Department of Computer Science
Series of Publications A
Report A-2021-8

Scalable computational methods for high-throughput sequencing data analytics in population genomics

Altti Ilari Maarala

Doctoral dissertation, to be presented for public examination with the permission of the Faculty of Science of the University of Helsinki in Auditorium CK112, Exactum, on December 17, 2021, at 12 o'clock.

University of Helsinki
Finland

Supervisor
Keijo Heljanko, University of Helsinki, Finland

Pre-examiners
Ulf Leser, Humboldt-Universität zu Berlin, Germany
Zaid Al-Ars, Delft University of Technology, Netherlands

Opponent
Rayan Chikhi, Institut Pasteur, Paris, France

Custos
Keijo Heljanko, University of Helsinki, Finland

Contact information

Department of Computer Science
P.O. Box 68 (Pietari Kalmin katu 5)
FI-00014 University of Helsinki
Finland

Email address: info@cs.helsinki.fi
URL: http://cs.helsinki.fi
Telephone: +358 2941 911

ISSN 1238-8645
ISBN 978-951-51-7745-2 (paperback)
ISBN 978-951-51-7746-9 (PDF)
Helsinki 2021
Unigrafia

Scalable computational methods for high-throughput sequencing data analytics in population genomics

Altti Ilari Maarala

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
ilari.maarala@helsinki.fi

PhD Thesis, Series of Publications A, Report A-2021-8
Helsinki, December 2021, 98+61 pages
ISSN 1238-8645
ISBN 978-951-51-7745-2 (paperback)
ISBN 978-951-51-7746-9 (PDF)

Abstract

High-throughput sequencing (HTS) technologies have enabled rapid DNA sequencing of whole-genomes collected from various organisms and environments, including human tissues, plants, soil, water, and air. As a result, sequencing data volumes have grown by several orders of magnitude, and the number of assembled whole-genomes is increasing rapidly as well. This whole-genome sequencing (WGS) data has revealed the genetic variation in humans and other species, and advanced various fields from human and microbial genomics to drug design and personalized medicine. The amount of sequencing data has almost doubled every six months, creating new possibilities but also big data challenges in genomics. Diverse methods used in modern computational biology require a vast amount of computational power, and advances in HTS technology are even widening the gap between the analysis input data and the analysis outcome.

Currently, many of the existing genomic analysis tools, algorithms, and pipelines are not fully exploiting the power of distributed and high-performance computing, which in turn limits the analysis throughput and restrains the deployment of the applications to clinical practice in the long run. Thus, the relevance of harnessing distributed and cloud computing in bioinformatics is more significant than ever before. Besides, efficient data compression and storage methods for genomic data processing and retrieval

integrated with conventional bioinformatics tools are essential. These vast datasets have to be stored and structured in formats that can be managed, processed, searched, and analyzed efficiently in distributed systems.

Genomic data contain repetitive sequences, which is one key property in developing efficient compression algorithms to alleviate the data storage burden. Moreover, indexing compressed sequences appropriately for bioinformatics tools, such as read aligners, offers direct sequence search and alignment capabilities with compressed indexes. Relative Lempel-Ziv (RLZ) has been found to be an efficient compression method for repetitive genomes that complies with the data-parallel computing approach. RLZ has recently been used to build hybrid-indexes compatible with read aligners, and we focus on extending it with distributed computing. Data structures found in genomic data formats have properties suitable for parallelizing routine bioinformatics methods, e.g., sequence matching, read alignment, genome assembly, genotype imputation, and variant calling. Compressed indexing fused with the routine bioinformatics methods and data-parallel computing seems a promising approach to building population-scale genome analysis pipelines. Various data decomposition and transformation strategies are studied for optimizing data-parallel computing performance when such routine bioinformatics methods are executed in a complex pipeline. These novel distributed methods are studied in this dissertation and demonstrated in a generalized scalable bioinformatics analysis pipeline design.

The dissertation starts from the main concepts of genomics and DNA sequencing technologies and builds routine bioinformatics methods on the principles of distributed and parallel computing. This dissertation advances towards designing fully distributed and scalable bioinformatics pipelines focusing on population genomic problems where the input data sets are vast and the analysis results are hard to achieve with conventional computing. Finally, the methods studied are applied in scalable population genomics applications using real WGS data and experimented with in a high performance computing cluster. The experiments include mining virus sequences from human metagenomes, imputing genotypes from large-scale human populations, sequence alignment with compressed pan-genomic indexes, and assembling reference genomes for pan-genomic variant calling.

Computing Reviews (2012) Categories and Subject Descriptors:

Software and its engineering → Software organization and properties → Software system structures → Software system models → Massively parallel systems
Applied computing → Life and medical sciences → Genomics → Computational genomics
Information systems → Information retrieval → Search engine architectures and scalability → Search index compression

General Terms:
Algorithms, Software design, Distributed computing, Genomics

Additional Key Words and Phrases:
Population genomics, Next-generation sequencing, Data engineering

Acknowledgements

This ride towards my dissertation has been, shall I say, *epic*, including enchanting feelings of success due to accepted publications, memorable conferences, and colourful summer schools in various countries with all the great experience and people. Not to forget downside moods, or even despondency, while struggling with difficult concepts and complex problems, failed trial arrangements, rejected publications, and sometimes just with an excessive workload. The epic ride is all about pushing the limits, and to score a successful one, it takes several sacrifices along with falls and failures. All this would not have been possible without excellent people who have been guiding, helping, and counselling me on this ride.

The research work for my dissertation actually started at the Aalto University Department of Computer Science in 2016 and continued at the University of Helsinki Department of Computer Science in 2019. First and the most I am grateful to my supervisor, Prof. Keijo Heljanko, for helping me to takeoff towards doctorate, challenging me and pushing my limits on this ride, and guiding me frequently from incoherent verses to complete the dissertation. I am also grateful to my mentor, Prof. Veli Mäkinen, for giving me the opportunity to collaborate with the Genome-Scale Algorithmics (GSA) research group and letting me to work on his insightful research topics. I would like to thank my second mentor Dr. Daniel Valenzuela for his guidance and good advices for solving challenging algorithmic problems in computational genomics. Full credit goes to M.Sc. Ossi Arasalo for his excellent work in our group as a summer trainee at Aalto University. I would like thank Dr. Kalle Pärn, Dr. Javier Nuñez-Fontarnau, and Paavo Häppölä from Institute for Molecular Medicine Finland (FIMM) for offering their knowledge and expertise in the field of genomics. I am grateful to Prof. Joakim Dillner from the Karolinska Institute Dept. of Laboratory Medicine for providing me the opportunity to work with excellent researchers in exiting and novel tumor virology project. Special thanks to Dr. Davit Bzhalava from the Karolinska Institute for offering his in-depth expertise in bioinformatics without fearing to get his hands dirty in coding.

Many thanks to the pre-examiners, Prof. Ulf Leser and Assoc. Prof. Zaid Al-Ars, for their time and consideration in reviewing my dissertation thoroughly. I sincerely appreciate the consent of Dr. Rayan Chikhi to act as my opponent. I would also like to thank our research coordinator Dr. Pirjo Moen for her precise guidance regarding to my Ph.D. studies and practicalities during the dissertation process.

I am also grateful to Prof. Jukka Riekki for opening the academic path for me at the University of Oulu Department of Computer Science and Engineering in 2014 when I finished my M.Sc. thesis and started working full time in the Interactive Spaces research group. I want to thank also Dr. Xiang Su, Dr. Mika Rautiainen, Assoc. Prof. Susanna Pirttikangas, and many other excellent people I have worked with in multiple research projects that have supported me in the beginning of my academic path.

The following funding institutions that have made this work practically possible are gratefully acknowledged: Academy of Finland (projects 336092 and 313469), Helsinki Institute for Information Technology, and Nordic Information for Action eScience Center of Excellence. The providers of computing infrastructure are also gratefully acknowledged: FGCI - Finnish Grid and Cloud Infrastructure and CSC - IT Center for Science Ltd.

Last but not least, I want to thank my lovely wife Jonna for supporting me in many ways that I have not yet even realized. When I was in the shadows she cast the light upon me. I would like to thank also our lovely cats who healed me by purring on my stomach while displacing the laptop and apathy.

In Helsinki, Finland, November 2021
Altti Ilari Maarala

Original publications

This dissertation consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Altti Ilari Maarala, Zurab Bzhalava, Joakim Dillner, Keijo Heljanko, and Davit Bzhalava. **ViraPipe: Scalable parallel pipeline for viral metagenome analysis from next generation sequencing reads.** In Bioinformatics, 34(6): 928-935, 2018. https://doi.org/10.1093/bioinformatics/btx702

II Altti Ilari Maarala, Kalle Pärn, Javier Nuñez-Fontarnau, and Keijo Heljanko. **SparkBeagle: Scalable Genotype Imputation from Distributed Whole-Genome Reference Panels in the Cloud.** In Proceedings of the 11th International Conference on Bioinformatics, Computational Biology and Health Informatics (BCB 2020). https://doi.org/10.1145/3388440.3414860

III Altti Ilari Maarala, Ossi Arasalo, Daniel Valenzuela, Keijo Heljanko, and Veli Mäkinen. **Distributed hybrid-indexing of compressed pan-genomes for scalable and fast sequence alignment.** In PLoS One, 16(8): e0255260, 2021. https://doi.org/10.1371/journal.pone.0255260

IV Altti Ilari Maarala, Ossi Arasalo, Daniel Valenzuela, Keijo Heljanko, and Veli Mäkinen. **Scalable Reference Genome Assembly from Compressed Pan-Genome Index with Spark.** In Proceedings of the 9th International Conference on Big Data (BIGDATA 2020). https://doi.org/10.1007/978-3-030-59612-5_6.

Contents

Chapter 1

Introduction

Ever since the double helix structure of DNA was revealed in 1953 [1], scientists have been developing methods for detecting single nucleotides in a DNA molecule and exploring biology from basic DNA molecules to the origin of life. It took almost 50 years of research and billions of dollars to assemble the first human genome draft, which was published in 2001 as a result of the Human Genome Project [2]. Current advances in high-throughput sequencing (HTS) technology have increased the DNA sequencing throughput and decreased the sequencing price drastically over the last 15 years [3]. Recently, so-called next-generation sequencing (NGS) technologies have enabled rapid sequencing of multiple samples collected from any organism and environment, including human tissues, plants, soil, water, and air [4]. NGS now provides relatively cheap and rapid whole-genome, -exome, -transcriptome, and targeted sequencing, enabling large-scale and profound genome analytics. The decrease in sequencing cost and the accelerating sequencing speed are bringing genomics closer to everyday clinical use by enabling accurate and fast genetic analyses [5, 6]. Sequencing technologies can advance various disciplines, from human genetics to microbiology and personalized medicine [7, 8]. Whole-genome sequencing (WGS) analyses have revealed the genetic variation and their associations with diseases in humans and other species on a genome-wide scale [9]. As a consequence of advancing DNA sequencing, the quantity of sequencing data is growing at a rapid pace and the number of assembled whole-genomes is increasing quickly as well. It has been estimated that 2-40 exabytes of storage space will be required just for human genomes by the year 2025 [10].

At present, the race between sequencing data throughput and computing speed is ongoing, widening the gap between sequencing data volume and analysis results [11]. The existing computational methods used in

NGS analysis are partly slowing down the computational analysis output, and thus restraining research and development, a situation also known as the bioinformatics bottleneck [12, 13]. To boot, repealing Moore's law and the end of Dennard scaling have slowed down the sequential CPU performance improvement rate [14]. Performance gain is now achieved by adding more cores to the system to increase transistor count, and thus scalability to a high number of cores is the major driver in distributed and high-performance computing. To utilize distributed multi-core processors efficiently, the fundamentals of distributed and parallel programming have to be applied to bioinformatics software and algorithm design. However, shifting from single workstations to distributed computing with current bioinformatics tools is challenging, as the current bioinformatics tools, algorithms, storage formats, and pipelines have been designed under the premise of sequential computing [15]. Thus, the relevance of developing and harnessing novel scalable computing methods for bioinformatics and computational genomics is more significant than ever before.

This dissertation aims to alleviate the accumulating biological data analysis burden by applying scalable distributed computing methods to the tools and pipelines routinely used in bioinformatics and computational genomics. Genomic data consist of repetitive sequences that support developing efficient compression algorithms and of structures that enable distributed and parallel data processing, indexing, storage, and retrieval methods [16, 17]. All this fused with the routine bioinformatics methods, e.g., sequence matching, read alignment, genome assembly, genotype imputation, and variant calling offer a powerful toolset to release the increasing bioinformatics analysis burden. Moreover, modularity plays a key role in designing robust pipelines and reproducible analyses from reusable data processing components.

Distributed computing is a promising, if not the only option, in current technologies for overcoming the computational challenge of analyzing ever-growing sequence data volumes. However, designing and developing parallel programs for distributed computing clusters is far more complex than for conventional computers, especially so when support for existing sequential bioinformatics methods and data formats must be considered. The programming methods learned to implement sequentially executing programs do not apply directly anymore. This dissertation emphasizes the design of efficient genomic analysis pipelines to accelerate the data processing phases using distributed CPU computing. In this dissertation, scalable distributed methods are developed for compressing and indexing genomes, sequence alignment with compressed genomes, genotype imputation, min-

ing metagenomes from multiple NGS samples, and assembling pan-genomic reference sequences. The methods developed are demonstrated in a computing cluster with practical applications.

1.1 From DNA to genome sequencing

Deoxyribonucleic acid (DNA), the code of life, is the basis of all living organisms [1]. DNA stores an instruction set, genes, to code for proteins and make it possible for organisms to function, and thus to live. A DNA molecule consists of a long nucleotide sequence including four types of bases: A (adenine), C (cytosine), T (thymine), and G (guanine), forming base pairs A-T and C-G. In humans, these double-stranded DNA sequences are stored in a cell in 23 chromosome pairs where one chromosome is inherited from the father and another one from the mother. A genome is a collection of all the DNA in a cell. The cell machinery processes the DNA to construct needed proteins in a process known as protein synthesis, which is regulated by other proteins and complex chemical processes [18]. Simplified, the cell machinery is triggered to initiate gene expression, to read the instructions from the gene transcription start site, to transcribe the genes to ribonucleic acid (RNA) molecules, and further translate the RNA to output amino acids forming the proteins. Each amino acid contains three nucleotides forming codons, thus there exist 64 different codons [18]. However, because of their chemical properties, multiple codons can represent the same amino acid, and thus 20 different amino acids appear in the DNA. The rate of gene expression is regulated by transcription factors, which are also proteins, and the rate of expression influences the amount of final protein product. Genes and the rate of their expression can also regulate other genes, making biological processes very complex. Here, the analogy to computer science can be seen with the difference that computers process binary sequences, consisting of ones and zeros, instead of base-pair sequences. The cell can be then imagined as a central processing unit (CPU) and a set of genes can be imagined as a computer program that can have multiple functions depending on the expression of the genes and proteins produced. All this is executed simultaneously with multiple genes in parallel by trillions of distributed cells in this living 'computer'.

Advances in sequencing technologies have enabled us to read the DNA bases. The first complete human genome draft was published in 2004 by the Human Genome Project (HGP) [2]. The human genome comprises approximately 3.2 billion base pairs and is estimated to include 30 000-40 000 protein-coding genes [18]. However, only a tiny proportion of the genome

makes us different: on average, the base pairs of two individual genomes differ only by 0.5% [19]. This draft genome has been complemented from time to time and it has become a standard reference genome that is used as a comparative reference in the majority of scientific contributions in human genetics. The first reference genome version, GRCh36, is a mosaic of genomes assembled from thirteen volunteer donors. Therefore, it has been argued to represent the variation of various ethnicities across the world poorly but has lately been complemented with multiple donors and more diverse ancestry [20].

The first genomes in the HGP were sequenced with the early Sanger chain-termination-based sequencing [21] technology, capable of sequencing 160 kilobases per day at a cost of 0.75 dollars per base. However, Sanger's technology has been refined since [22, 23] and is still in use in many laboratories. Next-generation sequencing (NGS) [4] technologies are successors to Sanger technology. While the Sanger method is capable of sequencing a single DNA fragment at a time, NGS sequences millions of fragments simultaneously in parallel. The latest NGS sequencers have reached throughput of up to terabases per day at a cost of 0.05 dollars per million bases, which is approximately a million times faster and 10 million times cheaper than the first sequencers (Figure 1.1). To put it in scale, approximately 1.3 human genomes were sequenced in the year 2005 and 18,000 genomes were sequenced in 2014, while the cost per genome was 10 million dollars in 2005 and fell almost to 1000 dollars in 2015.

NGS has advantages such as higher sensitivity, higher sample volumes, greater genomic coverage, and higher throughput compared to previous sequencing technologies. NGS can be divided mainly into whole-genome,

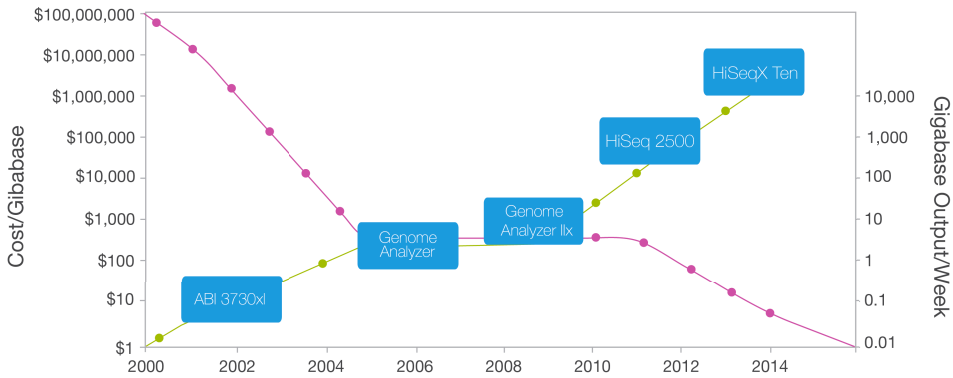

Figure 1.1: Next-generation sequencing data output rate versus sequencing cost on Illumina NGS platforms [24].

-exome, -transcriptome, and targeted sequencing methods, where whole-genome and -exome sequencing is applicable for DNA and the other methods for RNA detection [4]. Depending on the genomic study, the inspector can choose from prepared NGS analysis libraries or prepare a custom library based on species, genomic target, and a sample type such as blood, cell culture, or fresh-frozen tissues.

An NGS sequencer produces short sequences, called reads, from the original DNA, covering each genomic position multiple times. Sample DNA is first broken up into small fragments (from a few hundred to a few thousand bases), typically with enzymes depending on the technology used [4]. The fragments are then sequenced in a single-read or paired-end mode, with paired-end sequences on both ends of the DNA fragment providing more accurate alignment capabilities and improving detection of insertions and deletions [25]. NGS sequencing loses the information of which chromosome or DNA strand the detected sequence originates from and the origin can be detected only with computational methods using reference genomes [25]. Sequencing read length is set while preparing the library and is typically between 50 bp to 100 bp in RNA sequencing and 100 bp to 300 bp in DNA sequencing. The output volume of a sequencing run depends mostly on the genome size, the coverage, and the read length settings. NGS produces approximately 0.1-1 terabytes of data per human genome [4].

Third-generation sequencing (TGS) technologies are currently being actively developed by research institutions and private companies. The fundamental difference between NGS and TGS technology is that TGS can sequence longer DNA fragments continuously and produce sequence data in real-time. The current dominating TGS technologies are PacBio Single-molecule Real-time sequencing (SMRT) and Oxford Nanopore sequencing [26, 27]. The major improvement is that TGS produces longer reads, typically ranging from tens to hundreds of kilobases, offering more accurate genome assemblies [28]. At the moment, the weaknesses of TGS technology are low throughput, high error rates, and high cost. This dissertation focuses mainly on NGS-based methods, but most of the computational methods studied are potentially applicable to TGS data. The distributed and parallel computing methods studied in this dissertation can potentially be applied to upstream analysis pipelines using tools developed especially for long-read alignment and assembly. Moreover, some of the routinely used read alignment and *de novo* assembly methods developed for short-reads are applicable to long-reads as well, but should be comprehensively assessed if applied [29, 30].

1.2 Problem statement

We address these three main research questions:

I How to find an optimal problem and data decomposition that maximizes the scalability of a distributed bioinformatics pipeline.

II How to scale routine bioinformatics methods using the existing genomic data structures with distributed computing.

III How to improve the sequence alignment to perform at pan-genome scale and alleviate the storage burden using compressed indexes in combination with distributed computing.

Typical data volumes are massive at upstream in a bioinformatics pipeline and decrease through the analysis towards the downstream. Analysis pipelines tend to be both I/O and CPU bound where preprocessing steps represent the former and analysis steps the latter. Optimizing computing performance in bioinformatics pipelines requires minimizing the computing time consumed in all processing steps to maximize the throughput. Genomic properties allow us to decompose data into distributable partitions, e.g., by sample, chromosome, genomic region and locus, genes, or by *k-mers*. Segmentation of database or sequence queries may suggest an alternative option for decomposing the computing task into parallel tasks. These properties provide versatile data partitioning schemes in distributed file systems, thus enabling efficient data retrieval and highly granular parallel processing. However, most of the generic algorithms and file formats for genomics were not originally designed for distributed file system data partitioning schemes.

Manipulating big data in parallel on conventional shared-memory computers is not feasible for many reasons. Parallel computing with a fixed number of cores does not satisfy big data analytics goals, as computing time can get unbearable while data volumes grow. To keep the computing time low, more cores need to be assigned to complete the tasks. Scalable distributed programming models [31, 32] have been developed to foster adaptation of the data-parallel paradigm and distributed computing.

Appropriate problem decomposition and data partitioning are often the first and perhaps most important steps towards transforming an existing sequential bioinformatics pipeline into a scalable one using distributed and parallel computing. Data partitioning is a fundamental pre-processing phase in the distributed data processing pipelines, and greatly affectins the performance of implemented analysis algorithms as well. The computational problem is decomposed into tasks that are executed simultaneously

on multiple processors. Problem decomposition requires the identification of tasks and their relation to input data structures that expose parallelism. In the data-parallel approach, identical CPU-intensive tasks can be executed in parallel on different partitions of the underlying data structure. Assuming that data partitions implement the same data structure, the data can be partitioned and distributed to the distributed filesystem for data-parallel computing. Data partitioning combined with a distributed filesystem enables data to be manipulated locally in relation to the CPU without loading data over a network. Apache Spark [32] automates data distribution and parallel processing using internal distributed abstract data types. Spark operates on distributed memory by reading the data partitions into a distributed data structure, thus enabling fast data access and transformations. Data processing pipelines are described as a sequence of data transformations where each transformation passes manipulated data to the following pipeline stage.

We are interested in the problem decomposition, data partitioning, data structures, and data transformations that help to minimize the communication overhead, the number of I/O operations, and memory footprint, and also to maximize CPU utilization. Transforming sequential analysis pipelines to run on distributed computing clusters using optimized partitioning and data transformation methods can greatly improve next-generation sequencing data analysis throughput and scalability. These methods have been studied in Publication I for scaling metagenomic data-mining pipelines to identify viruses from several human samples in parallel on an Apache Spark computing cluster.

Most of the data structures and file formats used in bioinformatics applications have not been developed under the premise of parallel computing and distributed file systems, restricting the direct exploitation of distributed storage solutions. The Hadoop-BAM [33, 15] library has enabled the distributed manipulation of SAM (Sequence Alignment Map format), BAM (binary SAM) [34], VCF (Variant Calling Format) and BCF (binary compressed VCF) [35] files relying on the Hadoop Distributed File System (HDFS) [36]. Disq[1] is the successor project for Hadoop-BAM, developing better Apache Spark compatibility. Gzip compatible block compression method (BGZF)[2] allows random access to BGZF compressed BAM and BCF files. BGZF compressed files can be read in blocks from the given offset without decompressing the entire file. Tabix indexing enables the partitioning of BGZF compressed files on distributed file systems and the

[1] https://github.com/disq-bio/disq
[2] https://samtools.github.io/hts-specs/SAMv1.pdf

retrieval of portions of compressed data for efficient parallel processing. The data-parallel distributed genotype imputation method is applied to a large-scale genotype imputation problem in Publication II.

Compressing and constructing genomic indexes that are compatible with direct sequence alignment with common bioinformatics tools can enable large-scale pan-genomic analyses in practice in tolerable time. Advanced genomic data compression methods can contribute greatly to the scalability and analysis throughput, assuming that the compressed data is randomly accessible without decompressing the entire data set. Accessing the data efficiently requires the indexing of massive genomic datasets, which is computationally expensive. In bioinformatics applications, it is important that the compressed index is compatible with routinely used tools such as BWA [37], Bowtie [38], and BLAST [39] aligners. A variant of essential lossless data compression algorithm Lempel-Ziv, named Relative Lempel-Ziv (RLZ) [40, 41], has been found to be efficient in compressing repetitive sequences like genomes. RLZ is based on the idea that with a highly similar set of sequences, a relatively good compression ratio can be achieved by using a prefix of the data set as a reference to compress the rest of the set. Moreover, RLZ compressed genomes can be indexed with the hybrid-indexing method and accessed with conventional sequence aligners [42]. The methodological background of distributed RLZ is reviewed in Section 3.6. The distributed RLZ compression is used with the hybrid-indexing method to compress and index pan-genomes, enabling fast read alignment and sequence searching in Publication III. Compressed hybrid-indexes are used for assembling novel reference genomes from pan-genomes in Publication IV.

1.3 Contributions

The main contributions of this dissertation are given in the original Publications I-IV as follows.

I In this paper, distributed and parallel computing methods are studied to increase throughput and scalability on detecting viruses from multiple human samples sequenced by NGS technologies. The methods studied are applied to solving the computational problem in mining and identifying viral metagenomes. It has been predicted that10^{12} microbial species exist on Earth, and 10^5 species exclusively in the human microbiome [43]. Performing large-scale metagenomic studies on ever-growing NGS data volumes is computationally demanding and too time-consuming with the traditional sequential analy-

sis pipelines. ViraPipe speeds up the identification of metagenomes by distributing read alignment, contig assembly, and database search phases in the High-Performance Computing (HPC) cluster. This work is based on the previous research on discovering virus-disease associations from NGS samples carried out by Bzhalava D, Bzhalava Z, and Dillner J from the Department of Laboratory Medicine, Karolinska Institutet [44, 45, 46, 47]. Maarala AI and Bzhalava D carried out the software design and implementation and wrote the manuscript together. Maarala AI, Bzhalava D, and Bzhalava Z performed the experiments and evaluations. Data curation and Spark cluster management were carried out by Bzhalava D and Bzhalava Z. The research was supervised by Heljanko K and Dillner J. This paper addresses Research Questions I and II as presented in Section 1.3.

II Data-parallel genotype imputation and distributed data compression methods have been studied in this paper and applied in practice to the implemented genotype imputation tool, SparkBeagle. The main goal is to respond to future scalability needs in determining genetic variation amongst a massive number of individuals from rapidly growing genotyping datasets. A distributed imputation method is implemented and trialed with 2500 human genotypes in a cloud computing cluster, with promising results. The work was carried out in collaboration with the Institute for Molecular Medicine Finland (FIMM). The overall design was carried out by all the authors. Maarala AI implemented the distributed imputation tool and performed the scalability experiments. The imputation accuracy assessment was designed and carried out by Pärn K and Nuñez-Fontarnau J. The paper was written by all the authors together. The research was supervised by Heljanko K. This paper addresses Research Questions I and II as presented in Section 1.3.

III This paper addresses distributed genomic data compression methods and indexing genomic data collections for efficient sequence alignment purposes. Compressing and indexing massive collections of genomes, and searching sequences from such collections, is a computationally demanding and time-consuming task. The distributed version of compressed hybrid-indexing and Relative Lempel-Ziv (RLZ) compression originally presented by Valenzuela et al. [48] have been designed and implemented in this paper. The data-parallel methods from Publications I and II have been applied to the RLZ compression and hybrid-indexing techniques in this work. Maarala AI and Arasalo O carried out the software design and implementation with the advice of Valen-

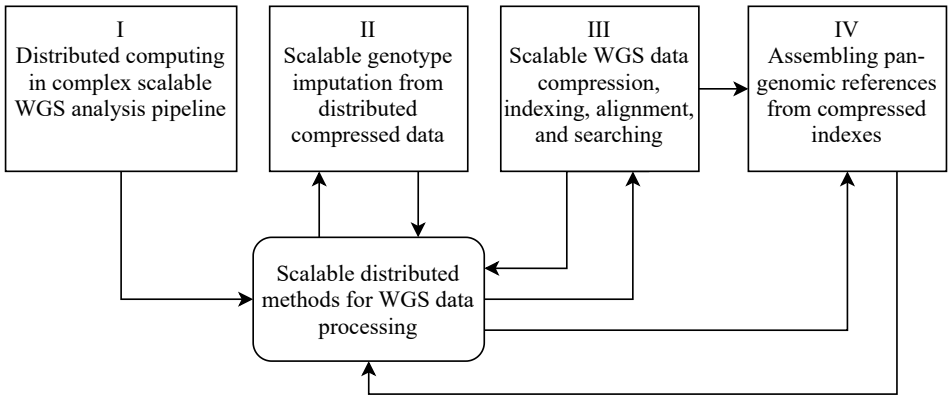

Figure 1.2: Publications and their contextual relations in this dissertation.

zuela D. The paper was written by Maarala AI, Arasalo O, and Heljanko K. The research was supervised by Mäkinen V and Heljanko K. This paper addresses Research Questions II and III as presented in Section 1.3.

IV In this paper, distributed pan-genome compression and hybrid-indexing methods from Publication III have been harnessed for scalable assemby of reference pan-genomes and enabling of variant calling over compressed pan-genomes. The original idea of calling variants over pan-genomes was presented by Mäkinen et al. [16]. A distributed reference pan-genome assembly pipeline has been implemented in this work based on the sequential pipeline presented by Valenzuela et al. [49]. Maarala AI and Arasalo O designed and implemented the software with the advice of Valenzuela D. The paper was written by Maarala AI, Arasalo O and Heljanko K. The research was supervised by and Mäkinen V and Heljanko K. This paper addresses Research Questions I, II, and III as presented in Section 1.3.

1.4 Dissertation outline

The main concepts in genomics and modern DNA sequencing were presented briefly in this chapter with the motivation and introduction to the problem area.

Chapter 2 presents fundamentals of the design of distributed and parallel algorithms and programming frameworks with regard to genomics. Chapter 3 introduces the basic concepts and routine methods used in high-

throughput sequencing data analytics, including genomic data compression and indexing techniques. Chapter 4 presents the fundamentals of the design of scalable high-throughput sequencing data analysis pipelines. The distributed computing methods from Chapter 2 are applied to routine sequence analysis methods introduced in Chapter 3. The method applieds are presented with examples through the illustrated analysis pipeline stages. Chapter 5 introduces four specific population genomics applications implemented using the methods presented in the former chapters. Conclusions and future research directions are presented in Chapter 6.

Chapter 2

High-performance computing in genomics

Distributed and parallel computing frameworks enable scalable, reliable, efficient, and relatively low-cost computing in the cloud and high-performance computing (HPC) clusters. Parallel data analysis with multiple distributed computing nodes can deliver a huge performance advantage compared to single workstations. Cloud services provide infrastructure for deploying computing clusters in a flexible and cost-effective manner. Big data infrastructures offer appropriate platforms and tools for processing and analyzing massive and diverse datasets that are computationally too expensive for conventional computing [50].

The characteristics of genomic data structures allow the computational problems to be decomposed into small granular tasks that can be executed in parallel without rewriting the inner loops of the complex algorithms. Decomposing problems into distributable data partitions, e.g., by species, chromosomes, bacterial strains, samples, or gene variants enables data to be stored in distributed file systems and processed flexibly and efficiently in parallel in computing clusters [51].

The main idea in distributed and parallel computing is that independent tasks of the program or algorithm can be executed concurrently on different processing units, thus reducing the computation time. Computing in general consists of a system providing an execution and programming environment for programs that implement algorithms for processing data. In the traditional Von Neumann model [52], the instruction set (program) and data are loaded into memory and the central processing unit (CPU) fetches the instructions and data from the memory and executes the instructions, i.e., arithmetic operations on the data. A sequential algorithm

contains several instructions that are executed in a serial process whereas a parallel algorithm executes the instructions simultaneously.

Modern computer architectures allow concurrent instruction execution with multiple processing units using multi-core processors. The concurrent processing architectures use either shared memory, which can be accessed simultaneously by processing units, or distributed shared memory (DSM), where each processing unit has a separate memory. Symmetric Multiprocessor (SMP) is a common shared-memory architecture that shares the same physical memory space symmetrically amongst all cores through the same bus [53]. SMP provides the same view of the data in memory for all processors while DSM memory blocks are private for separate processors. Most modern multi-core computer architectures use Non-uniform Memory Access (NUMA), where each core has its local physical memory and also has access to other core's memory blocks, albeit with longer latency than accessing local memory; thus memory access time is dependent on the physical memory location of the data [53]. Concurrent execution typically requires more physical memory than sequential, as simultaneous execution increases the system's total memory usage. Distributed memory architectures are typically used in high-performance computing clusters consisting of multiple interconnected computing nodes, where each node is an SMP or a DSM system [52].

Alongside CPU computing, General-purpose computing on graphics processing units (GPGPU) and Field-programmable gate arrays (FPGA) offer fine-grained data-parallel and energy-efficient computing [54]. FPGAs are integrated circuits developed for accelerating special computing tasks that make them highly optimized and efficient, but also relatively slow to design, hard to program, and expensive. However, FPGAs are becoming more reconfigurable, i.e., with reconfigurable fabrics, offering configurable logic blocks, digital signal processors, input/output interfaces, and memory blocks in a single chip, i.e., in a System-on-Chip [55]. GPUs fall into the category of Single Instruction Multiple Data (SIMD) processor architectures in Flynn's taxonomy[1], where the same operation is executed on multiple data elements that have similar data structures in parallel [53]. As GPUs use relatively less power than CPUs, more cores can be packed into a single GPU for offering massively parallel computing on a single chip [54]. However, GPU computing is not displacing CPU computing; instead, they complement each other, and in common heterogenous SMP systems GPUs work under the control of CPUs. GPUs have evolved from graphics processing to general-purpose needs, and thus, they are very ef-

[1] https://en.wikipedia.org/wiki/Flynn's_taxonomy

ficient with properly decomposed compute-intensive arithmetic tasks such as matrix operations [54].

2.1 Distributed and parallel algorithm design

Distributing workload to multiple cores requires a strategy for partitioning the computation and is crucial for efficient parallel execution. The partitioning can be done either for data or computation or both. The computational problem is decomposed into tasks that are executed simultaneously on multiple processors. *Problem decomposition* requires the identification of tasks and data structures that expose parallelism [52]. Appropriate task and data decomposition patterns allow the design of rapid data flows through the distributed data processing pipelines and affect the whole application development cycle. Problem decomposition depends greatly on the data structures, algorithms, control flow, and computing architecture. The input data structures can vary between separate pipeline stages, requiring different data decompositions for optimal distribution to ensure scalability. In a task-parallel approach, the problem is decomposed into independent tasks that can be executed in parallel, and pipelined tasks that are executed sequentially (task decomposition) [52]. In a data-parallel approach, the identical tasks are executed in parallel on different partitions of the same data structure (data decomposition).

Decomposition can be further divided into fine-grained and coarse-grained approaches [52]. Granularity affects load balancing as the workload generated by smaller tasks can be distributed more evenly. Fine-grained parallelism considers decomposing the algorithm into subtasks operating on the smallest possible data elements that can expose parallelism, such as matrix cells, whereas coarse-grained parallelism involves operating on data partitions including multiple data elements such as matrix rows or columns. Shared-memory systems have certain advantages over distributed memory systems in implementing fine-grained parallelism, such as faster intercommunication between the processes and memory blocks .

The data-parallel approach is applicable when independent parallel tasks at the same stage do not exchange data, and thus there is no need for synchronized access to shared variables [52]. Data-parallel algorithms fit well with distributed memory architectures as each node works on its local data partitions. In complex pipelines, hybrid decomposition methods may be required in the partitioning stages. For example, in distributed memory architectures, distributed data partitions are processed locally on separate nodes and fine-grained parallelism can be implemented for each local data

partition (data locality) using local shared-memory space. Caching is a technique to improve memory access performance, where cache memory is used as a temporary storage for the data that is accessed frequently. Increasing data locality is a key principle used in cache optimization. Data locality can be temporal or spatial, where the former one assumes that recently used data will be used also in the near future, and the latter one assumes that nearby memory addresses of accessed data element will be accessed in the near future [53]. For improved read/write efficiency, cached data is typically stored in tagged blocks (cache blocking) and further in sets of blocks that can be searched in parallel. Simplest way to reduce cache misses is to increase the block size or cache memory size. On the other hand, the bigger block size increases the miss penalty (time to replace the block increses) and bigger cache size can lead to longer hit times and higher cost [53].

Minimizing memory footprint is important and it can be optimized through algorithm design, efficient data structures, and data transformations. The performance is also affected by the number of I/O operations and communication overhead, which can be minimized by appropriate problem decomposition and data partitioning. Figure 2.1 presents a data-parallel approach for distributing genotype data by coarse-grained chromosomal partitions and further partitioning chromosomes to regions for fine-grained parallelism.

Assuming a computer has n processors, the parallel process runtime should be divided by n in an ideal case. However, the processes in distributed and parallel systems need to interact with some communication

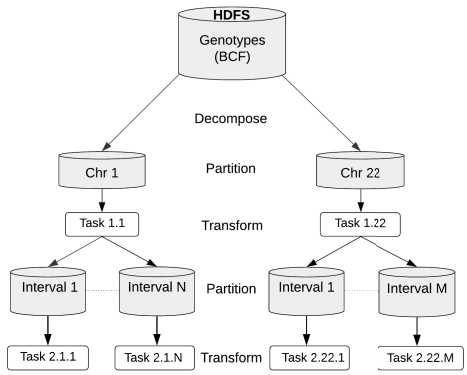

Figure 2.1: Partitioning genotyping data by chromosomes and chromosomal regions.

mechanism. Thus, the parallel performance is affected by interacting processes generating communication overhead. The communication is managed through an interconnection network that handles the communication and synchronization between the tasks and stages. An intercommunication network connects processing elements such as processor and memory blocks in a shared-memory system and nodes in a distributed memory architecture. Typically, high-speed ethernet, such as InfiniBand or Gigabit Ethernet, is used as an intercommunication network in high-performance computing clusters.

Distributing data partitions for parallel processing tasks increases the communication between the computing units. In Figure 2.1, the independent parallel tasks do not exchange data, thus synchronization can be omitted between the tasks at the same stage. However, some synchronization mechanism is required between sequential tasks because Tasks 2.1.x have to wait for Tasks 1.x to be completed. Sometimes tasks can be combined to reduce the communication overhead and minimize the I/O operations: for example, many filtering and data pre-/post-processing steps can be combined within parallel tasks, assuming that there are no data dependencies between the data partitions. In sophisticated distributed programming frameworks, such as Apache Spark [32], a task scheduler takes care of control messaging and synchronization, and parallel abstraction layers are hidden. In such frameworks, the developer simply defines a sequence of data transformation functions that process data partitions and the framework handles the synchronization between the transformation stages.

The granularity of decomposition greatly affects parallelism, i.e., the more parallel tasks the more speedup, but also the more communication overhead is generated. Communication can be decreased by coalescing tasks into larger ones at the expense of parallelism and load balancing. For example, the data in our example (Figure 2.1) can be decomposed into fewer partitions by using wider chromosomal regions.

Input/output (I/O) and data transformation operations increase the communication overhead between the processing steps as large data volumes are distributed and read from the disk into memory. The amount of computing work may not be evenly distributed between the tasks, causing a *load balancing* problem where the most loaded tasks stall the execution (while other processors are idling). With massive data volumes, uneven workload distribution may become prominent [56]. In the optimal case, the number of tasks is equal to or greater than the number of available cores at each parallel stage, but the available memory often becomes the limiting factor in CPU-intensive computing. In such cases, we can increase

the granularity by using smaller data partitions to decrease memory usage. In turn, too few partitions can lead to load imbalance and suboptimal performance.

Data aggregation is an essential process in between the tasks of a distributed data processing pipeline, as partitioned data often needs to be collected, merged, joined, or reduced after each processing step. The aggregation step stores intermediate or final results and may include data processing as well, but with the combined data partitions. Moreover, the data partitions can be aggregated from the tasks that have already been processed to release CPU time for the unfinished tasks, e.g., when the number of tasks is greater than the number of processors. MapReduce [31] is a powerful data-parallel programming paradigm intended for processing large data sets and it provides easy to use templates for programmers to execute code in parallel in a distributed computing cluster. In the MapReduce programming model [31], the map stage assigns the local data partitions in a data node for parallel task execution, and the reduce stage aggregates intermediate results by given data elements such as keys in ⟨ key, value ⟩ pairs and continues computing with aggregated data sets in parallel (Figure 2.2) [57].

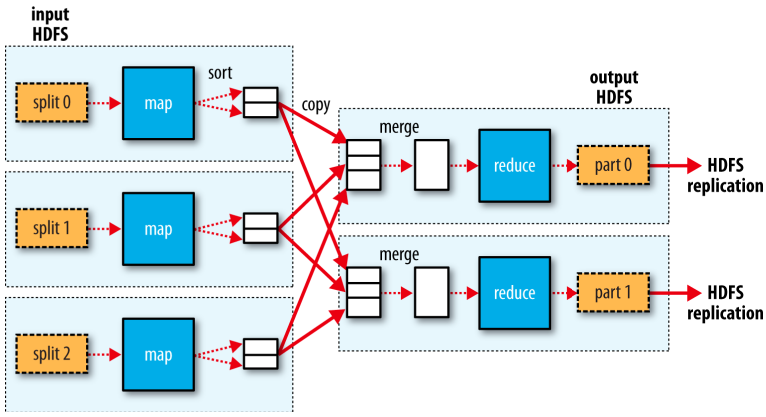

Figure 2.2: MapReduce method [57].

2.2 Data-intensive computing frameworks

Modern distributed computing frameworks foster parallel program execution in distributed computing clusters. Apache Spark [58] and Apache

Hadoop[2] have become common frameworks in big data analytics and cloud computing, offering a powerful toolset for scalable data processing. Apache Hadoop enables the execution of MapReduce programs [57] with large datasets stored in the Hadoop Distributed File System (HDFS) [36]. In a Hadoop computing cluster, each node acts as an independent processing unit with its internal physical memory, CPU, and disks, thus no resources are shared between the nodes, i.e., shared-nothing architecture. Efficient mechanisms are usedto minimize data transfer between the nodes through data replication, providing fault tolerance [31]. In case of failure, the independent tasks even can be re-scheduled to a different node thus making MapReduce programs robust. Spark is closely related to Hadoop and is often used on top of the Hadoop platform as it can be integrated with the HDFS and YARN[3] resource manager provided with the Hadoop stack. However, Spark uses its own Direct Acyclic Graph (DAG) [32] execution engine instead of MapReduce. Spark loads the input data into distributed memory partitions and accelerates data analysis with in-memory data processing, where distributed working data sets can be reused and pipelined from one pipeline stage to another. In-memory processing enables multiple times faster data-parallel computing compared to MapReduce, in which the data is loaded from the disk between the pipeline stages [32]. A Spark job is launched with a DAG engine, which defines the execution order of tasks and stages in a Spark job [32].

In Apache Spark, serialized code is moved to the data to be executed as soon as possible and the data is loaded fully in distributed memory (RDD) if it fits, otherwise the data is spilled to disk producing unwanted I/O operations and latency. Scheduling of task execution in Spark is based on the data locality principle and the locality level can be configured to process-local (data in the same Java Virtual Machine as the running code), node-local (data on the same node), rack-local (data on the same server rack), or any (data can be anywhere)[4]. Data locality level affects greatly to the amount of data transfer, and thus, to the performance. If no free execution core has been found at the given location level (e.g., on the same node), Spark waits for cores to be freed for a configured period of time. If the wait period expires, Spark broadens the locality level requirement and moves the data to other JVM, node, or rack that has free execution cores.

Data models in Spark allow distributed in-memory data manipulation between the pipeline stages through higher-order transformation functions.

[2]https://hadoop.apache.org

[3]https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[4]https://spark.apache.org/docs/latest/tuning.html#data-locality

Distributed computing in Spark was originally based on the Resilient Distributed Dataset (RDD) [32], which stores the collection of data that is partitioned into the distributed memory of computing nodes in a cluster. DataFrame and Dataset are successors of RDD, and offer better support for structured data processing and querying. Distributed data is processed in parallel tasks managed by a task scheduler, where the tasks are launched by Spark executors in data nodes. Distributed computing is executed on local parts of the distributed data set on each node. Each node assigns an executor or executors for its local tasks, which are run in parallel on multiple cores of the node. Caching enables iterative analysis, where the same working sets of data can be reused several times within the same analysis job. Moreover, HDFS [36] offers distributed data storage for Spark and provides fault tolerance through data replication. Files in HDFS are stored in fixed-size blocks that are distributed and replicated over the computing cluster. These blocks can be read into RDDs and processed in parallel tasks by Spark executors. In case of a task failing, the task is re-scheduled and run on the replicated data partition without re-launching the whole job. Spark provides functional programming options with Scala, Java, Python, and R programming languages.

Message Passing Interface (MPI) [59] has been a dominant framework in high-performance computing clusters. MPI offers great control for the programmer through low-level programming routines but is complex to use in complex pipelines, as it requires the programmer to control the data flow explicitly with its communication protocol. MPI is well applicable to fine-grained parallelism and compute-intensive work, although it relies on storage area network (SAN) controlled shared filesystems that can become a bottleneck with very large datasets as communication overhead becomes prominent [57]. A fundamental difference in data-parallel programming is that the programmer is intended to think in terms of data-level parallelism and avoid moving the data over the network, i.e., favor data-locality. Moreover, the data-parallel approach fosters the design of scalable applications in terms of increasing data volume and computing resources.

Real-time stream processing engines such as Apache Flume[5], Kafka[6], Flink[7], and Spark Streaming[8] can be potentially used in complex batch processing pipelines to avoid intermediate data staging and read/write latency between the pipeline phases. Load balancing could benefit from stream processing as the following phase does not have to wait the previous phase

[5] https://flume.apache.org/
[6] https://kafka.apache.org/
[7] https://flink.apache.org/
[8] https://spark.apache.org/docs/latest/streaming-programming-guide.html

to be finished. Stream processing typically uses memory buffering for storing incoming data, and thus, can waste available memory used in batch processing phases if a proper scheduling has not been implemented.

Together with CPUs, distributed systems can be accelerated greatly with GPU computing, where each processing unit type can have different roles. Appropriate problem and data decompositions that consider workload management play important roles in such heterogeneous systems as the workload should be distributed evenly over several CPUs and a high number of GPU cores, and possibly in a distributed computing cluster where each node comprises multiple CPUs and GPUs. The computational power of a GPU is mainly driven by a massive number of GPU cores and threads. GPUs follow the Single Instruction Multiple Thread (SIMT) programming model, where multiple threads execute the same program code with different data elements in parallel [60]. GPUs consist of multiple computing units called streaming multiprocessors (SM) that contain multiple GPU cores. In a GPU program, threads are organized into blocks and thread blocks are assigned to SMs that process threads simultaneously. Theoretically, each GPU core processes one thread simultaneously and private memory space and register are allocated for each running thread [60]. A thread block size is defined by the programmer and each block reserves its own address space from the shared memory, which allows access to thread-local memory simultaneously from multiple threads (i.e., memory coalescing), which increases the memory bandwidth [60]. In practice, available memory restricts the thread block size, and a new block is not processed until there is enough free shared memory and free registers for all threads in a block [61]. Simultaneous memory access from multiple threads can cause correctness issues, making software verification more challenging [62]. Currently, the two dominant GPU manufacturers are Nvidia and AMD. Nvidia provides the CUDA[9] parallel computing development platform, where programs are written in CUDA C language. The Nvidia Ampere A100 includes 6912 CUDA cores whereas Intel Xeon Platinum 8380 includes 40 cores per CPU. Theoretical maximum memory bandwidth in the Nvidia A100 is 1555 GB/s whereas in Intel Xeon Platinum 8380 with eight DDR4 3200 memory channels it is approximately 204.8 GB/s. Recently, Nvidia has added tensor cores in GPUs to speed up matrix calculations that foster, e.g., deep neural network training [63, 64].

[9]https://developer.nvidia.com/cuda-zone

2.3 Frameworks for computational genomics

Leveraging the accumulating genomic data in analytics requires high-performance computing including a large amount of storage space, working memory, computing power, and network [10]. Big data infrastructures, cloud computing frameworks, distributed filesystems, and databases have evolved while the price of DNA sequencing, data storage, and computing memory has fallen, and now it is economically viable to move on to distributed and cloud computing. To exploit NGS data efficiently in genomic studies, it is essential to compress and store genomic data sets in readable and reusable form for bioinformatic analysis tools [10]. Moreover, these enormous datasets should be stored and structured in distributed formats so that they can be processed, queried, and analyzed efficiently in parallel.

Broad Institute's Genome Analysis Toolkit (GATK)[10] is an open-source framework designed for efficient and scalable genomic data processing and analysis, originally powered by Hadoop MapReduce [65]. Apache Spark [58] is used in the latest GATK [66] and Adam[11] to improve scalability. Genomic data is partitionable at the chromosomal level and genomic regions, which enables distributed data processing in the read alignment, genome assembly, genotype imputation, and variant calling steps (see Chapter 3). Most of the generic file formats and algorithms for genomics are not designed for distributed systems and binary file formats in particular are not intrinsically distributable. However, Hadoop-BAM [33] is a library originally developed for distributed processing of common genomic file formats on HDFS. It has been developed further purely in Spark under the Disq[12] project. Hadoop-BAM includes the Hadoop Input/Output interface for distributing genomics file formats into HDFS and tools for sorting, merging, and filtering of genomic data formats including BAM, SAM, CRAM, FASTQ, FASTA, QSEQ, BCF, and VCF files. The GATK[13] genomic analysis framework adopted Hadoop-BAM as its parallel I/O interface, which has recently been replaced by its successor, Disq. Hadoop-BAM is also used in other genomics tools such as Adam[14], Halvade [67], Halvade-RNA [68], Seal [69] and SeqPig [70]. Al-Ars et al. propose SparkRA [71] for scaling GATK RNA-seq Pipeline with Apache Spark, and they achieved 7.7x speedup while using 16 nodes. SparkRA uses a similar input read data partitioning approach as was used in Halvade [67] and in Publication I by

[10] https://gatk.broadinstitute.org/
[11] https://github.com/bigdatagenomics/adam
[12] https://github.com/disq-bio/disq
[13] https://gatk.broadinstitute.org/
[14] https://github.com/bigdatagenomics/adam

extending the read alignment phase with static and dynamic load balancing methods. However, the premise of distributed read alignment in SparkRA and in the method used in our pipeline differ in that SparkRA uses RNA-seq data and ViraPipe uses WGS data for different use cases. General workflow managers such as Apache Airflow[15] can be utilized in complex pipelines for scheduling and monitoring tasks, and for managing workflows dynamically using web GUI and Directed Acyclic Graph presentations. More bioinformatics specific workflow engines such as Nextflow [72] and Snakemake [73] have been also developed for alleviating effort of developing reproducible analysis workflows and for running bioinformatics pipelines on different computing platforms.

Galaxy is a versatile bioinformatics analysis platform and workflow manager that provides web-based user interface for building and controlling workflows. Galaxy integrates several bioinformatics tools that can be combined in a workflow dynamically and workflows can be run on cloud computing clusters and deployed with provided virtual machine images [74].

The technological development of GPGPUs and FPGAs have recently increased the adaptation of hardware acceleration to bioinformatics applications [75, 76]. The popular BLAST [39] sequence aligner (see Section 3.2) has been accelerated by 14.80x with GPU in G-BLASTN aligner [77] compared to sequential BLAST running on a single-core CPU. The Arioc read aligner [77] shows 25x speedup using 8 Nvidia v100 GPUs compared to the Bowtie2 [38] aligner running on Intel Xeon Platinum 8175M CPUs (48 cores). Ahmed et al. [78] are using the Apache Arrow framework to enable parallel SAM [34] file processing on GPUs such as alignment sorting and removing duplicate and low quality reads. They have integrated the tool, ArrowSAM, with the GATK variant calling pipeline [79] and they report 4.85x speedup using WGS data. Houtgast et al. [80] propose a GPU accelerated BWA-MEM [37] based read aligner with promising results, and with GASAL2 [81] the same research group accelerates the BWA-MEM with an NVIDIA Tesla K40c GPU, showing 20x speedup in the local alignment phase and overall speedup of 1.3x compared to BWA-MEM running on 28 CPU cores (2 x Intel Xeon E5-2680 v4 CPUs). The authors of GASAL2 implemented a GPU-accelerated sequence alignment in GATK HaplotypeCaller pipeline to speedup variant calling phase showing 2.3x speedup compared to CPU implementation [82]. *De novo* assembly [83] is another routine method in bioinformatics (see Section 3.4) that has been GPU accelerated [84, 85], and also at larger scale by Goswami S. et al. [86] with encouraging results. Houtgast et al. [87] propose FPGA acceleration

[15] https://airflow.apache.org/

for Smith-Waterman sequence alignment algorithm and for BWA-MEM read aligner [88], with promising results. An FPGA accelerated Hidden Markov Model-based algorithm has been implemented by Ren S. et al. [89] for sequence alignment prediction, giving 67x speedup compared to a CPU counterpart. Alser et al. [90] propose FPGA accelerated pre-alignment, resulting in 11x overall speedup compared to read alignment purely on CPUs. GPU and FPGA acceleration results in the bioinformatics application field also encourage the enhancement of the distributed pipelines presented in this dissertation (see Chapter 5) with hardware acceleration, and this is a promising future research direction.

Chapter 3

Routine methods in high-throughput sequencing data analytics

This chapter introduces the basic concepts and routine methods used in high-throughput sequencing data analytics that are related to Publications I-IV. The methods presented here consider sequence alignment, search, and assembly techniques, as well as genotyping, genomic data compression, and indexing techniques.

3.1 Sequence matching

Sequence matching is a routinely used method in basic biological sequence searching. In a genome, the alphabet is restricted to letters A, C, T, G, and N (unknown), denoting the nucleotides of a DNA sequence [18]. The basic application is to find out if a given sequence of nucleotides is contained in a longer sequence. A naive algorithm for searching for an exact sequence within another sequence can be done with the brute force method by checking the matching letters in a sequence position by position. In the example (Figure 3.1), we use short artificial nucleotide sequences for illustration, consisting of bases A, C, T, and G.

Let the reference DNA sequence be S = CGGCGAGCGCAC and the query sequence Q = GCGCA. To find Q in S, we start from the first position S(0) and compare each character Q(j) to S(i) by increasing the index j by one until the mismatch occurs. In the case of mismatch, the algorithm advances to the next position S(i+1). The length of S is m = 12 and the length of Q is n = 5. If Q is present at the very beginning of the S, then the minimal number of iterations is 5, which is the length of the Q. In the

worst case, the algorithm has to execute m times n comparison operations, thus the time complexity of this brute force algorithm is $\mathcal{O}(mn)$. In the example (Figure 3.1), the first match is found on the 7th iteration after 16 comparison operations.

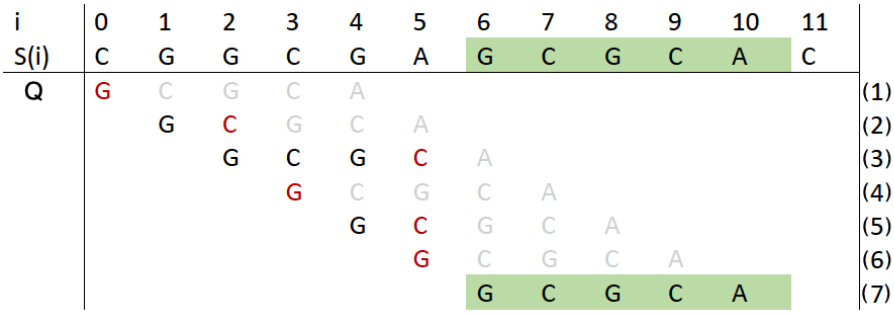

Figure 3.1: Brute-force algorithm for searching exact match. Here, black colour denotes matching character, red denotes a mismatch, and grey is a skipped character.

Knuth, Morris, and Pratt [91] proposed the first linear-time algorithm for exact string matching. They discovered that the naive brute-force algorithm includes unnecessary comparison operations that can be bypassed. Next, we go through the KMP string matching example in Figure 3.2 step by step. If the current character Q(j) does not equal S(i), the algorithm skips checking the following characters in Q, and it advances to S(i+P(j)+1), where P(j) is an integer defined in the prefix table (Figure 3.3). The prefix table is initialized using a prefix function as explained by Erciyes [51]. Basically, it specifies how many positions to skip when a

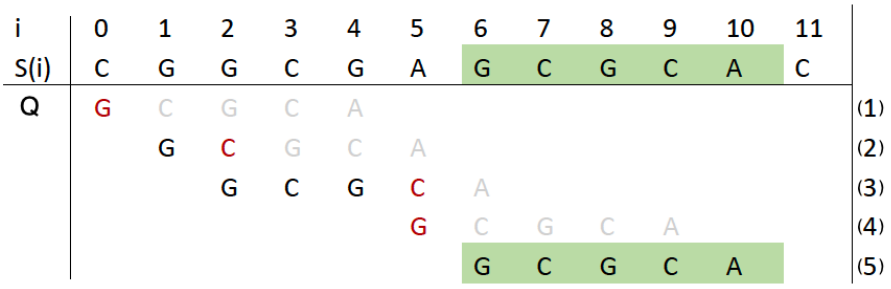

Figure 3.2: Knuth-Morris-Pratt algorithm step by step. Black colour denotes matching character, red denotes a mismatch, and grey is a skipped character.

j	0	1	2	3	4
Q(j)	G	C	G	C	A
P(j)	0	0	1	2	0

Figure 3.3: Pre-computed prefix table used with Knuth-Morris-Pratt algorithm.

mismatch occurs in Q(j). In Step 1, a mismatch occurs in the first position Q(0)≠S(0). Prefix table P(0)=0, thus we advance to S(0+0+1)=S(1). In Step 2, Q(0)=S(1), thus we increment i and j by 1. Then Q(1)≠S(2) and from the prefix table we get P(1)=0 and we advance to S(1+0+1)=S(2). In Step 3, the first three letters of Q match with corresponding positions in S, and a mismatch is found when j=3 and i=5. Then Q(3)≠S(5) and from the prefix table we get P(3)=2 and we advance to S(2+2+1)=S(5). In Step 4, the mismatch occurs in the first position Q(0)≠S(5). Thus we increment i by 1 (S(5+0+1)) and continue from S(6). In Step 5, all the characters in Q match with S, and the exact match has been found. The exact match was found using 13 comparison operations in total.

Boyer-Moore [92] is an exact string matching algorithm that also uses the properties of similar suffixes to skip unnecessary comparisons. It can search strings in $\mathcal{O}(n/m)$ time by using optimization rules and long strings but has the worst-case performance of $\mathcal{O}(nm)$. Another exact string matching algorithm is Shift-Or [93], which can perform string matching in $\mathcal{O}(n)$ time in the worst case enhanced by bit-wise operations but has its maximum string length limited by the memory-word size of the machine. Rabin-Karp [94] is an exact string matching algorithm that uses hashing of substrings to improve its search performance. It has an average performance of $\mathcal{O}(m+n)$ time and $\mathcal{O}(mn)$ in the worst case.

An alternative approach is to use approximate string matching, where a restricted amount of mismatches are allowed. For example, we can use the naive approach from Figure 3.1 and count the matches and mismatches over the alignment of Q. Assuming mismatches are restricted to c = 1, the alignment of Q is considered as a match if the count of matching nucleotides is greater than or equal to 2.

Assuming we have k reference sequences that Q should be aligned to in parallel. A straightforward coarse-grained decomposition is to partition the reference sequences and perform a search algorithm for each reference sequence in parallel. Ideally, this would result in k times speedup compared to the sequential approach where Q is aligned to each S^1,...,S^k reference sequence in parallel.

Further, each S^1,...,S^k can be decomposed into fine-grained partitions by splitting a sequence into subsequences. Similar ideas of distributed KMP algorithm have been discussed by Erciyes [51]. However, this would require storing the matches in memory and preserving the order of subsequences after the parallel alignment stage is finished, as Q may stretch over the partition boundaries. A possible solution for this could be to combine the results from each indexed partition by incrementing the positions of matches using the partition length m, index number, and the position in subsequence. The sequence that spans across the partition boundary, e.g., Q = CGT in Figure 3.4 cannot be found with this naive approach. With a more sophisticated parallel algorithm, the partition boundary matches can be handled, e.g., by storing the partial match if it starts from the first position of S^1,...,S^k or is located at the rightmost position in the S^1,...,S^k at range [m-n+1,m], and by sorting the results by the partition index and combining the partial matches. Another approach is to partition the sequence S into overlapping partitions, where the overlapping sequence length is at least the length of the Q.

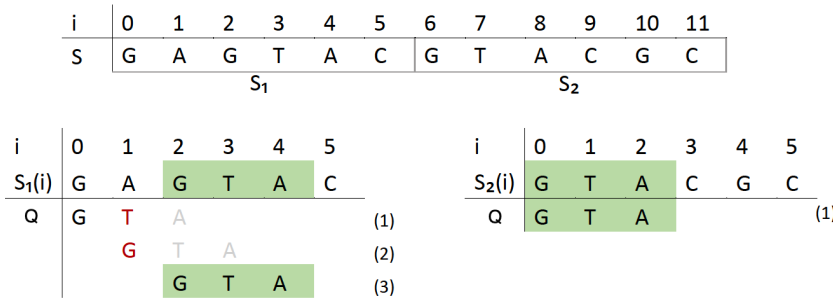

Figure 3.4: Fine-grained parallel brute-force method for searching subsequences. The first match is found from S^1 at position 3 on the third iteration and the second match from S^2 at position 1 on the first iteration.

3.2 Sequence alignment

Biological sequence alignment is a fundamental problem in comparative genomics and routinely used to search for any kind of sequence from viruses to human individuals [95]. The comparison of sequences is needed to analyze the biological functionalities of the species. This is based on the premise that similar sequences of nucleotides (homologous) provoke similar molecular functions. Basically, sequence alignment measures how similar

or distant two sequences P and Q are (pairwise alignment). The similarity of P and Q can be computed simply by measuring the difference between all elements in sequence P and sequence Q. Biological sequences experience evolution: they change through time. These changes, called mutations, happen in genetic recombination during cell division where some changes are inherited through reproduction such as germline mutations, while somatic mutations persist only through a species life cycle [18]. Generally, three types of edit operations mutate DNA: substitutions (one base changes to another), insertions (consecutive bases are added to the sequence) and deletions (consecutive bases are removed from the sequence) [18]. In biological sequences, some edit operations happen more frequently than others. By giving some penalty/score for different edit operations, we can measure the alignment relevancy by counting the total sum of operations in an alignment and minimizing or maximizing the sum over the alignment scores. Local alignment is the case when sequence P is longer than Q, and preceding and trailing insertions and deletions (indel) are allowed without penalty. An example of basic local alignment scoring with maximization is given in Figure 3.5.

P	A	A	C	C	T	T	G	C	C	T	
Q	A	T	C	C	T	T	T	C	T		
Σ	0	-1	0	0	0	0	-1	0	-1		-3

P	A	A	C	C	T	T	G	C	C	T	
Q	A	T	C	C	T	T	T	-	C	T	
Σ	0	-1	0	0	0	0	-1	-2	0	0	-4

Figure 3.5: Local alignment scoring. Here, the scoring scheme defines 0 for a match, -1 penalty for a mismatch and -2 for a deletion. The maximum alignment score is -3.

Global alignment is the case when sequence P is longer than Q and the alignment covers all positions in P. In that case, the preceding and trailing indels are also penalized. Semi-global alignment is the case when the alignment is restricted to the subsequence of P. In practice, semi-global alignment is used when short sequences are aligned with longer sequences such as genomes. Smith-Waterman is one optimal algorithm for searching for local alignments in $\mathcal{O}(mn)$ time [96]. Needleman-Wunch is an optimal $\mathcal{O}(mn)$ time algorithm for finding global alignment [97]. However, the sequential performance of these algorithms is not satisfactory for searching for a number of sequences from a large sequence database.

In practice, one or more input sequences are pairwise aligned to multiple reference sequences listed in the reference database. Regardless of the algorithm, similar coarse-grained and fine-grained data-parallel approaches discussed in Section 3.1 can be applied to the problem. The sequence database can be split into multiple partitions and multiple pairwise alignments can be run in parallel per partition and/or each input sequence Q can be pairwise aligned to each reference sequence P inside the partition in parallel (coarse-grained). Fine-grained approaches are then applicable to speed up the alignment algorithm itself by decomposing the aligned sequences into partitions as described in Figure 3.1. In any case, the alignment results need to be merged and recalculated in the end to find the best scoring alignment amongst all the alignments.

BLAST [39], Basic Local Alignment and Search Tool is perhaps the most popular tool for finding sequence matches. BLAST relies on a heuristic algorithm to restrict the search space and to speed up searching. Online BLAST searches the database maintained by the National Center for Biotechnology Information (NCBI), consisting of a huge number of sequences assembled from different species. The offline BLAST version is also available and it can be used with custom sequence databases, although scaling to a large number of simultaneous searches and long sequences requires specialization to the distributed and parallel approaches presented in Section 4.3.1.

Multiple sequence alignment (MSA) is used in comparative genomics to find an alignment between multiple sequences or genomes. In MSA, three or more sequences are aligned pair-wise and the best scoring alignment is computed over all the sequences. As solving MSA is computationally heavy in practice, MSA algorithms typically use some heuristics and approximations for finding the near-optimal solution in a feasible time. Clustal [98] is one popular MSA algorithm utilizing the Hidden Markov Model (HMM), MAFFT [99] relies on the Fast Fourier Transform, and MUSCLE [100] on log-expectation heuristics.

3.3 Read alignment

Read alignment is a routine step in sequencing data analysis [4]. It is a special case of a sequence alignment where often billions of short sequences of a donor genome produced by NGS are aligned with a known reference genome. These short reads are sequenced from random DNA fragments without foreknowledge about where each chromosome, haplotype, or DNA strand read has come from. Thus, read alignment is performed to find

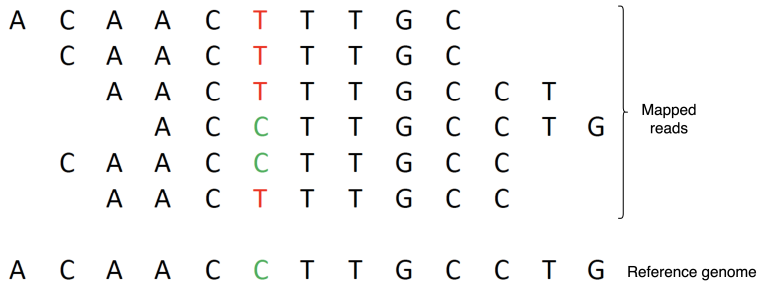

Figure 3.6: A simplified scenario of read alignment with a reference genome. Red base marks a difference from the reference base in aligned position.

the locations of donor genome reads in a reference genome and to identify differences between the donor and the reference genome [101]. The reads are said to be mapped when they are similar enough to a reference genome and unmapped otherwise. Figure 3.6 illustrates a simple read alignment scenario. The coverage of sequencing defines how many times each DNA fragment is sequenced per base position on average: the more coverage, the better the accuracy. To find the best aligning reads covering a substring of a reference in some genomic location, every read should be aligned to each base position in a reference genome. That is, multiple reads can align to the same region, resulting in different alignment scores. Due to sequencing errors and duplicate reads, raw read data is typically cleaned before the alignment.

Efficient read alignment methods utilize advanced compressed indexing techniques to reduce alignment computation time and improve space efficiency. The Burrows-Wheeler transformation (BWT) is a powerful method for constructing an index of the reference sequence to speed up the read alignment [102]. In practice, candidate sequences are searched from the Burrows-Wheeler index and the final alignment is done to candidate sequences. Variants of the Burrows-Wheeler index are used in popular read aligners such as BWA [37], Bowtie [38], and SOAP [103].

3.4 Sequence and genome assembly

De novo assembly [83] is an appropriate method when novel genomes or sequences are studied and there is no reference sequence available. *de novo* sequence assembly basically joins short sequences (e.g., Next-generation sequencing reads) to longer ones – contigs – and joins contigs to ordered scaffolds (Figure 3.7) to construct whole-genome assemblies [83]. Longer

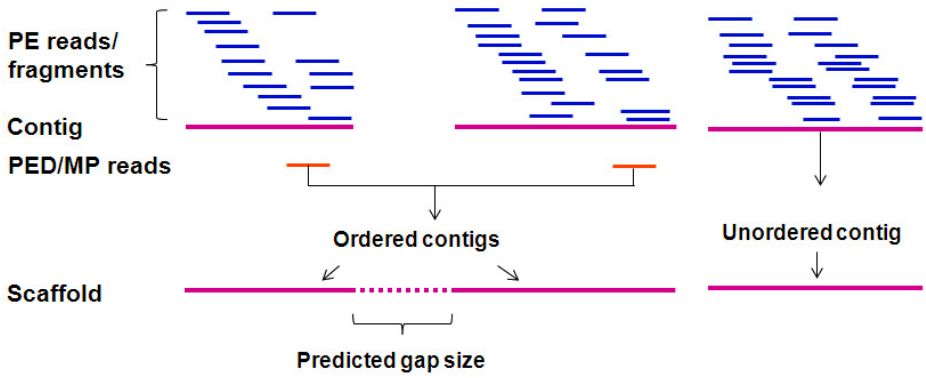

Figure 3.7: *De novo* assembly from paired-end reads by Kuo-Ping [25].

sequences can be then identified and classified more precisely into species. *De novo* assembly is also used to improve the existing assemblies and has revealed many non-reference unique insertions (NUI) in human genomes, i.e, sequences present in individuals but not found in the standard human reference genome [104]. *De novo* methods typically exploit graphs such as de-Bruijn and overlap graphs to find and join overlapping sequences using sequence prefixes and suffixes. Constructing sequence graphs and assembling contigs are computationally expensive and memory-intensive operations, making them problematic with WGS data and especially with large genomes. Moreover, assembling short NGS reads accurately is challenging, as the reads are only a few hundred bases long and genomes often include long repeat regions [105]. Paired-end (PE) sequencing can foster the detection of sequence repeats and improve contig concatenation as read pairs are sequenced from both ends of a longer DNA fragment [83]. Modern *De novo* assembly tools such as Abyss [106], SOAPdenovo[1], Trinity[2], and MegaHit[3] have improved assembly performance and decreased assembly time through shared-memory parallel implementations. The common data-parallel assembly strategy is to split sequences into k-mer partitions, construct subgraphs from k-mers, and assemble partitions in parallel. Typically, k-mer sorting and indexing with hash-tables are exploited in similar implementations [107]. The use of several assembly algorithms can improve the final assembly result. Reference-guided *De novo* assembly methods are applicable when the species under investigation is known beforehand and

[1] http://soap.genomics.org.cn/
[2] http://trinityrnaseq.github.io
[3] https://github.com/voutcn/megahit

an assembled genome of the representative species is available [108, 109]. In that case, sequenced reads are aligned to the reference genome and the mapped reads are *De novo* assembled, and unassembled reads are used to detect NUIs [108].

3.5 Genotyping

One of the main sources of variation in biological sequences arises from genetic recombination occurring during cell division [18]. Genotype represents a collection of all gene variants, alleles, in a genome. Genetic variation is traditionally detected by DNA microarrays [110, 111], requiring plenty of laboratory work. Next-generation sequencing has enabled rapid genotyping by sequencing (GBS) [112] utilizing computational methods such as variant calling (see Section 3.6.2). In spite of the high precision of microarray genotyping, the variants obtained do not often provide sufficient coverage for comparative genetics studies as it is able to identify only the variants the microarray is prepared for. In contrast, GBS offers a hypothesis-free method for genotyping that does not require any prior information about the gene variant itself and provides a way to discover all the variants, including novel variants.

3.5.1 Genotype imputation

Genotype imputation from a subset of genotyped variants provides a way to infer the rest of the known variants without using laboratory methods [113]. Genotype imputation utilizes large collections of genotype data sets, reference panels, where imputation is performed with computational methods by estimating the missing alleles of the target panel using already genotyped alleles in a reference panel [113]. The accumulating reference panel sizes offer increasing imputation accuracy at the cost of computation time. The current imputation methods are based on machine learning and related statistical methods. Hidden Markov Models (HMM) perform well on genotype imputation where relationships between alleles are taken into account, that is, nearby alleles are related through linkage disequilibrium [114]. HMM is used in popular tools such as Impute, Mach, and Beagle [115]. The current imputation algorithms are mainly based on the Hidden Markov Model (HMM), which enables accurate parallel imputation in adjacent sets of alleles inside Linkage disequilibrium [114].

3.5.2 Variant calling

Variant calling is a computational process for detecting variants used with genotyping by sequencing [116]. Variant calling begins by aligning sequencing reads to a reference genome and filtering out the unaligned reads. Next, mapped alignments are typically filtered by examining the quality of the alignments and filtering out the worst scoring alignments using some threshold for the alignment score.

Read pileup [117] based variant calling can detect single-nucleotide polymorphism (SNP) and short indels (insertions and deletions of a few base-pairs, typically shorter than 10 bases). Sequencing reads are piled up over mapped reference positions and reference mapped alleles are counted by using the information of the read alignment. Thus the most probable alleles covering a genomic position can be detected. SNPs can easily be detected with reasonable accuracy by counting the most presented nucleotide at a reference genome position. For example, if position i in genome Q is covered by n reads of which more than 90 percent contain nucleotide C, it is reasonable to infer that C is the nucleotide in position i. If inferred nucleotide differs from the reference nucleotide of position i in Q, it is inferred as an SNP. If genome Q is from a diploid organism such as a human, whether it is heterozygous or homozygous SNP can not be inferred as we do not yet know which haplotype it belongs to. Indel detection requires a more comprehensive analysis of a read pileup. Read pileup can be done with Samtools[4] and within the GATK[5] best practices variant discovery pipeline. The GATK pipeline also provides a Hidden Markov Model-based HaploTypecaller variant calling algorithm.

However, variant calling is always an error-prone process, especially in genomic regions including large structural variations, e.g., copy number variation, segmental duplications, or pseudogenes [118, 119]. Falsely called variants are often the consequence of NGS read misalignments typically occurring in these highly homologous regions, thus the variants observed should be comprehensively assessed and validated [120, 121]. Targeted methods for improving variation analysis in these regions have been developed, such as read-pairing-, split-read-, read-depth-, and *de novo* assembly-based methods, and hybrids of these [122, 123].

[4] http://www.htslib.org/
[5] https://github.com/broadinstitute/gatk

3.6 Compressing and indexing genomic sequences

This section presents genomic data compression and indexing methods that are applicable to parallel and distributed computing enabling fast data access with sequence aligners.

3.6.1 Sequence compression

The rapidly accumulating genomic datasets make efficient compression methods more significant. Moreover, sequences need to be retrieved quickly to enable fast sequence alignment and search capability. Virtually, human genetic variation is relatively low: genomes differ only by 1 base pair per 1000 bases on average, and most of the genome consists of repetitive sequences [18]. This property offers an excellent basis for compressing and representing large collections of genomes. The amount of storage needed for a collection of genomes can be reduced hugely with data compression methods utilizing the characteristics of repetitive sequences between the individual genomes [17], that is, the human genome includes a large proportion of repetitive sequences that can be found in every individual.

General-purpose lossless data compression tools such as gzip[6], 7-zip[7], and Snappy[8] are based on Lempel-Ziv (LZ) [124] compression method variants using dictionary coding. Typically, a compression dictionary is calculated on-the-fly and in gzip, using the DEFLATE[9] algorithm: the LZ77 dictionary size is limited to 32 Kb by its maximum sliding window size [125]. 7-zip uses the LZMA[10] compression algorithm, which is based on a variable dictionary size of up to 4 GB[11]. However, they were not developed for genomic data compression where long sequence repetitions are present. The performance of several methods on compressing sequence data has been compared by Kryukov K. et al. [126]. LZ variants have been used in many compression approaches to decrease database index size and retrieval time [127, 128, 129]. However, the original LZ does not scale to compress a large number of genomes as the uncompressed input data is read into memory as a whole for dictionary parsing. Thus, novel methods for LZ-based compression have been proposed for genomic data indexing and retrieval using the Relative Lempel-Ziv (RLZ) [17, 130, 131, 132] method. RLZ uses

[6] https://www.gzip.org
[7] https://www.7-zip.org
[8] http://google.github.io/snappy
[9] https://www.w3.org/Graphics/PNG/RFC-1951
[10] https://en.wikipedia.org/wiki/Lempel-Ziv-Markov_chain_algorithm
[11] https://www.7-zip.org/7z.html

a sample of the data as a dictionary to compress the rest of the data relative to the dictionary that makes it more memory efficient.

RLZ [40, 41] can be enhanced with suffix-array [133] based dictionaries to decrease encoding time and speed up the searching. Suffix-array is a data structure that represents a string with suffixes of the original string in a sorted array that can be constructed in linear time [134, 135]. Suffix-array data structure enables fast dictionary searching of the successors of the query substring. The suffix-array is calculated from the dictionary and the rest of the sequences are LZ factorized using an LZ77-parse relative to the dictionary utilizing the suffix-array [41]. However, efficient compression requires that the dictionary represents the repetition in the input sequences comprehensively. In decompression, the successors of dictionary-matched substrings are backtracked using an LZ77-parse. That is, all the original input sequences can be decompressed from the dictionary sequence.

Wandelt and Leser [136, 137] propose a referential compression method that has been used to construct a Referentially Compressed Search Index (RCSI) for a large collection of genomes [138]. The compression method in RCSI is very similar to RLZ except that RCSI compresses only differences in genomes relative to their common reference genome instead of a dictionary constructed from the sequences. RCSI utilize a compressed suffix tree [139] of the reference sequence to decrease the search and compression time. With the RCSI they achieved an index compression ratio of 26:1 with 1092 whole human genomes in 54 hours on a single laptop [138].

Kärkkäinen et al. [140] present three external memory algorithms for LZ factorization that are proposed as suitable for distributed and parallel implementation. Hoobin et al. [40] propose RLZ-like dictionary coding where a representative dictionary is sampled from the entire input data and used to encode the data in fixed-size blocks. This approach provides faster retrieval times and a feasible compression ratio when the block size is large enough. Moreover, it is potentially suitable for data-parallel compression as separate blocks can be encoded in parallel. In Publication III, we implement a distributed RLZ compression method that is a hybrid of these methods using KKP3 [140, 141] algorithm for data-parallel dictionary coding (see Section 5.3).

3.6.2 Indexing

For a human, the reference genome is over three billion base pairs long (approximately 3 Gigabytes)[12]. NGS read data to be aligned can exceed

[12] https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.39

hundreds of Gigabytes, depending mainly on the sequencing technology, sequencing coverage, and the size of the donor genome [4]. Typical NGS analysis workflow is preceded by a read alignment process (see Section 3.3) where billions of short-read sequences from a donor sample are aligned with a reference genome.

Basically, sequence alignment compares short sequences to subsequences of a genome (see Section 3.2). Assembled reference sequences are typically indexed to speed up the sequence alignment and searching, which can be computationally heavy with massive sequence databases. For example, the NCBI GenBank database included in April 2021 over 227.12 million sequences comprising over 832.40 billion bases and the Whole-genome sequencing (WGS) database over 1590.67 million reads comprising over 12.73 trillion bases[13]. Moreover, aligning sequences with such a massive database requires a vast amount of computing time and working memory.

Compressed full-text indexes, also called *self-indexes*, are compressed and indexed data structures that do not need the original data to be accessed after the index has been constructed [142]. Self-index has interesting advantages as it can provide random access to compressed sequences with efficient pattern matching properties without the need for data decompression [42]. FM-index [143] is a self-index based on the Burrows-Wheeler transformation (BWT) [102] which can be, e.g., implemented based on the suffix-array (SA) [133]. FM-index is used in popular read aligners such as BWA [37] and Bowtie [38]. However, these tools do not exploit highly repetitive sequences in compression, unlike hybrid-indexing, which we will cover next.

Hybrid-indexing

Hybrid-indexing was first presented by Ferrada et al. [42] based on the concept of compressed text indexes by Kärkkäinen and Ukkonen [144]. Ferrada et al. [143] show that the LZ77 compressed repetitive sequences can be indexed and searched with conventional indexes such as the FM-index. In genomics, hybrid-index is an efficient data structure that allows fast alignment of sequences to compressed sequences using traditional indexing methods such as Burrows Wheeler and Bowtie2.

Valenzuela et al. [48] propose a CHICO indexer based on hybrid-index implementation using RLZ compression with kernelization to compress and index collections of genomes, i.e., pan-genomes. CHICO uses kernelization to compress pan-genomes to a single kernel sequence that can be indexed

[13] https://www.ncbi.nlm.nih.gov/genbank/statistics/

using conventional tools, such as BWA and Bowtie. Gagie and Puglisi [145] describe kernelization for representing a collection of genomes in a non-repetitive string that is a concatenation of LZ77 parsed phrases separated by some boundary symbol, e.g., \$. The genomes are first compressed with RLZ relative to the sample dictionary parsed from a partial pan-genome. RLZ parsed suffix phrases are used to represent the original genome in a non-repetitive kernel sequence. The positions of LZ suffix phrase boundaries in a genome are stored into auxiliary data structures using a sorted list of phrase boundary positions in kernel sequence and with a list of corresponding phrase boundary positions in a genome. The kernel sequence is then indexed for conventional read aligners, e.g., BWA [37] or Bowtie [38].

On the CHICO index, the read alignment is performed with the CHIC aligner [146], which aligns reads to an indexed kernel sequence instead of aligning to each individual genome separately using BWA or Bowtie2. The CHIC aligner maps the resulting kernel mapped reads to the original genomic positions using the LZ parse and phrase position lists of the CHICO index [48]. The alignment of a sequence that crosses a phrase boundary in the kernel sequence is said to be a primary occurrence and secondary otherwise (contained in the phrase) [144]. The kernel mapped reads are initially potential primary occurrences [48]. Thus, the parameters used in the conventional read aligner (BWA or Bowtie2) affect the number of kernel mapped reads, and therefore, potentially the number of reads mapped to the pan-genome. The primary occurrences are searched in the first phase and the secondary occurrences are searched afterward using LZ parse and primary occurrences as an input [144]. In many real use cases, it is enough to find primary occurrences only. The distributed RLZ method with hybrid-indexing presented in Publication III extends the CHICO indexing (see Section 5.3).

Chapter 4

Scalable high-throughput sequencing analysis pipelines

A complete bioinformatics workflow typically consists of various analysis components that are combined into complex analysis pipelines. High-throughput sequencing (HTS) analysis pipelines are traditionally broken down into three parts: primary, secondary, and tertiary analysis [147]. The primary analysis involves sequencing of biological samples with a sequencing instrument that generates sequencing read data. Sequencing itself follows laboratory instrument- and analysis-specific protocols that do not fall within the scope of this dissertation. We focus on the secondary and tertiary analysis steps that are more computationally versatile and can be programmatically parallelized. However, the analysis steps here are presented in chronological order and the primary analysis step is introduced briefly first as it is the fundamental premise for conducting HTS analysis itself. Secondary analysis typically comprises quality analysis, filtering, and preprocessing of raw sequencing data as well as read alignment, sequence assembly, and variant calling steps. Tertiary analysis can comprise, e.g., genomic data aggregation, functional and taxonomical classification and annotation, genome-wide association studies, and exploratory analysis [147].

4.1 Primary analysis

In primary analysis (Figure 4.1), the sequencing instrument detects base pairs from the DNA sample fragments in the base calling step. Basically, the sequencer determines bases by measuring signal intensity from the signal emitted while a DNA fragment is synthesized or ligated [148]. Each

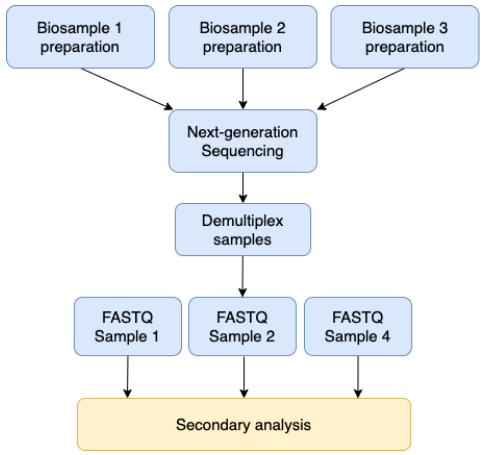

Figure 4.1: Primary analysis steps.

```
==> reads_1/ERR251691_1.filt.fastq <==
@ERR251691.1 FCD1LUTACXX:4:1101:1241:2128/1
TGTGCTTTGTGTTCACCGAAGAAGCAAGCAGACTTTGCATGTTCCATTATGTATTTATTCCTCTGTCCTGCAGTGAGNTGGCGAAAAAGCAGAGCAGTGT
+
@BBFFFFFFHHHHFIIIJIIIJIIIIJIIIGGHIGJIJIGEHIJII>GIHJHDHGICIGIIJJGGIGIJJJJJI@DHE#-;BDFCD?@DBACDC@DAC>A@

==> reads_2/ERR251691_2.filt.fastq <==
@ERR251691.1 FCD1LUTACXX:4:1101:1241:2128/2
CCTTCATCTACCAGGGAGTTGGTTATAGTACATACTATTTAAAACATGATTTCTCATTAAAATAACCCTGAAAAGGAAACATATTAGGTTTNANGAAAGC
+
@??DDDDFHFHHHIIE+CF;E3CC4CAFB94??C?CCFGGDGIG@FB<<DFF0B?FAHGIJGDHDGIHG>EC;=EECCECD;@C################
```

Figure 4.2: FASTQ paired-end file format (NCBI SRA accession number ERX226225).

detected base is validated by measuring the precision of the base call and the quality score is calculated for each base as a result. Multiple samples are multiplexed for simultaneous sequencing and the output is de-multiplexed to separate the sample reads by identifiers [4].

The next-generation sequencing (NGS) base calling step generates single or paired-end short sequencing reads with the typical length of 50 to 300 bp with quality scores in an instrument-specific data format. The read data is typically converted to the FASTQ sequencing file format [149] for the read alignment and sequence assembly tools. Paired-end (PE) sequencing generates read pairs which are sequenced from both ends of the DNA fragment. PE has advantages compared to single-end as the relative positions of the pairs are known. PE enables detection of sequence rearrangements and it fosters *de novo* assembling by easing the contig concatenation [83]. Figure 4.2 shows paired-end read pair extracted from two FASTQ files. One

```
>NZ_CP014997.1 Salmonella enterica subsp. enterica serovar Weltevreden str. 1655 plasmid, complete sequence
AGCGATACCCACGGTACGGTTTTCCAAAACTTTTCCAGGTTCTGCGGCGGCAGGGATACCCGTGGAATCACAAAAGGATC
CATCGTATTTATTGTCTGCTGAAGCTGAATTTTCGCCGTAAAGGCAAACAACGGTTGCCGGTGCGTAATCCCTCGCCACT
GGCCACACCGGAAGCGCTGAACCAGAGCTGGTCTGTCGATTTTATGCATGATGCCCTGGTCTGTGGCCGTCGTTTTCGCA
CGTTCAATGTTGTTGATGACTTTAACCGTGAGGCGTTGTCGATTGAAATCGATCTGAATCTGCCAGCTCTGCGCGTGGTC
CGTGTACTCGACAGGATCGCGGTAAGTCGCGGCTATCCGGCCATGCTACGCATGGATAATGGTCCGGAATTTATCTCACT
--
>NZ_CP012985.1 Salmonella enterica subsp. enterica serovar Typhimurium strain RM9437, complete genome
AGAGATTACGTCTGGTTGCAAGAGATCATGACAGGGGGAATTGGTTGAAAATAAATATATCGCCAGCAGCACATGAACAA
GTTTCGGAATGTGATCAATTTAAAAATTTATTGACTTAGGCGGGCAGATACTTTAACCAATATAGGAATACAAGACAGAC
AAATAAAAATGACAGAGTACACAACATCCATGAACCGCATCAGCACCACCACCATTACCACCATCACCATTACCACAGGT
AACGGTGCGGGCTGACGCGTACAGGAAACACAGAAAAAAGCCCGCACCTGAACAGTGCGGGCTTTTTTTTTCGACCAGAG
ATCACGAGGTAACAACCATGCGAGTGTTGAAGTTCGGCGGTACATCAGTGGCAAATGCAGAACGTTTTCTGCGTGTTGCC
--
>NZ_CP014658.1 Salmonella enterica subsp. enterica serovar Anatum str. USDA-ARS-USMARC-1736 plasmid pSAN1-1736, complete sequence
GATAGGCTCAGATAAACAGACCTTACCCTCGCATCGAGAACCGCTTGCCCTCCAGCATCGAGAGACGGTGGTAAAGAGGC
ATTTGGAATCTTTGATGCCATATCCAATATATCTGGAATCTTTAAATATAGATTCATATGTAAAGAGGCTGTGAAAGAAT
AAGAGCATCAAGATTCCAGATAGATAGAGGGAAATTTGACAAATTCCAAAGATGGGTTAGCCTAGTGACAGAACTAGATT
CCAGTATTGGAATAATCAGCTTTAAATTCCAGATAGATAGTTATGTGGATAGGAATTGGATAGGAATTGGGAGGGTATTG
AGGTGAGTCTACCAACAGAGCGATGGCTAGATCTGCTAGATCCGAAGCTCGAAGAAGAACGATCCGAGATAACAGAGGAT
```

Figure 4.3: Salmonella sequences in FASTA file format (data available in NCBI Assembly database).

FASTQ record[1] contains four lines where the first line describes the read id and mate-pair number as well as instrument-specific information. The second line is the actual sequence of detected nucleotides. Line three is for optional descriptions. The fourth line contains the Phred[2] quality score, which measures the quality of the identification of each base.

4.2 Secondary analysis

The secondary analysis utilizes various methods and tools using different data formats [147]. Typically, assembled sequences such as contigs, chromosomes, and genomes are represented in FASTA format (Figure 4.3) [150], where each sequence record starts with a header line defined by the > symbol and followed by a unique sequence identifier. The actual sequence is stored as a character string in the standard IUB/IUPAC nucleic or amino acid codes [151] after the header line in a single line or multiple lines of the same length. Generally, this step can be divided into two methodological categories: re-sequencing and *de novo* assembly. Re-sequencing (Figure 4.4) implies read alignment (see Section 3.3) with some reference sequence assembled from the same species and is often used with longer genomes such as humans and other mammals [152]. *De novo* assembly (see Section 3.4) from NGS data basically joins short sequences from FASTQ reads to longer FASTA sequences and is typically used with shorter genomes such as microbes and species that do not yet have a reference genome assembled [83], but is also used with human genomes, especially for complementing the re-sequencing method [107]. In both approaches, the reads are preprocessed

[1] https://www.ncbi.nlm.nih.gov/sra/docs/submitformats/#fastq-files
[2] https://en.wikipedia.org/wiki/Phred_quality_score

as the read data may contain sequencing artefacts such as low quality and duplicate reads. Various methods exist for preprocessing FASTQ data such as removing adapter bias, trimming read length, trimming Poly-A/T tails, read quality-based filtering, filtering duplicates, read normalization, or fixing the formats for different tools [153]. Read data can be also used for, e.g., calculating and estimating genome sizes.

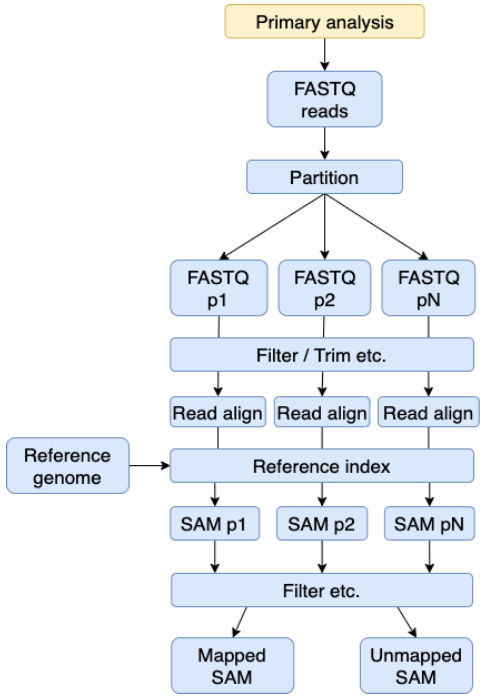

Figure 4.4: Whole genome re-sequencing secondary analysis step.

4.2.1 Whole-genome re-sequencing

After the preprocessing steps, the read aligner is used to align reads to a reference genome [154]. To specify the concept of an alignment (see Section 3.2), an alignment is said to be mapped when it is determined to be a subsequence of a reference sequence and unmapped otherwise. As whole-genome sequencing detects every organic material including microbes from a species, reference mapped reads are assumed to belong to the same species as the reference genome represents. In turn, unmapped reads contain all organisms divergent from the sample species. At this stage, unmapped reads may be invaluable, e.g., for metagenomics studies where unmapped

reads are *de novo* assembled or re-aligned to a reference genome of a specific microbe under study (see Publication I). Various methods are used to modify, analyze and filter the alignment data before the tertiary analysis such as quality filtering, filtering mapped and unmapped reads, calculating mapping depth, marking duplicates, etc [153].

```
SRR233147.1191  419     HG00151-2.01    103375687       1       76M     =       6487154117      178     AAGAGTTGTTACATGTTTGTAATGCTGTAAA
AGAAATTCCCAAACCAATAATACACTATGTTAACAATAACACAGT   HHHHHHHHHHHHHHBHFGHGHHHHHHHHHHHHHHHGGGHHHHHGHHHHHHHHHHHHHHHHHHGIGGG1?BBAGEGEGF    AS:i:0X
S:i:0   XN:i:0  XM:i:0  XO:i:0  XG:i:0  NM:i:0  MD:Z:76 YS:i:0  YT:Z:CP
SRR233147.1401  323     HG00151-2.01    146389095       6       76M     =       6488381332      175     AGAAATCAACGAGAAATTGCTCGCCCAGCTG
GCAGAGAACAAACAGCAGTTCAGAAACCTCAAAGAGAAATGTTTT   =9>+@##########################################################################    AS:i:-X
S:i:-8  XN:i:0  XM:i:4  XO:i:0  XG:i:0  NM:i:4  MD:Z:20G1C5T39G7        YS:i:0  YT:Z:CP
SRR233147.1401  435     HG00151-2.01    146389194       6       76M     =       6488381233      -175    GGCCAACCGACAGAAGAAATACAGTAAGATC
TATAGGCTCACCATCATGAAAGTGATGAATGATGTCCTGTCTTCT   #######?EEEDFEBFD=8DC>9><=4.-2,:7:5:,=<;=CDCBDD8;EEC5DCC>:8>==?@@..94:;->45>    AS:i:0X
S:i:0   XN:i:0  XM:i:0  XO:i:0  XG:i:0  NM:i:0  MD:Z:76 YS:i:-8 YT:Z:CP
---
SRR233147.4714268       77      HG00151-2.12    7470314 0       *       *       0       0       TTGAGTGTATGGTCTTTTTTTGGGGGTTTTTTGGCATTT
TTTGTTTGGTGTGGGTGGGAGTGTTTGTGTGGGTTTG   ##########################################################################    YT:Z:UP
SRR233147.4714268       141     HG00151-2.12    7470314 0       *       *       0       0       TGTGTTGTTGTGCGTATTATACGGGGACTTCTTGGTATT
TGTGGTTTCTGGATTCTGGGGGTGTTATCTTGGGTTT   ##########################################################################    YT:Z:UP
SRR233147.4714269       77      HG00151-2.12    7470314 0       *       *       0       0       CCACTATAGTTTTGGATGTTTTGCTTTGGTGGGTTGGGT
GGGGGGTTTCGTGGTGTTGGTTGTTTTTGGTGTTATT   ##########################################################################    YT:Z:UP
```

Figure 4.5: SAM alignment file format.

To continue with the re-sequencing (Figure 4.4) approach, the mapped reads are persisted. The majority of read aligners produce Sequence Alignment Map (SAM) formatted files (Figure 4.5) or binary SAM (BAM) formatted files [34]. SAM format represents each alignment record in one line, where each field is delimited by a tabular symbol[3].

Read alignment is computationally expensive, especially with multiple high-coverage sequencing samples. A large majority of read aligners use the Burrows-Wheeler-based index (see Section 3.6.2) which is constructed first and reused when multiple samples are aligned in parallel. Aligning RNA sequencing reads to a reference genome requires considering transcript splice junctions as RNA is spliced from the DNA during the transcription [18]. Tools such as TopHat are used to compose RNA spliced alignments before the read alignment [155]. The straightforward data-parallel approach for distributed read alignment is to decompose reads into partitions and align each partition to the reference genome in parallel on multiple computing nodes. This method has been implemented in a few distributed read alignment tools such as CloudBurst [156], SparkBWA [157], BigBWA [158], Seal [69], and Halvade [67].

Another use case is to align reads sequenced from multiple samples to the reference genome in parallel. Data-parallel decomposition by the sample can provide sufficient coarse-grained parallelization if the number of samples is relatively large in proportion to the computing nodes available in a cluster. Partitioning a reference genome index is not as practical, as the optimal alignment has to be found and the reference index data

[3]https://www.ncbi.nlm.nih.gov/sra/docs/submitformats/#bam-files

typically remains several times smaller than NGS read data. The more fine-grained approach would be to decompose the read data into partitions. Thus, it is feasible to provide a reference index as a whole for each parallel alignment execution cycle by replicating and caching the index to all nodes in a computing cluster. Read data partitions are then aligned with the whole reference index in parallel on all distributed computing nodes. Both sample-level and read data partitioning approaches have been applied in Publication I to scale the read alignment step with multiple human samples in a distributed Spark computing cluster.

Distributed computing methods and frameworks such as Apache Spark [58] and Hadoop Distributed Filesystem (HDFS) [36] are well applicable here. By distributing reads into HDFS, the partitioning is done automatically when reads are loaded into Spark. As the FASTQ record is not line-based, the custom I/O interface is needed to split the read records correctly. This is because Spark splits data by lines into RDD by default. Hadoop-BAM [33] provides a custom record reader and writer for the FASTQ format, enabling robust distributed FASTQ data manipulation with Spark. However, the reference index must be available in memory in each parallel read partition alignment stage. Spark provides a broadcast mechanism for distributing shared variables with immutable data types. Distributing the index files in such a format that the native read aligner code can be executed inside the Spark transformation function requires custom methods. The straightforward method would be to use Spark's RDD *pipe* operator to execute a native process. However, the native process input interface may restrict the use of the RDD *pipe* as it writes an RDD record directly to *stdin* of the given process. Java Native Interface provides a more robust way to run native code in Spark, giving the programmer more flexibility at the expense of simplicity.

4.2.2 *De novo* assembly

De novo assembly (see Section 3.4) is widely used in metagenomics studies, where the most novel sequences are found and the reference sequence may be absent [12, 159]. However, recent findings of novel sequences amongst human genomes have raised the need for more scalable and efficient *De novo* assembly methods [160, 161]. Assembling sequences from WGS data is challenging as reads are much shorter (even with long reads using Third-generation sequencing, which is not always an option) than the molecules they originate from and the molecules can consist of long regions of repeating sequences and large structural variations [123]. As *de novo* assembler algorithms mainly use graph-based data structures (see Section 3.4), fine-

grained decomposition strategies are challenging to apply directly without parallelizing assembly routines in existing algorithms. Moreover, processing a large sequence graph is computationally demanding and requires a lot of memory. Chromosomal and sample-level partitioning provides a basis for the data-parallel approach. However, chromosomal decomposition requires reference-guided read alignment to be performed beforehand as we need to determine which chromosome a read comes from. This approach also enables more fine-grained partitioning by chromosomal regions as alignments store the positional information. Without the reference genome, read clustering techniques can enable a data-parallel assembly approach. Basically, reads can be clustered by overlapping k-mers and the data-parallel assembly can be then performed for each cluster [167]. However, overlap read clustering is also a compute-intensive process and needs further investigation and experiments on massive sequencing data. Sample-level decomposition is a straightforward method when multiple NGS samples are to be assembled but its efficiency depends on the number of samples relative to the number of available computing nodes. Note that sample size distribution can greatly affect the load balance and the overall runtime would be roughly the assembly time of the largest sample.

4.2.3 Metagenomics

In metagenomics, microbes under investigation, e.g., viral or bacterial pathogens, can be identified from sequencing data using computational methods [168]. Analysis often starts with separating the unrelated genetic material such as host DNA, e.g., human DNA in human microbiome samples, from the related material such as viruses [169]. If the reference genome of a pathogen is available, a re-sequencing approach (see Section 4.2.1) can be conducted. If the reference genome is not available, as is the case with many microbes, *de novo* assembling can be used [170]. In that case, the reads belonging to unrelated species can be filtered out by aligning the reads to the host reference genome, e.g., to the human reference genome if the metagenomic sequencing is based on human samples [46, 47, 171]. A simplified parallel pipeline to assemble and identify genomes from metagenomic next-generation sequencing (NGS) samples is shown in Figure 4.6.

A read normalization method is often used in a pre-processing step to reduce sequencing bias, which improves the estimation of pathogen abundance in the sample [172]. As normalization reduces the number of reads, it also reduces the analysis data volume and computational burden. Species-related reads are normalized by k-mer (similar sequences of length k) abundances and *de novo* assembled to contigs. Basically, read normalization

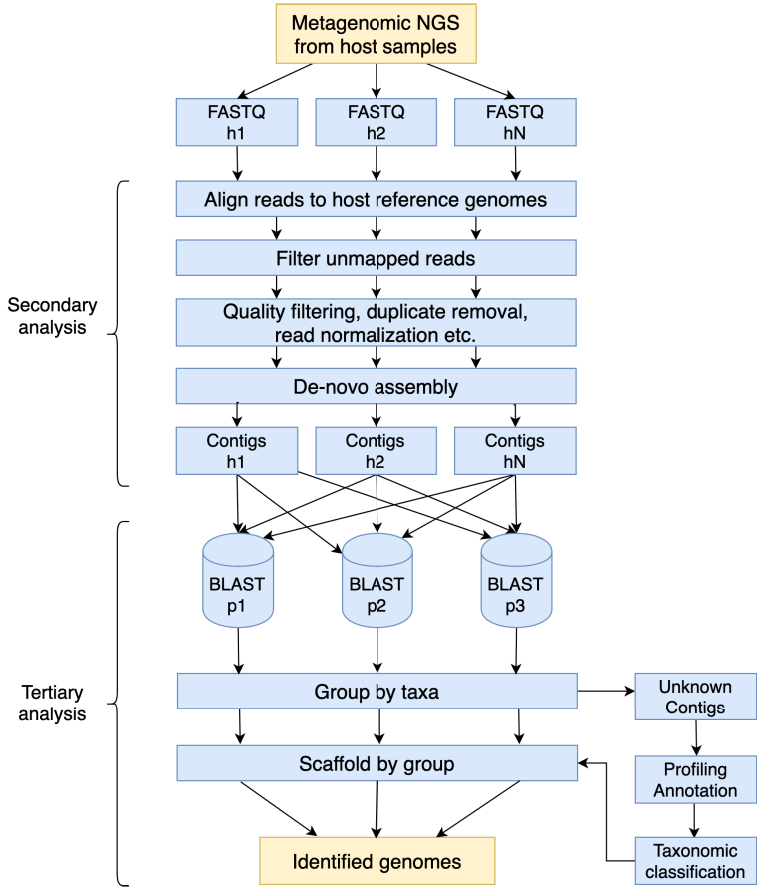

Figure 4.6: Identifying microbial genomes from metagenomic NGS samples in parallel.

flattens the distribution of read coverage by down-sampling the reads in high-depth areas considering the abundances of k-mers [173]. This technique is used in Publication I to scale the *de novo* assembly of multiple metagenomic samples in parallel.

4.3 Tertiary analysis

The tertiary analysis explores and discovers new biological insights through more precise analytics using the analysis-ready data processed in the previous steps combined with external data sources. This step comprises comparative, statistical, and taxonomical genetic analyses and classifica-

tion [147]. The tertiary analysis covers a wide range of various analysis methods and tools for different disciplines of computational genomics. Sequences of interest are annotated by discovering known genomic elements such as promoters, enhancers, motifs, CpG islands, open reading frames, introns, or exons [174]. Here, we address the most important routines with regard to big data analytics in genomics.

4.3.1 Sequence alignment and search

Sequence alignment basics have been reviewed in Chapter 3. Sequence alignment is one of the routine methods when novel assembled sequences must be searched, compared, classified, and annotated. Various sequence databases for different organisms and species are available online. However, online alignment search tools are typically restricted in terms of the number of queries and sequence length. Thus, custom methods and databases for performing study-specific comparisons on a large scale are often implemented. Strategies to scale the sequence alignment in a distributed environment depend on the database size, the number of queries, and the algorithms used.

The popular BLAST sequence alignment tool is routinely used by bioinformaticians, and thus researchers have exploited parallel computing methods to improve its performance, such as multithreading, Message Passing Interface, GPUs, database replication, and/or distributed databases [77, 175, 176, 177]. The main factors affecting BLAST search performance can be identified as database size, query batch size, and search sequence length. Database replication enables a data-parallel approach to be applied to query batches by chunking query batches into partitions, resulting in greater granularity while distributing partitions to multiple nodes and searching smaller batches in parallel. Once a database is distributed by segmenting it into partitions and distributing partitions to multiple computing nodes, the query batches can be searched in parallel on multiple nodes [178].

Decomposing large sequencing datasets into partitions and distributing the data partitions into a distributed file system, such as HDFS [36], enables scalable data-parallel sequence alignment with Apache Spark [58]. Spark can be further utilized to refine aligned sequence data through pipeline stages with rich analytic functions in parallel. Aggregating sequence alignments from multiple samples is often needed in the analysis pipeline and can be performed efficiently with dataset operations such as reduce, group, join, and aggregate in Spark. Apache Spark has been used for distributed BLAST sequence alignment in a metagenomics data analysis pipeline in

Publication I. Moreover, efficient indexing and compressing methods for massive datasets become important when large-scale sequence alignment has to be performed. Large sequencing datasets can be compressed and indexed to reduce storage space usage and sequence alignment time (see Section 3.6) with a distributed compressed hybrid-indexing method, as shown in Publication III.

4.3.2 Metagenomic identification

Alignment routines are sometimes used iteratively in both secondary and tertiary analysis. In a metagenomic sample, multiple species are abundant and thus classification of the species is needed [179]. *De novo* assembled sequences are typically aligned to known species for screening, i.e., filtering the contaminated or unrelated sequences, and for finding related sequences [46, 47, 170]. After the screening, the sequences discovered are often annotated taxonomically and functionally [174]. These annotated novel assemblies can be then used as references for further studies on related species. As more sequences from the same species are assembled and annotated, the consensus reference genome can finally be constructed and used for re-sequencing purposes.

In metagenomic studies, the bacterial, viral, phage, and vector sequences can be identified by aligning assembled contigs with known microbial reference sequences or searching, e.g., with BLAST from nucleotide databases such as RefSeq [180], Enterobase [181], and VIPR [182]. The remaining sequences are then identified by searching databases corresponding to the species under investigation. Tertiary analysis in metagenomics may require the profiling and classification of unknown species, strains, and serotypes with methods such as nucleotide or amino acid homology searching [171], or Multilocus sequence typing (MLST) [183]. Gene annotation is typically performed at this point for enabling further downstream analysis. Figure 4.6 introduces a simplified parallel pipeline to identify genomes from metagenomic NGS samples.

The database replication and parallel sequence search methods are applied in Publication I to profile metagenomes from human NGS samples using BLAST and HMMER databases in the distributed Spark computing cluster.

4.3.3 Variant calling

Variant calling [116] often follows the read alignment step in re-sequencing analysis. Variant calling is a set of computational methods for detecting

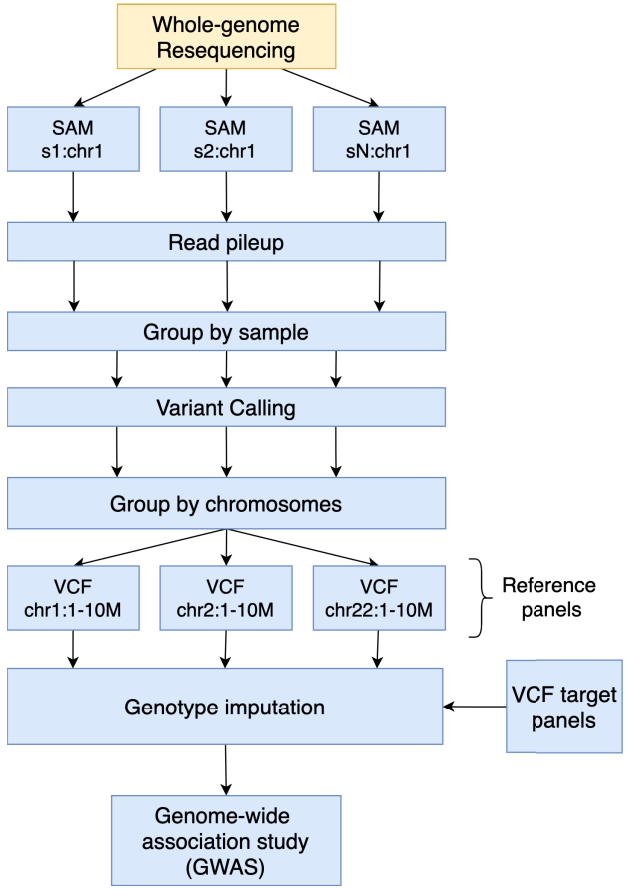

Figure 4.7: Tertiary analysis from re-sequencing to genome-wide association study.

genotypes and is a premise for various further analyses. In addition, the existing genotype data is used in many comparative studies. Basically, variant calling estimates the likelihood of the allele in a genomic locus relative to a reference genome and outputs the most probable allele in that position. Assuming that mapped alignments are filtered and other preprocessing steps have already been performed in secondary analysis, traditionally the next step would be read pileup [117], previously described in Section 3.5.2. Figure 4.7 illustrates how the read pileup and variant calling are related in tertiary analysis. We can start by decomposing read alignments (SAM files [34]) into partitions by sample and chromosome. Pileup is then done in parallel per chromosome in every sample. Variant calling typically produces a VCF [35] file where each line represents a genomic position in a

reference genome and stores reference allele and Single Nucleotide Polymorphism (SNP) likelihoods. Figure 4.7 shows a simple illustration of how the read pileup is done in parallel when data is decomposed by sample and chromosome (s1:chr1, s2:chr1, sN:chr1). Read pileups are then grouped by sample for actual variant calling, which produces VCF files. Reference panels are composed of VCF files by grouping SNPs by chromosomes and aggregating samples into a single file. Read pileup and some of the general pre- and postprocessing methods for variant and alignment data have been parallelized with Spark in the latest GATK [66].

4.3.4 Genotype imputation

Genotype imputation is routinely used in genome-wide association studies (GWAS) when only a subset of variants is available and large-scale comparison of variants is needed. The subset of variants can be detected by any available genotyping method, e.g., by DNA microarrays [110] or by sequencing [112]. Imputation is done by estimating unknown Single Nucleotide Polymorphism (SNP) from the previous observations in the genotype reference panel. Data-parallel decomposition can be applied to chromosomal regions where groups of variants are known to be inherited together. These variants are said to be linked by linkage disequilibrium (LD) [184]. Linked variant sets allow highly parallel imputation in partitioned regions. Widely used Beagle [115] and Impute [185] imputation tools impute genotypes in allele marker intervals with high accuracy in parallel. Figure 4.7 illustrates how genotype imputation is related to variant calling in tertiary analysis. SNP data for imputation is parallelizable at the chromosomal level and regions. With the simplified example in Figure 4.7, imputation is done in chromosomal intervals of size 10 million SNPs. This partitioning strategy enables data-parallel imputation without rewriting the existing algorithms.

Distributed file systems enable the storage of sets of variants in distributed data partitions and thus imputation can be done in parallel on local data without reloading or moving data outside the computing node, which reduces the communication overhead. This approach is applied in Publication II for massive-scale distributed genotype imputation.

Chapter 5

Scalable population genomics applications

This chapter introduces the scalable whole-genome sequencing data analytics applications developed in this dissertation. The implemented applications originate from Publications I-IV and are presented in this respective order. A brief introduction to each application is presented, followed by an overview of the overall design and key findings.

5.1 Large-scale identification of pathogens from whole-genome sequencing data

Metagenomics studies the diversity of genomes in a complex sample comprising various organisms of different species [179]. Metagenomes can be discovered by sequencing microbiological genomes that may be present in a sample or host. An estimated 10^{12} microbial species exist on Earth [43]. Rapid identification of microbes is essential to design antibodies for vaccines and improve drug development for quick response to epidemics and pandemics [13, 186, 187, 188]. Whole-genome sequencing (WGS) of human, food, or environmental samples now enables the identification of all the microorganisms present in a biospecimen. Moreover, WGS enables the analysis of total microbial DNA directly from its natural environment, i.e., microbial community, thus reducing the time consumed in laboratory work [168]. Moreover, this also enables the identification of novel unknown sequences by assembling sequences computationally. The source of pathogens can be tracked with high confidence when an animal or foodborne pathogen is linked to the pathogens identified from human patients. For example, ongoing SARS-Cov-2 (COVID-19) outbreak cases have been tracked in near

real-time with the Nextstrain analysis tool from sequenced viruses [189]. Epidemiological tracing of cases and contacts during the Ebola outbreak in western Africa in 2016 was conducted utilizing Next-generation Sequencing data analysis [190]. The origin of the 2011 E. coli outbreak in Europe was tracked by identifying genetic mutations of bacterial strains from WGS data [191]. The human microbiome comprises bacteria and viruses, among other microbes [192, 193], and their proportion and composition vary between individuals [159, 194]. Systematic classification of microorganisms is challenging, especially because of their rapid recombination rate, the relatively small genomes, and low concentration and incidence [195]. WGS-based analysis methods can segregate bacterial serotypes and strains with high precision and resolution more rapidly than traditional laboratory methods using cultured microbes [196]. Thus, algorithms and tools for metagenomics will have to cope with a huge amount of WGS data in the near future.

5.1.1 Distributed identification of viruses from multiple metagenomic NGS samples with ViraPipe

Chapter 4 reviewed distributed approaches for metagenomic read alignment and *de novo* assembly. A combination of the methods reviewed was used in Publication I to develop a scalable distributed metagenome analysis pipeline, ViraPipe. ViraPipe is designed especially for detecting viral sequences from NGS reads sequenced from human samples, but can easily be configured for identifying other microbes. The pipeline was developed on the Apache Spark framework and it comprises the following distributed sequence analysis methods (Figure 5.1):

1. NGS read data quality filtering and duplicate removal. Read alignment with distributed read data partitions.

2. Filtering out the human genetic material by discarding the mapped reads from the distributed read data partitions. The unmapped SAM formatted alignments are transformed back to FASTQ format in the same distributed phase.

3. Normalization of unmapped reads. The normalization is executed on distributed read data partitions in parallel.

4. *De novo* assembly of preprocessed unmapped reads in parallel per sample with MegaHit.

5. Taxonomical identification of known viral sequences from the replicated BLAST database from distributed contigs in parallel.

6. Novel viral sequence profiling with HMMER search from the vFam protein database from distributed contigs in parallel.

More detailed descriptions of the pipeline phases are given in Publication I. The experiments have been conducted to analyze viral fractions from various human biospecimens sequenced with Next-generation sequencing technologies. The analysis pipeline consists of mining viral candidate sequences from heterogenous NGS reads, assembling viral contigs, discovering viral genomes, and profiling viruses. ViraPipe analyzed 768 human samples (570.9 GB dataset) in 210 minutes with Apache Spark on the Hadoop computing cluster, including 5.12 TB of RAM and 1288 cores distributed to 23 worker nodes (Figure 5.2). 11x speedup was achieved on 23 computing nodes compared to the sequential pipeline run on a single node.

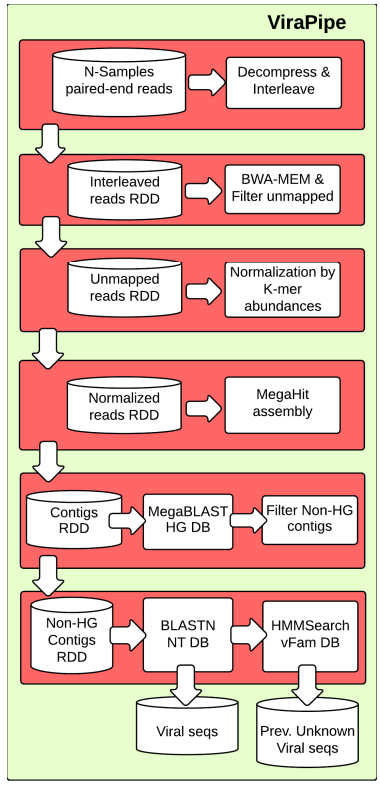

Figure 5.1: Overview of the analysis phases in ViraPipe pipeline [197].

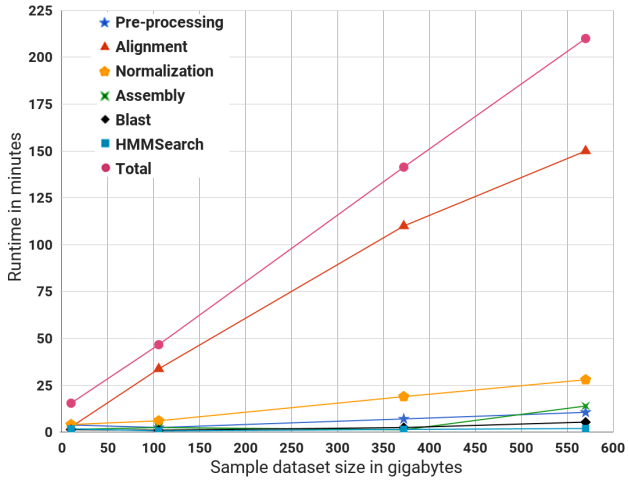

Figure 5.2: Scalability of the ViraPipe in respect to the NGS dataset size [197].

ViraPipe discovered 12.5% more viral sequences than the sequential pipeline with the test dataset due to distributed read alignment and alignment normalization. This can be possibly compensated with post-filtering of normalized alignments by their similarity in expense of execution time. The pipeline was performing better with larger number of samples, which indicates that the granularity of data partitions should have been higher with a small number of samples. Granularity can be configured with Spark RDD repartitioning or coalescing transformations, and by setting the HDFS block size. The optimal granularity depends on the underlying computing cluster size and resources such as available memory and cores. The complete assessment of the pipeline is given in the original Publication I. The ViraPipe source code is available in Github[1].

5.1.2 Bacterial pathogen identification with ViraPipe

The ViraPipe pipeline (Figure 5.1) has been further extended with Multilocus sequence typing (MLST) to identify bacterial pathogens from Next-generation sequencing food samples presented in Figure 5.3. This work was done for the PrecisionFDA CFSAN Pathogen detection challenge[2] and is not part of Publication I.

U.S. Food and Drug Administration (FDA) provided NGS data for 32 food samples for evaluation from which 24 samples had to be identified

[1] https://github.com/NGSeq/ViraPipe
[2] https://precision.fda.gov/challenges/2

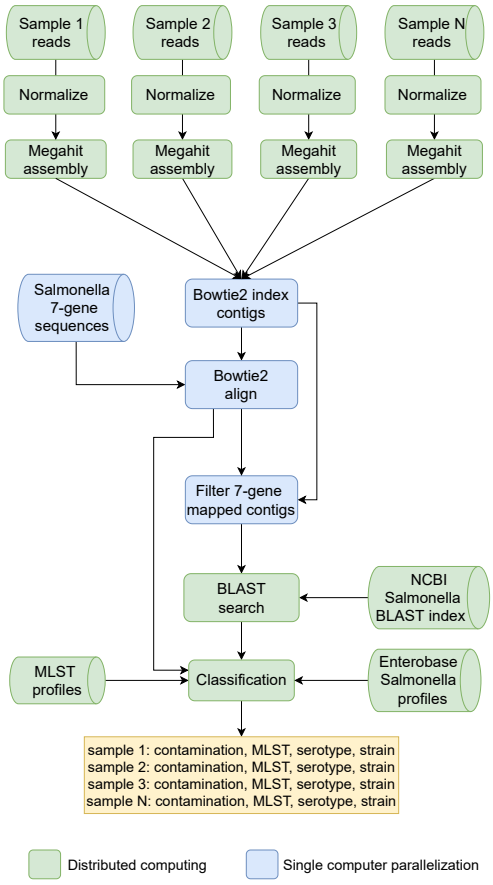

Figure 5.3: Salmonella pathogen identification pipeline.

and 8 samples were example samples. The goal was to identify the strain, serotype, and MLST type of Salmonella-positive samples. MLST scheme defines 7-gene profiles of known Salmonella strains where the gene profile refers to the corresponding allele. MLST is a method for identifying bacterial isolates based on the comparison of variants of selected genes [183]. MLST uses the genetic loci of multiple genes to profile the strain of a bacterium. MLST scheme defines allelic profiles of multiple genetic loci of known bacterial strains that have been already identified, e.g., by using DNA microarrays.

24 NGS read data sets (82 GB uncompressed data in FASTQ format) are first normalized to remove duplicates and closely similar sequences. After pre-processing, the candidate contigs are *de novo* assembled with a MegaHit [198] assembler from NGS reads in parallel per sample. 7-gene

profiles are used as a reference sequence and candidate contigs are aligned with the Salmonella 7-gene reference index using the BWA-MEM [37] algorithm. The best scoring alignments are then filtered from the produced SAM files. MLST types are inferred from the MLST database using the best matching gene profiles of each sample. Candidate serotypes and strains are searched from the Enterobase database by 7-gene MLST types and 7-gene matched contigs from the BLAST Salmonella database. The final classification is performed with an algorithm that scores MLST types by the number of correctly matched 7-gene profiles and infers serotypes and strains from the best-matched BLAST and Enterobase search results. If closely related MLST profile was not found from the Enterobase database, the strain was identified only by the BLAST database and serotype was left unknown.

Executing the whole pipeline took 341.8 minutes using Spark computing cluster including 1 TB of RAM and 400 cores distributed to 25 nodes (Table 5.1). The most computing intensive phase of the pipeline is the Megahit *de novo* assembly which took 84% of the execution time. The assembly input data is decomposed by sample and 24 Megahit tasks are run in parallel on distributed nodes. As the sample data set sizes vary, the load balancing is not optimal and the execution time is the time taken to assemble the largest sample data. After the assembly phase, the input data sizes decrease to few gigabytes (3 GB of contigs). Bowtie2 indexing of contigs and 7-gene alignment are done on a single node as partitioning of the index was not currently possible, but has been later studied in Publication III. Searching the 7-gene mapped contigs from the Salmonella BLAST database is done in parallel on distributed contig partitions. As contigs length vary a lot (minimum length 714 bp and maximum length 214785 bp with mean length of 20814 bp) and the BLAST performs slowly with long contigs, the load balancing here was unequal causing performance degradation. Classification phase was performing without load balancing issues as it is parallelized with constant size data partitions using Spark RDDs. Salmonella contamination was correctly predicted from 18 out of 24 samples including no false positives and six false negatives. From thirteen true positive samples four MLST profiles, three serotypes, and zero strains were correctly predicted. The overview of the official results reported by the FDA is available at the competition website[3].

The strain prediction was done last and it turned out to be a challenging task in the limited competition time frame. Lately, a few k-mer alignment based tools have been developed for bacterial strain detection

[3]https://precision.fda.gov/challenges/2/results

such as Kaiju [162], Centrifuge [163], and Kraken2 [164], that can be potentially used to improve the pipeline. Machine learning classification methods could potentially be used to improve MLST, serotype and strain prediction accuracy, and should be further explored [165, 166]. The performance in assembly phase can be increased by decomposing sample data sets into smaller partitions, however, this would affect to assembly accuracy as some contigs would be lost if sequences are not concatenated between the partitions. Read clustering methods [167] can be useful here for grouping reads having closely similar prefixes and suffixes as *de novo* assembly uses them to concatenate read sequences (see Section 3.4 and Section 4.2.2). The ViraPipe bacterial pathogen identification extension is available in Gitlab[4].

Pipeline phase	Exec. time (min.)
Normalization	11
Megahit assembly	287
Bowtie2 indexing	36
7-gene align	0.5
Salmonella DB indexing	0.7
BLAST 7-gene contigs	4.6
Classification	2.0
Total time	341.8

Table 5.1: Salmonella type identification pipeline execution times.

5.2 Distributed large-scale genotype imputation

Genotyping basics are presented in Section 3.5 and in Section 4.3.4. A hybrid of the methods reviewed was applied in Publication II to develop a scalable distributed genotype imputation tool, SparkBeagle[5].

Existing WGS-based large genotype reference panels offer accurate detection of missing genotypes through genotype imputation. Genotype imputation from WGS-based reference panels is a cost-efficient way to computationally detect all the variants in the human genome [113, 199]. Genotype imputation panels may contain tens of thousands of samples and tens of millions of variants, generating massive data sets. The current imputation algorithms use complex imputation methods, and imputing the missing genotypes of multiple target samples from thousands of reference genotypes is too computationally heavy for algorithms running on a single computer.

[4]https://gitlab.com/aimaarala/pathospark
[5]https://github.com/NGSeq/SparkBeagle

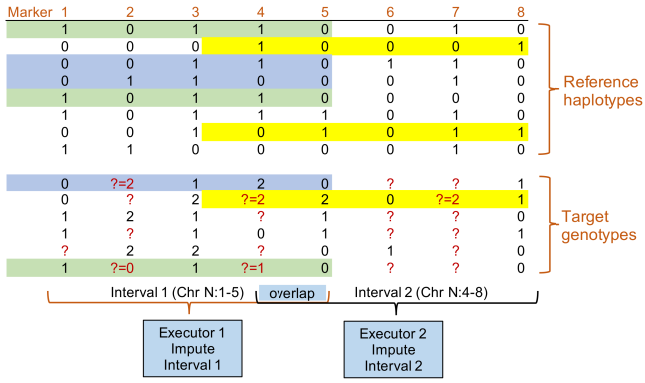

Figure 5.4: A simplified parallel imputation scenario [200].

A simplified parallel imputation scenario with diploid genomes is illustrated in Figure 5.4. The goal is to impute missing target genotypes denoted with a question mark. In a real-life scenario, the marker numbers would represent physical chromosomal positions of Single Nucleotide Polymorphism (SNP). A row represents one sample haplotype and a column represents an allele in some genomic position. The genotypes in a target sample are imputed from reference haplotypes highlighted in the same color. The genotypes are imputed in parallel across overlapping markers. In Figure 5.4 only one marker is overlapped as an example.

In SparkBeagle, imputation reference and target panels are decomposed by chromosomes and stored into the Hadoop Distributed File System in block compressed (BGZF)[6] variant call format (BCF). The distributed compressed reference panel blocks are read from the HDFS locally on each node into distributed memory using Resilient Distributed Datasets (RDD) with Spark. The imputation is done massively parallel from a distributed reference panel in chromosomal intervals with Spark. SNP data is processed in parallel by splitting SNP markers into overlapping windows, which enables more fine-grained data decomposition in chromosomal regions considering linkage disequilibrium [184]. The overlapping partition strategy is used to mimic the original Beagle's interval overlapping so that the imputation algorithm performs accurately on partition boundaries as well (Figure 5.5). However, imputation interval and overlap size can affect the performance and imputation accuracy: too large an imputation interval and overlap size can result in uneven load balancing, thus decreasing scalability and performance. In contrast, smaller imputation intervals give better load balancing

[6] https://samtools.github.io/hts-specs/SAMv1.pdf

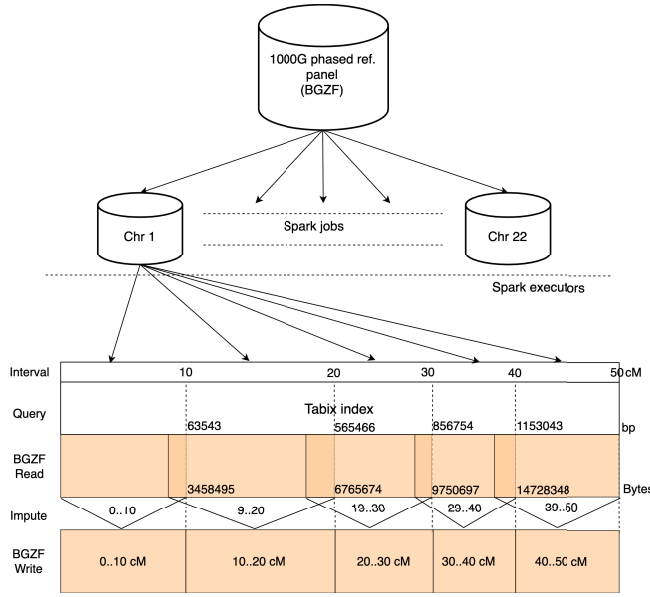

Figure 5.5: Distributed imputation of overlapping allele marker intervals in SparkBeagle. The input reference panel markers are decomposed into imputation intervals by genomic regions defined in units of centimorgans (cM) [200].

and are imputed faster in parallel but too small an interval and overlap size can reduce the imputation accuracy, an effect that was also pointed out in the original Beagle results [201].

The input data is partitioned into the distributed memory by the intervals using Spark's Resilient Distributed Datasets (Figure 5.6). The distributed partitions are mapped to Spark executors across the computing cluster for parallel task execution. Imputed intervals are written to HDFS and numbered according to the chromosomes and marker positions. Ordered file numbering reduces post-processing operations like sorting the output data.

In SparkBeagle experiments, 1000 Genomes phase 3 genotype data[7] was used as a reference panel including 2504 samples (n=81,214,785 variants, 769 GB). The target panel for the scalability study was created by extracting every fifth marker from the reference panel (n=16,238,469 variants, 154 GB). The total number of imputed variants was 64,976,316. A subset of 861 individuals of HapMap 3 data[8] was derived for the concordance analysis,

[7] ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/
[8] https://www.sanger.ac.uk/resources/downloads/human/hapmap3.html

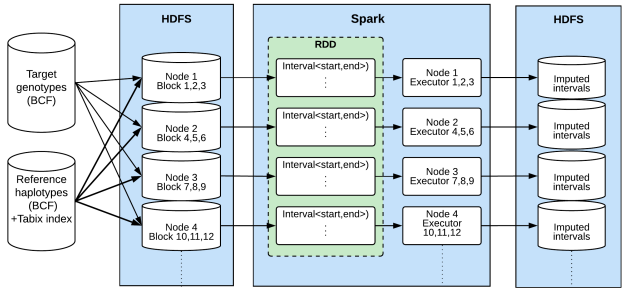

Figure 5.6: Distributed Spark architecture implemented in SparkBeagle [200].

resulting in the target panel of size 220,017 variants. Near linear scaling with the increasing number of nodes can be observed from Figure 5.7.

The whole dataset was imputed in 18 minutes, resulting in 30x speedup on 40 Spark worker nodes. Near identical imputation accuracy was observed between the original Beagle and SparkBeagle. As Beagle and SparkBeagle were executed with the same interval and overlap sizes, a small variation in accuracy is associated with the pseudorandom number generators used in heuristic imputation algorithm[9].

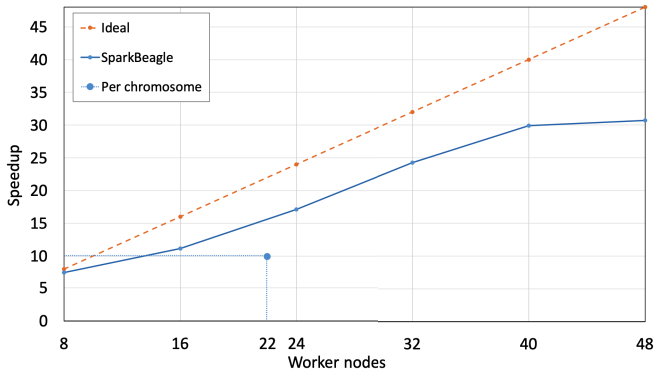

Figure 5.7: Speedup with increasing cluster size compared to Beagle running in parallel per chromosome on 22 nodes. Dashed line denotes an ideal speedup [200].

[9]https://faculty.washington.edu/browning/beagle/beagle.html

5.3 Scalable compressed indexing and sequence alignment with pan-genomes

Pan-genomics is an effort to exploit information in multiple genome assemblies for comparative variation analysis in population and evolutionary genomics [202, 203, 204]. Pan-genomic references can improve genetic variation discovery amongst species or subpopulations by emphasizing the genetic diversity of individuals and recombinations. Thus, pan-genomes have been used successfully for, e.g.,genetic ancestry discovery, evolutionary genomics, and pathogen discovery [205, 206, 207, 208]. The first studies were performed on Streptococcus strains to study pathogenic isolates and developing vaccines [209]. Pan-genomic variant calling and analysis basically require re-sequencing or *de novo* assembling of all genomes in a population under investigation, which is computationally demanding. A pan-genome can be presented as a set of genomes, i.e. core genome, which contains the common sequences across all genomes, and with dispensable genes that are absent in one or more genomes [210]. Alternatively, a pan-genome can be represented with a consensus genome by applying the most frequent variants to the reference genome of a species or with a Direct Acyclic Graph (DAG) including common sequences and all variants [16, 211]. Consensus pan-genome represents the most frequent variations across the population in a single genome and can thus be directly indexed and aligned with conventional read aligners [49]. Sequence compression and indexing methods become even more important on a pan-genomic scale as data volumes grow directly in proportion to the number of genomes included.

In Publication III, a distributed compressed hybrid-indexing technique was studied and found practical in indexing large collections of repetitive genomes, i.e., pan-genomes. Hybrid-index [42] supports also direct read alignment and sequence search with compressed pan-genomes [49]. Genomic data compression and indexing methods have been reviewed in Section 3.6.

Genomic data volumes are growing fast while sequencing technologies are advancing, making genomic data compression and indexing methods eer more significant. Current sequence compression and indexing methods often use sequential algorithms that are computationally time-consuming, and do not provide efficient sequence alignment performance on very large collections. Scalable compressed indexing methods for repetitive sequences can foster massive genomic data storage and analysis burdens when coupled with sequence search and alignment capabilities.

Lempel-Ziv [124] based lossless data compression algorithm, Relative Lempel-Ziv (RLZ) [40, 41] is a proven space- and time-efficient method for compressing highly repetitive sequences. RLZ is a modification to the classical LZ77 method with a difference that RLZ uses a predefined subset of the input data as a compression dictionary instead of previous characters such as LZ77's sliding window does. To reduce the dictionary coding and compression time, a distributed version of the RLZ compression method was developed in Publication III. Distributed RLZ partly follows the method provided by Hoobin et al. [41], with the modification that blocks (i.e., partitions) are encoded separately in parallel with the coding dictionary extracted from the prefix of each block instead of using the same dictionary to encode every block. Moreover, for the encoding part we use the LZ factorization algorithm KKP3 presented by Kärkkäinen et al. [141, 213]. The suffix-array for the dictionary is constructed in parallel for each partition using a Scala implementation of the DC3 algorithm originally presented by Kärkkäinen et al. [135]. Technically, the sequence is processed character by character by searching a longest common prefix (LCP) from the sorted suffix array [213]. The sorted suffix array enables that all the possible matches can be found quickly with a binary search that searches the upper and lower bound indexes of LCP in the suffix array. LCP is searched by increasing the query substring length until difference to the upper and lower bound is minimized using Succint Data Structures Library (SDSL)[10]. The found LCP is then encoded in 64 bit $\langle$ pos, len $\rangle$ pairs referring to dictionary where *pos* is a starting position of the LCP and *len* is the LCP length indicated by the suffix array.

Figure 5.8 presents the developed distributed compression and indexing pipeline. The pan-genome is decomposed into chromosomes per genome that are distributed across the cluster using Hadoop Distributed File System (HDFS) [36] and the chromosomes having the same identifier, e.g., number, are decomposed into constant length partitions while reading data into memory using Spark RDDs. Each partition is RLZ compressed in parallel, and eventually the compressed partitions are merged keeping the original order. LZ is used for compression method as it has data structures that are decomposable into distributable partitions and RLZ has been already integrated with hybrid-indexing method [48].

Figure 5.9 illustrates the sequence alignment workflow with the compressed hybrid-index and CHIC [146] aligner. Valenzuela et al. [48] propose a CHICO indexer based on hybrid-index implementation using RLZ compression with kernelization for compressing pan-genomes.

[10] https://github.com/simongog/sdsl-lite

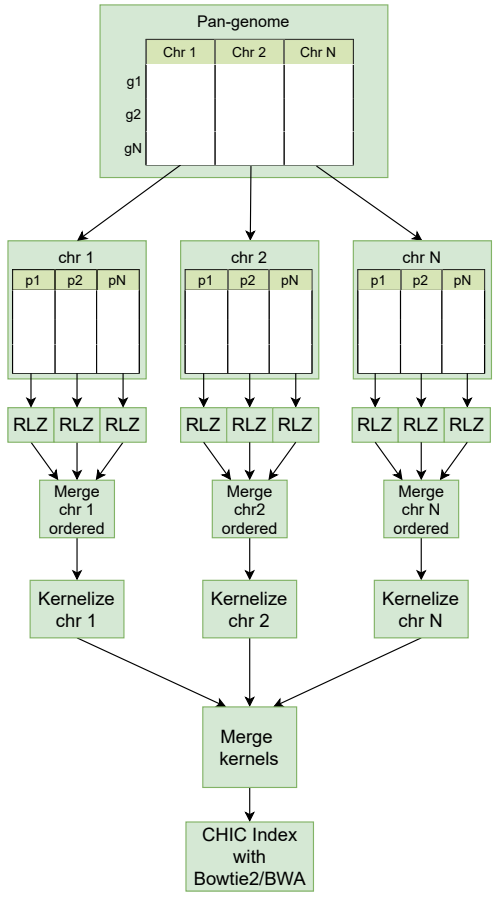

Figure 5.8: Distributed Relative Lempel-Ziv compression and hybrid-index pipeline [212].

Hybrid-indexing presented in Publication III uses the CHICO [48] indexer extended with distributed RLZ (DRLZ) and kernelization technique (see Section 3.6.2). The kernel representation represents the original sequences in a non-repetitive sequence utilizing RLZ encoding. Eventually, the kernel sequence produced is indexed with CHICO using some of the supported conventional indexes including BWA, Bowtie, and BLAST. The resulting index is directly accessible with the supported aligners.

Apache Spark computing cluster comprising 448 cores distributed over 25 nodes was used for experimenting both with human and bacterial genomes (Figure 5.10). The BLAST index for (n=1,000) human pan-genome was built in 1520 minutes with a compression ratio of 532:1, and the Bowtie2

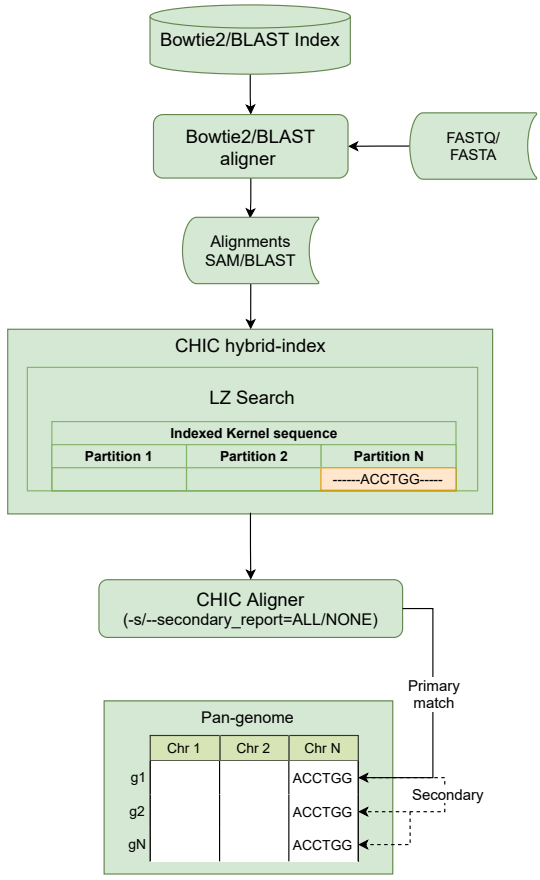

Figure 5.9: Workflow for using hybrid-index with conventional aligners [212].

index was built in 1938 minutes with a compression ratio of 76:1 CR (see Publication III). 14.6 GB of paired-end reads aligned with the compressed human pan-genome (n=1,000) index in 31.7 minutes on a single node with Bowtie2 (Table 5.2). Searching 189,864 Crispr-Cas9 gRNA target sequences (23 MB in total) from the compressed human pan-genome (n=1,000) with BLAST took 45 minutes on a single node (Table 5.2).

Compressing the BLAST index from the bacterial sequence database (n=13,4M) took 575 minutes, resulting in a compression ratio of 62:1 (Figure 5.11). Compressing the 18 GB E. coli data set took 33 minutes using the whole computing cluster, and indexing the compressed E. coli data set

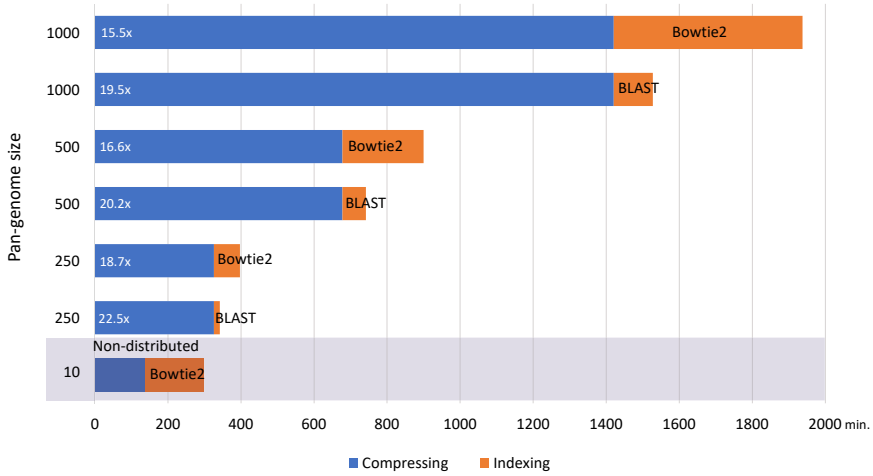

Figure 5.10: Distributed compressing and indexing performance with complete human pan-genomes compared to non-distributed method. Calculated speedup relative to the sequential pipeline is denoted by a number with x [212].

Tool	n	Query sequences	min	Mapped
Bowtie2	1000	2x28.86M (2x7.3 GB)	31.7	10.12M
BLAST	1000	189.9k (23 MB)	45.2	639k

Table 5.2: Aligning sequences to a compressed pan-genome index including 1000 human genomes.

on a single node took 2 minutes, resulting in a compression ratio of 102:1 with BLAST. Indexing the compressed E. coli dataset with Bowtie2 took 8 minutes, resulting in a compression ratio of 51:1. 4,200 bacterial sequences aligned with the compressed E. coli index containing 745k sequences (18 GB) in 5.38 minutes with BLAST (Table 5.3). Aligning 599 bacterial sequences (30 MB) with the compressed GenBank BLAST index including 13,4 million sequences (488 GB) took 26 minutes (Table 5.3). Bowtie2 read alignment took 12 minutes with the compressed index of 488 GB bacterial sequences and 5.94 minutes with the compressed E. coli index (Table 5.4).

If chromosomes in pan-genome are divided into constant size partitions, the shorter chromosomes would produce less partitions, thus, shorter chromosomes have been divided into smaller partitions for achieving more fine-grained parallelization. However, too small partitions can affect negatively

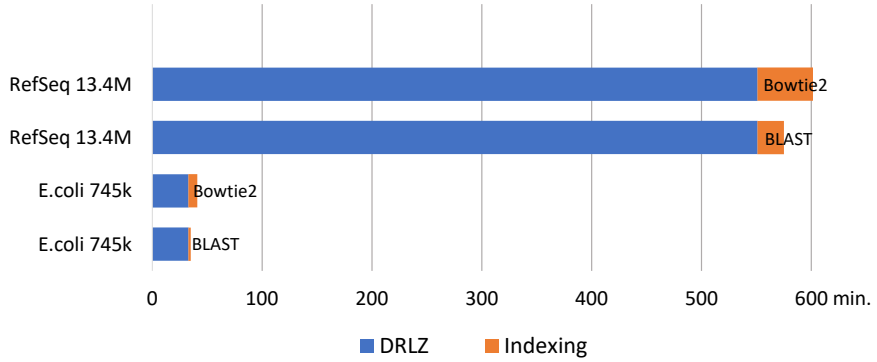

Figure 5.11: Distributed compressing and indexing performance with bacterial sequences [212].

Pan-genome (seqs., size)	Query seqs. (size)	Time (min)
E.coli (745k, 18 GB)	4.2k (78 MB)	5.38
GenBank (13.4M, 488 GB)	599 (30 MB)	26.17

Table 5.3: Aligning bacterial sequences to a compressed index with BLAST.

Pan-genome (seqs., size)	Reads (size)	Time (min)
E.coli (745k, 18 GB)	3.1M (792 MB)	5.94
GenBank (13.4M, 488 GB)	27.2M (4334 MB)	12.0

Table 5.4: Aligning next-generation sequencing reads to a compressed index with Bowtie2.

to the compression ratio. Each partition is compressed in parallel, and eventually the RLZ compressed partitions are merged by chromosomes. BLAST indexing has been done in parallel per chromosome and thus indexing of longer chromosomes took more time creating uneven load balancing as each chromosomal partition was indexed in separate nodes in parallel. The same rule applies to Blast alignment that has been done in parallel per chromosome and to distributed kernelization phase used with BWA and Bowtie2 indexes. The most I/O heavy task observed is the DRLZ compression as it requires loading the pan-genome and chromosomal suffix arrays into Spark RDDs as well as writing temporary pan-genome plus encoded sequences to the HDFS. However, HDFS performed fluently with the largest pan-genome when default block replication factor of 3 was used. One Spark transformation – *groupBy* – produces remarkable network and I/O overhead inside the DRLZ method due to its shuffle operation that redistributes the data across

the nodes. The most CPU heavy and memory intensive tasks observed are indexing and read alignment with the BWA and Bowtie2 indexes as they are not using distributed parallelization. However, the compression ratio affects to the amount of memory and indexing time used in the indexing phase as the input data size varies directly proportional to the CR. Feasible compression ratios and indexing times were achieved with all pan-genome sizes.

5.4 Assembling reference pan-genomes using compressed hybrid-indexes

The pan-genomic variant calling approach can improve genetic variation discovery within populations by considering genetic diversity and recombination in individuals [16, 211, 214, 215]. The idea is to identify the most similar sequences in a population, assemble these sequences into a reference pan-genome, and finally call the donor variants using this reference pan-genome [16]. The original idea has been realized by Valenzuela et al. [49, 216] by implementing a PanVC pipeline (Figure 5.12) to call variants over pan-genomes using a consensus reference genome (i.e. ad hoc) computed from the pan-genome. Figure 5.12 presents the sequential PanVC pipeline [49] that the distributed version implemented in Publication IV is based on. Like the traditional variant calling with a single reference genome (see Section 4.3.3), the pan-genomic variant calling is also preceded by the read alignment, except that the donor genome reads are first aligned with all genomes using a compressed pan-genomic index with CHIC aligner (Assume that pan-genome index has been already constructed as described in Section 5.3). The similar sequences to the pan-genome are then identified from the mapped reads by scoring the mapped reads by their pan-genomic coverage with the score matrix, calculating the heaviest path from the score matrix, and extracting the consensus (i.e., *ad hoc* in Figure 5.12) reference pan-genome by following the heaviest path in a pan-genome. Finally, in the variant calling step, the donor reads are aligned with a reference pan-genome instead of a standard reference genome.

In Publication IV, a distributed pipeline PanGenSpark (Figure 5.13) was implemented to assemble a reference genome from a compressed pan-genome index based on the sequential pipeline (Figure 5.12) presented by Valenzuela et al. [49]. The distributed hybrid-indexing and compression techniques presented in Publication III are used in this work to improve the scalability of the index construction in Phase b. The proof of concept pipeline has been experimented with a population of 500 haploid genomes.

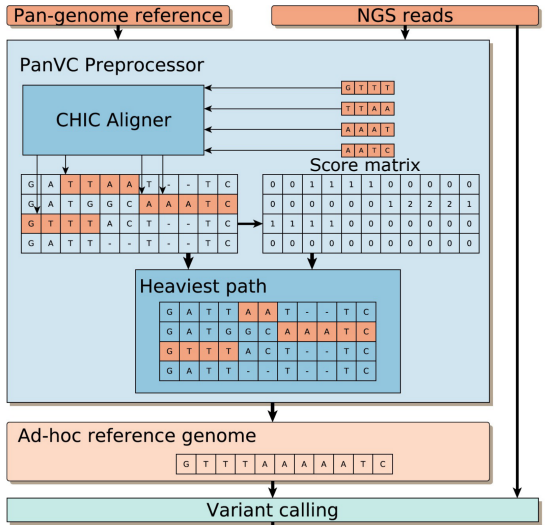

Figure 5.12: Sequential reference pan-genome construction in PanVC by Valenzuela et al. [49].

The PanGenSpark pipeline consists of the phases denoted with a letter (a - e) in Figure 5.13 as follows (the methods are presented in Publication IV in greater detail).

a) Pan-genome preparation

The pan-genome is constructed of multiple genome assemblies. If assemblies are already provided, the pan-genome can be constructed by loading genomes in FASTA format into a Hadoop File System (HDFS) directory. In our experiment, genomes are assembled by applying variants from VCF files to a standard reference genome with the `vcf2multialign`[11] tool. If diploid genomes are used, `vcf2multialign` generates both haploids per individual. `vcf2multialign` alignment also generates gaps (-) in case of deletion that are stored into gap position files to fill the score matrix in Phase d correctly. The standard reference genome is added on top of the pan-genome.

b) Compressing and indexing the pan-genome

The pan-genome is compressed with the Distributed Relative Lempel-Ziv (DRLZ) method and the hybrid-index is constructed using the methods

[11] https://github.com/tsnorri/vcf2multialign

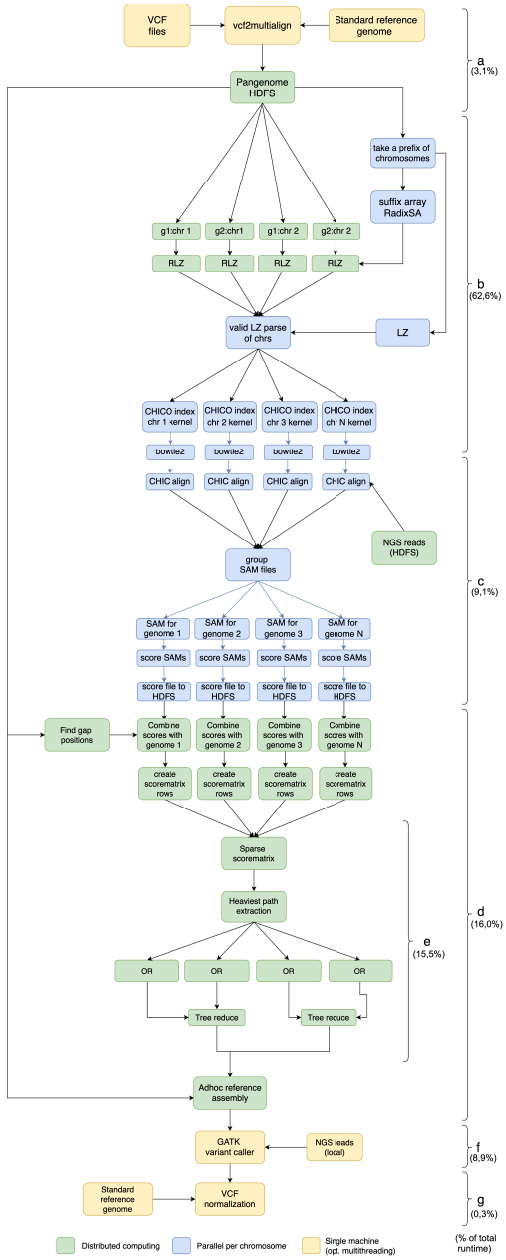

Figure 5.13: PanGenSpark, a distributed reference pan-genome assembly pipeline. Percentage denotes relative amount of computation time spent in each pipeline phase (a - g) [217].

presented in Publication III. The partitioned pan-genome is compressed using distributed RLZ factorization (see Sections 3.6 and 5.3). Each individual human genome in the pan-genome is stored into HDFS in FASTA format divided by chromosomes. The data is loaded into the Spark Resilient Distributed Dataset from the HDFS and repartitioned into similar-sized partitions to equalize distribution and improve load balancing. Each distributed chromosome partition is RLZ compressed in parallel generating N x 22 partitions where N denotes the number of genomes in a pan-genome. Hybrid-indexing is based on the CHICO indexer [48], which we modify to exploit distributed computing. The LZ parse generated by RLZ factorization is collected from the HDFS by chromosomes to the local filesystem in separate nodes and the chromosomal kernel representations are then constructed and indexed with CHICO [48] using the Bowtie2 index.

c) Read alignment with the compressed pan-genome index

The donor genome reads are aligned with distributed chromosomal pan-genomic indexes with CHIC aligner using Bowtie2. The read alignment process follows the workflow presented in Figure 5.9. The mapped reads are grouped by the individual genomes, and scored by their mapping coverage in each genomic position (see Figure 5.12). The score files of each genome (corresponds to the row in the score matrix) are loaded to HDFS and read into the score matrix in the next phase.

Bowtie2 aligns reads with an indexed kernel sequence and alignments are transformed to original pan-genomic positions using the LZ parse and auxiliary data structures generated by the CHICO index in Phase b. The CHIC aligner has different parameters (see Figure 5.9) for searching only primary matches (default), primary plus secondary matches (-sALL), and a Bowtie2 specific `bowtie2 -all` option that reports all approximate alignments. Aligning with chromosomally distributed indexes can find the secondary matches only from the same chromosome that the primary match aligns with. As we align the total reads with every chromosome separately, it may result in better scoring in some chromosomes than would be the case if we aligned with a single index. This can be fixed, e.g., by removing duplicate alignments or filtering by alignment quality.

d) Consensus reference genome assembly

If the pan-genome is produced with the gaps (as is the case when using vcf2multialign in Phase a, the alignment scores are fixed to correspond to the gapped positions in the pan-genome. The scores are loaded into a dis-

tributed score matrix using Spark Mllib BlockMatrix and used to calculate the heaviest path for each chromosome in Phase e.

e) Heaviest path

The reference genome is assembled from the heaviest path extracted from Spark RDD distributed score matrix blocks in parallel by chromosome. The heaviest path is the path through the score matrix that maximizes the sum of the scores over the score matrix (see Figure 5.12). In practice, the score matrix is sparse having the same dimensions as the pan-genome, where the top vector (row) contains the most of the matches. To obtain the consensus (i.e., *ad hoc*) reference genome, the corresponding nucleotides pointed by the heaviest path vector are extracted from the pan-genome in parallel. To transform from the heaviest path number representation to the corresponding DNA reference sequence, the pan-genome is read into Spark RDD in parallel by chromosomes and individual genomes from the HDFS and the heaviest path vector is broadcasted to each node for enabling distributed transformation. The distributed heaviest path to consensus genome transformation is done for each chromosomal partition in parallel. If the current pan-genome row index matches the row number pointed by the heaviest path in the current sequence position (Figure 5.14) the corresponding nucleotide is returned and otherwise '?' is returned. Finally, the sequence branches are combined into a single consensus reference genonme in parallel using the Spark RDD tree-reduce transformation.

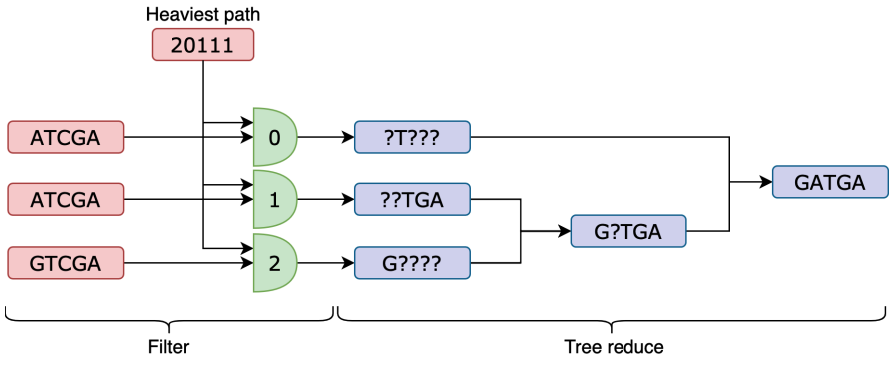

Figure 5.14: Example of extracting the consensus reference sequence from the calculated heaviest path with three sequences (Erratum: Publication IV, Figure 4.) [217].

f+g) Variant calling with the consensus reference genome

Variant calling with the obtained consensus reference genome can be performed using common variant calling pipelines such as Samtools+bcftools[12] or GATK[13]. If the standard reference genome has been used in Phase a, the variants called with consensus reference can be projected to the standard reference genome with the PanVC normalization tool[14]. The normalization applies the consensus-based variants to the standard reference genome using VCF files and gap position files. The projection performs sequence alignment between the normalized reference and standard reference genome. The insertions, deletions, and mismatches are reported for further analysis.

The PanGenSpark was able to assemble a reference genome from a pan-genome including 500 human haploid genomes in 1468 minutes (Figure 5.15) with 11x speedup (calculated from the runtime measurements while using 50 genomes as a baseline). The experiments have been run on a distributed Spark computing cluster consisting of 448 cores distributed on 26 computing nodes.

The indexing phase was distributed only by chromosomes causing uneven load balancing, and thus, the indexing time is the time taken to index the largest chromosome. The Distributed RLZ compression scales well with the increasing number of genomes and the execution time stays tolerable. However, the DRLZ computation time is relatively large with the small number of genomes decreasing the speedup that can be noticed from the results in Figure 5.15. This side-effect is mostly due to time consuming suffix array creation phase and its relatively large size. To notice, the size of the chromosomal partitions vary with the chromosome size which can cause uneven load balancing but was not affecting in this workflow as each chromosome was compressed in a separate Spark job. The distributed heaviest path calculation and consensus genome extraction scaled nicely with the increasing pan-genome size.

PanGenSpark now enables the assembly of a reusable pan-genomic reference genome in a tolerable time from several hundred human genomes in practice, and the pipeline is scalable to process even larger pan-genomes. The compression and indexing phases can be potentially improved with the more fine-grained parallelization (see Figure 5.8) that was used in Publication III and resulted in a better compression ratio and faster indexing (Figure 5.10).

[12] http://samtools.sourceforge.net/mpileup.shtml
[13] https://gatk.broadinstitute.org/
[14] https://gitlab.com/dvalenzu/PanVC

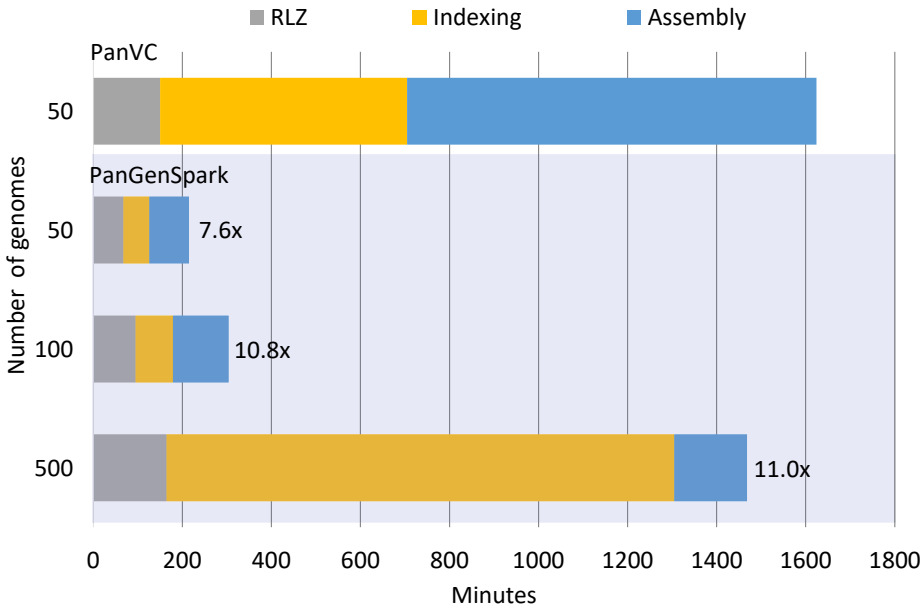

Figure 5.15: PanGenSpark performance compared to single node PanVC execution. Calculated speedup relative to the sequential pipeline is denoted by a number with [217].

Chapter 6

Conclusions

This dissertation presents scalable distributed and data-parallel computing methods for processing and analyzing massive whole-genome sequencing data efficiently. Whole-genome sequencing and high-performance cloud computing costs have dropped to a level that makes it viable to move sequence analytics from research and development purposes to clinical practice and public healthcare. However, the computing time used to refine data from multiple sequencing samples to actionable analytical results is critical and can currently be too long to apply the methods in clinical use. The ultimate goal is to maximize computational efficiency and minimize the computational analysis time. In complex distributed analysis pipelines this is achieved by minimizing the communication overhead, I/O operations, search space, and memory footprint, and by maximizing CPU utilization. The key elements that contribute to the optimal performance of distributed pipelines are noted as appropriate problem decomposition, reusable and minimal data structures, fast data access and transformations, and load balancing. Compression and indexing techniques contribute greatly to minimizing the sequence search space, communication overhead, and memory footprint.

Apache Spark with distributed in-memory data structures was found to be highly suitable for large-scale distributed cluster computing and efficient in reusing and transforming large datasets from one pipeline stage to another in parallel. Hadoop Distributed File System (HDFS) provided fast data access through data locality and contributed to minimizing the communication and I/O operation overhead. Together these technologies offer an efficient and flexible toolset for manipulating big data in the diverse problem space of computational genomics.

Data-parallel methods have been used to transform existing bioinformatics algorithms and tools to operate in distributed analysis pipelines. In

Publication I, an Apache Spark-driven distributed metagenomic analysis pipeline was developed to speed up virome mining from Next-generation sequencing (NGS) data sequenced from several human DNA samples. The pipeline comprises distributed NGS read alignment, alignment filtering and normalization, *de novo* assembly, BLAST, and HMMER sequence database searching, and taxonomic sequence profiling. Different data decomposition patterns have been used in each pipeline phase to achieve data-parallelism on HDFS that enables the running of tasks of each phase efficiently in parallel on a Spark cluster. Data decomposition contributes greatly to load balancing and CPU utilization through granularity, and decomposition depends greatly on the algorithmic complexity: computationally heavy algorithms require fine-grained decomposition for efficient parallelization. Moreover, the workload of smaller tasks can be distributed more evenly.

Scalable data-parallel methods for imputing population genotypes from massive distributed genome-wide Single Nucleotide Polymorphism (SNP) reference panels were implemented in Publication II. Imputation-assisted genotyping increases the coverage of genotypes and the SNP resolution which is often required in Genome-wide Association Studies (GWAS). The distributed imputation method developed relies on imputing SNPs from distributed block compressed (BGZF) SNP reference panels that are decomposed to overlapping SNP marker intervals. This method utilizes the Tabix index to allow fast random access to block compressed SNP data by seeking the offset from the data distributed over HDFS. The same distributed imputation method is applicable in many parallel genotype imputation algorithms that allow imputation over SNP marker intervals. Imputation quality control protocol was used to analyze the imputation accuracy and no degradation in the accuracy was obtained due to the distributed method. However, the imputation interval size affects the imputation speed and accuracy: the shorter interval causes more granularity and decreases the imputation time due to higher parallelism but can also reduce accuracy.

Hybrid-indexing and Distributed Relative Lempel-Ziv (DRLZ) compression has been proven efficient and practical for fast sequence alignment with large compressed collections of human genomes in Publication III. A distributed Relative Lempel-Ziv (DRLZ) compression method was designed and implemented to compress multiple genomes into a single reusable hybrid-index. The hybrid-index provides direct access to compressed genomes and mates the space-efficient RLZ compression with popular sequence alignment tools such as Bowtie2, BWA, and BLAST. DRLZ compression with hybrid-indexing is the first method tested so far on multiple whole human genomes and the results are promising in terms of scal-

ability and query times. As bacterial genomes tend to contain less repetition than human genomes, the compression ratio is lower, but improves slightly with taxonomical grouping. Sequence clustering would be a potential method for improving compression ratio with relatively dissimilar genomes such as microbes and should be studied further. The compressed hybrid-index is directly reusable with the Bowtie2 and BWA read aligners, and with the BLAST search tool.

In Publication IV, the DRLZ compression and hybrid-indexing were applied to assembling reference genomes from compressed and indexed pan-genomes. The standard reference genome is argued to represent the variation of individuals or specific populations poorly. A pan-genome can instead represent populations without disabling population- or individual-specific variants. Pan-genomic references can potentially be used for variant calling purposes when the standard reference genome does not provide enough resolution to discover specific individual variants, and for discovering variants between the population-based pan-genomic references. This work extends the PanVC [49] pipeline with a distributed indexing and reference assembly phase to scale pan-genomic variant calling to multiple human genomes. The compression ratio directly affects the indexing time as the index is constructed from the compressed sequence (i.e., kernel sequence). This phase can be further improved with the highly distributed RLZ compression method presented in Publication III, which allows the use of longer compression dictionaries with RLZ.

The characteristics of genomic data structures allow the decomposition of computational problems into coarse-grained tasks, enabling tasks to be executed in parallel without rewriting the inner loops of the complex algorithms. Decomposing data in to distributable partitions, e.g., by species, samples, chromosomes, gene locus, or SNP markers, provides various alternatives for designing efficient data flows and transformations through bioinformatics analysis pipelines. Moreover, compressed data structures and file formats should be chosen or redeveloped so that the data can be stored and indexed in distributed file systems, and thus retrieved and processed efficiently in parallel on high-performance computing clusters and cloud computing platforms. Thus, routine upstream bioinformatics tasks such as sequence alignment and searching, *de novo* assembly, alignment sorting, filtration, variant calling, and genotype imputation can be executed efficiently in parallel on different partitions of the underlying distributed data structures. Distributed and parallel algorithms with appropriate task and data decomposition patterns allow the implementation of efficient high-throughput data flows in distributed data processing pipelines for future

needs. The focus of this dissertation has been on scalable distributed and parallel computing methods on CPUs. In addition, GPGPU [54, 75, 218] and FPGA [55, 76, 219] computing offer complementary approaches for increasing the performance with fine-grained data parallelism nested with coarse-grained data-parallel computing on CPUs in complex bioinformatics pipelines and algorithms, which is a promising area of our future research.

References

[1] J. Watson and F. Crick. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.

[2] E. Lander, L. Linton. B. Birren, et al. Initial sequencing and analysis of the human genome. *Nature*, 409:860–921, 2001.

[3] S. Goodwin, J. McPherson. and W. McCombie. Coming of age: ten years of next-generation sequencing technologies. *Nature reviews Genetics*, 17:333–351, 2016.

[4] J. K. Kulski. Next-generation sequencing: an overview of the history, tools, and omic applications. In *Next generation sequencing advances, applications and challenges*, pages 3–60, Intech, 2016.

[5] S. Yohe and B. Bharat Thyagarajan. Review of clinical next-generation sequencing. *Archives of Pathology and Laboratory Medicine*, 141:1544–1557, November 2017.

[6] E. Giannopoulou, T. Katsila, C. Mitropoulou, E. E. Tsermpini, and G. P. Patrinos. Integrating next-generation sequencing in the clinical pharmacogenomics workflow. *Frontiers in pharmacology*, 10:384, 2019.

[7] W. Gu, S. Miller, and C. Y. Chiu. Clinical metagenomic next-generation sequencing for pathogen detection. *Annual review of pathology*, 14:319–338, 2019.

[8] P. Suwinski, C. Ong, M. Ling, Y. M. Poh, A. M. Khan, and H. S. Ong. Advancing personalized medicine through the application of whole exome sequencing and big data analytics. *Frontiers in genetics*, 10:49, 2019.

[9] V. Tam, N. Patel, M. Turcotte, et al. Benefits and limitations of genome-wide association studies. *Nature reviews Genetics*, 20:467–484, 2019.

[10] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big data: Astronomical or genomical?. *PLoS biology*, 13:7, 2015.

[11] M. C. Schatz, B. Langmead, and S. Salzberg. Cloud computing and the dna data race. *Nature Biotechnology*, 28(7):691–693, 2010.

[12] M. B. Scholz, C. C. Lo, and P. S. Chain. Next generation sequencing and bioinformatic bottlenecks: the current state of metagenomic data analysis. *Current Opinions in Biotechnology*, 23:9–15, Feb 2012.

[13] E. B. Hodcroft, N. De Maio, R. Lanfear, D. R. MacCannell, B. Q. Minh, H. A. Schmidt. A. Stamatakis, N. Goldman, and D. C., Want to track pandemic variants faster? fix the bioinformatics bottleneck. *Nature*, 591:30–33, 2021.

[14] J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[15] M. Niemenmaa. Analysing sequencing data in hadoop: The road to interactivity via sql. Master's Thesis, Aalto University, 2013.

[16] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing. Cambridge University Press, 2015.

[17] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[18] T. A. Brown. Genomes. Wiley-Liss, 2nd ed., 2002.

[19] L. E. Rosenberg and D. D. Rosenberg. Population and evolutionary genetics. In *Human Genes and Genomes*, pages 317–338, Academic Press, 2012.

[20] Y. Guo, Y. Dai, H. Yu, S. Zhao, D. C. Samuels, and Y. Shyr. Improvements and impacts of grch38 human reference on high throughput sequencing data analysis. *Genomics*, 109(2):83–90, 2017.

[21] F. Sanger, S. Nicklen, and A. R. Coulson. Dna sequencing with chain-terminating inhibitors. In *Proceedings of the National Academy of Sciences of the United States of America*, 74(12):5463–5467, 1977.

[22] T. J. Kasper, M. Melera, P. Gozel, and R. G. Brownlee. Separation and detection of dna by capillary Electrophoresis. *Journal of Chromatography*, 458:303–312, 1988.

[23] B. L. Karger and A. Guttman. Dna sequencing by ce. *Electrophoresis*, 30:S196–S202, 2009.

[24] Illumina Inc. An introduction to next generation sequencing technology. Illumina Inc., 2016.

[25] C. Kuo-Ping. Next-generation Sequencing and Sequence Data Analysis. Bentham Science Publishers, 2016.

[26] J. Quick, N. Grubaugh, S. Pullan, et al. Multiplex pcr method for minion and illumina sequencing of zika and other virus genomes directly from clinical samples. *Nature Protocols*, 12:1261–1276, 2017.

[27] A. D. Tyler, L. Mataseje, C. J. Urfano, et al. Evaluation of oxford nanopore's minion sequencing device for microbial whole genome sequencing applications. *Nature Science Reports*, 8:10931, 2018.

[28] S. L. Amarasinghe, S. Su, X. Dong, et al. Opportunities and challenges in long-read sequencing data analysis. *Genome Biololgy*, 21:30, 2020.

[29] S. Thankaswamy-Kosalai, P. Sen, and I. Nookaew. Evaluation and assessment of read-mapping by multiple next-generation sequencing aligners based on genome-wide characteristics. *Genomics*, 109(3-4):186–191, 2017.

[30] H. Gamaarachchi, S. Parameswaran, and M. A. Smith. Featherweight long read alignment using partitioned reference indexes. *Nature Science Reports*, 9:4318, 2019.

[31] D. Jeffrey and G. Sanjay. Mapreduce: simplified data processing on large clusters. *Communications ACM*, 51:107–113, 2008.

[32] M. Zaharia, M. Chowdhury, T. Das, et al. 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*, pages 2–2, Berkeley, 2012.

[33] M. Niemenmaa, A. Kallio, A. Schumacher, P. Klemelä. E. Korpelainen, and K. Heljanko, Hadoop-bam: directly manipulating next generation sequencing data in the cloud, *Bioinformatics*, 28:6, 2012.

[34] H. Li, et al. The sequence alignment/map format and samtools, *Bioinformatics*, 25(16):2078–2079, 2009.

[35] P. Danecek, et al. The variant call format and vcftools, *Bioinformatics*, 27(15):2156-2158, 2011.

[36] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, pages 1–10, 2010.

[37] H. Li and R. Durbin. Fast and accurate short read alignment with burrows-wheeler transform, *Bioinformatics*, 25(14):1754–1760, 2009.

[38] B. Langmead, et al. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

[39] S. F. Altschul, et al. Basic local alignment search tool. Journal of Molecular Biology, 215(3):403–410, 1990.

[40] S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In *String Processing and Information Retrieval*. Lecture Notes in Computer Science, vol 6393. Springer, 2010.

[41] C. Hoobin, S. J. Puglisi, and J. Zobel. Relative lempel-ziv factorization for efficient storage and retrieval of web collections. *CoRR*, 2011.

[42] H. Ferrada, T. Gagie, T. Hirvola, and S. J. Puglisi. Hybrid indexes for repetitive datasets. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 372:20130137, 2014.

[43] K. J. Locey and J. T. Lennon. Scaling laws predict global microbial diversity. *PNAS*, 113:5970–5975, 2016.

[44] L. S. Arroyo Muhr, et al. Viruses in case series of tumors: consistent presence in different cancers in the same subject. *Plos One*, 12, 2017.

[45] D. Bzhalava, et al. Unbiased approach for virus detection in skin lesions. *Plos One*, 8, 2013.

[46] D. Bzhalava, et al. Deep sequencing extends the diversity of human papillomaviruses in human skin. *Scientific Reports*, 4:5807, 2014.

[47] V. Smelov, et al. Detection of dna viruses in prostate cancer. *Scientific Reports*, 6:25235, 2016.

[48] D. Valenzuela. Chico: A compressed hybrid index for repetitive collections. In *Proceedings of the 15th International Symposium on Experimental Algorithms*, 9685:326–338, 2016.

[49] D. Valenzuela, T. Norri, N. Välimäki, E. Pitkänen, and V. Mäkinen. Towards pan-genome read alignment to improve variation calling. *BMC Genomics*, 19:87, 2018.

[50] D. C. Marinescu. Cloud Computing : Theory and Practice. Morgan Kaufmann Publishers Inc, 1st ed., 2013.

[51] K. Erciyes. Parallel and distributed computing. In *Distributed and Sequential Algorithms for Bioinformatics*, pages 51–77, Springer, 2015.

[52] E. Aubanel. Elements of Parallel Computing. Chapman & Hall/CRC, 1st ed., 2016.

[53] J. L. Hennessy and D. A. Patterson. Computer Architecture, Fifth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc, 5th ed., 2011.

[54] C. Navarro, N. Hitschfeld-Kahler, and L. Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329, 2014.

[55] J. M. P. Cardoso, J. G. F. Coutinho, and P. C. Diniz. Embedded Computing for High Performance: Efficient Mapping of Computations Using Customization, Code Transformations and Compilation. Elsevier Inc, 2017.

[56] J. Dean and L. A. Barroso. The tail at scale. *Communications ACM*, 56, 2:74–80, 2013.

[57] T. White. Hadoop: The Definitive Guide. O'Reilly Media, 4th ed., 2015.

[58] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10, 2010.

[59] T. M. Forum. Mpi: a message passing interface. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 878–883, Association for Computing Machinery, 1993.

[60] H. Sarbazi-Azad. Advances in GPU Research and Practice. Elsevier Inc, 2016.

[61] H. Sarbazi-Azad. Software-level task scheduling on gpus. In *Advances in GPU Research and Practice*, pages 83–103, Elsevier Inc, 2016.

[62] S. Sarkar, P. Kandelwal, S. Bandyopadhyay, and H. Giese. Analysis of gpgpu programs for data-race and barrier divergence. In *13th International Conference on Software Technologies*, pages 494–505, 2018.

[63] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares. Accelerating sparse matrix-matrix multiplication with gpu tensor cores. *Computers and Electrical Engineering*, 88, Dec 2020.

[64] S. Mittal and S. Vaishay. A survey of techniques for optimizing deep learning on gpus. *Journal of Systems Architecture*, 99:101635, 2019.

[65] A. McKenna et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome Research*, 20, 9, pages 1297–1303, 2010.

[66] J. R. Heldenbrand, S. Baheti, and M. A. Bockol, et al. Recommendations for performance optimizations when using gatk3.8 and gatk4. *BMC Bioinformatics*, 20:557, 2019.

[67] D. Decap, J. Reumers, C. Herzeel, P. Costanza, and J. Fostier. Halvade: scalable sequence analysis with mapreduce, *Bioinformatics*, 31(15):2482–2488, 2015.

[68] D. Decap, J. Reumers, C. Herzeel, P. Costanza, and J. Fostier. Halvade-RNA: Parallel variant calling from transcriptomic data using mapreduce. *Plos One*, 12:3, 2017.

[69] L. Pireddu, S. Leo, and G. Zanetti. Seal: a distributed short read mapping and duplicate removal tool, *Bioinformatics*, 27:2159–2160, Aug 2011.

[70] A. Schumacher, L. Pireddu, M. Niemenmaa, A. Kallio, E. Korpelainen, G. Zanetti, and K. Heljanko. Seqpig: Simple and scalable scripting for large sequencing data sets in hadoop, *Bioinformatics*, 30(1):119–120, 2014.

[71] Z. Al-Ars, S. Wang, and H. Mushtaq. Sparkra: Enabling big data scalability for the gatk rna-seq pipeline with apache spark. *Genes*, 11(1):53, 2020.

[72] P. Di Tommaso, M. Chatzou, E. Floden, and et al.. Nextflow enables reproducible computational workflows. *Nature Biotechnology*, 35:316–319, 2017.

[73] J. Köster and S. Rahmann. Snakemake - a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2017.

[74] E. Afgan. The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. *Nucleic acids research*, 46:W537–W544, 2018.

[75] M. S. Nobile, P. Cazzaniga, A. Tangherloni, and D. Besozzi. Graphics processing units in *Bioinformatics*, computational biology and systems biology. *Briefings in bioinformatics*, 18(5):870–885, 2017.

[76] L. Hasan and Z. Al-Ars. An overview of hardware-based acceleration of biological sequence alignment. In *Computational Biology and Applied Bioinformatics*, InTech, 2011.

[77] K. Zhao, X. ChuK. G-blastn: Accelerating nucleotide alignment by graphics processors, *Bioinformatics*, 30:1384–91, 2014.

[78] T. Ahmad, N. Ahmed, J. Peltenburg, and Z. Al-Ars. Arrowsam: In-memory genomics data processing using apache arrow. In *3rd International Conference on Computer Applications & Information Security*, pages 1–6, 2020.

[79] T. Ahmad, N. Ahmed, Z. Al-Ars, and et al. Optimizing performance of gatk workflows using apache arrow in-memory data framework. *BMC Genomics*, 21:683, 2020.

[80] E. J. Houtgast, V. M. Sima, K. Bertels, and Z. Al-Ars. An efficient gpuaccelerated implementation of genomic short read mapping with bwamem. *ACM SIGARCH Computer Architecture News*, 44(4):38–43, 2016.

[81] N. Ahmed, J. Lévy, S. Ren, H. Mushtaq, K. Bertels, and Z. Al-Ars. Gasal2: A gpu accelerated sequence alignment library for high-throughput ngs data. BMC *Bioinformatics*, 20:520, 2019.

[82] S. Ren, N. Ahmed, K. Bertels, and Z. Al-Ars. Gpu accelerated sequence alignment with traceback for gatk haplotypecaller. *BMC Genomics*, 20:184, 2019.

[83] M. Baker. De novo genome assembly: what every biologist should know. Nat Methods, 9:333–337, 2012.

[84] N. Ahmed, T. D. Qiu, K. Bertels, and Z. Al-Ars. Gpu acceleration of darwin read overlapper for de novo assembly of long dna reads. *BMC Bioinformatics*, 21, 2020.

[85] A. Swiercz, W. Frohmberg, M. Kierzynka, P. Wojciechowski, and P. Zurkowski, et al. Grasshopper - an algorithm for de novo assembly based on gpu alignments. *Plos One*, 13:8, 2018.

[86] S. Goswami, K. Lee, S. Shams, and S. J. Park. Gpu-accelerated large-scale genome assembly. In *Proceedings of the IEEE 32nd International Parallel and Distributed Processing Symposium*, pages 814–824, Institute of Electrical and Electronics Engineers Inc., 2018.

[87] E. J. Houtgast, V. M. Sima, and Z. Al-Ars. High performance streaming smith-waterman implementation with implicit synchronization on intel fpga using opencl. In *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering*, pages 492–496, 2017.

[88] E. J. Houtgast, V. Sima, G. Marchiori, K. Bertels, and Z. Al-Ars. Power-efficient accelerated genomic short read mapping on heterogeneous computing platforms. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 28–28, 2016.

[89] S. Ren, V. M. Sima, and Z. Al-Ars. Fpga acceleration of the pair-hmms forward algorithm for dna sequence analysis. In *2015 IEEE International Conference on Bioinformatics and Biomedicine*, pages 1465–1470, 2015.

[90] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan. Gatekeeper: a new hardware architecture for accelerating pre-alignment in dna short read mapping, *Bioinformatics*, 33(21):3355–3363, 2017.

[91] D. E. Knuth, J. H., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[92] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications ACM*, 20(10):761–772, 1977.

[93] K. Fredriksson and S. Grabowski. Practical and optimal string matching. In String Processing and Information Retrieval. Lecture Notes in Computer Science, vol 3772, Springer, 2005.

[94] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[95] A. R. Mushegian. Foundations of Comparative Genomics. Elsevier Science and Technology, 1st ed., 2007.

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

[97] S. B. Needleman and C. D. A. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology, 48:443–453, 1970.

[98] M. A. Larkin, et al. Clustal w and clustal x version 2.0, *Bioinformatics*, 23:2947–2948, 2007.

[99] K. Katoh, K. Misawa, K. Kuma, and T. Miyata. Mafft: a novel method for rapid multiple sequence alignment based on fast fourier transform. Nucleic Acids Res, 30:59–66, 2002.

[100] R. C. Edgar. Muscle: a multiple sequence alignment method with reduced time and space complexity. *BMC Bioinformatics*, 5:113, 2004.

[101] R. Pereira, J. Oliveira, and M. Sousa. Bioinformatics and computational tools for next-generation sequencing analysis in clinical genetics. *Journal of Clinical Medicine*, 9:132, 2020.

[102] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, CA, 1994.

[103] R. Li, et al. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[104] K. H. Y. Wong, M. Levy-Sakin, and P. Kwok. De novo human genome assemblies reveal spectrum of alternative haplotypes in diverse populations. *Nature Communications*, 9:3040, 2018.

[105] X. Liao, M. Li, and Y. Zou. Current challenges and solutions of de novo assembly. *Quantitative Biology*, 7:90–109, Springer, 2019.

[106] J. T. Simpson, K. Wong, S. D.Jackman, et al. Abyss: A parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009.

[107] Y. Li, Y. Hu, L. Bolund, and J. Wang. State of the art de novo assembly of human genomes from massively parallel sequencing data. Human genomics, 4(4):271–277, 2010.

[108] H. E. L. Lischer and K. K. Shimizu. Reference-guided de novo assembly approach improves genome reconstruction for related species. *BMC Bioinformatics*, 18:474, 2017.

[109] K. Schneeberger, et al. Reference-guided assembly of four diverse arabidopsis thaliana genomes. In *Proceedings of the National Academy of Sciences*, 108:10249, 2011.

[110] R. W. Ye, T. Wang, L. Bedzyk, and K. M. Croker. Applications of dna microarrays in microbial system. *Journal of Microbiological Methods*, 47(3):257–272, 2001.

[111] G. Redei, Microarray hybridization. In *Encyclopedia of Genetics, Genomics, Proteomics and Informatics*, Springer, 2016.

[112] M. De Donato, S. O. Peters, S. E. Mitchell, T. Hussain, and I. G. Imumorin. Genotyping-by-sequencing (gbs): A novel, efficient and cost-effective genotyping method for cattle using next-generation sequencing. *Plos One*, 8:5, 2013.

[113] J. Marchini and B. Howie. Genotype imputation for genome-wide association studies. *Nature reviews Genetics*, 11:499–511, 2010.

[114] K. Ardlie, L. Kruglyak, and M. Seielstad. Patterns of linkage disequilibrium in the human genome. *Nature reviews Genetics*, 3:299–310, 2002.

[115] B. L. Browning and S. L. Browning. Genotype imputation with millions of reference samples. *American Journal of Human Genetics*, 98:116–126, 2016.

[116] R. Nielsen, J. S. Paul, A. Albrechtsen, and Y. S. Song. Genotype and snp calling from next-generation sequencing data. *Nature reviews Genetics*, 12(6):443–451, 2011.

[117] P. Danecek, et al. Twelve years of samtools and bcftools. *GigaScience*, 10, 2012.

[118] J. L. Freeman, G. H. Perry, L. Feuk, R. Redon, et al. Copy number variation: new insights in genome diversity. *Genome Research*, 16:949–961, 2006.

[119] A. J. Sharp, D. P. Locke, and S. D. McGrath, et al. Segmental duplications and copy-number variation in the human genome. *American Journal of Human Genetics*, 77:78–88, 2005.

[120] D. Mandelker, R. Schmidt, A. Ankala, et al. Navigating highly homologous genes in a molecular diagnostic setting: a resource for clinical next-generation sequencing. *Genetics in Medicine*, 18:1282–1289, 2016.

[121] C. Lee, H. Yen, A. W. Zhong, et al. Resolving misalignment interference for ngs-based clinical diagnostics. *Human Genetics*, 140:477–492, Springer 2021.

[122] M. Pirooznia, F. S. Goes, and P. P. Zandi. Whole-genome cnv analysis: advances in computational approaches. *Frontiers in genetics*, 6:138, 2015.

[123] S. Kosugi, Y. Momozawa, X. Liu, C. Terao, M. Kubo, and Y. Kamatani. Comprehensive evaluation of structural variation detection algorithms for whole genome sequencing. *Genome biology*, 20, 1:117, 2019.

[124] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[125] A. Langiu. On parsing optimality for dictionary-based text compression - the zip case. *Journal of Discrete Algorithms*, 20:65–70, 2013.

[126] K. Kryukov, M. T. Ueda, S. Nakagawa, and T. Imanishi. Sequence compression benchmark (scb) — database a comprehensive evaluation of reference-free compressors for fasta-formatted sequences. *GigaScience*, 9, 2020.

[127] S. Arroyuelo, G. Navarro, and K. Sadakane. Stronger lempel-ziv based compressed text indexing. *Algorithmica*, 62:54–101, 2012.

[128] P. Bille, R. Fagerberg, and I. L. Gørtz. Improved approximate string matching and regular expression matching on ziv-lempel compressed texts. *ACM Transactions on Algorithms*, 6, 2009.

[129] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings. In *Proceedings of the 22nd Symposium on Discrete Algorithms*, pages 373–389, 2011.

[130] S. Deorowicz and S. Grabowski. Robust relative compression of genomes with random access, *Bioinformatics*, 27:2979–2986, 2011.

[131] H. Do, J. Jansson, K. Sadakane, and W.-K. Sung. Fast relative lempelziv self-index for similar sequences. *Theoretical Computer Science*, 532:14–30, 2014.

[132] T. Gagie, P. Gawrychowski, and S. J. Puglisi. Faster approximate pattern matching in compressed repetitive texts. In *Proceedings of the 22nd International Conference on Algorithms and Computation*, pages 653-662. Springer-Verlag, 2011.

[133] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *1st annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, 1990.

[134] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.

[135] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53:6, November 2006.

[136] S. Wandelt and U. Leser. Adaptive efficient compression of genomes. *Algorithms for Molecular Biology*, 7:30, 2012.

[137] S. Wandelt and U. Leser. String searching in referentially compressed genomes. In *Proceedings of the International Conference on Knowledge Discovery and Information Retrieval*, pages 95–102. SciTePress, 2012.

[138] S. Wandelt, J. Starlinger, M. Bux, and U. Leser. Rcsi: scalable similarity search in thousand(s) of genomes. In *Proceedings of the VLDB Endowment*, 6:1534–1545, 2013.

[139] E. Ohlebusch, J. Fischer, and S. Gog. Cst++. In *Proceedings of the String Processing and Information Retrieval Symposium*. Lecture Notes in Computer Science, vol 6393. Springer 2010.

[140] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lempel-ziv parsing in external memory. In *Data Compression Conference*, pages 153–162, 2014.

[141] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time lempel-ziv factorization: Simple, fast, small. In *Combinatorial Pattern Matching*. Lecture Notes in Computer Science, vol 7922. Springer, 2013.

[142] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39:1, 2007.

[143] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly fm-index. In *String Processing and Information Retrieval*, Lecture Notes in Computer Science, vol 3246. Springer, 2004.

[144] J. Kärkkäinen and E. Ukkonen. Lempel-ziv parsing and sublinear-size index structures for string matching. In *Proceedings of the 3rd South American Workshop on String Processing*, pages 141–155, Carleton University Press, 1996.

[145] T. Gagie and S. Puglisi. Searching and indexing genomic databases via kernelization. *Frontiers in Bioengineering and Biotechnology*, 3:12, 2015.

[146] D. Valenzuela and V. Mäkinen. Chic: a short read aligner for pan-genomic references. *bioRxiv*, 178129, 2017.

[147] D. Sexton. Computational infrastructure and basic data analysis for next-generation sequencing. In *Tag-Based Next Generation Sequencing*, John Wiley & Sons, 2011.

[148] J. A. Garrido-Cardenas, F. Garcia-Maroto, J. A. Alvarez-Bermejo, and F. Manzano-Agugliaro. Dna sequencing sensors: An overview. *Sensors*, 17(3):588, 2017.

[149] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic acids research*, 38(6):1767–1771, 2010.

[150] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. In *Proceedings of the National Academy of Sciences of the United States of America*, 85(8):2444–2448, 1988.

[151] S. A. Leonard. Iupac/iub single letter codes within nucleic acid and amino acid sequences. Current Protocols in *Bioinformatics*, 99:A.1A.1–A.1A.1, 2003.

[152] M. Stratton. Genome resequencing and genetic variation. *Nature Biotechnology*, 26:65–66, 2008.

[153] A. Conesa, P. Madrigal, S. Tarazona, D. Gomez-Cabrero, A. Cervera, A. McPherson, M. W. Szczeŝniak, D. J. Gaffney, L. L. Elo, X. Zhang, and A. Mortazavi. A survey of best practices for rna-seq data analysis. *Genome biology*, 17:13, 2016.

[154] S. Pfeife. From next-generation resequencing reads to a high-quality variant data set. Heredity, 118:111–124, 2017.

[155] D. Kim, G. Pertea, C. Trapnel, et al. Tophat2: accurate alignment of transcriptomes in the presence of insertions, deletions and gene fusions. *Genome Biololgy*, 14:R36, 2013.

[156] M. C. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[157] J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo. Sparkbwa: Speeding up the alignment of high-throughput dna sequencing data. *Plos One*, 11:5, 2016.

[158] J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo. Bigbwa: approaching the burrows-wheeler aligner to big data technologies, *Bioinformatics*, 31:4003–4005, 2015.

[159] T. Thomas et al. Metagenomics - a guide from sampling to data analysis. *Microbial Informatics and Experimentation*, 2:3, 2012.

[160] R. M. Sherman, J. Forman, et al. Antonescu. Assembly of a pan-genome from deep sequencing of 910 humans of african descent. *Nature Genetics*, 51:30–35, 2019.

[161] S. Mallick, H. Li, and M. a. Lipson. The simons genome diversity project: 300 genomes from 142 diverse populations. *Nature*, 538:201–206, 2016.

[162] P. Menzel, K. Ng., and A. Krogh. Fast and sensitive taxonomic classification for metagenomics with Kaiju. *Nature Communications*, 7, 11257, 2016.

[163] D. Kim, L. Song, F.P. Breitwieser, S.L Salzberg. Centrifuge: rapid and sensitive classification of metagenomic sequences. *Genome Research*, 12(26):1721-1729, 2016.

[164] D.E. Wood, J. Lu, and B. Langmead. Improved metagenomic analysis with Kraken 2. *Genome biology*, 20:257, 2019.

[165] B. Vilne, I. Meistere, L. Grantina-Ievina, J. Kibilds. Machine Learning Approaches for Epidemiological Investigations of Food-Borne Disease Outbreaks. *Frontiers in Microbiology*, 10:1722, 2019.

[166] A. Fiannaca, L. La Paglia, M. La Rosa, et al. Deep learning models for bacteria taxonomic classification of metagenomic data. *BMC Bioinformatics* 19:198, 2018.

[167] C. S. Kim, M. D. Winn, V. Sachdeva, and K. E. Jordan. K-mer clustering algorithm using a mapreduce framework: application to the parallelization of the inchworm module of trinity. *BMC Bioinformatics*, 18:467, 2017.

[168] C. U. Köser, M. J. Ellington, E. J. P. Cartwright, et al. Routine use of microbial whole genome sequencing in diagnostic and public health microbiology. *Plos Pathogens*, 8:8, 2012.

[169] S. A. Clark, R. Doyle, J. Lucidarme, R. Borrow, and J. Breuer. Targeted dna enrichment and whole genome sequencing of neisseria meningitidis directly from clinical specimens. *International Journal of Medical Microbiology*, 308:256–262, 2018.

[170] J. Cheval, et al. Evaluation of high-throughput sequencing for identifying known and unknown viruses in biological samples. *Journal of clinical microbiology*, 49(9):3268–3275, 2011.

[171] J. Yuan, W. Li, E. Qiu, S. Han, and Z. Li. Metagenomic ngs optimizes the use of antibiotics in appendicitis patients: bacterial culture is not suitable as the only guidance. *American journal of Translational research*, 13(4):3010–3021, 2021.

[172] P. Chouvarine, L. Wiehlmann, M. Losada, D. P. DeLuca, and B. Tümmler. Filtration and normalization of sequencing read data in whole-metagenome shotgun samples. *Plos One*, 11:10, 2016.

[173] D. A. Durai and M. H. Schulz. Improving in-silico normalization using read weights. *Scientific Reports*, 9:5133, 2019.

[174] C. A. Steward, A. Parker, B. A. Minassian, S. M. Sisodiya, A. Frankish, and J. Harrow. Genome annotation for clinical genomic diagnostics: strengths and weaknesses. *Genome medicine*, 9(1):49, 2017.

[175] N. Dhanker. Parallel implementation & performance evaluation of blast algorithm on linux cluster. *International Journal of Computer Science and Information Technologies*, 5(3):4818–4820, 2014.

[176] M. Mikailov, F. Luo, S. Barkley, et al. Scaling bioinformatics applications on hpc. *BMC Bioinformatics*, 18:501, 2017.

[177] J. D. Grant, R. L. Dunbrack, F. J. Manion, and M. F. Ochs. Beoblast: distributed blast and psi-blast on a beowulf cluster, *Bioinformatics*, 18:765–766, 2002.

[178] A. Y. Zomaya. Parallel Computing for Bioinformatics and Computational Biology: Models, Enabling Technologies, and Case Studies. John Wiley & Sons, 2006.

[179] K. Chen and L. Pachter. Bioinformatics for whole-genome shotgun sequencing of microbial communities. *Plos computational biology*, 1(2):106–112, 2005.

[180] K. D. Pruitt, T. Tatusova, and D. R. Maglott. Ncbi reference sequences (refseq): a curated non-redundant sequence database of genomes, transcripts and proteins. *Nucleic acids research*, 35:61–65, 2007.

[181] Z. Zhou, et al. The enterobase user's guide, with case studies on salmonella transmissions, yersinia pestis phylogeny, and escherichia core genomic diversity. *Genome Research*, 30:138–152, 2020.

[182] B. E. Pickett, et al. Vipr: an open bioinformatics database and analysis resource for virology research. *Nucleic acids research*, 40:593–598, 2012.

[183] M. V. Larsen, S. Cosentino, and S. Rasmussen, et al. Multilocus sequence typing of total-genome-sequenced bacteria. *Journal of clinical microbiology*, 50(4):1355–1361, 2012.

[184] M. Stephens and P. Scheet. Accounting for decay of linkage disequilibrium in haplotype inference and missing-data imputation. *American Journal of Human Genetics*, 76(3):449–462, 2005.

[185] B. N. Howie, P. Donnelly, and J. Marchini. A flexible and accurate genotype imputation method for the next generation of genome-wide association studies. *Plos Genetics*, 5, 6, 2009.

[186] E. C. Carbo et al. Coronavirus discovery by metagenomic sequencing: a tool for pandemic preparedness. *Journal of clinical virology*, 131, 10459:4, 2020.

[187] W. C. Koff, et al. Accelerating next-generation vaccine development for global disease prevention. *Science*, 340, 6136, 2013.

[188] R. J. Noad, et al. Uk vaccines network: Mapping priority pathogens of epidemic potential and vaccine pipeline developments. *Vaccine*, 37, 43:6241–6247, 2019.

[189] J. Hadfield, C. Megill, S. B. Bell, J. Huddleston, B. Potter, C. Callender, P. Sagulenko, T. Bedford, and R. A. Neher. Nextstrain: real-time tracking of pathogen evolution, *Bioinformatics*, 34:4121–4123, 2018.

[190] A. Armando, J. W. Simon, and A. Danny. Rapid outbreak sequencing of ebola virus in sierra leone identifies transmission chains linked to sporadic cases. *Virus Evolution*, 1, 2:vew016, 2016.

[191] Y. H. Grad, M. Lipsitch, M. Feldgarden, et al. Genomic epidemiology of the escherichia coli o104:h4 outbreaks in europe. *PNAS*, 109, 8:3065–3070, 2012.

[192] J. A. Gilbert, M. J. Blaser, J. G. Caporaso, J. K. Jansson, S. V. Lynch, and R. Knight. Current understanding of the human microbiome. *Nature Medicine*, 24, 4:392–400, 2018.

[193] K. M. Wylie et al. Emerging view of the human virome. *Translational research*, 160:283–290, 2012.

[194] K. M. Wylie, et al. Sequence analysis of the human virome in febrile and afebrile children. *Plos One*, 7, 2012.

[195] P. Hugenholtz, M. Chuvochina, A. Oren, et al. Prokaryotic taxonomy and nomenclature in the age of big sequence data. *ISME Journal*, 2021.

[196] W. Gu, X. Deng, M. Lee, et al. Rapid pathogen detection by metagenomic next-generation sequencing of infected body fluids. *Nature Medicine*, 27:115–124, 2021.

[197] A. Maarala, Z. Bzhalava, J. Dillner, K. Heljanko, and D. Bzhalava. Virapipe: scalable parallel pipeline for viral metagenome analysis from next generation sequencing reads, *Bioinformatics*, 34(6):928–935, 2018.

[198] D. Li et al. Megahit v1.0: a fast and scalable metagenome assembler driven by advanced methodologies and community practices. *Methods*, 102:3–11, 2016.

[199] J. Huang and B. Howie. Improved imputation of low-frequency and rare variants using the uk10k haplotype reference panel. *Nature Communications*, 6:8111, 2015.

[200] A. I. Maarala, K. Pärn, J. Nuñez Fontarnau, and K. Heljanko. Sparkbeagle: Scalable genotype imputation from distributed whole-genome reference panels in the cloud. In *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, 97:1-8, 2020.

[201] B. L. Browning, Y. Zhou, and S. R. Browning. A one-penny imputed genome from next-generation reference panels. *American Journal of Human Genetics*, 103(3):338–348, 2018.

[202] Computational Pan-Genomics Consortium. Computational pangenomics: status, promises and challenges. *Briefings in bioinformatics*, 1(19):118-135, 2016.

[203] R. M. Sherman and S. L. Salzberg. Pan-genomics in the human genome era. *Nature reviews Genetics*, 21:243–254, 2020.

[204] A. Dilthey, C. Cox, Z. Iqbal, M. Nelson, and G. McVean. Improved genome inference in the mhc using a population reference graph. *Nature Genetics*, 47:682–688, 2015.

[205] L. Rouli, V. Merhej, P. E. Fournier, and D. Raoult. The bacterial pangenome as a new tool for analysing pathogenic bacteria. *New Microbes New Infects*, 7:72–85, 2015.

[206] E. Trost, J. Blom, S. C. Soares, et al. Pangenomic study of corynebacterium diphtheriae that provides insights into the genomic diversity of pathogenic isolates from cases of classical diphtheria, endocarditis, and pneumonia. *Journal of Bacteriology*, 194:3199–3215, 2012.

[207] Q. Zhao, Q. Feng, H. Lu, et al. Pan-genome analysis highlights the extent of genomic variation in cultivated and wild rice. *Nature Genetics*, 50:278–284, 2018.

[208] B. Zhang, W. Zhu, and S. Diao, et al. The poplar pangenome provides insights into the evolutionary history of the genus. *Communications Biology*, 2:215, 2019.

[209] T. Hervé, et al, Implications for the microbial "pan-genome". In *Proceedings of the National Academy of Sciences*, 102:13950–13955, 2005.

[210] L.C. Guimarães, et al. Inside the pan-genome - methods and software overview. *Current genomics*, 16(4):245–252, 2015.

[211] J. Sirén, N. Välimäki, and V. Mäkinen. Indexing graphs for path queries with applications in *Genome Research*. In *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):375–388, 2014.

[212] A. Maarala, O. Arasalo, D. Valenzuela, V. Mäkinen, and K. Heljanko, Distributed hybrid-indexing of compressed pan-genomes for scalable and fast sequence alignment. *Plos One*, 16:8, 2021.

[213] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lazy lempel-ziv factorization algorithms. *ACM Journal of Experimental Algorithmics*, 2.4:1-19, 2016.

[214] K. Schneeberger et al. Simultaneous alignment of short reads against multiple genomes. *Genome Biololgy*, 10:R98, 2009.

[215] B. Paten, A. Novak, and D. Haussler. Mapping to a reference genome structure. *arXiv preprint*, arXiv:1404.5010, 2014.

[216] D. Valenzuela. Algorithms and Data Structures for Sequence Analysis in the Pan-Genomic Era. PhD thesis, University of Helsinki, Finland, 2017.

[217] A. Maarala, O. Arasalo, D. Valenzuela, K. Heljanko, and V. Mäkinen. Scalable reference genome assembly from compressed pan-genome index with spark. In *Proceedings of the 9th International Conference on BigData*, pages 68–84, Springer, 2020.

[218] Z. Yin, H. Lan, G. Tan, M. Lu, A. V. Vasilakos, and W. Liu. Computing platforms for big biological data analytics: Perspectives and challenges. *Computational and Structural Biotechnology Journal*, 15:403–411, 2017.

[219] T. Robinson, J. Harkin, and P. Shukla. Hardware acceleration of genomics data analysis: Challenges and opportunities, *Bioinformatics*, 13(37):1785-1795, 2021.