Supercomputing for Big Data ET4310 (2016)
Assignment 3

Lorenzo Gasparini (4609905)

November 8, 2016

1 Introduction

This assignment is about the implementation of an in-memory distributed version of the GATK DNA analysis pipeline using Apache Spark.

The figure below shows the steps involved in a typical DNA analysis pipeline.

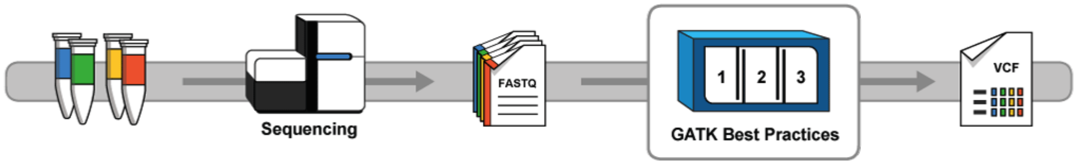

Figure 1: DNA analysis pipeline

After the DNA is sequenced, the output is stored in FASTQ files with sizes in the order of hundreds of gigabytes. Then the DNA *reads* are aligned to a reference genome, a process called *DNA assembly*. This is done through the Burrows-Wheeler Aligner in the form of the BWA-MEM algorithm, and stored in SAM format files.

Afterwards, the short reads in the SAM file are sorted and deduplicated, using the Picard tool. The output of this tool consists of files in the BAM format. Next, *variant calling* is operated on these files to identify the DNA mutations. The output of this step is a VCF (variant call format) file which contains the identified mutations in the DNA.

The assignment is divided in three steps:

1. *Chunking of the input FASTQ files*: in this step two FASTQ files are interleaved and divided into multiple chunks to enable parallelization of the pipeline
2. *DNA sequence analysis*: implementation of the pipeline itself, which takes as input the ouput of step1 and outputs the VCF file(s)
3. *Porting the solution to a cluster*: interfacing with the *Hadoop Distributed File System* to obtain the input data, save the logs and store the results.

The next sections are organised as follows: *Background* contains an overview on the Spark and Scala features used in this assignment. In *Implementation* the technical implementation details are discussed. In *Results* it's discussed whether the initial goals were achieved successfully. Finally, *Conclusion* contains a summary of the report with additional remarks on what was learnt during the assignment.

2 Background

2.1 Broadcast variables

Broadcast variables are used in Spark to keep read-only variables cached on each executor instead of shipping a copy of the variable with each task.

Typically when a function passed to a Spark operation is executed on a remote node, it uses local copies of the variables used in the function. With broadcast variables every node gets access to some common data in an efficient manner.

Broadcast variables are created by using `SparkContext.broadcast(variable)` and can then be read with `variable.value`.

They were not used in the solution of the assignment since the local mode of Spark was used.

2.2 Scala process package

The Scala `process` package is used to handle the execution of external processes. It is based on the Java `Process` and `ProcessBuilder` packages. It provides a DSL (Domain Specific Language) for running and chaining processing, in a similar fashion as the Unix shell ability to pipe the output of a process in the input of another.

There are 3 main ways to execute a command:

- Using `!` methods, if we're interested in the return value of the process
- With the `!!` methods, if we want the output of the process as a string
- Using the `lines` methods, which return the continuous output of the process as a `Stream[String]`.

Additionally, instances of `java.io.File` can be used both as an input and output to the processes, using the #> syntax for output redirection.

2.3 `mapPartitions`, `foreachPartition`

The Spark RDD programming model uses partitions to distribute data over the cluster. Sometimes it's useful to operate on each each partition individually, and that's where these methods come in.

`mapPartitions` is the analogue to the `map` method, and can be used for example to do some heavyweight initialization once for each partition instead of once for each element. `mapPartitionsWithIndex` is used when the index of the partition is needed in the computation.

Similarly, `foreachPartition` is used when there is no need to return a value but only side effects (like writing to a database or saving to a file) are done.

2.4 Hadoop, YARN

Apache Hadoop is composed of many modules, including YARN (Yet Another Resource Negotiator) which is a resource-management platform.

Spark offers the option to run the applications on a YARN cluster manager, and in this case resource management, scheduling and security are controlled by YARN.

To run a Spark application YARN the first step is to configure the `HADOOP_CONF_DIR` enviroment variable to point to the directory containing the configuration files for the cluster, which are used to connect to HDFS.

Then it's possible to submit an application to YARN using the `--master yarn` flag of the `spark-submit` script.

3 Implementation

3.1 Chunking of the input FASTQ files

This part of the assignment asked to interleave 2 given FASTQ files and to split the result in multiple chunks before saving it in a compressed format. A FASTQ file consists of multiple DNA reads, where each read uses 4 lines.

3.1.1 Input format

To read this file format a custom Hadoop `FileInputFormat` named `FASTQInputFormat` was implemented. This format can be used together with the method `newAPIHadoopFile` to obtain a key-value RDD in which the values are the DNA reads, as `Strings` (the keys are not used and therefore set to null).

To enable parallel processing, the input files are splitted in multiple `InputSplits` so that each task can read his own split improving the efficiency of the program. This brings some technical challenges. Since the files are split without taking the lines into account, each split can include incomplete lines and thus reads.

We can be assured that each task outputs only valid reads if:

1. Each task starts reading from the first valid read in the split
2. Each task reads after the end boundary of the split to complete the last read if it's needed.

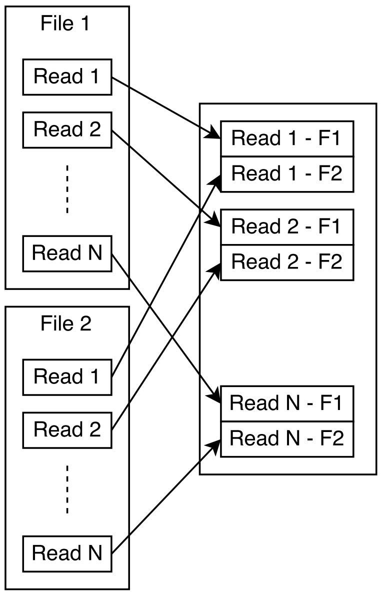

Figure 2: FASTQ interleaving

The rule 1 is implemented by the `positionAtFirstValidRead` method. This method gets called when the task starts reading the split and it reads lines from the input file until it finds a sequence of lines in which the first starts with `@` and the third starts with `+`, which are part of the FASTQ format. If the starting position corresponds to byte 0 of the file, we assume we're positioned on a valid read.

After positioning on the first valid read, the `nextKeyValue` method gets called to obtain the sequence of reads. The method checks first that the current position doesn't exceed the boundaries of the split and then reads 4 lines, so that rule 2 is respected.

3.1.2 Main program

In the main program the two input files are read into different RDDs using the custom input format. Then the first is `zipped` with the second to obtain an RDD of tuples of the respective elements of the two RDDs. The tuples are then concatenated to form the interleaved read, and the `repartition` method is used so that the resulting RDD has the same number of partitions as the number of tasks chosen with the command line arguments.

Afterwards the `foreachPartition` method is called on the resulting RDD to save each partition to a different compressed file. The output stream is compressed in the GZIP format by means of `GZIPOutputStream`.

3.2 DNA sequence analysis

Concerning the second part, the first step is to read the list of the FASTQ chunks from the input folder. Then the list is turned into a RDD by means of the `parallelize` method, and the `flatMap` method is used to invoke the BWA-MEM algorithm on each of the input chunks, with the `bwaRun` method.

This method runs the BWA-MEM algorithm passing as command line arguments the input file path and the reference file, and redirecting the output to a temporary file. The output gets read using the `BWAKeyValues` class and becomes the value returned by the method.

The `chrToSamRecord` RDD resulting from the `flatMap` is then cached since it gets used multiple times.

Afterwards, load balancing is performed. For this purpose, we first compute the SAM record count associated with each chromosome, sorted by descending number of SAM records. This is done by means of the `reduceByKey` and the `sortBy` methods, and the resulting list of (chrNumber, SAMrecordscount) elements is saved in the `chrToNumSamRecords` variable.

Next, we assign to each chromosome number a chromosome region. The number of chromosome regions is the same as the number of parallel tasks. We iterate over `chrToNumSamRecords` and each iteration we assign the current chromosome number to the region that has the least amount of records assigned, as illustrated by the figure on the side. The association between chromosome number and region is stored in the `loadBalancingMap` map. This gives a final record count for each region of {277662, 279898, 273353, 278221}.

The next step is to map the `chrToSamRecord` RDD so that the chromosome number is turned into the chromosome region, and to repartition it on the basis of the newly computed key to have the same number of partitions as the number of chromosome regions. The resulting balanced RDD is stored in the `chrToSamRecsBalanced` variable.

Then, variant call is operated on each partition, by means of the `mapPartitionsWithIndex` method. For each region the SAM records are sorted using a custom ordering function, then the `variantCall` method is called passing the sorted records and the region number, which happens to be the same as the partition index.

The variant calling method executes all the Java commands part of the pipeline, storing the results in temporary files. The execution of the commands is logged in a separate file for each region. The final output of the variant calling pipeline is the VCF file, which gets read and transformed into the return value of the `variantCall` method. All the temporary files used by the pipeline are deleted as soon as they are not needed anymore.

The combined results are sorted by the chromosome number and the position, and they are written to the output VCF file.

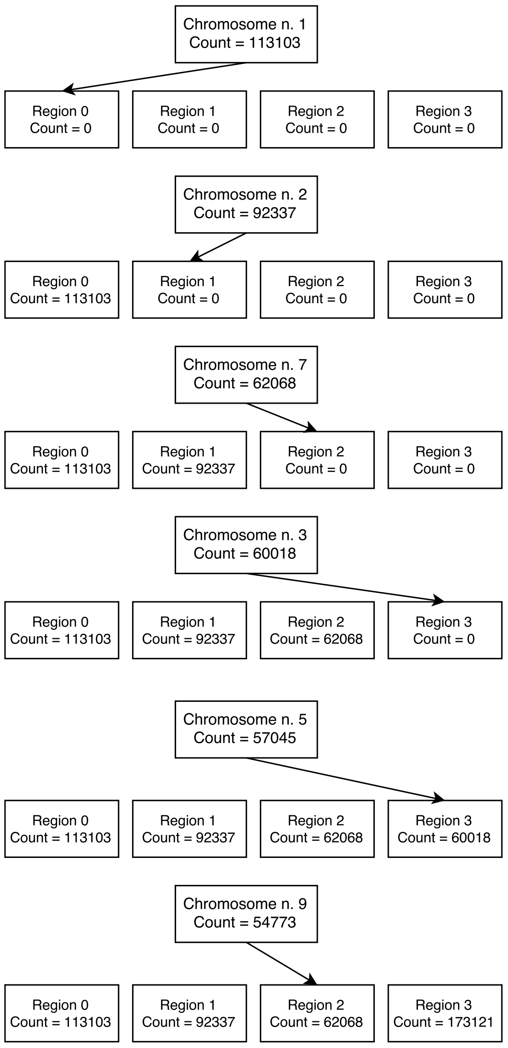

Figure 3: Load balancing

3.3 Porting the solution to a cluster

The third part asked to interact with HDFS, writing a custom library. The custom library consists in the `HDFSHelper.scala` file.

Here the `FileSystem.get()` method is used to get the resource corresponding to the HDFS. Then the methods implemented are simply wrappers around the methods offered by the HDFS library.

4 Results

The programs were ran on the Kova machine, which is an IBM POWER7 system.

4.1 Chunking of the input FASTQ files

The FASTQ chunking exercise was run with 2GB of RAM, giving the following execution time

```
$ ./run.sh 8 /data/spark/fastq out/
Number of parallel tasks = number of chunks = 8
Input folder = /data/spark/fastq
Output folder = out/
[...]
|Execution time: 0 mins 14 secs|
```

4.2 DNA sequence analysis

The second part of the assignment was run with 32GB of RAM and 4 instances, giving the following output

```
$ ./run.py
spark-submit --jars lib/htsjdk-1.143.jar --class "DNASeqAnalyzer" --master local[*]\
 --driver-memory 32g target/scala-2.11/dnaseqanalyzer_2.11-1.0.jar
***** Configuration *****
refFolder:        /data/spark/ref/
toolsFolder:        /data/spark/tools/
tmpFolder:        /tmp/spark/lgasparini/
inputFolder:        /home/lgasparini/SBD_2016_Lab3/part1/out/
outputFolder:        output/
numInstances:        4
numThreads:        4
*************************
[...]
|Execution time: 12 mins 21 secs|
```

The resulting VCF file was compared using the Python script, which resulted in the following

```
$ ./compare.py ref.vcf ../output/result.vcf
total = 4031
matches = 4013
diff = 18
```

that is satisfactory since the instructions stated that the difference should be less than 35.

4.3 Porting the solution to a cluster

For the third part of the assignment the input files needed to be retrieved from the HDFS and the output was written there together with the logs that were continuously update during the execution.

This resulted in a execution time of 12 mins 42 secs, and the same output of the comparison script.

5 Conclusion

This assignment has given me a good insight into DNA processing pipelines using Apache Spark. Spark seems the ideal choice for highly parallel applications like this one, especially when it's necessary to deal with large amounts of data. HDFS, in fact, has been proven to be able to scale up to 200PB of storage.