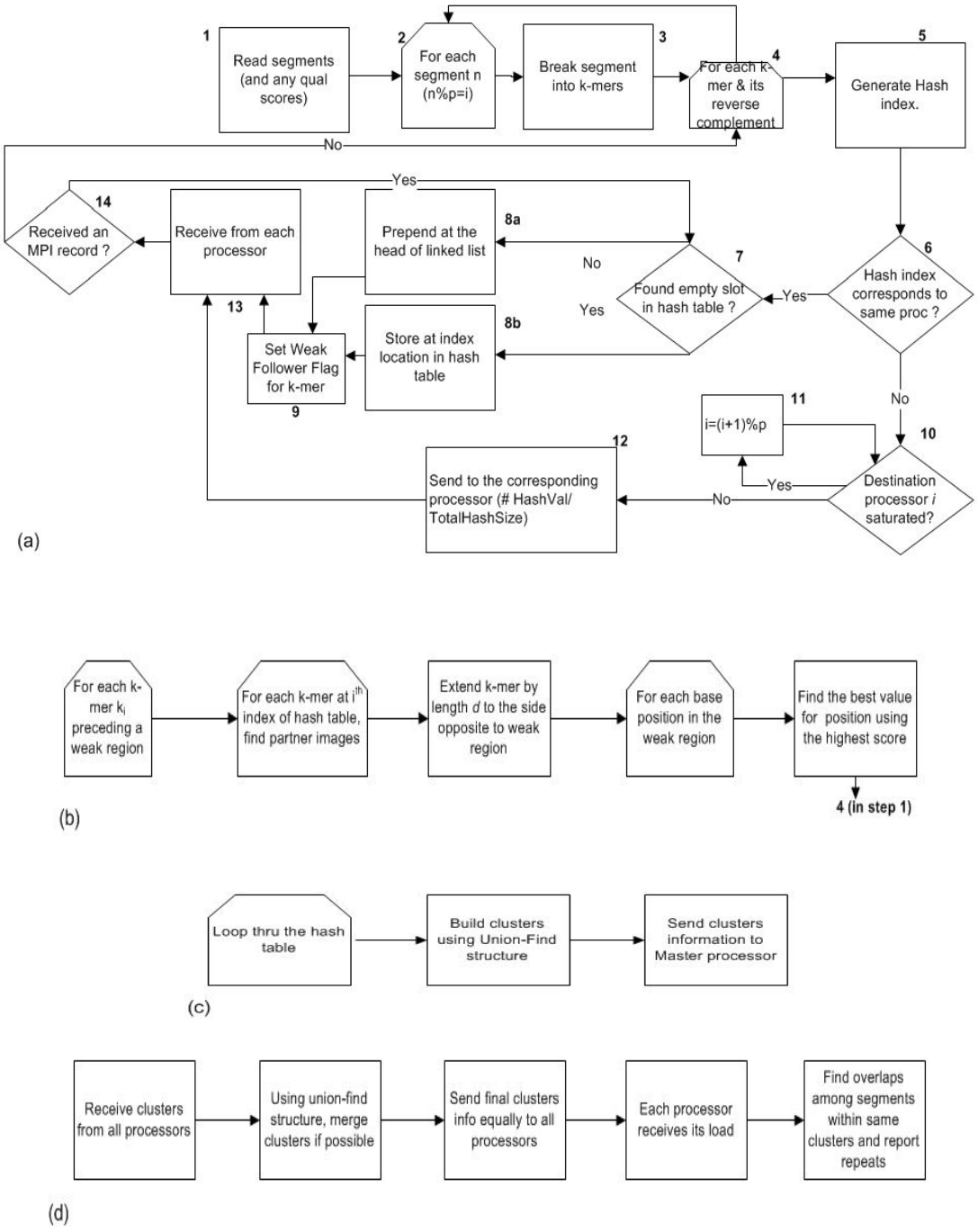


Figure 4.10 Algorithm Flowchart. (a) Data is read and k-merized in parallel. Hash table and linked lists are populated with k-mers and related data, including a flag indicating a following weak region (b) Weak regions are processed and errors are corrected (c) Each processor builds clusters of potentially overlapped segments and sends that information to the master processor (d) Master processor receives all individual clusters and merges them before sending the final disjoint data sets to individual processors.

Like any typical parallel algorithm, the most significant overhead is incurred during communication. The communication overhead is contributed by passing k-mers information to other processors when a hash value falls within a range outside of the local hash table. Let $T_c$ be the overhead related to communication. A vector ${T_c}^j \ \forall\ 1 \leq j \leq x$ represents *x* components of overhead $T_c$ incurred at various modules of the algorithm. The total overhead for all communication phases is given by

$$T_c = \sum_{j=1}^{x} T_c^j \tag{4.9}$$

Following is a discussion of various components that contribute to the total communication overhead.

4.3.3.1 Initial Hashing

As noted earlier, each processor reads and parses $\frac{n}{p}$ segments and k-merize each one into *l*-*k*+1 k-mers. Assuming *l*>>*k* and an evenly distributed data set, a processor will typically store $\frac{l}{p}$ records locally and transmit $(l - \frac{l}{p})$ messages to other processors for each segment. The total number of transmissions from all *p* processors can be estimated to be $n\ (l - \frac{l}{p})$.

The total number of words (k-mer information) to transmit during initial hashing is found to be $\frac{n}{p}\left(l - \frac{l}{p}\right)$ for a given processor, i.e. $n\left(l - \frac{l}{p}\right)$ for all *p* processors. It should be noted that this includes the words transferred after errors have been corrected as we considered the full length of the segment. The communication cost for this phase can be estimated as

$$T_c^1 = (T\mathrm{s} + T\mathrm{t}) * (\mathrm{n}\left(l - \frac{l}{p}\right)) \tag{4.10}$$

4.3.3.2 Sending Clusters To Master Processor

Now we estimate the overhead involved in sending clusters from each worker processor to the master processor as follows.

**Lemma 4.1** There is a minimum overlap of *c* and maximum overlap of *n* segments; therefore, the number of clusters formed by a processor is in the range of 1 to $\frac{n}{c}$ and the size of a typical i$^{th}$ cluster $C_i$ formed by a processor is $|C_i|$, such that $c \leq |C_i| \leq n \; \forall \; i \; \in \boldsymbol{Z}, \; 1 \leq i \leq \frac{n}{c}$.

As we cannot determine the exact number and size of clusters that would be dynamically formed by each processor, we take an average of lower and upper bounds and sum it for all processors to estimate communication cost, which is for the total number of words transmitted, associated with this phase.

$$T_c^2 = p * \; T_s * \frac{1}{2}\left(1 + \frac{n}{c}\right) + p * \; T_t * \frac{1}{2}\left(1 + \frac{n}{c}\right)\frac{1}{2}\;(c + n) \qquad \text{or}$$

$$T_c^2 = p * \frac{1}{2}\left(1 + \frac{n}{c}\right)(T_s + T_t * \frac{1}{2}\;(c + n)) \qquad (4.11)$$

The startup latency holds for only the number of clusters whereas transmit time is counted for the total number of words transmitted, i.e. size * number of total clusters. It can be readily observed from the above equation that the total communication cost can be lowered by reducing the number of clusters (by either decreasing *n* or increasing coverage *c* which would result in fewer clusters thus reducing the $T_s$ component of the communication cost). However, the transmittal portion of the communication cost associated with $T_t$ increases with increase in either coverage or number of input segments.

4.3.3.3 Distributing Clusters To All Worker Processors

After the master processor has merged all the clusters it received from the worker processors, it distributes them to all available processors. This communication overhead can be estimated as follows.

The startup latency occurs only once for each worker processors to which the clusters are being sent by the master processor because master processor uses a block partitioning scheme in allocating the load. The transmittal time, however, depends upon the number and size of the clusters. Once again, the maximum number of clusters can be n/c and a theoretically minimum is 1. Using a similar approach as used in previous section, we can derive the following.

$$T_c^3 = pT_s + T_t * \frac{1}{2}\left(1 + \frac{n}{c}\right)\frac{1}{2}(c+n) \quad (4.12)$$

Using equation (4.10) through (4.12),

$$T_c = (T_s+T_t) * \left(n\left(l - \frac{l}{p}\right)\right) + p * \frac{1}{2}\left(1 + \frac{n}{c}\right)(T_s + T_t * \frac{1}{2}(c+n)) + pT_s + T_t * \frac{1}{2}\left(1 + \frac{n}{c}\right)\frac{1}{2}(c+n) \quad (4.13)$$

We can conclude from this analysis that the communication overhead has a linear dependency on both the number of processors and the data size and that the algorithm is medium grained as it sends a significant number of small messages incurring a startup latency cost that cannot be ignored. However, the main part of the communication is associated with the transmittal of clusters where a minimum startup time is required compared to the overall transmission of messages.

#### 4.3.3.4 Communication Word Length

The size of a word, i.e. each message sent by one processor to another, is denoted by vector $W_s{}^j \; \forall \; 1 \leq j \leq x$ where x (3 for our algorithm) is the number of different communication phases where the word size and the number of words transmitted are unique. The word size is

machine dependent. Table 4.1 shows some commonly used data types in communication structures and their size on the Lonestar cluster used for this research work.

Table 4.1 Data Types And Sizes On Lonestar

| Data type | Size (bits) |
|---|---|
| long | 64 |
| short | 32 |
| int | 16 |
| pointer | 64 |

The word length of the message sent by each processor to other processors while creating k-mers can be calculated based on declaration in Figure 4.9 as 64+32+16+16+16 = 144 bits.

The word length of the message sent by each processor to the master processor while sending clusters can be calculated based on declaration in Figure 4.9 Declaration of communication structures COMM_REC and COMM_CLUSTER_REC followed by definition of an array to package a cluster for transmissions.Figure 4.9 by first calculating the records size which is (32+32) =64 bits. These records are packed into an array to minimize the $T_s$. The array size is a user defined parameter, i.e. COMM_ARRAY_SIZE. The array is dynamically resized if needed. A typical value of array size is 100; therefore, a minimum of 6400 bits are sent for each cluster.

Finally, the word size used by the master processor, to send merged clusters to the worker processors, is also same as that used by worker processors before, i.e. 6400 bits for each cluster. Therefore,

$$ {}_1^3W_s = \begin{bmatrix} 144 \\ 64 * |C| \\ 64 * |C| \end{bmatrix} \tag{4.14} $$

The startup latency $T_s$ and the transmission time $T_t$ on Lonestar architecture have been found to be 2.5μs and 40Gbits/sec respectively, i.e.

$T_s$ = 0.0000025 seconds

and

$T_t = \begin{bmatrix} 0.0000000036 \\ 0.00000016 \\ 0.00000016 \end{bmatrix}$ in seconds.

## 4.4 Analyzing Scalability

We will use the time complexities we noted above along with other criteria for evaluating the scalability of our algorithm in the following sections. This theoretical analysis will be used to study and validate the experimental results in the subsequent sections.

### *4.4.1 Isoefficiency Analysis*

An algorithm is usually characterized by its time complexity that represents a relationship between runtime and the input data. In case of a parallel algorithm, the speedup, a runtime comparison of the execution speed of the parallel algorithm to its sequential version, is normally used as a metric for measuring performance. The efficiency, which measures the fraction of time a processor spends in useful work, is the speedup per processor and provides a gauge of utilization in a parallel algorithm. Assuming a constant data size *W*, the efficiency decreases with the increasing number of processors. This is due to the fact that more processors are available to do the same work that previously was achieved by fewer processors causing lower utilization. It also results in more communication overhead and start-up times, denoted together by $T_0$ and $T_s$ respectively, as discussed before. A parallel algorithm is considered scalable if increasing resources, which are usually the processors, requires only a proportional increase in the input data to keep the efficiency from decreasing significantly. The sequential time complexities for different phases described above can be summed up as:

$$T_1 = nl + k + nl + n^2 l + nl * \log(nl) + n^2 \quad (4.15)$$

Using equations 4.3 through 4.14, an overall time complexity of the parallel algorithm, including both computation and communication time complexities, is

$$T_{\text{p}} = \frac{nl}{p} + k + \frac{nl}{p} + (T_s + T_t) * \left(n\left(l - \frac{l}{p}\right)\right) + p * \frac{1}{2}\left(1 + \frac{n}{c}\right)(T_s + T_t * \frac{1}{2}(c + n))$$

$$+ pT_s + T_t * \frac{1}{2}\left(1 + \frac{n}{c}\right) * \frac{1}{2}(c + n) + \frac{n^2 l}{p} + \frac{nl}{p}\log\frac{nl}{p} + n^2 \qquad (4.16)$$

Using equations (4.1) and (4.2), assuming p >> 1, *l* >> 1, and n >> c, we simplify the above as

$$W = K(p^2\left(n\frac{T_s}{2} + \frac{n^2}{4}T_t + T_s\right) + pn\frac{T_t}{4} + nl(p(T_s + T_t) + (logn - log(nl)) \qquad (4.17)$$

It should be noted that all factors of the equation (4.17) contribute to the isoefficiency computation. However, we are more interested in the component that causes the data size to grow at the fastest rate with respect to *p* [33] and, asymptotically, our algorithm has a quadratic polynomial isoefficiency in terms of the number of processors. Hence, we can deduce that an increase in the number of processors from *p* to $p_{\text{new}}$ would require input data to be increased by a factor of $p_{\text{new}}^2/p^2$. Such an Isoefficiency function indicates that the parallel algorithm is reasonably scalable considering a small to mid-range parallel computing environment.

*4.4.2 Multi-objective Optimization Functions*

In this section, we will define the objectives and their maximization functions. However, we first need to introduce some concepts and definitions related to optimization functions and decision variables. Although there may be many different objectives depending upon the need and available resources, we have identified two that are usually considered to be the top priority in this domain. These often inter-conflicting objectives are **speedup** and **accuracy** with the data

size as the most important factor both depend upon. A typical maximization problem for *q* objectives can be stated as follows.

Given a set of variables that makes up all of the objective functions, a vector $x = \{x_1,\ldots,x_n\}$ in the solution space *X*, the goal is to find a vector $x^*$ which maximizes a given set of *q* objective functions $z(x^*) = \{z_1(x^*),\ldots,z_q(x^*)\}$. As each variable usually has lower and upper bounds, the solution space *X* is generally restricted by a series of constraints that lead to a feasible and calculable solution space.

4.4.2.1 Pareto Optimality

Assuming all objective functions are for maximization, a feasible solution *f* is said to dominate another feasible solution *g,* if and only if $z_i(f) \geq z_i(g)$ and $z_j(f) > z_j(g)$ for at least one objective $z_j \; \forall \; i, j \in \{1,\ldots,q\}$. A solution is said to be *Pareto optimal,* or *Pareto efficient*, if it is not dominated by any other solution in the solution space, i.e. a Pareto optimal solution cannot be improved with respect to any objective without worsening at least one other objective. A set of all such non-dominated solutions in *X* is referred to as the Pareto optimal set and the corresponding objective function values in the objective space make up the *Pareto front*. The Pareto set can be enormously large and very difficult, if not impossible, to calculate for many problems; therefore, a Pareto optimal set with reasonable coverage of the ranges of decision variables can be considered sufficiently representative of the entire set of feasible non-dominated solutions [55].

4.4.2.2 Decision variables

Some of the decision variables relevant to the following analysis are listed below.

Let

*n* = number of input segments

*l* = average length of a segment

$d$ = length of each partner image used in error correction logic

$p$ = number of processors

$m$ = memory size of a processor

$k$ = size of each token (k-mer) $1<k<\log_4(m)$

$c$ = coverage, i.e. no. of times a particular base in original DNA is sequenced

$s$ = size of the genome

$\mu$ = complexity of the algorithm

For the sake of simplicity, we will not discuss the space required for communication as it can be assumed as factored into the communication cost.

4.4.2.3 Decision Constraints

As discussed above, constraints define the bounds on the values of decision variables to yield feasible solutions. Based on the experiments we conducted, Table 4.2 lists the decision variables and their constraints used in this analysis.

Table 4.2 Decision Variable Constraints

| Variable | Minimum | Maximum |
|---|---|---|
| n | 1000 | 16000 |
| l | 600 | 800 |
| k | 10 | 12 |
| m | 2 | 4 |
| d | 6 | 12 |
| p | 1 | 128 |

*4.4.3 Increasing Data Size*

The size of the data can be increased by increasing coverage or adding reverse compliments if those are not already in the input data set. Increasing data size adversely affects processing speed but provides extra coverage, which in turn yields more partner images for the error correction logic to be more effective and results in enhanced accuracy. As long as there is

enough memory to support the data structures used in the program, the size of the data can be arbitrarily increased. We will present these relationships in the next sections.

Although increasing data size is desirable and often an objective, it is usually more of a decision variable as it affects other objectives directly whereas it is not affected by other objectives reciprocally. Therefore, our optimization model is based upon only accuracy and speedup as the two key objectives and assumes that all other parameters, including data size, make up the decision space.

*4.4.4 Enhancing Accuracy*

It follows that error correction logic works better with larger memory, smaller k-mers, larger extensions for partner images, and increased number of processors to allow larger data set, which in turn improves the accuracy. We can formulate these relationships for accuracy, *A*, for the analytical modeling purposes as

$$A \propto ndp$$

Also,

$$A \propto 1/k$$

therefore,

$$A = \mu ndp/k$$

where $\mu$ represents the algorithm complexity to be remained constant

It should be noted that a mere increase in the number of data segments *n* cannot guarantee a linear increase in the accuracy. The success of error correction logic depends upon the availability of partner images of weak regions, i.e. level of coverage and the percentage of that coverage available to the program.

Assuming a uniformly distributed multiple instances of a particular base, an increase in data size *n* by *x* segments can be expected to aid the error correction logic by a percentage of

(*n*+*x*)*100/(*c**s*/l). For example, adding 1000 segments to the input data of 1000 segments from a genome size of 161,000,000 bases with a coverage of 7x and average length of 700 bases/segment, an increase of (2000)*100/7*161000000/700 or ~1% increased accuracy is theoretically possible. Hence, it is not just *n*, but coverage *c*, genome size *s*, and average segment length *l*, all play an important role in modeling the increase in accuracy when increasing data size. The maximization function for enhancing accuracy can therefore be given as

$$Z_1 = \max\left(\frac{ndp}{k}\right) \tag{4.18}$$

*4.4.5 Increasing Speedup*

We will now present a theoretical model of speedup which can be used to forecast the actual speedup attained by the implementation of our algorithm. From equations (4.15) and (4.16), knowing that speedup is the ratio of sequential runtime to the parallel runtime, i.e. $= \frac{T_1}{T_p}$, we can derive the following.

$$S = \frac{(nl+k+nl+n^2l+ nl*\log(nl)+ n^2)}{{}_2\frac{nl}{p}+k+\frac{n^2l}{p}+\frac{nl}{p}\log\left(\frac{nl}{p}\right)+ n^2+T_c}$$

Assuming $1 < c < 6,\ l \gg 1, n \gg 1, and\ l \cong p$, we can simplify the ratio as

$$S = \frac{n}{\frac{2}{p}+\frac{1}{2l}+\frac{n}{p}+\frac{1}{p}\log\left(\frac{nl}{p}\right)+T_c} \tag{4.19}$$

Although $T_s$ and $T_t$ are small, the communication overhead grows with both the number of processors and data size; therefore, equation (4.19) includes the communication overhead $T_c$. It can be inferred from equation that the speedup should increase with the number of input sequences and with increasing number of processors. In the next section, we will plot our

theoretical speedup vs. actual speedup obtained from the empirical results. The maximization function for speedup can therefore be given as

$$Z_2 = \left(\frac{n}{\frac{2}{p}+\frac{1}{2l}+\frac{n}{p}+\frac{1}{p}log\frac{nl}{p}+T_c}\right) \quad (4.20)$$

*4.4.5 Discussion of Pareto Frontier*

It should be noted that there is a wide range of data points that could be used in the decision space. However, based upon the real life constraints, as shown in Table 4.2, the objectives are modeled using the data size and processors available for this study. It follows that a larger data set is critical for increasing accuracy as well as improving speedup though it eventually affects the speedup. Also, smaller values of *k*, i.e. k-mer size, improve accuracy but affect the speedup due to increased communication as after reaching a certain threshold point, the communication to computation ratio begins to have an adverse impact. We will present more results and a detailed analysis in the following sections.

## 4.5 Experiment and Results

To test and validate the time complexity analysis discussed above, we implemented our algorithm using C programming language in MPICH environment [36], an implementation of Message Passing Interface (MPI) framework, using up to 128 nodes in a high performance computing environment as described below. Although finding repeats was not the primary focus of this research, areas with concentration of repeats can be inferred using the depth count as explained earlier. The resulting clusters consisting of overlapping segments were randomly selected for various sizes and validated using SeaView graphical tool [31].

*4.5.1 Experimental Platform*

Following sections provide in-depth details of the system and environment used to develop and test the parallel implementation of our algorithm explained in previous sections.

4.5.1.1 System Overview

The Lonestar Linux Cluster, part of Texas Advanced Computing Center (TACC), consists of 1,888 compute nodes, with two 6-Core processors per node, for a total of 22,656 cores. It is configured with 44 TB of total memory and 276TB of local disk space. The theoretical peak compute performance is 302 TFLOPS. The system supports a 1PB global, parallel file storage, managed by the Lustre file system. Nodes are interconnected with InfiniBand technology in a fat-tree topology with a 40Gbit/sec point-to-point bandwidth. A 10 PB capacity archival system is available for long term storage and backups.

4.5.1.2 Hardware Details

A regular node consists of a Dell PowerEdge M610 blade running the 2.6 x86_64 Linux kernel from kernel.org. Each node contains two Xeon Intel Hexa-Core 64-bit Westmere processors (12 cores in all) on a single board, as an SMP unit. The core frequency is 3.33GHz and supports 4 floating-point operations per clock period with a peak performance of 13.3 GFLOPS/core or 160 GFLOPS/node. Each node contains 24GB of memory (2GB/core). The memory subsystem has 3 channels from each processor's memory controller to 3 DDR3 ECC DIMMS, running at 1333 MHz. The processor interconnect, QPI, runs at 6.4 GT/s.

4.5.1.3 Interconnectivity

The InfiniBand (IB) interconnect topology is a Clos fat tree (no oversubscription). Each of the 16 blades in a chassis is connected to a Mellanox M3601Q InfiniBand Switch Blade (leaf switch) within the chassis. From 16 ports (40Mb/s per port) of each switch blade a bundle of 4

links is connected to each of the 4 large core switches. Each core switch is a Mellanox IS5600 648-port (40Gb/s each) switch delivering up to 51.8Tb/s of bandwidth. Even with IO servers connected to the switches there is still about 25% capacity remaining for system growth. Any processor is only 3 hops away from any other processor.

#### 4.5.1.4 Software Environment

The cluster runs Linux OS where nodes are managed by the Rocks 4.1 cluster toolkit with 64-bit version of MPI MVAPICH available for use. MPICC compiler is used to build the executable and the program is run in batch mode using Single-Program-Multiple-Data (SPMD) paradigm in this distributed memory configuration.

#### 4.5.1.5 Data

The raw traces of Drosophila Yakuba were obtained from ENSEMBL [44] along with the corresponding quality scores. The program was executed as a batch routine and total runtime was used to compute efficiency at different data points. Each data point represents number of processors from a range of 2 to 128 and the size of the dataset from 1000 to 16000 sequences.

### *4.5.2 Results*

The runtime was calculated using values returned by the MPI function *MPI_WTime* at the start and end of different segments within the program code. We have plotted percentages of different sections of the program, e.g. percentage of error correction logic processing compared to the rest of the program to predict the cost associated with error correction, i.e. runtime attributed to error correction logic. Also, increasing the data size and studying variation of error correction portion of runtime provides an insight of the relationship between accuracy and data size.

4.5.2.1 Performance vs. Data Size

We tested our program using a set of up to 16000 sequences as mentioned in the previous section. One of the objectives of this research is to allow larger data sets to be run through within reasonable duration of time. Using well designed data structures and functions like hashing and sorting, this is achieved within the available resources and is illustrated by the graphs below.

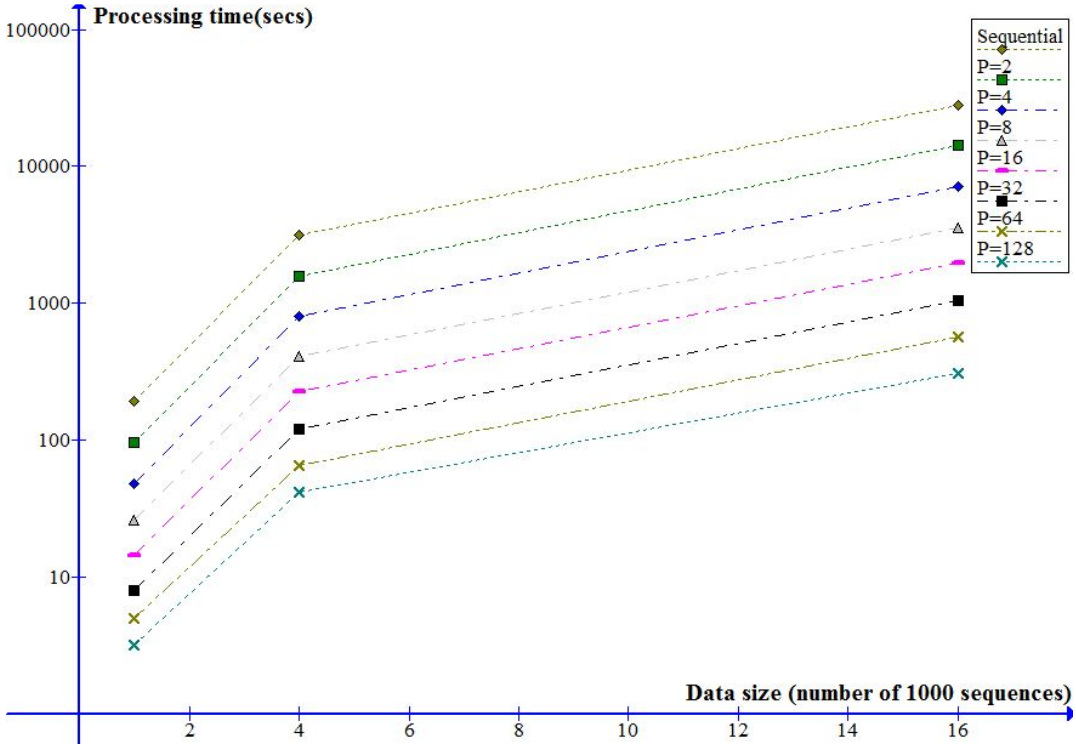


Figure 4.11 Exectuion time vs. data size for different number of processors.

Figure 4.11 illustrates the benefits of many optimization techniques and data structures used in the algorithm increase with the data size resulting in less steeper curves for larger data sets. Also, as the number of processors grows, the savings in the processing time, represented by the distance between two consecutive curves, improve before diminishing due to an increasing communication-to-computation ratio.

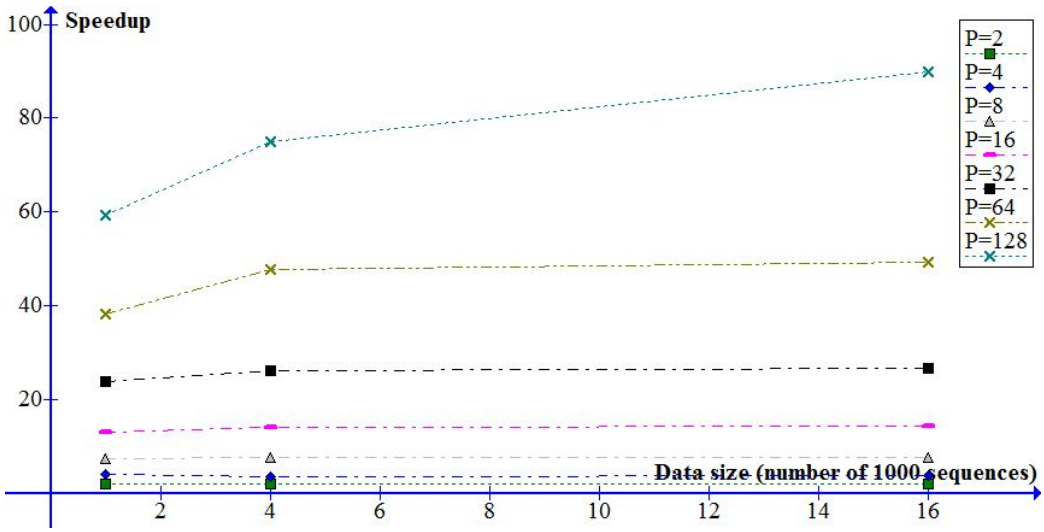


Figure 4.12 Speedup vs. data size for different number of processors.

As illustrated in Figure 4.12, the speedup increased with both increasing data size and number of processors. The increase in speedup due to the increasing data size shows how larger volumes of data better leverage the distributed processing compared to the smaller data sets. The percentage increase in speedup due to increasing number of processors also tapers off for the same size dataset due to increased communication.

For a given number of processors, as shown in Figure 4.13, the efficiency tends to increase with the increasing data size. This increase is relatively sharper in the beginning but starts to diminish as the level of communication rises. Also, for a given data size, the efficiency drops with the increase in the number of processors. Both observations confirm our theoretical analysis.

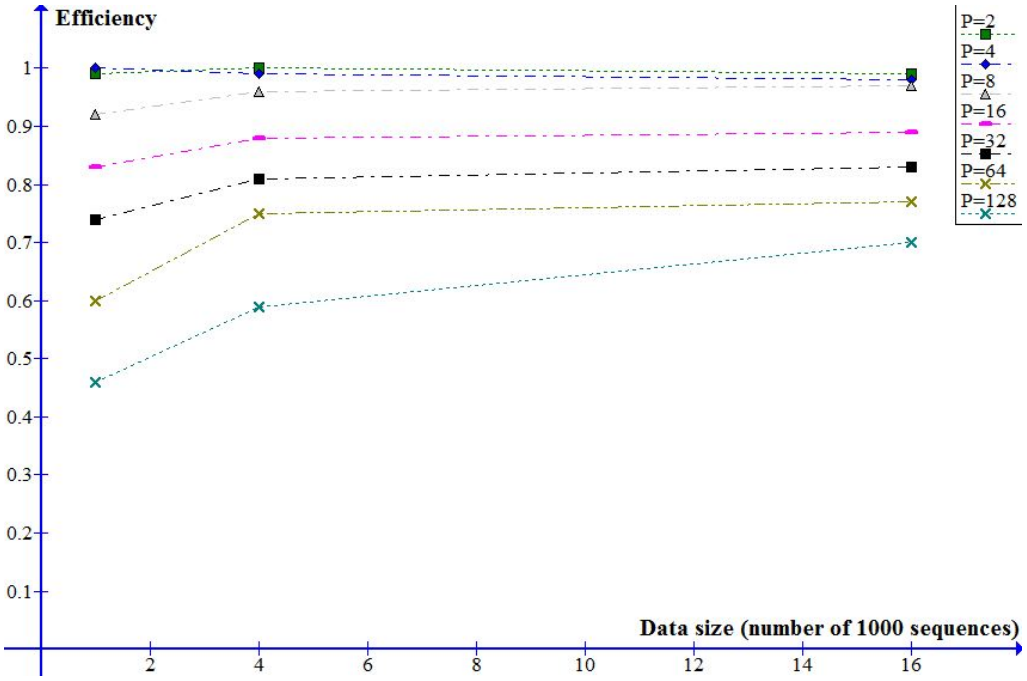


Figure 4.13 Efficiency vs. data size for different number of processors.

4.5.2.2 Enhancing Accuracy

The algorithm attempts to correct errors that are mostly found towards the beginning and the end of the sequences details of which are explained in previous sections. Here, we will take a look at how we validated the logic using real data set with some artificially created errors.

The input data consisted of three sets of sequences where only end regions were used to test and validate the error correction logic of the algorithm. In order to simplify the testing, we identified and tracked 60 erroneous bases at various locations and manually checked the results after the program was run. Following is the summary of the test results which is also plotted below.

With reference to our prior discussion of how weak regions are identified, Table 4.3 shows the actual values of the program parameters we used to test the error correction logic.

Table 4.3 Parameters Used To Test And Validate The Error Correction Logic of The Algorithm

| No. | Parameter | Value |
|---|---|---|
| i | K-mer size, i.e. k | {10, 12} |
| ii | Number of errors identified and tracked in the dataset | 60 |
| iii | Minimum threshold value to identify a weak base, i.e. $T_q$ | 20 |
| iv | Minimum length of the weak region, i.e. $T_y$ | 15 |
| v | Minimum number of bases with quality score <= $T_q$ within an expecte weak region, i.e. $T_w$ | 7 |
| vi | Length of the extension delta *d* used to extend the partner images | {6, 12} |

The test run used input of varying data sizes, ranging from 1000 to 16000 reads, each with an average length of 700 bases. In order to evaluate the error correction logic, we describe the four types of outcomes for each error as follow.

- **True Positive**: An error was found and corrected using the right partner image(s) leading to a correction.
- **False Positive**: An error was found but corrected with the wrong partner image(s) resulting in a false correction.
- **True Negative**: An error may have been found but no correction took place because there was no matching partner image found.
- **False Negative**: An error was not detected or it was not part of the input data.

The sensitivity and specificity values have been calculated using the formulae as below:

$$Sensitivity = \frac{No.\ of\ True\ Positives}{No.\ of\ True\ Positives + No.\ of\ False\ Negatives}$$

and

$$Specificity = \frac{No.\ of\ True\ Negatives}{No.\ of\ True\ Negatives + No.\ of\ False\ Positives}$$

As the data size grew, the number of potential partner images increased and therefore we see that the error correction rate improved. Table 4.4 summarizes the test results using the parameters defined in Table 4.3. It can be observed that a few of the errors corrected by the program turned out to be false when compared with the real data. This was due to the fact that the input data did not contain the entire set of segments and therefore some of the partner images were not available for consideration. However, this does not prevent us from testing and validating the error correction logic and extrapolating the results for our research.

Table 4.4 Results of Error Correction Scheme

| **Results** | **Dataset 1: 1000 sequences** | **Dataset 2: 4000 sequences** | **Dataset 3: 16000 sequences** |
|---|---|---|---|
| Program identified errors for correction based on criteria for identifying the weak regions and using the quality scores | 60 | 60 | 60 |
| | | | |
| *d*=6 | | | |
| Errors corrected | 17 | 33 | 60 |
| True positives | 11 | 22 | 45 |
| False positive | 2 | 5 | 8 |
| True negative | 4 | 6 | 7 |
| False negative | 43 | 27 | 0 |
| Sensitivity | .20 | .45 | 1.0 |
| Specificity | .67 | .55 | .47 |
| | | | |
| *d*=12 | | | |
| Error corrected | 17 | 33 | 60 |
| True positives | 12 | 27 | 55 |
| False positive | 2 | 3 | 3 |
| True negative | 3 | 3 | 2 |
| False negative | 43 | 27 | 0 |
| Sensitivity | .22 | .50 | 1.0 |
| Specificity | .60 | .50 | .40 |

The error correction logic requires extra processing of data, adding to the overall processing time. However, a parallel program allows us to develop complex algorithm to correct errors while keeping the overall processing times under a reasonable limit. In some instances, as evident from the results, making the error correction logic more restrictive yields better

accuracy (true positivity) though it requires more processing time as well as space due to increased size of the extension delta used to validate the partner images.

A set of charts is plotted and produced here for reference based upon various relationships.

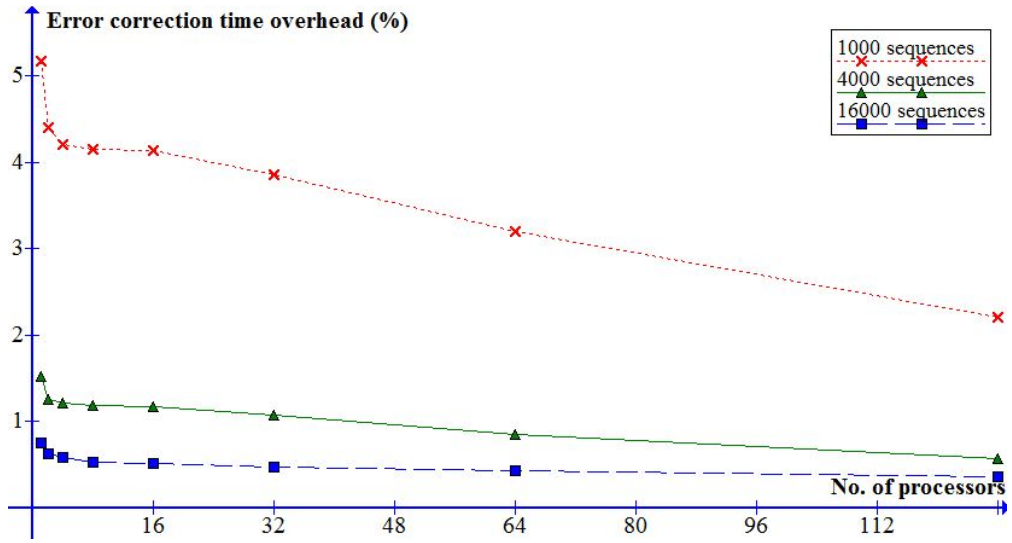


Figure 4.14 Error correction overhead (%) with increasing number of processors.

It can be inferred from curves in Figure 4.14 that the error correction overhead tends to decrease fast in the beginning before stabilizing as the number of processors become significantly large. This is due to the following reason. There are two main parts of the error correction logic. First, it scans the input data and the local hash table, finds partner images, and corrects errors. All of this processing takes place locally without any inter-process communication. However, it then breaks the corrected bases into k-mers, which in turn may be sent over to another processor thereby requiring message passing among processors. As the number of processors grows, the communication-to-computation ratio grows and the overall reduction in overhead slows down as shown by the flattening curves. The other characteristic of these curves is that the fraction of time spent in error correction decreases as the data size

grows. This is because the error correction logic is more data parallel than the other parts of the algorithm and therefore its runtime does not grow as fast as that of the other modules.

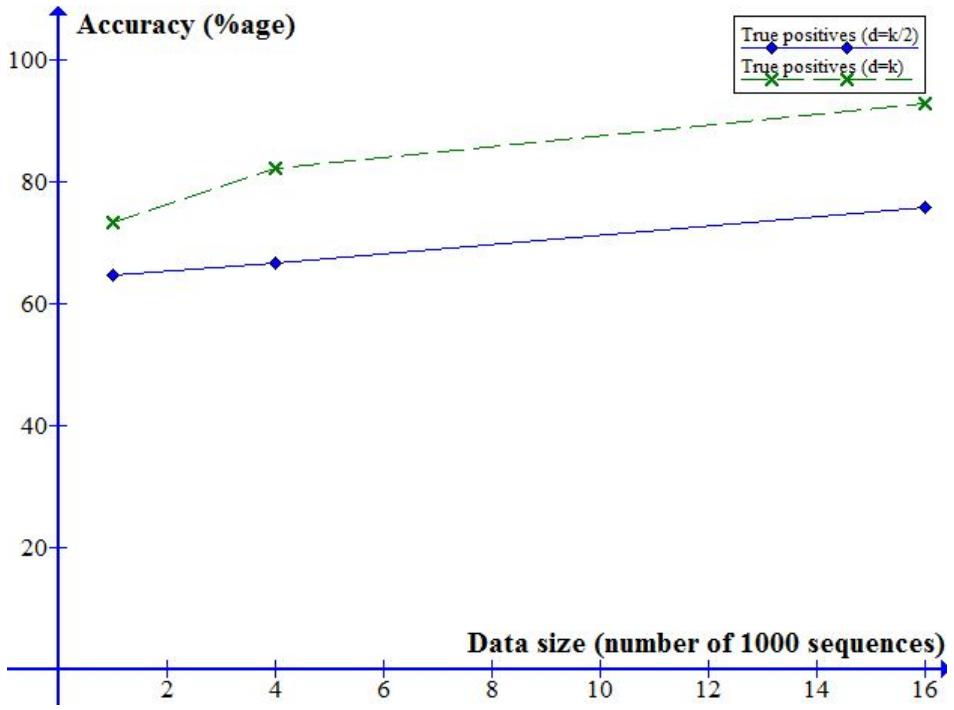


Figure 4.15 Accuracy vs. data size for d=k and d=k/2

Figure 4.15 shows the relationship between data size and the rate of error correction for two different values of extension delta *d*. For a given *d*, the accuracy tends to improve for larger data volumes as the number of correctly partnered images grows. It should be noted here that we simulated this test case and the results using real raw data might vary as the number of matches found with every increment of data may be different depending upon the selection of sequences.

It can also be observed that increasing the value of extension delta *d* adds to the overall accuracy rate by avoiding some false positive results. This is true because increasing *d* makes the algorithm more restrictive.

*Accuracy in terms of sensitivity and specificity of the algorithm*

Figure 4.16 illustrates how increasing data size and tuning the extension delta value affects the sensitivity and specificity of the program. Here, sensitivity provides the measure of how accurately the program processes and corrects the true error whereas the specificity, in simple terms, helps measure the false alarm, i.e. how many corrections were made for the erroneous bases by matching with false partner images. As the data size and the extension delta *d* increase, the sensitivity rises as the program can process more errors that are truly errors. However, the specificity decreases as the algorithm tends to process more false positives.

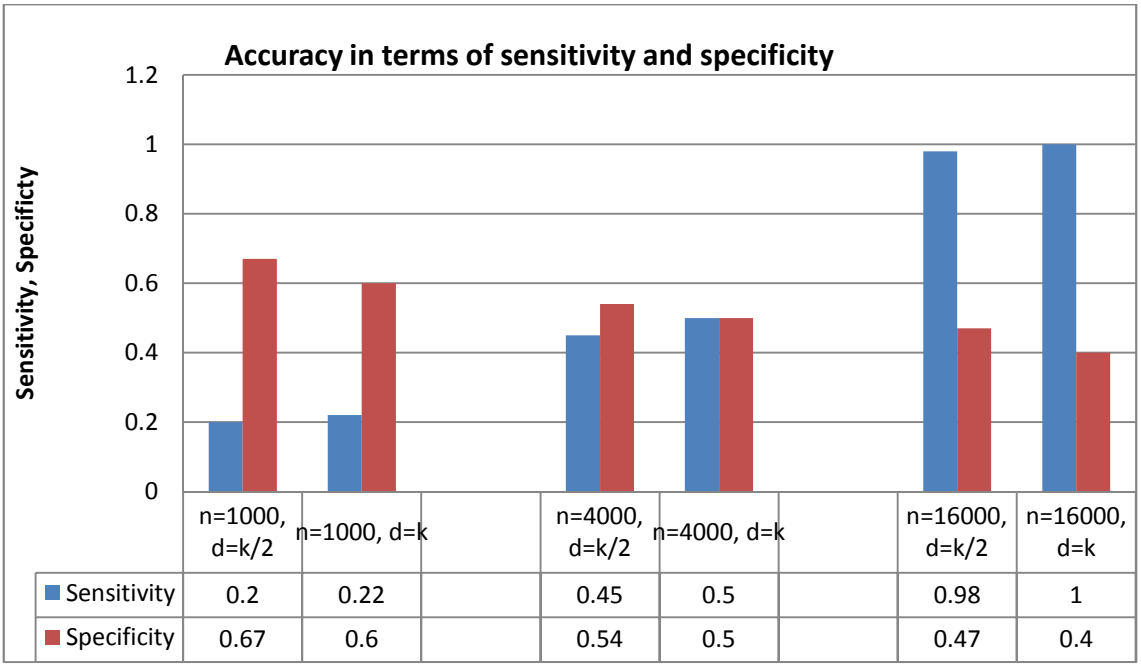


Figure 4.16 Accuracy in terms of sensitivity and specificity.

4.5.2.3 Optimizing Speedup

Following is a discussion of the results to measure the performance of the parallel processing leading to the validation of the scalability analysis we performed earlier.

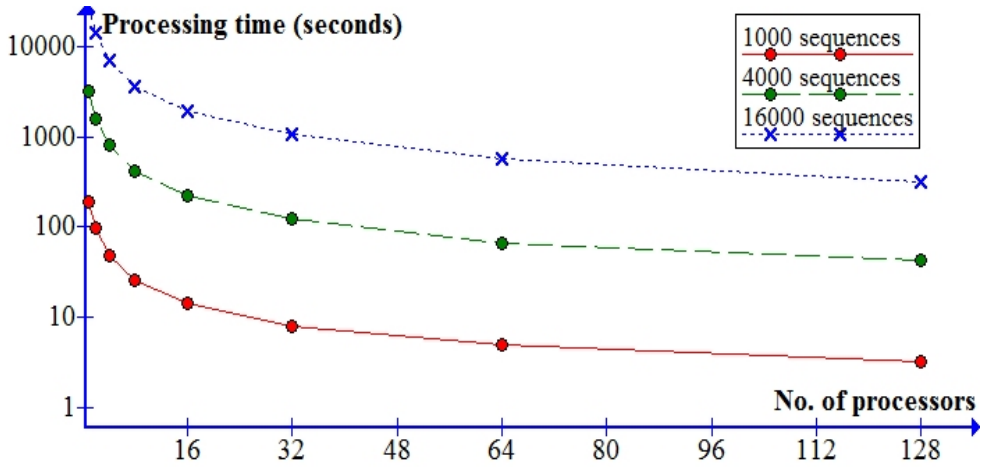


Figure 4.17 Processing time vs. number of processors for different data sizes.

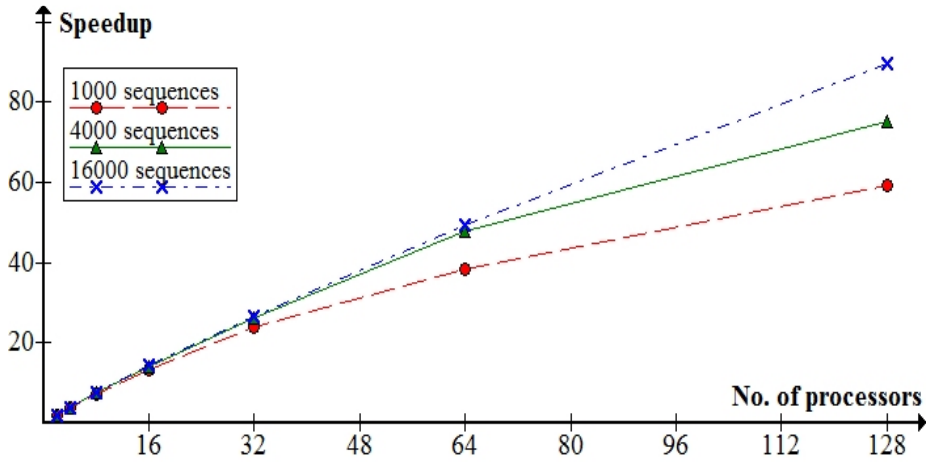


Figure 4.18 Speedup for different sizes of data.

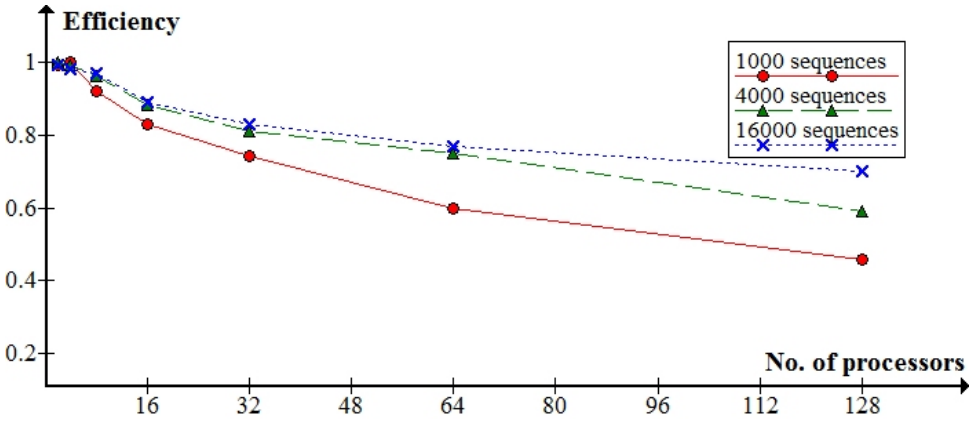


Figure 4.19 Efficiencies for different sizes of data

As shown in Figure 4.19, a steady decline in efficiency was observed with increasing number of processors. It should also be noted that for larger data set the curve was less steep and showed relatively higher efficiencies.

***Interrelationship among different performance measures***

Our system allows fine tuning of performance using different parameters that can be set before running the program. As discussed earlier, these parameters include *k*, the size of k-mer tokens, the extension delta size *d* and various threshold values used in the error correction logic. Each has an impact on system performance in one way or another. Figure 4.20 depicts the interconnectivity among different performance measures, i.e. how increasing the data size and desired accuracy affect the speedup. It can be observed that increasing the size of extension delta *d* (causing a more restrictive matching of partner images) and decreasing the size of k-mers, i.e. *k* (making more overlaps possible), improve accuracy; however, the overall speedup is adversely affected due to changing these parameters. We learn from the Pareto frontier curve that both of our objectives, i.e. accuracy and speedup, can be initially improved at the same time to a certain level. After initial improvement, the conflict becomes more obvious and maximizing one objective can only be achieved at the expense of the other. Finally, we observe that the Pareto frontier comprise a only a few data points indicating a limited range of

optimality, i.e. there is not much leeway for optimizing both accuracy and speedup without an adverse impact on one or another.

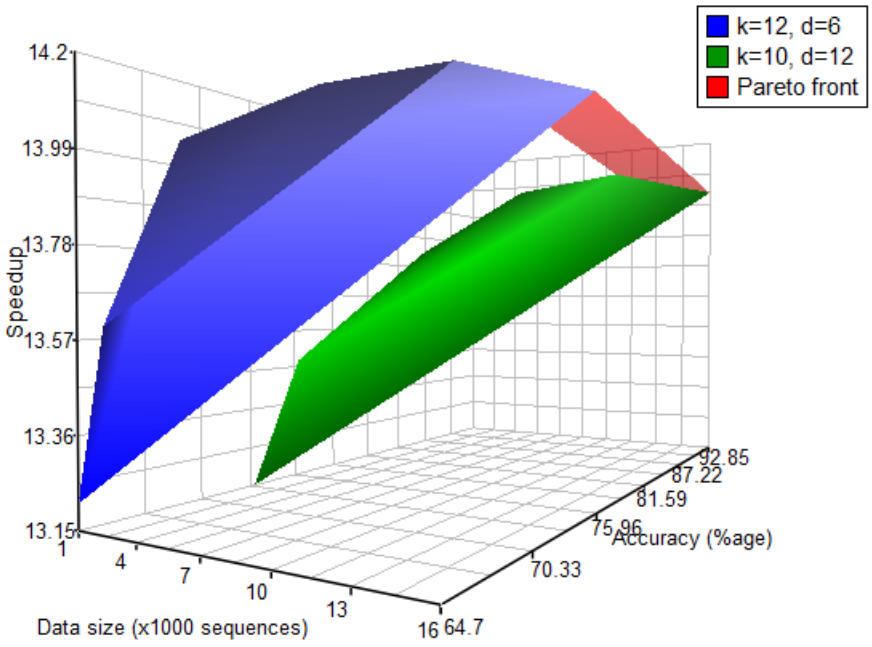


Figure 4.20 A 3-dimensional illustration of multiple performance measures with lower curve: (d,k) = (6,12) and upper curve:(d,k) = (12,10). The small area joining the two curves comprise Pareto frontier. No. of processors=16.

*4.5.3 Empirical Results vs. Theoretical Analysis*

In this section, we compare the predicted results with the experimental results for three test cases as follow.

4.5.3.1 Test Case 1: *Isoefficiency Curves*

Earlier, we derived isoefficiency function for our algorithm as given by equation (4.17) to be a quadratic polynomial function in terms of the number of processors, i.e., every time we

double the number of processors, a four-fold increase in the size of the input data will be required to keep efficiency near constant.

Figure 4.21 shows Isoefficiency curves plotted for a range of 8 to 128 processors and the data size ranging from 1000 to 16000 sequences of an average length of 600 to 800 as given in Table 4.2. These isoefficiency curves confirm the theoretical analysis of our algorithm as the two-fold increase in the number of processors require a four-fold increase in the data size to keep the efficiency at a near constant. A minor deviation observed is due to the fact that we ignored a few low order time complexities during our analysis. As the number of processors grows, the gap between the theoretical and actual curve starts to narrow. This may also suggest that, at some point, where the number of processors is very large, we would see a less than desired scalability. However, with the execution speed and speedups we have achieved for this particular problem, there is no need to use such a large number of processors. Also, this comparative analysis confirms that Isoefficiency is a valid and useful metric for a parallel system and sets precedence for other related problems in bioinformatics where a similar analysis should help classify the algorithms based upon required scalability.

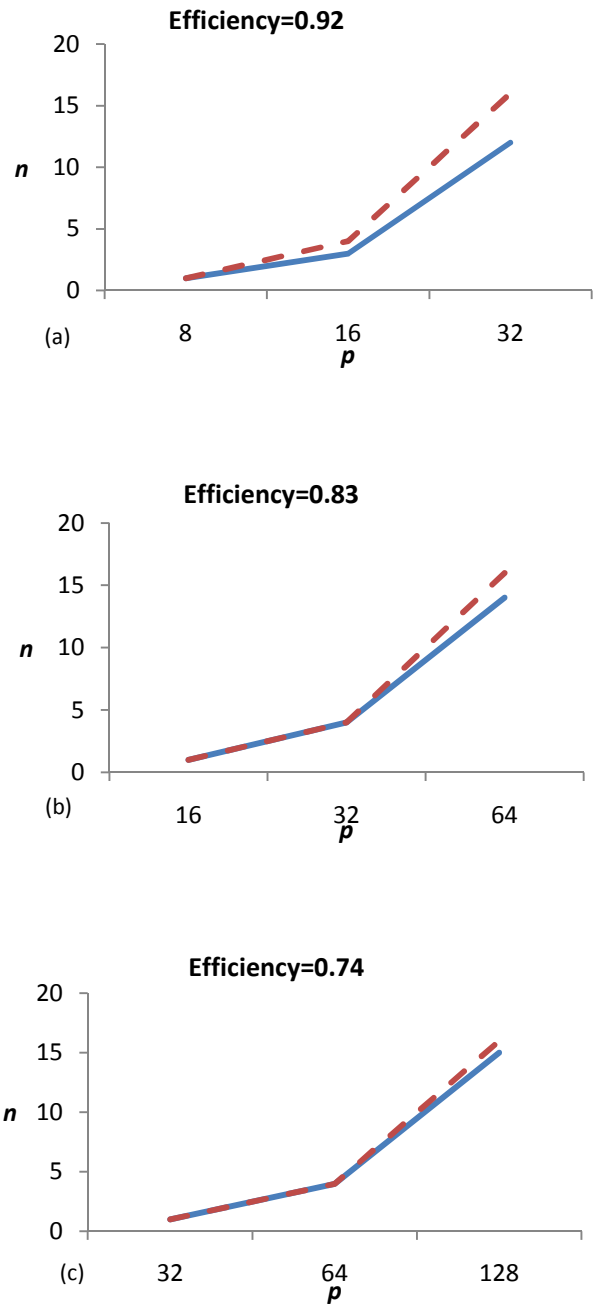


Figure 4.21 Isoefficiency curves. *p*=no. of processors, *n*=no. of sequences in data (x1000) Dashed line for theoretical and solid line for real curves, each illustrating a near constant efficiency of (a) e=0.92 (b) e=0.83, (c) e=0.74.

4.5.3.2 Test Case 2: *Actual vs. theoretical Speedup Curve*

Figure 4.22 illustrates how actual speedup curve deviates from the predicted gains calculated using the derived analytical model of the system. The speed gain is ideal for lower number of processors for a given data size. However, the gains start to decrease as the number of processors increase, affecting the overall runtime of the program. To explain this behavior, we first consider the fractions of the algorithm that are serial in nature, i.e. performed by the master processor. The final clusters are merged together by the master processor only and this part of the algorithm has relatively higher time complexity, i.e. quadratic polynomial in terms of the number of sequences, compared to other parts of the algorithm. This tends to have a significant impact on the execution time as the size of the data grows. Also, the communication-to-computation ratio (CCR) increases with increase in data size as more messages are now sent and received by the processors, which adds to the total processing time of a parallel program. The communication cost of our algorithm as given by equation (4.13) shows that CCR grows with both data size and the number of processors.

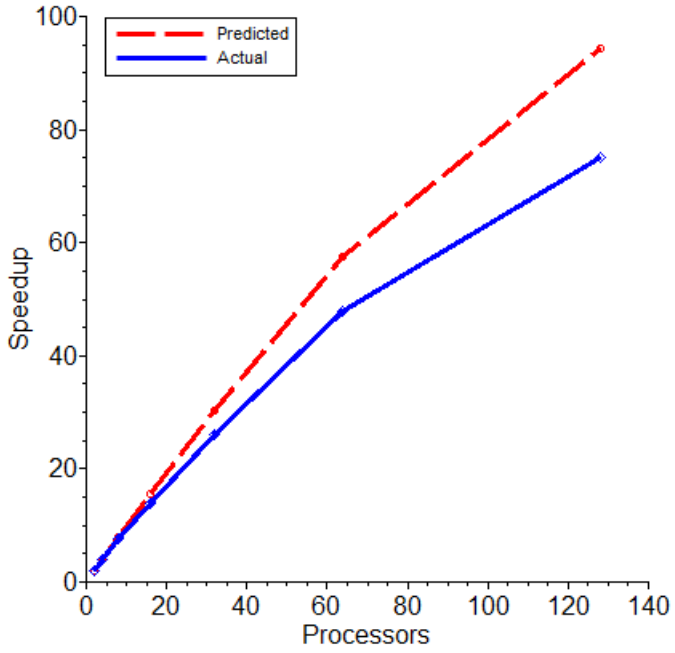


Figure 4.22 Predicted vs. actual speedup for $n$ = 4000 sequences

4.4.3.2 Test Case 3: *Actual vs. theoretical Accuracy*

This test case shows a comparison of the predicted accuracy and the actual error correction attained by the program. Figure 4.23 illustrates how the actual data deviates from the predicted values calculated by the analytical model.

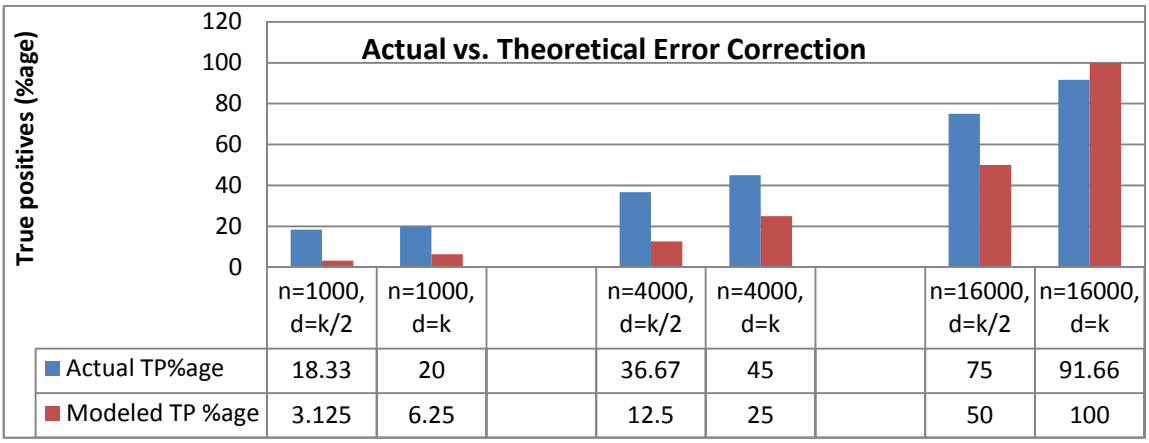


Figure 4.23 Predicted vs. actual accuracy enhancement for increasing data size and extension delta.

Although the pattern is the same, the actual accuracy achieved is much better than the predicted level. This is caused by more than the average concentration of copies of the bases to be corrected. However, as more data is added for processing, this gap narrows. Also, we see that the model predicts a much sharper increase in accuracy when the extension delta is doubled whereas the actual numbers show that the gain is achieved but at a lower rate. In reality, the model cannot portray all possible scenarios with every related decision variable factored in. The curves show, however, that the pattern predicted by the model is fairly close to the actual levels achieved in the experiment.

## 4.6 Conclusion

The key objective of this dissertation is to perform a detailed scalability analysis of a challenging problem in bioinformatics and to present it as a model that can be leveraged in other similar areas, especially in bioinformatics, e.g. multiple sequence alignment, phylogenetics, etc. In order to achieve this objective, we have carved out a set of tasks from the whole genome shotgun sequencing process, identified multiple objectives that need to be optimized, and implemented a new algorithm that improves the overall assembly process from many aspects.

We have also demonstrated that using the low quality end regions, instead of trimming them off, may be a better option in the parallel computing environment where we can leverage the available computing cycles to make use of certain data that would otherwise be either trimmed or would introduce errors and in both cases would require significant time to process. This also improves the utilization of resources. Compared to some contemporary work where the concept of clustering has been used, the use of enhanced version of Union-Find data structure and a special purpose hash function allow space efficient storage and fast searching capability enhancing the overall performance. Our algorithm is different from some other comparable tools in many aspects. For example, PaCE [52] clusters ESTs to feed a sequential assembler whereas our framework also lends itself to parallel computation in the subsequent phases of the assembly process. Also, our hash function and the supporting data structure lend themselves well to the parallel processing as compared to some other software tools that build suffix arrays with a disadvantage of slow construction and large memory requirement.

Also, being a stand-alone modular component, our implementation can be used to study how different objectives conflict with each other due to common decision variables, e.g. number of processors, data size, etc. With the aim to achieve multi-objective optimization of sequence assembly algorithms, we have proved that multiple and often conflicting objectives in the area of genome assembly can be optimized leveraging the parallel computing environment.

Finally, and most importantly, we analyzed our algorithm using the well-established metrics of parallel computing paradigm to derive a theoretical model based on the scalability analysis of our algorithm and compare that with the statistical results that we obtained by running real data through our program on a cluster of 128 processors. Finally, we tested and validated our assumptions and plotted the results confirming our analysis and illustrating the deviations from the theoretical models.

## 4.7 Future Directions

Ever since its first successful use, Sanger sequencing [83] technique has been used almost exclusively in all large-scale projects and has been the method of choice to obtain the reads used by assemblers. However, with the advent of new technologies [11], [65], sometimes referred to as next-generation sequencing, shorter reads containing fewer than 100 bases are obtained and are comparatively more cost effective, i.e. various methods are being deployed to generate low cost small-sized sequences in larger numbers. This is achieved by parallelizing the sequencing process producing millions of sequences at once. Following work has been mentioned in the research and is considered to be promising a successful and long overdue shift in sequencing technologies that would open new frontiers for bioinformaticians, especially in the field of genomics. There is a need to develop a new set of parallel algorithms based upon the next generation sequencing.

One of the methods used to determine the nucleotide sequence is based upon synthesizing a complementary strand of the DNA under consideration. This method uses a technique reported by Ronaghi et al. [80], which detects the activity of DNA polymerase when introduced to other enzymes, unlike the chain termination using dideoxynucleotides as used in Sanger sequencing. An implementation by 454 Sequencing, which is based on pyrosequencing, uses a large-scale parallel system to process around 400 to 600 Mbp taking 10-hour of runtime [53]. Another method developed by Illumina, also based on the concept of sequencing by synthesis like *pyrosequencing*, uses reversible terminators. A fluorescently-labeled terminator is imaged as each deoxyribonucleotide triphosphate (dNTP) is added and then cleaved to allow incorporation of the next base [62]. There is also a technique that is based on the idea of “sequencing by ligation”. Invitrogen’s SOLiD technology employs this technique which uses DNA ligase enzyme to detect the nucleotide at a given position in the sequence and does not use a DNA polymerase to create a complementary strand like other sequencing techniques mentioned above. The mismatch sensitivity of DNA ligase enzyme helps determine the

sequence of the DNA molecule under study [93]. Other techniques also exist and new ones are being developed as we write these lines. Those include Nanoball sequencing [76] and Hydrogen ion based sequencing [81]. A detailed review of all such technologies is outside of the scope of this dissertation. However, we strongly recommend the references provided here for more insight of these emerging sequencing methodologies and the potential for parallelization of many of them.

REFERENCES


[1] ABIABI PRISM, “DNA sequencing analysis software, user’s manual”. PE Applied Biosystems, Foster City, CA. 1996.

[2] Ahmed, M., Ahmad, I., and Khan, S. “A Theoretical Analysis of Scalability of the Parallel Genome Assembly Algorithms”, 2nd International Conference on Bioinformatics Models, Methods and Algorithms, p. 234-237, January 2011

[3] Ahmed, M., Ahmad, I., and Khan, S. “A Comparative Analysis of Approaches to Parallel Genome Assembly”. Journal of Interdisciplinary Sciences: Computational Life Sciences, 3: 1-7, 2011 DOI 10.1007/s12539-011-00

[4] Ahmed, M., Ahmad, M., and Ahmad, I. "A Multi-pronged Parallel Approach to Enhance Speed and Accuracy of Sequence Assembly", Biotechnology and Bioinformatics Symposium. 2008.

[5] Aho, A. V., Hopcroft, J. E. and J. D. Ullman. The Design and Analysis of Computer Algorithms. Addison -Wesley, Reading, Mass., 1974.

[6] Altschul, S.F., Madden, T.L., Schaffer, A.A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D.J. “Gapped BLAST and PSI-BLAST: A new generation of protein database search programs”. Nucleic Acids Res. 25: 3389–3402. 1997.

[7] Aluru, S., Futamura, N., and Mehrotra, K. "Parallel biological sequence comparison using prefix computations," Journal of Parallel and Distributed Computing, Vol. 63, No. 3, pp. 264-272, 2003.

[8] Anbarasu, L. “Multiple sequence alignment using parallel genetic algorithms”, Second Asia-Pacific Conference on Simulated Evolution. 1998.

[9] Bao, Z. and Eddy, S. “Automated De Novo Identification of Repeat Sequence Families in Sequenced Genomes”, Genome Res. 8: 1269-1276. 2002.

[10] Batzoglou, S., Jaffe, D., Stanley, K., Butler, J., Gnerre, S., Mauceli, E., Berger, B., Mesirov, J., and Lander, E. “Arachne: A Whole-Genome Shotgun Assembler”, Genome Research, vol. 12, no. 1, pp. 177-189. 2002.

[11] Bentley, D. “Whole-genome re-sequencing”, Curr. Opin. Genet. Dev. 16:545–552. 2006.

[12] Berger, M. and Munson, P. "A novel randomized iterative strategy for aligning multiple protein sequences", CABIOS 7, 479-484. 1991.

[13] Blackshields G., Wallace, I., Larkin. M., and Higgins, D. “Analysis and comparison of benchmarks for multiple sequence alignment”, In Silico Biol. 2006;6:321–339. 2006.

[14] Blazewicz, J., Figlerowicz, M., Jackowiak, P., Janny, D., Jarczynski, D., Kasprzak, M., Nalewaj, M., Nowierski, B., Styszynski, R., Szajkowski, L., and Widera. P. “Parallel DNA Sequence Assembly”, In Proceedings of the Fifth Mexican International Conference in Computer Science (ENC '04). IEEE Computer Society, pp378-382. 2004.

[15] Brudno, M. and Batzoglou, S. “ProbCons: probabilistic consistency-based multiple alignment of amino acid sequences”, Proceedings Nineteenth National Conference on Artificial Intelligence, 703-708. 2004.

[16] Chao, K., Pearson, W., and Miller, W. “Aligning two sequences within a specified diagonal band”, Comput. Applic. Biosci. 8: 481–487. 1992.

[17] Cheetham, J., Dehne, F., Pitre, S., Rau-Chaplin, A., and Taillon, P. “Parallel CLUSTAL W for PC Clusters”, ICCSA, LNCS 2668, pp. 300–309. 2003

[18] Darling, A., Carey, L., and Feng, W. “The Design, Implementation, and Evaluation of mpiBLAST”, 4th International Conference on Linux Clusters: The HPC Revolution 2003 in conjunction with the ClusterWorld Conference & Expo, 2003

[19] Deng, X., Li, E., Shan, J., and Chen, W. “Parallel implementation and performance characterization of MUSCLE”, Parallel and Distributed Processing Symposium. 2006.

[20] Dovichi, N. and Zhang, J. “How Capillary Electrophoresis Sequenced the Human Genome”, Angewandte Chemie, International Edition, vol. 39, pp. 4463-4468. 2000.

[21] Du, Z. and Lin, F. “pNJTree: A parallel program for reconstruction of neighbor-joining tree and its application in ClustalW”, J. of Parallel Computing, vol. 32, 5-6. 2006.

[22] Ebedes, J. and Datta, A. “Multiple sequence alignment in parallel on a workstation cluster”, Bioinformatics, vol. 20, no. 7. 2004.

[23] Edgar, R. 2004. MUSCLE: multiple sequence alignment with high accuracy and high throughput. Nucleic Acids Res, vol. 32, pp. 1792-1797. 2004.

[24] Edgar, R. and Myers, E. “PILER: Identification and Classification of Genomic Repeats”, Bioinformatics, 1;21 Suppl 1:i152-i158. 2005.

[25] Essoussi, N., Boujenfa, K., and Limam, M. “A comparison of MSA tools”, Bioinformation. 2008.

[26] Ewing, B., Hillier, L., Wendl, M., and Green, P. “Base-calling of automated sequencer traces using phred. I. Accuracy assessment”, Genome Res. 8(3):175-185. PMID 9521921. 1998.

[27] Felsenfeld, A., Peterson, J., Schloss, J., and Guyer, M. “Assessing the Quality of the DNA Sequence from The Human Genome Project”, Genome Research, 9:1-4. 1999.

[28] FlyBase: A database of Drosophila genes and genomes. http://flybase.net/

[29] Gao, W. and Qiao, S. “Multithreaded implementation of a biomolecular sequence alignment algorithm-software/information technology,” Canadian Conference on Electrical and Computer Engineering, Vol1, pp. 494 -498. 2000.

[30] Gao, G. “Landing ATCG on EARTH: Parallel Computing and Whole-Genome Alignment,” A presentation at Delaware Biotechnology Institute, University of Delaware. 2001.

[31] Galtier, N., Gouy, M. and Gautier, C., “SeaView and Phylo_win, two graphic tools for sequence alignment and molecular phylogeny”, Comput. Applic. Biosci., 12, 543-548, 1996.

[32] Galil, Z. and Italiano, G. “Data structures and algorithms for disjoint set union problems”, in ACM Computing Surveys, Volume 23, Issue 3, pp. 319-344, Sept. 1991.

[33] Grama, A., Gupta, A., and Kumar, V. “Isoefficiency: Measuring the scalability of parallel algorithms and architectures”, IEEE parallel and Distributed Technology, 1(3):12-21. 1993.

[34] Green, P. http://bozeman.mbt.washington.edu/phrap.docs/phrap.html

[35] Griffiths, A., Gelbart, W., Lewontin, R., and Miller, J. “Modern Genetic Analysis, Integrating Genes and Genomes”, 2nd ED., W. H. Freeman and Company, New York, 2002.

[36] Gropp W., Lusk, E., Doss, N., and Skjellum, A. “A high-performance, portable implementation of the MPI message passing interface standard”, Parallel Computing, vol. 22, pp. 789-828. 1996.

[37] Gordon, D., Abajian, C., and Green, P. “Consed: A Graphical Tool for Sequence Finishing”, Genome Research. 8:195-202. 1998.

[38] Gusfield, D. “Algorithms on Strings, Trees and Sequences”, Cambridge, England: Cambridge University Press. 1997.

[39] Higgins, D. and Sharp, P. "CLUSTAL: a package for performing multiple sequence alignment on a microcomputer". Gene 73 (1): 237–44. 1988.

[40] Higgins, D. “CLUSTAL V: multiple alignment of DNA and protein sequences”, Methods Mol Biol., 25, 307-318. 1994.

[41] Hirosawa, M., Totoki, Y., Hoshida, M., and Ishikawa, M. “Comprehensive Study on Iterative Algorithms of Multiple Sequence Alignment”, Comput. Applic. Biosci., 11:13–18. 1995.

[42] Huang X., Wang, J., Aluru, S., Yang, S., and Hillier, L. “PCAP: a whole-genome assembly program”, Genome Research, vol. 13, 2164–2170. 2003.

[43] Huang, X. and Madan, A. “CAP3: A DNA Sequence Assembly Program”, Genome Research, vol. 9, no. 9, pp. 868-877. 1999.

[44] Hubbard, T., Aken, B., Beal, K., Ballister, M., Caccamo, Y., Chen, L., Clarke, G., Coates, F., Cunningham, T., Cuts, T., Down, S., Dyer, S., Fitzgerald, J., Fernandez-Banet, S., Graf, S., Haider, M., Hammond, J., Herrero, R., Holland, K., Howe, K., Howe, N., Johnson, A., Kahari, D., Keefe, F., Kokocinski, E., Kulesha, D., Lawson, I., Longden, C., Melsopp, K.,

Megy, P., Meidl, B., Ouverdin, A., Parker, A., Prlic, S., Rice, D., Rios, M., Schuster, I., Sealy, J., Severin, G., Slater, D., Smedley, G., Spudich, S., Trevanion, A., Vilella, J., Vogel, S., White, M., Wood, T., Cox, V., Curwen, R., Durbin, X., Fernandez-Suarez, P., Flicek, A., Kasprzyk, G., Proctor, S., Searle, J., Smith, A., Ureta-Vidal, E. “Ensembl 2007”, Nucleic Acids Research., vol. 35, Database issue:D610-D617. 2007.

[45] Kalyanaraman, A., Kothari, S., Brendel, V., Aluru, S. “Efficient clustering of large EST data sets on parallel computers”, in Nucleic Acids Research, 31(11):2963-2964, 2003.

[46] Isokawa, M., Wayama, M., and T. Shimizu, "Multiple sequence alignment using a genetic algorithm," Genome Informatics, 7, 176-177. 1996.

[47] Ishikawa, M., Toya, T., Hoshida, M., Nitta, K., Ogiwara, A., and Kanehisa, M. “Multiple sequence alignment by parallel simulated annealing”, Bioinformatics, vol. 9, no. 3. 1993.

[48] Jeanmougin, F., Thompson, J., Gouy, M., Higgins, D., Gibson, T. “Multiple sequence alignment with Clustal X”, Trends Biochem Sci., 23, 403-405. 1998.

[49] Johnson, D., Metaxas, P. “Connected Components in O(log3/2n) Parallel Time for the CREW PRAM”, Journal of Computer and System Sciences, Volume 54, Number 2, pp. 227-242(16). 1997.

[50] Jones, T., Federspiel, N., Chibana, H., Dungan, J., Kalman, S., Magee, B., Newport, G., Thorstenson, Y., Agabian, N., Magee, P., Davis, R., Scherer, S., “The diploid genome sequence of Candida albicans”, Proceedings of the National Academy of Science of the United States of America. 2004.

[51] Kahsay, R., Dongre, N., Gao, G., Wang, G., and Dunbrack, R. “CASA: A Server for The Critical Assessment of Sequence Alignment Accuracy,” Bioinformatics, vol. 18, No. 3. 2002.

[52] Kalyanaraman, A., Aluru, S., Brendel, V. and Kothari, S. “Space and time efficient parallel algorithms and software for EST clustering”, IEEE Trans. Parall. Distrib. Syst. 14, 1209-1221. 2003.

[53] Voelkerding, K., Dames, S., and Durtschi, J. "Next Generation Sequencing: From Basic Research to Diagnostics", Clinical Chemistry 55 (4): 41–47. doi:10.1373/clinchem.2008.112789. PMID 19246620. 2009.

[54] Kao, W., Song, Y. “naiveBayesCall: an efficient modelbased base-calling algorithm for high-throughput sequencing”, Research in Computational Molecular Biology, volume 6044/2010 of Lecture Notes in Computer Science. Springer Berlin/Heidelberg: 233–47. 2010

[55] Konak, A., Coit, D., and Smith, A. “Reliability Engineering & System Safety”, in Special Issue - Genetic Algorithms and Reliability, Vol. 91, No. 9, pp. 992-1007. 2006.

[56] Larkin, M., Blackshields, G., Brown, N., Chenna, R., McGettigan, P., McWilliam, H., Valentin, F., Wallace, A., Wilm, R., Lopez, Thompson, J., Gibson, T., and Higgins, D. “Clustal W and Clustal X version 2.0”, Bioinformatics, 23, 2947-2948. 2007.

[57] Lander, E., Linton, L., and Birren, B. “Initial sequencing and analysis of the human genome”, Nature, 409, 860 – 921. 2001.

[58] Li, K. “ClustalW-MPI: ClustalW analysis using distributed and parallel computing”, Bioinformatics, vol. 19 no. 12. 2003.

[59] Lipman, D., Pearson, W. "Rapid and sensitive protein similarity searches". Science 227 (4693): 1435–41. doi:10.1126/science.2983426. PMID 2983426. 1985.

[60] Lipman, L. and Kececioglu, "A Tool for Multiple Sequence Alignment", Proc. Natl. Acad. Sci. 86:4412-4415. 1989.

[61] Luo, J., Ahmad, I., and Ahmed, M. “Parallel Multiple Sequence Alignment using Dynamic Scheduling”, International Conference on Information Technology: Coding and Computing, vol. 1, page(s): 8- 13. 2005.

[62] Mardis, E. "Next-generation DNA sequencing methods". Annu Rev Genomics Hum Genet 9: 387–402. doi:10.1146/annurev.genom.9.081307.164359. PMID 18576944. 2008.

[63] Martins, W., Cuvillo, J., Cui, W., and Gao, G. “Whole Genome Alignment Using a Multithreaded Parallel Implementation,” Symposium on Computer Architecture and High Performance Computing, pp 1-8. 2001.

[64] Martins, W., Cuvillo, J., Francisco, B., Theobald, J., and Gao, G. “A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. In Proceedings of the Pacific Symposium on Biocomputing, pages 311–322. 2001.

[65] Metzker, M. “Emerging technologies in DNA sequencing”, Genome Research, vol 15:1767–1776. 2005.

[66] Miller, P., Nadkarni, P., and Carriero, N. “Parallel computation and FASTA: confronting the problem of parallel database search for a fast sequence comparison algorithm”, Comput Appl Biosci 7(1): 71-78 doi:10.1093/bioinformatics/7.1.71. 1991.

[67] Morgenstern, B. “DIALIGN 2: improvement of the segment-to-segment approach to multiple sequence alignment”, Bioinformatics, 15:211-218. 1999.

[68] Mullikin, J. and Ning, Z. “The Phusion Assembler”, Genome Research. (1):81-90. 2003.

[69] Myers, E., Sutton, G., Smith, H., Adams, M., and Venter, J. “On the sequencing and assembly of the human genome”, Proc Natl Acad Sci 99(7):4145-6. 2002.

[70] Needleman, S. and Wunsch, C. “A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Sequences,” Journal of Molecular Biology, vol. 48, pp. 443-453. 1970.

[71] Notredame, C., “Recent progress in multiple sequence alignment: a survey”, Pharmacogenomics, 3(1):131-144. 2002.

[72] Notredame, C., Higgins, D. G. and Heringa, J. “T-Coffee: A novel method for fast and accurate multiple sequence alignment”. J. Mol. Biol. 302, 205-217. 2000.

[73] Ogden, T. and Rosenberg, M. “Multiple Sequence Alignment Accuracy and Phylogenetic Inference”, Syst. Biol. 55(2):314–328. 2006.

[74] Pevzner, P., Tang, H., and Waterman, S. “An Eulerian Path Approach to DNA Fragment Assembly”, Proceedings of National Academy of Sciences of the United States of America, 98(17):9748-9753. 2001

[75] Pevzner, P., Tang, H., and Tesler. G. “De Novo Repeat Classification and Fragment Assembly Genome Res.”, 14(9): 1786 - 1796. 2004.

[76] Porreca, G. "Genome sequencing on nanoballs". Nature Biotechnology 28 (1): 43–4. doi:10.1038/nbt0110-43. PMID 20062041. 2010.

[77] Ralston, A. “De Bruijn Sequences--A Model Example of the Interaction of Discrete Mathematics and Computer Science”, Math Mag. 55, 131-143. 1982.

[78] Reif, J. and Spirakis, P. “Expected parallel time and sequential space complexity of graph and digraph problems”, Algorithmica 7, 597--630. 1992.

[79] Sanger, F., Nicklen, S., and Coulson, A. “DNA sequencing with chain-terminating inhibitors”, Proceedings of the National Academy of Sciences, USA 74: 5463-7. 1977.

[80] Ronaghi, M., Uhlen, M., Nyren, P. "A sequencing method based on real-time pyrophosphate". Science 281 (5375): 363. doi:10.1126/science.281.5375.363. PMID 9705713. 1998.

[81] Rusk, N. "Torrents of sequence." Nat Meth 8(1): 44-44. 2011.

[82] Saitou, N. and Nei, M. “The neighbor-joining method: a new method for reconstructing phylogenetic trees”, Mol. Biol. Evol., 4, 406–425. 1987.

[83] Sanger, F., Nicklen, S., and Coulson, A. “DNA sequencing with chain-terminating inhibitors”, Proceedings of the National Academy of Sciences, USA 74: 5463-7. 1977.

[84] Schmollinger, M., Nieselt, K., Kaufmann, M., and Morgenstern, B. “DIALIGN P: Fast pair-wise and multiple sequence alignment using parallel processors”, BMC Bioinformatics. 2004.

[85] Simpson, J., Wong, K., Jackman, S., Schein, J., Jones, S., and Birol, I. “ABySS: a parallel assembler for short read sequence data”, Genome Res. 2009;19:1117-1123. 2009.

[86] Shi, W. and Zhou, W. “A Parallel Euler Approach for Large-Scale Biological Sequence Assembly”, Proceedings of the Third International Conference on Information Technology and Applications. 2005.

[87] Smit, A., Hubley, R. and Green, P. “RepeatMasker Open-3.0”, http://www.repeatmasker.org. 1996-2010.

[88] Smith, T. and Waterman, M.. “Identification of common molecular subsequences”, Journal of Molecular Biology 147, 195-197. 1981.

[89] Southern, E. “Detection of Specific Sequences Among DNA Fragments Separated by Gel Electrophoresis”, Journal Of Molecular Biology. 98, 503-517. 1975.

[90] Sutton, G., White, O., Adams, M., Kerlavage, A. “TIGR Assembler: A new tool for assembling large shotgun sequencing projects”, Genome Science and Technology. 1(1): 9-19. 1995.

[91] Teo, Y., Wang, X., and NG, Y. “GLAD: A system for developing and deploying large-scale bioinformatics Grid”, Bioinformatics. 2004.

[92] Thompson, J., Plewniak, F., and Poch, O. "A comprehensive comparison of multiple sequence alignment programs", Nucleic Acids Research 27 (13): 12682–2690. 1999.

[93] Valouev, A., Ichikawa, J., Tonthat, T., Stuart, J., Ranade, S., Peckham, H., Zeng, K., Malek, J., Costa, G., McKernan, K., Sidow, A., Fire, A., Johnson, S. "A high-resolution, nucleosome position map of C. elegans reveals a lack of universal sequence-dictated positioning". Genome Res. 18 (7): 1051–63. doi:10.1101/gr.076463.108. PMC 2493394. PMID 18477713. 2008.

[94] Venter, J., Adams, M., and Myers, E. “The Sequence of the Human Genome," Science 16; 291: 1304-1351. 2001.

[95] Vinson, J., Jaffe, D., O'Neill, K., Karlsson, E., Stange-Thomann, N., Anderson, S., Mesirov, J., Satoh, N., Satou, Y., Nusbaum, C., Birren, B., Galagan, J., Lander, E. “Assembly of polymorphic genomes: Algorithms and application to Ciona savignyi”, Genome Research, 15:1127-1135, 2005.

[96] Volfovsky, N., Haas, B., and Salzberg, S. “A Clustering Method for Repeat Analysis in DNA Sequences”, Genome Biol. 2: RESEARCH0027, 2001.

[97] Watson, J. and Crick, F. “Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid”, Nature, 171:737-738. 1953.

[98] Yap, T., Munson, P., Frieder, O., and Martino, R. “Parallel Multiple Sequence Alignment Using Speculative Computation,” Proceedings of the 1995 International Conference on Parallel Processing. 1995.

[99] Yap, T., Frieder, O., and Martino, R. “Parallel Computation in Biological Sequence Analysis,” IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 3. 1998.

[100] Zhang, C. and Wong, A. "A genetic algorithm for multiple molecular sequence alignment," Comput. Applic. Biosci., 13(6), 565-581. 19997.

[101] Zhao, F., Zhao, F., Li, T., and Bryant, D. “A new pheromone trail-based genetic algorithm for comparative genome assembly”, Nucleic Acids Res., 36(10): 3455 - 3462. 2008.

[102] Zola, J., Yang, X., Rospondek, S., Aluru, S., “Parallel T-Coffee: A Parallel Multiple Sequence Aligner”, In Proc. of ISCA PDCS-2007, pp. 248-253, 2007.

## BIOGRAPHICAL INFORMATION

Munib Ahmed earned his Bachelor of Engineering degree from N.E.D University of Engineering and Technology at Karachi in 1990. He received Master's in Computer Science Engineering in 1992 and Ph.D. in Computer Engineering in 2011, from the University of Texas at Arlington. He began his professional career in the field of Information Technology in 1992 and worked at American Airlines, Sabre Group, and Electronic Data Systems before joining Hewlett-Packard (HP) in 2008, where he is currently employed as a Senior Systems Architect. His near term plan is to continue his career at HP while staying involved with academia taking research and part-time teaching opportunities as they become available. His favorite pastimes are reading, listening to country music, and watching documentaries on science and nature, with biking, cricket, and swimming being the sports of choice.