SparkGA

A Spark Framework for Cost Effective, Fast and Accurate DNA Analysis at Scale

Mushtaq, Hamid; Liu, Frank; Costa, Carlos; Liu, Gang; Hofstee, Peter; Al-Ars, Zaid

DOI
10.1145/3107411.3107438

Publication date
2017

Document Version
Final published version

Published in
ACM-BCB '17 Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology,and Health Informatics

SparkGA: A Spark Framework for Cost Effective, Fast and Accurate DNA Analysis at Scale

Hamid Mushtaq
TU Delft
h.mushtaq@tudelft.nl

Frank Liu
IBM
frankliu@us.ibm.com

Carlos Costa
IBM
chcost@us.ibm.com

Gang Liu
IBM
gliu@us.ibm.com

Peter Hofstee
IBM
hofstee@us.ibm.com

Zaid Al-Ars
TU Delft
z.al-ars@tudelft.nl

ABSTRACT

In recent years, the cost of NGS (Next Generation Sequencing) technology has dramatically reduced, making it a viable method for diagnosing genetic diseases. The large amount of data generated by NGS technology, usually in the order of hundreds of gigabytes per experiment, have to be analyzed quickly to generate meaningful variant results. The GATK best practices pipeline from the Broad Institute is one of the most popular computational pipelines for DNA analysis. Many components of the GATK pipeline are not very parallelizable though. In this paper, we present SparkGA, a parallel implementation of a DNA analysis pipeline based on the big data Apache Spark framework. This implementation is highly scalable and capable of parallelizing computation by utilizing data-level parallelism as well as load balancing techniques. In order to reduce the analysis cost, SparkGA can run on nodes with as little memory as 16GB. For whole genome sequencing experiments, we show that the runtime can be reduced to about 1.5 hours on a 20-node cluster with an accuracy of up to 99.9981%. Moreover, SparkGA is about 71% faster than other state-of-the-art solutions while also being more accurate. The source code of SparkGA is publicly available at https://github.com/HamidMushtaq/SparkGA1.git.

ACM Reference format:
Hamid Mushtaq, Frank Liu, Carlos Costa, Gang Liu, Peter Hofstee, and Zaid Al-Ars. 2017. SparkGA: A Spark Framework for Cost Effective, Fast and Accurate DNA Analysis at Scale. In *Proceedings of ACM-BCB '17, Boston, MA, USA, August 20-23, 2017,* 10 pages.
https://doi.org/10.1145/3107411.3107438

1 INTRODUCTION

The fast reduction in the cost of DNA sequencing is making it an accessible method to use in clinics for diagnosing genetic diseases. However, the computational cost of DNA analysis has become the bottleneck in deploying this technique at scale. DNA analysis is needed to process the generated data to identify mutations in the DNA that could indicate specific susceptibilities to certain diseases. One of the most popular methods for DNA analysis is the GATK best practices pipeline from the Broad Institute [Auwera13]. The data generated by the DNA sequencing platforms is typically hundreds of GBs in size and therefore requires a lot of computational resources to process. This means that for efficient analysis, we require parallelizable solutions which can be easily scaled. While some stages of the analysis pipeline, such as the Burrows-Wheeler Aligner (BWA mem) [Li13], are highly scalable, other stages are not, when executed on a distributed computing infrastructure.

In recent years, a number of big data frameworks have emerged to efficiently manage and process large datasets in an easy way, the most popular of which are MapReduce [Dean08] and Apache Spark [Zaharia10]. Both of these frameworks provide good scalability and fault tolerance. The MapReduce framework has a limitation on programmability though, as it requires the programmer to write code where a map step is always followed by a reduce step. Moreover, it saves intermediate data into disk, which increases disk access overhead. Spark removes those restrictions, allowing programmers to perform many other transformations besides just map and reduce, while keeping data in memory between these transformations.

There are existing big data frameworks that attempt to improve the scalability and efficiency of genomics pipelines. One example is SparkBWA [Abuin16], which implements a Spark API to allow easy cluster scalability of BWA. However, SparkBWA only addresses the first stage of the GATK pipeline. Like SparkBWA, [Al-Ars15] also only deals with the first stage of the pipeline. Other solutions, such as the Churchill DNA analysis pipeline, efficiently utilize the available computational resources using a custom-made tightly-integrated computational pipeline [Kelly15]. The disadvantage of Churchill though is that it combines the different analysis stages into a single tightly-integrated computational pipeline, making it non-generic, meaning that parts of the pipeline cannot be easily replaced. Another solution proposed to solve the scalability challenge of the GATK pipeline is the Halvade framework, which implements the different GATK pipeline stages using a Hadoop-based MapReduce approach [Decap15]. The advantage of the Halvade's approach is that it leaves the pipeline tools unmodified, meaning that one can easily replace a pipeline tool with another one. A drawback of the Halvade framework though is that it is implemented using a classic Hadoop MapReduce based big data framework that is heavily disk oriented, which therefore leads to large disk access overhead due to the large data set sizes of genomics applications.

ACM-BCB '17, August 20-23, 2017, Boston, MA, USA

ACM ISBN 978-1-4503-4722-8/17/08…$15.00
https://doi.org/10.1145/3107411.3107438

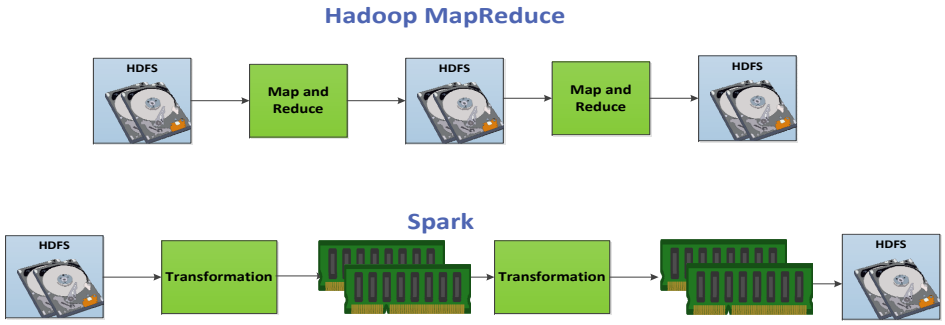


Figure 1: Spark vs Hadoop MapReduce

[Mushtaq15] discusses a Spark implementation of the full GATK DNA analysis pipeline that targets smaller Exome input files. The system described there attempts to process the GATK pipeline fully in memory, which creates memory size bottlenecks resulting in runtime failure for input files of larger sizes. In addition, that framework does not differentiate between the requirements of the various stages of the GATK pipeline, and runs the whole pipeline as a single run, thereby not fully optimizing system utilization for each stage.

In this paper, we propose a new Spark based framework called SparkGA (Spark Genome Analysis) that allows a DNA pipeline to run efficiently and cost-effectively on a scalable computational cluster. SparkGA uses Spark's in-memory computation capabilities to improve performance of the framework. At the same time and depending on available system resources, it can be configured to reduce its memory needs as a trade off for reduced performance. Due to this feature, it can even be run on a single node computer with just 16 GB of RAM. SparkGA implements the genomics computational pipeline as three distinct steps, and therefore allows for optimizing various Spark runtime parameters independently for each step of the framework. Moreover, like Halvade, it uses the pipeline tools unmodified, meaning that one tool can easily be replaced with another one. There has been recent work to accelerate individual stages of the pipeline using accelerators, such as in [Ahmed15] and [Ren15]. Since, like Halvade, SparkGA can also use the individual tools unmodified, such tools can directly rely on SparkGA for a scalable solution.

The contributions of this paper are as follows.

- We implemented SparkGA: a highly scalable, cost-effective, accurate and generic Apache Spark-based framework for a DNA analysis pipeline, which uses the original unmodified tools of a classical DNA pipeline.
- We optimized the framework using a memory-efficient load balancing step to ensure reducing the memory requirements of the compute nodes. Due to this, SparkGA can run even on a single node computer with just 16 GB of RAM.
- We ensured a modular implementation of the framework as three distinct steps that allow us to optimize the Spark runtime parameters for each step.

This paper is organized as follows. In Section 2, we discuss big data techniques, different stages of a DNA analysis pipeline and related work. Section 3 presents the Apache Spark framework we implemented to enable pipeline scalability. This is followed by Section 4, which discusses the performance and accuracy evaluations. We finally conclude the paper with Section 5.

2 BACKGROUND

In this section, first we discuss big data techniques available for executing parallel applications on scalable computational infrastructure, and then we discuss the GATK best practices DNA analysis pipeline. Lastly, we discuss the related work.

2.1 Big data techniques

The MapReduce programming model has been one of the most prevalent approaches used to manage data intensive computational pipelines that need to be processed on multiple compute nodes in a scalable cluster. This model divides the computation into two phases: map and reduce. During the map phase, input data is first formatted as key-value $<K, V>$ pairs followed by performing a specific mapping function on all of these pairs, resulting in a mapping of the input to an output set of $<K, V>$ pairs. The mapping function is executed in a distributed fashion using various mapping tasks that run locally on the data present in the nodes of the cluster. The output is then taken by the reduce tasks which first shuffle the data by grouping all the values that belong to the same key together. Afterwards, the reduce tasks compute a single output from the grouped $<K, V>$ pairs. Apache Hadoop is an open source implementation of the MapReduce programming model, which consists of three components: 1) the Hadoop Distributed File System (HDFS) that allows storing large data files on multiple nodes, 2) a resource manager called YARN that distributes the tasks to be executed to the various nodes in the cluster, and 3) the Hadoop MapReduce framework itself.

One disadvantage of the MapReduce framework is that it stores all data generated between the two phases (map and reduce) on disk, and if multiple map/reduce stages are chained together, it stores the output of each stage on the HDFS. This implies significant overhead due to the intensive disk access. Another disadvantage is that the only transformations allowed are map and reduce. If a different transformation has to be applied, it has to be done by modifying the map and reduce functions, thus making development with this framework very cumbersome.

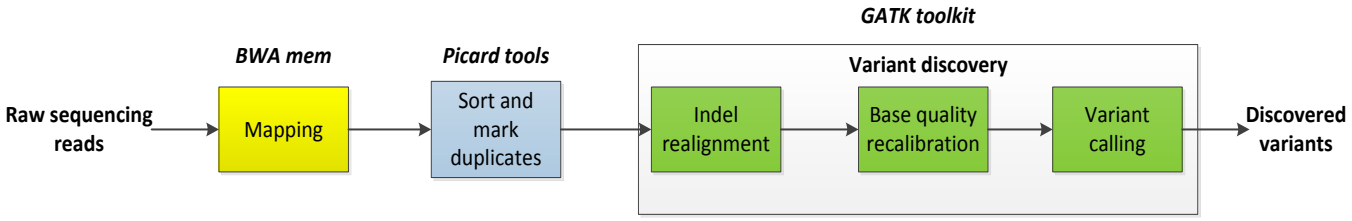


Figure 2: Classical DNA analysis and variant calling pipeline.

Apache Spark is a more recent big data framework that addresses the disadvantages of MapReduce listed above, as illustrated in Figure 1. First, it allows more transformations than mere map and reduce. It provides transformations such as join, cogroup, intersection, distinct and many others as predefined operations. This makes development of programs much easier as compared to the MapReduce framework. Moreover, the output of each transformation is saved in memory, but still allowing disk to be used to save data that does not fit within the available memory size. In this way, Spark avoids the overhead of disk access that is so prevalent in the MapReduce framework. In addition, Spark provides a programming interface to implement algorithms in various programming languages such as Scala, Python and Java, which allows writing programs in the language that is best suitable for the problem.

2.2 DNA analysis pipeline

Figure 2 shows a typical DNA analysis and variant calling pipeline. The input data set to this pipeline are the raw sequencing reads, which are obtained from a DNA sequencing machine. Since the DNA is usually over-sampled by as much as 30x to 100x by the sequencing machine, the number of such reads for a human are very large, and therefore the file storing these reads is of a very large size, typically in the range of several hundreds of gigabytes. One standard file format used today to store these reads is the FASTQ file format [Jones12].

The first step performed in the DNA analysis pipeline is DNA mapping. In this step, raw reads are mapped to a reference genome. Many classic alignment tools can be used in the GATK pipeline, including Bowtie2 [Langmead12] or the popular Burrows-Wheeler Aligner (BWA).

Since the DNA sequencing machine over-samples the DNA, the input data set might have multiple copies of the same raw read. These copies are reduced to a single copy by marking the duplicates. In the GATK best practices pipeline, this is usually done by using the mark duplicates tool from Picard [Picard]. The output of mark duplicates is a compressed file of BAM format.

Finally, the last part of the GATK best practices pipeline is done using tools from the GATK toolkit built by the Broad institute. The final output of the pipeline is a VCF (variant call format) file that contains all of the discovered variants. The first step is the indel realigner, which performs local realignment of reads around indels. The output of this part goes to the base quality recalibrator, which corrects quality scores for those reads which were assigned incorrect quality scores by the sequencing machine. Finally, a variant calling tool, such as the haplotype caller, discovers all the variants and saves them in a VCF file.

While the BWA mem DNA alignment tool of the best practices pipeline can scale well on a multicore system using multi-threading, it is difficult to exploit parallelism for other tools in the rest of the pipeline. In addition, other pipelines, not as well constructed and optimized as the best practices pipeline, are much less able to utilize the available computational infrastructure efficiently. The objective of our framework is to provide a generic method that is easy to use to ensure scalability to many of the various genomics analysis pipelines.

2.3 Related work

Recently, there have been approaches to tackle the scalability problem of a DNA analysis pipeline, by using big data parallelization techniques. One such example is the Churchill [Kelly15] DNA analysis pipeline, which utilizes the available computational resources using a custom-made tightly-integrated computational pipeline. Due to this tight integration, it is not trivial to replace parts of that pipeline. Halvade on the other hand, which uses the Hadoop MapReduce framework, allows the original unmodified pipeline tools to be used, meaning that parts of the pipeline can easily be replaced by new and better tools. The DNA mapping part is done by the map tasks while marking of duplicates and remaining parts of the pipeline are done by the reduce tasks. Within a map task, the unmodified DNA mapping tool is called using the Java's process package. Similarly, marking of duplicates and the remaining parts of the pipeline are done using unmodified original tools. The output of the mapping phase contains mapped reads keyed by chromosomal regions, where each chromosomal region is a part of a chromosome. Each reduce task therefore works on a chromosomal region, and outputs a corresponding vcf file. These vcf files are later combined into a single file.

There are a few limitations with Halvade's approach though. Firstly, it employs the MapReduce framework, which is heavily disk oriented, which leads to disk access overhead due to the large data set sizes of genomics applications. Secondly, it creates chromosomal regions based on the length of the chromosomes, while not checking the actual number of reads. This tends to create unbalanced regions, meaning that some regions may have significantly larger number of reads than other regions. This in turn, slows down the execution time of tasks working on bigger regions significantly, compared to tasks working on smaller regions, thus limiting the overall performance.

Our previous work [Mushtaq15] addressed the problems with MapReduce based solutions like Halvade by having a Spark based implementation of the DNA analysis pipeline. The system described there attempts to process the DNA pipeline fully in memory, and

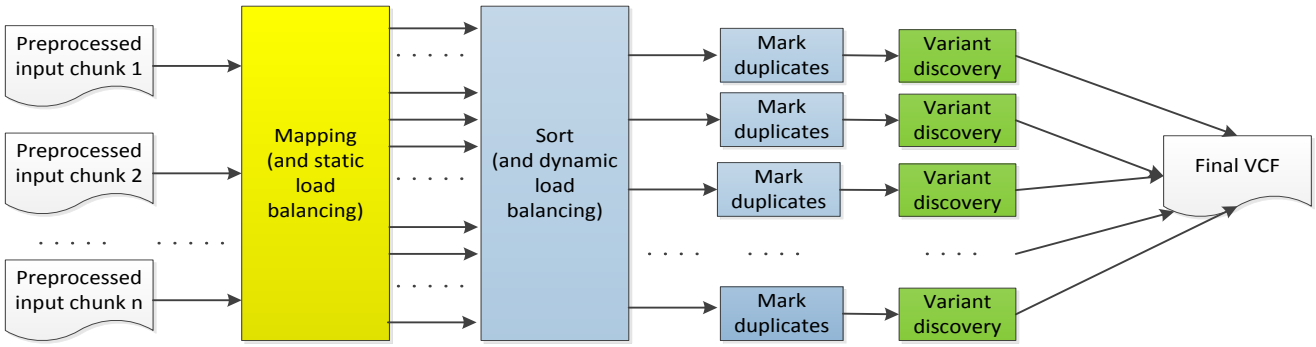


Figure 3: Data flow of the Spark-enabled GATK best practices pipeline using our framework

Table 1: Comparison of tools used in GATK best practices pipeline and SparkGA

Step	GATK best practices	Our framework
Align reads	BWA mem	BWA mem
SAM to BAM	Picard	Picard's Java library
Sort reads	Picard	Sorting in Scala
Mark duplicates	Picard	Picard
Indel realignment	GATK	GATK
Base recalibration	GATK	GATK
Haplotype caller	GATK	GATK

thus is able to create chromosomal regions based on the number of reads. However, since its load balancing step is done fully in memory, it results in out of memory errors for large input files. In addition, that framework does not differentiate between the requirements of the various stages of the GATK pipeline, and runs the whole pipeline as a single run, thereby not fully optimizing system utilization for each stage.

SparkGA, addresses the problem of [Mushtaq15] by implementing a memory efficient load balancing step. Moreover, since the memory and computational requirements for different steps vary, instead of running the whole pipeline with one program, SparkGA runs the pipeline in three different steps: DNA mapping & static load balancing; dynamic load balancing & SAM to BAM; and marking of duplicates and variant discovery. Through an xml configuration file, SparkGA allows the user to tune memory and cores of executors for all those three different steps. It has to be noticed here that Apache Spark does not allow to modify the memory and number of cores for executors at runtime. Therefore, the only way we can achieve this is by running the program in three steps, like SparkGA does.

3 IMPLEMENTATION

We used Apache Spark to implement our framework to parallelize a DNA classical pipeline. We use all the tools of a DNA pipeline unchanged. As shown in Table 1, the only parts of the pipeline that our framework replaces are sorting and SAM to BAM conversion. Those two parts are straightforward to implement anyway and do not require complex tools.

We discuss the overview of our parallelized GATK approach in Section 3.1, followed by our mapping and static load balancing approach in Section 3.2. Next, we discuss our sorting and dynamic load balancing method in Section 3.3 followed by our method to discover the variants in Section 3.4.

3.1 Overview

The dataflow of the execution of SparkGA is shown in Figure 3. In a typical DNA sequencing experiment, two FASTQ files are generated, that represent the two ends of a pair of sequences. These two input FASTQ files are divided into interleaved chunks using a chunks segmentation tool built by us, that also uploads these files to HDFS, thereby making them accessible to all nodes in a distributed cluster. Each chunk is then processed by a separate DNA mapping task that performs DNA alignment of the short reads against a reference genome, using a DNA mapping tool, such as BWA mem. The output of this step represents a list of read alignments (so-called SAM records) that will normally be stored in the SAM file. Since, we already know the length of each chromosome through the given reference file, we can already apply an approximate load balancing by reading the output of the SAM files produced by each BWA mem task. We call this approach static load balancing, since it depends upon information available even before running the program. Later, we apply further load balancing that evenly distributes the data for further computation. We call this approach dynamic load balancing, since it depends upon actual reads, which are only available at runtime. Each load balanced region contains reads from a part of a chromosome. By being able to create regions based on actual reads, rather than just the chromosomal length, like in the case of Halvade, we are able to achieve a better performance, as shown by the evaluation results section (Section 4 of this paper). Moreover, unlike Halvade, our tool is able to run even with less free disk space. Lastly, unlike [Mushtaq15], which consumes too much memory in the load balancing step, and is therefore prone to memory crashes for big data sets, our tool can work even on a single node with just 16 GB of RAM.

Subsequently, for each chromosome region, the aligned reads are sorted using their alignment positions (which replaces the sorting step in the best practices pipeline). Then, the rest of the analysis steps are performed in parallel using a tool for marking duplicates and other tools of the pipeline, such as indel realigner, base recalibration and haplotype caller, by applying those tools on different chromosomal regions separately, resulting in various VCF files. Finally, the contents of those VCF files are sorted and merged into a single file that contains all variants identified in the analysis.

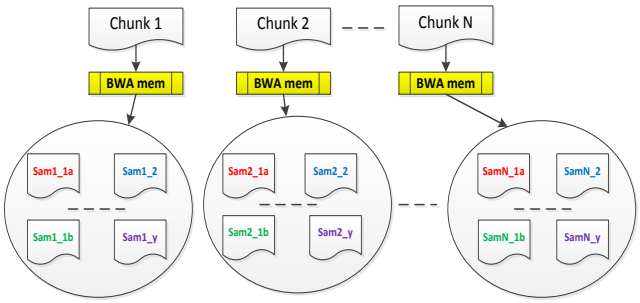


Figure 4: Mapping and static load balancing

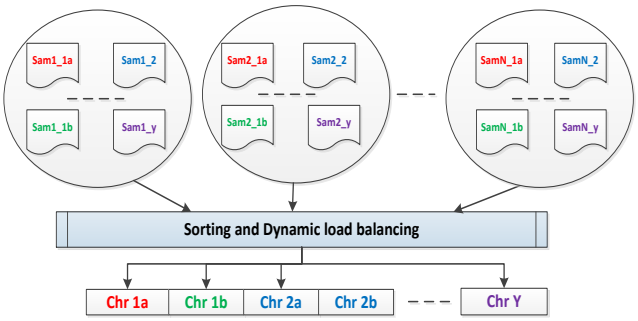


Figure 5: Sorting and dynamic load balancing

We perform the execution in three different steps: mapping and static load balancing; sorting and dynamic load balancing; and mark duplicates and variant discovery. The benefit of this approach is that we can set different optimized Spark execution parameters (such as executor memories and number of executors) for each of these three steps separately. Spark has limitations that do not allow it to change those parameters at runtime.

3.2 Mapping and static load balancing

Figure 4 illustrates this step of our implementation. In this step, each input chunk is fed to a mapping and static load balancing task. Mapping can be done by using a mapping tool, such as BWA mem, while load balancing is done by creating sub-groups (or chromosomal regions) of the resulting mapped reads output by the mapping tool, according to the chromosome number and mapping position. The chromosomal regions are created by equally dividing chromosomes based on their length. This means that longer chromosomes will have more regions than smaller ones. All the SAM files produced by this step are placed into the HDFS in preparation to perform the next step. Each of the mapping and static load balancing tasks creates SAM files that map to any region of the whole genome. For example, it is possible that task 1 produces a SAM file for region $<1, a>$ (Chromosome 1, part a) and task 2 also produces a SAM file for that same region ($<1, a>$). That is why we tag each SAM file with the task number. For example, for task 1, we can name this file as Sam1_1a, while for task 2, name it as Sam2_1a. All the data from SAM files corresponding to the same region will then be grouped together by the next step of the framework, which is the sorting and dynamic load balancing step.

The advantage of static load balancing is that it incurs almost no performance cost, as we rely on the length of the chromosomes which are already known. We notice that after this step, most of the regions made are already balanced according to the number of reads, which reduces the amount of work that needs to be done by the dynamic load balancing part of the next step. Lastly, since we communicate data from this step to the next step through the HDFS, the memory requirement for the nodes is reduced significantly, which enables SparkGA to run on nodes with as little memory as 16 GB.

SparkGA also provides the feature of multiplicating the number of regions after the static load balancing step, through a region multiplicating factor. For example, if the user sets the region multiplication factor of 4, the dynamic load balancing step would create approximately 4 times more regions than created by the static load balancing step. The benefit of that is that fewer number of files would have to be uploaded by the static load balancing step, thus reducing the overhead of accessing HDFS.

3.3 Sorting and dynamic load balancing

Approach. In this step, all the SAM files produced for different regions in the mapping and static load balancing part are combined and sorted to produce BAM and BED files. A BED file is used to restrict the computation in the variant discovery part. The idea is that since a BAM file for a region contains data for only that region, the variant discovery part does not need to scan the whole genome for finding mutations, but instead just inspect within that region. The way SAM files are combined is illustrated in Figure 5. For example, all SAM files produced for region 1a (for example, Sam1_1a, Sam2_1a, etc.) are combined into one BAM file.

The dynamic load balancing step can further divide a region if it has too many reads, since the execution time for several tools in the pipeline depends on the number of reads being processed. We note that after static load balancing, most of the regions during the dynamic load balancing step are already well-balanced. Only a limited number of regions need further division by the dynamic load balancer. The information about the number of reads is already provided by the mapping and static load balancing part, so by reading that information, we already know which regions are too big.

Creating BAM and BED files for a region which is already balanced is simple. The SAM records for such a region are simply sorted and put into a BAM file. Moroever, the BED file is created by looking at the positions of the SAM records in that region.

However, for regions which are not balanced, dynamic load balancing has to be applied. For that purpose, for each SAM chunk produced by the static load balancer, a corresponding file which contains positions of all the reads inside that SAM chunk, is also produced. This is done because otherwise, when dividing a region into sub regions, a dynamic load balancing task would have to collect all the SAM records for that region and sort them first. Sorting the SAM records themselves is memory intensive. That is why, in SparkGA, instead of sorting the SAM records themselves, a dynamic load balancing task just sorts the positions that come from the files containing the positions of those SAM records. Afterwards, as it reads each SAM record from the corresponding chunks, it

Algorithm 1: Dynamic load balancing

procedure createBAMFiles
▹ **Elements of the RDD are of type <regID, <chunkID, numOfReads>>**
$inputData \leftarrow readInfoFile(dynLBInfo).cache()$
▹ **Get the number of reads for each chunk**
$readsPerChunk \leftarrow inputData.map\{case(regID, (chunkID, numOfReads)) => numOfReads\}$
▹ **Get the total number of reads**
$totalReads \leftarrow readsPerChunk.reduceByKey(\_ + \_)$
▹ **Group the input data by region**
$chrReg \leftarrow inputData.groupByKey()$
$chrReg.cache()$
▹ **Calculate average number of reads per statically load balanced regions**
$avgReadsPerRegion \leftarrow totalReads/chrReg.count()$
▹ **makeBAM divides a region further if required, & creates BAM and BED files for each (sub)region**
$chrReg.foreach\{case(regID, infoList) => makeBAM(regID, infoList, avgReadsPerRegion)\}$
end procedure

checks which subregion that SAM record would go to by finding its position in the sorted positions array, using binary search. After finding out this information, it directly appends that SAM record to the SAM file representing that subregion. To accelerate this step, SparkGA allows to perform this step with multiple threads. In this way, different threads can read the SAM records from the different chunks representing a region, in parallel. Afterwards, all SAM files created by a dynamic load balancing task are read, sorted and converted into BAM files. The BED files are also created in that step. The sorting and conversion from SAM to BAM for subregions can also be done in parallel by using multiple threads.

Algorithm. The main algorithm for dynamic load balancing is shown as algorithm 1. Besides creating the SAM chunks and their corresponding positions files, the static load balancing step also uploads a global information file to help the dynamic load balancer. Each line of this file corresponds to a SAM chunk and contains information about the region ID that chunks belongs to, the ID of that chunk itself, and the number of reads it has. This file is then read by the dynamic load balancing step as shown in algorithm 1. The information from this file is first placed in a Spark's RDD (Resilient Distributed Dataset), which is the basic distributed data structure of Spark. This RDD is transformed into another RDD where each element of the RDD is assigned to the number of reads the corresponding chunk contains. Next, these number of reads are summed up using reduceByKey, giving us the total number of reads. After that, we group the input data by region, so that the information for each region is now contained in a list. The total number of reads divided by the total number of regions then gives us the average number of reads per region. Then for each statically load balanced region, the function *makeBAM* is called which performs the dynamic load balancing. The *makeBAM* function reads all the SAM files pertaining to the region its working on. The information passed to it contains a list of tuples (the 2nd parameter), where each tuple contains the chunk ID and the number of reads that chunk contains. Since the number of reads for each chunk is known as well as the average number of reads per statically load balanced regions (the 3rd parameter), the *makeBAM* function can divide a

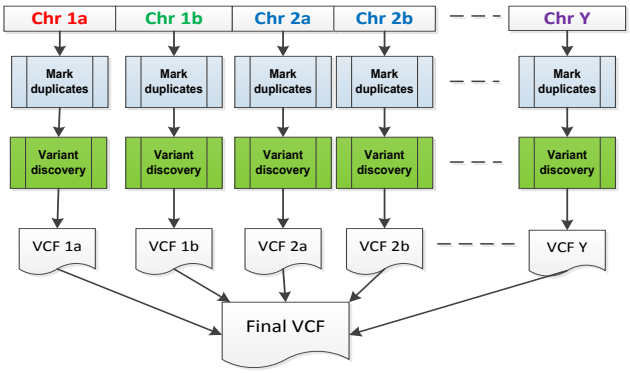


Figure 6: Mark duplicate and variant discovery

statically load balanced region into subregions of approximately equal sizes if that region has too many reads. Through a region multiplication factor, the user can also set how many regions the dynamic load balancer step would create. For example, if the user sets a region multiplication factor of 4, the dynamic load balancing step would create approximately 4 times more regions than what the static load balancing step created. The dynamic load balancing tasks can be run with multiple threads to improve the performance, as discussed in the previous paragraph.

3.4 Mark duplicates and variant discovery

In the last step, the BAM and BED files produced by the previous step, are used as an input for the rest of the pipeline. Each task works on a separate set of BAM and BED files, and outputs a VCF file. Later, all these VCF files are combined into one final VCF file. This is illustrated in Figure 6 as discussed next.

Mark duplicates—In this step, all identical short reads mapping to the exact same location in the reference are marked as copies of each other. This is done because allele frequencies and genotype calls can be skewed if certain individual reads are preferentially amplified relative to others.

Table 2: Runtime in minutes on a 4-data node Power7+, and a 5, 10, 15 & 19-data node Power8 (+1 name node) IBM cluster

	4-node P7+		5-node P8		10-node P8		15-node P8		19-node P8	
Step	**Time**	**%**	**Time**	**%**	**Time**	**%**	**Time**	**%**	**Time**	**%**
Mapping and static LB	128	30%	78.7	32%	40.6	29%	29.0	28%	24.4	27%
Sorting and dynamic LB	45	11%	22.1	9%	19.5	14%	18.8	18%	18.7	21%
Variant discovery	249	59%	145.9	59%	77.7	56%	57.2	54%	47.7	52%
Total	422	100%	246.7	100%	137.7	100%	105.0	100%	90.9	100%

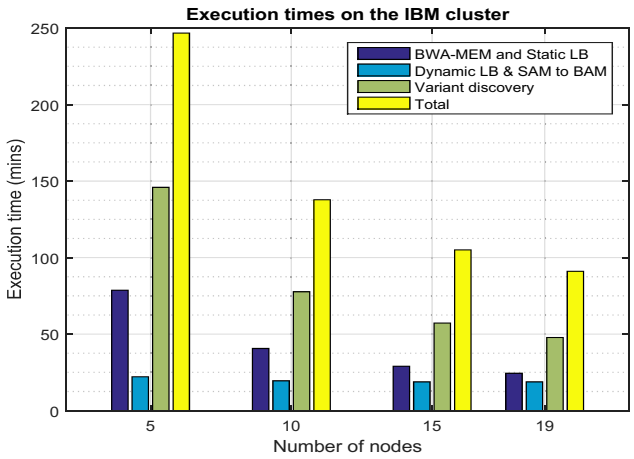


Figure 7: Performance on the IBM cluster

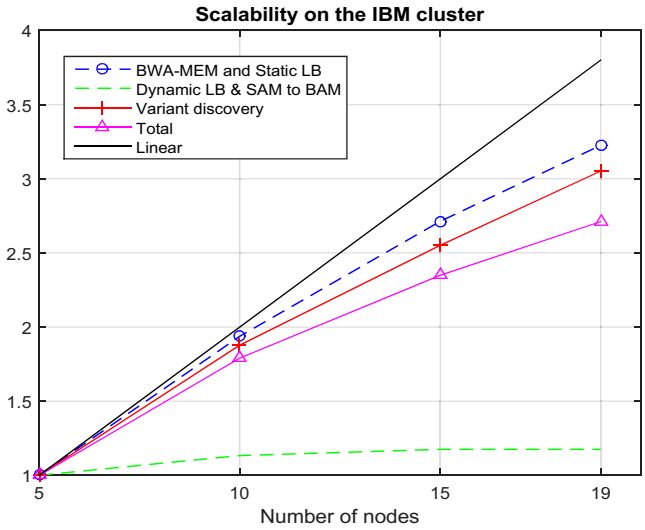


Figure 8: Scalability on IBM cluster

Indel realigner—The indel realigner tool performs local realignment of reads around indels, so as to minimize mismatching bases across all reads.

Base quality recalibration—When a sequencing machine outputs data, it assigns a quality score to each read. This quality score is to signal how much a read can be trusted to be accurate. However, sometimes a sequencing machine can also generate erroneous quality scores. The base quality recalibrator tries to correct this problem by looking at a human genome database. Since it is estimated that 99% of all variants in the Caucasian population have been put in the dbSNP database, the vast majority of mapping differences not in dbSNP should basically be sequencing errors. By using this database and and applying machine learning models on the output from indel realigner, the base recalibrator tries to correct any erroneous quality scores given to the reads.

Haplotype caller—The haplotype caller is the variant discovery tool of choice for the GATK pipeline. Whenever the haplotype caller sees a region which shows signs of variations, it discards the existing mapping information and assembles the reads itself for that region. This helps in making the haplotype caller more accurate for regions which are difficult to call, such as those that contain variants close to each other.

4 EVALUATION RESULTS

We tested the results on three different types of hardware platforms. The first platform is an IBM Power7+ cluster with 4 data nodes + 1 master node. Each node has two sockets that host a Power7+ processor. In total, a node there has 16 physical cores and 128GB of memory. Each Power7+ core is capable of 4-way simultaneous multithreading. Spark is run over YARN in that platform. The second platform consists of 20 IBM Power8 S822LC nodes (19 data nodes + 1 master node), with each node having two sockets, where each socket hosts a Power8 processor. In total, a node there has 20 physical cores and 512GB of physical memory. Each Power8 core can support 8-way simultaneous multithreading. Spark was run using a Spark cluster on that platform. Lastly, the third platform used was the Dutch national e-infrastructure's SURFsara cluster [SURFsara], which allows up to 72 Intel/AMD nodes to be used using YARN as a resource manager. This cluster was used to compare our results with Halvade.

We tested SparkGA with the publicly available whole human genome dataset G15512.HCC1954.1 from Genomics Data Commons (formally Cancer Genomics Hub) [GDC], with the reference genome human_g1k_v37_decoy. The raw reads have 65x coverage with size of over 402GB. SparkGA is able to compute the pipeline in 422 minutes (7 hours and 2 minutes) on the 4-node Power7+ cluster. The breakdown of this runtime for different steps is shown in Table 2, indicating that the variant discovery step consumes most of the time amounting to 59%, followed by the mapping step at 30%, then the sorting at 11%. On the 19-node IBM Power8 cluster, the execution consumed a total of 90.9 minutes, with similar distribution of runtime. However, the table shows that the for this shorter runtime, the sorting step is becoming more dominant, with its share

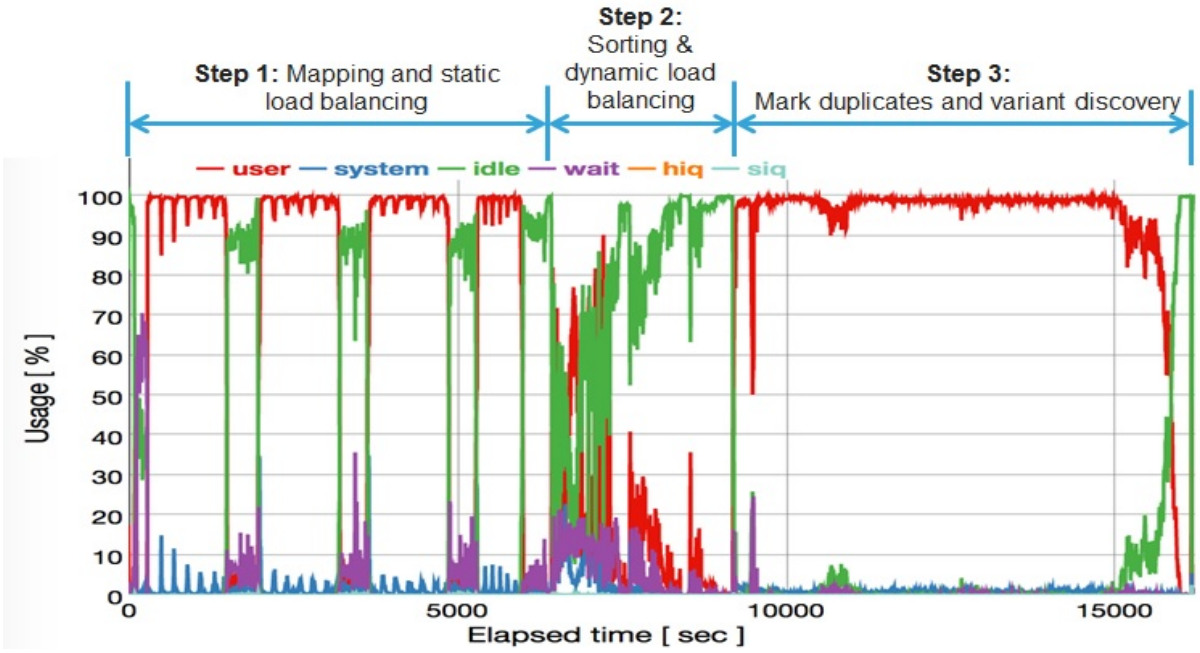


Figure 9: Utilization of the compute resources of a single data node in a 5-node Power8 cluster

increasing to 21%. In the same table, execution times for different sizes of IBM Power8 clusters are also shown.

4.1 Scalability analysis

Figure 7 shows a bar chart of the execution times registered for different steps and the total runtime for the IBM Power8 cluster with an increasing number of data nodes activated, ranging from 5, 10, 15 to 19 data nodes. The corresponding speedup graph is shown in Figure 8. Here, we observe that the BWA mem and variant discovery steps scale quite well, while the load balancing step plateaus. BWA mem was expected to scale well, but here we see that the variant discovery step scales almost as well. The reason it scales so well is that tools such as the haplotype caller can make very efficient use of vector instructions to improve the performance. Moreover, simultaneous multithreading also improves the performance. Each IBM Power8 processor has 8 high quality simultaneous threads for each physical core, which can benefit the workload greatly.

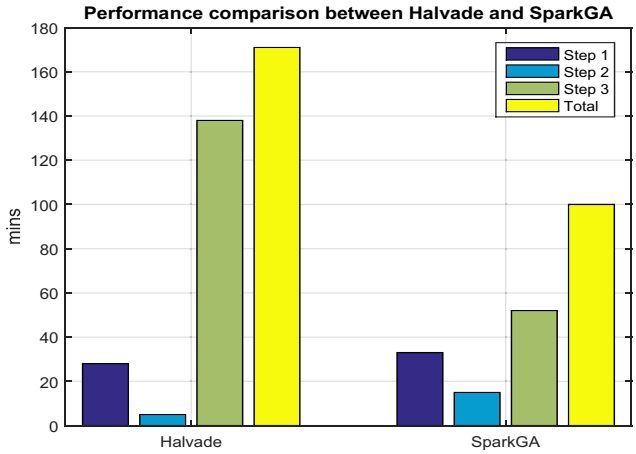


Figure 10: Halvade vs SparkGA performance comparison

4.2 Performance measurements

In order to understand the bottlenecks in the three steps, we plot the utilization of system resources in Figure 9. The figure shows rather high CPU utilization in the mapping as well as in the variant discovery steps. We also observe a drop in the CPU utilization for the mapping step at regular intervals. This is caused by the files that this step needs to upload to the HDFS. In addition, we observe some idle time in the sorting and dynamic load balancing step. This is due to the heavy I/O activity done by the sorting and dynamic load balancing step. This explains the limited scalability of the sorting and dynamic load balancing step as we increase the size of the cluster. However, we see a very high utilization in the last step (mark duplicates and variant discovery). The reason is that in this step there is no interaction with the HDFS, except for when the BAM files from the dynamic load balancing part are read, and when the final output VCF files (which are very small in size anyway) are written.

4.3 Performance comparison with Halvade

We also compared the performance with Halvade on a YARN based cluster. For that purpose, we used the maximum 72 nodes allowed by the SURFsara cluster. The SURFsara cluster nodes have mixed

Table 3: Different steps of Halvade and SparkGA

Steps	Halvade	SparkGA
1	Mapping and static load balancing	Mapping and static load balancing
2	Sorting and grouping of data by regions	Sorting, Dynamic load balancing and SAM to BAM
3	SAM to BAM, Mark duplicates and Variant discovery	Mark duplicates and Variant discovery

Table 4: Performance comparison between Halvade and SparkGA

	Halvade		SparkGA	
Step	**Minutes**	**% of Total**	**Minutes**	**% of Total**
1	28	16%	33	33%
2	5	3%	15	15%
3	138	81%	52	52%
Total	**171**	100%	**100**	100%

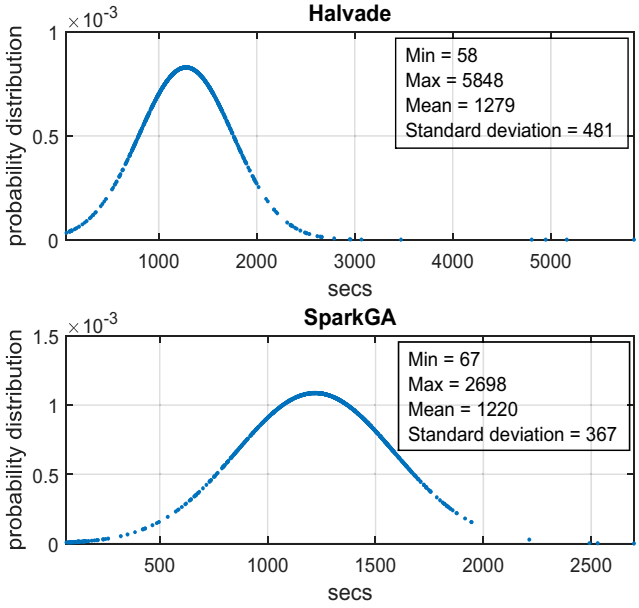
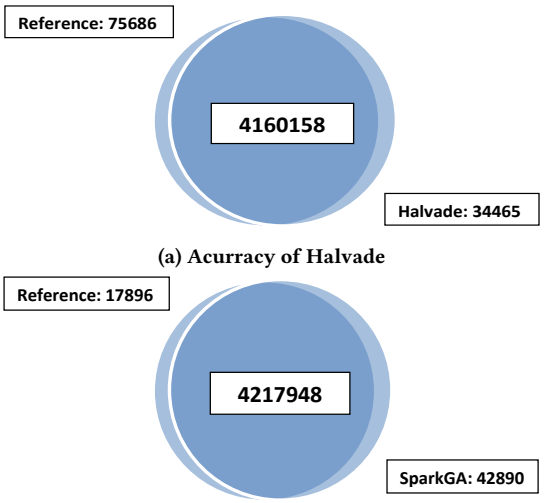


Figure 11: Load balancing of tasks in step 3

Intel and AMD processors, with each processor having 8 processing cores, and each node having 56 GB of RAM. We also tried to run Halvade on the 4 node IBM P7+ cluster, but it exceeded the available disk space. On the other hand, [Mushtaq15] exceeded the memory available on the 4 node IBM P7+ cluster. However, SparkGA is able to run on the 4 node IBM P7+ cluster, despite the low hard disk space and memory on the nodes.

It has to be noted first that the steps taken by SparkGA are not exactly the same as those taken by Halvade. This difference is shown in Table 3. In the second step, Halvade does not perform dynamic load balancing nor it creates the BAM and BED files, which it creates in the third step rather. However, since creation of BAM



(a) Acurracy of Halvade

(b) Accuracy of SparkGA

Figure 12: Accuracy comparison of SparkGA and Halvade with the classical GATK pipeline

and BED files take on average, just around 5% of a reduce task in Halvade, The execution times of Halvade and SparkGA for the third step can still be compared fairly.

The comparison with Halvade is shown in Table 4 and illustrated in Figure 10. The time for Step 1 is slightly higher for SparkGA because SparkGA also uploads some files required for its next step: sorting and dynamic load balancing part. In Step 2, Halvade performs sorting without dynamic load balancing, that is why it consumes less time than SparkGA. In the variant discovery step (Step 3) however, SparkGA is much faster than Halvade due to its improved load balancing. Overall Spark took 71% less time than Halvade for the selected data set.

The improvement achieved due to better load balancing can be understood from the Figure 11, which shows bell curve of the time taken by the tasks of Step 3. The mean and standard deviations of the execution time of these tasks are also shown in that figure. We can see that the standard deviation for Halvade is poorer (481 as compared to 367 seconds for SparkGA). This is because some of the tasks in Halvade take too much time. On the other hand, the task execution times for SparkGA are concentrated more towards the mean.

Since the cost of using a cluster depends directly on the time spent running the applications, this means that compared to Halvade, our solution is much more cost effective, as it takes significantly less time in comparison.

4.4 Correctness of the results

We evaluate the correctness of our framework by comparing the variants discovered in the VCF files of SparkGA with those resulting from the GATK best practices pipeline as a reference. One obvious concern is that since our method partitions the input dataset into chunks and the reference genome into regions, there is a risk of reduction in accuracy of the results. The correctness analysis is

Table 5: Accuracy comparison of SparkGA and Halvade

Parameter	Formula	Halvade (%)	SparkGA (%)
Sensitivity	TP / (TP + FN)	98.2%	99.6%
Specificity	TN / (TN + FP)	99.999%	99.999%
Precision	TP / (TP + FP)	99.18%	99.0%
False discovery rate	FP / (TP + FP)	0.82%	1.01%
Accuracy	**(TP + TN) / (TP + TN + FP + FN)**	**99.9966%**	**99.9981%**

based on measuring: the true positive (TP) variants detected by the analysis in the sample (those detected by both SparkGA and GATK); the false negatives (FN) detected by GATK but not by SparkGA; the false positives (FP) detected by SparkGA but not by GATK; and the true negatives (TN) representing all possible mutation locations in the genome that are not detected by either pipeline.

We first experiment by varying the number of chunks and regions from 512 to 8192 in powers of 2 and measure the accuracy of the results. Our method achieves a constant accuracy rate of 99.9981% for all experiments, which is better than Halvade's accuracy of 99.9966%. The Venn diagrams of Figure 12 show the comparison results of the VCF output of SparkGA and Halvade with that of the traditional GATK pipeline. The lower Venn diagram in that figure shows that both SparkGA and GATK pipelines detect 4217948 variants (TP). Moreover, SparkGA detects 42890 false positives and 17896 false negatives. In comparison, Halvade detects less variants, meaning it has more false negatives (75686 compared to SparkGA's 17896), as shown by the upper Venn diagram in Figure 12.

Table 5 also compares SparkGA's accuracy with that of Halvade. Moreover, it defines a number of parameters to quantify the correctness. Besides, better accuracy, SparkGA also has better Sensitivity compared to Halvade, due to its ability to detect more variants. Halvade however is slightly better when it comes to precision and false discovery rate.

5 CONCLUSIONS

Next generation sequencing has dramatically increased the speed and throughput of DNA sequencing. However, DNA analysis requires efficient and scalable solutions to ensure high computational performance at an affordable cost. We propose SparkGA, a big data framework based on Apache Spark that runs efficiently on a multi-node cluster. By applying novel static and dynamic load balancing techniques, our framework can spread the input data more uniformly across the computational nodes, hence ensuring better scalability. It also allows to run the framework on lower-cost nodes with as little as 16GB of memory. SparkGA enables scaling up the GATK best practices pipelines, a well accepted industry standard, achieving a high accuracy of up to 99.9981%. Experimental results show that when deployed on a 20-node IBM Power8 cluster, SparkGA can complete the GATK best practices pipeline in about 90 minutes. Moreover, it is 71% faster than the state-of-the-art solution. Having better performance also means that it is more cost-effective.

The source code of SparkGA is publicly available at the GitHub repository (https://github.com/HamidMushtaq/SparkGA1.git).

ACKNOWLEDGMENTS

Special thanks to SURF cooperative for providing support in testing our framework on the Dutch national e-infrastructure's SURFsara cluster [SURFsara].

REFERENCES

[Auwera13] G.A. van der Auwera, M. Carneiro, C. Hartl, R. Poplin, G. del Angel, A. Levy-Moonshine, T. Jordan, K. Shakir, D. Roazen, J. Thibault, E. Banks, K. Garimella, D. Altshuler, S. Gabriel, M. DePristo, "From FastQ Data to High-Confidence Variant Calls: The Genome Analysis Toolkit Best Practices Pipeline", Current Protocols in Bioinformatics, 43:11.10.1-11.10.33, 2013.

[Decap15] D. Decap, J. Reumers, C. Herzeel, P. Costanza and J. Fostier, "Halvade: scalable sequence analysis with MapReduce", *Bioinformatics*, btv179v2-btv179, 2015.

[Picard] https://broadinstitute.github.io/picard/

[GDC] https://gdc.cancer.gov/

[SURFsara] https://www.surf.nl/en/services-and-products/big-data-services/access/index.html

[Dean08] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", *Commun. ACM*, vol. 51, no. 1, 2008.

[Zaharia10] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker and I.Stoica, "Spark: cluster computing with working sets", *HotCloud'10*, USENIX Association, Berkeley, CA, USA, 10-10.

[Abuin16] J.M. Abuin, J.C. Pichel, T.F. Pena and J. Amigo, "SparkBWA: Speeding Up the Alignment of High-Throughput DNA Sequencing Data" Ed" *PLoS ONE 11.5 (2016)*, e0155461. PMC. Web. 31 Oct. 2016.

[Jones12] D.C. Jones, W.L. Ruzzo, X. Peng and M.G. Katze, "Compression of next-generation sequencing reads aided by highly efficient de novo assembly", *Nucleic Acids Research*, 2012.

[Kelly15] B.J. Kelly, J.R. Fitch, Y. Hu, D.J. Corsmeier, H. Zhong, A.N. Wetzel, R.D. Nordquist, D.L. Newsom and P. White,"Churchill: an ultra-fast, deterministic, highly scalable and balanced parallelization strategy for the discovery of human genetic variation in clinical and population-scale genomics", *Genome Biology*, vol. 16, no. 6, 2015.

[Ahmed15] N. Ahmed, V. M. Sima, E. Houtgast, K. Bertels and Z. Al-Ars, "Heterogeneous hardware/software acceleration of the BWA-MEM DNA alignment algorithm", 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, 2015, pp. 240-246.

[Ren15] S. Ren, V. M. Sima and Z. Al-Ars, "FPGA acceleration of the pair-HMMs forward algorithm for DNA sequence analysis", 2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), Washington, DC, 2015, pp. 1465-1470.

[Al-Ars15] Z. Al-Ars and Hamid Mushtaq "Scalability Potential of BWA DNA Mapping Algorithm on Apache Spark", SIMBig 2015, Cusco, Peru, 2015, pp. 85-88.

[Li13] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM", arXiv:1303.3997 [q-bio.GN], 2013.

[Mushtaq15] H. Mushtaq and Z. Al-Ars, "Cluster-based Apache Spark implementation of the GATK DNA analysis pipeline", 2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), Washington, DC, 2015, pp. 1471-1477.

[Langmead12] B. Langmead and S.L. Salzberg, "Fast gapped-read alignment with Bowtie 2", Nature Methods, vol. 9, no. 4, pp. 357-359, 2012.