Scaling up the GATK RNA-seq Variant Calling Pipeline with Apache Spark

Saiyi Wang

Thesis no. CE-MS-2018-20

Delft University of Technology

Abstract

Next-generation sequencing (NGS) technology has dramatically increased the availability of RNA-seq data. Though primarily used for novel gene identification, expression quantification, and splice analysis, RNA-seq is also a cheap and efficient alternative for variant calling to genome sequencing data. RNA sequencing costs less than genome sequencing. Plus, the variants discovered from RNA-seq data are expressed, which is a desired feature for researchers who want to study the relation between genotype and phenotype. What's more, variants called in RNA-seq data can be used to validate the discoveries from whole-genome sequencing (WGS) or whole-exome sequencing (WES). The GATK team has adapted the Best Practices pipeline to be able to process RNA-seq data from raw FASTQ reads to variants. However, some components of the pipeline are not optimized to process large datasets efficiently. We have studied several scalable solutions that scale up the DNA-seq Best Practices pipeline in order to apply the most efficient framework among them to scaling up the RNA-seq pipeline. We select Spark for its ease of implementation and efficient in-memory computation and implement a parallel RNA-seq variant calling pipeline based on the GATK Best Practices recommendations. Whereas the original sequential pipeline takes ~29 hours to process a dataset of 50 GB with one thread, and ~16 hours with 40 threads on a node with 20 Hyper-Threading cores, our implementation takes only ~2 hours with 16 nodes, each of which has 8 CPU cores without Hyper-Threading. Our implementation is 13.58x faster overall, while it achieves 27.03x speedup for the Picard preprocessing and GATK variant calling part of the pipeline compared to the single-thread execution of the original pipeline. On a large cluster, our implementation is 24.77% faster than the alternative scalable solution while keeping equally accurate results. For the specific Picard and GATK part of the pipeline, our implementation is 67.38% faster.

TU Delft Delft University of Technology

Quantum & Computer Engineering

Scaling up the GATK RNA-seq Variant Calling Pipeline with Apache Spark

by

Saiyi Wang

in partial fulfillment of the requirements for the degree of

Master of Science
in Emebedded Systems

at the Delft University of Technology,
to be defended publicly on Thursday August 30th, 2018 at 14:00.

Supervisor:	Dr. Z. Al-Ars	
Thesis committee:	Dr. Z. Al-Ars,	TU Delft
	Dr. T. Abeel,	TU Delft
	Dr. C. G. Almudever,	TU Delft
	Dr. H. Mushtaq,	Upsilon.lu

Dedicated to my family and friends.

CONTENTS

LIST OF FIGURES

List of Tables

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my supervisor, Dr. Zaid Al-Ars for guiding me through my thesis project. He was always ready to help in all matters regarding research, thesis writing or formalities. His patience, encouragement and immense knowledge helped me get through many challenges. I would also like to thank my daily advisor, Dr. Hamid Mushtaq for proposing this fascinating topic and providing me with a lot of support. When he was working in the QCE department, his office door was always open and he answered every question of mine. Even after he left the department, I can still reach him through email and get suggestions from him.

Furthermore, special thanks must go to the SURFsara cooperative for providing their Big Data Services so I can test my implementation in the SURFsara cluster. I also want to thank Dr. Thomas Abeel and Dr. Carmen G. Almudever for being part of my thesis committee.

Last but not least, I am grateful to my parents. Without their emotional and financial support, I would not be able to come to the Netherlands and have so many wonderful experiences here in both study and life. I must also thank my friends who were also working on their MSc theses in the QCE department. I enjoyed having meals and chatting with them every day. I would also like to thank all my friends in the Netherlands. We had a lot of good times together.

Saiyi Wang
Delft, August 2018

1

INTRODUCTION

1.1. MOTIVATION

With the development of next-generation sequencing (NGS) technology, both DNA-seq and RNA-seq data become increasingly accessible. Identifying variants from DNA-seq has drawn wide spread attention and a number of tools have been developed to address the problem. The GATK Best Practices [3] is one of the most popular DNA-seq pipelines, which recommends a series of tools to process DNA-seq data from raw reads to variant calls. However, due to the usually large size of DNA-seq input data and the poor scalability of some tools in the pipeline, it usually takes tens of hours to run the complete pipeline. The built-in multi-threading of the tools does not help much.

In order to improve the performance of DNA pipelines and get the results faster, a number of solutions are proposed. Churchill [4] customizes the GATK Best Practices pipeline and integrates the different steps in the pipeline to more efficiently utilize the computational resources. Halvade [5] tries to scale up the pipeline using the Hadoop MapReduce [6] [7] framework. SparkGA [2], on the other hand, is based on another big data framework Spark [8]. Both Halvade and SparkGA try to optimize the scalability of the GATK pipeline at a cluster level. The logic behind them is the same: the input data is divided into chunks and each step in the pipeline can be performed on those independent chunks to achieve data-parallel computations. The advantage of the latter two solutions is the ease of implementation of the pipelines on a scalable cluster without having to know the exact details of the cluster setup. In addition, the tools remain unchanged, so they can be easily replaced by other tools or even removed if a step is undesired.

Although primarily used for novel gene identification, expression quantification and splicing analysis [9], RNA-seq data is also a cheaper and efficient alternative to genome sequencing data. First, RNA sequencing costs less than genome sequencing [10]. Second, variants discovered from RNA-seq data are from expressed regions, which is more convenient for researchers to study the relation between genotypes and phenotypes. Plus, calling variants in RNA-seq is also an efficient option to validate the discoveries from whole-genome sequencing (WGS) or whole-exome sequencing (WES) [10][22]. There are also limitations in variant calling in RNA-seq data. For example, the coverage of different chromosomes can vary largely. What's more, the intrinsic complexity, i.e., the noncontiguous reads, of RNA-seq data makes alignment and the later processing difficult. However, a number of aligners intended for RNA-seq data such as STAR [11], TopHat [12][13], GSNAP [14], MapSplice [15], etc. have been developed, trying to address the problem of spliced reads alignment. The GATK team also released a tool especially for RNA-seq variant discovery (see Section 2.3). Such efforts make calling variants in RNA-seq technically possible.

The GATK team has adapted its Best Practices pipeline to be able to process RNA-seq data [16]. However, processing two paired-end raw FASTQ files [17] of 25GB each with this pipeline takes ~29 hours for 1 thread to process. On a Hyper-Threading [18] enabled 20-core node, the pipeline still takes up to ~16 hours to finish with 40 threads, which means that the execution time is reduced by only a factor of 2. Similar to the DNA-seq version, this pipeline also scales poorly with multi-threading and the bottleneck is mainly the Picard preprocessing [19] and GATK variant calling parts. A more scalable solution is thus needed to make it possible to process RNA-seq data efficiently. Considering the similarity of bottlenecks to the DNA-seq version, we expect that it would be possible to scale this pipeline up through the same methods which use scalable analysis techniques mentioned earlier in the section.

1.2. THESIS SCOPE AND CONTRIBUTIONS

In this thesis, the main problem we try to address is how to improve the speed of the RNA analysis pipeline yet keep the accuracy of the results. An implementation is presented intended to scale up the GATK pipeline as described in [16] using a scalable cluster solution, such as the Apache Spark big data framework. The thesis focuses on answering the following research questions.

- Is it possible to scale up the RNA analysis in a cluster environment efficiently? What is the efficiency achieved?
- What would the impact of scalability be on the accuracy of the performed analysis? What is the accuracy to be achieved?
- Which is the most effective framework to use for scaling up such a pipeline?

The contributions of this thesis can be presented as follows:

- A Sparked-based framework is implemented to scale up RNA analysis pipelines by tackling the computational bottleneck of the original sequential pipeline.
- On a single node, our implementation achieves 7.78x speedup overall and 9.12x speedup in Picard preprocessing and GATK variant calling parts compared to the original GATK pipeline running with 1 thread. It also takes 24.29% and 48.72% less time in total and in Picard preprocessing and GATK variant calling part, respectively, compared to the alternative solution while achieving almost the same accuracy.
- In a cluster composed of nodes each with 56 GB RAM and 8 CPU cores without Hyper-Threading, our implementation is able to finish in ~2 hours running on 16 nodes, while the original sequential pipeline takes ~29 hours with one processing thread and ~16 hours with 40 threads on a node with 20 Hyper-Threading cores. It is 13.58x speedup overall and 27.03x speedup in the Picard and GATK part compared to the single-thread execution of the original pipeline. Our implementation is also faster than the alternative solution by 24.77% and 67.38% in total and in the Picard and GATK part while achieving equal accuracy.

1.3. THESIS ORGANIZATION

The rest of the thesis is organized as follows. Background information about RNA sequencing, variant discovery, the RNA analysis pipeline and big data frameworks will be presented in Chapter 2. In Chapter 3, alternative solutions to accelerate the genomic analysis pipeline and the evaluation criteria are presented. The proposed scalable Spark framework is introduced in Chapter 4. Details of the implementation are also explained in the same chapter. Chapter 5 presents the results of evaluation and corresponding analyses, while Chapter 6 draws the conclusions and proposes future work directions.

2
BACKGROUND

2.1. RNA SEQUENCING

RNA sequencing (RNA-seq) is a technology that benefited from the advent of high-throughput DNA sequencing, i.e., next-generation sequencing (NGS) technology. Typically, in the first step, because most of the sequencing platforms are capable of providing only short reads (about 40 - 400 bp), long RNAs have to be fragmented to improve the coverage [20]. Then the fragmented RNAs are reverse transcribed to complementary DNA (cDNA). Subsequently, sequencing adapters are added to the cDNA fragments. The adapters allow the sequencing platform to recognize the fragments. They can also carry other functional elements if needed. The cDNA fragments with adapters are now ready for sequencing, and short reads can be obtained through NGS technology. The output reads are often stored in FASTQ files [17] as they are in DNA-seq. This process is depicted in Figure 2.1.

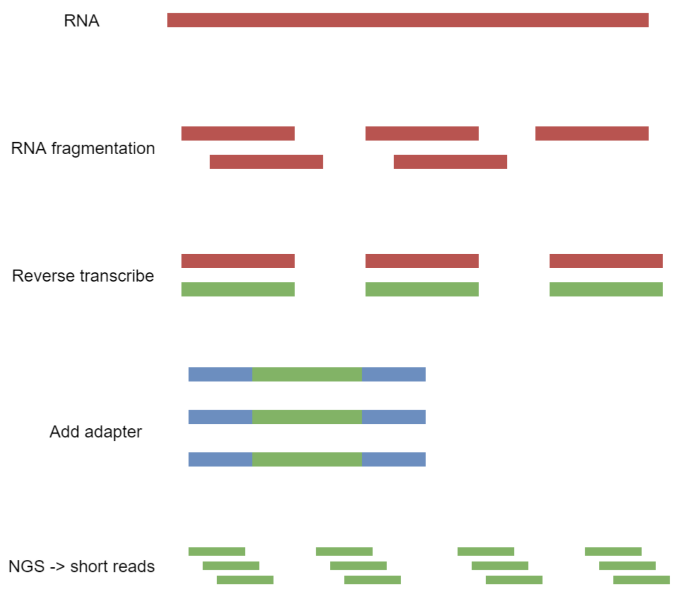

Figure 2.1: A brief description of RNA sequencing

At first, RNA-seq was meant for transcriptome profiling, that is, to catalog the species of transcripts, to determine the transcriptional structure for genes, and to quantify the expression level of the transcripts [21]. However, in recent studies, people are also interested in discovering variants from RNA-seq data [10][22][23][24].

One of the biggest challenges of calling variants in RNA-seq data is the noncontiguous nature of the reads. Figure 2.2 gives an idea of the reasons behind this. The output short reads from the sequencing platform are usually composed of 40 - 400 bp [20], while the average length of human exons is 150 bp [10]. This results in many reads of RNA-seq spanning over the intronic regions, being noncontiguous. The gapped reads make alignment (a necessary step before calling variants) rather challenging.

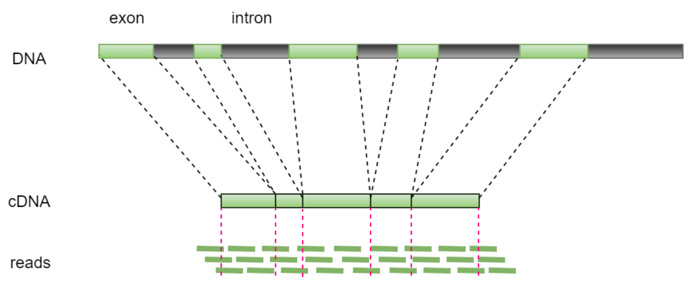

Figure 2.2: Splice junctions in the RNA-seq reads

Alignment is the process where the short reads are mapped to certain positions on the chromosomes according to the reference genome. In DNA alignment this is usually easier as they are continuous sequences. One common challenge that DNA and RNA alignments both have is that the reads may contain genomic variations and sequencing errors. But for RNA alignment, gapped reads increase the difficulty. Apart from finding the right position for continuous reads, aligners also have to consider the possibility of splice junctions. Plus, some other genome regions may have identical or related sequences as the spliced reads, and if they are also transcribed, the problem gets even worse [11]. In recent years, several alignment tools meant for RNA-seq have been developed to address these challenges [11]-[15], making alignment, a crucial step of variant calling more accurate. In Section 2.3, one of those aligners, which is used in our implementation will be briefly introduced.

2.2. SNP AND INDEL DISCOVERY

The GATK variant calling pipeline for RNA-seq aims at discovering two types of variants, SNP and indel. SNP and indel are two of the most common kinds of variants in human genomes [25]. In this section, basic knowledge about SNP and indel and the significance of identifying them will be briefly introduced.

SNP is short for single nucleotide polymorphism. If more than 1% of a population carry different bases at a specific base position, then the variation at this position can be categorized as an SNP [26]. As its name suggests, an SNP is only a single base difference between the reference genome and the studied sequence as shown in Figure 2.3 [1]. SNPs are not necessarily present in genes only; they can also be present in areas between genes. However, in the variant calling pipeline for RNA-seq, only exons (mostly coding regions [27]) in the gene would be analyzed because cDNA is reverse transcribed from mRNA. Among the SNPs in coding regions, some would not change the amino acid sequence of the protein that is produced [28] while others do. Discovering SNPs is important for people to find out the impact of SNPs on gene functions. It is also very valuable to reveal the association between SNPs and diversity in the population, individuality, susceptibility to diseases, individual response to medicine etc [29].

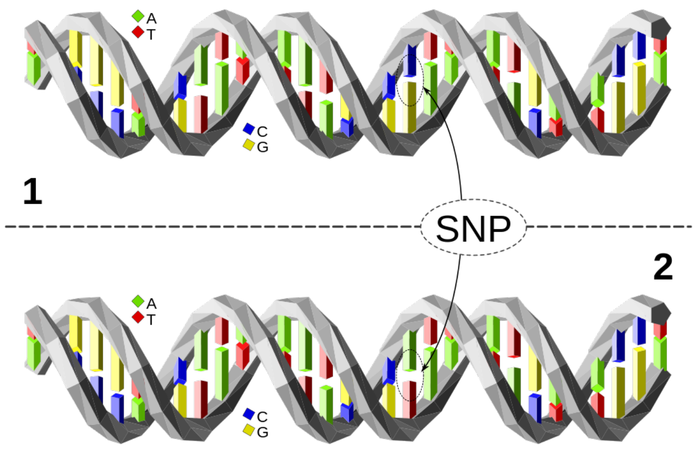

Figure 2.3: A Single Nucleotide Polymorphism is a change of a nucleotide at a single base-pair location on DNA [1]. Author: David Eccles. Creation date: 2014-12-18. Distributed under the GNU Free Documentation License, Version 1.2. Retrieved from `https://en.wikipedia.org/wiki/File:Dna-SNP.svg` on 2018-08-07.

Indel is the abbreviation of "insertion or deletion". An indel is the addition or deletion of bases in the genome. The length of an indel variation is usually 1 to 10000 bp [25]. Unlike SNPs, indels in coding areas almost always change the amino acid sequence of the protein if the length of the indel is not multiple of 3. That is because codons in RNA consist of 3 bases and changes in other lengths than multiple of 3 results in disorder of the rest coding in this gene. For example, the original stop codon might be lost, resulting in longer protein. Or a new stop codon is unexpectedly formed, making the translation end earlier and the protein is shorter. Figure 2.4 is an illustration of an example of an earlier stopped translation caused by the deletion. Because of its significant affect on gene transcription and translation, identifying indels is important to find out the relation between indels and diseases, traits etc [25].

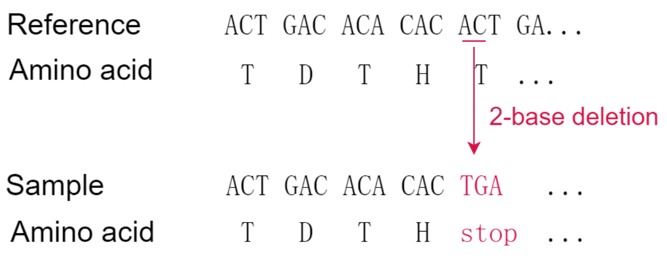
Reference ACT GAC ACA CAC ACT GA...
Amino acid T D T H T ...
2-base deletion
Sample ACT GAC ACA CAC TGA ...
Amino acid T D T H stop ...

Figure 2.4: An example of an earlier stopped translation caused by a 2-base deletion

2.3. GATK RNA-SEQ VARIANT CALLING PIPELINE

In this thesis, the GATK RNA-seq variant calling pipeline (mentioned as GATK pipeline in the rest of this chapter) [16] is chosen to discover the variants from short reads (FASTQ files [17]). An overview of the GATK pipeline is illustrated in Figure 2.5. In this section, the focus will be on the parts that are different from the DNA version.

The STAR [11] aligner is chosen by the GATK team as the first step in the GATK pipeline. The team found that they can achieve the highest sensitivity to both SNPs and indels using the STAR 2-pass protocol [16]. The logic behind 2-pass mapping is simple. In the first mapping pass, splice junctions are discovered and used to rebuild the genome index. Then the new genome index with the splice junctions information is used in the second mapping pass. Compared to the STAR 1-pass protocol, 2-pass mapping discovers much more splice junctions and produces more spliced alignments, unless the 1-pass mapping is guided with external transcript annotations [30], which means the 2-pass protocol is independent of known annotations. STAR, the tool itself is also concluded as doing better than seven other aligners studied in [30].

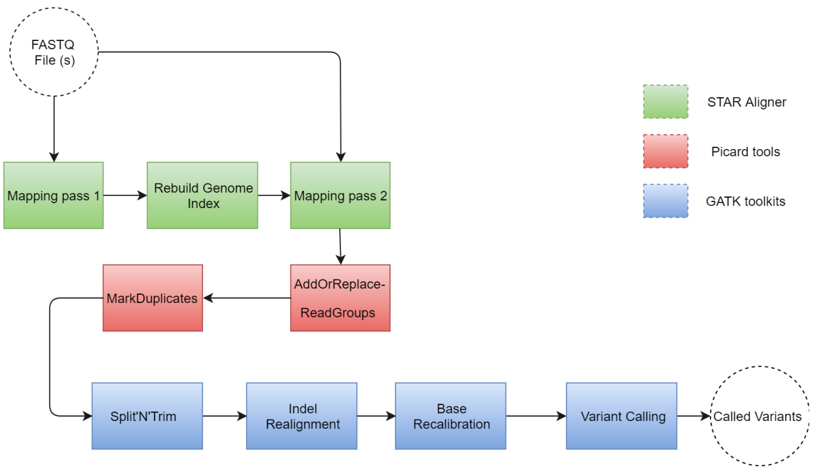

Figure 2.5: An overview of the GATK variant calling pipeline in RNA-seq data

The output of the alignment phase is a SAM file [31], which contains the mapped reads and extra information such as position, mapping quality, etc. Before forwarding the SAM file to GATK toolkits, some preprocessing needs to be done by Picard [19]. AddOrReplaceReadGroups adds the read group information to the reads. Sorting and conversion to BAM file [31] can also be done at this step. MarkDuplicates can tag artifact duplicates to prevent bias in the variant calling process.

As the first step of the GATK processing, Split'N'Trim (tool name SplitNCigarReads) is designed specifically for RNA-seq data. As mentioned in 2.1, RNA reads are very possible to be gapped by intronic regions. In the SAM file, CIGAR string describes how each base in a read is mapped. "M" stands for "match / mismatch", "N" for missed bases on the reference genome, and some other options for other meanings. In an RNA-seq read, "N" is for bases in the missing intronic regions that are not present in the sequence. A noncontiguous read of 101 bp, for example, can have CIGAR string like "46M12144N55M". It can be seen that there are 12144 bases skipped, possibly an intronic region. Some of the GATK tools would not be able to properly deal with those Ns. Hence, Split'N'Trim has to be done before any other GATK procedures. For this reason, SplitNCigarReads would split a spliced read into exon segments and removes the Ns. The second reason why the tool is needed is as follows. Some overhang sequences are very short. For example, in a read "97M3356N4M", there is an overhang of length 4. Even the best aligner can hardly tell that the 4 matched bases are from the other exon, it would, therefore, remain overhanging into the intronic region while actually it belongs to a different region on the genome [32]. This would result in false variants in the variant calling. Thus, any sequences overhanging into intronic regions would be hard clipped by this tool to prevent the false positives. Last but not least, reads with mapping quality 255 mean a good alignment in STAR, but mean "unknown" in GATK, therefore, it must be reassigned at this step to be consistent with the rest GATK tools [16].

Indel realignment performs local re-alignment to correct the mismatching due to indels [33]. However, according to [16], the effect is limited. Therefore, it is an optional step and not used in the final implementation. Base recalibration, on the other hand, is to re-estimate the base qualities empirically and produce more accurate base quality scores. As the base qualities are critical to variant calling, this in turn improves the accuracy of the variant calls [33].

The last step is variant calling, where the tool HaplotypeCaller is recommended by GATK team as it shows much higher true positive rate than UnifiedGenotyper. Also, the team has added new features to HaplotypeCaller to better take care of the intron-exon split regions [16].

2.4. BIG DATA FRAMEWORKS

Big data frameworks are developed for problems where the volume of data to be processed exceeds the capacity or computational power of a single computer. The underlying framework schedules tasks, handles node failure, manages communication etc [6]. The advent of big data frameworks allows programmers to focus on processing the data itself without paying much attention to low-level deployment or management.

Hadoop MapReduce [7] and Spark [8] are two of the most popular big data frameworks. They have a couple of features in common, but still, several differences distinguish them from each other. In this section, we will introduce the main features of the two frameworks, which are useful to know to understand some contents in later chapters.

MapReduce can also be interpreted as a programming model. So here we refer to the big data framework as Hadoop MapReduce [7] and the programming model as MapReduce [6]. Hadoop MapReduce usually consists of the following three components:

- HDFS. HDFS (Hadoop Distributed File System) [34] is a distributed file system designed for data storage in the Hadoop framework. The data is physically stored across the cluster and the names of those data are managed by a dedicated node, NameNode. So data on HDFS can be accessed by any worker nodes in the cluster via network if the data does not present locally.

- YARN. YARN (Yet Another Resource Negotiator) [35] is like a manager dealing with the underlying resource management and task scheduling in the cluster. This is almost invisible to the programmers once it is deployed in the cluster. A programmer only needs to tell YARN how much resources he or she wants for an application.

- MapReduce. The programming model according to which the programmers implement the data processing. This model will be explained below.

In a MapReduce application, the input data is read from HDFS and parsed into <key, value> pairs format. A `Map` function written by the programmer takes these <key, value> pairs as input and produces intermediate <key, value> pairs. These new pairs are written back to HDFS (physically on local disk) so they can be accessed by other worker nodes where the `Reduce` function will be run. These new pairs are shuffled and all the values with the same key are grouped together. Then, the <key, list(value)> are processed by the `Reduce` function and the final results, i.e., output of `Reduce` function are written back to HDFS. Like `Map`, the `Reduce` function is also written by programmers. Figure 2.6 is a simple illustration of how a MapReduce application is executed.

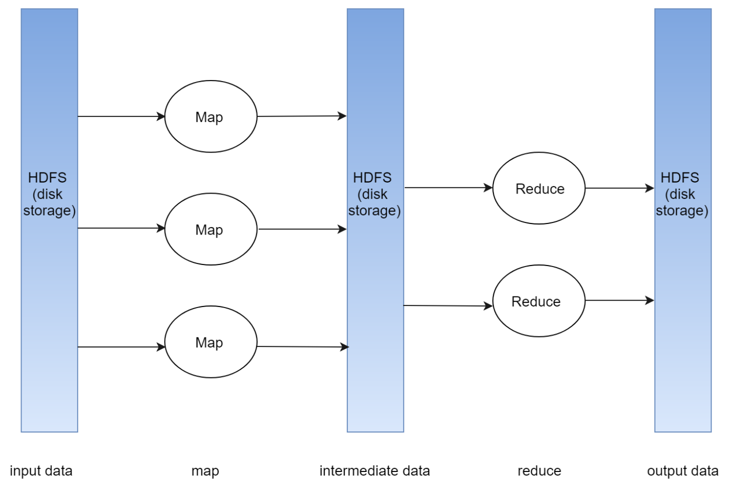

Figure 2.6: Execution flow of a MapReduce application

From Figure 2.6 we can see that MapReduce relies heavily on the disk storage. The buffered intermediate pairs are periodically written to HDFS (disk storage). When the amount of `Map` tasks increases, the disk access overhead would be large, and the execution would be very slow.

What's more, when programming with the MapReduce model, there is a limitation that the only transformations provided are `Map` and `Reduce`. If other kinds of transformation are desired, the programmer has to play some tricks with these two functions. Plus, usually a map phase has to be followed by a reduce phase, meaning one cannot execute tasks like "map-map-reduce" or any other order. These restrictions make development difficult.

One other challenge for the programmers to use MapReduce is that one has to implement a runner class to configure the MapReduce job before it can be launched. In the runner class, everything of the job is set manually: the mapper class, the reducer class, type of the input file, type the intermediate keys and values, type of the sorting and grouping comparators, type of the output keys and values, etc. It can be seen that programming with the MapReduce model is not flexible and the learning process is not trivial.

On the other hand, Spark, a more recently developed big data framework, addresses the concerns stated above. Spark can hook into the Hadoop environment, serving as the substitution of MapReduce in the three components presented earlier in this section. That is to say, it can work with YARN and HDFS, which is easy for programmers who are used to Hadoop MapReduce to get started with Spark.

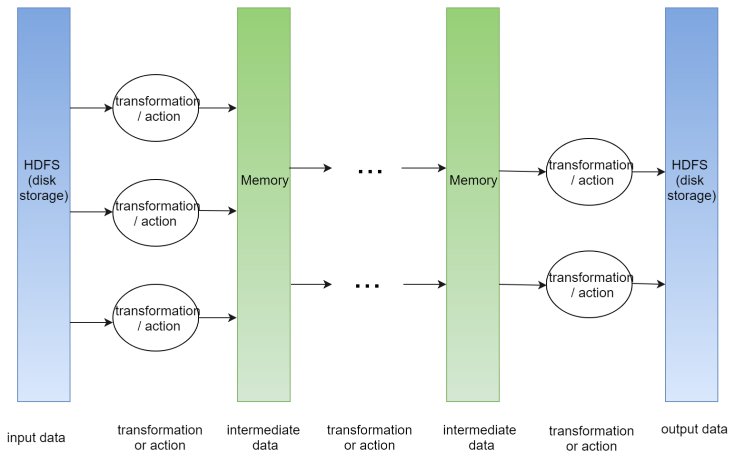

Figure 2.7: Execution flow of a Spark application

Several main features distinguish Spark from MapReduce. First, unlike MapReduce, Spark keeps all the intermediate results within memory, as shown in Figure 2.7. A Spark application accesses HDFS only when reading the input and writing back the final results. It would store intermediate data to disk only if the data cannot fit in the memory. This enables it to process the data much faster than MapReduce.

Secondly, Spark provides programmers with a lot more operator functions besides `map` and `reduce`. In an application, they can be used without restriction of order and amount. These operators are applied to a unique form of the input dataset, RDD (resilient distributed datasets). The operators can be categorized into two types, transformations and actions. Transformations can transform the original RDD and returns a new RDD. To name a few transformations, `map`, `flatMap`, `reduceByKey`, `distinct` etc. are often used. More transformations can be viewed in Spark RDD Programming Guide [36]. Among the mentioned 4 operators, only `reduceByKey` requires RDD to be in the form <key, value> pairs, the rest can be applied to RDD of any type of objects. Transformation `map` and `flatMap` both take a function as parameters and apply this function to every element in RDD. Transformation `reduceByKey` also takes a function describing how two elements of the same key can be aggregated. Programmers can execute these transformations in a row as many times as needed and at any order. For example, executing "flatMap-reduceByKey-map" is quite easy. Actions, on the other hand, returns non-RDD types. Some examples of actions are `reduce`, `collect`, `take(n)` etc. As the returned type is not an RDD, actions cannot be executed one after another. But because of the existence of RDD, multiple actions can still be invoked within one application. The following pseudo-code shows how this is possible.

```
rdd1 = parallelize(inputdata)
array1 = rdd1.map.distinct.collect
array2 = rdd1.reduceByKey.map.take(100)
```

In the above pseudo-code, different transformations are performed to the same RDD `rdd1`. Then actions `collect` and `take` are invoked respectively. Action `collect` returns all the elements of the RDD as an array, while `take(100)` returns the first 100 elements in the RDD as an array.

Thirdly, as can be already seen from the pseudo-code above, Spark APIs are very easy and flexible to use. The programmer can perform a series of transformations and action in only one line. The code also becomes more readable than MapReduce. It is clear that how data flows through transformations and actions. As discussed earlier in this section, in MapReduce, before running a job which executes "map-reduce", one has to set up the runner class, mapper class, reducer class, input format, intermediate format, output format etc. in advance. It makes MapReduce cumbersome to implement and hard to read. For example, from the mapper class, one can only see what a mapper would do. To find out the input source or what the reducer does, one has to go to the runner class and check.

From the discussion above, it can be concluded that Spark can outperform MapReduce because of its in-memory computation. Spark is also easier and more flexible to implement, understand, and maintain than MapReduce. Although sometimes it may be claimed that memory is more expensive than disk, in many cases, people can offset this cost by shorter execution time in a cluster where users pay by time. In general, Spark is a preferable choice to MapReduce.

3

ALTERNATIVE SOLUTIONS

In this chapter, several solutions to address the problem of the slow execution of the typical genomic analysis pipeline are introduced. Then the evaluation criteria according to which we select the method are defined. Next, the pros and cons of each method will be compared and discussed. Finally, the selection will be made.

3.1. ALTERNATIVE 1: BUILT-IN MULTI-THREADING IN THE TOOLS

The first solution is to use the built-in multi-threading [18] function in the tools themselves. Some tools in the GATK Best Practices RNA-seq variant calling pipeline (referred to as the GATK RNA pipeline in the rest of this chapter) support it, while others do not. STAR aligner and genome generation both support multi-threading. However, in the Picard and GATK steps, only BaseRecalibrator can be run with more than one thread [19] [37]. To evaluate the performance when the number of threads varies, we tested the variation of execution time with increased CPU cores on a Hyper-Threading enabled 20-core node. The number of threads is adjusted from 1 to 40 to see the scalability. The results are shown in Figure 3.1.

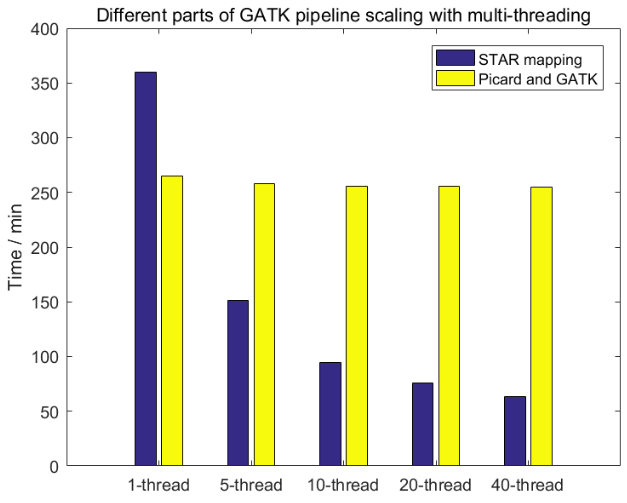

Figure 3.1: The performance of the GATK RNA pipeline with multi-threading

It can be observed that STAR scales much better than the rest of the pipeline, which hardly changes when more threads are applied. In the Picard and GATK parts, only BaseRecalibrator can be run with multiple threads. Figure 3.2 shows the percentage runtime of the different tools for the least overall pipeline time (when 40 threads are used). BaseRecalibrator only takes 4% of the whole time. Its execution time is only reduced from 22.78 min to 14.13 min when the number of threads is changed from 1 to 40.

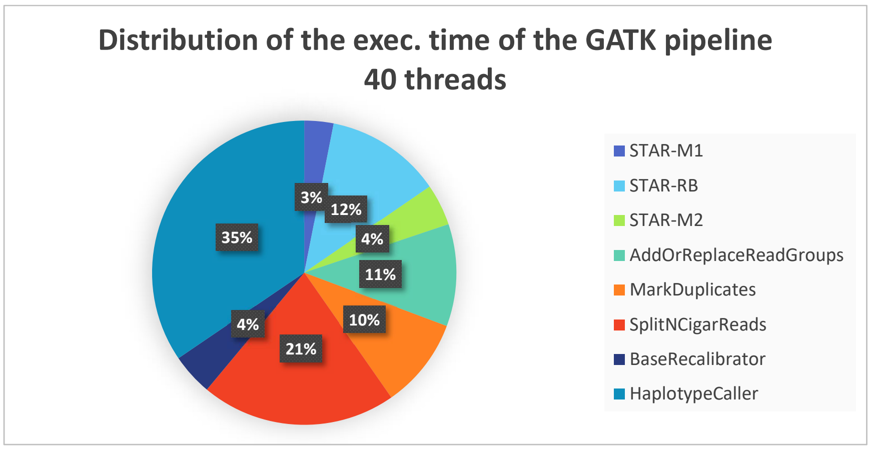

Figure 3.2: The distribution of the execution time of each step in the GATK RNA pipeline. Used threads: 40.

As for STAR, the execution time reduces significantly when the number of threads increased. The variation of time of the three STAR steps is illustrated in Figure 3.3. The flat line after 20 threads can be explained that there are actually only 20 physical cores, thus raising the number of threads to be larger than 20 does not work as expected. However, when the number of threads varies from 10 to 20, the speed only increases by 18.67% on average while when the threads are added from 1 to 10, the speedup is 4.14x on average.

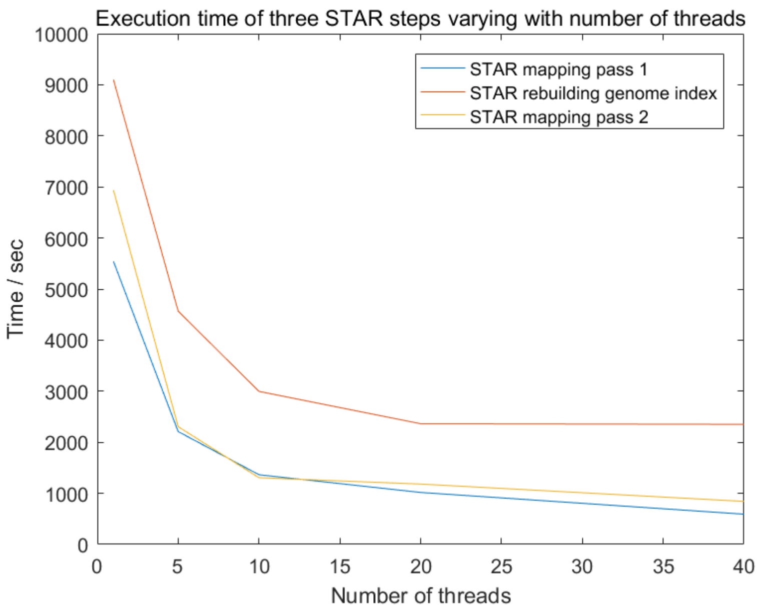

Figure 3.3: Execution time of three STAR steps varying with number of threads

In summary, in the GATK RNA pipeline, though STAR scales well with built-in multi-threading, the rest of the pipeline do not. Plus, even STAR scales slower when the number of threads keeps growing. What's more, the built-in multi-threading only allows the pipeline to scale on a single node. If one has computational power of more than one node, nothing could be done to utilize the computational power of the other nodes.

3.2. ALTERNATIVE 2: CUSTOMIZING THE PIPELINE IN AN INTEGRATED WAY

Churchill [4] is a parallel solution meant for scaling up the GATK Best Practices DNA variant calling pipeline [3] (referred to as the GATK DNA pipeline below). It integrates the GATK DNA pipeline, performs customized processing on the outputs of the tools to make the most use of computational resources. It can be run on a single multi-core node or in an HPC cluster environment.

In the alignment part, raw input sequencing reads (FASTQ format) are first partitioned to FASTQ chunks, then multiple instances of the aligner run in parallel taking these chunks as input. The output SAM files are converted to BAM files. Supposed that *N* BAM files are produced by the alignment part, and Churchill further splits each BAM file into *M* subsets, corresponding to *M* chromosome subregions, then totally $M \times N$ BAM files are generated. In this process, if two reads in a pair would be split into different subregions, both of them will be stored in a separate BAM file called `chrI.bam`, which is dedicated for reads at the border of subregions. This is one of the places where Churchill tries to keep the data integrity while partitioning data. Then, for each of the *M* chromosome regions, the corresponding *N* BAM files are merged, yielding *M* BAM files in total. This complicated split and merge have to be done because FASTQ chunks may contain reads from different chromosomes and the output *N* raw BAM files cannot be forwarded directly to the next processing step. In fact, this can be done naturally by shuffling and sorting in a big data framework.

Then `chrI.bam`, the BAM file containing reads at the border, is processed by Picard MarkDuplicates, the output BAM file is again split into *M* subregions `chrI_region0.bam` to `chrI_regionM.bam` and they are merged with the *M* BAM files mentioned in the last paragraph. Then the new *M* BAM files are processed in parallel by MarkDuplicates and local indel realignment. This way, duplicates of paired reads at borders can also be marked and removed.

Then the pipeline comes to the base recalibration step. In the current version (GATK 3+) and also in GATK Best Practices [3], the output of the GATK tool BaseRecalibrator, a recalibration table, is used for base recalibration. The recalibration can be applied to the reads by either subsequently running the tool PrintReads or inputting the recalibration table to the variant caller (e.g. HaplotypeCaller). But in Churchill, CountCovariates plus TableRecalibration tools are used to recalibrate base quality score. The two tools were deprecated in GATK version 2.0 in 2012 [38]. It is not trivial for Churchill developers to migrate from the older version. Because in their pipeline, using this two tools is a desired optimization of data integrity. The outputs of parallel instances CountCovariates are CSV files containing covariate results. These files are merged to a single CSV and then fed to all the TableRecalibration instances so every instance knows the covariate results from the entire genome instead of only its own subregion. This is the second place where Churchill tries to keep the integrity of data.

After base recalibration, variant calling is also done in parallel and the output VCF files are combined into the final single VCF file as the output of the pipeline.

Churchill pipeline is tightly integrated. It optimizes the way how data is parallelized by taking care of reads spanning chromosomes or sub-regions after division and merging information from the entire genome range (e.g. `chrI.bam` and merged CSV file). But some of the optimizations are done using deprecated tools and in fact do not comply with GATK Best Practices. In summary, Churchill is tightly integrated to achieve high utilization of the resources as well as high data integrity (maintaining read pairs at the division borders and keep some information genome-wise). But, on the other hand, it is also difficult to replace some components in it. The programming difficulty to customize the GATK DNA pipeline is also high. The resulting pipeline is not generic and cannot be adapted easily to other analysis pipelines such as the GATK RNA pipeline.

3.3. ALTERNATIVE 3: SCALING UP WITH MAPREDUCE

Halvade [5] [39] is another solution trying to address the scalability problem of both GATK DNA and RNA pipeline [3] [16]. It is based on the big data framework Hadoop MapReduce. To be consistent with Section 2.4, here we also refer to the big data framework as Hadoop MapReduce [7], including HDFS [34], YARN [35] and MapReduce [6]. The programming model, also the third component in the Hadoop framework mentioned above is referred to as MapReduce.

The MapReduce programming model has been introduced in Section 2.4. One MapReduce application typically consists of a map phase and a reduce phase (some applications may be map-only or reduce-only). Halvade-DNA [5] comprises 2 MapReduce applications. In the first application, the map phase corresponds to read alignment. Output of all the map tasks are <key, value> pairs with the chromosome region information in the key and the aligned SAM records in the value. The pairs are then sorted and grouped by the key and become <key, list(value)> pairs. The reduce phase corresponds to the variant calling part. Reduce tasks take the <key, list(value)> pairs as input and process the SAM records of chromosome regions in parallel. The output of the reduce phase is called variants, i.e. the VCF files of each subset of data. The second MapReduce application combines the VCF files into a single VCF file. This VCF file is also the final output of the entire pipeline. An overview of the framework of Halvade-DNA is shown in Figure 3.4. Halvade-RNA [39] scales the GATK RNA pipeline as presented in Section 2.3. It has 3 MapReduce applications. The map phase in the first application is corresponding to STAR mapping pass 1 while in the reduce phase, genome index regeneration is done. Output of the map phase in this application is <key, value> pairs containing splice junction information with the position information of the junction in the key and the splice junction file text line in the value. In the reduce phase, all the splice junction information is first merged into a single file. Then this file is used in the genome index regeneration. The second MapReduce application is similar to that of Halvade-DNA, only the map phase is corresponding to STAR mapping pass 2. The third application does the combination of VCF files, just the same as in Halvade-DNA.

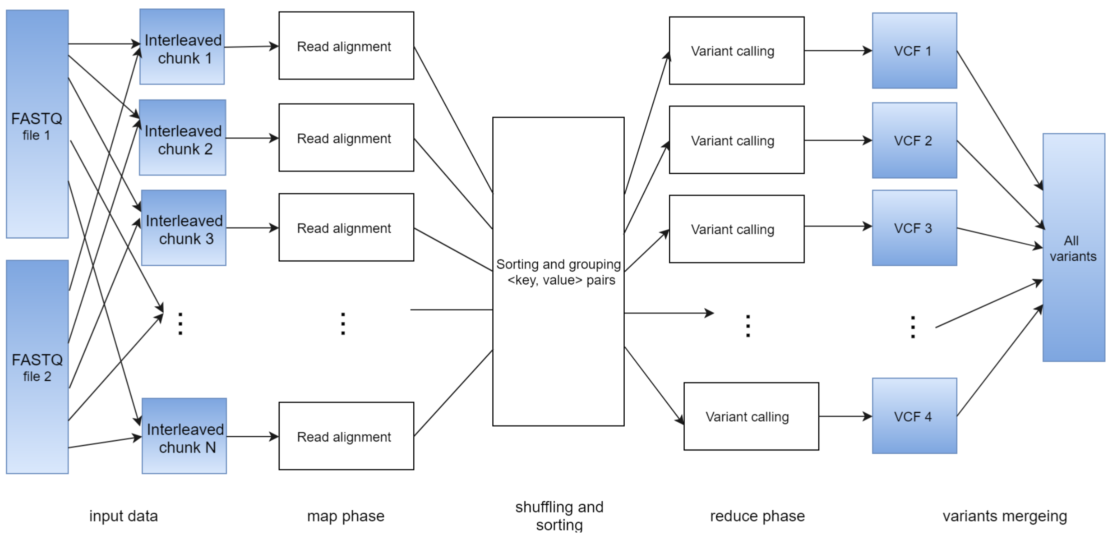

Figure 3.4: Overview of Halvade-DNA

Compared to Churchill, Halvade is more flexible regarding component replacement as it does not perform optimizations by changing the workflow of the GATK pipelines. Plus, Hadoop MapReduce framework takes care of the SAM sorting and grouping in between the map phase and reduce phase. However, this is implemented in a rather complicated way in Churchill.

Halvade is more flexible in terms of updating the components than Churchill, some parts of the pipeline can also be better dealt with in the big data framework. However, the programming model of MapReduce is still cumbersome as discussed in Section 2.4. MapReduce provides only `Map` and `Reduce` APIs. The programmers need to make quite some efforts if any other kinds of transformations are desired. Before one MapReduce application can be launched, input data format, intermediate key and value format, output format, mapper class, reducer class etc. need to be configured in a runner class. Even a small MapReduce application like combining the VCF files into one needs such configuration. The execution order is also fixed. What's more, intermediate data is stored on disk, which leads to slow data processing.

3.4. Alternative 4: Scaling up with Spark

SparkGA [2] also adopts the big data framework to scale up the GATK Best Practices DNA pipeline [3]. Unlike Halvade, it is based on the Spark framework.

In one Spark application, Spark transformations can be used as many times as needed in any order if applicable. SparkGA consists of 3 parts. The first part is read alignment. FASTQ chunks are processed by multiple map tasks in parallel and the output SAM stream is formatted in <key, value> pairs for the later sorting and grouping. In the second part, SAM streams are sorting and grouping, then SparkGA performs load balancing, dividing the SAM files to approximately equal size. The SAM files are then converted to BAM files also in this part. In the third part, data preprocessing and variant calling are done for each BAM file and then the corresponding VCF files are produced. Then the VCF files are combined into one as the final output. The whole program can be started from either of the 3 parts given the required dataset is ready. An overview of SparkGA can be seen in Figure 3.5.

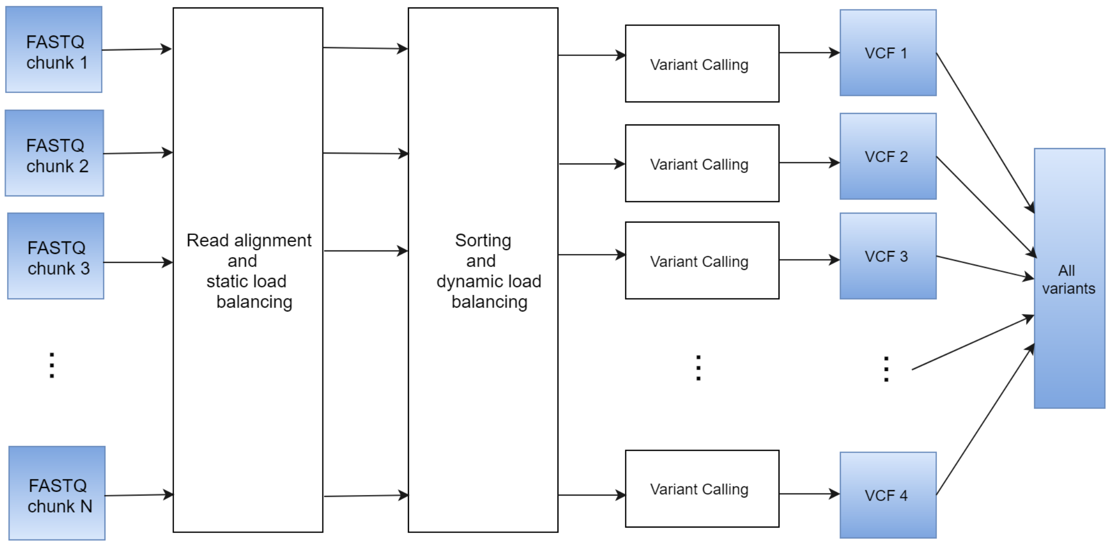

Figure 3.5: Overview of SparkGA

In Spark, there is no explicit limitation that a transformation has to be followed by an action. Therefore, the programmers can easily perform any transformations on the dataset with the built-in APIs. Although due to the lazy manner of Spark, a series of transformations would not be really performed until an action is called. That is to say, the outcome data will only be produced when it is needed. Lazy execution allows Spark not to execute each transformation immediately. Instead, it figures out the best way to get the final results in a lineage graph. Once an action on this dataset is called, Spark executes all the transformations according to this graph at once. This manner also avoids generating unnecessary intermediate data. As Spark keeps the intermediate data in memory, the space is not as ample as in disk storage, lazy execution saves both the computational and memory resource.

Compared to Halvade, SparkGA benefits from the Spark framework in the following aspects. First, Spark keeps intermediate data in memory, resulting in faster processing compared to disk-oriented computation in MapReduce. Secondly, Spark provides more built-in APIs of transformations and actions [36] so that the programmers save a lot of efforts on customizing the `Map` and `Reduce` APIs. Thirdly, the amount and order of used transformations and actions are not fixed. Within an application, they can be used as many times as needed. For example, the VCF combination task needs a separate MapReduce application in Halvade, but is easily done in SparkGA within the same application as variant calling. Last but not least, Spark applications can be written in a more readable way while MapReduce is written in a rather hard-to-understand way because of the nature of the programming model requiring separate job runner class, mapper class, reducer class, etc.

3.5. EVALUATION CRITERIA

To select a more suitable method for scaling up the GATK RNA pipeline, several evaluation criteria are defined in this section. Then in the next section, discussion of the alternative solutions according to these criteria will be presented. The evaluation criteria are defined as follows:

- Ease of implementation
- Maintainability
- Extensibility
- Performance
- Correctness of the results

Now, these criteria will be explained one by one. Ease of implementation includes the ease of using the framework itself and the complexity of the way to scale up the original pipeline. Maintainability here is about the readability of the code or ease of understanding the code. Extensibility concerns whether the parts in the implementation can be easily replaced so it can make use of newer versions of the tools or enable new functions such as RNA analysis. Performance is about the speed of execution and resource utilization. Correctness of the results is obviously the correctness of variants discovered by each solution.

3.6. DISCUSSION

In this section, Alternative 1 to 4 are compared in terms of the criteria defined in Section 3.5. At the end of this section, selection will be made to implement the scalable GATK RNA pipeline.

An overview of the differences among the 4 alternatives is presented in Table 3.1. Items in blue, i.e., the first 5 items, are relevant to ease of implementation, maintainability, and extensibility. Items colored to teal, i.e., item 6 to 8, are relevant to performance while the last green item is about the correctness. But there are also items that are about more than one criterion. Criteria and the corresponding comparison items are listed below.

- Ease of implementation: programming language; framework; cluster environment; availability; whether to customize the original pipeline; SAM sorting and grouping.
- Maintainability: programming language; framework.
- Extensibility: whether to customize the original pipeline.
- Performance: framework; basis of load balancing; parallel efficiency.
- Correctness of the results: special processing of data integrity.

One thing to notice is that the comparisons in Table 3.1 are all based on the techniques that the solutions use and the architecture of the solutions. The only experimental results there is the parallel efficiency. But the parallel efficiency of Churchill, Halvade-DNA, and SparkGA are from their literature [4] [5] [2]. They are not based on the same set of experiments. The numbers can only give a first glance of comparison on the parallel efficiency of these solutions. Therefore, we add another set of numbers in Table 3.2. Those numbers are from SparkGA paper [2], and they are results based on the same dataset and the same platform.

First, for ease of implementation, the native multi-threading can be achieved trivially. Apart from it, SparkGA should be the best among the rest three. Because Churchill is not open-source, the framework of it is not known. Comparing SparkGA and Halvade-DNA, Spark is easier to use than MapReduce as we have discussed in Section 2.4, 3.3 and 3.4. In terms of programming language, Scala and Python are more user-friendly than Java. As for the cluster environment, it does influence the implementation method, but it is more up to the programmers' accessible facilities. As for availability, Churchill is not open-source, making it hard to analyze its method to scale up the GATK RNA pipeline. Halvade-DNA and SparkGA do not customize the original pipeline, which makes them easier to implement than Churchill. Besides, big data frameworks can naturally perform sorting and grouping between read alignment part and variant calling part, which omits the complicated implementation done in Churchill.

Secondly, maintainability or readability is not applicable for the native multi-threading solution. Among the rest three, Python and Scala are again better than Java. As for the framework, as discussed in Section 2.4, 3.3 and 3.4, Spark is much easier to understand than MapReduce. Therefore, SparkGA is also a preferable solution in this aspect.

Thirdly, for extensibility of the solutions, for the native multi-threading method, it is quite flexible for the users to replace any components. Halvade-DNA and SparkGA also leave the tools of the original pipeline as they are. As discussed in Section 3.2, Churchill has customized the original pipeline and optimized with deprecated tools, making it more difficult to change components in the pipeline or adapt it to another pipeline.
Next, the scalability of the 4 solutions can be compared by the parallel efficiency. Although benchmarked

Table 3.1: An overview of the differences among alternative solutions.

Alternative	**Native multi-threading**	**Churchill**	**Halvade-DNA**	**SparkGA**
Programming language	-	Python & Bash	Java	Scala&Java
Framwork	-	Not known	MapReduce	Spark
Cluster environment	Not supported yet[1]	HPC	Big data	Big data
Availability	Built-in	Download on permission	Open-source and free for download	Open-source and free for download
Customizing the original pipeline	-	Yes	No	No
SAM sorting and grouping	-	Dedicated implementation	Built-in processing by framework	Built-in processing by framework
Basis of load balancing	-	Real data size	Static chromosome size	Real data size
Parallel efficiency[2]	9.42% (1.88x, 1 core to 20 cores)	54.17% (13x, 8 cores to 192 cores)	92.05% (18.11x, 18 cores to 354 cores)	71.32% (2.71x, 100 cores to 380 cores)
Special processing of data integrity	-	Yes	No	No

in different experiments, a general idea can be obtained through these numbers. The native multi-threading method scales slowly with the increment of cores. The efficiency result of native multi-threading is calculated from the experiments in Section 3.1. Halvade-DNA achieves the highest efficiency, but when it comes to the absolute performance, it is slower than SparkGA in the same experiment, which can be viewed in Table 3.2. Another comparison can be taken into consideration for Churchill, Halvade-DNA, and Spark. It is the basis according to which the load balancing is done. Churchill and SparkGA both balance the according to the real data size in this execution while Halvade-DNA only divides the load according to the static chromosome size in the reference genome, which might not be identical to the real read counts in each chromosome. Unbalanced load can influence the performance of variant calling part. Framework is also an important factor that may influence the performance. Spark is recognized to outperform MapReduce because of its in-memory computation, which is much faster than the disk-oriented computation of MapReduce.

Last but not least, we take a look at the correctness of the results. Churchill, Halvade, and SparkGA all achieve parallelism based on the observation that the read alignment is independent of other reads and variant calling in one chromosome region is conceptually parallel to other chromosome regions. They divide the data into subsets so many instances can be run in parallel on different subsets of the data. Because the native multi-threading method does not partition the data, the correctness is seldom influenced. In the same set of experiments as in Section 3.1, the original pipeline running with 20 cores achieves the sensitivity and precision of 99.97% and 99.97%, respectively, with the result of the single-core execution as the baseline. The results of the original GATK pipeline is also used as the baseline of the variants called by Halvade-DNA and SparkGA. The correctness results of these two solutions are shown in Table 3.2. It can be seen that both of them achieve a high rate of coincident variants with the baseline. Churchill, on the other hand, takes care of the data integrity and claims that it produces deterministic variants and it is even more accurate than the GATK original pipeline itself because of GATK's default use of downsampling [4].

[1] As of August 2018, Spark version of tools in GATK4 [40] is still in beta stage (uncompleted).
[2] The efficiency of Churchill, Halvade-DNA and SparkGA is based on the results in the literature of these alternatives. They are not results of the same dataset.

To be concluded, among the three scalable alternative solutions, i.e., customizing and integrating the orig-

Table 3.2: Performance and correctness comparison between Halvade-DNA and SparkGA. The results are from SparkGA publication [2] and are based on the same dataset and the same resources.

	Halvade-DNA	**SparkGA**
Execution time (min)	171	100
Sensitivity	98.2%	99.6%
Precision	99.18%	99.0%

inal pipeline, scaling up with MapReduce and scaling up with Spark, Spark is preferable in terms of ease of implementation, readability, extensibility. Although results from different experiments show that SparkGA does not scale as efficient as Halvade-DNA, it exceeds Halvade-DNA in terms of absolute performance in the same experiment. Churchill although keeps the data integrity, the pipeline itself is integrated so it is hard for implementation, component replacement and upgrading. This thesis project selects the Spark framework to scale up the GATK variant calling pipeline for RNA-seq data because of its excellent in-memory calculation feature, user-friendly programming language and APIs. The solution SparkGA which uses Spark shows equal or better scalability, performance and correctness when scaling up the GATK DNA pipeline, compared to the other alternatives. This is also an important consideration when making the selection.

4 IMPLEMENTATION

In this chapter, the implementation details are presented. First, the parallel model is shown to form a general idea of the architecture. Next, important techniques used in the implementation are introduced. Finally, the important features of the implementation are summarized.

4.1. PARALLEL MODEL

Before looking at the parallel model, the data flow of the original GATK pipeline (mentioned as "the original pipeline" in the rest of the chapter) is first shown in Figure 4.1. This way, it is easier to understand how the data is controlled in the parallel mode.

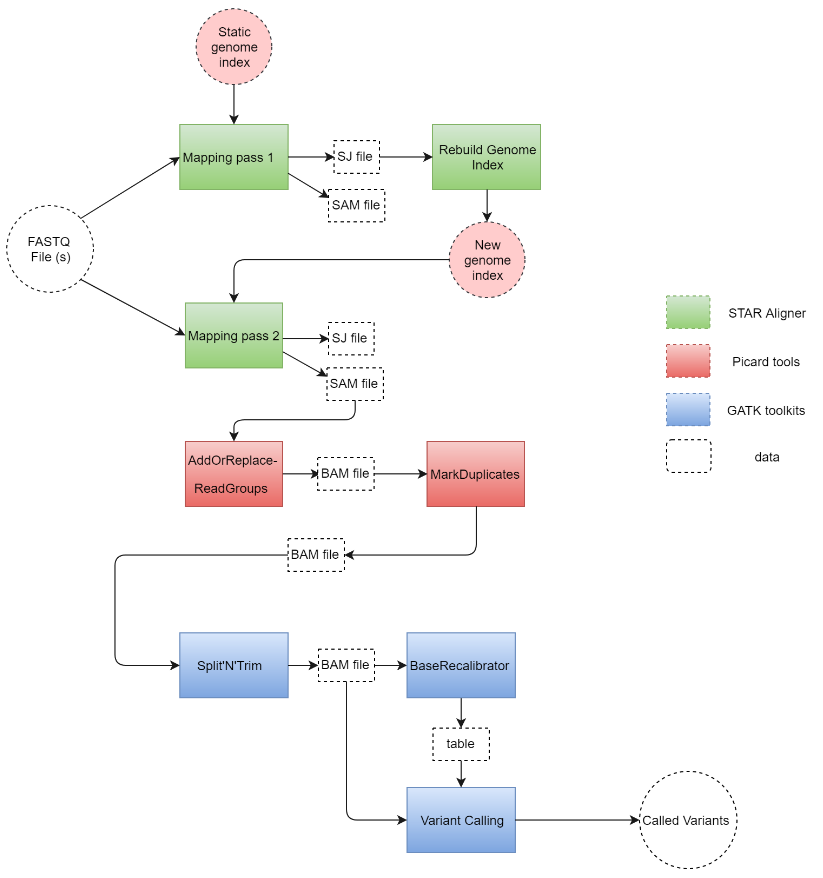

Figure 4.1: Data flow in the original GATK pipeline

In the beginning, reads are inputted to the pipeline as one FASTQ file (or two if paired-end reads). Then a file named SJ.out.tab containing the splice junction information is produced by the first mapping pass, while the generated SAM file at this step is discarded. The SJ.out.tab file is used as a guide in the rebuilding of genome index and a new genome index with splice junction information is referred to in the second mapping pass. The SAM file outputted at this phase is forwarded to the next stage: Picard preprocessing. The SAM file is processed and converted to BAM file for better performance in the rest of the pipeline. At base recalibration step, a report is first generated and then it is used to guide the rewriting of base qualities in the BAM file, which is done in the variant calling step. Finally, the found variants are written in a VCF file [41] and the pipeline ends.

As stated before, the basic logic behind the framework is to create multiple instances of the tools to process the data partitions simultaneously. More specifically, the instances are created across the cluster and run on different nodes, which is done by the Spark APIs.

From the results in Section 3.1, it is known that for the Picard and GATK tools, as many instances as possible should be created in the parallel model, and one thread for an instance is enough. As for STAR, giving one STAR instance around 10 threads has already achieved almost the best that multi-threading can reach according to Figure 3.3. Assigning more than that is a waste of resources. When it comes to the parallel scenario, this number may be further reduced to fit the real need. To cooperate with the instances, the data must also be divided and inputted as chunks. Figure 4.2 gives an idea of this scalable model. Before starting the pipeline, the FASTQ files are divided into chunks of nearly equal size. The reference genome and the static version of genome index have to be put somewhere every task can access. Then the chunks are processed in parallel by a certain number of STAR instances. Next, as the genome generation cannot be parallelized, the splice junction files must be merged into one. The genome generation is subsequently run and the output new index is again put to a shared place. The output SAM strings of the second mapping pass have to be sorted and grouped by chromosome before flowing to the preprocessing and variant calling phases. Because of the very possible uneven distribution of the size of each chromosome, for better load balancing, bigger chromosomes have to be divided into regions. This will be explained more detailed in Chapter 4. When it comes to the preprocessing and variant calling, things become more straightforward. The output SAM / BAM files of the previous step are taken as the input of the next step, until it reaches the end and the VCF files are produced. Finally, the VCF files are merged into one to generate the final result for further analysis.

ANALYSIS OF THE MODEL

In this model, there are three synchronization points, which make the tasks not fully independent and parallel. The first one is when the splice junction information has to be collected across the tasks. The second point is when the SAM stream has to be shuffled and grouped by chromosome regions. The third is, of course, the merge of VCF files.

Different from the DNA pipeline, where only one-time mapping is needed, the sequential step in between the two mapping passes will definitely influence the performance. This step should be made as short as possible as presented in Section 4.5. Between the second and third synchronization point, the pipeline is fully parallel. Making the task size as even as possible before this stage is thus important as discussed in Section 4.4. The last point is near the end of the pipeline, therefore, it does not influence the overall performance much.

Another problem the model might have is that STAR requires about 27 GB of RAM when doing human RNA mapping job [11] to load the huge genome index files into memory. It removes the genome from memory after execution. Imagine that if a set of small datasets are processed by multiple instances frequently, the loading overhead would be huge enough to cover the performance benefits from parallel execution. To better deploy the resources, one node is commonly assigned more than one instance of STAR, each processes a small dataset. If each of them needs to load the genome once, it would not only limit the number of simultaneous instances, but also increase the time one instance needs to process the data because it has to load and unload the index before and after the execution. It is very time consuming for small and frequent tasks to occupy so much resource each time. This problem will also be addressed in 4.3.

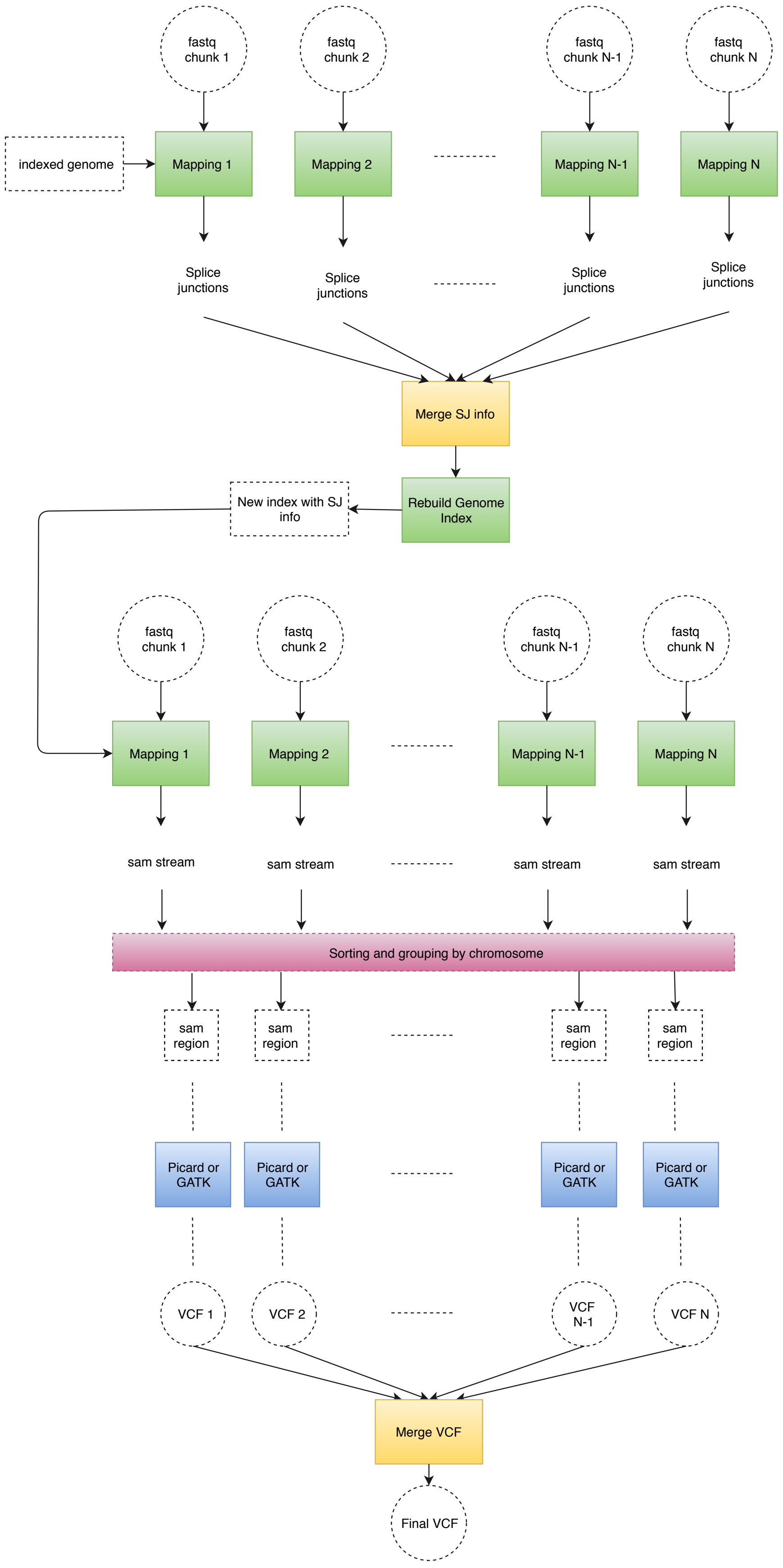

Figure 4.2: The scalable model of the GATK pipeline

4.2. SOFTWARE OVERVIEW

The model presented in Section 4.1 has been implemented as a software program. The program is mostly written in Scala [42], while some utilities in Java and the spark submitting script in Python. The pipeline is separated into three parts. The program can be started from any part provided given that the output files of the preceding part exist. Part 1 does the alignment and static load balancing according to the size of each chromosome provided by the reference genome. Part 2 performs dynamic load balancing to the SAM files according to the real chromosome size of the input data. Part 3 is the preprocessing and variant discovery phase. In each part, the corresponding kernels are mapped to the nodes in the cluster using the Spark API `map`. The returned values of the kernels form a new RDD and further transformations or actions can be easily done.

An XML script is provided for the user to specify parameters regarding paths to required data, cluster configuration, etc. Most of the user-specified parameters will be acquired by a utility named `Configuration` at runtime, while some have to be read before the execution, e.g., the number of Spark executors. The configuration XML script gives the user fine-grained control over the spark parameters, which allows the user to assign the most suitable parameters to different parts to make the best use of the available resources. Table 4.1 is a list of the performance related parameters which the user can tune through the XML script.

The program can be run not only on a cluster, but also in spark local mode, which means one who does

Table 4.1: List of user-specified performance related parameters

Name of performance related parameters	Description
Non Spark parameters	
numRegions	An approximate to the total number of SAM regions
regionsFactor	More SAM regions will be created in part 2 if this is larger than 1.
Spark parameters	
mode	The mode Spark will be run in. It can be yarn-client, yarn-cluster or local.
executorMemGB1/2/3	Memory of each executor. Can be different for each part, same below.
driverMemGB1/2/3	Memory of the driver.
numInstances1/2/3	Number of executors.
numTasks1/2/3	Number of parallel tasks that can be run in each executor.
numThreads1/2/3	Number of thread for each task. Should be $\#executor/\#task$.

not have access to a cluster can still gain a considerable speedup (details can be viewed in Chapter 5) without doing any configuration for cluster or shared file system, e.g. YARN manager and HDFS.

4.3. USING SHARED MEMORY IN ALIGNMENT PHASE

As discussed at the end of 4.1, there is a problem parallelizing the STAR reads mapping, that is, STAR needs to load and remove genome index files to memory to use, which causes big overhead for small and frequent tasks like what will be done in our implementation.

The genome index files, whether in pass 1 or pass 2, once generated, are the same for all runs. Thus, the first thought is to try to load the files only once and make them available to all later runs. Fortunately, STAR provides options to load the genome index to Linux shared memory [43]. Once loaded, shared memory objects will persist until the system is shut down, or no process is attached to it and it is explicitly deleted. In STAR, options of `LoadAndExit`, `LoadAndKeep`, and `Remove` can be chosen when doing read alignment. According to the manual of STAR [44], running mapping with `LoadAndExit` option would simply load the genome index to shared memory and exit without doing the alignment. It attaches the shared memory Id (shmid) to the path to the genome index directory. If the genome at the given path has already been in the shared memory, it will not load it again. The loaded files will remain in shared memory and be ready for later runs. `LoadAndKeep` tells the aligner to access the genome index files from shared memory and leave them there after using. `Remove` explicitly deletes the shared memory segment.

Given these options, the basic idea of the optimization can be formed. Before the alignment tasks are launched, first special tasks with `LoadAndExit` have to be sent to the nodes. Then alignment tasks are deployed to the nodes. After all the tasks finish, special tasks performing the `Remove` are again sent to the nodes to delete the shared memory segments. When there are multiple loading/removing tasks running on the same node, which is often the case, mutual exclusion (mutex) must be applied to avoid several tasks loading/removing at the same time, causing unpredictable results. Here, a lock file mechanism is implemented to function as the mutual exclusion.

The lock mechanism can be viewed in Figure 4.3. The local lock file is in a location where every thread on the machine can access. Before a task does job about the genome index files (loading or removing), it will first check if the lock file for this job is being occupied (busy). If not, the task has to wait until it is free. If the lock file is free, the task obtains exclusive access to it. Then it has to check the content of the lock file. If a certain byte has already been written to the lock file, it means that the loading or removing has already been done so the task will release the exclusive access and exit directly. If the lock file is still empty, the task can start the loading or removing job. After the task finishes, it writes a byte to the lock file, releases the exclusive access and exits.

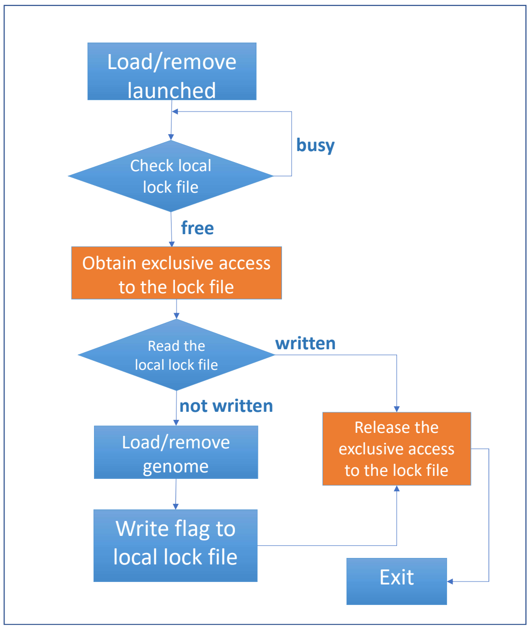
Load/remove launched
busy
Check local lock file
free
Obtain exclusive access to the lock file
Read the local lock file
written
not written
Load/remove genome
Release the exclusive access to the lock file
Write flag to local lock file
Exit

Figure 4.3: Lock file mechanism for loading/removing tasks

As the lock file mechanism forces the other tasks to wait, even if they wait for just a few seconds, it is an overhead and should be avoided to the biggest extent among the parallel instances. In order to reduce the frequency of `LoadAndExit` and `Remove` tasks which require mutex, the Spark API `mapPartitions` is used. Figure 4.4 shows how the loading and removing tasks are sent at the beginning and the end of an alignment pass. From the figure, it can be observed that the number of loading tasks equals the number of numTasks1 (as in Table 4.1, also the M in Figure 4.4). So the number of times that mutex is acquired on one node can be calculated as:

$$\#(mutex - acquire - per - node) = \#(executor - per - node) * numTasks1 \tag{4.1}$$

Removing tasks are launched by another `mapPartitions` to ensure that the shared genome is not deleted before all alignments are done. As opposed, if another Spark APImap is used here instead of `mapPartitions`, we can also implement the loading at the beginning of `STARRunX` method. However, the number of times of acquiring the mutex will be increased to be as much as the number of `STARRun` to be run on that node as shown in Figure 4.5. Its count can be represented as:

$$\#(mutex - acquire - per - node) = \#(STARRun - execution - per - node) \tag{4.2}$$

Comparing Equation 4.1 and Equation 4.2, it can be concluded from experience that using `mapPartitions` acquires less mutex per node than using `map`. For example, in a cluster with 4 nodes, each node is with 56 GB RAM and 8 cores. If we have 100 input chunks (do not consider the paired-end case here) to be processed, approximately every node will run 25 `STARRuns`. Because of the RAM limit, we can assign at the maximum 2 simultaneous alignment tasks (no matter 1 executor with two tasks or 2 executors with 1 task each), each with 4 threads. Then the $\#(mutex - acquire - per - node)$ for using `mapPartitions` will be 2, while this count for using `map` will be approximately 25.

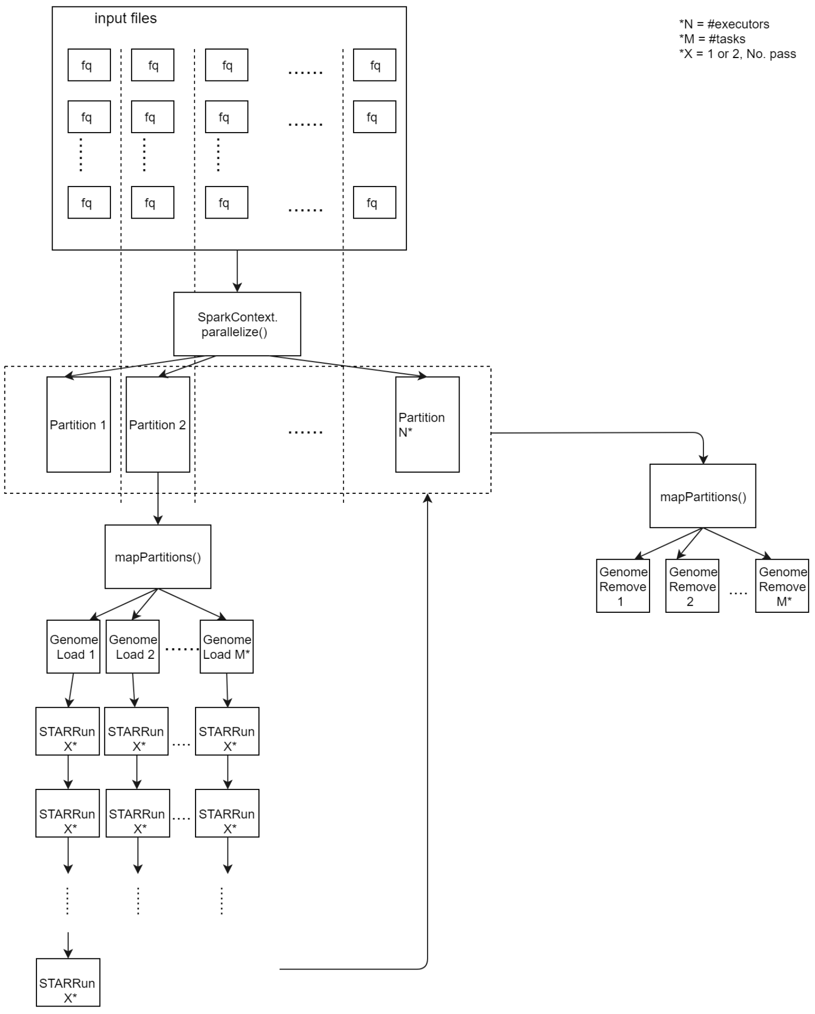

Figure 4.4: Load and remove the genome index at the beginning and the end of each STAR mapping phase.

4.4. STATIC AND DYNAMIC LOAD BALANCING

As discussed at the end of Section 4.1, part 3 (Picard preprocessing and GATK variant discovery part) is run fully parallel. Thus, an even division of the data is crucial to the overall computational efficiency. For example, if a parallel task has to run for 3 hours while other tasks can finish within an hour, most of the computational resources would be idle for 2 hours. In our case, chromosomes that contain more reads than average would lead to unbalanced parallel tasks. Thus, dividing big chromosomes (in SAM format) into smaller chromosome SAM regions before part 3 is what we want to achieve. Here we apply the static and dynamic load balancing method as proposed in SparkGA [2] in our implementation.

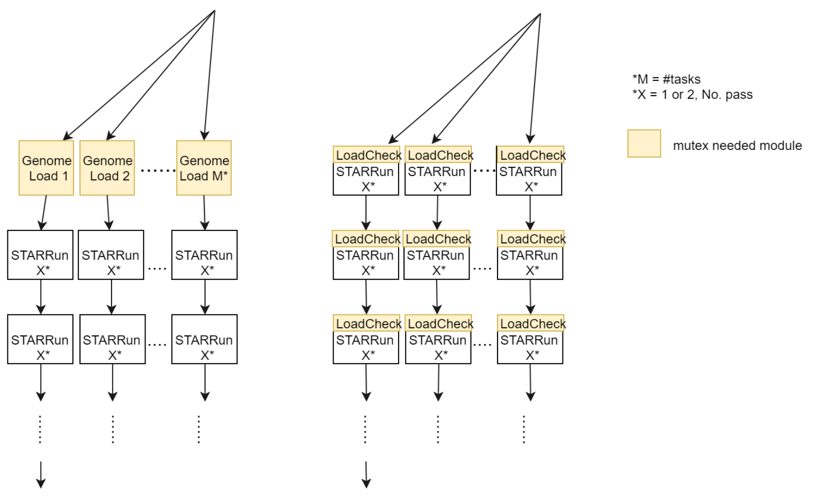

Figure 4.5: Comparison of the mutex needed times between using `map` and `mapPartitions`.

4.4.1. STATIC LOAD BALANCING

Static load balancing is done according to the sequence dictionary file of the reference genome, which is known before runtime. The sequence dictionary file lists the size of each chromosome of the reference genome. The file is parsed to get the sum of the static size of each chromosome ($sizeStatic_{sum}$). Then the average of the chromosome size ($avgSizeStatic$) is calculated as follows.

$$avgSizeStatic = \frac{sizeStatic_{sum}}{numRegions} \tag{4.3}$$

where $numRegions$ is specified by the user to give an expected number of regions as listed in Table 4.1. Supposed that one particular chromosome (e.g. chromosome i) is likely to be divided into as many regions as r_{i_sLB} after the static load balancing, and let $sizeStatic_i$ be the size of this chromosome in the genome, then r_{i_sLB} can be obtained as follows.

$$r_{i\_sLB} = \max(1, \left\lfloor \frac{sizeStatic_i}{avgSizeStatic} \right\rfloor) = \max(1, \left\lfloor \frac{sizeStatic_i}{sizeStatic_{sum}} \times numRegions \right\rfloor) \tag{4.4}$$

In case the size of a chromosome is smaller than average, its number of regions would be set to 1 obviously. The size of one region of chromosome i can be calculated as follows.

$$regionSizeStatic_i = \left\lfloor \frac{sizeStatic_i}{r_{i\_sLB}} \right\rfloor \tag{4.5}$$

All of the above calculation can be performed at the beginning of the part 1 execution because only a user-specified parameter and the information from the reference genome are needed. Then, at the end of the second pass STAR mapping, this region size is used as margins to tag the real SAM reads with different chromosome regions. $ChrPos$ represents the position of the left-most base of a read on the reference genome. This position information is available from the aligned SAM reads. SAM reads are tagged in the form (chr, reg), which indicates their chromosome and region number respectively. In such a tag, chr is obvious and reg is calculated as follows.

$$reg = \left\lfloor \frac{ChrPos}{sizeStatic_i} \right\rfloor \tag{4.6}$$

This process can be viewed in Figure 4.6. In the figure, "region size" means $sizeStatic_i$ in Equation 4.6 Next, SAM reads with the same tag are obviously placed in the same SAM region file. This is how static load balancing works.

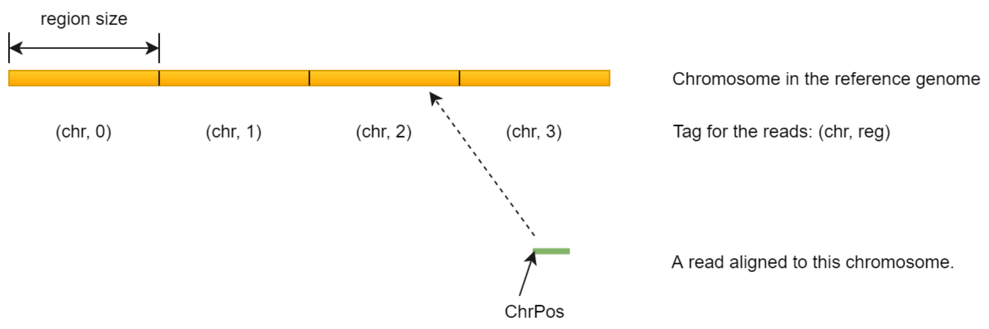
region size
Chromosome in the reference genome
(chr, 0)
(chr, 1)
(chr, 2)
(chr, 3)
Tag for the reads: (chr, reg)
A read aligned to this chromosome.
ChrPos

Figure 4.6: A read tagged with (chr, reg) and placed to its corresponding chromosome region

From the explanation above, some summaries about static load balancing can be made:

- Static load balancing is done at the end of the second pass STAR mapping. The output of the second pass is SAM reads stream as shown in Figure 4.2 and the reads are processed in the way described above. It can help to initially divide the chromosome into smaller regions, and save some time for part 2, dynamic load balancing. Not much extra computation needs to be done for static load balancing, as the calculations in Equation 4.3 to Equation 4.5 are all done at the beginning of part 1 and only Equation 4.6 needs $ChrPos$, which is generated at runtime. Thus, it would improve the performance at a trivial price.

- Let R_{sLB} be the actual number of regions which we will get after static load balancing, then from the equations above, we easily get that $numRegions \neq R_{sLB}$. How $numRegions$ affects R_{sLB} in a real experiment can be viewed in Section 5.1.2.

4.4.2. DYNAMIC LOAD BALANCING

Unlike in static load balancing, in dynamic load balancing, the actual total number of reads ($sizeDynamic_{sum}$) is counted at run time. This is easily and quickly done in spark within one line. The average size of each chromosome region ($avgSizeDynamic$) can be obtained as follows.

$$avgSizeDynamic = \frac{sizeDynamic_{sum}}{R_{sLB}} \tag{4.7}$$

where R_{sLB} is the actual number of regions we get after static load balancing. Next, a chromosome region will be divided into sub-regions if it is bigger than average or if the user-specified $regionsFactor$ multiples its size and makes it bigger than average. The number of sub-regions for a certain chromosome region (tagged as `(chr,reg)`) is calculated as follows.

$$r_{(chr,reg)\_dLB} \approx \frac{sizeDynamic_{(chr,reg)} \times regionsFactor}{avgSizeDynamic} \tag{4.8}$$

where the result on right side is rounded to the nearest integer. When $r_{(chr,reg)\_dLB}$ is smaller than or equal to 1, this chromosome region will not be further divided into sub-regions. The present of $regionsFactor$ allows the user to create more regions during dynamic load balancing instead of during static load balancing. For example, if one wants 300 regions after both load balancing steps, he can specify a combination of: 1. $(numRegions = 150, regionsFactor = 2)$ or 2. $(numRegions = 300, regionsFactor = 1)$. Combination 2 will create more files in static load balancing. The files have to be uploaded to HDFS at the end of part 1 and downloaded again during part 2, i.e. dynamic load balancing. Rather, combination 1 create fewer files and requires less access to HDFS.

Together, static and dynamic load balancing would divide chromosomes that are too big into smaller regions to a user expected extent. Although it is hard to control the exact number of regions that we will finally get, they improve the parallelism and efficiency of the program by creating more tasks of approximate equal sizes.

4.5. OPTIMIZING THE SEQUENTIAL STEP: REGENERATING GENOME INDEX

From Figure 4.2, it is known that regeneration of the genome index is totally a sequential step. When executing regeneration, the rest of the nodes are idle and have to wait until it finishes. Therefore, this step has to be made as short as possible.

STAR uses a suffix array to search during alignment. STAR provides with an option to generate this suffix array in a sparse way [44]. A sparse suffix array can be generated much faster than a normal one. In our experiment, the speed of generating a suffix array of sparsity 8, i.e. distance between indices is 8, is as 6.3x fast as generating a suffix array of sparsity 1 (39.2 minutes versus 6.27 minutes). The price of generating sparse suffix array is the slowing down of the following alignment where the sparse suffix array is used. Here it means that alignment pass 2 is slowed down. However, alignment can be parallelized and available computational resources can be utilized instead of remaining idle. Thus, the overall performance is still good enough.

Furthermore, sparse suffix array is also smaller than the normal one, which also reduces the time needed for uploading it to and downloading it from HDFS. As genome index generation is sequential, it can only be run on a single node. Once finished, the generated index files (including the suffix array) has to be uploaded to HDFS to be later used by all of the nodes. Rest of the nodes need to download them from HDFS as well. In our experiment, the size of the suffix array is reduced from 22.3 GB to 2.8 G when sparsity is 8. The total size of index files is reduced from 26.7 GB to 7.2 GB. The overhead of moving files to and from HDFS is thus reduced significantly, leading to better scalability.

Apart from generating sparse suffix array, it is also important to make sure that the regeneration step is utilizing all the CPU cores of the node where it is running. Unlike alignment, there will only be a single task, i.e. the regeneration step running on the node. If assigning the same number of threads as the alignment tasks to the regeneration task, there will be idle threads on the node, leading to inefficient utilization of the resources. Thus, when the regeneration task runs, it acquires the maximum number of threads it can get on the node.

To be summarized, we optimize the regeneration of genome index mainly by:

- Generating sparse suffix array instead of normal one.
- Instead of assigning the same resource as to the alignment tasks, assigning all the available resources of the node to the regeneration task.

4.6. SUMMARY

This implementation mainly takes care of the data flow in the original GATK pipeline. Creating multiple instances of the tools to process different divisions of data in parallel is the main idea of the implementation. In the process of implementing, however, several technical challenges arise and can be summarized as follows.

- STAR alignment requires relatively huge memory, no matter how small the being processed FASTQ file(s) is, which is challenging for small frequent alignment tasks as we have.
- There is a sequential step, genome index regeneration in the middle of part 1 execution, which influences the efficiency and parallelism of the program.
- Uneven distribution of the size of each chromosome, which leads to inefficient utilization of resources and slows down the execution in part 3.

These challenges have been addressed and illustrated in this chapter. Solutions can be summarized as follows.

- To solve the STAR memory issue, we load the genome index files to Linux shared memory in advance for all the later alignment tasks on the node to use.
- To make the sequential step as short as possible, we only generate a sparse index in regeneration. Plus, instead of assigning the same computational resources as alignment tasks to the regeneration task, full resources on the node are assigned to it.

- To balance the workload of each parallel task in part 3, static and dynamic load balancing are applied in part 1 and 2, respectively. They together divide the data more equally in size and thus make the resources be utilized more efficiently.

Apart from addressing the technical challenges, the implementation also allows the user to tune the Spark configuration parameters (number of executors, executor memory, etc.) for different parts. As tasks in different parts can require very different resources, tuning the parameters differently can make better use of the available resources.

Last but not least, the implementation is also suitable to run in local mode and get quite a good speedup. This means that even if the user only has a single node, he or she can get quite a good speedup without doing any configuration for cluster management or shared file system. All that the user needs is installation of Spark and our implementation.

5 EVALUATION

5.1. PERFORMANCE IN LOCAL MODE

First, the implementation is tested on a single node, running in Spark local mode. This section is organized as follows: 1. Influence on the execution time of varying the number of chromosome regions. 2. Speedup against the original GATK pipeline (referred to as "the original pipeline" in the rest of the chapter). 3. Comparison of execution time with Halvade-RNA [39] (referred to as "halvade" below) on the same node.

5.1.1. EXPERIMENT SETUP

Considering the capability of a single node, a pair of relatively small data sets is chosen, which is publicly available from Encyclopedia of DNA Elements (ENCODE) [45]. They are ENCFF005NLJ and ENCFF635CQM and are about 16 GB each after extraction. The node has 2 Intel processors, each with 10 physical cores with hyper-threading enabled (2 threads per core) and 196GB RAM. The execution of the three different parts of our implementation is configured as follows. For part 1 of the implementation, where we perform alignment and static load balancing, the number of simultaneous tasks is 10, each with 4 threads. For part 2, where we perform sorting and dynamic load balancing, the number of tasks is 20, each with 2 threads. For part 3, where we perform preprocessing and variant calling, 40 tasks with 1 thread for each is set. As for halvade, as long as the available resources are provided, the program will assign the number of mappers and reducers by itself.

5.1.2. VARYING NUMBER OF CHROMOSOME REGIONS

As shown in Table 4.1, the non-spark parameters $numRegions$ and $regionsFactor$ also influence the performance as they affect the number of chromosome regions to be processed in parallel in part 3, i.e., the preprocessing and variant calling phase. As explained in Section 4.4, $numRegions$ will define only approximately the number of chromosome regions created after static load balancing, and $regionsFactor$ will define only approximately the multiplication factor for the number of regions in the dynamic load balancing. Table 5.1 lists the actual number of regions we get after static and dynamic load balancing. In the table, R_{sLB} and R_{dLB} represent the number of regions obtained after static and dynamic load balancing, respectively. The

Table 5.1: Number of regions after static and dynamic load balancing given different $numRegions$ and $regionsFactor$ value.

Index	$numRegions$	$regionsFactor$	R_{sLB}	R_{dLB}
1	180	1	224	304
2	144	2	193	441
3	144	1	193	262
4	108	1	156	217
5	72	2	123	294
6	72	1	123	183
7	50	1	103	161

listed combinations are marked with index 1 to 7 and the execution time of each part of each combination is presented in Table 5.2. In the table, t_1, t_2, t_3 and "Time total" represent the execution time of part 1, 2, 3 and the whole program, respectively. Theoretically, bigger R_{sLB} (created by static balancing in part 1) means fewer regions need to be divided further by dynamic balancing in part 2. Therefore, t_2 should decrease, since there is less effort taken by dynamic load balancing. The results in Table 5.2 support this conclusion. This is further illustrated in Figure 5.1 as a bar diagram showing the relation between t_2 and R_{sLB}. To eliminate other influences, combination 2 and 5 which use the $regionsFactor$ are not plotted in the figure.

Table 5.2: Execution time of each part given different $numRegions$ and $regionsFactor$ value.

Index	t_1 (min)	t_2 (min)	t_3 (min)	Time total (min)
1	39.38	9.33	32.48	81.19
2	39.78	9.27	39.23	88.28
3	39.67	10.30	30.85	80.82
4	40.65	11.17	29.20	81.02
5	39.65	10.43	32.55	82.63
6	39.81	11.40	29.10	80.31
7	40.01	13.8	29.35	83.16

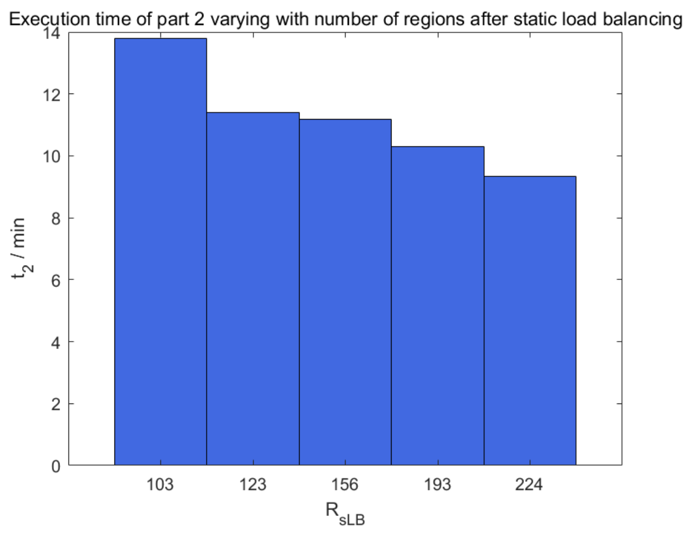

Figure 5.1: Influence of R_{sLB} on run time of part 2 of the implementation

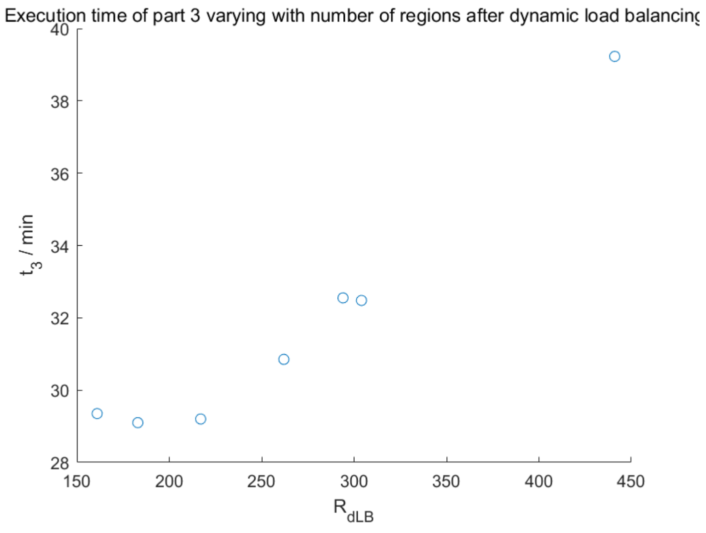

Figure 5.2: Influence of R_{dLB} on run time of part 3 of the implementation

As for t_1, it does not show a correlation with the $numRegions$. This means that the static load balancing, which is done at the end of alignment pass 2, hardly affects the performance. It is done at no cost, but it helps to partition the chromosome into regions roughly and save some time in the dynamic load balancing. When it comes to t_3, it shows a positive correlation with R_{dLB}. This relation can be viewed in Figure 5.2. This seems contrary to the impression that more parallel tasks lead to faster overall execution. However, after looking at the size of the regions in combinations 2 and 7 (where R_{dLB} has the highest and lowest values, respectively), we may get an explanation as discussed next. Figure 5.3 shows the distribution of the size of all region files

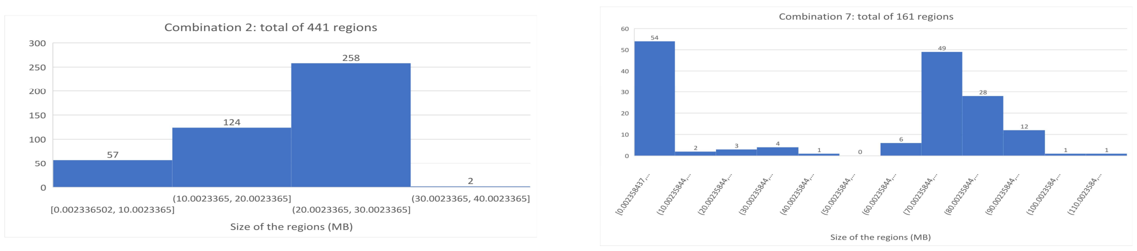

Figure 5.3: Size of the bam file of each region for combinations 2 and 7

(BAM files). In combination 2, almost all the BAM files are smaller than 30MB and 40% of the files are smaller than 20MB. On the other hand, in combination 7, 59% of the files are between 60MB and 100MB. It can be concluded that, if data is partitioned into too small pieces, the benefits brought by parallel computation cannot cover the overhead of launching one parallel task. When the total number of tasks is smaller than or comparable to the parallelism that the machine can provide (i.e., 40 threads at most on this platform), too small tasks may not be a problem. This is because the total execution time would be equal to the biggest task anyway. However, in combination 2, 441 parallel tasks will be launched. It is possible that a task is launched and occupies a computational thread only to process a couple of reads, causing the overall performance to decrease.

It is also noticed that, in both combinations, more than 50 files are smaller than 10MB. This is caused by the very few reads that are mapped to the unknown or random region indexed in the dictionary file of the reference genome. As the program would never put reads mapped to different chromosomes into one region, it is possible that one SAM or BAM file only contains several reads, thus making the file very small.

5.1.3. COMPARISON WITH ALTERNATIVE SOLUTIONS

Based on the discussion in Section 5.1.2, the best performance of our implementation on the single node can be obtained is with combination 3 or 6 with a total run time of around 80.57min. Here we use results from combination 6 in the comparison. As halvade cannot be tuned with user-specified parameters, we provided it with all available computational resources and ran it three times, and the average is taken to be used in our comparison. As for the original pipeline, because of the long execution time, it is only run twice, once with 1 thread and once with 40 threads. Table 5.3 lists the execution time comparison[1] between the original pipeline with 1 thread (origi-1) and 40 threads (origi-40), halvade and our implementation (spark).

A diagram of the speedup of origi-40, halvade and spark execution compared to origi-1 is made in Figure 5.4

Table 5.3: Comparison of the execution time (min) on one node

	origi-1	origi-40	halvade	spark(local mode)
STAR pass 1	92.36	9.87	9.60	10.53
STAR genome rebuild	151.65	39.20	6.70	6.27
STAR pass 2	115.61	14.04	33.02	22.85
Dynamic load balancing	n	n	n	11.40
Preprocessing & variant calling	265.26	255.15	56.75	29.10
Total	624.89	318.26	106.07	80.31

for easy comparison. It can be concluded that our implementation achieves the best performance among the three. The overall speedup reaches 7.78× compared to origi-1. And it is 24.29% faster than halvade. Looking back at Table 5.3, compared to halvade, spark implementation is faster mostly in the mapping pass 2 and preprocessing & variant calling step. In fact, even though the overall pass 1 time is slower, the actual mapping pass 1 is faster. This slowdown is caused by the fact that the loading process of the genome index to shared memory is a little bit slower than halvade. The main reason the mapping step of spark is faster is due to the programming approach of MapReduce, since halvade has to first parse paired-end read lines from the interleaved input FASTQ file and has to create <key,value> pairs and rewrite them to two new files as the input of STAR aligner. In this process, the <key,value> pairs are written to disk and then the FATSQ file lines in the value are written to the new files. But for spark, this step is not necessary. It can simply use the list of the names of input files as its initial RDD and input the files directly to STAR. The second faster part is Picard and GATK processing. Obviously, dynamic load balancing works well and makes part 3 of the pipeline run smoothly. It is almost twice as fast as the same part of halvade. To further support this observation, Figure 5.5 shows the distribution of sizes of the BAM files before being processed by Picard in halvade. It can observed in the figure that halvade launches big parallel tasks for the preprocessing and variant discovery part. The parallelism level is low and the execution is not efficient.

[1]For spark, "Dynamic load balancing" includes SAM to BAM conversion while for halvade it is included in the "Preprocessing & variant calling" steps.

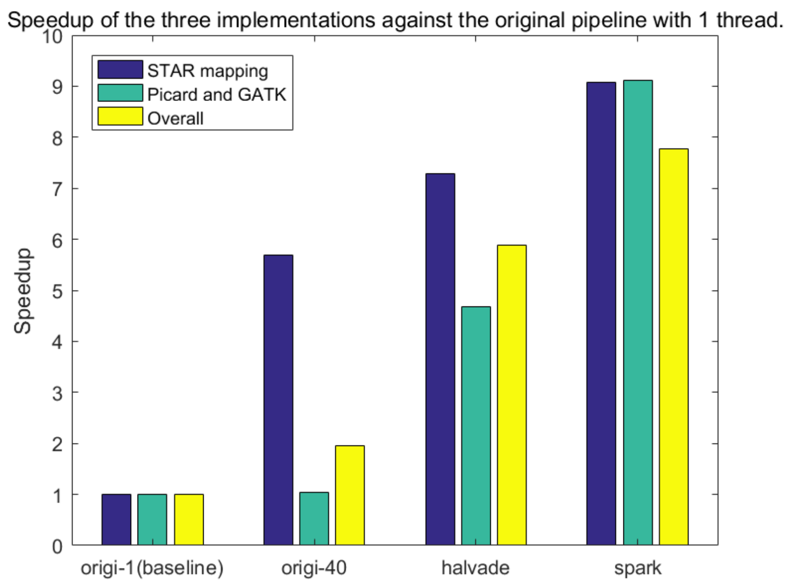

Figure 5.4: Speedup comparison over the original pipeline with 1 thread.

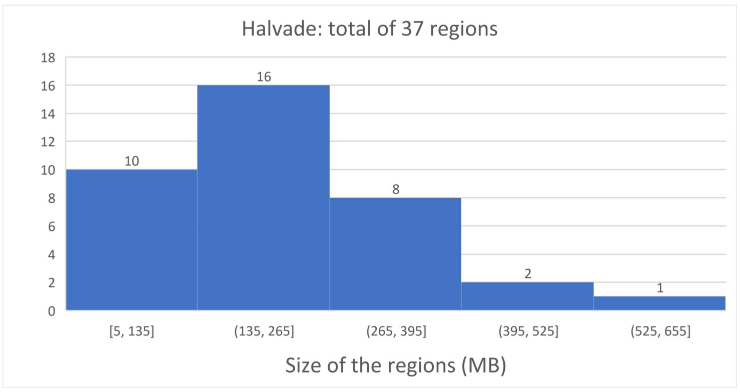

Figure 5.5: Region files size distribution in halvade. Total regions: 37.

5.2. PERFORMANCE IN CLUSTER MODE

In this chapter, experiments are done in cluster mode on SURFsara cluster [46]. First, the scalability of our implementation will be shown by varying the number of nodes in Section 5.2.1. Then the utilization numbers of the resources will be presented in Section 5.2.2. Lastly, a comparison of performance with halvade and the original pipeline will be done in Section 5.2.3.

5.2.1. SCALABILITY: VARYING NUMBER OF NODES

Each node in the cluster has 8 CPU cores and 56 GB RAM. How the total number of CPU varies with the number of nodes is presented in Table 5.4. First, we use the same dataset as used in local mode, that is, the ENCFF005NLJ and ENCFF635CQM paired-end FASTQ files. Both of them are 16 GB in size, giving a total of 32 GB of input data. The scalability is plotted in Figure 5.6.

Table 5.4: Total number of CPU cores changes with number of nodes.

# node	1	4	8	16
# cores	8	32	64	128

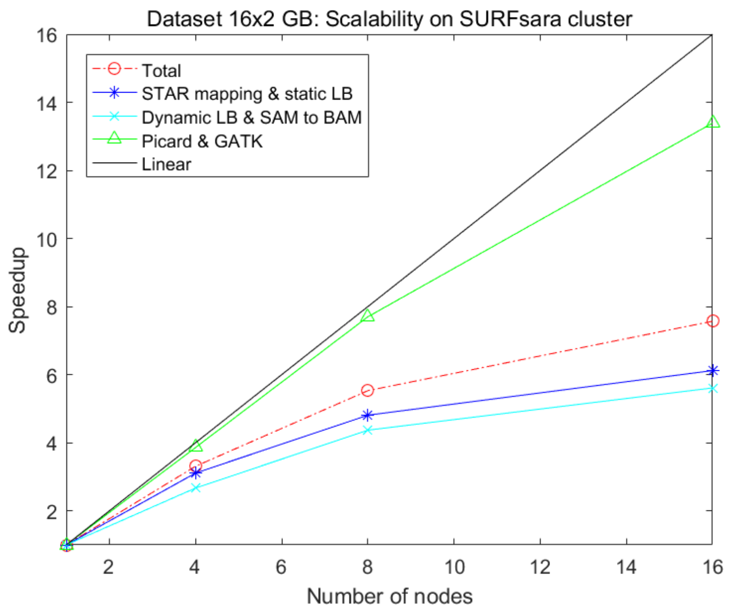

Figure 5.6: Scalability of the implementation when varying the number of nodes from 1 to 16. Data size: $16GB \times 2$.

From Figure 5.6 it can be seen that part 3, i.e. Picard preprocessing and GATK variant calling part scales with the highest efficiency. The speedup of this part reaches 13.39× when the number of nodes is 16. This is in accordance with our expectation in the static analysis of the parallel model in Section 4.1. It can be observed from the model that part 3 is fully parallel and without synchronization point except for the combination point of the VCF files near the end of execution.

Part 1, i.e. the read alignment part, scales slower than part 3. The speedup is only 6.12× when the number of nodes is 16. This can also be explained by the parallel model, which shows there are a synchronization point and a sequential execution (genome index regeneration) in between the first and second mapping pass. The synchronization point and the regeneration definitely influence the parallelism of the program and lead to lower efficiency. As the number of nodes increases, the total execution time decreases, but the sequential execution time remains the same and becomes more dominant, which also makes the curve flatter when the number of nodes is increased. Change of the proportion of the execution time of the regeneration part in the total execution time can be viewed in Figure 5.7.

As for part 2, i.e. sorting, dynamic load balancing and SAM to BAM conversion, it steps the slowest among the three parts. The speedup is 5.61× when the number of nodes is 16. This is because the sorting and dynamic load balancing both rely on shuffling data among nodes, which means the network transport plays an important role in this part. Increased nodes also means more communication and more network overhead. Therefore, it scales slower compared to the other two parts.

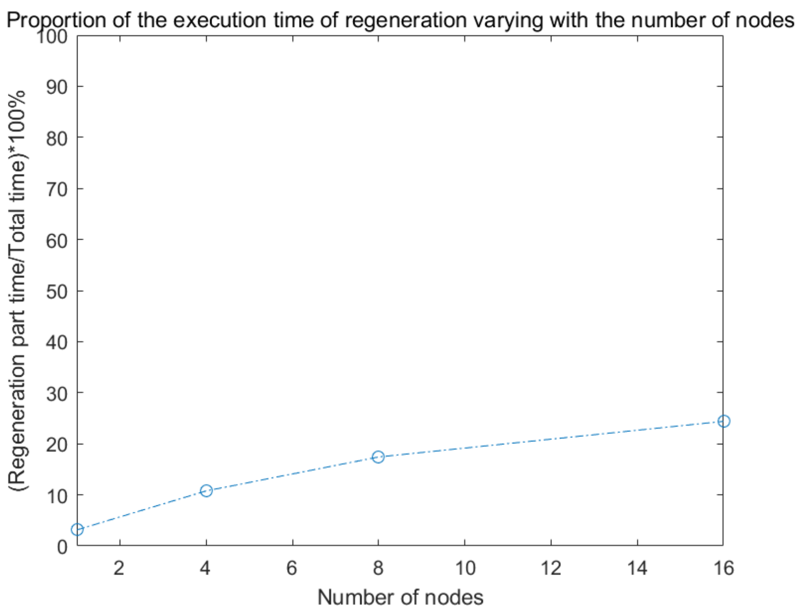

Figure 5.7: Change of the proportion of the execution time of the regeneration part in the total execution time. Data size: $16GB \times 2$.

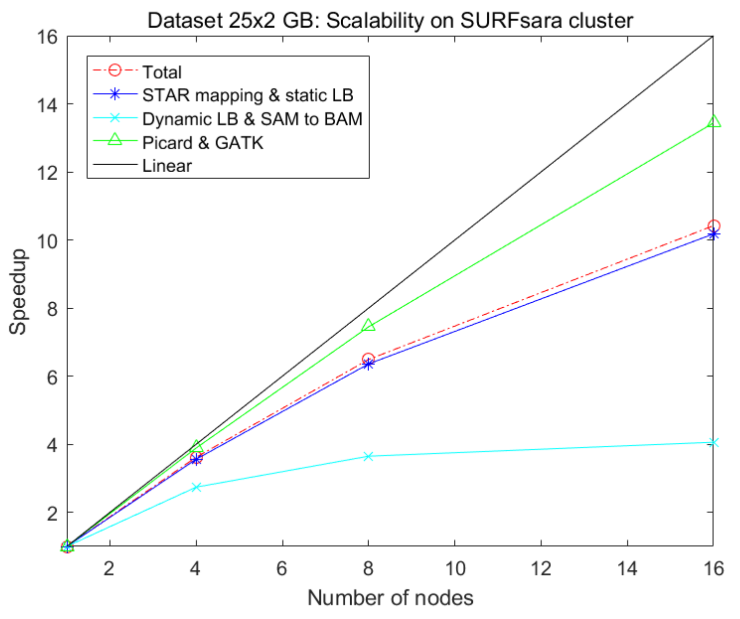

Figure 5.8: Scalability of the implementation when varying the number of nodes from 1 to 16. Data size: $25GB \times 2$.

Next, we test the scalability with a bigger dataset ENCFF174YCI and ENCFF584LJW. They are also paired-end FASTQ files from ENCODE project [45]. They are $25GB$ each after extraction, so there is a total of 50 GB of input data. The scalability is plotted in Figure 5.8. It can be viewed from the figure that part 1, i.e., the STAR read alignment part scales more quickly than the last set of experiments with the $16 \times 2GB$ dataset. When the number of nodes is 16, speedup of part 1 reaches 10.18× while in the last set of experiments, the speedup of part 1 is only 6.12×. Speedup of part 3 remains almost the same (this 13.44× vs last 13.39×). Part 2 scales even more slowly than the last set of experiments (this 4.06× vs last 5.61×) due to more data to be shuffled than in the last set of experiments. The overall speedup is 10.42× while that of the last set of experiments is 7.57×. This means that in the last experiments, the cluster has spare computational power, mainly when executing part 1. The reason of this difference can be explained by the different number of FASTQ chunks. The $25GB \times 2$ dataset is divided into 320 chunks (160 pairs) while the $16GB \times 2$ dataset is divided into 200 chunks (100 pairs).

This means that there are totally 160 parallel tasks in part 1 in this set experiments and 100 parallel tasks in part 1 in the last set experiments.

To be summarized, in cluster mode, when 16 nodes are used, the program achieves an overall parallel efficiency of 65.13%, and the Picard preprocessing and GATK variant calling part achieves a parallel efficiency of 84.00%. The parallel efficiency is defined as follows. If a program achieves a speedup of $S\times$ when running on N workers compared to running on 1 worker, the parallel efficiency E can be obtained:

$$E = S/N \tag{5.1}$$

5.2.2. CPU UTILIZATION IN THE CLUSTER

To better analyze the performance of the implementation in cluster mode, CPU utilization of the worker nodes during the execution is monitored. This is done with the Linux profiler iostat [47]. On a multi-core system, it reports the CPU statistics on average among all the cores. The CPU utilization numbers are reported at 8 seconds interval and are plotted as functions of time.

The two experiments with the input ENCFF174YCI and ENCFF584LJW running with 8 and 16 nodes are profiled and the plots are shown in Figure 5.9 and Figure 5.10, respectively. The plots are the CPU utilization on one of the 8 or 16 worker nodes. In both figures, 3 different statistics are plotted as shown in the legend. "Process" plots the CPU utilization of the computations. "IOwait" plots the percentage of time since the last report during which the CPUs are idle because there are outstanding disk I/O calls. "Idle" plots the percentage of time since the last report during which the CPUs are idle and there is no outstanding disk I/O calls. There are two graphs for part 1 presented in each figure. The upper graph in each figure shows the statistics of the whole execution from the node where genome index regeneration happened, while the lower graph show statistics of part 1 from one of the rest nodes that did not perform regeneration.

It can be seen at a first glance that, the overall CPU utilization is rather high in both part 1 and 3 (blue curves). It means that the implementation balances the load well and utilizes the resources efficiently. Idle CPUs are mostly observed during STAR regeneration and part 2, i.e. sorting, dynamic load balancing and SAM to BAM conversion. In the rest of this section, different stages of the execution reflected in Figure 5.9 and Figure 5.10 will be analyzed to obtain more insights into the implementation.

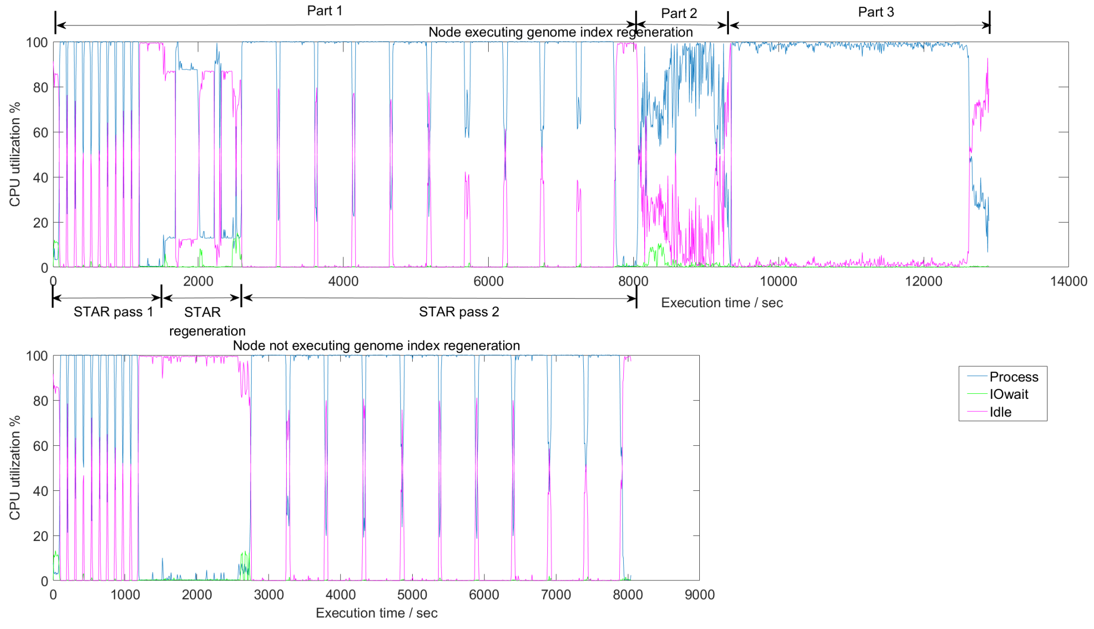

Figure 5.9: CPU utilization of one of the 8 nodes. Data size: $25GB \times 2$.

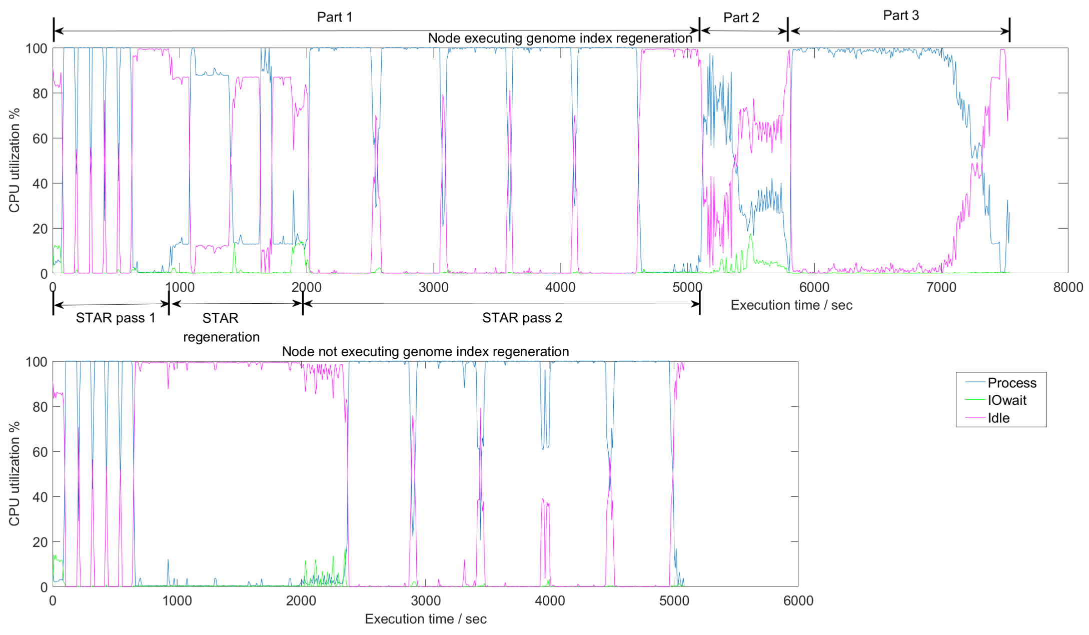

Figure 5.10: CPU utilization of one of the 16 nodes. Data size: $25GB \times 2$.

For part 1, the first thing we can observe is the clear borders between stages mapping pass 1, genome regeneration and mapping pass 2. After mapping pass 1 finishes, splice junction information is sorted, shuffled and finally collected at the driver node. That is why a small portion of "Idle" (pink curve) presents after mapping pass 1 in all the 4 graphs in Figure 5.9 and Figure 5.10.

Then it comes to the regeneration step of part 1. It can be seen that no matter in the 8-node execution or in the 16-node execution, the time of regeneration is around 1000 seconds. Comparing Figure 5.9 and Figure 5.10, it is easily found that the proportion of this part increases in the total execution, which is in concordance with the discussion in Section 5.2.1. From the upper graphs in each figure, it can be seen that although it is a sequential execution, it is provided with all the 8 cores and STAR makes use of multi-threading and still keeps the utilization relatively high. There are two peaks reaching more than 90% utilization. The two-peak pattern results from the way how STAR regenerates the genome index. In both upper graphs, at the end of STAR regeneration, a small green peak lasting about 100 seconds appears. This represents the I/O calls for uploading the new genome index files to HDFS. In both lower graphs, the CPUs just stay idle throughout the regeneration time.

Next, at the beginning of STAR mapping pass 2, in both lower graphs, a period of I/O calls (green curve) is present. This is because, on nodes where genome regeneration is not done, the new genome index files need to be downloaded from HDFS. Similar to mapping pass 1, at the end of pass 2, information about SAM files outputted from each task is collected at the driver node. Therefore, a small portion of "Idle" (pink) is present after mapping pass 2 in all the graphs in Figure 5.9 and Figure 5.10. In Figure 5.10, the lower graph shows that the last mapping pass 2 task finished later than that in the upper graph. This is also because the node reflected by the lower graph has to download the new genome index files. The node reflected by the upper graph, although finishes earlier, has to wait until the other nodes also finishes.

Comparing part 1 execution on 8 nodes and 16 nodes, another interesting observation can be obtained. In Figure 5.9, in both STAR pass 1 and pass 2, there are 10 intervals during which the CPU utilization reaches 100%. Considering our configuration that 2 parallel tasks per node for part 1, i.e., in each interval there are 2 simultaneous tasks running, we can conclude that in each mapping pass, 20 tasks are done per node. Then a total of $20 \times 2 = 160$ tasks are done for each mapping pass. In Section 5.2.1, we have mentioned that we generate 160 pairs of FASTQ chunks for this dataset which completely coincides the measures. It also shows the balance loads in part 1. In Figure 5.10, the number of intervals is halved, showing ideal scalability of the mapping passes.

When it comes to part 2, the CPU utilization is not as high as the other two parts. This is because this part contains sorting and dynamic load balancing, both of which relying rely on data shuffling and I/O activity. The computation load is not as heavy as the rest parts. This is also supported by the rising of "IOwait" plots (green curve) in this part.

Part 3, the Picard preprocessing and GATK variant calling part, is where the highest utilization is achieved. This also makes it achieve the best scalability among the three parts. Each task for this part is completely parallel until the VCF files are generated and combined at the driver node.

In summary, the implementation achieves high utilization in most stages of the execution in a cluster environment. This makes it a scalable solution that can process bigger datasets with more computational resources.

5.2.3. COMPARISON WITH ALTERNATIVE SOLUTIONS

In this section, the performance of our implementation (referred to as "spark" in the rest of the section) and that of halvade will be compared in the SURFsara cluster [46]. The two datasets: ENCFF005NLJ and ENCFF635CQM ($16 \times 2GB$), and ENCFF174YCI and ENCFF584LJW ($25 \times 2GB$) are both tested. As stated before, each node in the cluster has 8 CPU cores and 56 GB RAM. There are 76 active nodes in total at the time the experiments are done. For halvade, it is not possible to control the number of nodes. MapReduce spreads the map or reduce tasks throughout the cluster. Therefore, halvade runs on all the 76 nodes. Our implementation, on the other hand, runs on 16 nodes in the cluster and is still faster for both datasets. Of course, running on 76 nodes introduces more communication overhead than on 16 nodes, but not being able to configure the desired number of nodes is thus also a disadvantage of halvade or MapReduce. What's more, we will show that our implementation (spark) is faster not just because there is less communication overhead.

Figure 5.11 and Figure 5.12 show the speedup of three scalable solutions against the original pipeline running with 1 thread (origi-1 or baseline). The three solutions are: original pipeline with multi-threading enabled running with 40 threads (denoted as origi-40), halvade running in SURFsara cluster and spark running in SURFsara cluster. Figure 5.11 presents the comparison running with the $16 \times 2GB$ dataset while Figure 5.12 presents that of the $25 \times 2GB$ dataset.

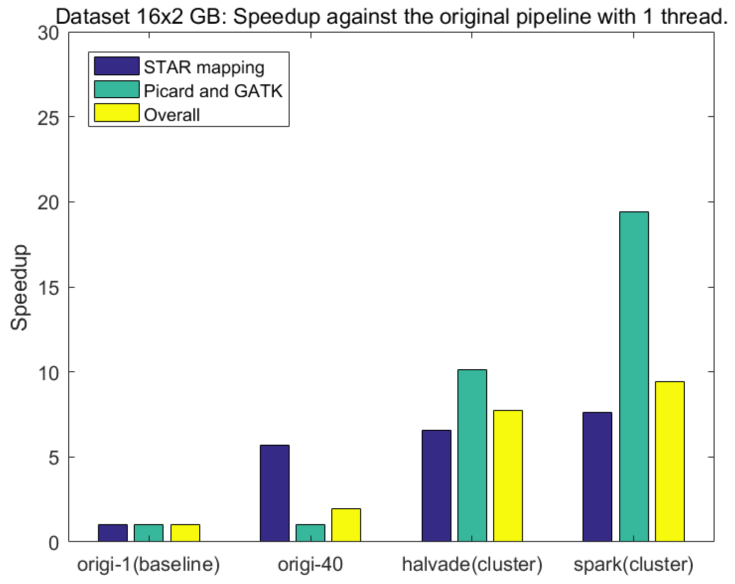

Figure 5.11: Speedup against the original pipeline with 1 thread. Data size: $16GB \times 2$. origi-1 and origi-40 run on node with 20 HT cores and 196 GB RAM. Halvade and spark run in SURFsara cluster where one node is with 8 cores and 56 GB RAM.

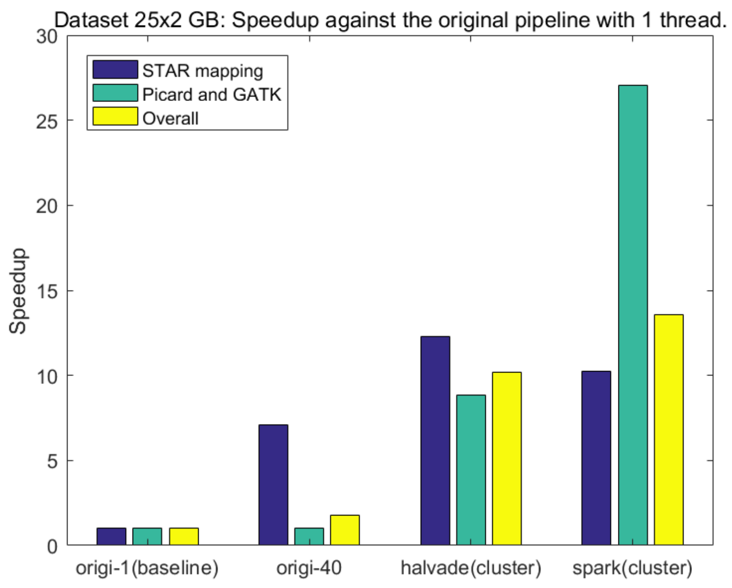

Figure 5.12: Speedup against the original pipeline with 1 thread. Data size: $25GB \times 2$. origi-1 and origi-40 run on a node with 20 HT cores and 196 GB RAM. Halvade and spark run in SURFsara cluster where one node is with 8 cores and 56 GB RAM.

Table 5.5: Comparison of the execution time (min) in cluster mode

dataset	part	origi-1	halvade	spark (cluster mode)
$16 \times 2GB$	STAR alignment	359.63	54.65	47.17
	Dynamic load balancing	n	n	5.62
	Preprocessing & variant calling	265.26	26.15	13.65
	Total	624.89	80.80	66.44
$25 \times 2GB$	STAR alignment	833.51	67.78	81.48
	Dynamic load balancing	n	n	12.62
	Preprocessing & variant calling	891.96	101.17	33.00
	Total	1725.47	168.95	127.10

It has to be noticed that baseline and origi-40 run on a node that is different from nodes in SURFsara. It is with 20 physical cores with hyper-threading enabled (40 threads in total) and 196 GB RAM. Whereas a node in SURFsara is only with 8 physical cores without hyper-threading functions (8 threads in total) and 56 GB RAM. This means that origi-1 and origi-40 run on a more powerful node. Thus, although running on different machines, it is still meaningful to compare halvade and spark with origi-1 and origi-40. Even running in a cluster composed of less powerful nodes, scalability makes the big data solutions much faster than the baseline and the native multi-threading solution.

It can be observed from Figure 5.11 that, for the dataset of size $16 \times 2GB$, spark is faster in all the three comparisons. The speedup of the Picard and GATK part against the baseline reaches 19.43× while the overall speedup reaches 9.41×. These two numbers of halvade are 10.14× and 7.73×, respectively. The biggest difference lies in the Picard and GATK part. Our implementation (spark) outperforms the others in this part due to its dynamic balancing step, which efficiently utilizes the resources. As the Picard and GATK part is completely parallel that does not require any communication or synchronization among parallel tasks, the notable speedup in this part can be concluded as a result of the well-balanced computation of spark itself instead of the result of less overhead due to fewer used nodes. From Figure 5.12, similar observation can be obtained that spark outperforms the other mainly owing to the well-balanced loads in the Picard and GATK part. Spark reaches a speedup of 27.03× in this part and 13.58× in total. These numbers for halvade are 8.82× and 10.21×, respectively. We also notice that for this dataset, spark is slower than halvade in the read alignment part. This is because halvade runs on more nodes than spark does. Compared this with the Picard and GATK part, it further supports our conclusion by showing that even with more nodes, unbalanced load disables halvade to run faster than spark in the Picard and GATK part.

Table 5.5 shows the execution time of each part for baseline, halvade, and spark. It can be seen that for dataset $16 \times 2GB$, spark is 17.77% faster than halvade and for dataset $25 \times 2GB$, spark is 24.77% faster.

5.3. CORRECTNESS EVALUATION

Last but not least, the correctness of the results is also inspected and analyzed. A couple of the output VCF files of our framework are compared to that of the original pipeline. The tool used for comparison is RTG Tools[48]. The following outcomes are measured: true positive (TP), false positive (FP) and false negative (FN). In our case, true positives represent variants discovered by both the original pipeline and our implementation. False positives are those discovered by our implementation but not by the original pipeline, while false negatives are those called by the original pipeline but not by our implementation. Given these values, two other important measures can be obtained:

$$Sensitivity = TP/(TP + FN) \tag{5.2}$$

$$Precision = TP/(TP + FP) \tag{5.3}$$

One assumption we can make about the results is that the output correctness will be influenced since the input data is divided into chunks and processed independently. This way, a tool might lose accuracy because of lacking information from another data chunk which is not known to it when processing the current data. For example, according to the tool documentation of the HaplotypeCaller [33], HaplotypeCaller applies local de-novo assembly of haplotypes in regions where signs of variation show. More explicitly, it ignores the given mapping information and re-assembles those reads in that region. If one such region is accidentally separated in two data segments, the local information is partly lost at either side. Thus, it can be predicted that the more chromosome region segments are divided into, the lower accuracy the result might have.

5.3.1. VARYING NUMBER OF REGIONS

The same set of experiments discussed earlier in Section 5.1 are used to inspect the correctness of the results in this section. That is to say, the compared results are variants called from the paired-end ENCFF005NLJ and ENCFF635CQM datasets. In Table 5.6, the indices correspond to the same combination in Table 5.1. The symbol R_{dLB} still represents the number of regions after the dynamic load balancing, which is the actual parallel tasks that will be in part 3.

It can be seen that the correctness is indeed negatively correlated with the number of chromosome regions.

Table 5.6: Correctness of spark when running on one node.

Index	R_{dLB}	TP	FP	FN	Precision	Sensitivity
1	304	109265	7487	6509	93.59%	94.38%
2	441	109104	8031	6670	93.14%	94.24%
3	262	109323	7234	6451	93.79%	94.43%
4	217	109410	7041	6364	93.95%	94.50%
5	294	109309	7515	6465	93.57%	94.42%
6	183	109529	6798	6245	94.61%	94.38%
7	161	109575	6515	6199	94.39%	94.65%

This is clearer in Figure 5.13, which shows the relation between the true positive calls and R_{dLB}.

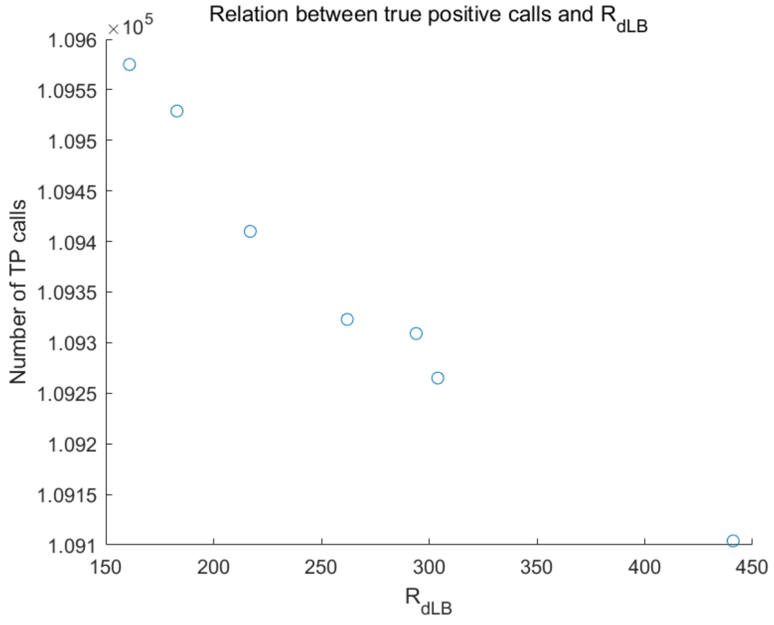

Figure 5.13: True positive calls decrease as R_{dLB} increases.

Halvade, on the other hand, as shown in Section 5.1.3, only forms 37 divisions in total. It achieves better correctness than our implementation at the price of 95.01% more execution time in Picard and GATK part. The correctness of halvade is presented in Table 5.7.

Table 5.7: Correctness of halvade when running on one node.

Index	#Regions	TP	FP	FN	Precision	Sensitivity
halvade	37	110206	5761	5568	95.03%	95.19%

5.3.2. CORRECTNESS IN CLUSTER MODE

First, the same dataset ENCFF005NLJ and ENCFF635CQM as discussed in the last section are used to inspect the change of correctness when migrating from local mode to cluster mode. Table 5.8 shows the comparison of correctness between halvade and spark. It can be observed that compared to combination 6 in Table 5.6, although with the same number of regions, spark in cluster mode yields results of lower precision while higher sensitivity. On the other hand, halvade generates more chromosome regions compared to that when running on one node, and the precision and sensitivity of its results both decrease. Comparing halvade with spark running in the same cluster, the difference of correctness is smaller but halvade is still with higher precision and sensitivity. This can also be explained by the difference in the number of regions as halvade generates fewer regions than spark.

Then the correctness of another dataset, ENCFF174YCI and ENCFF584LJW ($25 \times 2GB$), is also checked in cluster mode. Table 5.9 shows the comparison of correctness between halvade and spark running in the cluster. From the table it can be seen that for this dataset, spark achieves higher sensitivity than halvade, though the precision is still lower.

Table 5.8: Correctness comparison running in cluster mode. Dataset: ENCFF005NLJ and ENCFF635CQM ($16 \times 2GB$).

Implementation	#Regions	TP	FP	FN	Precision	Sensitivity
spark	183	109411	6886	6363	94.08%	94.50%
halvade	142	109669	6850	6105	94.12%	94.73%

Table 5.9: Correctness comparison running in cluster mode. Dataset: ENCFF174YCI and ENCFF584LJW (25 × 2GB).

Implementation	#Regions	TP	FP	FN	Precision	Sensitivity
spark	276	144793	15625	13821	90.26%	91.29%
halvade	142	144424	15465	14190	90.33%	91.05%

5.4. SUMMARY

In this chapter, a number of results have been presented. Now we will make a summary of these results.

- The user-specified parameters $numRegion$ and $regionsFactor$ influence the number of chromosome regions we will get after static and dynamic load balancing. The number of chromosome regions in turn influences the execution time of part 2 and 3. Keeping the number of chromosome regions between 150 and 300 makes the loads balance enough and yields good performance.
- In local mode, our implementation achieves 7.78× overall speedup compared to the original pipeline running with 1 thread (baseline) on the same machine. It is also faster by 24.29% than the alternative solution halvade. As for the Picard preprocessing and GATK variant calling part, our implementation achieves 9.12× speedup compared to the baseline and it is 48.72% faster than halvade.
- In cluster mode, when 16 nodes are used, the program achieves an overall parallel efficiency of 65.13%, and the Picard preprocessing and GATK variant calling part achieves a parallel efficiency of 84.00%.
- In cluster mode, CPU utilization of the our implementation is very high, reflecting well balanced loads in the execution. Except for the sequential execution part, the rest of the program efficiently utilizes the computational resources.
- In cluster mode, our implementation achieves 27.03× speedup in the Picard preprocessing and GATK variant calling part compared to the original pipeline running with 1 thread (baseline) on a node which is more powerful than those in the cluster. In this part, it is 67.38% faster than halvade running in the same cluster. The overall speedup compared to the baseline reaches 13.58× and it is overall 24.77% faster than halvade.
- In local mode, our implementation discovers variants with a precision of 94.61% and a sensitivity of 94.38% with the output of the original pipeline as baseline. This is a little lower than halvade's precision of 95.03% and sensitivity of 95.19%. The main reason is that more chromosome regions are generated in our implementation than halvade. This brings more balance computation but small decrease in the correctness rate.
- In cluster mode, correctness of the results of halvade decrease while that of our implementation remains almost the same. Thus, in cluster mode, difference in correctness between our implementation and halvade is smaller. For the ENCFF174YCI and ENCFF584LJW dataset, we have even higher sensitivity than halvade.

6 CONCLUSIONS

This thesis project targets scaling up the GATK variant calling pipeline for RNA-seq data. We propose and implement a scalable solution based on the big data framework Spark.

We have presented that out implementation can scale in a cluster environment with a parallel efficiency of 84.00% for the preprocessing and variant calling part and an overall parallel efficiency of 65.13%, when the overall speedup reaches 10.42×. We have also shown that our implementation achieves very high resource utilization in parts that are parallelizable, which indicates that our implementation is able to process datasets in a very efficient way because the resources keep working most of the time. We have also compared our implementation with the original pipeline as well as the alternative solutions in terms of performance and correctness. For a dataset of $32GB$ in size, the original pipeline takes 10.41 hours to finish with 1 thread on a node with 20 Hyper-Threading physical cores and 196 GB RAM. This time is reduced to 5.30 hours when the built-in multi-threading is enabled and it runs with 40 threads. On the same node, our implementation, running in local mode, only takes 1.34 hours (7.78× speedup), which is also faster than the 1.77 hours of the alternative solution, halvade, by 24.29%. For a dataset of $50GB$, the original pipeline takes 28.76 hours to finish with 1 thread while it takes 16.34 hours with 40 threads still on that node with 20 Hyper-Threading cores and 196 GB RAM. In a cluster composed of less powerful nodes (each with 56 GB RAM and 8 cores without Hyper-Threading), our implementation takes only 2.12 hours (13.58× speedup) running with 16 nodes while halvade takes 2.82 hours running with all the 76 active nodes in the cluster. Thus, in the cluster environment, our implementation is faster than the alternative solution by 24.77%.

Our implementation outperforms the alternative solutions mainly due to the well-balanced parallel loads in the preprocessing and variant calling part. This can be supported by the fact that for the dataset of $50GB$, our implementation takes only 33.00 minutes for this part on 16 nodes while the alternative solution halvade takes 101.17 minutes for the same part on 76 nodes. This also indicates that for a parallel program, load balancing is so important that without it, 5 times more resources do not improve the performance.

For the same dataset, the correctness of the results of our implementation is negative-related to the number of divided chromosome regions. The more regions we have divided in the load balancing part, the fewer correct discoveries we would get. Running on different machines also influences the correctness. In local mode, the best correctness measures we get is a precision of 94.61% and a sensitivity of 94.38%. On the same node, halvade has slightly higher measures of 95.03% and 95.19% for precision and sensitivity, respectively. In the cluster environment, for our implementation, the precision is reduced to 94.08% and the sensitivity is increased to 94.50%. For halvade, they are reduced to 94.12% and 94.73%. Thus, in the cluster environment, the difference of correctness between our implementation and the alternative solution is smaller. For the other dataset, the sensitivity of our results is even higher than that of halvade.

In total, our implementation is able to parallelize the original pipeline efficiently in a cluster environment. It can process an RNA-seq dataset of $50GB$ in 2.12 hours on 16 nodes, each of which is with 8 CPU cores and 56 GB RAM. This is about 26 hours 38 minutes less than the original pipeline running with 1 thread, and is about 14 hours 13 minutes less than the original pipeline running with 40 threads. It is also 24.77%

faster than the alternative scalable solution while the correctness of results is almost the same. Due to the well-balanced parallel loads in Picard preprocessing GATK variant calling part, a high speedup of 27.03× is achieved compared to the same part of the original pipeline running with 1 thread. The Picard and GATK part in our implementation is also 67.38% faster than the same part in the alternative scalable solution. We can conclude that our implementation scales up the GATK RNA-seq variant calling pipeline efficiently in both single node execution and in a cluster environment using the following techniques. First, applying the big data Apache Spark framework enables transparent distribution of data and computation throughout the cluster, which greatly eases the implementation process and allows efficient scalability when the number of nodes varies. Spark also provides more efficient computation by keeping intermediate data in memory compared to Hadoop MapReduce. Secondly, well-balanced parallel loads make our implementation finish the Picard preprocessing and GATK variant calling part much faster than the original pipeline and the alternative scalable solution. Another conclusion we can make is that our implementation is a better cluster-based scalable solution for scaling up the GATK RNA variant calling pipeline than the alternative one because it is more efficient and of almost the same accuracy.

FUTURE WORK

The bottleneck of the parallel model of our implementation mainly lies in the STAR genome index regeneration, which cannot be parallelized. Although as stated in Section 4.5, we have made optimization for this sequential step, it still takes a considerable proportion of the overall time, especially when the rest parallel parts get shorter. In fact, according to STAR manual [44], regeneration can be avoided by running the mapping pass in a special 2-pass mode, and the splice junctions from the 1st pass will be inserted to the genome on the fly. But in this way, shared memory cannot be used and the genome has to be loaded and unloaded by every task. Thus, one future research direction could be finding a way to allow inserting the junctions on the fly but still keeping the memory access in a low frequency.

Another improvement direction is about the correctness of the results. Although as accurate as the alternative big data solution, our implementation discovers variants with precision and sensitivity ranging from 90% to 95% when compared to the original pipeline. This is much lower than the Spark solution for the GATK DNA analysis pipeline, which achieves the precision and sensitivity of more than 99% [2]. About the cause of inaccuracy, one assumption is that, compared to the DNA-seq data, RNA-seq data needs some special processing at the border before being divided into chromosome subregions. More researches can be done to learn why scalable RNA analysis solutions such as our implementation and Halvade-RNA are not so accurate as the scalable DNA analysis solutions, such as Churchill, SparkGA, and Halvade-DNA.

Last but not least, GATK version 4 starts to provide the users with the Spark version of many tools. The GATK team has also combined some tools to form a partial Spark pipeline of the Best Practices. For example, `ReadsPipelineSpark` takes BAM file as input and performs MarkDuplicates, base quality recalibration, and HaplotypeCaller one after another in Spark. But until the moment this thesis project is finalized (August 2018), the Spark version of the tools is not completed yet (in beta version). Plus, the necessary tool SplitNCigarReads for RNA-seq data processing does not have a Spark version nor is it included in a GATK Spark pipeline yet. In the future, we can either 1. compare the corresponding part of our implementation with the GATK partial Spark pipeline or 2. compare the whole pipeline if Spark version of SplitNCigarReads is released.

BIBLIOGRAPHY

[1] D. Eccles, *A single nucleotide polymorphism is a change of a nucleotide at a single base-pair location on dna,* (2014), `https://en.wikipedia.org/wiki/File:Dna-SNP.svg`, Accessed on 2018-08-07.

[2] H. Mushtaq, F. Liu, C. Costa, G. Liu, P. Hofstee, and Z. Al-Ars, *Sparkga: A spark framework for cost effective, fast and accurate dna analysis at scale,* in *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology,and Health Informatics*, ACM-BCB '17 (ACM, New York, NY, USA, 2017) pp. 148–157.

[3] G. 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. V. Garimella, D. Altshuler, S. Gabriel, and M. A. DePristo, *From fastq data to high-confidence variant calls: The genome analysis toolkit best practices pipeline,* Current Protocols in Bioinformatics **11(1110)** (2013).

[4] 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 **16**, 6 (2015).

[5] D. Decap, J. Reumers, C. Herzeel, P. Costanza, and J. Fostier, *Halvade: scalable sequence analysis with mapreduce,* Bioinformatics (2015).

[6] J. Dean and S. Ghemawat, *Mapreduce: Simplified data processing on large clusters,* Communications of the ACM **51**, 107 (2008).

[7] *Apache hadoop,* `http://hadoop.apache.org/`, Accessed on 2018-08-07.

[8] M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, *Spark: Cluster Computing with Working Sets,* Tech. Rep. UCB/EECS-2010-53 (EECS Department, University of California, Berkeley, 2010).

[9] Y. Han, S. Gao, K. Muegge, W. Zhang, and Z. B., *Advanced applications of rna sequencing and challenges,* Bioinform Biol Insights (2015).

[10] R. Piskol, G. Ramaswami, and J. B. Li, *Reliable identification of genomic variants from rna-seq data,* The American Journal of Human Genetics (2013).

[11] A. Dobin, C. A. Davis, F. Schlesinger, J. Drenkow, C. Zaleski, S. Jha, P. Batut, M. Chaisson, and T. R. Gingeras, *Star: ultrafast universal rna-seq aligner,* Bioinformatics (2013).

[12] C. Trapnell, L. Pachter, and S. Salzberg, *Tophat: discovering splice junctions with rna-seq,* Bioinformatics (2009).

[13] D. Kim, G. Pertea, C. Trapnell, H. Pimentel, R. Kelley, and S. L. Salzberg, *Tophat2: accurate alignment of transcriptomes in the presence of insertions, deletions and gene fusions,* Genome Biology (2013).

[14] T. D. Wu and S. Nacu, *Fast and snp-tolerant detection of complex variants and splicing in short reads,* Bioinformatics (2010).

[15] K. Wang, D. Singh, Z. Zeng, S. J. Coleman, Y. Huang, G. L. Savich, X. He, P. Mieczkowski, S. A. Grimm, C. M. Perou, J. N. MacLeod, D. Y. Chiang, J. F. Prins, and J. Liu, *Mapsplice: Accurate mapping of rna-seq reads for splice junction discovery,* Nucleic Acids Research (2010).

[16] *Gatk | calling variants in rnaseq,* `https://software.broadinstitute.org/gatk/documentation/article.php?id=3891`, Accessed on 2018-07-02.

[17] *Wikipedia: Fastq file format,* `https://en.wikipedia.org/wiki/FASTQ_format`, Accessed on 2018-07-04.

[18] *Intel® hyper-threading technology,* `https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html`, Accessed on 2018-07-10.

[19] *Picard,* `https://broadinstitute.github.io/picard/`, Accessed on 2018-07-03.

[20] *Rna-seqlopedia,* `https://rnaseq.uoregon.edu/`, Accessed on 2018-07-04.

[21] Z. Wang, M. Gerstein, and M. Snyder, *Rna-seq: a revolutionary tool for transcriptomics,* Nat Rev Genet **10**, 57 (2009).

[22] B. B. Cummings, J. L. Marshall, T. Tukiainen, M. Lek, S. Donkervoort, A. R. Foley, V. Bolduc, L. Waddell, S. Sandaradura, G. L. O'Grady, E. Estrella, H. M. Reddy, F. Zhao, B. Weisburd, K. Karczewski, A. O'Donnell-Luria, D. Birnbaum, A. Sarkozy, Y. Hu, H. Gonorazky, K. Claeys, H. Joshi, A. Bournazos, E. Oates, R. Ghaoui, M. Davis, N. G. Laing, A. Topf, , A. Beggs, P. B. Kang, K. N. North, V. Straub, J. Dowling, F. Muntoni, N. F. Clarke, S. T. Cooper, C. G. Bonnemann, and D. G. MacArthur, *Improving genetic diagnosis in mendelian disease with transcriptome sequencing,* Science Translational Medicine **9** (2017), 10.1126/scitranslmed.aal5209.

[23] A. Cánovas, G. Rincon, A. Islas-Trejo, S. Wickramasinghe, and J. F. Medrano, *Snp discovery in the bovine milk transcriptome using rna-seq technology,* Mamm Genome **21**, 592 (2010).

[24] H. Lopez-Maestre, L. Brinza, C. Marchet, J. Kielbassa, S. Bastien, M. Boutigny, D. Monnin, A. E. Filali, C. M. Carareto, C. Vieira, F. Picard, N. Kremer, F. Vavre, M.-F. Sagot, and V. Lacroix, *Snp calling from rna-seq data without a reference genome: identification, quantification, differential analysis and impact on the protein sequence,* Nucleic Acids Research **44**, e148 (2016).

[25] J. M. Mullaney, R. E. Mills, W. S. Pittard, and S. E. Devine, *Small insertions and deletions (indels) in human genomes,* Hum Mol Genet **19**, R131–R136 (2010).

[26] *Snp | learn science at scitable,* (), `https://www.nature.com/scitable/definition/single-nucleotide-polymorphism-snp-295`, Accessed on 2018-08-06.

[27] *Wikipedia: Exon,* `https://en.wikipedia.org/wiki/Exon`, Accessed on 2018-08-06.

[28] *Wikipedia: Snp,* (), `https://en.wikipedia.org/wiki/Single-nucleotide_polymorphism`, Accessed on 2018-08-06.

[29] B. S. Shastry, *Snp alleles in human disease and evolution,* Journal of Human Genetics **47**, 561–566 (2002).

[30] P. G. Engström, T. Steijger, B. Sipos, G. R. Grant, A. Kahles, G. Rätsch, N. Goldman, T. J. Hubbard, J. Harrow, R. Guigó, and P. Bertone, *Systematic evaluation of spliced alignment programs for rna-seq data,* Nat Methods **10**, 1185 (2013).

[31] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin, *The sequence alignment/map format and samtools,* .

[32] *Broade: Gatk/mapping and processing rnaseq,* `https://www.youtube.com/watch?v=0jy82R86Bm4`, Accessed on 2018-07-09.

[33] *Gatk tool documentation index,* `https://software.broadinstitute.org/gatk/documentation/tooldocs/3.8-0/index`, Accessed on 2018-07-09.

[34] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, *The hadoop distributed file system,* in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010) pp. 1–10.

[35] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, *Apache hadoop yarn: yet another resource negotiator,* in *SoCC*, SOCC '13 (2013).

[36] *Rdd programming guide,* https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations, Accessed on 2018-08-06.

[37] *How can i use parallelism to make gatk tools run faster?* (), https://gatkforums.broadinstitute.org/gatk/discussion/1975/how-can-i-use-parallelism-to-make-gatk-tools-run-faster, Accessed on 2018-07-15.

[38] *Countcovariates - gatk-forum - broad institute,* https://gatkforums.broadinstitute.org/gatk/discussion/1248/countcovariates, Accessed on 2018-07-31.

[39] D. Decap, J. Reumers, C. Herzeel, P. Costanza, and J. Fostier, *Halvade-rna: Parallel variant calling from transcriptomic data using mapreduce,* PLOS ONE **12**, e0174575 (2017).

[40] *Github - broadinstitute/gatk: Official code repository for gatk versions 4 and up,* (), https://github.com/broadinstitute/gatk, Accessed on 2018-08-07.

[41] *Wikipedia: Variant call format,* https://en.wikipedia.org/wiki/Variant_Call_Format, Accessed on 2018-07-10.

[42] *The scala programming language,* https://www.scala-lang.org/, Accessed on 2018-07-10.

[43] *shm_overview(7)-linux manual page,* http://man7.org/linux/man-pages/man7/shm_overview.7.html, Accessed on 2018-07-15.

[44] *Star manual,* https://github.com/alexdobin/STAR/blob/master/doc/STARmanual.pdf, Accessed on 2018-07-15.

[45] *An integrated encyclopedia of dna elements in the human genome,* Nature **489**, 57 (2012).

[46] *Surfsara,* https://www.surf.nl/en/about-surf/subsidiaries/surfsara/, Accessed on 2018-07-31.

[47] *iostat(1) - linux man page,* https://linux.die.net/man/1/iostat, Accessed on 2018-08-08.

[48] *Rtg tools-real time genomics,* https://www.realtimegenomics.com/products/rtg-tools, Accessed on 2018-07-15.