UNIVERSITÀ DEGLI STUDI ROMA TRE

Engineering Department
Master Degree Course in Ingegneria Informatica

Thesis

An Efficient Design of a Next Generation Sequencing pipeline with Apache Spark

Majoring
Nicholas Tucci
Freshman 461669

Supervisor
Prof. Riccardo Torlone

Correlator
Prof. Paolo Missier

Accademic Year 2016/2017

In a way this project has the goal to detect and annotate genome variants which may be deleterious, such as cancer, heart attacks or alzheimer.

So this project is dedicated to my grandma, who left us during my Master Degree due to alzheimer illness.

Thanks

Thanks to the Prof. Paolo Missier, Dr. Jacek Cała and Dr. Jannetta Steyn, for having guided me in the elaboration of research work, and for advice, indications and suggestions put to use in the implementation of the pipeline.

Inoltre vorrei ringraziare tutte le persone che mi sono state vicine in questi 5 anni di universita', che mi hanno supportato in piu' occasioni.
In particolare con i miei colleghi abbiamo condiviso momenti leggeri di risate e spensieratezza, ma anche di ansia a causa degli esami. Ma col lavoro di gruppo ed aiutandoci a vicenda nei momenti difficili, ci ha aiutato a superare ogni problema.
Non ci sono parole per esprimere la mia gratitudine per tutti quanti voi, siete stati degli indispensabili compagni di viaggio!

Abstract

The goal of this project is to implement a Next Generation Sequencing Pipeline with Apache Spark. This is a pipeline which processes human genomes in order to detect and annotate Variants which in literature are known as deleterious. These kind of pipeline are much time and resource consuming, so has been introduced Apache Spark, a Big Data technology which can introduce efficiency in the work load. With Spark there is the possibility to deploy this pipeline over a computer-cluster, so through Docker I delivered the necessary technologies to execute it in cluster-mode on Microsoft Azure VMs.

Introduction

This pipeline follows the GATK Best Practices, in particular is a Germline short variant discovery pipeline [Ins07]; it means that processing a cohort of human genomes, with a machine learning approach, allows to predict and enumerate the mutations that are present in a human patient's genome, identifying mutations that are known, from research literature, to be deleterious.

Few years ago, to execute these pipelines were used HPC (High Performance Computing) clusters, data centre designed in order to process these genomes, making very high the "Entry Gate Cost"; but since the recent advent of the Cloud Computing, the Entry Gate Cost decreases, allowing even other organizations to execute these pipelines and introducing improvements in it. [Ste05]

For this reason many solutions were created in order to execute these pipelines with more efficiency; since that each patient's genome is made of tens of Gigabytes, and to process the pipeline are required at least ten patients, it is possible to talk about Big Data in this domain. One recent important Big Data technology is Apache Spark, whose basic idea is to keep in memory the results produced by intermediate steps, avoiding disk access, a very time consuming task. Since that these genome pipelines produce different intermediate files of big size, of course introducing a Spark approach to these pipelines would increase their efficiency.

This thesis is structured in this way:

1. Background: in this chapter will be discussed the previous topics with more de-

tails, explaining the pipeline's functionalities from a Bioinformatics point of view. Moreover will be given an overview of the technologies and software used to delivery the pipeline.

2. Pipeline: description of the pipeline implementation.

3. Clustering: since that has been used Apache Spark to implement this pipeline, this technology gives the opportunity of distribute the computation on a computer-cluster. This chapter describe the delivery of the pipeline on a cluster, using Docker for provisioning the nodes of the cluster.

4. Datasets: description of the input samples used for the pipeline and all the required data in order to execute the GATK tools (used in this pipeline): from the reference genome to the known sites and annotation databases.

5. Performance Analysis: numbers that help to understand which is the best parameters configuration in order to obtain maximum performance and compare with other solutions.

Contents

List of Figures

List of Tables

List of Charts

Chapter 1

Background

The human genome is the complete set of nucleic acid sequences for humans, encoded as DNA (deoxyribonucleic acid) within the 23 chromosome pairs, adding up to 6.4 billion base pairs [Wika]. The DNA is divided in sequences called *gene*, the basic unit of heredity. Every person has two copies of each gene, one inherited from each parent. Most genes are the same in all people, but a small number of genes (less than 1 percent of the total) are slightly different between people. These small differences contribute to each person's unique physical features. In humans, genes vary in size from a few hundred DNA bases to more than 2 million bases [Ref20b]. A *base* (nucleotide) is the fundamental unit which codifies the DNA (adenine (A), cytosine (C), guanine (G) and thymine (T)) and RNA (like DNA but uracil (U) instead of thymine).

In this thesis is discussed the implementation of a NGS (Next Generation Sequencing) pipeline, which aims to detect **variants** (mutations) of a human genome from a reference genome. These variants are then loaded in databases and statistical tools, in order to identify harmful mutations. Variants called inside this pipeline include SNPs (Single Nucleotide Polymorphism) and INDELs (insertions/deletions).

SNP is the most common type of genetic variation among people and represents a difference of a DNA nucleotide (for example, a SNP may replace the nucleotide cytosine (C) with the nucleotide thymine (T) in a certain point of DNA). They can act as biological markers, to locate genes that are associated with disease. SNPs can also be used to track the inheritance of disease genes within families. Future studies will work

to identify SNPs associated with complex diseases such as heart disease, diabetes, and cancer [Ref20a].

An **INDEL** polymorphism is a variation in which a specific nucleotide sequence is present (insertion) or absent (deletion). While not as common as SNPs, INDELs are widely spread across the genome [RMS20].

Figure 1.1: Pipeline Overview

As shown in Figure 1.1, this human genome processing starts with a biological sample (e.g. blood, saliva...) as input of sequencing machines and finishes with a report of differences from the reference genome and what they mean. This is usually divided into three stages:

1. Primary analysis: analyses a sample biochemically and produces raw data
2. Secondary analysis: takes the raw data, aligns it to the reference, and identifies variants (differences) from the reference
3. Tertiary analysis: analyses the variants and adds annotations to help interpret their biological or clinical implications

The primary analysis stage is done in the laboratory using specialized **sequencing** instruments. The genome sequencer replicates and fragments the DNA and then reads the base pairs, in a massively parallel process combining biochemistry, optics, electronics, and image processing. Identifying the base pairs in a sequence of DNA (base calling) is hard and errors do happen, so in addition to the A, C, T, or G base called for each location, the sequencer also produces a *quality score* to record how confident the sequencer is in the base call. Sequencing an entire human genome typically produces about a billion roughly 100-250 character strings (**reads**) of A, C, T, and G (bases), covering

the genome with an average of 30 copies for redundancy, over-sampling the DNA.
Ideally, DNA sequencing includes the entire genome, Whole-Genome Sequencing (WGS). While WGS technology is rapidly coming on the market at affordable prices, in the last few years most research labs have adopted **Whole-Exome Sequencing** (WES) as interim technology. WES is limited to the exome information, that is to the regions of DNA that are responsible for protein expression by way of RNA translation. These are the priority areas of the genome, where mutations can be more easily identified as deleterious, as they have a higher chance of directly compromising protein synthesis. WES-based diagnosis provides a good trade-off between diagnostic power and the cost of data processing, as exomes only account for about 1.5% of the entire genome and can, therefore, be processed using in-house computational resources [P.M12].
This process produces about 15 GB of compressed WES raw data per sample (instead of 1 TB for WGS) ready for secondary analysis, typically stored in FASTQ files. In particular we talk about *paired end reads*, which means that a DNA fragment is sequenced in both ends, generating high quality sequence data collected in 2 FASTQ files [Ill, Gen].

This project focuses on second and third stage, the process that goes from *reads* to *variants* discovery and annotation illustrated in 1.1 and deepened in next paragraphs.

1.1 GATK Best Practices

The FASTQ files produced from the sequencing machine are used as input of the GATK tools: here begins the Big Data processing. Indeed these tools take over a day to process a 30x whole genome sample on a 16-core server, starting with the FASTQ files from the sequencer, and producing a file of aligned reads and a file of variant calls.
In a discovery approach, I used the GATK tutorials in order to follow their Best Practices, which can be divided in 3 phases (as even the Best Practices Figure 1.2 are divided): Preprocessing, Variant Discovery and Call Set Refinement; at the time was available GATK version 3.8 and it was possible to execute their tools step by step. The basic concept of this pipeline is that each tool processes the file generated by the previous tool, in a sequential manner.

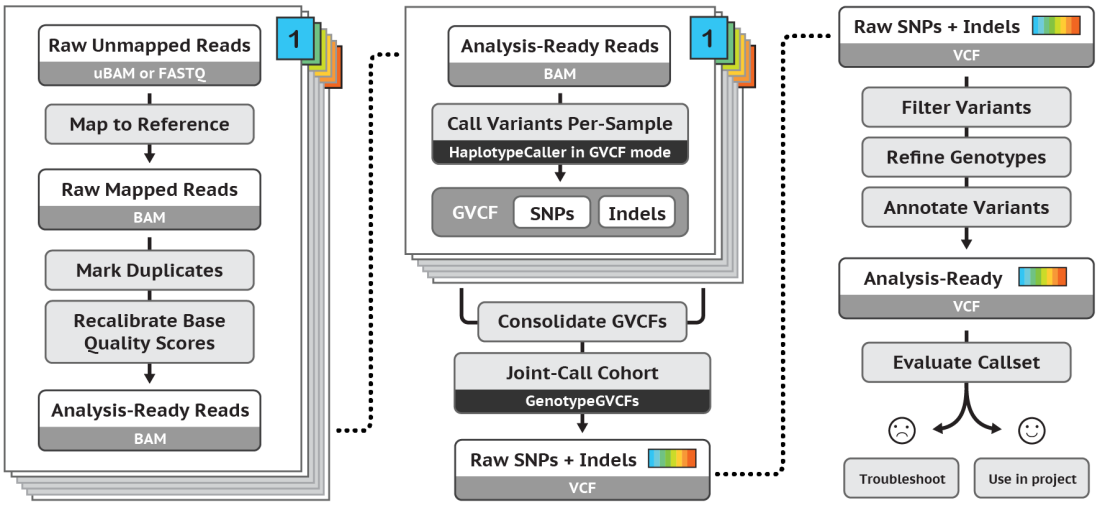


Figure 1.2: GATK Best Practices

1.1.1 Preprocessing

The Preprocessing phase is the most resource and time consuming phase. As is possible to observe in the Best Practices Figure 1.2, this phase goes from the *Raw Unmapped Reads* to the *Recalibrate Base Quality Score* (but I included here even HaplotypeCaller in GVCF mode, because of designing decision), and the tools used here must process each of the human genomes in input (the FASTQ files) in a sequential manner.

Since human DNA varies relatively little among individuals (typically about 1 in 1000 locations) we usually compare a given person's DNA to a reference genome, a composite genome based on the results of the first Human Genome Project [Gen]. So this is the first step, aligning the sequenced reads collected in FASTQ files against a human reference genome, mapping each read to one or more positions, in order to reconstruct the sequenced genome. For this pipeline was used the reference genome HG19, but the only reference file is not enough: are necessary even indexes of the reference genome, for which GATK provides tools that allow to create them. Indexes are necessary to other tools in order to improve access performance to reference genome. A *de facto standard* software for the alignment is the **Burrows Wheeler Aligner**, in particular using the algorithm **BWA-MEM** [HR]. The output generated by this software is stored in a

SAM (Sequence Alignment Map) file.
After that, using tools provided by Picard, the reads inside the SAM file will be **sorted**, for then **Marking Duplicates**: a process that flags the paired reads mapped to exactly the same start and end positions as duplicates. These reads often originate erroneously from DNA preparation methods. They will cause biases that skew variant calling and hence should be removed, in order to avoid them in downstream analysis; this process generates a BAM (Binary Alignment Map, a compressed binary representation of SAM) file [Ins].
The following step is the **BQSR** (Base Quality Score Recalibration): since that the sequencing machine emits error (for example due to physics or chemistry of how the sequencing reaction works), for each base call is even provided a quality score, which represents how confident the machine was that it called the correct base. Here a machine learning approach adjusts these scores [Ins29]. Known sites (which will be discussed in Datasets chapter) are required in order to execute this step.
Even if **Haplotype Caller** is considered part of Variant Discovery phase, I treated here because of designing reasons. Indeed it aims to call variants on a single genome, to discovery germline variants in DNA. But as previous tools, this must process each of the DNA sample (in particular the output file generated by the previous tool). The output file is a raw (unfiltered) VCF (Variant Calling Format). This tool calls SNPs and INDELs simultaneously.

1.1.2 Variant Discovery

After that all the previous steps processed each human genome sample in input (in other words, the Haplotype Caller generated a raw VCF file for each human genome), it is possible to execute the **GenotypeGVCFs**. As possible to notice in the Best Practices Figure 1.2, this tool takes in input the VCF files generated from Haplotype Caller (in particular, since the GATK version 3, the file extension of Haplotype Caller is .g.vcf) to create the raw SNP and INDEL VCF that are usually emitted by the callers [Ins21]. From this point of the pipeline the execution goes on with a single file: the file produced from a tool will be the input for the following tool.
Following this approach, is now necessary to apply **VQSR** (Variant Quality Score Re-

calibration), a method that uses machine learning algorithms to learn from each dataset what is the annotation profile of good variants vs. bad variants, and does so in a way that integrates information from multiple dimensions (5 to 8 typically). This allows us to pick out clusters of variants in a way that frees us from the traditional binary choice of "is this variant above or below the threshold for this annotation?" [Ins30, Ins09]. This is possible through the **Variant Recalibrator** and **Apply Recalibration** tools, in order to recalibrate variant quality scores and produce a callset filtered for the desired levels of sensitivity and specificity. In particular the process is so structured:

1. Prepare recalibration parameters for SNPs
    a) Specify which call sets the program should use as resources to build the recalibration model
    b) Specify which annotations the program should use to evaluate the likelihood of INDELs being real
    c) Specify the desired truth sensitivity threshold values that the program should use to generate tranches
    d) Determine additional model parameters
2. Build the SNP recalibration model
3. Apply the desired level of recalibration to the SNPs in the call set
4. Prepare recalibration parameters for INDELs
    a) Specify which call sets the program should use as resources to build the recalibration model
    b) Specify which annotations the program should use to evaluate the likelihood of INDELs being real
    c) Specify the desired truth sensitivity threshold values that the program should use to generate tranches
    d) Determine additional model parameters
5. Build the INDEL recalibration model

6. Apply the desired level of recalibration to the INDELs in the call set

At the end of this process, a file containing the recalibrated variants is produced [Ins06b].

1.1.3 Call Set Refinement

This phase of the pipeline, according to GATK Best Practices 1.2, starts with the **Genotype Refinement**; it takes in input the VCF file generated by the previous VQSR and it is composed of three steps:

1. Derive posterior probabilities of genotypes: we are deriving the posteriors of genotype calls in our callset, the VCF file generated by VQSR.

2. Filter low quality genotypes: after the posterior probabilities are calculated for each sample at each variant site, genotypes with GQ < 20 based on the posteriors are filtered out. GQ20 is widely accepted as a good threshold for genotype accuracy, indicating that there is a 99% chance that the genotype in question is correct. Tagging those low quality genotypes indicates to researchers that these genotypes may not be suitable for downstream analysis (a filter tag is applied, but the data is not removed from the VCF).

3. Annotate possible de novo mutations: using the posterior genotype probabilities, possible de novo mutations are tagged [Ins06a].

Finally there is the **Variant Annotation** process, which is not supported by GATK. Indeed their tutorial suggests to use third-party software. There is a wide variety of tools and databases in use for this process. Depending on the purpose of the analysis, these might add information about evolutionary conservation, protein structure, drug response, disease risk, genomic interactions, etc. The choice of databases and tools depends greatly on the purpose of the analysis, and the overall methodology of the clinician or researcher.
Due to union of VCF files in only one through GenotypeVCFs, the current VCF file at this point of the pipeline is a sort of a CSV file with many fields, among which the input samples names. But now is necessary to split this VCF in many file as the number of input samples, since that following annotation steps require a VCF relative

to each sample. In order to deliver this task, GATK provides the tool *SelectVariants*, which allows us to do it. From this point of the pipeline to the end, we have the same approach used during the Preprocessing phase, where each tool processed each sample; here each tool annotates each sample.

As a first annotation tool here we used **ANNOVAR**, an efficient software tool (a command line Perl program) that allows to functionally annotate genetic variants detected from diverse genomes. This is possible through well known Databases in literature, that ANNOVAR allows to download. It is used extensively by the scientific community. This efficient program enables annotation of a whole genome in less than four minutes and filtering of important variants in less than 15 minutes [Hak10]. Finally the pipeline concludes with the annotation though **IGM Anno** and a filtering called **Exonic Filter**. The finally VCF file generated in this last step were comparable with the VCF realized in another pipeline.

1.2 Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware (scale "out" rather than "up"), designed to be highly fault-tolerant and deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets.

HDFS has a master/slave architecture. An HDFS cluster consists of a single **NameNode**, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of **DataNodes**, usually one per node in the cluster, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients [Bor08].

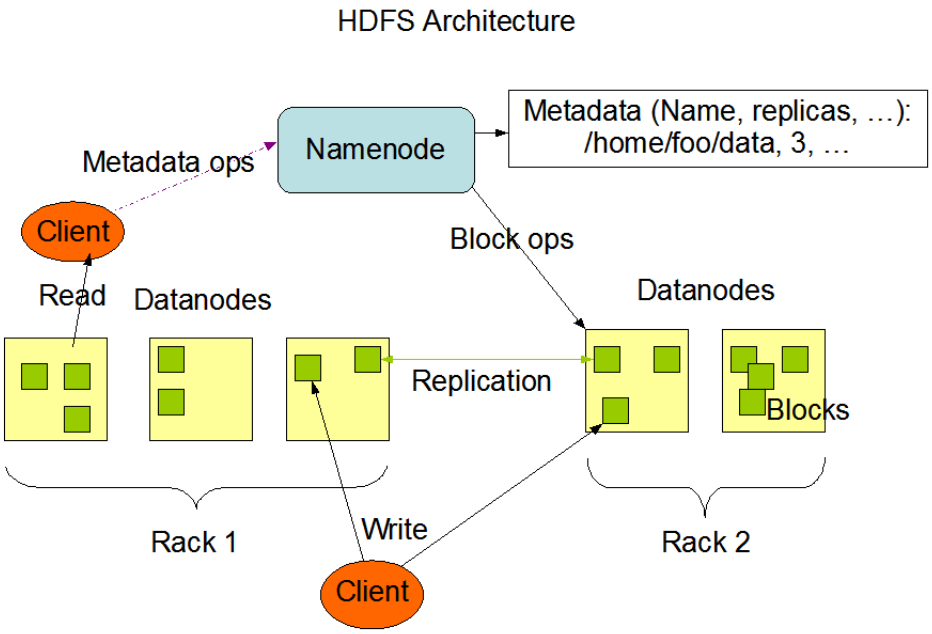


Figure 1.3: HDFS Architecture

In this project HDFS has been necessary in order to distribute mandatory files for the pipeline over the cluster. Some GATK tools in cluster mode expect to find in input HDFS files path.

1.3 Apache Spark

Since that NGS pipelines process big quantity of data, Apache Spark is a fast and general-purpose cluster computing system. It would introduce efficiency in these pipelines not only because allows to distribute the work load on a cluster of more computers, but even because runs programs up to 100x faster than *Hadoop MapReduce*. This is due to Spark in-memory data storage, for very fast iterative queries, using general execution graphs and powerful optimizations. Provides high-level APIs in Java, Scala, Python and R, and can run on top of Hadoop Cluster.

1.3.1 Cluster Architecture

Spark applications run as independent sets of processes on a cluster, coordinated by the **SparkContext** object in the main program (**Driver Program** which is situated in *Driver node*), according to the developer requirements. There is another machine where the Spark cluster manager is running, called the *Master node*. Along side, in each of the machines in the cluster, there is a *Worker* process running which reports the available resources in its node to the *Master*. Once connected, Spark acquires **Executors** on nodes in the cluster (*Worker nodes*), which are processes that run computations and store data for the application. After that it sends the application code (defined by high-level APIs) to the executors. Finally, SparkContext sends tasks to the executors to run.

In other words, when the *Driver* process needs resources to run jobs/tasks, it ask the *Master* for some resources. The *Master* allocates the resources and uses the *Workers* running through out the cluster to create *Executors* for the *Driver*. Then the Driver can run tasks in those *Executors*.

Each application gets its own executor processes, which stay up for the duration of the

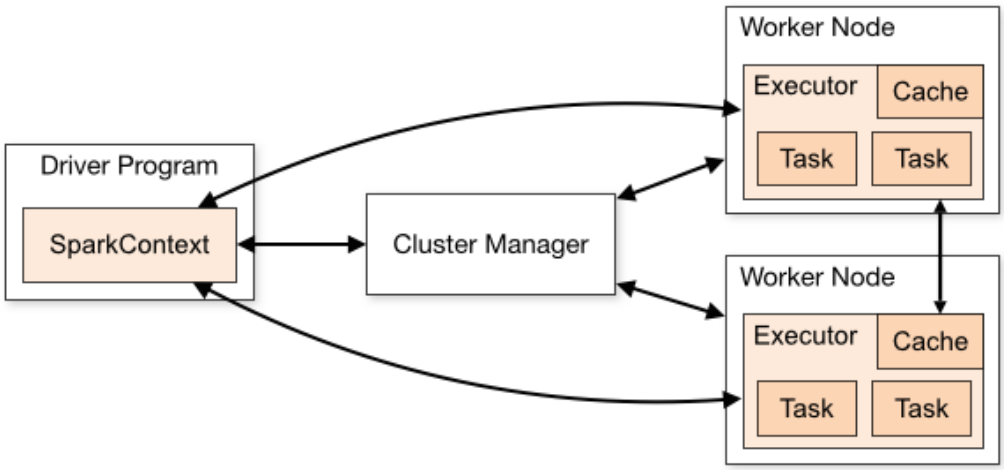


Figure 1.4: Apache Spark Cluster overview

whole application and run tasks in multiple threads. This has the benefit of isolating applications from each other, on both the scheduling side (each driver schedules its own

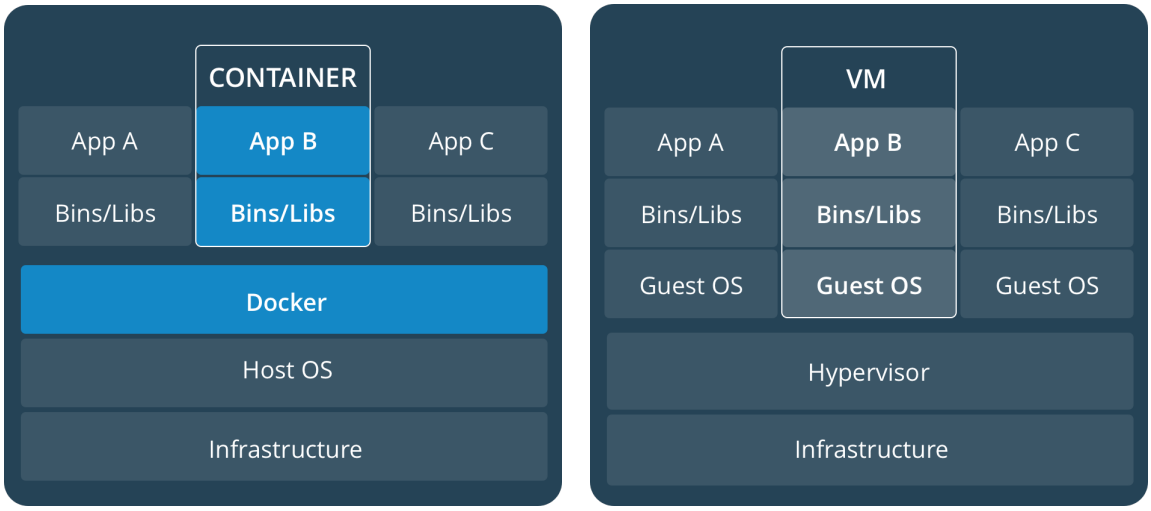
tasks) and executor side (tasks from different applications run in different JVMs).
Spark revolves around the concept of **Resilient Distributed Dataset** (RDD), distributed collections of objects that can be cached in memory across cluster nodes. It is fault-tolerant (automatically rebuilt on failure) and can be operated on in parallel. RDD are the distributed memory abstractions that lets programmer perform in-memory parallel computations on large clusters in a highly fault tolerant manner.
When the user application creates RDDs and run actions, the result is a DAG (Directed Acyclic Graph) of operators, which is compiled into stages. Each stage is executed as a series of Task (one Task for each Partition) [Apa].

Even GATK in the 4th version introduced Spark because of all these benefits. So in this moment some of their tools support Spark; for other tools which are not in Spark, I implemented them in a manner discussed in next chapters, in order to introduce Spark in them too.

1.4 Docker

Docker is a platform to develop, deploy, and run applications with containers. The use of Linux containers to deploy applications is called containerization. Containerization is increasingly popular because containers are: *lightweight* (containers leverage and share the host kernel), *interchangeable* (deploy updates and upgrades on-the-fly), *portable* (build locally, deploy to the cloud, and run anywhere) and *scalable* (possibility to increase and automatically distribute container replicas).
A container is launched by running an **image**, an executable package that includes everything needed to run an application (the code, a runtime, libraries, environment variables, and configuration files). So a container is a runtime instance of an image (what the image becomes in memory when executed).
The Figure 1.5 shows the difference between Docker Container and a generic Virtual Machine: containers are used to execute one or more specific applications, defining a virtual environment required by an application set. Host does not need an hypervisor. Containers do not have the complete OS (the only constraint is a compatible host OS), contain only the strictly necessary to execute an application and decoupling it from the

underlying system. Containerized software will always run the same, regardless of the environment.



(a - Docker Container) (b - Virtual Machine)

Figure 1.5: Comparison of Docker Container and Virtual Machine.
(a) A **container** runs natively on Linux and shares the kernel of the host machine with other containers. It runs a process taking no more memory than any other executable, making it lightweight.
(b) A **virtual machine (VM)** runs a full-blown guest operating system with virtual access to host resources through a hypervisor. In general, VMs provide an environment with more resources than most applications need.

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs and start almost instantly.
Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, one or more apps, necessary binaries and libraries. VMs can also be slow to boot [Doca].

1.4.1 Swarm

A swarm is a group of machines that are running Docker and joined into a cluster. The usual Docker commands are executed on a cluster by a **Swarm Manager**. The machines, after joining a swarm, are referred to as *nodes*.

Using **Compose** file, is possible to instruct the swarm manager to use several strategies to run containers, such as "emptiest node", which fills the least utilized machines with containers. Or "global", which ensures that each machine gets exactly one instance of the specified container.

Swarm managers are the only machines in a swarm that can execute user commands, or authorize other machines to join the swarm as workers. **Workers** only provide capacity and do not have the authority to tell any other machine what it can and cannot do.

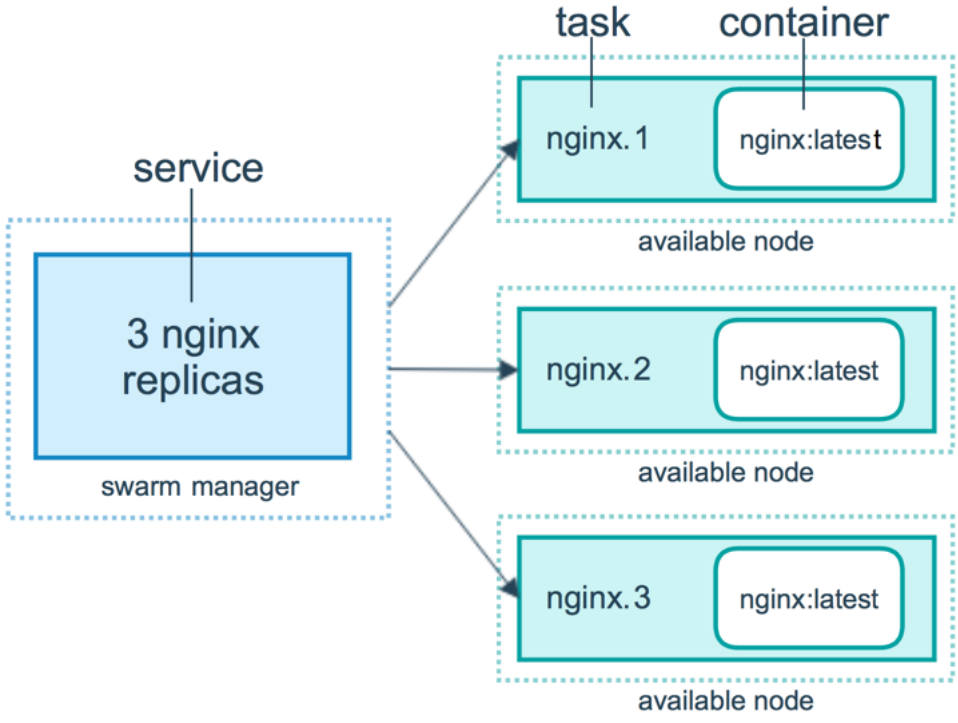


Figure 1.6: Service deploy over a Swarm

When a user deploys an application image in swarm mode, he is creating a **service**.

Frequently a service is the image for a *microservice* within the context of some larger application.
In the service creation context, must be specified which container image to use and which commands to execute inside running containers, with the opportunity of defining:

- the port where the swarm makes the service available outside the swarm
- an overlay network for the service to connect to other services in the swarm
- the number of replicas of the image to run in the swarm

When a service is deployed to the swarm, swarm manager accepts the service definition as the desired state for the service. Then it schedules the service on nodes in the swarm as one or more replica tasks. Tasks run independently of each other on nodes in the swarm [Docb].

In this project Docker is necessary in order to deploy Apache Spark and HDFS on a cluster of computers and so distribute the execution (horizontal scalability). In next chapters will be discussed how this service is deployed.

1.5 Microsoft Azure

Microsoft Azure is a Cloud Computing service created by Microsoft for building, testing, deploying, and managing applications and services through a global network of Microsoft-managed data centres. It provides Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) and supports many different programming languages, tools and frameworks.
Cloud Computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. Indeed with a power plant analogy, we can say it used to be that everyone had their own power source; then people started to build large, centralized power plants with very large capacity, connected to customers by a network. Their usage is metered, and everyone pays only for what they actually

use.

So using this approach, as shown in Figure 1.7, we can observe how the unused resources quantity, given as

$$[ComputationalCapacity - ComputationalDemand]$$

decreases using the cloud, instead that provisioning a proper data centre.

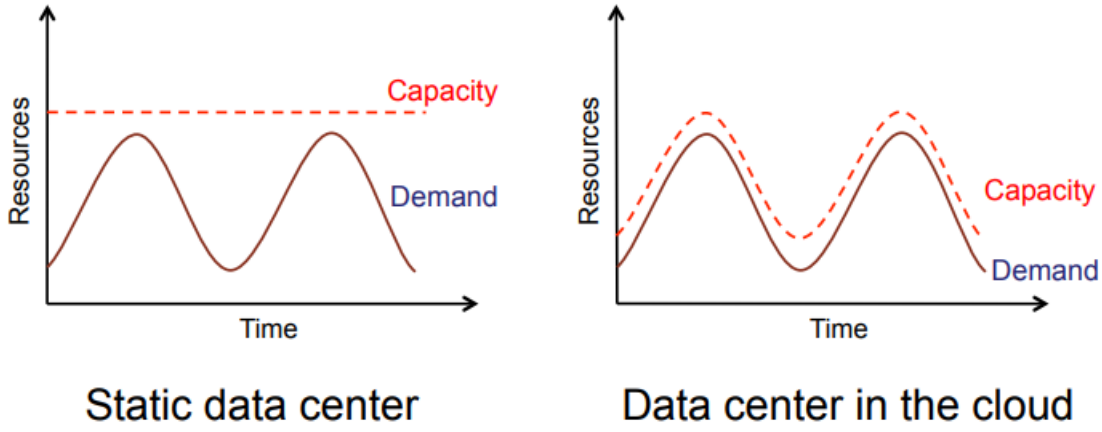


Figure 1.7: Static data centre and Data centre in cloud

Chapter 2

Pipeline

As exposed in the Background chapter and according to the GATK Best Practices, the pipeline can be divided in three phases: Preprocessing, Variant Discovery and Call Set Refinement. Here is exposed a solution which deploys the pipeline on a single node. Details to delivery the pipeline on a cluster will be discussed in next chapters.

2.1 Preprocessing

It is interesting to note that this phase is the most resource/time consuming phase and only for this phase, in this moment, GATK 4.0 provides Spark tools; for the remaining two phases I produced a solution in Spark with a particular approach.
In particular for this phase they provide three interesting tools: *BwaAndMarkDuplicatesPipelineSpark*, *BQSRPipelineSpark* and *HaplotypeCallerSpark*. But, as mentioned more times by GATK, all Spark tools and pipelines are in BETA in this moment. Using these tools in a sequence manner, all the steps of the GATK Best Practices of Preprocessing phase (considering the Preprocessing expressed in Background chapter) are executed using Spark. An interesting consequence of using only these three tools is the production of only three intermediate files; indeed the previous version (GATK 3.8) produced 7 intermediate files. In this way much of the disk access time is reduced, avoiding in part a process that is known to be very time consuming.

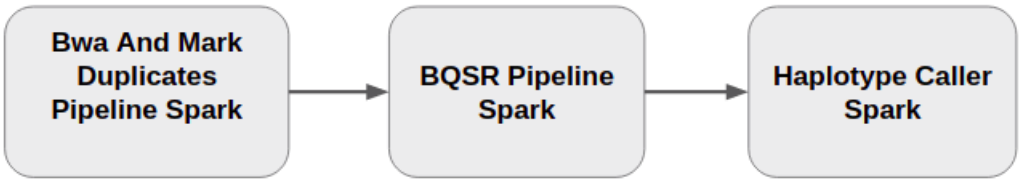


Figure 2.1: Preprocessing steps in GATK 4.0

2.1.1 Requirements Satisfaction

The first step, BwaAndMarkDuplicatesPipelineSpark, differently by previous version tools, requires only one input file. Indeed while previously two paired end FASTQ files were accepted as input, now GATK prefers using an *unmapped BAM* (uBAM) file format. So before of executing these commands, a further preprocessing is required: through the Picard tool FastqToSam is possible to convert two FASTQ files in a single uBAM file.
Moreover, the reference genome indexes used in previous GATK versions are not enough: indeed it is required an other file in the folder containing the reference genome, a .fasta.img file. It is possible to create it through BwaMemIndexImageCreator.
The following tools, BQSRPipelineSpark and HaplotypeCallerSpark, in addition to parameters described in Background chapter, require an other file in the reference folder, a .2bit file. For the reference genome HG19 used in this project, it is possible to download it from Internet.

2.1.2 Pipeline Implementation

After satisfying the requirements, there are two approaches to implement the Preprocessing phase: GATK 4.0 and Integrated Pipeline:

1. **GATK 4.0**: a bash script that calls the Pipelines commands sequentially, executing the first command on every input sample (a for loop), for then continuing this approach with following tools. So with the tools illustrated in Figure 2.1 it is possible to execute this part of the pipeline for many input samples; after that, the outputs generated for each sample will be used as input for GenotypeGVCFs, for then executing the Variant Calling.

With this approach, this part of the pipeline will produce 2 intermediate files that will cause disk access.

2. **Integrated Pipeline**: ; in this approach was attempted to avoid intermediate disk access. Deepening the Java code of tools BwaAndMarkDuplicatesPipelineSpark, BQSRPipelineSpark and HaplotypeCallerSpark, and exploring their class code, is possible to notice that each of these class is made of different JavaRDD which pass their output to the following JavaRDD, with a disk writing of the final result. So instead of writing on the file system the result of BwaAndMarkDuplicatesPipelineSpark for instance, it may be used as input of the first JavaRDD of BQSRPipelineSpark and so on, pipelining JavaRDDs. In this way it would be possible to avoid some of the intermediate writings. I asked about a possible design like this in GATK Forum. They agree that this solution would avoid part of the heavy process of disk access, but they are still benchmarking whether is convenient or not having the entire solution in memory.
I tried to produce a solution that follows this approach; observing at the tool output, is possible to notice that the BWA runs normally. After some time started the BQSR normal output, alternating to the BWA output (the two tools start working in parallel), for then raising an exception:

Listing 2.1: Sequence Dictionary

```
java.lang.IllegalArgumentException:
Reference index for 'chr4' not found in sequence dictionary.
```

suggesting a conflict of a reference index inside the sequence dictionary. After sharing the problem of this approach in the GATK Forum, an issue in their Git Hub project was open. But I think that to carry out this approach will take much more time. I have not enough knowledge in this domain in order to achieve it. So I continued to use the only tools provided by GATK, without apporting any alteration.

2.2 Variant Discovery

The approach used here is already exposed in Background chapter. Since that GATK does not provide Spark tools, for this phase has been used GATK v3.8-0, because this version provide complete tools (it will be clearer in a few lines). After that all samples reached the HaplotypeCallerSpark step, are ready for the Variant Discovery phase and so apply the GenotypeGVCFs: this tool takes in input all files generated by HaplotypeCallerSpark executions and creates an output to be used for following steps; in particular this tool requires of an index, but the GATK v3.8 creates the index himself (this reason leads to choice GATK v3.8).
After that is necessary to calculate and then apply Variant Recalibration for SNP before and INDEL later. All theses tasks are sequential and require few time compared to Preprocessing phase.

2.3 Call Set Refinement

This phase was already discussed in the Background chapter, here will be expressed more details. This phase starts with the output generated by the Variant Discovery phase. As GATK Best Practices suggests, it is necessary to execute sequentially these tools: CalculateGenotypePosteriors, VariantFiltration and VariantAnnotator.
After that GATK Best Practices provide only some guidelines, because GATK does not cover following steps. So the last generated file (from VariantAnnotator) will be used by SelectVariants to create as many files as the input samples in Preprocessing phase. Each of these new files will be then processed by ANNOVAR and IGM Anno; and finally an Exonic Filter will be applied.
In particular to apply annotation of ANNOVAR and IGM Anno have been used the same proceedings of a previous project. Before of applying ANNOVAR annotation, this software requires some human Databases relative to the reference genome used in the pipeline. ANNOVAR provides tools in order to download them and after that is possible to annotate variants.

2.4 *Sparkifying* not-Spark tools

At this point is possible to see the entire pipeline at Figure 2.2 (GATK tools have been marked in *italics*); it is possible to notice that only part of the Preprocessing phase uses Spark tools. But it was required that the entire pipeline is implemented in Spark, due to continuity of the system.

To achieve this task has been used the **Pipe** Transformation method provided by Spark. From the documentation, this method "pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings". In other words, it allows to execute inside Spark a software which is inside a Bash script. Since that to call each GATK tool must be used a bash command, with this method is possible to *"import"* all the other tools in Spark.

For tools like *FastqToSam*, this approach may introduce parallelism; indeed as observable in Figure 2.2, this tool must process each input sample in order to convert it in a uBAM file. As there are multiple samples in input, Spark will parallelise the conversion. There is an inconvenient: as this method receive in input a bash script, this must be saved on the File System. It means that it works only when executed in local on a single node. When executed in cluster mode, a *FileNotFoundException* will be raised. So the parallelism level of this approach is limited to the only number of cores of the local machine, does not scale to the number of nodes in the cluster. As further development, it may be found a way to scale this tool in a cluster.

Referring to Figure 2.2, the remaining tools after Preprocessing phase (for how has been defined in this project) are not implemented in Spark. So has been adopted the same approach used for *FastqToSam*: calling each tool through bash command inside the Pipe command. As possible to notice in Figure 2.2, tools from *GenotypeVCFs* to *SelectVariants* are execute only "one time" in the entire pipeline and not for each input sample as previous tools. It means that using Pipe will not lead any effective speed-up to the process. But as already exposed, a continuity solution was required.

After splitting the VCF file in many files as input samples through *SelectVariants*, following tools must process many files. Here the pipeline gains a speed-up similar to the one gained with *FastqToSam*. Since that ANNOVAR and IGM Anno tools are programmed in Perl language, it is possible to use the Perl command inside Pipe function and import this tools in Spark.
The Exonic Filter is a simple filter which filters the input TSV file rows, according to specific rules. To create this function, Spark offers the method Filter, which is enough to achieve this task.

All these *Sparkified* tools are written in Java using three classes:

1. **FastqToSam**: encapsulates the *FastqToSam* tool, requiring in input the Picard jar path, comma separated input files (the paired end fastq files to be converted in uBAM) and an output folder path.

2. **VariantDiscovery**: here have been encapsulated the Variant Discovery tools, so referring to Figure 2.2, are included tools from *GenotypeVCFs* to *INDEL Recalibration*, which means that this part is without speed-up.

3. **CallsetRefinement**: this class encapsulates tools from *Genotype Refinement* to the *Exonic Filter*. As exposed before, the part after *SelectVariants* gains speed-up.

All three classes extend an Abstract Class which provides them support methods such as creating an executable bash script on the file system, executing bash commands, locating required input files on the file system.
The choice to include all the Variant Discovery and Call Set Refinement tools in two classes is due to give more flexibility to the user of these tools; indeed to execute respectively Variant Discovery and Call Set Refinement tools will be necessary to use only two bash command. These classes are callable through the *spark-submit* script provided inside the Spark package.

At this point the entire pipeline meant to run on a local machine can be expressed as a bash script. This will contain a spark-submit calling FastqToSam class, *for loops*

respectively for BwaAndMarkDuplicatesPipelineSpark, BQSRPipelineSpark and HaplotypeCallerSpark, processing each sample; two spark-submit calling respectively VariantDiscovery and CallsetRefinement.
The generated output files have been compared with a previous similar pipeline [P.M12] results and they are much similar; it should be necessary only to apply an higher filter in VariantFiltration.

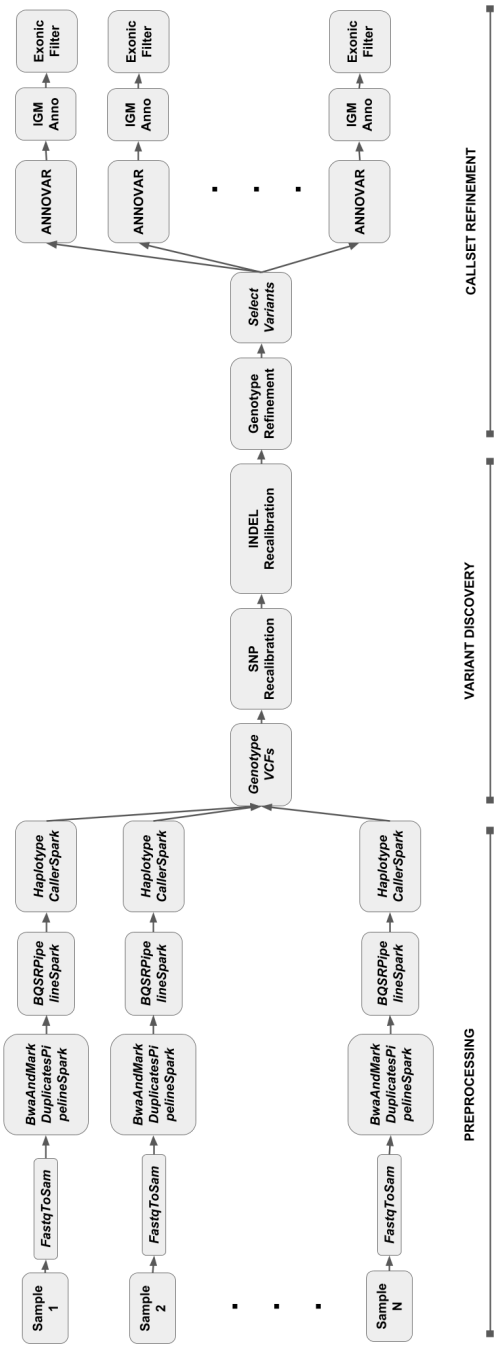


Figure 2.2: NGS Pipeline

Chapter 3

Clustering

Until now has been discussed pipeline execution on a single node. Since now will be exposed how to delivery the execution of the pipeline in cluster mode.
In Background chapter explains how Spark works and the possibility that offers Spark to distribute computation in cluster. Microsoft Azure offers a solution, *HDInsight*, which provides a pre-configured environment ready to execute Spark in cluster mode. The inconvenient is the pricing, which is the double of a normal VM without the HDInsight provisioning. Instead of invest moneys in HDInsight, is better spending them in further hardware resources.
Rather than spending resources in HDInsight, the choice is to provision VMs "by hand". To delivery this provisioning has been chosen Docker Swarm.

3.1 VMs Provisioning with Docker Swarm

GATK 4.0 Spark tools give the opportunity to run them in cluster mode. But they expect to read input files or reference genome in HDFS. So Docker should be able to provide both services (Spark over HDFS). The solution Docker Swarm, as exposed in Background chapter, offers the opportunity to deploy a service over a cluster in order to have Docker Containers distributed in different nodes, able to communicate with each other. Through **Docker Stack** has been possible distributing the required services on cluster. It is based on a *docker-compose* file, just like services in local machine; with some differences, Docker Stack allows to distribute services over a Swarm.

Big Data Europe offers a solution that allows to achieve this task; indeed they provide in Docker Hub the Docker Images necessary to deploy containers over a Swarm.
According to Spark and HDFS architectures exposed in Background chapter, the required actors for this cluster are **Namenode**, **Datanode**, **Spark Mater** and **Spark Worker**, illustrated in Figure 3.1. First of all it is necessary to create a Swarm: the first node will be **Swarm Manager**, following nodes that will join the *swarm* are **Swarm Worker**. After that is possible to deploy the stack.

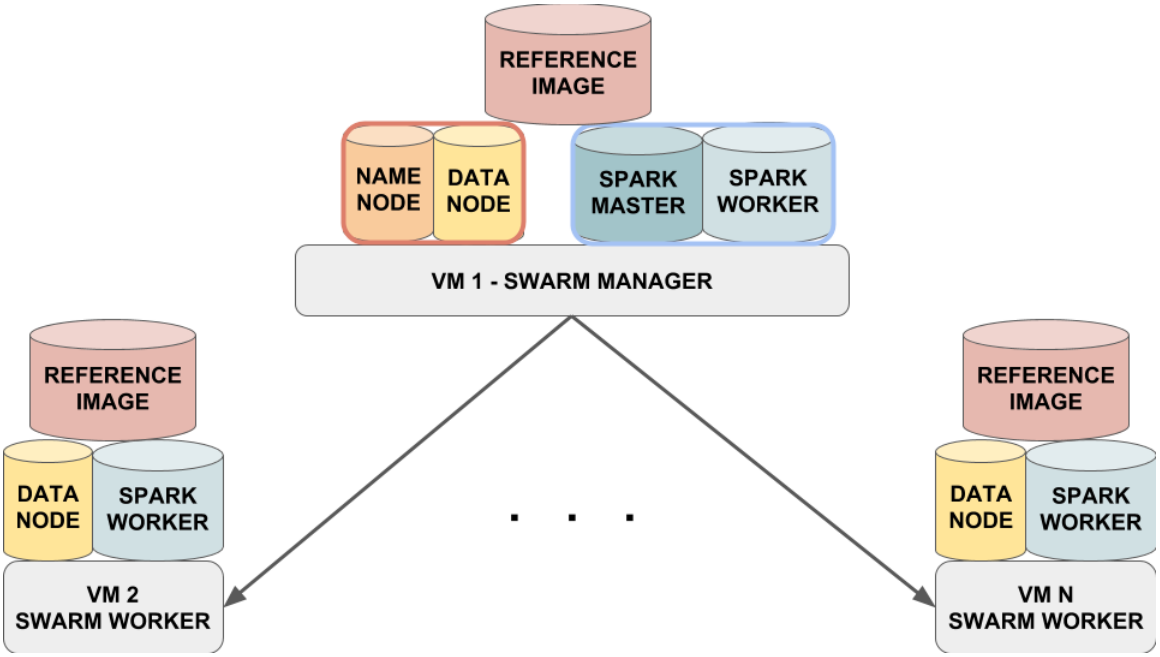


Figure 3.1: Services distribution of the NGS Pipeline over a cluster

As described in Background chapter, both HDFS and Spark have a Master-Slave architecture, where Spark Master and Namenode are masters, Datanode and Spark Worker are slaves. In a consistent manner, the masters have been deployed in the Swarm Manager, specifying an only replica in docker-compose file. Slaves may be deployed in global mode, distributing for each slave a replica in each node which is part of the Swarm. These containers will be able to communicate through a declared Overlay Network.
It is important to expose these ports, in order to get relative benefits:

- 4040 (Spark Master): provides a monitoring service of the running job, through a graphic user interface; it is very useful for debugging and understanding the job status.
- 8080 (Spark Master): gives an overview of the Spark-cluster, useful to understand if there is any communication issue and all Spark Workers correctly joined the cluster.
- 7077 (Spark Master): used to launch Spark jobs in cluster mode, indeed at the moment of executing *spark-submit* script and specifying Spark parameters, the Spark Master will be expressed in this way: *–spark-master spark://SPARK-MASTER-HOST:7077*
- 8081 (Spark Worker): gives information about the Spark Worker; indeed when executing in cluster-mode, Spark stack-trace is really poor of information. Here is possible to observe the usual stack trace printed in local-mode, or even in the Spark Worker container at */spark/work/[0-9]+/stderr*
- 50070 (Namenode): a graphic interface that allows to navigate the HDFS and other operations.

But the environment is not ready yet. Indeed just like in local execution mode, it is necessary to provide to the pipeline reference genome files, input samples, annotation databases, GATK 4.0 & 3.8, Picard ecc.
So has been used the Spark Master Docker image provided by Big Data Europe as a starting point, for then adding the required dependencies. Indeed the Spark Master Dockerfile will clone the GATK GitHub repository for then installing it, obtaining in this way GATK 4.0. With COPY command have been put GATK 3.8 and Picard jars inside the container. Eventually have been added environment variables and Spark logs service. The image of this container is available in my Docker Hub repository .
In docker-compose files have been specified **volumes**, a feature that allows to Docker containers inside a stack to access the host machine's file system. In this way the Spark Master is able to access to input samples and all annotation databases.
The reference genome in cluster mode must be treated in particular manner. As exposed

in a GATK GitHub issue, the GATK Spark tools require for .2bit reference genome in HDFS and the reference image distributed for each Spark Worker file system. This means that is necessary to distribute this file in each node of the cluster, so that the Spark Worker is able to access it. This explains the Reference Image container in Figure 3.1, indeed has been used a Docker container with global deployment mode, to globally distribute to all the Swarm nodes the reference genome image. Through the volumes is then possible to give to Spark Worker the access to Reference Image container file system and access in this way the reference genome image file. This approach has been preferred since that in this way is easy to change reference genome. If the user would like to use another reference genome, he only needs to create a container with the relative reference image inside.

3.2 Cluster-mode Execution

Now that the environment has been provisioned, is possible to run the service. An overview of the work-flow has been illustrated in Figure 3.2, where the entire process can be divided in 4 steps:

1. **FastqToSam** - as previously anticipated in section *Sparkifying not-Spark tools*, for this moment it is not possible to use in distributed mode these tools, only in local mode. Indeed as shown in the picture, only the Spark Master container is accountable to execute this tool, without distributing the work-load to Spark-Worker containers in the various Swarm nodes. Anyway this processing gets benefit from the number of cores of the host machine, when there are more sample files to convert in uBAM.

2. **Load files to HDFS** - when the uBAM files produced by FastqToSam are available, is possible to distribute them in HDFS. Not only the uBAM files, but even reference files and known sites files will be loaded in HDFS. In particular the script will use the Namenode container as interface, in order to use HDFS commands and distribute the specified files over Datanodes.

3. **GATK Spark Tools** - these tools, when executed in distributed mode, require

some input files from HDFS, that are the files distributed in the previous step. In particular are:

a) *BwaAndMarkDuplicatesPipelineSpark*: this tool requires the uBAM file and reference genome .2bit file in HDFS as previously mentioned, and reference genome in local file system. This last requirement has been achieved through Reference Image container as previously explained. Moreover in this GATK GitHub issue has been discussed about an important parameter, *–bam-partition-size*. Without this parameter, the relative default value is set to 0, with the result of getting stuck during the execution of this tool. Setting this parameter to 4000000, as expressed in that topic, the execution proceeds and reaches the completion. From tool documentation: *maximum number of bytes to read from a file into each partition of reads. Setting this higher will result in fewer partitions. Note that this will not be equal to the size of the partition in memory. Defaults to 0, which uses the default split size (determined by the Hadoop input format, typically the size of one HDFS block). Default value: 0.*

b) *BQSRPipelineSpark*: here is requested the output file of the previous tool in HDFS (not a problem, since is possible to write it directly in HDFS), the reference .2bit file in HDFS (the same for BwaAndMarkDuplicatesPipelineSpark) and known sites in HDFS, which have already been loaded previously in HDFS. When trying to execute it in cluster mode, I faced with a Spark performance regression, that has been patched in following GATK update in version 4.0.2.0.

c) *HaplotypeCallerSpark*: similar to BQSRPipelineSpark, is requested the output generated by the previous tool in HDFS and reference genome .2bit file in HDFS. The Spark performance regression was even here, but the patch made by GATK resolved this issue too.

4. ***Sparkified* Tools** - just like for FastqToSam, even other Sparkified Tools (already described in Pipeline chapter) may be executed only in local mode, taking the input file from normal file system. So before of executing this tools is necessary

to move the output generated by HaplotypeCallerSpark in HDFS to normal file system. Even here tools get benefit from the number of cores of the host machine.

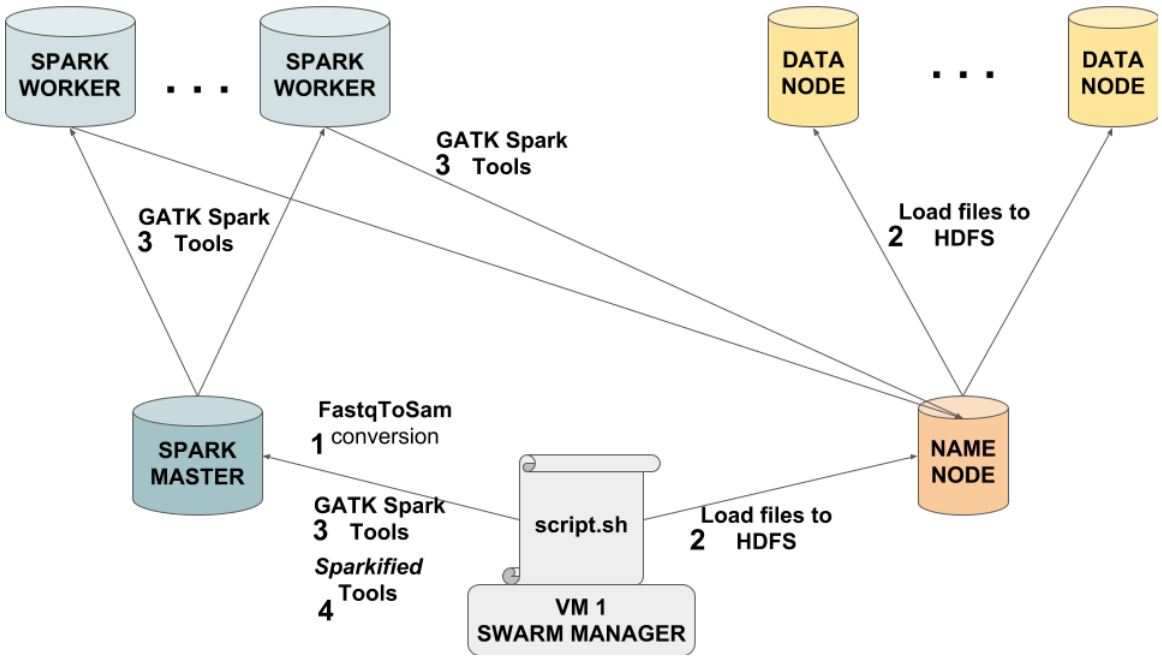


Figure 3.2: Pipeline work-flow in Distributed mode

Chapter 4

Datasets

As mentioned in previous chapters, many tools of the pipeline required specific datasets in input on the file system.

4.1 Input samples

The input samples of the pipeline, as exposed in Background chapter are compressed Whole Exome Sequencing genomes, of about 15 GB size. In particular has been used a cohort of anonymised patients, provided by the Institute of Genetic Medicine in Newcastle upon Tyne.

4.2 Reference Genome

A reference genome is a digital nucleic acid sequence database, assembled by scientists as a representative example of a species' set of genes. As they are often assembled from the sequencing of DNA from a number of donors, reference genomes do not accurately represent the set of genes of any single person. Instead a reference provides a haploid mosaic of different DNA sequences from each donor.
As the cost of DNA sequencing falls, and new full genome sequencing technologies emerge, more genome sequences continue to be generated. Reference genomes are typically used as a guide on which new genomes are built, enabling them to be assembled much more quickly and cheaply than the initial Human Genome Project. For much

of a genome, the reference provides a good approximation of the DNA of any single individual [Wikb].

In Table 4.1 have been reported the recent human genome assemblies. In this project has been used the hg19 provided by the University of California, Santa Cruz. Is possible to download it from Internet, from one of the mirror servers. For the pipeline are even necessary further files in the folder containing the reference genome. Through proper GATK tools they can be created.

Table 4.1: Recent human genome assemblies

Release name	Date of release	Equivalent UCSC version
GRCh38	Dec 2013	hg38
GRCh37	Feb 2009	hg19
NCBI Build 36.1	Mar 2006	hg18
NCBI Build 35	May 2004	hg17
NCBI Build 34	Jul 2003	hg16

4.3 Known Sites

Each tool uses known sites differently, but what is common to all is that they use them to help distinguish true variants from false positives, which is very important to how these tools work. If not provided, the statistical analysis of the data will be skewed, which can dramatically affect the sensitivity and reliability of the results. GATK provides sets of known sites in the human genome as part of their resource bundle, giving specific recommendations on which sets to use for each tool in the variant calling pipeline [Ins27]. Indeed in this pipeline have been used for BaseRecaibrator and VariantRecalibrator the known sites reported in Table 4.2.

Table 4.2: Required Known Sites

Tool	dbSNP 129	dbSNP >132	Mills indels	1KG indels	HapMap	Omni
BaseRecalibrator		X	X	X		
VariantRecalibrator		X	X		X	X

- **BaseRecalibrator**: This tool requires known SNPs and INDELs passed with the *–known-sites* argument to function properly. Have been used all the following files:
  - The most recent dbSNP release (build ID $>$ 132)
  - Mills_and_1000G_gold_standard.indels.hg19.vcf
  - 1000G_phase1.indels.b37.vcf (currently from the 1000 Genomes Phase I indel calls)
- **VariantRecalibrator**: resource datasets and arguments that GATK recommends for use in the two steps of VQSR (i.e. the successive application of VariantRecalibrator and ApplyRecalibration).
  - Resources for SNPs
    - True sites training resource: *HapMap*
      This resource is a SNP call set that has been validated to a very high degree of confidence. The program will consider that the variants in this resource are representative of true sites (truth=true), and will use them to train the recalibration model (training=true). These sites will be used later on to choose a threshold for filtering variants based on sensitivity to truth sites. The prior likelihood assigned to these variants is Q15 (96.84%).
    - True sites training resource: *Omni*
      This resource is a set of polymorphic SNP sites produced by the Omni geno- typing array. The program will consider that the variants in this resource are representative of true sites (truth=true), and will use them to train the recalibration model (training=true). The prior likelihood assigned to these variants is Q12 (93.69%).
    - Non-true sites training resource: *1000G*
      This resource is a set of high-confidence SNP sites produced by the 1000 Genomes Project. The program will consider that the variants in this re-

source may contain true variants as well as false positives (truth=false), and will use them to train the recalibration model (training=true). The prior likelihood assigned to these variants is Q10 (90%).

* Known sites resource, not used in training: *dbSNP*
  This resource is a call set that has not been validated to a high degree of confidence (truth=false). The program will not use the variants in this resource to train the recalibration model (training=false). However, the program will use these to stratify output metrics such as Ti/Tv ratio by whether variants are present in dbsnp or not (known=true). The prior likelihood assigned to these variants is Q2 (36.90%).

- Resources for INDELs
  * True sites training resource: *Mills*
    This resource is an INDEL call set that has been validated to a high degree of confidence. The program will consider that the variants in this resource are representative of true sites (truth=true), and will use them to train the recalibration model (training=true). The prior likelihood assigned to these variants is Q12 (93.69%).
  * Known sites resource, not used in training: *dbSNP*
    This resource is a call set that has not been validated to a high degree of confidence (truth=false). The program will not use the variants in this resource to train the recalibration model (training=false). However, the program will use these to stratify output metrics such as Ti/Tv ratio by whether variants are present in dbsnp or not (known=true). The prior likelihood assigned to these variants is Q2 (36.90%) [Ins25].

4.4 Variant Annotation

The process of Variant Annotation has been delivered through ANNOVAR. The easiest way to use ANNOVAR is to use the *table_ annovar.pl* program. This program takes an input variant file (such as a VCF file) and generate a tab-delimited output file with many columns, each representing one set of annotations. Additionally, if the input is a

VCF file, the program also generates a new output VCF file with the INFO field filled with annotation information.

After downloaded and unpacked the ANNOVAR package, is possible to observe that the bin/ directory contains several Perl programs with .pl suffix. First, is necessary to download appropriate database files using *annotate_variation.pl*, for then running the *table_annovar.pl* program to annotate the variants in the input file. In this project have been downloaded and used these databases: knownGene, ensGene, refGene, phastConsElements46way, genomicSuperDups, esp6500si_all, 1000g2012apr_all, cg69, snp137, ljb26_all. Similarly to the previous pipeline [P.M12].

Fields that does not have any annotation will be filled by "." string. Opening the output file in Excel shows what it contains [ANN].

Chapter 5

Performance Analysis

The goal of this project is to introduce Apache Spark in these NGS Pipelines in order to improve their efficiency. Here will be exposed the execution times of tools with different Spark and hardware configurations.

5.1 Local Mode

As mentioned in previous chapters, the Preprocessing phase is the most time and resource consuming, with some numbers will be expressed the different execution times between the three phases.

5.1.1 Preprocessing 6 genomes

In Tables 5.1 5.2 5.3 are listed the execution times of the preprocessing phase using 6 input samples, reporting the Spark configuration parameters too. The first step is the conversion through *FatqToSam*: since that it is a *Sparkified* tool, it benefits from only the cores of the host machine. Indeed while it takes 46 minutes to process an only genome, processing 6 genomes takes only 70 minutes on a single VM, applying each of the 6 conversions in parallel.
As exposed in previous chapters, GATK Spark tools are meant to process a sample for each, so with a first *for loop* have been executed the BwaAndMarkDuplicatesPipelineSpark on each input sample; each of the generated output has been used as input of *for*

loop of BQSRPipelineSpark and the same thing for HaplotypeCallerSpark. This means that the preprocessing phase for 6 samples lasted 3695,31 minutes (obtained summing the execution times of the 3 tables).
Moreover this computation has been executed one more time, with a Spark parameters tuning, with results observable in Tables 5.4 5.5 5.6. By simply summing the execution times of these tables (as already done previously), we obtain 3511,7 minutes. This means that by simply tuning Spark parameters and using the same VM is possible to obtain some speed-up. Even comparing, for example, the execution times of BwaAndMarkDuplicatesPipelineSpark in Tables 5.1 and 5.4 it is observable that the time values reported in the second configuration are better than the first one. It is the same even for BQSRPipelineSpark, while HaplotypeCallerSpark is quite unpredictable: indeed looking at Charts 5.1 5.2, HaplotypeCallerSpark shows a strange behaviour, since in certain executions took less time with bigger size samples than smaller ones.

The analysis performance expressed in this subsection was conducted through GATK 4.0 BETA, when it was not an official release.

5.1.2 Spark parameters tuning

So it is worth to do some performance analysis on Spark parameters for these tools, due to find the best configuration. In Tables 5.7 5.8 5.9 are respectively reported BwaAndMarkDuplicatesPipelineSpark, BQSRPipelineSpark and HaplotypeCallerSpark benchmarking with different Spark parameters, using the same input sample and same VM.
For BwaAndMarkDuplicatesPipelineSpark seems that using driver-memory 10GB num-executors 4 executor-cores 2 executor-memory 8 is obtainable the best execution time. Even if has been executed two times with these parameters and obtained a little bit slower performance (to demonstrate that these processes are not perfectly time predictable).

Even here analysis performance was conducted through GATK 4.0 BETA.

5.1.3 Variant Discovery and Call-set Refinement

Comparing these numbers with the execution times reported for Variant Discovery and Call-set Refinement makes understandable the great difference of time executions between these three phases. In Tables 5.10 5.11 are reported the execution times of the simple tools (not *Sparkified*) respectively of the Variant Discovery and Call-set Refinement phase. These tools used the output files generated by the HaplotypeCallerSpark on the 6 samples (5.3), because the GenotypeGVCFs require more input files, with only one does not work. Has been noticed that 6 is a good number of input samples and in this this condition the tool works.
With the *Sparkified* version, which gathers respectively the tools of Variant Discovery and Call-set refinement (listed in Tables 5.10 and 5.11), was measured respectively 13,82 and 31,37 minutes. Comparing these values with the previous calculated of preprocessing, through the Chart 5.3 is possible to have an idea of the different execution times.

5.1.4 Scale up

In order to introduce parallelism in this pipeline (or in general) there are two approaches: *scale up* and *scale out*.
With the first approach the work load is distributed on more virtual cores of the same VM, which means using VM with better hardware resources. In the second one the work load is distributed on hardware resources of different VMs, a computer cluster that cooperate to processing data. This is possible due to sharing intermediate results of the computation that lead to the final result. This data sharing between nodes in the same cluster is possible through networking and shared file system, which allow the nodes of the cluster to communicate each other. The inconvenient of this approach is the introduction of overhead, because these intermediate results must be communicated over the network (even if it is a local network). Scaling-up avoids these delays, and with the same combined hardware resources, this approach gives better results. So executing BwaAndMarkDuplicatesPipelineSpark on a VM with 16 cores and 110 GB of RAM takes 129,29 minutes. While BQSRPipelineSpark takes 36,62 minutes, as reported in Chart 5.4. Comparing these numbers with values reported in Tables 5.1 5.2, we almost

have an ideal speed-up, since doubling the number of cores, almost halves the execution time (in next section will be discussed the scale out approach).
While HaplotypeCallerSpark with 16 cores took more time. Analysing the log output of the execution, there is a warning that explain the reason of so much long execution time:

Listing 5.1: Missing AVX instruction

```
WARN  PairHMM - ***WARNING:
Machine does not have the AVX instruction set support needed for
the accelerated AVX PairHmm.
Falling back to the MUCH slower LOGLESS_CACHING implementation!
```

It seems that resizeing the Azure VM to 16 cores, provides a different CPU. This one is not compatible with HaplotypeCallerSpark requirements and so it uses a much slower implementation.
Moreover these tools have been executed even on a VM with 32 cores and 120 GB of RAM, reporting all these results of scaleing up in Chart 5.4. As observable from this chart, using 32 cores lead to an unpredicted behaviour: instead to further decrease the execution time of the tool, this value just increased instead. Probably this is due to the limit of obtainable speed-up with this configuration. Further analysis may be required to understand the reason of this behaviour.

Here was used GATK 4.0.2.0 release. It is interesting to notice that using this release for previous exposed cases changes their execution times, increasing this value. They are still working on these tools and there are still known issues to resolve.

5.2 Cluster Mode

As mentioned in Clustering chapter, finding the right configuration was pretty hard. After finding the right parameter in a GitHub issue, the pipeline have been executed on a cluster of two VMs. The results are observable in Table 5.12 for BwaAndMarkDuplicatesPipelineSpark. Unfortunately there was still some unresolved problem in executing HaplotypeCallerSpark.

Anyway from the reported results, is possible to notice that comparing the second row with results reported in Table 5.7 (which means using of VMs with same hardware, changing only the number of VMs) have been obtained a certain speed-up, passing from about 250 minutes to 169,29. Comparing this value with the 129,29 minutes obtained scaling up, confirms what exposed previously. In both cases have been used 16 cores and 110 GB of RAM, but in the scale-up the resources were on a single VM, giving better performance. In the scale-out these resources have been distributed in 2 nodes (other results are observable at Chart 5.5).
While analysing the result of the first row of Table **??**, in which has been used a 2 Nodes Cluster with double hardware resources per VM, which means each VM has 16 cores and 110 GB of RAM, gives even an higher speed-up, with 94,63 minutes.

Here was used GATK 4.0.2.0 release too.

5.3 Microsoft Genomics

Microsoft give the possibility to execute their genomic pipeline described in their paper [Gen]. Their approach is to run separately each genome on a single high-capacity virtual machine, to maximize throughput and minimize communication and storage overhead within each run. So instead of using a cluster which would introduce delays due to network and disk accesses, using an only high-capacity virtual machine avoids these delays. Their pipeline arrives until the Haplotype Caller (in other words is only the Preprocessing phase described in this thesis), without Variant Calling and Callset Refinement phases. Moreover has an option to produce a gVCF file, which can be merged across multiple samples for joint genotyping, but for this moment it is not available. Only VCF files are produced at the end of this process, while the pipeline implemented in this project produces gVCF files.
The key difference between a regular VCF and a gVCF is that the gVCF has records for all sites, whether there is a variant call there or not. The goal is to have every site represented in the file in order to do joint analysis of a cohort in subsequent steps. The records in a gVCF include an accurate estimation of confidence in the determination that the sites are homozygous-reference or not. This estimation is generated by the

HaplotypeCaller's built-in reference model [Ins22].
Analysing their performance, processing the sample PFC_0028 reported in Tables 5.7 5.8 5.9 takes only 76,88 minutes. Unfortunately they do not provide info about the VM specifics which process this sample.

5.3.1 Pricing Analysis

What they provide is a pricing rate, where is reported the £0.217/gigabase pricing. Which means that processing the 6 samples exposed in subsection *Preprocessing 6 genomes* through the Microsoft Genomics pipeline costs £18.61, since that should be processed roughly 85 gigabases. Making a comparison with the pipeline implemented in this project, for processing the first mentioned 6 genomes were necessary 3511,7 minutes on a VM with 8 cores and 55 GB of RAM, which pricing is of £354,89/month. With some calculations, processing 6 genomes costs £28,84 here. So the Microsoft Genomics pipeline is even cheaper.

Table 5.1: BwaAndMarkDuplicatesPipelineSpark Execution Times

Tool	Input	driver-memory	num-executors	executor-cores	executor-memory	VM	Time (m)
BWA&MD	PFC_0033 15,9 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	306,57
BWA&MD	PFC_0032 14,4 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	270,10
BWA&MD	PFC_0031 10,8 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	194,21
BWA&MD	PFC_0030 13,2 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	261,93
BWA&MD	PFC_0029 13 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	264,71
BWA&MD	PFC_0028 14,2 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	282,81

Table 5.2: BQSRPipelineSpark Execution Times

Tool	Input	driver-memory	num-executors	executor-cores	executor-memory	VM	Time (m)
BQSR	PFC_0033 15,9 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	95,91
BQSR	PFC_0032 14,4 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	78,66
BQSR	PFC_0031 10,8 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	60,38
BQSR	PFC_0030 13,2 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	72,55
BQSR	PFC_0029 13 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	76,92
BQSR	PFC_0028 14,2 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	73,53

Table 5.3: HaplotypeCallerSpark Execution Times

Tool	Input	driver-memory	num-executors	executor-cores	executor-memory	VM	Time (m)
HC	PFC_0033 15,9 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	380,19
HC	PFC_0032 14,4 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	296,59
HC	PFC_0031 10,8 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	222,01
HC	PFC_0030 13,2 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	226,70
HC	PFC_0029 13 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	300,07
HC	PFC_0028 14,2 GB	20 GB	2	4	16 GB	8 Cores 55g RAM	231,47

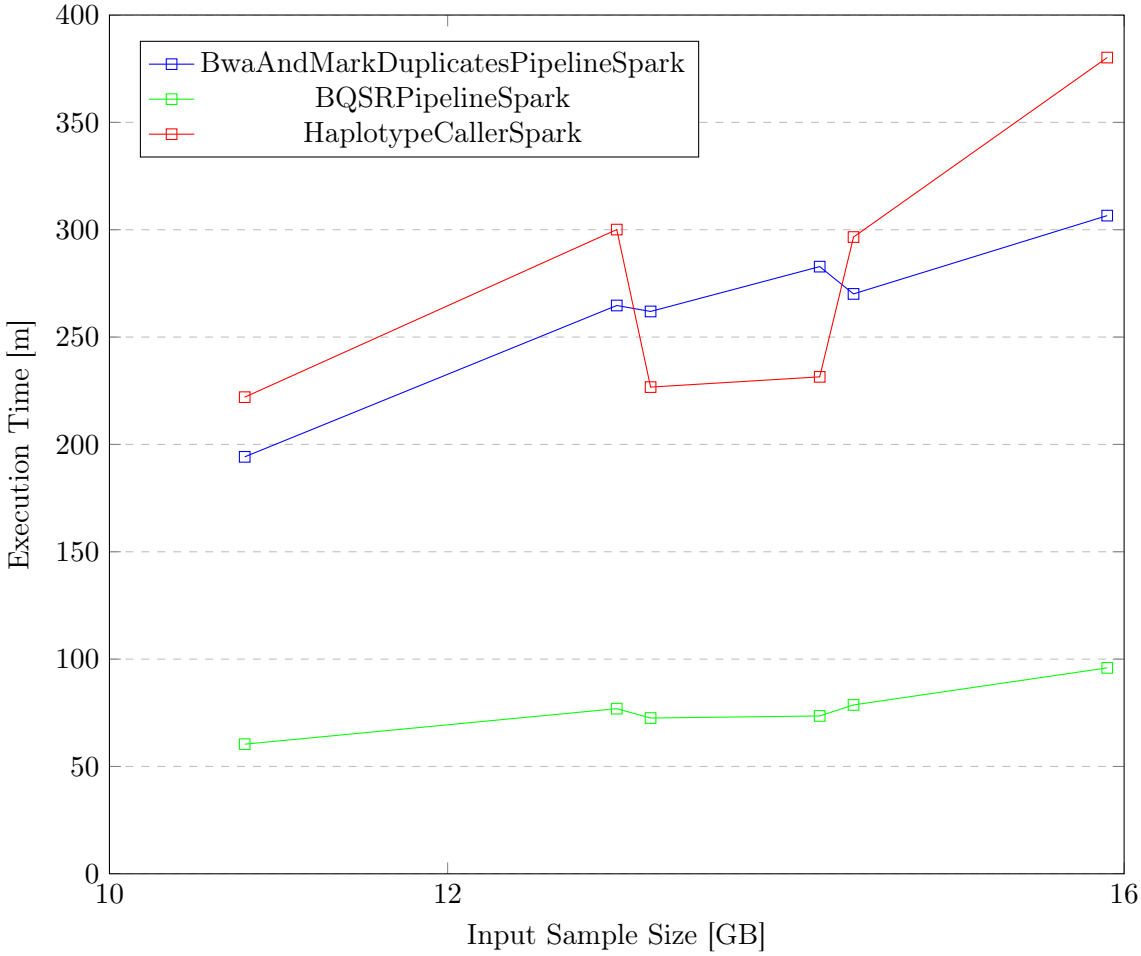


Chart 5.1: Preprocessing of 6 samples in local-mode (8 Cores 55GB RAM) using Spark parameters: –driver-memory 20GB –num-executors 2 –executor-cores 4 –executor-memory 16GB

Table 5.4: BwaAndMarkDuplicatesPipelineSpark Execution Times

Tool	Input	driver-memory	num-executors	executor-cores	executor-memory	VM	Time (m)
BWA&MD	PFC_0033 15,9 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	292,06
BWA&MD	PFC_0032 14,4 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	258,65
BWA&MD	PFC_0031 10,8 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	188,03
BWA&MD	PFC_0030 13,2 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	247,00
BWA&MD	PFC_0029 13 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	249,50
BWA&MD	PFC_0028 14,2 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	265,71

Table 5.5: BQSRPipelineSpark Execution Times

Tool	Input	driver-memory	num-executors	executor-cores	executor-memory	VM	Time (m)
BQSR	PFC_0033 15,9 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	92,31
BQSR	PFC_0032 14,4 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	77,08
BQSR	PFC_0031 10,8 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	54,50
BQSR	PFC_0030 13,2 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	69,45
BQSR	PFC_0029 13 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	73,28
BQSR	PFC_0028 14,2 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	72,25

Table 5.6: HaplotypeCallerSpark Execution Times

Tool	Input	driver-memory	num-executors	executor-cores	executor-memory	VM	Time (m)
HC	PFC_0033 15,9 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	434,23
HC	PFC_0032 14,4 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	260,60
HC	PFC_0031 10,8 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	226,21
HC	PFC_0030 13,2 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	186,72
HC	PFC_0029 13 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	268,43
HC	PFC_0028 14,2 GB	20 GB	4	2	8 GB	8 Cores 55g RAM	195,69

Table 5.7: BwaAndMarkDuplicatesPipelineSpark Benchmarking

Tool	Input	driver-memory	num-executors	executor-cores	executor-memory	VM	Time (m)
BWA&MD	PFC_0028	20 GB	4	2	8 GB	8 Cores 55g RAM	270,27
BWA&MD	PFC_0028	10 GB	4	2	8 GB	8 Cores 55g RAM	251,8
BWA&MD	PFC_0028	10 GB	4	2	8 GB	8 Cores 55g RAM	258,3
BWA&MD	PFC_0028	10 GB	2	4	8 GB	8 Cores 55g RAM	255,74
BWA&MD	PFC_0028	10 GB	2	4	8 GB	8 Cores 55g RAM	254,91
BWA&MD	PFC_0028	10 GB	8	1	6 GB	8 Cores 55g RAM	281,85
BWA&MD	PFC_0028	5 GB	4	2	12 GB	8 Cores 55g RAM	279,18
BWA&MD	PFC_0028	5 GB	4	2	8 GB	8 Cores 55g RAM	264,99

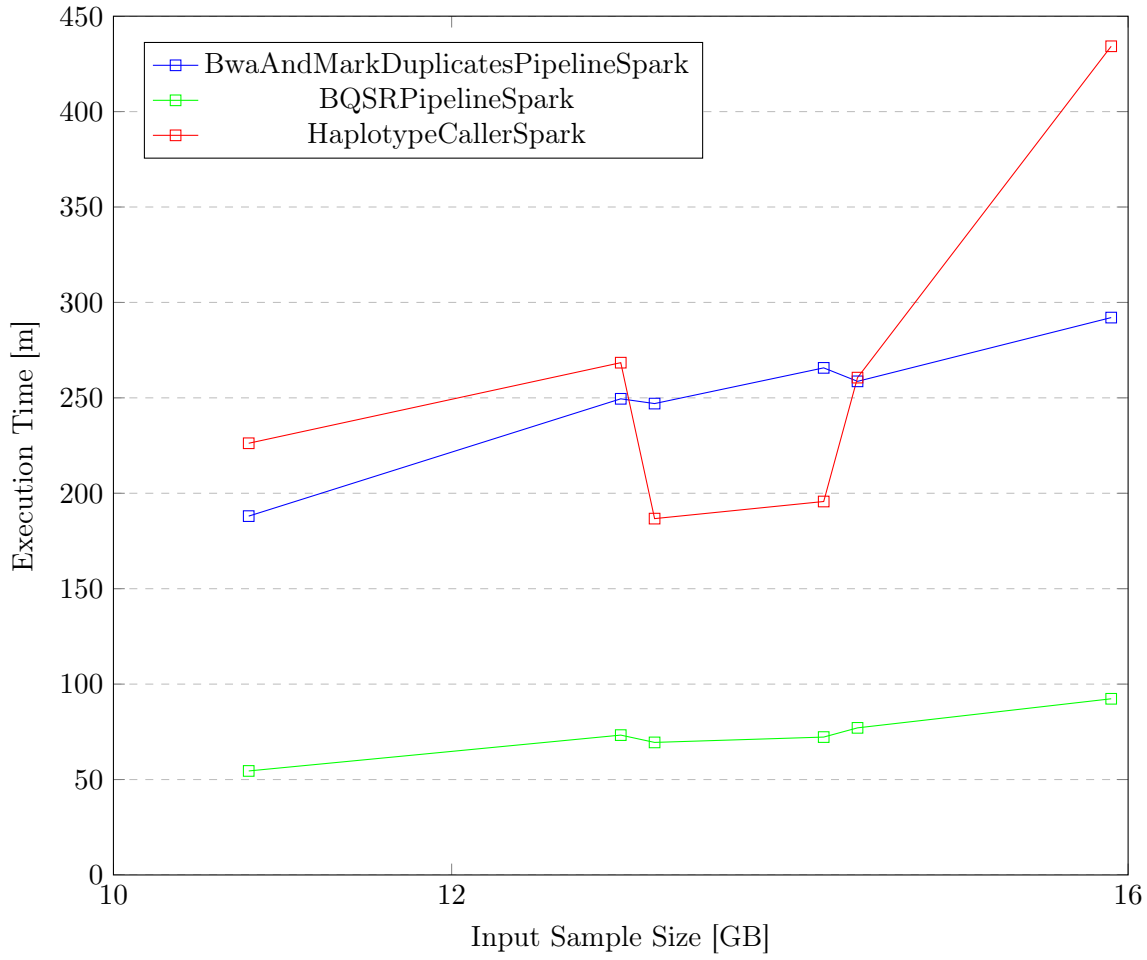


Chart 5.2: Preprocessing of 6 samples in local-mode (8 Cores 55GB RAM) using Spark parameters: –driver-memory 20GB –num-executors 4 –executor-cores 2 –executor-memory 8GB

Table 5.8: BQSRPipelineSpark Benchmarking

Tool	Input	driver-memory	num-executors	executor-cores	executor-memory	VM	Time (m)
BQSR	PFC_0028	5 GB	4	2	8 GB	8 Cores 55g RAM	71,66
BQSR	PFC_0028	10 GB	4	2	8 GB	8 Cores 55g RAM	71,43
BQSR	PFC_0028	10 GB	8	1	6 GB	8 Cores 55g RAM	74,98

Table 5.9: HaplotypeCallerSpark Benchmarking

Tool	Input	driver-memory	num-executors	executor-cores	executor-memory	VM	Time (m)
HC	PFC_0028	10 GB	8	1	6 GB	8 Cores 55g RAM	227,36
HC	PFC_0028	10 GB	4	2	8 GB	8 Cores 55g RAM	230,7
HC	PFC_0028	20 GB	4	2	8 GB	8 Cores 55g RAM	225,85
HC	PFC_0028	20 GB	2	4	8 GB	8 Cores 55g RAM	191,49
HC	PFC_0028	20 GB	1	8	8 GB	8 Cores 55g RAM	195,28
HC	PFC_0028	20 GB	2	4	16 GB	8 Cores 55g RAM	194,07
HC	PFC_0028	20 GB	2	4	4 GB	8 Cores 55g RAM	233,31

Table 5.10: Variant Discovery

Tool	Input	VM	Time (m)
GenotypeGVCFs	PFC(28-29-30-31-32-33)	8 Cores 55g RAM	9,47 + 8,38
VariantRecalibratorSNP	PFC(28-29-30-31-32-33)	8 Cores 55g RAM	9,53
ApplyRecalibrationSNP	PFC(28-29-30-31-32-33)	8 Cores 55g RAM	0,42
VariantRecalibratorINDEL	PFC(28-29-30-31-32-33)	8 Cores 55g RAM	1,06
ApplyRecalibrationINDEL	PFC(28-29-30-31-32-33)	8 Cores 55g RAM	0,42

Table 5.11: Call-set Refinement

Tool	Input	VM	Time (m)
CalculateGenotypePosteriors	PFC(28-29-30-31-32-33)	8 Cores 55g RAM	1,75
VariantFiltration	PFC(28-29-30-31-32-33)	8 Cores 55g RAM	0,36
VariantAnnotator	PFC(28-29-30-31-32-33)	8 Cores 55g RAM	13
SelectVariants	PFC(28-29-30-31-32-33)	8 Cores 55g RAM	1,62
convert2annovar.pl	PFC(28-29-30-31-32-33)	8 Cores 55g RAM	0,28
table_annovar.pl	PFC(28-29-30-31-32-33)	8 Cores 55g RAM	31,86
ExonicFilterSpark	PFC(28-29-30-31-32-33)	8 Cores 55g RAM	1,23

Table 5.12: Cluster execution times

Tool	Input	VM	Time (m)
BWA&MD	PFC_0028 14,2 GB	16 Cores 110g RAM (2 nodes)	94,63
BWA&MD	PFC_0028 14,2 GB	8 Cores 55g RAM (2 nodes)	169,29
BWA&MD	PFC_0028 14,2 GB	8 Cores 55g RAM (3 nodes)	111,96
BQSR	PFC_0028 14,2 GB	8 Cores 55g RAM (2 nodes)	60,54
BQSR	PFC_0028 14,2 GB	8 Cores 55g RAM (3 nodes)	55,87

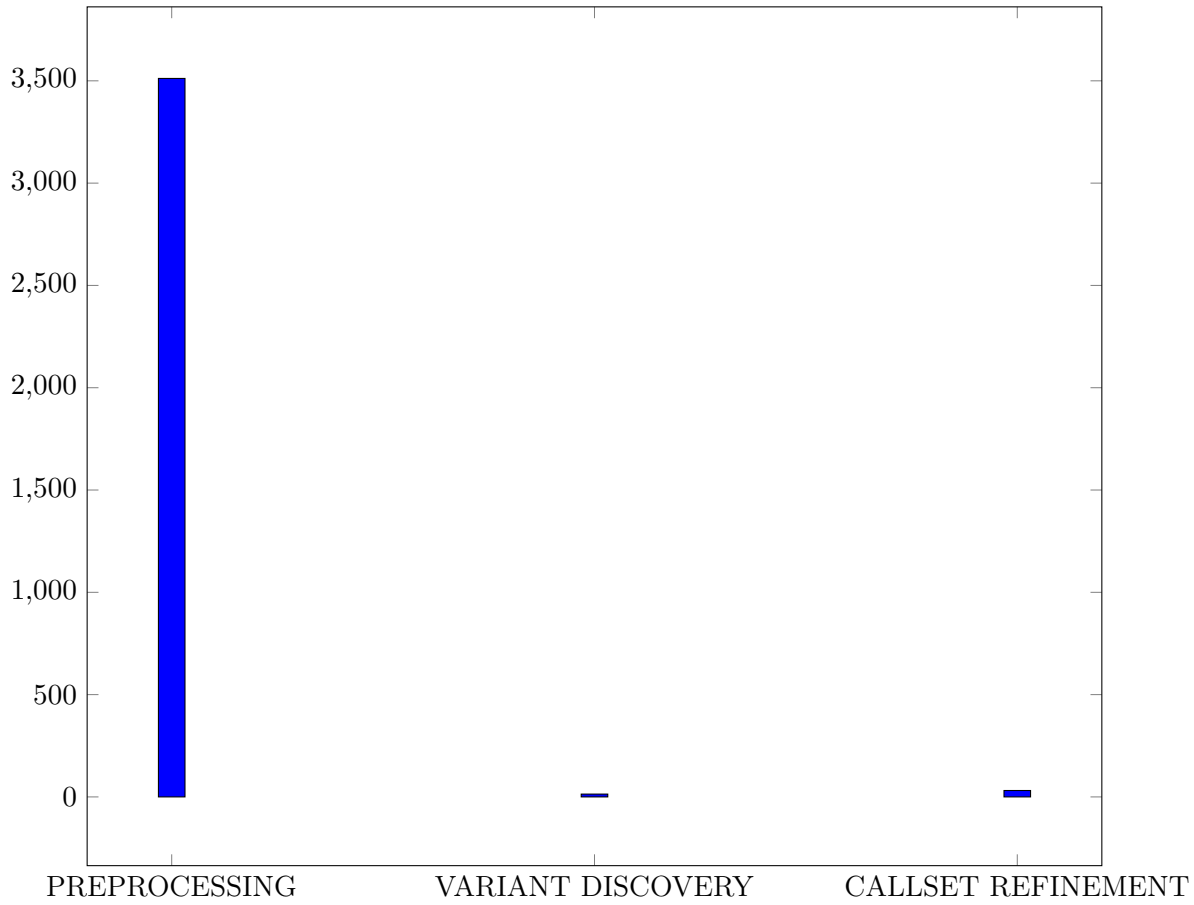


Chart 5.3: Preprocessing, Variant Discovery and Call-set Refinement execution times with 6 input samples)

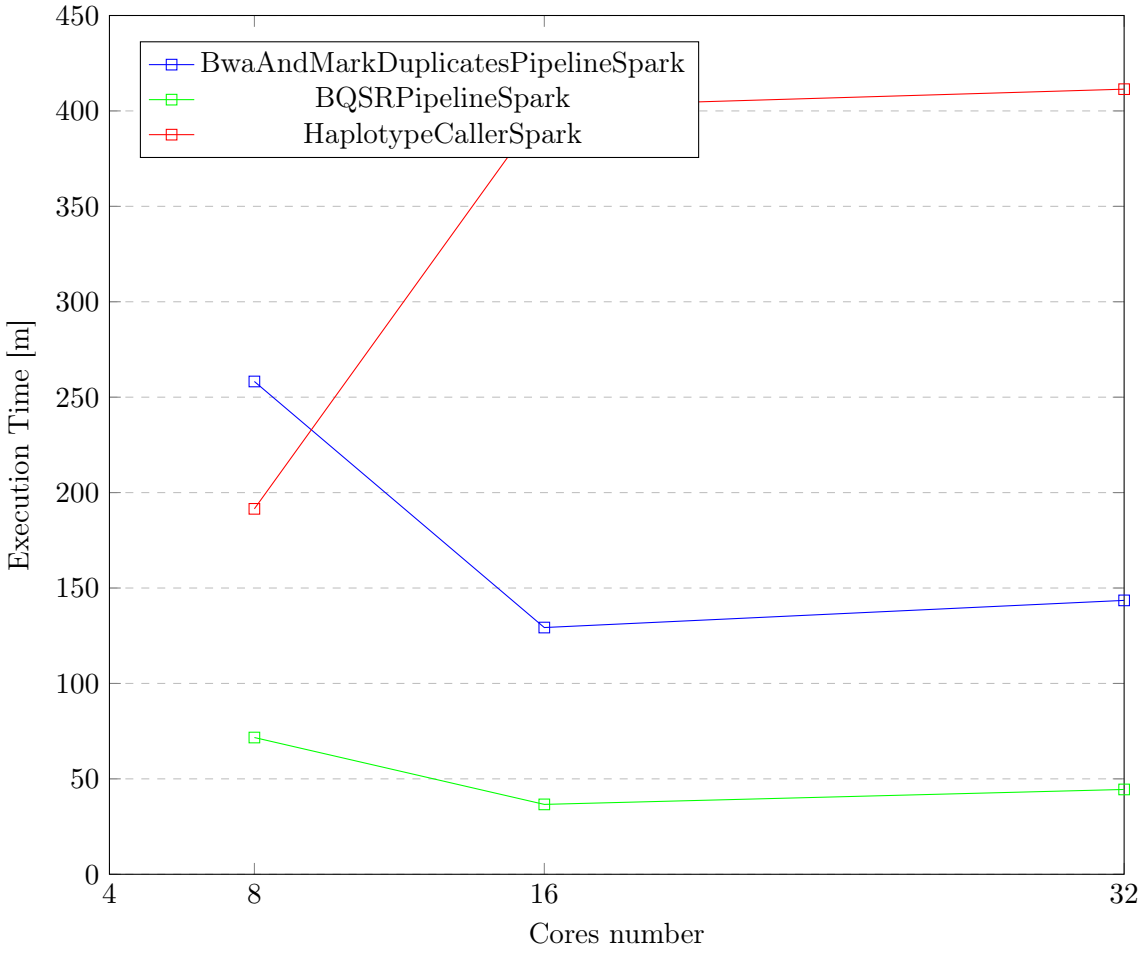


Chart 5.4: Analysis performance variating the VM cores number (scale-up)

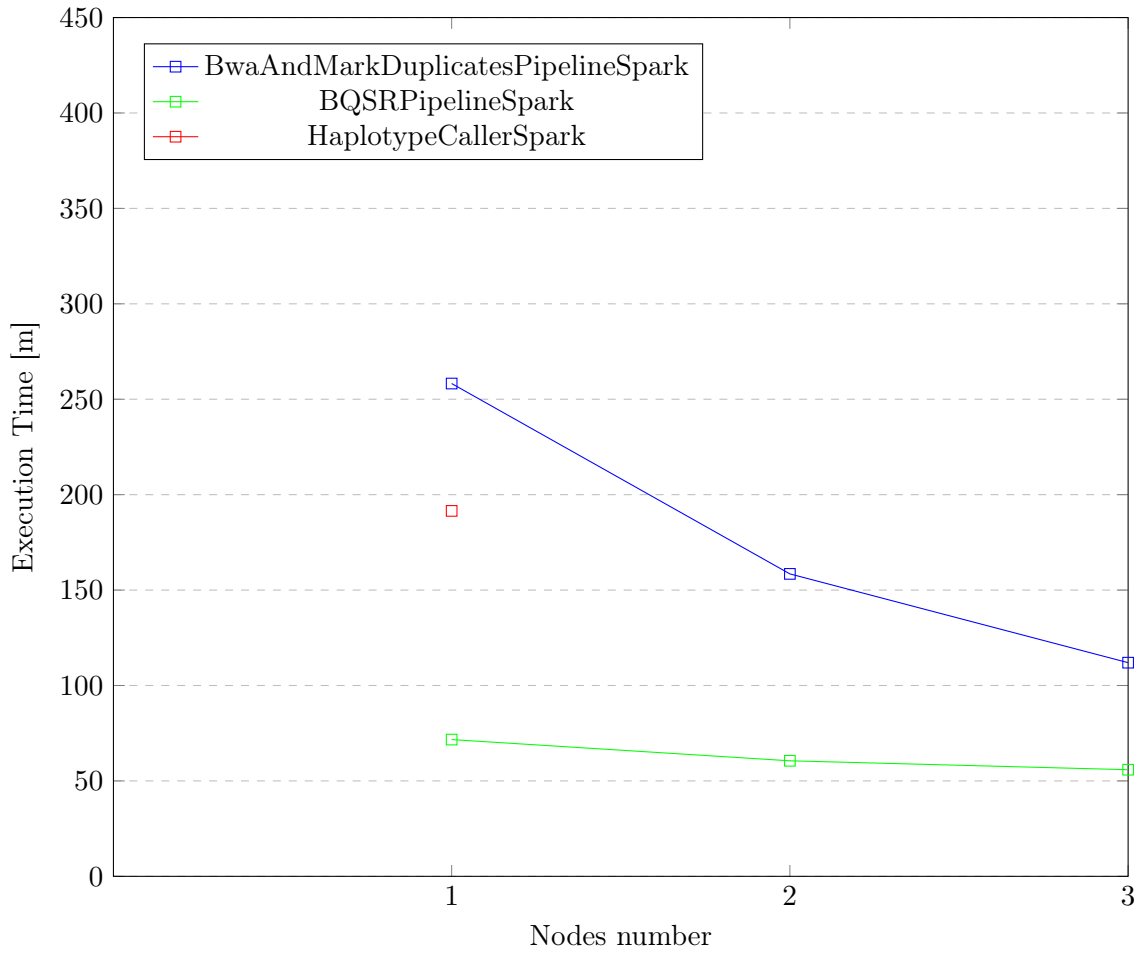


Chart 5.5: Analysis performance variating the nodes number (scale-out), where each node is a 8 cores 55 GB RAM VM

Conclusions and Future Developments

The analysis performance conducted in the final chapter was conducted using different realeases of the GATK 4.0 version. This led to pretty different execution times for the same GATK tool, according to the release version used. So it is valuable to re-conduct these analysis performance when there will be a stable version of these tools. In particular for cluster-mode execution, since that in the last GATK release there are some problems in executing HaplotypeCallerSpark.

As exposed in previous chapters, Sparkifed tools actually are implemented to be executed on a single machine, since a bash script must be generated on the local file system. It could be implemented in cluster-mode, maybe trying to generate this script on HDFS and so all the Spark Workers would be able to access to the bash script, due to execute it in distributed mode through Spark.

The final output result generated from the pipeline is reliable, since compared with the result generated from an other pipeline [P.M12]. But there are some differences and things to do:

- Files generated by this pipeline are nearly four times larger than what the e-SC pipeline returns, but that is mainly because in this pipeline was not filtered VCF by quality. In the e-SC pipeline was used Phred=30. It should be filter out the variants that have quality less than 30.

- There are some extra rows which the other pipeline filter out like 'chrM', 'chr1_gl000...', 'chrUn_gl000...'. The first is the mitochondrial DNA, the other are some unknown reads, possibly coming from sample contamination. These may be filtered too.
- There are some redundant columns in output, e.g. column 38 is exactly the same as column 11 (both include some ensGene references like ENSG00000198888), column 17 includes only dots '.'

When executed in cluster-mode, as exposed in chapter Clustering, after converting the input FASTQ samples through *FastqToSam*, is then necessary to load the generated outputs in HDFS (within other files). Instead of waiting the output generated from the tool, maybe is possible to put it in a *named pipe*, which will be the input for the HDFS loading command. In this way probably the overhead of loading the converted samples in HDFS could be avoided, so is valuable to explore this approach.

HaplotypeCallerSpark presented unexpected execution times when scaling up or out, because of this warning:

Machine does not have the AVX instruction set support needed for the accelerated AVX PairHmm. Falling back to the MUCH slower LOGLESS_CACHING implementation !

Maybe it is due to use of GATK 4.0.2 release or maybe for some hardware alteration; anyway this led to unexpected results. It is interesting to understand the reason of this warning and attempt to resolve it in order to report correct execution times.

Execution of GATK Spark tools on a 32 cores VM did not lead to any speed-up, while executing on 2 VM with 16 cores or 3 VM with 8 cores led to further speed-up instead. It is interesting to understand the reason why a 32 cores VM did not lead speed-up.

Bibliography

[ANN] Quick start-up guide - annovar documentation. http://annovar.openbioinformatics.org/en/latest/user-guide/startup/.

[Apa] Apache. Cluster mode overview. https://spark.apache.org/docs/latest/cluster-overview.html.

[Bor08] Dhruba Borthakur. Hdfs architecture guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2013-04-08.

[Doca] Docker. Get started, part 1: Orientation and setup. https://docs.docker.com/get-started/.

[Docb] Docker. Get started, part 4: Swarms. https://docs.docker.com/get-started/part4/.

[Gen] Microsoft Genomics. Accelerate precision medicine with microsoft genomics.

[Hak10] Kai Wang Mingyao Li Hakon Hakonarson. Annovar: functional annotation of genetic variants from high-throughput sequencing data. http://dx.doi.org/10.1093/nar/gkq603, 2010.

[HR] Li H. and Durbin R. Fast and accurate short read alignment with burrows-wheeler transform.

[Ill] Illumina. Paired-end vs. single-read sequencing technology. https://emea.illumina.com/science/technology/next-generation-sequencing/paired-end-vs-single-read-sequencing.html.

[Ins] Broad Institute. Picard. https://broadinstitute.github.io/picard/.

[Ins22] Broad Institute. What is a gvcf and how is it different from a 'regular' vcf? https://software.broadinstitute.org/gatk/documentation/article.php?id=4017, 2014-10-22.

[Ins07] Broad Institute. Germline short variant discovery (snps + indels). https://software.broadinstitute.org/gatk/best-practices/workflow?id=11145, 2018-01-07.

[Ins09] Broad Institute. Variant quality score recalibration (vqsr). https://software.broadinstitute.org/gatk/documentation/article?id=11084, 2018-01-09.

[Ins21] Broad Institute. Calling variants on cohorts of samples using the haplotypecaller in gvcf mode. https://software.broadinstitute.org/gatk/documentation/article.php?id=3893, 2017-01-21.

[Ins27] Broad Institute. What should i use as known variants/sites for running tool x? https://software.broadinstitute.org/gatk/documentation/article.php?id=1247, 2016-01-27.

[Ins29] Broad Institute. Base quality score recalibration (bqsr). https://software.broadinstitute.org/gatk/documentation/article?id=11081, 2017-12-29.

[Ins30] Broad Institute. What is a vcf and how should i interpret it? https://software.broadinstitute.org/gatk/documentation/article.php?id=1268, 2017-05-30.

[Ins06a] Broad Institute. Genotype refinement workflow for germline short variants. https://software.broadinstitute.org/gatk/documentation/article?id=11074, 2017-09-06.

[Ins06b] Broad Institute. (howto) recalibrate variant quality scores = run vqsr. https://software.broadinstitute.org/gatk/documentation/article?id=2805, 2017-09-06.

[Ins25] Broad Institute. Which training sets / arguments should i use for running vqsr? https://software.broadinstitute.org/gatk/documentation/article.php?id=1259, 2017-09-25.

[P.M12] J.Cała E.Marei Y.Xu K.Takeda P.Missier. Scalable and efficient whole-exome data processing using workflows on the cloud. *Future Generation Computer Systems*, 65:153–168, 2016-12.

[Ref20a] Genetics Home Reference. What are single nucleotide polymorphisms (snps)? https://ghr.nlm.nih.gov/primer/genomicresearch/snp, 2018-02-20.

[Ref20b] Genetics Home Reference. What is a gene? https://ghr.nlm.nih.gov/primer/basics/gene, 2018-02-20.

[RMS20] Laura Rodriguez-Murillo and Rany M. SalemEmail. Insertion/deletion polymorphism. https://link.springer.com/referenceworkentry/10.1007%2F978-1-4419-1005-9_706, 2018-02-20.

[Ste05] L.D. Stein. The case for cloud computing in genome informatics. http://dx.doi.org/10.1186/gb-2010-11-5-207, 2010-11-05.

[Wika] Wikipedia. Human genome. https://en.wikipedia.org/wiki/Human_genome.

[Wikb] Wikipedia. Reference genome. https://en.wikipedia.org/wiki/Reference_genome.