Czech Technical University
Faculty of Electrical Engineering

Distributed cloud-based approaches to the genomic data analysis

(Master's thesis)

Bc. Filip Mihalovič

Supervisor: doc. Ing. Jiří Kléma, PhD.

Study programme: Open Informatics
Specialization: Software Engineering

May 2016

Acknowledgements

I wish to express my sincere thanks to my supervisor doc. Ing. Jiří Kléma, PhD. for sharing his expertise and for his continuous guidance. I am grateful to my family and friends for their encouragement and support during my studies. Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Filip Mihalovič**

Study programme: Open Informatics
Specialisation: Software Engineering

Title of Diploma Thesis: **Distributed cloud-based approaches to the genomic data analysis**

Guidelines:

1. Learn about the origin and role of NGS (next-generation sequencing) data.
2. Get familiar with the most popular cluster computing platforms such as Apache Spark or Hadoop.
3. Review the existing bioinformatics tools for NGS cloud-based processing that are based on the general platforms.
4. Propose a cloud-based distributed algorithms for NGS-based sequence classification and assembly (data will be provided by the supervisor).
5. Develop, test and deploy a prototype of a NGS distributed workflow that implements the algorithms ad 4.
6. Provide performance tests that compare the proposed algorithms with their serial counterparts.

Bibliography/Sources:

1. Wiewiórka, M.S. et al.: SparkSeq: fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision. Bioinformatics, Sep 15;30(18):2652-3, 2014.
2. Massie, M. et al: ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing. University of california at Berkeley, Tech. Rep. No. UCB/EECS-2013-207, Dec 2013.
3. Metzker, M. L.: Sequencing technologies &#8212; the next generation. Nature Reviews Genetics 11, 31-46, Jan 2010.
4. Kwon, T. et al: Next-generation sequencing data analysis on cloud computing. Genes & Genomic, Vol. 37, Issue 6, pp 489-501, June 2015.
5. Pop, M., Salzberg, S. L.: Bioinformatics challenges of new sequencing technology. Trends Genet. 24, 142&#8211;149 2008.

Diploma Thesis Supervisor: doc. Jiří Kléma Ing., Ph.D.

Valid until the end of the summer semester of academic year 2016/2017

[illegible]

prof. Ing. Filip Železný, Ph.D.
Head of Department

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
L.S.
-4-

[illegible]

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 18, 2016

Declaration

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

In Prague on 24th May 2016

[illegible]
…………………………………
Author

Abstract

The advance of genome analysis bound to next-generation sequencing has allowed scientists to conduct research to deeper understand the biological structure of organisms. A problem of computationally demanding genome assembly based on a high volume of sequence reads is introduced. Several sequential solutions for de novo genome assembly are reviewed. Two fundamental types of genome assembly approaches exist, the sequence reconstruction via de Bruijn graph and the overlap graph method. We focus on parallelization of the genome assembly task using the overlap graph approach and the utilization of Apache Spark big data engine. We demonstrate that subtasks of genome assembly can be parallelized and computed in a distributed manner. We present the results of parallelization on a proof of concept implementation by executing performance and functional tests. The test results indicate a sufficient degree of parallelization and a satisfying assembly quality when compared to the referential sequential assembler.

Abstrakt

Výzkum v oblasti analýzy genomu spojený se sekvenováním nové generace poskytl vědcům možnost provádět experimenty pro lepší porozumění biologické struktury organismů. Nadefinujeme problém výpočetně náročného sestavení genomů na základě velkého množství přečtených vzorků sekvencí. Následně prozkoumáme několik sekvenčních algoritmů pro de novo sestavování genomů. Dva fundamentální přístupy k sestavení genomů jsou známé, rekonstrukce sekvencí na základě de Bruijn grafů a na základě grafů překrytí. Zaměříme se na paralelizaci sestavování genomů pomocí grafů překrytí s využitím systému pro zpracování velkých dat Apache Spark. Demonstrujeme paralelizaci dílčích úkolů sestavování genů a jejich zpracování distribuovaným systémem. Výsledky paralelizace ověřujeme na vyvinutém konceptu provedením testů zaměřených na výkon a správnou funkcionalitu. Dosažené výsledky testů indikují dostatečnou úroveň paralelizace a uspokojivou kvalitu sestavení ve srovnání s referenčním řešením.

Table of Contents

List of Figures

List of Abbreviations

DNA	Deoxyribonucleic acid
NGS	Next-Generation Sequencing
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
RDD	Resilient Distributed Dataset
DBG	De Bruijn Graph
OLC	Overlap Layout Consensus
SAGE	String-overlap Assembly of Genomes

1. Introduction

Next-generation sequencing has been changing how scientists look at genome analysis for the past ten years. It has brought very bright ideas on genomic data gathering and analysis, providing essential information on the evolution of organisms, the relationships between them, and last but not least, the personalization of medical treatment. In comparison with First-Generation Sequencing, obtaining data from biological material is massively parallel and fast. On the other hand, assembly and analysis of genomes are computationally demanding. Therefore, an efficient software depending on graph algorithms is needed.

Sequential assembly approaches already exist. Despite high complexity of algorithms, their performance is often faster than the worst-case expectations. Existing algorithms are divided into two main groups, according to the genome reconstruction method. The first one depends on the further division of reads into k-mers, the latter reconstructs the genome based on overlaps between reads. Enhancements in their performance are accomplished by applying graph optimization heuristics. A great room for improvements still exists, originating in the parallelization of the genome assembly. Parallelization expands the assembly with high scalability, distribution and potential detachment to cluster computing resources.

In this thesis, I focus on design, development and testing of a parallel assembly algorithm. An assembler capable of real biological application is out of the scope of this thesis. However, I try to present a proof of concept identifying the main assembly subtasks and challenges to be parallelized. Eventually, the design is inspired by a successful sequential algorithm, and I parallelize it using a big data engine. Although I do not outperform the initial algorithm with its full implementation containing numerous heuristic improvements, I show that parallelization has the capacity to speed up the assembly. I base this proof on the fact, that the key assembly steps can be efficiently distributed among available resources, and the measured runtime can be decreased by resource scaling.

The distributed design of genome assembly is turned into a functional implementation. A workflow necessary to execute the solution is built and tested. The usage of the selected platform is evaluated from the practical, performance and quality point of view. A series of tests are executed to provide relevant information for the final comparison of the assembler based on a selected algorithmic approach and the parallelized implementation. The strengths and the weaknesses of both solutions are emphasized. In the end, thoughts on the further applications of big data engines in genomic data analysis are provided.

2. Next-Generation Sequencing

Deoxyribonucleic acid (DNA) is a molecule that was first isolated by Friedrich Miescher in 1869 [1]. But it was not until 1953 that James Watson and Francis Crick discovered its structure with two helical chains [2], each curled round the same axis, currently known as double-helix structure. DNA is what makes each organism unique in the entire world by carrying its genetic information for proper functionality, growth, and reproduction. This explains why DNA molecule has been the subject of experiments and studies that provide and definitely will continue to provide essential information about the world around us. While ingredients of DNA are already known, its uniqueness responsible for differences is still a subject that is being paid much attention. Although before studying DNA, it has to be read on molecular level first.

Initial attempts at reading DNA molecule were executed by Fred Sanger in 1955 [3]. He succeeded in determining the order of nucleotides for insulin protein. Afterward, Sanger proposed a method for sequencing with chain-terminating inhibitors [4]. This approach allowed only quite short strands to be sequenced, with its maximum at around 1000 base pairs. The shotgun sequencing mechanism overcomes these cons of Sanger's sequencing by breaking up molecules into random small segments. Reads of DNA are executed multiple times to assure that each segment is present in numerous reads. In the next step, computer software is needed to reconstruct the whole DNA by matching subsequent reads based on overlapping. Thoughts and strategies on utilizing computer programs to reveal the DNA sequence from particular reads were shared by R. Staden in 1979 [5].

Human Genome Project [6] was supposed to be a breakthrough in DNA sequencing. Starting in 1984, its goal was to sequence the whole human DNA, while identifying and mapping genes within it. The Human Genome Project took advantage of shotgun sequencing, performed in 20 research centers and universities. In 2003 the project was completed, providing the first sequenced human genome. It took nearly 20 years and \$3.8 billion to sequence the first human genome containing approximately 3 billion base pairs. It is said to be the best single investment ever made in science [7].

Sanger's sequencing method did create a breakthrough and is still widely used, but it has downsides mainly in its biological bias, capacity, performance metrics and cost [8]. That is where next-generation sequencing (NGS) methods come in place. NGS provides inexpensive, high throughput and high volume way of sequencing DNA. The development of human genome sequencing cost is visible in Figure 1.

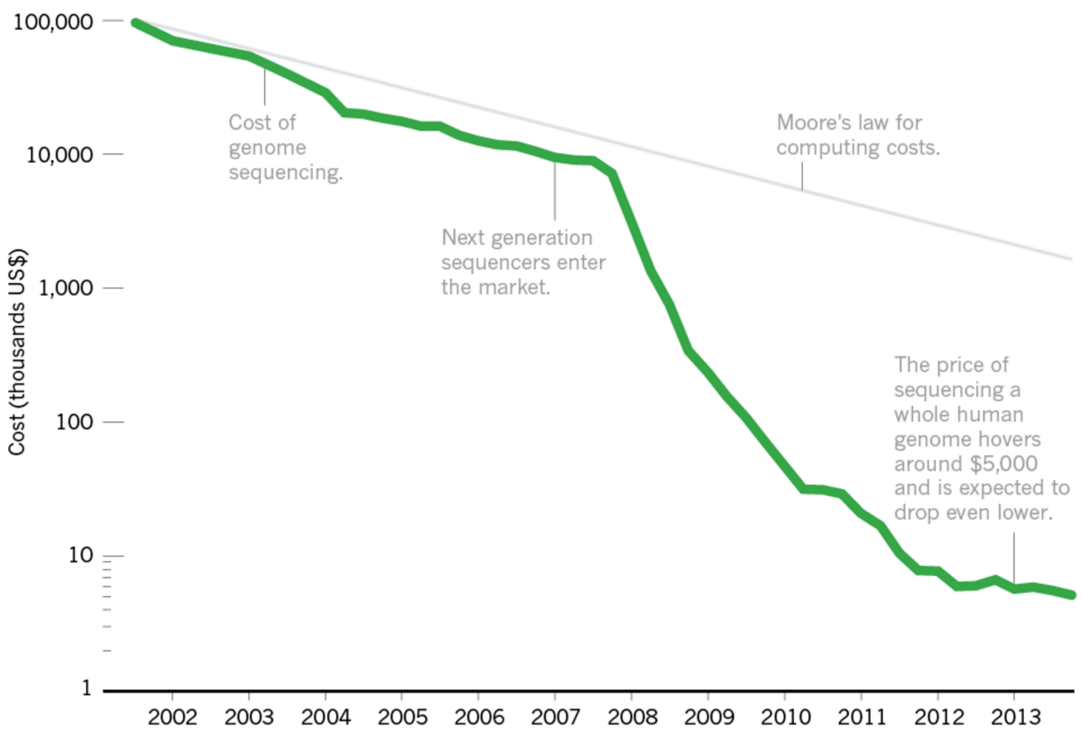

Figure 1 - Cost of Human Genome Sequencing [9]

Next-generation sequencing achieves such results by applying a set of steps consisting of template preparation, imaging and genome alignment. NGS technologies make sequencing faster and easier to execute, typically providing enormous volume. Data gathered from NGS usually consists of a huge number of short reads, which make it currently a very cheap technology with regards to base pair discovery. After obtaining genome fragments from the reads, reconstruction of the original genome must happen. The genome reconstruction is a very difficult task requiring complex algorithms and considerable computation resources.

2.1. Relevant terms

The following chapters present important biological terms used throughout the text. They are explained in a limited detail sufficient to understand their role in the problem of genome assembly algorithms. The terms explanations are mainly based on the article by Niranjan Nagarajan and Mihai Pop [10].

2.1.1. Base pair

Nucleotide bases in DNA sequence are connected in duplets called base pair [10]. These bases are created on the principle of nucleotide complementarity. During sequencing process, bases are discarded, and only the single stranded sequence is further analyzed. This is sufficient since the second strand can be easily deducted from the first one.

2.1.1. Sequence

A set of letters representing nucleotides is called a sequence [10]. The letters A, C, G and T drawn from the alphabet are ordered in a linear manner, defining the structure of the DNA. These structures are capable of carrying biological information of the sequence owner. Letters A, C, G and T stand in for nucleotides adenine, cytosine, guanine and thymine [11]. Sequences can be pushed to DNA sequencing mechanism for further interpretation and analysis.

2.1.2. Read

DNA sequencing methods provide an output consisting of reads [10]. Reads are used in further stages to assemble the original sequence. Reads can be of various length, and they do originate from throughout the sequence. The more reads are obtained for each position of the sequence, the higher the quality of original sequence reconstruction is. Reads are not ordered one after another, but they do get created with overlaps between each other.

2.1.3. Overlap

Overlap is a succession of nucleotides common for two different reads if the first read starts with this succession and the second read ends with it. The length of overlap between reads is a number defining a number of nucleotides occurring in the final section of precedent and as well at the beginning of descendant. Overlaps are essential in further assembly because the ordering of reads can be estimated based on their occurrence.

2.1.4. Contig

Overlapping reads are put together one after another resulting in a contig [10]. Th contig is, therefore, a subsequence of the original sequence built in a growing manner by adding more and more reads to it. If no reads are left and there is only one continuous contig left, the original sequence has been successfully assembled. However, there can be multiple contigs originating from the reads' set. That means a part of the original sequence is not sufficiently covered by reads and the scaffolding occurs. Contigs, due to their nature, are important when it comes to assessing the quality of genome assembly and are used in multiple quality assembly computation methods.

2.1.5. Scaffold

Scaffold [10] is a set of correctly ordered contigs, which are not connected in continuous linear sequence. Scaffold consists of mentioned contigs and gaps between them. Gaps occur in places where relationships between reads do not pass the confidence threshold. Based on the complexity of the original sequence and read coverage, multiple scaffolds can coexist after the assembly is finished. In the ideal case, the output of assembly method shall provide only one scaffold, representing the original sequence.

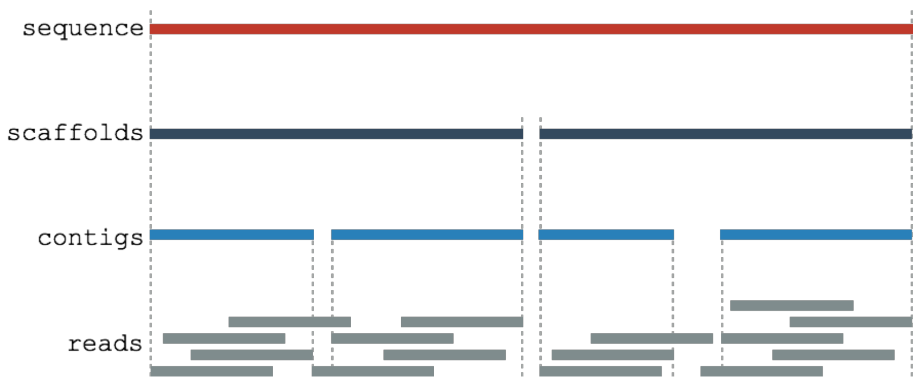

Figure 2 – Visualization of the important terms

2.2. Sequencing Principles

Sanger sequencing, referred to as the First-Generation Sequencing, has reached its limits in terms of relatively low throughput cause by in vitro process and high cost bound to it. After automating almost all steps in Sanger sequencing process, it was clear that the limiting factor is huge and new sequencing methods must evolve. More on how Sanger sequencing was automated can be found in the research by Metzker [12].

Next-generation sequencing refers to newer methods that differ in ways used in process of obtaining DNA information from samples. The process itself is quite stable and consists of template preparation, sequencing, imaging, genome alignment and assembly steps [13].

2.2.1. Template preparation

DNA sequencing starts by preparing a template consisting of genomic DNA strands. Todays' NGS methods perform breaking DNA into smaller pieces. Afterward, the templates are attached to support altogether with other broken DNA pieces. This allows multiple fragments to be read concurrently, speeding up the process. To increase the quality of reads by imaging systems, broken DNA strands are duplicated in the same regions of the template. This increases the glow and recognition factor in later stages. The process of putting the same broken strands next to each other on the template creating clusters of strands is called amplification.

2.2.2. Sequencing

After templates with broken genomic DNA strands are inserted into NGS device, DNA polymerase is added. This starts up the process of building corresponding seconds strand to the broken piece on the template. Each nucleotide added by DNA polymerase is terminated with special fluorescence chemical. This chemical called terminator can distinguish nucleotides of DNA by shining with different light spectrum. Terminators are used for two important things in the process. The first one is controlling the building of second DNA strand, and the other one is to distinguish between different nucleotides located on the top of the second strand.

2.2.3. Imaging

In the imaging stage, the template is being photographed every time a layer of nucleotides is washed away. The image consists of multiple broken DNA pieces on the template, each piece being a consensus of multiple duplicates at the same position. Duplicates might be at a different nucleotide stage due to errors while washing away the top layer. This behavior is called signal dephasing and limits the size of reads to 10-1000 base pairs, depending on the platform used.

2.2.4. Genome alignment and assembly

Alignment is a step performed after all NGS reads are available. There are two possible ways of sequence construction, either to align it to an already known reference genome or to perform de novo assembly [14]. The precondition of the first way is that there exists a trusted reference genome for DNA being sequenced, which means de novo assembly was done earlier. De novo assembly requires advanced algorithms to provide final DNA sequence. Approaches to de novo assembly task will be discussed further in the following chapters.

2.2.5. Sequencing errors

Sequencing process is not error-free, and some flaws in the sequencer output may occur. Physical or chemical thresholds mostly cause the nature of these errors. This can be either wrong recognition of the nucleotide or addition of a base that is not terminated by blocking polymerase. Therefore, it is necessary to assign a quality value to each nucleotide that has been sequenced. FASTQ format for next generation sequencing provides such option and is explained in Chapter 2.3.

As mentioned earlier, NGS converted domain of DNA analysis to an entirely different level, making it a commercial technology easily usable by research labs all over the world. Most NGS methods provide as many as one billion reads per instrument run. This makes obtained data cheap and ready for further analysis. Sequencing technologies were the focus of the research in many commercial and state centers, which means that various methods are available, each having its strength in specific use cases. There are two main methods, clonally amplified templates and sequencing by synthesis [13]. The latter one is more widely used, though each has its pros and cons.

Next-generation sequencing process was described in a limited manner due to nature of this text. For more information on NGS processes, methods and tools, please refer to the more detailed overview provided in the paper Sequencing technologies – the next generation [13].

2.3. Data Input

FASTQ data format is the golden standard when it comes to storing reads produced by sequencers [15]. Each sequence read is saved in four lines, each having its specific function. The first line starting with '@' character is defined as read identifier, with no limitation to character types or length. Therefore, this line can also store additional comments referring to particular read.

The second line is reserved for the particular sequence read; that means a set of nucleotides represented by A, C, G and T. Uppercase is the convention for storing read sequence. The length of this line depends on how long reads the sequencer can produce.

The third line starts with the '+' character and signals the end of the sequenced read. It can also contain the copy of read description and comment from the first line, but lately, this has been omitted due to storage savings reasons.

The last comes the line specifying read quality. It contains a subset of ASCII characters, where each character defines the quality for one nucleotide. That is why the length of the quality line must be equal to the read length. The encryption of quality into ASCII character is easy and efficient and works based on equation

$$Q=-10*log10e$$

where e is the estimated probability of the wrong classification of read base and Q is the character encoding in ASCII [15].

3. Distributed Systems for Parallel Processing

Massive parallel processing is the key to big data analytics. There are three views which define big data, these are velocity, variety and volume [16]. To establish a data source as big data, at least two properties must be met. For DNA sequence reads, it is mainly volume and variety of data, though velocity can be also taken into account when real-time sequencing is to be discussed. Massive parallel processing not only provides parallelism, but it also divides huge task into small pieces making real-time analytics available. Big data could not be a phenomenon on its own. Indeed, it requires distributed parallel systems to help get value out of it. That is why big data analytics and cloud computing go and develop hand in hand [17]. The following chapters present available cloud computing technologies and their relevance to processing NGS datasets.

3.1. NGS in Cloud

Cloud is providing shared configurable resources, scalability and optimization of resource usage. Cloud is not only something that is available in a third party data center, but cloud can also be hosted inside an organization, so-called “on-premise” cloud. There are three different service models in which cloud services can be offered.

3.1.1. Infrastructure as a Service

Provider of Infrastructure as a Service (IaaS) runs a solution that offers infrastructure ready for deployments. Users are abstracted from the physical layer of data center and storage, physical servers, network and other data center related resources are provided. On the other hand, end users are responsible for Operating System installation and patches, including OS and application level management. Machines in IaaS model are provided using virtualization technologies.

From the perspective of deploying distributed algorithms for DNA sequencing into IaaS platform, one could rent resources from a provider and install analytical tools of choice onto it. This option provides least limitations in terms of tools used.

3.1.2. Platform as a Service

If specific developer tools and application programming interfaces are provided for usage, cloud is available via Platform as a Service (PaaS) model. In this case, user receives a limited number of tools which can be used. There is no management overhead for user when it comes to hardware and middleware, all patching, upgrades and availability is ensured by cloud provider.

Various platforms are targeting DNA analysis on PaaS. Amazon Web Services provide data from 1000 Genomes project available. Researchers then pay only for additional resources needed to process and analyze data [18]. Google Genomics project also provides API to store, process, explored and share DNA sequence reads on cloud utilizing Google's cloud infrastructure [19].

3.1.3. Software as a Service

Software as a Service (SaaS) model provides user access to third party software and data. Available software is hosted and managed by the cloud provider, giving users only permissions to access functionality. Therefore, end users can start working without the need of gaining computer resources and without performing any installation. SaaS is also called on-demand software.

Next generation DNA sequencing and analysis is making its way to SaaS platforms, one of the first is GenomeNext [20] which provides online software for cost-effective, reproducible and deterministic DNA analysis. Algorithms for sequencing, gene expression and molecular diagnostics are available as well [20]. More SaaS providers for DNA sequencing and analysis can be found at [21].

3.2. MapReduce

Cloud development has provided computing power for the era of big data, but it would not be essential without creating a new style of application design. MapReduce was first proposed by Google in 2004 at the Operating systems design and implementation conference [22]. MapReduce technology can process huge volumes of data by performing Map and Reduce steps, all of this on commodity hardware. Map step takes key and value pairs and computes auxiliary result. Reduce step then collects all auxiliary results by common key and provides final result.

3.3. Hadoop

Hadoop is an open-source computing environment based on distributed file system [23]. Hadoop implements MapReduce programming style, which makes it an effective, highly scalable system able to run on commodity hardware. Hadoop strength is in its design, where huge file objects are traversed and functions are applied as close to data location as possible. This way functions can be applied to enormous volume of data, while satisfying redundancy and mentioned horizontal and vertical scalability. Files are stored in specific file system component called Hadoop Distributed File System (HDFS).

Hadoop-BAM is a novel library for the scalable manipulation of aligned next-generation sequencing data in the Hadoop distributed computing framework. It acts as an integration layer between analysis applications and BAM files that are processed using Hadoop. Hadoop-BAM solves the issues related to BAM data access by presenting a convenient API for implementing map and reduce functions that can directly operate on BAM records. Hadoop-BAM can be utilized for coverage calculation, variant detection or many other analysis tasks to be executed on next-generation sequencing data [24].

3.4. Apache Spark

Apache Spark is an open-source cluster computing framework with in-memory processing capabilities. The main properties of Apache Spark are speed, ease of use and sophisticated analysis of big data. It was originally developed in 2009 in UC Berkeley's AMPLab and open sourced in 2010 as an Apache project [25]. Spark offers big data processing framework that is able to process diversified data, in terms of its structure as well as source. In other words, Spark can provide data analysis on text data, graph structures and on other raw formats. MapReduce is implemented by Spark as well, but on the top of standard MapReduce, it provides multi-step data pipelines using a directed acyclic graph pattern and enables in-memory sharing across directed acyclic graphs. Standard Hadoop MapReduce jobs have to be executed in defined order so that data from actual computation job are fully processed until handed over to next MapReduce job. This is the reason why Spark is multiple times faster than other big data technologies in specific use cases [26].

Spark's strengths come from its features [27], which include:

- Support for MapReduce functions as well as standard algorithms
- In-memory allocation of frequently accessed data
- Lazy evaluation of big data queries
- API in Java, Scala and Python
- Compatibility with HDFS
- Interactive shell for Python and Scala
- Sandboxed runtime inside Java Virtual Machine

The hidden secret of Spark's performance lays in a simple fact, rather than writing temporary results to disk, it saves them in memory. This becomes even more efficient when the temporary results are used more times. However, Spark is not only limited to in-memory data storage. In specific cases, it can store data directly to disk or potentially to Hadoop cluster. As a result of that, Spark can handle computation which are much larger than its available in-memory storage, while still maintaining performance advantage due to the utilization of in-memory capabilities for some temporary results.

Spark supports lazy evaluation as well. Big data queries are optimized since their execution depends on the specific workflow in which they are processed. The evaluation time is triggered by lower levels of the framework, from which the application developer is shadowed by using high-level APIs [25].

Resilient Distributed Datasets (RDD) is the foundation of Spark framework. RDD is a data storing structure, similar to a table that can store any type of data. RDDs are immutable and partitioned. They are used to relocate data over the cluster and optimize it for processing. RDDs are highly available and fault tolerant, because RDD can be recreated by a finite number of steps from another RDD. The creation of a new RDD has two possible scenarios, either it is created from data in persistent storage or through deterministic operations on a different RDD. In general, there are two operations supported on Resilient Distributed Datasets [25].

1) Actions

Actions executed on an RDD will return a new value. All data computation required to produce this value are executed immediately, and the result of an action are returned. Examples of action commands are reduce, collect, count, first, take, countByKey and foreach.

2) Transformations

Transformation executed on an RDD will result in creation of new RDD. Also transformation tasks do not get executed at the time they are called, but they do create a relationship between the old and new RDDs for further computations. Examples of transformation commands are map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, pipe and coalesce.

Spark comes with multiple additional tools that extend its capabilities to enable graph processing with GraphX framework, advanced machine learning algorithms available via Mllib, SparkSQL to enable support of query language and many more.

3.4.1. GraphX

GraphX is a distributed graph processing library built on the top of Spark [28]. GraphX provides a framework for implementing new algorithms to model relationships between objects. GraphX together with Spark creates an environment that has the functionality available for graph processing as well as for table-structured data. GraphX extends standard Spark RDD with Resilient Distributed Property Graph. Such multi-graph has properties assigned to each node and edge. GraphX comes bundled with a significant number of algorithms for graph analysis, including PageRank, ConnectedComponents and many more.

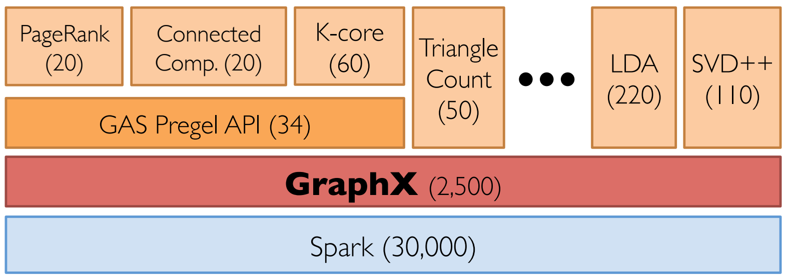

Figure 3 - GraphX framework position in the Spark framework [28]

3.4.2. SparkSeq

SparkSeq is a fast, scalable and cloud-ready tool for the interactive genomic data analysis [29]. It utilizes Spark's MapReduce capabilities to provide genomic analysis in an interactive way via Spark shell. It also opens possibility for applying Spark ecosystem tools like Machine learning to next-generation sequencing data. SparkSeq combines pros of Hadoop-BAM and Spark to improve filtering of sequence reads, summarizing genomic features and additional statistical analyses. Particular algorithms offered by SparkSeq are Position encoding, Exon encoding and Coverage calculations.

3.4.3. ADAM

ADAM is a fast, scalable genome analysis library for exploring genomic data [30]. ADAM defines a new data type meant for storing and processing genomic data, which is called ADAM format. This new format saves up to 50% of space which brings another plus in terms of processing and transfer performance. ADAM is rather good at detecting variants in genomes. Variant is a difference between two genomes which causes uniqueness of DNA samples per each human. ADAM offers functionality to convert from most standard genomic data formats into its own, but what is more, it implements algorithms for genome sorting, base quality recalibration, duplicate marking and other read quality recalibration functions. To sum up, ADAM is scalable, data is compact, read performance is improved and code is simpler.

3.4.4. Spark Internals

To write efficient map-reduce applications in Apache Spark, one must study how submitted jobs are internally handled within Spark cluster. It is quite obvious that submission of tasks to the whole cluster does happen from a master node. The master node is aware of all spark worker nodes and data location amongst them. That is the reason why it can efficiently distributed the code closest to the data, minimizing the runtime and movement of data over network. Following sections will explain, what exactly happens to a job after being submitted to Spark master node [31].

3.4.4.1. Spark Cluster Setup

Spark cluster consists of three different types of nodes, each having its specific function. Master node is responsible for managing cluster resources and keeping the cluster available. Spark driver node requests cluster resources from master node and executes applications on worker nodes provided for computation by master node. Spark worker nodes are nodes, which process the tasks assigned by the driver node. Worker nodes are effectively doing all the productive work.

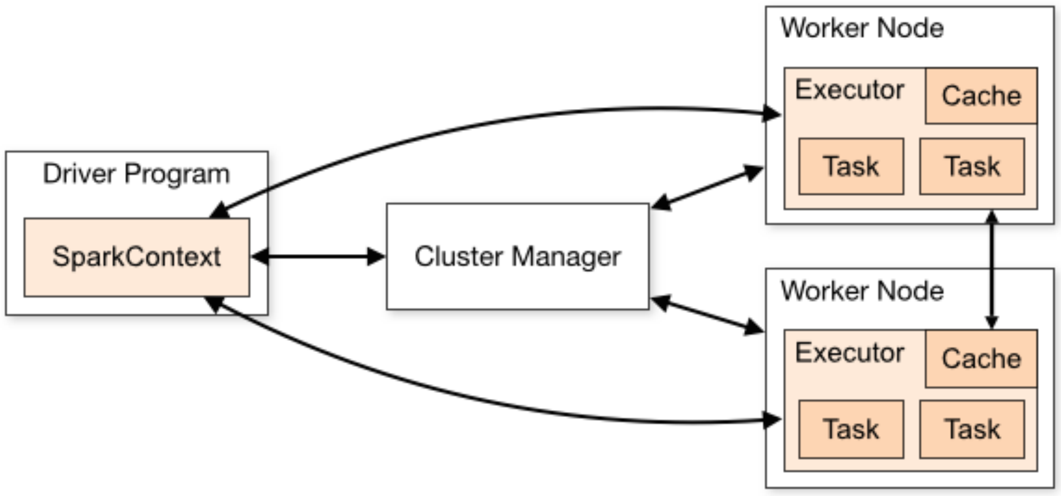

Figure 4 - Spark cluster setup [32]

3.4.4.2. Directed Acyclic Graph Creation

Each action on an RDD triggers a necessary creation of directed acyclic graph, which specifies the stages of execution that must be passed for a valid termination of requested action. A stage is a set of actions that cannot be pipelined, i.e. stage can be completed only with full set of data used in the stage already available. Stages can also be referred to as super-operations.

3.4.4.3. Logical Plan of Execution

Shuffling, redistribution of data, and directed acyclic graph validations happened during the planning step. During this step, a plan how to move data to particular nodes is created, in order to have relevant data for relevant stages easily and quickly accessible. The data movement is an expensive operation and it can be optimized by partial aggregations or by minimizing the actual data flow by proper partitioning. The partitions are subsets of data, which cannot be further divided. These subsets provide concurrency, because different partitions can withstand on different nodes in the cluster. Setting the correct number of partitions is crucial to overall performance, since a number too high can reduce concurrency which will lead to resource lower utilization of available hardware. On the other hand, a high number of partitions might suffer from data skew and operational overhead.

3.4.4.4. Scheduling

The main purpose of scheduler is to assign tasks to machines in the cluster based on data availability and location. As stated earlier, the speedup of map-reduce programming style is by sending code, in this case tasks, directly to data. This is what scheduler is responsible for. Since dealing with distributed system, failures must be considered as a part of normal operation. In case of task failure on one of the nodes, the scheduler retries running the task on another node, where the replica of the corresponding partition resides.

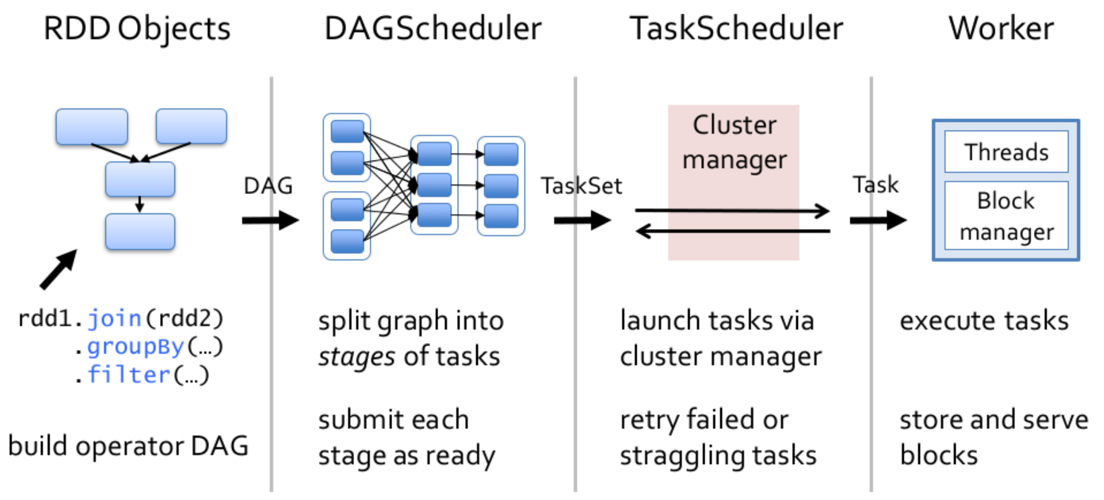

Figure 5 – Spark execution flow [33]

4. Genome Assembly

In the following chapters, algorithmic approaches to genome assembly are addressed. First, a general view of assembly principles is discussed, followed by specific algorithms that are currently available for implementing de novo assemblers. The first algorithm is based on the Overlap graph, whereas the latter takes advantage of de Bruijn graphs. A description of ways to improve performance by optimizing input dataset is provided as well. In the end of this Chapter 4, both algorithms are compared and based on the comparison, one is chosen for further distributed implementation, evaluation and performance testing.

4.1. General Approach

The output of a sequencer is a set of short reads which is used to reconstruct the genome. This means reads should be ordered in a way they occur in the original genome sequence. These reads cover each section of genome multiple times with some overlap, which is the key to deciding on the order of reads. The overlap between two reads is a property of their relationship while read content is its representation amongst the set. The length of the overlap increases the probability of co-occurrence of these reads one after another and strengthens the relationship between them. This all leads to the creation of a graph, which will represent the relationships between reads in an effective way. Multiple ways of representing nodes, edges and relationships between them exist [13]. All of them are essentially focused on the same thing, and that is to truly reflect genome that was at the input of the sequencer. On the other hand, all of these representations take a different way and shortcuts on how to provide a high-quality output.

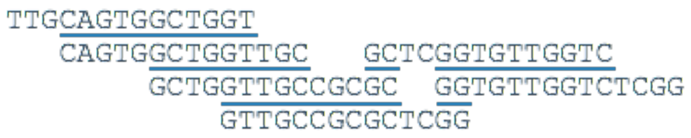

Figure 6 – A set of overlapping reads

4.2. Sequence Assembly Algorithms

The most popular algorithms for sequence assembly are explained in the next chapters, including issues that may occur during algorithm usage and how to avoid them or fix them. Firstly Overlap Layout Consensus (OLC) algorithm is discussed, as it is more traditional than the de Bruijn approach and has been widely used for longer reads in the first generation sequencers. After that, de Bruijn algorithm introduced with next-generation sequencing is addressed.

4.2.1. Overlap Layout Consensus Algorithm

The OLC algorithm relies on the construction of overlap graph [34]. The overlap graph is a directed graph representing reads and relationships between them. The approach using this technique dates back to Sanger sequencing methods and was used mainly due to good performance with longer reads. OLC algorithm requires discovery of all overlaps between all reads. That is why the graph may get huge and impractical. Due to this property, numerous optimization algorithms are applied to make this method more time and memory efficient. These methods will be further described in the chapters below.

4.2.1.1. Input Specification

The algorithm expects all reads provided by sequencing machine on input, including their quality assessment on a base level. Further on, it is necessary to provide an identity function that will reflect quality into the comparison of reads. In the ideal case of no expected errors in the input read set, the identity function can be set to exact matching only. An important input parameter requiring tuning is the minimum overlap of reads. Minimum overlap is a threshold that needs to be met to identify two reads as overlapping.

4.2.1.2. Pre-processing of Input Data

Before building the actual overlap graph, overlaps between all pairs of reads need to be identified based on input conditions. The overlap length defines how many bases of a read suffix are identical to some others read prefix. Only the longest overlap is considered, preventing multiple relationships between the same pair of processed reads. The overlap discovery starts from the minimum overlap threshold and grows until the maximum possible value, also called as the seed and extend strategy [34].

4.2.1.3. Graph Construction

The vertices in the overlap graph are referring to reads in the input dataset. This means the graph can have as many vertices as there are reads provided. The edges connecting vertices define the relationship of the overlap, where each edge has an integer property specifying the length of the overlap. The higher the number, the more similar the reads are. The resulting graph is a multigraph in the most of the cases.

4.2.1.4. Optimization of Graph Structure

Multiple optimization techniques exist to improve OLC algorithm performance. Some implementations focus on minimizing the dataset even before the graph construction, by deleting duplicated reads from the input dataset [35]. This is very likely to speed up the graph construction and sequence assembly due to fewer comparisons executed when looking for overlaps between reads.

Other approaches focus on minimizing the graph size by pruning spurs [35]. Spurs are dead ends to which a path exists only due to sequencing errors. The constructed overlap graph can also contain small disconnected structures, which could be essentially removed due to their short length. Moreover, multiple edges can contain the same information, creating transitive edges in the graph. This transitive edges can be eliminated, because by removing them no information about original genome is lost. Bubbles in the graph occur when there are two very similar subsequences of the genome. These bubbles can be collapsed into a linear structure in order to compress the graph, but the genome information from the bubble must be stored in the linear structure [34].

4.2.1.5. Sequence Assembly

Once the graph is constructed and optimized for sequence assembly, the graph represents the approximate layout of reads. To generate the sequence, maximal simple paths in the graph must be found. Finding a maximal path in the graph is an NP-hard problem that can be reduced from Hamiltonian path [36]. In the special case of a directed acyclic graph, the longest path problem can be transformed into the shortest path problem that is polynomial [37]. This is achieved by the negation of every edge weight, where the cost of each edge e_i is calculated as an inverse value of overlap between vertices v_1 and v_2, where e_i is the connecting edge between these vertices. The assumption, in this case, is that the longest overlaps will provide the most probable order of vertices [38].

The output of the sequence assembly is not one single sequence. The cause of this are repeats present in the sequence, which make the sequence assembler unable to decide how many times the repeat is occurring in the sequence, or in other words, how many times should the cycle be iterated. This problem is nowadays practically unsolvable in the OLC algorithm, and therefore, the output of assembly is a list of maximal simple paths, each being a separate contig.

4.2.1.6. Overview

The OLC algorithm based on Overlap graphs is a traditional, well-performing algorithm that is very suitable for larger genomes and deals efficiently with longer reads. In the case of the input consisting of short reads, its advanced heuristics for optimization of stored data structure helps performing similarly to de Bruijn graph based algorithms described in the next chapter [39].

4.2.2. De Bruijn Algorithm

The algorithm for sequence assembly based on de Bruijn graph dates back to the introduction of machines producing short reads from input genome. K-mer structures are the ground elements on which de Bruijn graph assemblers rely. By design, the approach does not store and utilize reads during graph creation and analysis, but it rather uses compressed k-mers for the benefit of more compact graphs. Even more, not all overlaps within all k-mers are necessary to be discovered, and the produced graph compresses repeated parts of the genome. The final assembly is also not an exhaustive operation, it is considered as a byproduct of graph construction.

4.2.2.1. K-mer Creation

K-mers are defined as all substrings of a given string with length `k`. Given a read with length `L` and a required k-mer with length `k`, there are `L-k+1` k-mers for each such read. In practice, this means each read is divided into smaller chunks that are used instead of input reads during overlap calculations. The read length `L` is given by sequencing machines, whereas k-mer size is one of the algorithm inputs. The correct choice of length `k` is essential for further successful assembly and depends on the preliminary assessment of genome structure. The lower size of k-mers produces a smaller set of unique k-mer and therefore results in smaller de Bruijn graph with fewer edges. On the other hand, small k-mers are more probable to cause issues during sequence assembly due to generally higher input and output degrees of edges. Longer k-mers negatively impact the graph size by increasing the number of edges. Also preferring larger k-mer may lead to a situation of more contigs in the assembly because more k-mers are less likely to have sufficient overlap. The pros are that shorter repeats are absorbed and less likely to cause incorrect repeat ordering in the final sequence assembly.

4.2.2.2. Graph Construction

De Bruijn graph construction can be initiated once all k-mers for given reads are available. The next step is to extract left and right (k-1)-mers from given set of k-mers. Given this step, the output of such operation consists of two (k-1)-mers containing bases $i_1..i_{k-1}$ and $i_2..i_k$, where k is the length of original k-mer and i_j is the base `i` on j-th position. These extracted (k-1)-mers will become vertices of the de Bruijn graph. Directed edges are placed between left and right vertices of (k-1)-mers respectively. In the end, for each original k-mer of length `k`, there is a corresponding edge in de Bruijn graph. Also, it is important to note that there is a vertex for each unique (k-1)-mer extracted from the original k-mer set. Another fact is that multiple edges can exist between two vertices, caused by the same k-mers in the original set.

Given this process of graph construction, it is visible that repeats in the genome will reflect as cycles in the de Bruijn graph. It is as well essential to keep (k-1)-mers in a separate data structure so that it is not necessary to traverse the whole graph to identify distinct edge. The compressed structure of the graph is guaranteed by the uniqueness of edges and the fact that k-mers are smaller than reads and that k-mers repeat in the worst case as many times as reads.

4.2.2.3. Sequence Assembly

The reconstruction of the original genome from de Bruijn graph corresponds to a path in the graph meeting the criteria that each edge is visited exactly once. The path starts in an edge and restricted by edge direction, passes through all edges. Since edges are indeed a representation of k-mers, each k-mer is present in the final path one time. The final genome assembly is the concatenation of k-mers in the order they are visited during path construction.

The construction of path explained above is called Eulerian walk. Eulerian walks exist only in Eulerian graphs. The issue is, that de Bruijn graph for sequence assembly is Eulerian graph only in the ideal case of k-mers with no errors and full coverage of the genome. In real-life sequencing, Eulerian graphs are not common since the data is more complicated. The issue hides in natural genome structure, where repeats are common. Repeats are problematic from a simple point. They create cycles in de Bruijn graph, resulting in the existence of multiple Eulerian paths. As stated earlier, cycles can be limited by choosing longer k-mer sizes. Some other approaches refer to original reads for deciding on correct path when the path is interfered by cycles.

4.2.2.4. Overview

De Bruijn graph based algorithm provides a way of saving sequenced data in compressed form and minimizes storage and computation requirements for assembly. On the other hand, it introduces input variable k-mer length and by its nature is more suitable for short reads. In addition to all these facts, handling of repeats is an important part of each assembly utilizing de Bruijn graphs.

4.3. Comparison of Assembly Algorithms

Both de Bruijn graph (DBG) based and Overlap Layout Consensus algorithms have the majority of their features common. They both rely on overlaps of reads and represent data as directed graphs. OLC algorithm stores direct connections between reads, whereas DBG prefers storing overlaps between k-mers. Both algorithms require cleaned and pre-processed data at the input, and also both implement post-processing techniques on created graph to reduce errors and the size of resulting graphs. DBG algorithm is very sensitive to choosing a correct k-mer length, but the similar applies to OLC with its minimum overlap length threshold input parameter.

The biggest difference lies in the expected read length on the input. While OLC algorithm prefers longer reads (100 to 800 base pairs) with smaller coverage acceptable, the DBG algorithm is suited for today's next-generation sequencing reads ranging from 25 to 100 base pairs per read [39]. On the other hand, the OLC algorithm with its advanced heuristics discovered lately competes with DBG algorithms quite well even on this field.

None of these sequence assembly algorithms is perfect, as both have their issues that have not been fixed yet. DBG algorithm's performance strongly depends on the length of the Eulerian path in the graph, which can get very short for repetitive genomes [39]. OLC algorithm also has problems identifying repeats longer than read length.

In the future, a technical advance in sequencing machines will bring longer reads that would better deal with repetitive genomes [40]. Once the supplied reads are long enough to house the whole repetitive subsequence inside it, the OLC algorithm might have a very strong benefit on its side. However, the main task will still be to provide a well-performing algorithm for rapidly changing conditions in the area of next-generation sequencing. The algorithms must work and scale well with provided read length and quality, adapt to increasing volume and utilize growing computation resources effectively.

4.4. Existing Assemblers

The next chapters are providing an insight into today's most popular sequence assembly software. Each tool is briefly described and its key advantages and disadvantages are highlighted for easier understanding of their usage.

4.4.1. SOAPdenovo2

SOAPdenovo2 [41] is an assembly method specializing on Illumina GA short reads. The tool's performance has been presented by building a de novo draft assembly of a human-sized genome based on short reads. SOAPdenovo2 is based on de Bruijn graph construction, with a high focus on read error correction and decrease in memory consumption. SOAPdenovo2 works best with reads from 35 to 50 base pairs, has embedded scaffold identification algorithm and excels in assembly speed. It is known to be deployed on supercomputers and is widely used [41].

4.4.2. MEGAHIT

One of the most advanced single computer time and cost efficient de novo assemblers is MEGAHIT [42]. It can perform a de novo assembly of large and complex metagenomics data, but has its bounds in terms of machine memory capacity. MEGAHIT uses improved compressed representation of de Bruijn graph, also known as succinct de Bruijn graphs [43]. Also, additional edge properties are added to make a dynamic removal of edges more efficient. There are a lot of pros to MEGAHIT, but the main construction of succinct de Bruijn graphs is still a very expensive task. These graphs are built iteratively, from k-mers of small sizes to very large. Due to improvements to standard algorithms that MEGAHIT provided, it guarantees great completeness and contiguity, and at the same time, efficient assembly of large and complex metagenomics data [42].

4.4.3. Velvet

Another assembly method utilizing de Bruijn graphs and focusing on very short reads is Velvet [44]. The key distinguisher for Velvet is its ability to produce valid assemblies leveraging very short reads while taking into account read pairs and removing errors that could appear during de novo assembly. Velvet is fairly easy to setup and run, can handle reads of various length, but its main drawbacks remain the extensive usage of memory and sensitivity to the setting of input parameters [44].

4.4.4. MIRA

MIRA is a powerful tool capable of both sequence assembly and mapping [45]. It has extensive documentation and can be used for many use cases due to its modularity and configuration possibilities. MIRA uses a unique trace signal analysis routines which are suitable for shotgun sequencing. It has acceptable performance for smaller genomes, but can struggle when dealing with larger ones.

4.4.5. AbySS

MPI is a message parsing protocol used by AbySS to be able to run de novo sequencing on multiple nodes [46]. While this can be considered as a distributed approach, it still suffers from extensity of message sizes. The protocol is highly specific and tailored for ABySS, but in connection with de Bruijn graph usage, its performance struggles in comparison with other assemblers. From the bright side, a set of commodity hardware can be used for assembling larger genomes. The algorithm behind ABySS works in two basic steps. The first step extends contigs until it is no longer possible. In the second step, these contigs are merged. ABySS comes the closest to distributed de novo genome assembly, but it still has its downsides in error correction, performance and deployment architecture.

4.4.6. SAGE

SAGE (String-overlap Assembly of Genomes), unlike others, performs de novo assembly via the construction of overlap graphs. SAGE is inspired by advanced research completed in string overlap graph construction, analysis and maximum likelihood assembly. Moreover, it adds on improvements such as graph reduction. Due to its improvements, SAGE is highly competitive with de Bruijn assemblers [47]. The process of de novo assembly with SAGE includes error correction followed by construction of bi-directed graph. In the bi-directed graph, read and its reverse complement are described by the same edge. Once string overlap graph is built, the optimizing part of the process is invoked. This consists of reduction of transitive edges, simplification of graph and removal of duplicated information. In the end, maximum likelihood is used to determine the final sequence. Successful assemblies of short to medium-sized genomes are currently supported by SAGE while the misassemblies are still impacting its overall quality. SAGE is developed in C++, is platform independent and does not support distributed deployments on cluster [47].

4.5. Misassembly

Two leading causes of misassemblies can be distinguished. The first one is caused by collapsing or expanding repeated reads, originating in incorrect estimation of how many repeats of genome part shall be present in the final assembly. The final assembly can either contain too many or not enough copies of reads. Sequence rearrangement and inversion cause the second main type of misassembly, originating in the wrong order of subsequences, e.g. not putting a subsequence in the correct position of the final assembly. It is essential to take misassemblies into account when performing analysis of the assembly since this can prevent invalid conclusions when examining the final assembled sequence. A good metric of identifying assembly with many misassemblies is to check whether sequencing errors do occur in regular locations across multiple reads.

4.5.1. Repeat collapse and expansion

If a collapse misassembly happens, reads that refer to distinct locations of the sequence are joined into a single location [48]. This results in a higher density of reads in a specific location of the sequence. The effect of collapsing is unpredictable since reads attached to a specific location take more reads to the same location with them. The other effect of this kind is an expansion, happening when reads that are the same repeatedly occur in the assembly. The effect of expansion causes subsequences with a lower density of reads due to their relocation to locations with a higher density.

4.5.2. Rearrangements and inversions

Misassemblies are not limited only to correct placing of the reads within the sequence. Other special cases can cause a sequence structure to be broken. An example is an inversion of two reads, putting the two reads in an incorrect orientation. This triggers a process that inverts the whole region where these reads are occurring. In addition to the wrong direction of the sequence, mate-pair constraints can be wrongly oriented as well. Mate-pairs are known to identify such issues in early stages, but a whole assembly can be constructed without violating any mate-pair constraint. In that case, it is difficult to identify such misassembled sequences [48].

4.6. Assessment of Assembly Quality

To measure the quality of genome assembly, various methods are present. The most traditional one is called N50. It is a statistical method taking into account lengths of contigs and scaffolds. The value of N50 represents the smallest contig (or scaffold) meeting the condition that 50% of contigs (or scaffolds) is the size of N50 value or higher. To compute the exact value of N50 for assembled contigs, the following procedure shall be performed:

1) Order all assembled contigs of the sequenced genome by their length in descending order.
2) Start putting contigs into a set and summing up their contig lengths from the beginning. Stop when the growing summation is equal or higher than 1/2 of all contigs lengths summed up.
3) The value of N50 is equal to the length of the shortest contig in the set.

A similar procedure can be applied to calculating N50 value based on scaffold lengths, the only difference is that scaffold lengths are used. In case N50 value is equal to half of the genome length, it can be said that 50% of the genome will be present in a single contig. N50 value of an assembly which resulted in a single contig is the length of the contig. The reads prior to assembly have the N50 length equal to read length.

The quality of assembly grows with N50 value, with increasing N50 value the quality of genome assembly is considered as better. Such way of quality assessment directly prefers larger contigs to be of higher quality. The issue is that contigs of greater length can be erroneous and therefore decrease the quality of the whole assembly. Each sequence assembly algorithm is deciding on the contig merging during its run. Therefore, some of the assemblers may apply an optimistic strategy for contig merging in trade off higher assembly quality. This risk must be accepted when using N50 for assessment purposes. On the other hand, conservative assemblers preferring shorter contigs with increased probability of correct merges may score very low in N50 criteria. All in all, N50 may be a misleading statistics under particular circumstances, but it is considered as the leading quality evaluation criterion when it comes to evaluating genome assembly [49].

5. Implementation of Distributed Assembly Algorithm

The subsequent chapter offers an insight into the implementation of the distributed assembly algorithm using a specific big data workflow. The scope of the task is defined, and the building blocks of the workflow are introduced, including the description of use during the development and testing process. Further on, the assembly algorithm implementation is described, focusing on the explanation of each executed task and implementation details. The runtime aspects of the workflow are provided. Chapter 5 concludes with more information on the executing process of the solution.

5.1. Scope

The target of this implementation is to utilize an existing distributed computing platform to develop a sequence assembly software taking advantage of its parallel capabilities. The proposed solution must be scalable and reusable from the platform and source code perspective. This means the platform is a multi-purpose computing instance and the source code can be further enhanced to support more functionality. All these aspects must be taken into account when choosing suitable tools, algorithms and processes during implementation

The scope of the implementation is defined as follows:

1. develop a distributed sequence assembler tool based on one of the known sequence assembly algorithms,
2. extend the algorithm with heuristics to improve its performance and assembly quality,
3. use exact matching during overlap identification,
4. propose and commission scalable platform,
5. assess the scalability and resource distribution of the proposed solution,
6. accomplish performance testing and compare it with the referential solution,
7. perform analysis of sequence assembly quality and compare it with a similar existing solution.

Following requirements are out of scope of this implementation:

1. correct errors in input reads,
2. read quality assessment during the whole assembly process,
3. Re-implement all assembly heuristics available in the referential solution, only the functionality that mostly contributes to the parallelization will be considered.

5.2. Workflow Preparation

Prior to the start of the implementation, the setup of environments and workflow is necessary. Technologies and tools described in the following chapters provided a good starting point. Two separate environments needed to be built, the first one is a development environment running on the development computer. The development environment must be flexible, easily accessible and must support execution with low volume data. The second environment needed for experiments is test environment. Requirements for the test environment are entirely different. It must be vertically and horizontally scalable, versatile and able to work with medium to high volumes of data.

5.2.1. Development Environment

The development environment and usage of proper technologies are the keys to an effective solution. Next chapters provide an overview and a swift description of tools used within the development of assembly program.

5.2.1.1. Big data processing engine

For development purposes, a single node standalone Apache Spark cluster was installed on the development computer. Version 1.6 with pre-built Hadoop 2.6. was chosen as the latest version at the time of download. The package comes with an executable shell and submit scripts and can be easily installed based on the guide on the official Apache Spark site [50]. The package also contains required library GraphX.

5.2.1.2. Programming language

Spark can accept jobs written in JAVA, Python or Scala programming languages. The Spark shell is available only for Python and Scala. Scala was chosen due to the availability of interactive shell and its similarity with JAVA. Spark itself is written in Scala and GraphX library is also Scala based [51].

5.2.1.3. Integrated Development Environment

IntelliJ IDEA was a definite choice when it came to choosing IDE, due to its support of Scala via an easily installable plugin and ability to deploy directly to local or remote Spark clusters [52].

5.2.1.4. Build and Packaging

SBT is an exclusive tool for building Scala applications. It offers continuous compilation, integration with IntelliJ and most importantly it is easy to set up and use [53].

5.2.2. Deployment On Cloud

Development and testing purposes required building a specific infrastructure for execution of Spark applications. Since the shared cluster can result in affecting of performance testing by applications submitted by other users, an exclusive cluster just for the needs of genome assembly was built. The environment for performing test was provisioned on the infrastructure provided by MetaCentrum [54]. It offers private cloud services with a self-service portal for provisioning cloud servers with pre-installed images.

The logical design is as per following diagram in Figure 7, details about exact HW and SW configuration of servers used during performance testing is stated in Chapter 6.1 within environment description.

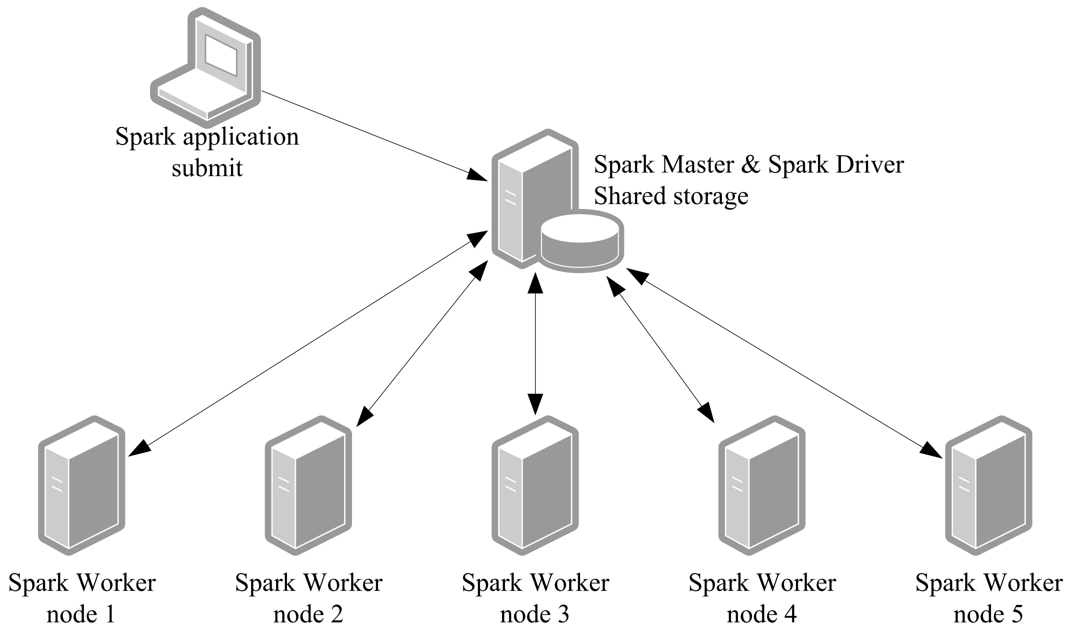

Figure 7 - Spark infrastructure diagram for the test environment

5.3. Implementation

For the implementation purposes, an instance of OLC algorithm is chosen. In particular, I use the SAGE assembler described in Chapter 4.4.6 as the reference. This is due to its numerous advantages in comparison with de Bruijn graph based algorithms and future potential, as described in Chapter 4.3. The following chapters are focused on walking through the distributed implementation of OLC algorithm with regards to efficient usage of the Spark framework. The functionality is developed according to details provided in Chapter 5.1.

5.3.1. Application Configuration

Spark applications offer a variety of configuration to be present during runtime. This implementation uses application name property to identify it easily in application scheduler during runtime on Spark cluster. Additionally the implementation benefits from using Kryo serialization library [55], which is more memory and speed efficient library for serialization and deserialization of registered objects. This means serialization times are lower and data transfer between worker nodes is reduced.

Since it is expected that the implementation can be executed on a shared Spark cluster, the mode of job scheduling is set to FAIR. Such mode introduces Round Robin scheduling [56] of jobs within the application, as well as within all concurrently running applications on the cluster.

5.3.2. Input

The application expects three parameters on the input. The first one is the source of reads in FASTA or FASTQ format. It can be either an address to a file stored on a local disk or an address of the file in Hadoop HDFS storage. The input file is loaded directly into Resilient Distributed Dataset (RDD). The second input argument is the location where the output of the application is saved, which can also be a local directory or a Hadoop HDFS address. The last argument is an integer specifying the minimum allowed overlap of reads, used in overlap detection between reads.

5.3.3. Pre-processing of Reads

The pre-processing of input data starts with removing all the unused information from the input file. Since quality assessment on base pair level is out of scope, all the input lines except those containing base pairs are removed. Input file cleaning is performed via filter operation, only lines satisfying given regular expression are left in RDD. The next steps are to optimize the number of reads for further processing. Firstly distinct operation is used to delete all duplicated reads, as duplicates carry only a copy of input data and would result in duplicated nodes and edges. Onwards each read stored in RDD is assigned with a unique number identification, which is used instead of read string for read identification during graph related operations. Map operation creating a hash code of read string can easily assign a unique id to each read. RDD with pre-processed reads is cached into memory for fast access, due to its usage in multiple following operations. Mentioned operations on the input dataset are shown in Listing 1. In the end of the pre-processing stage, essential information is computed from reads, namely the read length and the range of overlap lengths to be considered. The range is defined as an interval from minimum overlap to read length.

```
val reads = rawReads.filter{ case (line) => line.matches("[ATGC]+")}.
  distinct.
  map(read => (read.hashCode.toLong, read)).
  cache();
```

Listing 1: Pre-processing of raw reads loaded from the input file

5.3.4. Overlap Discovery

An important task of the application is to discover overlaps between reads. This task is executed by iterating over the range of allowed overlap lengths and performing an all-to-all comparison of read suffixes and prefixes. The prefix is extracted as first `n` characters of read, the suffix as last `n` characters of read, where `n` is iterated over the range of overlap lengths. Both suffix and prefix strings are hashed into an integer for faster comparison with each other. That means exact string matching is used during the comparison. If an overlap meeting the criteria between a suffix and prefix exists, identification of both reads and the length of respective overlap quality between them is saved. Listing 2 represents actions performed during overlap detection of reads with the length of overlap equal to `overlapSize` parameter. Figure 8 provides a graphical representation of overlap discovery on an example set of reads. Overlap quality is defined as the length of overlap subtracted from read length. Such computation of quality parameter guarantees that the shortest path algorithm described in the further chapters finds the path with longest overlaps. The lower the attribute, the longer overlap length between reads is.

```
val readPrefix = reads.map { case (id, read) =>
      (read.take(overlapSize).hashCode, id) }
val readSuffix = reads.map { case (id, read) =>
      (read.takeRight(overlapSize).hashCode, id) }
val readEdges = readSuffix.
  join(readPrefix).
  map { case (readSubstring, (sourceId, destinationId)) =>
      (sourceId, destinationId, overlapLength) }
```

Listing 2: Overlap detection algorithm

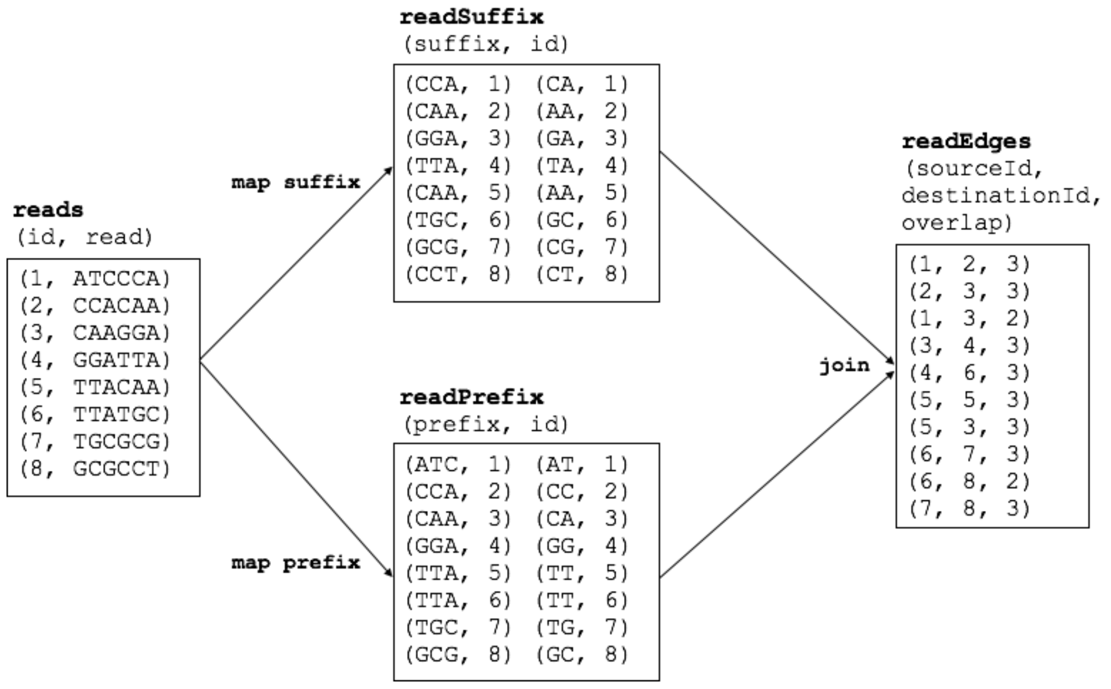

Figure 8 - Example identification of overlaps (minimum overlap equals 2)

Once lists of overlapping reads with the respective length of overlap are constructed for each allowed overlap length, all these overlaps are joined. The edges where the identification of prefix and suffix match are deleted. Items in the list represent edges in the overlap graph. Therefore, they can be converted to Edge objects provided by GraphX library. The process of edge creation is as per Listing 3.

```
val edges = rawEdges.
  repartition(partitionSize).
  distinct().
  filter { case (sourceId, destinationId, overlapLength) =>
      sourceId != destinationId }.
  map { case (sourceId, destinationId, overlapLength) =>
      Edge(sourceId, destinationId, overlapLength) };
```

Listing 3: Pre-processing and cleaning of raw edges

5.3.5. Graph Construction

Edges of Overlap graph are identified during the overlap discovery. Graph nodes are represented by reads. Before creating a graph object, there is one more optimization that can be performed on the list of nodes. All nodes that are not present in any of edges can be removed from the list since they are remote with no overlap between them and any other read. Implementation depends on edges object, from which the destination and the source vertex ids are mapped and joined to create a unique set. The creation of this unique set is visible in Listing 4. Graph node objects require an attribute, but to decrease storage requirements an empty string is used instead. The removal of vertices is a heuristics referred to as removing disconnected structures in the OLC algorithm description in Chapter 4.2.1.4. It guarantees that constructed graph contains only connected reads. After performing this optimization, graph object can be initialized as shown in Figure 9.

```
val connectedNodes = edges.map { case Edge(sourceId, _, _) =>
      (sourceId, "") }.
  join(edges.map { case Edge(_, destinationId, _) =>
      (destinationId, "") }).
  map { case (id, attribute) => (id, "") }
```

Listing 4: Removing disconnected nodes

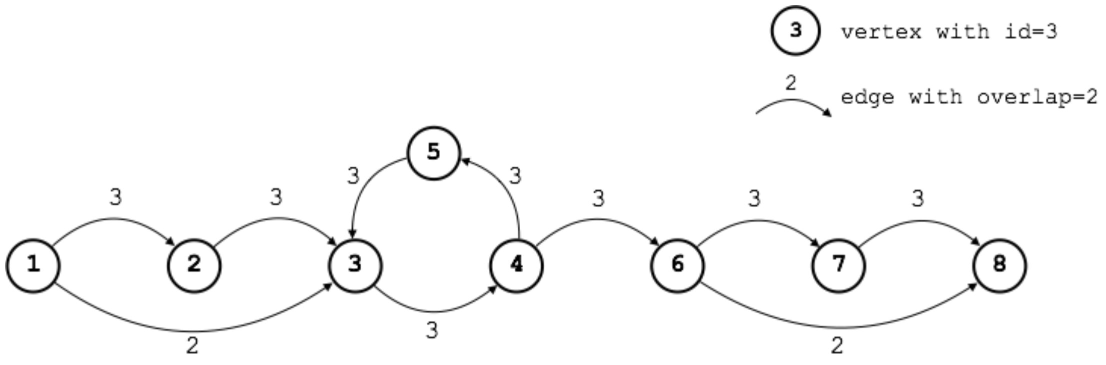

Figure 9 - Example of constructed overlap graph

5.3.6. Graph Optimization

The constructed graph is rather big, and there are multiple ways it can be made more effective for computation and suitable reaching a result of a better quality. The first import task is to identify cycles in the constructed overlap graph that do result in subsequence with an unknown number of repeats. This is performed by subtracting all strongly connected components from the original graph computed by GraphX function for finding strongly connected components. A strongly connected component of size 2 or higher corresponds to a part of the graph containing a cycle, in Figure 10 component SCC3 matches the criteria. Therefore, a sub-graph is created, containing only strongly connected components of size equal to 1.

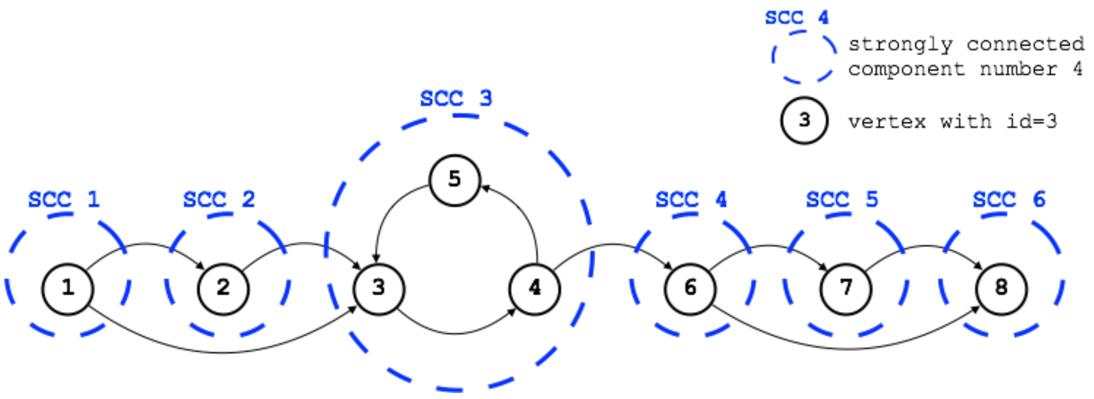

Figure 10 - Strongly connected components in an overlap graph

The implementation provided also in Listing 5, alters each vertex identification to its respective strong component id and attribute 1 is added. Afterward, a reduction by id is performed while summing the attributes, resulting in the list of strongly connected components and the number of vertices in them. Then filtering is performed to identify those strongly connected components with size exactly 1, creating a list of component ids. A sub-graph is created by keeping only those vertices, whose strongly component ids are present in the created list.

```
val oneNodeSCCs = scc.vertices.
  map { case (id, sccId) => (sccId, 1) }.
  reduceByKey { case (a, b) => a + b }.
  filter { case (sccId, sccSize) => sccSize == 1 }.
  map { case (sccId, sccSize) => sccId }.
  collect;
```

Listing 5: Creating list of strongly connected components with exactly 1 vertex

The resulting sub-graph is directed and acyclic as well. The identification of respective connected component is added to each vertex as an additional attribute, all edges are preserved.

Before the actual finding of the longest paths in the graph, it is essential to cut the graph into distinctive sub-graphs, that means into multiple graphs that are not connected by any edge. This task can be transformed into the identification of all weakly connected components. The algorithm for connected components will parse the graph into smaller pieces, in which algorithm for finding the longest path is applied in parallel. During this step, small distinct parts of the graph are removed, that means removing connected components which size does not pass the pre-defined threshold. Weakly connected components present in the graph after removing cycles are visible in Figure 11.

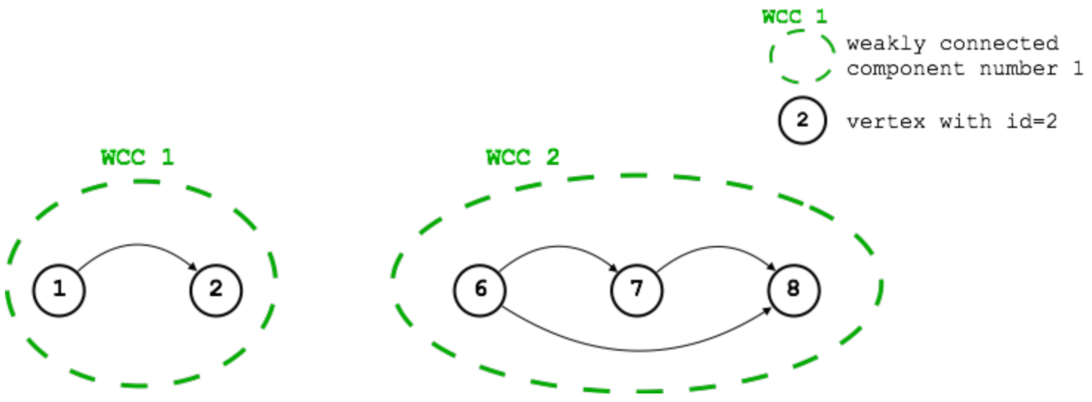

Figure 11 - Weakly connected components in overlap graph

Additionally, transitive reduction of edges can be performed at this stage of the algorithm. The proposed algorithm uses greedy implementation, which always selects the edge with maximum edge attribute heading to a processed node.

5.3.7. Identification of the Longest Paths

Identification of the longest path in the graph can be translated as a path with the longest overlaps due to the setup of edge attributes and the fact that all cycles were removed during graph optimization. This means edge attributes can be set to negative numbers without causing infinite loops in the implementation. The reformulation of the problem allowed using algorithms for finding shortest paths in a directed graph. To initialize the search for the long path, source node must be known. Since the source node, which is the start of the genome sequence, cannot be directly identified, a list of potential source nodes is created. A node becomes a potential source node, if it belongs to nodes with zero in degree value, i.e. no edges are heading to this node.

The longest path discovery is iteratively executed from all potential source nodes to all other nodes present in the graph. To identify all longest paths from a source node including the nodes passed on the way, firstly all nodes are extended with additional attributes defining the length of the path from the source node to themselves and a list of nodes visited on the path. Initial values for the longest path are zero for source node and infinity for all other nodes. The list is initialized as empty.

The computation of the longest paths from source nodes is executed via Pregel operator [28], which sends a message with a specific task to each triplet. The triplet is a structure containing information about two vertices and the respective edge between them. The first step is to define vertex program choosing the vertices for which computation is happening. In the current implementation, the computation starts from the source node with path length attribute equal to zero and continues to nodes with higher values of the attribute. Since path length attributes are initialized as infinity, consequently all nodes are processed in this vertex program. The next operation is to send the message with computation details to all triplets. This message contains information how edges' attributes should be updated once being processed. The path length attribute is equal to the path length in the source nodes added with current edge attribute in the triplet, if source node attribute is smaller than destination attribute subtracted by the triplet attribute. Additionally, the path steps attribute is updated with values of source nodes joined with the source node itself. This will guarantee that the triplet attribute always comes from the longest known path at the time of computation. Tasks for updating triplet attributes are merged for each triplet, which has final path length attribute that is not a subject to change. This ensures the computation of the longest path is deterministic. The exact implementation is provided in Listing 6.

```
val graphShortestPaths = graphInfinite.pregel((
            Double.PositiveInfinity,
            List[VertexId]()),
            Int.MaxValue,
            EdgeDirection.Out)(
  (nodeId, distance, newDistance) =>
    if (distance._1 < newDistance._1) distance
    else newDistance,
  triplet => {
    if (triplet.dstAttr._1 - triplet.attr > triplet.srcAttr._1){
            Iterator((triplet.dstId,
            (triplet.srcAttr._1 + triplet.attr,
             triplet.srcAttr._2 :+ triplet.dstId)))
    } else {
      Iterator.empty
    }
  },
  (a, b) => if (b._1 > a._1) a else b);
```

Listing 6: Identification of the longest path

Executing the discovery of the longest paths outputs a graph structure with computed attributes for each node, i.e. how to get to the node and what is the path length. The graph can still contain unvisited nodes that are not connected to the graph structure. These are identified by having their path length attribute equal to infinite and are consequently removed from the working set. In the following steps, the longest path is found for each source node, paths connecting less than three nodes are deleted, and all remaining paths are sorted from the longest to the shortest for further analysis and performing contig discovery.

5.3.8. Contig Discovery

Contigs are iteratively discovered by processing the list of all long paths. The processing is executed while there is a long path, extracted by the process described in the previous chapter, left in the set. One of the long paths is chosen according to ordering by primarily the number of nodes in the longest paths. If two paths with the same number of visited nodes exist, the one with higher path length is considered as more superior.

Once a path is being analyzed, all other long paths containing any of the nodes being currently processed are deleted. Therefore, the list of long paths is getting smaller after each iteration. Furthermore, the nodes present in the path are filtered to enable concatenation of resulting contig. Before the actual contig is provided, overlaps between duplets of nodes are found and substituted by relevant sequence string. In the end, the first read is taken as a whole and concatenated with suffixes of all other reads visited on the analyzed path. The contig is then written to disk with a unique identification. The process is repeated until the set of long paths is empty. Output file on disk can, therefore, contain multiple contigs.

5.4. Execution

The genome assembly application is submitted to the Spark cluster via the spark-submit program from development computer. Prior to the submission of the application, it is necessary to place input data, the application binary file and all required libraries on shared storage available to the whole cluster. The Spark submit program has a wide range of configuration that can be applied on submitting an application. Each parameter has a significant effect on application execution. Therefore, much testing was performed to fine tune parameter configuration to suit developed genome assembly application perfectly. Description of used parameters and advised values are following.

Parameter **deploy-mode** defines the location of Spark driver node. Mainly due to latency issues, cluster mode is selected. In cluster mode, the driver is started and runs within the cluster. The address of master node managing the cluster where the application is executed is configured by the **master** parameter. Respectively, **executor-memory** and **executor-cores** specify the maximum amount of RAM and CPU cores to be available per worker node during the application runtime. The total number of executors utilized is set up by **num-executors**. Since Spark driver handles delegation of computation to worker nodes, it needs additional resources for bigger tasks. RAM available to the driver can be increased by **spark.driver.memory**. The driver is also responsible for gathering results of operations on serialized data. If the result is higher than the configuration of **spark.driver.maxResultSize**, the execution of the job is halted. Movement of data between workers and disk I/O operations are very expensive from the time perspective, but fortunately the size of data can be lowered using compression controlled by **spark.rdd.compress** switch.

Partitioning and caching of data turned out to be a crucial aspect of execution. Caching preserves selected data structure in RAM memory of worker nodes for easy and fast access. Only structures accessed frequently should be cached to save RAM memory for other computations. Caching is enabled for objects on code level by using **cache()** function. The number of partitions of created RDDs is defined by **spark.default.parallelism** parameter. RDD is split into chunks and configured level of parallelism is provided on the data level, meaning that all partitions are processed concurrently if enough resources are available. Too many partitions may cause a communication overhead between worker nodes and the driver. Also, the testing indicates that the size of partitions should be at least 10 MB. For larger datasets, calculating parallelism as **executor-cores** * **num-executors** works perfectly fine.

Spark submit program also requires specification of the main class in the binary file by parameter **class** followed by the location of the binary file. Afterward, arguments for application are provided in the following order

```
<input file> <output directory> <minimum overlap>
```

The input file contains reads in FASTA or FASTQ format that are assembled during execution. Output directory is a location where assembled genome is saved. Minimum overlap defines the minimum number of the same base pairs between reads to consider them as overlapping.

The configuration of submitting program was studied from Spark documentation [57], and testing was performed on a Spark standalone cluster later used for testing purposes. An illustration of submit command is provided in Listing 7.

```
spark-submit \
    --deploy-mode cluster \
    --master spark://147.251.253.10:6066 \
    --executor-memory 8G \
    --executor-cores 4 \
    --num-executors 5 \
    --conf spark.driver.memory=16g \
    --conf spark.driver.maxResultSize=16g \
    --conf spark.default.parallelism=4 \
    --conf spark.rdd.compress=true \
    --class "main.scala.run.CCAssemblyJob" \
    /ngs/run/assembly_2.10-1.0.jar \
    /ngs/input/ecoli.fastq /ngs/output/ 90
```

Listing 7: Example of the Spark submit command

The result of genome assembly execution is a set of contigs. The Spark implementation groups found contigs into a single file and saves it into specified output directory. The output file consists of separator lines, lengths of each contig and respective contig sequence as shown in Listing 8.

```
############## CONTIG #################
 Length=185bp
GGAGGAGCACGAAGGTTGGCTAATCCTGGTCGGACATCAGGAGGTTAGTGCAATGGCATAAGCCAGCTT
GACTGCGAGCGTGACGGCGCGAGCAGGTGCGAAAGCAGGTCATAGTGATCCGGTGGTTCTGAATGGAAG
GGCCATCGCTCAACGGATAAAAGGTACTCCGGGGATAACAGGCTGAT

############## CONTIG #################
 Length=172bp
TTGCTGATTACGTGCAGCTTTCCCTTCAGGCGGGATTCATACAGCGGCCAGCCATCCGTCATCCATATC
ACCACGTCAAAGGGTGACAGCAGGCTCATAAGACGCCCCAGCGTCGCCATAGTGCGTTCACCGAATACG
TGCGCAACAACCGTCTTCCGGAGACTGTCATACG
```

Listing 8: Example of genome assembly output consisting of 2 contigs

5.5. Beyond the Assembly

The huge amount of data produced by next-generation sequencing requires an effective and targeted analysis to be implemented. After the reads are provided by a sequencer, some analysis can be done without even constructing the whole sequenced genome. One of such points of interest could be to classify a set of reads into two groups, those which contain positive and negative samples of a disease or DNA subsequence. The simple approach would be to assemble the whole sequence and to search for a particular part of the genome within its content. However, there is a second possibility brought by PN algorithm [58]. Using this algorithm, classification can be done even without gene assembly by analyzing the short reads. The next chapters suggest a parallel implementation of the PN algorithm.

5.5.1. Problem Definition

Two types of reads can be considered as an output of NGS machine. Each read from the initial multiset S can either belong to a negative S^N or a positive S^P group. The reads coming from the negative samples, e.g. cover healthy samples, can be described as

$$S^N = \{\{s_1, s_2, \ldots, s_n\}\}$$

where s_i are particular reads, the reads coming from positive samples, e.g. diseased samples, as

$$S^P = \{\{s_1, s_2, \ldots, s_m\}\}$$

The training set contains samples marked with corresponding negative or positive label. The goal is to compute PN ratio PN_i for each single read in the input multiset S such that

$$PN_i = n_{Pi} / n_{Ni}$$

Where n_{Pi} is the number of occurrences of read i in positive set S^P and n_{Ni} is the number of occurrences of read i in the negative set S^N. A trained classifier consists of distinguished sets of positive and negative reads and a threshold value Θ. If the PN ratio is higher than proposed threshold Θ, then the sample is marked as positive. In the other case, the sample is marked as negative [58].

5.5.2. Distributed Implementation

The proposed distributed implementation of PN algorithm utilizes MapReduce programming style using the map and reduce jobs available in Spark. First of all, negative and positive samples are loaded into Spark RDDs, creating RDDs that represent sets S^N and S^P. In the later step, each read sequence is mapped to value 1, which is later used to represent the number of occurrences of that particular read inside the set. The same is done for positive and negative sample sets.

In the reduce step, the read string is used as a key. All occurrences of the key are counted and new RDDs are created, consisting of key $\mathtt{s_i}$ and value $\mathtt{n_i}$, where $\mathtt{s_i}$ is the read string and $\mathtt{n_i}$ is the number of occurrences within the particular set.

The final stage of the proposed distributed PN algorithm using MapReduce has two functions applied to the dataset. Both positive $\mathtt{S^N}$ and $\mathtt{S^P}$ RDDs are joined and at the same time the occurrence count is mapped to calculate PN ratio for each of the read strings presented in any of the RDDs creating final RDD $\mathtt{S^{PN}}$. Figure 12 shows step by step the execution of actions including temporary results.

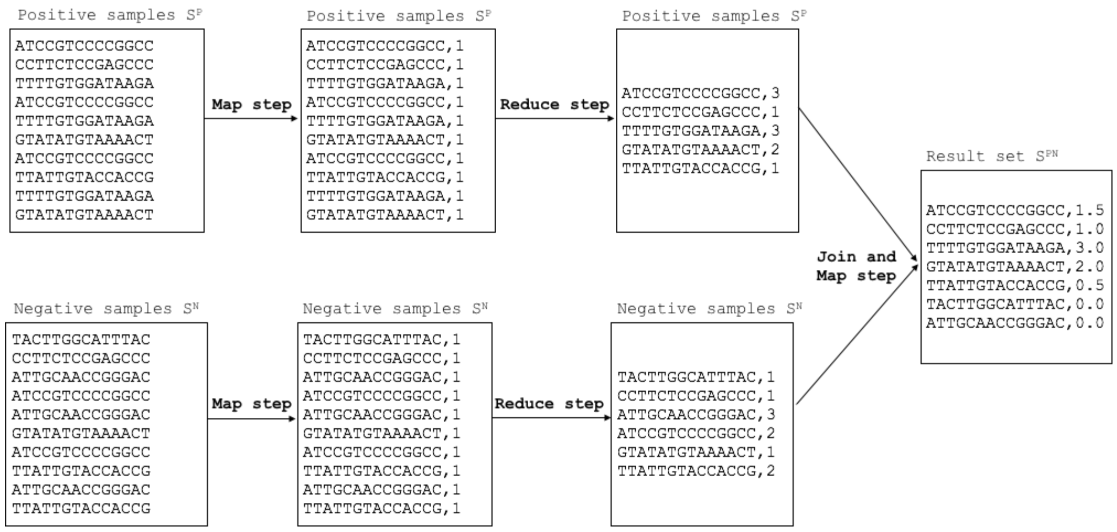

Figure 12 - MapReduce implementation of the PN Algorithm

5.6. Summary

Deliverables of the implementation consist of end-to-end assembly workflow and commissioned test environment. The implementation covers main steps of de novo genome assembly, starting with filtering of the input dataset. Only unique reads are preserved and cached. The next step is to identify overlaps between reads that are further used to build edges between reads. An overlap graph with only connected edges is created. Identification of strongly connected components is used to remove unwanted cycles in the graph. Subgraphs, each consisting of exactly one weakly connected component, are extracted for the discovery of the longest paths. The longest path for each such subgraph is found and contigs are reconstructed based on the visited nodes in the path. Additionally, an example of possible genomic data analysis represented by PN algorithm is developed in a distributed manner.

On the one hand there are the strengths of the solution, which are a generally parallelized approach to tasks and usage of standard and verified libraries where applicable. On the other hand, unavailability of heuristics such as pruning of spurs or the usage of exact matching can be considered as solution weaknesses.

The expectations before executing tests are that proposed distributed solution will perform worse than the referential application on the same amount of computation resources, due to Spark cluster overhead and omit of some heuristics. Though, it is expected that the scalability on a growing number of computation resources will be clearly visible. An identity function implemented as exact matching might produce shorter contigs, resulting in a lower N50 quality value. These are only preliminary expectations, which can be denied or confirmed by test results.

6. Assembler Evaluation

The subsequent chapters describe the comparison testing between a referential sequential assembler and the distributed assembler developed per description in Chapter 5. Referential sequence assembler will be represented by SAGE, described in detail in Chapter 4.4.6. SAGE is selected because it is also based on the construction of an overlap graph. Also, it implements many of graph optimization techniques currently available for overlap graphs. This makes it a very powerful representative of overlap graph based de novo assemblers.

Firstly the environment setup and test details are provided, following by the definition of test scenarios and metrics. There are 2 separate groups of performed tests, one focusing on the performance and scalability, the second one on sequence assembly quality. Additionally, the distribution of tasks across available resources is examined as well to identify the level of parallelism. In the end, test results are analyzed and strong points of both compared solutions are identified. The goal is not to outperform SAGE, but rather to indicate the correct routing of Spark based assembler and fulfillment of requirements in the scope stated in Chapter 5.1.

6.1. Environment Description

The test environment for SAGE was a CentOS based server scaled from 1 CPU with 4GB RAM up to 8 CPUs with 32GB RAM. Runs comparing the performance with the distributed solution were executed with the top configuration.

The distributed implementation ran on a standalone Spark cluster built specifically for the purpose of testing and not used by any other users. It consisted of one master node and 5 additional worker nodes. Each node had the same configuration, 4 CPUs and 16GB RAM. The shared storage was implemented as a shared file system located on the master node. Limiting of resources during scalability testing was done by setting maximum available resources via configuration during submitting jobs to Spark cluster.

The testing input data in FASTQ format was obtained from Assemblathon challenges [59], it is an ecoli bacteria reads set with total 6,355,877 reads. These reads were cleaned before the test execution, each read had length 101 base pairs. During the execution, the minimum overlap length for both compared solutions was set to 90. To identify the behavior of algorithms with different size of input data, FASTQ file was truncated to 100,000 reads and then gradually incremented by 100,000 reads up to the total length. The file size in a non-compressed format was 766MB.

6.2. Performance

During performance testing, scalability of both solutions is evaluated. The scaling is considered from the data perspective as well as increasing computation resource. Both viewpoints provide outlook how solutions behave in very data and resource complex applications in genome assembly field. The main purpose is to assess, whether the Spark based assembler is heading the correct way to provide a computationally scalable solution while making no trade-offs in quality. To support this, a series of figures and comparisons with SAGE assembler are set up and evaluated.

6.2.1. Data Growth

The test consisted of providing both solutions with equal resources for computation and iteratively executing the run with changing size of the input dataset. At the start, 100,000 reads were used. Each following run, additional 100,000 reads were added to the dataset, until 5 million reads input dataset was reached. Every run was repeated 10 times to approximate the results. The main observed metric for this particular test was runtime. Available computation resources were 8 CPU cores and 32 GB of RAM.

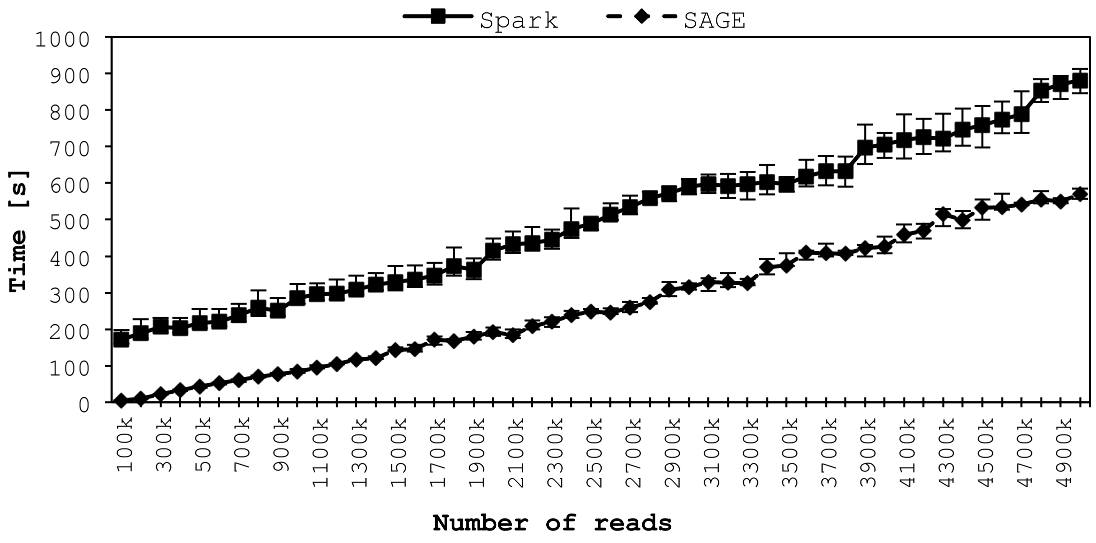

Figure 13 - Data growth test results

Figure 13 provides the number of reads in the input dataset on the horizontal axis and time needed to assemble the reads on the vertical axis. It is clearly visible that SAGE has a faster performance throughout the growing dataset. The solution based on Spark seems to perform especially slower than SAGE on small datasets up to 900 thousand reads, being more than 3 times slower. The slowdown ratio gets lower with growing dataset and stabilizes between 1,5 and 1,9 slowdown rate on datasets with more that 1,5 million reads. The figure also shows that runtimes for the same dataset differ per runs for both solutions. Whereas SAGE shows minimal differences for small datasets and stays under 5% for datasets over 1,5 million reads, the Spark based assembler runs with a higher deviation of assembly times reaching up to 10%.

There are a couple of reasons behind the gap between the two implementations. The first one is that SAGE uses more graph optimizations and not all of them were implemented into the Spark based assembler. An example of such graph optimization technique missing in Spark implementation is the removal of dead ends. Further improvements and optimizations may shrink the runtime gap or completely remove it. Another reason for gap occurrence is the architecture difference between the two solutions. SAGE, based on C++, runs directly on a host system as a process with no middleware layer. On the other hand, to run a Spark application, it is necessary to have a Java Virtual Machine (JVM) started on all cluster nodes. Spark nodes are then started within these JVM and only after that application can be deployed to Spark cluster. This provides an overhead in resource consumption and communication between nodes, but thanks to this a high level of scalability is available.

6.2.2. Scalability

The focus of hardware scalability test was to investigate and compare runtime of both assembly solutions while changing the number of available computation resources. The entry tests were performed with the configuration of 1 CPU core with 4 GB of RAM. For testing of SAGE, it was necessary to alter the whole server configuration via the cloud self-service portal after each test execution. It means that the server configuration was changing from 1 CPU core with 4 GB of RAM up to 8 CPU cores with 32 GB RAM, which was the maximum available server configuration at the time of test execution. The resource unit added after each test was 1 CPU core with 4 GB of RAM. Performing the scaling test on Spark cluster was rather simpler since the Spark application submit command provides a configuration for resources to be used during computation. The configuration of parameters executor-memory and executor-cores was gradually increased to model growing resources. The starting configuration was 1 CPU core with 4 GB of RAM, and by the same unit size, it was increased up to 16 CPU cores with 64 GB of RAM. The resources could grow even higher by adding more nodes, but stated configuration was sufficient to investigate scalability possibilities. Each test was executed 10 times to approximate the results. The input dataset used for all tests consisted of 1 million reads.

The scalability test discovers that SAGE has a very limited scalability potential. It is able to run only on one server and can utilize only one CPU core. Adding more CPU cores does not result in any significant improvements. Also, SAGE uses only an amount of RAM proportional to the amount of input data. Once all input data can be stored in RAM, no advance in runtime is visible. Although Spark implementation still preserves a runtime gap after SAGE, it proves that it is scalable. When running the Spark assembler on 1 or 2 CPU with up to 8 GB of RAM, the speed of assembly is extremely poor. A huge decrease in runtime is visible after adding third CPU core and increasing RAM to 12 GB. This boosts up performance and starts a gradual decline in the time needed for assembly. From this point, the scaling results in a decrease of runtime until it is at half of its origin. In other words, the algorithm works two times faster on 10 CPU cores with 40 GB RAM than on 3 CPU cores with 12 GB RAM. After this point, no significant decline was recorded, and runtimes oscillate around 3 minutes.

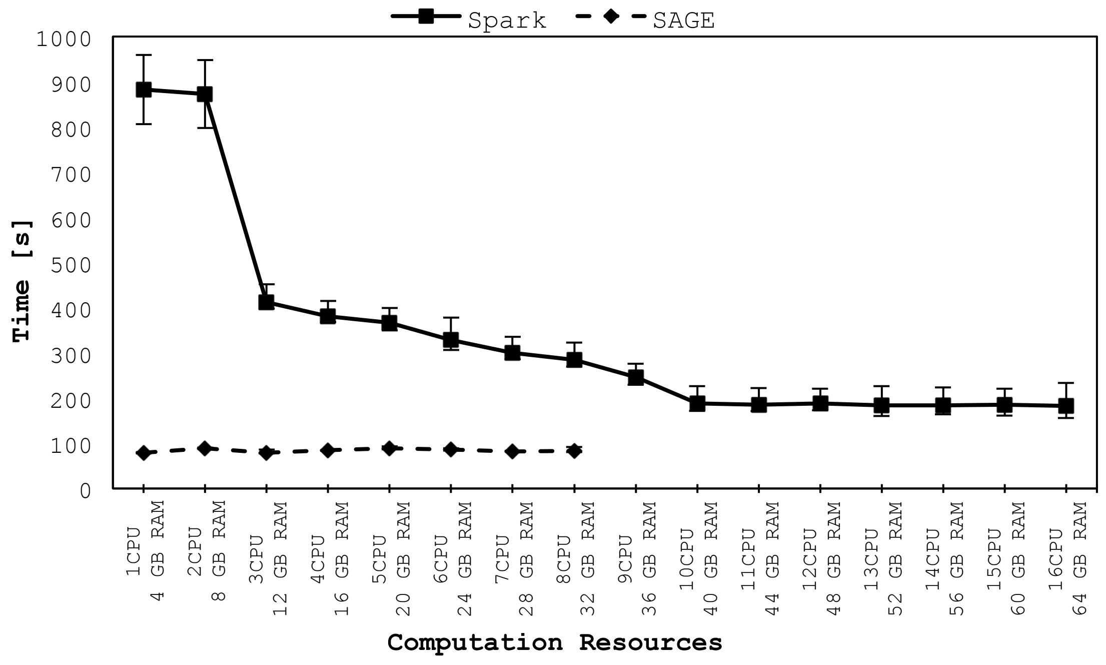

Figure 14 - Resource scalability

The results of resource scaling test can be divided into three segments from the Spark assembler perspective. The first segment is during the tests with 1 and 2 CPU cores. These assemblies suffer from insufficient resources and therefore a lot of cache clearing and usage of persistent storage for temporary data. Since file access times to move data from and to persistent storage is higher than the operations within RAM, total runtimes are heavily impacted. This issue is overcome by adding sufficient resources, which in this test are 3 CPU cores and 12 GB of RAM. From this point, the second segment of results is visible, characterized by efficient scaling. From 3 CPU cores and 12 GB of RAM up to 10 CPU cores and 40 GB of RAM, adding resources results in the dropping of processing time. Effective scaling happens during this segment, where resources are utilized efficiently to provide the best possible runtime. Afterward, over-allocation of resources happens. The same runtime characterizes the over-allocation segment despite increased computation power. Since all resources are already allocated efficiently, adding more does not improve runtime and results only in unused computing power.

6.3. Quality

The quality of genome assembly was measured by the standard technique N50, which is described in detail in Chapter 4.6. N50 quality assessment depends on a contig length. The contig length can be controlled by minimum overlap threshold input parameter of both assemblers. Lower values of the parameter give the assembler more flexibility during overlap identification since lower thresholds generally mean more options how to organize reads into the overlap graph. Input dataset for the quality test consisted of 1 million reads and the minimum overlap parameter was iterated from 66 up to 95.

Figure 15 represents an evaluation of N50 assembly quality, higher values are considered of better quality. As it is visible, quality strongly depends on the selection of the proper value for the minimum overlap parameter.

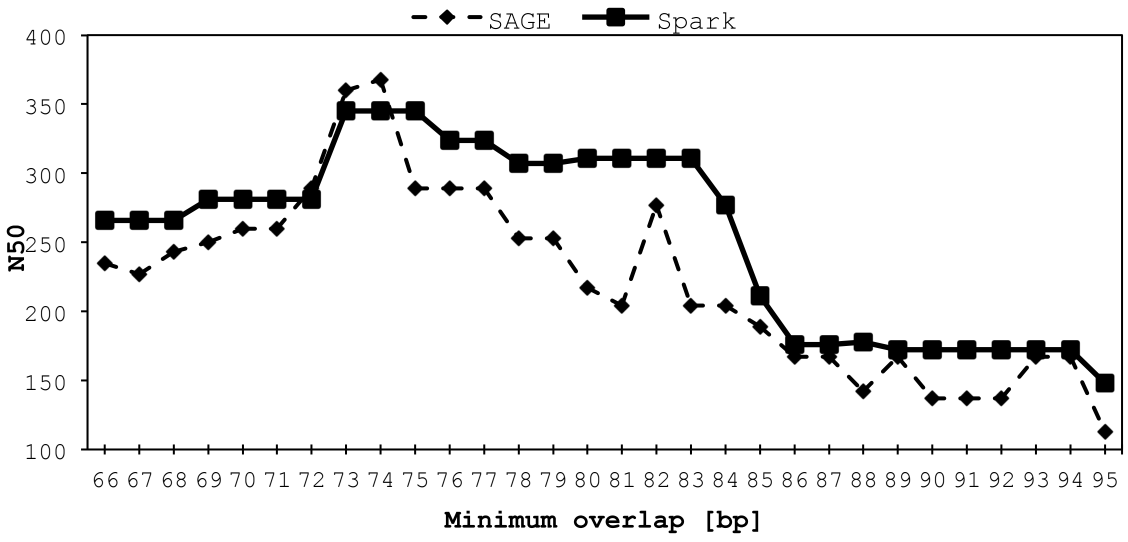

Figure 15 - N50 quality assessment

SAGE managed to obtain the highest N50 value in the investigated parameter interval. Spark generally assembles output of a better quality and is less turbulent during the changes of minimum overlap parameter. Also, the maximum N50 value of Spark implementation is exactly for the same minimum overlap value as in SAGE's case. Spark based assembler achieved excellent results and well-assembled contigs from N50 quality perspective.

6.4. Parallelization Properties

Parallelism assessment of Spark based assembler is done via Spark's history event user interface [60]. The investigation provides information at all execution levels, on all jobs level, in the specifically selected jobs, and on the stage level.

Overall execution of jobs that together represent application run is provided in Figure 16. The main application functions such as pre-processing of reads, overlap discovery, graph optimization, identification of the longest paths and assembly of contigs are identified and highlighted. It is also visible that five executors are added at the beginning of runtime and utilized throughout the whole execution. Parallelism in Spark can be represented on job, stage and data level. Figure 16 presents parallelism property on the job level and clearly shows that concurrent job execution occurs.

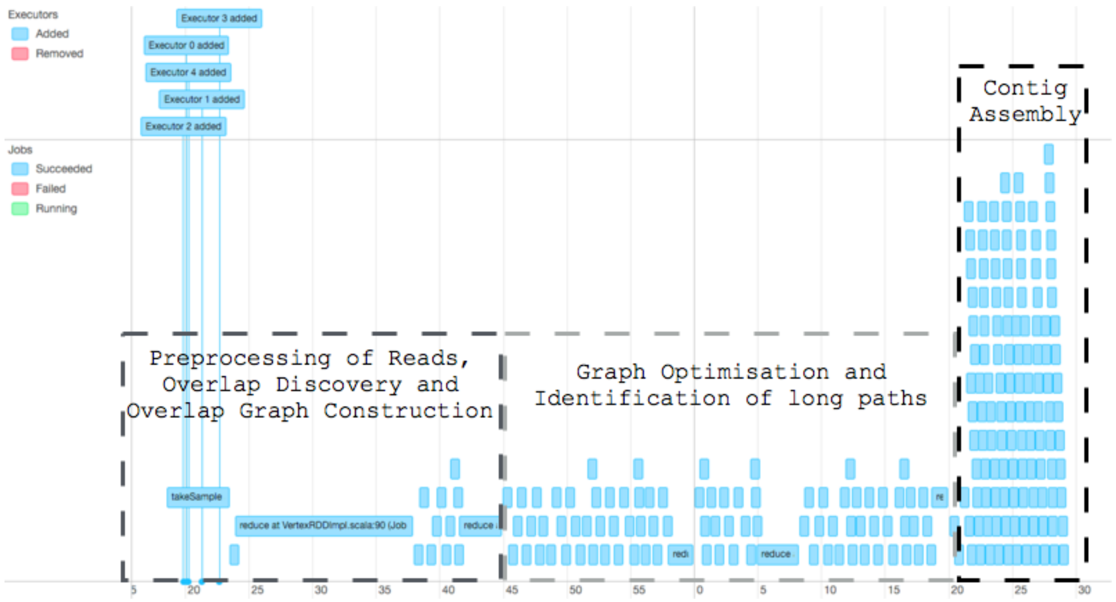

Figure 16 - Overview of all executed jobs

Job length and number of jobs in three highlighted parts of application differ. The reasons behind this are data operations executed on the dataset. During data loading and all to all read overlap discovery, the application works with a relatively huge amount of data in one structure that is processed within one job. The parallelism in such jobs happens on stage and data levels. Further in the algorithm, data is split into smaller chunks due to their attachment to various connected components. Such granulated data is then processed via multiple jobs at once since graph optimizations or the longest path discoveries run in each weakly connected component separately. Jobs performing tasks on different connected components can run concurrently and are rather quick, resulting in short jobs in the second part of the application run. The last part of application responsible for contig assembly based on provided paths looks up read strings based on their identification and concatenates them into final contigs. Jobs executing this part of the application are highly parallelized, atomic and fast.

The last job in Figure 16 in the bottom right represents saving assembled sequence to file.

To gain more information on long running jobs, a breakdown to the stage level is essential. Stages of long running jobs such as the one provided in Figure 17 consist of particular transformations and actions, for example map, reduce, collect or repartition. Map tasks can be executed in parallel since they are applied per partition and do not depend on each other. On the other hand, repartition task needs all map tasks to be finished due to its dependence on the result. Only after that, repartition is performed. Following repartition, the dataset is processed to remove duplicates leaving only distinct values in the result. The distinct action also waits until all preceding stages are completed. The map, repartition and distinct actions described refer to the part of the application, where all to all read comparison is performed to identify all overlaps and create unique edges out of the results. Repartition is applied to lower the increased number of partitions obtained while joining the overlap sets.

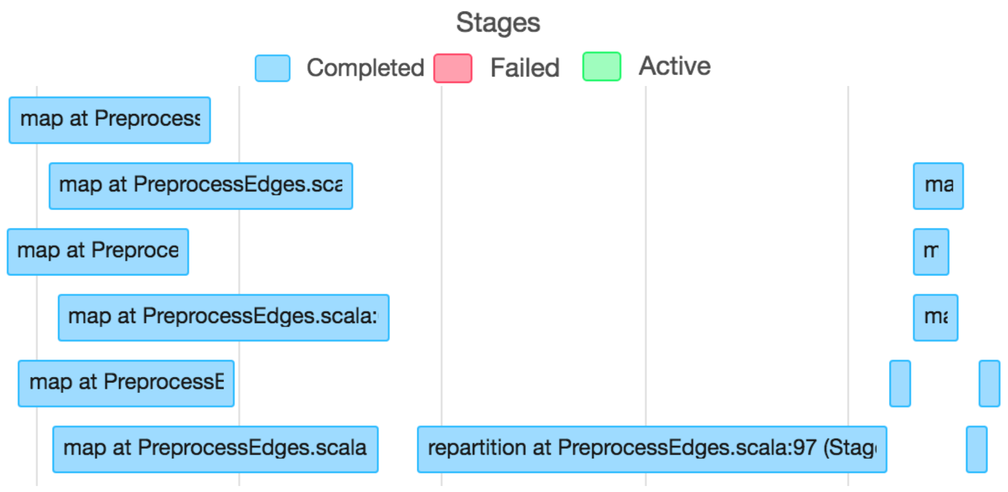
Stages
Completed
Failed
Active
map at Preprocess
map at PreprocessEdges.sca
ma
map at Preproce
m
map at PreprocessEdges.scala:
ma
map at PreprocessE
map at PreprocessEdges.scala
repartition at PreprocessEdges.scala:97 (Stag

Figure 17 - Job detail

The details for each stage in Figure 18 show how partitions of data are processed within its execution. Parallelism on data partition level is the lowest visible concurrency available on Spark cluster. Partition level parallelism means that same stage is applied to data present in different partitions on worker nodes.

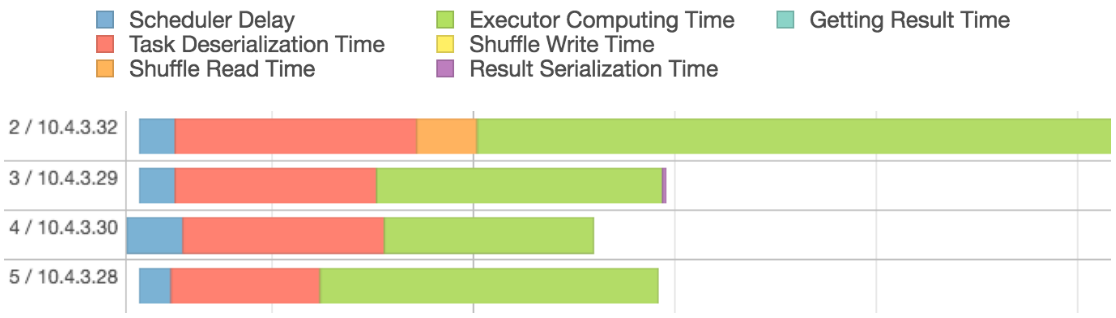

Figure 18 - Stage detail

6.5. Result Discussion

The performance and quality tests were executed successfully. It was possible to generate a sufficient amount of runtime data for the comparison of solutions. During the test execution, a few additional metrics were discovered and assessed, which are not directly connected to sequence assembly itself, but are generally important for product evaluation. These metrics are ease of use, configuration possibilities, deployment and portability properties.

During the performance testing, a runtime gap between SAGE and the Spark implementation in favor of the first solution was revealed. SAGE implementation gains its advance due to more heuristics applied during the graph optimization. However, it is important to note, that overall complexity of both solutions is similar and therefore the gap can be closed by adding more heuristics to the Spark assembler. Moreover, Spark introduces additional computation layers for the purposes of resiliency and vertical scalability. These layers consist of the big data computing engine, communication between Spark nodes and Java virtual machines. Despite the overhead and runtime gap, no significant issues of the Spark implementation were identified when the input dataset is growing.

The tests also indicate, that the Spark implementation of genome assembler can efficiently lay out over available resources. It scales with growing resources until the optimal allocation is reached. Available Spark cluster resources can, therefore, be utilized evenly and solution adapts well to changing conditions.

When analyzing the assembly quality with N50 criteria, the tests show no impact of parallelization to the overall quality. SAGE and Spark assemblers use different approaches to assembly, but the quality results on different datasets and with different input parameters have the same trend for both solutions. Generally, Spark constructs longer contigs, but the maximum N50 value measured during testing was produced by SAGE. The tests reveal that the quality of assembly using the Spark implementation is comparable to the referential assembler's quality. This is even emphasized by the maximum and minimum quality points achieved with the same settings of the two solutions.

The parallelism of Spark solution is on a very high level and is visible throughout all monitored layers, from jobs down to data location. The distribution of work over available executors does happen as well, providing horizontal scalability. Vertical scalability on Spark nodes is also essential and results in improved performance.

The comparison of assemblers from deployment perspective is difficult because the installation of environment differs. SAGE requires a Linux-based system with numerous installed libraries to be prepared. On the other hand, Spark assembler requires ready to use Spark instance or cluster installed and properly configured. The advance of the Spark assembler is that it can be deployed to any Spark cluster available, either public or built on-premise. Also, such Spark clusters can be reused for multiple purposes, not only for genome assembly. Additional positives of using Spark based assembler are high availability and resilience to node failures. That means genome assembly on Spark cluster will not be negatively affected in case of node failure or outage.

Spark based assembler also excels in the field of easy usage. It gives a very practical insight into application runtime via monitoring tool of the cluster with its graphical user interface. Location of large files in very common HDFS storage can be retrieved only by Spark based assembler, which is a significant advantage. Spark applications can also be remotely executed on a cluster from development computer using submit actions, which is a feature not available in SAGE. SAGE must be executed directly from the machine with the data and computation resources. Both assemblers provide easily readable output ready for further analysis.

From the configuration perspective, both assemblers expect the same parameters on the input. Spark excels in configuration on job level, making it possible to set resource handling based on specific dataset being processed.

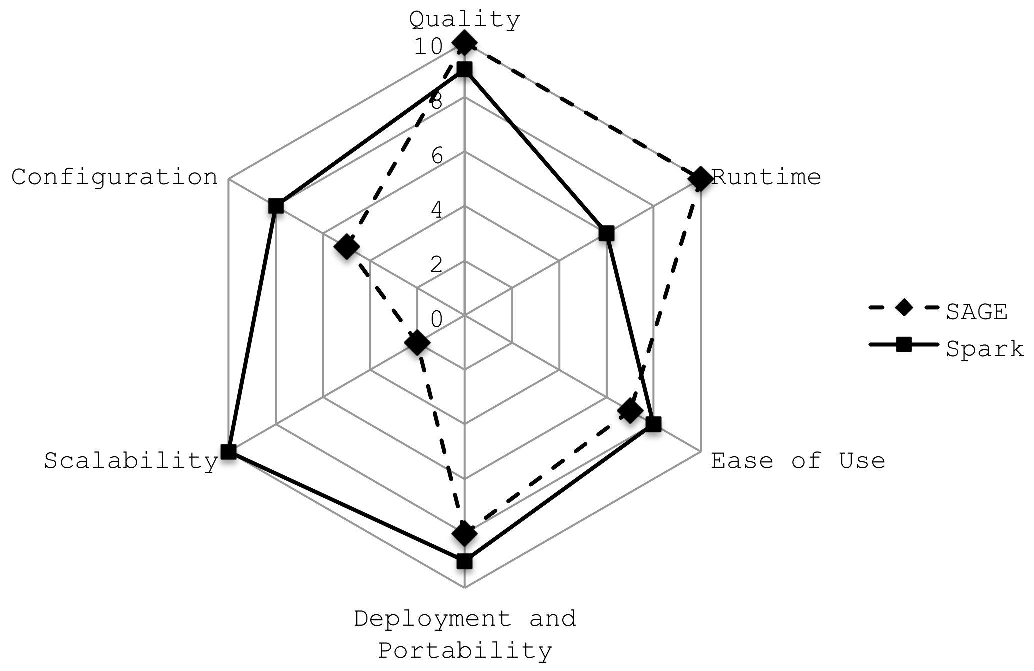

Figure 19 – Overall results illustration

In order to sum up the overall results taking into account all investigated metrics, Figure 19 is extracted. The review is based on the individual test results and experience with both assembly solutions. A brief analysis of the figure shows SAGE outperforming the Spark implementation in runtime and achieving slightly better N50 quality values in certain cases. Spark assembler accomplishes much better rating in scalability and configuration perspective. It also has improved portability and ease of use, when compared with SAGE.

7. Future Work

The evaluation of distributed Spark implementation for de novo genome assembler shows that the chosen direction and idea have a potential to bring cloud computing closer to the genome assembly and analysis. Though the provided implementation is not yet ready for genome assembly commercial use, it does provide a concept that can be further enhanced.

Further improvements shall be mainly focused on closing the runtime gap between traditional assemblers and the Spark implementation. This can be achieved by further optimizing the overlap graph. Suitable candidates for implementation are reducing of bubble structures and removing dead ends that will result in faster traversing of a more compact graph. Other used heuristics can also be improved to perform better, such as a more advanced implementation of transitive edge reduction. Quality assessment of base pair level was put out of scope for this implementation, but for the Spark solution to be competitive in the future, it should be added to consider read quality during assembly. Today's DNA sequencers provide reads output of a high quality, but there is still a risk of specific read being sequenced with an error.

As for any application focused on great performance, it is necessary to debug and profile it using available tools. Despite studying vast information about the correct distributed design and Spark's functionality, current distributed implementation should be assessed by experts on distributed data processing. This may result in further improvements in the performance. Also, there are multiple tools on the market for application profiling and application performance monitoring, which could be applied to find weak spots requiring further design and development modifications.

From the usage perspective, the developed application can be extended to support multiple input and output formats. The most common were chosen and implemented, but there are many other that could be enabled to widen the supportability of multiple DNA sequencers.

What is more, the applicability of big data computing engines on the cloud should not be limited only to sequence assembly. Multiple other problems can benefit from distributed computation. Classification of reads is one of the possible applications that was provided in 5.5. However, there are tons of other opportunities waiting for big data challenges, starting from searching of specific subsequences in large genomes up to clustering of organisms based on their DNA structure and similarity.

8. Conclusion

Next-generation sequencing is changing the world around us. The biological footprint of the technology is enormous, diagnostic medicine based on DNA is about to enter our daily lives. The sky is the limit when it comes to next-generation sequencing, due to applicability in fields such as agriculture, forensics, studying of organisms' evolution and much more. Innovation happening in the field of genomics is enormous, strongly impacting the rapid development of sequencing and analysis technologies. The bioinformatics sector must keep pace with these developments in order to support the constantly growing requirements. This thesis shows that using big data technologies can be the way forward, putting de novo sequence assembly as an example.

A truly distributed, scalable and cloud-ready de novo sequence assembler was developed. The chosen engine for big data processing and cluster setup was Apache Spark. This choice was a perfect fit for this purpose since additional Spark libraries for graph processing efficiently utilized distributed architecture. Scalability was provided out of the box and development was pleasant thanks to interactive shell and monitoring interface. Spark also seamlessly integrates with existing big data solutions including, but not limited to Hadoop. Similarly to NGS, Apache Spark is an innovative service improving big data solutions in terms of in-memory capabilities, resiliency, multi-tenancy and development speed.

A de novo sequence assembly application designed on top of the Spark framework was built with parallelism, performance, quality and innovation in mind. A lot of available solutions and approaches were analyzed to choose the right strategy. The final decision was to build a solution based on overlap graph, utilizing heuristics based on graph theory, especially strongly and weakly connected components to parse the overlap graph. This turned out to be a very good decision that provided fulfillment of the requirements. Algorithmically, the distributed solution remains a proof of concept. Sequence assembly is an intricate task whose complete resolution lies outside the scope of this thesis.

Spark cluster setup was one of the most important work items performed during workflow preparation. A dedicated cluster on standalone servers was built and configured for maximum performance for developed application. The application was deployed to the Spark cluster and deeply analyzed during functional and performance testing. All test results were evaluated and compared with traditional de novo assembler based on the OLC algorithm. The application developed on the top of Spark framework proved to be competitive, but still having points in which it could be improved.

The gained knowledge during the design, implementation and testing is an enormous benefit for myself. I have extended my knowledge of distributed computation engines and learned how to build, configure and execute big data processing jobs. Also, I studied under the hood properties of Spark engine, which helped me better understand distributed systems and develop a better application for that purpose. My knowledge improved not only in the field of big data, but I also made a huge step in learning about next-generation sequencing and how it works from the biological perspective. Deeply studying the sequence assembly problem and available solutions helped me understand the challenges bioinformatics and geneticists face when it comes to DNA assembly.

The usage and importance of next-generation sequencing are rising and there is no sign of this trend to be changed. Cloud computing is bringing NGS-related processes into the future, and Spark sequence assembly solution developed within this thesis fits into this landscape. By combining high-throughput sequencing technologies with cheap cloud computing resources, maximum potential can be achieved. This way NGS solutions and DNA analysis could be really available to everyone.

References

[1] MIESCHER-RÜSCH, Friedrich. Ueber die chemische Zusammensetzung der Eiterzellen (On the chemical composition of pus cells) , 4 : 441–460. 1871.

[2] WATSON, J. D.; CRICK, F. H. C. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. Nature, 1953, 171.737-738: 3.

[3] SANGER, Fred; COULSON, Alan R. A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase. Journal of molecular biology, 1975, 94.3: 441-448.

[4] SANGER, Frederick; NICKLEN, Steven; COULSON, Alan R. DNA sequencing with chain-terminating inhibitors. Proceedings of the National Academy of Sciences, 1977, 74.12: 5463-5467.

[5] STADEN, Rodger. A strategy of DNA sequencing employing computer programs. Nucleic acids research, 1979, 6.7: 2601-2610.

[6] ADAMS, Mark D., et al. Complementary DNA sequencing: expressed sequence tags and human genome project. Science, 1991, 252.5013: 1651-1656.

[7] TRIPP, Simon; GRUEBER, Martin. Economic Impact of the Human Genome Project. Battelle Memorial Institute, 2011, 4-7.

[8] JANITZ, Michal (ed.). Next-generation genome sequencing: towards personalized medicine. John Wiley & Sons, 2011, 1.3: 7-8.

[9] Technology: The $1,999 genome [online]. Nature Publishing Group. [visited on 15.5.2016]. Available at: http://www.nature.com/news/technology-the-1-000-genome-1.14901

[10] NAGARAJAN, Niranjan; POP, Mihai. Sequence assembly demystified. Nature Reviews Genetics, 2013, 14.3: 157-167.

[11] WATSON, James D., et al. Molecular structure of nucleic acids. Nature, 1953, 171.4356: 737-738.

[12] METZKER, Michael L. Emerging technologies in DNA sequencing. Genome research, 2005, 15.12: 1767-1776.

[13] METZKER, Michael L. Sequencing technologies—the next generation. Nature reviews genetics, 2010, 11.1: 31-46.

[14] POP, Mihai; SALZBERG, Steven L. Bioinformatics challenges of new sequencing technology. Trends in Genetics, 2008, 24.3: 142-149.

[15] COCK, Peter JA, et al. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. Nucleic acids research, 2010, 38.6: 1767-1771.

[16] ZIKOPOULOS, Paul, et al. Understanding big data: Analytics for enterprise class hadoop and streaming data. McGraw-Hill Osborne Media, 2011.

[17] KWON, Taesoo, et al. Next-generation sequencing data analysis on cloud computing. Genes & Genomics, 2015, 37.6: 489-501.

[18] 1000 Genomes Project and AWS [online]. Amazon.com, Inc. [visited on 5.2.2016]. Available at: https://aws.amazon.com/1000genomes/

[19] Google Genomics Docs [online]. Google Inc. [visited on 5.2.2016]. Available at: https://cloud.google.com/genomics/what-is-google-genomics

[20] GenomeNext Launches Deterministic, Turn-Key, SaaS (Software-as-a-Service) Genomics Solution Addressing Computational Bottlenecks in DNA Sequencing Analysis [online]. GenomeNext, LLC. [visited on 7.2.2016]. Available at: http://www.genomenext.com/press_post/genomenext-launches-deterministic-turn-key-saas-software-as-a-service-genomics-solution-addressing-computational-bottlenecks-in-dna-sequencing-analysis/

[21] SaaS providers [online]. OMICtools. [visited on 7.2.2016]. Available at: http://omictools.com/saas-providers-c415-p1.html

[22] DEAN, Jeffrey; GHEMAWAT, Sanjay. MapReduce: simplified data processing on large clusters. Communications of the ACM, 2008, 51.1: 107-113.

[23] BORTHAKUR, Dhruba. The hadoop distributed file system: Architecture and design. Hadoop Project Website, 2007, 11.2007: 21.

[24] NIEMENMAA, Matti, et al. Hadoop-BAM: directly manipulating next generation sequencing data in the cloud. Bioinformatics, 2012, 28.6: 876-877.

[25] Big Data Processing with Apache Spark [online]. InfoQ. [visited on 9.1.2016]. Available at: http://www.infoq.com/articles/apache-spark-introduction

[26] ARMBRUST, Michael, et al. Scaling spark in the real world: performance and usability. Proceedings of the VLDB Endowment, 2015, 8.12: 1840-1843.

[27] Spark Programming Guide [online]. The Apache Software Foundation. [visited on 14.4.2016]. Available at: http://spark.apache.org/docs/latest/programming-guide.html

[28] GONZALEZ, Joseph E., et al. Graphx: Graph processing in a distributed dataflow framework. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 2014. p. 599-613.

[29] WIEWIÓRKA, Marek S., et al. SparkSeq: fast, scalable, cloud-ready tool for the interactive genomic data analysis with nucleotide precision. Bioinformatics, 2014, btu343.

[30] MASSIE, Matt, et al. Adam: Genomics formats and processing patterns for cloud s cale computing. University of California, Berkeley Technical Report, No. UCB/EECS-2013, 2013, 207.

[31] ZAHARIA, Matei, 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. USENIX Association, 2012. p. 2-2.

[32] Cluster Mode Overview [online]. The Apache Software Foundation. [visited on 3.5.2016]. Available at: http://spark.apache.org/docs/latest/cluster-overview.html

[33] Apache Spark: A Look under the Hood [online]. Sigmoid. [visited on 27.4.2016]. Available at: https://www.sigmoid.com/apache-spark-internals/

[34] MILLER, Jason R.; KOREN, Sergey; SUTTON, Granger. Assembly algorithms for next-generation sequencing data. Genomics, 2010, 95.6: 315-327.

[35] POP, Mihai; SALZBERG, Steven L.; SHUMWAY, Martin. Genome sequence assembly: Algorithms and issues. Computer, 2002, 35.7: 47-54.

[36] KARGER, David; MOTWANI, Rajeev; RAMKUMAR, G. D. S. On approximating the longest path in a graph. Algorithmica, 1997, 18.1: 82-98.

[37] EPPSTEIN, David. Finding the k shortest paths. SIAM Journal on computing, 1998, 28.2: 652-673.

[38] SIMPSON, Jared T.; DURBIN, Richard. Efficient construction of an assembly string graph using the FM-index. Bioinformatics, 2010, 26.12: i367-i373.

[39] LI, Zhenyu, et al. Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. Briefings in functional genomics, 2011, elr035.

[40] OZSOLAK, Fatih; MILOS, Patrice M. RNA sequencing: advances, challenges and opportunities. Nature reviews genetics, 2011, 12.2: 87-98.

[41] Luo et al.: SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. GigaScience 2012 1:18.

[42] LI, Dinghua, et al. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. Bioinformatics, 2015, btv033.

[43] BOWE, Alexander, et al. Succinct de Bruijn graphs. In: Algorithms in Bioinformatics. Springer Berlin Heidelberg, 2012. p. 225-235.

[44] ZERBINO, Daniel R.; BIRNEY, Ewan. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome research, 2008, 18.5: 821-829.

[45] CHEVREUX, Bastien, et al. Using the miraEST assembler for reliable and automated mRNA transcript assembly and SNP detection in sequenced ESTs.Genome research, 2004, 14.6: 1147-1159.

[46] SIMPSON, Jared T., et al. ABySS: a parallel assembler for short read sequence data. Genome research, 2009, 19.6: 1117-1123.

[47] ILIE, Lucian, et al. SAGE: string-overlap assembly of genomes. BMC bioinformatics, 2014, 15.1: 1.

[48] PHILLIPPY, Adam M., et al. Genome assembly forensics: finding the elusive mis-assembly. Genome Biol, 2008, 9.3: R55.

[49] SALZBERG, Steven L., et al. GAGE: A critical evaluation of genome assemblies and assembly algorithms. Genome research, 2012, 22.3: 557-567.

[50] Apache Spark [online]. The Apache Software Foundation. [visited on 9.1.2016]. Available at: http://spark.apache.org/

[51] A Scalable language [online]. EPFL and Lightbend, Inc. [visited on 9.1.2016]. Available at: http://www.scala-lang.org/what-is-scala.html

[52] Making Development and enjoyable experience [online]. JetBrains s.r.o. [visited on 9.1.2016]. Available at: https://www.jetbrains.com/idea/features/

[53] The interactive build tool [online]. Typesage Inc. [visited on 9.1.2016]. Available at: http://www.scala-sbt.org/

[54] MetaCentrum – Virtual Organization [online]. MetaCentrum VO. [visited on 2.5.2016]. Available at: https://metavo.metacentrum.cz/en/index.html

[55] Kryo [online]. GitHub, Inc. [visited on 5.4.2016]. Available at: https://github.com/EsotericSoftware/kryo

[56] HAHNE, Ellen L. Round-robin scheduling for max-min fairness in data networks. Selected areas in Communications, IEEE Journal on, 1991, 9.7: 1024-1039.

[57] Spark Configuration [online]. The Apache Software Foundation. [visited on 9.1.2016]. Available at: http://spark.apache.org/docs/latest/configuration.html

[58] JALOVEC, Karel; Železný, Filip. Binary classification of metagenomic samples using discriminative DNA superstrings. CTU in Prague FEE Dept. of Computer Science and Engineering, 2014.

[59] BRADNAM, Keith R., et al. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. BioMed Central Ltd., 2013.

[60] Monitoring and Instrumentation [online]. The Apache Software Foundation. [visited on 14.4.2016]. Available at: http://spark.apache.org/docs/latest/monitoring.html

Appendix A

CD Content

Attached CD contains:

- **Assembler Source Code** - /Source code/Assembly
- **PN algorithm Source Code** - /Source code/PNRatio
- **Example input and output dataset** - /Data
- **Thesis in PDF** - /Thesis