Supercomputing for Big Data ET4310 (2016)

Assignment 3

Name (Student ID)

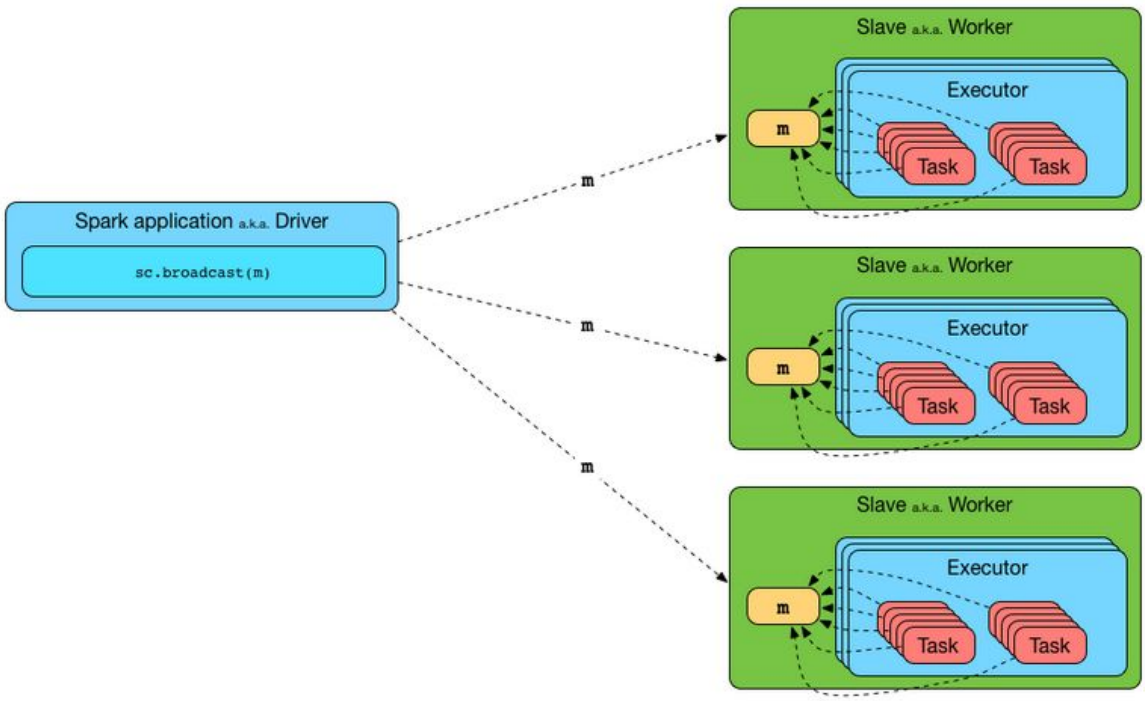
Introduction

Fast progress in next generation sequencing of DNA has resulted in the availability of large DNA datasets ready for analysis. However, DNA analysis has become the bottleneck in using these data sets, as it requires powerful and scalable tools to perform the needed analysis. A typical analysis pipeline consists of a number of steps, not all of which can readily scale on a distributed computing infrastructure. In this assignment, a framework that implements an in-memory distributed version of the GATK DNA analysis pipeline using Apache Spark, is created. To distribute the work parallely amongst multiple processes as uniformly as possible, a good load balancer is needed. An efficient algorithm is proposed in the implementation section.

This report is structured as follows: Introduction, Background, Implementation, Results, Conclusion and References.

Background

Broadcast variables



Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a

copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost. [1,2]

Spark actions are executed through a set of stages, separated by distributed "shuffle" operations. Spark automatically broadcasts the common data needed by tasks within each stage. The data broadcasted this way is cached in serialized form and deserialized before running each task. This means that explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important. [1]

Scala process package

This package handles the execution of external processes. The contents of this package can be divided in three groups, according to their responsibilities: [3]

- Indicating what to run and how to run it.
- Handling a process input and output.
- Running the process.

MapPartitions and ForeachPartitions

This function returns a new RDD by applying a function to each partition of this RDD. This function is more useful than the map function in certain case. For example, when we are initializing a database. If we are using `map` or `foreach`, the number of times we would need to initialize will be equal to the no of elements in RDD. Whereas if we use `mapPartitions`, the no of times we would need to initialize would be equal to number of Partitions. [4]

ForeachPartitions is used to perform side-effects given an iterator of records for each partition. This is useful to for example to write the content of each partition to files.

Implementation

Chunking the input Fastq files

The first step is to divide the two provided FASTQ input files into many smaller chunks that are used as input to the parallelized GATK Spark pipeline. Each DNA short read consists of 4 lines in the FASTQ files. These reads are interleaved from data of these two files. First, the lines are numbered according to the corresponding read sequence number. Second, these lines are grouped by the read sequence number to group the lines of a read as a record. This is done to both files. Both rdds have the same number of partitions and the same hash function. Zip function is used to create an interleaved rdd from the two original rdds. Finally, foreachPartition is used to write the reads of each partitions to different files.

DNA Sequence Analysis

The records are load balanced over the given number of partitions as follows: First, the number of records (size) per chromosome number gets computed and and sorted in a descending order with the size as key. Second, this array gets iterated, and each element is assigned to a partition with the least number of records i.e. the chromosome numbers are assigned in descending order to the partition with the least number of records.

Porting the solution to a cluster

The scala object HDFSInterface is responsible for communicating with and updating the hdfs file system. These are the available methods:

- Copy the given local file $src to hdfs file $des
  - def copyFromLocalFile(src: String, des: String)
- Copy the given hdfs file $src to local file $des
  - def copyToLocalFile(src: String, des: String)
- Add the given line to the hdfs file $path
  - def addLine(path: String, line: String)
- Create a directory in hdfs file system
  - def mkdirs(hdfsPath: String)
- return a list of file names in the given hdfs path
  - def getFiles(hdfsPath: String)
- Create a new hdfs file, if it already exits it gets emptied
  - def createNewFile(hdfsPath: String)

Results

I succeed in producing a correct output. Using the given script, to compare my output with the sample output, only 18 differences occurred, which is within the acceptable margin.

I ran my code on the Kova machine using four parallel tasks. The load balancer was the most important part of the code regarding performance of the code. Without the load balancer i.e. just using the hash partitioner on the chromosome number of each record, the execution time was 26 minutes. While with the above mentioned load balancer the execution time was 24 minutes.

Conclusion

This was an interesting assignment. It demonstrated that spark can be used in different domains. After this assignment, I have a better understanding of how sparks' load balancer works and how we can create a custom load balancer according the problem.

References

1: http://spark.apache.org/docs/latest/programming-guide.html#broadcast-variables

2: https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-broadcast.html

3: http://www.scala-lang.org/api/rc2/scala/sys/process/package.html

4: http://apachesparkbook.blogspot.nl/2015/11/mappartition-example.html