University of South Carolina
Scholar Commons

Theses and Dissertations

12-15-2014

High-Performance Meta-Genomic Gene Identification

Ibrahim Savran
University of South Carolina - Columbia

Follow this and additional works at: https://scholarcommons.sc.edu/etd

 Part of the Computer Engineering Commons

Recommended Citation

Savran, I.(2014). *High-Performance Meta-Genomic Gene Identification.* (Doctoral dissertation). Retrieved from https://scholarcommons.sc.edu/etd/2991

This Open Access Dissertation is brought to you by Scholar Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact digres@mailbox.sc.edu.

High-performance meta-genomic gene Identification

by

Ibrahim Savran

Bachelor of Engineering
Selçuk University 2003
Master of Science
Selçuk University 2006

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2014

Accepted by:

John R. Rose, Major Professor

Manton M. Matthews, Committee Member

Homayoun Valafar, Committee Member

Achraf El Allali, Committee Member

Sean Norman, Committee Member

Lacy Ford, Vice Provost and Dean of Graduate Studies

© Copyright by Ibrahim Savran, 2014
All Rights Reserved.

Dedication

Dedicated to

Dearest Mother and beloved Merve

Acknowledgments

It took me a long period of time to finish this dissertation. It involved the support, patience and guidance of my friends, family and professors. Any words of gratitude would not be sufficient for expressing my feelings.

I would like to thank my committee members Dr. Manton M. Matthews, Dr. Homayoun Valafar, Dr. Sean Norman and Dr. Achraf El Allali. Their guidance, suggestions and feedback at various instances have greatly benefited this work and my knowledge. I would also thank them for patiently reading this dissertation and approving of this research.

It would have been impossible for me to pursue the research without the guidance and instruction of Dr. John R. Rose. He not only has been supportive but also encouraging. His dedication, commitment and enthusiasm kept me motivated and inspired. Foremost, he has been patient with me throughout. I cannot imagine a better mentor, guide and adviser for my PhD.

I would like to express my gratitude towards my colleges in the Heterogeneous and Reconfigurable Computing Lab especially Zheming Jin and Fan Zhang and Jeremy Lane with whom I benefited from their valuable suggestions and discussions. I would also thank Dr. Krishna Nagar, with whom I prepared for the qualifying exam and had discussion on various many research topics.

My family has always provided me unconditional support and have always kept me motivated. I sincerely appreciate their support.

Abstract

Computational Genomics, or Computational Genetics, refers to the use of computational and statistical analysis for understanding the structure and the function of genetic material in organisms. The primary focus of research in computational genomics in the past three decades has been the understanding of genomes and their functional elements by analyzing biological sequence data.

The high demand for low-cost sequencing has driven the development of high-throughput sequencing technologies, next-generation sequencing (NGS), that parallelize the sequencing process, producing thousands or millions of sequences concurrently. Moore's Law is the observation that the number of transistors on integrated circuits doubles approximately every two years; correspondingly, the cost per transistor halves. The cost of DNA sequencing declines much faster, which implies more new DNA data will be obtained.

This large-scale sequence data, produced with high throughput sequencing technologies, needs to be processed in a time-effective and cost-effective manner.

In this dissertation, we present a *high-performance meta-genome gene identification framework*. This framework includes four modules: *filter, alignment, error correction*, and *gene identification*. The following chapters describe the proposed design and evaluation of this pipeline.

The most computationally expensive kernel in the framework is the alignment procedure. Thus, the filter module is developed to determine unnecessary alignment operations. Without the filter module, the alignment module requires 1.9 hours to complete all-to-all alignment on a test file of size 512,000 sequences with each sequence

average length 750 base pairs by using ten Kepler K20 NVIDIA GPU. On the other hand, when combined with the filter kernel, the total time is 11.3 minutes. Note that the ideal speedup is nearly 91.4 times faster when new alignment kernel is run on ten GPUs (10*9.14). We conclude that accuracy can be achieved at the expense of more resources while operating frequency can still be maintained.

Table of Contents

Dedication iii

Acknowledgments iv

Abstract v

List of Tables ix

List of Figures xi

Chapter 1 Introduction 1

Chapter 2 Background 5

2.1 DNA: Deoxiribo Nucleic Acid 5

2.2 Sequencing Technologies 6

2.3 Sequence Alignment 9

2.4 High-performance Computing 12

Chapter 3 Filter Design 16

3.1 Motivation 16

3.2 Implementation 17

3.3 Results 26

Chapter 4 Alignment Module 33

4.1 Motivation 33

4.2 Background 33

4.3 Results 39

Chapter 5 Genetic sequence error correction 44
5.1 Motivation 44
5.2 Background 44
5.3 Method 47
5.4 Implementation 51
5.5 Results 53

Chapter 6 Meta-Genome Gene Identification 57
6.1 Motivation 57
6.2 Background 58
6.3 The MGC algorithm 59
6.4 Implementation of MGC model 62
6.5 Results 64

Chapter 7 Conclusion 68
7.1 Future work 69

Bibliography 70

List of Tables

Table 2.1 Comparison of next-generation Sequencing Methods 9

Table 3.1 The suffixes of the string, "banana", and its suffix array and LCP array 20

Table 3.2 The Skew Algorithm on the string, "banana": (a,b) sorting the subarrays, and (c) merging 21

Table 3.3 Performance Summary of the SACA Algorithms 24

Table 3.4 Similar data set : genomes are from – **Bacillus Genus** 26

Table 3.5 Dissimilar data set : genomes are from –**Proteobacteria Phylum** 27

Table 3.6 Test files, average sequence length is represented by l_{avg} 28

Table 3.7 Comparison of Stampede computing node with GPU used in our tests. 29

Table 3.8 PaCE timing results on the benchmarks on different clusters 40, 60, 80, 100 CPUs (fixed match length – 12) 29

Table 3.9 1 NVIDIA K20 GPU timing results for running filter kernel 30

Table 3.10 10 NVIDIA K20 GPUs timing results for running filter kernel . . . 31

Table 3.11 The filter kernel run-time results on 20 NVIDIA K20 GPUs . . . 32

Table 4.1 CPU vs. Single GPU execution time in seconds. 39

Table 4.2 Multiple alignment run time results of the modified alignment kernel on a 10-GPU cluster 41

Table 4.3 New filter-align module run time results on 1-GPU cluster 42

Table 4.4 Multi-GPU performance results 42

Table 4.5 Run time comparison of G-DNA with our Alignment kernel only – AK 43

Table 5.1 Correction code table 48

Table 5.2 Substitution error in $String_4$, and deletion error in $String_1$ 49

Table 5.3 Two strings share a substring m, where $|m| \geq min_match_length$ 52

Table 5.4 Comparing timing of DecGPU and my Error correction Kernel (when number of iterations is set to 10 for DecGPU) 54

Table 5.5 Definitions for the classification test 55

Table 5.6 Summary of the classification test for simulated datasets 56

Table 6.1 The timing result for extracting ORF sequences 66

Table 6.2 Orphelia run time results for gene identification 66

Table 6.3 HP-MGC model timing results 67

List of Figures

Figure 1.1 **Cost per Genome:** the cost of sequencing a human-sized genome [8, 9, 10]. 2

Figure 1.2 High-performance meta-genome gene identification framework. . 3

Figure 2.1 Memory hiearachy of a GPU thread 14

Figure 3.1 Filter module takes raw meta-genomic sequences and create coarse groupings. 17

Figure 3.2 The suffix tree for the string "BANANA" (The first leaf node which represents $ character is removed). 18

Figure 3.3 PaCE timing results on the benchmarks on different clusters 40, 60, 80, 100 CPUs 30

Figure 3.4 Filter module timing: 1 NVIDIA K20 GPU timing results for running filter kernel 31

Figure 3.5 10 NVIDIA K20 GPUs timing results for running filter kernel . . 31

Figure 3.6 20-GPU cluster timing results 32

Figure 4.1 Alignment Module 33

Figure 4.2 Example Needleman-Wunsch alignment between two sequences. . 36

Figure 5.1 Error correction module phase 1: scanning process is for determining and reporting misinterpreted read positions and error codes into a table for all sequences that are approved by alignment module. 51

Figure 5.2 Error correction module phase 2: voting and fixing process gets the error table, orders the table which helps decide whether there should be a correction reporting misinterpreted read positions and error codes for all sequences that are selected by the alignment module. 51

Figure 5.3 Comparing timing of DecGPU and my Error correction Module (when number of iterations is set to 10 for DecGPU) 53

Figure 5.4 The efficiency of DecGPU and my Error correction Module: (a) on the S-256 and S-512 test files 55

Figure 6.1 **The possible ORF positions within the forward strand of a fragment.** The fragment is depicted by the outside box and gray bars represent possible ORFs. Candidate translation initiation sites are represented by green pentagons and red squares indicate stop codons. (Obtained from El Allali and Rose [109]) 60

Figure 6.2 **MGC's scoring scheme:** The first steps computes six features from the ORF based on the corresponding linear discriminant and two additional features are computed directly from the ORF. The last feature is derived from directly the fragment. The neural network model from the corresponding GC-range is used to combine features from the previous step in order to compute a final gene probability. (Obtained from El Allali and Rose [109]) 61

Chapter 1

Introduction

The traditional approach of isolating and culturing microbes has had limited success in determining the diversity of a microbial community. It is estimated that only 1-10% of all microbial species can be cultured [1, 2, 3, 4]. The new approach is to access this wealth of genetic information through environmental DNA extraction, which has provided a means of avoiding the limitations of culture-dependent genetic exploitation.

From the pioneering experiment of Sanger and Coulson until now, sequence analysis has been the core study of molecular biology. In the past three decades, numerous projects have successfully deciphered the genomes of various species with corresponding structural and functional annotations.

Sequencing technology has evolved rapidly over the last decade, especially after 2007, with the advantages of lowering cost-per-base and increasing the throughput. As an example, the cost of the Human Genome Project (HGP) was around three billion dollars in 2001 and it took over ten years to complete.

The ongoing revolution of next-generation sequencing (NGS) technologies has led to the production of high-throughput short read (HTSR) data at dramatically lower cost compared to conventional sequencing technologies. As an example, the cost of the Human Genome Project (HGP) was around three billion dollars in 2001 and it took over ten years to complete. Using today's next-generation sequencing (NGS) techniques, a human-sized genome can be sequenced for the cost of one thousand dollars in a day.

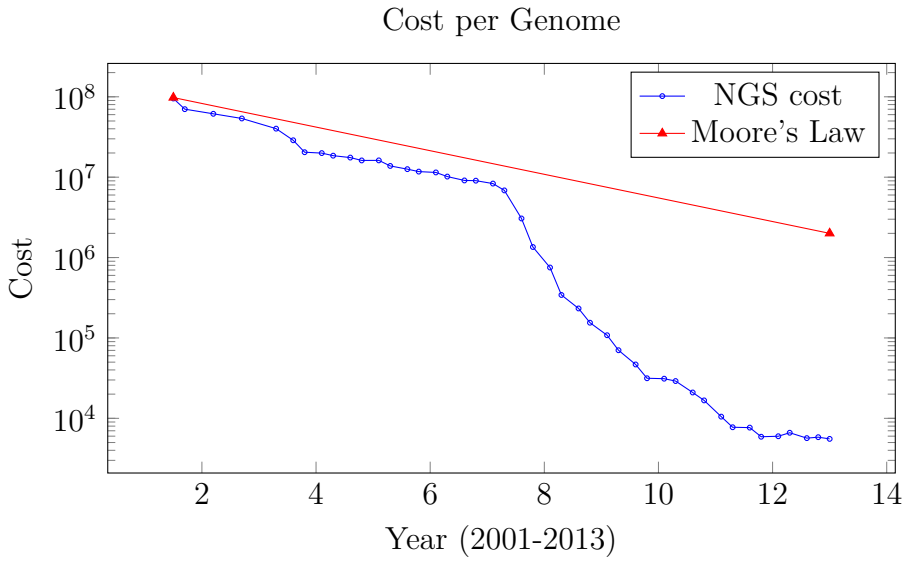


Figure 1.1: **Cost per Genome:** the cost of sequencing a human-sized genome [8, 9, 10].

Figure 1.1 shows the reduction of DNA sequencing costs over the past decade relative to Moore's Law. In the graph, a logarithmic scale is used on the Y axis. The sudden and profound out-pacing of Moore's Law can be seen after 2007.

Designing computational solutions for analyzing NGS data is challenging for a number of reasons:

1. **Large-scale data:** The rapid rate of biological sequence cultivation with NGS technology has led to a rapid growth of publicly available sequence data sets. Processing large-scale data imposes huge memory and run-time requirements.

2. **Computational requirements:** Many problems that involve sequence analysis are computationally difficult. Even polynomial solutions often require a large run-time and a huge memory for large-scale data sizes. As an example, one main operation called "alignment" is the computation between two strings. Using dynamic programming, computing an optimal alignment takes $O(l^2)$ time, and $O(l)$ space where l is the length of two strings $l = |s_1| = |s_2|$. However,

solving the dynamic programming problem for multiple sequence alignments rapidly becomes intractable – $O(\binom{n}{2} \times l^2)$.

My focus will be on solving problems for making core level discoveries of genomic data. The contribution of this study will be to construct a *high-performance meta-genome gene identification framework*.

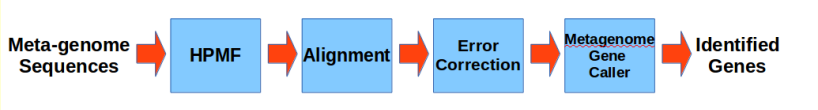


Figure 1.2: High-performance meta-genome gene identification framework.

The pipeline of the framework can be seen from the figure 1.2, the raw meta-genome sequences are the input for the filter module. The filter will make a coarse clustering of the raw set. Then, the alignment procedure will refine the boundaries of each cluster. Next, the error correction module will repair errors made by the sequencer which will help to boost specifying open reading frames for the MGC module. Finally the MGC module will extract genes from the sequences.

1. **Filter Module:** This module makes a coarse grouping of the raw set.

2. **Alignment Module:** The alignment module will obtain the list of promising pairs as its input and run a global alignment procedure to produce the final clusters from the collection of sequences produced by the filter module.

3. **Error correction module:** This method will fix the misinterpreted bases by the sequencer which will help to boost specifing open reading frames for the next module.

4. **MGC module:** Finally MGC module will extract genes from the sequences.

The main goal of this work will be the development of methods that can scale to the largest available sequence data sets. This work is organised as follows. Chapter

2 provides a brief overview of the biological concepts required to understand the problems and applications described in this study.

The following list depicts the portions of the proposed framework I will implement and test:

I will formulate the meta-genome gene identification problem. Then, I will provide an extensive review of literature describing the various computational methods. From Chapter 3 to chapter 6 I will describe the high-performance meta-genome gene identification framework approach. Chapter 3 will cover the filter partition. Chapter 4 will cover the alignment module. Chapter 5 will cover the error correction module, as well as identification of possible start and stop codon. Chapter 6 will cover MGC, the module that identifies genes. Chapter 7 concludes the study with a summary and with a discussion of future research directions.

Chapter 2

Background

2.1 DNA: Deoxiribo Nucleic Acid

DNA or Deoxyribo-Nucleic Acid is one of the fundamental molecular entities inside a cell of a living organism. DNA is not only the hereditary material of an organism, but also encodes the genetic instructions to carry out its cellular development. The DNA of an organism is inherited primarily from ancestors. For example, we resemble our parents simply because our bodies were formed using DNA inherited from them. All cells in an organism contain copies of the same set of DNA molecules. In terminology, genome refer to all the DNA molecules within a cell which includes chromosomes and the mitochondrial DNA of our cell.

A DNA molecule is a helical form that contains two strands intertwined. Each strand has *base molecules* bonded to one another as a sequence of four nucleotides: Adenine A, Cytosine C, Guanine G and Thymine T. The sequence of one strand can be inferred from the sequence of the other because of the complementary relations of bases $A \leftrightarrow T$ and $C \leftrightarrow G$. Thus, the sequence length of a DNA molecule is typically measured in base pairs (bp). In contrast to a DNA molecule, a Ribo Nucleic Acid (RNA) molecule is single stranded and contains the Uracil base, U, instead of Thymine.

Specific segments of the genome, called genes, encode proteins and Ribonucleic Acids (or RNAs) that carry out cellular functions. Transcription is a biological process by which portions of a gene are translated into an RNA molecule. These RNA

molecules are subsequently released into the cytoplasm of the cell, where they are translated into their corresponding protein molecules.

"Sequencing" is the process of determining the sequence of a DNA molecule. Even though the structure of DNA was established as a double helix in 1953, [11] it was not until 1975 that the first practical procedure to sequence DNA was designed [12]. Subsequent advances in high-throughput cost-effective sequencing technologies have resulted in tremendous growth in genomic databases. A wide range of genomes have been sequenced, from short viral genomes to larger more complicated plant and mammalian genomes. In order to make sense of the rapidly increasing amount of genomic data, new analytic tools and computational methods must be developed.

2.2 Sequencing Technologies

Sequencing is the process of determining the precise order of nucleotides in a DNA/RNA molecule. The first method was "plus and minus" designed in 1975 by Sanger and Coulson [12] to sequence DNA molecules. Two years later, Sanger et al. designed another similar method, called the "chain termination method" [13]. Currently, most sequencing methods are based on this approach. Since the invention of the "chain termination method," a great deal of technological advancements have been made towards increasing the throughput, and towards reducing the cost per base.

Today's sequencing techniques for DNA are capable of sequencing $\sim 100-1000$bp nucleotides with high accuracy (>98%). However, genetic molecules are much longer; – DNA and RNA molecules span a few tens of thousands to tens of millions of nucleotides, and a protein may contain hundreds of amino acids. The current approach to assemble the target molecule is a two phase strategy: (1) sequencing randomly chosen "fragments" from many copies of the molecule, and (2) subsequently relying on computational approaches to assemble the target molecule's sequence.

In the following section, I will briefly review the different sequencing technologies

and the types of sequences that can be derived from the specified methods.

2.2.1 Whole genome shotgun sequencing

One popular way to sequence an entire genome is whole genome shotgun (WGS) sequencing (or shotgun cloning), first used to sequence the genome of a bacteriophage [14]. Since the "chain termination method" of DNA sequencing can only be used for fairly short sequences (100 to 1000bp), the shotgun sequencing method is a large chunk approach which samples random locations of a target genome, and short sequences($\sim$ 5000bp) are then extracted starting at these locations. Next, these sequences are cloned in bacterial vector colonies, (BAC) and are finally sequenced from both sides. The resulting sequences are of length $500-1000$bp and are called shotgun fragments.

In WGS sequencing, multiple overlapping reads for the target genome are obtained by performing several rounds of these processes, such that each target genome base can be expected to be covered by a specified number of fragments. This number is called 'sequencing coverage' and is denoted by 'X'. The number of fragments sequenced in a WGS project is specified by the length of the target genome and the desired sequencing coverage. For example, a 10X coverage of a one billion base pair genome will result in approximately 14 million fragments, assuming an average sequence length of 700bp.

Since the shotgun process is random, it is hard to guarantee that each base will be covered by at least one fragment. In practice, some genome stretches are left uncovered in sequencing, and each uncovered stretch is called a "sequencing gap". There is a trade-off in that specifying a high coverage decreases the frequency of gaps, although this choice will lead to a higher sequencing cost.

Whole genome shotgun sequencing is relatively cheap when compared to other sequencing technologies. This easy and cheap approach has been used in many projects

including the Human Genome Project [15, 16, 17].

2.2.2 Hierarchical sequencing

In this approach, a genome is first broken down into clones of up to 150-200Kbp each called a "Bacterial Artificial Chromosome (BAC)". Next, a combination of these BACs that provide a minimum tiling path based on their locations along the genome is determined. Each selected BAC is then individually sequenced using a shotgun approach generating numerous short $(500-1000\text{bp})$ shotgun fragments. This method is also called clone-by-clone sequencing because of its hierarchical strategy. Even though Hierarchical sequencing is a costlier method than whole genome shotgun sequencing, this method provides additional information that facilitate an accurate analysis of the fragments. Hierarchical methods involve different types of colonies such as Yeast Artificial Chromosomes and Fosmids. This approach has been used for sequencing several complex eukaryotic genomes including that of maize [NSF (2005)] and the human [Consortium (2001)].

2.2.3 Next-generation methods

The high demand for low-cost sequencing has driven the development of high-throughput technologies to parallelize the process, producing thousands of sequences concurrently. In ultra-high-throughput sequencing as many as one million sequencing operations may be run in parallel [18, 19].

In the table 2.1, the NGS technologies are summarised. The specifics of NGS systems such as Solid/Ion Torrent PGM from Life Sciences, HiSeq and MiSeq from Illumina, and GS FLX Titanium from Roche are presented. This table summarises many of the important features of computing approaches to next-generation sequencing. However, this table does not include the initial equipment cost. For example,

the sequencers from Pacific Bio and Illumina are much more expensive than the Ion sequencer [20].

Table 2.1: Comparison of next-generation Sequencing Methods

Method Name	Read Length	Accuracy	Reads/run	Time/run	Cost/Mbp
Single-molecule sequencing Pacific Bio	5500 - 8500bp	99.9%	400Mbp	30-120m	<$1
Ion Torrent sequencing Ion semiconductor	<400bp	98%	80Mbp	120m	$1
Pyrosequencing 454	700bp	99.9%	1Mbp	24h	$10
Sequencing by synthesis Illumina	50-300bp	98%	3Bbp	1-10d	5-15¢
Sequencing by ligation SOLiD sequencing	50+50bp	99.9%	1.4Bbp	1-2w	15¢
Chain termination Sanger sequencing	400-900bp	99.9%	-	3h	>$2000

2.3 Sequence Alignment

All biological sequences can be represented as strings over a finite alphabet. As shown in equation 2.1 the alphabet size is 4 for DNA or RNA sequences, and 20 characters is enough for proteins. The relationship between two sequences is typically established by comparing the two sequences and detecting any potential overlap between them.

Sequences typically represent much smaller pieces of the original source sequence. As such the following questions need to be answered.

1. How can we construct the whole sequence from many small strings?

2. How can we cluster the sequences in a set?

The presence of overlap can be used as an evidence to find similarities between two sequences or to link two sequences without prior knowledge.

For the remainder of the study, I will use the terms sequence and string interchangeably. Also throughout the remainder of this study, I will use the term subsequence to mean a substring.

$$\begin{aligned} \sum\nolimits_{DNA} &= \{A, C, G, T\} \\ \sum\nolimits_{RNA} &= \{A, C, G, U\} \\ \sum\nolimits_{Pro} &= \{A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, V\} \end{aligned} \tag{2.1}$$

A suitable alignment method will help us to detect an overlap between two sequences. An alignment between two strings is an ordered list of matches, mismatches, insertions, and deletions. A gap in an alignment stands for one or more insertions (alternatively, deletions). An alignment score is computed from the number of its matches, mismatches and gaps. An optimal alignment is one with the optimum score.

2.3.1 Alignments Types

There are several types of alignments that can be computed between two strings for different purposes. Given two strings, s_1 and s_2, of lengths l_1 and l_2 (where $l_1 > 0$ and $l_2 > 0$) respectively:

Using dynamic programming, computing an optimal alignment takes $O(l_1 \times l_2)$ time, and $O(l_1 + l_2)$ space [21].

Alignments are typically computed using a $(l_1 + 1) \times (l_2 + 1)$ table. Computing a global alignment between a pair of strings of similar lengths and in which you expect

high sequence similarity can be accelerated using a banded computation technique [22]. In this technique, the alignment computation starts on the diagonal of the dynamic programming table and progressively expands either side in a band until it can be guaranteed that no optimal alignment can lie outside of the band. The main idea is to avoid computing the entire table, although it may be necessary in the worst case. This banded technique can also be extended for non-global alignments if individual pairs of local regions that are potentially aligning can be identified through other, quicker means.

For the above alignments, alignment scoring could vary depending on the mechanism used to penalise gaps. A straightforward method is to penalise gaps proportional to their lengths. Another popular gap function is the affine gap penalty function [23], in which gaps exceeding a cutoff length are given a constant penalty. Affine gap penalty functions are generally preferred because they provide a better model for biological events such as mutations and polymorphisms.

Besides alignment scoring, there are several other ways to measure pairwise sequence similarity [24, 25]. While computing these measures may not accurately model the problem for sequence errors and expected patterns in overlaps, these techniques are usually sought as faster alternatives to alignment-based methods. For a survey of alignment and other sequence similarity measures and methods, see [26, 27].

2.3.1.1 Global Alignments

Global alignments, which attempt to align every base in two sequences, are useful when the sequences are similar and of roughly equal size. A widely accepted and general global alignment technique is the Needleman-Wunsch algorithm, which is based on dynamic programming [28].

2.3.1.2 local Alignments

Local alignments are suitable when s_1 and s_2 dissimilar sequences are suspected to contain regions of similarity or similar sequence motifs. The Smith-Waterman algorithm is a widely accepted local alignment method also based on dynamic programming [29].

2.3.1.3 Hybrid Alignment Methods

Hybrid methods (or semi-global methods) attempt to find the best possible alignment that includes the prefix and suffix of one or the other sequence [30]. This method is a choice when neither global nor local alignment is entirely appropriate because a global alignment would attempt to force the alignment to extend beyond the region of overlap, while a local alignment might not fully cover the region of overlap [30]. Another useful case for semi-global alignment is when one sequence is short and the other is very long ($l_1 \gg l_2$ or $l_2 \gg l_1$).

2.4 High-performance Computing

Energy consumption is a major problem for integrated circuit designers. Not only is it difficult to provide energy to a chip, the power-driven heat can cause major malfunctions. Scaling for chips reached to maximum density, is ultimately limited by the system capability to cool down the circuit. Consequently, the semiconductor industry has settled on two trajectories for designing microprocessors. The many-core approach pays more attention to the execution throughput of parallel applications. On the other hand, the multi-core idea maintains the execution speed of sequential programs while using multiple cores. The many-core architecture is split into a large number of smaller cores. As an example, in NVIDIA GPUs, each core is an in-order, heavily multi-threaded, single-instruction issue processor that shares cache with other

cores.

CUDA, which stands for Compute Unified Device Architecture, is NVIDIA's GPU programming environment. The CUDA programming model consists of both host and device functions. The kernel function which is specific device function and runs on the GPUs in order to accelerate highly parallel and computationally intensive procedures.

In modern software applications, most of the program segments often includes a rich amount of data parallelism, a property which allows many arithmetic operations to be safely performed on program data structures in a simultaneous manner.

Because current GPUs are built on the *single-instruction multiple-data (SIMD)* model [31, 32], each SIMD lane can execute its own logical thread for independent branching and load/store instructions. This native support for diverging scalar threads allows memory accesses to exhibit fine-grained characteristics, as memory addresses are determined at a per-thread granularity.

2.4.1 GPU architecture

2.4.1.1 Streaming Multiprocessor

NVIDIA's new *streaming multiprocessor (SMX)* introduces several architectural innovations. One SMX has 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units. The SMX count may vary between 7 and 15 for different chipsets.

Hyper-Q enables multiple CPU cores to launch work on a single GPU simultaneously, thereby dramatically increasing the % of temporal occupancy on the GPU. Hyper-Q increases the total number of connections between the host and the GPU by allowing 32 simultaneous processes.

GPUDirect is a capability that enables GPUs within a single computer, or GPUs in different nodes located across a network, to directly exchange data without needing to use system memory. The RDMA feature in GPUDirect allows third party devices

to directly access memory on multiple GPUs within the same system, significantly decreasing the latency of MPI send and receive messages to/from GPU memory. It also reduces demands on system memory bandwidth and frees the GPU DMA engines for use by other CUDA tasks.

2.4.1.2 GPU memory model

The memory model of new generation GPU models is slightly different from the older versions. It has extra cache memory which is dedicated to read-only data.

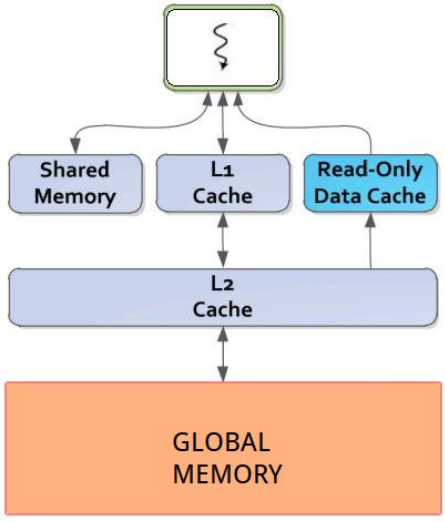


Figure 2.1: Memory hiearachy of a GPU thread

2.4.1.3 CUDA and MPI programming models

Compute Unified Device Architecture, (CUDA) is a parallel programming language extending general programming languages, such as C, C++ and Fortran. CUDA enables users to write parallel scalable programs for CUDA-enabled processors [33]. A CUDA program is includes two parts: a host program running one or more sequential threads on a host CPU, and one or more parallel kernels able to execute on Tesla, Fermi, and Kepler unified graphics and computing architectures [34, 35, 36].

A kernel is a device function launched on a set of lightweight concurrent threads. The parallel threads are organized into a grid of thread blocks, where all threads in a thread block can synchronize through barriers and communicate via a high-speed shared memory. This hierarchical organization of threads enables thread blocks to implement coarse-grained task and data parallelism, and lightweight threads provide fine-grained thread-level parallelism. Threads from different thread blocks in the same grid are able to cooperate through atomic operations on global memory shared by all threads.

MPI is a de facto standard for developing portable parallel applications using the message passing mechanism. MPI works on both shared and distributed memory machines, offering a highly portable solution to parallel programming on a variety of machines and hardware topologies. In MPI, each node is defined as a process and enables the processes to execute different programs. This multiple program, multiple data model offers more flexibility for data-shared or distributed parallel program design. Within a computation, processes communicate by calling runtime library routines, specified for the C/C++ and Fortran programming languages, including peer-to-peer and global communication routines. Peer-to-peer communication is used to send and receive messages between two specific nodes, suitable for unstructured communications. Global communication is used to perform commonly used operations e.g. reduction and broadcast operations.

Chapter 3

Filter Design

3.1 Motivation

Next generation sequencing (NGS) produces huge amount of data. At the same time pairwise techniques for processing sequences are computationally expensive. One way of addressing these two problems is to first filter the data by splitting it into several clusters on the bases of sequence similarity. The filter produces cluster of sequences as can be seen from the Figure 3.1.

Substring search methods are well suited to finding similarities of sequences [37, 38, 51, 52, 53, 54, 55]. This chapter includes descriptions of two filter approaches. The first idea is not efficient because it does not have a parallel data structure and the method requires many data transactions during the execution. For these reasons, I propose to implement the second approach and compare the timing results with that of the filter portion of PaCE [37, 38].

PaCE is chosen for comparison because its data structure is closely related to my approach. PaCE is an open source tool and only PaCE outputs filtering results comparable to those of the method I implemented. In addition, PaCE was implemented for a parallel environment and tested thoroughly [37, 38].

Prior to discuss of the filter structure, we should review the appropriate data structures for the module. The suitable data structures are: the suffix tree and suffix array.

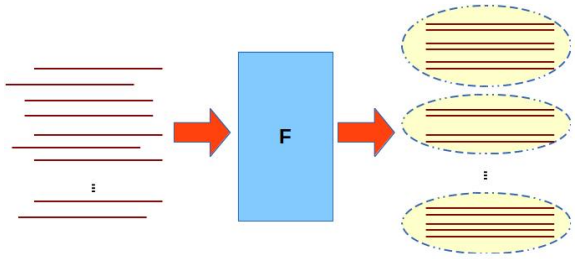


Figure 3.1: Filter module takes raw meta-genomic sequences and create coarse groupings.

3.2 Implementation

Suffix trees and suffix arrays are versatile data structures fundamental to string processing applications. The following subsections describe the suffix tree and suffix array data structures as well as their construction and use.

3.2.1 Suffix tree

Let r denote a string over the alphabet Σ. Let \$ be a unique termination character – the lexicographically smallest character–, and s be the string resulting from appending \$ to r where $s = r\$$. Let $suff_i = s_i s_{i+1} \dots s_{|s|}$ be the suffix of the new string starting at ith position. The suffix tree of s is a compacted trie of all suffixes. Let $n = |s|$. The suffix tree of s has the following properties: [39].

- The suffix tree has n leaves, and each leaf corresponds to a suffix of s.
- Each internal node has at least 2 children.
- Each edge in the tree is labeled with a substring of s.
- The concatenation of edge labels from the root to the leaf labeled i is $suff_i$.

The two paths from the root to the leaves i and j corresponding to two different suffixes $suff_i$ and $suff_j$ and share up to their longest common prefix (LCP), at which character they bifurcate. By using the unique \$ symbol, we can create a leaf node if

a suffix of the string is a prefix of another longer suffix, by using $ symbol, we can assign a new leaf node for the shorter suffix. The suffix tree of the string "BANANA" is shown in Figure 3.2.

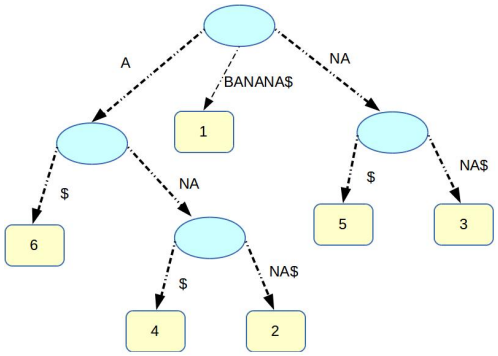


Figure 3.2: The suffix tree for the string "BANANA" (The first leaf node which represents $ character is removed).

3.2.2 Suffix trees and suffix array relation

Manber and Myers proposed the suffix array as an alternative to suffix trees and explained the first algorithm for constructing it in 1990 [40, 41]. They also provided an algorithm to compute an auxiliary data structure, the longest common prefix (LCP) array, alongside the suffix array in $O(nlog(n))$ time. the LCP array stores the lengths of the longest common prefixes between pairs of consecutive suffixes in the sorted suffix array.

Suffix arrays and suffix trees are closely related data structures. Each one can easily be converted to the other. A suffix array can be derived from a suffix tree by performing a depth-first traversal on the tree. A suffix tree can be constructed in linear time by using a combination of suffix and LCP array [42, 43].

It has been shown that every suffix tree algorithm can be systematically replaced with an algorithm that uses a suffix array enhanced with additional information (such as the LCP array) and solves the same problem in the same time complexity.

Advantages of suffix arrays over suffix trees include improved space requirements, and easier linear time construction algorithms [40, 42].

3.2.3 Suffix array construction methods

The suffix array is as an alternative data structure to suffix trees which is more suitable for GPU computing. Since it was announced, suffix array based applications and suffix array construction algorithms (SACAs) have proliferated. This chapter provides summaries that highlight the features of these algorithms, while avoiding as much as possible going into exhaustive detail. I provide comparisons of the algorithms' worst-case time complexity and space complexity.

After the Manber and Myers announcement, there has been a great deal of research on the construction and use of suffix arrays. Over this period, it has been shown that practical space-efficient SACAs exist that require worst-case time linear in string length [44, 46].

It has also been proven that suffix arrays and suffix trees have same asymptotic complexity [42]. Thus, suffix arrays have become the data structure of choice for many string processing applications for which suffix tree data structure is applicable.

In this section, I do not attempt to cover the entire suffix array literature. The goal is to provide a comprehensive overview of SACAs, organized into a "taxonomy" based primarily on the methodology used and their complexity [47].

3.2.4 Suffix arrays basics

Consider a finite nonempty string s of length $n = |s| \geq 1$, defined on an alphabet Σ. The suffix array A of s is defined to be an array of integers providing the starting positions of suffixes of S in lexicographical order. This means, an entry A_i contains the starting position of the i-th smallest suffix in the string and thus

$\forall i,\ 1 < i \leq n,\ s[A_{i-1}, n] < s[A_i, n]$

Table 3.1: The suffixes of the string, "banana", and its suffix array and LCP array

i	Suffix	Suffix	A[i]	LCP[i]
0	banana$	$	6	
1	anana$	a$	5	0
2	nana$	ana$	3	1
3	ana$	anana$	1	3
4	na$	banana$	0	0
5	a$	na$	4	0
6	$	nana$	2	2

Consider the text $s = 'banana\$'$ to be indexed:

i	0	1	2	3	4	5	6
s	b	a	n	a	n	a	$

The text ends with the special letter $ that is unique and lexicographically smaller than any other character.

Suffix arrays usually need an auxiliary data structure called the *longest common prefix array (LCP array)*. The LCP array stores the lengths of the longest common prefixes between pairs of consecutive entries in the suffix array. Combining the suffix array with the LCP array supports an efficient simulation of the suffix tree, [41, 42] and speeds up pattern matching on the suffix array [43]. After computing the suffix array the LCP array is constructed by comparing lexicographically consecutive suffixes to determine their longest common prefix:

The rule for constructing an LCP array is,

$\forall j,\ 1 < j \leq n,$

lcp_j is just the length of the longest common prefix of suffixes A_{j-1} and A_j.

The string *banana* has the following suffixes and sorted suffixes as shown in Table 3.1:

Table 3.2: The Skew Algorithm on the string, "banana": (a,b) sorting the subarrays, and (c) merging

i	Suffix
5	a$
1	anana$
4	na$
2	nana$

(a)

+

i	Suffix
6	$
3	ana$
0	banana$

(b)

⇒

i	Suffix
6	$
5	a$
3	ana$
1	anana$
0	banana$
4	na$
2	nana$

(c)

3.2.5 The skew algorithm

The *skew algorithm* is much simpler than previous linear time algorithms [46]. The algorithm also works for the case of an integer alphabet. Let s be a string of length n over a fixed alphabet Σ. For convenience, assume n is a multiple of three and last two symbols are empty – $s_{n+1} = s_{n+2} = 0$. The main idea of this algorithm is to divide suffixes into 3 groups. Table 3.2 depicts the steps of the skew algorithm. For suffix array construction over the alphabets that can be implemented to run in linear time using the following sorting subroutine:

1. Recursively sort suffixes beginning at positions:

   $i \bmod 3 = 0$

2. Sort the remaining suffixes using the information obtained from the previous step,

   $i \bmod 3 \neq 0$

3. Merge the sorted sequences which are computed in steps one and two.

$G^{\neq 0}$ = { (1, 'anana$'), (2, 'nana$'), (4, 'na$'), (5, 'a$') }

$G^{=0}$ = { (0, 'banana$'), (3, 'ana$'), (6, '$')}

s	b	a	n	a	n	a	$
Type	L	S	L	S	L	L	S/L
Pos	0	1	2	3	4	5	6

bucket	$	a	a	a	b	n	n
Step-2	6	5	3	1	0	2	4
Sorted Order	6	5	3	1	0	4	2

3.2.6 Ko and Aluru's Algorithm

The algorithm of Ko and Aluru [44, 45] partitions suffixes based on the lexicographic ordering of a suffix with the suffix of its neighbour. Consider a string s of size n over a fixed alphabet Σ. Again, I use \$ to mark the end of s, considered lexicographically the smallest and $\$ \notin \Sigma$.

The symbol $\prec$ denotes lexicographic ordering. The statement $a \prec b$ indicates if the string a is smaller than b. The algorithm starts by classifying suffixes into two types, S and L. The classification is done as follows: a suffix $suff_i$ is in the S class if $suff_i \prec suff_{i+1}$, and is of type L if $suff_{i+1} \prec suff_i$. The very last suffix, $suff_n$, is labelled as S/L. The positions of the type S suffixes partition the string into a set of substrings. I substitute each of these substrings by its rank among all the substrings and produce a new string t. The suffixes of the new string are then recursively sorted. The suffix array of t gives the lexicographic order of all type S suffixes. The order of all other type suffixes can be deduced from this order.

The first step of the algorithm is to classify suffixes into types S and L.

Algorithm 1 Ko and Aluru's Suffix Array construction Algorithm

1: $suff_{n-1} = S/L$
2: **for** $i = n - 2$ downto 0 **do**
3: If $s_i < s_{i+1}, suff_i$ is of type S
4: If $s_i > s_{i+1}, suff_i$ is of type L.
5: If $s_i = s_{i+1}, suff_i$ is of type $suff_{i+1}$.
6: **end for**

Initially, let B be an array containing all suffixes of the string. Let C be a sorted array of all suffixes of type S. Using C, the sorted order of all suffixes of s can be computed as follows:

1. group all suffixes of the string according to their first character in the first array B.

2. The array C is scanned. For each suffix encountered in the scan, move the suffix to the current end of its bucket in array C, and advance the current end by one position to the left. After this step, all type S suffixes are in their correct positions in B.

3. Scan array B. For each entry of B_i, if $suff_{B_i-1}$ is a type L suffix, move it to the current front of its bucket in the current array, and advance the front of the bucket by one. At the end of this step, B is the suffix array of s.

We can characterise the main types of SACAs as follows:

Prefix Doubling

Algorithms are based on the idea of Karp, Miller and Rosenberg (1972). The idea is to find prefixes that mark the ordering of suffixes. The determined prefix length doubles in every iteration of the algorithm until a unique prefix is found and this prefix provides the rank of the corresponding suffix. The time required for prefix doubling SACAs is O(nlog n). There are two algorithms in this class: Manber and Myers [40, 41] and Larsson and Sadakane [48, 49].

Recursive algorithms

Recursive algorithms follow the same idea that used for constructing a suffix tree by Farach. These algorithms recursively sort a subset of suffixes. Later, the sorted subset is then transfered to the suffix array. The overall time requirement

of these algorithms is $\Theta(n)$. There are three main algorithms in this class: Ko and Aluru [44], Kärkkäinen and Sanders [46], and Kim et. al. [57].

Induced Copying

Induced copying methods and recursive algorithms are similar in the sense that they use an already sorted subset to induce a fast sort of the remaining suffixes. The difference is that induce copying methods are non-recursive. In general, these induce copying algorithms are very efficient in practice, but may have worst-case asymptotic complexity as high as $O(n^2 logn)$. [58, 59, 60, 61, 62, 63]. A detailed survey of SACAs has been put together by Puglisi et. al. [64].

Table 3.3: Performance Summary of the SACA Algorithms

Algorithm	Year	Worst Case	Memory
Prefix-Doubling			
Manber and Myers	1993	$O(nlogn)$	8n
Larsson and Sadakane	1999	$O(nlogn)$	8n
Recursive			
Ko and Aluru	2003	$O(n)$	7-10n
Kärkkäinen and Sanders	2003	$O(n)$	10-13n
Kim et al.	2004	$O(nloglogn)$	13-16n
Induced Copying			
Itoh and Tanaka	1999	$O(n^2 logn)$	n
Seward	2000	$O(n^2 logn)$	n
Burkhardt and Kärkkäinen	2003	$O(nlogn)$	5-6n
Manzini and Ferragina	2004	$O(n^2 logn)$	5n
Schürmann and Stoye	2005	$O(n^2)$	9-10n
Baron and Bresler	2005	$O(n\sqrt{logn})$	8n
Maniscalco and Puglisi	2007	$O(n^2 logn)$	5-6n
Nong et. al. [50]	2009	$O(n)$	–

3.2.7 Filter implementation

In the filter module, I implemented a kernel that computes a cumulative suffix array for concatenated string t. The concatenated string t is the concatenation of a set of strings s_i in S. After forming the suffix array SA of t, the inputs are ready for filter

kernel. However, an additional step should be taken here for the kernel to prevent the filter from matching a sequence s_i with its replica in the total sequence t.

The algorithm of the the filter kernel that I implemented and evaluated is given in Algorithm 2.

Algorithm 2 Filter Algorithm 2

1: $Input : S = \{s_0, s_1, \ldots, s_{n-1}\}$
2: $t = \varepsilon$
3: **for** $s = s_0 : s_{n-1}$ **do**
4: $\quad t = t + s$
5: **end for**
6: Compute suffix array of t, SA
7: A = FilterKernel(SA, S)
8: Align (S, A)

I can achieve this by blocking out the corresponding part of the total sequence for each sequence. Algorithm 2 describes the second filter idea. Finally, I need to prevent the finding of matches that overlap consequent sequences. Consider the following situation: Algorithm 2

$$\begin{aligned} s_j &= \ldots TTCCCAT \ldots \\ s_i &= \ldots ACCTTCC. \\ s_{i+1} &= CATTG \ldots \end{aligned} \tag{3.1}$$

The string s_j has the following substring "TTCCCAT" which overlaps both of the strings s_i and s_{i+1} which is an artificial match. In order to eliminate this situation, I will add another unique symbol between the sequences. This can be shown as follows:

$$\begin{aligned} \sum &= \sum\nolimits_{DNA} \cup \{\#\} \\ t &= s_0 + \# + s_1 + \# + \ldots + \# + s_{n-1} \end{aligned} \tag{3.2}$$

3.3 Results

The filtering module generates pairs of sequences (promising pairs) that are sent to the alignment procedure. We compare our filter timing results with the filter portion of parallel clustering tool, PaCE. I chose PaCE, because it is an open-source MPI-based tool which is using suffix tree for filtering purpose. The output format of PaCE and my filter module are similar which will make easier to compare results. Since PaCE is a developed in MPI, it will be a fair comparison between GPU timing with PaCE timing.

Data

A genome set were obtained from NCBI for constructing meta-genome sets. We gathered a total of 25 complete bacteria genomes shown in Table 3.4 and Table 3.5. From these genomes we constructed two data sets, namely, *similar and* dissimilar, reflecting taxonomic relationships. The similar group consists of thirteen Bacillus Genus bacteria genomes. The dissimilar set elements are in the Proteobacteria Phylum.

Table 3.4: Similar data set : genomes are from – **Bacillus Genus**

No.	**Name of the sequence**	**Size**
1	Bacillus anthracis str. A0248	5227419
2	Bacillus atrophaeus UCMB-5137 chromosome	4116019
3	Bacillus bombysepticus str. Wang	5295783
4	Bacillus cellulosilyticus DSM 2522 chromosome	4681672
5	Bacillus cereus B4264 chromosome	5419036
6	Bacillus clausii KSM-K16	4303871
7	Bacillus coagulans 36D1 chromosome	3552226
8	Bacillus halodurans C-125 chromosome	4202352
9	Bacillus licheniformis ATCC 14580 chromosome	4222597
10	Bacillus megaterium DSM 319 chromosome	5097447
11	Bacillus pseudomycoides DSM 12442 chromosome	5782514
12	Bacillus subtilis subsp. subtilis str. 168 chromosome	4215606
13	Bacillus weihenstephanensis KBAB4 chromosome	5262775
	Total base-pairs	**61379317**

The purpose is to start based on a common truth. MetaSim is simulating NGS and printing error positions that we will use later in the Error correction module.

Table 3.5: Dissimilar data set : genomes are from –**Proteobacteria Phylum**

No.	**Name of the sequence**	**Size**
1	Achromobacter xylosoxidans NH44784-1996	6916670
2	Acidobacteria bacterium KBS 146	4996384
3	Aeromonas hydrophila ML09-119	5024500
4	Anaeromyxobacter sp. Fw109-5 chromosome	5277990
5	Azoarcus sp. BH72 chromosome	4376040
6	Geminicoccus roseus DSM 18922	5421495
7	Herbaspirillum seropedicae SmR1	5513887
8	Laribacter hongkongensis HLHK9	3169329
9	Marinobacter adhaerens HP15 chromosome	4421911
10	Pandoraea pnomenusa 3kgm	5429298
11	Rhizobium sp. LPU83	4195305
12	Rhodospirillum centenum SW	4355543
	Total base-pairs	**59098352**

NGS DATA MODELING

To simulate the sequencing process, I used MetaSim tool. MetaSim is a sequencing simulator [65]. Based on a database of given genomes, MetaSim allows the user to design a meta-genome by specifying the number of genomes present at different levels, and then to collect reads from the meta-genome using a simulation of a number of different NGS technologies. The MetaSim sequencing simulator is used to generate collections of synthetic reads of specified meta-genome data sets.

Coverage (read depth) is the average number of reads representing a given nucleotide in the reconstructed sequence.

$$Coverage = N \times \frac{l}{\sum_i G_i} \tag{3.3}$$

Equation 3.3 depicts the coverage computation. In the equation, G_i and N represent the length of the ith genome and the number of reads respectively; the average read length is shown with (l).

If we want to compute 6X read depth for the Bacillus Genus data set, then we derive N from the equation as shown by the calculation in (3.4).

$$
\begin{aligned}
Coverage &= N \times \frac{l}{\sum_i G_i} \\
6 &= N \times \frac{750}{61379317} \\
N &\sim 500K
\end{aligned}
\tag{3.4}
$$

Considering a real sequencing process, we wanted to test different coverages. For each data set, we obtained two files, 512K sequences and 1024K sequences which correspond approximately to 6X coverage and 12X coverage, respectively. Altogether, we produced four test files, S-512K, S-1024K, D-512K and D-1024K. We set average sequence length ∼750 base pairs for all data sets.

In addition to the artificial data files, we also obtained a real meta-genome. The meta-genome file is under the sample name Obese Human Gut which has the sample code SRS009825 [66]. It has 2038516 reads and the average read length is 270. From this meta-genome we created 3 more test files which have 256K, 512K and 1024K, respectively.

Table 3.6: Test files, average sequence length is represented by l_{avg}

File Name	**Size**	**Coverage**	l_{avg}
O-256K	256000		270
O-512K	512000		270
O-1024K	1024000		270
O-2048K	2038516		270
S-512K	512000	6X	750
D-512K	512000	6X	750
S-1024K	1024000	12X	750
D-1024K	1024000	12X	750

Table 3.6 shows the benchmarks for testing filter module. The performance of PaCE is summarized in Table 3.8. PaCE can only extract pairs of sequences where match length is 12. PaCE encounters a memory deficiency when running larger data files. Thus, on the clusters of 40 and 60 CPUs, PaCE cannot handle D-1024K, S-1024K, O-2048K files.

PaCE does not have 1492.266

Table 3.7: Comparison of Stampede computing node with GPU used in our tests.

	Intel Xeon E5-2680	**NVIDIA K20 GPU**
Architecture	Sandy Bridge	Kepler GK110
Processor cores	8	13
Threads /core	2 threads/core, Superscalar speculative out of order	2048 threads/core, 8 instructions dispatched per cycle per core in program order
Clock rate	2.7 GHz	745 MHz
Memory bandwidth	51.2 GB/s	208 GB/s
Transistors	2.27 billion	7.1 billion
On chip memory	20 MB L3 cache	∼1 MB (64KB L1/multiprocessor core)
Thermal Dynamic Power	130 Watts	225 Watts

Table 3.8: PaCE timing results on the benchmarks on different clusters 40, 60, 80, 100 CPUs (fixed match length – 12)

File Name	**40**	**60**	**80**	**100**
O-256	251.194	227.053		
S-512	635.120	597.231	425.129	381.185
D-512	627.483	593.110	411.634	375.124
O-512	515.789	480.159	382.986	288.840
S-1024			962.193	790.931
D-1024			937.878	715.689
O-1024	1195.919	962.683	707.607	638.061
O-2048		1492.266	1254.101	1128.76

The filter module is tested thoroughly on three different size of clusters. The first one is the base cluster and it includes only one NVIDIA K20 GPU. The filter timing results which is running on one NVIDIA K20 GPU is given Table 4.1. The filter module requires more time once the minimum match length is decreased, the filter extracts more pairs of

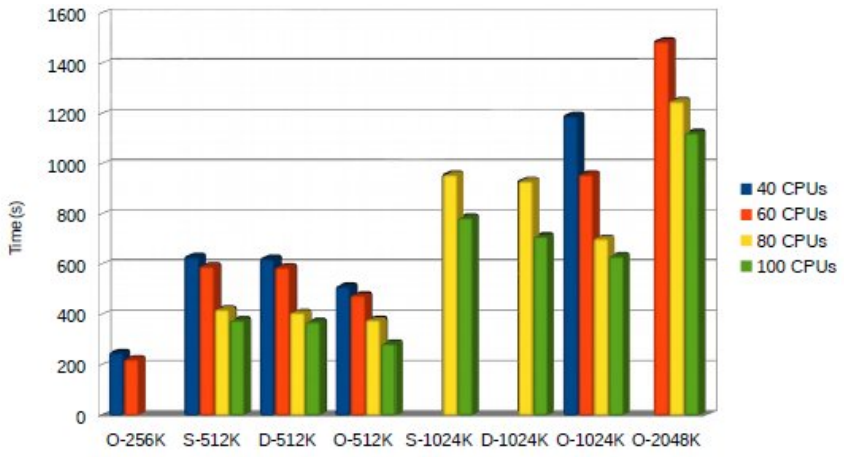


Figure 3.3: PaCE timing results on the benchmarks on different clusters 40, 60, 80, 100 CPUs

strings.

Table 3.9: 1 NVIDIA K20 GPU timing results for running filter kernel

File Name	**40**	**30**	**20**	**15**	**12**
O-256	72.302	73.008	77.115	78.179	82.425
S-512	244.086	245.733	247.115	251.594	255.962
D-512	241.708	243.206	246.760	247.745	250.601
O-512	145.686	151.749	152.332	153.249	158.653
S-1024	498.332	506.403	514.875	521.220	
D-1024	480.538	488.640	497.512	512.953	
O-1024	289.987	298.071	305.200	311.848	

The longer sequences that a test file has, the more time the filter kernel requires. For example when match length is set to 15, the filter procedure requires 521.220s for the test file S-1024 which has an average sequence length 750. On the other hand, the filter completes the process in 311.848s for the file O-1024 consists of sequences of average length of 270bp.

The second test combination uses a cluster of size 10 NVIDIA K20 GPUs. The results shown in Table 3.10 are total time for the kernel. The same behavior can be observed here as we saw in table 4.1 The average performance ratio of 10-GPU cluster/1-GPU cluster is over 7. Figure 3.5 represents the performance of the 10-GPU cluster.

Finally, the third test run uses a 20-GPU cluster where each GPU has the same specifi-

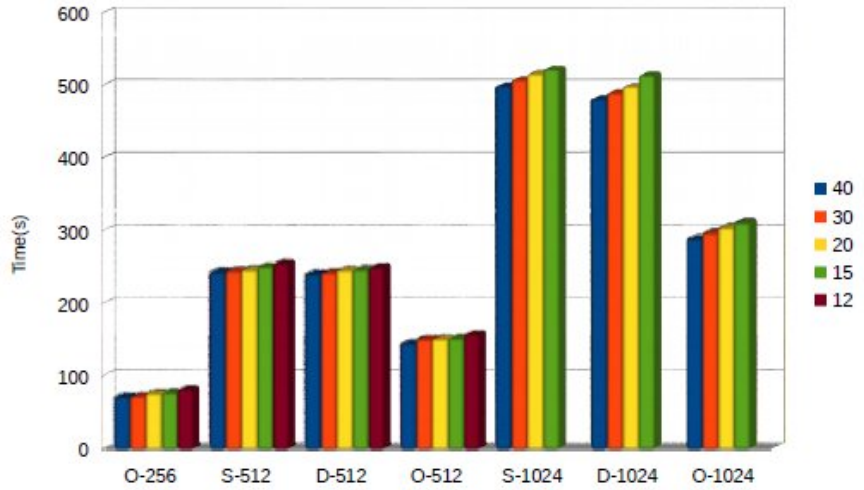


Figure 3.4: Filter module timing: 1 NVIDIA K20 GPU timing results for running filter kernel

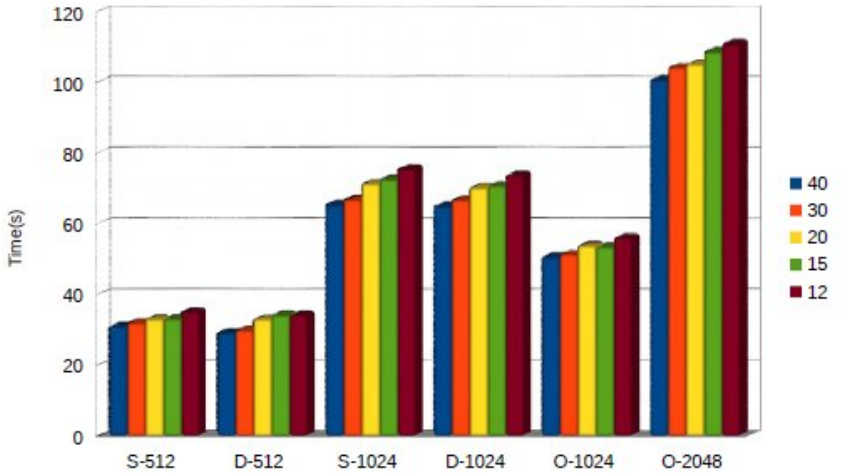


Figure 3.5: 10 NVIDIA K20 GPUs timing results for running filter kernel

Table 3.10: 10 NVIDIA K20 GPUs timing results for running filter kernel

File Name	**40**	**30**	**20**	**15**	**12**
S-512	31.098	32.128	33.115	33.182	35.224
D-512	29.319	30.081	33.129	34.305	34.339
S-1024	65.710	66.987	71.440	72.766	75.764
D-1024	65.267	66.879	70.426	70.961	73.812
O-1024	50.865	51.543	53.909	53.618	56.229
O-2048	99.015	102.426	104.367	108.944	109.159

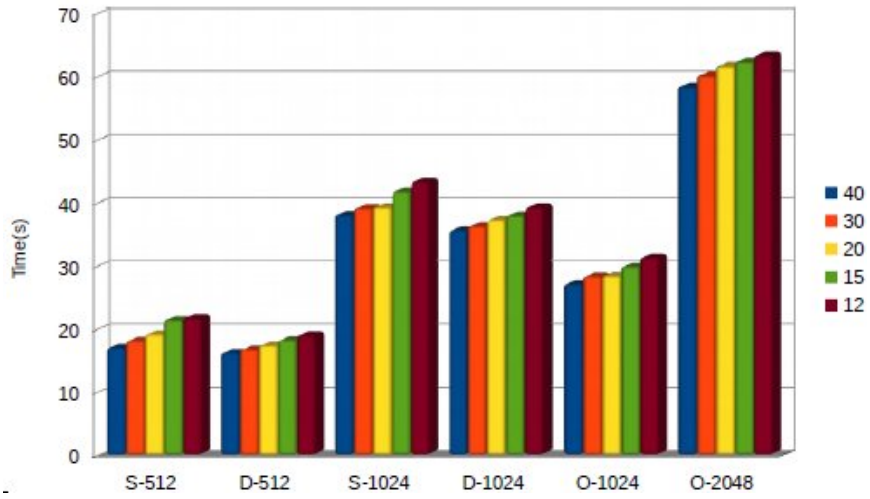


Figure 3.6: 20-GPU cluster timing results

cations. Table 3.11 shows total time for the kernel, as well. The average performance ratio of 20-GPU cluster/1-GPU cluster is about 13.6. Figure 3.6 represents the performance of the 20-GPU cluster.

Table 3.11: The filter kernel run-time results on 20 NVIDIA K20 GPUs

File Name	**40**	**30**	**20**	**15**	**12**
S-512	17.102	18.171	19.245	21.484	21.845
D-512	16.319	16.812	17.529	18.308	19.110
S-1024	38.221	39.242	39.404	41.869	43.443
D-1024	35.716	36.421	37.347	38.110	39.412
O-1024	27.200	28.373	28.442	29.941	31.356
O-2048	56.498	60.256	61.719	61.429	63.423

Chapter 4

Alignment Module

4.1 Motivation

The filtering module in the previous chapter only matches the sequences that are similar based on a threshold value. In order to organise the coarse groups produced by the filter module into clusters additional processing is required. The alignment module will obtain the list of promising pairs as its input and run a global alignment procedure to produce the final clusters from the collection of sequences produced by the filter module.

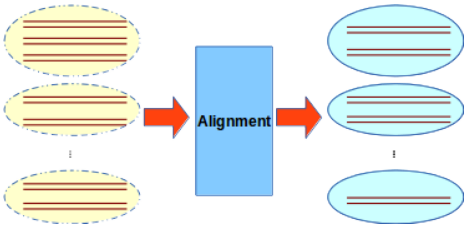


Figure 4.1: Alignment Module

4.2 Background

There has been recent interest in processing and clustering sequences generated by NGS sequencing tools [67, 68, 69, 70, 71, 72, 73, 74].

The kernel of our alignment tool uses the Needleman-Wunsch algorithm to compute the pairwise distances among a large set of short sequences [28]. There are several examples in the literature that describe GPU accelerated local and global alignment algorithms such as Needleman-Wunsch [75, 76], Smith-Waterman [77], and BLAST [78]. However, the

emphasis of these efforts is on local sequence alignment for genomic database searching, in which a relatively short sequences are aligned against a very long database sequence.

Manavski provided the work in accelerating Smith-Waterman using CUDA [79]. This early work has been improved upon by the development of libraries such as CUDASW++ [77]. More recently, Razmyslovich has developed an OpenCL implementation of Smith-Waterman [80] that can achieve three times the performance of CUDASW++ 2.0 in some circumstances [81].

4.2.1 Needleman-Wunsch global alignment

The Needleman-Wunsch algorithm is a comparison operation between two sequences A and B given an implicit assumption that when the sequences are not exactly equal, their similarity can be characterised as the number of edit operations that would transform one sequence into the other. Possible edit operations are *character substitutions*, *substring insertions*, and *deletions*. The objective of an alignment is to align the matching or substituted characters that are common in both sequences and to add blank spaces-or gaps- to one of the sequences when the characters do not align. The distance "penalty" that is contributed by each edit operation can be specified using a substitution table T and a gap penalty d.

The algorithm works by constructing two matrices, where each matrix has $l_1 + 1$ rows and l_2+1 columns, where l_1 and l_2 are the lengths of the two strings to be aligned. The score matrix records the alignment score for every possible alignment between the two strings, while the movement matrix provides a path through the matrix, from the bottom-right cell to the upper-left cell, that represents the alignment configuration that yields the minimal alignment score. In this path, a move to the left (or up) represents a gap that is inserted into the first (or second) sequence, while a diagonal move represents a matching.

Each cell of the score and movement matrix is computed as shown in Equations 4.1 and 4.2.

$$S_{i,j} = min \begin{cases} S_{i-1,j-1} & + & T_{a,b} \\ S_{i-1,j} & + & d \\ S_{i,j-1} & + & d \end{cases} \tag{4.1}$$

$$M_{i,j} = \begin{cases} Diagonal & if\ S_{i,j} & = & S_{i-1,j-1} + T_{a,b} \\ Up & if\ S_{i,j} & = & S_{i-1,j} + d \\ Left & if\ S_{i,j} & = & S_{i,j-1} + d \end{cases} \tag{4.2}$$

In these equations, $S_{i,j}$ is the score matrix, $T_{a,b}$ is the substitution penalty resulting from comparing element $a = A_i$ and element $b = B_j$, and d is the gap penalty. T and d are specific to the sequencer technology and are represented as floating-point values. In order to differentiate minor variations between flows, some authors choose to use double precision floating-point to perform the comparisons, score accumulation, and score normalisation [75, 76].

Figure 4.2 shows an example of global alignment. In this example, sequence 1 undergoes three edit operation to produce sequence 2. The final alignment score is taken from the lower-right cell of the resultant score matrix. The move matrix is depicted in the figure and shows how characters present in sequence 1 but not in sequence 2 produce moves to the left, characters present in sequence 2 but not in sequence 1 produce moves up, and characters that match or are substituted produce a move diagonally. In this example, for computing the normalised score the final score is divided by the alignment length of 13.

4.2.2 Space optimization

The construction of the two matrices represents a major challenge when performing large-scale batch alignments on GPUs, as the memory requirement will often become a constraint well before the execution time. Since the actual alignment is not needed, only the last row of the score matrix and move matrix need to be stored in memory. However, since our alignment procedure finds the total movement distance in order to compute normalised score, it originally scored the entire movement matrix [75].

In our improved kernel, we store only one row of the movement matrix as well. In order to avoid storing the entire movement matrix, the kernel maintains only a single vector V, where V_j represents the accumulated number of minimal alignment moves beginning from the current row and from column j. In addition to this vector, we establish two extra registers, N_m and L_m, to hold intermediate values. N_m holds the newly computed number of moves and L_m holds the previous number of moves

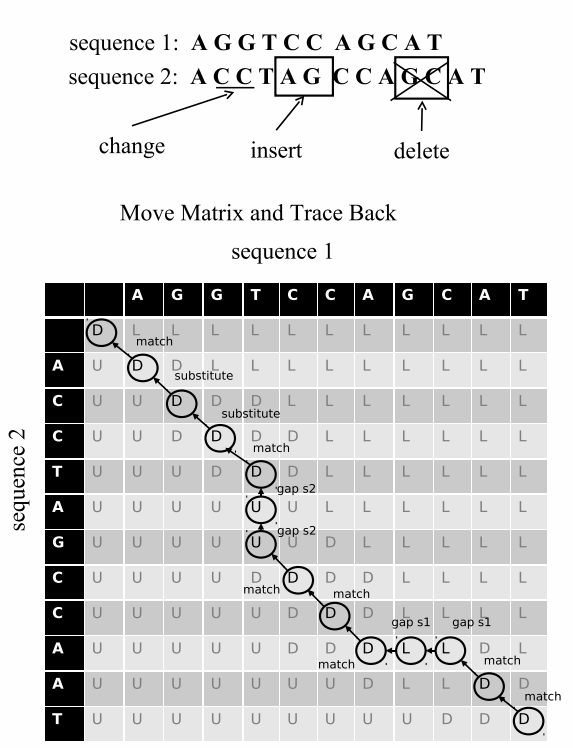


Figure 4.2: Example Needleman-Wunsch alignment between two sequences.

from the left cell. If the current move is determined to be diagonal, then we set $N_m = V_{i-1} + 1$, if the current move is determined to be left, then we set $N = L + 1$, and if the current move is determined to be up, and we set $N_m = V_i + 1$. After this we set $V_{i-1} = L_m$, $L_m = N_m$, and increment j. This processes is described in more detail below in Algorithm 3.

Algorithm 3 Single Vector Needleman-Wunsch Alignment

```
1: Input : A, B, T, d
2: Output : normalised score value
3: for i = 1 : |A| do
4:     L_m = 0
5:     L_S = i × d
6:     for j = 1 : |B| do
7:         N_S = min(S_{j-1} + T_{A[i],B[j]}, L_S + d, S_j + d)
8:         if N_S = S_{j-1} + T_{A[i],B[j]} then
9:             N_m = V_{j-1} + 1
10:        else if N_S = L_S + d then
11:            N_m = L_m + 1
12:        else
13:            N_m = V_j + 1
14:        end if
15:        S_{j-1} = L_S
16:        L_S = N_S
17:        V_{j-1} = L_m
18:        L_m = N_m
19:    end for
20:    S_{j-1} = L_S
21:    V_{j-1} = L_m
22: end for
23: return L_S/L_m
```

4.2.3 Arithmetic Intensity

Algorithm 3 shows the version of the Needleman-Wunsch algorithm used in our kernel. The innermost loop performs all the operations required to calculate a single cell of both the score and the movement matrices. As shown in the algorithm, this requires 9 double precision floating-point operations, the loading of 2 bytes for A_i and B_j,

the access of 24 bytes accessed from the score vector, and 8 bytes accessed from the distance vector. Since all of our test sequences are of length 400, each alignment requires $400^2 = 160,000$ cell updates.

If we begin with the assumption that our kernel is compute bound we consider that each of the thirteen streaming multiprocessor cores (SMXs) on the NVIDIA K20 can dispatch 128 double precision operations per cycle. Consequently, we should be able to achieve a throughput of $1/160,000$ alignment/cells $\times 1/9$ cell/ops $\times (13 \times 128)$ ops/cycle $\times 706e6$ cycles/second $= 815,820$ alignments/second per GPU. On the other hand, if we assume that the kernel is memory bound, we should be able to compute $1/34$ cell/bytes $\times 1/160,000$ alignment/cells $\times 208$ GB/s $= 38,235$ alignments/second per GPU. Since the second throughput is lower we conclude that the kernel is indeed memory bound. We can compute the utilization of this kernel, $38,235/815,820 = 4.7\%$ of the computational capability of the GPU.

In order to compute the pairwise distances among sequences, an input dataset with n sequences will perform a N-W alignment $(n^2 - n)/2$ times to compute. For a dataset of 2^{18} sequences there are approximately 34 billion required alignments, which would ideally require 468 minutes on 32 K20 GPUs.

4.2.4 Multi-GPU implementation

Since each individual alignment is independent, the host can assign each GPU a workload consisting of a subset of the alignments in order to parallelize the pairwise alignments across multiple GPUs. In our multi-GPU implementation, we divide the workload across each GPU using MPI.

4.3 Results

4.3.1 Single GPU performance results

Each node in the TACC Stampede cluster contains dual 2.7 GHz eight-core Intel Xeon E5-2680 CPUs that can each execute 16 MPI processes. Our first set of experimental results seeks to determine how many of these cluster nodes are equivalent, in performance, to a single NVIDIA GTX680 GPU for performing a set of pairwise alignments.

Table 4.1: CPU vs. Single GPU execution time in seconds.

		Execution Time for 8192 Seq.	Execution Time for 6144 Seq.
Cluster processes	32	2479	1394
	64	1241	698
	128	620	698
	256	620	349
	512	310	175
	1024	155	87
GPU	GTX680	1230	700

Table 4.1 shows the performance results for alignment, using only Stampede's CPUs. The 8K and 6K sequences are nodes on two through 64, and on all 16 processors on each node. Our CPU implementation uses the same optimized algorithm described in Algorithm 4.1, which is approximately six times faster than the base implementation in AmpliconNoise due to the movement vector optimization. Note that the speedup is nearly ideal as we scale to larger numbers of processors, except for the case when scaling from 128 to 256 processors for the 8K dataset and from 64 to 128 processors in the 6K dataset. We assume this is due to communication overhead related to the placement of the MPI processes on the cluster.

We also ran the same datasets using a single NVIDIA GTX680 GPU. For both datasets the GPU is equivalent to 64 processors.

4.3.2 Performance analysis

Combining the alignment method and filter kernel, a partial performance equation of my framework can be derived by using Amdahl's Law (see equation 4.3). Let f be the number of pairs of sequences that are filtered in a second through the filter engine and let a be the number of alignments that are produced by the alignment algorithm in a second. The filtering method is removing the sequences that should not be in a cluster. However, some portion of the sequence pairs need to be sent to the alignment kernel for further analysis. q denotes the proportion of the sequences that are sent to alignment algorithm from the filter method. Finally r represents the overhead.

$$P = \frac{1}{\frac{1}{f} + q \times \frac{1}{a} + r} \tag{4.3}$$

Let S be the set of sequences to be clustered, where $n = |S|$ denotes the number of input sequences, and l denote the average length of a sequence.

I can give an ideal picture by eliminating overhead from the equation. If the filter throughput is 5×10^6 sequence-pairs/s and alignment method is capable of producing 4×10^4 alignment/s, and assuming that the filter is conveying 1% of the sequences to the alignment module for getting further detail, the algorithm can cluster 2.2×10^6 sequence/s.

4.3.3 Faster Alignment Module

The alignment module described above is computing in all-to-all fashion. The pipeline cannot process millions of sequences it will take days or weeks to align (recall Figure 1.2 on the page 3). This process becomes a bottleneck in the framework. Two options can be considered: (1) Alignment module is computing double precision which

requires more data to be transferred. Thus the module was limited with memory bandwidth. So, we can use integer or single precision floating point to speed up. (2) The number of alignments can be reduced with a pre-scan process.

The alignment kernel in [76] is modified so that the function is returning an integer score. Table 4.2 depicts the performance of the new alignment kernel. Test files are a

Table 4.2: Multiple alignment run time results of the modified alignment kernel on a 10-GPU cluster

File Name	**Time(s)**
S-20	114.328
D-20	112.882

Since we updated the alignment kernel, we can compute the ideal performance of a GPU. One GPU can achieve ideally 232056 alignments/second (1/(750×750) alignment/cells ×1/9 cell/ops ×(13 × 128)ops/cycle ×706e6 cycles/second). A 10-GPU cluster is able to process 2320560 alignments per second.

There are 20000×20000/2 seq all to all sequence alignment

Ideal time for aligning a test file which has 20K sequences (20000×20000/2) alignment (1/2320560) = 86.186 seconds.

In our tests, 10-GPU cluster finished aligning 20K (S-20) test file in 114.328 seconds (see Table 4.2). Consider that we measured whole kernel time which includes initialization, data transfer and printing the result.

The filter module extracts the sequences that are sharing a substring. This process is decreasing number of alignment significantly.

When we analyze Table 4.3, the required time increases significantly when we decrease minimum match length because there are more sequence pairs to be aligned.

Table 4.3: New filter-align module run time results on 1-GPU cluster

File Name	40	30	20	15	12
S-512	362.590	514.961	821.042	1067.495	1134.608
D-512	311.917	468.566	607.93	962.268	1027.849

4.3.4 Multi-GPU performance results

Stampede has 128 nodes that all contain one NVIDIA K20 GPU. For our multi-GPU experiments, we scaled our GPU kernel up to 10 NVIDIA K20s on Stampede using data sets of 512K sequences. For this test, workload is shared by GPUs and CPUs in the Filter kernel, however Stampede's CPUs remained idle while the GPUs executed the alignment kernel.

Table 4.4: Multi-GPU performance results

File Name	40	30	20	15	12
S-512	46.811	62.593	101.558	122.336	135.473
D-512	39.973	53.024	75.708	112.612	119.910

The synergy of 10 GPUs is depicted in Table 4.4. When we fix the minimum match length to 12, a 10-GPU cluster requires 135.473 second to align the test file, S-512, which is derived from a set of **Bacillus Genus** genomes (please see Table 3.4). On the other hand, the same cluster aligns dissimilar test file, D-512, in two minutes.

G-DNA is a multi-GPU/MPI tool for aligning nucleotide reads [82]. G-DNA requires two files as its input: a) sequence file, b) list of pairs of sequences to be aligned. According to the author, one NVIDIA GeForce GTX 580 GPU reached 89 Giga cell updates per second –GCUPs, which is a speed measure used for alignment tools. In our test, G-DNA achieved 58.7 GCUPs on one K-20 GPU. The kernel time of G-DNA was 3933.560 ms to align 370000 pairs. In Table 4.5, T_{G-DNA}, T_{AK}, and T_t are run time results for G-DNA, our alignment kernel and theoretical time

respectively.

Theoretically, our kernel can finish same alignment process in 370000 / 232056= 1594.440 ms. Thus, the performance ratio of the ideal time over the G-DNA is $T_t/T_{G-DNA} = 2.46$. G-DNA has been outperformed by our kernel [76]. Our alignment kernel is 1.6 times faster than G-DNA.

Table 4.5: Run time comparison of G-DNA with our Alignment kernel only – AK

# of pairs	$\mathbf{T}_{G-DNA}$	$\mathbf{T}_{AK}$	$\mathbf{T}_t$
370000	3933.560	2431.198	1594.440

Chapter 5

Genetic sequence error correction

5.1 Motivation

NGS is an important tool for many areas of molecular biology, however, its output data is noisy and is hard to interpret especially for meta-genomics. Even a low error rate can cause a large number of errors due to the high number of bases being sequenced. Identifying sequencing errors from true biological variants is a challenging task. For organisms without a reference genome, this difficulty is even more challenging. A newer approach in metagenomics follows another strategy which is to identify genes directly from sequences, rather than constructing the whole genome. As mentioned before, the cost and time effort was enormous for the Human Genome Project (HGP) which was launched in 2001. Using today's next-generation sequencing (NGS) techniques, a human-sized genome can be sequenced for the cost of one thousand dollars in a single day [5]. However, the resulting sequences are much shorter and contain more errors. In this chapter, the most common sequencing error types (insertion/deletions and substitutions) are addressed.

5.2 Background

The difficulty of identifying possible sequencing errors is very important, necessitating the development of alternate error correcting methods. The importance of identifying and correcting sequence errors has been highlighted by the recent discussion prompted by the report of the presence of the widespread differences between the human genome

and sequence reads derived from the corresponding RNA [83]. Once these differences were considered due to RNA editing. However, after a thorough analysis of the same data set, it is clear that a huge amount of the differences arose from sequencing errors [84].

Sequencing errors can occur for a variety of reasons. One source of error originates from a phenomena referred to as "crosstalk." Crosstalk occurs when there is an overlap in the signals used in sequencing machines. This overlap can lead to a possible substitution error, confusion of the nucleotide G with nucleotide T, and of A with C [85, 86]. A second cause of errors is referred to as either phasing or dephasing. Since sequencing is done in cycles, an error in a former cycle may propagate to and effect subsequent cycles. For an extensive discussion see the review of Ledergerber et al., which discusses other possible sources of sequencing errors such as signal decay, mixed clusters and boundary effects [87]. Additionally, sequence-specific error patterns have been proposed as an important cause of sequencing errors through dephasing [88].

The issue of sequencing errors is so unavoidable that being able to detect and correct them is essential in many areas of molecular biology, particularly in the case of gene identification. In the study of Dohm et al. , the occurrence of errors and their corresponding rates were investigated by examining Illumina data sets (2.8 million sequences, each 27 base long) taken from *Beta vulgaris* and *Helicobacter acinonychis*. By aligning reads to the known genomes of these bacteria, error rates were derived for each of the 12 possible nucleotide substitutions.

There has been considerable research resulting in many methods and tools in recent years. Early error correction algorithms were based on the spectral alignment problem (SAP) in [89] and [90].

Another error correcting method based on an algorithm for correcting sequencing errors uses a "generalized suffix trie" [91]. However, this method requires a reference genome and assumes that the error distribution is uniform. A similar method

using suffix arrays is that of Ilie et al. [92]. In an alternative method based on a "position-dependent error model," error probabilities are estimated for each nucleotide substitution type [93]. Another approach, that does not rely on a reference genome, was adopted by Qu et. al. [94]. Short reads are clustered into trees where the most abundant sequence is taken to be the root of a tree, and children, that differ by n nucleotide substitutions, are placed at the nth level. These children are classified either as sequencing errors or biological variants. This approach uses the Illumina quality scores, which are adjusted by means of actual error rates determined by BAC sequencing data used as a control [95].

Schreiber et. al. propose a probabilistic model for predicting the occurrence of sequencing errors in short RNA reads. This method does not require a reference genome or quality scores [96]. Instead, it is based on the observed frequencies of the sequence variants. A graph is constructed where reads are connected if they differ by a single nucleotide substitution.

Another error correction algorithm based on the SAP, called SHREC [97] has been proposed using a generalized suffix trie data structure. The extended version of SHREC, Hybrid SHREC is able to correct a mixed set of reads produced from different sequencers [91]. Due to the large size of NGS datasets, error correction is both a time and memory consuming process.

GPU computing architectures have evolved rapidly and have already demonstrated the ability to reduce the execution time of a wide range of demanding bioinformatics applications such as multiple sequence alignment [76, 77, 81], and motif finding [98]. As a first step, Shi et al. [99] implemented CUDA-EC, a parallel error correction algorithm, using NVIDIA's compute unified device architecture (CUDA). This algorithm is based on the SAP approach [100], where a Bloom filter data structure [101] is used to gain memory space efficiency. This algorithm has been further optimized by incorporating quality scores and a filtration approach in the work of Shi

et al. [102].

DecGPU is the first parallel and distributed error correction algorithm for large-scale NGS datasets using a hybrid combination of CUDA and message passing interface (MPI) [103] parallel programming models. DecGPU provides two versions: a CPU-based version and a GPU-based version. Compared to the hSHREC algorithm, DecGPU shows superior error correction quality for both simulated and real datasets [104].

In a meta-genome analysis, an efficient error correction module is required due to enormous size of data. The performance of hSHREC decreases while data size increases which makes hSHREC a weak candidate. DecGPU is fast but its sensitivity is low when the dataset has long sequences

There are other GPU based sequence error correction methods. Unfortunately, most of them are focused on repairing short sequences. For example CUDA-EC is tested with data sets which consists of 35-70bp sequences.

5.3 Method

We propose a parallel error correction algorithm based on the suffix array that works in two phases as follows:

In the first stage, every sequence in the sequence set is compared with others in order to explore the error types and positions. Every match, at a greater than a threshold minimum, are extracted using suffix array. Once a match is determined between two sequences, a scanning process scans backward and forward to extract bases that do not match.

Obviously, error correction requires more than pairwise string comparisons. The second phase of this procedure is the gathering and deciding process. All the update information for candidate nucleotide errors on a specific string is gathered first. Then a decision algorithm grades these scores and decide if an update necessary.

We use a coding scheme that covers all error types. The second phase of the module uses the coding scheme to make repairs. An extra symbol, x, is used to specify indel errors. Table 5.1 depicts the code that prposed error correction module use. From the table, if there is a proposed substitution error from A to C, e_{AC} is the corresponding label. For a deletion error we use e_{Ax}.

$\sum_{fix} = \sum_{DNA} \cup \{x\}$

Table 5.1: Correction code table

	A	C	G	T	x
A	NA	e_{AC}	e_{AG}	e_{AT}	e_{Ax}
C	e_{CA}	NA	e_{CT}	e_{CT}	e_{Cx}
G	e_{GA}	e_{GC}	NA	e_{GT}	e_{Gx}
T	e_{TA}	e_{TC}	e_{TG}	NA	e_{Tx}
x	e_{xA}	e_{xC}	e_{xG}	e_{xT}	NA

The format of a proposed error code can be given by a triple.

(String Id, Error index in the string, Error code)

Consider the following substring "banana" which all strings share in Table 5.2. Table 5.2 depicts a deletion error at the position j_1 in $String_1$ and a substitution error in $String_4$.

The scanning algorithm scans the prefixes and suffixes to explore mismatches. Once a mismatch is found it is written down as shown below.

– Deletion error entries for the $String_1$

($String_1$, j_1, e_{xs}) – from the $String_2$

($String_1$, j_1, e_{xs}) – from the $String_3$

($String_1$, j_1, e_{xs}) – from the $String_4$

In order to fix a deletion error at the position j_1 of the $String_1$, there will be three entries that suggest a new s character should be inserted.

– Substitution error entries for the $String_4$

($String_4$, i_4, e_{ae})

(String_4, i_4, e_{ae})

(String_4, i_4, e_{ae})

These entries suggest that there is a substitution error at i_4 in the String_4. The voting stage will count the proposed error entries and decide that the character should be e.

Table 5.2: Substitution error in String_4, and deletion error in String_1

	$\overset{1}{\leftarrow}$	i_1		j_1	$\overset{2}{\rightarrow}$
String_1	orang	e	banana		trawberry
		i_2		j_2	
String_2	orang	e	banana	s	trawberry
		i_3		j_3	
String_3	orang	e	banana	s	trawberry
		i_4		j_4	
String_4	orang	a	banana	s	trawberry

5.3.1 MPI and GPGPU programming models

The Error correction model is implemented using the Compute Unified Device Architecture (CUDA) and Message Passing Interface (MPI). CUDA is a parallel computing platform and programming model created by NVIDIA.

Moreover, CUDA is a parallel programming language extending general programming languages (C, C++ and Fortran). CUDA enables users to write parallel programs for NVIDIA GPUs [105]. A typical CUDA program includes two parts, a host program running one or more sequential threads on a host CPU, and one or more parallel kernels able to execute on Tesla [106], Fermi [107] and Kepler [108] NVIDIA unified computing architectures.

A kernel is a piece of program launched on a set of concurrent threads. These threads are organized into a grid of thread blocks, where all threads in a block can synchronize through barriers and communicate via a high-speed shared memory. Threads from different thread blocks in the grid are able to cooperate through atomic oper-

ations on global memory. Unified graphic and computing devices are featured with multi level memory hierarchy, including global and local memory, cached texture and constant memory as well as shared memory and registers.

The CUDA-enabled processors are built around a fully programmable scalable processor array, organized into a number of streaming multiprocessors (SMXs in Kepler architecture, SMs in older architectures). Each streaming multiprocessor contains 8 scalar processors (SPs), 32 SPs and 192 SPs in the Tesla, the Fermi and the Kepler architectures respectively.

While the on-chip memory size was fixed 16 KB on in the Tesla series, in the later generations each multiprocessor has a configurable shared memory size from the 64 KB on-chip memory. This on-chip memory can be configured as 48 KB of shared memory with 16 KB of L1 cache or as 16 KB of shared with 48 KB of L1 cache. When executing a thread block, all the threads in the block are split into small groups of 32 parallel threads, called warps, which are scheduled in a single instruction, multiple thread (SIMT) fashion. Since all threads of a warp take the same execution path, branch divergence or warp divergence is allowed for threads when some threads may need to execute different instructions.

Message Passing Interface (MPI), is a de facto standard for developing portable parallel applications on a variety of hardware topologies [103]. MPI works on both shared and distributed memory systems. In MPI, it defines each worker as a process and enables the processes to execute different programs. This multiple program, multiple data model offers more flexibility for data-shared or data-distributed parallel program design. Within a computation, processes communicate data by calling runtime peer-to-peer and collective communication routines, specified for the C/C++ and Fortran programming languages.

5.4 Implementation

The error correction module has two phases. The first phase checks substring matches between given two strings. The process in this stage is similar to the process of the filter module.

The promising errors stored in a list which is the byproduct of the scanning stage. Figure 5.1 depicts the scanning process in the error correction module.

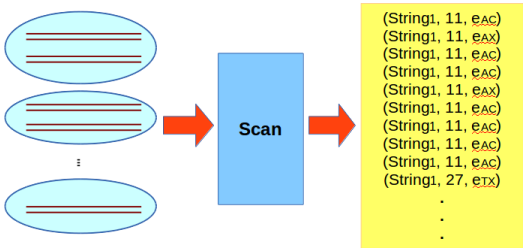


Figure 5.1: Error correction module phase 1: scanning process is for determining and reporting misinterpreted read positions and error codes into a table for all sequences that are approved by alignment module.

In the implementation, the scanning procedure runs on a CPU.

The list of promising errors is processed by another function – voting and fixing. Figure 5.2 depicts the scanning process for error correction module.

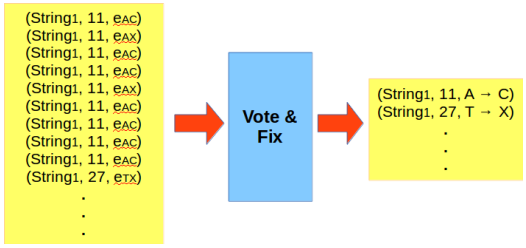


Figure 5.2: Error correction module phase 2: voting and fixing process gets the error table, orders the table which helps decide whether there should be a correction reporting misinterpreted read positions and error codes for all sequences that are selected by the alignment module.

My target set will contain longer sequences around >300bp. Since CUDA-EC is able to handle only short sequences, I compared the results of my module with DecGPU in terms of time and correction efficiency.

Even though NGS sequencers can read billions of sequences, the length of the sequences are generally quite short which is not suitable for my framework. Yet there are some NGS platforms such as Illumina MiSeq and Ion PGM that are able to read up to 400 base-pairs.

Table 5.3: Two strings share a substring m, where $|m| \geq min_match_length$

	$\overset{1}{\leftarrow}$		$\overset{2}{\rightarrow}$
String_1	α	m	γ
String_2	β	m	ω

Algorithm 4 Error correction module phase 1: scanning function

```
1: scan the prefixes α and β backward
2: for all mismatches between String_1 and String_2 do
3:     Record the mismatch positions
4:     For String_1 (String_1, Error position in String_1, error code)
5:     For String_2 (String_2, Error position in String_2, error code)
6: end for
7: scan the suffixes γ and ω forward
8: for all mismatches between String_1 and String_2 do
9:     Record the mismatch positions
10:    For String_1 (String_1, Error position in String_1, error code)
11:    For String_2 (String_2, Error position in String_2, error code)
12: end for
```

Algorithm 4 depicts the scan process on the two sequences listed in Table 5.3. The function scans prefixes (α, β) and suffixes (γ, ω) to extract more mismatch characters and record the new promising errors.

Analysis

Since each thread is assigned to update one string in the voting Algorithm 5, we can analyze the complexity per thread.

Assuming a 2% average sequencing error rate there are approximately 6 and 15 errors for the sequences 300bp and 760bp of length respectively.

Let l, c, and r be the average sequence length, coverage and error rate for a sequence set respectively. The size of proposed error count for a specific string s_i at

Algorithm 5 Error correction module phase 2: voting function

```
Input: S={s_0, s_1, s_2, ...s_{n-1}, }, sequences
Input: PE={P_0, P_1, P_2, ...P_{n-1}, }, Promising error records for each sequence
KernelVote function (S, PE)
Thread_i on s_i
for j =0 to |s_i| do
    for k =0 to |P_i| do
        count the error recordings for the position j
    end for
    Update the base at the position j in string s_i.
end for
```

most can be:

$$P_i = (c-1) \times l \times r$$

For example, when coverage is 10, error rate 2% and average length is 750, the size of the proposed error list becomes 135. Keeping the coverage and error rate the same, for a set of sequences $l = 300$ the number of proposed error entries becomes 54. Thus, the complexity of the voting algorithm per thread is

$$O(|P_i| \times |s_i|).$$

5.5 Results

I compared DecGPU and my error correcting kernel on two sets that contain longer sequences of around 750bp.

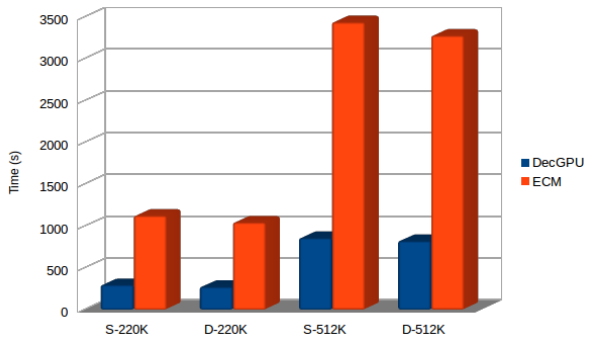


Figure 5.3: Comparing timing of DecGPU and my Error correction Module (when number of iterations is set to 10 for DecGPU)

Table 5.4: Comparing timing of DecGPU and my Error correction Kernel (when number of iterations is set to 10 for DecGPU)

Test File	DecGPU	Error correction Kernel	Performance Ratio
S-220K	295.813	1124.325	3.80
D-220K	273.351	1042.478	3.81
S-512K	857.958	3437.482	4.03
D-512K	822.314	3276.657	3.98

In terms of speed, DecGPU is approximately 3.9 times faster than my error correction tool because the scan procedure is running on CPU rather than GPU. After the scanning process is completed, error code list is transferred to GPU to execute the vote and fix kernel.

All the tests are conducted on a Stampede computing node which has the specifications in Table 3.7. The efficiency of the two error correction tools is displayed in Figure 5.4.

I have evaluated the performance of my algorithm using the simulated datasets in terms of the run time and the ability to correct erroneous reads. Table 5.5 shows the corresponding definitions of true positive (TP), false positive (FP), true negative (TN) and false negative (FN).

The sensitivity and specificity measures are defined in Equation 5.1. Liu et. al. proved that when the the coverage is high (≥ 30) and sequences are short the DecGPU achieves high accuracy. Unfortunately, when the coverage is low and sequences are long DecGPU sensitivity decreases gradually.

$$Sens = \frac{TP}{TP + FN}$$

$$Spec = \frac{TP}{TP + FP} \tag{5.1}$$

Table 5.5: Definitions for the classification test

Read Condition		**Classification**
Erroneous	Error-free	
TP	FP	Detected as erroneous
FN	TN	Detected as error-free

The performance of correcting erroneous reads is evaluated using the simulated datasets. The error rates are calculated by doing a base-by-base comparison with

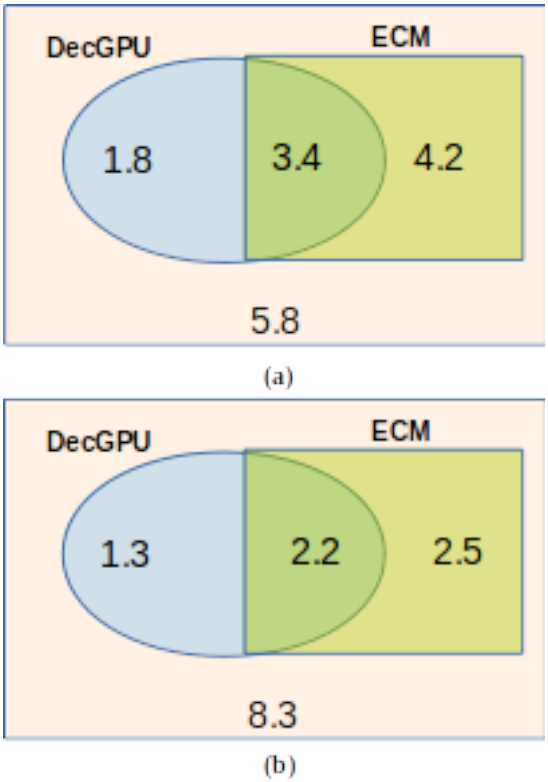


Figure 5.4: The efficiency of DecGPU and my Error correction Module: (a) on the S-256 and S-512 test files

Table 5.6: Summary of the classification test for simulated datasets

Data set	Algorithm	TP	FP	FN	Sens.	Spec.
S-220	DecGPU	1098944	424531	2102375	34.32%	72.13%
	ECM	1224083	395868	1270282	49.07%	75.56%
S-512	DecGPU	17926822	13354468	39510846	31.21%	57.30%
	ECM	20185446	12185673	28150195	41.76%	62.35%

their respective original reads (without errors).

DecGPU has a feature called the number of fixing iterations. This feature can be modified with the intention to find and correct more than one erroneous base in a single read. Since the read-length of the test files are significantly long, I set the number of fixing iterations variable to 10, so that DecGPU produces the highest sensitivity and specificity scores.

The results of the classification test are shown in Table for the six simulated datasets, where the sensitivity and specificity values have been multiplied by 100. From the sensitivity measure, ECM achieves better performance for all datasets. But the sensitivity is $< 41\%$, meaning that more than half of the erroneous reads remain undetected. Due to long read length and low coverage (S-220 has 10X coverage and S-512 has 6X) both of the algorithm suffer to find erroneous bases.

Chapter 6

Meta-Genome Gene Identification

6.1 Motivation

Once the error correction module fixes sequencing errors, it also extracts all possible *open reading frames (ORFs)* present in the fragments. The MGC module evaluates these ORFs by implementing a parallel version of the MGC algorithm developed by El Allali and Rose [109].

Next generation sequencing (NGS) is preferred in meta-genomics to traditional sequencing since NGS can produce a much larger amount of data. However, the resulting sequences are not complete and may come from many different species. Therefore, the assembly and annotation of the meta-genomic data is a challenging task. Meta-genome assembly does not work well due in part to the presence of homologous sequences from the many species present. One way to deal with these difficulties is to go directly to gene identification and bypass the assembly step.

Computational gene finding methods have proven their robustness in identifying genes in complete genomes. However, meta-genomic sequencing has presented new challenges due to the incomplete and fragmented nature of the data. During the last few years, attempts have been made to extract complete and incomplete ORFs directly from short reads and to identify the coding ORFs.

6.2 Background

New methods are being developed to predict genes specifically in meta-genomics. The best known methods in this field are MetaGene [110], Orphelia [111], and FragGeneScan [112]. MetaGene and GeneMark.hmm [113] have similar approaches. The method of MetaGene takes into account the GC-content sensitive monocodon and dicodon models computed from fully annotated genomes.

Orphelia obtains better performance than MetaGene by using a two-stage machine learning approach. The first stage builds linear discriminants for monocodon and dicodon usage as well as the translation initiation start (TIS) features extracted from the ORFs. In this step, the features are linearly extracted from the high dimensional search space [111]. The next stage combines the features obtained from the linear discriminants as well as length and GC-content features using a non-linear neural network which produces the probability that a given ORF encodes a protein. As a final step, Orphelia uses probabilities from the scoring mechanism in order to find the overlap.

FragGeneScan is an algorithm based on hidden Markov models (HMM). It is able to predict genes in both complete genomes and metagenomic fragments [112]. The algorithm combines codon usage, sequence patterns for start/stop codons and sequencing error models using HMMs. The Viterbi algorithm is used to decide the best path of hidden states that generates the observed nucleotide fragment. For further information see Rho et al. [112].

I propose to implement a parallel version of the metagenomics gene caller, MGC [109], which is based on a two-stage machine learning approach similar to that of the program Orphelia [111]. According to Chan and Stolfo [114], the models for machine learning classification learned from disjoint partitions of a dataset performs better than a single model learned from the entire dataset.

MGC learns separate models for several pre-defined GC ranges as opposed to the single model approach used by Orphelia and applies the appropriate model to each fragment based on its GC-content. Separating the training data by GC-content provides MGC with mutually exclusive partitions of the data in order to train multiple models [109].

GC-content is used to partition the training dataset for the MGC method. The use of GC-content for this purpose was inspired by the relationship between nucleotide bias and amino acid composition. Singer and Hickey [115] demonstrated that nucleotide bias can have a strong effect on the amino acid composition of the encoded proteins. This effect is not only proven in complete genomes but it is also valid for individual genes [115]. Separating the models by GC-content can ensure that different compositions are accounted for instead of combining them into one model [109].

GC-content influences codon usage which in turn influences the amino acid usage. Lightfield et al. demonstrated that use of amino acids encoded by GC-rich codons increased by approximately 1% for each 10% increase in genomic GC-content, the conversee was also true for GC-poor codons. Separating GC-contents into several GC ranges will ensure that the different linear discriminants can separate the codon and amino acid usage more precisely [116].

6.3 The MGC algorithm

Like Orphelia, MGC has a two-stage machine learning approach [109, 111]. The first stage includes linear discriminants that are used to compact any high dimensional feature space into smaller ones.

Several linear discriminants were trained based on GC-content ranges. First the training data is split into GC ranges which are defined so that the number of training sequences in all these ranges is the same. For example, El Allali and Rose split the GC spectrum into ranges where each partition contains 10% of the sequences

in the training data [109]. They then used the data from each range to create all the necessary discriminants to compute the features. The first phase of MGC in the Figure 6.2 illustrates the linear discriminant stage of MGC for a particular GC range and shows all nine features used in the second stage of the MGC algorithm [109]. The different locations of an ORF in a fragment is illustrated in Figure 6.1.

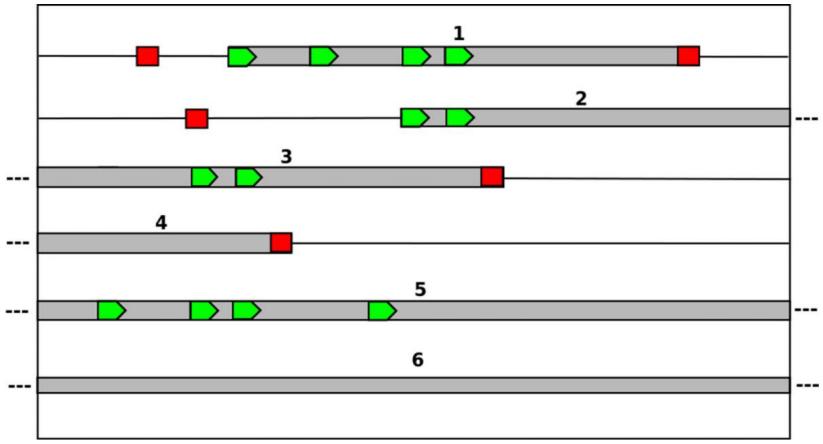


Figure 6.1: **The possible ORF positions within the forward strand of a fragment.** The fragment is depicted by the outside box and gray bars represent possible ORFs. Candidate translation initiation sites are represented by green pentagons and red squares indicate stop codons. (Obtained from El Allali and Rose [109])

Once the models are trained, all possible complete and incomplete ORFs are extracted from the test set as was done in the training phase [109]. Based on the GC-content of the fragment, the corresponding neural network model is used to score the ORF. The output of the neural network is the approximation of the posterior probability that the ORF is coding. Step 2 in Figure 6.2 illustrates the neural network model. After scoring all hypothetical ORFs, overlapping ORFs resolved. The same greedy algorithm used by Orphelia is used to determine the overlap between all candidate ORFs that have a probability greater than 0.5. Given the candidate list for a particular fragment containing all ORFs i with probability $P_i > 0.5$, Algorithm

6 [109] describes the selection scheme used to generate the final list of genes. The maximum allowed overlap $o_{max} = 60$bp which is the minimal gene length considered for prediction.

Algorithm 6 The final candidate selection [109]

1: **while** $L \neq \emptyset$ **do**
2: Find i_{max} $\text{argmax}_i P_i$, $\forall i \in L$
3: Move ORF i_{max} from L to ł
4: Remove all the ORFs in L that overlap with ORF i_{max} by more than o_{max}
5: **end while**

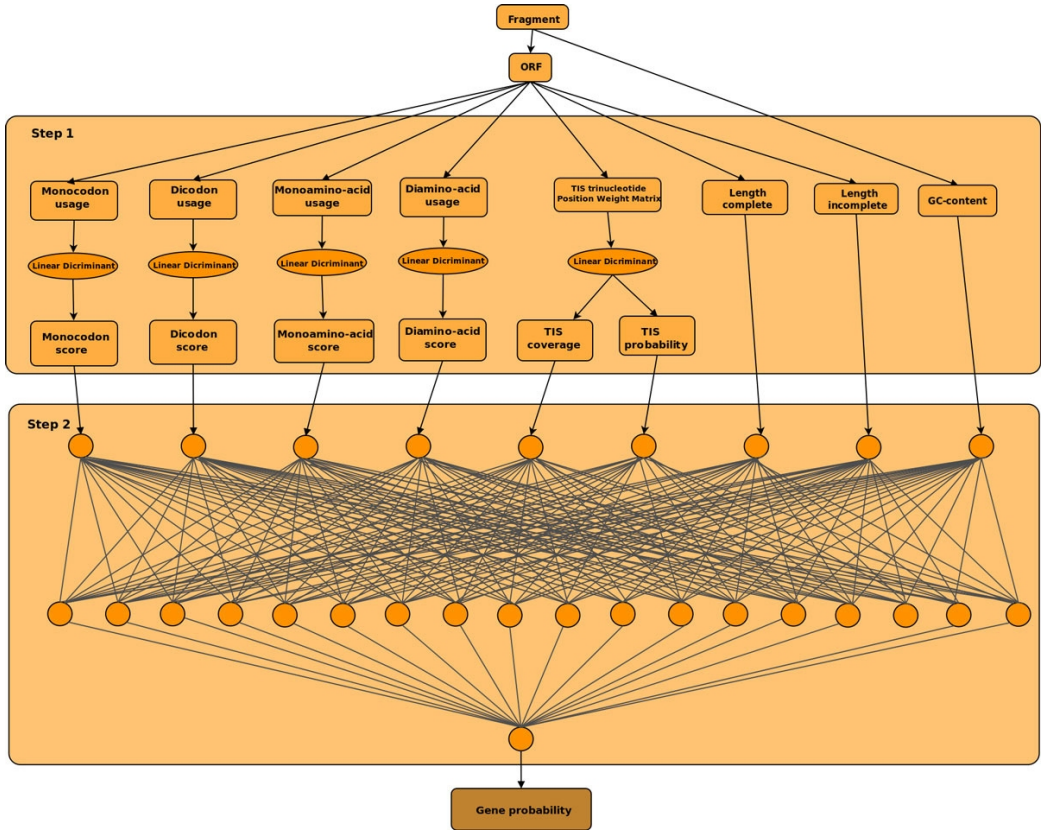


Figure 6.2: **MGC's scoring scheme:** The first steps computes six features from the ORF based on the corresponding linear discriminant and two additional features are computed directly from the ORF. The last feature is derived from directly the fragment. The neural network model from the corresponding GC-range is used to combine features from the previous step in order to compute a final gene probability. (Obtained from El Allali and Rose [109])

Features of the Model

In order to train the models in my multi-threaded MGC module, I use the same nine features that El Allali and Rose derived [109]. Standard discriminant codon features and the amino discriminant features are derived from amino acid usage in a similar way. Since there are 20 amino-acids and one stop codon, the monoamino-acid usage is based on the 21 amino-acid frequencies that represent the occurrences of successive single amino-acids in the training sequences while the diamino-acid usage is extracted from the 21^2 diamino-acid frequencies which represent the occurrences of successive half-overlapping amino acid tuples in the training sequences. Linear discriminant analysis based on the monoamino and diamino-acid usage is then used to reduce this high dimensional space to two features. For further information regarding how the derivation of linear discriminants see the work of El Allali and Rose [109].

Neural networks

El Allali and Rose combined the nine features in each GC-range in a multilayer neural network [109]. The output of each network is the posterior probability of an ORF encoding a protein. This is similar to Orphelia [111] with the exception that El Allali introduces two more features, and his models are GC-range specific. For each GC range El Allali obtains a model using features computed from all the sequences in the training dataset that have GC-content within the GC range. The same GC ranges used to compute the linear discriminants are used to build the neural network models. Different splits by GC-content were used to study the effect of the GC range size on the performance of MGC. In the study of El Allali and Rose, the MGC models were trained using the 10%, 5% , 2.5% ranges. In my case, if the number of models that was trained is multiple of the SMX count in a GPU, the occupancy of the GPU increases which increases throughput of module.

The neural network used by Orphelia [111] is a standard multilayer perceptron with one layer of k hidden nodes and a single logistic output function. A binary classification is produced with classification labels:

$y_i = 1$ for coding and

$y_i = 0$ for noncoding.

The output of the neural network is considered an approximation of the posterior probability of the coding class which is used in the final step to select the final ORFs. The k hidden activations z_i for a given input feature vector x are:

$$z_i = tanh(w_I^i \times x + b_I^i) \tag{6.1}$$

where w_I^i are input weight vectors and b_I^i are the bias parameters. The prediction function based on weight vector w_o and bias b_o is

$$g(z) = \frac{1}{1 + e^{-(w_o \times z + b_o)}} \tag{6.2}$$

where z is a vector containing all the z_i vectors.

The output of the trained network $f(x_i; \theta) \forall i \in (1..N)$ is computed by minimizing the objective function $E(\theta)$ in equation 6.3 where x_i represent the training examples, N is the number of training examples, the weight and bias parameters are referred to by the vector θ and the matrix A contains the regularization parameters.

$$E(\theta) = \sum_{i=1}^{N} (f(x_i; \theta) - y_i +)^2 + \theta^T A \theta. \tag{6.3}$$

Four strictly positive hyper-parameters are needed in the regularization matrix $A = diag(a1, \ldots, a1, a2, \ldots, a2, a3, \ldots, a3, a4)$ for separate scaling of the parameters w_I^i, b_I^i, w_o , b_o.

MacKay [118] introduced the evidence framework which is based on a Gaussian approximation of the posterior distribution of network weights. This adaptation of the hyper parameters is incorporated into the network training and uses the same training points [109, 119].

6.4.1 Implementation of HP-MGC

My HP-MGC module follows the same algorithm as the original MGC [109]. The only difference is that the learning phase and the classification are parallelized.

There are two approaches to running the MGC program in parallel. The first choice is to run MGC parallel on GC ranges. For example if 10% GC ranges are considered, the MGC module can compute 10 models in parallel. Let θ_j, where $j \in 1..10$, denote the resulting neural network model for a given GC range j. Training the model θ_j is similar to training the single model θ as described above only the training examples that have GC-content within the GC range j. The network output for a given test sample x_i is computed as $\mathrm{f}(x_i; \theta_j) = P_i$, where the GC-content of the fragment that contains x_i is within the GC range j.

The second option is to split a dataset into smaller pieces. This approach is much better than the first one because Orphelia cannot extract ORF information when the test file is huge. When the size of the dataset is over 100K reads Orphelia encounters out of memory error after running 73 minutes. The smallest data set that we have is 220000 sequences.

Splitting a dataset into smaller pieces helps Orphelia to extract ORF sequences. Then, MGC module uses this data.

6.5 Results

Hoff et al. measured the performance of the neural network by using the sensitivity and specificity measures in equations 6.4 and 6.5 to measure the capability of detecting

annotated genes and the reliability of gene predictions respectively [119]. TP_{gene} is the number of ORFs that match at least 60bp on an annotated gene in the same reading-frame, while FN_{gene} is the number of overlooked genes and FP_{gene} refers to the number of predicted ORFs that do not match the annotation. For comparison reasons we follow the same use of the positive likelihood score as a measure of specificity, this score does not take into account the number of true negatives and is used by metagenomic gene finders such as Orphelia, FragGeneScan, MetaGene, and MGC.

$$Sens = \frac{TP_{gene}}{TP_{gene} + FN_{gene}}. \tag{6.4}$$

$$Spec = \frac{TP_{gene}}{TP_{gene} + FP_{gene}}. \tag{6.5}$$

The harmonic mean which can be constructed by merging the sensitivity and specificity:

$$HarmonicMean = \frac{2 \times Sens \times Spec}{Sens + Spec}. \tag{6.6}$$

In this study, we did not consider the accuracy of the HP-MGC algorithm because it will be identical to which has been thoroughly tested and reported in [109]. Consequently, we focus on reporting the timing results of HP-MGC and Orphelia tool.

In this chapter we give the timing results for classification of HP-MGC only.

Before starting gene identification process, both Orphelia and MGC requires ORF sets. ORF sequences are extracted by Orphelia, and Table 6.2 summarizes the run time for this phase. In the table, S-220 and D-220 are datasets which we explained

Table 6.1: The timing result for extracting ORF sequences

Partitions	$T_S - 220$ (s)	# of ORFs	$T_D - 220$ (s)	# of ORFs
0-20K	105.084	689498	239.452	1564093
20-40K	93.225	651825	215.455	1572245
40-60K	152.087	920056	209.194	1563241
60-80K	143.571	879489	207.671	1559486
80-100K	142.812	868452	211.409	1565714
100-120K	121.644	658522	212.365	1561857
120-140K	97.680	635460	207.857	1557170
140-160K	91.654	661298	210.916	1565213
160-180K	224.836	1120229	211.257	1562981
180-200K	128.586	652557	203.755	1530482
200-220K	117.847	695548	202.373	1525938

Table 6.2: Orphelia run time results for gene identification

Partitions	S-220	D-220
0-20K	657.33	1631.341
20-40K	746.375	1638.475
40-60K	916.405	1638.073
60-80K	924.385	1540.895
80-100K	899.933	1589.206
100-120K	983.878	1606.096
120-140K	966.104	1554.535
140-160K	985.852	1560.147
160-180K	866.905	1627.653
180-200K	675.524	1568.865
200-220K	1063.637	1410.300

in Table 3.6. Each test file is fragmented into partitions where each segment includes 20K sequences. Extraction phase run time varies because each fragment has different gene content. Since D-220 dataset has more diversity, it has more ORF sequences.

There is a strong correlation between number of ORF sequences and run times for identifying genes. In the gene identification phase Orphelia is approximately 4.3 times faster than HP-MGC per processing a partition. In the future, I plan to implement a GPU kernel to speed up HP-MGC module.

Table 6.3: HP-MGC model timing results

Partitions	S-220	D-220
0-20K	3347.409	7830.050
20-40K	3159.125	7885.205
40-60K	3150.280	7846.274
60-80K	3174.445	7392.043
80-100K	3013.026	7528.547
100-120K	3225.641	7409.265
120-140K	3273.300	7285.859
140-160K	3164.474	7327.085
160-180K	3161.145	7517.305
180-200K	3142.785	7226.415
200-220K	3177.740	7029.609

Table 6.3 depicts the timing result of HP-MGC model. In Stampede supercomputer we are able to run 12 Matlab threads simultaneously. In this test, it takes at most 3177.74 seconds for a fragment. Totally HP-MGC can identify genes in an hour for each test file.

Chapter 7

Conclusion

Traditional sequencing techniques are not suitable for determining the diversity of a microbial community because only a very small portion of all microbial species can be cultured. Thus, researchers are following an alternative approach in meta-genomics.

Bypassing the assembly process provides a means of avoiding the limitations of culture-dependent genetic exploitation. The contribution of this study was to construct a *high-performance meta-genome gene identification framework*. The pipeline of the framework was presented in Figure 1.2 in Chapter 1.

The main goal of this work is the development of methods that can scale to the largest available sequence data sets.

During the research, I used the Planck hybrid supercomputer and the Maxwell super computer here at USC mainly for development purposes. The Planck Cluster combines a mixture of 264 CPU cores. It also includes 57 Nvidia GPGPU accelerators boards. The theoretical peak performance of Planck Cluster is 59 Teraflops. The hardware of the Maxwell supercomputer combines a mixture of 40 GL390 Nodes each with 12 cores per node, Intel Xeon 2.4 GHz, 24 GB RAM and 6 SL250 nodes with 16 cores per node, Intel Xeon 2.60GHz, 32 GB RAM. The head node is a DL380 with 12 core, 48GB RAM. The storage attached to the Maxwell supercomputer is 24 TB.

In order to test modules in the framework, I used the TACC Stampede supercomputer. Each node in the TACC Stampede cluster contains dual 2.7 GHz eight-core Intel Xeon E5-2680 CPUs that can each execute 16 MPI processes. Also each node

has a NVIDIA K20 accelerator (see Table 3.7).

The contribution of this study is to design and develop the *high-performance meta-genome gene identification framework*. The following list depicts the portions of the framework I designed, implemented and tested:

1. **Filter Module:** This module makes a coarse clustering of the raw NGS set.
2. **Alignment Module:** The alignment module receives the list of promising pairs as its input and run a global alignment procedure to produce the final clusters from the collection of sequences produced by the filter module.
3. **Error correction module:** This method corrects bases that were misinterpreted by the sequencer, which helps boost identification of open reading frames by the next module.
4. **HP-MGC module:** Finally, HP-MGC module extracts genes from the sequences.

The filter and error correction modules were implemented from scratch. The alignment kernel function was inserted between these modules. Finally the core function in MGC module was modified so that it can run parallel.

7.1 Future work

In the future, we plan to improve the error correction module by implementing a GPU kernel function for scanning process. Consequently, all the component of the error module will run on GPU.

Even though we accelerated the filter, alignment, and error correction modules, the HP-MGC module still runs on the host device. We will improve our *high-performance meta-genomic gene identification framework* by implementing a GPU version HP-MGC.

Bibliography

[1] Ward D.M., Weller R., Bateson M.M. 16S rRNA sequences reveal numerous uncultured microorganisms in a natural community. *Nature*, 345:63–65, 1990.

[2] Goebel U.B. Phylogenetic amplification for the detection of uncultured bacteria and the analysis of complex microbiota. *Journal of Microbiological Methods*, 23:117–128, 1995.

[3] Gonzalez J.M., Saiz-Jimenez C. Application of molecular nucleic acid based techniques for the study of microbial communities in monuments. *Intl Microbiol*, 8:189–194, 2005.

[4] Chen K., Pachter L., Bioinformatics for whole-genome shotgun sequencing of microbial communities. *PLoS Comput. Biol.* 1:106–112, 2005.

[5] HiSeq X^{TM} Ten, datasheet $1000 human genome and extreme throughput for population-scale sequencing. *http://systems.illumina.com /content/dam/illumina-marketing /documents/products /datasheets/datasheet-hiseq-x-ten.pdf*, 2014.

[6] Drmanac R., Sparks A.B., Callow M.J., Halpern A.L., Burns N.L., Kermani B.G., Human genome sequencing using unchained base reads on self-assembling DNA nanoarrays. *Science*, 327:78–81, 2010.

[7] Zhou X., Ren L., Li Y., Zhang M., Yu Y., Yu J., The next-generation sequencing technology: A technology review and future perspective. *Science China Life Sci.*, 53:44–57, 2010.

[8] —— Toward precision medicine building a knowledge network for biomedical research and a new taxonomy of disease National Research Council U.S. Committee on A Framework for Developing a New Taxonomy of Disease. 2011.

[9] Trajanoski Zlatko, *Computational Medicine IX*, 203p, 2012.

[10] Wetterstrand K.A., DNA sequencing costs: Data from the NHGRI large-scale genome sequencing program [http://www.genome.gov/sequencingcosts] accessed Jan 1 2014.

[11] Watson J.D., Crick F.H., The structure of DNA. *Cold Spring Harb. Symp. Quant. Biol.*, 18: 123–31, 1953.

[12] Sanger F., Coulson A.R. A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase. *J. Mol. Biol.* 94/3:441–8, 1975.

[13] Sanger F., Nicklen S., Coulson A.R., DNA sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci. U.S.A.* 74/12:5463-7, 1977.

[14] Sanger F., Air G. M., Barrell B. G., Brown N. L., Coulson A. R., Fiddes J. C., Hutchison C. A., Slocombe P. M. et al. Nucleotide sequence of bacteriophage ϕX174 DNA. *Nature*, 265/5596: 687–95, 1977.

[15] Adams M.D. et al., The genome sequence of Drosophila melanogaster. *Science*, 24/287(5461):2185-95, 2000.

[16] Venter J.C. et al. The sequence of the human genome. *Science*, 16/291(5507):1304-51, 2001.

[17] Venter J.C. et al. Environmental genome shotgun sequencing of the Sargasso Sea. *Science*, 2/304(5667):66–74, 2004.

[18] Ten Bosch J.R., Grody W.W., Keeping Up with the Next Generation. *The Journal of Molecular Diagnostics*, 10(6):484–492, 2008.

[19] Tucker T., Marra M., Friedman J.M., Massively Parallel Sequencing: The Next Big Thing in Genetic Medicine. *The American Journal of Human Genetics*, 85(2):142–154, 2009.

[20] Lin Liu, Yinhu Li, Siliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, Maggie Law, Comparison of Next-Generation Sequencing Systems. *Journal of Biomedicine and Biotechnology*, 1–11, 2012.

[21] Hirschberg D.S., A linear space algorithm for computing maximal common subsequences. *Communications of the ACM* 18/6:341–343, 1975.

[22] Fickett J.W., Fast optimal alignment. *Nucleic Acids Res.*, 12:175–180, 1984.

[23] Gotoh Osamu. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162:705, 1982.

[24] Burke J., Davison, D., and Hide W. d2_cluster: A validated method for clustering EST and fulllength cDNA sequences. *Genome Research*, 9(11):1135–1142, 1999.

[25] Ukkonen E. Approximate stringmatching with qgrams and maximal matches. *Theoretical Gomputer Science*, 92(1):191–211, 1992.

[26] Jackson B. and Aluru S. Chapter 1: Pairwise sequence alignment. *In Handbook of computational molecular biology*, CRC Press, 2005.

[27] Gusfield D. Algorithms on Strings, Trees, and Sequences *Cambridge University Press*, 1997.

[28] Needleman S., Wunsch C., A general method applicable to the search for similarities in the amino acid sequence of two proteins, *Journal of molecular biology*, 48/3:443–453, 1970.

[29] Smith T.F., Waterman M.S., Identification of common molecular subsequences, *Journal of Molecular Biology*, 147:195–197, 1981.

[30] Brudno M., Malde S., Poliakov A., Do C.B., Couronne O., Dubchak I., Batzoglou S. Glocal alignment: finding rearrangements during alignment. *Bioinformatics*, 19. Suppl 1 (90001): i54–62, 2003.

[31] NVIDIA Corporation, NVIDIA CUDA Programming Guide, June, 2011.

[32] AMD Corporation, ATI Stream Computing OpenCL Programming Guide, Aug. 2010.

[33] Nickolls J., Buck I., Garland M., Skadron K., Scalable parallel programming with CUDA. *ACM Queue*, 6/2:40–53, 2008.

[34] Lindholm E., Nickolls J., Oberman S., Montrym J., NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28/2:39–55, 2008.

[35] NVIDIA: Fermi: NVIDIA's Next Generation CUDA Compute Architecture. [www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf]

[36] NVIDIA: Kepler GK110 Architecture [www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf]

[37] Kalyanaraman A., Aluru S., Brendel V., Kothari S., Space and time efficient parallel algorithms and software for EST clustering, *IEEE Transactions on parallel and distributed systems*, 14:1209–1221, 2003.

[38] Kalyanaraman A., Emrich S.J., Schnable P.S., Aluru S., Assembling genomes on large-scale parallel computers. *Journal of Parallel and Distributed Computing (JPDC)*, 67(12):1240–1255, 2007.

[39] Aluru S., Handbook of Computational Molecular Biology *(Chapman & All/CRC Computer and Information Science Series). Chapman & Hall/CRC*, 2005.

[40] Manber U., Myers G., Suffix arrays: a new method for on-line string searches. *First Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 319-327, 1990.

[41] Manber U., Myers G., Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22/5: 935, 1993.

[42] Abouelhoda Mohamed Ibrahim, Kurtz Stefan, Ohlebusch Enno, Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2:53, 2004.

[43] Kasai T., Lee G. Arimura H., Arikawa S., Park K. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*. Lecture Notes in Computer Science 2089:181–192, 2001.

[44] Ko P., Aluru S. Space-efficient linear-time construction of suffix arrays. *In Proc. 14th Annual Symposium, Combinatorial Pattern Matching*, 200–210, 2003.

[45] Ko P., Aluru S. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3:143-156, 2005.

[46] Kärkkäinen J. Sanders P. Simple linear work suffix array construction, *Automata, Languages and Programming*, Springer Berlin Heidelberg, 943–955, 2003.

[47] Crauser A. Ferragina P., A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32:1–35, 2002.

[48] Sadakane, K. A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation. *In DCC: Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, CA, 129–138, 1998.

[49] Larsson, J. N. Sadakane, K. Faster suffix sorting *Tech. Rep*, Department of Computer Science, Lund University, Sweden, LU-CS-TR:99-214, 1999.

[50] Nong G., Zhang S., Chan W. H. Linear Suffix Array Construction by Almost Pure Induced-Sorting. *Data Compression Conference*, p.193, 2009.

[51] Kurtz S., Phillippy A., Delcher A.L., Smoot M., Shumway M., Antonescu C., and Salzberg S.L., Versatile and open software for comparing large genomes. *Genome Biology*, 5:R12, 2004.

[52] Delcher A.L., Phillippy A., Carlton J., and Salzberg S.L., Fast Algorithms for Large-scale Genome Alignment and Comparision. *Nucleic Acids Research*, 30/11:2478–2483, 2002.

[53] Schatz M.C., Trapnell C., Delcher A.L. and Varshney A., High-throughput sequence alignment using Graphics Processing Units *BMC Bioinformatics*, 8:474, 2007.

[54] Gharaibeh A. and Ripeanu M., Accelerating Sequence Alignment on Hybrid Architectures, *Scientific Computing Magazine*, January/February 2011.

[55] Gharaibeh A. and Ripeanu M., Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance, *IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2010)*, New Orleans, LA, Nov. 2010.

[56] Gusfield D., An increment-by-one approach to suffix arrays and trees, *Technical Report CSE-90-39*, UC Davis, Dept. Computer Science, 1990.

[57] Kim D.K., Jo J., Park H. A fast algorithm for constructing suffix arrays for fixed-size alphabets. *In Proceedings of the 3rd Workshop on Experimental and Efficient Algorithms (WEA 2004)*, C. C. Ribeiro and S. L. Martins, Eds. Springer-Verlag, Berlin, Germany, 301–314, 2004.

[58] Itoh H., Tanaka H. An efficient method for in memory construction of suffix arrays. *In Proceedings of the 6th Symposium on String Processing and Information Retrieval (Cancun, Mexico)*, IEEE Computer Society, Los Alamitos, CA, 81–88, 1999.

[59] Seward J. On the performance of BWT sorting algroithms. *In DCC: Data Compression Conference*. IEEE Computer Society Press, Los Alamitos, CA, 173–182, 2000.

[60] Manzini G., and Ferragina P., Engineering a lightweight suffix array construction algorithm. *Algorithmica* 40:33–50, 2004.

[61] Schürmann K. and Stoye J., An incomplex algorithm for fast suffix array construction. *In Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX05) SIAM*, 77–85, 2005.

[62] Burkhardt S. Kärkkäinen J. Fast lightweight suffix array construction and checking. *In Proceedings of the 14th Annual Symposium CPM 2003*, R. Baeza-Yates, E. Chavez, and M. Crochemore, Eds. Lecture Notes in Computer Science, vol. 2676. Springer-Verlag, Berlin, Germany, 55–69, 2003.

[63] Maniscalco M. A. *MSufSort. 2005.* [http://www.michael-maniscalco.com/msufsort.htm.]

[64] Puglisi S.J., Smyth W.F., Andrew T.H. A Taxonomy of Suffix Array Construction Algorithms *ACM Computing Surveys*, Vol. 39, No. 2/4, 2007.

[65] Richter D.C., Ott F., Auch A.F., Schmid R., Huson D.H. MetaSim - A Sequencing Simulator for Genomics and Metagenomics. *PLoS ONE* 3(10): e3373, 2008.

[66] Sample analysis results: Obese human gut (patient TS28), EBI metagenomics, Date Accessed: August 15, 2014.

[67] Kumar S., Carlsen T., Mevik B., Enger P., Blaalid R., Shalchian-Tabrizi K., Kauserud H., CLOTU: an online pipeline for processing and clustering of 454 amplicon reads into OTUs followed by taxonomic annotation, *BMC bioinformatics*, 12/1, 182p, 2011.

[68] Cole J., Wang Q., Cardenas E., Fish J., Chai B., Farris R., Kulam-Syed-Mohideen A., McGarrell D., Marsh T., Garrity G. et al., The ribosomal database project: improved alignments and new tools for rRNA analysis, *Nucleic acids research*, 37/1:D141-D145, 2009.

[69] Amber H., Sean R., Timothy M., Bertram L., and Jonathan E., Introducing WATERS: a workflow for the alignment, taxonomy, and ecology of ribosomal sequences, *BMC Bioinformatics*, 11, 2010.

[70] Ram P., Viola N., and Christian S., CANGS: a user-friendly utility for processing and analyzing 454 GS-FLX data in biodiversity studies, *BMC Research Notes*, 3, 2010.

[71] Juan F., Antonio L., No F., Francisco C., Guillermo P., and Gonzalo C., SeqTrim: a high-throughput pipeline for preprocessing any type of sequence read, *BMC Bioinformatics*, 11, 2010.

[72] Caporaso J., Kuczynski J., Stombaugh J., Bittinger K., Bushman F., Costello E., Fierer N., Pea A., Goodrich J., Gordon J. et al., QIIME allows analysis of high-throughput community sequencing data, *Nature methods*, 7/5:335–336, 2010.

[73] Quince C., Lanzn A., Curtis T., Davenport R., Hall N., Head I., Read L., and Sloan W., Accurate determination of microbial diversity from 454 pyrosequencing data, *nature methods*, 6/9:639–641, 2009.

[74] Quince C., Lanzen A., Davenport R., and Turnbaugh P., Removing noise from pyrosequenced amplicons, *BMC Bioinformatics*, 12/1: 38p, 2011.

[75] Gao Yang, Bakos Jason D., GPU Acceleration of Pyrosequencing Noise Removal, *Proc. 2012 Symposium on Application Accelerators in High Performance Computing (SAAHPC'12)*, Argonne National Laboratory, July 10-11, 2012.

[76] Savran Ibrahim , Gao Yang , Bakos Jason D., Large-scale Pairwise Sequence Alignments on a Large-scale GPU Cluster, *IEEE Design and Test*, January/February 2014.

[77] Liu, D. Maskell, and B. Schmidt, CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA enabled graphics processing units, *BMC Research Notes*, 2/1:73p, 2009.

[78] Altschul S., Gish W., Miller W., Myers E., and Lipman D., Basic local alignment search tool, *Journal of molecular biology*, 215/3:403–410, 1990.

[79] Manavski S., and Valle G., CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment, *BMC bioinformatics*, 9/2, S10, 2008.

[80] Razmyslovich D. , Marcus G., Gipp M., Zapatka M., and Szillus A., Implementation of Smith-Waterman algorithm in OpenCL for GPUs, *In Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop*, 48–56, 2010.

[81] Liu Y., Schmidt B., Maskell D.L., CUDASW++ 2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*, 3:93, 2010.

[82] Fromberg W., Kierzynka M., Blazewic J., Gawron P., and Wojciechowski P., G-DNA âĂŞ a highly efficient multi-GPU/MPI tool for aligning nucleotide reads *Bull. of the Polish Academy of Sciences Technical Sciences*, Vol. 61, No. 4, 2013.

[83] Li M., Wang I.X., Li Y., Bruzel A., Richards A.L., Toung J.M., Cheung V.G., Widespread RNA and DNA sequence differences in the human transcriptome. *Science*, 333(6038):53-58, 2011.

[84] Pickrell J.K., Gilad Y., Pritchard J.K., Widespread RNA and DNA sequence differences in the human transcriptome *Science* 1302/335(6074).

[85] Whiteford N., Skelly T., Curtis C., Ritchie M.E., Löhr A., Zaranek A.W., Abnizova I., Brown C., primary data analysis for the Illumina Solexa sequencing platform. *Bioinformatics*, 25(17):2194-2199, 2009.

[86] Li L., Speed T., An estimate of the crosstalk matrix in four-dye fluorescence-based DNA sequencing. *Electrophoresis*, 20:1522-2683, 1999.

[87] Ledergerber C., Dessimoz C., Base-calling for next-generation sequencing platforms. *Brief Bioinform*, 12(5):489–497, 2011.

[88] Nakamura K., Oshima T., Morimoto T., Ikeda S., Yoshikawa H., Shiwa Y., Ishikawa S., Linak M.C., Hirai A., Takahashi H., Altaf-Ul-Amin M., Ogasawara N., Kanaya S., Sequence-specific error profile of Illumina sequencers. *Nucleic Acids Res*, 39, 2011.

[89] Pevzner PA., Tang H., Waterman MS., An Eulerian path approach to DNA fragment assembly. *Proc Natl Acad Sci USA*, 98(17):9748-9753, 2001.

[90] Chaisson MJ., Pevzner PA., Short read fragment assembly of bacterial genomes. *Genome Res*, 18(2):324-330, 2008.

[91] Schröder J., Schröder H., Puglisi S.J., Sinha R., Schmidt B., SHREC a short-read error correction method. *Bioinformatics*, 25(17):2157-2163, 2009.

[92] Ilie L., Fazayeli F., Ilie S., HiTEC: accurate error correction in high-throughput sequencing data. *Bioinformatics*, 27(3):295-302, 2011.

[93] Wijaya E., Frith M.C., Suzuki Y., Horton P., Recount: expectation maximization based error correction tool for next generation sequencing data. *In Genome Inform*, 23:189-201, 2009.

[94] Qu W., Morishita S., Hashimoto S., Efficient frequency-based de novo short-read clustering for error trimming in next-generation sequencing. *Genome Res*, 19(7):1309-1315, 2009.

[95] Sleep Julie A. , Schreiber Andreas W. and Baumann Ute, Sequencing error correction without a reference genome. *BMC Bioinformatics*, 14:367, 2013.

[96] Schreiber A., Shi B.J., Huang C.Y., Langridge P., Baumann U., Discovery of barley miRNAs through deep sequencing of short reads. *BMC Genomics*, 12:129, 2011.

[97] Salmela L., Correction of sequencing errors in a maxed set of reads. *Bioinformatics*, 26(10):1284-1290, 2010.

[98] Liu Y., Schmidt B., Liu W., Maskell D.L., CUDA-MEME: accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units. *Pattern Recognition Letters*, 31(14):2170-2177, 2010.

[99] Shi H., Schmidt B., Liu W., Müller-Wittig W., A parallel algorithm for error correction in high-throughput short-read data on CUDA-enabled graphics hardware. *Jour. Comp. Biol.*, 17(4):603-615, 2010.

[100] Pevzner P.A., Tang H., Waterman M.S., An Eulerian path approach to DNA fragment assembly. *Proc Natl Acad Sci USA*, 98(17):9748-9753, 2001.

[101] Bloom B.H., Space/time trade-offs in hash coding with allowable errors. *Commu ACM*, 13:422-426, 1970.

[102] Shi H., Schmidt B., Liu W., Müller-Wittig W., Quality-score guided error correction for short-read sequencing data using CUDA. *Procedia Computer Science*, 1(1):1123-1132, 2010.

[103] Message Passing Interface (MPI) tutorial [https://computing.llnl.gov/tutorials/mpi]

[104] Liu Yongchao , Schmidt Bertil and Maskell Douglas L., DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI. *BMC Bioinformatics*, 12:85, 2011.

[105] Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with CUDA. ACM Queue 2008, 6(2):40-53.

[106] Lindholm E, Nickolls J, Oberman S, Montrym J.,

NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro 2008, 28(2):39-55.

[107] NVIDIA: Fermi: NVIDIA's Next Generation CUDA Compute Architecture. [http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf]

[108] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[109] El Allali Achraf, Rose John R., MGC: a metagenomic gene caller. *BMC Bioinformatics*, 14/9:S6, 2013.

[110] Noguchi H., Park J., Takagi T., MetaGene: prokaryotic gene finding from environmental genome shotgun sequences. *Nucleic Acids Research*, 34(19):5623–5630, 2006. [http://www.ncbi.nlm.nih.gov/pubmed/17028096].

[111] Hoff K.J., Lingner T., Meinicke P., Tech M., Orphelia: predicting genes in metagenomic sequencing reads. *Nucleic acids research*, 37 W101-5, 2009. [http://www.ncbi.nlm.nih.gov/pubmed/19429689].

[112] Rho M., Tang H., Ye Y., FragGeneScan: predicting genes in short and error-prone reads. *Nucleic Acids Research*, 38(20):e191, 2010.

[113] Borodovsky M., Mills R., Besemer J., Lomsadze A., Prokaryotic gene prediction using GeneMark and GeneMark.hmm. *Current protocols in bioinformatics editoral board Andreas D. Baxevanis et al.* Chapter 4:Unit4.5., 2003. [http://www.ncbi.nlm.nih.gov/pubmed/18428700].

[114] Chan P.K., Stolfo S.J., A comparative evaluation of voting and meta-learning on partitioned data. *Proc 12th International Conference on Machine Learning Morgan Kaufmann*, 90-98, 1995. [http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.7713].

[115] Singer G.A., Hickey D.A., Nucleotide bias causes a genomewide bias in the amino acid composition of proteins. *Molecular Biology and Evolution*, 17(11):1581-1588, 2000. [http://www.ncbi.nlm.nih.gov/pubmed/11070046].

[116] Lightfield J., Fram N.R., Ely B., Across Bacterial Phyla, Distantly-Related Genomes with Similar Genomic GC Content Have Similar Patterns of Amino Acid Usage. *PLoS ONE* 6(3):12, 2011.

[117] Oliver J.L., Marin A., A relationship between GC content and coding-sequence length. *Journal of Molecular Evolution*, 43(3):216-223, 1996.

[118] MacKay D.J.C., A Practical Bayesian Framework for Backpropagation Networks. *Neural Computation*, 4(3):448-472, 1992. [http://www.mitpressjournals.org/doi/abs/10.1162/neco.1992.4.3.448].

[119] Hoff K.J., Tech M., Lingner T., Daniel R., Morgenstern B., Meinicke P., Gene prediction in metagenomic fragments: a large scale machine learning approach. *BMC bioinformatics*, 9:217, 2008. [http://www.ncbi.nlm.nih.gov/pubmed/18442389].