Efficient agglomerative hierarchical clustering for biological sequence analysis

Nguyen, Thuy Diem

2015

https://hdl.handle.net/10356/65568

https://doi.org/10.32657/10356/65568

NANYANG TECHNOLOGICAL UNIVERSITY

EFFICIENT AGGLOMERATIVE HIERARCHICAL CLUSTERING FOR BIOLOGICAL SEQUENCE ANALYSIS

NGUYEN THUY DIEM
SCHOOL OF COMPUTER ENGINEERING

2015

EFFICIENT AGGLOMERATIVE HIERARCHICAL CLUSTERING FOR BIOLOGICAL SEQUENCE ANALYSIS

NGUYEN THUY DIEM

School of Computer Engineering

A thesis submitted to the Nanyang Technological University in partial fulfillment of the requirement for the degree of Doctor of Philosophy

2015

Abstract

Cluster analysis or clustering is an important data mining technique widely used for pattern recognition and information retrieval. In the context of computational biology and bioinformatics, clustering has been successfully applied in many subfields such as in evolutionary biology to group homologous genetic sequences into gene families, in transcriptomics to group genes with related expression patterns, or in ecology to describe communities of organisms in heterogeneous environments.

Cluster analysis originated from an anthropology study by Driver and Kroeber in 1932. Since then, over a hundred clustering algorithms have been developed to target input datasets with different characteristics. Out of these algorithms, two most prevalent methods are hierarchical clustering and k-means clustering. The former algorithm is particularly useful for analyzing genetic datasets in evolutionary biology studies because there are inherent hierarchical relationships amongst the genetic sequences extracted from related organisms in these datasets.

The hierarchical clustering analysis of biological sequences is however computational expensive in terms of both execution time and memory usage. Consequently, this analysis was rarely applied to input datasets with more than ten thousand sequences. In recent years, new high-throughput sequencing technologies can produce more data in a shorter time and for a cheaper cost. As a result, more and more raw sequence data is efficiently produced from the genetic materials of live organisms, many of which are examined for the first time. Hence, there is a pressing need for more effective computational techniques to study the relationships among the newly-discovered species.

Motivated by the necessity for more effective clustering algorithms to study biological sequence data, I explore the use of parallel computing technologies with new algorithms to perform agglomerative hierarchical sequence clustering in a more effective way without compromising the accuracy of the results. Specifically, I develop new parallel algorithms using general-purpose computing on graphics processing units (or GPGPU) to speedup the most compute intensive part of the hierarchical clustering process: the pairwise distance matrix computation of the input sequences. Graphic cards from NVIDIA are becoming a commodity in recent years and are available in many personal computers nowadays. With an NVIDIA GPU card, any laptop or desktop running these parallel algorithms can speed up the matrix computation process by ten times to a hundred times faster compared to the computation on a CPU using a traditional sequential (single-threaded) algorithm.

Besides reducing execution time, I have built a more memory-efficient and robust agglomerative hierarchical clustering algorithm. This new clustering method reduces memory usage by applying a data summarization technique to maintain a compact version of only a part of the distance matrix instead of loading the whole matrix into the main memory. An important feature of this algorithm is the capability to produce the same hierarchical structure as the standard method. The new algorithm supports all three popular linkage schemes including: average-linkage, single-linkage, and complete-linkage. Among them, average-linkage clustering is widely used in many research and real-world applications, making a memory-efficient average-linkage clustering algorithm in great demand.

Amongst various types of biological sequence cluster analyses, this thesis focuses on a particular type of sequence clustering application called operational taxonomic unit (OTU) clustering to demonstrate the usefulness of the afore-mentioned efficient algorithms for processing large genomic datasets. I have developed two OTU clustering pipelines for 454 pyrosequencing datasets called CRiSPy-Embed and CRiSPy-CUDA. A comprehensive evaluation benchmark using randomly simulated datasets and popular mock datasets has been designed to test the performance of these pipelines against existing tools. The benchmark results show that these tools can produce similar or more accurate OTU groupings than most existing OTU hierarchical clustering tools in a much more efficient manner.

Acknowledgements

First and foremost, I would like to express my sincere thanks to my supervisors, Prof Bertil Schmidt and Assoc Prof Kwoh Chee Keong for introducing me to the exciting fields of high performance computing and bioinformatics. I am grateful for their support, guidance and suggestions. Their expertise and academic experience have helped me become a capable researcher. In addition, I thanks Prof Kyle Rupnow for his brief but valuable guidance during the period of my PhD qualification examination.

Secondly, I would like to sincerely thank Ms. Irene Ng-Goh Siew Lai and Mr. Poliran Kenneth Caballes, the laboratory executives of the Parallel and Distributed Computing Center, for their efficient assistance and instant support in setting up and troubleshooting the hardware resources.

In addition, I would like to also thank my best friend, Pham Chau Khoa for many fruitful discussions about computers and programming. His passion for technologies and excellent programming skill have inspired me to become a better programmer.

Last but not least, my gratitude goes to my mother, Pham Thi Dieu Huong and my former mentors, Dr. Timo Bretschneider and Dr. Ian McLoughlin for the inspiration and encouragement they have provided me throughout my academic journey.

Contents

List of Figures

List of Tables

Abbreviations

BLAST	**B**asic **L**ocal **A**lignment **S**earch **T**ool
CRiSPy	**C**omputing Species **R**ichness **i**n 16**S** rRNA **Py**rosequencing Datasets
CRiSPy-Embed	**CRiSPy** using Sequence **Embed**ding
CRiSPy-CUDA	**CRiSPy** using **CUDA**
CPU	**C**entral **P**rocessing **U**nit
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
DNA	**D**eoxyribo**n**ucleic **A**cid
DP	**D**ynamic **P**rogramming
GB	**G**iga **B**ytes
GBPS	**G**iga **B**ytes **P**er **S**econd
GCUPS	**G**iga **C**ell **U**pdates **P**er **S**econd
GPU	**G**raphical **P**rocessing **U**nit
GPGPU	**G**eneral **P**urpose Computing using **GPU**
MPI	**M**essage **P**assing **I**nterface
MCUPGMA	**M**emory-**C**onstraint **UPGMA**
NGS	**N**ext-**G**eneration **S**equencing
NMI	**N**ormalized **M**utual **I**nformation
OpenMP	**Open** **M**ulti**p**rocessing
OTU	**O**perational **T**axonomic **U**nit
PCI	**P**eripheral **C**omponent **I**nterconnect
PCR	**P**olymerase **C**hain **R**eaction
RAM	**R**andom **A**ccess **M**emory
rRNA	**R**ibosomal **R**ibo**n**ucleic **A**cid

SIMT	**S**ingle **I**nstruction **M**ultiple **T**hreads
SP	**S**calar **P**rocessor
SM	**S**treaming **M**ultiprocessor
UPGMA	**U**nweighted **P**air **G**roup **M**ethod with **A**rithmetic Mean

Symbols

For distances

$d_e(V_x, V_y)$	Euclidean distance between R_x and R_y
$d_k(R_x, R_y)$	k-mer distance between R_x and R_y
$d_g(R_x, R_y)$	genetic distance between R_x and R_y
θ_e	Euclidean distance cutoff
θ_g	genetic distance cutoff
θ_k	k-mer distance cutoff
k	tuple size for computing k-mer distances
V_S	standard vertical scoring matrix for global alignment
H_S	standard horizontal scoring matrix for global alignment
D_S	standard diagonal scoring matrix for global alignment
V_B	binary vertical scoring matrix for global alignment
H_B	binary horizontal scoring matrix for global alignment
D_B	binary diagonal scoring matrix for global alignment in
SM	scoring matrix for global alignment in CRiSPy-CUDA
AL	aligned length matrix for global alignment in CRiSPy-CUDA
ML	mismatch length matrix for global alignment in CRiSPy-CUDA
$sbt(a = b)$	match score
$sbt(a \neq b)$	mismatch score
α	gap opening penalty
β	gap extension penalty

For SparseHC

$E^{(t)}$	exact link map at clustering iteration step t
$I^{(t)}$	inexact link map at clustering iteration step t
M	amount of memory allocated to SparseHC
θ	the maximum distance value in sparse distance matrix
λ	the maximum distance value that has been loaded by SparseHC

For CRiSPy

Σ	DNA alphabet
Ω	set of all substrings over DNA alphabet Σ of length k
R	set of input reads
L	average read length of input dataset
N_B	the number of seeds in seed dataset
N_R	the number of reads in input dataset
N_O	the number of output OTU
O	set of output OTU
V	set of embedding vectors
B	set of seeds
S	sparse pairwise distance matrix
T	number of clustering iterations in SparseHC

Chapter 1

Introduction

This thesis focuses on the development of efficient agglomerative hierarchical clustering (AHC) algorithms for processing large biological datasets generated by new deoxyribonucleic acid (DNA) sequencing technologies. This chapter provides some background information about next-generation sequencing (NGS) technologies, 16S rRNA metagenomic datasets and the operational taxonomic unit (OTU) clustering problem that motivates the development of new clustering methods. This chapter also defines the objectives of this research and describes the overall structure of the thesis.

1.1 Background

1.1.1 Next-generation sequencing

DNA sequencing is the process of determining the sequence of nucleotide residues of a DNA. Knowledge of DNA sequences is indispensable to study the molecular structure and functions of the genetic materials from living organisms. NGS [5, 10, 11] refers to new technologies for high-throughput DNA sequencing which produces millions of short DNA sequences (called *reads*) in a shorter time and at lower cost compared to traditional Sanger-based sequencing technologies [12].

Table 1.1: Next-generation sequencing versus Sanger sequencing based on the guide to next-generation sequencers by Glenn [5]

Sequencing platform	**Chemistry for sequencing**	**Read length (in bps)**	**Reads per run**	**Throughput per run**	**Runtime**	**Cost per run**	**Cost per MB**
ABI 3730xI	Sanger method	650	96	0.06MB	2 hrs	$145	$2,300
454 GS Junior	Pyrosequencing	400	0.1M	50MB	10 hrs	$1,100	$22
454 FLX Titaninum	Pyrosequencing	400	01M	0.4GB	10 hrs	$6,200	$12
454 GS FLX+	Pyrosequencing	650	1.0M	0.65GB	20 hrs	$6,200	$7
Illumina MiSeq	Reversible terminators	2x150	4.0M	1.2GB	26 hrs	$1,150	$1
Illumina GA IIx	Reversible terminators	2x150	0.3B	96GB	14 days	$17,500	$0.19
ABI SOLiD 5500xl	Sequencing by ligation	2x50	1.4B	155GB	8 days	$10,500	$0.07
Illumina HiSeq 1000	Reversible terminators	2x100	1.5B	300GB	8.5 days	$10,200	$0.04
Illumina HiSeq 2000	Reversible terminators	2x100	3.0B	600GB	11.5 days	$23,500	$0.04

Table 1.2: Characteristics of next-generation sequencing datasets

Data	Data size	Read length
Pyrosequencing reads from 454/Roche sequencers e.g. GS FLX+	Raw: up to 10^6 reads Cleaned: up to 5×10^5 reads	Up to 700 base pairs per read
Paired-end reads from Illumina sequencers e.g. MiSeq and HiSeq	Raw: up to 6×10^9 reads Cleaned: up to 10^9 reads	Up to 100 base pairs per read

Table 1.3: Examples of targeted 454 16SrRNA pyrosequencing datasets

16S rRNA data	Statistics	Estimated number of OTUs
Twin gut [6]	$1.2 - 1.5 \times 10^6$ reads 130bp	4949 OTUs in 27 body sites of a twin pair
Children gut [7]	4.4×105 reads 260bp	1500-2000 OTUs in the gut of a child
Human skin [8]	3.5×10^5 reads 230bp	4742 OTUs in the hands of 51 people
Human wound [9]	2.2×10^5 reads 350bp	487 OTUs from 49 wounds

NGS carries great potential since the cost of NGS is decreasing considerably while the data generation capacity of NGS is increasing significantly, both with the rate multiple times faster than Moore's law [13]. Table 1.1 lists the most popular NGS sequencing platforms and compares them to a Sanger-based sequencing platform - the ABI 3700xl. Table 1.1 shows that NGS can generate a large amount of short reads in a shorter time at significantly lower costs.

There are a couple of existing and upcoming NGS platforms in the market. However, the most commonly used technologies with corresponding platforms are the pyrosequencing platforms from 454/Roche and the recently paired end sequencing platforms by Illumina [14]. The specifications of these two technologies are summarized in the Table 1.2. Examples of 16S rRNA NGS datasets are shown in Table 1.3. This research aims to support the analysis of these datasets in metagenomic studies. This thesis focuses on the problem of microbial diversity of a metagenome using 16S rRNA pyrosequencing datasets.

1.1.2 Metagenomics

Microorganisms or microbes are living things, generally less than 0.2mm and invisible to human eyes. There are four main groups of microbes: bacteria, archaea, microbial eukarya and viruses. Microbes play an important role to all living organisms on Earth including human , *e.g.*, (1) perform photosynthesis (2) provide plant with essential nutrition and fight off pathogens (3) enable human to extract calories from food, synthesize vitamins and amino acids, and defend infectious pathogens [15].

Due to the omnipresence of microorganisms on Earth and their essential roles to other living organisms, microbiology has been an important branch of life sciences. Traditionally, a microbe must be clonally cultured in a laboratory before its genome is sequenced and studied. There are two major drawbacks of traditional microbiology. Firstly, only a small proportion of microorganisms in nature can be culture in wet laboratories. Hence the existing knowledge of cultured microbes is

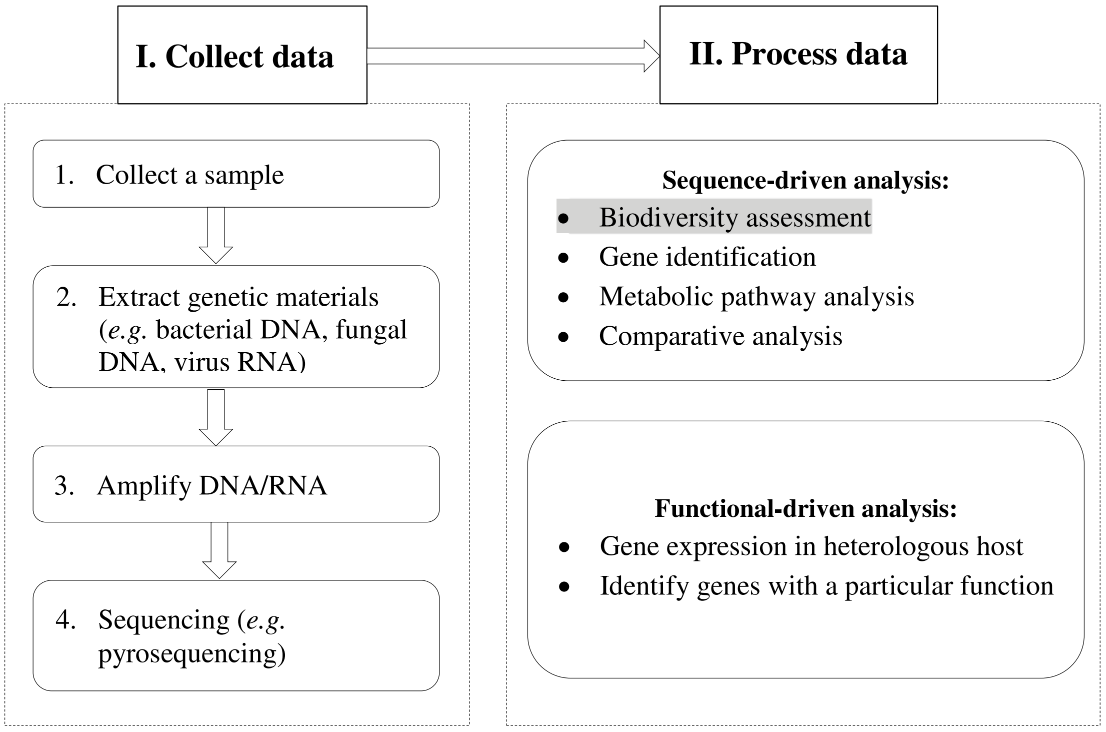

I. Collect data
II. Process data
1. Collect a sample
2. Extract genetic materials (*e.g.* bacterial DNA, fungal DNA, virus RNA)
3. Amplify DNA/RNA
4. Sequencing (*e.g.* pyrosequencing)
Sequence-driven analysis:
Biodiversity assessment
Gene identification
Metabolic pathway analysis
Comparative analysis
Functional-driven analysis:
Gene expression in heterologous host
Identify genes with a particular function

FIGURE 1.1: The metagenomic processing pipeline: biodiversity assessment is the focus of this research.

highly biased and does not represent the whole picture of the genomes of microorganisms [16, 17]. Secondly, microbes rarely live independently. They usually live in multi-species communities. Therefore, they often interact with each other and with their surrounding environment (including host organisms such as human). Cultured approaches fail to represent these interactions [18].

Recent advances in NGS technologies and computational methods have given rise to a new way to study the genetic materials of microbial communities called metagenomics. A metagenome is defined as a collection of genomes obtained from many different microorganisms in a microbial community. Subsequently, metagenomics is the analysis of the collective genetic materials obtained from a sample [19]. Figure 1.1 shows the common procedures for collecting and processing genetic materials in a metagenomic study.

Metagenomics can address the aforementioned drawbacks of traditional microbiology. A community (group) of micro-organism is sequenced together using new NGS sequencing platforms and the large amount of sequenced data is then analyzed in silico. With metagenomics, scientists can understand better the interaction of microorganisms within a community and the interaction of these microbes with their living environment.

A significant amount of genetic data generated by NGS platforms is unseen and unknown. Traditionally with Sanger sequencing, only one or a few species of interest are chosen and sequenced, then analysed. With the high throughput of NGS, $10^2 - 10^4$ species which all live in a community are sequenced and analysed together. Many species are unknown. Therefore, it is natural to start by clustering them into groups. This process in a metagenomic project is called biodiversity assessment.

1.1.3 16S rRNA datasets

The genetic material most commonly used to analyze the biodiversity of a metagenome is the 16S rRNA gene. RNA molecules are responsible for protein synthesis. They are essential and present in all organisms. The gene that encodes RNA is stored

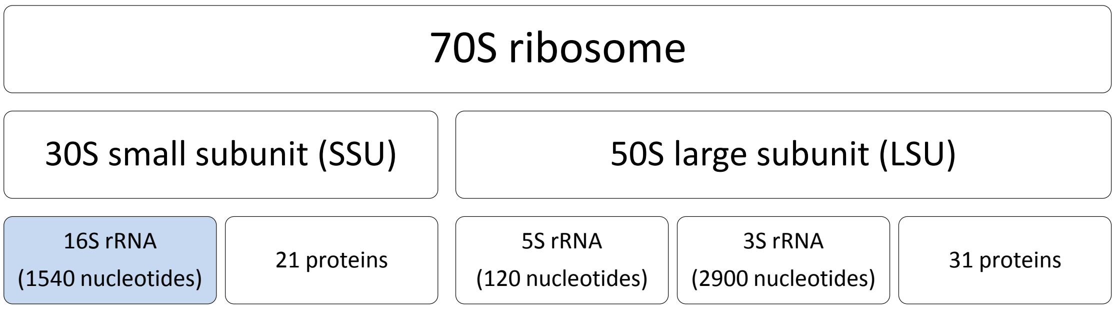


FIGURE 1.2: The structure of a prokaryote ribosome: the 16S rRNA gene is the marker gene for biodiversity assessment.

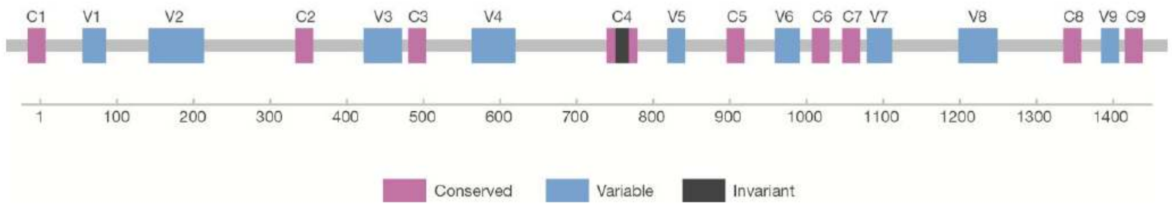


FIGURE 1.3: The schematic representation of the 16s rRNA gene in a canonical bacteria: hypervariable regions are in blue and conserved regions are in purple. The grey regions are invariant in all bacteria [1]

in the rDNA. Due to the essence of RNA, rDNA is the most conversed gene in all organisms [20]. Furthermore, rDNA contains highly conserved domains interspersed with variable regions [21] making it suitable for phylogenetic analysis by comparative methods to infer relatedness and differences among different organisms.

Figure 1.2 illustrates the structure of prokaryote ribosomes as introduced by [22] *S* stands for *Svedberg* which is a unit for sedimentation rate. The larger the sedimentation coefficient, the bigger the RNA subunit, *e.g.*, 50S subunit is larger than 30S subunit. Since a 5S RNA does not contain sufficient information for comparative analysis, 16S RNA and 23S RNA are often chosen. However due to the lack of primers and being larger, 23S RNA has lost favor to 16S RNA in phylogenetic analysis.

RNA is less stable in DNA in the cells and all the information carried in RNA can be found in the DNA. Therefore, rDNA is used for sequencing and analysis instead of RNA. The 16S rDNA strand is often used as a gene marker to identify microorganisms and assess the biodiversity of a microbial metagenome [23]. The full-length 16S rDNA sequence contains nine hypervariable regions interspersed with conversed regions as illustrated in Figure 1.3. For the purpose of species richness estimation, we often use one or more regions to identify the species that are present in a community. In this thesis, the terms ”rDNA dataset” and ”rRNA dataset” are used interchangeably.

1.1.4 Biodiversity measures

Estimating the biodiversity of a microbial community is important to infer the functionalities and the impact of that community on the hosts (*e.g.*, plants and animals including humans) or its surrounding environment (*e.g.*, soil, water).

According to [24], there are three types of biodiversity:

- *α-diversity* is a measure of biodiversity in an ecosystem. It refers to the number of species i.e. the species richness within an ecosystem. α-diversity is the intra-community diversity.
- *β-diversity* is a measure of biodiversity in an area of many ecosystems. It is assessed by the total number of species that are unique to each of the ecosystems in the area. β-diversity is the inter-community diversity.
- *γ-diversity* is a measure of biodiversity of a region. It is computed as the total species richness over that region, *i.e.*, $\gamma = \alpha \times \beta$

This project targets only the α-diversity - the species richness of a microbial community. Two important aspects that should be considered when measuring α-diversity are *species richness* and *relative abundance* [25]. Species richness refers to the number of species that coexist in a microbial community. Species abundance is calculated by the number of individuals per species and relative abundance measures the evenness with which the individuals are spread out among the species. The mathematical formula for computing species richness and species abundance of a community are listed in Appendix A.

1.2 Motivation

Next sequencing technologies such as 454 and Illumina platforms has generated a massive amount of genomic data. At the same time, modern computing hardware has quickly evolved from traditional single-core central processing units (CPUs) to multi-core CPUs and subsequently high performance computing (HPC) platforms such as general-purpose graphical processing units (GP-GPUs), compute clusters and the most recent trend of compute clouds. The purpose of this work is to build the bridge between data-processing and data-generating platforms by developing new algorithms that utilizes the power of HPC hardware to analyze high-throughput sequencing data in an effective manner. Specifically, this works focuses on building parallel and memory-efficient algorithms and tools using GP-GPUs to analyze genomic data produced by 454 platforms.

This research is motivated by a specific bioinformatic problem namely taxonomic profiling for biodiversity assessment of metagenomic samples. The purpose of this processing is to identify the components (and possibly structure) of a biological sample consisting of many different (and most often novel) microorganisms. The most popular approach is to extract the most representative genetic materials from these organisms and then to find the inherent groups in this genetic pool. In data mining terms, grouping can be done in a supervised manner by classification or in an unsupervised manner by clustering or in a semi-supervised manner by the combination of both classification and clustering. Due to the speculation that the proportion of novel species in each metagenomic sample could be significantly high (up to 90%), more focuses have been placed in the development of clustering methods.

1.3 Objectives

Motivated by the biodiversity assessment problem, this research focuses on building new hierarchical clustering algorithms for 16s rRNA genes produced by 454 sequencing platforms. Hierarchical clustering is chosen due to (1) its capability to construct biologically meaningful tree structure (2) its prospect to produce better grouping (3) its potential usefulness to many other bioinformatic applications and text mining problems and last but not least the lack of competent hierarchical clustering algorithms to process genomic datasets with more than 10000 sequences.

Traditional computational tools for Sanger sequencing datasets such as *dotur* + MUSCLE, *mothur* typically process $10^3 - 10^4$ reads. These tools do not work on next-generation sequencing datasets, each of which typically contains 10^5 reads and above, either because of the runtime limitation or memory constraints. Hence, there is a need for more effective and scalable programs to process these datasets.

The datasets generated by these two platforms should be considered separately due to the following reasons. Firstly, the difference in the read length implies difference processing e.g. shorter read length implies more duplicates, different amplification and sequencing techniques implies different types of sequencing errors. Secondly, the difference in the data size (pyrosequencing typically produces $10^5 - 10^6$ reads while Illumina sequencing typically outputs $10^8 - 10^9$ reads per run) implies the need for different algorithms. This thesis focuses on the analyses for pyrosequencing datasets.

The objective of this research is to provide scalable and efficient clustering techniques to tackle the biodiversity assessment problem *i.e* estimating the species richness and relative abundance of large-scale 16S rRNA metagenomic datasets.

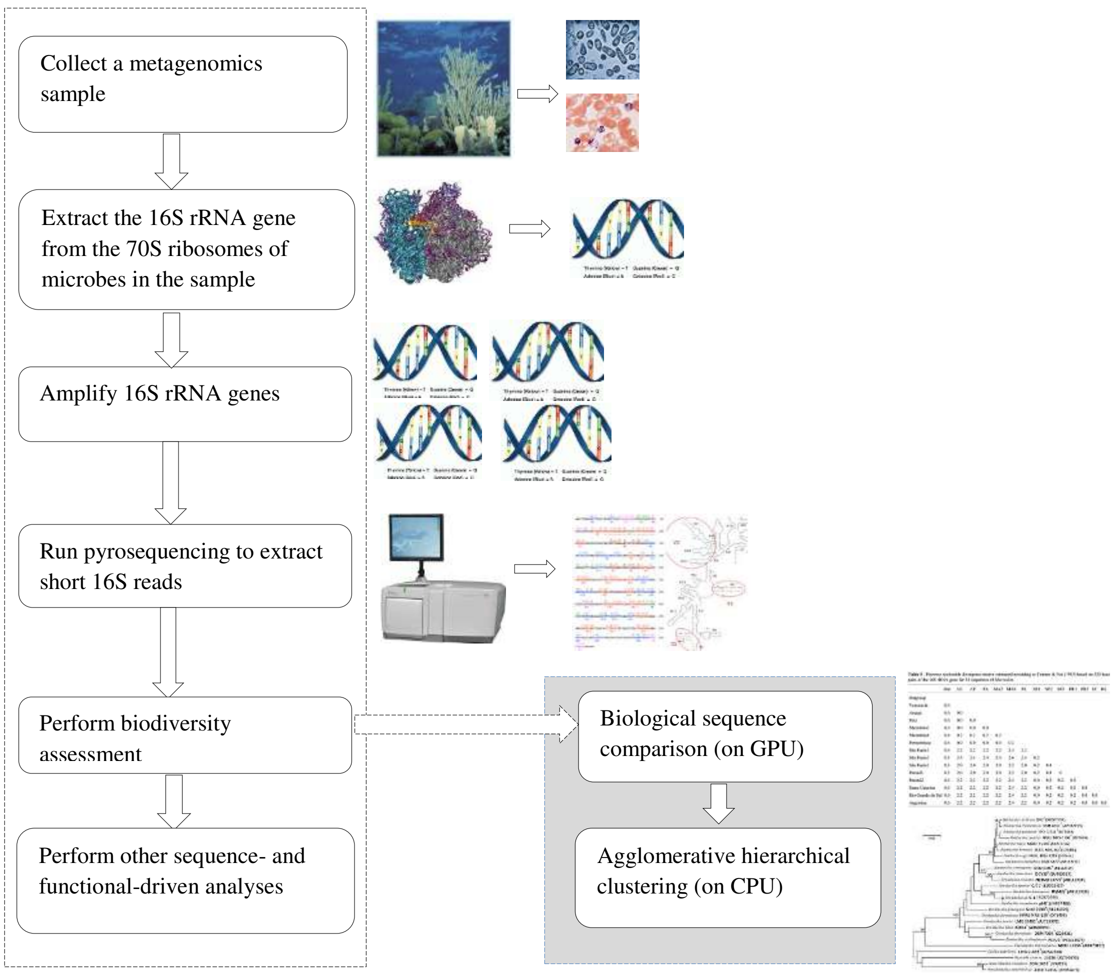

Collect a metagenomics sample
Extract the 16S rRNA gene from the 70S ribosomes of microbes in the sample
Amplify 16S rRNA genes
Run pyrosequencing to extract short 16S reads
Perform biodiversity assessment
Perform other sequence- and functional-driven analyses
Biological sequence comparison (on GPU)
Agglomerative hierarchical clustering (on CPU)

Figure 1.4: The biodiversity assessment flowchart: the shaded box shows the main focus of this research - biological sequence clustering.

Especially this research focuses on designing and implementing parallel algorithms on the GPUs as well as space-efficient hierarchical clustering algorithms. Figure 1.4 illustrates how biodiversity assessment is performed using 16S rRNA genes extracted from a metagenomic sample. Besides, it also highlights how biological sequence clustering is often used in a metagenomic study.

1.4 Contributions

The major contributions of this thesis are compute-efficient algorithms for distance matrix calculation on parallel architectures and a space-efficient hierarchical clustering algorithm. Specifically, parallel algorithms have been developed to compare biological sequence data using either the k-mer distance or the genetic distance or the Euclidean distance metric.

Each of these parallel algorithms is implemented either in CUDA to support execution on a general-purpose graphical processing unit (GPGPU) or in OpenMP to support execution on a multi-core CPU. While OpenMP implementation can be straightforward, the parallel implementation in CUDA is often a non-trivial task. To achieve maximum speedup on GPUs, the parallel implementations in CUDA often involve various tasks such as data compression, task division and usage of different memory types available on the GPUs.

Another important contribution of this research is the SparseHC algorithm, a general-purpose agglomerative hierarchical clustering algorithm to tackle the high space requirement of the traditional AHC approach. SparseHC can perform fast and memory-efficient clustering on both partial and full distance matrices. SparseHC supports three most commonly used linkage schemes: complete-linkage, single-linkage and average-linkage schemes.

In the domain of OTU clustering, this research contributes two processing pipelines namely the CRiSPy-Embed and the CRiSPy-CUDA pipeline. In general, the results of the cluster analysis are often hard to access since there is no universally accepted independent data to validate the clustering outcomes. Therefore, to evaluate the performance of these pipelines against existing tools, a comprehensive OTU evaluation framework using simulated and mock datasets has also been created. The evaluation process involves the creation of test datasets, the deliberate choice of sensible accuracy measurements as well as the interpretation and presentation of the evaluation results.

1.5 Thesis Structure

The remainder of the report is organized as follows. Chapter 2 presents the literature survey on general text clustering and OTU clustering. Chapter 3 and 4 provide the details of the compute- and space-efficient algorithms developed during the course of this research. Specifically, chapter 3 focuses in parallel computing and parallel algorithms for computing pairwise distance matrices. Chapter 4 concentrates on the memory-efficient clustering algorithm SparseHC and related topics. Subsequently, chapter 5 presents two new OTU binning pipelines called CRiSPy-Embed and CRiSPy-CUDA built by assembling the individual components discussed in 3 and 4. Chapter 6 describes an evaluation framework used to validate the accuracy and efficiency of each processing pipeline in comparison with other state-of-the-art OTU clustering pipelines. And finally, chapter 7 concludes and discusses the future work of this thesis.

Chapter 2

Literature Survey

Cluster analysis is a popular technique in data mining with numerous applications in research and commercialized applications. As discussed in 1, clustering is the main analysis used for estimating species richness of a 16S metagenomic datasets. This survey chapter provides a brief overview of cluster analysis with a focus on aspects related biological sequence clustering. Subsequently, it reports a detailed survey of existing state-of-the-artifacts techniques and algorithms in the domain of de novo OTU clustering - the focus of this thesis.

2.1 Cluster analysis

Clustering or cluster analysis refers to the process of grouping a set of data points in such a way that members in the same group are more similar to each other that those in different groups. Other similar terms to clustering in the literature include *automatic classification*, *numerical taxonomy*, *typological analysis* and *botryology*. The subtle differences are in the usage of the result: the resulting grouping patterns are of interest in data mining and the resulting discriminative power is of interest in automatic classification in machine learning.

Cluster analysis is an essential procedure used in statistical data analysis and exploratory data mining. Cluster analysis was first used in social sciences such as anthropology [26], psychology [27], personality psychology for trait theory classification [28]. Since then, this technique has been widely used in many fields of knowledge discovery including machine learning, pattern recognition, information retrieval, image analysis and bioinformatics.

Cluster analysis can be performed using various algorithms that differ notably in two aspects: their definition of a cluster and the efficacy of the algorithm. Popular cluster notions include group with high level of similarities amongst members, close-packed regions in the data space or certain statistical distributions. Therefore, clustering can be formulated as a multi-objective optimization problem. The choices of clustering algorithms and related parameter settings (e.g. the distance function, the expected number of clusters, the density threshold) depend on the characteristics of the input dataset and the intended use of the results.

2.1.1 General clustering approaches

Clustering techniques are often categorized into two groups: the hierarchical clustering group and the partitional clustering group, depending on the cluster structure they produce. Each approach has its own pros and cons and often different approaches yield different insights to the data. Table 2.1 lists out the most commonly used clustering algorithms available in the popular statistical analysis tool

TABLE 2.1: Most commonly used clustering algorithms and their implementations in the popular statistical language R

Algorithm	R package & func.	Category	Note
Hierarchical	**base** *hclust()* or	Agglomerative	Hierarchical clustering
	cluster *agnes()*		Agglomerative nesting
DIANA	**cluster** *diana()*	Divisive	Divisive analysis clustering
SOTA	**clValid** *sota()*	Divisive	Self-organizing tree algorithm
K-means	**base** *kmeans()*	Partitional	K-means clustering
PAM	**cluster** *pam()*	Partitional	Partition around medoids
CLARA	**cluster** *clara()*	Partitional	Clustering large applications
FANNY	**cluster** *fanny()*	Fuzzy	Fuzzy analysis clustering
SOM	**kohonen** *som()*	Neural network	Self organizing map
Model	**mclust** *Mclust()*	Statistical	Model-based clustering

R. There are three main categories presented in this table: the hierarchical clustering group (including two subgroups: agglomerative and divisive), the partitional clustering group and the others group (e.g. fuzzy clustering, self-organizing map).

Besides using output structure, clustering methods can also be distinguished by the distance metric and the size of the data. There are data points in the Euclidean space where a cluster can be summarized using a centroid versus data points in non-Euclidean spaces where other methods of summarization are required e.g. the longest common subsequence of all sequences in a cluster can be used to summarize that cluster. Another way to differentiate clustering algorithms is by the type of memory they use during the clustering processes. Algorithms targeting very large datasets often require external memory besides the main memory. Hence, these methods often need additional techniques to reduce the hard disk assessing time.

Figure 2.1 shows the taxonomy of the most commonly used clustering algorithms in the literature, incorporating all the various ways to differentiate clustering techniques as discussed above. This research focuses on biological sequence clustering. More specifically, this research involves the development of new clustering methods for very large biological sequence data in high-dimensional (probably non-Euclidean) space.

2.1.2 Biological sequence clustering

Clustering is used to analyze high-throughput genomic data by grouping genes or proteins with similar expression patterns or sharing similar biological pathways. This analysis is particular important when dealing with novel genomic data produced by newly-developed high-throughput sequencing platforms such as 454 and Illumina sequencers.

Clustering starts with *data points* which are objects in some *space*. For example, DNA clustering starts with DNA sequences which are objects in the *sequence space*. In evolutionary biology, the sequence space contains all possible biological sequences (gene or protein) [29]. Each letter represents a dimension in the sequence

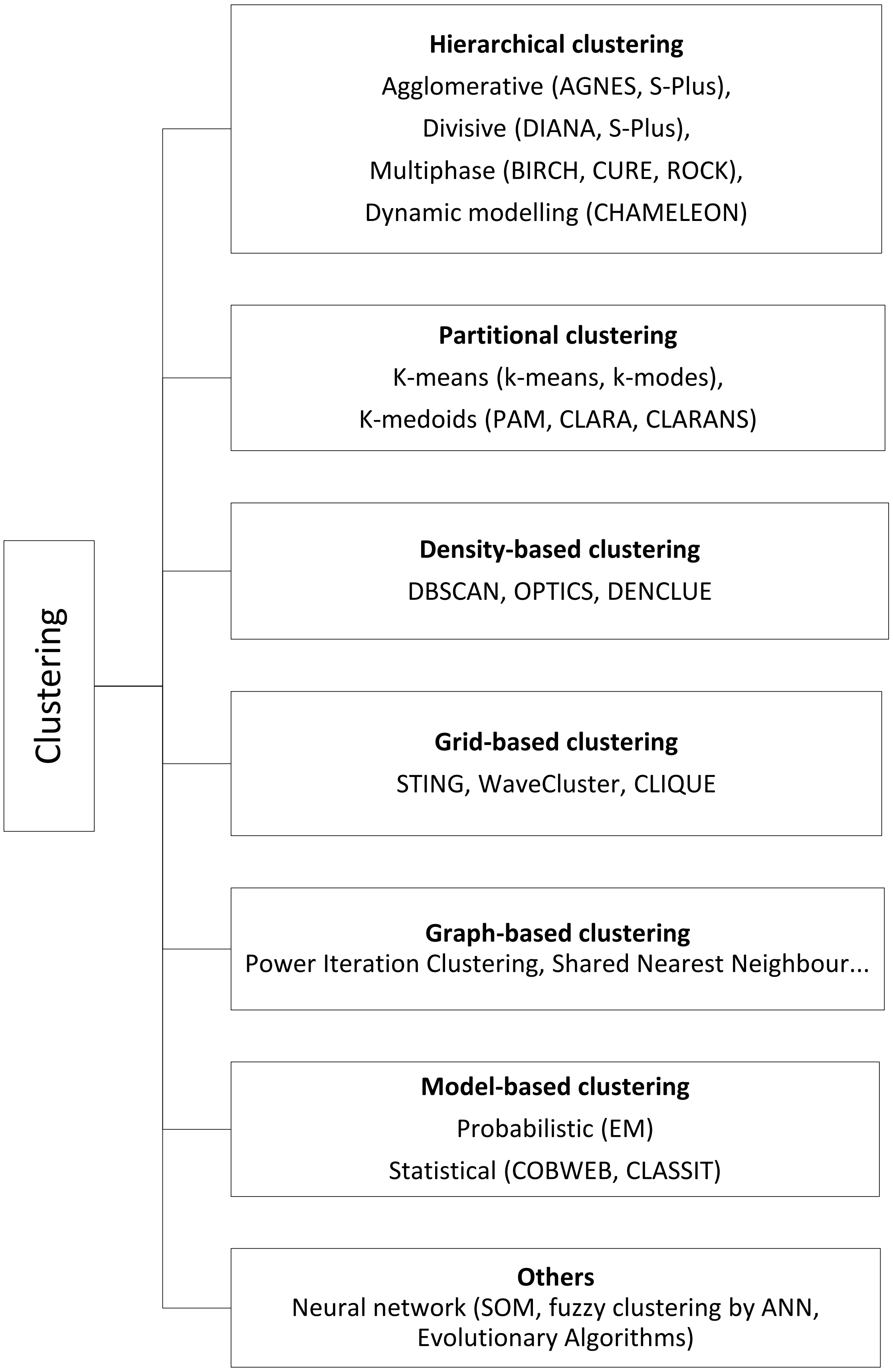

Clustering
Hierarchical clustering
Agglomerative (AGNES, S-Plus),
Divisive (DIANA, S-Plus),
Multiphase (BIRCH, CURE, ROCK),
Dynamic modelling (CHAMELEON)
Partitional clustering
K-means (k-means, k-modes),
K-medoids (PAM, CLARA, CLARANS)
Density-based clustering
DBSCAN, OPTICS, DENCLUE
Grid-based clustering
STING, WaveCluster, CLIQUE
Graph-based clustering
Power Iteration Clustering, Shared Nearest Neighbour...
Model-based clustering
Probabilistic (EM)
Statistical (COBWEB, CLASSIT)
Others
Neural network (SOM, fuzzy clustering by ANN,
Evolutionary Algorithms)

FIGURE 2.1: Taxonomy of clustering algorithms.

space [30]. Each nucleotide in a DNA sequence is represented by an axis with 4 possible positions, each of which corresponds to an item in the DNA alphabet $\Sigma = A, T, G, C$. A DNA sequence of length n can be viewed as a data point in a Σ^n space with 4^n possibilities. In other words, clustering is about grouping data points in a most sensible way possible for a particular purpose.

Popular distance measures in the Euclidean space include the Euclidean distance (L_2 norm), the Manhattan distance (L_1 norm) and the L_∞ distance (L_∞ norm). On the other hand, common non-Euclidean distance measures include the Jaccard distance, cosine distance, Hamming distance, edit distance and genetic distance. To be considered as distance metrics, these measures must satisfy the following three properties:

1. The non-negative property: $d(a, b) \geq 0$ and $d(a, b) = 0$ when $a = b$
2. The symmetric property: $d(a, b) = d(b, a)$
3. The triangular inequality property: $d(a, b) + d(b, c) > d(a, c)$ if a, b, c are different data points

The same distance metric concept apply to biological sequences which are not in the Euclidean space, but in the sequence space. For sequence comparison metrics, three aforementioned conditions are often relaxed. Most commonly used measures for DNA and protein sequences include the edit distance and the genetic distance.

2.2 OTU clustering

2.2.1 Background

Taxonomic profiling using 16S rRNA marker genes is an important step in a metagenomic analysis pipeline [31]. Examples include profiling of microbiomes in the human gut [32, 33] and in seawater [34]. Pyrosequencing is a next-generation sequencing technology capable of producing a large amount of sequencing data in a short period. This sequencing technology has been frequently used in metagenomic projects to sequence the hypervariable regions of the 16S rDNA marker gene for the purpose of biodiversity assessment [35].

Clustering is an important technique to perform taxonomic profiling of a microbial community by binning 16S rRNA amplicon reads into operational taxonomic units (OTUs). Existing OTU clustering tools for biodiversity assessment of 16S amplicon datasets can be grouped into three approaches: the closed-reference, the de novo and the open-reference approach (a hybrid approach of the former two approaches) [36, 37]. The closed-reference approach identifies a taxonomy composition of a metagenome by referencing the input dataset against a reference database of known micro-organisms. Although known microbes can be efficiently classified, this approach lacks the ability to identify novel species. According to the 'rare biosphere' theory [34, 38], many microbes have not been identified in existing reference databases. Therefore, grouping unknown microbes is an important task, which motivates the de novo approach.

The de novo approach performs biodiversity profiling by grouping the DNA sequences in an input dataset into clusters. The open-reference approach is a hybrid approach that first identifies known microbes by using the closed-reference and subsequently channeling the genetic materials of unknown organisms to a de novo algorithm for clustering. In this thesis, I focus on the de novo approach that can be used either independently or together with a closed-reference algorithm for open-reference OTU picking.

2.2.2 De novo OTU clustering

The de novo (or taxonomy-independent) method, does not need reference databases for grouping. This approach works based on an underlying assumption that a full-length 16S rDNA (in some cases, a region or a group of regions on the full-length sequence) is representative of a particular species. It performs sequence clustering and then groups input reads into clusters called operational taxonomic units (OTUs) based on a similarity threshold. Most existing studies and tools use heuristic threshold values of 97% and 95% for grouping at the species and at the genus level respectively. Depending on the structure of output groupings, existing algorithms for the de novo approach can be further grouped into two categories: greedy heuristic clustering (GHC) and agglomerative hierarchical clustering (AHC).

Greedy heuristic clustering (GHC) is a partitional clustering method that requires a specific distance level as input. Greedy heuristic clustering is typically computed by first choosing an input read as a seed. Each subsequent input read is then compared against the existing set of seeds. If this sequence is similar to one of the seeds within a predefined level (often at 97% sequence similarity), it will be added to the cluster represented by that seed. Otherwise, this sequence is considered as a new seed.

State-of-the-art tools in this category include UCLUST [39], USEARCH6, UPARSE [40], CD-HIT-OTU [41], DySC [42] and QIIME's *pick_otus* [37]. UCLUST chooses cluster seeds based on the percentage identity between a sequence and a seed. USEARCH and UPARSE perform a similar seed choice as UCLUST with additional filtering of clusters with low abundance i.e. small cluster sizes. CD-HIT-OTU groups similar sequences above 97% identity threshold and keeps the longest sequence as seeds. It also employs a fast heuristic to find high identify segments between sequences and hence avoid more costly full alignments. QIIME's *pick_otus* implements a number of popular reference-based and de novo OTU algorithms with the UCLUST algorithm chosen as the default method. DySC uses a dynamic readjustment of cluster seeds. All GHC methods have linear space and time complexity in terms of the number of input reads.

Agglomerative hierarchical clustering (AHC) is a bottom-up clustering technique that requires a distance matrix computed from the pairwise comparisons of all input reads. Notable tools in this category include *mothur* [43], ESPRIT [44] and ESPRIT-Tree [45].

ESPRIT employs the traditional hierarchical approach of first computing an alignment-based all-against-all distance matrix and then performs either complete-linkage or average-linkage clustering on that matrix. ESPRIT reduces computational complexity by generating only the lower part of a dendrogram. Although ESPRIT has a time and space complexity of $\mathcal{O}(N^2)$ as traditionally AHC algorithms, the empirical runtime and memory usage are significantly reduced by a factor proportional to the sparsity of the distance matrix used for generating the partial dendrogram.

The approach of *mothur* is similar to ESPRIT but instead of pairwise global alignment *mothur* uses multiple sequence alignment produced by a third-party alignment tool such as MUSCLE [46] to compute the pairwise distance matrix. Several studies have shown that using pairwise alignment produces better clustering outcomes than multiple sequence alignments [38, 47].

Different from ESPRIT and *mothur*, ESPRIT-Tree employs both greedy and hierarchical strategies. Instead of seeds, it uses “probabilistic sequences” to present a group of similar sequences and then applies a BIRCH-like [48] clustering method to build and refine a “pseudo-metric based partition tree” using these probabilistic sequences instead of the original input sequences. ESPRIT-Tree has quasilinear space and time complexity [45].

Overall, the AHC approach has quadratic or cubic time and space complexity due to the all-against-all pairwise read comparison. Meanwhile, the GHC implementations often has linear complexity, hence faster and more memory-efficient than the AHC approach. On the other hand, AHC tools have been shown to produce grouping results of higher quality than GHC tools [47]. Since the AHC approach is highly compute-intensive, it does not scale well for large datasets. For example, to cluster a dataset with N reads of length L, the time complexity to calculate the pairwise distance matrix is $O(N^2L^2)$ and the space complexity to perform clustering ranges from $O(N^2)$ to $O(N^3)$ [49].

Current OTU clustering tools use a default distance cutoff value of 0.03. This threshold value is chosen based on the presumption that the similarity between a pair of 16S rDNA short reads created from the same full-length 16S rDNA (hence from the same species) is higher than 97%. This assumption holds and hence is only applicable for datasets in which the pairwise distances between reads from the same species are less than 0.03 and the distances between reads from different species are larger than 0.03. However, under circumstances where this assumption does not hold, we prefer a more robust method to determine the cutoff threshold.

Recent OTU binning tools supporting dynamic cutoff thresholds are CROP [50] and M-pick [51]. CROP applies Bayesian clustering on input sequences which are described by Gaussian mixture models. Meanwhile, M-pick builds a graph from the input dataset and subsequently detects groups of edges in which the number of connections are "significantly higher than expected by chance". M-pick employs the concept of "modularity of a grouping" to indicate if a graph partitioning reveals the underlying community structure.

2.3 GPGPU in bioinformatics

Parallel computing has been heavily employed in the field of bioinformatics due to the heavy computation often occurs during a bioinformatic study. Especially, general-purpose processing using graphical processing units (GPGPU) has become more and more popular, especially amongst applications for sequence analysis and 3D molecular simulation.

For sequence analysis, GPUs are mainly used for sequence alignment for the purpose of sequence comparison for sequence database search. A notable GPU-based application is CUDASW++ for Smith-Waterman protein sequence global alignment [52–54]. For protein multiple sequence alignment we have MSA-CUDA [55]. For protein sequence BLAST search, two popular tools are CUDA-BLASTP [56] and GPU-BLASTP [57].

For error correction of next-generation sequencing data, we have CUSHAW for short read alignment [58–60]. We also have DecGPU [61] and Musket [62] for short read error correction [63]. In the context of 3D molecular simulation, we have applications such as molecular dynamics simulation [64] and 3D protein docking [65, 66].

Besides bioinformatics, GPGPU has also been used in other data-intensive applications including graph processing [67, 68] and deep learning for big data applications such as general-purpose unsupervised learning [69, 70], image processing and classification [71, 72].

Chapter 3

Parallel Distance Matrix Computation

As discussed in Chapter 2, the clustering analysis using the agglomerative hierarchical clustering approach consists of two stages: the distance matrix computation stage and the cluster hierarchy construction stage. The former stage is often a compute intensive procedure, hence in the context of this research, is accelerated using compute-efficient algorithms on parallel processors. Specifically, parallel algorithms in either Compute Unified Device Architecture (CUDA) language or Open Multi-Processing (OpenMP) are developed to execute this stage on a general-purpose graphical processing unit (GPGPU) or a multi-core central processing unit (CPU) respectively. This chapter details the background of parallel programming, describes three main parallel distance matrix computation algorithms developed during the course of this research and reports the acceleration achieved by these parallel algorithms compared to their sequential counterparts. The work described in this chapter has been published in [73] and [42].

3.1 Parallel computing

Parallel computing refers to the use of a parallel computer to reduce the processing time required to solve a computational problem. A parallel computer is a computer system with multiple processors. There are two main categories of parallel computers: multicomputers and centralized multiprocessors. As suggested by its name, a *multicomputer* is a parallel computer that consists of multiple computers connected via a network. A *centralized multiprocessor* (also known as a *symmetrical multiprocessor* or SMP) is a highly integrated parallel computer consisting of multiple CPUs sharing a single memory.

This section introduces two parallel programming techniques for SMPs: OpenMP programming for multicore CPUs and CUDA programming for GPGPUs, as well as a parallelization technique for multicomputers using the message passing interface (MPI). These technologies can also be combined together in one program to support heterogeneous systems. For examples, a program written in C using OpenMP and MPI libraries can run on a cluster of multicore CPUs. Similarly, a program written in C using CUDA and MPI can run on a cluster of GPGPUs. A program written in C using CUDA, MPI and OpenMP libraries can use all the parallel computation potentials of a cluster of both CPUs and GPUs.

There are three main types of parallelism: *data parallelism* where the same operation is applied to different data elements, *functional parallelism* where different operations are applied to different data elements and *pipelining* where a problem is divided into multiple computation stages and the stages from multiple problem instances are chained together into an assembly line.

Data dependency graph is a directed graph that is often used to identify the parallelism amongst subtasks of a computational problem. Each vertex represents a subtask and an edge from vertex u to vertex v denotes the dependency of subtask v on subtask u i.e. subtask u must finish before subtask v starts. If there is no edge between a pair of subtasks, they are independent of each other, hence can be executed concurrently.

3.1.1 OpenMP programming

OpenMP is a popular and easy-to-use application programming interface (API) used for parallel programming on multiprocessors i.e. a CPU with multiple cores. OpenMP can be used together with C, C++ or FORTRAN.

OpenMP uses the shared memory model for parallel computation. The shared memory model is used when multiple processors have access to the same memory resource so that the can synchronize and communicate with each other using shared variables. Figure 3.1 illustrates this concept.

OpenMP employs *fork/join* parallelism [74] in which there is a master thread running the sequential portion of the code and creating additional threads to execute the parallel sections (fork). At the end of a parallel portion, the child threads are suspended and the control returns to the master thread (join).

Two main components of OpenMP are *compiler directives* and *functions*. Examples of OpenMP compiler directives include: *parallel* (precedes the code block to be executed in parallel with multiple threads), *for* (precedes a for loop with independent iterations that can be divided among multiple parallel threads), *critical* (precedes a critical section - a code section that should be executed by only one thread at a time), etc. Examples of OpenMP functions include: *omp_get_num_procs* (returns the number of cores in a multiprocessor), *omp_set_num_threads* (set the

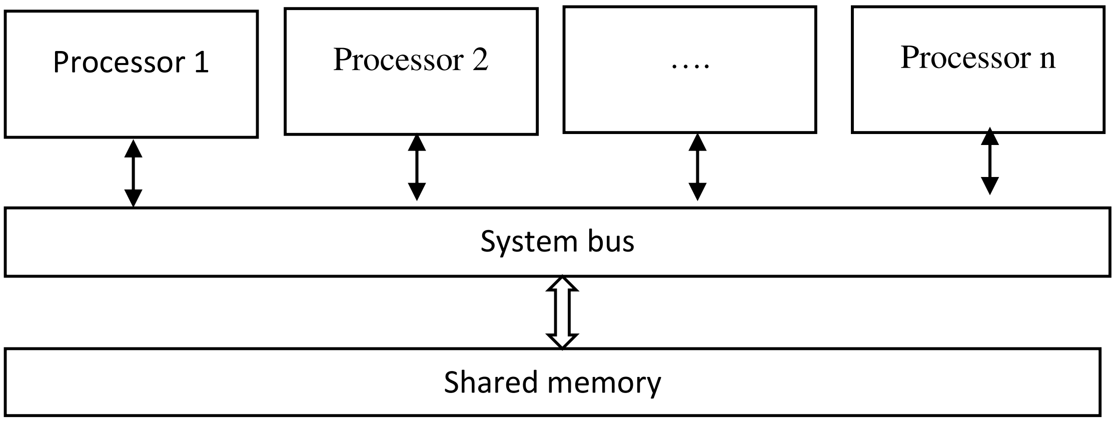


FIGURE 3.1: Shared memory model for parallel computation.

number of threads used to execute a code block), *omp_get_thread_num* (returns the thread ID), etc. Directives and functions allow easy transformation of a serial program into a parallel program one code block at a time (incremental parallelization).

OpenMP are often used in conjunction with MPI to program a cluster of multiprocessors. It can also be used in conjunction with CUDA to program a workstation with multiple GPU cards installed.

3.1.2 MPI programming

MPI is the most popular specification used by parallel programs to pass messages amongst computers in a cluster. In this work, I use the OpenMPI library [75] which implements the MPI specification for parallel programs implemented in C.

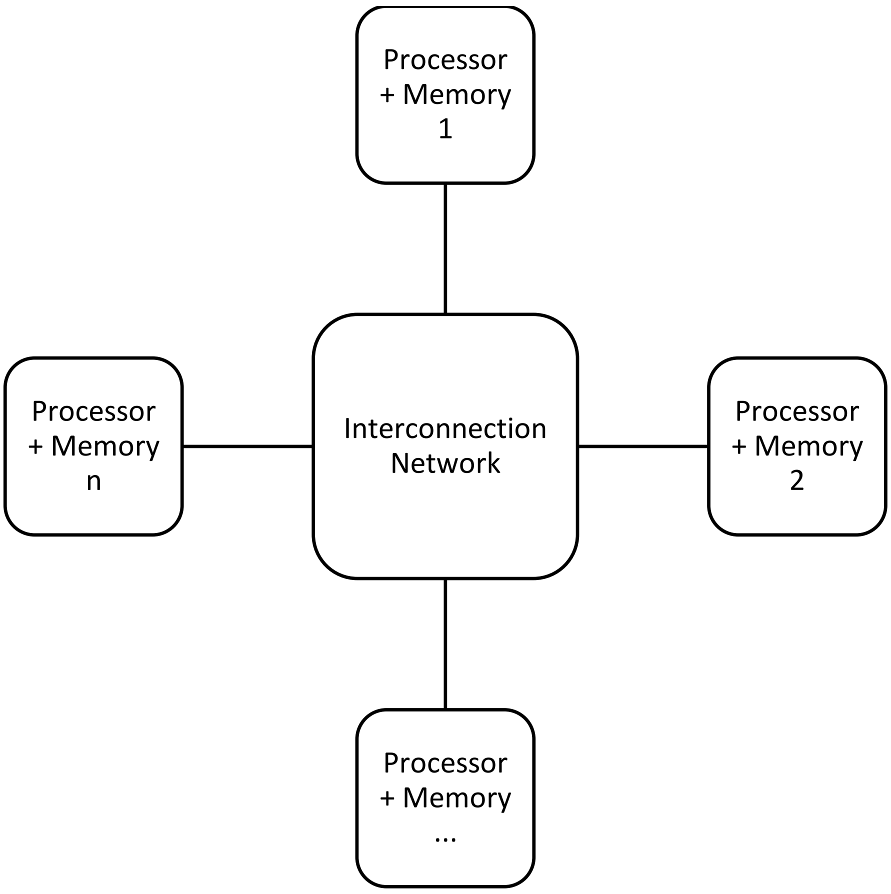


FIGURE 3.2: Message passing model for parallel computation.

The message passing parallel programming model assumes that the underlying hardware is a collection of independent processors, each with its own local memory. They communicate with each other by passing messages via an interconnection network instead of assessing a common shared memory as in the OpenMP model. Figure 3.2 illustrates this concept.

When running, a parallel program using MPI spawns multiple processes running on multiple processors. Since all processes executes the same program, each process is given a unique process ID which is used by the parallel program to distinguish them in order to assign different tasks to different processes. Examples of MPI functions include: *MPI_Init* and *MPI_Finalize* to initialize and shutdown MPI, *MPI_Comm_rank* to determine a process ID, *MPI_Comm_size* to find the number of processes running an MPI program.

3.1.3 CUDA programming

An NVIDIA graphical processing unit is created around a scalable programmable processor array consisting of a number of streaming processors. The number of streaming processors per device varies depending on the GPU type. Each streaming processor is in turn contains a number of scalar processors and a configurable on-chip shared memory. For the Fermi-based GPUs used in this project, each GPU has 16 streaming processors, each of which has 32 scalar processors. The on-chip memory on each streaming processor can be configured as 16 KB of PBSM with 48 KB of L1 cache or as 48 KB of PBSM with 16 KB of L1 cache. Figure 3.3 [2] illustrates the architecture of a Fermi NVIDIA GPU.

CUDA is a parallel programming language used to program an NVIDIA GPU. It extends general programming languages with a minimal set of abstractions for expressing parallelism. Similar to OpenMP, CUDA enables users to write parallel code for CUDA-enabled processors using familiar languages such as C, C++ and FORTRAN [76].

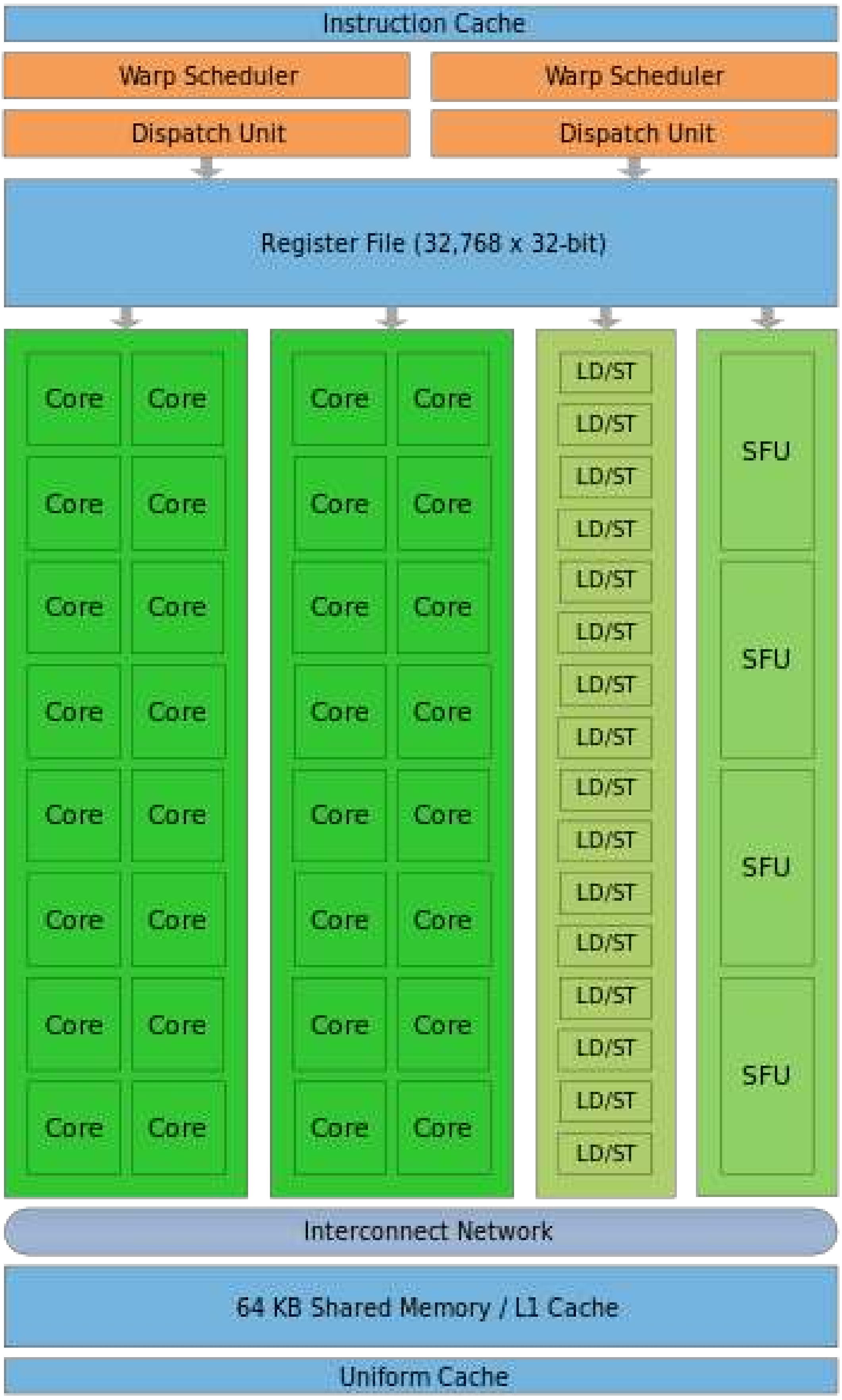
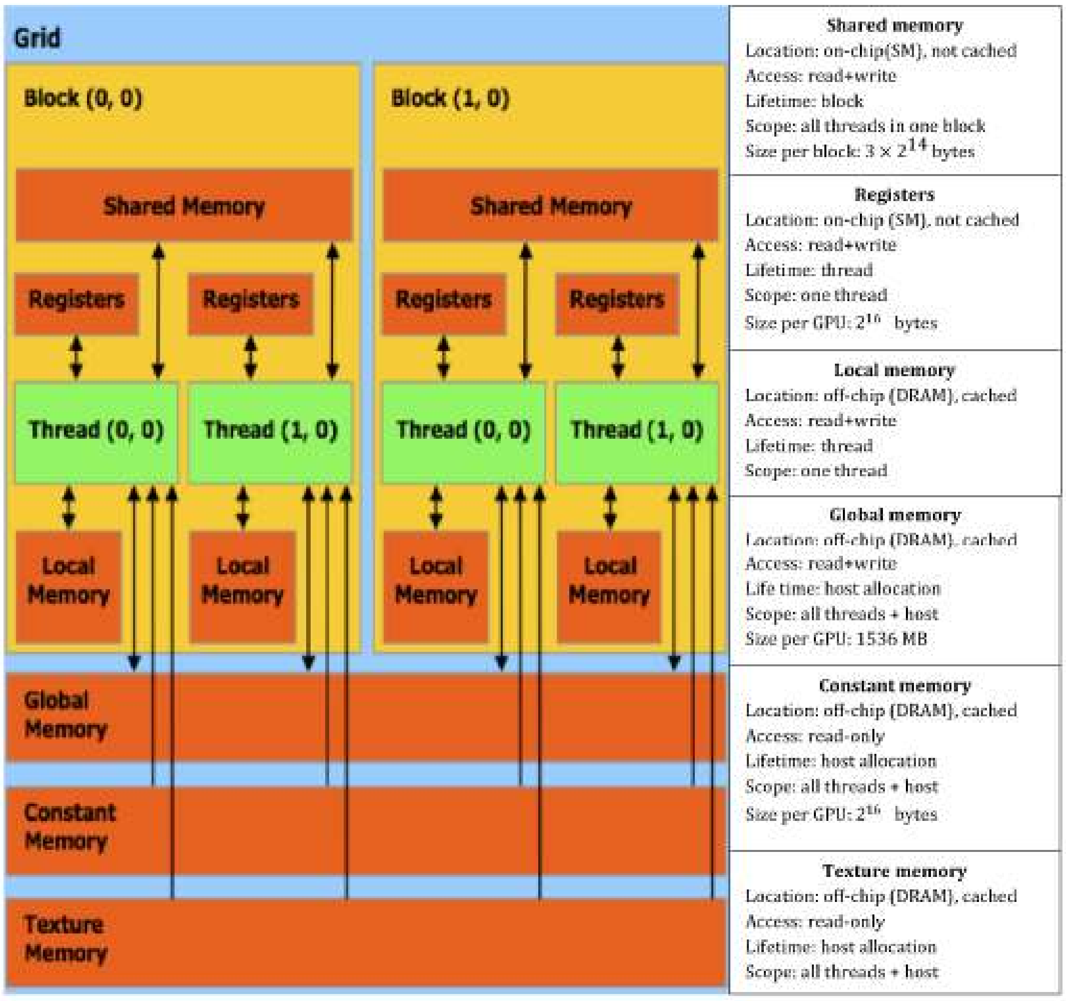


FIGURE 3.3: Fermi architecture [2]



Shared memory
Location: on-chip(SM), not cached
Access: read+write
Lifetime: block
Scope: all threads in one block
Size per block: 3×2^{14} bytes

Registers
Location: on-chip (SM), not cached
Access: read+write
Lifetime: thread
Scope: one thread
Size per GPU: 2^{16} bytes

Local memory
Location: off-chip (DRAM), cached
Access: read+write
Lifetime: thread
Scope: one thread

Global memory
Location: off-chip (DRAM), cached
Access: read+write
Life time: host allocation
Scope: all threads + host
Size per GPU: 1536 MB

Constant memory
Location: off-chip (DRAM), cached
Access: read-only
Lifetime: host allocation
Scope: all threads + host
Size per GPU: 2^{16} bytes

Texture memory
Location: off-chip (DRAM), cached
Access: read-only
Lifetime: host allocation
Scope: all threads + host

FIGURE 3.4: Thread hierarchy and memory hierarchy of a Fermi GPU: the GTX 580 based on the GPU memory architecture by Nickolls and Dally [3]

A CUDA program is comprised of two parts: a host program running one or more threads on a host CPU, and one or more parallel kernels which are executed on a GPU device [77]. A CUDA kernel is a serial function launched on a set of lightweight parallel threads. A group of lightweight concurrent threads is called a thread *block* and a group of thread blocks is called a *grid*. When executing a block, all threads in the block are split into small groups of 32 parallel threads, called *warps*, which are scheduled in a single-instruction multiple-threads (SIMT) fashion. Divergence of execution paths is allowed for threads in a warp, but multiprocessors realize full efficiency and performance when all threads in a warp take the same execution path.

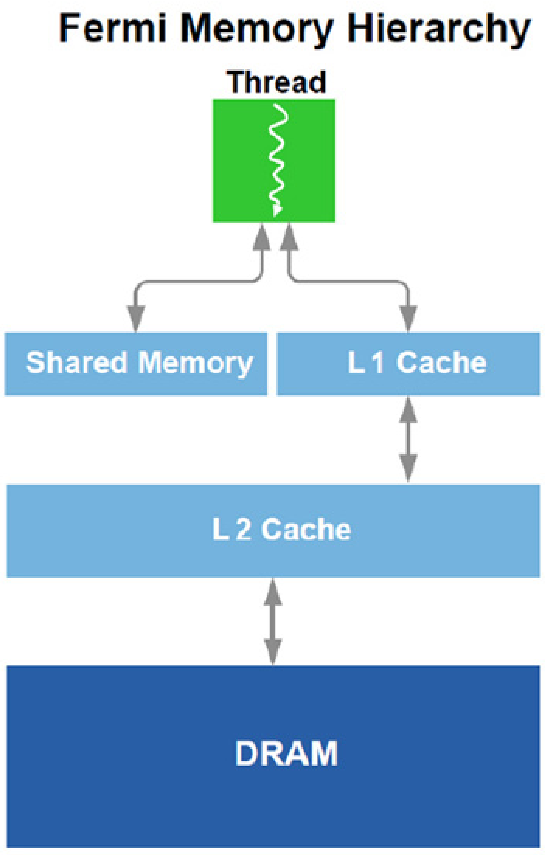


FIGURE 3.5: The memory hierarchy of a Fermi GPU [2].

Threads within a block can access a fast per-block shared memory (PBSM) and can synchronize through barriers. Threads within the same grid can communicate with each other through atomic operations on the global memory shared amongst all threads. The layered structure of the concurrent lightweight threads allows coarse-grained *data parallelism* and *task parallelism* at the block level and fine-grained *data parallelism* and *thread parallelism* parallelization at the thread level.

Figure 3.4 illustrates the thread hierarchy on a SM, which includes a grid of multiple blocks, each of which consists of many threads. It also shows the memory hierarchy on a standard Fermi-based GPU device - the GTX 580, which includes registers, shared, global, constant, shared memory as well as the characteristics of each memory type .

Figure 3.5 shows the memory hierarchy of a Fermi GPU card. A Fermi card has 48KB/16KB of L1 cache and 768KB (6 $\times$ 128KB) of L2 cache that serves all data accesses to GPU DRAM and system memory. L2 cache can be controlled through inline modifiers in Parallel Thread Execution (PTX) assembly statements. In this work, we focus on optimizing the L1 cache and texture cache, leaving the L2 cache optimization to the hardware.

3.2 Biological sequence distance comparison

In this research, we study and employ two alignment-free distance metrics: the Euclidean distance based on sequence embedding and the k-mer distance, together with an alignment-based distance metric: the genetic distance.

The genetic distance is chosen since it is the most sensible and widely-used distance for biological distance comparison. However, for large datasets the execution time for the pairwise genetic distance matrix can be significantly long albeit computed on GPUs. K-mer distance and Euclidean distance based on sequence embedding are chosen since they have been shown to be highly correlated with the genetic distance. On the other hand, they require significantly less execution time to compute. Therefore, these alignment-free distances are used to substitute the genetic distance in cases where accuracy can be compromised such as for the filtration step in CRiSPy-CUDA.

3.3 k-mer distance

3.3.1 Definition

Definition 1

For a text sequence, a k-mer (also known as k-tuple or k-shingle) is defined as a substring of length k found in that sequence. A k-mer tuple *bag* of a sequence S is set of all k-mers (including duplicates) found in S.

The k-mer distance between two text sequences R and S is calculated based on the k-tuple bags (i.e. duplicates are included) B_R and B_S created from the these two sequences:

$$d_k(R, S) = 1 - \frac{|B_R \cap B_S|}{\min(|B_R|, |B_S|)} \tag{3.1}$$

Example:

Given two DNA sequences: R = “ATGAT” and S = “ATTTAAT”.

R consists of 4 k-mers of length 2. The tuple bag of R is: $B_R = AT, TG, GA, AT$.

S consists of 6 k-mers of length 2. The tuple bag of S is: $B_S = AT, TT, TT, TA, AA, AT$.

The k-mer distance between $d_k(R, S) = 1 - \frac{2}{min(4,6)} = 1 - \frac{2}{4} = 0.5$

Definition 2

Here is another formulation of k-mer distance that is useful when k is small, particularly for long sequences. Given two sequences R and S of length l_R and l_S and a value $k > 0$, their k-mer distance is calculated using the following formula:

$$d_k(R, S) = 1 - \frac{\sum_{p=1}^{|\Omega|} \min(n_R[p], n_S[p])}{\min(l_R, l_S) - k + 1} \tag{3.2}$$

The set Ω contains all substrings over the alphabet Σ of length k. The substrings in Ω are enumerated in lexicographically sorted order. $n_R(p)$ and $n_S(p)$ represent how frequent the substring with index p appear in sequence R and sequence S respectively.

Using this second definition, we can represent a sequence of an arbitrary length by a vector of fixed length $|\Omega| = |\Sigma|^k$ as illustrated in the example below.

Example:

For DNA sequences $\Sigma = \{A, T, G, C\}$.

If $k = 2$, $\Omega = \{AA, AC, AG, AT, CA, CC, CG, CT, GA, GC, GG, GT, TA, TC, TG, TT\}$.

Given two aforementioned DNA sequences: R = "ATGAT" and S = "ATTTAAT".

$n_R = [0, 0, 0, 2, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0]$

$n_S = [1, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2]$

$d_k(R, S) = 1 - \frac{2}{min(5,7)-2+1} = 1 - \frac{2}{4} = 0.5$

Relationship with Jaccard similarity

The concept of k-mer distance between two sequences is closely related to the concept of Jaccard similarity of sets.

The Jaccard similarity (also known as the Jaccard index or the Jaccard similarity coefficient) of two sets A and B is defined as:$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$

The Jaccard distance, which measures the dissimilarity between two sets, is complementary to the Jaccard index: $d_J(A, B) = 1 - J(A, B)$

3.3.2 Sorting-based k-mer distance calculation

Algorithm 1 Sorting-based k-mer distance calculation for two reads R_1 and R_2 of length l_1, l_2

```
$count \leftarrow 0$
$i \leftarrow 0$
$j \leftarrow 0$
while $i < L_1 - k + 1$ and $j < L_2 - k + 1$ do
  if $V_1(i) < V_2(j)$ then
    $i \leftarrow i + 1$
  else if $V_1(i) > V_2(j)$ then
    $j \leftarrow j + 1$
  else
    $count \leftarrow count + 1$
    $i \leftarrow i + 1$
    $j \leftarrow j + 1$
  end if
end while
$distance \leftarrow count/(\min(L_1, L_2) - k + 1)$
```

This subsection introduces a sorting-based algorithm for calculating pairwise k-mer distances. This algorithm has been designed based on the first definition of k-mer distance as discussed above.

In the first step, we compute a value array V_i for each input read R_i. V_i contains all substrings of length k in the sequence R_i. The substrings in V_i are sorted in lexicographical order. To compute the distance between two reads, we scan their corresponding value arrays in ascending order and in the meantime keep track of the indices of these arrays. In each step, we compare two elements, each element from one array. If the two elements are the same, we increment the values of two indices and also the counter "count". If the two elements are different, only the index pointing to the array of the smaller value is increased by one. The computation stops when the end of either array is reached. At that time, the value "count" contains the number of similar substrings that the two input reads have in common.

The sorting-based algorithm is illustrated in Algorithm 1. It requires time and space $O(l)$ for two reads of average length l. This k-mer distance computation algorithm has been published in [73].

In CRiSPy-CUDA (Section 5.2), before computing the genetic distance matrix, in the filtration step we need to compute the pairwise distance matrix using the k-mer distance. The aforementioned sorting-based algorithm helps to speed up the pairwise matrix calculation since (1) the decomposition of the original biological sequences into k-mers is only done once (2) the string comparison operations of k-mers are faster since each k-mers has been converted into a value (3) the number of comparisons performed for a sequence pair is fewer thanks to the strategy of sorting the value array beforehand. The next two subsections present the parallel computation of the k-mer pairwise distance matrix using OpenMP and CUDA.

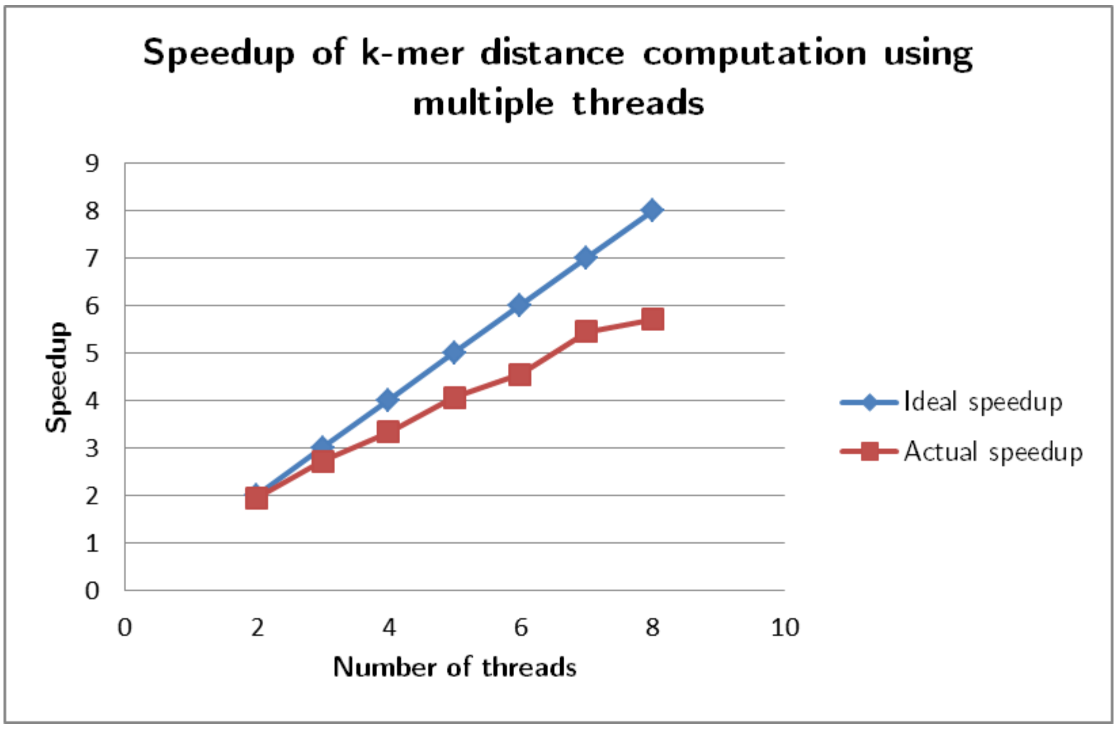


FIGURE 3.6: The speedup of the parallel k-mer distance computation running on a multicore CPU with respect to the number of threads

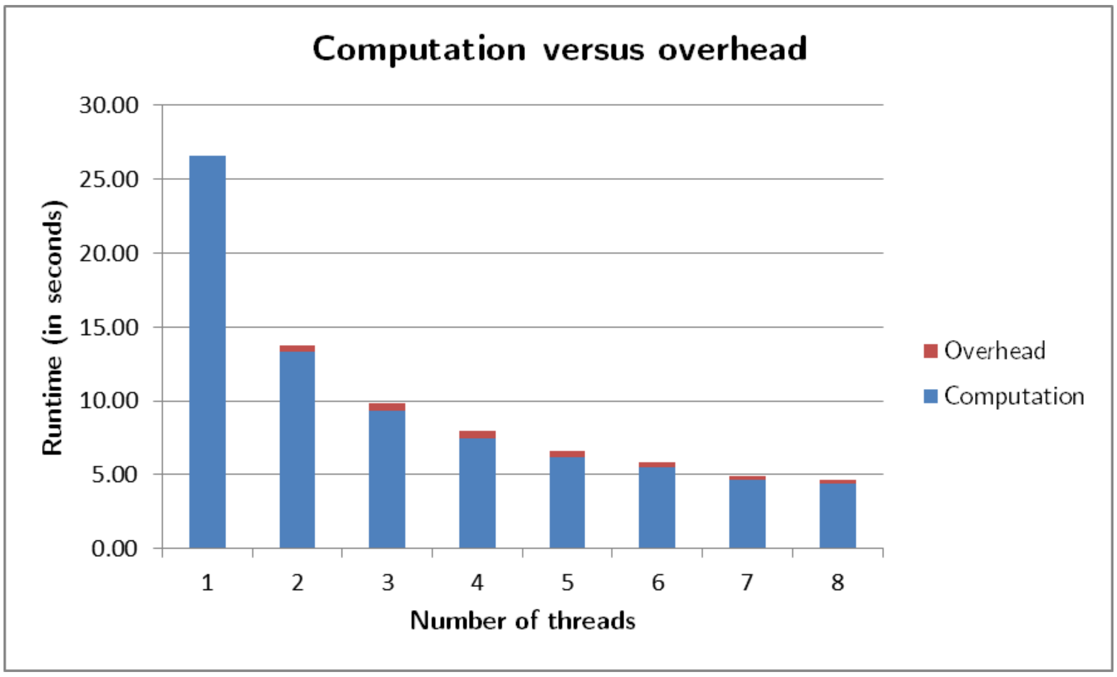


FIGURE 3.7: The overhead of the parallel OpenMP program for computing k-mer distances

3.3.3 OpenMP implementation

Figure 3.6 shows the speedup achieved by the parallel k-mer distance computation running on a multi-core CPU. Each thread runs on a processor. The speedup is the ratio between the serial runtime (i.e. the execution time using 1 thread) and the parallel runtime (i.e. the execution time using 2 or more threads) [74]. The actual speedup is less than the theoretical speedup due to the part of the code that writes the output to file that does not run in parallel.

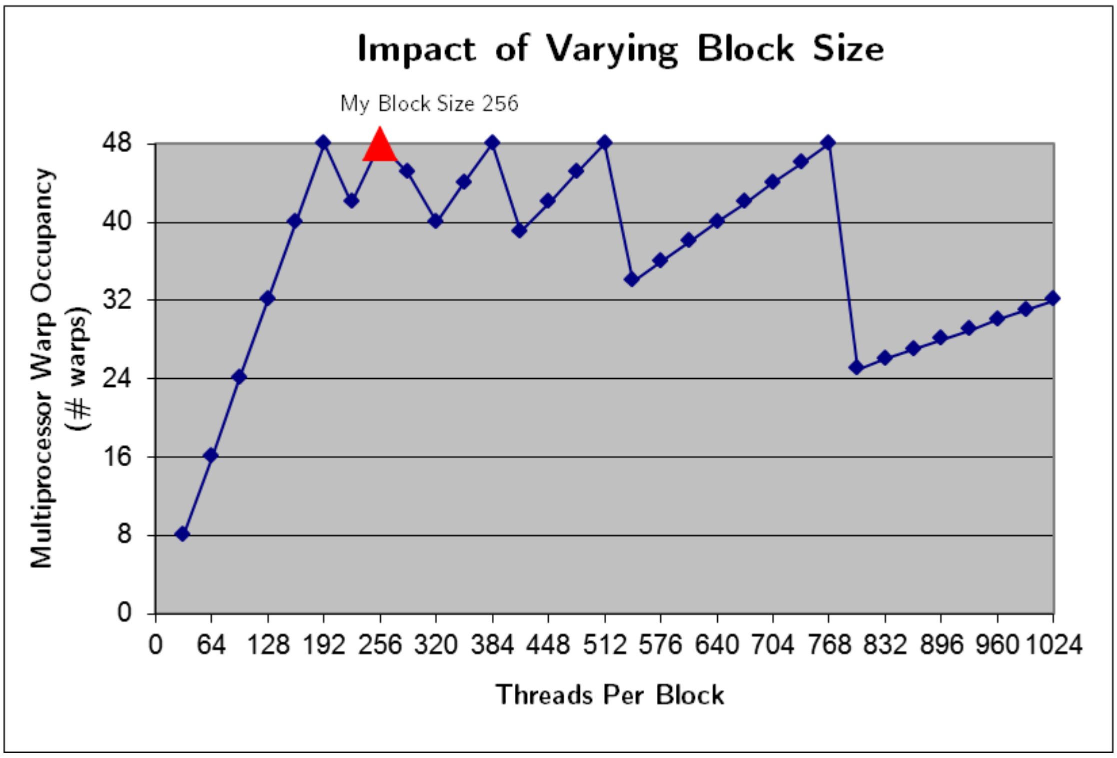
Figure 3.7 shows the overhead of the OpenMP program as the number of threads increases. Using more threads often reduces computation time but increases communication time. Because there is little communication between independent parallel tasks, we can observe that the overhead is negligible making the parallel program very efficient.

The efficiency is computed using the following formula:

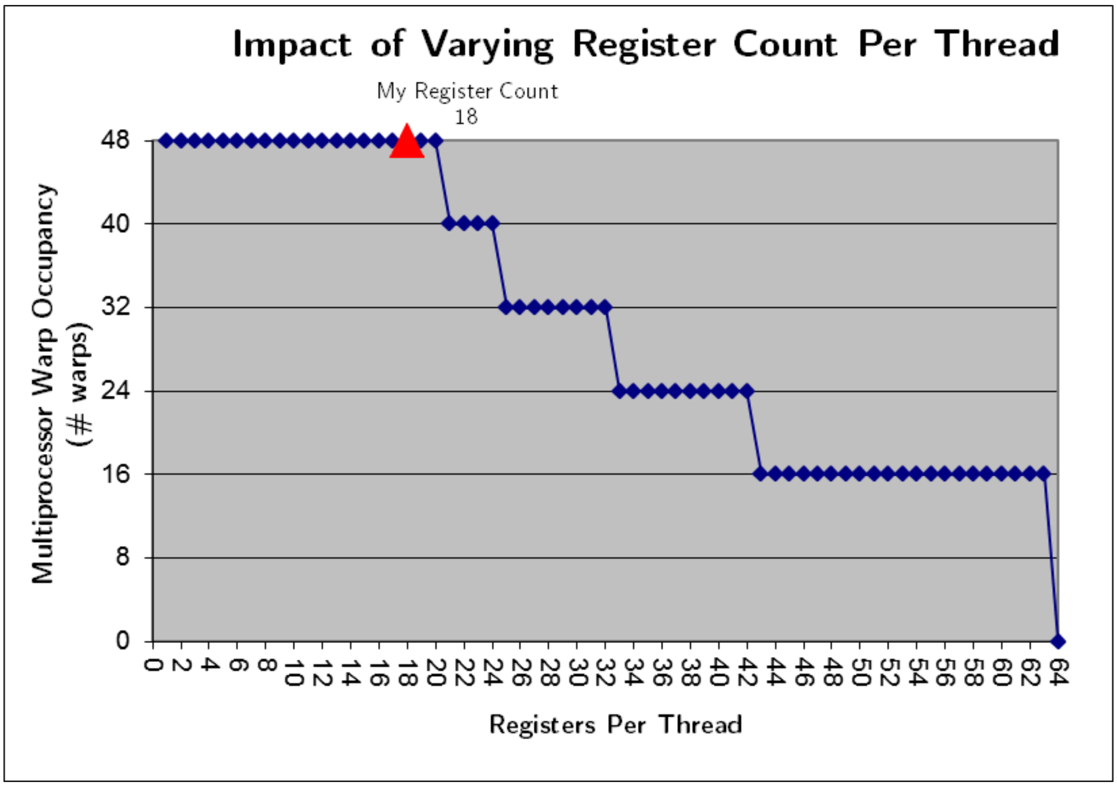
$$Efficiency = \frac{Serialexecutiontime}{Numberofthreads \times Parallelexecutiontime} \tag{3.3}$$

3.3.4 CUDA implementation

The massively parallel CUDA implementation first pre-computes all value arrays on the CPU in parallel using OpenMP. They are then transferred to read-only CUDA texture memory. Each pairwise distance k-mer computation is independent of each other and thus each CUDA thread can calculate a pairwise distance in parallel. To optimize memory accesses within a CUDA thread block, one value array is stored in shared memory. Each thread within the block then fetches value arrays from texture memory and compares the fetched array with the value array stored in shared memory in parallel. Finally, the calculated k-mer distance values are transferred back to CPU. The pair indices with k-mer distance smaller than a given threshold value are kept in a sparse matrix index for the subsequent processing stage.



(a) Block size



(b) Register per thread

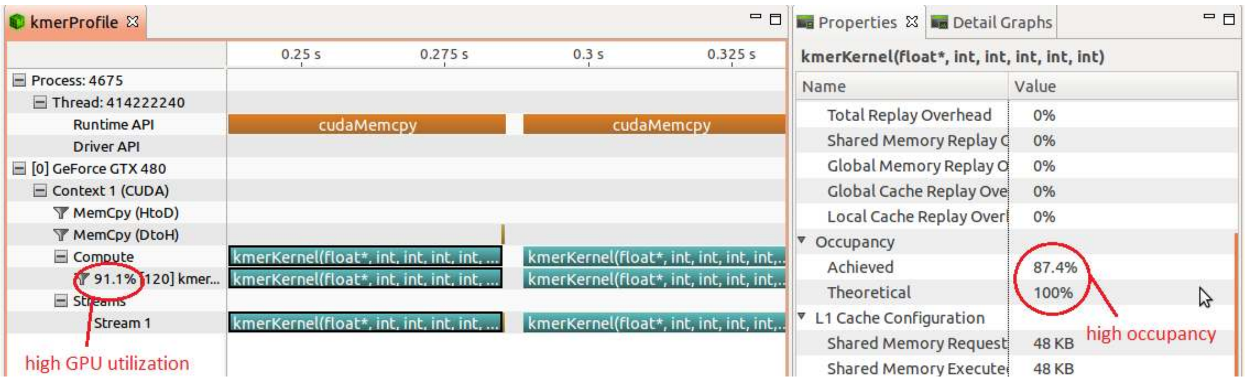
Figure 3.8: Choosing the block size and the register count per thread in order to maximize the multiprocessor warp occupancy. The red triangles mark the chosen settings for the CUDA program to compute k-mer distances: 256 threads per block and 18 registers per thread.

Figure 3.8 shows the result from the CUDA occupancy calculator for choosing the optimal block size and the register count per thread for the targeted GPU in order to achieve maximum multiprocessor warp occupancy. To optimize the performance of the CUDA parallel code, I use the NVIDIA Visual Profiler to benchmark the occupancy and GPU utilization. Figure 3.9 shows the profiling result for the k-mer distance computation module. The parallel k-mer program achieves a practical occupancy value of 85.8% over the theoretical occupancy of 100%. It also has a high utilization level: only 1.6% runtime is spent for memory transfer while 91.1% execution time is used for calculation.

3.4 Euclidean distance

3.4.1 Definition

The Euclidean distance is perhaps the most commonly used distance measure between data points in the Euclidean space. In this work, we use the Euclidean distance to measure the distances between a pair of sequence vector objects. A sequence vector is used to represent a biological sequence in the Euclidean space. A DNA sequence is transformed into a sequence vector via a process called sequence embedding which is described in details in the section about CRiSPy-Embed of Chapter 6.

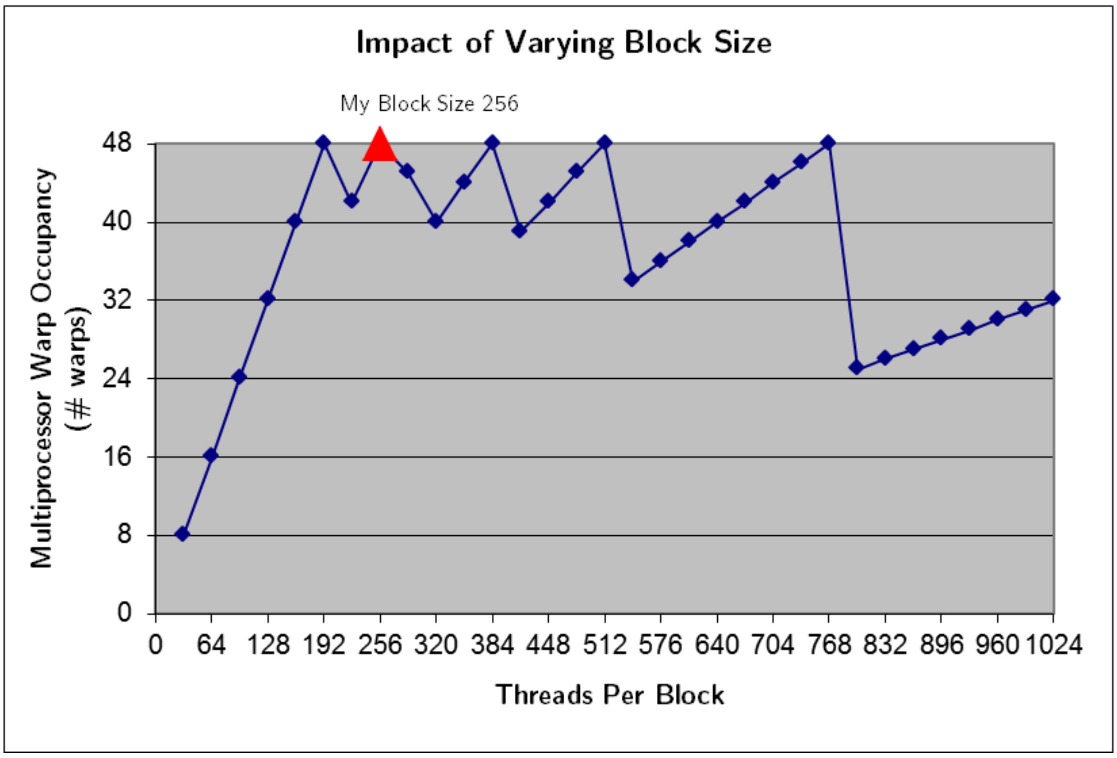


Process: 4675
Thread: 414222240
Runtime API
Driver API
[0] GeForce GTX 480
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
91.1% [120] kmer...
Streams
Stream 1

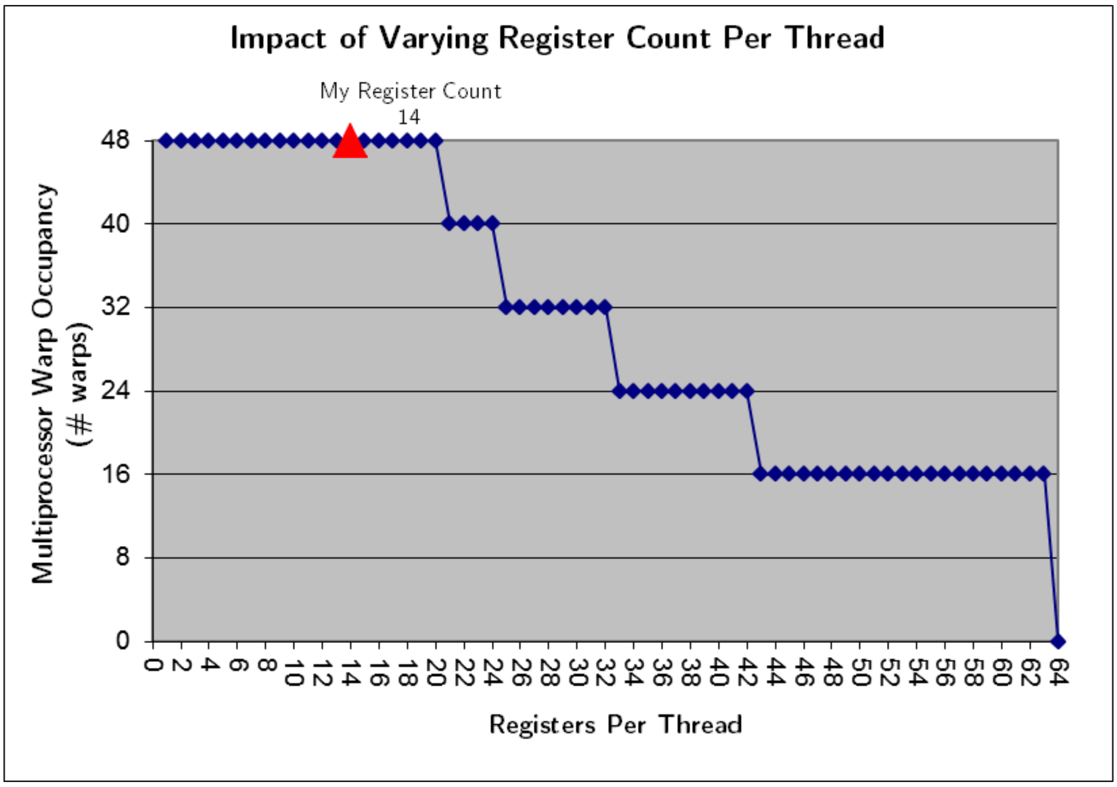
kmerKernel(float*, int, int, int, int, int)

Name	Value
Total Replay Overhead	0%
Shared Memory Replay O	0%
Global Memory Replay O	0%
Global Cache Replay Ove	0%
Local Cache Replay Over	0%
▾ Occupancy	
Achieved	87.4%
Theoretical	100%
▾ L1 Cache Configuration	
Shared Memory Request	48 KB
Shared Memory Execute	48 KB

FIGURE 3.9: Profiling the CUDA program for computing k-mer distance running on an NVIDIA GTX480 using the NVIDIA Visual Profiler. The parallel program achieves a high occupancy value of 87.4%.



(a) Block size



(b) Register per thread

FIGURE 3.10: Choosing the block size and the register count per thread in order to maximize the multiprocessor warp occupancy. The red triangles mark the chosen settings for the CUDA program to compute Euclidean distances: 256 threads per block and 14 registers per thread.

After transform input reads into sequence vectors using the sequence embedding procedure in CRiSPy-Embed, the normalized Euclidean distance $d(i, j)$ between a pair of sequence vectors V_i, V_j is computed as follows:

$$d(V_i, V_j) = \sqrt{\frac{\sum_{m=1}^{M}(V_{im} - V_{jm})^2}{M}} \tag{3.4}$$

Similar to the standard Euclidean distance, the normalized distance is a metric on the set $\Re^M$. Therefore, the normalized Euclidean distance also complies to the three properties of a metric, specifically the symmetry, the non-negativity, and the triangle inequality property. The triangle inequality property of the Euclidean metric allows partial Euclidean distance matrices to be effectively clustered by SparseHC used in CRiSPy-Embed.

3.4.2 Parallel implementations

The Euclidean distance computation is parallelized using the OpenMP (Open Multi-Processing) on multi-core CPUs and CUDA (Compute Unified Device Architecture) programming language on GPUs. Since the matrix is large and cannot fit into the GPU memory, I divide the triangular matrix into small sub-matrices and process each sub-matrix at a time. I use the pinned host memory to ensure high memory transfer throughput between host and device. The effective throughput is 5.94 Giga Bytes Per Seconds (GBPS) compared to maximum possible throughput of 8GBPS. Figure 3.10 shows the block size and the register count per thread obtained by profiling the Euclidean distance computation program using the CUDA occupancy calculator.

Besides, CUDA asynchronous streams are used to allow concurrent data transfer and kernel execution. Therefore, the Euclidean module achieves 97% of GPU device utilization. The computation on a Fermi-based GPU (GeForce GTX 580) is 4.5x - 5.5x faster than the computation on a quad-core CPU (Intel Xeon W3540) using four threads, and 13x - 15x faster than the single-threaded CPU computation.

	-	A	T	G	A	T
-	U L D M 0 0 0 0 ML AL 0 0	U L D M 0 0 0 0 ML AL 0 1	U L D M 0 0 0 0 ML AL 0 2	U L D M 0 0 0 0 ML AL 0 3	U L D M 0 0 0 0 ML AL 0 4	U L D M 0 0 0 0 ML AL 0 5
A	U L D M 0 0 0 0 ML AL 0 1	U L D M 0 0 1 5 ML AL 0 1	U L D M 0 1 0 0 ML AL 1 2	U L D M 1 0 0 0 ML AL 1 4	U L D M 0 0 1 5 ML AL 0 4	U L D M 0 1 0 0 ML AL 1 5
T	U L D M 0 0 0 0 ML AL 0 2	U L D M 0 1 0 0 ML AL 1 3	U L D M 0 0 1 10 ML AL 0 2	U L D M 0 1 0 5 ML AL 1 3	U L D M 1 0 0 0 ML AL 1 5	U L D M 0 0 1 10 ML AL 0 5
T	U L D M 0 0 0 0 ML AL 0 3	U L D M 0 1 0 0 ML AL 1 4	U L D M 0 0 1 5 ML AL 1 4	U L D M 0 0 1 6 ML AL 1 3	U L D M 0 0 1 1 ML AL 2 4	U L D M 0 0 1 5 ML AL 1 6
A	U L D M 0 0 0 0 ML AL 0 4	U L D M 0 0 1 5 ML AL 0 4	U L D M 0 1 0 0 ML AL 1 5	U L D M 0 0 1 1 ML AL 2 5	U L D M 0 0 1 11 ML AL 1 4	U L D M 0 1 0 6 ML AL 2 5
A	U L D M 0 0 0 0 ML AL 0 5	U L D M 0 0 1 5 ML AL 0 5	U L D M 0 0 1 1 ML AL 1 5	U L D M 0 0 1 -4 ML AL 2 6	U L D M 0 0 1 6 ML AL 2 6	U L D M 0 0 1 7 ML AL 2 5
T	U L D M 0 0 0 0 ML AL 0 6	U L D M 0 1 0 0 ML AL 1 7	U L D M 0 0 1 10 ML AL 0 6	U L D M 0 1 0 5 ML AL 1 7	U L D M 1 0 0 1 ML AL 3 7	U L D M 0 0 1 11 ML AL 2 7

“

Figure 3.11: Scoring matrices for aligning two input sequences ATGAT and ATTAAT using the new linear memory formula with the scoring scheme: match score is 5, mismatch score is -4, gap penalties are -10 and -5 for gap opening and gap extension respectively.

”

3.5 Genetic distance

3.5.1 Definition

In order to run as many threads as possible on GPUs, the amount of memory used in each kernel to compute an alignment of two reads needs to be optimized. As a result, a linear memory formula for semi-global alignment with affine gap penalty as derived from the standard formula of the Needleman-Wunsch dynamic programming (DP) algorithm with 3 scoring matrix: horizontal, vertical and diagonal [78].

Three standard scoring matrices M, V, H are replaced with one scoring matrix M and three mutually exclusive binary DP matrices U, L, D to store immediate traceback pointers. This replacement results in little effect on the alignment outcomes though significantly reduce the amount of memory required to store an alignment cell from 3 integers to 1 integer and 3 booleans (embedded in a char), thus increases the parallelism level of the CUDA program.

The scoring matrix of the alignment is computed using the following formula:

$$M(p,q) = \max \begin{cases} M(p-1,q-1) + sbt(R_i[p], R_j[q]) \\ M(p,q-1) + \alpha D(p,q-1) + \beta U(p,q-1) \\ M(p-1,q) + \alpha D(p-1,q) + \beta L(p-1,q) \end{cases} \tag{3.5}$$

where D, L and U are binary DP matrices to indicate which neighbor (diagonal, left or up) the maximum in cell $M(p,q)$ is derived from. Matrices D, L and U are defined as follows:

$$\begin{aligned}
U(p,q)=0,\quad & L(p,q)=0, D(p,q)=1 \\
& \text{if}\quad M(p,q)=M(p-1,q-1)+sbt(R_i[p],R_j[q]) \\
U(p,q)=0,\quad & L(p,q)=1, D(p,q)=0 \\
& \text{if}\quad M(p,q)=M(p,q-1)+\alpha D(p,q-1)+\beta U(p,q-1) \\
U(p,q)=1,\quad & L(p,q)=0, D(p,q)=0 \\
& \text{if}\quad M(p,q)=M(p-1,q)+\alpha D(p-1,q)+\beta L(p-1,q)
\end{aligned} \tag{3.6}$$

Note that except for the first row and the first column, $D(p,q)+L(p,q)+U(p,q)=1$ for $p=0,\ldots,l_i, q=0,\ldots,l_j$.

To make the genetic distance calculation more suitable for parallelization, a trace-back-free linear space solution was implemented by merging the *ml* and *al* calculation into the DP computation of the optimal global alignment score. To obtain the values *ml* and *al*, two more matrices ML and AL are introduced with the recurrent relations as follows:

$$\begin{aligned}
ML(p,q)= \;& U(p,q)ML(p,q-1)+L(p,q)ML(p-1,q) \\
& +D(p,q)ML(p-1,q-1)-m(R_i[p],R_j[q])+1 \\
AL(p,q)= \;& U(p,q)AL(p,q-1)+L(p,q)AL(p-1,q) \\
& +D(p,q)AL(p-1,q-1)+1
\end{aligned} \tag{3.7}$$

where $m(R_i[p],R_j[q])=1$ if $R_i[p]=R_j[q]$ and $m(R_i[p],R_j[q])=0$ otherwise. Initial conditions are given by $ML(0,q)=ML(p,0)=0, AL(0,q)=q, AL(p,0)=p$ for $p=0,\ldots,l_i, q=0,\ldots,l_j$.

Figure 3.11 illustrates an example for the computation of the DP matrices M, U,D, L, ML and AL. The dark shaded cells and arrows show the semi-global alignment path from the cell with the largest value in the final row or final column (in this case value 7 in cell(6,5)) to any cell from the first row or the first col (in this example, cell(2,0)) . Note that this is a score-only computation and therefore requires only linear space.

Furthermore, the banded alignment concept has been employed to reduce the number of computed DP matrix cells. In this approach, only cells within a narrow band along the main diagonal are calculated. Even though some of the distance values might change, the pairwise distances can still result in a similar OTU structure after clustering.

3.5.2 CUDA implementation

An overview of the CUDA implementation on a single GPU of the genetic distance computation is shown in Figure 3.12. The pair indices and input reads are transferred to CUDA global memory, whereby reads are represented as binary strings using two bits per base: A=00, T=01, G=11, C=10.

Multiple CUDA threads can calculate the pairwise distances in parallel. During the computation, one row of DP matrix values per pairwise alignment is stored in CUDA global memory which is accessed using coalesced data transfer to reduce transfer time. Moreover, each thread within a thread block computes a DP matrix

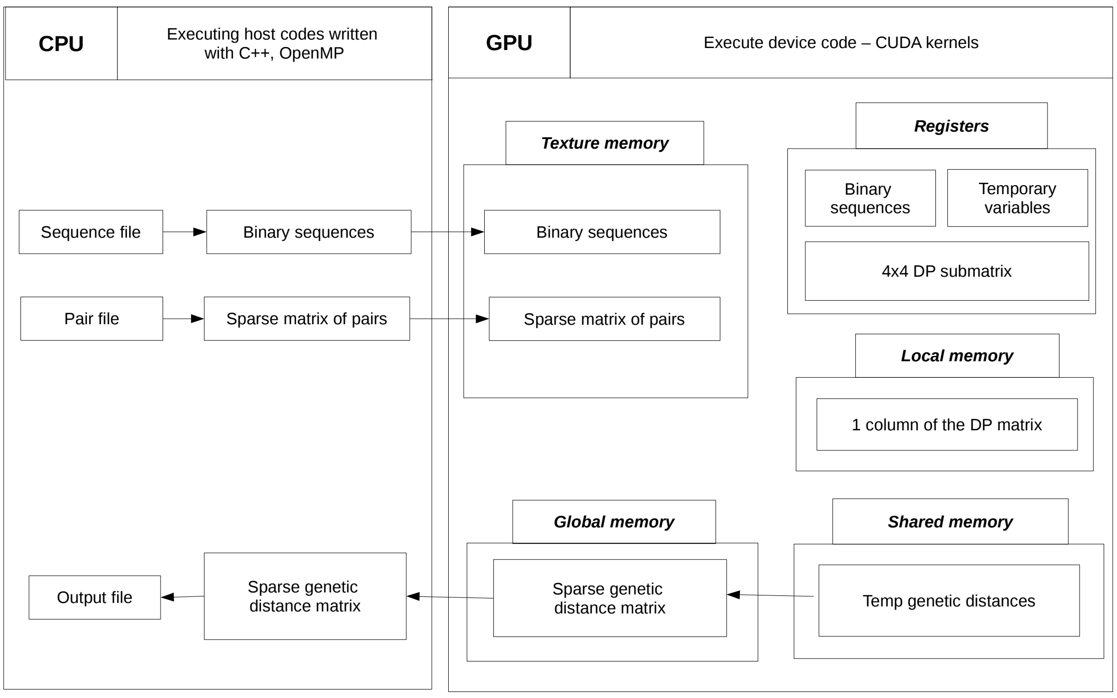
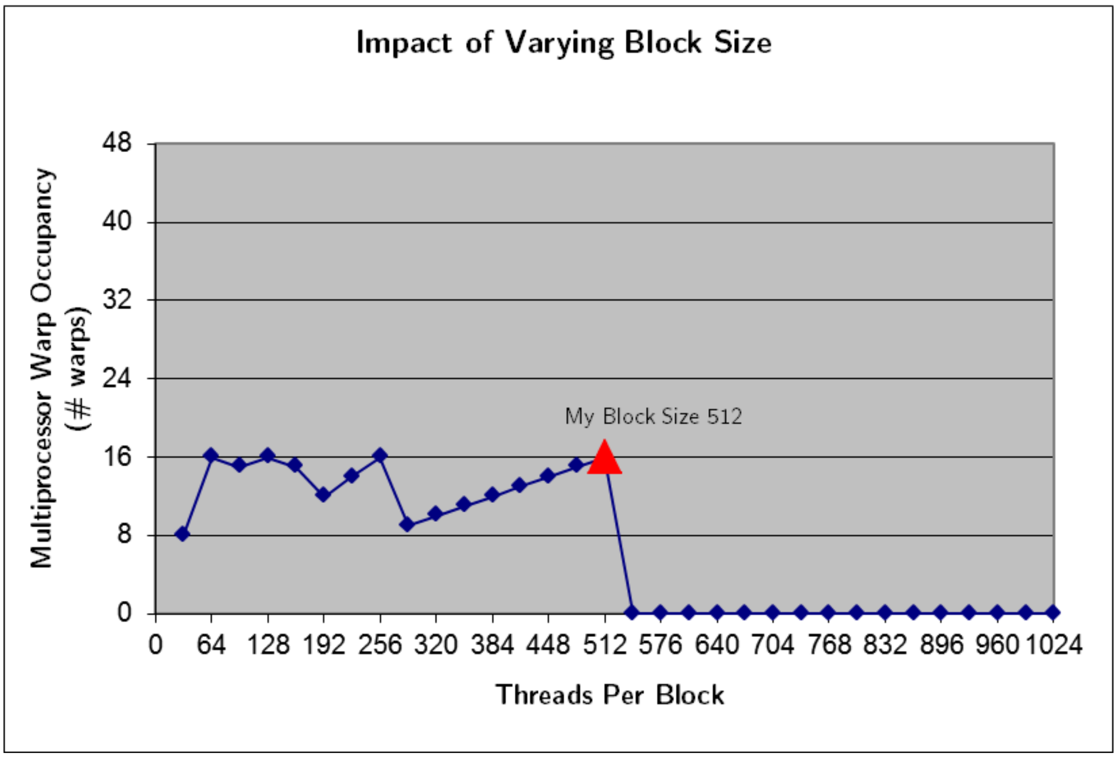
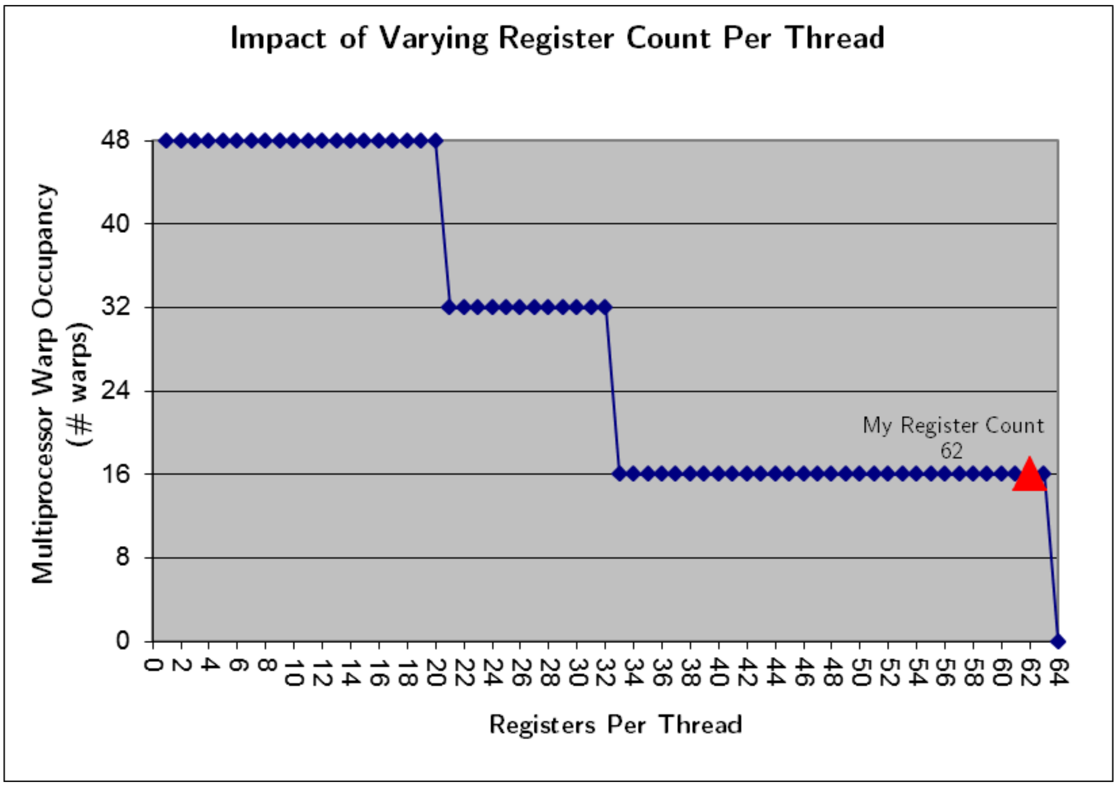


FIGURE 3.12: The overall structure of the parallel CUDA program for computing genetic distances on a GPU.

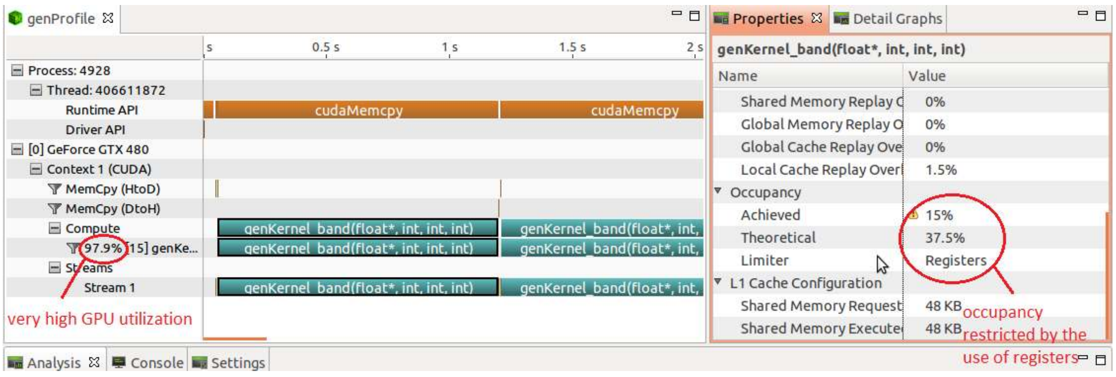


(a) Block size



(b) Register per thread

FIGURE 3.13: Choosing the block size and the register count per thread in order to maximize the multiprocessor warp occupancy. The red triangles mark the chosen settings for the CUDA program to compute genetic distances: 512 threads per block and 62 registers per thread.



genKernel_band(float*, int, int, int)

Name	Value
Shared Memory Replay O	0%
Global Memory Replay O	0%
Global Cache Replay Ove	0%
Local Cache Replay Over	1.5%
Occupancy	
Achieved	15%
Theoretical	37.5%
Limiter	Registers
L1 Cache Configuration	
Shared Memory Request	48 KB
Shared Memory Execute	48 KB

Figure 3.14: Profiling the CUDA program for computing genetic distance running on an NVIDIA GTX480 using the NVIDIA Visual Profiler. The parallel program achieves a high GPU utilization value of 97.9%.

block of size 4 × 4 using register - the memory type with fastest access, which reduces the costly accesses to global memory by a factor of 4 and makes the kernel compute-bound rather than memory-bound. At the end of the computation, each thread returns a distance value to a buffer located in the per block shared memory. The result buffer is then transferred to CUDA global memory, back to the host memory and the CPU creates the final sparse genetic distance matrix.

Figure 3.13 shows the result from the CUDA occupancy calculator for choosing the optimal block size and the register count per thread for the targeted GPU. Figure 3.14 shows the profiling result for the genetic distance computation. Although it has fully utilized the GPU for computation with 97.9% of wall time, it has an achieved occupancy of 12.4% over the theoretical occupancy of 37.5%. The genetic distance computation is more compute-intensive and requires much more memory than the k-mer distance computation. Therefore, its theoretical occupancy is limited by the number of registers required by each thread for computation. The compute/memory ratio is 417, showing that the genetic kernel is compute-bound rather than memory-bound.

I suggest the following practical CUDA optimization guidelines which have proven effective throughout this work:

- Use pinned host memory and minimize the data transfer between host and device: Using pinned host memory, the CUDA implementation for genetic distance computation achieves an average memory throughput of 5.9 gigabytes per second (GBPS) (the maximum host to device bandwidth on the PCI Express x16 Gen2 is 8 GBPS) in both k-mer distance and genetic distance modules, which is two times faster than if the non pinned host memory was used.

- Optimize the use of registers and threads per block: A *Fermi*-based GPU has 32768 32-bit registers per streaming processor. Each processor can have maximum 1024 threads (32 warps x 32 threads). For the occupancy of 100%, each thread can use up to $32768/1024 = 32$ registers. One can use the CUDA occupancy calculator to help visualize the effects of the number of threads per block and the number of registers per thread to occupancy.
- Maximize the use of shared memory in place of local or global memory.
- Use texture memory for coalesced access of CUDA two-dimensional array.
- Replace the more expensive calculation such as multiplication and division less expensive ones like addition and subtraction whenever possible.
- Use binary instead of text data files to speed up the file input/output process.

3.6 Summary

One of the main contributions of this thesis are compute-efficient algorithms for distance matrix calculation on parallel architectures. Specifically, parallel algorithms have been developed to compare biological sequence data using either the k-mer distance or the genetic distance or the Euclidean distance metric.

Each of these parallel algorithms is implemented either in CUDA to support execution on a general-purpose graphical processing unit (GPGPU) or in OpenMP to support execution on a multi-core CPU. While OpenMP implementation can be straightforward, the parallel implementations in CUDA are often a non-trivial tasks. To achieve maximum speedup on GPUs, the parallel implementations in

CUDA often involve various tasks such as data compression, task division and usage of different memory types available on the GPUs.

In this chapter, we use small to medium datasets for profiling purpose. For the sensitivity of these parallel implementations to larger datasets, please refer to section 6.3.2 of chapter 6.

The parallel distance matrix computation modules discussed in this chapter although targeting biological sequence data can also be used for other text analysis tasks. Furthermore, parallel Euclidean distance can be used as an independent module for computing pairwise distances between real-valued data points.

Chapter 4

Memory-Efficient Agglomerative Hierarchical Clustering

Computing a hierarchical clustering of objects from a pairwise distance matrix is an important algorithmic kernel in computational science. Since the storage of this matrix requires quadratic space with respect to the number of objects, the design of memory-efficient approaches is of high importance to research . This chapter provides some background information on standard agglomerative hierarchical clustering algorithms, discusses in details a new memory-efficient algorithm namely SparseHC and reports the evaluation results to compare SparseHC with other algorithms for the same purpose. The work described in this chapter has been published in [79].

4.1 Background

4.1.1 Agglomerative hierarchical clustering

Clustering is an important unsupervised machine learning technique to group similar objects in order to uncover the inherent structure of a given dataset. Depending on the output, clustering algorithms are broadly divided into two main

TABLE 4.1: The Lance Williams formulation of agglomerative hierarchical clustering: parameters for seven commonly-used linkage schemes

Linkage	α_1	α_2	β	γ	Alternative formula
Single	0.5	0.5	0	-0.5	$d_{ij} = \min_{x \in C_i, y \in C_j} d_{xy}$
Complete	0.5	0.5	0	0.5	$d_{ij} = \max_{x \in C_i, y \in C_j} d_{xy}$
Average	$\frac{\lvert C_i\rvert}{\lvert C_i\rvert+\lvert C_j\rvert}$	$\frac{\lvert C_j\rvert}{\lvert C_i\rvert+\lvert C_j\rvert}$	0	0	$d_{ij} = \frac{1}{\lvert C_i\rvert\lvert C_j\rvert} \sum_{x \in C_i, y \in C_j} d_{xy}$
Weighted	0.5	0.5	0	0	
Centroid	$\frac{\lvert C_i\rvert}{\lvert C_i\rvert+\lvert C_j\rvert}$	$\frac{\lvert C_j\rvert}{\lvert C_i\rvert+\lvert C_j\rvert}$	$-\frac{\lvert C_i\rvert\lvert C_j\rvert}{(\lvert C_i\rvert+\lvert C_j\rvert)^2}$	0	
Median	0.5	0.5	-0.25	0	
Ward	$\frac{\lvert C_i\rvert+\lvert C_m\rvert}{\lvert C_i\rvert+\lvert C_j\rvert+\lvert C_m\rvert}$	$\frac{\lvert C_j\rvert+\lvert C_m\rvert}{\lvert C_i\rvert+\lvert C_j\rvert+\lvert C_m\rvert}$	$-\frac{\lvert C_m\rvert}{\lvert C_i\rvert+\lvert C_j\rvert+\lvert C_m\rvert}$	0	

categories: hierarchical clustering and partitional (or flat) clustering [80–82]. The structured output produced by hierarchical clustering algorithms is often more informative than the unstructured set of clusters returned by partitional clustering algorithms [49, 83]. Thus, hierarchical clustering is a crucial data analysis tool in many fields including computational biology and social sciences [84]. Nonetheless, the quadratic time and especially the quadratic memory complexity have limited the use of hierarchical clustering software to rather small datasets [83]. Since many areas of computational science face a data explosion, addressing the problem of computing a hierarchical clustering from a large and possibly sparse pairwise distance matrix in a memory-efficient way is becoming increasingly important. I tackle this problem by presenting a new general-purpose online hierarchical clustering algorithm called SparseHC.

Hierarchical clustering can be divided into two categories: the agglomerative "bottom-up" approach and the divisive "top-down" approach [83]. I focus on the former category: agglomerative hierarchical clustering (AHC). AHC algorithms can be characterized as sequential, agglomerative, hierarchical, and non-overlapping [85, 86]. In AHC algorithms, objects or data points are first treated as singletons and subsequently merged one pair of clusters at a time until there is only one cluster left. There are seven commonly used linkage schemes: single, complete, average (UPGMA), weighted (WPGMA), centroid (UPGMC), median

(WPGMC) and Ward's method. The properties of each scheme are discussed in [87]. The merging criteria used by all these schemes can be neatly represented with the recurrence formula by Lance and Williams [88].

Given that two clusters C_i and C_j have previously been merged into cluster C_k, the distance between cluster C_k and any unmerged cluster C_m is defined as:

$$d_{km} = d(C_i \cup C_j, C_m) = \alpha_1 d_{im} + \alpha_2 d_{jm} + \beta d_{ij} + \gamma |d_{im} - d_{jm}|$$

The specific parameters for each scheme are defined in Table 4.1.

Depending on the input data, AHC algorithms can be divided into the "stored data approach" and the "stored matrix approach" [89, 90]. The stored data approach requires the recalculation of pairwise distance values for each merging step. Since only data points are stored in the main memory, algorithms in this approach can achieve $\mathcal{O}(N)$ space complexity often at the expense of $\mathcal{O}(N^3)$ time complexity [91], where N is the number of input data points.

One notable algorithm in the stored data approach is the nearest-neighbor chain algorithm, which achieves $\mathcal{O}(N)$ space complexity and $\mathcal{O}(N^2)$ time complexity for the Ward's method linkage scheme. However, this algorithm is not applicable to the centroid and median linkage schemes because these schemes do not fulfill the required reducibility criterion i.e. $d(C_i \cup C_j, C_m) \geq \min(d(C_i, C_m), d(C_j, C_m))$ [90, 91]. For the single-, complete- and average-linkage schemes, this algorithm requires $\mathcal{O}(N^2)$ space and time complexity [92]. On the contrary, in the stored matrix approach an all-against-all pairwise distance matrix of size N^2 is first computed and then used for clustering. As a result, this approach requires $\mathcal{O}(N^2)$ time and memory complexity [82].

To overcome the low memory efficiency of classical AHC algorithms, new techniques perform either data reduction by random sampling (e.g. data sampling and partitioning in CURE [93]) or data summarization by using a new data structure to represent the original data (e.g. the CF tree in BIRCH [48]). Although these algorithms have linear memory complexity [82], the dendrograms produced by these algorithms are indeterministic and are dissimilar those produced by standard AHC tools because of the random procedures being used.

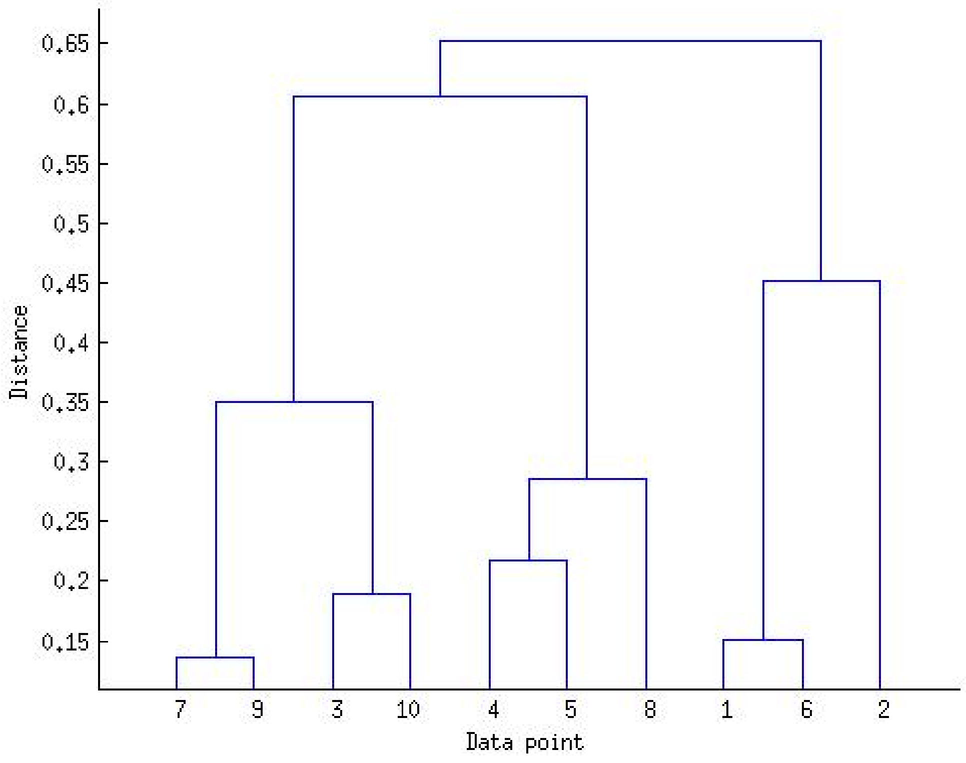
In this work, I focus on reducing the primary memory consumption of the AHC stored matrix approach. I have developed SparseHC, a general-purpose memory-efficient AHC algorithm for single-, complete- and average-linkage schemes. SparseHC is an online algorithm. Borodin and El-Yaniv [94] defined online algorithms as algorithms that focus on scenarios where "*the input is given one piece at a time and upon receiving an input, the algorithm must take an irreversible action without the knowledge of future inputs*". Because online algorithms only require partial input in the main memory for processing, they are often used to target problems with high space complexity. To my knowledge, there are only a few existing online hierarchical clustering algorithms for the stored matrix approach including MCUPGMA [95] for the average scheme and ESPRIT *hcluster* [44] for single and complete schemes.

SparseHC employs a similar strategy as in MCUPGMA and *hcluster* where the input distance matrix is first sorted and then processed in a chunk-by-chunk manner. SparseHC incorporates two new techniques in order to achieve significantly better performance:

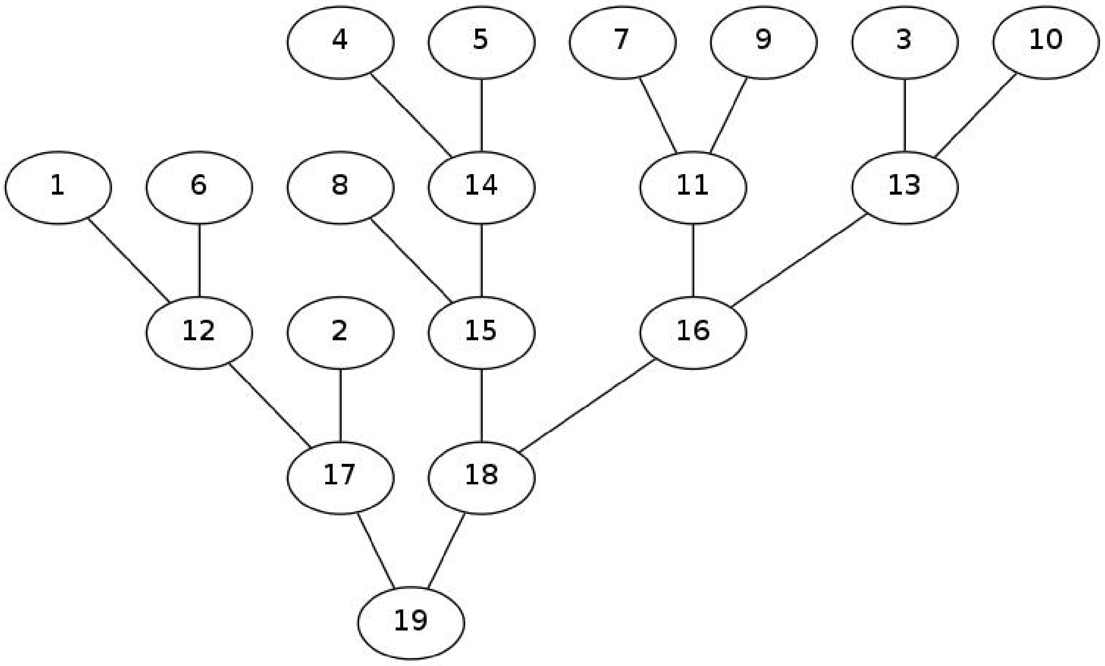
1. Compression of the information in the currently loaded chunk of the input matrix into the most compact form.
2. Usage of an efficient graph representation to store unmerged cluster connections, which allows constant access to these connections for faster speed.

4.1.2 Memory-efficient AHC algorithms

SparseHC and other online AHC algorithms work based on the observation that once the values of an input distance matrix are sorted in ascending order and

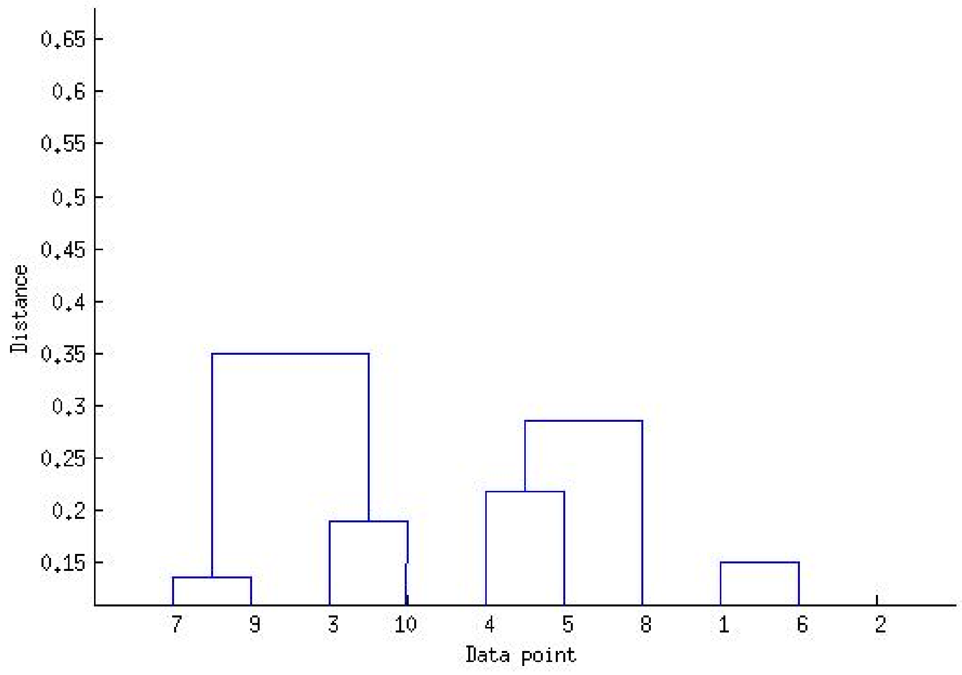


(a) Full dendrogram

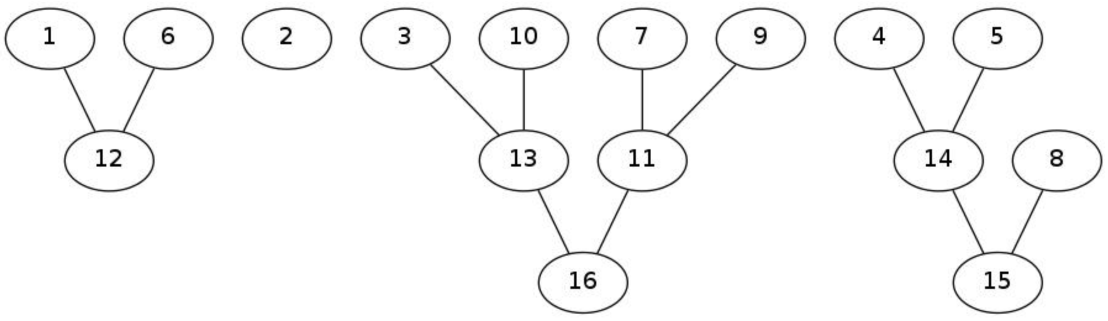


(b) Complete binary tree

Figure 4.1: Example of the standard dendrogram and the equivalent complete binary tree generated by clustering an input dataset of 10 data points using the average-linkage scheme.



(a) Partial dendrogram



(b) Incomplete binary tree

FIGURE 4.2: Example of the partial dendrogram and the equivalent incomplete binary tree generated by clustering the same input dataset as above (Figure 4.1) but using only the pairwise distances smaller than the cutoff threshold of $\theta = 0.4$ instead of the whole distance matrix.

TABLE 4.2: The time and space complexity of query and update operations for different graph implementations. SparseHC uses the adjacency map representation which is derived from the adjacency list to accelerate edge-related operations.

Representation	Storage	Add edge	Remove Edge	Query edge
Incidence matrix	$\mathcal{O}(\|V\|\|E\|)$	$\mathcal{O}(\|V\|\|E\|)$	$\mathcal{O}(\|V\|\|E\|)$	$\mathcal{O}(E)$
Adjacency matrix	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Incidence list	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(1)$	$\mathcal{O}(E)$	$\mathcal{O}(E)$
Adjacency list	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(1)$	$\mathcal{O}(E)$	$\mathcal{O}(V)$
Adjacency map	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

loaded chunk-by-chunk from the top, the merge order and the dendrogram distances can be accurately determined using only the loaded part i.e. without any knowledge about the unseen portion.

SparseHC takes a sorted distance matrix D as input and iteratively builds a dendrogram from reading only a part of D in each iteration step as shown in Figure 4.1. Depending on the available main memory, a sequence of values $0 = \lambda_0 < \lambda_1 < \ldots < \lambda_T = \theta$ is built on-the-fly. In each iteration step $1 \leq t \leq T$, all distances d_{xy} with $\lambda_{t-1} \leq d_{xy} < \lambda_t$ are read from D. Starting from the a tree consisting of only N leaves where a leaf node i $(1 \leq i \leq N)$ represents the singleton cluster $C_i = \{i\}$, a binary tree (which is the dendrogram) is built from bottom up. Since only two clusters are merged at a time, the full binary tree has a height of $N-1$ and consists of $2N-1$ nodes (see Figure 4.1).

In offline AHC algorithms, D has to be a full pairwise distance matrix. However, in online AHC algorithms such as SparseHC, D can be either full or sparse. A sparse distance matrix D_θ uses a predefined distance cutoff θ $(0 \leq \theta < 1)$ and stores only distance values up to θ $(0 \leq d_{xy} \leq \theta, \forall d_{xy} \in D)$. For sparse matrix clustering, the output dendrogram has a height in the range of $[1, N-1]$ and a size in the range of $[N, 2N-1]$ as shown in Figure 4.2.

The input to SparseHC is a sorted full or sparse distance matrix stored in a list of tuples (i, j, d_{ij}) format. The maximum element of a full matrix is 1.0 while that of a partial matrix is a pre-defined distance cutoff $\theta < 1.0$. The ability to process sparse distance matrices is particularly useful in applications like taxonomic studies in bioinformatics [44, 96] where only the lower part of the final dendrogram is of interest. In these situations, runtime and memory usage are further reduced depending on the sparsity of the input matrix. The memory efficiency and ability of SparseHC to process sparse matrices come at the cost of pre-sorting the input matrices. Nonetheless, the memory performance of SparseHC is not affected if an external merge sort algorithm [97] is used for the sorting stage.

Similar to offline AHC algorithms, during the clustering process, SparseHC needs to store all the connections amongst unmerged clusters to figure out which cluster pair will be merged next. However, same as other online AHC algorithms, SparseHC only stores the connections amongst active clusters. A cluster pair is called *active* in iteration step t when (1) both clusters do not have a parent and (2) at least one distance value between the member data points has been read from the input file during the first t iteration steps. I observe that active clusters contribute to only a small subset of unmerged clusters. The memory efficiency of online AHC algorithms is determined by their ability to store active connections in a compact way.

SparseHC uses an undirected weighted graph to model the connections amongst active cluster pairs. This graph consists of a set of vertices V and a set of edges E. The vertices are the nodes of the binary tree i.e. $V = \{C_1, C_2, \ldots, C_{2N-1}\}$. A fixed size array is used to store all possible vertices, hence allowing $\mathcal{O}(1)$ vertex query and update. The undirected weighted edges are the active connections amongst the clusters.

Graphs are typically implemented using an adjacency matrix, an adjacency list, an incidence matrix or an incidence list [98]. The time and space complexity of each representation are shown in Table 4.2. To facilitate its cluster merging process, SparseHC prefers a graph representation that requires minimum storage for the graph and allows constant time to perform edge insertion, edge deletion, and edge update. Therefore, I have modified the standard adjacency list to assist these operations. I call this graph representation the adjacency (hash) map.

The adjacency map is a collection of unordered hash maps, one for each vertex of the graph. Each hash map records the set of neighbors of its vertex using the neighbor vertex identification number as the key. Because of this adjacency map representation, SparseHC can use $\mathcal{O}(|V| + |E|)$ space to store all the clusters and their active connections. More importantly, these connections can be accessed and updated in $\mathcal{O}(1)$ time.

4.2 SparseHC

This section presents a memory-efficient online hierarchical clustering algorithm called SparseHC [79]. SparseHC scans a sorted and possibly sparse distance matrix chunk-by-chunk. Meanwhile, a dendrogram is built by merging cluster pairs as and when the distance between them is determined to be the smallest among all remaining cluster pairs. The key insight used is that for finding the cluster pair with the smallest distance, it is unnecessary to complete the computation of all cluster pairwise distances. Partial information can be utilized to calculate a lower bound on cluster pairwise distances that are subsequently used for cluster distance comparison. Experimental results show that SparseHC achieves a linear empirical memory complexity, which is a significant improvement compared to existing algorithms.

4.2.1 Algorithm

The definition of the edge $e_{ij}^{(t)}$ between two active clusters C_i and C_j in iteration step t is defined in Table 4.3 depending on the clustering scheme. $d_{ij}^{(t)}$ is the minimum possible distance between C_i and C_j and is computed according to Table 4.3. $s_{ij}^{(t)}$ ($n_{ij}^{(t)}$) is the sum (number) of distance values between any member of C_i to any member of C_j that has been read from the input file so far. λ_t is the maximum distance value loaded from the input matrix so far.

In each iteration step t, active edges are partitioned into two sets: a set of *complete edges* $K^{(t)}$ and a set of *incomplete edges* $I^{(t)}$ (both sets are stored in the adjacency

TABLE 4.3: Distance d_{ij} between cluster C_i and C_j for clustering sparse matrices

Linkage	Edge definition	Cluster distance		Complete condition
		Incomplete edge	Complete edge	
Single	$e_{ij}^{(t)} = ()$	$d_{ij}^{(t)} = 1.0$	$d_{ij}^{(t)} = d_{xy}$	$n_{ij}^{(t)} = 1$
Complete	$e_{ij}^{(t)} = (n_{ij}^{(t)})$	$d_{ij}^{(t)} = 1.0$	$d_{ij}^{(t)} = d_{xy}$	$n_{ij}^{(t)} = \|C_i\|\|C_j\|$
Average	$e_{ij}^{(t)} = (s_{ij}^{(t)}, n_{ij}^{(t)})$	$d_{ij}^{(t)} = \frac{s_{ij}^{(t)}+\lambda^{(t)}(\|C_i\|\|C_j\|-n_{ij}^{(t)})}{\|C_i\|\|C_j\|}$	$d_{ij}^{(t)} = \frac{s_{ij}^{(t)}}{\|C_i\|\|C_j\|}$	$n_{ij}^{(t)} = \|C_i\|\|C_j\|$

Algorithm 2 SparseHC algorithm for a sorted input matrix D from N data points stored as a list of tuples (x, y, d_{xy}).

```
$C_i \leftarrow \{i\} \quad \forall i = 1, \ldots, N$
$E$.max_size $\leftarrow N$ {$E$ is the adjacency map $E = K \cup I$}
$k \leftarrow N; t \leftarrow 0; \lambda_0 \leftarrow 0$ {initialize cluster id $k$, iteration $t$, distance threshold $\lambda$}
while $D \neq \emptyset$ do
  $t \leftarrow t + 1$
  while $D \neq \emptyset$ and $E$.size $\leq$ E.max_size do
    $d_{xy} \leftarrow D$.get_next(); $D = D \setminus \{d_{xy}\}$
    $C_i \leftarrow C_x$.get_ancestor(); $C_j \leftarrow C_y$.get_ancestor()
    $e_{ij}^{(t)}$.update($d_{xy}$) {create $e_{ij}^{(t)}$ if it does not exist}
    compute $d_{ij}^{(t)}$ {use the cluster distance formula in Table 4.3}
    if $e_{ij}^{(t)}$ is complete then
      $C_i$.minK $\leftarrow$ min($C_i$.minK, $d_{ij}^{(t)}$); $C_i$.merge_candidate $\leftarrow C_j$
    else
      $C_i$.minI $\leftarrow$ min($C_i$.minI, $d_{ij}^{(t)}$)
    end if
  end while
  $\lambda_t \leftarrow d_{xy}$ {$\lambda_t$ is the largest distance in an iteration}
  while $d_{ij} = \min(K^{(t)}) \leq \min(I^{(t)})$ and $k \leq 2N - 1$ do
    $k \leftarrow k + 1; \quad C_k \leftarrow C_i \cup C_j$ {merge clusters $C_i$ and $C_j$ into cluster $C_k$}
    for all $C_m$ such that $e_{im}^{(t)} \in E \vee e_{jm}^{(t)} \in E$ do
      $e_{km}^{(t)} \leftarrow$ merge($e_{im}^{(t)}, e_{jm}^{(t)}$) {$s_{km}^{(t)} \leftarrow s_{im}^{(t)} + s_{jm}^{(t)}; \quad n_{km}^{(t)} \leftarrow n_{im}^{(t)} + n_{jm}^{(t)}$}
      $E = E \cup \{e_{km}^{(t)}\} \setminus \{e_{im}^{(t)}, e_{jm}^{(t)}, e_{ij}^{(t)}\}$
      compute $d_{km}^{(t)}$ {use the cluster distance formula in Table 4.3}
      if $e_{km}^{(t)}$ is complete then
        $C_k$.minK $\leftarrow$ min($C_k$.minK, $d_{km}^{(t)}$); $C_k$.merge_candidate $\leftarrow C_m$
      else
        $C_k$.minI $\leftarrow$ min($C_i$.minI, $d_{km}^{(t)}$)
      end if
    end for
  end while
  if $E$.size $\geq E$.max_size then
    $E$.max_size $\leftarrow 2 \times E$.max_size {dynamically increase the adjacency map size}
    if $E$.size $\geq$ RAM.size then
      return partial result {when the memory limit is reached}
    end if
  end if
end while
return full result
```

map). A complete edge is a connection between two active clusters that are ready to be merged. An incomplete edge is a connection between two active clusters that are yet to be merged. For complete- and average- linkage schemes, an edge is complete when $n_{ij}^{(t)} = |C_i||C_j|$. Otherwise, when $n_{ij}^{(t)} < |C_i||C_j|$, the edge is considered incomplete. For single-linkage scheme, an edge is complete when $n_{ij}^{(t)} = 1$ i.e. the connection between two clusters is complete as soon as the first distance value between any member reads has been read from the input.

Let $\min(I^{(t)})$ $(\min(K^{(t)}))$ denote the smallest distance value in $I^{(t)}$ $(K^{(t)})$. The high-level description of the SparseHC algorithm in each iteration t $(1 \leq t \leq T)$ consists of three steps:

1. Read the distance values d_{xy} from matrix D in ascending order until the adjacency map is full and determine the value $\lambda^{(t)}$.
2. Update/create the edges for all active cluster pairs with the new distances and partition them into $I^{(t)}$ and $K^{(t)}$.
3. Retrieve the edge $e_{ij}^{(t)}$ for which $d_{ij}^{(t)} = \min(K^{(t)}) \leq \min(I^{(t)})$. Merge the cluster pair C_i and C_j into cluster C_k. Delete $e_{ij}^{(t)}$ from $K^{(t)}$ and combine existing edges to either cluster C_i or C_j into new edges to cluster C_k. Repeat until $\min(K^{(t)}) > \min(I^{(t)})$.

Algorithm 2 shows the details of SparseHC.

4.2.2 Correctness

To show the dendrogram produced by SparseHC is correct, we need to prove that up to the distance cutoff θ both the merge distance values and the merge order are preserved.

Merge distances: Let $d_{ij}^{(t)}$ be the merge distance between two clusters C_i and C_j assuming that they are being merged by SparseHC in an iteration t. Let d_{ij} be the merge distance between C_i and C_j produced by a traditional AHC algorithm. We need to show that $d_{ij}^{(t)} = d_{ij}$. Indeed, when C_i and C_j are merged by SparseHC, the edge $e_{ij}^{(t)}$ is complete. By definitions of d_{ij} in Table 4.1 and in $d_{ij}^{(t)}$ when $e_{ij}^{(t)}$ is complete in Table 4.3, it holds that $d_{ij}^{(t)} = d_{ij}$. Therefore, the merge distance values are preserved.

Merge order: To prove that SparseHC preserves the merge order, we show that if C_i and C_j are merged before C_k and C_m, then $d_{ij} \leq d_{km}$. At the time when C_i and C_j are being merged in an iteration t, we have $d_{ij}^{(t)} = d_{ij} = \min(K^{(t)})$. In step t, after C_i and C_j are merged, the status of the edge $e_{km}^{(t)}$ is one of the followings:

1. $e_{km}^{(t)}$ is active and complete $\Rightarrow e_{km}^{(t)} \in K^{(t)}$. For a complete edge, it holds that $d_{km}^{(t)} = d_{km}$.
   Besides, $e_{km}^{(t)} \in K^{(t)} \Rightarrow \min(K^{(t)}) \leq d_{km}^{(t)}$. Therefore, $d_{ij} \leq d_{km}$
2. $e_{km}^{(t)}$ is active and incomplete $\Rightarrow e_{km}^{(t)} \in I^{(t)}$. For an incomplete edge, it holds that $d_{km}^{(t)} \leq d_{km}$. In SparseHC, it always holds that $\min(K^{(t)}) \leq \min(I^{(t)}) \Rightarrow d_{ij} \leq \min(I^{(t)})$.
   Besides, $e_{km}^{(t)} \in I^{(t)} \Rightarrow \min(I^{(t)}) \leq d_{km}^{(t)}$. Therefore, $d_{ij} \leq d_{km}$
3. $e_{km}^{(t)}$ is inactive $\Rightarrow e_{km}^{(t)} \notin \{K^{(t)} \cup I^{(t)}\}$. For an inactive edge, it holds that $\lambda_t < d_{km}^{(t)} \leq d_{km}$. Since C_i and C_j have been merged in iteration t, $d_{ij} \leq \lambda_t$. Therefore, $d_{ij} < d_{km}$

For all cases, we have $d_{ij}^{(t)} = d_{ij}$ and $d_{ij} \leq d_{km}$ i.e. both the merge distances and the merge order are preserved in SparseHC.

4.2.3 Memory efficiency

While standard offline AHC algorithms store all the connections amongst unmerged clusters in memory (i.e. $|C_i| \times |C_j|$ values for a cluster pair (C_i, C_j), SparseHC uses *at most* two values per cluster pair: the number of connections $n_{ij}^{(t)}$ and the sum of distances $s_{ij}^{(t)}$ (see Table 4.3). Specifically, SparseHC maintains only one value per cluster pair ($n_{ij}^{(t)}$) for complete-linkage clustering, two values per pair ($n_{ij}^{(t)}, s_{ij}^{(t)}$) for average-linkage clustering and none for single-linkage clustering.

Compared to offline AHC tools, SparseHC uses less primary memory because of two reasons: (1) SparseHC stores only the information from the currently loaded chunks and (2) It stores a compact version of the seen information: at most two values per active cluster pair.

Compared to existing online AHC tools such as *hcluster* and MCUPGMA, SparseHC is better because of three reasons. Firstly, SparseHC uses an array of hash maps to store the compact cluster connections. This efficient data structure allows $\mathcal{O}(1)$ query, insert and delete, which contributes to the compute efficiency of SparseHC.

Secondly, for average-linkage clustering, SparseHC uses two values instead of four values per cluster connection as in MCUPGMA. More importantly, SparseHC dynamically allocates the amount of memory needed and returns partial results if all the available memory is consumed. MCUPGMA and *hcluster* require the user to specify the amount of memory beforehand and return error if the allocated amount is insufficient.

Thirdly, SparseHC supports three linkage types while ESPRIT *hcluster* supports only single- and complete-linkage clustering and MCUPGMA supports only average-linkage. ESPRIT has another sub-module called *aveclust* which performs fast average-linkage clustering. However, *aveclust* is not memory-efficient and still requires quadratic memory complexity. Finally, SparseHC stops after performing $N - 1$ merges. This termination condition is particularly useful for single-linkage clustering where the clustering process converges early.

4.3 Evaluation

4.3.1 Experiment setup

To benchmark the performance of SparseHC, I use two standard AHC implementations: the MATLAB *linkage* function and the *fastcluster* Python program [90]. I also use three online AHC implementations including EPSRIT's *hcluster* for complete-linkage clustering, ESPRIT's *aveclust* for average-linkage clustering and MCUPGMA also for average-clustering. These tools are chosen for their compute and/or memory efficiency as well as the availability of executable source codes.

These performance profiling experiments are executed on a Dell T3500 workstation running 64-bit Ubuntu operating system. This station has a quad-core Intel Xeon W3540 2.93 GHz processor and 8GB of RAM. The peak memory usage is measured with the Valgrind Massif profiler [99] and the execution time is recorded using Linux *time* command.

4.3.2 Performance profiling

I use eight Euclidean sparse distance matrices (with distance cutoff $\theta = 0.10$) computed by CRiSPy-Embed on eight V2Mice datasets [33] listed in Table 4.4 as inputs for these average linkage clustering tools. The size of these sparse matrices

TABLE 4.4: Datasets used for performance profiling of average-linkage hierarchical clustering tools

Dataset	Num of annotated reads	Num of unique reads	Average length
V2Mice	1,003,674	159,329	234.0
V2Mice-100k	100,000	26,248	234.0
V2Mice-200k	200,000	45,736	234.0
V2Mice-300k	300,000	62,613	234.3
V2Mice-400k	400,000	77,052	234.6
V2Mice-600k	600,000	95,494	234.6
V2Mice-700k	700,000	110,896	234.6
V2Mice-800k	800,000	130,999	234.6
V2Mice-1000k	1,000,000	158,808	234.6

ranges from 2.2GB to 64.5GB as shown by the "Sparse matrix size" curve in Figure 4.3.

Figure 4.3 and 4.4 show the memory consumption and runtime of SparseHC compared to *aveclust* from the ESPRIT package and MCUPGMA on a Dell workstation with 12GB RAM.

aveclust requires the whole input matrix to be loaded into memory. From the memory profiling graph, we can observe that the amount of memory used by *aveclust* is approximately the size of the input matrix. Therefore with 12GB RAM, the largest dataset that *aveclust* can process is V2Mice-400k with $\textbf{size}(S) = 6.7$GB. *aveclust* runs out of memory while processing larger datasets. On the other hand, MCUPGMA uses the maximum amount of memory allocated to it. This maximum value can be controlled by setting the *heap-size* parameter in MCUPGMA to the amount of available RAM (12GB for our workstation). Empirical results show that MCUPGMA indeed uses approximately 10GB of RAM.

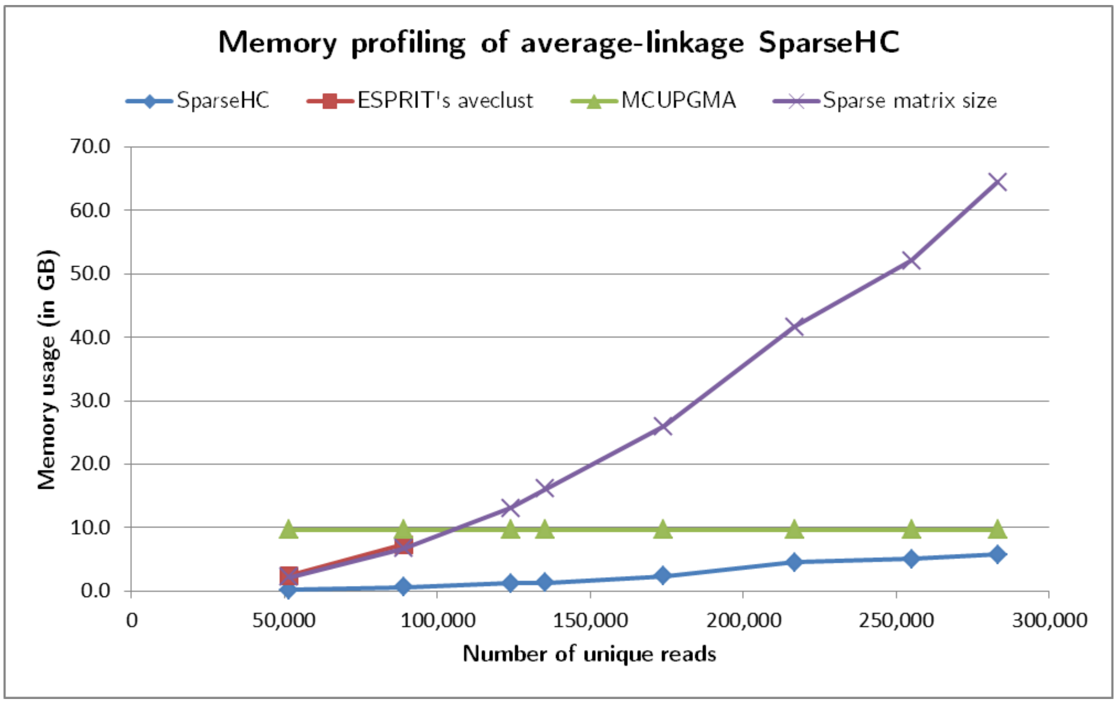


FIGURE 4.3: Memory profiling of SparseHC versus ESPRIT's *aveclust* and MCUPGMA. *aveclust* runs out of memory for datasets larger than V2Mice-400k

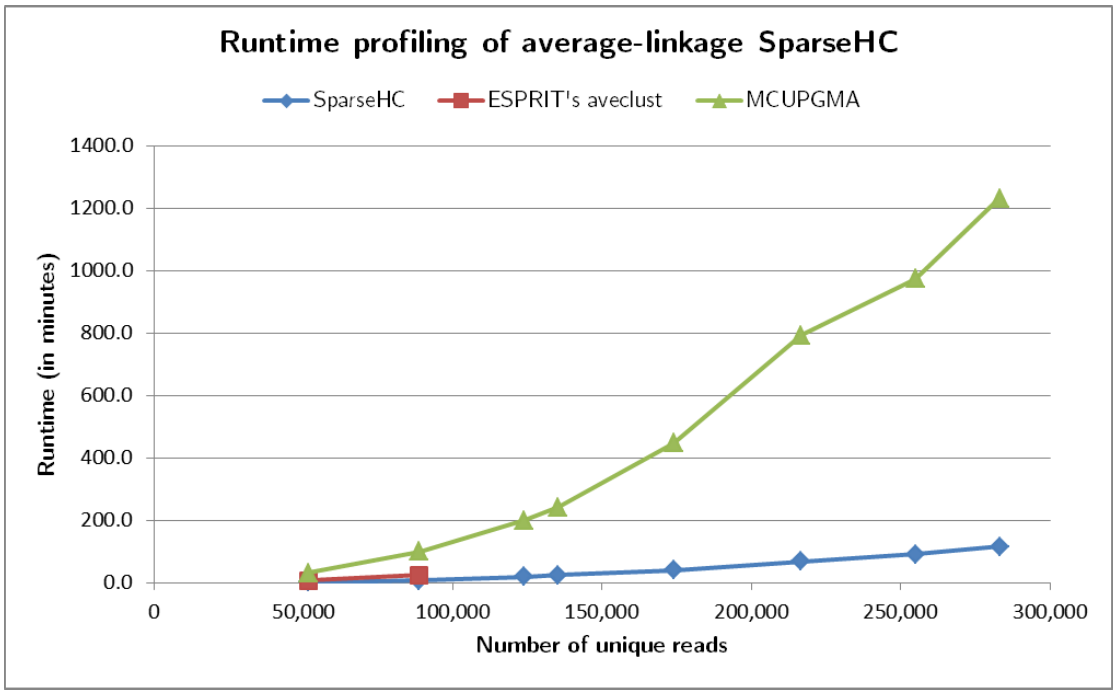


FIGURE 4.4: Runtime profiling of SparseHC versus ESPRIT's *aveclust* and MCUPGMA. *aveclust* runs out of memory for datasets larger than V2Mice-400k

SparseHC tops the other sparse matrix clustering tools in terms of memory efficiency. Figure 4.3 shows that it uses much less memory than the input matrix since it neither requires the whole matrix as in *aveclust* nor uses all available RAM as in MR-MC-UPGMA. On contrary, SparseHC dynamically allocates a necessary amount of memory by only increasing the value M in Algorithm 2 when needed. This dynamic allocation mechanism makes SparseHC more memory-efficient than MCUPGMA. The recorded memory usage shows that SparseHC uses only 5.8GB of RAM to process an input matrix of 64.5GB. Furthermore, SparseHC is 3x faster than *avelust* and 10x - 15x faster than MCUPGMA as shown in Figure 4.4. The distributed link map greatly contributes to the runtime efficiency of SparseHC. In conclusion, SparseHC for average linkage clustering is better than *aveclust* and MCUPGMA in terms of both speed and memory usage.

Table 4.5: The empirical time and space complexity ($f(n) = Cn^k$) of SparseHC versus other online and offline AHC programs. This benchmark experiment employs 20 distance matrices calculated from datasets with sizes ranging from 1000 to 20000 objects. Execution time $f_r(n)$ is assessed in seconds and memory usage $f_s(n)$ is assessed in megabytes. The data input size n is measured in thousand data points.

The empirical runtime growth $f_r(n)$			
AHC tool	Single-linkage	Complete-linkage	Average-linkage
SparseHC	$0.003 \times n^{1.855}$	$0.190 \times n^{2.047}$	$0.216 \times n^{2.040}$
hcluster/aveclust	$0.340 \times n^{2.015}$	$0.378 \times n^{2.000}$	$0.216 \times n^{2.047}$
MATLAB *linkage*	$0.352 \times n^{1.996}$	$0.344 \times n^{1.996}$	$0.336 \times n^{2.003}$
fastcluster	$0.221 \times n^{2.085}$	$0.306 \times n^{1.955}$	$0.236 \times n^{2.073}$
MCUPGMA	not available	not available	$1.313 \times n^{2.120}$
The empirical memory growth $f_s(n)$			
AHC tool	Single-linkage	Complete-linkage	Average-linkage
SparseHC	$0.886 \times n^{0.456}$	$1.272 \times n^{0.848}$	$1.155 \times n^{0.962}$
hcluster/aveclust	$0.242 \times n^{0.482}$	user-defined	$1.007 \times n^{1.982}$
MATLAB *linkage*	$7.674 \times n^{1.998}$	$7.673 \times n^{1.998}$	$7.674 \times n^{1.998}$
fastcluster	$79.166 \times n^{1.995}$	$78.343 \times n^{2.001}$	$78.336 \times n^{2.001}$
MCUPGMA	not available	not available	user-defined

4.3.3 Empirical complexity

The theoretical memory complexities of online AHC algorithms are often difficult to estimate because of their heuristic nature. To benchmark the online algorithms of interest, I use the regression model of memory and execution time introduced by Coffin et al. [100] to calculate the empirical complexities [101] instead of the theoretical values.

Assuming the runtime and memory usage follow the power rule i.e. $f(n) \approx Cn^k$ where n is the input size, the constant factor C and the order k can be estimated using regression on the log-transformed model where ϵ is the error term:

$$\log f(n) = k \log n + \log C + \epsilon \tag{4.1}$$

The values k and C are then estimated as follows:

$$f(n) = Cn^k \rightarrow \frac{f(n_2)}{f(n_1)} = (\frac{n_2}{n_1})^k \tag{4.2}$$

$$\rightarrow k = \frac{\log \frac{f(n_2)}{f(n_1)}}{\log(\frac{n_2}{n_1})} \tag{4.3}$$

$$\rightarrow C = \frac{f(n)}{n^k} \tag{4.4}$$

Table 4.5 reports the average empirical runtime and memory growth of the tested AHC clustering implementations of interest. I use full pre-sorted pairwise Euclidean distance matrices as inputs in this experiment. These matrices are computed from 1000 - 20000 randomly-generated data points. Although the values of C and k in Table 4.5 are only representative of the performance of these algorithms on the tested random datasets, our results on larger datasets in Table 4.7 and on biological sequence datasets in Table 4.6 further confirm and strengthen the validity of the regression model for evaluating empirical complexity and the estimated values in Table 4.5.

The upper sub-table of Table 4.5 shows that all algorithms have quadratic runtime with $k \approx 2$ as expected. Nevertheless, if I plot these functions in the domain $[0, 10^6]$ data points, we can see that SparseHC is the fastest amongst them. Especially for single-linkage clustering, the constant factor C of SparseHC is two orders of magnitude smaller than other tools. For the complete- and average-linkage schemes, the main reason for the fast runtime of SparseHC is the efficiency of edge operations of the adjacency map data structure. For the single-linkage scheme, the significant improvement in speed is due to the edge completion condition ($n_{ij}^{(t)} = 1$). This condition allows two clusters to be merged as soon as the connection between them becomes active, making it unnecessary for SparseHC to store and query active connections of unmerged clusters. Moreover, because of this condition, the merging process for the single-linkage scheme often completes before all values of the input file are loaded, effectively reducing the amount of runtime spent for file input.

The lower sub-table of Table 4.5 shows that offline algorithms have quadratic memory complexity with $k \approx 2$ as anticipated. Python clustering modules such as *fastcluster* or SciPy *cluster* function are less memory-efficient than MATLAB *linkage* since they require additional intermediate data besides the input matrix. On the contrary, the memory usage of SparseHC grows sublinearly/linearly with the input size. SparseHC mainly uses memory to store the adjacency map of unmerged cluster connections.

For the "user-defined" cases in Table 4.5, our experiments show that SparseHC uses less memory than *hcluster* and MCUPGMA. For example, to cluster a 4GB matrix, SparseHC consumes 16MB while *hcluster* uses up 192MB of main memory. Similarly, to cluster a 2.2GB matrix, SparseHC consumes 21MB while MCUPGMA uses up 312MB of main memory. Therefore, SparseHC is the most space-efficient for complete- and average-linkage clustering. For single-linkage, SparseHC and *hcluster* achieve similarly good memory performance.

4.3.4 Clustering DNA datasets

To showcase the usefulness of SparseHC for clustering biological sequences, I tested it with partial pairwise matrices computed from genetic datasets. These sparse matrices are half the sizes of the full matrices. As shown in Table 4.6, I use four matrices generated from datasets of 10000, 20000, 30000, and 40000 DNA sequences.

TABLE 4.6: Execution time and memory usage of SparseHC, MCUPGMA, and ESPRIT's *aveclust* for processing partial matrices calculated from genomic datasets with the sparsity level of 0.5

Number of	Sparse matrix	Runtime (in seconds)			Memory usage (in MB)		
sequences	size (in MB)	SparseHC	*aveclust*	MCUPGMA	SparseHC	*aveclust*	MCUPGMA
10000	483	13.3	15.0	169.2	8.4	96.4	311.2
20000	2035	54.2	67.3	651.2	14.7	383.3	311.9
30000	4706	126.0	174.8	1477.9	24.3	860.9	312.7
40000	8415	229.8	321.1	2815.9	30.9	1529.6	313.8

The matrices are computed using the sequence embedding module in CRiSPy-Embed. Each input sequence is converted into a sequence vector of real coordinates by calculating the k-mer distances between that sequence and a set of representative sequences in the input dataset called seeds. After the conversion process, we compute the pairwise Euclidean distance matrix using the embedding vectors. The elements in the matrix are then sorted in ascending order and only the elements in the lower half are retained for clustering. The linkage scheme used in this experiment is the average-linkage.

Table 4.6 reports the sparse matrix size, together with the memory usage and execution time of MCUPGMA, ESPRIT's *aveclust* and SparseHC. We can observe that SparseHC uses significantly less memory than both *aveclust* and MCUPGMA. Notably, the amount of main memory used by SparseHC is 50x - 280x smaller than the size of the input sparse matrices. In terms of execution, SparseHC is about 10 times faster than MCUPGMA and 1.2x - 1.4x faster than *aveclust*.

4.3.5 Clustering large matrices

To highlight the memory efficiency of SparseHC, I report the $\frac{matrix\_size}{memory\_usage}$ ratio for four representative large datasets in Table 4.7. These datasets contains randomly-generated data points in the coordinate space. The pairwise distance matrices computed from these datasets are 2 - 28 times bigger than the amount of RAM available on the test platform.

Table 4.7 shows that SparseHC can process distance matrices three to four orders of magnitude (i.e. 10^3 - 10^4) larger than the memory capacity. The memory

Table 4.7: The memory efficacy of SparseHC measured by the $\frac{matrix\_size}{memory\_usage}$ ratio

Number of data points	Matrix size (in GB)	Memory usage (in MB) Single	Complete	Average	Memory efficiency of SparseHC Single	Complete	Average
50000	14	7.0	30.5	44.9	2055	469	318
100000	56	12.2	60.0	90.2	4673	954	635
150000	126	17.7	89.6	143.4	7272	1437	897
200000	224	22.9	119.4	198.9	10013	1917	1151

complexity of the single-linkage scheme is linear with respect to the number of input data points. The average-linkage scheme uses about 1.5 times more memory than the complete-linkage scheme.

4.4 Summary

In this chapter, I have introduced the SparseHC algorithm to address the high space complexity of the standard agglomerative hierarchical clustering approach. SparseHC a new online AHC tool which can perform accurate single-, complete- and average-linkage hierarchical clustering with linear empirical space complexity.

SparseHC can be used for general-purpose cluster analysis, not limited to biological sequence analysis. SparseHC is especially useful for clustering large datasets on computing platforms with a limited amount of main memory. As long as a distance matrix is provided as input, SparseHC can process both full and sparse matrices. Nonetheless, it should be noted that in order for SparseHC to provide a sensible dendrogram, the sparse matrix should be a *contiguous* part extracted from a *sorted* full pairwise distance matrix.

It is also useful to note that the amount of memory used by SparseHC depends on (1) the type of distance metric and (2) the type of linkage. SparseHC is more efficient for the distance metrics that satisfy the triangle inequality such as the Euclidean distance that for those that do not adhere to the inequality such as the genetic distance. For the same input matrix, the average-linkage uses more memory than the complete-linkage, which in turn uses more memory than the single-linkage. Therefore, we recommend the average-linkage scheme when cluster quality is the main concern. On the other hand, when memory resource is limited or when execution time is a concern, it is better to use the complete-linkage or the single-linkage scheme.

Since many scientific areas are facing a data explosion issue, memory-efficient algorithms like SparseHC are of high importance to research. SparseHC can be applied to many real-world applications such as genetic clustering in bioinformatics, market segmentation in business and marketing, crime hot spot identification or student group identification in social science. Furthermore, SparseHC can also used to power some of the most popular data mining applications such as social network analysis, search result grouping or user preference prediction in recommender systems.

The SparseHC source code is available at the following Bitbucket repository: `https://bitbucket.com/ngthuydiem/sparsehc`. The Euclidean distance matrix simulator can also be accessed at `https://bitbucket.com/ngthuydiem/simmat`.

Chapter 5

OTU Clustering Pipelines

This chapter describes two new OTU clustering pipelines for species richness estimation of 16S rRNA pyrosequencing datasets called CRiSPy-Embed and CRiSPy. These pipelines are formed by combining the parallel distance matrix computation and the space-efficient clustering algorithms discussed in previous chapters as well as additional preprocessing and post-processing procedures. The evaluation of these pipelines against other OTU clustering tools is reported in the next chapter. The review of state-of-the-art OTU binning tools has been reported in Chapter 2. The work described in this chapter has been published in [102] and [103].

5.1 CRiSPy-Embed

This section introduces a new hierarchical OTU clustering pipeline called CRiSPy-Embed [102]. CRiSPy-Embed stands for "*C*omputing *Ri*chness in 16*S* *Py*rosequencing Datasets using Sequence *Embed*ding". CRiSPy-Embed addresses two issues of the agglomerative hierarchical clustering approach: the high computational cost of the sequence alignment for distance matrix construction and the high memory requirement of average-linkage hierarchical clustering.

5.1.1 Approach

A main idea in CRiSPy-Embed is to replace the costly sequence alignment for sequence comparison with a more efficient computation called sequence embedding. This approach is based on the embedding method used by Clustal-Omega

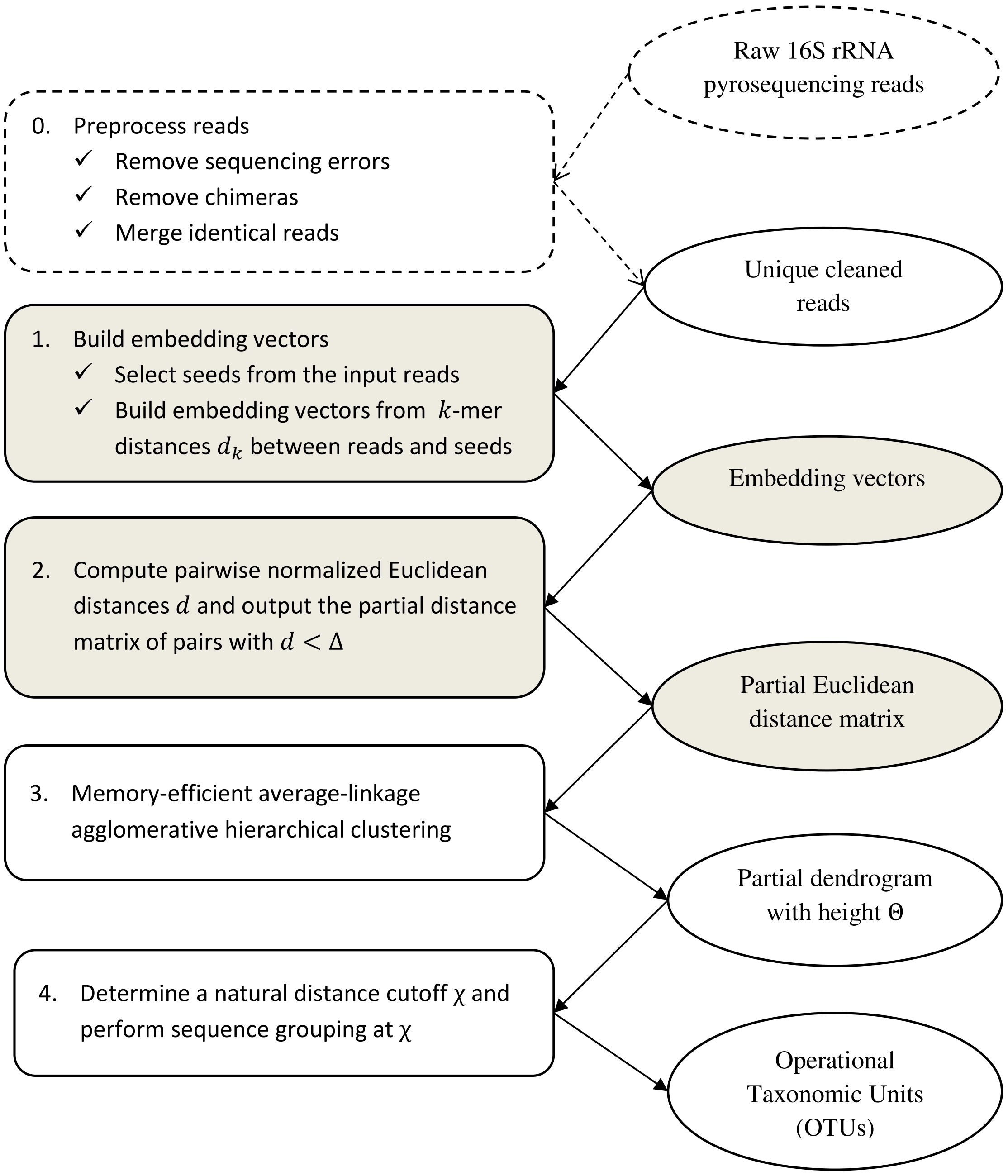
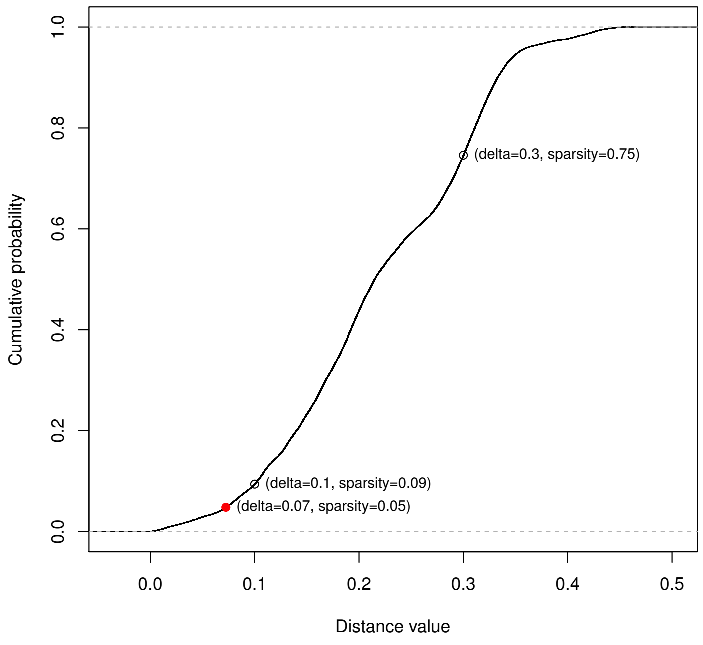
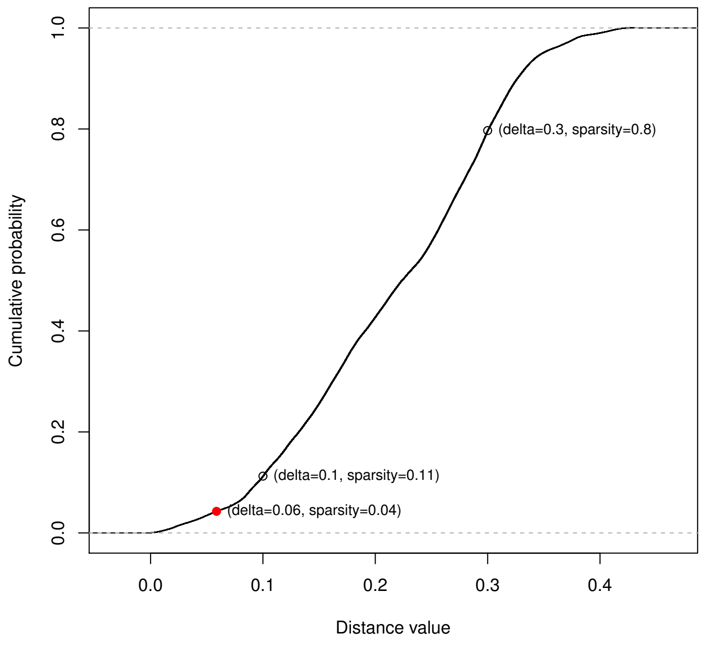

Raw 16S rRNA pyrosequencing reads
0. Preprocess reads
✓ Remove sequencing errors
✓ Remove chimeras
✓ Merge identical reads
Unique cleaned reads
1. Build embedding vectors
✓ Select seeds from the input reads
✓ Build embedding vectors from k-mer distances d_k between reads and seeds
Embedding vectors
2. Compute pairwise normalized Euclidean distances d and output the partial distance matrix of pairs with $d < \Delta$
Partial Euclidean distance matrix
3. Memory-efficient average-linkage agglomerative hierarchical clustering
Partial dendrogram with height Θ
4. Determine a natural distance cutoff χ and perform sequence grouping at χ
Operational Taxonomic Units (OTUs)

FIGURE 5.1: The CRiSPy-Embed processing pipeline: oval boxes indicate data and rectangular boxes represent computation. The dashed rectangular box contains the cleansing steps using external chimera removal and error correction tools.



(a) Dataset with 1000 reads



(b) Dataset with 5000 reads

FIGURE 5.2: The cumulative distribution functions (CDFs) of two matrices computed from two datasets consisting of 1000 and 5000 sequences. At the same distance threshold, the sparsity of the matrix computed from the larger dataset is higher than that computed from the smaller dataset. The red dots mark the cutoff values used in CRiSPy-Embed.

[104, 105]. The input reads are first converted into vectors of real coordinates. The distance matrix is then constructed from pairwise Euclidean distances between these embedding vectors. The alignment-free sequence comparison technique requires $N^2(\log_2 N)^2 + N(\log_2 N)^2 L$ computations instead of $3N^2L^2$ for a dataset with N sequences of average length L. Furthermore, the distance matrix computation module has been parallelized for both multi-core CPU and many-core GPU architectures. As a result, the matrix computation in CRiSPy-Embed only takes 10% - 25% of the total runtime as opposed to 90% - 99% in ESPRIT.

CRiSPy-Embed addresses the memory issue of average-linkage clustering by using the SparseHC clustering algorithm described in Chapter 5 which is faster and more space-efficient than other space-efficient clustering algorithms such as *aveclust* and MC-UPGMA. Moreover, CRiSPy-Embed uses dynamic dendrogram cutting procedure to cut the dendrogram produced by the hierarchical clustering process. This allows us to obtain the final OTU grouping using a more natural cutoff of the dendrogram instead of using a static cutoff value as other OTU binning tools. Our clustering accuracy assessment using various simulated datasets shows that CRiSPy-Embed achieves more accurate clustering outcomes compared to several notable OTU clustering pipelines such as QIIME, USEARCH6, CD-HIT-OTU, ESPRIT-Tree and ESPRIT.

Figure 5.1 shows the processing pipeline of CRiSPy-Embed. The following techniques have been used to reduce runtime and memory requirements of the standard AHC approach:

- *Sparse matrix representation*: CRiSPy-Embed uses the default value $\Delta = \frac{1}{2ln(N)}$ where N is the dataset size because in practice only about 10% of a full matrix is needed to obtain a partial dendrogram of interest. To maintain the same sparsity level of 10%, smaller distance thresholds should be used for larger datasets (see Figure 5.2). Hence, the threshold Δ in CRiSPy-Embed is inversely proportional to the dataset size N.
- *Embedding and parallelization*: CRiSPy-Embed uses the sequence embedding approach to compute a pairwise distance matrix. Furthermore, two

parallel programs have been implemented to support computing the matrix on either a multi-core CPU (Central Processing Unit) or a GPU (Graphical Processing Unit). The OpenMP version on a quad-core CPU with four threads is 2.7x - 2.9x faster than the single-threaded CPU computation. The computation on a Fermi-based GPU is 13x - 15x faster than on a single CPU core.

5.1.2 Sequence embedding

Given an input dataset $R = \{R_1, R_2, \ldots, R_N\}$ consisting of N unique reads (or sequences) over the four-element DNA alphabet $\Sigma = \{\text{A, C, G, T}\}$, the embedding process consists of two steps:

1. *Seed selection:* The input reads R are first sorted by read length in ascending order. M' seeds are then sampled from R at an equal distance of $L = \lfloor N/M' \rfloor$. Following the LLR algorithm by [106], M' is set as $M' = (log_2 N)^2$. An all-against-all comparison is performed amongst these seeds using k-mer distance to eliminate identical seeds and retain a set $S = \{S_1, S_2, \ldots, S_M\}$ of M unique seeds ($M \leq M'$).

2. *Embedding vector construction:* For each input read R_i ($1 \leq i \leq N$), an embedding vector V_i of length M (the number of unique seeds) is constructed. An embedding vector $V_i = (V_{i1}, V_{i2}, \ldots, V_{i(M)})$ is built from the k-mer distances between the read R_i and all the seeds in S i.e. $V_{ij} = d_k(R_i, S_j)$ ($1 \leq i \leq N$, $1 \leq j \leq M$).

   Given a read R_i and a seed S_j of length l_i and l_j and a positive integer k, their k-mer distance is defined as:

$$d_k(R_i, S_j) = 1 - \frac{\sum_{p=1}^{|\Omega|} \min(n_i[p], n_j[p])}{\min(l_i, l_j) - k + 1} \tag{5.1}$$

   where Ω is the set of all substrings over Σ of length k enumerated in lexicographically sorted order and $n_i(p)$ and $n_j(p)$ are the numbers of occurrences of substring number p in R_i and S_j respectively. k should be chosen such that the chance of finding a particular k-mer in an input read is low. This probability depends on two factors: the alphabet size and the average read length. For DNA reads with a small alphabet size ($|\Omega| = 4$), CRiSPy-Embed uses $k = 15$ resulting in $4^{15} \approx 10^9$ possible k-mers. A typical pyrosequencing read is 450 characters long, which is indeed much smaller than one million. Besides, for $k = 15$, a substring 15-mer can be packed into an integer of 32 bits with two bits for each DNA letter.

After the embedding process, the original dataset R of N sequences is converted into the dataset $V = \{V_1, V_2, \ldots, V_N\}$ of N vectors, each of which has M coordinates.

CRiSPy-Embed uses the distance threshold $\Delta = \frac{1}{2ln(N)}$ where N is the dataset size (ESPRIT uses $\Delta = 0.3$ and ESPRIT-Tree uses $\Delta = 0.1$). This default threshold is chosen because in practice only about 5% of a full matrix is needed to obtain a partial dendrogram of interest. To maintain the same sparsity level of 5%, smaller distance thresholds should be used for larger datasets (see illustration in Figure 5.2). Thus, the threshold Δ in CRiSPy-Embed is inversely proportional to the dataset size N.

The parallel distance matrix computation is described in Chapter 4.

5.1.3 Dendrogram construction

CRiSPy-Embed uses SparseHC for space-efficient average-linkage hierarchical clustering of sparse Euclidean distance matrices. Traditionally, hierarchical clustering takes a full pairwise distance matrix D and progressively builds a complete binary tree called a dendrogram. For the average-linkage scheme, the distance between two clusters C_i and C_j is defined as follows:

$$d_{ij} = \frac{\sum_{r \in C_i, s \in C_j} d_{rs}}{|C_i||C_j|} \tag{5.2}$$

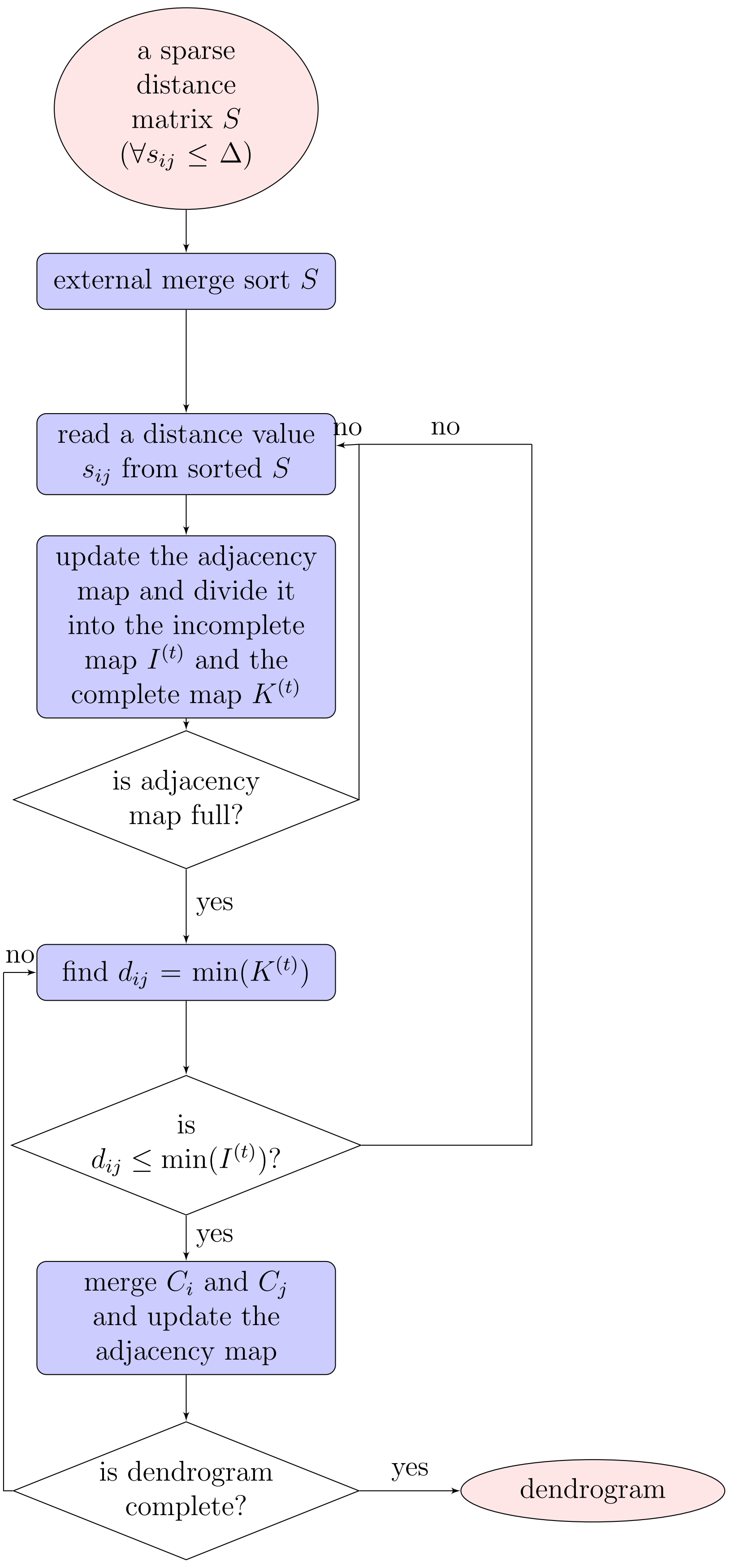

a sparse distance matrix S $(\forall s_{ij} \leq \Delta)$
external merge sort S
read a distance value s_{ij} from sorted S
no
no
update the adjacency map and divide it into the incomplete map $I^{(t)}$ and the complete map $K^{(t)}$
is adjacency map full?
yes
no
find $d_{ij} = \min(K^{(t)})$
is $d_{ij} \leq \min(I^{(t)})$?
yes
merge C_i and C_j and update the adjacency map
is dendrogram complete?
yes
dendrogram

FIGURE 5.3: The flow diagram of the SparseHC algorithm

When two clusters C_i and C_j are merged into a cluster $C_k = C_i \cup C_j$, then the distance between C_k to any other cluster C_m can be calculated as:

$$d_{km} = \frac{d_{im}|C_i| + d_{jm}|C_j|}{|C_i| + |C_j|} \tag{5.3}$$

CRiSPy-Embed uses a sparse distance matrix S which only contains all distances less than Δ instead of storing the full matrix D. In-memory sparse average-linkage clustering can be performed on S by replacing a missing distance by Δ [95].

However, even when using a sparse distance matrix, for large-scale datasets, it is often still not possible to store all elements of the matrix in the main memory. Assuming $N = 10^6$ unique input reads and 5% of pairwise distances are $\leq \Delta$, S would contain $5\% \times (10^6)^2/2 = 25$ billion elements. If each distance element takes 12 bytes (8 bytes to store two indices and 4 bytes to store the distance), it requires $12 \times 25 = 300$ billion bytes = 300 GB just to store the sparse distance matrix.

In order to deal with large sparse distance matrices, CRiSPy-CUDA uses SparseHC to perform memory-efficient out-of-core hierarchical clustering. SparseHC iteratively builds a dendrogram from reading only a part of S in each iteration step. SparseHC is discusses in details in the previous chapter.

Figure 5.3 summarizes the main steps in SparseHC.

1. *Sorting*: The sparse distance matrix is divided into smaller chunks each of which fits into the memory. Each chunk is then sorted and written into an individual file on the hard drive. The sorting module is implemented in parallel for both CPUs and GPUs using the key-value sorting algorithm called *sort_by_key* from the Thrust library [107].

2. *Merging*: The merging module reads all the sorted files on disk and uses the priority queue data structure to merge these files together and passes the results to the clustering module. The sorting and merging are known collectively as *external merge sort* [97].

3. *Clustering*: Depending on the available main memory, a sequence of values $0 = \lambda_0 < \lambda_1 < \ldots < \lambda_T = \Delta$ is built on-the-fly. In each iteration step $1 \leq t \leq T$, all distances s_{ij} with $\lambda_{t-1} \leq s_{ij} \leq \lambda_t$ are read from S. Starting from a tree consisting of only N leaves (where leaf i represents the singleton cluster $C_i = \{i\}$), a binary tree is built from bottom up. A cluster pair is called *active* in iteration step t when (1) both clusters do not have a parent and (2) at least one distance value between the member reads has been read from the input file during the first t iteration steps.

   Consider an active cluster pair (C_i, C_j) in iteration step t. The link $l_{ij}^{(t)}$ is a triple $(sum_{ij}^{(t)}, n_{ij}^{(t)}, d_{ij}^{(t)})$ where $sum_{ij}^{(t)}$ $(n_{ij}^{(t)})$ is the sum (number) of distance values between any member of C_i to any member of C_j that has been read from the input file so far. $d_{ij}^{(t)}$ is the minimum possible distance between C_i and C_j and is computed as follows:

$$d_{ij}^{(t)} = \frac{sum_{ij}^{(t)} + \lambda_t(|C_i||C_j| - n_{ij}^{(t)})}{|C_i||C_j|} \tag{5.4}$$

   A link is defined as *complete* if $n_{ij} = |C_i||C_j|$. Otherwise, it is called *incomplete*. In iteration step t, links are partitioned into a set of complete links $K^{(t)}$ and a set of incomplete links $I^{(t)}$.

   Let $\min(I^{(t)})$ $(\min(K^{(t)}))$ denote the smallest distance value in $I^{(t)}$ $(K^{(t)})$. The clustering procedure in each iteration $t (1 \leq t \leq T)$ involves the following steps:

   (a) Read the distance values s_{ij} from S in ascending order until the adjacency map is full and determine the value $\lambda^{(t)}$.

   (b) Update/create the links for all active cluster pairs with the new distances and partition them into $I^{(t)}$ and $K^{(t)}$.

   (c) Retrieve the link l_{ij} for which $d_{ij} = \min(K^{(t)}) \leq \min(I^{(t)})$. Merge the cluster pair C_i and C_j into C_k. Delete l_{ij} from $K^{(t)}$ and combine existing links to C_i and C_j into new links to C_k. Repeat until $\min(K^{(t)}) > \min(I^{(t)})$.

5.1.4 OTU grouping

Most existing OTU binning tools, for both approaches AHC and GHC, use the de facto cutoff value of 97% sequence similarity. This de facto choice is based on the assumption that the pairwise genetic distance between a pair of full-length 16S rDNA from different species differ by more than 3% and that 97% similarity in 16S rDNA amplicons reflects 3% dissimilarity in the full sequences. When the actual distance distribution does not follow this assumption, a more flexible approach to determine the final OTU grouping is preferred. A recent tool aiming to provide a more dynamic OTU picking functionality is M-pick [51]. M-pick works by detecting groups of edges in a graph where the number of edges within such groups are "significantly higher than expected by chance".

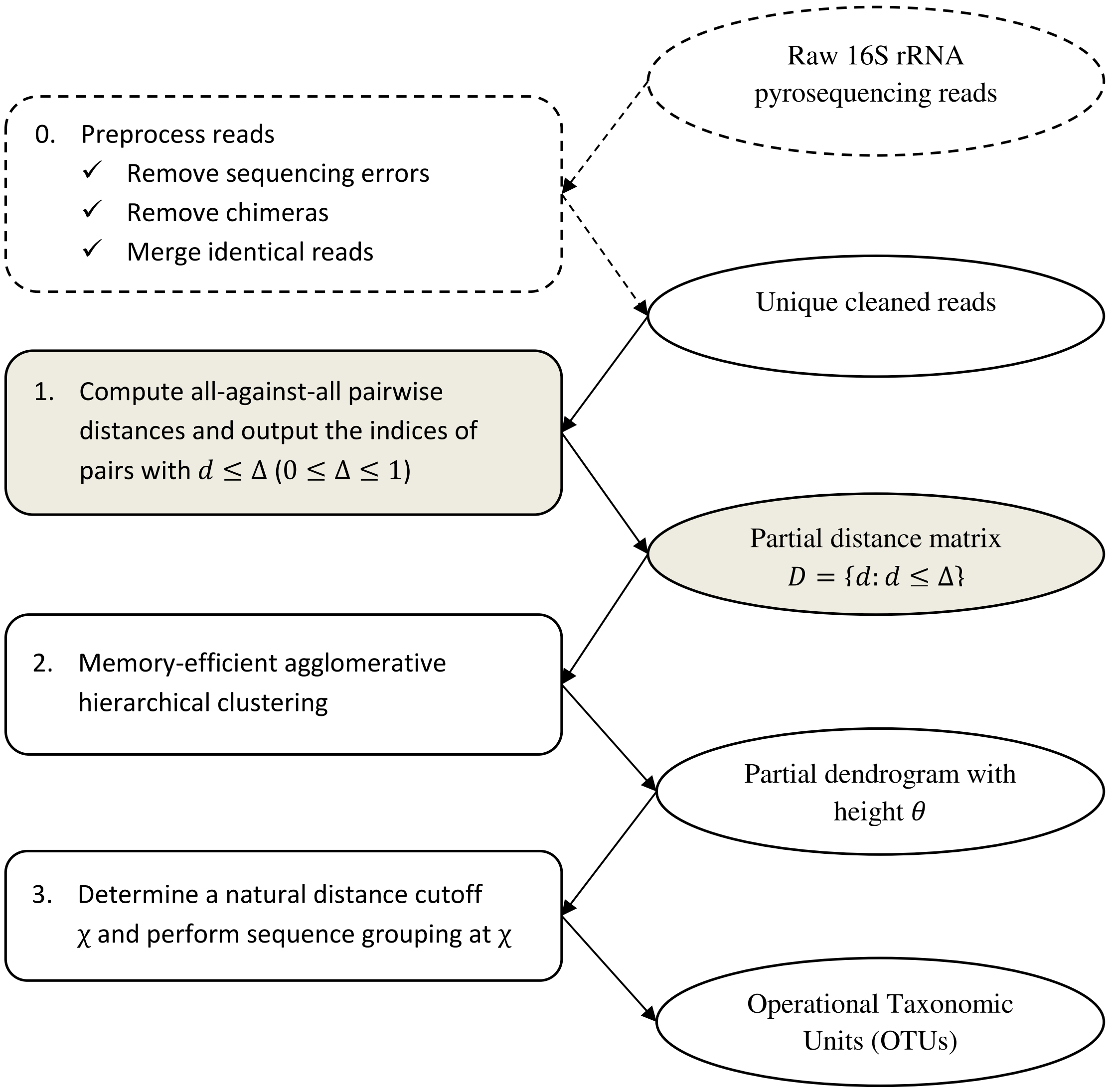
CRiSPy-Embed uses new dendrogram-based method to automatically determine the distance cutoff for a dendrogram. The aim of this method is to find a dynamic cutting method to discover the natural grouping in 16S rDNA short read datasets. The key insight used is that a big jump in the merging distances of a dendrogram indicates the grouping of two intrinsically unrelated clusters. Hence, cutting the dendrogram right before this jump occurs will result in a more natural grouping of data points in a dataset.

A data mining technique called anomaly detection is used to find a natural cut-off point χ. Given the (possibly incomplete) merging sequence produced by SparseHC, CRiSPy-Embed first computes the pairwise differences between every two consecutive merging distances. Subsequently, the change in *variance* of the distance differences is detected using the R *changepoint* package [108]. The merging distance at which the most significant change in variance occurs is the cutoff point χ that is used by CRiSPy-Embed.

5.2 CRiSPy-CUDA

5.2.1 Approach

This section describes an OTU binning pipeline called CRiSPy-CUDA. CRiSPy-CUDA stands for "*C*omputing *Ri*chness in 16*S* *Py*rosequencing Datasets with CUDA" [103]. Figure 5.4 shows the processing pipeline of CRiSPy-CUDA.



Raw 16S rRNA pyrosequencing reads
0. Preprocess reads
✓ Remove sequencing errors
✓ Remove chimeras
✓ Merge identical reads
Unique cleaned reads
1. Compute all-against-all pairwise distances and output the indices of pairs with $d \leq \Delta$ $(0 \leq \Delta \leq 1)$
Partial distance matrix $D = \{d : d \leq \Delta\}$
2. Memory-efficient agglomerative hierarchical clustering
Partial dendrogram with height θ
3. Determine a natural distance cutoff χ and perform sequence grouping at χ
Operational Taxonomic Units (OTUs)

FIGURE 5.4: The CRiSPy-CUDA processing pipeline: oval boxes indicate data and rectangular boxes represent computation. The dashed rectangular box contains the cleansing steps using external chimera removal and error correction tools.

While CRiSPy-CUDA uses the static cutoff value of 0.03, CRiSPy-CUDA has a dynamic method to automatically determine the distance threshold to obtain the natural grouping of a dataset. Due to this dynamic dendrogram cutting, CRiSPy-CUDA produces better grouping results than most existing OTU binning tools including QIIME, CD-HIT-OTU, UPARSE, USEARCH6, ESPRIT-Tree, ESPRIT and CRiSPy-CUDA in terms of the adjusted rand index (ARI) score.

Pyrosequencing datasets often contain sequencing errors and PCR artifacts including mutations (insertions and deletions), homopolymers, and chimeras [96]. Therefore, input reads are preprocessed (e.g. filtered, trimmed, de-replicated, denoised, and chimeras removal) by third-party tools such as AmpliconNoise [109] and UCHIME [110]. As a result, the inputs to CRiSPy-CUDA are preprocessed datasets of unique cleaned 16S rRNA reads.

CRiSPy-CUDA uses the following techniques to reduce runtime and memory requirements of the standard AHC approach:

- *Parallel distance computation*: CRiSPy-CUDA supports parallel distance matrix computation on a multi-core CPU with one or multiple threads, a GPU or a GPU cluster with multiple GPUs with well-designed parallel algorithms that aim to leverage the computational power of the hardware (see Chapter 4.

- *Sparse matrix representation*: For the purpose of species richness estimation, I are often interested in only the lower part of the dendrogram where the species grouping can be found. Since the dendrogram is built bottom-up and only its lower part is required, I can stop building the dendrogram after passing a certain height of interest Θ. As a result, at the end of the clustering procedure I often obtain a partial dendrogram instead of the typical full dendrogram.

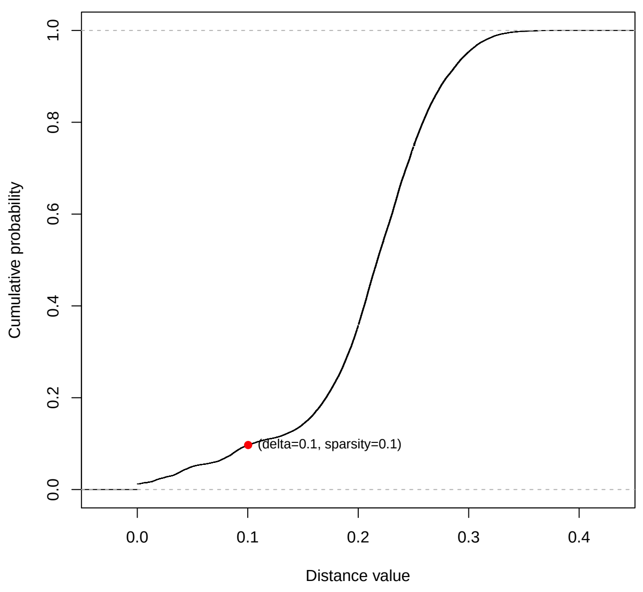
  To build such partial dendrograms, I only need a partial distance matrix, which contains a subset of the full pairwise distance matrix. Keeping partial matrices reduces the processing required in the later steps and allows the use of dendrogram-based methods on larger datasets. Partial matrices for building partial dendrograms consist of distance values smaller than a certain threshold Δ. Please note that *cluster* distance threshold Θ is different from the pairwise *read* distance threshold Δ. Depending on the clustering algorithm, the value Δ is chosen, often by experiments, such that there are sufficient distance values to build a dendrogram up to the height of Θ. The default value of Δ in ESPRIT is 0.3 [44] and in ESPRIT-Tree is 0.1 [45].

- *Memory-efficient sparse matrix clustering*: Even though I use a sparse representation of the calculated distance matrices, typical sizes of large matrices can still exceed the RAM capacity of a standard workstation. Thus, I have developed a fast and memory-efficient hierarchical clustering algorithm for sparse matrices called SparseHC (see Chapter 5). SparseHC scans a sorted sparse distance matrix chunk-by-chunk and a dendrogram is built by merging cluster pairs as soon as the distance between them is determined to be the smallest among all remaining cluster pairs. The main insight used is that it is unnecessary to wait for the completion of all cluster pair distance computations for finding the cluster pair with the smallest distance. Partial information can be used to determine lower bounds on cluster pair distances that are used for cluster distance comparison.

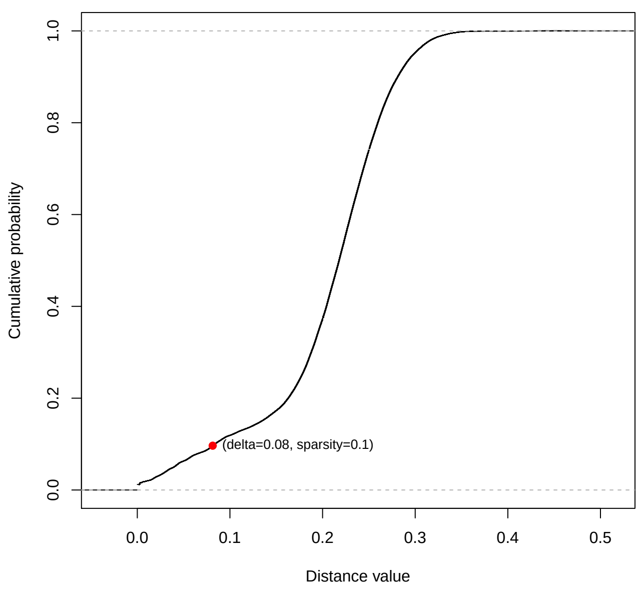
5.2.2 Sequence comparison

I use the following techniques to reduce runtime and memory requirements of the standard AHC approach:

- *Banded sequence alignment and parallelization*: To compute a genetic distance matrix, CRiSPy-CUDA uses a modified dynamic programming formula, which allows efficient banded sequence alignment on parallel computing systems. I have developed two parallel implementations to support the matrix computation on either a multi-core CPU (Central Processing Unit) or a many-score GPU (Graphical Processing Unit).



(a) Dataset with 1000 reads



(b) Dataset with 5000 reads

FIGURE 5.5: The cumulative distribution functions (CDFs) of two matrices computed from two datasets consisting of 1000 and 5000 sequences. The red dots mark the cutoff values used in CRiSPy-CUDA.

The OpenMP version on a quad-core CPU with four threads is on average 4.7x faster than the single-threaded CPU computation by ESPRIT. The computation on a Fermi-based GPU is on average 100x faster than ESPRIT on a single CPU core.

- *k-mer distance filtration and parallelization*: Even with our efficient alignment algorithm, computing the pairwise genetic distance matrix is still costly. Therefore, prior to this step, CRiSPy-CUDA performs pairwise k-mer distance computation to reduce the amount of sequence alignments in the second step. This filtration technique was first introduced by ESPRIT [44] based on the observation of the high correlation between the k-mer distance and the genetic distance [111].

  The OpenMP version on a quad-core CPU with four threads is on average 4.3x faster than the single-threaded CPU computation by ESPRIT. The computation on a Fermi-based GPU is on average 50x faster than ESPRIT on a single CPU core.

- *Sparse matrix representation*: In CRiSPy-CUDA, I use the default value $\Delta = \frac{1}{log_2(N)}$ where N is the dataset size because I observe that only about 10% of a full matrix is needed to obtain a partial dendrogram of interest. To maintain the same sparsity level of 10%, smaller distance thresholds should be used for larger datasets (see Figure 5.5). Hence, the threshold Δ in CRiSPy-CUDA is inversely proportional to the dataset size N.

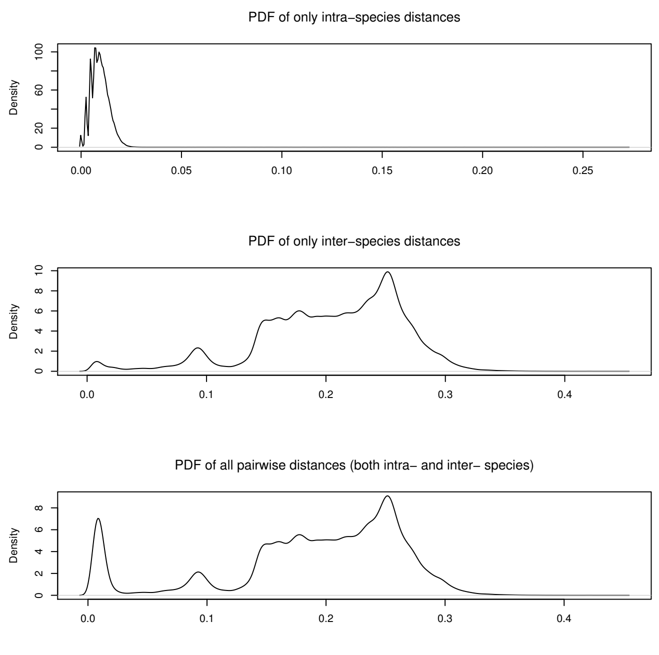
5.2.3 Dendrogram construction

The hierarchical clustering in CRiSPy-CUDA is a procedure of shaping an incomplete binary tree (i.e. a partial dendrogram), where each leaf node represents a unique read. Since most existing hierarchical clustering tools require full distance matrices as inputs, I develop our own clustering algorithm called SparseHC to target sparse distance matrices. SparseHC scans a sorted (possibly sparse) distance matrix chunk-by-chunk and a dendrogram is built by merging cluster pairs as and when the distance between them is determined to be the smallest among all remaining cluster pairs. The key insight used is that for finding the cluster pair with the smallest distance, it is unnecessary to wait for completing the computation of all cluster pair distances. Partial information can be used to determine a lower bound on cluster pair distances used for cluster distance comparison.

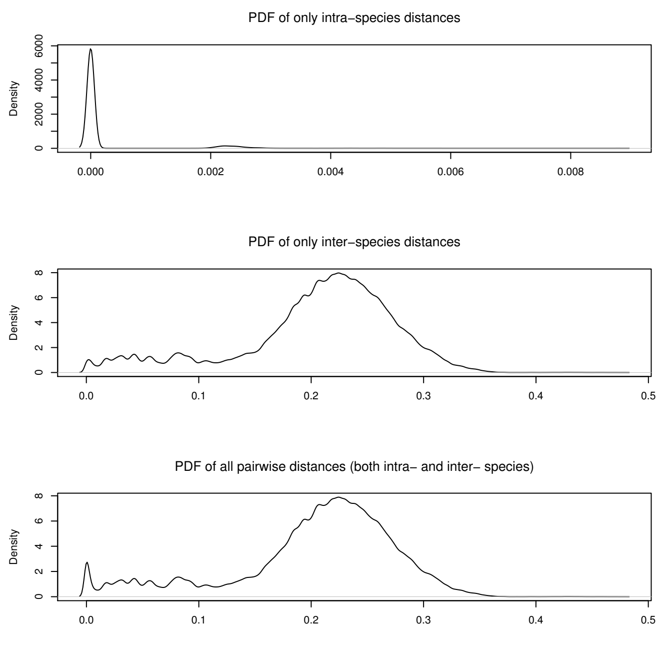
SparseHC achieves a linear empirical memory complexity, which is a significant improvement compared to other algorithms for the same purpose. SparseHC supports memory-efficient single-linkage, complete-linkage and average-linkage hierarchical clustering for both sparse and full matrices. Prior to SparseHC, the average-linkage scheme cannot be used on large datasets due to memory usage restriction albeit it has been shown in several benchmark studies to provide better clustering results than the single-linkage and complete-linkage schemes [38, 47]. In CRiSPy-CUDA, I also use the complete-linkage scheme for building the dendrogram. In CRiSPy-CUDA, I use the average-linkage scheme for better dendrogram building thanks to SparseHC.

5.2.4 OTU grouping

The OTU binning approach works based on an underlying assumption that a full-length 16S rDNA (or in some cases, a region or a group of regions on the full-length sequence) is representative of a particular species. Static grouping methods further assume that the pairwise genetic distance between a pair of 16S rDNA short reads from the same full-length rDNA sequence (i.e. from the same species) is less than 0.03. Although most tools allow users to set this threshold parameter before running the tool, the value of 0.03 or 97% is most commonly used unless a user has other reasons to change this assumption and pick another value. This assumption is reasonable in cases where the distances between 16S read pairs from the same species are at most 0.03 and the distances between pairs from different species are larger than 0.03. Most existing OTU binning tools such as UPARSE, CD-HIT-OTU, ESPRIT-Tree, etc. use this de facto threshold.

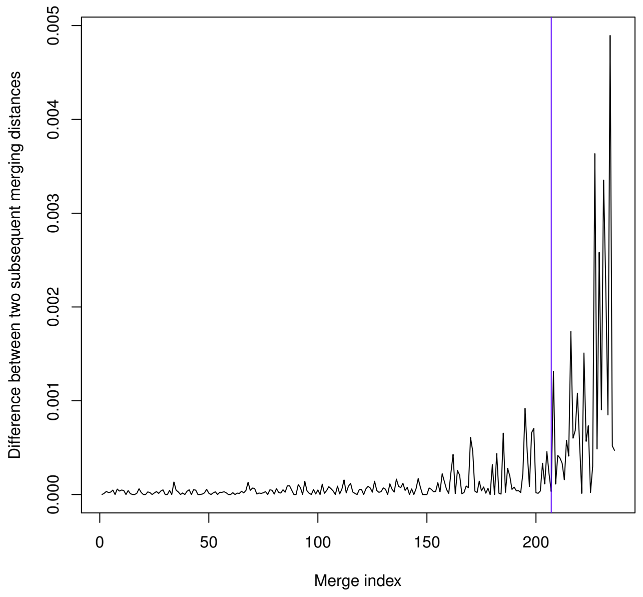


(a) dataset with errors

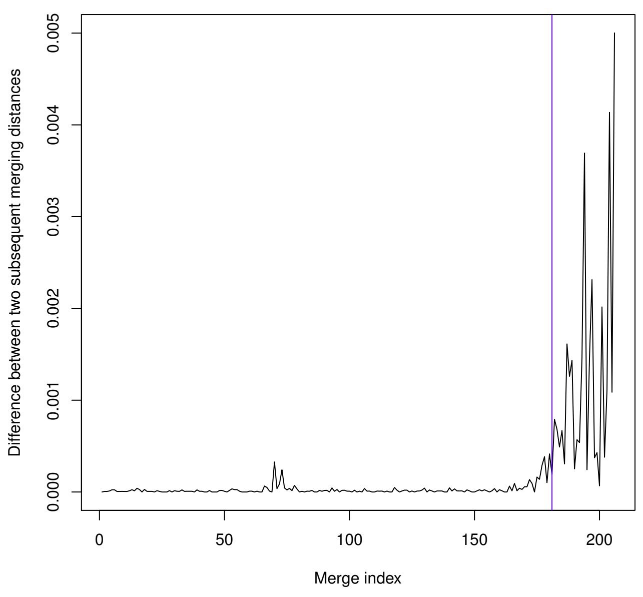


(b) dataset without errors

Figure 5.6: The probability distribution functions (PDFs) of pairwise distance matrices for two typical datasets: a raw dataset that contains errors on the left and a cleaned dataset that contains few errors on the right.



(a) dataset with errors



(b) dataset without errors

FIGURE 5.7: Determination of the natural distance cutoffs of two datasets with different characteristics by detecting the first sudden change merging distances of a dendrogram. These are same datasets as in Figure 5.6.

Several recent studies [38] have questioned the generality of the default value when using on datasets with different characteristics. In practice, there are two common cases where this assumption might not hold true. Firstly, two 16S full-length sequences from two different species share a higher or lower resemblance than 97% sequence distance. Subsequently, this assumption causes an under- or over-estimation of the biodiversity of a dataset. Secondly, this assumption is based on the specific definition of genetic distance for comparing reads. In the literature, there are many ways of computing sequence similarity/distance depending on the type of sequence comparison used including alignment-based methods e.g. local, global, glocal sequence alignment or alignment-free methods e.g. k-mer distance based on word counting [112]. Under these circumstances, a more robust/dynamic way for identifying the distance threshold is preferable to the conventional value of 0.03. Recently, Wang et al. [51] have developed M-pick to provide a more dynamic way to address this issue. This method works well for datasets with low diversity. For datasets with higher diversity (100 species or more), M-pick faces the problem of underestimation.

Figure 5.6 illustrates a common case where the de facto choice of 0.03 pairwise genetic distance for OTU grouping at the species level is suboptimal by showing the effect of errors on the distribution of inter-species and intra-species pairwise distances. Each full pairwise distance matrix (bottom sub-figures) can be decomposed into two sets: the set of intra-species distances (top sub-figures) and the set of inter-species distances (middle subfigures). The aim of dynamic dendrogram cutting is to find a distance threshold that best separates these two distance sets.

The aim of dynamic dendrogram cutting is to find a distance threshold that best separates the intra-species distance set from the inter-species set, that is to group sequences from the same species together. This figure shows that for cleaned datasets a smaller distance threshold than 0.03 is often more appropriate to group short reads from the same species and differentiate reads from different species. Potential factors that can affect the distribution of pairwise distances include those directly alter the DNA sequence of each read, directly affecting the distances of that read to other sequences. Besides PCR and sequencing errors, other community-related characteristics such as abundance, coverage and diversity can also affect the distribution of pairwise distances.

In CRiSPy-CUDA, I use a new dendrogram-based method to automatically determine the distance threshold to cut a dendrogram. The aim of this method is to discover the natural grouping in 16S rDNA short read datasets. The key insight used is that a big jump in a dendrogram indicates the grouping of two intrinsically unrelated clusters. Hence, cutting the dendrogram right before this jump occurs will result in the natural grouping of data points in a dataset. The main issue is to formulate the definition of big jump. A straight forward way to determine these big jumps is to take the absolute differences between two consecutive merging distances and then simply take the maximum difference. However, this method is susceptible to noises and outliers. Instead, I apply a technique called change point detection.

In CRiSPy-CUDA, given the (possibly incomplete) merging sequence produced by SparseHC, I first compute the sequence of differences between every pair of consecutive merging distances. I then detect the change in *variance* of the difference sequence using the R *changepoint* package [108]. Figure 5.7 illustrates the detected change points in two different merging sequences with different statistical characteristics produced by CRiSPy. The natural cutoff distance of the dataset with errors is 0.0350, which corresponds to the difference of 0.0013 (indicated by the vertical line in the figure on the left). The cutoff of the dataset without errors is 0.0073, which corresponds to the difference of 0.0008 (indicated by the vertical line in the on the right).

The *changepoint* package consists of several search algorithms used to identify the points at which changes in the probability distribution of a time series occur. Other R packages in the category of multiple change point search algorithms are *cumSeg* [113] and *ecp* [114]. While most change point detection packages provide only a particular search method, the *changepoint* package implements a variety of popular search algorithms and test statistics such as the binary segmentation algorithm, the segment neighborhood algorithm and the PELT algorithm. I employ this assortment of algorithms to find the best test statistics and search algorithm to detect the change points in the merging sequence produced by the clustering procedure of CRiSPy.

The particular *changepoint* method used in CRiSPY is described as follows. Given the merging sequence $\Theta = (\Theta_1, \Theta_2, \ldots, \Theta_n)$ ($n \leq N - 1$ where N is the number of input reads), a change point τ is a value in the range $[1, n]$ such that the statistical properties of $\Theta_1 = (\Theta_1, \ldots, \Theta_\tau)$ and $\Theta_2 = (\Theta_{\tau+1}, \ldots, \Theta_n)$ are significantly different. Statistical properties including the mean and/or variance of the distributions of Θ_1 and Θ_2 are summarized in the sets of parameters $\hat{\Theta}_1$ and $\hat{\Theta}_2$ correspondingly. The single change point detection problem can be posed as a hypothesis test. The null hypothesis H_0 is that there is no change point in the merging sequence. The alternative hypothesis H_1 is that there is one change point τ in the merging sequence. I can then compute the test statistic to reject the null hypothesis using the maximum likelihood approach:

$$\begin{aligned}\Lambda &= -2\log(\frac{\text{likelihood for null hypothesis}}{\text{likelihood for alternative hypothesis}}) \\ &= 2(\text{log likelihood for alternative hypothesis} \\ &- \text{log likelihood for null hypothesis}) \\ &= 2\{\max_{\tau}(\log p(\Theta_1|\hat{\Theta}_1) + \log p(\Theta_2|\hat{\Theta}_2)) - \log p(\Theta|\hat{\Theta})\}\end{aligned}$$

If Λ is larger than a certain threshold c, I can reject the null hypothesis and conclude that there exists a change point τ. The value of τ is then estimated such that the log likelihood $(\log p(\Theta_1|\hat{\Theta}_1) + \log p(\Theta_2|\hat{\Theta}_2))$ is maximized.

5.3 Summary

This research introduces CRiSPy-CUDA and CRiSPy-Embed - two scalable pipelines for de novo OTU clustering of large 16S rRNA pyrosequencing datasets. Using a workstation with a single CUDA-enabled GPU, CRiSPy-CUDA and CRiSPy-Embed can efficiently perform biodiversity estimation of large metagenomic datasets.

Based on algorithms which are designed for massively parallel CUDA-enabled GPU, CRiSPy-CUDA and CRiSPy-Embed achieve significant speed-up over the same serial pipeline.

With the recent development in bioinformatics, the CRiSPy pipelines can have many other real-world applications besides biodiversity assessment for metagenomic datasets. One related application is in plant systematics to generate artificial phylogenies of plants at the species, genus or higher level. Another bioinformatic application is in transcriptomics to group genes with similar expression patterns or in sequence analysis to group homologous sequences into gene families. They can also be used to cluster human genetic data to infer population structures. Since large-scale genomic datasets become more and more accessible to scientists, scalable yet accurate tools like CRiSPy-CUDA and CRiSPy-Embed are crucial for research in this area.

CRiSPy-CUDA is available at `https://github.com/ngthuydiem/crispy-cuda` and CRiSPy-Embed is available at `https://github.com/ngthuydiem/crispy-embed`.

Chapter 6

OTU Clustering Evaluation

Similar to general cluster analysis, benchmarking OTU clustering algorithms and tools is a nontrivial problem itself due to the lack of benchmark datasets with groundtruth information, especially for NGS metagenomic datasets in which many organisms are sequenced for the first time. This chapter describes the methodology taken in this research to reasonably compared the accuracy of the clustering algorithms of interest. The work described in this chapter has been published in [102] and [103].

6.1 Benchmark Framework

6.1.1 Simulated test datasets

I simulate 16S rDNA amplicon read datasets using Grinder [115] to obtain a more comprehensive benchmark on different datasets with various controlled parameters. In the context of this study, I focus on 454 pyrosequencing datasets. Test datasets are simulated using full-length 16S reference sequences and universal primers for 16S rRNA. I obtain the reference sequences from the Greengenes database [116] and two universal primers for 16S rDNA: 926F (AAACTYAAAK-GAATTGACGG) and 1492R (CGGTTACCTTGTTACGACTT).

In these experiments, I have varied the following attributes: read length, diversity, fold coverage, rank abundance, dataset size, sequencing and PCR error model.

Read length is simulated with three variables: the average length, the distribution, and the standard deviation. Following the recommendation by Grinder for 454 datasets, I use the average length of 450 bp, the normal distribution and the standard deviation of 50 bp for all test datasets.

Diversity of a simulated dataset refers to the number of full-length reference sequences used to simulate that dataset. For small datasets with 5000 reads, I use 100 reference sequences as in a 16S rRNA case study of Grinder [115]. For larger datasets, I use diversity values ranging from 500 to 5000 species depending on the dataset size.

Fold coverage is defined as the total number of base pairs in an output simulated dataset divided by the total number of base pairs of the input reference sequences. When I fix the average read length and the diversity, the fold coverage is determined by the number of output reads. In these experiments, I control the fold coverage by choosing the desired number of reads instead.

Rank-abundance is used to specify how rarity or commonness of a species in relativity to other species in the same community. Except for a particular experiment to study the impact of different rank-abundance models, I use the power law distribution i.e. $P(n) = n^{\alpha}$ with $\alpha = 0.1$ to model the relative species richness in these test datasets since the power law rank-abundance is often observed in real-world 16S rDNA datasets [31, 117].

Errors refer to sequencing and PCR errors including mutations, homopolymers and chimeras.

- Mutations are simulated by introducing substitutions, insertions and deletions at positions that follow a specified distribution. In these experiments, I use the constant mutation rate of 0.1% i.e. 1 out of 1000 base pairs is a mutation.

- Homopolymers are introduced by adding homopolymeric stretches to using a specified model where the length of each stretch follows a normal distribution. As recommended by Grinder, I use the Balzer model [115, 118].
- Chimeras refer to reads generated from the fusion of two or more amplicon template sequences. In the test datasets that contain chimeras, I specify that 10% of amplicon templates are chimeric.

In consideration of the fact that the preprocessing step is often recommended for raw datasets before OTU binning [96], most of these test datasets are cleaned datasets with little errors. Except for a particular experiment to test the robustness of OTU binning tools in the presence of errors, I experiment with the assumption that 90% sequencing and PCR errors are removed during the preprocessing stage.

6.1.2 Mock datasets

Eight mock communities are also used to validate the clustering results. Two V5 datasets were sequenced from a library of 94 16S rRNA clones [119]: "Divergent" was sequenced from 23 clones that differed at least by 7% mixed in equal proportions while "Artificial" was sequenced from 90 clones some of which differed by only 1% and mixed in uneven proportions to simulate a real-world community. Six V2 datasets were extracted from a pool of 87 isolated organisms [6]: "Even1",

TABLE 6.1: Mock communities used for validating OTU clustering accuracy (# means "the number of")

Dataset	#original reads	#labeled reads	#cleaned, unique reads	average length	#species	#multi-labeled OTUs	#single-labeled OTUs
Divergent	35190	34905	13,364	268	23	22	22
Artificial	31867	31765	10,790	269	90	117	48
Even1	53,771	49,006	7,144	250	87	133	70
Even2	45,178	41,104	5,995	248	87	124	70
Even3	54,153	49,588	6,996	245	87	133	73
Uneven1	44,926	39,369	5,583	247	87	92	62
Uneven2	44,176	41,914	5,868	254	87	114	58
Uneven3	50,931	48,365	7,098	243	87	105	55

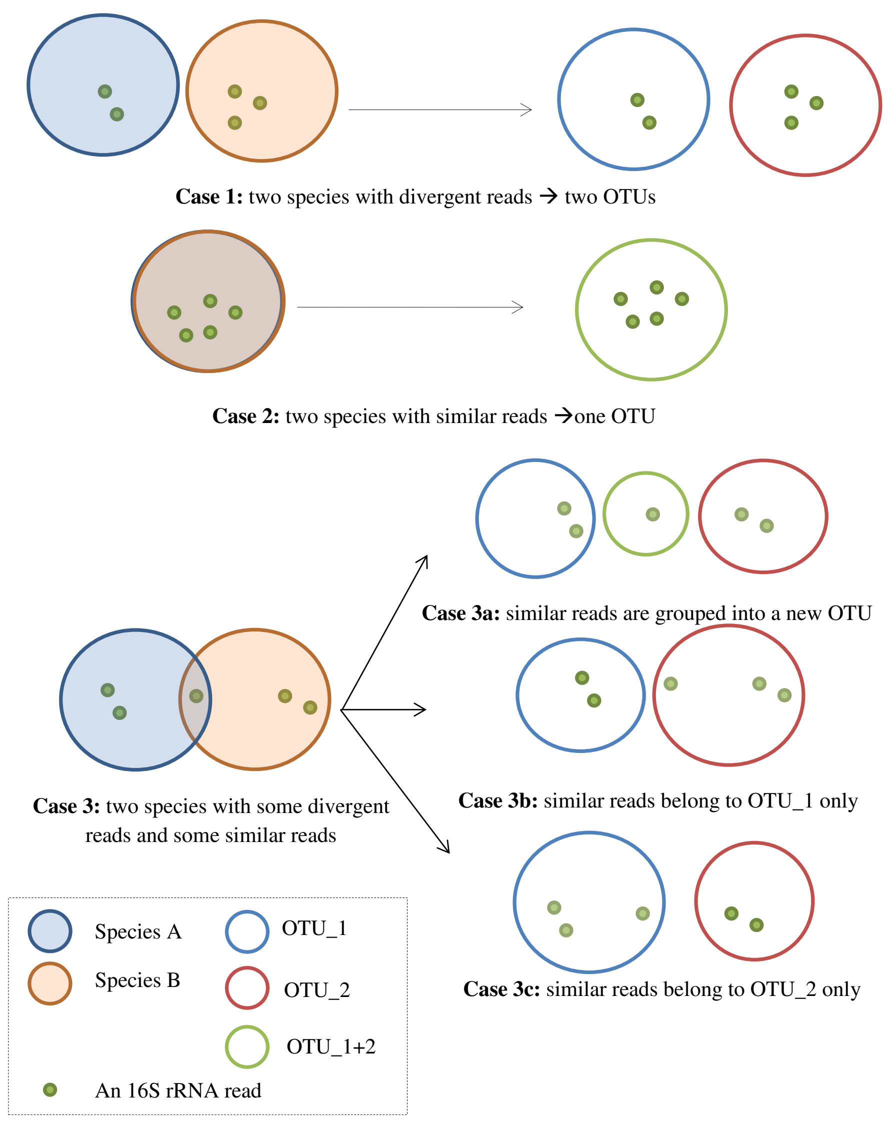

Case 1: two species with divergent reads → two OTUs
Case 2: two species with similar reads →one OTU
Case 3a: similar reads are grouped into a new OTU
Case 3: two species with some divergent reads and some similar reads
Case 3b: similar reads belong to OTU_1 only
Case 3c: similar reads belong to OTU_2 only
Species A
Species B
OTU_1
OTU_2
OTU_1+2
An 16S rRNA read

FIGURE 6.1: Complications in labelling mock datasets to evaluate clustering accuracy.

"Even2", "Even3" were mixed in equal proportions and "Uneven1", "Uneven2", "Uneven3" were mixed in uneven proportions. These datasets are all labeled and preprocessed before input to the OTU clustering tools. The characteristics of these mock datasets are summarized in Table 6.1.

Clustering is knowledge discovery process to identify patterns in unlabeled datasets. However, to evaluate the accuracy of a clustering algorithm, I need the *ground truth* or labeled datasets. Obtaining the ground truth for a 16S rRNA dataset can be a tricky issue. It is often impossible to know the actual *taxa assignment* (e.g. Read 1 is from Species A, Read 2 is from Species B) of a 16S segment since it might belong to more than one taxa (species, genus, etc.) as illustrated in Figure 3 (Case 2 and 3). From a sequenced dataset, I can only determine the *OTU assignment* of a 16S segment (e.g. Read 1 is from Species A hence assigned to OTU_1, Read 2 may be from either Species A or B or both and hence assigned to OTU_1+2). As a result, the OTU assignment is used as the ground truth instead of the taxa assignment.

I determine the OTU assignment of a read by using *USEARCH -usearch_global* [39] to blast a mock dataset against its corresponding reference dataset with the stringent criteria of minimum 97% identity. After blasting, I retain about 90%-99% the original datasets. One read may be annotated with multiple labels due to the fact that several species or several full-length 16S rRNA clones might have identical 16 rRNA segments thus producing multiple BLAST hits.

For sequences that are common among multiple species in Figure 3 - Case 3, I can either consider that they belongs to a new OTU (*OTU_1+2* in Case 3a) or consider that they belongs to one of the original OTUs (*OTU_1* in Case 3b or *OTU_2* in Case 3c). In the former case, I have multi-labeled OTUs as ground truth and in the latter case, I have single-labeled OTUs as ground truth. The expected numbers of single-labeled and multi-labeled OTUs in the mock datasets are shown in the last two columns of Table 6.1.

It is important to remember that due to the complications when determining the ground truth, I can only measure how good a clustering result is compared to the ground truth label generated from BLAST results but not the actual ground truth. Thus, the following accuracy evaluation is a reasonable but by no means an exact indication to how good an clustering algorithm is.

6.1.3 Accuracy measures

There are many existing measures to validate clustering results. 22 of these measures are summarized in [120]. Among these measures, the corrected-by-chance measures such as the adjusted mutual information (AMI) or the adjusted rand index (ARI) measures are recommended over the equivalent uncorrected indices such as the mutual information index (MI) or the Rand index (RI) [120, 121] due to two reasons. Firstly, unadjusted measures are affected by the number of clusters in the results while adjusted measures are not [121]. Secondly, adjusted measures provide a constant baseline (ideally the baseline value is 0) to indicate no similarity for two clusterings sampled independently at random while unadjusted measure often provide a high and non constant value. Hence, although several OTU clustering studies [45, 47, 96] have used the NMI score to measure OTU clustering accuracy, I recommend the ARI or AMI score for validation.

Normalized Mutual Information

Normalized Mutual Information (or NMI) [49] score is defined as follows:

$$NMI(\varOmega, C) = \frac{2MI(\varOmega|C)}{H(\varOmega) + H(C)}$$

where $\varOmega = \omega_1, \omega_2, ..., \omega_K$ is the set of ground truth clusters and $C = c_1, c_2, ..., c_J$ is the set of clustering outcomes.

TABLE 6.2: The simplified contingency table

Number of pairs	same label	different labels
same cluster	a	b
different clusters	c	d

Given a dataset of N raw sequences, the *entropies* $H(\Omega)$ and $H(C)$ are computed as:

$$H(\Omega) = -\sum \frac{|\omega_k|}{N} \log_2 \frac{|\omega_k|}{N}, H(C) = -\sum \frac{|c_j|}{N} \log_2 \frac{|c_j|}{N}$$

where $|\omega_k|$ is the number of sequences in cluster ω_k .

The mutual information $MI(\Omega|C)$ can be computed as follows:

$$MI(\Omega|C) = H(\Omega) + H(C) - H(\Omega|C)$$

where $H(\Omega|C)$ is the *conditional entropy* of Ω on C:

$$H(\Omega|C) = -\sum_k \sum_j \frac{|\omega_k \cap c_j|}{N} \log_2 \frac{|\omega_k \cap c_j|}{N}$$

$|\omega_k \cap c_j|$ denotes the number of sequences that are present in both clusters ω_k and c_j.

Adjusted Mutual Information

AMI score is based on normalized mutual information score (NMI) (used in [47], [96]) with adjustment for the chance that two data objects are randomly in the same cluster, hence is a more appropriate measure than the NMI score to evaluate clustering accuracy. AMI score attains a minimum value of 0 when the clustering result is totally different from the ground truth and a maximum value of 1 when the clustering result is exactly the same as the ground truth. A higher AMI score indicates a higher accuracy of a clustering algorithm. For the detailed formula of AMI score, see [121].

Adjusted Rand Index

ARI is the corrected for chance version of the simpler Rand index [122] which measures the percentage of decisions that are correct:

$$RI = \frac{a+d}{a+b+c+d} = \frac{a+d}{\binom{n}{2}} \tag{6.1}$$

where a, b, c, d are defined in Table 6.2 and n is the dataset size. a and d denote two types of correct decisions. a counts all true positive decisions that assign two reads from the same reference sequence to the same cluster and d counts all true negative decisions that assign two reads from two reference sequences to different clusters. Meanwhile, b and c denote two types of errors. b counts all false positive decisions that assign two reads from two reference sequences to the same cluster and c counts all false negative decisions that assign two reads from the same reference sequence to different clusters.

Like other measures for clustering validation, ARI aims to measure the similarity between two groupings: the actual grouping (the labels) and the predicted grouping (the clustering results). I want to measure the similarity between the grouping produced by an OTU clustering pipeline and the actual grouping of the input reads. I use the identity of original full-length reference sequence from which a read is extracted as the label of that read. ARI score is defined as:

$$ARI = \frac{\binom{n}{2}(a+d) - [(a+b)(a+c)+(b+d)(c+d)]}{\binom{n}{2}^2 - [(a+b)(a+c)+(b+d)(c+d)]} \tag{6.2}$$

A random labeling in which the predicted grouping is very different from the actual grouping results in an ARI score close to 0.0 while a perfect labeling has a score of 1.0.

6.2 Evaluation of clustering accuracy

In this section, I evaluate the accuracy and efficiency of CRiSPy-CUDA and CRiSPy-Embed against M-pick - a *dynamic* OTU binning tool together with other state-of-the-art *static* OTU binning tools including CD-HIT-OTU v0.0.2, UPARSE (a part of USEARCH v7.0 package), USEARCH v6.1, QIIME *pick_otus* v1.8, ESPRIT (2011 distribution), and ESPRIT-Tree (2011 distribution). All static tools work with the assumption that the genetic distance between a pair of 16S rDNA short reads from the same species is less than 0.03. Although these tools allow users to set the distance threshold parameter before clustering, the value of 0.03 or 97% is most commonly used unless there are reasons to change this assumption and pick another value before running a tool. In these experiments, I use the distance cutoff of 0.03 for these static tools as in their default settings.

All experiments are conducted on a 64-bit Ubuntu operating system using an HP ENVY 700 PC with Intel Core i7-4790 3.6 GHz processor and 8GB of RAM. CRiSPy-CUDA and CRiSPy-Embed run on an NVIDIA GTX 745 graphic card with CUDA 6.5 toolkit installed on the same PC. The runtime is measured using the Linux *time* command and the peak memory usage is measured with the Valgrind Massif profiler [99]. To minimize statistical influence, every experiment is carried out multiple times and the result from each run is stored in a MySQL database for later retrieval and reporting purposes. Each measurement (such as runtime, memory usage or clustering accuracy) reported in this section is the average value computed from multiple runs.

6.2.1 On simulated datasets

In the first experiment, I compare the overall performance of all tools in terms of clustering accuracy, runtime and memory usage. This experiment runs five times on five different datasets with typical characteristics of cleaned 16S amplicon datasets: 5000 pyrosequencing amplicon reads from 100 reference sequences with few sequencing errors. I report the means (bars) and standard deviations (whiskers) of each parameter from five runs in Figure 6.2. Asterisks above each box plot of an existing tool indicate the level of significance (p-value) of the statistical hypothesis test to show that the mean scores achieved by CRiSPy-CUDA and

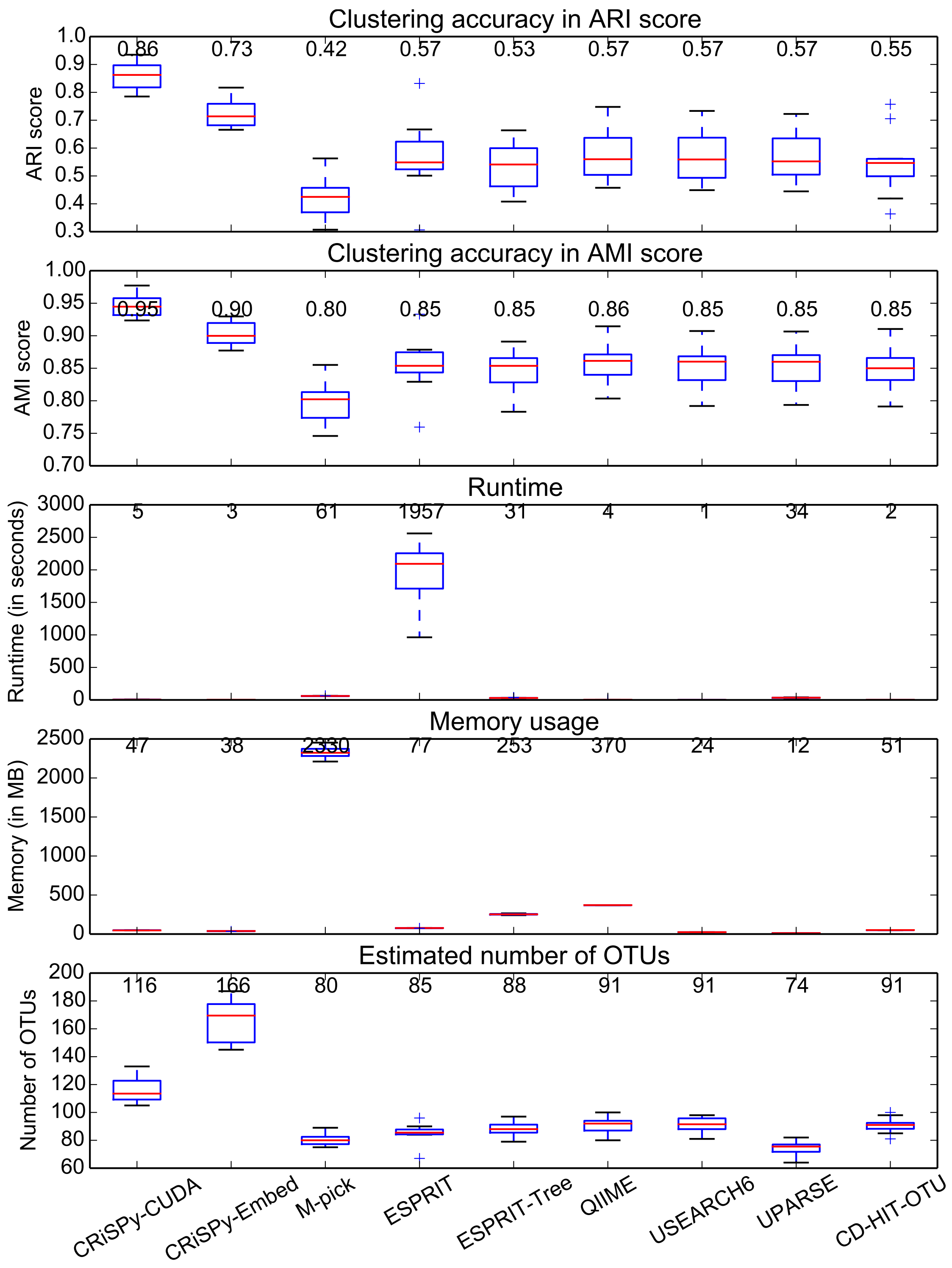


FIGURE 6.2: The benchmark of computational performance and accuracy of CRiSPy-CUDA and CRiSPy-Embed against other OTU binning tools using 10 different datasets, each of which contains 5000 16S rDNA reads that are generated randomly from 100 species. The bars and whiskers represent the means and standard deviations from 10 runs on these 10 datasets.

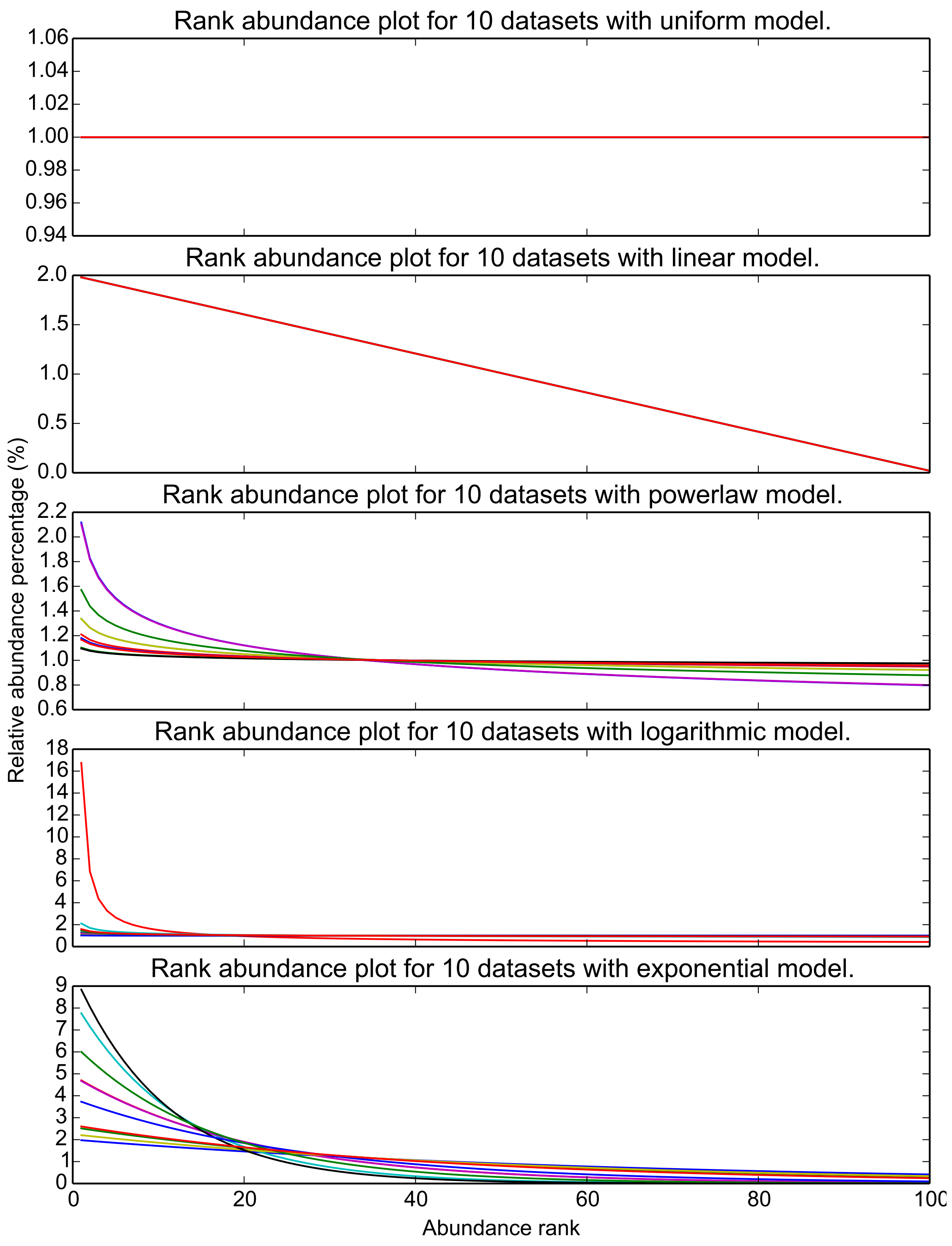


FIGURE 6.3: Different rank abundance models used for benchmarking: exponential, logarithmic, power law, linear and uniform distributions.

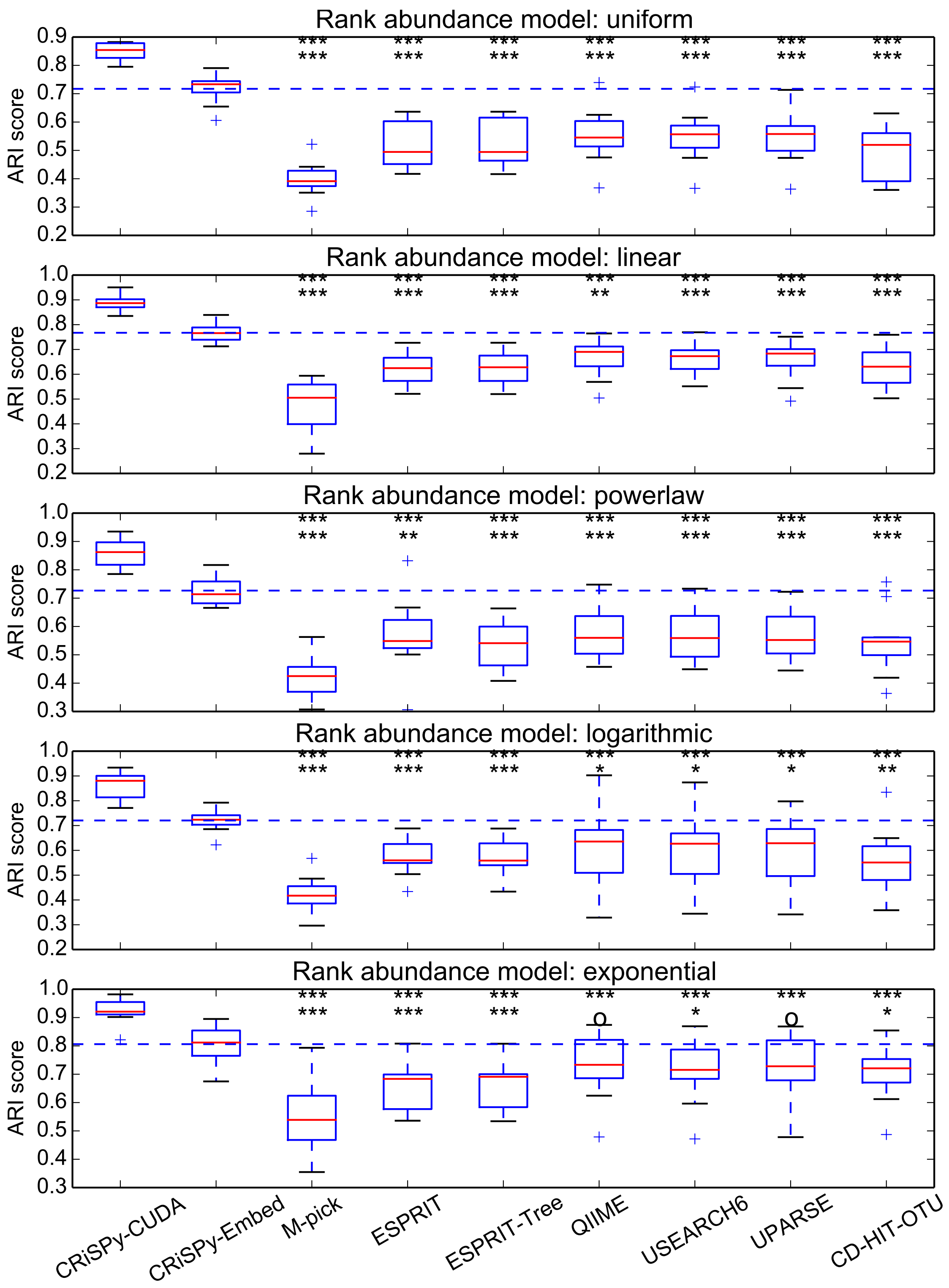


FIGURE 6.4: ARI scores achieved by the OTU binning tools of interest when clustering datasets with the rank abundance models shown in Figure 6.3. Each dataset contains 5000 reads and is simulated from 100 full-length 16S rDNA reference sequences chosen randomly from the Greengenes database.

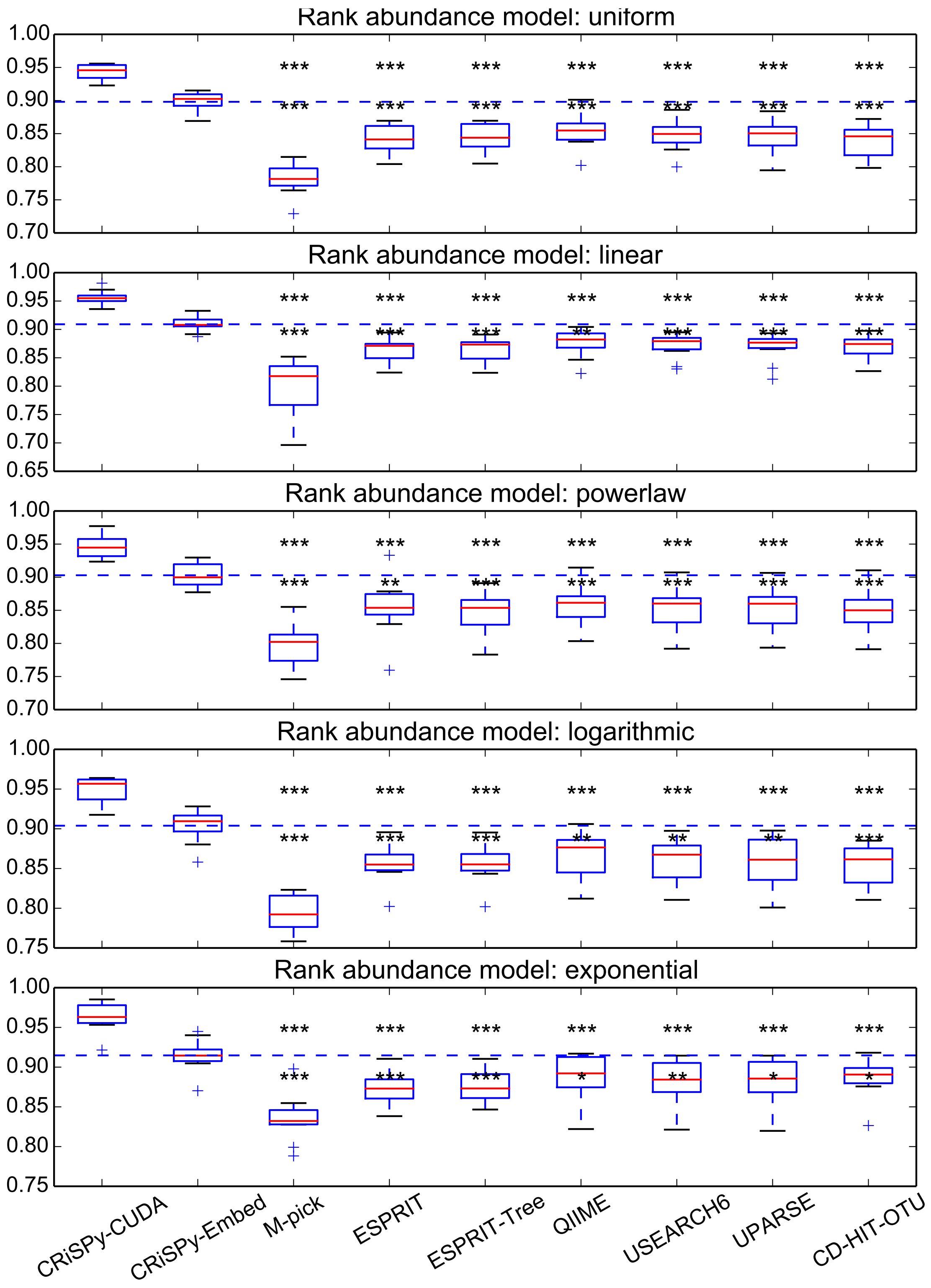


FIGURE 6.5: AMI scores achieved by the OTU binning tools of interest when clustering datasets with different rank abundance models. These AMI scores are computed using the same clustering results as the ARI scores in Figure 6.4. Both accuracy measures are reported for comparison purpose.

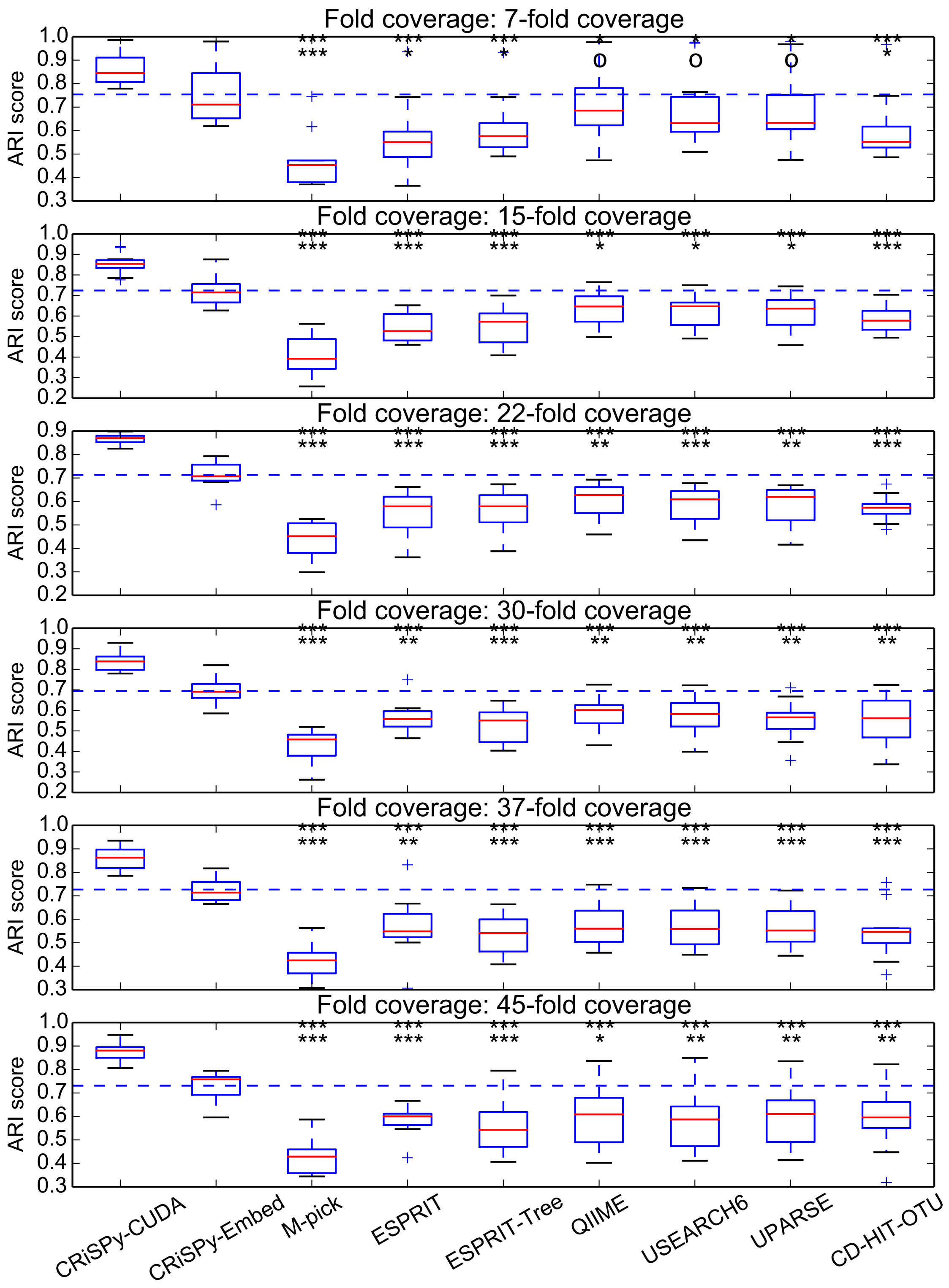


FIGURE 6.6: ARI scores achieved by the OTU binning tools of interest when clustering datasets with fold coverage values: 7 (1000 reads), 15 (2000 reads), 22 (3000 reads), 30 (4000 reads), 37 (5000 reads) and 45 (6000 reads). Each dataset is simulated from 100 full-length 16S rDNA reference sequences chosen randomly from the Greengenes database.

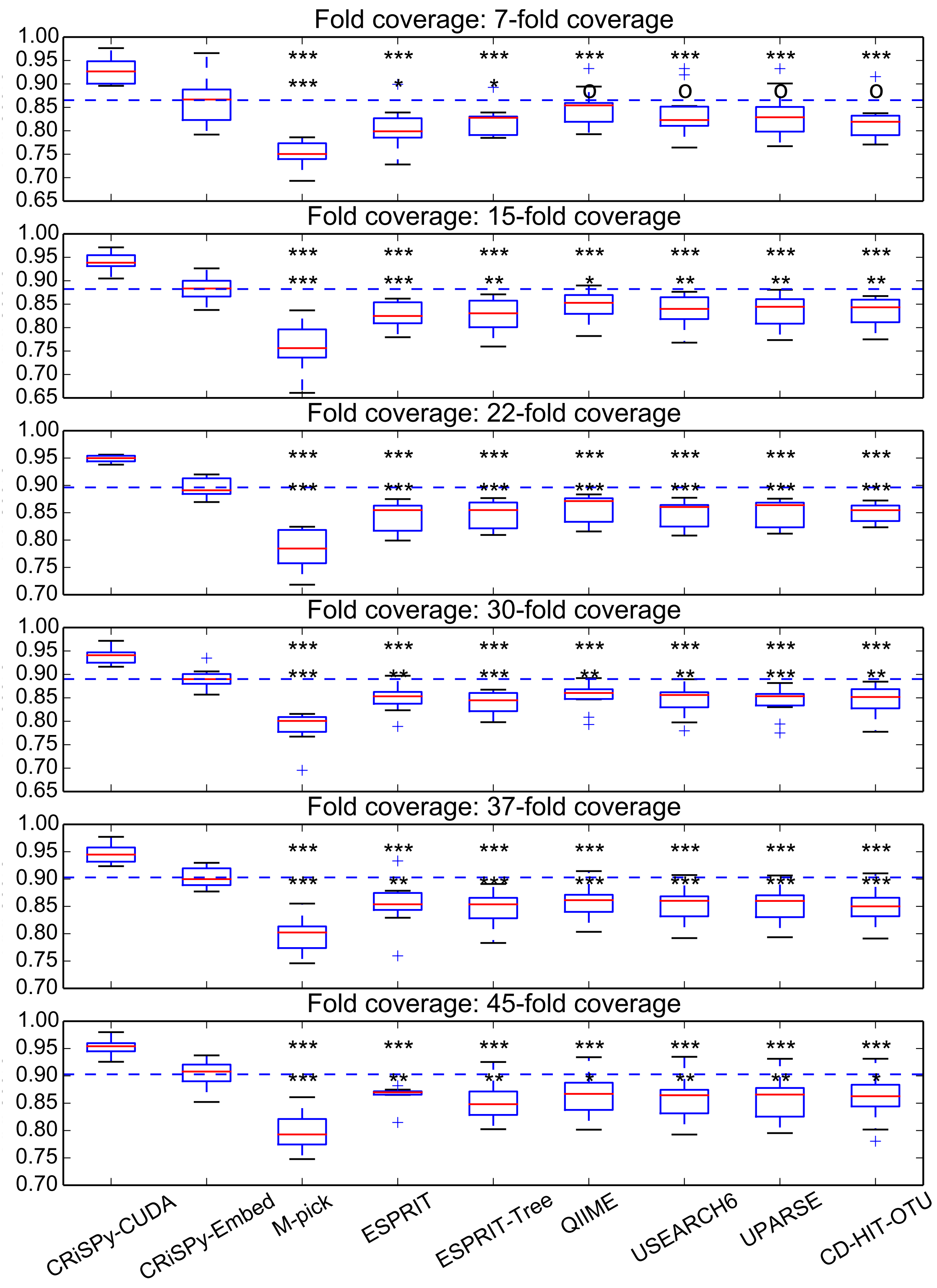


FIGURE 6.7: AMI scores achieved by the OTU binning tools of interest when clustering datasets with different fold coverage values. These AMI scores are computed using the same clustering results as the ARI scores in Figure 6.6.

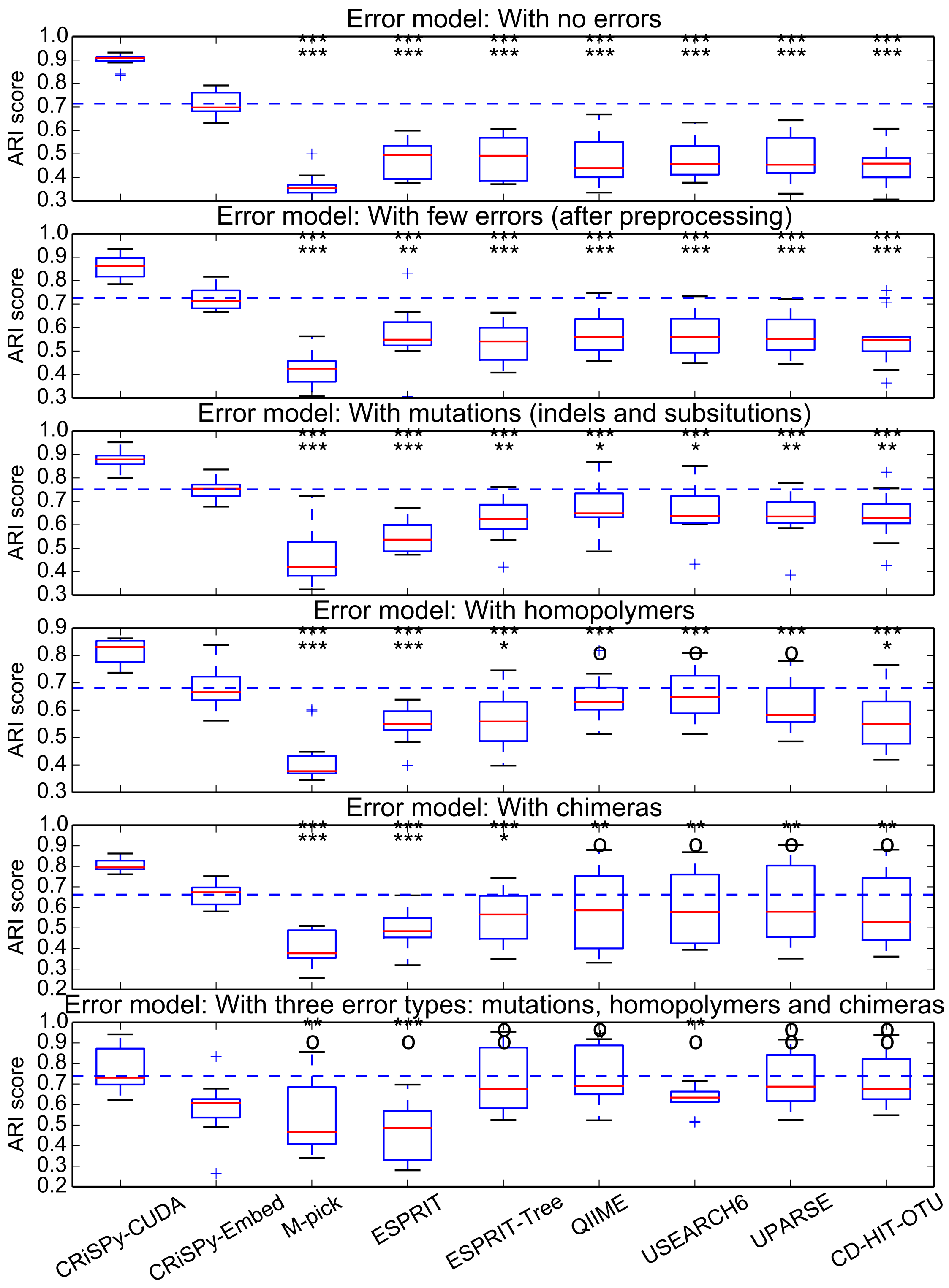


Figure 6.8: ARI scores achieved by the OTU binning tools of interest when clustering datasets with different types of sequencing and PCR errors (mutations, homopolymers, chimeras). Each dataset contains 5000 reads and is simulated from 100 full-length 16S rDNA reference sequences chosen randomly from the Greengenes database.

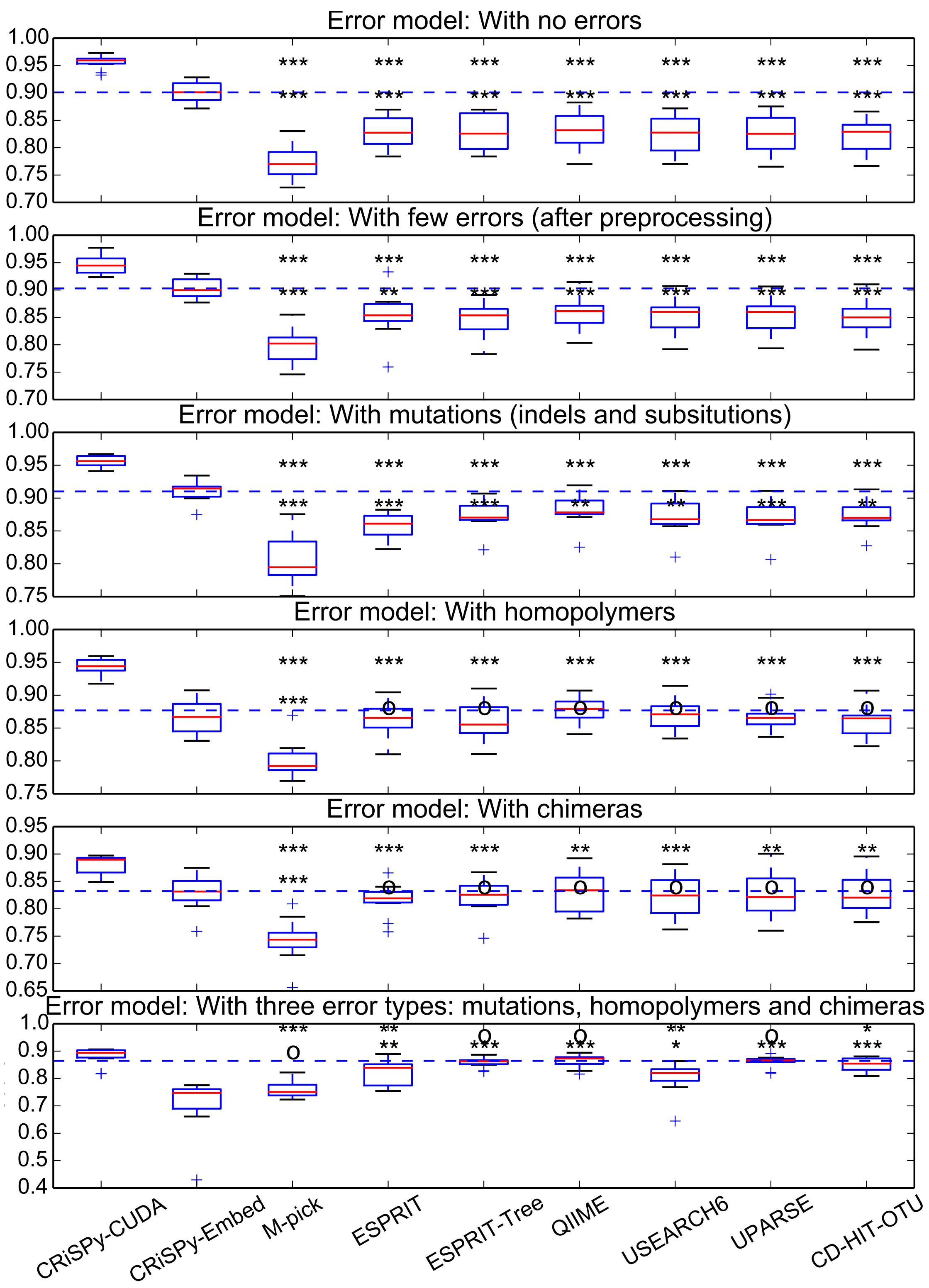


Figure 6.9: AMI scores achieved by the OTU binning tools of interest when clustering datasets with different types of sequencing and PCR errors. These AMI scores are computed using the same clustering results as the ARI scores in Figure 6.8.

CRiSPy-Embed are different (in most cases, higher) than those achieved by other tools. Besides, for each scenario the horizontal dashed line shows the maximum mean score achieved by tested existing tools.

CRiSPy-CUDA and CRiSPy-Embed provide better clustering accuracy mainly due to the dynamic dendrogram cutting method. The clustering accuracy of M-pick is low because the modularity-based approach employed by M-pick often does not work well for a large number of clusters. The memory usage of M-pick is also higher than other tools due to the storage required to build and store a graph of N vertices for modularity detection. Most GHC tools employ similar clustering strategies with slightly different seeding scheme, thus producing similar accuracy.

CRiSPy-CUDA generally runs faster than most AHC tools thanks to the parallel algorithms running on GPU. Particularly, CRiSPy-CUDA is 1.8x - 2.9x faster than ESPRIT-Tree, the currently most efficient AHC tool according to the benchmark by Sun et al. [47]. CRiSPy-CUDA is slower than some GHC tools due to the different natures of the two approaches: quadratic versus linear complexity.

To benchmark the robustness of these OTU clustering tools on different types of datasets, I vary two important aspects of a 16S datasets namely the rank abundance model and the fold coverage and the diversity. Figure 6.4 and 6.5 report the clustering quality for five common abundance models supported by Grinder using two different accuracy measures: the ARI and the AMI score. Similarly, Figure 6.6 and 6.7 report the clustering accuracy for five fold-coverage values often encountered in real-world datasets. These results further illustrate the robustness of the dynamic dendrogram cutting employed in CRiSPy-CUDA and CRiSPy-Embed.

Although using any OTU clustering tool directly on uncleaned datasets is not recommended, I also benchmark the clustering quality on datasets with sequencing errors to compare their robustness in the presence of errors. Figure 6.8 and 6.9 report the results from this experiment.

CRiSPy-CUDA is able to handle datasets with mutations and chimeras as well as cleaned datasets with sequencing errors. Out of three types of sequencing errors, homopolymers and chimeras have the greater impact on CRiSPy's clustering accuracy. This phenomenon can be explained by the fact that homopolymers induce a more significant effect on the pairwise distance values of the alignment-based genetic distance matrix. Meanwhile, chimeric sequences are often treated as new OTUs and cause an overestimation of the number of OTUs. Nonetheless, CRiSPy-CUDA still produces better clustering results than existing OTU clustering tools for datasets with or without errors.

6.2.2 On external mock datasets

To validate the performance of CRiSPy-CUDA and CRiSPy-Embed on non-simulated datasets, I use eight mock communities produced in vitro by Quince et al. [119]

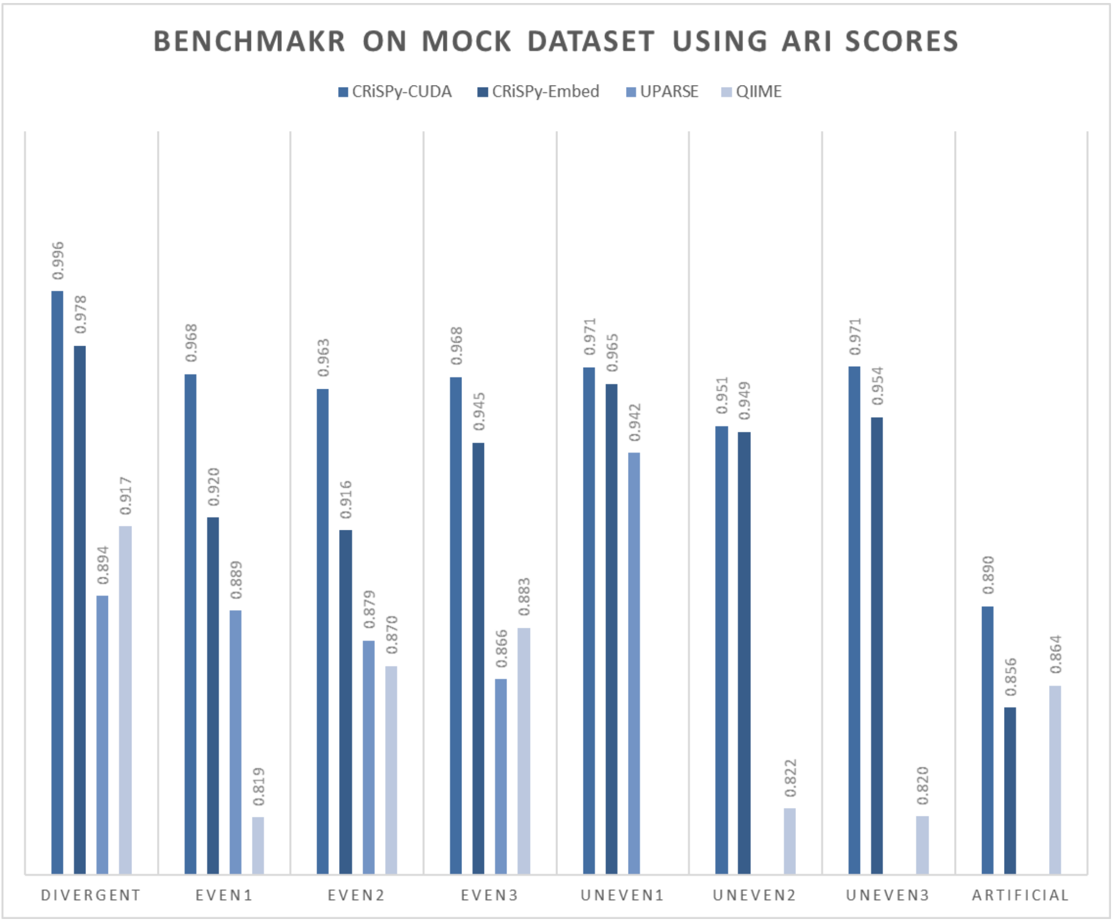


FIGURE 6.10: ARI scores achieved by the OTU clustering tools of interest when running on eight mock datasets in Table 6.1.

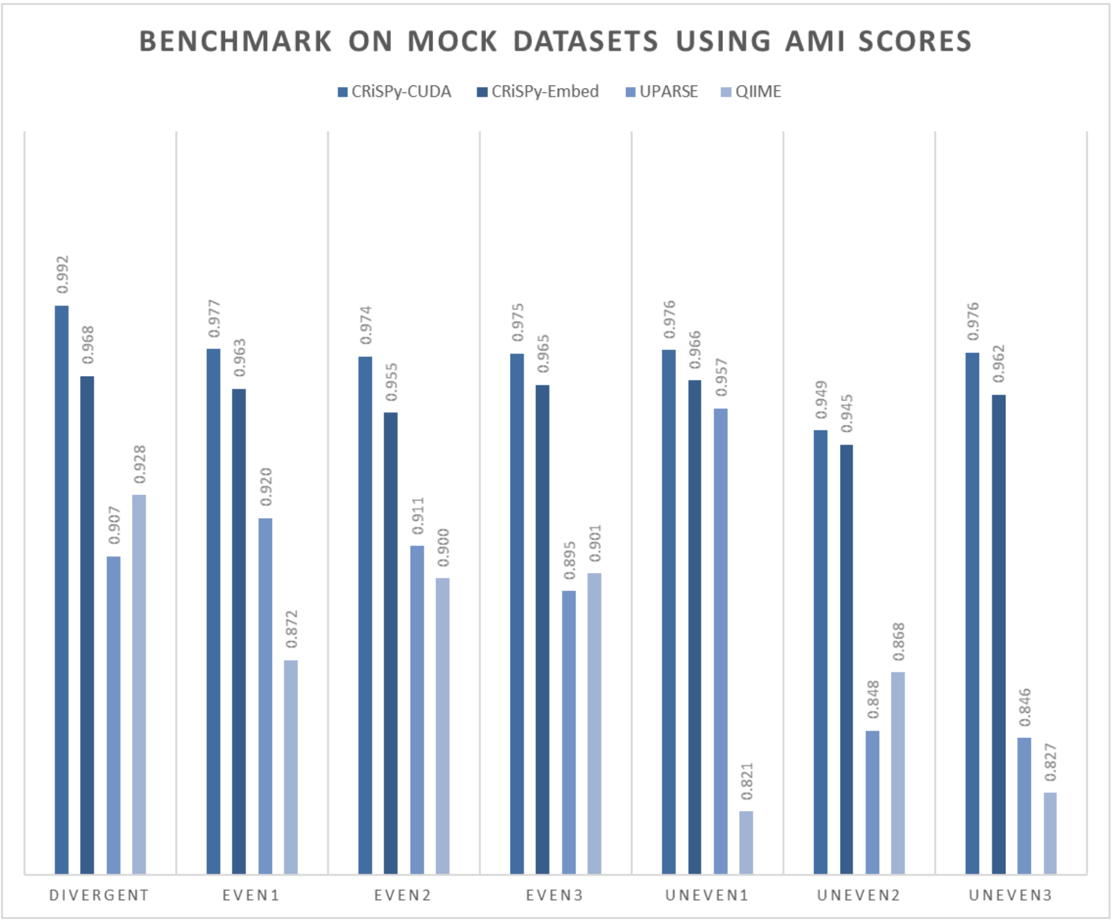


FIGURE 6.11: AMI scores achieved by the OTU clustering tools of interest when running on eight mock datasets.

and Turnbaugh et al. [6]. Figure 6.10 and 6.11 show the performance of CRiSPy-CUDA and CRiSPy-Embed against QIIME and UPARSE on these eight mock datasets. Overall, CRiSPy-CUDA and CRiSPy-Embed are more accurate than both UPARSE and QIIME.

6.3 Evaluation of computational efficiency

6.3.1 On large simulated datasets

In this experiment, I compare the computational performance CRiSPy-CUDA against other scalable binning tools. As shown in Figure 6.2, M-pick and ESPRIT

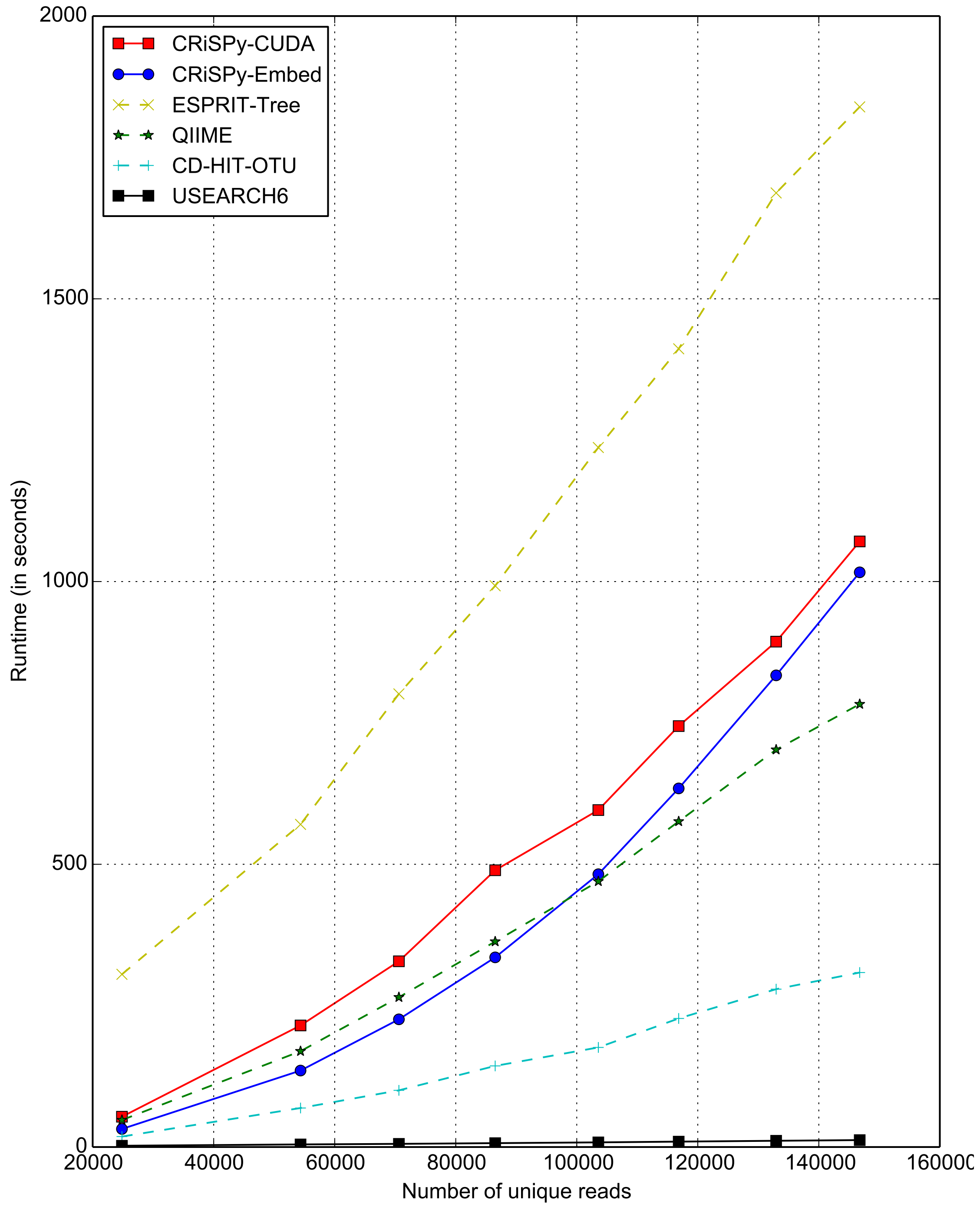


Figure 6.12: Execution time of the OTU clustering tools of interest running on large datasets of 25000 to 150000 cleaned reads. CRiSPy-CUDA and CRiSPy-Embed are faster than ESPRIT-Tree, the fastest amongst existing AHC tools.

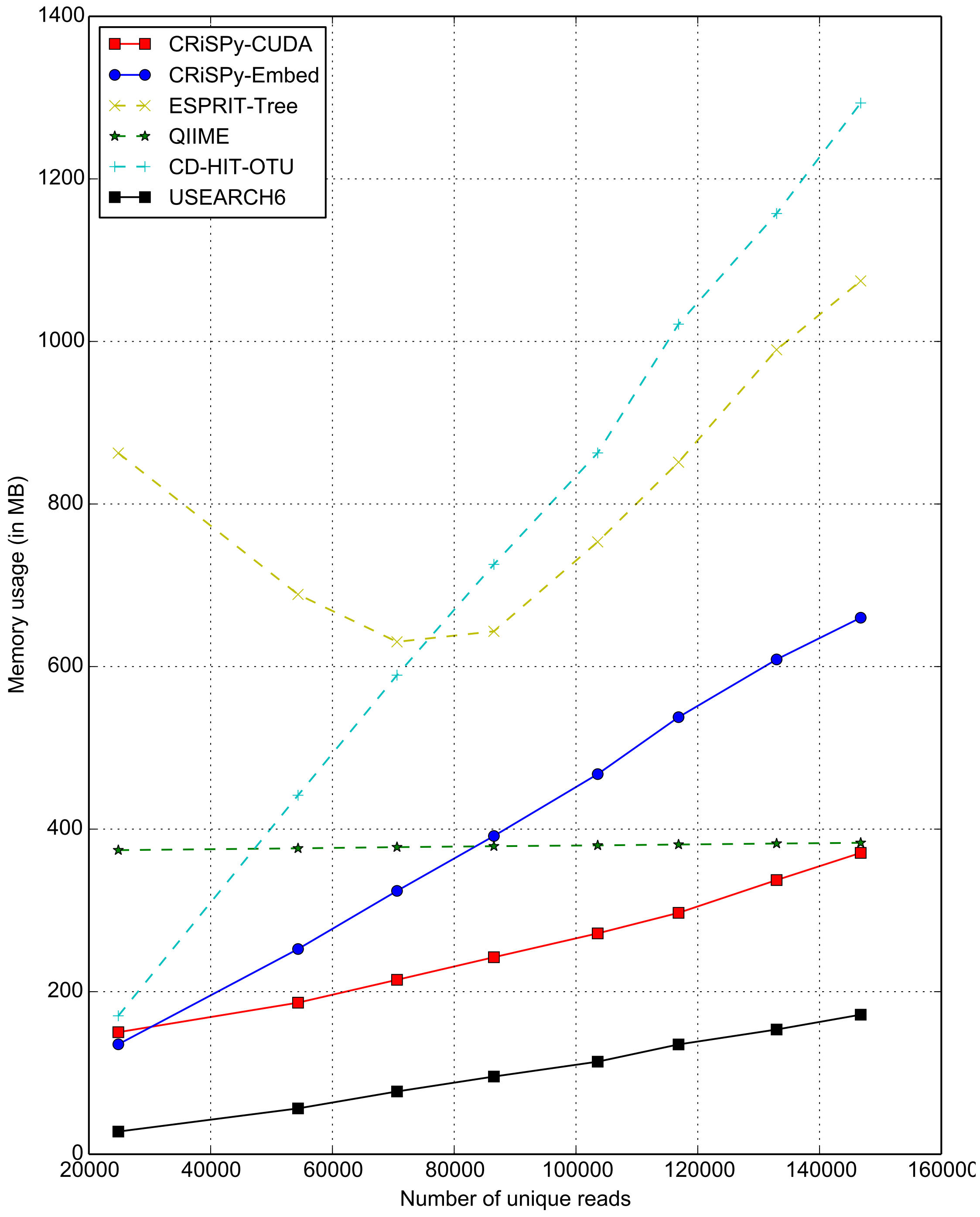


FIGURE 6.13: Memory usage by the OTU clustering tools of interest when running on large datasets with sizes ranging from 25000 to 150000 reads. CRiSPy-CUDA and CRiSPy-Embed achieve quasilinear memory complexity compared to the quadratic complexity of other AHC tools such as ESPRIT-Tree.

are less efficient than other tools. Therefore, they are not included in this experiment. I use eight large datasets with different sizes ranging from 25000 to 150000 unique cleaned reads and the same rank abundance model (power law) and fold coverage value (about 35x).

The memory usage of CRiSPy-CUDA scales quasilinear with respect to the dataset size as shown in Figure 6.13. Furthermore, Figure 6.12 shows that both CRiSPy-Embed and CRiSPy-CUDA are faster than ESPRIT-Tree, the currently most efficient AHC tool according to the benchmark by Sun et al. [47]. In particular, CRiSPy-CUDA is 1.8x - 2.9x faster than ESPRIT-Tree.

6.3.2 On public datasets

To evaluate the performance of CRiSPy, we have acquired eight 16S rRNA pyrosequencing datasets from the NCBI Sequence Read Archive (SRA) including three medium datasets and five large datasets. These raw datasets usually contain reads of low quality which can introduce a considerable number of false diversity into the species richness estimation. Hence, we have preprocessed these raw datasets to remove reads which contains ambiguous nucleotides (N) and reads with lengths that are not within 1 standard deviation from the average length. The numbers of reads before and after preprocessing are recorded in Table 6.3.

In this study to benchmark CRiSPy-CUDA on both CPU and GPU, we compare it mainly against ESPRIT. ESPRIT package includes two different versions: a personal computer version (ESPRIT PC) and a computer cluster version (ESPRIT CC). ESPRIT PC implements sequential codes running on a single-CPU. ESPRIT CC takes a data parallel approach to parallelize both k-mer and genetic distance computation. The input data is split into several parts and distributed to compute nodes in a cluster through job submission. The partial distance matrices are then combined to form the full sparse distance matrix which is clustered by the same clustering module as ESPRIT PC.

Table 6.3: Runtime (in seconds) and speedup comparison of k-mer distance computation between ESPRIT and CRiSPy

Dataset	Number of raw reads	Number of cleaned reads	ESPRIT PC	ESPRIT CC		CRiSPy MT		Single-GPU		Quad-GPU	
			T	T	S	T	S	T	S	T	S
SRR029122	40864	15179	322	98	3.29	78	4.13	6.7	48	1.7	189
SRR013437	57902	22498	838	257	3.26	203	4.13	15.8	53	4.0	211
SRR064911	23078	16855	837	251	3.33	203	4.12	19.1	44	4.8	173
SRR027085	249953	131482	27600	8648	3.19	6014	4.59	473	58	119	232
SRR033879_81	1494071	300667	153000	48316	3.17	29130	5.25	3150	49	791	193
SRR026596_97	333630	178860	91300	29386	3.11	22432	4.07	1746	52	440	208
SRR058099	339344	162223	78000	25550	3.05	17920	4.35	1459	53	368	212
SRR043579	857248	256760	195000	66115	2.95	48221	4.04	4046	48	1015	192

Table 6.4: Runtime (in minutes) and speedup comparison of genetic distance computation between ESPRIT and CRiSPy

Dataset	Average length	Percentage of pairs left p	ESPRIT PC	ESPRIT CC		CRiSPy MT		Single-GPU		Quad-GPU	
			T	T	S	T	S	T	S	T	S
SRR029122	239	11.04%	161	64	2.50	33	4.88	1.7	94	0.4	374
SRR013437	267	11.82%	462	184	2.51	94	4.93	4.6	101	1.1	402
SRR064911	524	56.73%	4700	1964	2.39	958	4.90	44	108	11	429
SRR027085	260	0.84%	1074	423	2.54	212	5.07	10	107	2.6	416
SRR033879_81	268	9.03%	64806	27029	2.40	12683	5.11	611	106	153	424
SRR026596_97	541	28.45%	2815	1128	2.50	559	5.04	26	110	6.5	434
SRR058099	531	6.76%	52430	22568	2.32	10551	4.97	479	109	120	437
SRR043579	556	8.37%	166160	73289	2.27	32627	5.09	1496	111	374	444

Since the performance of ESPRIT CC depends on cluster setup and is subjected to communication overheads, we mainly use ESPRIT PC to benchmark the performance of CRiSPy. However, we also report the runtime and speedup of ESPRIT CC on a cluster of 4 CPUs to give readers a rough idea about the performance of ESPRIT CC. We use the latest ESPRIT version released on February 2011. ESPRIT source code is available upon request to Dr. Sun Yijun `http://plaza.ufl.edu/sunyijun/ESPRIT.html`.

We have implemented three different versions of CRiSPy:

1. a multithreaded C++ program with OpenMP (CRiSPy MT)
2. a CUDA program running on a single-GPU (CRiSPy single-GPU)
3. a CUDA program running on a multi-GPU cluster (CRiSPy multi-GPU)

We have conducted performance comparisons under the Linux OS with the following setup. ESPRIT PC and CRiSPy MT runtime are estimated on a Dell T3500 Workstation with a quad-core Intel Xeon 2.93 GHz processor, 4GB RAM.

ESPRIT CC runtime is estimated on a cluster for four Dell T3500 Workstations connected via Ethernet. Condor High-Throughput Computing System is used for job submission. CRiSPy single-GPU runtime is measured on a Fermi-based Tesla S2050 GPU card with 3GB RAM. CRiSPy quad-GPU runtime is measured on a quad-GPU cluster which consists of two host nodes, each of which connected to two S2050 GPU cards. These nodes are connected by a high-speed Infiniband network. We use the following parameters for our experiments: $k = 6$, $\theta_k = 0.3$ for large datasets, $\theta_k = 0.5$ for medium datasets, $\theta_g = 0.2$, $sbt(x = y) = 5$, $sbt(x \neq y) = -4$, $\alpha = -10$ and $\beta = -5$.

Performance comparison of k-mer distance computation

Table 6.3 records the runtime (in seconds) of the pairwise k-mer distance computation. For a dataset which contains n reads, the total number of pairs to be computed is $n(n-1)/2$. For both k-mer and genetic distance modules, T stands for runtime and S stands for speedup. T includes IO transfer time between host CPU and CPU, computation time of the algorithm and file IO time to output results to file.

CRiSPy MT is a multithreaded program written in C++ with OpenMP for a multi-core PC. It exploits the data parallelism of the pairwise distance matrix computation and achieves an average speedup 4.34 compared to ESPRIT PC. ESPRIT CC requires more time that CRiSPy MT mainly due to scheduling, communication overheads and data splitting stage. CRiSPy on a single-GPU (quad-GPU) runs 50.7(201.3) times faster than ESPRIT PC and 11.7 (46.4) times faster than CRiSPy MT.

Performance comparison of genetic distance computation

Table 6.4 shows the runtime (in minutes) of the genetic distance computation on the aforementioned datasets. The percentage of pairs p reported in Table 6.4 is defined as the number of read pairs with k-mer distance less than the threshold $\theta_k = 0.3$ divided by the total number of pairs which is $n(n-1)/2$ for a dataset of n reads. Hence,$1 - p$ is the percentage of pairs that k-mer distance module has effectively filtered out.

When running on multiple GPUs, the number of pair indices acquired from k-mer filtering step are divided equally amongst all the GPUs. Each GPU will then process its set of pair indices independently. As the runtime of ESPRIT PC, ESPRIT CC and CRiSPy MT are tremendous for large datasets, we sample representative parts of each dataset to get an estimated runtime.

In comparison with ESPRIT PC, the runtime taken by CRiSPy MT reduces by the factor of 5.00 on average. Furthermore, average speedup gain by CRiSPy single-GPU (quad-GPU) is 105.6 (419.8) compared to ESPRIT PC and 21.1 (84.0) compared to CRiSPy MT. Similar to the filtration stage, ESPRIT CC encounters even more signification communication overheads since the amount of input and output data are much larger.

The speedup gain of the genetic distance module is much more significant compared to the k-mer distance module since genetic distance computation is more compute-intensive and hence it can utilize more processing powers of GPUs. Furthermore, the performance of this module increases in correspondence to the average length and the size of the input dataset.

Execution time of CRiSPy full run

CRiSPy's processing pipeline includes three modules: parallel k-mer distance computation, parallel genetic distance matrix computation and sequential clustering of the resulting sparse distance matrix. Table 6.5 and Table 6.6 show the runtime of CRiSPy on a single-GPU and a quad-GPU cluster respectively for five large datasets.

TABLE 6.5: Runtime (in minutes) of CRiSPy full run on a single-GPU

Dataset	k-mer distance	Genetic distance			Clustering		Total runtime		
	distance	full	1/5	1/10	Sorting	Hcluster	full	1/5	1/10
SRR027085	7.9	10	4.8	3.3	2.6	3.0	24	18	17
SRR033879_81	53	611	286	195	162	285	1111	785	694
SRR026596_97	29	26	10	6.2	1.7	1.8	58	43	39
SRR058099	24	479	195	116	36	49	587	303	224
SRR043579	67	1496	607	360	102	152	1849	960	714

TABLE 6.6: Runtime (in minutes) of CRiSPy full run on a quad-GPU cluster

Dataset	k-mer	Genetic distance			Clustering		Total runtime		
	distance	full	1/5	1/10	Sorting	Hcluster	full	1/5	1/10
SRR027085	2.0	2.6	1.2	0.8	2.6	3.0	10	8.8	8.4
SRR033879_81	13	153	72	49	162	285	613	532	509
SRR026596_97	7.3	6.5	2.6	1.6	1.7	1.8	17	13	12
SRR058099	6.1	120	49	29	36	49	210	139	119
SRR043579	17	374	152	90	102	152	678	455	393

TABLE 6.7: Runtime (in minutes) and speedup comparison between ESPRIT and CRiSPy

Dataset	ESPRIT PC	CRiSPy MT		Full band		1/5 band		1/10 band	
	T	T	S	T	S	T	S	T	S
SRR029122	167	35	4.79	2.4	68	1.6	106	1.3	125
SRR013437	477	98	4.85	6.1	78	3.7	130	3.0	157
SRR064911	4720	968	4.88	50	94	24	196	17	276

For the genetic distance computation, we use three different global alignment schemes including full alignment, 1/5 banded alignment and 1/10 banded alignment. Please note that 1/5 (1/10) banded alignment reduces the runtime by a factor of 2.29 (3.58) compared to full alignment. We have observed from these experiments that 1/5 banded alignment produces identical OTUs as the full alignment and the 1/10 banded alignment results in only some minor difference in terms of outliers.

Table 6.7 shows the comparison between ESPRIT PC, CRiSPy MT and CRiSPy single-GPU full run for three medium datasets. We notice that by using banded alignment, CRiSPy single-GPU often requires two orders of magnitude less time than ESPRIT PC to execute the whole pipeline. Besides, the larger the dataset, the more significant speedup CRiSPy achieves.

6.4 Summary

This chapter reports the benchmark results achieved by the CRiSPy-Embed and CRiSPy-CUDA pipelines in comparison with other OTU clustering tools. It is often difficult to access the results of the cluster analysis due to the lack of universally accepted validation datasets with predefined labels. Therefore, to evaluate the performance of these pipelines against existing tools, a comprehensive OTU evaluation framework using simulated and mock datasets has also been created. The evaluation process involves the creation of test datasets, the deliberate choice of sensible accuracy measurements as well as the interpretation and presentation of the evaluation results. The benchmark results show that CRiSPy-Embed and CRiSPy-CUDA outperforms other OTU binning tools in terms of both accuracy and computational performance.

Chapter 7

Conclusion and Future Work

The main objective of this thesis is the development of efficient, accurate and robust hierarchical clustering algorithms for recognizing stratified patterns in biological sequence datasets. This chapter concludes the thesis by recounting the work done in this project towards this goal. This chapter also provides some recommendations to extend this work in future research.

7.1 Conclusion

Agglomerative hierarchical clustering is a useful analysis technique especially for biological sequence datasets. However, it is computational expensive with theoretical runtime complexities ranging from $O(n^2 logn)$ to $O(n^3)$ for a dataset of n data points. The AHC analysis involves two main steps: the pairwise distance matrix computation and the grouping procedure.

This research introduces CRiSPy-CUDA and CRiSPy-Embed - two scalable tools for taxonomy-independent analysis of large-scale 16S rRNA pyrosequencing datasets running on low-cost hardware. Using a workstation with a single CUDA-enabled GPU, CRiSPy-CUDA and CRiSPy-Embed can efficiently perform biodiversity estimation of large metagenomic datasets. Based on algorithms which are designed for massively parallel CUDA-enabled GPU, CRiSPy-CUDA and CRiSPy-Embed achieve speedup of up to two orders of magnitude over the equivalent sequential pipeline. Because large-scale genomic datasets become more and more accessible to scientists, scalable yet accurate tools like CRiSPy-CUDA and CRiSPy-Embed are crucial for research in this area.

To enable the application of hierarchical clustering analysis on large datasets, I have implemented a memory-efficient algorithm for agglomerative hierarchical clustering called SparseHC. SparseHC can also perform space-efficient average-linkage, single-linkage and complete-linkage on sparse distance matrices. Empirical results show that SparseHC outperforms other sparse hierarchical clustering tools in terms of both speed and memory usage, making it suitable for processing large sparse distance matrices. Besides OTU clustering, SparseHC can also be used as a general hierarchical clustering tool for full and sparse distance matrices.

Although CRiSPy-CUDA is designed for microbial studies targeting DNA sequence analysis, the individual k-mer distance and genetic distance modules on multi-core CPU and GPU can easily be extended to support protein sequence analysis and be used in other biological sequence analyses. For examples, k-mer distance is used for fast, approximate phylogenetic tree construction [46] and pairwise genetic distance computation is used in metagenomic processing pipelines such as CROP [50].

Similarly, CRiSPy-Embed can be used to transform data points from a sequence space (e.g. DNA space or protein space) into the coordinate space. After converting sequences into vectors of real coordinates, we can apply multidimensional scaling to these vectors to scale them to two or three dimensional space, which then can be plotted on 2D or 3D figures. This capability of CRiSPy-Embed is particularly useful for visualizing biological sequences.

7.2 Recommendation for Future Work

7.2.1 Target Illumina datasets

CRiSPy-CUDA and CRiSPy-Embed currently target preprocessed pyrosequencing datasets of up to a million reads. However, the data size generated by new sequencing platforms are increasing rapidly. Recent sequencers such the Illumina HiSeq can now generate up to 600 billion paired-end reads of average length 150 base pairs.

Besides 454 pyrosequencing, Illumina sequencing is now becoming another common sequencing technology used to generate 16S rRNA datasets. Compared to 454 datasets, Illumina datasets are often larger but contain much shorter reads. The Illumina read length is in the range of 50-150 base pairs as compared to 400-700 base pairs as in 454 reads. I have run CRiSPy and CRiSPy-Embed on several simulated Illumina datasets.

The CRiSPy tools are unable to scale up to datasets sized billions of reads because:

- The clustering module is memory constrained. The data size that the CRiSPy tools can currently process is proportional to the RAM capacity
- Illumina read length is very short compared to pyrosequencing reads. CRiSPy's approach uses distances between the sequences to distinguish them and to cluster them. When the read length is short, the reads become very similar and hence the metric-based approaches does not work well.

Thus, there is a need for faster and more memory-efficient algorithms and tools to process datasets of larger size. Designing more accurate and fast algorithms to analyze even large-scale datasets is therefore an important future research topic.

A possible direction for future research is to design a parallel/distributed algorithm for agglomerative hierarchical clustering because it can potentially solve the memory constraint and scalability issues of the current approaches.

7.2.2 Dimensionality reduction

OTU clustering for next generation sequencing datasets can be considered as a special application of a more general problem: clustering of high dimensional datasets. Another special characteristic of the OTU clustering problem is that the data points lie in a non-Euclidean space.

Throughout this work the main approach taken to tackle the high dimension issue is based on filtration. The data is first filtered by the cleansing process by removing contaminated data points (e.g. reads with the end points unmatched with the primers used in the DNA sequencing processes, chimeric sequences coming from two different original full-length DNA sequences) and by removing the redundant data points (duplicates). Subsequently, the data is filtered for the second round after the matrix computation stage where only a part of the full pairwise distance matrix.

Another common approach often used in machine learning to tackle the high dimension issue is dimensionality reduction using techniques such as principal component analysis. Dimensionality reduction for large biological sequence datasets is another interesting future research direction.

7.3 Publications

1. ***Nguyen, T. D.***, Schmidt, B., Zheng, Z., & Kwoh, C. K.(2015). **Efficient and Accurate OTU Clustering with GPU-based Sequence Alignment and Dynamic Dendrogram Cutting**. In *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (no. 1, pp. 1). IEEE Computer Society. ISSN 1545-5963. doi: 10.1109/TCBB.2015.2407574.

2. ***Nguyen, T. D.***, Schmidt, B., & Kwoh, C. K. (2014). **Fast Dendrogram-based OTU Hierarchical Clustering using Sequence Embedding**. In *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics - BCB '14* (pp. 63-72). ACM Press. ISBN 9781450328944. doi: 10.1145/2649387.2649402.

3. ***Nguyen, T. D.***, Schmidt, B., & Kwoh, C. K. (2014). **SparseHC: a memory efficient online hierarchical clustering algorithm**. In *Procedia Computer Science* (vol. 29, pp. 8-19). Elsevier. ISSN 18770509. doi: 10.1016/j.procs.2014.05.001.

4. ***Nguyen, T. D.***, Schmidt, B., Zheng, Z., & Kwoh, C. K. (2013). **Large-Scale Clustering of Short Reads for Metagenomics On GPUs**. In *Biological Knowledge Discovery Handbook: Preprocessing, Mining, and Postprocessing of Biological Data* (chap. 44, pp. 1003-1022). Wiley. ISBN 9781118617151. doi: 10.1002/9781118617151.

5. Zheng, Z., ***Nguyen, T. D.***, & Schmidt, B. (2011). **CRiSPy-CUDA: Computing species richness in16S rRNA pyrosequencing datasets with CUDA**. In *Pattern Recognition in Bioinformatics* (vol. 7036, pp. 37-49). Springer. ISBN 978-3-642-24854-2.

Appendix A

Biodiversity Assessment

Let $R = \{R_1, \ldots, R_{N_R}\}$ denote the input read dataset, which consists of N_R reads (or sequences) over the DNA alphabet $\Sigma = \{A, C, G, T\}$. Let $O = \{O_1, \ldots, O_{N_O}\}$ denote the output OTUs (or clusters) produced by the OTU binning tools. The output set is composed of N_O clusters, each of which consists of one or more reads. Species richness and abundance are computed as follows:

A.1 Measuring species richness

Species richness is defined as the number of species in a community. When estimating the species richness of a community, only the presence or absence of taxa is considered. Species richness can be quantified by one of the following measures:

Rarefaction curve

A rarefaction curve is "a plot of the number of species as a function of the number of samples" [123]. It is commonly used in sampling processes to identify the sampling size. The rarefaction curve, f_n is defined as:

$$f_n = E[X_n] = N_O - \binom{N_R}{n}^{-1} \sum_{i=1}^{N_O} \binom{N_R - N_i}{n} \tag{A.1}$$

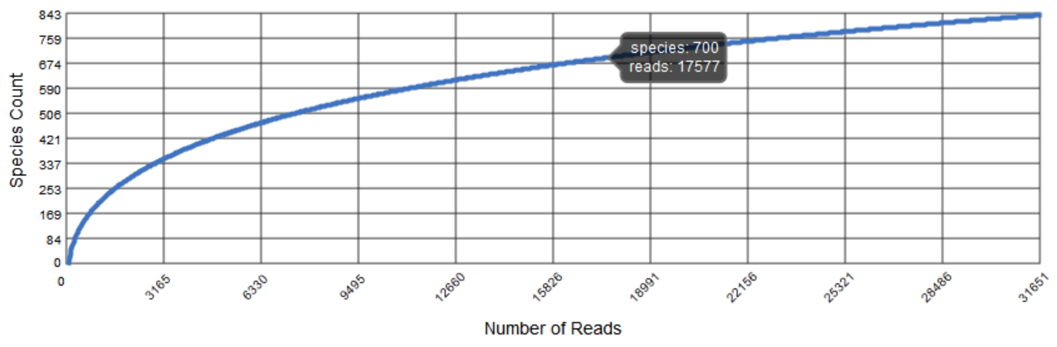


FIGURE A.1: A rarefaction curve generated by MG-RAST [4]. The species count (richness) increases with the number of sequence sampled. The optimal sample size for discovery applications is the size where the species richness no longer increases.

- N_R denotes the number of input sequences
- N_O denotes the number of output taxonomic units
- X_n denotes the number of output taxonomic units still present in the sub-sample of n sequences

Figure A.1 shows a sample rarefaction curve.

chao

Given the assumption that rare species also carry information about missing species, *chao*1 and *chao*2 measures use identified singletons (clusters that have one members each) and doubletons (clusters that have two members each) to estimate the amount of missing species [124].

$$S_{chao1} = S_{obs} + \frac{f_1(f_1-1)}{2(f_2+1)} \qquad S_{chao2} = S_{obs} + \frac{f_1^2}{2f_2} \tag{A.2}$$

- S_{chao2} is the estimate of species richness using species abundance information
- S_{obs} denotes the observed number of species
- f_1 denotes the number of output singletons (1-member clusters)

- f_2 denotes the number of output doubletons (2-member clusters)

Abundance-based Coverage Estimator (ACE)

Abundance-based coverage estimator or ACE is another commonly-used metric for species richness. The ACE measure is defined as follows:

$$S_{ACE} = S_{abundance} + \frac{S_{rare} + f_1\hat{\gamma}^2}{\hat{C}} \tag{A.3}$$

- $S_{abundance}$ measures the number of abundant species
- S_{rare} measures the number of rare species
- f_1 measures the number of output singletons (1-member clusters)
- $\hat{C}$ denotes the estimated sample coverage $\hat{C} = 1 - f_1/\sum_{i=1}^{K} if_i$
- $\hat{\gamma}^2$ denotes the estimated squared coefficient of variation
  $\hat{\gamma}^2 = \max(0, S_{rare}\sum_{i=1}^{K} i(i-1)f_i/[\hat{C}(\sum_{i=1}^{K} if_i)^2] - 1)$

A.2 Measuring species abundance

Species abundance takes additional accounts for the number of occurrences of a taxon. Species abundance can be quantified by the Shannon index or the Simpson index.

Shannon index

The Shannon index or the Shannon entropy is a popular measure of biodiversity. It is defined as:

$$H' = -\sum_{i=1}^{N_O} i_i \log i_i \tag{A.4}$$

i_i is the proportion of individuals belonging to the i-th OTU (O_i) and N_O is the richness or total number of species in the community of interest.

The Shannon index quantifies the uncertainty in predicting the species identity of an individual that is taken at random from the dataset.

Simpson index

The Simpson index expresses the probability that two entities taken at random from the dataset of interest represent the same type.

$$\lambda = \sum_{i=1}^{N_O} i_i^2 \tag{A.5}$$

A.3 Measuring relative abundance

Relative abundance or species evenness defines how equally abundant are each of the species. Relative abundance is defined as:

$$J' = \frac{H'}{H'_{max}}, H'_{max} = lnN_O \tag{A.6}$$

where N_O is the total number of species, determined by species richness.

The rank abundance curve or Whittaker plot can be used to display the relative species abundance. Abundance rank (x-axis) versus the relative abundance (y-axis). It can also be used to visualize species richness and species evenness. Figure A.2 shows an example of the rank abundance curve.

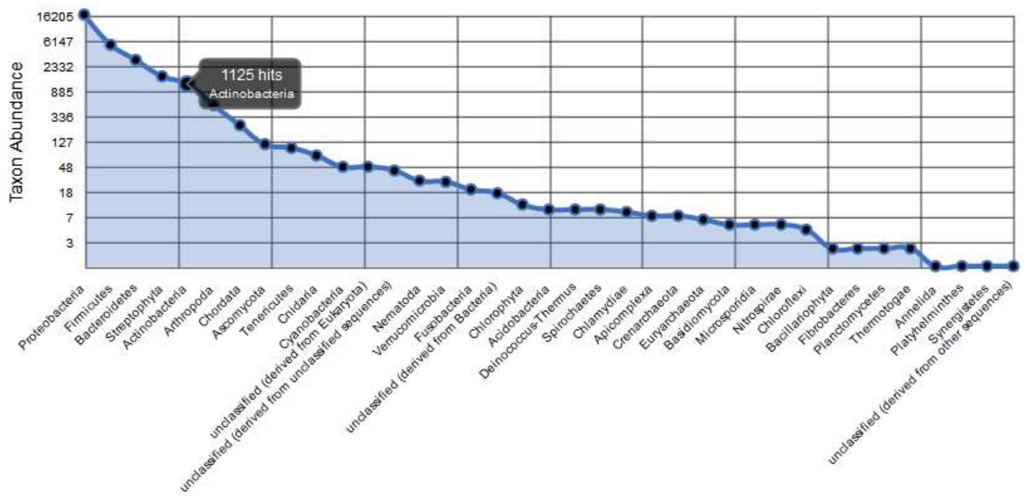


FIGURE A.2: A rank abundance curve generated by MG-RAST [4]. X-axis shows the species sorted by abundance rank (from left to right the most abundant to the least abundant species). Y-axis plots the abundance of each species on a log scale.

Bibliography

[1] J. G. Caporaso, C. L. Lauber, W. A. Walters, et al. "Ultra-high-throughput microbial community analysis on the Illumina HiSeq and MiSeq platforms". *The ISME Journal*, volume 6, no. 8, pp. 1621–1624, 2012.

[2] P. Glaskowsky. "NVIDIA Fermi: The First Complete GPU Computing Architecture". Technical report, 2009.

[3] J. Nickolls and W. J. Dally. "The GPU Computing Era". *IEEE Micro*, volume 30, no. 2, pp. 56–69, 2010.

[4] F. Meyer, D. Paarmann, M. D'Souza, et al. "The metagenomics RAST server - a public resource for the automatic phylogenetic and functional analysis of metagenomes". *BMC Bioinformatics*, volume 9, no. 1, p. 386, 2008.

[5] T. C. Glenn. "Field guide to next-generation DNA sequencers". *Molecular Ecology Resources*, volume 11, no. 5, pp. 759–69, 2011.

[6] P. J. Turnbaugh, C. Quince, J. J. Faith, et al. "Organismal, genetic, and transcriptional variation in the deeply sequenced gut microbiomes of identical twins". *Proceedings of the National Academy of Sciences*, volume 107, no. 16, pp. 7503–7508, 2010.

[7] C. De Filippo, D. Cavalieri, M. Di Paola, et al. "Impact of diet in shaping gut microbiota revealed by a comparative study in children from Europe and rural Africa". *Proceedings of the National Academy of Sciences*, volume 107, no. 33, pp. 14691–6, 2010.

[8] N. Fierer, M. Hamady, C. L. Lauber, et al. “The influence of sex, handedness, and washing on the diversity of hand surface bacteria”. *Proceedings of the National Academy of Sciences*, volume 105, no. 46, pp. 17994–17999, 2008.

[9] S. E. Dowd, T. R. Callaway, R. D. Wolcott, et al. “Evaluation of the bacterial diversity in the feces of cattle using 16S rDNA bacterial tag-encoded FLX amplicon pyrosequencing (bTEFAP)”. *BMC Microbiology*, volume 8, no. 1, p. 125, 2008.

[10] E. R. Mardis. “Next-generation DNA sequencing methods.” *Annual Review of Genomics and Human Genetics*, volume 9, pp. 387–402, 2008.

[11] J. Shendure and H. Ji. “Next-generation DNA sequencing”. *Nature Biotechnology*, volume 26, no. 10, pp. 1135–45, 2008.

[12] C. A. Hutchison. “DNA sequencing: bench to bedside and beyond.” *Nucleic Acids Research*, volume 35, no. 18, pp. 6227–37, 2007.

[13] A. Sboner, X. J. Mu, D. Greenbaum, et al. “The real cost of sequencing: higher than you think!” *Genome Biology*, volume 12, no. 8, p. 125, 2011.

[14] J. F. Siqueira, A. F. Fouad, and I. N. Rôças. “Pyrosequencing as a tool for better understanding of human microbiomes.” *Journal of Oral Microbiology*, volume 4, 2012.

[15] J. Handelsman, J. Tiedje, L. Alvarez-Cohen, et al. *Committee on Metagenomics: Challenges and Functional Applications*. National Academy of Sciences, 2007.

[16] R. I. Amann, W. Ludwig, and K. H. Schleifer. “Phylogenetic identification and in situ detection of individual microbial cells without cultivation.” *Microbiological Reviews*, volume 59, no. 1, pp. 143–69, 1995.

[17] M. S. Rappé and S. J. Giovannoni. “The uncultured microbial majority”. *Annual Review of Microbiology*, volume 57, pp. 369–94, 2003.

[18] J. C. Wooley, A. Godzik, and I. Friedberg. “A primer on metagenomics”. *PLoS Computational Biology*, volume 6, no. 2, p. e1000667, 2010.

[19] P. Hugenholtz and G. W. Tyson. “Microbiology: metagenomics”. *Nature*, volume 455, no. 7212, pp. 481–3, 2008.

[20] T. A. Isenbarger, C. E. Carr, S. S. Johnson, et al. “The most conserved genome segments for life detection on Earth and other planets”. *Origins of life and evolution of the biosphere : the journal of the International Society for the Study of the Origin of Life*, volume 38, no. 6, pp. 517–33, 2008.

[21] J. F. Petrosino, S. Highlander, R. A. Luna, et al. “Metagenomic pyrosequencing and microbial identification”. *Clinical Chemistry*, volume 55, no. 5, pp. 856–66, 2009.

[22] B. Alberts, A. Johnson, J. Lewis, et al. “From RNA to Protein”, 2002.

[23] J. E. Clarridge. “Impact of 16S rRNA gene sequence analysis for identification of bacteria on clinical microbiology and infectious diseases”. *Clinical Microbiology Reviews*, volume 17, no. 4, pp. 840—-62, table of contents, 2004.

[24] R. H. Whittaker. “Evolution and measurement of species diversity”. *Taxon*, pp. 213–251, 1972.

[25] S. P. Hubbell. *The Unified Neutral Theory of Biodiversity and Biogeography.* Princeton University Press, 2001.

[26] H. E. Driver and A. L. Kroeber. *Quantitative expression of cultural relationships.* University of California Press, 1932.

[27] R. C. Tryon. *Cluster analysis: correlation profile and orthometric (factor) analysis for the isolation of unities in mind and personality.* Edwards Brother, Inc., 1939.

[28] R. B. Cattell. “The description of personality: basic traits resolved into clusters.” *Journal of Abnormal and Social Psychology*, volume 38, no. 4, p. 476, 1943.

[29] M. A. DePristo, D. M. Weinreich, and D. L. Hartl. "Missense meanderings in sequence space: a biophysical view of protein evolution." *Nature Reviews Genetics*, volume 6, no. 9, pp. 678–87, 2005.

[30] E. Bornberg-Bauer and H. S. Chan. "Modeling evolutionary landscapes: mutational stability, topology, and superfunnels in sequence space." *Proceedings of the National Academy of Sciences of the United States of America*, volume 96, no. 19, pp. 10689–94, 1999.

[31] M. N. Ashby, J. Rine, E. F. Mongodin, et al. "Serial analysis of rRNA genes and the unexpected dominance of rare members of microbial communities". *Applied and Environmental Microbiology*, volume 73, no. 14, pp. 4532–42, 2007.

[32] P. J. Turnbaugh, M. Hamady, T. Yatsunenko, et al. "A core gut microbiome in obese and lean twins". *Nature*, volume 457, no. 7228, pp. 480–484, 2009.

[33] P. J. Turnbaugh, V. K. Ridaura, J. J. Faith, et al. "The effect of diet on the human gut microbiome: a metagenomic analysis in humanized gnotobiotic mice". *Science Translational Medicine*, volume 1, no. 6, p. 6ra14, 2009.

[34] M. L. Sogin, H. G. Morrison, J. A. Huber, et al. "Microbial diversity in the deep sea and the underexplored 'rare biosphere'". *Proceedings of the National Academy of Sciences*, volume 103, no. 32, pp. 12115–12120, 2006.

[35] S. M. Huse, J. A. Huber, H. G. Morrison, et al. "Accuracy and quality of massively parallel DNA pyrosequencing." *Genome Biology*, volume 8, no. 7, p. R143, 2007.

[36] M. Hamady and R. Knight. "Microbial community profiling for human microbiome projects: Tools, techniques, and challenges". *Genome Research*, volume 19, no. 7, pp. 1141–1152, 2009.

[37] J. G. Caporaso, J. Kuczynski, J. Stombaugh, et al. "QIIME allows analysis of high-throughput community sequencing data". *Nature Methods*, volume 7, no. 5, pp. 335–336, 2010.

[38] S. M. Huse, D. M. Welch, H. G. Morrison, et al. "Ironing out the wrinkles in the rare biosphere through improved OTU clustering". *Environmental Microbiology*, volume 12, no. 7, pp. 1889–1898, 2010.

[39] R. C. Edgar. "Search and clustering orders of magnitude faster than BLAST". *Bioinformatics*, volume 26, no. 19, pp. 2460–2461, 2010.

[40] R. C. Edgar. "UPARSE: highly accurate OTU sequences from microbial amplicon reads". *Nature Methods*, volume 10, no. 10, pp. 996–8, 2013.

[41] W. Li and A. Godzik. "Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences". *Bioinformatics*, volume 22, no. 13, pp. 1658–1659, 2006.

[42] Z. Zheng, T.-D. Nguyen, and B. Schmidt. "CRiSPy-CUDA: Computing species richness in 16S rRNA pyrosequencing datasets with CUDA". *Pattern Recognition in Bioinformatics*, pp. 37–49. Springer, 2011.

[43] P. D. Schloss, S. L. Westcott, T. Ryabin, et al. "Introducing mothur: Open-Source Platform-Independent Community Supported Software for Describing and Comparing Microbial Communities". *Applied and Environmental Microbiology*, volume 75, no. 23, pp. 7537–7541, 2009.

[44] Y. Sun, Y. Cai, L. Liu, et al. "ESPRIT: estimating species richness using large collections of 16S rRNA pyrosequences". *Nucleic Acids Research*, volume 37, no. 10, p. e76, 2009.

[45] Y. Cai and Y. Sun. "ESPRIT-Tree: hierarchical clustering analysis of millions of 16S rRNA pyrosequences in quasilinear computational time". *Nucleic Acids Research*, volume 39, no. 14, p. e95, 2011.

[46] R. C. Edgar. "MUSCLE: multiple sequence alignment with high accuracy and high throughput". *Nucleic Acids Research*, volume 32, no. 5, pp. 1792–1797, 2004.

[47] Y. Sun, Y. Cai, S. M. Huse, et al. "A large-scale benchmark study of existing algorithms for taxonomy-independent microbial community analysis." *Briefings in Bioinformatics*, volume 13, no. 1, pp. 107–121, 2011.

[48] T. Zhang, R. Ramakrishnan, and M. Livny. "BIRCH: A New Data Clustering Algorithm and Its Applications". *Data Mining and Knowledge Discovery*, volume 1, no. 2, pp. 141–182, 1997.

[49] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[50] X. Hao, R. Jiang, and T. Chen. "Clustering 16S rRNA for OTU prediction: a method of unsupervised Bayesian clustering." *Bioinformatics*, volume 27, no. 5, pp. 611–618, 2011.

[51] X. Wang, J. Yao, Y. Sun, et al. "M-pick, a modularity-based method for OTU picking of 16S rRNA sequences". *BMC Bioinformatics*, volume 14, p. 43, 2013.

[52] Y. Liu, D. L. Maskell, and B. Schmidt. "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units." *BMC research notes*, volume 2, no. 1, p. 73, 2009.

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

[54] Y. Liu, A. Wirawan, and B. Schmidt. "CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions." *BMC bioinformatics*, volume 14, no. 1, p. 117, 2013.

[55] B. Schmidt and D. Maskell. "MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA". *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 121–128. IEEE, 2009.

[56] W. Liu, B. Schmidt, and W. Müller-Wittig. "CUDA-BLASTP: accelerating BLASTP on CUDA-enabled graphics hardware." *IEEE/ACM transactions on computational biology and bioinformatics / IEEE, ACM*, volume 8, no. 6, pp. 1678–84, 2011.

[57] P. D. Vouzis and N. V. Sahinidis. "GPU-BLAST: using graphics processors to accelerate protein sequence alignment." *Bioinformatics*, volume 27, no. 2, pp. 182–8, 2011.

[58] Y. Liu, B. Schmidt, and D. L. Maskell. "CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform." *Bioinformatics*, volume 28, no. 14, pp. 1830–7, 2012.

[59] B. Schmidt. "CUSHAW2-GPU: Empowering Faster Gapped Short-Read Alignment Using GPU Computing". *IEEE Design & Test*, volume 31, no. 1, pp. 31–39, 2014.

[60] Y. Liu, B. Popp, and B. Schmidt. "CUSHAW3: sensitive and accurate base-space and color-space short-read alignment with hybrid seeding." *PloS one*, volume 9, no. 1, p. e86869, 2014.

[61] Y. Liu, B. Schmidt, and D. L. Maskell. "DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI." *BMC bioinformatics*, volume 12, no. 1, p. 85, 2011.

[62] Y. Liu, J. Schröder, and B. Schmidt. "Musket: a multistage k-mer spectrum-based error corrector for Illumina sequence data." *Bioinformatics*, volume 29, no. 3, pp. 308–15, 2013.

[63] H. Shi, B. Schmidt, W. Liu, et al. "A parallel algorithm for error correction in high-throughput short-read data on CUDA-enabled graphics hardware". *Journal of Computational Biology*, volume 17, no. 4, pp. 603–615, 2010.

[64] W. Liu, B. Schmidt, G. Voss, et al. "Molecular dynamics simulations on commodity GPUs with CUDA". pp. 185–196, 2007.

[65] B. Sukhwani and M. C. Herbordt. “GPU acceleration of a production molecular docking code”. *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-2*, pp. 19–27. ACM Press, New York, New York, USA, 2009.

[66] M. Ohue, T. Shimoda, S. Suzuki, et al. “MEGADOCK 4.0: an ultra-high-performance protein-protein docking software for heterogeneous supercomputers.” *Bioinformatics*, volume 30, no. 22, pp. 3281–3, 2014.

[67] J. Zhong and B. He. “Parallel graph processing on graphics processors made easy”. *Proceedings of the VLDB Endowment*, volume 6, no. 12, pp. 1270–1273, 2013.

[68] J. Zhong and B. He. “Medusa: Simplified Graph Processing on GPUs”. *IEEE Transactions on Parallel and Distributed Systems*, volume 25, no. 6, pp. 1543–1552, 2014.

[69] R. Raina, A. Madhavan, and A. Y. Ng. “Large-scale deep unsupervised learning using graphics processors”. *Proceedings of the 26th Annual International Conference on Machine Learning - ICML ’09*, pp. 1–8. ACM Press, New York, New York, USA, 2009.

[70] A. Coates, B. Huval, T. Wang, et al. “Deep learning with COTS HPC systems”. *Proceedings of the 30th international conference on machine learning*, pp. 1337–1345. 2013.

[71] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. F. Pereira, C. J. C. Burges, L. Bottou, et al., editors, *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc., 2012.

[72] A. Giusti, D. C. Cireşan, J. Masci, et al. “Fast Image Scanning with Deep Max-Pooling Convolutional Neural Networks”. p. 11, 2013.

[73] T.-D. Nguyen, B. Schmidt, Z. Zheng, et al. "Large-Scale Clustering of Short Reads for Metagenomics On GPUs". M. Elloumi and A. Y. Zomaya, editors, *Biological Knowledge Discovery Handbook: Preprocessing, Mining, and Postprocessing of Biological Data*, chapter 44, pp. 1003–1022. John Wiley & Sons, Inc., Hoboken, New Jersey, 2013.

[74] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[75] D. Kranzlmüller, P. Kacsuk, and J. Dongarra, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[76] J. Nickolls, I. Buck, M. Garland, et al. "Scalable parallel programming with CUDA". *Queue*, volume 6, no. 2, pp. 40–53, 2008.

[77] E. Lindholm, J. Nickolls, S. Oberman, et al. "NVIDIA Tesla: A unified graphics and computing architecture". *Micro, IEEE*, volume 28, no. 2, pp. 39–55, 2008.

[78] S. B. Needleman and C. D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". *Journal of Molecular Biology*, volume 48, no. 3, pp. 443–453, 1970.

[79] T.-D. Nguyen, B. Schmidt, and C.-K. Kwoh. "SparseHC: A Memory-efficient Online Hierarchical Clustering Algorithm". *Procedia Computer Science*, volume 29, pp. 8–19, 2014.

[80] P. Berkhin. "A survey of clustering data mining techniques". J. Kogan, C. Nicholas, and M. Teboulle, editors, *Grouping Multidimensional Data*, volume 10, chapter 2, pp. 25–71. Springer, 2006.

[81] A. K. Jain, M. N. Murty, and P. J. Flynn. "Data clustering: a review". *ACM Computing Surveys*, volume 31, no. 3, pp. 264–323, 1999.

[82] R. Xu and D. Wunsch. "Survey of clustering algorithms". *IEEE Transactions on Neural Networks*, volume 16, no. 3, pp. 645–678, 2005.

[83] M. Steinbach, G. Karypis, and V. Kumar. "A Comparison of Document Clustering Techniques". *KDD Workshop on Text Mining*, volume 400, no. X, pp. 1–2, 2000.

[84] M. Girvan and M. E. J. Newman. "Community structure in social and biological networks". *Proceedings of the National Academy of Sciences*, volume 99, no. 12, pp. 7821–6, 2002.

[85] W. H. E. Day and H. Edelsbrunner. "Efficient algorithms for agglomerative hierarchical clustering methods". *Journal of Classification*, volume 1, no. 1, pp. 7–24, 1984.

[86] P. H. A. Sneath and R. R. Sokal. *Numerical Taxonomy: The Principles and Practice of Numerical Classification.* A Series of books in biology. W.H. Freeman, 1973.

[87] B. S. Everitt, S. Landau, M. Leese, et al. *Cluster Analysis.* Wiley Series in Probability and Statistics. Wiley, 5th edition, 2011.

[88] G. N. Lance and W. T. Williams. "A General Theory of Classificatory Sorting Strategies 1. Hierarchical systems". *The Computer Journal*, volume 9, no. 4, pp. 373–380, 1967.

[89] M. R. Anderberg. *Cluster analysis for applications.* Academic Press, 1973.

[90] D. Müllner. "fastcluster: Fast Hierarchical, Agglomerative Clustering Routines for R and Python". *Journal of Statistical Software*, volume 53, no. 9, pp. 1–18, 2011.

[91] F. Murtagh and P. Contreras. "Methods of Hierarchical Clustering". *arXiv preprint arXiv:1105.0121*, 2011.

[92] I. Gronau and S. Moran. "Optimal implementations of UPGMA and other common clustering algorithms". *Information Processing Letters*, volume 104, no. 6, pp. 205–210, 2007.

[93] S. Guha, R. Rastogi, and K. Shim. "CURE: an efficient clustering algorithm for large databases". *ACM SIGMOD Record*, volume 26, no. 1, pp. 73–84, 1998.

[94] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[95] Y. Loewenstein, E. Portugaly, M. Fromer, et al. "Efficient algorithms for accurate hierarchical clustering of huge datasets: tackling the entire protein space". *Bioinformatics*, volume 24, no. 13, pp. i41–9, 2008.

[96] M. J. Bonder, S. Abeln, E. Zaura, et al. "Comparing clustering and pre-processing in taxonomy analysis". *Bioinformatics*, volume 28, no. 22, pp. 2891–2897, 2012.

[97] D. Knuth. "External Sorting". D. Knuth, editor, *The Art Of Computer Programming, Volume 3: Sorting and Searching*, chapter 5, pp. 248–379. Addison-Wesley, 2nd edition, 1998.

[98] R. Sedgewick. *Algorithms in C++: Graph Algorithms*. Addison-Wesley, 2002.

[99] N. Nethercote and J. Seward. "Valgrind". *ACM SIGPLAN Notices*, volume 42, no. 6, p. 89, 2007.

[100] M. Coffin and M. J. Saltzman. "Statistical Analysis of Computational Tests of Algorithms and Heuristics". *INFORMS Journal on Computing*, volume 12, no. 1, pp. 24–44, 2000.

[101] R. Sedgewick and P. Flajolet. *Analysis of Algorithms*. Addison-Wesley, 2013.

[102] T.-D. Nguyen, B. Schmidt, and C.-K. Kwoh. "Fast dendrogram-based OTU clustering using sequence embedding". *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics - BCB '14*, pp. 63–72. ACM Press, New York, New York, USA, 2014.

[103] T. D. Nguyen, B. Schmidt, Z. Zheng, et al. “Efficient and Accurate OTU Clustering with GPU-based Sequence Alignment and Dynamic Dendrogram Cutting”. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, volume PP, no. 99, p. 1, 2015.

[104] G. Blackshields, F. Sievers, W. Shi, et al. “Sequence embedding for fast construction of guide trees for multiple sequence alignment”. *Algorithms for Molecular Biology*, volume 5, p. 21, 2010.

[105] F. Sievers, A. Wilm, D. Dineen, et al. “Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega”. *Molecular Systems Biology*, volume 7, p. 539, 2011.

[106] N. Linial, E. London, and Y. Rabinovich. “The Geometry of Graphs and Some of Its Algorithmic Applications”. *Combinatorica*, volume 15, pp. 577–591, 1994.

[107] N. Bell and J. Hoberock. “Thrust: A Productivity-Oriented Library for CUDA”. *GPU Computing Gems*, pp. 359–371. Morgan Kaufmann Pub, jade edition, 2011.

[108] R. Killick, P. Fearnhead, and I. A. Eckley. “Optimal Detection of Changepoints With a Linear Computational Cost”. *Journal of the American Statistical Association*, volume 107, no. 500, pp. 1590–1598, 2012.

[109] C. Quince, A. Lanzen, R. J. Davenport, et al. “Removing noise from pyrosequenced amplicons”. *BMC Bioinformatics*, volume 12, no. 1, p. 38, 2011.

[110] R. C. Edgar, B. J. Haas, J. C. Clemente, et al. “UCHIME improves sensitivity and speed of chimera detection”. *Bioinformatics*, volume 27, no. 16, 2011.

[111] R. C. Edgar. “Local homology recognition and distance measures in linear time using compressed amino acid alphabets”. *Nucleic Acids Research*, volume 32, no. 1, pp. 380–385, 2004.

[112] D. Gusfield. *Algorithms on strings, trees, and sequences.* Cambridge University Press, 1997.

[113] V. M. R. Muggeo and G. Adelfio. "Efficient change point detection for genomic sequences of continuous measurements." *Bioinformatics*, volume 27, no. 2, pp. 161–6, 2011.

[114] N. A. James and D. S. Matteson. "ecp: An R Package for Nonparametric Multiple Change Point Analysis of Multivariate Data", 2013.

[115] F. E. Angly, D. Willner, F. Rohwer, et al. "Grinder: a versatile amplicon and shotgun sequence simulator". *Nucleic Acids Research*, volume 40, no. 12, p. e94, 2012.

[116] T. Z. DeSantis, P. Hugenholtz, N. Larsen, et al. "Greengenes, a chimera-checked 16S rRNA gene database and workbench compatible with ARB". *Applied and Environmental Microbiology*, volume 72, no. 7, pp. 5069–72, 2006.

[117] M. D. Dumas, S. W. Polson, D. Ritter, et al. "Impacts of poultry house environment on poultry litter bacterial community composition". *PloS One*, volume 6, no. 9, p. e24785, 2011.

[118] S. Balzer, K. Malde, A. Lanzén, et al. "Characteristics of 454 pyrosequencing data–enabling realistic simulation with flowsim". *Bioinformatics*, volume 26, no. 18, pp. i420–5, 2010.

[119] C. Quince, A. Lanzén, T. P. Curtis, et al. "Accurate determination of microbial diversity from 454 pyrosequencing data." *Nature Methods*, volume 6, no. 9, pp. 639–641, 2009.

[120] A. N. Albatineh, M. Niewiadomska-Bugaj, and D. Mihalko. "On Similarity Indices and Correction for Chance Agreement". *Journal of Classification*, volume 23, no. 2, pp. 301–313, 2006.

[121] N. X. Vinh, J. Epps, and J. Bailey. "Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance". *The Journal of Machine Learning Research*, volume 11, pp. 2837–2854, 2010.

[122] W. M. Rand. "Objective criteria for the evaluation of clustering methods". *Journal of the American Statistical Association*, volume 66, no. 336, pp. 846–850, 1971.

[123] N. J. Gotelli and R. K. Colwell. "Quantifying biodiversity: procedures and pitfalls in the measurement and comparison of species richness". *Ecology Letters*, volume 4, no. 4, pp. 379–391, 2001.

[124] A. Chao. "Estimating the population size for capture-recapture data with unequal catchability". *Biometrics*, volume 43, no. 4, pp. 783–91, 1987.