Cloud-based Bioinformatics Framework for Next-Generation Sequencing Data

Liren Huang

June 18, 2019
Version: The original thesis

Bielefeld University

Faculty of Technology
DiDy graduate school
Computational Metagenomics Group

Dissertation

Cloud-based Bioinformatics Framework for Next-Generation Sequencing Data

Liren Huang

1. Reviewer Alexander Sczyrba
Faculty of Technology
Bielefeld University

2. Reviewer Alexander Goesmann
Institute for Bioinformatics and Systems Biology
Justus-Liebig-Universität Gießen

Supervisor Alexander Sczyrba

June 18, 2019

Liren Huang
Cloud-based Bioinformatics Framework for Next-Generation Sequencing Data
Dissertation, June 18, 2019
Reviewers: Alexander Sczyrba and Alexander Goesmann
Supervisor: Alexander Sczyrba

Bielefeld University
Computational Metagenomics Group
DiDy graduate school
Faculty of Technology
Universitaetsstrasse 25
33615 Bielefeld , Germany

Abstract

The increasing amount of next-generation sequencing data introduces a fundamental challenge on large scale genomic analytics. Storing and processing large amounts of sequencing data requires considerable hardware resources and efficient software that can fully utilize these resources. Nowadays, both industrial enterprises and nonprofit institutes are providing robust and easy-access cloud services for studies in life science. To facilitate genomic data analyses on such powerful computing resources, distributed bioinformatics tools are needed. However, most of existing tools have low scalability on the distributed computing cloud. Thus, in this thesis, I developed a cloud based bioinformatics framework that mainly addresses two computational challenges: (i) the run time intensive challenge in the sequence mapping process and (ii) the memory intensive challenge in the *de novo* genome assembly process.

For sequence mapping, I have natively implemented an Apache Spark based distributed sequence mapping tool called Sparkhit. It uses the q-gram filter and Pigeonhole principle to accelerate the speeds of fragment recruitment and short read mapping processes. These algorithms are implemented in the Spark extended MapReduce model. Sparkhit runs 92–157 times faster than MetaSpark on metagenomic fragment recruitment and 18–32 times faster than Crossbow on data pre-processing.

For *de novo* genome assembly, I have invented a new data structure called Reflexible Distributed K-mer (RDK) and natively implemented a distributed genome assembler called Reflexiv. Reflexiv is built on top of the Apache Spark platform, uses Spark Resilient Distributed Dataset (RDD) to distributed large amount of k-mers across the cluster and assembles the genome in a recursive way. As a result, Reflexiv runs 8-17 times faster than Ray assembler and 5-18 times faster than AbySS assembler on the clusters deployed at the de.NBI cloud.

In addition, I have incorporated a variety of analytical methods into the framework. I have also developed a tool wrapper to distribute external tools and Docker containers on the Spark cluster. As a large scale genomic use case, my framework processed 100 terabytes of data across four genomic projects on the Amazon cloud in 21 hours. Furthermore, the application on the entire Human Microbiome Project

shotgun sequencing data was completed in 2 hours, presenting an approach to easily associate large amounts of public datasets with reference data.

Thus, my work contributes to the interdisciplinary research of life science and distributed cloud computing by improving existing methods with a new data structure, new algorithms, and robust distributed implementations.

Acknowledgements

I would like to express my most sincere gratitude and appreciation to my supervisors, Dr. Alexander Sczyrba and Prof. Dr. Alexander Goesmann, for their patiences, times, supports, guidances, and all their efforts.

I would also like to thank my colleagues: Jan Krüger, Peter Belman, and Christian Henke for the technical supports to my research project.

I am very grateful to Prof. Dr. Colin Collins and Dr. Faraz Hach for hosting me at Vancouver and providing a comfortable working atmosphere in the research group.

My deepest thanks to Dr. Roland Wittler and Prof. Dr. Jens Stoye for bringing me on board the big family of the DiDy international research training group.

To all the lovely DiDy students and friends: Georges Hattab, Tina Zekic, Markus Lux, Zhu Lu, Jia Yu, Nicole Althermeler, Kostas Tzanakis, Lukas Pfannschmidt, Benedikt Brink, Guillaume Holley, Tizian Schulz, Linda Sundermann, Omar Castillo and many more. It was an amazing four years together with all you guys.

I appreciate DFG and DiDy graduate school for the generous funding.

Finally I would like to thank my mother Xiaobo Huang and my father Chengsheng Huang for supporting me on pursuing my doctoral study in Germany.

Contents

1 Introduction

> *Mere data makes a man. A and C and T and G. The alphabet of you. All from four symbols. I am only two: 1 and 0*
>
> **— Joi**
> (A.I. of the movie *Blade Runner 2049*)

1.1 The big data challenge in life science

As the genetic material of living organisms, nucleic acids or "nucleins" were first discovered by the young Swiss doctor Friedrich Miescher in the winter of 1868/9 (Dahm, 2008). They are usually formed in molecular sequences consisting of four nucleotides: adenine, cytosine, guanine, and thymine (A, C, G, and T). The complete collection of nucleic acid sequences builds up the deoxyribonucleic acids (DNA) and ribonucleic acids (RNA) in an organism. Followed by the discovery of the DNA double helix structure in 1953 (Watson and Crick, 1953), the primary objective in life science studies is to decode the sequencial formation of the four nucleotides within a certain genome (Fig. 1.1A). However, to obtain and reconstruct the complete sequence of the genome can be challenging as the size of a genome ranges from two millions to several billions of nucleotides, making it impossible for modern technologies to read the genome in one continuous sequence.

Instead of reading the entire genome in a continuous sequence, whole genome shotgun (WGS) sequencing technology was introduced to capture the fragments of a genome in parallel and redundant ways (Green, 2001). By using the overlap information of the sequenced fragments (Fig. 1.1B and C), researchers are able to reconstruct most of the original genome with the help of computational tools (Chaisson et al., 2015). In recent decades, next-generation sequencing (NGS) technologies have been widely used by life scientists in nucleic acid studies. Moreover, the improving productivity and low running costs of NGS have introduced an exponential increase in sequencing data (Goodwin et al., 2016). Nowadays, a single flow cell of an Illumina sequencer can generate hundreds of gigabytes (10^9 bytes) raw data. These raw data sets are usually archived and stored in public data centers such as the European Nucleotide Archive (ENA) and the Sequence Read Archive

(SRA). The latter hosted around 14 petabases (10^{15} nucleotides) of raw data and its size is doubling every 10-20 months (Langmead and Nellore, 2018). Such a large amount of archived data not only comes from individual research laboratories, but also from different genome project consortia (Fig. 1.2A).

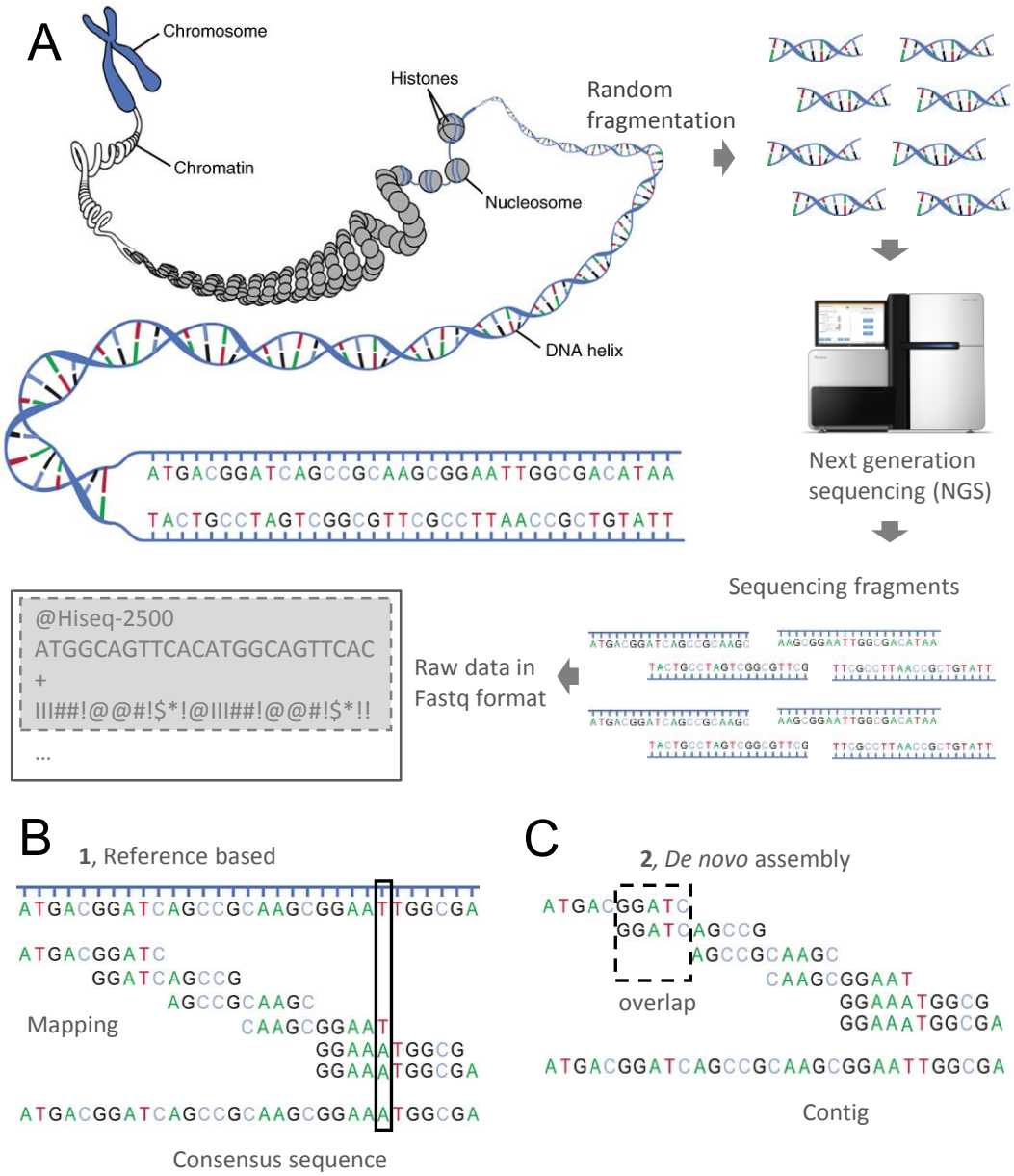

Fig. 1.1: Nucleic acids and next-generation sequencing: **(A)** The DNA sequences of the original genome are randomly fragmented and sequenced in a redundant way (Betts et al., 2013); **(B)** The original genome is reconstructed by mapping the fragments back to a reference template and building a consensus sequence; **(C)** Genome reconstruction using overlaps between fragments.

After the accomplishment of the Human Genome Project (HGP) in April 2003 (Collins et al., 2003), several consortia have initiated different genome projects with more specific focuses on different research domains (Fig. 1.2B). For instances, the 1000 Genome Project aims to provide a detailed catalogue of human genetic variants by

sequencing more than a thousand human individuals (Auton et al., 2015). To that end, the particular consortium has collected and sequenced 2504 human individuals from 26 populations in the 3rd phase of the project which has generated more than 100 terabytes (TB) of sequencing data. In the agricultural domain, the 3000 Rice Genome Project consortium sequenced more than 3000 rice samples which has generated more than 200TB of sequencing data (Consortium, 2014). This data enables a large scale discovery of novel alleles that are correlated to the changing environment. For microbial studies, metagenomic whole genome shotgun sequencing technology is frequently used to study the aggregated microbial community in an given environment. The Human Microbiome Project (HMP) investigates how the microbiome impacts human health and disease (Nih Hmp, Peterson, et al., 2009). The consortium sequenced the metagenomes of 5 major body sites and generated 5TB of metagenomic WGS data. These genome project consortia not only conducted comprehensive investigations into the sequencing data, but also provided large data warehouses for researchers to carry out association studies between the public datasets and their private datasets.

To carry out association studies using public datasets, researchers must tackle three computational problems:

1. transferring large amounts of raw data from public storage to computational resources

2. insufficient computational resources (hard disks, memories and processors) to handle large-scale sequencing data

3. high run time and memory consumptions using conventional bioinformatics tools to process the data.

For most research laboratories, a combination of high performance computing (HPC) clusters (hardware) and bioinformatics tools (software) is used to carry out their genomic studies.

On the hardware side, an HPC cluster provides a considerable amount of random access memories (RAMs) and central processing units (CPUs) for bioinformatics tools to carry out corresponding analyses. All computational tasks are usually scheduled by a queuing system such as the sun grid engine (SGE), where parallelizations are done by manually splitting and distributing smaller batches (Droop, 2016). Thus, by using an HPC cluster, researchers are able to address the insufficiency of computational resources. However, it also has two shortcomings: (i) significant costs to setup and manage the cluster; (ii) computational resources idling from time to time.

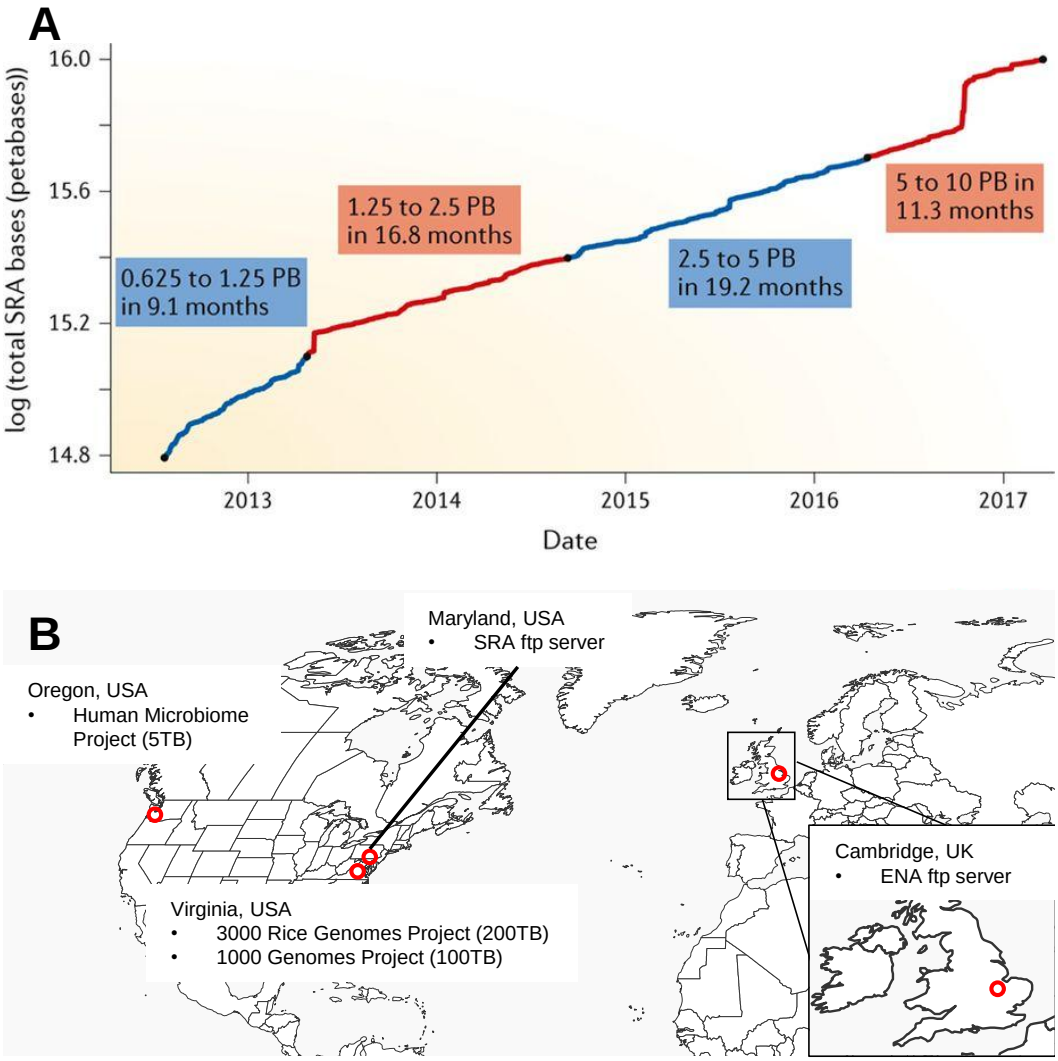

Fig. 1.2: NGS data increment and storage: **(A)** Archived NGS data in the SRA database doubled four times from July 2012 to March 2017 (Langmead and Nellore, 2018); **(B)** Different locations for public data storage and cloud storage.

On the software side, the first and most resource consuming step for analyzing sequencing data is to assemble the original genome using the sequenced fragments (sequencing reads). There are two approaches for the assembly: (i) reference based genome assembly and (ii) *de novo* genome assembly.

For model organisms whose genomes have been well assembled, reference based assembly is usually used (Fig. 1.3B). For this approach, the goal is to detect structural variants that differ from the reference genome templates. Thus, short read alignment is used to map sequencing read to the reference genome. Because of sequencing errors and structural variances (e.g. single-nucleotide polymorphism), the mapping process has to thoroughly compare each nucleotide between the sequencing reads and the reference genome. From a computational point of view, such a comparison is very time consuming.

For organisms whose genomes are unknown, *de novo* genome assembly approaches are used (Fig. 1.3C). As DNA was randomly fragmented before sequencing, the overlapping information between fragments can be used for genome reconstruction. To efficiently search for the overlap fragments and extend the fragments to longer sequences, *de novo* assemblers usually store fragments in memory. For organisms with large genome sizes (e.g. around 3 billion nucleotides for the human genome), this approach has a high demand on the size of the RAM.

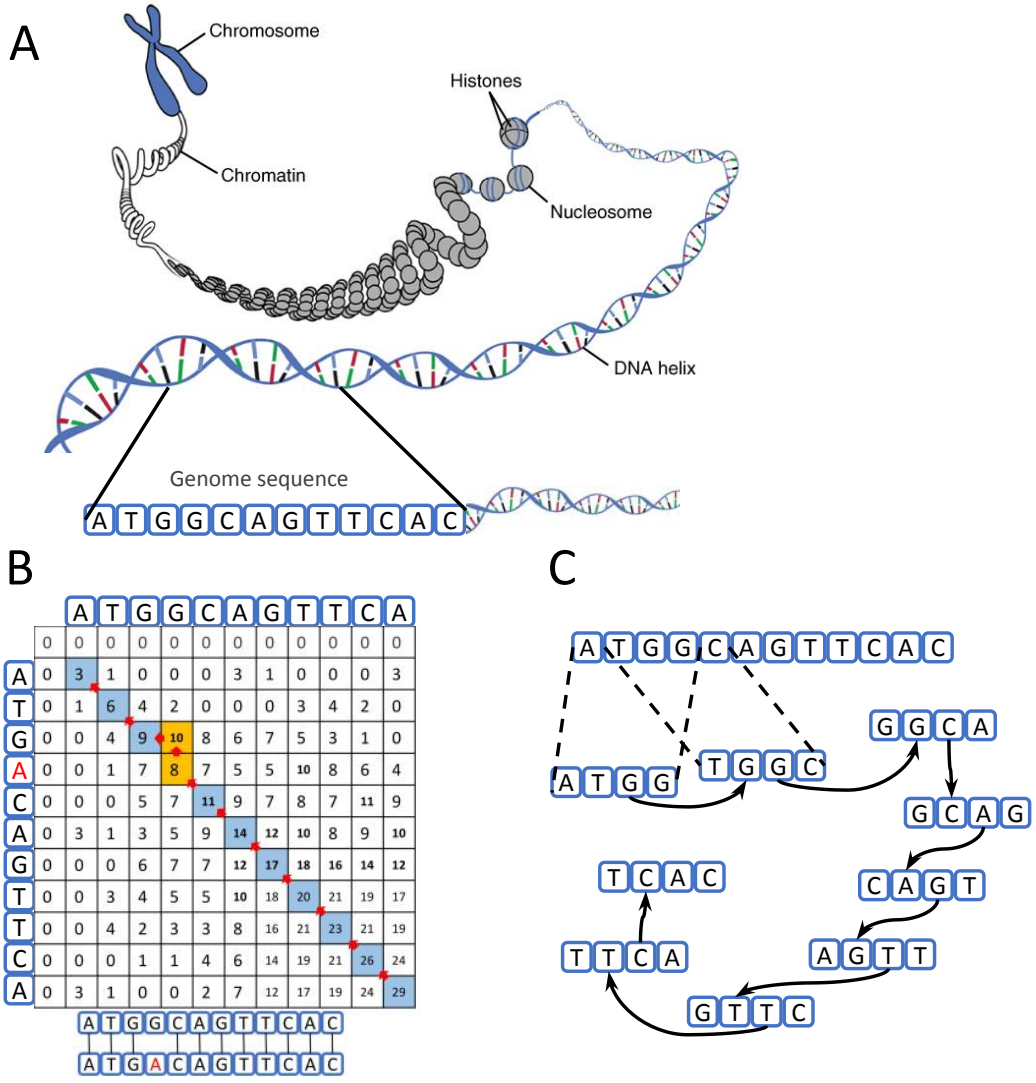

Fig. 1.3: Computational method for genome assembly: **(A)** A fragment of the genome; **(B)** reference based assembly maps sequenced fragments back to a reference sequence. The mapping process is usually computationally time consuming; **(C)** *de novo* assembly uses overlap information of the sequenced fragments to extend and reconstruct the sequence. To efficiently search the overlaps of all fragments, all sequences are stored in the memory. Thus, it is very memory consuming.

To address the big data challenge in life science, we need both easy-access to large computational infrastructures and bioinformatics tools that are compatible and scalable on such platforms.

1.2 Distributed cloud computing

When upgrading a personal computer (PC), four major hardware components are critical to its later performance: memory, hard disk, CPU, and network connection. The improvement of computing capacity on a single instance of an operating system is known as vertical scaling or "scale up" (Singh and Reddy, 2014). Yet, installing more processors or memory on one single computer instance will reach a certain limit and the cost will raise considerably. To further improve the computing capacity, multiple independent computer instances can be connected via network and organized to work together in a grid. By distributing workload across the network to each computer instance, the grid is able to process large data sets that can not fit into one single computer. This approach, which leverages multiple commodity computers for parallel computing, is known as horizontal scaling or "scale out" (Fig. 1.4). For instance, using a cluster of 80 Amazon web server (AWS) computer instances (c3.8xlarge), BiBiS3 (Henke, 2017) is able to download NGS data with an aggregate throughput of more than 22 GBytes per second (Fig. 1.5). The parallel downloading makes it possible to transfer large amounts of raw data from public storage to computational resources (see section 1.1, the first computational problem).

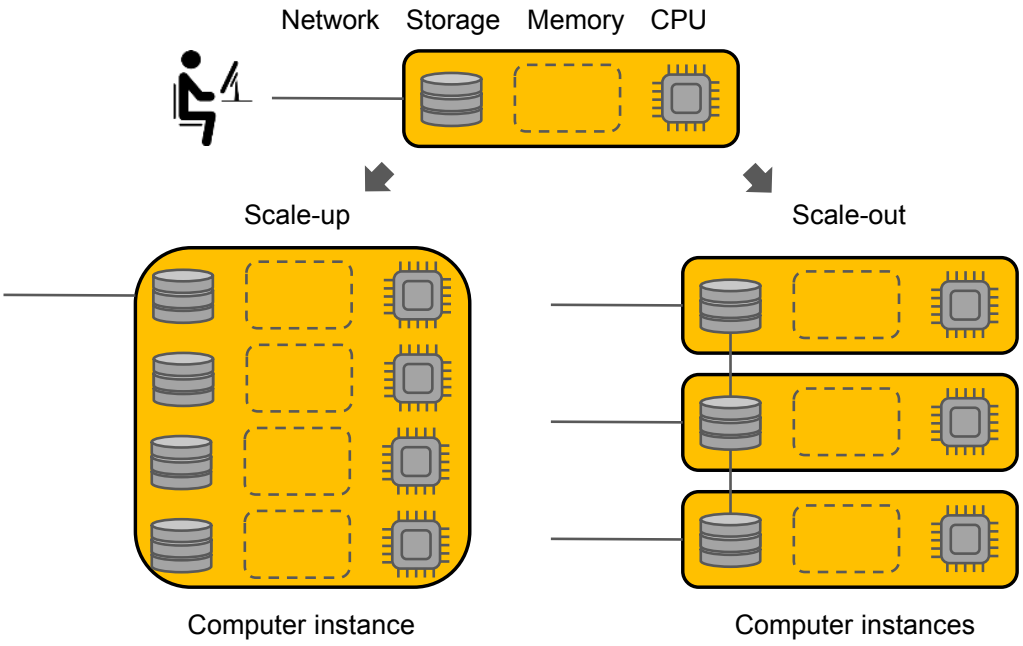

Fig. 1.4: Horizontal and vertical scaling (scale up and scale out): Scale up improves the computational capacity within one computer instance, whereas scale out connects more computer instances to increase the computational capacity

Scaling out, in theory, has no limit as long as more computers can be connected to the grid. However, there are limited software frameworks that can fully utilize and balance all computer instances. Both industrial enterprises and academic communities are developing distributed frameworks to maximize the usage of

computational resources. For data storage, *Yahoo* Inc. harnessed 25000 servers to store 25 PB of application data using a distributed system called the Hadoop distributed file system (HDFS) (Shvachko et al., 2010). For data processing, The more recent Apache Spark framework sorted 1 PB of data on 190 machines in under 4 hours (Zaharia et al., 2012). With the help of distributed frameworks and platforms, managing and using large pools of computers becomes much easier and more efficient. The robustness and flexibility of horizontal scaling promoted the rapid development of cloud computing.

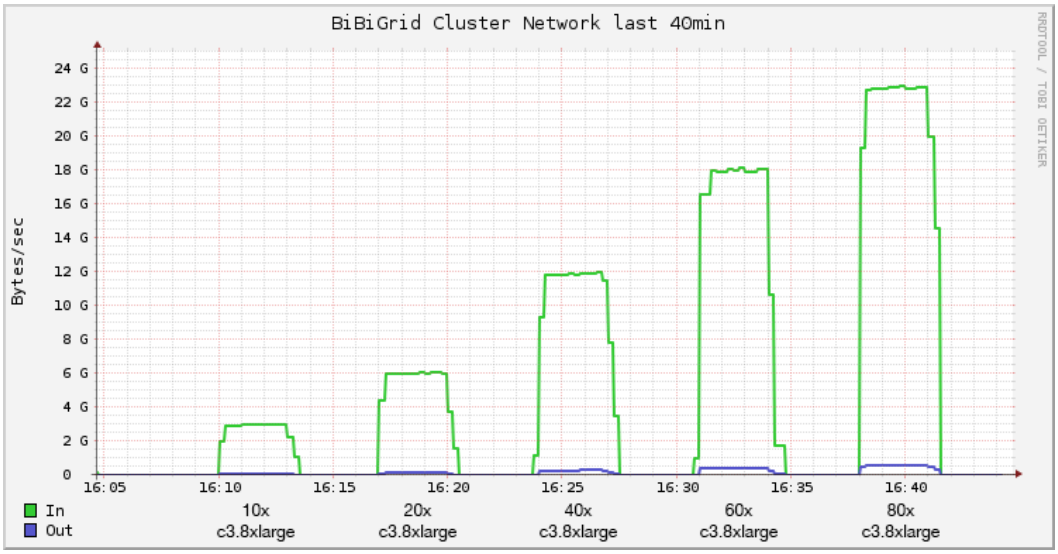

Fig. 1.5: Scaling out download workloads: Each computer instance has 10 Gigabit/s bandwidth. The test data sets are the NGS data of the human microbiome genome project stored on the AWS cloud in Oregon, USA region. All data were downloaded in parallel to a cluster located in Frankfurt, European region. The figure is a screen shot from Ganglia network I/O monitor (Henke, 2017).

Cloud computing, as defined by the US National Institute of Standards and Technology (NIST), is "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" (Mell and Grance, 2011). To simplify, it is a model of providing easy-access to on demand computational resources. These resources can be offered from lower-level computing infrastructure to higher-level platforms and software, also known as: infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). IaaS provides hardware such as CPUs, memories, hard disks, network bandwidth, and virtual machines (VMs). PaaS provides cloud-based platforms for users to run software in a distributed fashion, such as the MapReduce model (Dean and Ghemawat, 2008). SaaS directly provides executable software for various data analyses (Fig. 1.6).

In the industrial domain, major commercial enterprises, such as Amazon Web Service (AWS), Google cloud, and Microsoft Azure, provide all three levels of service for

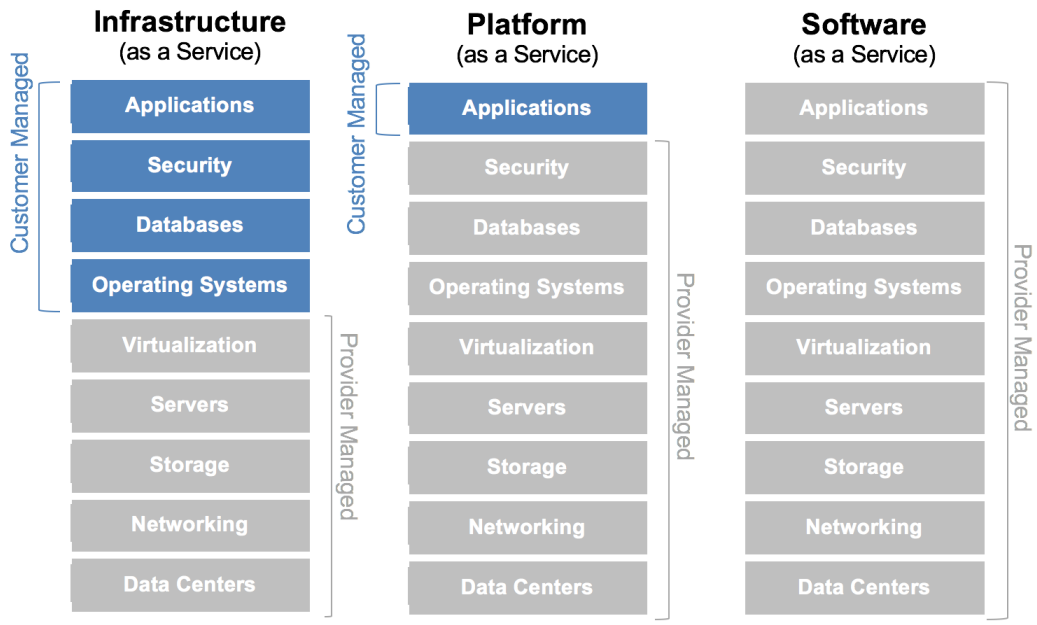

Fig. 1.6: Categories of cloud services. The figure is modified based on (*Ensi, 2017*).

general purposes. The services are charged based on how much resource is rented out and how long it will be rented, known as the Pay-as-you-go model. The advantages of the Pay-as-you-go cloud are (i) fast run time, (ii) low cost, and (iii) elastic scalability. Instead of spending 1000 hours running on a single computer, users can easily setup 1000 computers for 1 hour of computational run time on a cloud. Thus, with the same cost, the cloud fulfills computational requests quicker (Langmead and Nellore, 2018). Cloud service providers, such as AWS, introduce bidding systems that offer lower prices to avoid computers idling in its computing center. The bidding price is normally offered to compensate the electricity cost in a computing center. Therefore, using bidding price could further reduce the cost. In addition, without investing cluster management effort, users could easily configure their cloud clusters on a command console. Once the cluster is setup, it can be further re-configured based on the real time computational intensity. These features of the cloud make it suitable for genomic researchers to handle large-scale NGS data sets (see section 1.1, the second computational problem).

Conducting genomic studies on the cloud requires users to comprehend certain amount of knowledge on cloud computing. In the bioinformatics domain, there are nonprofit cloud services for genomic studies, such as the German Network for Bioinformatics Infrastructure (de.NBI), the Embassy Cloud, and the European Open Science Cloud. These cloud services provide platforms and software with a particular focus on bioinformatics applications. The platforms facilitate users to run bioinformatics software on the cloud. But most of them are built on top of a distributed system, such as the MapReduce programming model (Dean and

Ghemawat, 2008). Although existing bioinformatics tools have been developed and utilized on single computers (Langmead and Salzberg, 2012; Li and Durbin, 2009; Niu et al., 2011; Wood and Salzberg, 2014; Li et al., 2009; Li et al., 2008; Bray et al., 2016), most of the parallelizations on multi-computer networked clusters are done by manually splitting and scheduling in batches (Droop, 2016). To be compatible with a cloud environment, methods involving message passing and graph representation between computers must be re-implemented with higher level programming interfaces (Gropp et al., 1996).

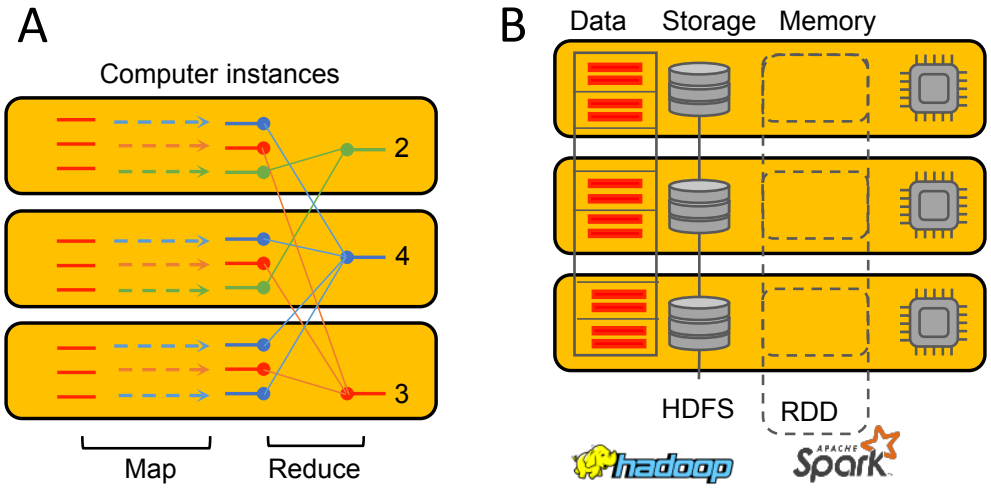

Fig. 1.7: Distributed computational model and frameworks: **(A)** An example of record counting in the MapReduce programming model. Each yellow box represents a computer instance. **(B)** Distributed data storage in the Hadoop distributed file system (HDFS) and the distributed memory cache in the resilient distributed datasets (RDD). Red dashes represent data partitions in a file.

To fully exploit distributed cloud computing systems, bioinformatics methods and programs should be (i) scalable, (ii) fault tolerant, and (iii) platform independent. In genomics applications, there are several tools (e.g. ABySS (Simpson et al., 2009) and Ray (Boisvert et al., 2010)) that uses the message passing interface (MPI) for distributed implementations. However, programming on top of MPI has to tackle thread synchronization and load balance. In addition, the performance of MPI-based assemblers is bonded to the performance of the network. The MPI-based Assemblers have much faster run times on an InfiniBand connected cluster than an Ethernet connected one. Moreover, there is no complete fault tolerance mechanism built inside of MPI. The conventional Apache Hadoop framework is designed to offer higher scalability and a supervised fault tolerance mechanism by providing a distributed data storage system called HDFS (Shvachko et al., 2010) and a distributed computing engine, Hadoop MapReduce (Dean and Ghemawat, 2008). The MapReduce model consists of 'map' and 'reduce' steps, where 'map' carries out independent processes and 'reduce' summarises the pre-computed results (Fig. 1.7A). However, Hadoop MapReduce reads and writes intermediate results to a distributed disk storage, hence

introducing a significant overhead for iterative algorithms. Furthermore, Hadoop has limited options for handling distributed data. Thus, more efforts are needed to implement MapReduce-based algorithms on top of Hadoop.

The more recent Apache Spark framework addresses these weaknesses with its unique data sharing primitive, called resilient distributed datasets (RDD) (Zaharia et al., 2012). RDD offers a 'cache' function to store distributed data in the memory across computers on a cluster (Fig. 1.7B), thus, avoiding run time overhead of iterative data input and output (I/O). Moreover, Spark has more build-in functions for RDD to facilitate methods implementation and data handling via its application programming interface (API). These advantages make Spark suitable for large-scale genomic data analyses.

1.3 Thesis structure

In this thesis, I will present my work on a cloud-based bioinformatics framework that facilitates NGS data analysis on the distributed cloud environment. The main contributions of the thesis can be summarized in three parts:

1. Sparkhit, a distributed implementation on top of the Apache Spark platform for short read mapping and fragment recruitment that addresses a run time intensive problem in NGS data analysis.

2. Reflexiv, a distributed *de novo* genome assembler using a newly developed data structure called Reflexible Distributed K-mer (RDK) that addresses a memory intensive problem in NGS data analysis.

3. Rapid analysis of large-scale NGS data associated with the public datasets on the cloud using different functional modules in our framework.

The main content of the thesis is structured as follows:

Chapter 2: Related work

First, I will introduce the current state-of-the-art methods and technologies for distributed sequence alignment and genome assembly. These methods are mainly implemented on top of the Apache Hadoop and Spark platform. Thus, I will start by giving a brief introduction into the Apache Hadoop and Spark eco-system. For existing bioinformtics tools, I will introduce their algorithms and their distributed implementations. I will also introduce the limitations of existing approaches on top of Hadoop and Spark. Hadoop-based tools mostly suffer from the overhead of iterative data I/O on a distributed storage system. As for Spark-based tools, the

algorithms implemented on the distributed system are not efficient and introduce large amounts of messages passing through the network that impacts their run time performances.

Chapter 3: Sparkhit: Distributed sequence alignment

Then, I will present Sparkhit, an open source computational framework that is easy to use on a local cluster or on the cloud. Sparkhit is built on top of the Apache Spark platform, integrates a series of analytical tools and methods for various genomic applications: (i) I have natively implemented a metagenomic fragment recruitment tool and a short-read mapping tool (Sparkhit-recruiter and Sparkhit-mapper). The short-read mapper implements the pigeonhole principle to report the best hit of a sequencing read. Whereas the fragment recruitment tool implements the q-gram algorithm to allow more mismatches during the alignment, such that extra information is provided for the metagenomic analysis; (ii) For using external software on Sparkhit, I built a general tool wrapper (Sparkhit-piper) to invoke and parallelize existing executables, biocontainers (e.g. Docker containers (Merkel, 2014)) and scripts; (iii) For downstream data mining, I integrated and extended Spark's machine learning library. All methods and tools are programmed and implemented in a new MapReduce model extended by Spark, where parallelization is optimized (load balanced) and supervised (fault tolerance).

The benchmarks on Sparkhit demonstrated its high scalability. In comparison, Sparkhit ran 18 to 32 times faster than Crossbow (Langmead et al., 2009) on data preprocessing and Sparkhit-recruiter ran 92 to 157 time faster than MetaSpark (Zhou et al., 2017) on fragment recruitment.

Chapter 4: Reflexiv: Parallel *de novo* genome assembly

As the second major implementation, I will introduce Reflexiv, an open source parallel *de novo* genome assembler and its core data structure called Reflexible Distributed K-mer (RDK). It is also built on top of the Apache Spark platform, uses Spark RDD to distribute large amounts of k-mers across the cluster and assembles the genome in a recursive way. Comparing RDK to the state-of-the-art *De Bruijn* graph, RDK stores only the nodes of the graph (k-mers) and discards all the edges. Since all k-mers are distributed in different compute nodes, RDK uses a random k-mer reflecting method to reconnect the nodes across the cluster (a reduce step of the MapReduce paradigm). This method iteratively balances the workloads between each node and assembles the genome in parallel.

The main contribution of Reflexiv is a new distributed data structure and a recursive implementation that leverages the memories of multiple instances in a standard ethernet connected cluster. Reflexiv ran slightly faster than existing distributed *de novo* assemblers, Ray and AbySS, on a single computer instance. When scaling out

to more instances, Reflexiv is 8-17 times faster than Ray and 5-18 times faster than AbySS on the Ethernet interconnected clusters deployed at the de.NBI cloud.

Chapter 5: Large-scale genomic data analyses

As the third part of the thesis, I will focus on showcasing a cloud application of my framework. Utilizing the powerful AWS compute cloud, my framework quickly analyzes large amounts of genomic data. My use case presents a 21 hours "pay-as-you-go" cloud application that analyzed 100 TB genomic data from 3 genome projects and a transcriptomics study (Wyatt et al., 2014). The analysis on the Human Microbiome Project (HMP), associates public 'big data' with private datasets, demonstrates how Sparkhit can be widely applied on different genomic studies. Thus, my framework enables the broader community to engage genomic investigations by leveraging cloud computing resources.

Chapter 6: Conclusion and outlook

In the end, I will conclude the contributions of my work and outlook the future works for the project. The thesis focuses on addressing the big data challenge in life sciences by leveraging distributed computing resources. In particular, I have implemented two distributed algorithms on top of the Apache Spark platform to enhance the performances of sequence alignment (run time intensive) and *de novo* genome assembly (memory intensive). Both approaches have significant run time improvements compared to existing tools. I also present an application on the cloud using my distributed bioinformatics framework. The application presents a use case on how to easily access and rapidly analyze large amounts of NGS data on the Amazon AWS cloud. Nevertheless, to further improve the framework, two functions can be added to the current implementations. For sequence alignment, the overhead of broadcasting reference index can be compensated by integrating a parallel reference download function. As for the *de novo* assembler, the pipeline of the assembler ends when the contigs are assembled (see discussion in chapter 4). Thus, I will add an extra scaffolding module in the assembler's pipeline for the downstream assembly process.

Related Work

2

> *If I have seen further, it is by standing upon the shoulders of giants*
>
> — **Isaac Newton**
> (Physicist)

2.1 The Apache Hadoop and Spark frameworks

When developing software for a distributed system, the Apache Hadoop and Spark frameworks are the two well known options for distributed implementations. The Hadoop project was first started in 2004 at Yahoo Inc. (Dean and Ghemawat, 2004). Its package mainly consists of a distributed storage module, known as Hadoop distributed file system (HDFS) (Shvachko et al., 2010), and a data processing module called Hadoop MapReduce (Dean and Ghemawat, 2008). HDFS is a distributed file system for storing large amounts of data on each computer node across a cluster. Whereas, MapReduce is a programming scheme for parallel data processing. The weakness of the MapReduce paradigm is its inefficiency for iterative algorithms. For the conventional MapReduce model, each 'map' step reads data from the disk and writes the processed result back to the disk. In an iterative implementation, the program reads and writes the data from/to the disk in a iterative loop. In such case, the overhead of disk access will be significantly magnified.

The Apache Spark framework was then introduced to tackle such problems. It was developed by the Algorithms Machines People (AMP) lab at the University of California at Berkeley (Zaharia et al., 2012). Spark introduces a new data abstraction called resilient distributed datasets (RDD). The major function of RDD is its ability to cache distributed dataset into memories of a distributed cluster. Such design overcomes the I/O overhead problem in the Hadoop MapReduce model. Moreover, the programming interface of RDD provides a variety of functions to facilitate distributed implementations.

Both frameworks are highly compatible with a collection of distributed computing modules and frameworks that provide additional functionalities. For instance, Hadoop can directly access data records from distributed databases (e.g. HBase).

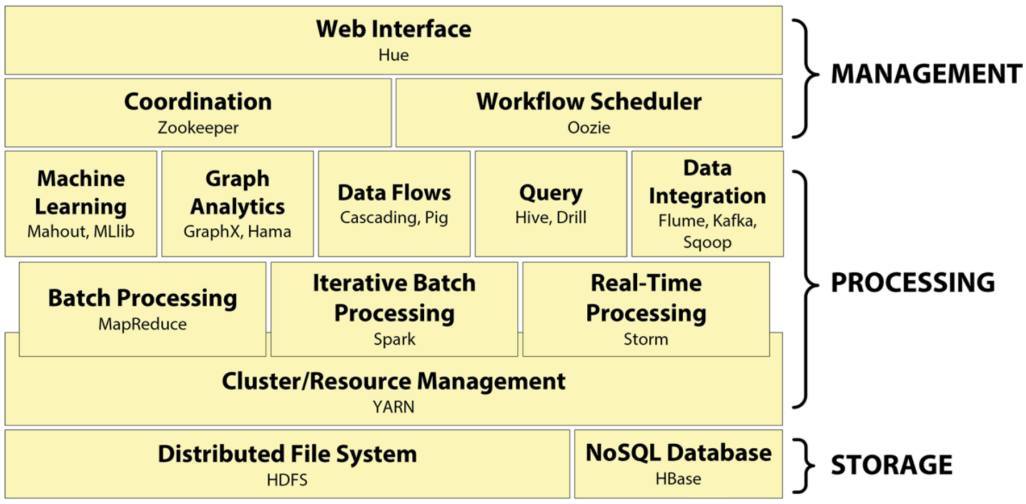

Fig. 2.1: The Hadoop ecosystem (Landset et al., 2015)

Spark Machine Learning Library (Spark-MLLIB) provides a series of distributed implementations of machine learning algorithms. This collection of distributed frameworks and functional modules is known as the Hadoop and Spark ecosystem (Fig. 2.1). Such an ecosystem increases the flexibility of Hadoop and Spark on the cloud environment. Therefore, Hadoop and Spark based tools are easily portable to other cloud platforms.

Spark and Hadoop excel other frameworks in the cloud environment with two native features: (i) fault tolerant mechanism and (ii) high scalability. To understand how Spark and Hadoop provide such features, let us start by looking at the topologies of Hadoop and Spark clusters.

2.1.1 Cluster topology

Both Hadoop and Spark cluster architectures can be simplified as master-worker networks. To organize all computer nodes, a master node is selected to supervise worker nodes, allocate resources, and balance workloads. Whereas worker nodes carry out assigned tasks.

In a distributed cluster of commodity computers, errors occur more frequently. A single failure in the system can jeopardize the entire process or the complete cluster. Fault tolerance is a mechanism of distributed frameworks to counter failures in a distributed system. The failure can be an error in a single computing process or a malfunction of a computer node. To prevent such failures, the master node constantly monitors not only the heartbeats of all worker nodes, but also the statuses of all processes. For instance, on a Hadoop distributed file system, the master node (named by HDFS as the 'name node') checks the heartbeats of all worker nodes

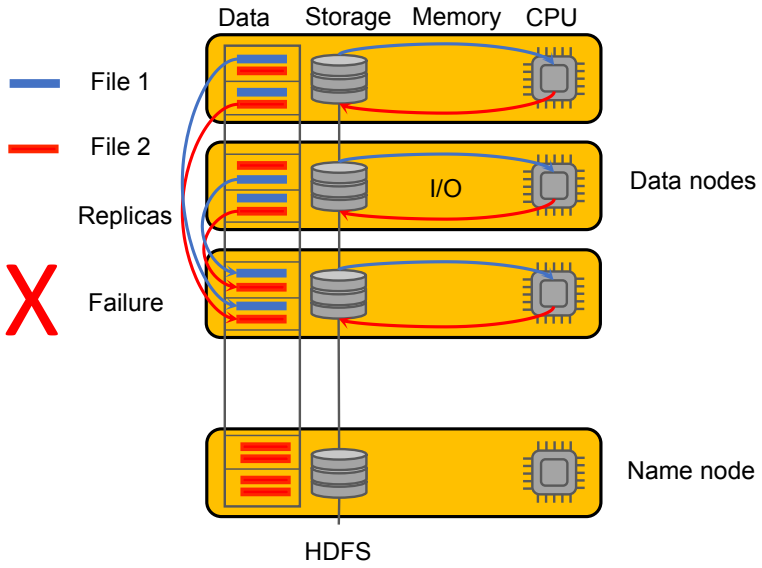

Fig. 2.2: The fault tolerant mechanism of HDFS: The blue and red dashes represent data blocks replicated and distributed by HDFS. In the event of a data node failure (e.g. data node disconnected to the name node), HDFS is able to recover the data using the replicas from other data nodes.

(named as 'data nodes'). When storing data on HDFS, the files are split into blocks and replicated onto several data nodes (Fig. 2.2). Once a data node is offline, the name node can recover the file using the replicated blocks stored on the other data nodes. When running a task on a Spark cluster, the master node (i.e. driver node) checks the statuses of all parallel processes. A Spark computing process consists of a series of executing operations. When a certain operation fails during the process, Spark trace back its parent operations (called lineage) and re-executes the failed process (Fig. 2.3). If the complete worker node failed during the process, the Spark driver node re-schedules the failed chain of operations to another worker node for continue processing.

Moving a large amount of data through a network is time-consuming. The advantage of distributed computing is to process and manipulate data locally on each node of the cluster. This way, each node can efficiently access and handle the data, thus maximizing their processing capacity. When storing a file on the Hadoop cluster, HDFS splits the file into blocks and distributes them redundantly on each data node throughout the cluster. To read and process files stored on the HDFS, Hadoop-MapReduce distributes tasks to each data node and each task processes the data blocks directly from local storage. Spark can also directly load data blocks from HDFS into RDD. An RDD allocates data into different chunks, known as partitions. Each partition carries out an independent task that processes the data stored in the partition. When loading data from HDFS, each RDD partition loads data blocks stored on the local Hadoop data node (worker node for Spark).

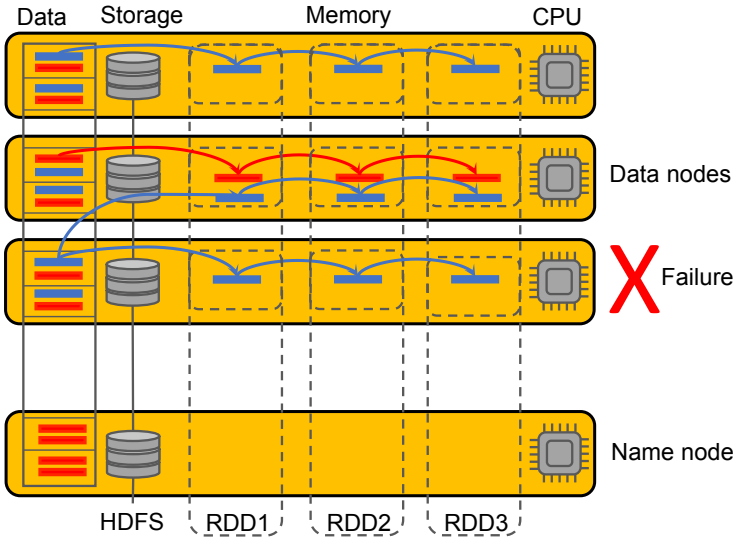

Fig. 2.3: The fault tolerant mechanism of Spark: each worker node carries out a series of operations as the linage of the task. In the event of a worker node failure, the linage of the task will be sent to anther worker node on the cluster and resumes running.

When connecting to the external storage, the distributed feature continues to benefit the performances of Hadoop and Spark clusters. For instance, the AWS Simple Storage Service (S3) is a distributed object storage system provided by Amazon cloud for persistent data storage. Files stored on the Amazon S3 are replicated and distributed across multiple servers within Amazon's data centers. When downloading a large genomic dataset from Amazon S3 to an HDFS, Hadoop-MapReduce sends download requests ('GET' requests) from each data node to the Amazon S3 server (Fig. 2.4). The data nodes download different parts of the dataset to their local disks. The Hadoop framework can also apply the same parallel download method to other object storage systems such as the OpenStack Swift service. The distributed downloading approach can fully exploit the bandwidth of distributed network connection between the cluster and the external storage.

2.1.2 Spark data processing paradigm

The Apache Spark framework has extended the classic MapReduce programming paradigm and introduced a new distributed data abstraction called resilient distributed datasets (RDD) (Zaharia et al., 2012). From a computational point of view, an RDD is a Scala/JAVA object created inside the Java virtual machine. Spark provides a collection of functions for RDD to facilitate methods implementation and data handling via its application programming interface (API). A computational pipeline on top of Spark is a series of functions applied to RDDs. These functions are distributed methods that operate on each partition of the RDD (Fig. 2.5). There are two types of functions: transformations and actions. Transformations apply

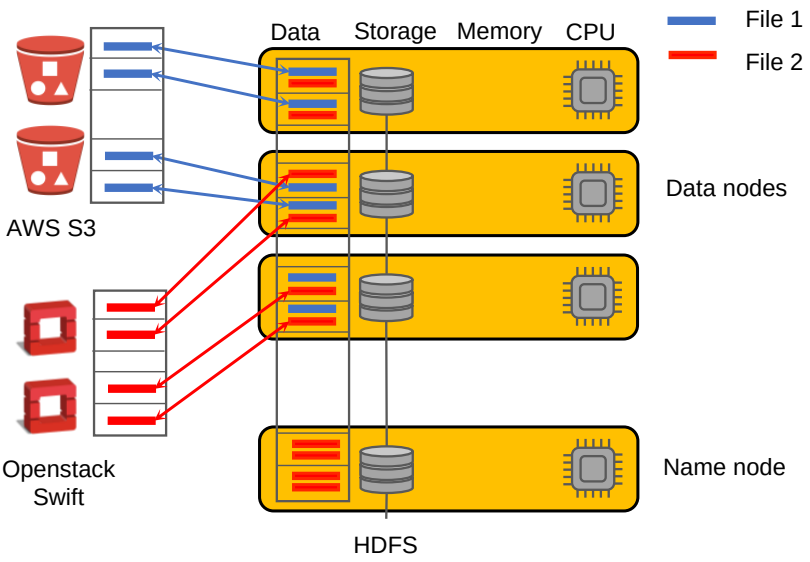

Fig. 2.4: Distributed network connection with external storages: blue and red dashes represent data blocks that are transferred independently by 'map' tasks.

functions to an RDD and send result to a new RDD. Whereas actions apply functions to one RDD and return a value to the driver program after processing. In a way, all transformations pass data between RDDs and can be carried out independently on each partition of the RDD as a workflow. As for actions, results must be summarized and sent to the driver program. For example, a 'repartition' function is a transformation that can be applied to divide an RDD into a certain number of partitions. A 'collect' function is an action that can be applied to aggregate processed results from an RDD to the driver node.

Spark introduces a 'lazy' feature for arranging tasks on a Spark cluster. When a Spark job is submitted to the cluster, the lineages of the operations are sent to each executor of the worker nodes. Transformations are not computed right away on executors. Instead, the executors continue to search the lineage of the operations until an action is found. As actions require to send results to the driver node, the executors start all transformations before the action and compute the result of each operation (Fig. 2.6). This lazy feature is designed for the fault tolerant mechanism. When a process fails during the 'map' step, the lineage of the process can be sent to another active worker node and re-computed from the last action checkpoint.

2.1.3 Sorting in Spark

In the reduce phase of the Spark extended MapReduce paradigm, intermediate results on the cluster are aggregated and summarized. To summarize results stored on different worker nodes (e.g. word counting), records with the same value must be sent to the same node for computing. The common way to do so is by sorting all

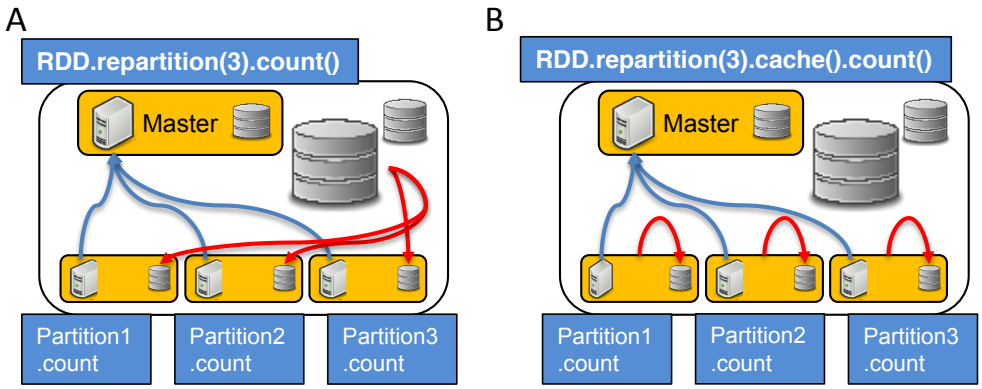

Fig. 2.5: Distributed computing on Spark clusters: **(A)** methods implemented via RDD's API will be operated on each partition of the RDD. Red lines indicate data input and blue lines indicate data output. **(B)** the 'cache' function stores distributed data in memory, so that the 'count' operation can read data directly from memory without loading from local disks.

data across the cluster. The core of sorting data in a distributed system is the 'shuffle' operation, which moves data across the worker nodes. In the shuffle operation, the task that emits the data in the source executor is 'mapper', the task that receives the data into the target executor is 'reducer', and what happens between them is 'shuffle'.

Spark sorts data in two stages: (i) the 'map' stage and (ii) the 'reduce' stage. In the map stage, each partition of the RDD is sorted locally by the executor on each worker node. After local sorting, Spark recorded the range of the sorted result, so that the reduce tasks can retrieve ranges of data quickly. The entire 'map' stage processes data locally without using the network connection between worker nodes.

In the reduce stage, the 'reducer' retrieves data in its own range from the 'mapper'. This 'shuffle' operation is carried out based on the TimSort algorithm implemented in the Spark 'sortByKey' function. TimSort is a derivation of merge sort and insertion sort. It performs better for datasets that are pre-sorted. At this stage, the performance is bounded by the network connection between worker nodes, as 'shuffle' actually takes place in this stage.

2.2 Sequence alignment and its cloud implementations

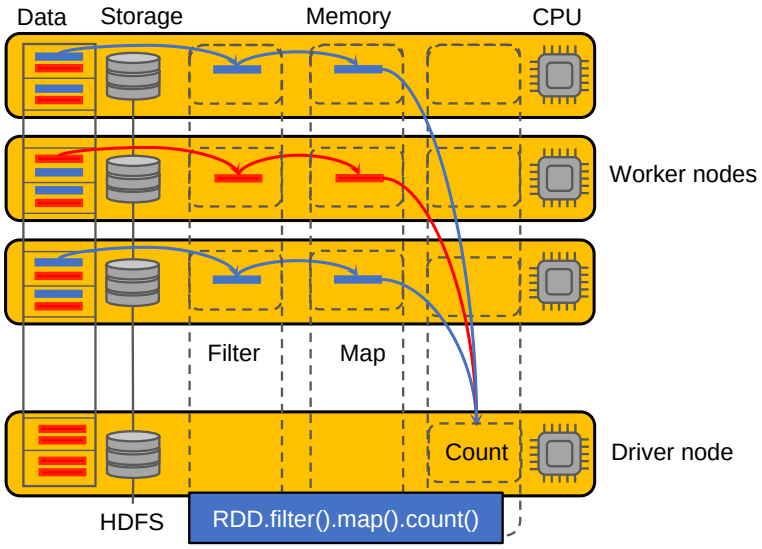

Fig. 2.6: Transformations and actions: the 'filter' and 'map' operations are transformations that operate on an RDD and send the result to a new RDD. The 'count' operation is an action that processes the data from an RDD and sends the result to the driver node. Spark only starts the job when encountering an action, which in this case is the 'count' operation.

2.2.1 Short read alignment and fragment recruitment

Pairwise sequence alignment is a method for comparing the similarity between two biological sequences (e.g. nucleotide sequences) that may have structural or functional relationships (Gollery, 2005). Since the lengths of two sequences usually vary from one another, sequence alignment can be classified into two categories: (i) global alignment and (ii) local alignment. Global alignment aligns two sequences from start to end, whereas local alignment searches for one or more short alignments describing the most similar regions within the two aligned sequences.

For short read alignment, the goal is to assign a short read (e.g. the 100 nucleotides (nt) sequence from Illumina Hiseq-2000 sequencer) to the most similar region on a reference genome. This type of alignment usually generates alignments with high similarities. However, fragment recruitment produces all possible matches between the short read and the reference genome. The goal of fragment recruitment is to recruit as many fragments as possible (even fragments with lower similarities, e.g. with only 50% identical nucleotides) to the reference genome so that more information can be used for downstream analysis. It is commonly used in metagenomic studies to understand the genome structure, evolution, phylogenetic diversity, and gene function of biological samples (Rusch et al., 2007). As a special case of sequence alignment, fragment recruitment supports more mismatches during the alignment. Thus, the computational complexity for fragment recruitment can be higher than standard short read alignment.

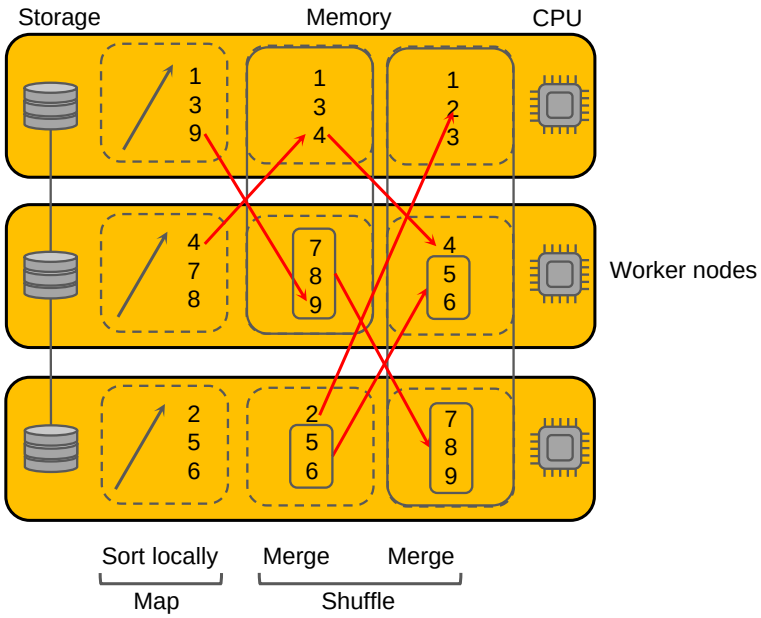

Fig. 2.7: The sorting process in a Spark cluster: the process consists of two stages: the 'Map' stage and the 'Reduce' stage. Each grey dash frame represents a partition of an RDD. The grey solid frames represent the merged result of TimSort.

2.2.2 Algorithms for sequence alignment

Due to the single nucleotide polymorphisms (SNPs) and insertions/deletions on genome sequences, sequence alignment must thoroughly compare each nucleotide of the two sequences to find differences. There are two types of methods for sequence alignment: (i) dynamic programming methods and (ii) heuristic methods. One of the classic dynamic methods for sequence alignment is the Smith-Waterman algorithm (Smith and Waterman, 1981). Like the Needleman-Wunsch algorithm (Needleman and Wunsch, 1970), Smith-Waterman creates a scoring matrix *M(m, n)*, where *m* denotes the length of the query sequence and *n* denotes the length of the reference sequence. First, the algorithm goes through all cells of the matrix and scores each cell based on the pre-set mismatch and gap penalty scores. In the next step, the algorithm traces back the cell and finds the optimal path with the highest scores. Based on the traversed path, the detailed alignment and the mapping identity (i.e. similarity) can be presented. The Smith-Waterman algorithm has been widely implemented in sequence alignment tools, such as JAligner (Moustafa, 2005) and the FASTA package (Lipman and Pearson, 1985).

The Burrows-Wheeler Transform (BWT) algorithm is also widely used in dynamic programming for sequence alignment. BWT was initially developed for data compression techniques such as bzip2 (Seward, 1996). In the particular application of sequence alignment, BWT was widely implemented in a collection of alignment

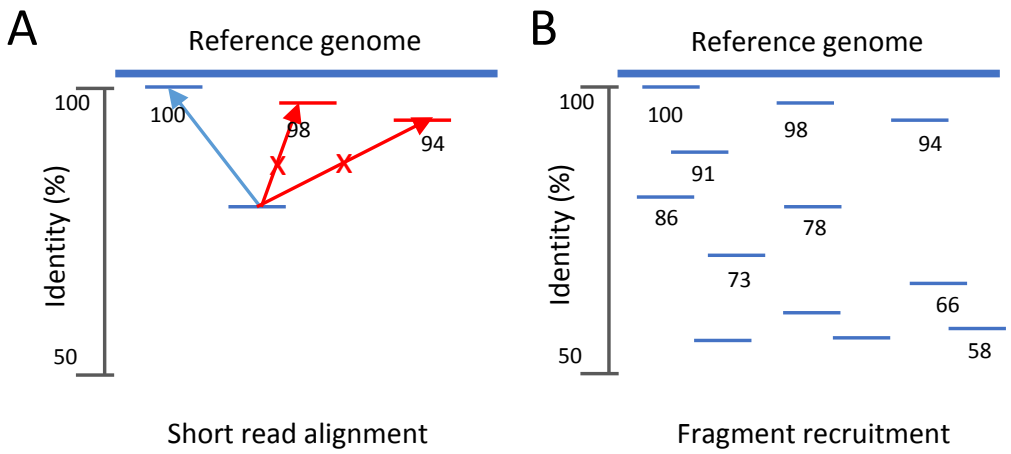

Fig. 2.8: Short read alignment and fragment recruitment: the major difference between the two approaches is the goals they want to achieve. **(A)** Short read alignment tries to find the best match of a given read. Whereas **(B)** fragment recruitment tries to report all possible matches that have higher identities than a given threshold. Blue dashes represent sequencing reads.

tools, such as SOPA2, Bowtie2, and the Burrows-Wheeler Alignment tool (BWA) (Li and Durbin, 2009). BWA first builds an index of the reference genome for the later alignment process. To build the index, it first constructs circulated strings (produce of a consecutive head-to-tail nucleotide shift of the original string (Fig. 2.9)) from the reference sequence. Then, all circulated strings are sorted lexicographically. The position of the first symbol in the sorted strings is used to build a suffix array, while the last symbol of the circulated strings is concatenated to build the BWT string. Once the index is built, BWA runs a backward search to align short reads back to the reference genome.

As for fragment recruitment, most of the methods are implemented with heuristic algorithms. Since fragment recruitment produces alignments with much lower mapping identities, the mapping process must be tolerant for more mismatches and gaps between the query sequence and the reference sequence. Thus, more candidate fragments are able to pass the pre-filtering stage and a considerable amount of computing run time is consumed for mapping these candidate fragments to the reference genome. Basic Local Alignment Search Tool (BLAST) (Altschul et al., 1990b) is one of the most widely used bioinformatics programs for sequence alignment. It can also be used for fragment recruitment. Like most alignment tools, a BLAST application consists of two phases: building the reference index and querying short reads to the index. BLAST builds the reference index by using a hash table data structure to store the locations of each k-mer on the genome (the seeds). Once the index is built, BLAST maps short reads to the reference index. The BLAST mapping process has two steps: seeding and extension. The seeding step searches perfect k-mer matches (as seeds) in the pre-built reference index. Once the seeds are found,

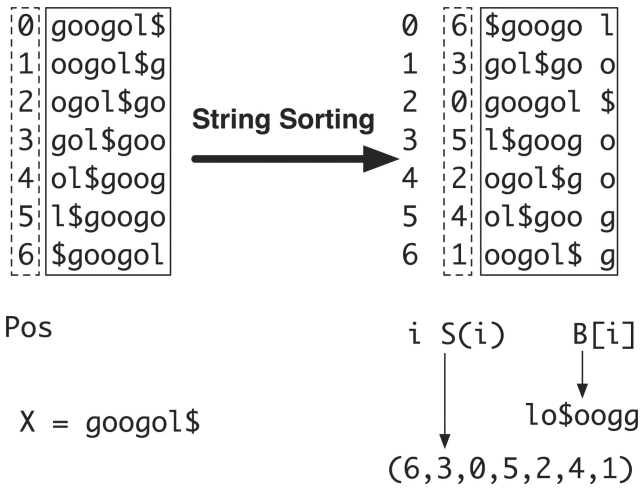

Fig. 2.9: BWT suffix array construction: the circulated strings are created by a head-to-tail shift of one nucleotide, where the $ sign serves as a marker to the end of the sequence. All circulated strings are then lexicographically sorted and the last symbols of the strings compose the BWT string (*lo$oogg* in the figure). The figure is modified from (Li and Durbin, 2009).

extensions are then carried out on both sides of the seeds. The extensions compare and score the similarity between the query sequence and the reference genome sequence using the BLOSUM62 scoring matrix.

A more recent fragment recruitment tool, called Fr-hit (Niu et al., 2011), introduces a q-Gram filter method (Rasmussen et al., 2006) to improve the run time performance for fragment recruitment. Compared to the 'seed and extend' approach used in BLAST, Fr-hit uses a longer k-mer (11nt) to plant seeds in the reference genome. Then, it creates candidate sequence blocks around the seeds on the reference genome. To remove disqualified candidate blocks that are unlikely to fulfill the minimal similarities required by the aligner (too many mismatches and gaps), Fr-hit implemented the q-gram filter method. Q-gram filter uses small continuous k-mers (4nt in Fr-hit) from a short read as probes to target the candidate blocks. A successful k-mer probing represents an exact K nucleotides match. Whereas one mismatch will remove at least K continuous k-mer matches. Thus, based on the number of matched k-mers, the q-gram filter rejects candidate blocks with more mismatches than the pre-set threshold.

2.2.3 Distributed implementations

Distributing sequence alignment tasks on a SGE cluster can be easily implemented, as each alignment can be an independent task running on a chunk of input sequencing data. Most research laboratories create their own in-house scripts to manually split input Fastq files into small chunks and submit to the SGE in batches. Each batch job carries out a standalone alignment on a chunk of the input file and the outputs of

all batch jobs will be concatenated as the final result. However, such an approach has the following limitations: (i) all data must go through the network of a shared volume on the cluster, introducing a bottleneck on the network, (ii) the in-house scripts usually do not have a built-in fault tolerant mechanism, (iii) the approach is not portable to a distributed cloud system.

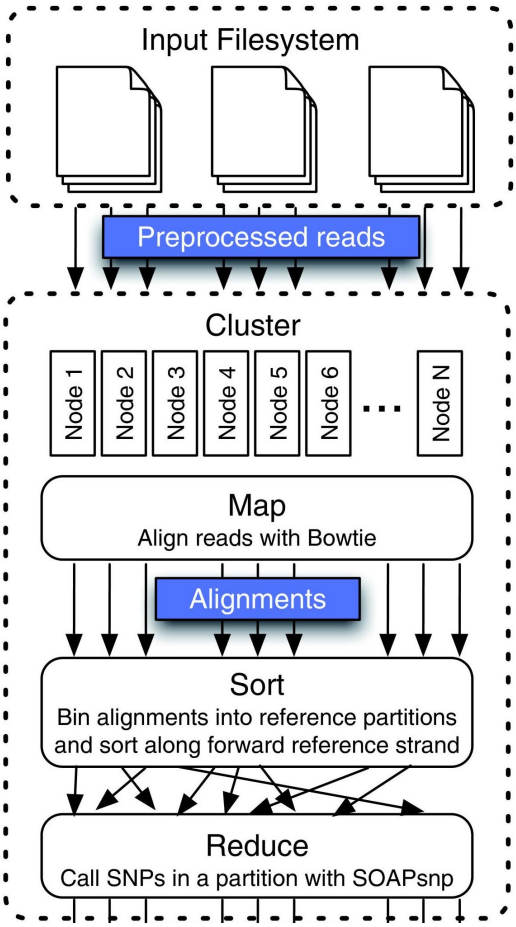

Fig. 2.10: Distributed sequence alignment in Crossbow: Preprocessed reads are split and distributed to different computing nodes. Each node carries out an independent Bowtie alignment on the split block of the sequencing reads. The alignments from Bowtie are binned and sorted for SNP calling. The figure is modified from (Langmead et al., 2009)

These limitations can be addressed by using distributed frameworks like Apache Hadoop and Spark. Cloudburst (Schatz, 2009) is a Hadoop based sequence mapping tool programmed in the MapReduce model. The common seed and extend alignment pipeline is split and implemented in 'map' and 'reduce' steps, where the 'map' step searches k-mer matches (as seeds) and the 'reduce' step extends the seeds to apply dynamic alignments. The limitation of such method is that the 'reduce' step introduces large data shuffling across cluster nodes that impacts its performance. Crossbow (Langmead et al., 2009), Halvade (Decap et al., 2015) and Myrna (Langmead et al., 2010), on the other hand, directly use Hadoop to invoke existing sequence aligner

(Bowtie (Langmead, 2010)) and SNP caller (SOAPsnp (Li et al., 2008) or GATK (McKenna et al., 2010)) for sequence mapping and genotyping on large datasets. The three tools have successfully reduced the run times for mapping, genotyping and gene expression quantification. Yet, their data preprocessing step introduces a heavy overhead and their options for handling distributed data are limited.

The Apache Spark framework has more built-in functions for RDD to facilitate methods implementation and data handling via its API. Nevertheless, existing Spark based bioinformatics tools have their own limitations. For instance, SparkBWA (Abuin et al., 2016) adopts the same idea of Crossbow by using Spark to invoke the BWA (Li and Durbin, 2009) aligner. However, it does not provide data preprocessing functions for large amounts of compressed sequencing data. Thus, manually decompressing the sequencing data introduces a significant run time overhead. Instead of directly invoking external aligners, MetaSpark (Zhou et al., 2017) re-implemented a fragment recruitment algorithm (Rusch et al., 2007). It has the same 'seed and extend' pipeline as Cloudburst, but implemented its algorithm on top of Spark. Therefore, it also introduces large data shuffling in the reduce step that impacts its run time performance. Moreover, it requires a self-defined input format rather than the standard Fastq and Fasta format, which introduces an overhead to manually convert large Fastq files.

2.3 De novo assembly and its cloud implementations

2.3.1 Algorithms for short read *de novo* assembly

The principle for *de novo* assembly is to detect overlaps between sequenced short reads. There are three major categories of algorithms for short read *De novo* assembly: (i) Greedy, (ii) overlap-layout-consensus (OLC), and (iii) *de Bruijn* graph (Nagarajan and Pop, 2013). Greedy method uses the full length short read sequences and conducts a pairwise search against each other for overlaps. It extends the assembly by joining the reads with the best overlap (as in greedy). Thus, this method solely considers the local connections of the short reads and does not take into account the global relationship of all short reads. Most assemblers developed in the early stage of genomic studies uses the greedy method, such as the TIGR (Sutton et al., 1995) and PHRAP (Melissa and Richard, 2007) assemblers.

Overlap-layout-consensus (OLC) method also uses the full-length sequences of the reads to extend the assemblies. Different from the greedy method, OLC constructs a graph to represent the global relationship of all overlaps between the reads. In the graph, each node denotes a read and each edge represents an overlap between two reads (Fig. 2.11A and B). Algorithms based on OLC method can traverse the

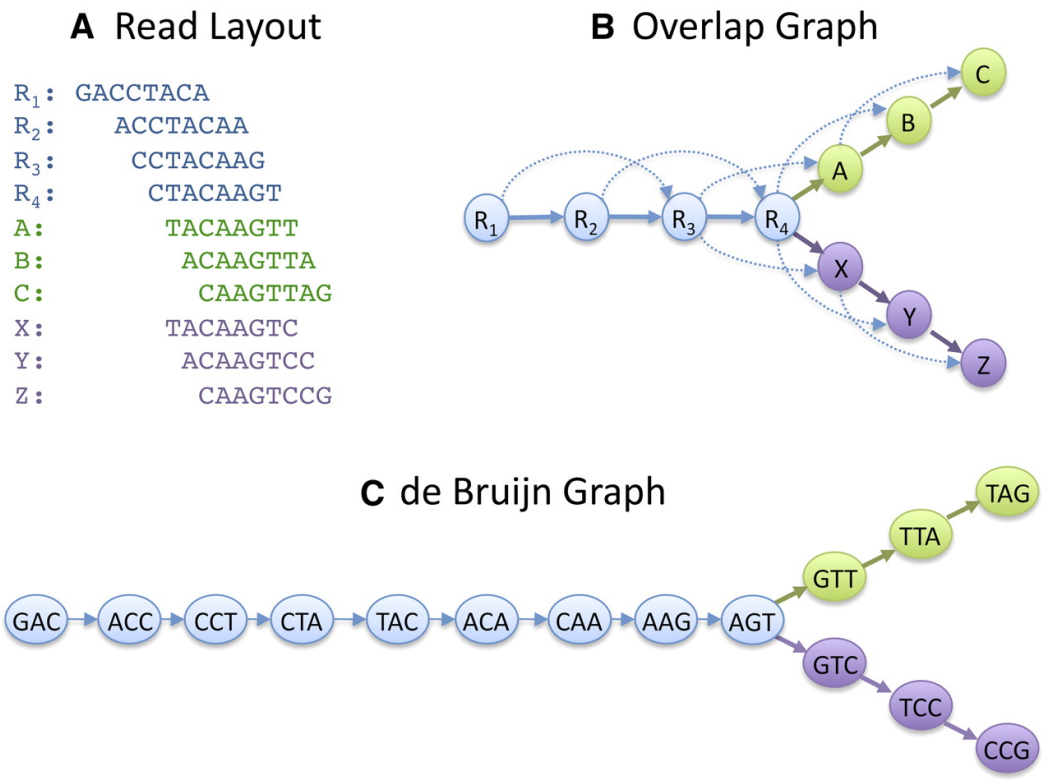

Fig. 2.11: *De novo* assembly methods: **(A and B)**, part of the overlap-layout-consensus (OLC) method. **(C)**, part of the *de Bruijn* Graph. The figure is from (Schatz et al., 2010)

graph and optimize the assembly by taking into account the global relationship between the reads. Celera (Myers et al., 2000) is one of the well-known assemblers developed using the OLC approach. However, the computational complexity of the OLC approach has limited its performance on the high throughput sequencing data. The more recent SGA assembler introduces a more efficient string indexing data structure to overcome the high complexity bottleneck of the OLC approach (Simpson and Durbin, 2012).

2.3.2 State-of-the-art *de Bruijn* graph

Instead of searching the overlaps of the full-length sequencing reads, the *de Bruijn* graph method extracts length *K* sub-sequences, known as k-mers (Fig. 2.12A), from the input reads and constructs a graph based on the overlaps of the k-mers. There are two types of *de Bruijn* graphs: Hamiltonian and Eulerian *de Bruijn* graphs (Fig. 2.11B and C). In a Hamiltonian *de Bruijn* graph, each node represents a k-mer and each edge denotes an overlap of two k-mers. The overlaps in the *de Bruijn* graph are normally *K-1* letters in length with only one nucleotide shifting between the adjacent (overlapped) k-mers. Thus, each step of the graph traversal extends one nucleotide on the assembly. In contrast to the Hamiltonian graph, each node of the Eulerian *de*

Bruijn graph is the overlap sequence (the (K-1)-mer) of two k-mers and each edge is the complete sequence of a k-mer.

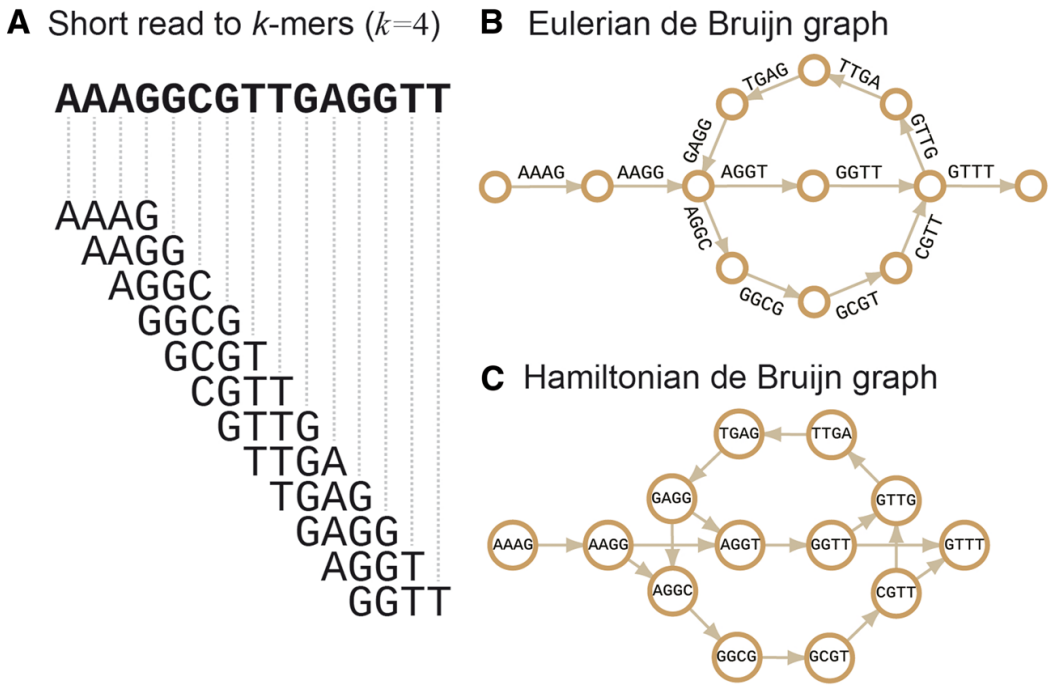

Fig. 2.12: The Hamiltonian and the Eulerian *de Bruijn* graphs: **(A)**, k-mers are extracted with 4 nucleotides in length. **(B)**, the Eulerian *de Bruijn* graph uses k-mers as the edges and (K-1)-mers as the nodes. **(c)**, the Hamiltonian *de Bruijn* graph uses (K-1)-mers as the edges and k-mers as the nodes. The figure is from (Sohn and Nam, 2018).

In a Hamiltonian *de Bruijn* graph, the genome is assembled by traversing Hamiltonian paths that go through all nodes in the graph, in which each node is visited only once. Such graph traversal is a typical nondeterministic polynomial time (NP)-complete problem. The computational complexity for searching the Hamiltonian paths is $O(m \times 2^n)$, where m denotes the number of all nodes in the graph, n represents the number of branching nodes (Thomason, 1989). Due to the sequencing errors in the sequencing process and repeat events on the genome sequence, the computational complexity of the Hamiltonian increases exponentially. Many Hamiltonian *de Bruijn* graph-based assemblers, such as ABySS (Simpson et al., 2009), Meraculous (Chapman et al., 2011), SOAPdenovo (Li et al., 2010), and Velvet (Zerbino and Birney, 2008), reduce the complexity of the graph by partially removing branch nodes. However, such reduction produces more short contigs in the result of the assembly.

Different from the Hamiltonian *de Bruijn* graph, the Eulerian *de Bruijn* graph method try to go through all edges in the graph (Fig. 2.12B), in which each edge is visited only once. In such case, the path can be found in polynomial time with an $O(n^2)$ computational complexity (Pevzner et al., 2001). Without simplifying the graph,

Eulerian-based *de novo* assemblers, such as EULER (Pevzner et al., 2001) and SPAdes (Bankevich et al., 2012), generally produce longer contigs.

2.3.3 Cloud based *de novo* assemblers

Implementing a *de novo* assembly algorithm on a distributed system can be more complicated than implementing a distributed alignment algorithm, as genome assembly cannot be carried out independently on each partition of the sequencing data. Take the *de Bruijn* graph as an example: to assemble the complete genome, all parts of the input data (all nodes in the graph) must be accessible by the processor to traverse the complete paths of the genome. In a distributed system, the complete nodes of the *de Bruijn* graph are separated and distributed to different computer instances of a cluster. In such case, traversing all nodes of the *de Bruijn* graph involves communications between different computer instances.

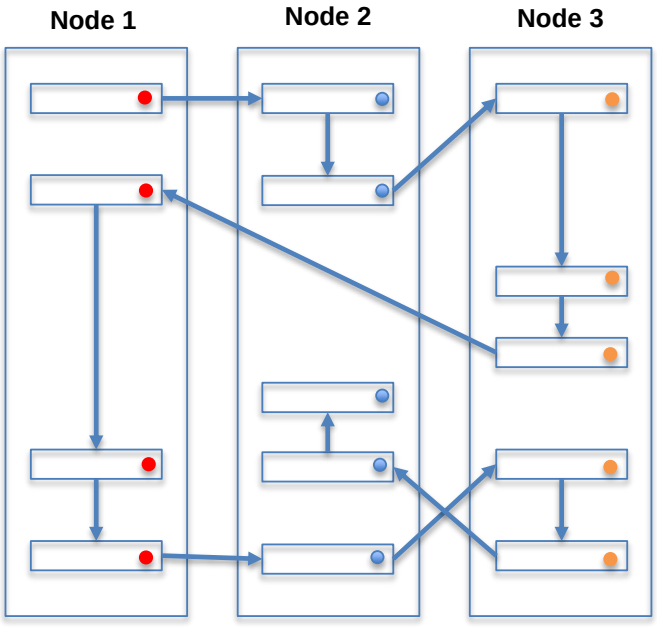

Fig. 2.13: The distributed *de Bruijn* graph of Velvet: Blue frames represent nodes of the *de Bruijn* graph. Figure modified from (Zerbino and Birney, 2008)

Most distributed assemblers still use the *de Bruijn* graph approach as their key algorithms (e.g. ABySS (Simpson et al., 2009), Ray (Boisvert et al., 2010), SWAP-Assembler (Meng et al., 2014), and Spaler (Abu-Doleh and Çatalyürek, 2015)). To organize data distribution and communication in a cluster of commodity computers, distributed frameworks are needed for implementing a distributed *de Bruijn* graph. The message passing interface (MPI) (Gropp et al., 1996) is the primary choice for most distributed assemblers, such as ABySS, Ray, and SWAP-Assembler. In the MPI implementation of ABySS assembler, all k-mers of the *de Bruijn* graph are distributed across all computer instances of the cluster. The physical location of each k-mer

(the index of the computer instance, on which the node is stored) is also computed, so that k-mers can locate its adjacent k-mers (overlapped k-mers) and pass the message to the computer instances accordingly. The weakness of such an approach is that it relies heavily on the performance of the network. On a standard Ethernet connected cluster, the constant messaging between different computer instances will significantly increase the latency of the network. To increase the scalability of such approach, a high-speed InfiniBand network is needed (Liu et al., 2011).

The more recent Spaler assembler implemented the *de Bruijn* graph on top of the Apache Spark framework. It employs the Spark graphic library, called GraphX, to distributed its *de Bruijn* graph. Spark-GraphX offers a series of functions for developers to handle the communications between worker nodes for traversing the distributed graph. The back ends of these functions are MapReduce based implementations to reduce the nodes of the *de Bruijn* graph and extends the assemblies (Gonzalez et al., 2014). Spaler has reported better performance in scalability comparing to other MPI based assemblers such as ABySS, SWAP-Assember and Ray. However, its executable file is, to the best of our knowledge, not available.

2.4 Conclusion

In this chapter, I presented related work in three different parts: (i) The Apache Hadoop and Spark framework, (ii) the current state-of-the-art methods and technologies for distributed sequence alignment, and (iii) distributed genome assembly. I have presented the cluster architectures of both Hadoop and Spark clusters. I also introduced the classic Hadoop MapReduce model and its weakness in iterative computations. The more recent Spark framework addresses the weakness with its in-memory computing function. Spark introduces a new data sharing primitive called RDD that provides built-in functions to facilitate distributed implementations. Thus, most of my distributed implementations are built on top of the Spark framework.

For distributed sequence alignment, existing tools either have limited options for preprocessing input data (e.g. Crossbow) or suffer from the overhead introduced by large amounts of messaging in the network (e.g. MetaSpark). Therefore, in the first section of my work (chapter 3), I will introduce my approaches for both short read alignment and fragment recruitment. Then, I will compare my tools to both Crossbow and MetaSpark.

For distributed *de novo* genome assembly, most existing tools use the state-of-the-art *de Bruijn* graph and implemented the distributed graph on MPI. However, the performances of the MPI based tools are banded by the network speed. In the second section of my work (chapter 4), I will introduce a new distributed data structure

and its implementation on a newly developed assembler called Reflexiv. I will also compare my tool to the existing assemblers including Ray and ABySS.

3 Sparkhit: Distributed sequence alignment

> *The future is already here. It's just not very evenly distributed.*
>
> — **William Ford Gibson**
> (Science fiction writer)

In this chapter, I mainly focus on addressing a computational intensive challenge (sequence alignment) in bioinformatics applications. I will present Sparkhit, an open source computational framework that is easy to use on a local cluster or on the cloud (Huang et al., 2018). Sparkhit is built on top of the Apache Spark platform, integrates a series of analytical tools and methods for various genomic applications:

1. I have natively implemented a metagenomic fragment recruitment tool and a short-read mapping tool (Sparkhit-recruiter and Sparkhit-mapper) on top of the Apache Spark platform. The short-read mapper implements the pigeonhole principle to report the best hit of a sequencing read. Whereas in the fragment recruitment tool, I implemented the q-gram algorithm to allow more mismatches during the alignment, so that extra mapping reads are provided for the downstream metagenomic analysis

2. For using external software on Sparkhit, I built a general tool wrapper (Sparkhit-piper) to invoke and parallelize existing executables, biocontainers (e.g. Docker containers (Merkel, 2014)) and scripts

3. For downstream data mining, I integrated and extended Spark's machine learning library

4. For data preprocessing, I developed a parallel decompression tool (Sparkhit-spadoop) that significantly increases the speed for NGS data decompression.

In the results section, I will present a series of performance benchmarks for Sparkhit. In general, the benchmarks demonstrated its high scalability on the AWS cloud. In comparison, Sparkhit ran 18 to 32 times faster than Crossbow on data preprocess-

ing. For fragment recruitment, Sparkhit-recruiter ran 92 to 157 time faster than MetaSpark.

3.1 The pipeline for sequence alignment

In metagenomic studies, fragment recruitment is a key step to understand the genome structure, evolution, phylogenetic diversity, and gene function of biological samples (Rusch et al., 2007). As a special case of read mapping, fragment recruitment supports more mismatches during the alignment. Thus, the computational complexity for fragment recruitment can be higher than that of the standard short read mapping.

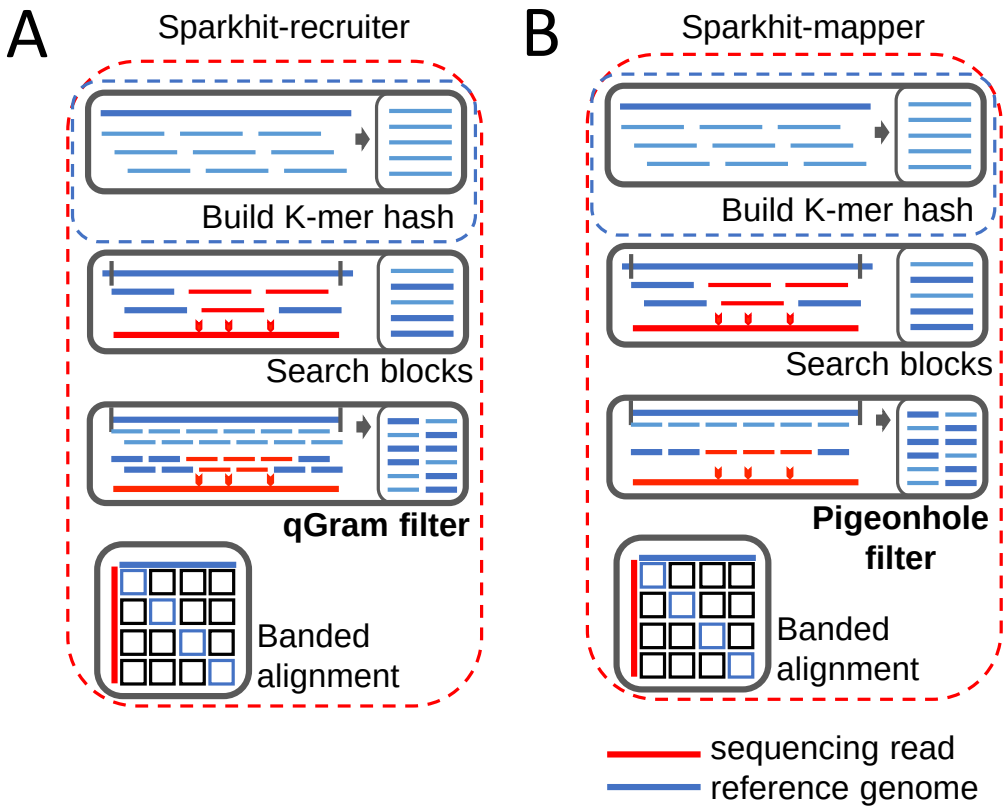

Fig. 3.1: The pipelines of Sparkhit-recruiter and Sparkhit-mapper: **(A)**The pipeline of Sparkhit-recruiter for fragment recruitment. Blue dashes represent k-mers extracted from the reference genome, whereas red dashes represent k-mers extracted from sequencing reads. **(B)** The pipeline of Sparkhit-mapper for short-read mapping. The third step of Sparkhit-mapper uses the pigeonhole filter instead of the q-gram filter.

I implemented a fast and sensitive fragment recruitment tool, called Sparkhit-recruiter. Sparkhit-recruiter extends the Fr-hit (Niu et al., 2011) pipeline and is implemented natively on top of the Apache Spark. The pipeline consists of four steps (Fig. 3.1A):

1. building reference index
2. searching candidate blocks
3. block filtering
4. banded alignment.

When building the reference index, Sparkhit-recruiter uses a k-mer hash table to store each K-mer's location on the reference genome (Fig. 3.1A in blue color). Once the index is built, the program extracts the k-mers from sequencing reads (Fig. 3.1A in red color) and searches against the reference hash table for exact matches. The matched k-mers will be placed on the genome as seeds to extend candidate blocks. The block filtering step incorporates a q-gram threshold to remove badly matched blocks, therefore improving run time performance. After filtering, banded alignment is applied to give a final mapping result.

I also implemented a short-read aligner, called Sparkhit-mapper (Fig. 3.1B). It adopts the same pipeline of Sparkhit-recruiter, but uses a more strict pigeonhole principle for the filtering step. Compared to the q-gram filter implementation in Sparkhit-recruiter, the pigeonhole principle allows fewer mismatches on the sequence so that less identical blocks are filtered and high similarity candidate blocks are preserved (see section "3.1.3 Pigeonhole principle"). In this case, less candidate blocks are sent to the next step for banded alignment, making it runs faster than Sparkhit-recruiter.

Here, I explain each step of the pipelines in detail:

3.1.1 Building reference index

First, the reference index is built on the driver node before the actual mapping (querying) starts (Fig. 3.2). The driver program first reads the input reference genome and extracts k-mers with a pre-selected length. The k-mers are, then, used to construct a hash table that records each k-mer's location on the reference genome. I encoded the hash table in a one dimensional array with a customized hash function. Four different nucleotides are encoded in two binary bits (A to 00, T to 01, C to 10, G to 11). As a result, a k-mer smaller than 16 nucleotides (encoded in 2^{32} bits) can be encoded as an arbitrary index number of an array. This hash function gives each k-mer an unique hash code. Thus, collision events are eliminated from the hash table, as each key is placed in a unique block within the array. Therefore, the run time performances of building and searching this hash table are increased.

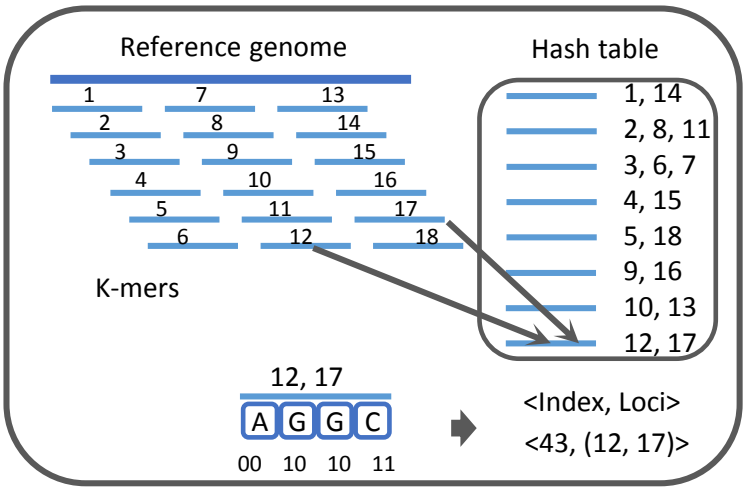

Fig. 3.2: Reference index construction: k-mers are extracted from the reference genome and their locations on the genome are stored in a hash table. Each k-mer is encoded into an integer, which servers as the index number (the Hash code) of the hash table.

3.1.2 Candidate block seraching and q-Gram filters

The querying process starts by finding the exact k-mer matches between queried sequencing reads and the reference genome (seeding). Once a match is found, the program extends a putative mapping block (a block that is longer than the read length plus the maximum mismatches) on the reference genome around the seed as a candidate (Fig. 3.3A). Since the seed can be a random match on the genome that results in a block with extensive mismatches, a q-gram filter is applied to reject such candidate blocks. The q-gram filter tries to find the worst scenario to plant a number of mapped short k-mers (q-grams) in a sequence to allow a certain number of mismatches. If the worst scenario cannot be fulfilled (less q-grams are found), more mismatches are present in the block than the maximum number of mismatches allowed. In this case, the block is rejected. For a candidate block with a length of n nucleotides, let e denote the number of mismatches, q the length of the q-gram. The q-gram Lemma can be expressed as:

$$T(n, q, e) \geq (n + 1) - q(e + 1) \tag{3.1}$$

Where $T(n, q, e)$ stands for the minimum number of q-gram matches expected in the block. Here, the worst scenario is that each mismatch will consume q number of q-gram matches (q-grams are extracted with 1 nucleotide offset). Since the maximum number of q-grams in a sequence of length n is $(n + 1 - q)$, the minimum q-gram matches found in the block should be $(n + 1 - q) - q \times e = (n + 1) - q\ (e + 1)$.

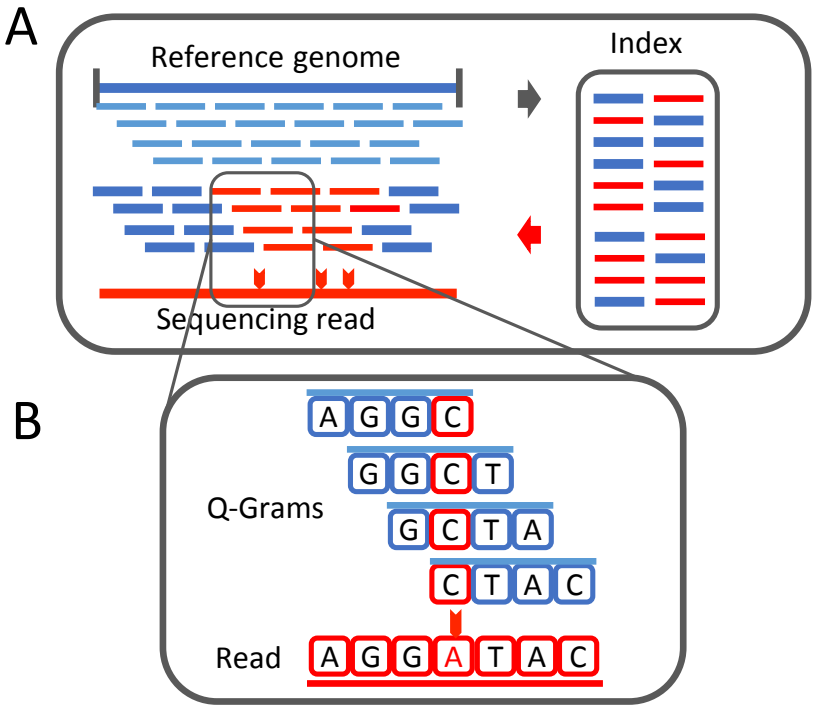

Fig. 3.3: An example of the q-gram filter: **(A)** three mismatches between the sequencing read and the candidate block knock out 10 q-grams (red short dashes). **(B)** One mismatch knock out maximally q number of q-grams.

3.1.3 Pigeonhole principle

When the mapping identity has been set to more than 94%, the pigeonhole principle is used to introduce faster and more robust filtering. It also tries to find the worst scenario to plant a number of k-mers that allows a certain number of mismatches. However, in the pigeonhole principle solution, k-mers are extracted without overlaps from a sequencing read (Fig. 3.4). It uses consecutive short k-mers as probes to target the candidate blocks. Each successful k-mer probing represents an exact k nucleotides match, whereas each unsuccessful k-mer probing represents at least one mismatch in this k-mer region. In sequence alignment, a mismatch is considered as a pigeon that occupies a container (in our case, a k-mer) on the sequence. A e number of mismatches on the sequence will cost the same number of k-mers in the worst scenario. For a sequencing read with a length of n nucleotides, the total number of k-mers are $\lfloor n/k \rfloor$, where k is the size of the k-mer. Thus, the minimum number of k-mer matches on the read should not be less than $\lfloor n/k \rfloor - e$. Otherwise, the sequencing read is rejected.

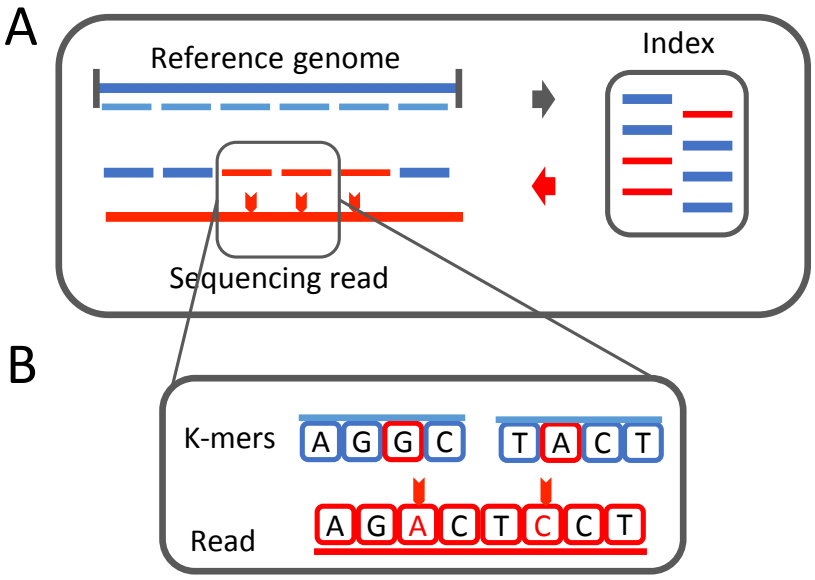

Fig. 3.4: An example of the pigeonhole principle: **(A)** when using pigeonhole principle for the filtering process, short k-mers are extracted consecutively without overlaps. Thus, each mismatch knocks out maximally one short k-mer. **(B)** An example of two mismatches knock out two k-mers from the candidate block

3.1.4 Banded alignment

I implemented the dynamic alignment method which constructs a $(m+1) \times (n+1)$ matrix, where m is the length of queried sequence and n is the length of filtered reference block, to traverse the optimal alignment between the two sequences. However, when scoring the matrix, a smaller band can be applied to avoid unnecessary computing on the outer bound of the matrix (Pearson and Lipman, 1988). After filtering the candidate blocks, a banded alignment is carried out with a pre-defined bandwidth (Fig. 3.5). The band center is set around the region with highest q-gram matches. After the matrix traversal, the mapping score and the identity are both calculated based on the point accepted mutation (PAM) 50 scoring matrix.

3.2 Distributed implementation

To implement the pipelines in a distributed fashion, I split the mapping processes into two parts: building the reference index and querying sequencing reads (Fig. 3.6A-B, blue and red dashed boxes). When starting a Sparkhit-recruiter job, the reference index is, firstly, built on the driver node (usually the master node), where the main Spark program runs. Once the reference index is built, the driver program executes a 'broadcast' command to ship one copy of the index to each worker node shared by all local querying tasks. On the worker nodes, sequencing data chunks are loaded from

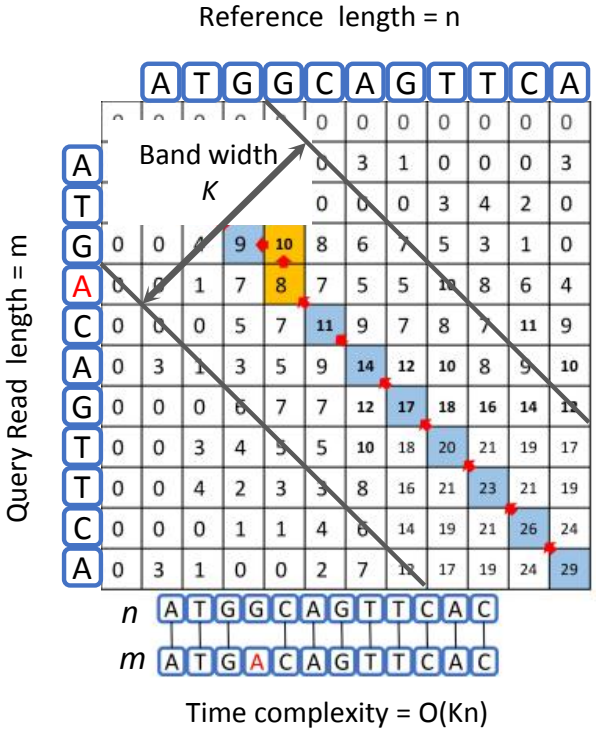

Fig. 3.5: Banded alignment: A K length band is applied on a $m \times n$ matrix for the pairwise alignment, where n is the length of the reference genome and m is the length of the sequencing read. Since the computation is limited in the banded area, the computational time complexity is $O(Kn)$.

HDFS to a Spark RDD. Each partition of the RDD is, then, independently queried to the broadcasted reference index as a 'map' step of the MapReduce pipeline. In the end, a 'reduce' step summarizes the mapping result (Fig. 3.6A).

3.2.1 Reference index serialization and broadcasting

To send a copy of the reference index to each worker node, a broadcast function is used. This broadcasting process has two steps: object serialization and network broadcasting. The driver program serializes an instance of the reference data structure (the reference index) to a binary file. Then, the driver program applies a 'broadcast' function to transmit the binary data to each worker node (Fig. 3.6A). For example, the Jave code can be expressed as:

```
  final Broadcast<KmerLoc[]>broadIndex =
sc.broadcast(ref.index);
```

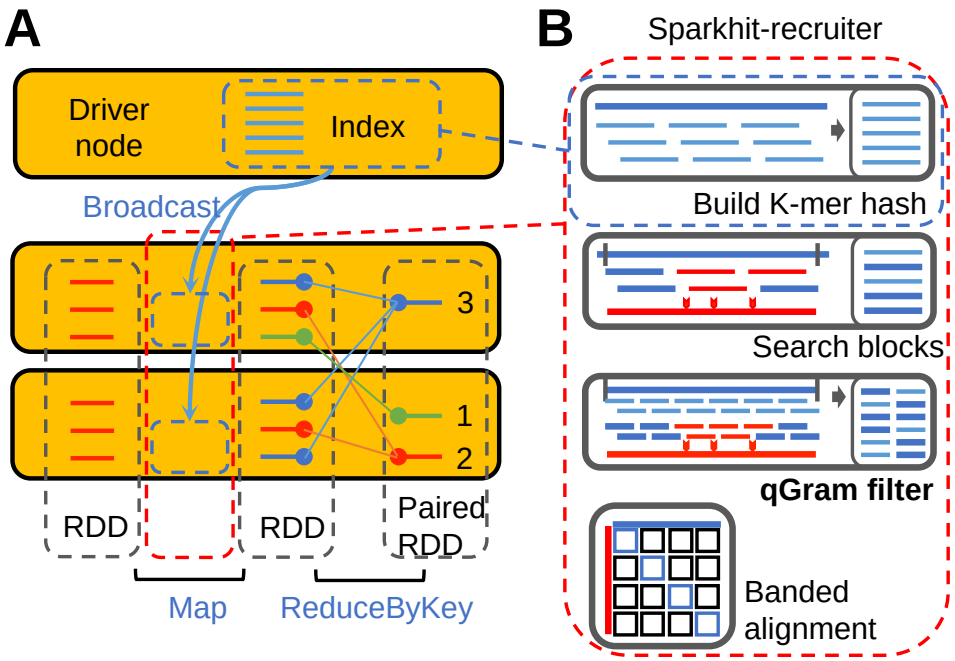

Fig. 3.6: Distributed implementation of the fragment recruitment pipeline: **(A)** Distributed implementation of Sparkhit-recruiter. The reference index, illustrated in blue dashed box, is built on a driver node and broadcasted to each worker node. Sequencing reads, illustrated in Red dashes, are loaded into an RDD and queried to the broadcasted reference index in parallel as a 'map' step. A 'reduce' step is followed to summarize the mapping result. **(B)** the reference index, illustrated in blue dashed box, is built on a driver node and broadcasted to each worker node. Sequencing reads, illustrated in bold red dash, will be searched against the reference hash table for exact matches. A smaller k-mer is used to apply the q-gram filter.

Where `ref.index` is a pre-built reference index from an input reference genome. `sc` stands for the Spark context, which is a driver program running on the master node. It applies the `sc.broadcast` function to broadcast the `ref.index` to each worker node.

On worker nodes, each executor applies a `get` function to de-serialize the input data stream into a Java object (in this case the de-serialized reference index) and stores only one copy of the index in memory for all mapping processes on this worker node.

I implemented the Kryo (Grotzke, 2017) serialization framework for faster reference index serializations. The reference data structure mainly consists of a k-mer hash table encoded in a one dimensional array. Our serializer first registers the Java class of this one dimensional array with an integer ID. The Java code can be expressed as:

```
kryo.register(KmerLoc[].class, 1);
```

Where `KmerLoc[].class` is the Java class for the reference data structure. Once the reference index is built, the Java object (an instance of the Java Class) is, then, serialized and the binary bits are written to an output file via a pre-defined output stream writer. As the Java Class is registered, it will write the registered ID first, then the binary bits of the Java object.

3.2.2 Data representation in the Spark RDD

The Spark RDD stores data chunks in line-based text format, where identifying entries requires finding line boundaries denoted by newline characters. After decompression, most NGS data is stored in line-based text files, e.g., fastq, SAM and VCF files. For SAM and VCF files, each line is an independent unit that contains its corresponding information (mapping records or genotypes). Thus, when loading these files, each line is read and stored as an element of an RDD. However, a fastq file stores its basic information in a four-line unit, where each line is an essential part of a sequencing read. When loading a fastq file into Spark RDD, a filter step is applied to check each four-line unit of the fastq file before the next step.

Both loading and saving run in parallel on each partition of the RDD (the sequencing data) with a default size of 256 MB per partition. Sparkhit can also directly load and save data from and to both HDFS and Amazon S3 by using the HDFS and the S3 URL scheme.

3.2.3 Concurrent in memory searching

Once the reference index is broadcasted to each worker node, each partition of the RDD runs an alignment task independently on the worker node. The alignment tasks go through each sequencing read in the partitions and search against the reference index using the pipeline of Sparkhit-recruiter or Sparkhit-mapper. The alignment results are sent to a new RDD with the same number of partitions (Fig. 3.6). In the case of a task failure during the process, the failed partition (a portion of the sequencing reads) will be reloaded into the RDD and are rescheduled for alignment in a new task.

3.2.4 Memory tuning for Spark native implementation

The random access memories (RAMs) of Sparkhit-recruiter and Sparkhit-mapper are mainly consumed by a copy of the reference index and loaded partitions of

an RDD. For the reference index, there are a hash table, which stores the locus of each k-mer, and a list of binary strings that represents the compressed reference genome sequences. To measure the memory consumption of the reference index, I can serialize the objects of the reference and output the binary files to the hard disk using Sparkhit's local recruiter, a Java based tool included in Sparkhit. The overall size of the output files is the size of memory used by the reference index on a worker node. As for the RDD, when not using the 'cache' function, the memory consumption on each worker node is the number of CPUs times the batch size of each partition (256 MB by default). When using the 'cache' function, all the input sequencing data will be loaded into memory. However, the limit for both reference index and the input sequencing data is set to 75% of the maximum RAM by default (see section 3.6.7 cluster configurations).

3.3 Using external tools and Docker containers

To be flexible for different kinds of analyses, I built a general tool wrapper called Sparkhit-piper. It extends Spark's 'pipe' function to invoke existing tools for analyses like sequence mapping, taxonomic profiling, gene expression quantification and genotyping (Fig. 3.7A). I use Spark RDD to split and distribute NGS data across cluster nodes. Then, distributed datasets are sent to the invoked tool via a standard input (stdin) stream. The tool processes input data in a batch and sends back the result to another RDD via a standard output (stdout) stream. In this case, the Spark RDD splits and distributes NGS data, while external tools carry out their corresponding computations. Sparkhit-piper is intuitive and flexible for users to parallelize their own scripts or tools directly without modifying their codes, as illustrated in Fig. 3.7.

The implementation is based on Spark RDD's 'pipe' function, where the RDD is able to send its data out of the JVM for processing in the operating system, like a Linux pipe operator ('|'). The Java code can be expressed as:

```
  JavaRDD<String> MapRDD = FastqRDD.pipe
(param.tool + param.toolOptions);
```

Where "`FastqRDD`" is the input RDD that stores sequence data and "`MapRDD`" is the output RDD that stores mapping result. The "`param.tool`" represents the full path of the tool executable while "`param.toolOptions`" represents the corresponding tool parameters. Together, they assemble an external command that runs as an independent process on each partition of the input RDD.

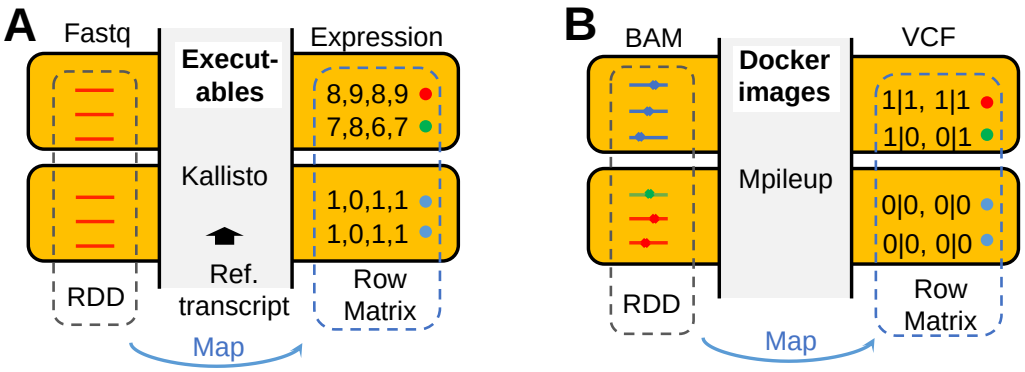

Fig. 3.7: Invoking external tools in Sparkhit: **(A)** Yellow boxes represent Spark worker nodes virtualized by the Spark JVMs. Spark RDD sends sequencing data (in fastq format) from Spark JVMs to the external executables via an STDIN channel. External executables process the input sequencing data independently and send the result back to Spark RDD via an STDOUT channel. **(B)** The same approach can also apply to external Docker containers.

This approach can also be applied to bio-containers (e.g. bioinformatics Docker images (Merkel, 2014)). By replacing an executable to a Docker container, Sparkhit-piper can easily assemble a Docker run command that runs an independent Docker task on the Spark cluster (Fig. 3.7B).

3.4 Integrating Spark's machine learning library (MLlib)

As a supplementary functional model for downstream genomic data mining, I extended Spark's machine learning library (MLlib) and integrated a variety of algorithms: (i) clustering, (ii) regression, (iii) chi-square test, (iv) correlation test, and (v) dimensional reduction. These algorithms are implemented with more RDD functions and iterative processes. For example, to implement the k-means clustering in a distributed way, the re-centering and re-clustering steps are split and implemented in a Spark extended MapReduce paradigm. The 'map' step assigns each data point to the closest centroids to form clusters. Whereas the 'reduce' step computes the new centroid for each cluster. Since data points are distributed across cluster nodes, the 'reduce' step applies Spark's 'reduceByKey' function to shuffle data points by clustering and calculating the centroids. The 'map' and 'reduce' steps iterate until a convergence status is reached. Detailed methods can be found in the *Appendix methods* section.

3.5 Parallel data preprocessing

For data preprocessing, I developed a parallel decompression tool called Sparkhit-spadoop. Since Bzip2 files are compressed in blocks (900 KB per block by default), a large Bzip2 compressed file with sequencing data can be decompressed in parallel. In particular, each block is an independent component that is delimited by a 48-bit pattern, which makes it possible to find the block boundaries. When block boundaries are found, parallel decompression can be applied to each block and processed by multiple CPUs, so that more computing cores are utilized (Fig. 3.8). Then, it distributes all processes in a Hadoop MapReduce job that creates a 'mapper' for each chunk of the input HDFS data. Each 'mapper' loads a data chunk and commences a decompression process on the Bzip2 blocks in the chunk.

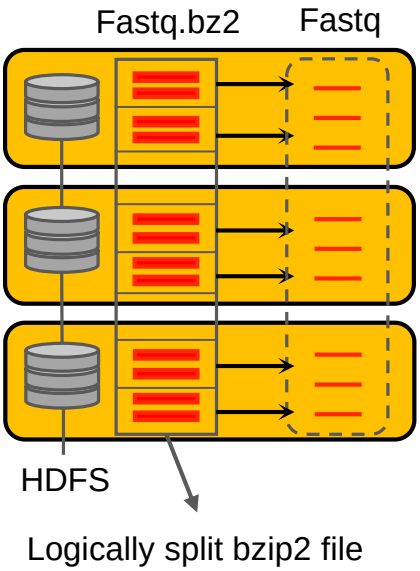

Fig. 3.8: Distributed decompression: A Bzip2 compressed fastq file is logically split on HDFS (replicas are physically distributed to different computer nodes) and each chunk of the file is decompressed by a 'mapper' process that runs a Bzip2 decompression program.

To implement the parallel decompression tool (Sparkhit-spadoop) on top of the Hadoop MapReduce framework, I set the corresponding input file format for the Hadoop 'mapper' by using the following Java code snippet:

```
  job.setInputFormatClass(Bzip2TextInput
Format.class)
```

Where `job` is a Hadoop MapReduce job created by the program. `Bzip2TextInputFormat.class` is the class type of the input Bzip2 file format. The

same method can also be applied to other "splittable" compressed file formats, such as the Lempel-Ziv-Oberhumer (LZO) format.

When input files are in the binary alignment/map (BAM) format, such as the mapping results of the 3000 Rice Genomes Project, Hadoop-BAM (Niemenmaa et al., 2012) was used to access and decompress BAM files stored on the HDFS. Hadoop-BAM is built on top of the Hadoop platform. It uses the Hadoop record reader to access HDFS data chunks. Since BAM files are compressed in the blocked GNU zip format (BGZF), the program starts decompression by locating the boundaries of compressed blocks using the BGZF magic code. Then, it searches the start of a BAM record within the blocks and decompresses the BAM file.

3.6 Results and Discussion

In this section, I present a series of performance benchmarks for Sparkhit and discuss its performances compared to other tools.

3.6.1 Run time comparison between different mappers

For sequence mapping, I compared run time performances between Sparkhit-recruiter, Sparkhit invoked fr-hit, Sparkhit-mapper, Sparkhit invoked BWA, Sparkhit invoked Bowtie2, and Crossbow, where Sparkhit-recruiter and fr-hit have a particular focus on fragment recruitment. The comparisons were carried out across different sizes of input sequence data (1.3TB and 545GB data of tongue dorsum samples from the HMP mentioned in the materials section as Data-1 and Data-2, see table. 3.3), different sizes of reference genomes (36 MB, 72 MB and 142 MB correspond to Ref-1, Ref-2 and Ref-3 mentioned in the NGS data sets section below) and different number of worker nodes (30 and 50 `c3.8xlarge` worker nodes) (Fig. 3.9A-D). I used mostly default parameters for each tool with slight modifications depending on the purpose of the evaluation. For Crossbow, I have set the corresponding Bowtie parameter to report all valid alignments with the '`-a`' option.

Our toolkit ran faster than Crossbow across different numbers of worker nodes (30 and 50), different sizes of input data (1.3 TB and 545 GB) and different sizes of reference genomes (36 MB, 72 MB and 142 MB). Although Sparkhit-recruiter was slower than other Sparkhit based mappers, it recruited many more reads than standard short-read mappers such as Bowtie (Fig. 3.10). I have used Data-1 (1.3 TB fastq files) for comparing the recruited numbers of reads between Sparkhit and Crossbow. Crossbow recruited 16,288,351 reads to a 72 MB reference genome, whereas Sparkhit-recruiter recruited 496,569,401 sequencing reads.

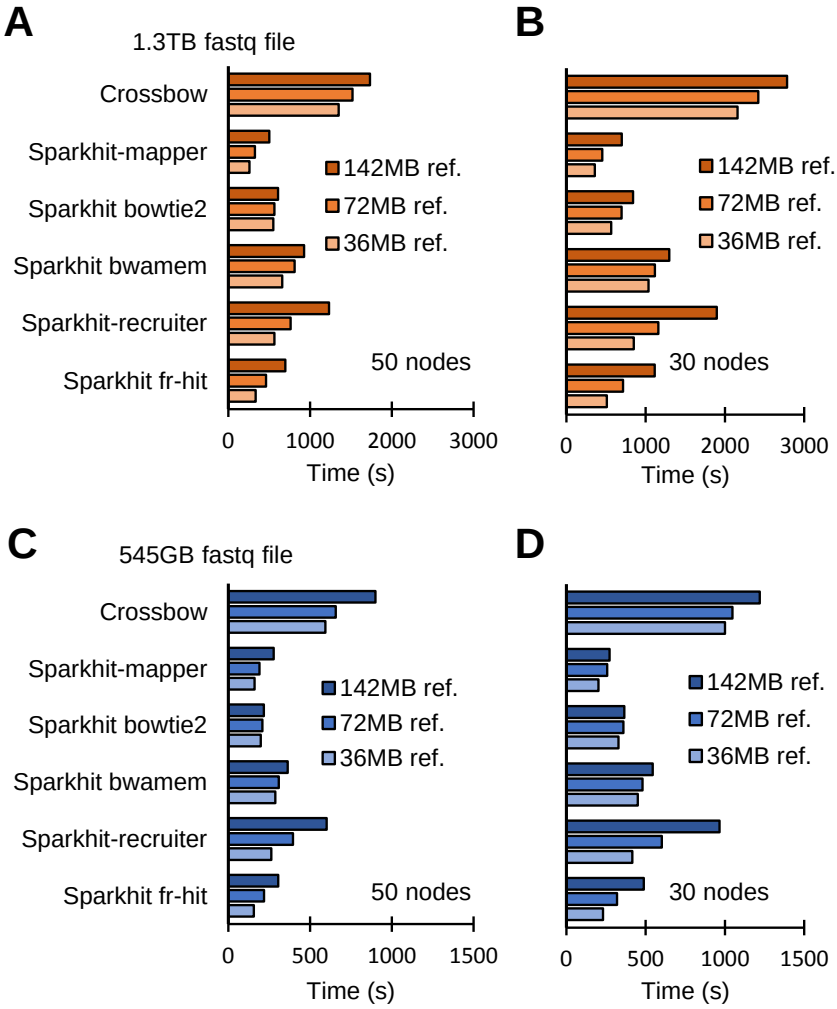

Fig. 3.9: Run time comparisons between different aligners: The comparisons were carried out across different sizes of input fastq files, different sizes of reference genomes and different numbers of worker nodes.

3.6.2 Scaling performance of Sparkhit-recruiter

To present the scalability of Sparkhit-recruiter along the increasing size of the input data (Fig. 3.11A), I used the larger dataset (Data-1) and ran Sparkhit-recruiter on 30 `c3.8xlarge` worker nodes with 100 GB increment. Whereas, for the scaling performance along different number of worker nodes (Fig. 3.11B), I used the smaller dataset (Data-2) and ran Sparkhit-recruiter on 10 to 100 worker nodes with 10 nodes increment.

Sparkhit-recruiter scaled linearly with the increasing amount of input data on a 30 worker nodes Spark cluster (Fig. 3.11A). When scaled-out to more compute nodes, Sparkhit experienced slight slowdown after I increased the number of worker nodes to 60 (Fig. 3.11B). The slowdown is introduced by the overhead of building the reference index. However, since metagenomic fragment recruitment applications

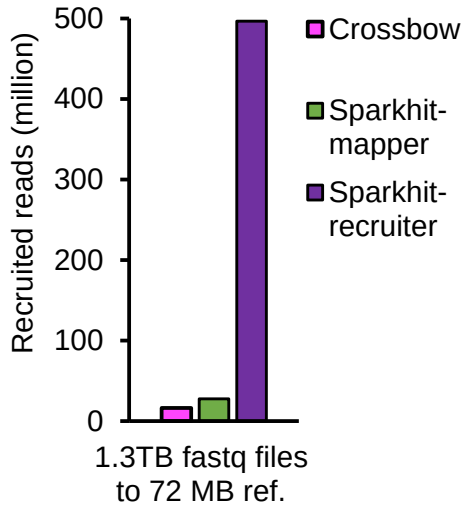

Fig. 3.10: Numbers of recruited reads: comparison was carried out between Crossbow and Sparkhit-recruiter when mapping 1.3 TB fastq files to a 72 MB reference genome.

actively change reference genomes between different studies, the index building overhead is quite low. Moreover, Sparkhit runs much faster on the index building process compared to other Burrows-Wheeler transform (BWT) based methods (see Discussion).

3.6.3 Accuracy and sensitivity of natively implemented tools

I have compared the sensitivity and accuracy between Sparkhit-recruiter, Sparkhit-mapper, Fr-hit, SOAP, BWA and Bowtie2. The evaluation was firstly carried out based on the 6 simulated datasets (see table.3.3 and URL: doi:10.4119/unibi/2914921). For BWA, bowtie2 and soap, I used their default parameters. Whereas for Sparkhit-recruiter and Fr-hit, I set the corresponding parameters to report the best hit (fragment recruitment tools usually report all valid hits). Evaluations were also carried out based on the public datasets of the Genome in a Bottle Consortium (GIAB) (Zook et al., 2014). I have used the 150nt pair-end sequencing data of the Chinese trio mother datasets (NA24695) and mapped all sequencing data to the GRCH37 human reference genome. Two reference benchmarks were used: (i) the mapping result of the GIAB project (the BAM file generated by NovoAlign: http://www.novocraft.com/), (ii) the consensus overlapping result of NovoAlign, BWA, Bowtie2 and SOAP. All tools were set to use the same parameters.

In general, Sparkhit-recruiter has slightly higher accuracy than Fr-hit, Bowtie2 and SOAP, while having slightly lower accuracy than BWA (Fig. 3.12). For sensitivity, Sparkhit-recruiter, Fr-hit, Bowtie2 and BWA are higher than SOAP on 100nt simulated reads. For 150nt simulated reads, all mappers have similar sensitivities.

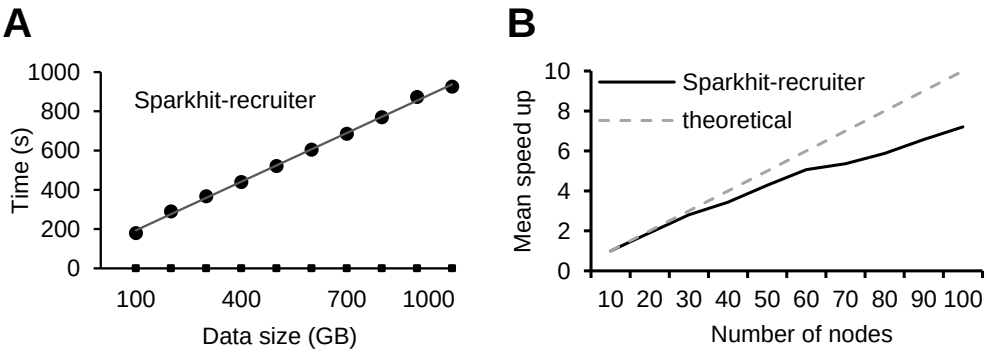

Fig. 3.11: Scaling performances of Sparkhit-recruiter: **(A)** Run time performance of Sparkhit-recruiter for recruiting 100-1000 GB sequencing data to a 72 MB reference genome on a 30 nodes Spark cluster deployed on the Amazon EC2 cloud. Each node has 32 vCPUs. **(B)** Scaling performance of Sparkhit-recruiter. When increasing the number of worker nodes, the mean speed ups are measured by comparing their run times to the run time on 10 worker nodes. We recruited 1.3 TB fastq files (Data-1) to a 72 MB reference genome (Ref-2) on the same cluster of (*A*).

Sparkhit-mapper has slightly higher accuracy and sensitivity than Sparkhit-recruiter on the GIAB data.

3.6.4 Fragment recruitment comparison with MetaSpark

The comparison between Sparkhit and Metaspark (Zhou et al., 2017) was carried out on clusters with 10, 20 and 30 `c3.4xlarge` worker nodes (Fig. 3.13). I mapped 6 million simulated reads to Ref-2 and 1 million simulated reads to Ref-3. For Metaspark, I first converted the simulated fastq files into the read file format specified by the tool, as well as the reference genome file. It is important to note that the run times for converting the files are not included in the comparison). The recruited number of reads and processing time were, then, measured and compared (see the *Appendix* Table S1, S2). When running the tools, I set the k-mer size of both tools to 11 (for Sparkhit, the default is 12).

Sparkhit-recruiter ran 92 to 157 times faster than MetaSpark across different numbers of worker nodes (10, 20 and 30), different numbers of input reads (1 million and 6 millions) and different sizes of reference genomes (72 MB and 142 MB). Although our tool recruited 10% to 12% reads less than MetaSpark using the same k-mer size, I have adjusted to a smaller k-mer size that recruits more reads than MetaSpark, while still running 47 to 124 times faster (see Discussion).

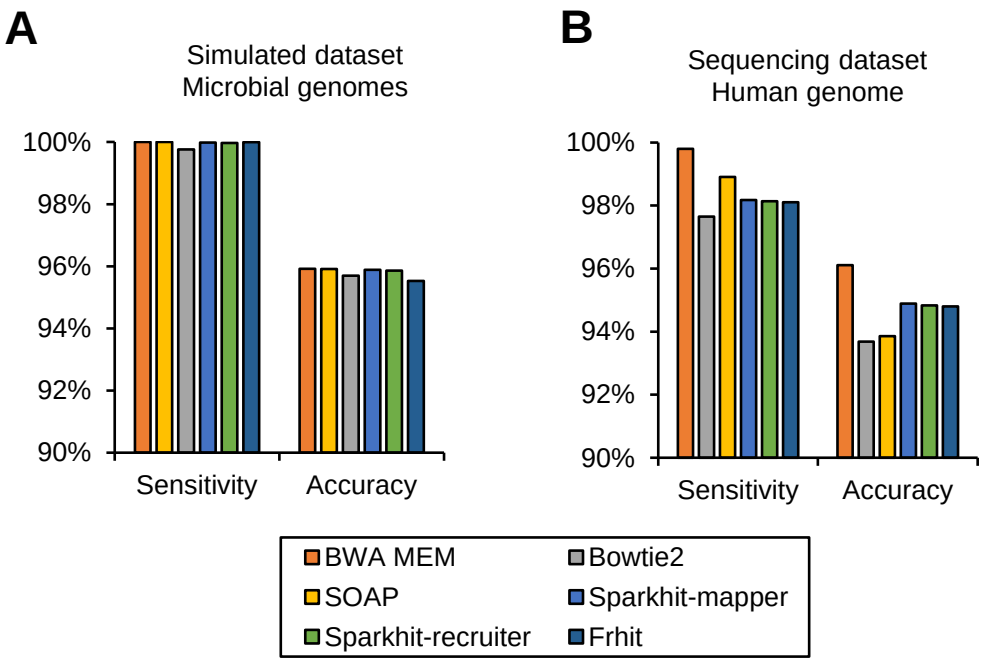

Fig. 3.12: Sensitivity and accuracy comparisons between mapping tools.

3.6.5 Preprocessing comparison with Crossbow

Data preprocessing is a critical step for interpreting cloud stored public datasets. Manually decompressed and distributed large amounts of genomic data on a cluster introduce significant overheads before data analysis. Although, several existing cloud tools have provided preprocessing functions (Schatz, 2009; Langmead et al., 2009; Decap et al., 2015), their preprocessing speeds are limited by their non-parallel implementations. For Sparkhit, the preprocessing was carried out by the Sparkhit-decompressor, a tool that applies parallel decompression to the compressed sequencing data.

I have compared the run time performances on data preprocessing between Sparkhit and Crossbow (Langmead et al., 2009). Cloudburst (Schatz, 2009) was not included in the comparison as its preprocessing step took too much time and was unable to finish. I used the larger dataset (Data-1) and ran Sparkhit and Crossbow respectively on Spark clusters with 50 and 100 worker nodes. For 338 GB Bzip2 compressed data (Data-1, 1.3 TB uncompressed), Sparkhit ran 18 to 32 times faster than Crossbow on 50 and 100 `c3.8xlarge` worker nodes (Fig. 3.14). Since Sparkhit utilizes all CPUs for parallel decompression, its run time performance almost doubled from 50 nodes to 100 nodes, whereas Crossbow had similar run times on both clusters.

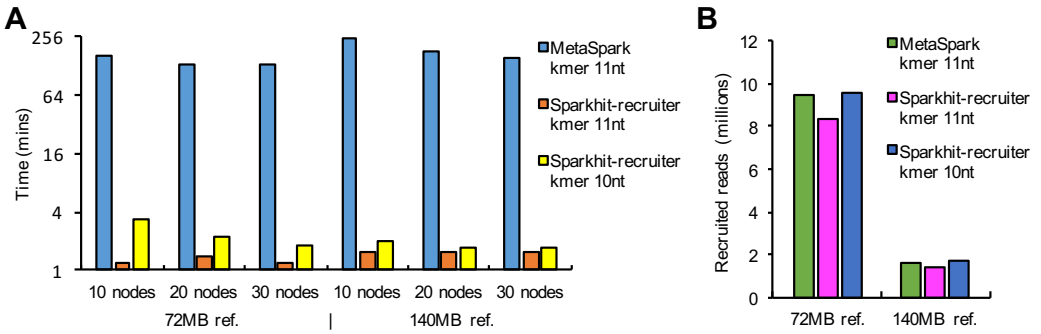

Fig. 3.13: Comparisons between Sparkhit-recruiter and MetaSpark on metagenomic fragment recruitment: **(A)** Run times on recruiting simulated sequencing reads to 72 MB and 142 MB reference genomes. All tests were carried out on 10, 20, and 30 worker nodes Spark clusters. Each worker node has 16 vCPUs. Run times are presented in logarithmic scale of base 2. **(B)** Number of recruited reads on recruiting 6 million simulated reads to 72 MB reference genome and 1 million simulated reads to 142 MB reference genome.

3.6.6 Machine learning library benchmarking and run time performances on different clusters

For the machine learning library, I have compared the run time of each module on a 200GB VCF file (Data-3) containing genotypes of 2504 samples from the 1000 Genomes Project (cohorts in phase 3). The VCF file was the raw input for Sparkhit and all data points were cached into memory. In addition to measure run times of different modules, I also compared their run time performances on a private cluster and the Amazon EC2 cloud (Fig. 3.15). I deployed two Spark clusters with 20 and 40 worker nodes (see the cluster configurations section), where each worker node had the same number of cores. For the private cluster, data was stored on a network file system (NFS) setup on a magnetic disk. Whereas on EC2, data was stored on a Hadoop distributed file system (HDFS) setup on the solid state drive (SSD) with three times redundancy. For k-means clustering, I measured the run times of both data caching and no data caching when increasing iterations from 1 to 40 with 10 iterations increment (Fig. 3.16). The benchmark was carried out on the private cluster with 20 nodes and 640 cores using Data-3.

Since each module opened 640 and 1280 I/O tasks (20 nodes and 40 nodes, each node has 32 cores) to read input data and write output results, the run time performance on the private cluster was significantly slower than on the Amazon EC2 cloud (Fig. 3.15). We also observed a significant improvement on run time for cached iterative computations (K-means clustering), compared to non-cached ones (Fig. 3.16).

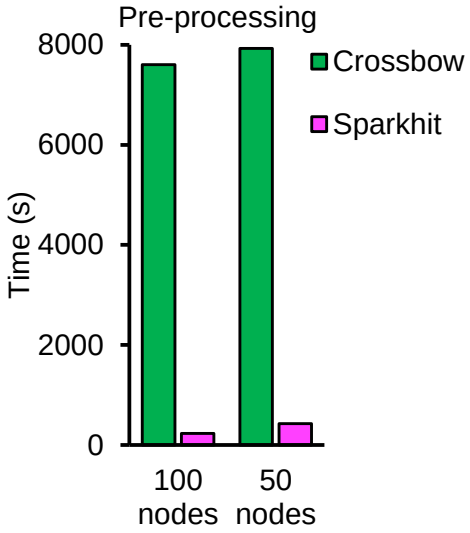

Fig. 3.14: Run time comparisons between Crossbow and Sparkhit for preprocessing 338 TB compressed fastq files on 50 and 100 worker nodes.

3.6.7 Cluster configurations for the benchmarks

All benchmarks on Amazon AWS EC2 were carried out on a Spark cluster that consists of one master node deployed on an `m1.xlarge` computer instance and 10 to 100 worker nodes (varies on different benchmark setups) deployed on the `c3.8xlarge` or the `c3.4xlarge` (when comparing with Metaspark) computing instances. The `m1.xlarge` is a type of general purpose instance, which is balanced on processing, storage, and network resources. It has 15 GB of random-access memory (RAM), 4 $\times$ 420 GB of magnetic disk storage and 4 vCPUs (Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz). When renting the instances from Amazon, the cost was $0.35 per hour, per instance. The `c3.4xlarge` and `c3.8xlarge` compute instances are two types of compute optimized instances, providing high performing processors. The `c3.4xlarge` has 30 GB RAM, 2 $\times$ 160 GB solid state disk (SSD) storage and 16 vCPUs (Intel Sandy Bridge, E5-2670), while the `c3.8xlarge` has 60 GB RAM, 2 $\times$ 320 GB solid state disk (SSD) storage and 32 vCPUs (Intel Sandy Bridge, E5-2670). The `c3.8xlarge` also has optimized network performance with a bandwidth of 10 Gigabit/sec. The standard price for the `c3.4xlarge` was $0.840 per hour, per instance, while the `c3.8xlarge` was $1.680 per hour, per instance.

On the private SGE cluster, Spark was deployed in the standalone mode. When setting up a Spark cluster on the SGE system (Red Hat Enterprise Linux 5.8), a master daemon is, firstly, started on the SGE login node using the "start-master.sh" script included in the Spark package. Once the master node is running, the worker daemons are submitted to the SGE computing nodes to setup worker nodes with designated resources using the "start-slave.sh" script. All worker nodes are registered

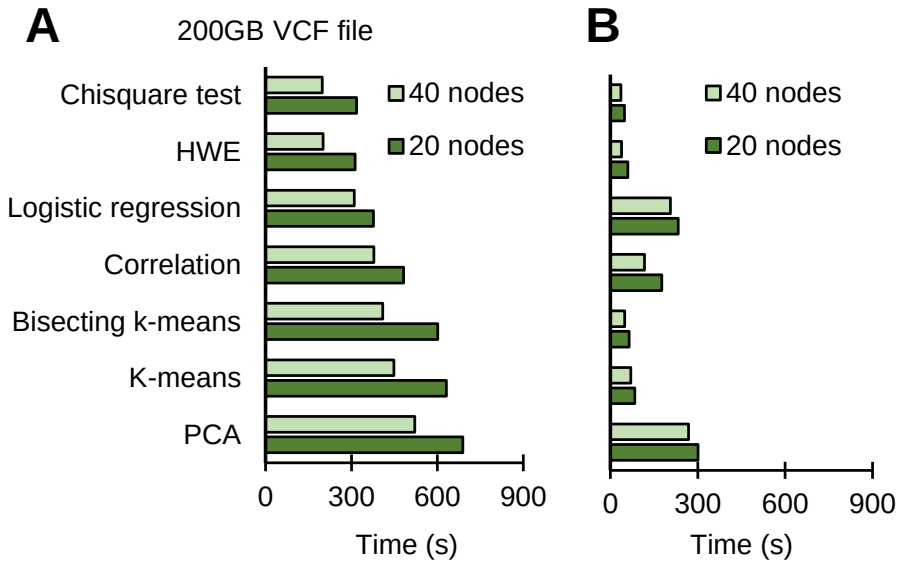

Fig. 3.15: Run times of the machine learning library on **(A)** a private cluster and **(B)** the Amazon EC2 cloud. All computations were performed on a 200 GB VCF file cached in the memory.

to the master node via Secure Shell (SSH) by assigning the master's Internet Protocal (IP) address to each worker daemon.

The Spark master node was deployed on the SGE login node with 24 cores (Intel(R) Xeon(R) CPU L5640 @ 2.27GHz) and 142 GB RAM. 20 to 40 Spark worker nodes were deployed on SGE computing nodes with 32 cores (Intel(R) Xeon(R) CPU E5-2658 0 @ 2.10GHz) and 252 GB RAM. When comparing the performance between the EC2 cloud and the private cluster, I used the same number of worker nodes, so that the total number of cores are equal between the two clusters. On the private cluster, I also requested 60 GB (Spark allocated 57.6 GB) RAM for each worker node,

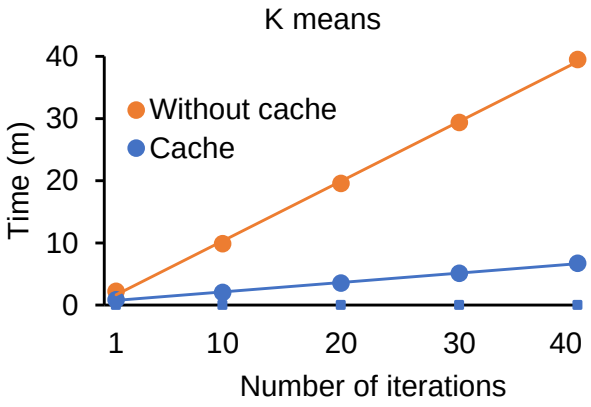

Fig. 3.16: Run times for different iterations of the K means clustering. We ran iterations on the same VCF file from Fig. 3.15, with data caching and non data caching.

Tab. 3.1: Configurations of different computer instances

Resources	Private Cluster		Amazon EC2		
	Login node (Master)	Computing node (Worker)	m1.xlarge (Master)	c3.4xlarge (Worker)	c3.8xlarge (Worker)
Memory	142 GB	252 GB	15 GB	30 GB	60 GB
Memory used	15 GB	60 GB	15 GB	30 GB	60 GB
vCPUs	24	32	4	16	32
vCPUs used	1	32	1	16	32
Hard disk	HDD	HDD	HDD	SSD	SSD
Storage	600 TB	NA	1.68 TB	320 GB	640 GB
File system	GPFS			HDFS	

even though they all have 252 GB available. A shared general parallel file system (GPFS) with 600 TB of disk volume was used to store benchmarking datasets.

When comparing the run time performance between Sparkhit and Metaspark, I used the `c3.4xlarge` instances instead of the `c3.8xlarge` instances. Spark-ec2 requested 30GB (Spark allocated 28.8GB) RAM for each worker node. 75% of the Java heap space was allocated for Spark's RDD memory cache (default is 60%). An HDFS was setup with 3 times redundancy. However, since the `c3.4xlarge` instance only has half the volume size of the `c4.8xlarge` instance, the maximum data size for HDFS storage is 105 GB per node.

Tab. 3.2: The standard and spot prices for different Amazon EC2 instances

	Standard price	Spot price
c3.8xlarge	$1.68/h	$0.35/h - $0.40/h
c3.4xlarge	$0.84/h	$0.17/h - $0.20/h

I was able to get spot prices between $0.17 and $0.20 per hour, per instance for the `c3.4xlarge` and between $0.35 and $0.40 per hour, per instance for the `c3.8xlarge` in the AWS Ireland region. The spot price, a rate of the computing instance bidding system, is introduced by Amazon to attract more users by offering lower prices to avoid computers idling in the Amazon computing center.

3.6.8 NGS data sets for the benchmarks

To benchmark the run time performances on different sizes of reference genomes (Fig. 3.9A-D), three sets of mixed microbial genomes were used: 36 MB reference genomes of 7 common human pathogens (Ref-1), 72 MB reference genomes of Ref-1 mixed with 16 bio-fuel microbes (Ref-2) and 142 MB genome sequences of Ref-2 mixed with 19 oral microbes (Ref-3). Ref-2 was also used in other performance benchmarks (Fig. 3.10, 3.11, and 3.13). See Table 3.3 for more details. The reference

Tab. 3.3: Datasets used for various benchmarks

Dataset	Data-1	Data-2	Data-3	Ref-1	Ref-2	Ref-3
Size	1.3 TB	545 GB	200 GB	36 MB	72 MB	142 MB
File format	Fastq	Fastq	VCF	Fasta	Fasta	Fasta
Run time comparisons between different mappers	✓	✓		✓	✓	✓
Scaling performance of Sparkhit-recruiter	✓				✓	
Machine learning library benchmarking			✓			
Data preprocessing	✓					
Data simulation				✓	✓	✓
Fragment recruitment comparison with MetaSpark					✓	✓
Fragment recruitment of HMP data				✓		

genome sequences were downloaded from the National Center for Biotechnology Information (NCBI) database.

I have used the entire whole genome sequencing (WGS) data (metagenomics) of the HMP project hosted on Amazon S3. These WGS data were sampled from 6 body sites and 15 sub body sites. In total, there are 2.3 TB compressed (8.6 TB uncompressed) fastq files in the bzip2 format. To compare the scalability of different mapping tools (Fig. 3.9A-D), I extracted two subsets of WGS data from tongue dorsum samples: a larger set (Data-1, Table 3.3) of 1.3 TB (uncompressed, 338 GB compressed in Bzip2 format) and a smaller set (Data-2, Table 3.3) of 545 GB (uncompressed). The larger one has also been used to test and compare pre-processing times between Sparkhit and Crossbow (Fig. 3.14). All HMP datasets are hosted in the Oregon region of the Amazon S3 storage.

The 200 GB genotype data (Data-3) in the variant call format (VCF) was used to evaluate the performance of the implemented machine learning library (Fig. 3.15). All datasets are hosted at the Amazon S3 (Virginia region). The complete list of files can be found in the supplementary file 3 Table S26 at (Huang et al., 2018).

3.6.9 Discussion

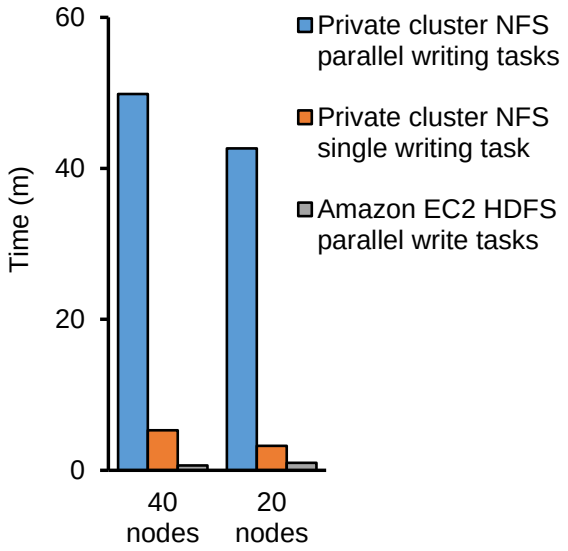

Fig. 3.17: I/O performance on different clusters: For 40 nodes cluster, parallel writing tasks operate on 1280 file handles. For 20 nodes cluster, parallel writing tasks operate on 640 file handles. The single writing task operates on 1 file handle.

In this chapter, I presented a Spark based distributed computational framework for large scale genomic analytics, called Sparkhit. Sparkhit incorporates a variety of tools and methods that are programmed in the Spark extended MapReduce model. I have described (i) the implementations of a fragment recruitment tool and a short-read mapping tool using Spark's RDD API, (ii) the construction of a general tool wrapper to invoke and parallelize external tools, and (iii) the integration of Spark's machine learning library for downstream data mining. I also presented the architecture of Sparkhit and the utilities that I used for deploying Spark clusters and downloading public datasets. Sparkhit outperforms most Hadoop and Spark based bioinformatics tools in computational run time. Using my framework, I analyzed large amounts of public genomic data on the cloud within a short time.

The performance benchmarks demonstrated the scalability of Sparkhit. Sparkhit-recruiter scaled linearly (Fig. 3.17) with the increasing amount of input data, as I utilized Spark RDD to balance data distribution and optimized the computational parallelization. In addition, the distributed data I/O via HDFS further reduces latency. On HDFS, data is distributed and loaded locally or from the closest node (depending on the redundancy setting of HDFS), avoiding massive data transfer across the network. I also observed the advantage of using HDFS when comparing the run times of the machine learning library between the Amazon EC2 cloud and the private cluster, which stored input data on an NFS shared by all worker nodes. On NFS, all data was read and written through the network connection to a mounted volume that saturated the bandwidth.

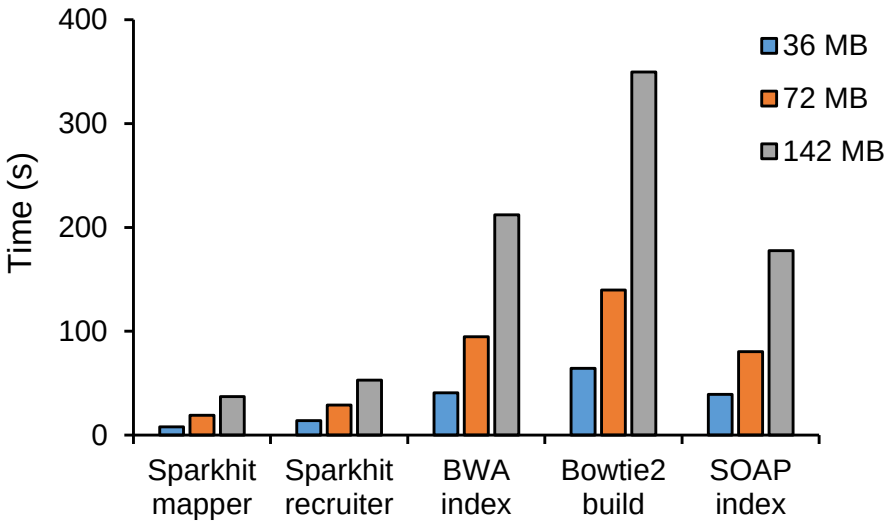

Fig. 3.18: Run time comparison of different tools for building reference index: The comparison was carried out on single computer node (the m1.xlarge Amazon EC2 instance). All tools ran on 36 MB, 72 MB and 142 MB reference genomes respectively.

When scaling out to more worker nodes, a slight slowdown was observed (Fig. 3.17). The slowdown was caused by the overhead of building the reference index, which runs solely on the driver node. This can be improved by pre-building the reference index using our locally implemented recruiter (a Java-based tool included in our framework). Moreover, the overhead for constructing the reference index is small compared to the run time of the fragment recruitment process. For Sparkhit, the reference index construction runs much faster compared to other Burrows-Wheeler transform (BWT) based methods (Fig. 3.18).

My tool had excellent run time performance on data preprocessing compared to Crossbow (18 to 32 times faster) and significant run time improvement on fragment recruitment compared to MetaSpark (92 to 157 times faster). Although Sparkhit recruits 10% to 12% less reads than MetaSpark, I can adjust to a smaller k-mer size that recruits slightly more reads than MetaSpark, while still ran 47 to 124 times faster (Fig. 3.13). In addition, our tool has a comparable accuracy and sensitivity on sequence mapping (Fig. 3.12). Sparkhit-recruiter also offers more options for fragment recruitment, such as an option for reporting the best match for each read and an option to choose between global or local alignment, whereas MetaSpark can only apply local alignment.

4 Reflexiv: Parallel De Novo genome assembly

> *You can't connect the dots looking forward; you can only connect them looking backwards.*
>
> **— Steve Jobs**
> Co-founder, Chairman, and CEO of Apple Inc.

In this chapter, I present a new parallel *de novo* genome assembler, called Reflexiv, and a distributed data structure implemented in the assembler. Reflexiv is built on top of the Apache Spark platform. It uses Spark RDD to distribute large amounts of k-mers across the cluster and assembles the genome in a recursive way. By distributing large amounts of k-mers across the computing cluster, Reflexiv addresses the memory intensive challenge in the *de novo* genome assembly process.

I will start by introducing the new data structure called *Reflexible Distributed K-mer* (RDK). The RDK is a higher level abstraction of the Spark RDD. I have implemented a random k-mer reflecting method to reconnect and extend the distributed k-mers. I describe how repeats in the genome are detected and how to pop bubbles in the assembly. I will also present the time complexity of the algorithm and how to measure the memory consumption of the program.

In the result section, I present the performance benchmarks on the assembler. I mainly focus on evaluating its run time performance and its assembly quality. I will also compare its performance to the other assemblers. In general, Reflexiv has a similar assembly quality to the other distributed assemblers, such as Ray and Abyss. However, Reflexiv has a much better run time performance than the other tools on an ethernet connected computer cluster.

4.1 Reflexible Distributed K-mer (RDK)

The primary objective of the Reflexible Distributed K-mer (RDK) data structure is to make the entire repertoire of k-mers in a given genome distributable. Distributable means that each item of the repertoire (each k-mer) can be independently assigned to, stored in, and retrieved from different computer instances, while still able to

re-establish its connections with other k-mers in the original genome sequence. This way, the original genome can be assembled in a distributed manner. To understand how RDK works, let's first take a look at the state-of-the-art *de bruijn* graph.

The conventional approach achieves distributed genome assembly by distributing a *de bruijn* graph in a computer cluster. A *de bruijn* graph re-establishes the connections between k-mers based on their overlaps of nucleotide sequences. An *n* nucleotides k-mer normally has *n-1* nucleotides overlap with its adjacent k-mer. In the graph, each k-mer is a vertex and the overlap with its adjacent k-mer is a directed edge. When traversing the *de bruijn* graph, an assembler program constantly searches the adjacencies in the memory and extends the sequences.

Yet, in a distributed system, the *de bruijn* graph is partitioned and stored in different computer instances with independent memories. To acquire the next adjacent vertex, the physical location (which computer instance the vertex is located) of the adjacent k-mer must be provided (Fig. 2.13). Thus, when constructing the distributed *de bruijn* graph, three components are required: (i) a k-mer as a vertex, (ii) its overlap with its adjacent k-mer as an edge, and (iii) an index pointing to the physical location of the adjacent k-mer as a pointer. In this way, the connections of all vertices in the distributed *de bruijn* graph are completed. When traversing a distributed *de bruijn* graph in a computer cluster, assemblers constantly search the adjacencies through the entire computer cluster. Once an adjacency of a vertex is found at another computer instance, the vertex is sent to the computer instance where the pointer points to. Most assemblers use the message passing interface (MPI) to implement their algorithms and handle their messaging processes (Simpson et al.,

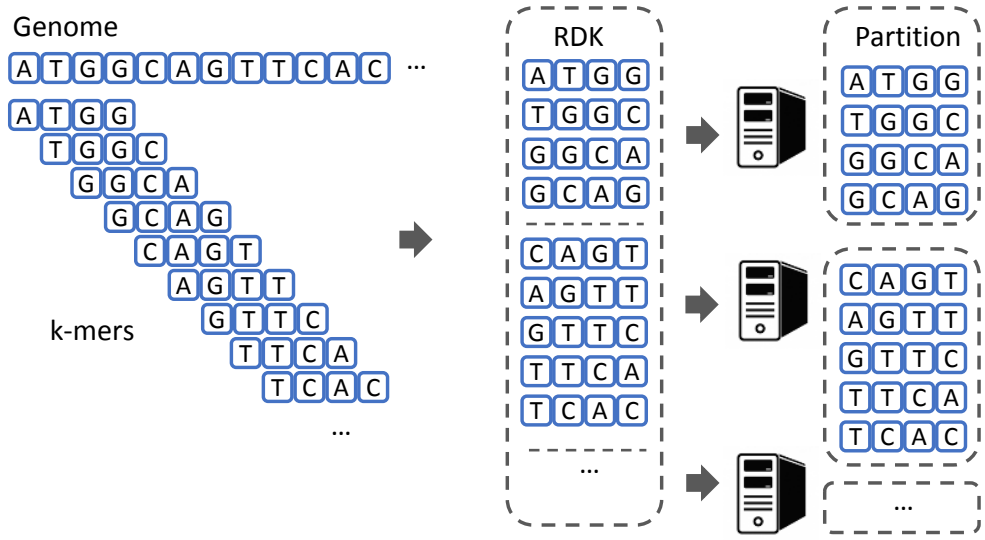

Fig. 4.1: A simplified representation of an RDK. An RDK is a long list of k-mers. It can be randomly partitioned and distributed to different computer instances. Compared to the state of the art *de bruijn* graph, an RDK only stores the vertices of the graph.

2009; Boisvert et al., 2010). However, this approach has two weaknesses: (i) storing the pointers of all k-mers consumes a considerable amount of memory and (ii) the constant messaging in the graph traversal process introduces significant overhead in the run time performance.

To address the two weaknesses, I have invented a new data structure called Reflexible Distributed K-mer (RDK). An RDK has two attributes: (i) Distributed and (ii) Reflexible. The attribute "distributed" states the fact that all k-mers are stored in different computer instances where communications are only viable through the provided network. Once separated, adjacent k-mers are not expected to be found by local in-memory searches. An RDK can be represented by a long list of k-mers, which can be partitioned and distributed to a certain amount of computer instances (Fig. 4.1). Each computer instance holds a part of the RDK represented by a sub list of k-mers. Hence, distributing an RDK can be completely arbitrary and simple. Compared to a distributed *de bruijn* graph, an RDK only stores k-mers without edges and pointers. As there is a large amounts of k-mers in a genome sequence, an RDK is much more memory efficient than a *de bruijn* graph. However, without edges and pointers, no adjacencies are provided to the k-mers of the RDK. To re-establish their connections, the second attribute of RDK, Reflexible, is needed.

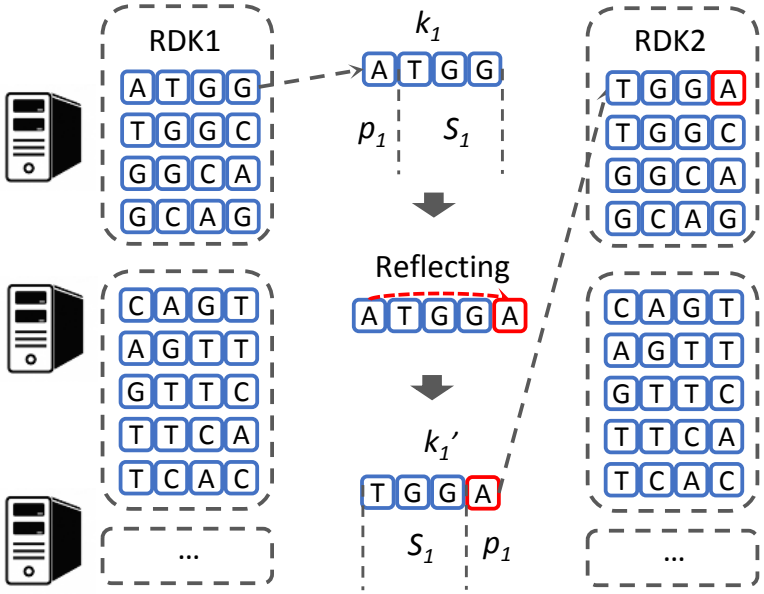

Fig. 4.2: K-mer reflecting in an RDK. A 4-nucleotide k-mer k_1 has a 1-nucleotide prefix p_1 and a 3-nucleotide suffix S_1. A k-mer reflecting step switches the positions of p_1 and S_1. The reflecting process creates a reflected k-mer k'_1.

Reflexible is defined as "capable of being reflected". The word 'Reflected' describes an effect that an object been throw back in a reversed status, e.g. an image been reflected by a mirror. In an RDK, we define a reflected k-mer as a k-mer with a swapped order of nucleotides. Different from a reversed k-mer which reverses the

complete order of its nucleotide sequence, a reflected k-mer only switches parts of its nucleotide sequences. For instance, an *n* nucleotides k-mer *k* has an *n-1* nucleotides of suffix *S* and an 1 nucleotide prefix *p* (Fig. 4.2). A reflected k-mer *k'* switches the position of the prefix and the suffix, and re-concatenates the sequences. Thus, for a k-mer k_1, its reflected k-mer k'_1 can be expressed as:

$$\begin{aligned} k_1 &= \{p_1, S_1\} \\ k'_1 &= \{S_1, p_1\} \end{aligned} \tag{4.1}$$

where a bracket '{}' represents a concatenation of strings of nucleotides, the capital *S* represents a fixed *n-1* nucleotides suffix, and the lower case *p* represents an unfixed nucleotides prefix (changeable after extension, see below). Once reflected, a k-mer's *n-1* nucleotides suffix is placed in the front of the k-mer.

When extracting k-mers from the sequencing data with 1 nucleotide shift, there are two adjacent k-mers k_1 and k_2, where the *n-1* nucleotides suffix S_1 of k_1 is overlapped with the *n-1* nucleotides prefix P_2 of k_2 (Fig. 4.2). After reflecting k_1, its *n-1* nucleotides suffix S_1 is placed in the front of the k-mer. Thus, the reflected k-mer k'_1 has an identical *n-1* nucleotides prefix as the k-mer k_2. The two overlapped k-mers can be expressed as:

$$\begin{aligned} k_1 &= \{p_1, S_1\} \\ k'_1 &= \{S_1, p_1\} \\ k_2 &= \{P_2, s_2\} \\ S_1 &= P_2 \end{aligned} \tag{4.2}$$

where the capital *P* represents a fixed *n-1* nucleotides prefix and the lower case *s* represents an unfixed suffix (changeable after extension, see below). In a distributed system, the distributed *de bruijn* graph uses the edge and the pointer of k_1 to help locating its adjacent k-mer k_2. However, in an RDK, only k-mers are stored as a long list in the cluster and two overlapped k-mers, k_1 and k_2, are likely to be stored in different computer instances. Since the reflected k-mer k'_1 has the same *n-1* nucleotides prefix with k_2, sorting the list of k-mers in an alphabetic order will rearrange the two k-mers to neighboring positions in the list (Fig. 4.3). Thus, the adjacency of k_1 and k_2 is found after the reflecting and sorting processes. Using the adjacency, we can connect the two k-mers and build a new extended *n+1* nucleotides k-mer k_{1+2}. The extended k-mer can be expressed as:

$$
\begin{aligned}
k'_1 &= \{S_1, p_1\} \\
k_2 &= \{P_2, s_2\} \\
S_1 &= P_2 \\
k_{1+2} &= \{p_1, (S_1||P_2), s_2\}
\end{aligned}
\tag{4.3}
$$

where the || symbol indicates that both S_1 and P_2 are applicable to the case.

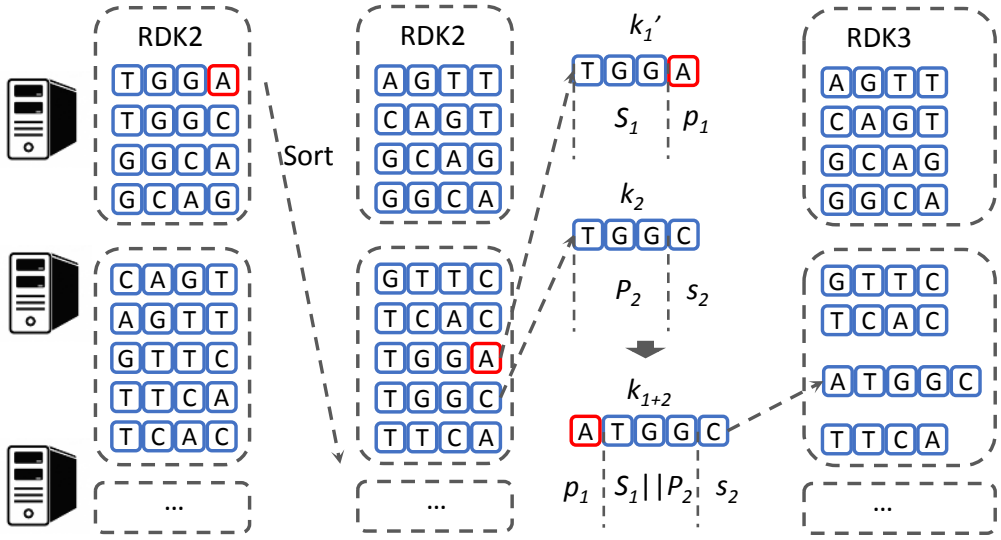

Fig. 4.3: Reestablishing k-mer adjacency: The sorting process places the reflected k-mer k_1 and its adjacent k-mer k_2 at neighboring positions. When going through the RDK k-mer list, the two adjacent k-mers are extended to k_{1+2}.

After the extension, the $n+1$ nucleotides k-mer k_{1+2} can be represented by an n-1 nucleotides suffix S_{1+2} and a 2 nucleotides prefix p_{1+2}. When reflecting the extended k-mer k_{1+2}, we still keep a fixed n-1 nucleotides suffix S_{1+2} and switch its position with the extended prefix p_{1+2} (Fig. 4.4). Thus, the extended k-mer k_{1+2} and its reflected k-mer k'_{1+2} can be expressed as:

$$
\begin{aligned}
k_{1+2} &= \{p_1, (S_1||P_2), s_2\} \\
k_{1+2} &= \{p_{1+2}, S_{1+2}\} \\
k'_{1+2} &= \{S_{1+2}, p_{1+2}\}
\end{aligned}
\tag{4.4}
$$

Since the extension is based on the sequence of the k-mer k_2, the fixed size suffix S_{1+2} of k-mer k_{1+2} is identical to the suffix S_2 of k-mer k_2. Whereas the extended prefix p_{1+2} is longer than the prefix p_2 of the k-mer k_2:

$$
\begin{aligned}
k_{1+2} &= \{p_{1+2}, S_{1+2}\} \\
k_2 &= \{p_2, S_2\} \\
S_{1+2} &= S_2 \\
p_{1+2} &> p_2
\end{aligned}
\tag{4.5}
$$

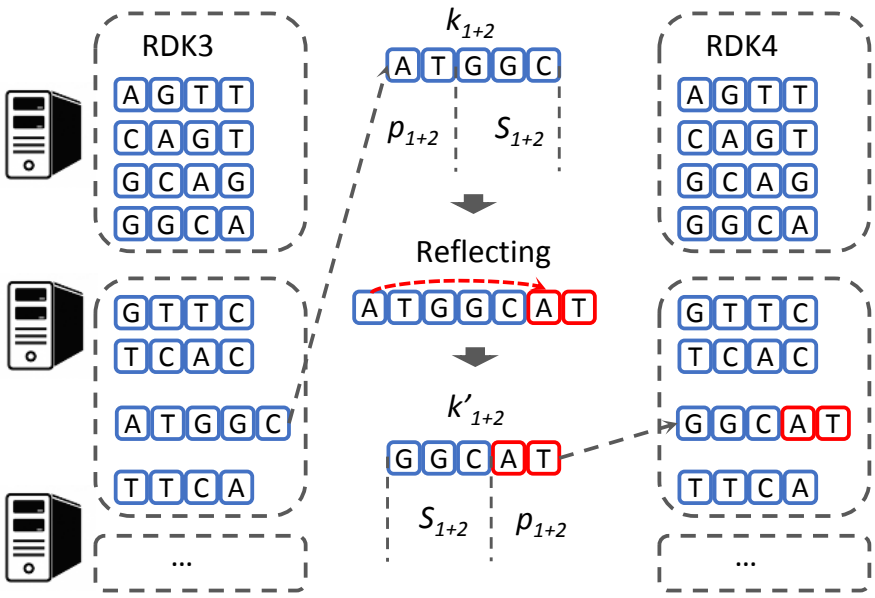

Fig. 4.4: Reflecting an extended k-mer: The extended k-mer k_{1+2} has a 2-nucleotide prefix p_{1+2} and a 3-nucleotide suffix S_{1+2}. The reflecting step switches the positions of p_{1+2} and S_{1+2}. After the k-mer reflecting process, a reflected k-mer k'_{1+2} is created in the RDK.

On the original genome sequence, the k-mer k_2 is likely (if not located at the end of the genome) to have another adjacent k-mer k_3 which has an *n-1* nucleotides prefix overlapped with the *n-1* nucleotides suffix of k_2. By applying the same reflecting and sorting methods to k_2 and k_3, we can acquire the adjacency of the two k-mers and connect k_2 with k_3. As k_2 and the extended k-mer p_{1+2} have the identical *n-1* nucleotides suffixes, we can also acquire the adjacency of p_{1+2} and k_3 (Fig. 4.5), thus further extend k-mer k_{1+2}. The extension can be expressed as:

$$
\begin{aligned}
k'_{1+2} &= \{S_{1+2}, p_{1+2}\} \\
k_3 &= \{P_3, s_3\} \\
S_{1+2} &= P_3 \\
k_{1+2+3} &= \{p_{1+2}, (S_{1+2}||P_3), s_3\}
\end{aligned}
\tag{4.6}
$$

where, P_3 is the fixed *n-1* nucleotides prefix of the k-mer k_3 and s_3 is the suffix of k_3. After the second round of extension, the extended *n+2* nucleotides k-mer k_{1+2+3} has a fixed *n-1* nucleotides suffix and a 3-nucleotide prefix.

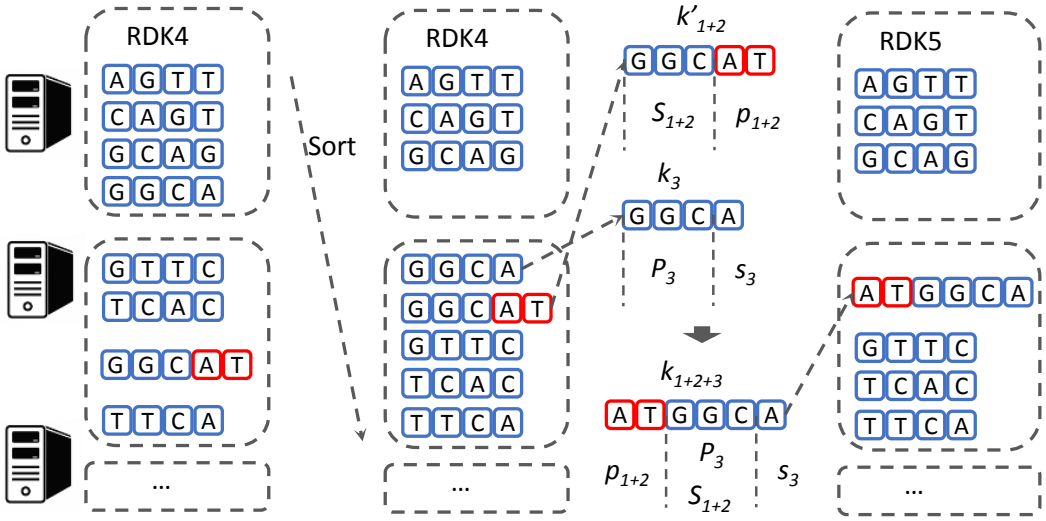

Fig. 4.5: Reconnecting adjacent k-mers: The extended k-mer k_{1+2} has an adjacent k-mer k_3, which has a 3-nucleotide prefix P_3 and a 1-nucleotide suffix s_3. P_3 is identical to the prefix S_{1+2} of the reflected k-mer k'_{1+2}. After sorting the RDK list, the reflected k-mer k'_{1+2} is placed at the neighboring position of its adjacent k-mer k_3. Thus, k'_{1+2} and k_3 can be merged as k_{1+2+3}.

After *m-1* iterations of the reflecting, sorting, and extension steps, the first k-mer k_1 is able to extend to the k-mer k_m. The last iteration extension of the k-mer can be expressed as:

$$\begin{aligned} k'_{1+\ldots+(\text{m-1})} &= \{S_{1+\ldots+(\text{m-1})}, p_{1+\ldots+(\text{m-1})}\} \\ k_m &= \{P_m, s_m\} \\ S_{1+\ldots+(\text{m-1})} &= P_m \\ k_{1+\ldots+\text{m}} &= \{p_{1+\ldots+(\text{m-1})}, (S_{1+\ldots+(\text{m-1})} || P_m), s_m\} \end{aligned} \tag{4.7}$$

where the subscript *1+...+(m-1)* represents *m-2* rounds of extensions and the subscript *1+...+m* denotes *m-1* rounds of extensions started from k-mer k_1. The extension procedure can be iterated until it reaches the end of the genome or the start of a repeat region (see the repeat detection section below) and, thereby, assembles a contig of the genome.

To sum up, the two attributes of the RDK data structure, (i) distributed and (ii) reflexible, enable RDK to store the entire repertoire of k-mers from a genome without storing the additional adjacencies for each k-mer. Whereas, the iteration of the (i) reflecting, (ii) sorting, and (iii) extension methods allow k-mers to reestablish their adjacenies and assemble the genome sequence in a distributed system.

4.2 Random k-mer reflecting and recursion

Sorting the entire list of k-mers is very time consuming, as there is a large number of k-mers on a genome sequence. Reflecting and extending just 1 k-mer in each iteration is not efficient and the entire assembly process is extremely time consuming. Thus, I introduce a random k-mer reflecting method to parallelize the assembly process and to improve the performance of RDK based methods.

On a genome sequence, each k-mer normally (neither at the end of the genome sequence nor at the edge of a repeat region) has two adjacent k-mers. For instance, an n nucleotides k-mer k_m has two adjacent k-mers, k_{m-1} and k_{m+1}, where k_{m-1}'s n-1 nucleotides suffix S_{m-1} is overlapped with the n-1 nucleotides prefix P_m of k_m and k_{m+1}'s n-1 nucleotides prefix P_{m+1} is overlapped with the n-1 nucleotides suffix S_m of k_m. If we reflect both k_{m-1} and k_m at the same time, the reflected k-mers k'_{m-1} and k'_m should be able to reconnect to the unreflected k-mers k_m and k_{m+1}, respectively. However, since the k-mer k_m has been reflected as k'_m, k'_{m-1} will not be able to find its adjacency with k_{m-1}. Thus, continuously reflecting the k-mers will not work efficiently.

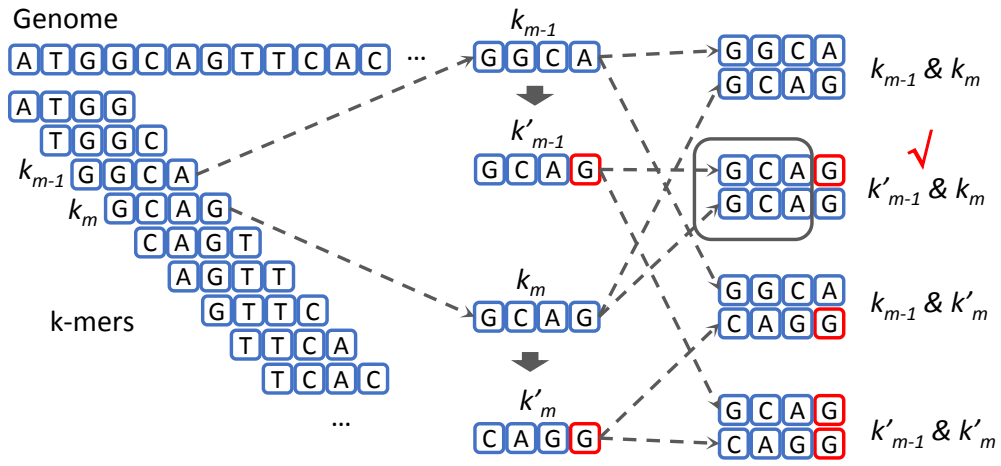

Fig. 4.6: Combinations of two adjacent k-mers after random k-mer reflecting: The two adjacent k-mers, k_{m-1} and k_m, will only be placed at neighboring positions when km-1 is reflected and k_m is not reflected.

The key to reestablish the adjacency between two adjacent k-mers, k_{m-1} and k_m, in an RDK is to reflect the first k-mer k_{m-1} and sort it together with the unreflected second k-mer k_m, so that they are placed at neighboring positions in the long k-mer list. Reflecting both k-mers k_{m-1} and k_m at the same time will not establish their adjacency as both k-mers placed their n-1 suffixes in the fronts of their reflected k-mers k'_{m-1} and k'_m. After sorting, the two reflected k-mers k'_{m-1} and k'_m will not be placed at neighboring positions in the list. In another case, reflecting the second k-mer k_m and sorting it together with the first k-mer k_{m-1} will also not work, as the

n-1 nucleotides suffix of k_m is only overlapped with the *n-1* nucleotides prefix of the third k-mer k_{m+1} (Fig. 4.6 and Fig. 4.7B). To summarize, in all the four scenarios:

(i) k_{m-1} and k_m,

(ii) k'_{m-1} and k_m,

(iii) k_{m-1} and k'_m,

(iv) k'_{m-1} and k'_m

only the (ii) k'_{m-1} and k_m scenario will establish their adjacency after the sorting step.

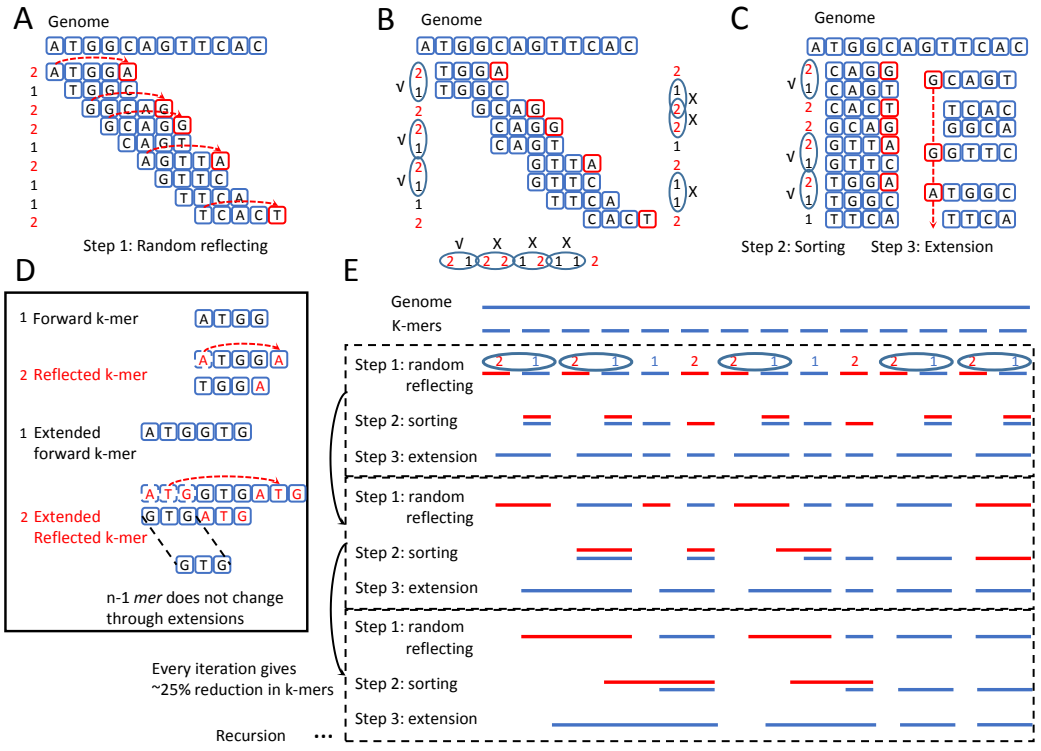

Fig. 4.7: Iterations of three steps in the random k-mer reflecting method: **(A)** Random k-mer reflecting. A reflected k-mer is marked with a red 2. Whereas an unreflected forward k-mer is marked with a blue 1. **(B)** An overview of all combinations. Only the 2-1 combinations can establish their adjacencies after sorting. **(C)** Sorting and extension steps. **(D)** After extension, the extended k-mers still keep a fixed *n-1* nucleotide suffix, where *n* is the length of the k-mers. **(E)** An overview of the extension events throughout the entire genome sequence. Each iteration reduces 25% of k-mers.

To maximize the likelihood of finding more adjacencies in each iteration, I implemented a random reflecting method for all the k-mers in the list. The method simply reflects half of the k-mers in an RDK in a random way (Fig. 4.7A). Thus, for a k-mer k_m, there is a 50% chance of being reflected as k'_m and a 50% chance of being

unreflected as k_m. For its adjacent k-mer k_{m-1}, there is also a 50% chance of being reflected as k-mer k'_{m-1}. As a result, the chance of finding the k'_{m-1} and k_m adjacency is 25%. For its other adjacent k-mer k_{m+1}, there is also a 25% chance of finding the k'_m and k_{m+1} adjacency. Put together, there is a 50% chance of finding an adjacency for each k-mer, if we randomly reflect half of the k-mers in the list. After sorting and merging all the adjacent k-mers, half of the k-mers are connected and extended. The extension process will reduce 25% of the total number of k-mers in the list. Once extended, we re-apply the random reflecting method to the new list of k-mers (Fig. 4.7E). Followed by the same sorting and extension processes, there are another 25% of k-mers reduction. Therefore, the number of k-mers can be expressed as:

$$\begin{aligned} T_1 &= (1 - 0.25) * T_0 \\ T_2 &= (1 - 0.25) * T_1 = (1 - 0.25)^2 * T_0 \\ T_m &= 0.75^m * T_0 \end{aligned} \tag{4.8}$$

where T_0 denotes the total number of initial k-mers in the genome, T_1 denotes the total number of k-mers after the first extension, and m represents the number of iterations. Based on the equation, the total number of k-mers decreases exponentially through the iterations. From the computational point of view, the reduction in k-mers significantly reduces the memory consumption and the computational intensity through such a process.

To simplify, the random k-mer reflecting method is a recursion of (i) random k-mer reflecting, (ii) sorting, and (iii) extension processes.

4.3 Distributed implementation

The parallelization of the random k-mer reflecting method is based on the divide and conquer paradigm. Since an RDK is a long list of k-mers, we can simply divide the whole list of k-mers into several sub lists of k-mers. For each sub list, the first step (i) random k-mer reflecting can be carried out independently on the k-mers in the sub lists. This way, all sub lists of k-mers are reflected simultaneously. Since adjacent k-mers can be placed in different sub lists, the (ii) sorting process must be carried out through the entire list of the k-mers in the RDK, so that adjacent k-mers can be placed in the neighboring positions of the list after sorting. Once the sorting is complete, the sorted list of k-mers can be divided again and (iii) extensions can be carried out simultaneously on each sorted sub list of k-mers. After the extension, we simply repeat all the three steps again until the k-mer number reaches a convergence.

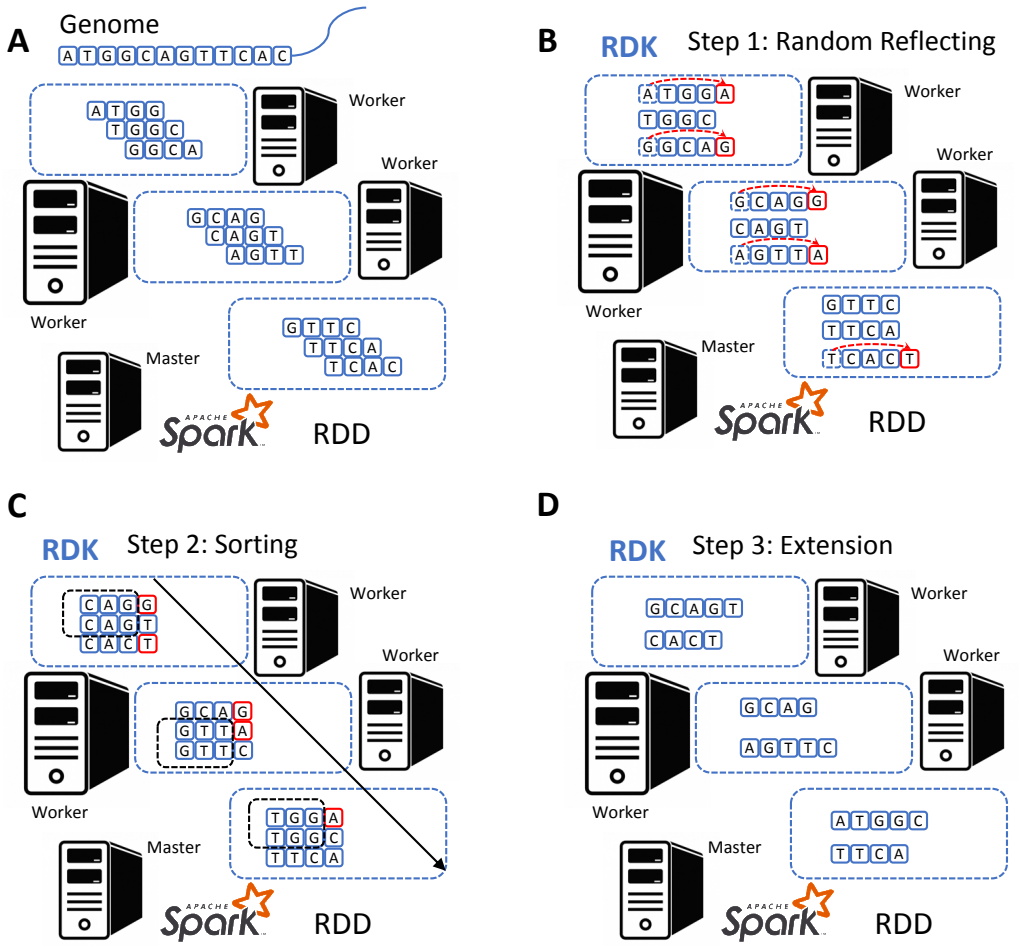

Fig. 4.8: Distributed implementation of the random k-mer reflecting method on top of the Spark platform. **(A)** all k-mers are loaded into an RDD that is distributed across a Spark cluster. **(B)** Each computer instance randomly reflects a sub list of k-mers stored in its memory. **(C)** The sorting process is carried out on the entire list of k-mers through the Spark cluster. **(D)** The extension step is carried out independently on each computer instance.

The distributed implementation of the random k-mer reflecting method is based on the Spark extended MapReduce paradigm. The RDK is built on top of the Spark RDD. An RDD is a collection of elements that can be distributed across the Spark cluster. Since an RDK is a long list of reflexible k-mers, each k-mer can be stored as an element in the RDD (Fig. 4.8A). On the lower level, Spark RDD supplies RDK with all the features such as fault tolerance, automatic parallelization, and distributed 'cache' function. On the higher level, I implemented the random k-mer reflecting method using the programming interfaces provided by the Spark RDD. The implementation mainly consists of three parts: (i) random k-mer reflecting, (ii) sorting, and (iii) k-mer extension (Fig. 4.6B-D).

On a distributed Spark cluster, Spark RDD automatically divides the k-mers into partitions (sub lists of k-mers) and distributes them across the cluster on the worker

nodes. The random k-mer reflecting step simply traverses each element of the partitions and randomly reflects half of the k-mers. This step is implemented in a 'map' function that applies the traversal to all partitions of the RDD simultaneously. After sorting the random reflected k-mers, a 'map' function is also implemented for the extension step. In the extension step, the function also traverses the sorted sub lists of k-mers and merges the adjacent k-mers in the neighboring positions.

Sorting is carried out on the entire list of k-mers in the RDD. I use the Spark build-in sorting algorithm, TimSort. TimSort is a derivation of merge sort, which is a divide and conquer based sorting algorithm. Thus, the sorting process can also be parallelized across the cluster. As mentioned in the related work section, sorting in the Spark cluster corresponds to a 'reduce' phase that shuffles data across different worker nodes. Therefore, the performance of the sorting step is bounded by the network connection between worker nodes. This step is implemented in the 'sortByKey' function that sorts all k-mers in parallel. All k-mers are encoded as a number stored in an 'Long' object. Thus, alphabetically sorting the k-mers is a process of numerically sorting a list of 'Long' objects. Compared to the alphabetically stored 'String' objects, the numerical k-mer encoding approach improves the run time performance of the sorting step (see discussion).

The performance of the recursion processes can benefit from the Spark's distributed in-memory computing. The conventional Hadoop MapReduce computing engine is a more disk-oriented processing model, where each step involves loading and writing large amounts of data from and to the disk. For iterative algorithms, the repetitive loading and writing has a significant impact on its run time performance. The Apache Spark, on the other hand, is a more memory-oriented processing model. It introduces a 'cache' function that allows distributed data to be stored in memory, so that iterative process can re-access the data directly from the RAM.

In theory, the assembly process is complete once the recursion reaches convergence (no more new contigs are generated). However, there are several common assembly problems we have to tackle.

4.4 Repeat detection and bubble popping

There are mainly three events that introduce challenges during the naive RDK based assembly: (i) repeats in the genome, (ii) nucleotide polymorphism, and (iii) sequencing errors. Let us start by looking at the *de bruijn* graph again. In a directed *de bruijn* graph, a repeat event creates two forks that lead to four paths at the beginning and the end of the repeat region (Fig. 4.9). Whereas a nucleotide polymorphism or a sequencing error introduces a bubble at the point of variation.

The bubble also creates two short branches on the original path. Thus, the key to resolve the repeats and pop the bubbles is to find the correct branch for the extension. In the random reflecting method, all k-mers are randomly reflected in each iteration. Therefore, the branch selection in the extension step is completely arbitrary, resulting in false assemblies. To solve this issue, let us first look at the repeats.

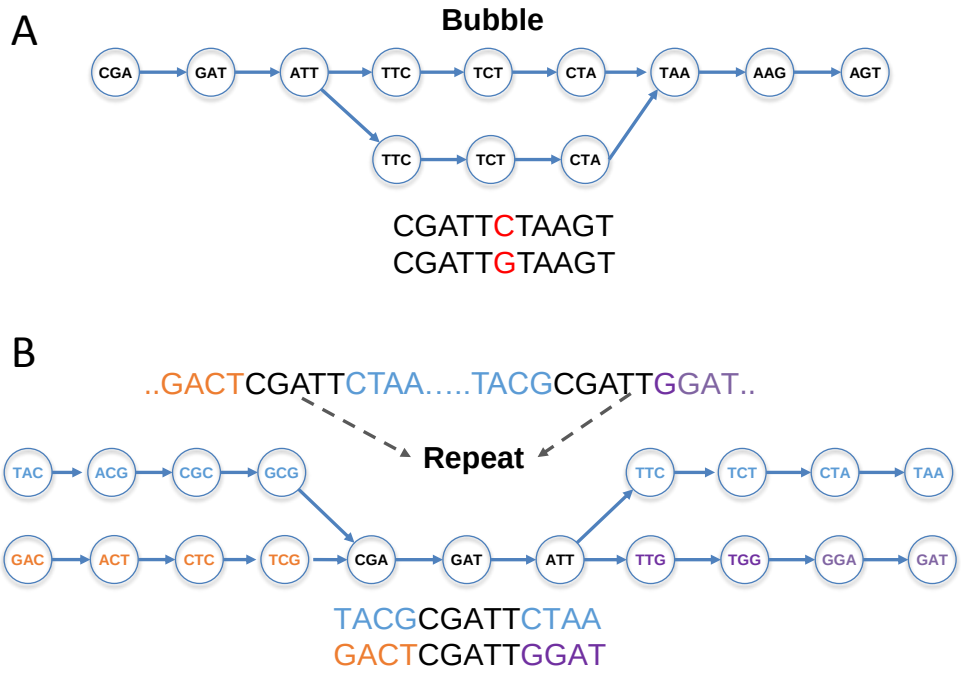

Fig. 4.9: Branches and forks on a de bruijn graph. **(A)** A bubble on a de bruijn graph creates two branches that will soon merge into one path. It also creates a forward fork and a backward fork. **(B)** A repeat event creates four branches and a repeat path. It creates a backward fork and a forward fork.

For a repeat event, there is no sufficient information provided by the *de bruijn* graph indicating the correct branches for the extension. At the contig assembly phase, a conservative approach is to stop the extension at the branches, so that no false assemblies are created. At the later assembly phases, repeats can be resolved by using the mate pair sequencing reads that connects two branches (Nagarajan and Pop, 2013). In the case of RDK, since there are no edges directing the path for the assembly, a branch can not be detected as the way it is found in a *de bruijn* graph.

To identify a repeat region in the RDK, a fork must be found even without the connective information from the edges. An RDK is a list of k-mers or, compared to a *de bruijn* graph, it can be considered as a collection of nodes without edges (Fig. 4.11). A forward fork in a *de bruijn* graph starts at two *n* nucleotides k-mers, k_{f1} and k_{f2}, with the same *n-1* nucleotides prefixes and different 1 nucleotide suffixes (Fig. 4.10). Whereas a backward fork in a *de bruijn* graph starts at two k-mers, k_{b1}

and k_{b2}, with the same *n-1* nucleotides suffixes and different 1 nucleotide prefixes. Since the reflected k-mers, k'_{b1} and k'_{b2}, are k-mers with swapped orders of suffixes and prefixes of k_{b1} and k_{b2}, the backward fork of k_{b1} and k_{b2} can also be represented as a forward fork of k'_{b1} and k'_{b2}.

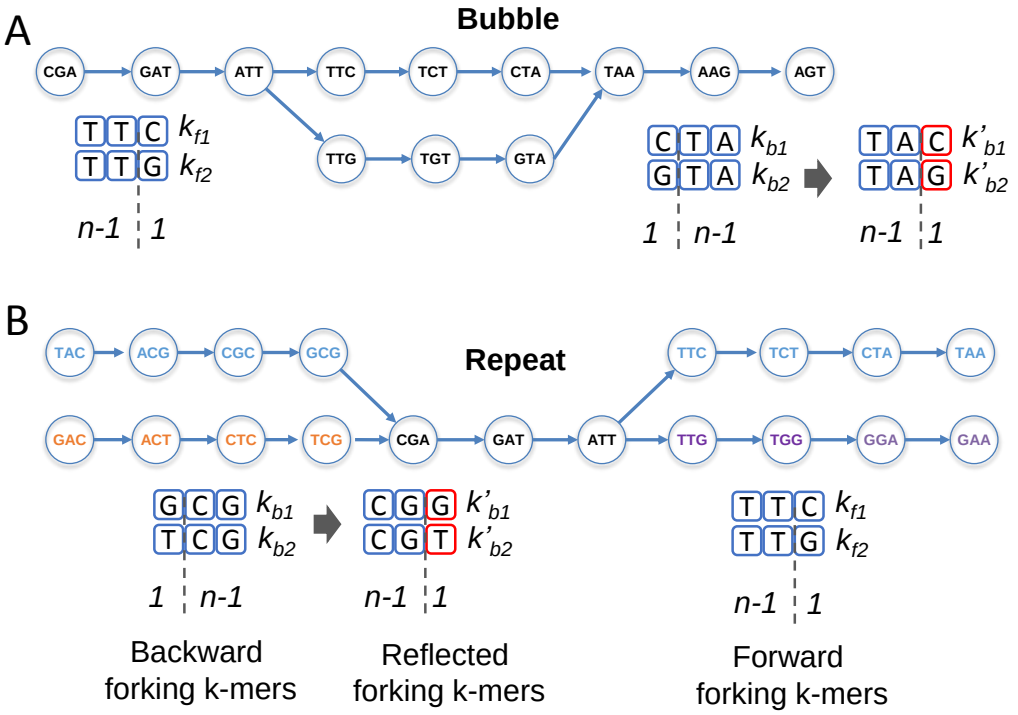

Fig. 4.10: Forward and backward forking k-mers: **(A)** A bubble creates two forward forking k-mers k_{f1} and k_{f2}. The two forward forking k-mers have the same *n-1* nucleotides prefixes and two different 1-nucleotide suffix. The forward forking k-mers will extend and connect to two backward forking k-mers k_{b1} and k_{b2} in *n-1* extensions. The two backward forking k-mers have identical *n-1* nucleotides suffix and two different 1-nucleotide prefixes. Both of the k-mers, k_{b1} and k_{b2}, can also be represented by two reflected forking k-mers k'_{b1} and k'_{b2}. **(B)** A repeat event also creates two forward forking k-mers and two backward forking k-mers. Compared to a bubble event, the forward and backward forking k-mers will not connect in *n-1* extensions.

To rebuild the adjacencies of k-mers in an RDK, a sorting process is needed to place reflected k-mers and non-reflected adjacent k-mers at the neighboring positions in the k-mers list. For finding the forks, I have used the same strategy. Before randomly reflecting k-mers in the RDK, I firstly sort the entire list of k-mers in an alphabetical order. After the sorting, the forward forking k-mers, e.g. k_{f1} and k_{f2}, are placed at the neighboring positions, as they have the same *n-1* nucleotides prefixes. Once the forward forking k-mers are recorded, we reflect all of the k-mers in the RDD to look for backward forks. Sorting all the reflected k-mers, e.g. k'_{b1} and k'_{b2}, will place backward forking k-mers, e.g. k_{b1} and k_{b2}, at neighboring positions as they have the same *n-1* nucleotides suffixes that has been reflected as the *n-1* nucleotides prefixes. After the two sorting processes, all forward and backward forking k-mers are found.

Now, let us take a look at bubbles. A bubble is created either by a sequencing error or a nucleotide polymorphism. A sequencing error creates a bubble with two branches. One of the two branches has a higher k-mer coverage and the other one has a significantly lower k-mer coverage. As for SNP in a diploid genome, the two branches should have similar k-mer coverages and assemblers normally just assemble one copy of the genome as a reference. Thus, to pop a bubble, only the higher coverage branch is assembled to the main path. Since all forking k-mers are in pairs, removing the lower coverage forking k-mers will stop the assembly process of the lower coverage branches of a bubble and a bubble can be popped. Let us come back to repeats. As mentioned above, the extension of a repeat region must be stopped to prevent false assemblies. Removing the lower coverage forking k-mer of a repeat fork will stop the extension of one branch of the repeat (Fig. 4.12), leaving the repeat region only extendable to the higher coverage branch (the event of stopping the repeat region from extending to the higher coverage branch is addressed in the next paragraphs). It is important to note that removing the lower coverage k-mer of the paired forking k-mers is beneficial for bubble popping and repeat detection. Once all sorting processes have been completed and the forking k-mers have been found, forking k-mers with the lower coverage are removed from the RDK.

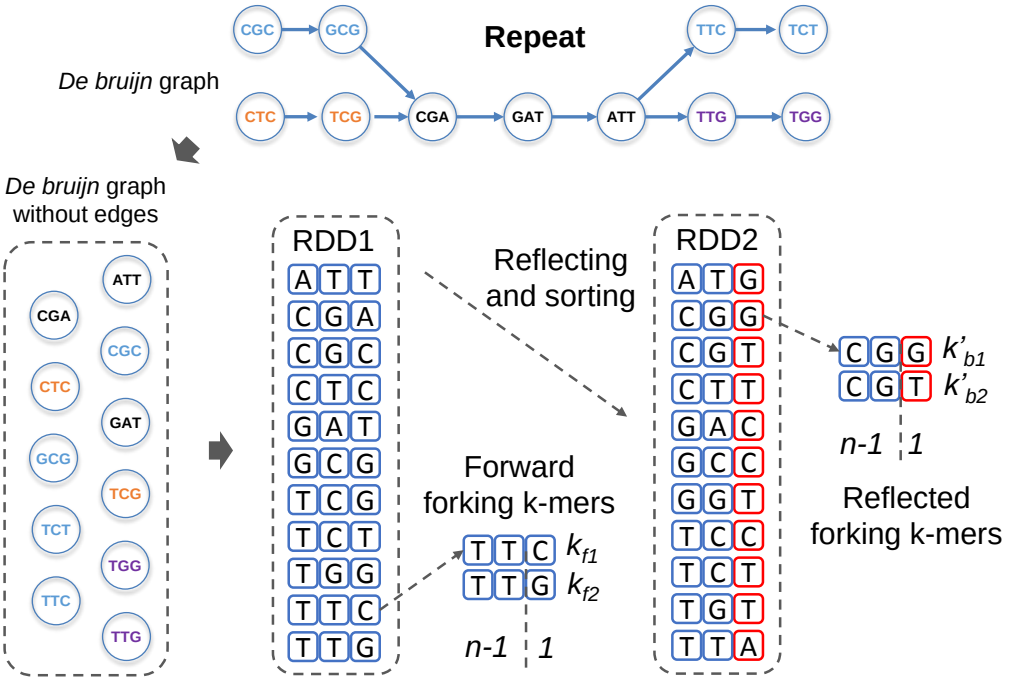

Fig. 4.11: Forward and backward forking k-mers detection: Sorting all forward k-mers will place forward forking k-mers at neighboring positions, as both forward forking k-mers k_{f1} and k_{f2} have the same *n-1* nucleotides prefix. Sorting all reflected forking k-mers will place backward forking k-mers at neighboring positions, as both reflected forking k-mers k'_{b1} and k'_{b2} have the same *n-1* nucleotides prefix.

Once we have identified the higher coverage forking k-mers, the next step is to distinguish repeat forks and bubble forks. A fork can be introduced either by a repeat

event or by a bubble. The difference between a repeat fork and a bubble fork is that a bubble fork will soon converge into the main path after the variation point, whereas a repeat fork will not (Fig. 4.12). As mentioned earlier, a backward fork is also a reflected fork. For a bubble fork, the higher coverage branch started from a forward fork will meet a reflected fork after the variation point. For instance, a single nucleotide polymorphism (SNP) on a diploid genome creates a bubble with two *2n-1* nucleotides branches, where *n* is the length of the k-mers. We can also understand it as a variation point which creates *n-1* variant k-mers. Thus, to detect a bubble in an RDK, a higher coverage forward forking k-mer should meet a higher coverage reflected forking k-mer in *n-1* extensions. As for a repeat, the higher coverage branch started from a forward fork or a reflected fork are not going to meet a reflected fork or a forward fork in *n-1* extensions.

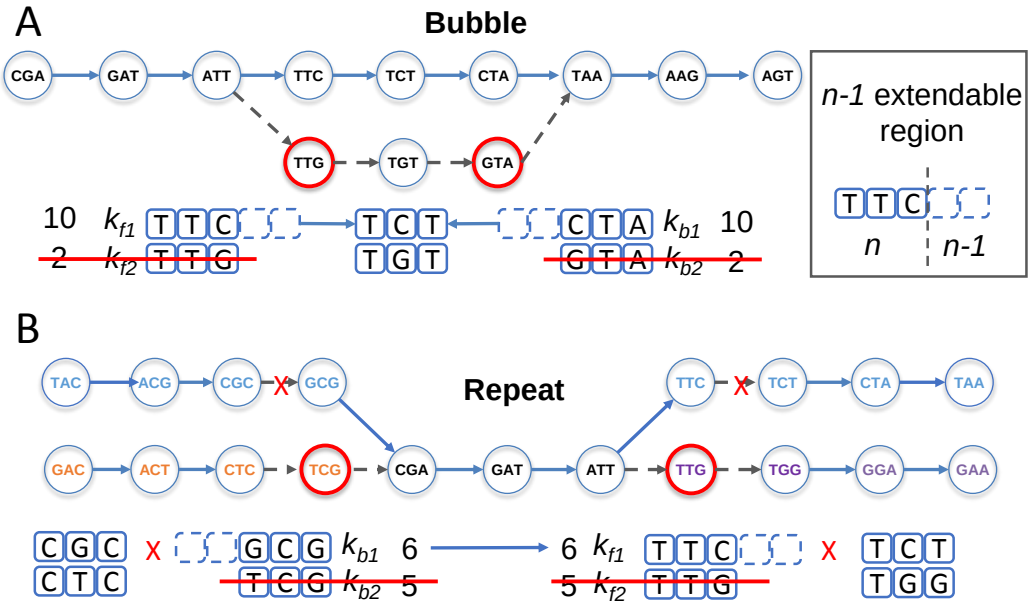

Fig. 4.12: Decision making for bubble forking k-mers and repeat forking k-mers. **(A)** Removing the lower coverage forking k-mers, k_{f2} and k_{b2}, will either correct a sequencing error or solve a SNP event. Extendable regions are given to the higher coverage forking k-mers, k_{f1} and k_{b1}. The extendable region allow both k-mers to extend maximum *n-1* nucleotides. In a bubble event, the two forking k-mers will connect in *n-1* nucleotides extensions. Once the two k-mers connect, the extendable regions are removed and the bubble has been popped. Red circled nodes represent removed lower coverage forking k-mers. Grey dashed arrows represent severed connections. **(B)** In a repeat event, removing the lower coverage forking k-mers, k_{f2} and k_{b2}, will stop the repeat region connecting to the two lower coverage branches. Whereas the extendable regions of the two higher coverage forking k-mers, k_{f1} and k_{b1}, will stop connecting to the two higher coverage branches, as the two forking k-mers will not meet backwark forking k-mers in *n-1* nucleotides.

I introduce a marker on each identified forking k-mer, called an extendable region. An extendable region restricts the maximum extensions allowed (*n-1* extensions) for a given forking k-mer until it meets a reflected forking k-mer, which has also

been marked with an extendable region. For instance, after sorting all the k-mers, a forward forking k-mer, k_{f1}, is found and an *n-1* nucleotides extendable region is given to k_{f1}. Assume k_{f1} is a forward forking k-mer at the start of a bubble. Then, k_{f1} will connect to its reflected forking k-mer k_{b1}, which also has been marked with an extendable region, in *n-1* extensions. In this way, only the higher coverage branch is connected to the main path (since the forking k-mer of the lower coverage branch has been removed) and a bubble has been popped. However, if k_{f1} is a forward forking k-mer at the start of a repeat. Without the restriction from the extendable region, it is able to connect to one of the two k-mers (the higher coverage forking k-mer, as the lower coverage forking k-mer has been removed), k_{r1} and k_{r2}, from the two branches. Now that the k-mer k_{f1} has been given an extendable region marker and the two k-mers, k_{r1} and k_{r2}, have not been marked with extendable regions, the extension of the forward forking k-mer k_{f1} is stopped and a repeat region will not be assembled to prevent false assemblies.

4.5 The assembly pipeline

The pipeline of the assembly process consists of 6 parts:

1. K-mer extraction
2. K-mer counting
3. Forward forking k-mer detection
4. Reflected forking k-mer detection
5. Assembly iterations
6. Result summary

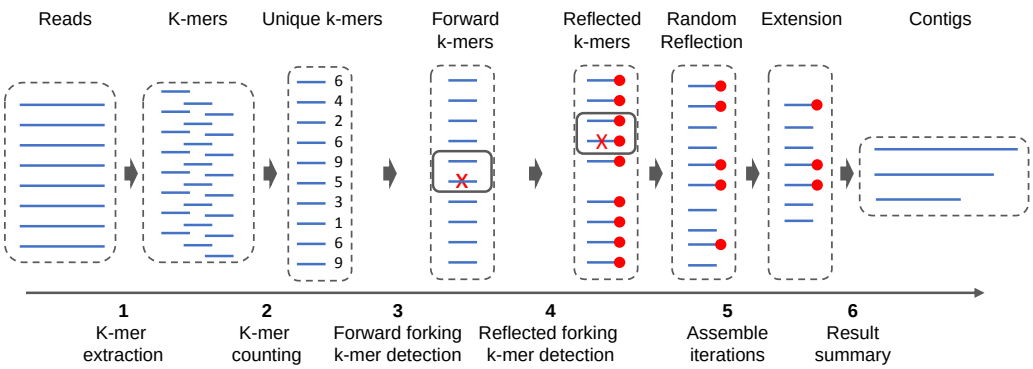

Fig. 4.13: The pipeline of the Reflexiv assembler. Blue dashes with red dots represent reflected k-mers. Step 5 iterates until convergence.

The pipeline starts with loading the sequencing data into an RDD. Then, a 'map' step is carried out to extract k-mers from the sequencing reads (Fig. 4.13). Once all k-mers are extracted from the reads, a 'reduce' function is used to count the copy number of each unique k-mer. After the k-mer counting step, a 'map' step is applied to convert the k-mers of the RDD into reflexible k-mers stored in a new RDD (as an RDK). In the newly created RDK, all reflexible k-mers are initiated as forward k-mers. The assembler sorts all the forward k-mers and looks for forward forking k-mers. After that, all forward k-mers are reflected and sorted again for searching reflected forking k-mers. When both forward and reflected forking k-mers are found, the assembler starts the recursion of the random k-mer reflecting method. Once recursion reaches convergence, the result of the assembly is summarized and the contigs are generated on the master node.

4.6 Time complexity

The time complexity of the RDK based assembly process can be calculated based on three parts that correspond to the three steps of the random k-mer reflecting method: (i) random k-mer reflecting, (ii) sorting, and (iii) extension. Random k-mer reflecting is a linear operation that goes through each element of the list and randomly reflects half of the k-mers. Thus, the time complexity of this step is *O(n)*, where *n* represents the number of k-mers in the RDK. The extension step has the same linear time complexity, *O(n)*, as it also traverses the list of k-mers and extends the adjacent k-mers.

The sorting process is implemented using the default sorting algorithm, Timsort, of the Apache Spark platform. Timsort is derived from merge sort and insertion sort. The average performance of Timsort is $O(n \times log(n))$. When we add all three parts together, the total time complexity for the first iteration is $2 \times O(n) + O(n \times log(n))$. It also can be expressed as:

$$\begin{aligned} T(n) &= 2n + n \times log(n) \\ T(n) &= n \times (2 + log(n)) \end{aligned} \tag{4.9}$$

Where *T(n)* represents the time complexity of assembling the genome from *n* number of k-mers in the RDK. As mentioned in the random k-mer reflecting method section, there is a 25% reduction of k-mers after each iteration. Thus, the time complexity for the second iteration can be expressed as:

$$T(n) = 0.75n \times (2 + log(0.75n)) \tag{4.10}$$

Let I denote the number of iterations until the recursion reaches convergence. The complete time complexity for the assembly process can be expressed as the sum of all iterations:

$$\begin{aligned} T(n) &= 0.75^0 n \times (2 + log(0.75^0 n)) \\ &+ 0.75^1 n \times (2 + log(0.75^1 n)) \\ &+ 0.75^2 n \times (2 + log(0.75^2 n)) \\ &+ 0.75^3 n \times (2 + log(0.75^3 n)) \\ &... \\ &+ 0.75^I n \times (2 + log(0.75^I n)) \end{aligned} \tag{4.11}$$

It can also be expressed as:

$$T(n) = \sum_{m=0}^{I} 0.75^m n \times (2 + log(0.75^m n)) \tag{4.12}$$

In the repeat detection and bubble popping section, I have introduced two rounds of sorting processes and one k-mer reflecting step for finding forward and reflected forking k-mers. These processes run before the random k-mer reflecting method begins. As mentioned above, the sorting step has a time complexity of $O(n \times log(n))$ and reflecting process has a time complexity of $O(n)$. Adding these computations to the run time pool, we have an aggregated time complexity of the assembly process:

$$\begin{aligned} T(n) &= 2n \times log(n) + n \\ &+ \sum_{m=0}^{I} 0.75^m n \times (2 + log(0.75^m n)) \end{aligned} \tag{4.13}$$

In addition to the repeat detection and bubble popping step, there is a k-mer counting step that generates unique k-mers with their correspond coverage for the downstream assembly process. At the very beginning of the assembly process, the assembler extracts k-mers from the raw sequencing data and counts the coverage of each k-mer. The k-mer counting in a distributed system is a typical MapReduce application, where the 'map' step extracts the k-mer and the 'reduce' step summarizes the number of each k-mer. As mentioned in the related work chapter, the 'reduce' step mainly consists of a sorting process. Thus, the final time complexity of the assembly process can be expressed as:

$$\begin{aligned} T(n) &= N + N \times log(N) \\ &+ 2n \times log(n) + n \\ &+ \sum_{m=0}^{I} 0.75^m n \times (2 + log(0.75^m n)) \end{aligned} \tag{4.14}$$

where N represents the number of k-mers extracted from the sequence data.

Although the complexity of the algorithm has a decisive effect on its time complexity on a single computer instance, the run time performance of a distributed program can be significantly affected by the network connection. Judging by the time complexity illustrated by equation 4.15, my assembler should have a low run time performance as it introduces a considerable amount of sorting processes. However, the sorting process is running in parallel and fully utilizes the bandwidth of the network. Thus, our tool has much better run time performance than other existing tools on standard Ethernet connected Spark cluster (see discussion).

4.7 Memory consumption

The memory consumption of the assembly process depends on both the number of unique k-mers and the length of the k-mers. The number of unique k-mers is effected by the complexity of the genome sequence and the sequencing quality. Whereas the length of the k-mers is usually assigned by the users. The length of the k-mer can also directly impact the number of unique k-mers. To better evaluate the memory consumption of the assembly process, let us preset the number of k-mers to n and the k-mer length to 31nt (commonly used default k-mer length).

The RDK is built on top of the Spark RDD. An RDK instance is a customized RDD instance distributed across the Spark cluster. In the Spark cluster, RDDs consume 75% (by default) of the JVM run time memory assigned to all worker nodes. However, the fraction of the memory used by RDDs can be reset accordingly (it can also be set to 100%). In the Reflexiv assembler, all the data for the assembly is stored in an RDD. Thus, the memory usage of the RDD is equal to the memory consumption of the assembly process. As mentioned before, an RDK is a long list of reflexible k-mers stored in an RDD. Each reflexible k-mer is a Java object created by the Spark JVM running on the worker nodes. Therefore, the memory consumption of an RDD is n times the size of a reflexible k-mer, where n stands for the total number of unique k-mers.

A reflexible k-mer is stored as a Java object. To be more specific, the object is a 'Tuple2' that contains 2 other objects: (i) a 'Long' object storing a fixed length (30nt with a default 31nt k-mer length) prefix or suffix (if the k-mer is reflected) of the k-mer and (ii) a 'Tuple4' object storing extra information for the reflexible k-mer.

For the first object of the Tuple2: since there are four types of nucleotides in a DNA sequence, each nucleotide can be encoded into two binary digits (bits). Thus, a 31nt k-mer can be encoded into 62 bits, which can be represented by a 'Long' object.

For the second object of the Tuple2: as illustrated by the name, a Tuple4 contains 4 objects: (i) a 'Boolean' object as a reflecting marker indicating whether the k-mer has been reflected or not, (ii) an 'Integer' as a marker for the forward extendable region, (iii) an 'Integer' as a marker for the reflected extendable region, and (iv) a 'Long' object to store the extended part of the k-mer (Fig. 4.14) (see discussion).

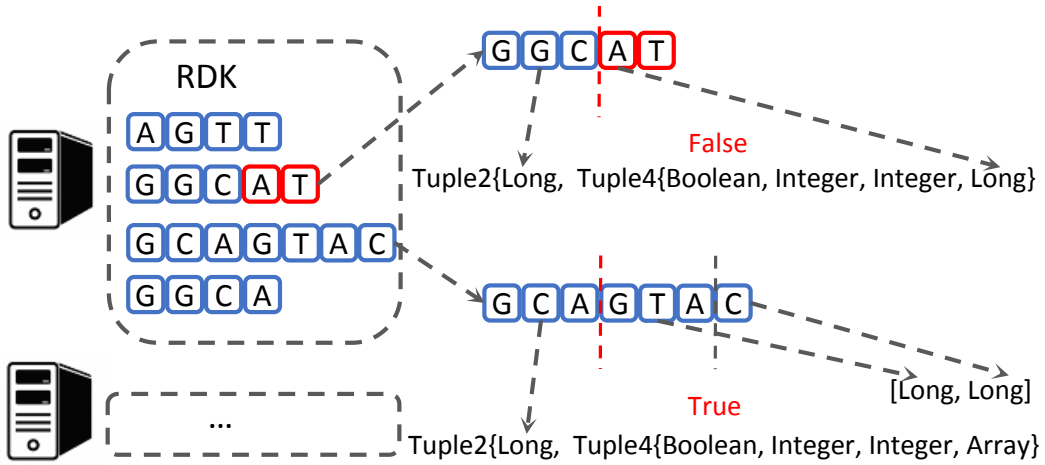

Fig. 4.14: Basic data structures of two reflexible k-mers in an RDK. For a reflected k-mer (denoted with red boxes) shorter than *2n* nucleotides, a Long object is used to store the extended suffix. For a forward k-mer longer *2n* nucleotides, an Array of Long objects is used to stored the extended suffix.

In the Java programming language, each object is stored in the JVM Heap memory and its reference is stored in the Stack memory. An object has a memory overhead that stores information related to the object (called 'housekeeping' information) and a value that is usually represented by a primitive type. For instance, an 'Integer' object has a value of an 'int' numeric primitive type that stores a number ranging from -2^{31} to $2^{31}-1$. Since the JVM allocates the memory in multiples of 8 bytes (64-bit JDK), the value of the object is rounded up to multiples of 8 bytes (also called padding). As the size of the 'int' primitive type is 4 bytes, the value of the object is padded to 8 bytes. In addition to the size of the value stored in the object, each object has an 12 bytes padding or overhead in a 64-bit JDK. Thus, the memory consumption of an 'Integer' object is 20 bytes. Based on the example, we can also calculate the size of the 'Long' object (20 bytes) and the 'Boolean' object (20 bytes). Adding up all the objects, a 'Tuple4' object consumes 92 bytes of memory. Whereas a 'Tuple2' object consumes 124 bytes of memory. As a result, the total memory consumption of the assembly process can be expressed as:

$$M(n) = 124 \times n \tag{4.15}$$

where M represents the memory size and n represents the number of unique k-mers.

4.8 Results and Discussion

4.8.1 Results

In this section, I present the run time performance and the assembly qualities of the Reflexiv assembler. I will also compare its run times and assembly qualities to other distributed assemblers, e.g. Ray and AbySS. I have used three different datasets for the benchmarking: (i) 500MB simulated Illumina Hiseq-2500 sequencing dataset based on the *E. coli* reference genome, (ii) 10GB simulated Illumina Hiseq-2500 sequencing dataset based on the chromosome 17 of the human genome, and (iii) a real Illumina MiSeq sequencing data of the *E. coli* genome (NCBI accesion number: PRJDB5271). Computing clusters were the de.NBI cloud. I have setup clusters with 1 master node and 1 to 20 worker nodes. The master node has 32 CPUs and 60GB of RAM. Each worker node has 28 CPUs and 60GB of RAM.

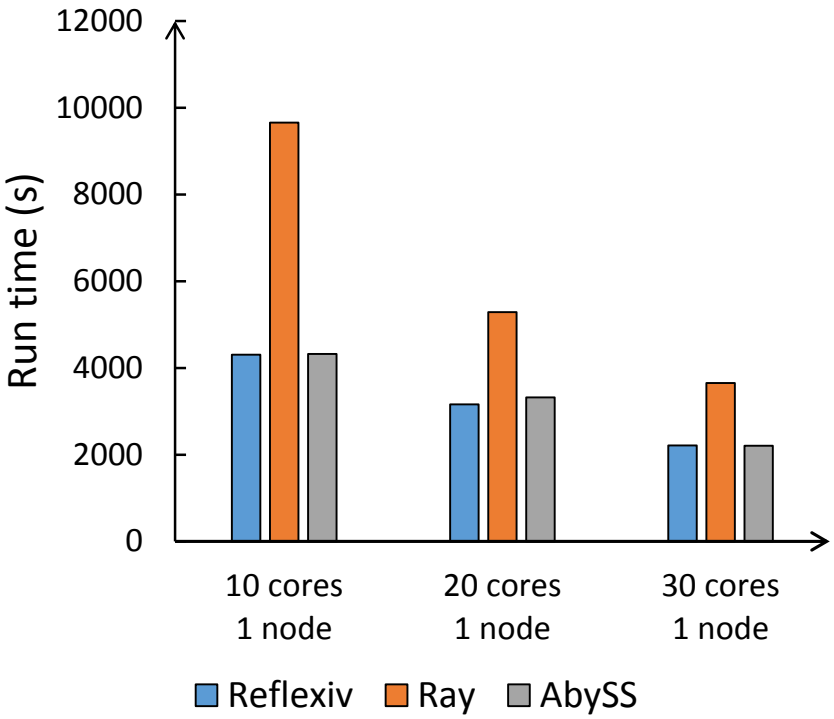

Fig. 4.15: Comparison of run time performances between different distributed *de novo* genome assemblers. The comparison was carried out on a single computer instance using 10, 20, and 30 CPUs. The 10GB simulated sequencing data of the human chromosome 17 was used for the benchmark. Detailed metrics can be found in Appendix Table S16.

For run time performances, I first compared the run times of all tools on one single computer instance (the master node) with 10, 20, and 30 CPUs (Fig. 4.15). Reflexiv runs slightly faster than AbySS on the 10GB simulated sequencing data

from the human chromosome 17. Both AbySS and Reflexiv run faster than the Ray assembler.

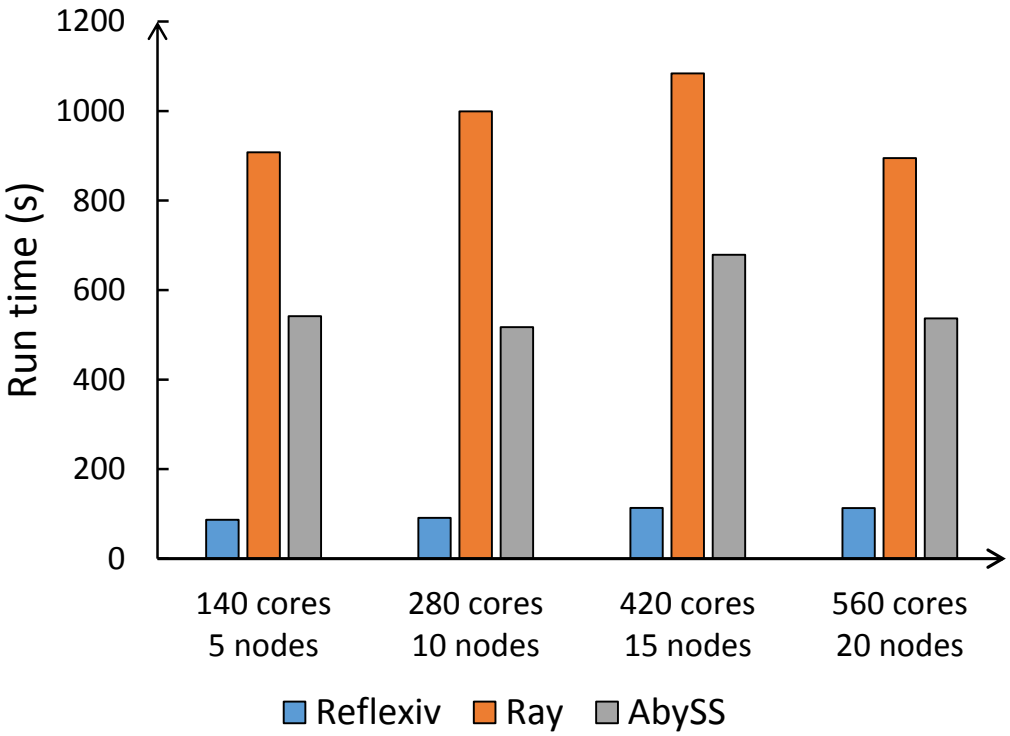

Fig. 4.16: Comparison of run time performances between different distributed *de novo* genome assemblers. The comparison was carried out on 5 to 20 worker nodes with 140 to 560 CPUs. The 10GB simulated sequencing data of the human chromosome 17 was used for the benchmark. Detailed metrics can be found in Appendix Table S17.

When scaling out to more than 5 worker nodes, Reflexiv runs much faster than both AbySS and Ray. For the 10GB simulated dataset of the human chromosome 17, Reflexiv runs 8-17 times faster than Ray and 7-18 times faster than AbySS (Fig. 4.16). Whereas for the 1.3GB read sequencing data of the *E. coli* genome, Reflexiv runs 7-10 times faster than Ray and 5 to 6 times faster than AbySS (Fig. 4.17).

To compare the assembly qualities between different *de novo* genome assemblers, I have used QUAST, a quality assessment tool for genome assemblies (Gurevich et al., 2013). In general, Reflexiv has similar assembly qualities to other distributed tools, e.g. Ray and AbySS. Velvet has slightly higher N50 values and assembles longer contigs. But it has more misassemblies. For the 500MB simulated dataset (Table 4.1), Reflexiv has similar quality to AbySS. Both Reflexiv and AbySS assemble slightly better than Ray. Whereas for the 1.3GB real sequencing dataset of the *E. coli* genome (Table 4.2), Reflexiv assembles longer contigs than both Ray and AbySS. For a larger dataset (10GB simulated sequencing data of the human chromosome 17), Reflexiv has similar performance to both Ray and AbySS

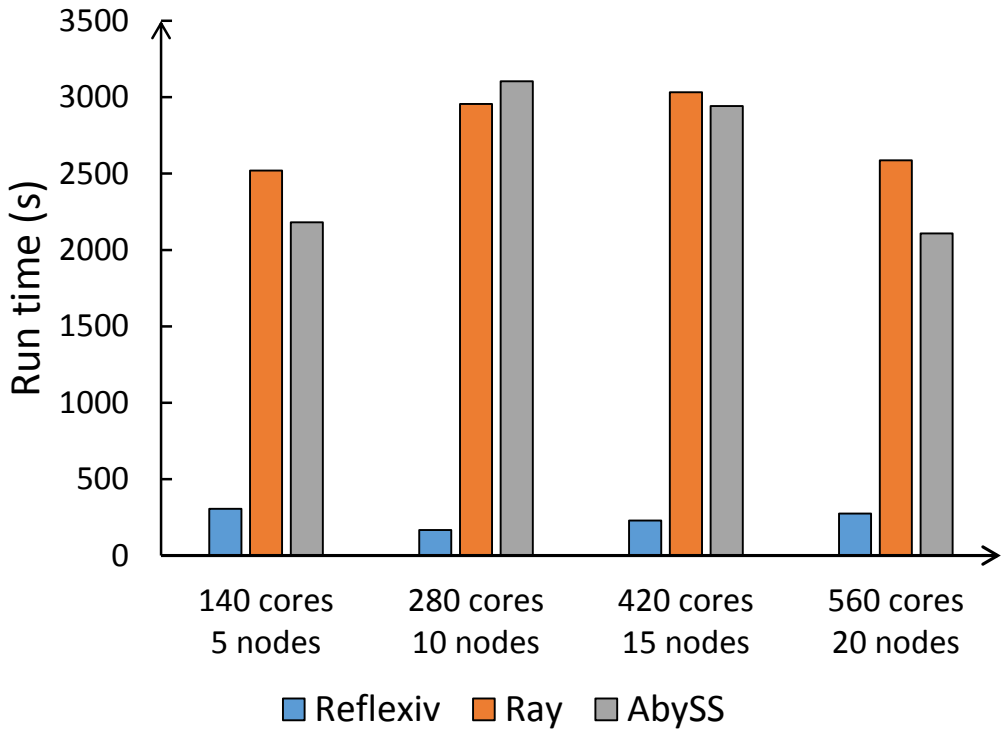

Fig. 4.17: Comparison of run time performances between different distributed *de novo* genome assemblers. The comparison was carried out on 5 to 20 worker nodes with 140 to 560 CPUs. The 1.3GB real sequencing data of the *E. coli* genome was used for the benchmark. Detailed metrics can be found in Appendix Table S18.

4.8.2 Discussion

In this chapter, I have presented a distributed *de novo* genome assembler called Reflexiv. The main innovation of the Reflexiv assembler is a new distributed data structure called Reflexible Distributed K-mer (RDK). The RDK is a higher level abstraction of the Spark RDD. It uses the Spark RDD to distribute large amounts of reflexible k-mers across the Spark cluster and assembles the genome in parallel. I have described how the random k-mer reflecting method retrieves the adjacencies between overlapping k-mers. I have also described how to solve repeats in the

Tab. 4.1: Comparison of the assembly qualities between different tools. The assemblies are carried out on a 500MB (50x) simulated dataset of an *E. coli* genome.

Tools	Reflexiv	Ray	AbySS	Velvet
Largest contig	127972	86660	127976	138261
N50	22171	21604	22173	33905
Misassemblies	0	0	0	8
Misassembled contigs	0	0	0	7
Misassembled contigs length	0	0	0	169224
Genome fraction (%)	97.174	96.058	97.195	97.562

Tab. 4.2: Comparison of the assembly qualities between different tools. The assemblies are carried out on a 1.3GB real sequencing dataset of an *E. coli* genome.

Tools	Reflexiv	Ray	AbySS
Largest contig	126555	85688	74508
N50	17892	16198	15124
Misassemblies	36	35	36
Misassembled contigs	31	30	31
Misassembled contigs length	911898	686404	686340
Genome fraction (%)	84.209	79.154	83.932

Tab. 4.3: Comparison of the assembly qualities between different tools. The assemblies are carried out on a 10GB (50x) simulated dataset of the chromosome 17 of the human genome.

Tools	Reflexiv	Ray	AbySS	Velvet
Largest contig	22610	22582	22612	70151
N50	2207	2303	2202	2858
Misassemblies	2	3	0	4212
Misassembled contigs	2	3	0	1857
Misassembled contigs length	3722	3843	0	8823201
Genome fraction (%)	62.13	65.205	62.919	67.459

genome and pop bubbles during the assembly. In addition, I have presented a formula to accurately measure the memory consumption of the assembly process.

In the results section, I have carried out a series of benchmarks on the Reflexiv assembler. My tool has excellent run time performances on the ethernet connected Spark cluster. Compared to existing tools, Reflexiv is the fastest to complete the assembly of the E. Coli genome and the chromosome 17 of the human genome. Moreover, Reflexiv is the only tool that is able to scale on an ethernet connected cluster. As for the assembly quality, Reflexiv has similar performance to both Ray and Abyss. Although Velvet assembles longer contigs than the other tools, it has more mis-assembled contigs in its result.

An RDK based assembler has three advantages compared to the MPI-based assembler: (i) the random k-mer reflecting method makes the k-mer extension step highly scalable. (ii) The sorting process fully utilizes the network connection and (iii) constantly balance the workload of each task.

Highly scalable means that the extension task can be divided into as many partitions as possible and the divided tasks can be simultaneously carried out throughout the entire cluster. An RDK is a long list of reflexible k-mers. In the list, each k-mer and its overlapping k-mer can be extended independently once their adjacency is found. The random k-mer reflecting method arbitrarily reflects the k-mers and reestablishes the adjacencies of overlapping k-mers. As the list can be easily divided, the adjacencies of

overlapped k-mers can be found simultaneously in each divided sub list. In addition, dividing a list of k-mers can be easily carried out and the proportion of each sub list can be managed based on the demand of the parallelization. Thus, the workloads of processing different sub lists of the k-mers can be easily balanced.

In each iteration of the random k-mer reflecting method, the sorting process shuffles the k-mers and re-distributes them evenly across the Spark cluster. Compared to the constant message passing in the MPI based assemblers, the shuffling process of Reflexiv is carried out to all the k-mers at the same time. On an ethernet connected cluster, there is a latency overhead for each data transmission. For a constant messaging process, the latency will create a significant overhead. Although the bottleneck can be solved by using low latency Infiniband network, most of the general purpose computing clusters are still using the economical ethernet network. Reflexiv assembler, on the other hand, does not suffer from the high latency overhead. Thus, it can be easily portable to different distributed system.

The current implementation of the reflexible k-mer uses a 'Long' object to encode the k-mers. As a nucleotide is stored in 2 bits, a k-mer longer than 31nt does not fit into a 'Long' object anymore. Therefore, the current implementation of the Reflexiv assembler has a 31nt limit for the k-mer length. I will upgrade the implementation to allow longer k-mers in the upcoming release (see future work). As mentioned in the memory consumption section, the extended part of the k-mer is stored as a 'Long' object inside a 'Tuple4' object. After 6 iterations of the assembly process, the extended part of the k-mer is switched and stored in an 'Array' object containing an array of 'Long' objects, as the extended part of the k-mer can be longer than 32nt (a nucleotide sequence longer than 32nt can not be encoded into a 'Long' object). However, after 6 iterations of the assembly process, the total number of the elements is reduced to $0.75^6 \times n$. Although an 'Array' object produces extra memory overhead, the total number of k-mers after 6 iterations of the assembly process is significantly lower.

The main focus of the Reflexiv assembler is to address the memory intensive challenge in the *de novo* genome assembly. Thus, the current version of the assembly pipeline is only implemented to assemble the contigs. For the scaffolding phase, the memory consumption is not as intensive as the contig assembly phase. Moreover, there is a collection of bioinformatics tools specialized to assemble scaffolds based on the pre-assembled contigs. These assemblers are both run time and memory efficient in assembling scaffolds (Yeo et al., 2018). Most of these tools use the paired sequencing reads to build up scaffolds. Since my tool already has a distributed read mapping function, I incorporate the function of the assembler and discuss the implementation of a scaffolding pipeline (see Chapter 6).

5

Large scale genomic data analyses

> *Big data is not about the data, the real value is in the analytics.*
>
> **— Gary King**
> Professor of Harvard University

In this chapter, I will present an use case on the Amazon cloud for analyzing large amounts of genomic dataset. In this cloud application, I have used my framework to analyze a collection of 100 TB of genomic data from 3 genome projects and a transcriptomics study (Wyatt et al., 2014). The entire process was completed in 21 hours, which included cluster deployment, data downloading, decompression and various data analyses. Based on the analytical results, I have also carried out a functional analysis to associate large scale public data with a private dataset.

The main focus of the application is to present: (i) an use case for users to easily access and analyze large amounts of public data on the cloud, (ii) the scalability of my framework on the powerful computing cloud, and (iii) a proof of concept functional analysis to bring additional biological insights from cloud hosted public data into private studies.

I will start by introducing how to deploy a Spark cluster on the cloud and the configurations of the Spark cluster. Then, I will present the genomic dataset employed in this use case. I will also introduce parallel data downloading and decompression methods used in this study.

In the result sections, I will present the run times of various analyses on the Amazon cloud. I will also present a functional study based on the fragment recruitment profile of the entire HMP data.

5.1 Cluster deployment and configuration

All analyses on the Amazon AWS EC2 cloud were carried out on Spark clusters that consist of one master node deployed on an `m1.xlarge` computer instance and 50 to 100 worker nodes deployed on the `c3.8xlarge` compute instances (see table 3.1).

The Spark cluster on the Amazon Elastic Compute Cloud (EC2) was deployed using (i) Spark-ec2, (ii) BiBiGrid, and (iii) Amazon Elastic MapReduce (EMR). For Spark-ec2, it launches a selected number of computing instances on EC2 using the AWS auto scaling function. It can also request a spot price when launching computing instances. A Linux system is deployed on all instances using the Amazon Machine Image (AMI) `ami-2ae0165d`. Once all instances are alive (once the Linux system deployment is completed), the Spark package is downloaded from the online repository to the master instance. Next, the package is copied to each worker instance and installed simultaneously. After that, a master Java virtual machine (JVM) daemon program is launched on the master node, followed by launching worker daemons on all worker nodes that connect to the master node. The entire cluster can be deployed or shutdown with a single command. BiBiGrid uses a customized image that has a pre-installed Spark framework in its operating system. Once the worker instances come alive, the Spark worker daemon program can be launched directly without downloading and installing Spark on the instances. EMR is a built-in module of the AWS cloud. It is optimized by AWS to deploy a Spark and Hadoop cluster in a short time. However, cost is needed for such service.

I have requested 60GB (Spark allocated 57.6 GB) RAM for each worker node on c3.8xlarge instances. By default, 75% of the Java heap space was allocated for Spark's RDD memory cache. A Hadoop distributed file system (HDFS) was setup on all worker nodes (named "Data nodes" by Hadoop) with 3 times of data redundancy (as default). Thus, the maximum data size for HDFS storage is 210 GB per node. On the HDFS, data is split into chunks and distributed across the data nodes with a default size of 128 MB per chunk (see table 3.1).

The standard price for the `c3.8xlarge` was $1.680 per hour, per instance. However, I have used the bidding system of the AWS cloud with spot prices between $0.17 and $0.20 per hour, per instance for the `c3.4xlarge` and between $0.35 and $0.40 per hour, per instance for the `c3.8xlarge` in the AWS Ireland region (see table 3.2). The bidding system massively reduced our costs.

5.2 Data storage and accessibility

The public datasets used in the analyses are selected from four genomic projects: (i) the Human Microbiome genome project, (ii) the 1000 Genome Project, (iii) the 3000 Rice Genome Project, and (iv) a prostate transcriptome project from (Wyatt et al., 2014). The size of all the NGS data is 26TB as compressed files. After decompression, the total size of the data is 100TB.

I have used the entire whole genome sequencing (WGS) data (metagenomics) of the HMP project hosted on Amazon S3. These WGS data were sampled from 6 body sites and 15 sub body sites. In total, there are 2.3 TB compressed (8.6 TB uncompressed, Fig. 5.1) fastq files in the bzip2 format. All HMP datasets are hosted in the Oregon region of the Amazon S3.

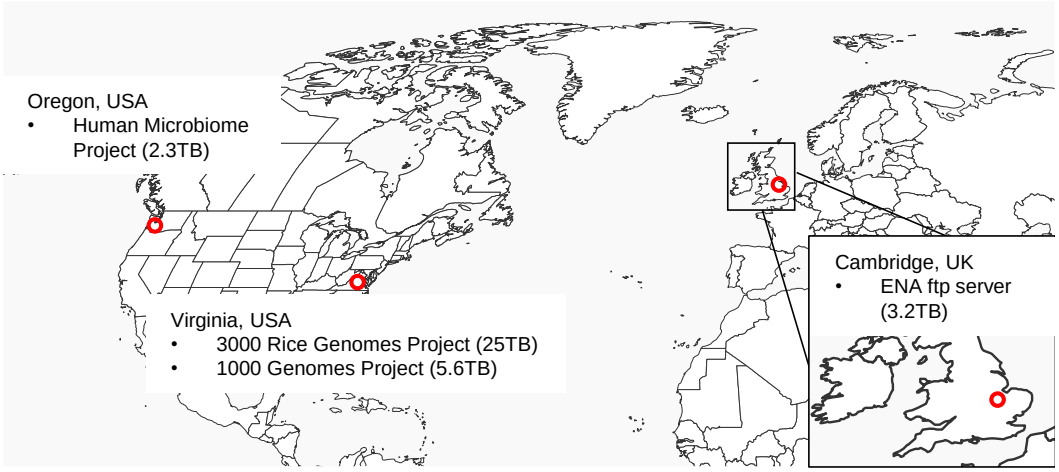

Fig. 5.1: Fast access to genomic data on public repositories. Data sets of the Human Microbiome Project, the 3000 Rice Genome Project and the 1000 Genomes Project are hosted in different regions on Amazon S3. Whereas the RNA-seq data of a prostate cancer transcriptomic study is stored on the ENA ftp server.

The WGS data of the 1000 Genomes Project was used for intensive performance evaluations on a EC2 cluster (100 nodes, Fig. 5.1). 5.6 TB (compressed) fastq files, sequenced from 106 samples, were mapped to the human reference genome (version GRCh38, hg19) using Sparkhit invoked BWA. All datasets are hosted at the Amazon S3 (Virginia region).

To benchmark the performance of genotyping on the Amazon EC2 cloud, I used 15 TB of BAM files (Fig. 5.1) that were already mapped to the *Oryza sativa L. 93-11* reference genome. The average sequencing depth is around 14 folds and the parameter for the pileup algorithm (implemented by Samtools mpileup) was set accordingly. All datasets are hosted at Amazon S3 (Virginia region).

In contrast to the other genome projects in this study, the RNA-seq data of the prostate transcriptome project is hosted on the ftp server of the Europe Nucleotide Archive (ENA) uploaded by a previous study (Wyatt et al., 2014). I used all sequencing data for gene expression profiling (Fig. 5.1). All datasets can be found with the study accession PRJEB6530 from the European Nucleotide Archive (ENA).

5.3 Distributed data downloading and decompression

Downloading and decompressing large genomic files are significant bottlenecks before the actual data analysis begins. Spark's architecture can greatly benefit from high-performance networks during large data transfers. In particular, when commencing a download task, files can be split into chunks and transferred from distributed storage system, such as Amazon simple storage service (S3), to each worker node. This parallel transfer method fully utilizes the high network bandwidth of a distributed cluster (Fig. 5.2).

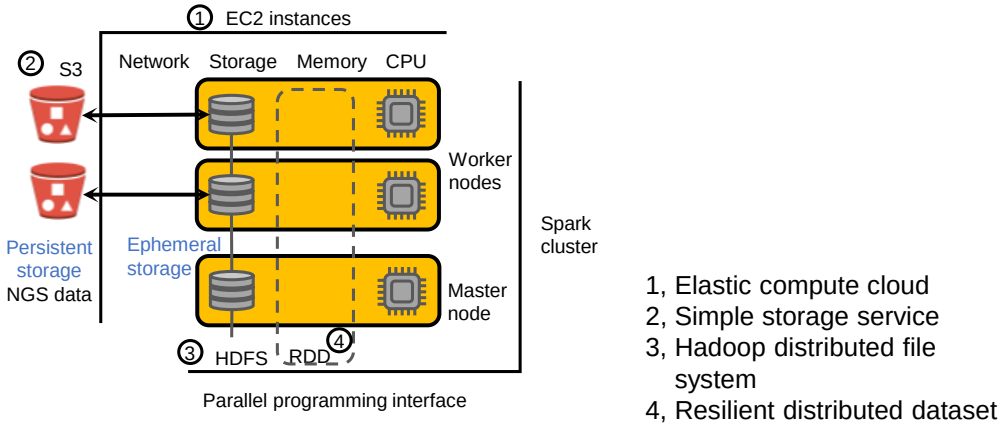

Fig. 5.2: The architecture of a Spark cluster deployed on the Amazon cloud. The yellow boxes represent Amazon EC2 instances that are virtualized into Spark master/worker nodes.

When downloading data from Amazon S3 to HDFS, I have used Hadoop Distcp (distributed copy), a tool designed for large inter-cluster copying. For downloading data from Amazon S3 to a shared network file system (NFS), I used a Java-based tool developed in our research group called BiBiS3 (https://wiki.cebitec.uni-bielefeld.de/bibiserv/index.php/BiBiS3). BiBiS3 not only parallelizes downloading jobs to multiple computer nodes, but also applies multi-thread downloading on each computer node to fully exploit the capacities of every computer's network connection.

`Distcp` is a Hadoop MapReduce program that launches a certain number of parallel 'mapper' processes on each worker node (known as slave node in a Hadoop cluster). Each 'mapper' invokes a download process in the 'map' step of the MapReduce job. The download process transfers input files to HDFS and stores them in small chunks (default is 128 MB per chunk).

In the case of downloading data from the Amazon S3 to a shared network file system (NFS), I have used BiBiS3 (Henke, 2017). It is designed to exploit the full network capacity between the cloud instances and Amazon S3 by applying multi-threaded download on each node. It is capable of downloading different chunks of the same data to an arbitrary number of machines simultaneously, so that maximum network bandwidth or disk I/O limit can be reached. When starting a BiBiS3 download job, it utilizes the Amazon S3 REST API to send a GET request on each chunk of the object, which in this case is a genomic data file. All chunks of data are then copied to the NFS file system. Moreover, BiBiS3 is also able to download a directory of files recursively.

Most genomic datasets are compressed and stored on public repositories. To directly access and analyze the compressed data, I also improved the performance on data preprocessing by introducing parallel decompression after downloading the data. I developed a parallel decompression tool, Sparkhit-spadoop, that is built on top of the Apache Hadoop (Spark's parallel decompression has a thread safety issue at 2.0.0 version) and included it in our toolkit (See chapter-3: parallel data preprocessing).

5.4 Rapid NGS data analyses on the AWS cloud

To demonstrate that large scale genomic analyses on the cloud can be easily accessed by my tools, I tested the framework on 100 TB (26 TB compressed) of genomic data from 3 genome projects and a transcriptomics study on the Amazon EC2 cloud (Fig. 5.3). This data was compressed and stored in different regions around the world on Amazon S3 and the European Nucleotide Archive (ENA) ftp server (Fig. 5.3F). I rented 100 c3.8xlarge Amazon EC2 instances (3200 cores, 6TB memory and 60TB disk space in total; see table 3.1) with a total spot price of 38 USD per hour. Sparkhit completed the entire process, including cluster deployment, data downloading, decompression and various data mining, in 21 hours (the entire duration that the cloud provider charges) with a total cost less than 800 USD (Fig. 5.3A-E).

I started deploying a Spark cluster of 100 worker nodes in Ireland region using Spark-ec2 script, which is the slowest one among three cluster deployment tools (the worst scenario for users' cloud budget). This step took 39 minutes and 54 seconds on the entire run time clock (Fig. 5.3A).

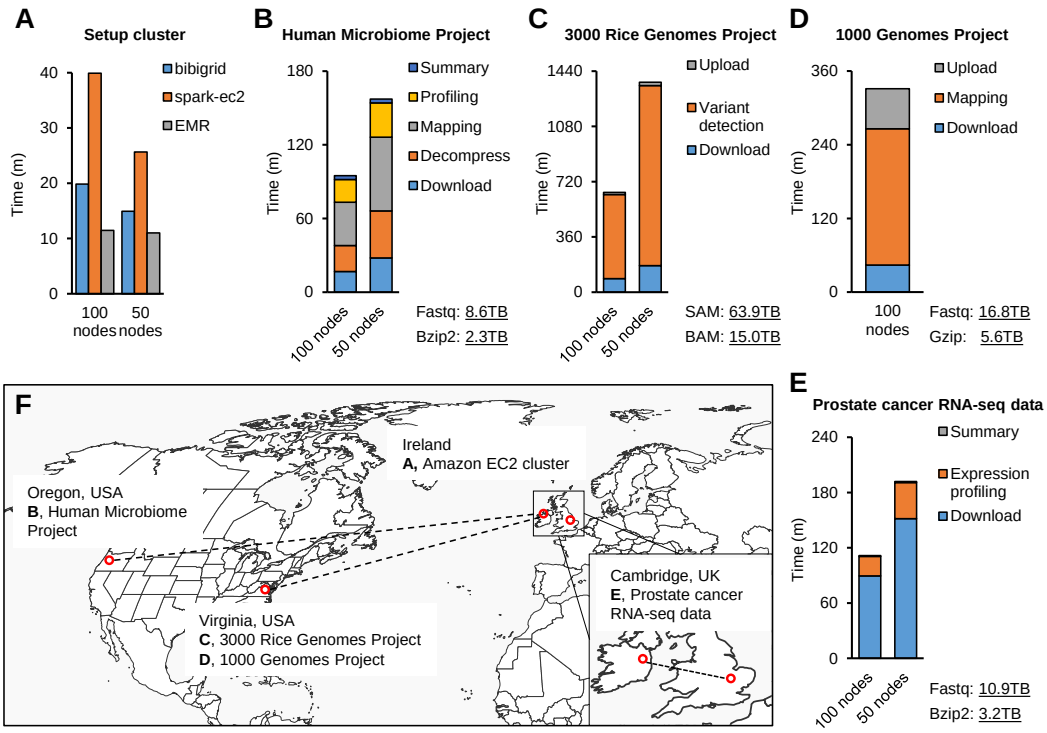

Fig. 5.3: Large scale genomic data analyses on the cloud: **(A)** Run time comparison between three auto-scaling tools for deploying a Spark cluster on the Amazon EC2 cloud. Durations include pending for approval of EC2 spot request and waiting for SSH connection to each EC2 instance. EMR, Amazon Elastic MapReduce service. **(B)** Run times for processing all WGS data from the Human Microbiome Project. Mapping was carried out using Sparkhit-recruiter while profiling was carried out using Sparkhit invoked Kraken. **(C)** Run times for processing 15 TB BAM files of the 3000 Rice Genome Project. I uploaded the variant calling result to Amazon S3. **(D)** Run times for processing 5.6 TB compressed sequencing data. Mapping was carried out using Sparkhit invoked BWA aligner. I uploaded the SAM files to Amazon S3. **(E)** Run times for processing 3.2 TB RNA-seq data. Gene expression profiling is carried out using Sparkhit invoked Kallisto. **(F)** Fast access to genomic data on public repositories. Data sets of the Human Microbiome Project, the 3000 Rice Genome Project and the 1000 Genomes Project are hosted in different regions on Amazon S3. Whereas the RNA-seq data of a prostate cancer transcriptomic study is stored on the ENA ftp server.

5.4.1 Processing all WGS data of the Human Microbiome Project

The Human Microbiome Project hosts 2.3 TB of compressed metagenomic whole genome shotgun (WGS) data on Amazon S3 located in the Oregon region. The data enables comprehensive characterization of the human microbiome and serves as a metagenomic database for microbiome studies. To associate these public datasets with microbial reference genomes, I downloaded all WGS data to the Spark cluster located in the Ireland region. After decompression, I recruited all sequencing reads to a collection of 21 microbial reference genomes (72 MB total, defined as Ref-2) and summarized the fragment recruitment results. I also profiled the taxonomy

abundance by using Sparkhit invoked Kraken (Wood and Salzberg, 2014). All processes were completed in 1 h 34 m (Fig. 5.3B and Appendix table S4).

5.4.2 Genotyping on 3000 samples of the 3000 Rice Genomes Project

The 3000 Rice Genomes Project hosts 200 TB of public data on Amazon S3 (Virginia region). This data enables large scale discovery of novel alleles for important rice phenotypes. Variant detection is a particularly expensive step in analyzing whole genome or exome sequencing data. To test the run time performance of Sparkhit on variant detection, I downloaded 15 TB BAM files to the Spark cluster. By using Sparkhit invoking Samtools-Mpileup (Li et al., 2009), I genotyped 3000 rice samples and uploaded detected variants to Amazon S3. This analysis took 10 h 49 m (Fig. 5.3C and Appendix table S5).

5.4.3 Mapping 106 samples of the 1000 Genomes Project

The 1000 Genomes Project hosts all WGS data on Amazon S3 (Virginia region). It provides a detailed catalogue of human genetic variants in the studied populations. Before variant detection, a mapping step is required to present all matches and mismatches on the reference genome. To test the run time performance of Sparkhit on whole genome sequence mapping, I downloaded 5.6 TB compressed sequencing data to the Spark cluster. Using Sparkhit invoking BWA (Li and Durbin, 2009), I mapped 106 samples to a human reference genome. After sequence mapping, all results (SAM format) were uploaded to Amazon S3 for persistent storage. The entire process was completed in 5 h 31 m (Fig. 5.3D and Appendix table S6).

5.4.4 Gene expression profiling on prostate cancer RNA-seq data

Profiling tumor specific gene expression is a critical step for cancer research. To enable fast transcriptome quantification on the cloud, I tested the run time performance of Sparkhit on profiling 3.2 TB compressed RNA-seq data of a prostate cancer transcriptomics study (Wyatt et al., 2014). Since all datasets are archived on the ENA ftp server (see *Data storage and accessibility* section), the downloading process consumed 1h 29m, whereas gene expression profiling with Sparkhit invoked Kallisto (Bray et al., 2016) was completed in 21 m 9 s (Fig. 5.3E and Appendix table S7). After uploading the final result to Amazon S3, all EC2 instances are manually terminated.

5.5 Metagenomic profiling and functional analysis

Metagenomics can capture and obtain genome fragments of uncultivated microbes (the microbial dark matter) by applying shotgun sequencing to aggregated microorganisms sampled directly from the environment (Rinke et al., 2013). The Human Microbiome Project consortium utilizes metagenomic shotgun sequencing to characterize microbial communities at different human body sites. As all HMP data is hosted on Amazon S3, we can directly access and analyze these metagenomic datasets on the Amazon cloud. Here, I present an use case to rapidly associate and analyze public HMP data with private dataset on the cloud.

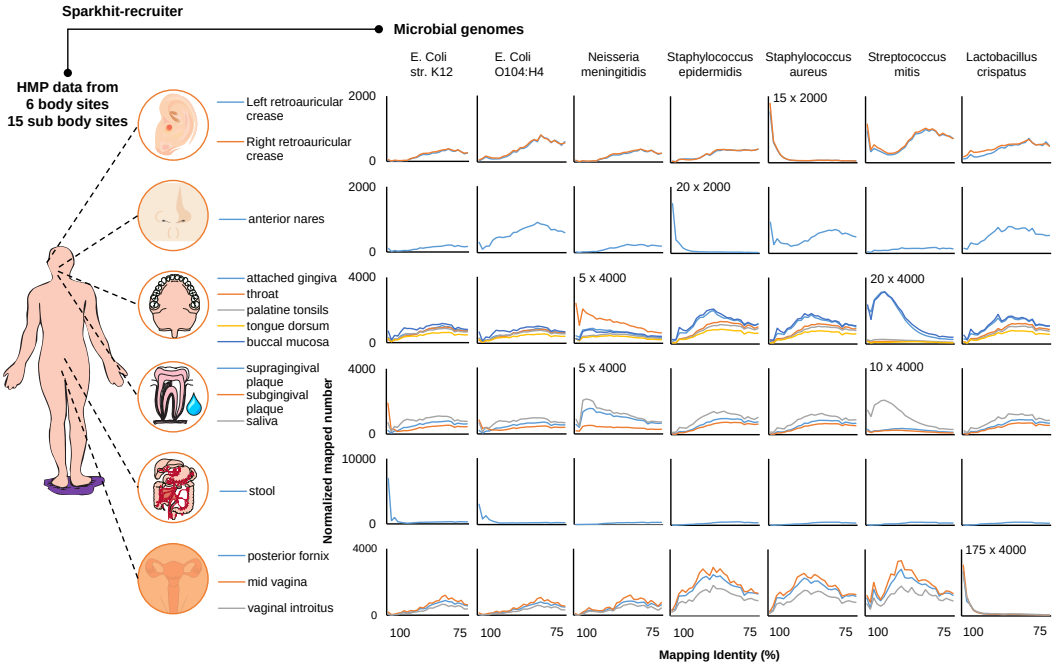

Fig. 5.4: Fragment recruitment profiles of different microbes at different sub body sites: Sparkhit-recruiter was used to map the entire HMP whole genome sequencing data to seven selected microbial genomes. For each line chart, normalized numbers of mapped reads are illustrated along different mapping identities from 75% to 100% with 1 percent increment. All line charts in the same row have the same scale indicated on the left, unless they are additionally annotated.

I started the use case by deploying a Spark cluster on the Cloud. Once the Spark cluster is deployed (see the "cluster deployment and configuration" section), all WGS data of the HMP is downloaded to HDFS in parallel using Hadoop-Distcp with maximum mapper number (parallel download processers) equal to the maximum number of available compute cores, which in our case is 3,200 cores from 100 `c3.8xlarge` instances. Then, the downloaded files of all samples (files of each sub-body site) are decompressed in parallel using our Sparkhit tool `Sparkhit-spadoop`. I mapped the sequence data of each sample to a reference genome containing 7 selected microbial genomes (Ref-1) using Sparkhit-recruiter and summarized the

recruitment result using Sparkhit-reporter. The abundance profile of each microbe is presented across different sub-body sites (Fig. 5.4). Considering that the sequencing depth varies between different sub-body sites and the genome size changes from one genome to another, the abundances are normalized by leveling the sequencing depth to 1 billion sequence reads and the genome size to 5 million bases. The normalization can be expressed as:

$$A = n \times (\frac{5000000}{L}) \times (\frac{1000000000}{N}) \tag{5.1}$$

Where *A* is the normalized abundance, *n* is the number of reads recruited, *L* is the length of a reference microbial genome and *N* is the total number of sequencing data in the sub-body site.

To demonstrate that associating HMP metagenomic data with private datasets (e. g. cultured microbial genomes or uncultured single cell genomes) can provide more biological insights, I downloaded and recruited all HMP whole genome shotgun data (2.3 TB compressed) to 12 selected microbial genomes (Ref-1, see table 3.3). Based on the metadata of the HMP samples, I present the fragment recruitment profile of 7 different microbial genomes across 6 different body sites (15 sub-body sites. Fig. 5.4). The abundance of each microbial genome was normalized and illustrated across different mapping identities (from 75 percent to 100 percent). In general, sub body sites of the same main body site have similar profiles, but with few exceptions. For instance, the abundances of *Neisseria meningitides* are different between saliva and gingival plaque from the oral body site. *Streptococcus aureus* has higher abundance in buccal mucosa, attached gingivae and saliva compared to other sub body sites in the oral body site. By changing the input reference genomes, users can easily produce the abundance profile of other microbes (see discussion).

Mapping metagenomic sequences to a new strain could reveal potential structural variances. A common method for functional studies is based on the hypothesis that such structural variances can have functional impact on the gene level. When recruiting HMP data to the reference genomes, the recruitment result was sensitive enough to pick up structural differences between genome sequences of two *E. coli* strains (Fig. 5.5A and 5.5B). Then, I followed up with a functional analysis (gene prediction and pathway enrichment) based on the sequences. I extracted sequence segments with lower read coverage from the reference genome (Fig. 5.5B). I used Prodigal (Hyatt et al., 2010) to predict genes on these sequence segments and annotated the predicted genes by mapping the peptide sequences of the genes to the NCBI non-redundant (nr) database using Blastp (Altschul et al., 1990a). The annotated genes were, then, sent to the KEGG (Kanehisa et al., 2014) database for pathway annotation. The pathway enrichment result was calculated using the hypergeometric test.

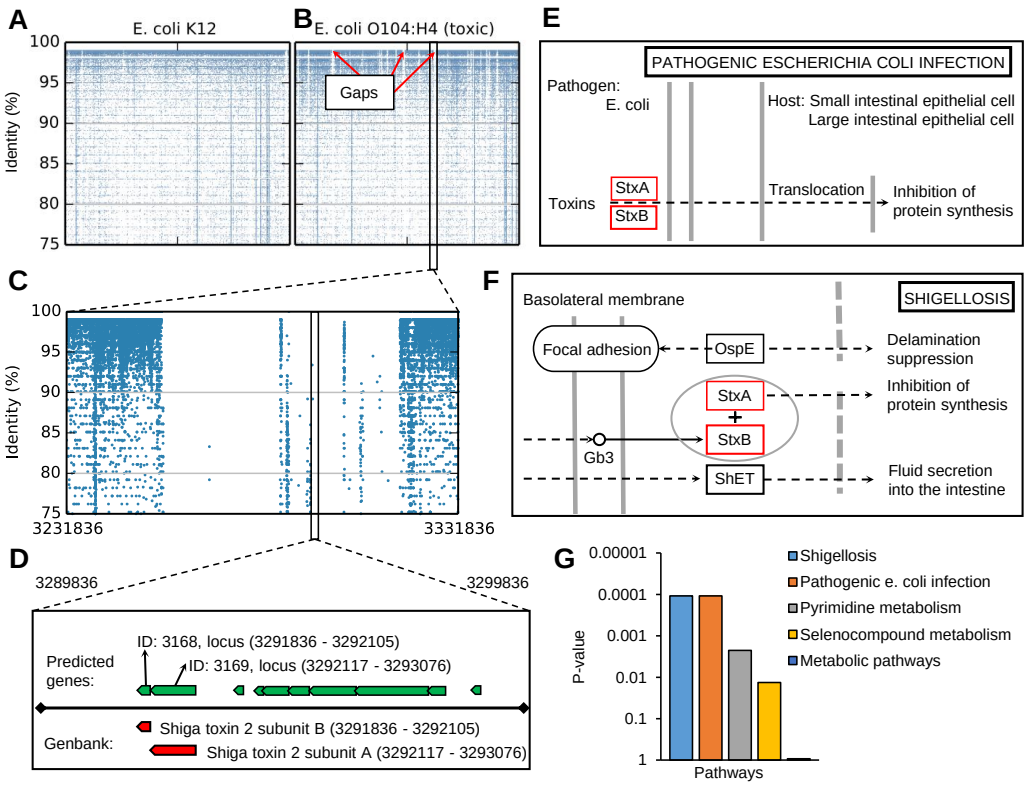

Fig. 5.5: Functional analyses on a toxic E. Coli. strain: **(A, B)** Distribution of recruited reads along the entire chromosomes of two E. Coli. strains. Reads that mapped to contigs and plasmid sequences are not included. **(C)** Enlarged fragment recruitment gap on the genomic sequence of the E. Coli. O104:H4 strain. **(D)** Predicted genes and their loci in the gap region. **(E, F)** Two enriched pathways: pathogenic Escherichia Coli infection and Shigellosis. **(G)** P-value of each annotated pathway. Pathway enrichment test was carried out using all predicted genes from all the gap regions.

I have predicted 158 new genes which are enriched in two pathogenic pathways: Shigellosis and Pathogenic *E. coli* infection (Fig. 5.5E-G). I annotated all genes and identified two toxic genes (Shiga toxin 2 subunit A and B) present in the *E. coli* strain that caused the 2011 German *E. coli* outbreak (Rasko et al., 2011).

5.6 Discussion

In this chapter, I have presented an use case of analyzing large amounts of public genomic data on the Amazon cloud. In total, my framework processed 100 TB of genomic data on a 100-node Spark cluster in 21 hours. The entire use case provided a thorough instruction on how to easily setup a cluster, download all datasets, and rapidly analyze the data with my framework. Furthermore, I have processed the entire HMP sequencing data in 2 hours, presented a proof of concept association study between public 'big data' and private datasets.

Spark and Hadoop based bioinformatics tools are not widely used in genomic studies as they require users to comprehend certain amount of knowledge on cloud computing. Therefore, I particularly focus on providing a simple and comprehensive cloud application to directly access and analyze public genomic datasets. I described a simple way to setup a Spark cluster on the Amazon cloud with one line of command. I also facilitated downloading and preprocessing large amounts of data by leveraging distributed Amazon S3 storage and optimizing parallel data decompression. In addition, Sparkhit enables users to parallelize their own tools or public bio-containers. Our large scale data analyses on 100 TB of data presented how I completed the entire cloud utilization cycle in only 21 hours. Moreover, the fragment recruitment application on all WGS data of HMP was completed in less than 2 hours on Amazon EC2.

The fragment recruitment application presented a study model in which users can easily query the entire HMP data to a personalized reference dataset on the cloud. In microbial studies, combining metagenomic data with microbial reference genomes has been commonly used (Eloe-Fadrosh et al., 2016). In our use case, I recruited all WGS data of the HMP to two different strains of *E. coli*: a toxic strain and a non-toxic strain. The intention is to find genome sequence segments that are not presented in the metagenomic samples. These sequence segments, which are unique for the toxic strain, might have functional impact that differs from the non-toxic one. I applied a functional analysis using the sequence segments and reproduced two toxic genes that caused the 2011 German *E. coli* outbreak. The same method can be applied to other isolates or single cell genomes.

6 Conclusion and outlook

6.1 Conclusion

In this thesis, I have developed a cloud based bioinformatics framework tackling two computational challenges introduced by large scale NGS data: (i) sequence mapping, a computationally intensive task and (ii) *de novo* genome assembly, a memory intensive task. By leveraging the powerful distributed computing engine, the Apache Spark, I have implemented two native applications, Sparkhit and Reflexiv, to address the two challenges. Both tools have better performances compared to existing tools. I have also integrated a series of analytical modules that enables users to carry out various data mining tasks on the cloud. Using the framework, I am able to rapidly analyze large amounts of genomic data on the Amazon EC2 cloud.

In the first part of my work, I have presented Sparkhit, a Spark based distributed computational framework for large scale genomic analytics. Sparkhit mainly focuses on addressing the computationally intensive challenge in sequence mapping. It incorporates a variety of tools and methods that are programmed in the Spark extended MapReduce model. In chapter 3, I have described (i) the algorithms and pipelines of a fragment recruitment tool and a short-read mapping tool, (ii) the implementation of a general tool wrapper to invoke and parallelize external tools and docker containers, and (iii) the integration of Spark's machine learning library for downstream data mining. I also presented the architecture of Sparkhit and the utilities that I used for deploying Spark clusters and downloading public datasets.

In the result section, I have carried out a series of performance benchmarks on Sparkhit. Our tool had excellent run time performance on data preprocessing comparing to Crossbow (18 to 32 times faster) and significant run time improvement on fragment recruitment comparing to MetaSpark (92 to 157 times faster). Although I recruited 10% to 12% less reads than MetaSpark, I can adjust to a smaller K-mer size that recruits slightly more reads than MetaSpark, while still running 47 to 124 times faster. In addition, my tool has a reasonable accuracy and sensitivity on sequence mapping. Sparkhit-recruiter scaled linearly with the increasing amount of input data, as I have used Spark RDD to balance data distribution and optimized the computational parallelization. When scaling out to more worker nodes, Sparkhit still keeps a good scaling performance with a minor slowdown at more then 60 worker nodes.

In the second part of the thesis, I have presented Reflexiv, a distributed *de novo* genome assembler that is built on top the Apache Spark platform. I have invented a new distributed data structure and implemented it in the Reflexiv assembler. The data structure is called Reflexible Distributed K-mer (RDK), which is a higher level abstraction of the Spark RDD. It uses the Spark RDD to distribute large amounts of reflexible k-mers across the Spark cluster and assembles the genome in parallel. I have introduced a random k-mer reflecting method to retrieve the adjacencies between overlapped k-mers and to assemble the genome in an iterative way. I have also described how to solve repeats in the genome and pop bubbles during the assembly.

In the results section, I have carried out a series of benchmarks on Reflexiv. My tool had similar assembly quality with both Ray and Abyss. Although Velvet assembles longer contigs than the other tools, it has more mis-assembled contigs in its result. Reflexiv has excellent run time performances on ethernet connected Spark clusters. Compared to existing tools, Reflexiv runs 8-17 times faster than the Ray assembler and 7-18 times faster than the AbySS assembler on the clusters deployed at the de.NBI cloud.

In the third part of the thesis, I presented a use case to rapidly analyze a collection of 100TB genomic dataset. In the genomic field, Spark and Hadoop based bioinformatics tools are not widely used, as users have little knowledge on distributed computing. Therefore, the use case focuses on providing a simple and comprehensive cloud application to directly access and analyze public genomic datasets. I presented an easy way to setup a Spark cluster on the Amazon cloud with just one line of command. I also presented parallel downloading and decompression methods to optimize the preprocessing of large amounts of genomic data on the cloud.

In the use case, I downloaded and analyzed 100 TB data in only 21 hours. Moreover, the fragment recruitment application on all WGS data of HMP was completed in less than 2 hours on Amazon EC2. The fragment recruitment application presented a study model that users can easily query the entire HMP data to a personalized reference dataset on the cloud. In the use case, I recruited all WGS data of the HMP to two different strains of *E. coli* and applied a functional analysis using the result. As a proof of concept application, I have found two toxic genes that caused the 2011 German *E. coli* outbreak.

In summary, my work contributes to the interdisciplinary research of life science and distributed cloud computing by improving existing methods with a new data structure, algorithms, and distributed implementations. The entire study involves theoretical algorithmic development, distributed software engineering and proof of concept biological applications. As a result, I have successfully accelerated run time

performances of two specific bioinformatics applications, sequence mapping and *de novo* genome assembly. I have also facilitated genomic research communities to engage large scale NGS data studies on the cloud.

6.2 Outlook

In the distributed implementation of the Sparkhit framework, I have used a broadcast function to ship just one copy of the reference index to each worker node for all the read mapping processes. Yet, the broadcasting process (building the reference index and the network broadcasting) introduces a slight run time overhead, as it is a single process running on just one CPU of the master node. The current work around is to use a local implemented tool to pre-built the reference index on a single computer. Then, the Sparkhit program can directly broadcast the pre-build reference index to each worker node. Thus, the overhead of building a reference index can be solved. Nevertheless, the network broadcasting process slows down the entire process, as it is not parallelized.

To further improve the scalability of the Sparkhit framework, I will implement a parallel reference index download function in the next upgrade of the software. A similar function has been used in the Sparkhit-piper module. When parallelizing external tools such as BWA and Bowtie2, I have used a parallel download function to download the reference index to the worker nodes. In this function, the pre-built reference index is usually pre-uploaded to a distributed storage system such as the Amazon S3 and the Openstack Swift file system. Once uploaded, the reference index can be downloaded simultaneously to all worker nodes of the Spark cluster by the parallel download function. I will implement the same function for both Sparkhit-recruiter and Sparkhit-mapper. Thus, the overhead of the reference index broadcasting process can be resolved.

For the distributed *de novo* genome assembler, there are two future works that can be implemented to add more functionality to the assembler. In the current implementation of the assembler, I have used a 'Long' object to encode a k-mer. Such an implementation restricts the program from using k-mer sizes longer than 31nt. In the future upgrade of the software, I will use an 'array' object to store the binaries of the encoded k-mer. In this way, there will be no limitations for the k-mer length.

The other future work is to add an extra scaffolding module to the Reflexiv assembler. The current version of the assembly pipeline is only implemented to assemble contigs. For the scaffolding phase, most of the scaffolding tools use the paired sequencing reads to build up scaffolds. Since my tool already has a distributed read mapping function, I will incorporate the function to firstly search for the mate pair sequencing

reads that connects two contigs. This function can be implemented in a ‘map’ step (in the MapReduce model) that maps all sequencing reads to the assembled contigs. Once the mate pair connections are found, a ‘reduce’ step is followed to lay out all the connected contigs and assemble the scaffolds.

Bibliography

Abu-Doleh, A. and Ü. V. Çatalyürek (2015). „Spaler: Spark and GraphX based de novo genome assembler“. In: *2015 IEEE International Conference on Big Data (Big Data)*, pp. 1013–1018 (cit. on p. 27).

Abuin, J. M., J. C. Pichel, T. F. Pena, and J. Amigo (2016). „SparkBWA: Speeding Up the Alignment of High-Throughput DNA Sequencing Data“. In: *PLoS One* 11.5, e0155461 (cit. on p. 24).

Altschul, S. F., W. Gish, W. Miller, E. W. Myers, and D. J. Lipman (1990a). „Basic local alignment search tool“. In: *J Mol Biol* 215.3, pp. 403–10 (cit. on p. 89).

Altschul, Stephen F., Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman (1990b). „Basic local alignment search tool“. In: *Journal of Molecular Biology* 215.3, pp. 403 –410 (cit. on p. 21).

Auton, Adam et al. (2015). „A global reference for human genetic variation“. In: *Nature* 526.7571, pp. 68–74 (cit. on p. 3).

Bankevich, Anton, Sergey Nurk, Dmitry Antipov, et al. (2012). „SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing“. In: *Journal of Computational Biology* 19.5. PMID: 22506599, pp. 455–477. eprint: https://doi.org/10.1089/cmb.2012.0021 (cit. on p. 27).

Betts, J.G., P. Desaix, J.E. Johnson, et al. (2013). *Anatomy & Physiology*. Open Textbooks. OpenStax College, Rice University (cit. on p. 2).

Boisvert, S., F. Laviolette, and J. Corbeil (2010). „Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies“. In: *J Comput Biol* 17.11, pp. 1519–33 (cit. on pp. 9, 27, 57).

Bray, N. L., H. Pimentel, P. Melsted, and L. Pachter (2016). „Near-optimal probabilistic RNA-seq quantification“. In: *Nat Biotechnol* 34.5, pp. 525–7 (cit. on pp. 9, 87).

Chaisson, Mark J. P., Richard K. Wilson, and Evan E. Eichler (2015). „Genetic variation and the de novo assembly of human genomes“. In: *Nature Reviews Genetics* 16. Review Article, 627 EP – (cit. on p. 1).

Chapman, Jarrod A., Isaac Ho, Sirisha Sunkara, et al. (2011). „Meraculous: De Novo Genome Assembly with Short Paired-End Reads“. In: *PLOS ONE* 6.8, pp. 1–13 (cit. on p. 26).

Collins, Francis S., Eric D. Green, Alan E. Guttmacher, and Mark S. Guyer (2003). „A vision for the future of genomics research“. In: *Nature* 422, 835 EP – (cit. on p. 2).

Consortium, Rice Genomes Project (2014). „The 3,000 rice genomes project“. In: *Gigascience* 3, p. 7 (cit. on p. 3).

Dahm, Ralf (2008). „Discovering DNA: Friedrich Miescher and the early years of nucleic acid research“. In: *Human Genetics* 122.6, pp. 565–581 (cit. on p. 1).

Dean, Jeffrey and Sanjay Ghemawat (2004). „MapReduce: Simplified Data Processing on Large Clusters“. In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, pp. 137–150 (cit. on p. 13).

– (2008). „MapReduce: simplified data processing on large clusters“. In: *Commun. ACM* 51.1, pp. 107–113 (cit. on pp. 7–9, 13).

Decap, D., J. Reumers, C. Herzeel, P. Costanza, and J. Fostier (2015). „Halvade: scalable sequence analysis with MapReduce“. In: *Bioinformatics* 31.15, pp. 2482–8 (cit. on pp. 23, 47).

Droop, A. P. (2016). „qsubsec: a lightweight template system for defining sun grid engine workflows“. In: *Bioinformatics* 32.8, pp. 1267–8 (cit. on pp. 3, 9).

Eloe-Fadrosh, E. A., D. Paez-Espino, J. Jarett, et al. (2016). „Global metagenomic survey reveals a new bacterial candidate phylum in geothermal springs“. In: *Nat Commun* 7, p. 10476 (cit. on p. 91).

Ensi, 2017. http://cloudonmove.com/iaas-paas-saas-what-do-they-mean/. IaaS, PaaS, SaaS – What do they mean? by Ensi Maria on 2017-08-01 (cit. on p. 8).

Gollery, Martin (2005). „Bioinformatics: Sequence and Genome Analysis, 2nd ed. David W. Mount. Cold Spring Harbor, NY: Cold Spring Harbor Laboratory Press, 2004, 692 pp., $75.00, paperback. ISBN 0-87969-712-1.“ In: *Clinical Chemistry* 51.11, pp. 2219–2219. eprint: http://clinchem.aaccjnls.org/content/51/11/2219.1.full.pdf (cit. on p. 19).

Gonzalez, Joseph E., Reynold S. Xin, Ankur Dave, et al. (2014). „GraphX: Graph Processing in a Distributed Dataflow Framework“. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, pp. 599–613 (cit. on p. 28).

Goodwin, Sara, John D. McPherson, and W. Richard McCombie (2016). „Coming of age: ten years of next-generation sequencing technologies“. In: *Nature Reviews Genetics* 17. Review Article, 333 EP – (cit. on p. 1).

Green, Eric D. (2001). „Strategies for the systematic sequencing of complex genomes“. In: *Nature Reviews Genetics* 2. Review Article, 573 EP – (cit. on p. 1).

Gropp, William, Ewing Lusk, Nathan Doss, and Anthony Skjellum (1996). „A high-performance, portable implementation of the MPI message passing interface standard“. In: *Parallel computing* 22.6, pp. 789–828 (cit. on pp. 9, 27).

Grotzke, Martin (2017). *Kryo: Fast, efficient Java serialization and cloning* (cit. on p. 38).

Gurevich, Alexey, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler (2013). „QUAST: quality assessment tool for genome assemblies“. In: *Bioinformatics* 29.8, pp. 1072–1075. eprint: /oup/backfile/content_public/journal/bioinformatics/29/8/10.1093_bioinformatics_btt086/2/btt086.pdf (cit. on p. 77).

Henke, Christian (2017). *BiBiS3, https://wiki.cebitec.uni-bielefeld.de/bibiserv-1.25.2/index.php/BiBiS3* (cit. on pp. 6, 7, 85).

Huang, Liren, Jan Krüger, and Alexander Sczyrba (2018). „Analyzing large scale genomic data on the cloud with Sparkhit“. In: *Bioinformatics* 34.9, pp. 1457–1465. eprint: /oup/backfile/content_public/journal/bioinformatics/34/9/10.1093_bioinformatics_btx808/2/btx808.pdf (cit. on pp. 31, 52).

Hyatt, D., G. L. Chen, P. F. Locascio, et al. (2010). „Prodigal: prokaryotic gene recognition and translation initiation site identification“. In: *BMC Bioinformatics* 11, p. 119 (cit. on p. 89).

Kanehisa, M., S. Goto, Y. Sato, et al. (2014). „Data, information, knowledge and principle: back to metabolism in KEGG“. In: *Nucleic Acids Res* 42.Database issue, pp. D199–205 (cit. on p. 89).

Landset, Sara, Taghi M. Khoshgoftaar, Aaron N. Richter, and Tawfiq Hasanin (2015). „A survey of open source tools for machine learning with big data in the Hadoop ecosystem“. In: *Journal of Big Data* 2.1, p. 24 (cit. on p. 14).

Langmead, B. and S. L. Salzberg (2012). „Fast gapped-read alignment with Bowtie 2“. In: *Nat Methods* 9.4, pp. 357–9 (cit. on p. 9).

Langmead, B., M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg (2009). „Searching for SNPs with cloud computing“. In: *Genome Biol* 10.11, R134 (cit. on pp. 11, 23, 47).

Langmead, B., K. D. Hansen, and J. T. Leek (2010). „Cloud-scale RNA-sequencing differential expression analysis with Myrna“. In: *Genome Biol* 11.8, R83 (cit. on p. 23).

Langmead, Ben (2010). „Aligning short sequencing reads with Bowtie“. In: *Current protocols in bioinformatics*, pp. 11–7 (cit. on p. 24).

Langmead, Ben and Abhinav Nellore (2018). „Cloud computing for genomic data analysis and collaboration“. In: *Nature Reviews Genetics* 19. Review Article, 208 EP – (cit. on pp. 2, 4, 8).

Li, H. and R. Durbin (2009). „Fast and accurate short read alignment with Burrows-Wheeler transform“. In: *Bioinformatics* 25.14, pp. 1754–60 (cit. on pp. 9, 21, 22, 24, 87).

Li, H., B. Handsaker, A. Wysoker, et al. (2009). „The Sequence Alignment/Map format and SAMtools“. In: *Bioinformatics* 25.16, pp. 2078–9 (cit. on pp. 9, 87).

Li, R., Y. Li, K. Kristiansen, and J. Wang (2008). „SOAP: short oligonucleotide alignment program“. In: *Bioinformatics* 24.5, pp. 713–4 (cit. on pp. 9, 24).

Li, Ruiqiang, Hongmei Zhu, Jue Ruan, et al. (2010). „De novo assembly of human genomes with massively parallel short read sequencing“. In: *Genome Research* 20.2, pp. 265–272. eprint: http://genome.cshlp.org/content/20/2/265.full.pdf+html (cit. on p. 26).

Lipman, DJ and WR Pearson (1985). „Rapid and sensitive protein similarity searches“. In: *Science* 227.4693, pp. 1435–1441. eprint: http://science.sciencemag.org/content/227/4693/1435.full.pdf (cit. on p. 20).

Liu, Yongchao, Bertil Schmidt, and Douglas L. Maskell (2011). „Parallelized short read assembly of large genomes using de Bruijn graphs“. In: *BMC Bioinformatics* 12.1, p. 354 (cit. on p. 28).

McKenna, A., M. Hanna, E. Banks, et al. (2010). „The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data“. In: *Genome Res* 20.9, pp. 1297–303 (cit. on p. 24).

Melissa, Bastide and McCombie W. Richard (2007). „Assembling Genomic DNA Sequences with PHRAP“. In: *Current Protocols in Bioinformatics* 17.1, pp. 11.4.1–11.4.15. eprint: https://currentprotocols.onlinelibrary.wiley.com/doi/pdf/10.1002/0471250953.bi1104s17 (cit. on p. 24).

Mell, Peter and Timothy Grance (2011). *The NIST Definition of Cloud Computing*. Tech. rep. 800-145. Gaithersburg, MD: National Institute of Standards and Technology (NIST) (cit. on p. 7).

Meng, Jintao, Bingqiang Wang, Yanjie Wei, Shengzhong Feng, and Pavan Balaji (2014). „SWAP-Assembler: scalable and efficient genome assembly towards thousands of cores“. In: *BMC Bioinformatics* 15.9, S2 (cit. on p. 27).

Merkel, Dirk (2014). „Docker: lightweight linux containers for consistent development and deployment“. In: *Linux Journal* 2014.239, p. 2 (cit. on pp. 11, 31, 41).

Myers, Eugene W., Granger G. Sutton, Art L. Delcher, et al. (2000). „A Whole-Genome Assembly of Drosophila“. In: *Science* 287.5461, pp. 2196–2204. eprint: http://science.sciencemag.org/content/287/5461/2196.full.pdf (cit. on p. 25).

Nagarajan, Niranjan and Mihai Pop (2013). „Sequence assembly demystified“. In: *Nature Reviews Genetics* 14. Review Article, 157 EP – (cit. on pp. 24, 67).

Needleman, Saul B. and Christian D. Wunsch (1970). „A general method applicable to the search for similarities in the amino acid sequence of two proteins“. In: *Journal of Molecular Biology* 48.3, pp. 443 –453 (cit. on p. 20).

Niemenmaa, M., A. Kallio, A. Schumacher, et al. (2012). „Hadoop-BAM: directly manipulating next generation sequencing data in the cloud“. In: *Bioinformatics* 28.6, pp. 876–7 (cit. on p. 43).

Nih Hmp, Working group, J. Peterson, et al. (2009). „The NIH Human Microbiome Project“. In: *Genome Res* 19.12, pp. 2317–23 (cit. on p. 3).

Niu, B., Z. Zhu, L. Fu, S. Wu, and W. Li (2011). „FR-HIT, a very fast program to recruit metagenomic reads to homologous reference genomes“. In: *Bioinformatics* 27.12, pp. 1704–5 (cit. on pp. 9, 22, 32).

Pearson, W. R. and D. J. Lipman (1988). „Improved tools for biological sequence comparison“. In: *Proc Natl Acad Sci U S A* 85.8, pp. 2444–8 (cit. on p. 36).

Pevzner, Pavel A., Haixu Tang, and Michael S. Waterman (2001). „An Eulerian path approach to DNA fragment assembly“. In: *Proceedings of the National Academy of Sciences* 98.17, pp. 9748–9753. eprint: http://www.pnas.org/content/98/17/9748.full.pdf (cit. on pp. 26, 27).

Rasko, D. A., D. R. Webster, J. W. Sahl, et al. (2011). „Origins of the E. coli strain causing an outbreak of hemolytic-uremic syndrome in Germany“. In: *N Engl J Med* 365.8, pp. 709–17 (cit. on p. 90).

Rasmussen, K. R., J. Stoye, and E. W. Myers (2006). „Efficient q-gram filters for finding all epsilon-matches over a given length“. In: *J Comput Biol* 13.2, pp. 296–308 (cit. on p. 22).

Rinke, C., P. Schwientek, A. Sczyrba, et al. (2013). „Insights into the phylogeny and coding potential of microbial dark matter“. In: *Nature* 499.7459, pp. 431–7 (cit. on p. 88).

Rusch, D. B., A. L. Halpern, G. Sutton, et al. (2007). „The Sorcerer II Global Ocean Sampling expedition: northwest Atlantic through eastern tropical Pacific“. In: *PLoS Biol* 5.3, e77 (cit. on pp. 19, 24, 32).

Schatz, M. C. (2009). „CloudBurst: highly sensitive read mapping with MapReduce“. In: *Bioinformatics* 25.11, pp. 1363–9 (cit. on pp. 23, 47).

Schatz, Michael C, Arthur L Delcher, and Steven L. Salzberg (2010). „Assembly of large genomes using second-generation sequencing“. In: *Genome Research*. eprint: http://genome.cshlp.org/content/early/2010/05/27/gr.101360.109.full.pdf+html (cit. on p. 25).

Shvachko, K., H. Kuang, S. Radia, and R. Chansler (2010). „The Hadoop Distributed File System“. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10 (cit. on pp. 7, 9, 13).

Simpson, J. T., K. Wong, S. D. Jackman, et al. (2009). „ABySS: a parallel assembler for short read sequence data“. In: *Genome Res* 19.6, pp. 1117–23 (cit. on pp. 9, 26, 27, 56).

Simpson, Jared T. and Richard Durbin (2012). „Efficient de novo assembly of large genomes using compressed data structures“. In: *Genome Research* 22.3, pp. 549–556. eprint: http://genome.cshlp.org/content/22/3/549.full.pdf+html (cit. on p. 25).

Singh, Dilpreet and Chandan K. Reddy (2014). „A survey on platforms for big data analytics“. In: *Journal of Big Data* 2.1, p. 8 (cit. on p. 6).

Smith, T.F. and M.S. Waterman (1981). „Identification of common molecular subsequences“. In: *Journal of Molecular Biology* 147.1, pp. 195 –197 (cit. on p. 20).

Sohn, Jang-il and Jin-Wu Nam (2018). „The present and future of de novo whole-genome assembly“. In: *Briefings in Bioinformatics* 19.1, pp. 23–40. eprint: /oup/backfile/content_public/journal/bib/19/1/10.1093_bib_bbw096/3/bbw096.pdf (cit. on p. 26).

Sutton, Granger G., OWEN WHITE, MARK D. ADAMS, and ANTHONY R. KERLAVAGE (1995). „TIGR Assembler: A New Tool for Assembling Large Shotgun Sequencing Projects“. In: *Genome Science and Technology* 1.1, pp. 9–19. eprint: https://doi.org/10.1089/gst.1995.1.9 (cit. on p. 24).

Thomason, Andrew (1989). „A simple linear expected time algorithm for finding a hamilton path“. In: *Discrete Mathematics* 75.1, pp. 373 –379 (cit. on p. 26).

Watson, J. D. and F. H. C. Crick (1953). „Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid“. In: *Nature* 171, 737 EP – (cit. on p. 1).

Wood, D. E. and S. L. Salzberg (2014). „Kraken: ultrafast metagenomic sequence classification using exact alignments“. In: *Genome Biol* 15.3, R46 (cit. on pp. 9, 87).

Wyatt, A. W., F. Mo, K. Wang, et al. (2014). „Heterogeneity in the inter-tumor transcriptome of high risk prostate cancer“. In: *Genome Biol* 15.8, p. 426 (cit. on pp. 12, 81, 83, 84, 87).

Yeo, Sarah, Lauren Coombe, René L Warren, Justin Chu, and Inanç Birol (2018). „ARCS: scaffolding genome drafts with linked reads“. In: *Bioinformatics* 34.5, pp. 725–731. eprint: /oup/backfile/content_public/journal/bioinformatics/34/5/10.1093_bioinformatics_btx675/2/btx675.pdf (cit. on p. 80).

Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, et al. (2012). „Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing“. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, pp. 2–2 (cit. on pp. 7, 10, 13, 16).

Zerbino, Daniel R. and Ewan Birney (2008). „Velvet: Algorithms for de novo short read assembly using de Bruijn graphs“. In: *Genome Research* 18.5, pp. 821–829. eprint: http://genome.cshlp.org/content/18/5/821.full.pdf+html (cit. on pp. 26, 27).

Zhou, W., R. Li, S. Yuan, et al. (2017). „MetaSpark: a spark-based distributed processing tool to recruit metagenomic reads to reference genomes“. In: *Bioinformatics* (cit. on pp. 11, 24, 46).

Zook, Justin M., Brad Chapman, Jason Wang, et al. (2014). „Integrating human sequence data sets provides a resource of benchmark SNP and indel genotype calls“. In: *Nature Biotechnology* 32, 246 EP – (cit. on p. 45).

Websites

Moustafa, Ahmed (2005). *JAligner: Open source java implementation of Smith-Waterman*. URL: http://jaligner.sourceforge.net/ (visited on Feb. 24, 2018) (cit. on p. 20).

Seward, Julian (1996). *Bzip2 data compressor*. URL: http://www.bzip.org/ (visited on Sept. 20, 2010) (cit. on p. 20).

List of Figures

List of Tables

Appendix

We have extended the Spark machine learning library (Mllib) for downstream data mining. Here we describe each module in detail.

Clustering

K-means is an iterative algorithm that clusters data points into k number of clusters based on a distance metric. It first randomly generates k number of centroids as the initial 'means'. Then, all data points are assigned to the closest centroid that creates k clusters. After that, a new centroid is elected within each cluster and all data points are reassigned to the newly created k clusters. These re-centering and re-clustering steps are iterated until convergence conditions are fulfilled.

To implement the k-means in a distributed way, the re-centering and re-clustering steps are split and implemented in a Spark extended MapReduce paradigm. The 'map' step assigns each data point to the closest centroids to form clusters. Whereas the 'reduce' step computes the new centroid for each cluster. Since data points are distributed across cluster nodes, the 'reduce' step applies Spark's 'reduceByKey' function to shuffle data points by clustering and calculating the centroids. The 'map' and 'reduce' steps iterate until the convergence is reached.

The bisecting k-means algorithm is used for hierarchical clustering. It uses a divisive approach that recursively splits all data points in a reverse hierarchical way. It starts with splitting one cluster of all data points into two sub-clusters using the k-means algorithm (Bisecting). Then, the splitting runs recursively to one cluster that is selected from the last recursion until the desired number of clusters is reached.

Principle component analysis

We used principle component analysis (PCA) to separate individuals from different continental regions based on their genotypes, as well as different tumor and benign samples based on their gene expression profiles (supplementary file 2 Fig. S4E and S4F). The genotypes or the abundances of genes are encoded in floats that forms

a matrix, where each row represents features of a sample in a study cohort. Spark loads the matrix into an RDD of vectors as a "RowMatrix", where each vector is a single row of the input matrix. Then, a distributed covariance method is used for the dimension reduction. It first applies a 'reduce' step to compute the empirical mean of each column. The empirical means are, then, used to calculate the deviations from the means as the outer product in a 'map' step. Followed by a 'reduce' step, a covariance matrix is created based on the outer product (a matrix computation operates on the conjugate transpose of the outer product matrix and the outer product matrix itself). The final step collects and computes the eigenvectors.

Correlation test

The correlation test is used to measure the linear dependence of two biological samples based on their genetic features (gene expression profiles or genotypes). The expression data or the genotype profile is imported into an RDD, where each element of the RDD is a vector of input data points (gene abundance or encoded genotypes) of different samples. All vectors must have the same number of data points, so that the correlation can be calculated based on two series of variables in the same length. A 'reduce' step applies the Pearson or the Spearman's rank correlation method to the two columns of the vector list (as a matrix) in the RDD. The returned correlation coefficients are sent back to the driver node by a 'collect' function.

Logistic regression

In genome wide association study (GWAS), the logistic regression is used to examine heterogeneous SNPs in a case cohort based on the training data of genotype variables from a control cohort. The limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm is implemented for the logistic regression analysis. The distributed implementation of the L-BFGS applies a 'map' function that reads each partition of the training dataset and calculates the logistic losses of each partition based on the current weights (the weighted scores in the current iteration) with the following equation:

$$L(w; x, y) := log(1 + exp(-yw^T x)) \tag{6.1}$$

where L is the function for computing the logistic loss, w is a vector of weights in the current iteration, x is a vector of input training data points (a row of variables) and y is a vector of corresponding labels to be predicted. After all logistic losses are calculated in the 'map' step, a 'reduce' step collects and summarizes the logistic losses on each worker node and updates the weights. The 'map' and the 'reduce' steps iterate until the approximate minimum is obtained (the weights and logistic losses are smaller than the default criteria).

Chi-square test

The Pearson's Chi-square test is used to perform statistical hypothesis tests that examine the biological variance, such as the gene differential expressions between case and control groups or the genotype profiles between two different cohorts. The expression data or the genotype profile is imported into an RDD where each element of the RDD is a vector (labeled points in Spark) of input data points (gene abundance or encoded genotypes) along different samples. Then, the chi-square test is carried out on each vector independently in a 'map' process. A 'collect' function aggregates statistical test results (p-values) from each worker node and sends the results back to the driver node.

Hardy-Weinberg equilibrium

In genome-wide association studies (GWAS), the Hardy-Weinberg equilibrium (HWE) is used to estimate genotyping errors and the population stratification by predicting genotype frequencies of a given cohort. According to the HWE, allele and genotype frequencies in a given population stays constant from generation to generation without other evolutionary interferences. Let *p* and *q* denote the ratios of two alleles. *A* and *a* denote the dominant and the recessive alleles of a diploid genome. The HWE can be expressed as:

$$(p+q)^2 = p^2 + 2pq + q^2 = 1 \tag{6.2}$$

Where p^2 is the expected genotype frequency of AA (the dominant allele), *2pq* is the expected genotype frequency of Aa, q^2 is the expected genotype frequency of aa (the resessive allele). When the ratios of homozygous and heterozygous genotypes significantly differ from the prediction under the HWE assumptions, genotyping errors or a population stratification is expected. Sparkhit loads the genotype data from a VCF file where each line represents the genotypes of one locus along different samples. A 'map' function is implemented to calculate the actual allele frequencies of each input line and predict the expected genotype frequencies.

Supplementary tables

Tab. S1: Run times on recruiting simulated sequencing data to 72 MB and 142 MB reference genome. All tests were carried out on Sparkhit clusters with 10, 20, and 30 worker nodes. Each worker node has 16vCPUs.

Tool	Run time (s)					
	72 MB			142 MB		
	10 nodes	20 nodes	30 nodes	10 nodes	20 nodes	30 nodes
MetaSpark (k-mer 11nt)	9725	7761	7807	14941	10779	9469
Sparkhit-recruiter (k-mer 11nt)	72	84	72	95	94	91
Sparkhit-recruiter (k-mer 12nt)	205	130	111	120	102	100

Tab. S2: Numbers of recruited reads for recruiting 6 million simulated reads to a 72 MB reference genome and 1 million simulated reads to a 142 MB reference genome.

Tool	Number of recruited reads	
	72MB reference genome 6 million fastq reads	142MB reference genome 1 million fastq reads
MetaSpark k-mer 11nt	9508168	1598186
Sparkhit-recruiter k-mer 11nt	8370739	1445459
Sparkhit-recruiter k-mer 10nt	9529937	1679323

Tab. S3: Run time comparison for setting up a Spark cluster on the Amazon EC2 cloud. Durations include pending for approval of EC2 spot requests and waiting for the SSH connection to each EC2 instance.

Tool	Run time (s)	
	100 nodes	50 nodes
Bibigrid	1190	896
Spark-ec2	2394	1539
EMR	688	660

Tab. S4: Run times for processing all HMP WGS data on the Amazon EC2 cloud. Recruitment was carried out using Sparkhit-recruiter while profiling was carried out using Sparkhit invoked Kraken.

Processes	Run time (s)	
	100 nodes	50 nodes
Download	1006	1673
Decompression	1273	2298
Mapping	2113	3603
Profiling	1114	1670
Summary	186	190
Total	5692	9434

Tab. S5: Run times for processing 15 TB BAM files of the 3000 Rice Genomes Project. Variant detection was carried out using Sparkhit invoked Mpileup. The detected variants were uploaded to Amazon S3 for persistent storage.

Processes	Run time (s)	
	100 nodes	50 nodes
Download	10356	5295
Variant detection	70363	32838
Upload	1326	861
Total	82045	38994

Tab. S6: Run times for processing 5.6 TB compressed sequencing data of the 1000 Genome Project. Mapping was carried out using Sparkhit invoked BWA aligner. Mapping result was uploaded to Amazon S3 for persistent storage.

Processes	Run time (s)
	100 nodes
Download	2652
Mapping	13304
Upload	3921
Total	19877

Tab. S7: Run times for processing 3.2 TB compressed sequencing data of a prostate cancer transcriptome study. Gene expression profiling is carried out using Sparkhit invoked Kallisto.

Processes	Run time (s)	
	100 nodes	50 nodes
Download	5368	9102
Gene expression profiling	1269	2348
Summary	61	73
Total	6698	11523

Tab. S8: Run time comparisons between different aligners on 1.3 TB input fastq files (Data-1). The comparisons were carried out across different sizes of reference genomes and different numbers of worker nodes.

Tool	Run time (s)					
	50 nodes			30 nodes		
	36 MB	72 MB	142 MB	36 MB	72 MB	142 MB
Sparkhit BWAMEM	658	810	927	1036	1116	1298
Crossbow	1349	1518	1732	2160	2421	2783
Sparkhit Bowtie2	550	563	608	566	697	842
Sparkhit-mapper	258	325	500	340	454	700
Sparkhit-recuiter	563	762	1230	850	1161	1897
Sparkhit Fr-hit	333	460	696	511	716	1116

Tab. S9: Run time comparisons between different aligners on 545 GB input fastq files (Data-2). The comparisons were carried out across different sizes of reference genomes and different numbers of worker nodes.

Tool	Run time (s)					
	50 nodes			30 nodes		
	36 MB	72 MB	142 MB	36 MB	72 MB	142 MB
Sparkhit BWAMEM	287	308	363	450	480	545
Crossbow	593	657	900	1000	1047	1220
Sparkhit Bowtie2	198	208	217	329	360	366
Sparkhit-mapper	160	190	277	202	258	272
Sparkhit-recuiter	262	395	602	416	602	966
Sparkhit Fr-hit	156	218	305	231	320	489

Tab. S10: Run times of Sparkhit-recruiter for recruiting 100 – 1000 GB sequencing data to a 72 MB (Ref-2) reference genome. Fragment recruitment was carried out on a 30 worker nodes Spark cluster deployed on the Amazon EC2 cloud. Each worker node has 32 vCPUs.

Sparkhit-recruiter	
Input data size (GB)	Run time (s)
95	179.599
187	289.762
290	366.968
391	439.932
496	520.955
600	604.355
704	685.301
799	769.355
894	872.818
988	925.38

Tab. S11: Scaling performance of Sparkhit-recruiter. We recruited 1.3 TB fastq files (Data-1) to a 72 MB reference genome (Ref-2) on a 30 nodes Spark cluster deployed on the Amazon EC2 cloud. Run times were used to measure the speed up at each scale.

Sparkhit-recruiter	
Number of nodes	Run time (s)
10	3262
20	1709
30	1161
40	949
50	761
60	644
70	608
80	555
90	496
100	453

Tab. S12: Resource consumptions between Crossbow and Sparkhit for preprocessing 338 TB Bzip2 compressed fastq files on 50 and 100 worker nodes.

Resources	100 nodes		50 nodes	
	Crossbow	Sparkhit	Crossbow	Sparkhit
Storage (input): Read data fetched	337.8G	337.8G	337.8G	337.8G
Storage (output): Read data pushed to HDFS	394.7G	1291.2G	394.7G	1291.2G
Memory: Total committed heap usage	9.7G	800.0G	10.8G	357.2G
Wall clock: program run time (s)	7601	234	7926	429
CPU: run time (ms)	202541780	418400492	210211966	408968576

Tab. S13: Comparing the recruited number of reads between Crossbow, Sparkhit-recruiter and Sparkhit-mapper when recruiting 1.3 TB fastq files to a 72 MB reference genome. Sparkhit-mapper uses more strict pigeonhole pricinple to filter candidate blocks.

Tool	Number of recruited reads
Sparkhit-recruiter	496569401
Sparkhit-mapper	27591565
Crossbow	16288351

Tab. S14: Run times of machine learning library on both private cluster and Amazon EC2 cloud. All computations were performed on a 200 GB VCF file (Data-3) cached in the memeory.

Machine learning module	Run time (s)			
	Private Cluster		Amazon EC2	
	20 nodes	40 nodes	20 nodes	40 nodes
PCA	689	521	301	268
K-means	631	448	84	70
Bisecting k-means	601	409	64	49
Correlation	482	378	176	117
Logistic regression	377	310	233	206
HWE	313	201	60	38
Chi-square test	318	198	48	36

Tab. S15: Run times for different iterations of K-means clustering. Comparisons were carried out between two sets of iterations: with data caching and non data caching. The input data set is a VCF file (Data-3).

K-means iterations	Run time (s)	
	Without cache	Cache
1	133	51
10	592	123
20	1175	212
30	1762	308
40	2371	403

Tab. S16: Comparison of run times between different distributed *de novo* genome assemblers. The comparison was carried out on a single computer instance using 10, 20, and 30 CPUs. The 10GB simulated sequencing data of the human chromosome 17 was used for the benchmark.

Clusters	Run time (s)		
	Reflexiv	Ray	AbySS
10 cores, 1 node	4307	9659	4323
20 cores, 1 node	3162	5287	3321
30 cores, 1 node	2215	3653	2205.9

Tab. S17: Comparison of run times between different distributed *de novo* genome assemblers. The comparison was carried out on clusters with 5, 10, 15, and 20 worker nodes (140, 280, 420, and 560 CPUs). The 10GB simulated sequencing data of the human chromosome 17 was used for the benchmark.

Clusters	Run time (s)		
	Reflexiv	Ray	AbySS
140 cores, 5 nodes	305.906	2519.66	2181.27
280 cores, 10 nodes	167.085	2955.6	3103.93
420 cores, 15 nodes	229.572	3032.36	2941.68
560 cores, 20 nodes	274.83	2586.32	2108.67

Tab. S18: Comparison of run times between different distributed *de novo* genome assemblers. The comparison was carried out on 5 to 20 worker nodes with 140 to 560 CPUs. The 1.3GB real sequencing data of the *E. coli* genome was used for the benchmark.

Clusters	Run time (s)		
	Reflexiv	Ray	AbySS
140 cores, 5 nodes	86.637	907.73	541.57
280 cores, 10 nodes	91.155	999.29	517.334
420 cores, 15 nodes	113.468	1084	678.95
560 cores, 20 nodes	112.813	894.8	536.67

Colophon

This thesis was typeset with LaTeX 2_ε. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Declaration

I have completed my work solely and only with the help of the references mentioned in the thesis.

Bielefeld, June 18, 2019

Liren Huang