# Performance Comparison of the Sequential and the MapReduce K-Means implementations

Dhoha Abid
Computing Department
Qatar University
Email: da1304123@qu.edu.qa

***Abstract*—Nowadays, clustering algorithms have been widely applied in many fields. K-means is one of the mostly used clustering algorithm that has been implemented in both the industrial and the research domains. However, the emergent big data makes applying the K-Means clustering algorithm an expensive task. So, in order to process such big amount of data, a distributed version of the K-Means algorithm is highly recommended. In this regard, this project aims to implement the distributed K-Means (DKMeans) algorithm using the Hadoop MapReduce framework.**

**Keywords.** MapReduce, Hadoop, *K*-means, Distributed clustering.

## I. Introduction

The K-Means algorithm is a popular data-clustering algorithm that has been applied in many fields such as biology, climate, medicine, and wireless sensor networks. It is a simple algorithm that clusters the data into sub-sets of $K$ clusters. However, the K-Means clustering algorithm becomes an expensive and challenging task when it processes a huge volume of data. So, in order to deal with such big data, the distributed implementation of the K-Means algorithm is highly required. In fact, it has been proven that such distributed algorithm is very efficient when dealing with big data [11].

In this regard, this project aims to implement a distributed version of the K-means clustering algorithm based on the MapReduce framework [5]. Moreover, this implementation is performed on the cloud using the Amazon Web Services [3].

The rest of this paper is divided as follows. The second section presents the sequential K-Means algorithm and the wireless sensor network application scenario . The third section presents the distributed implementation of K-Means algorithms using MapReduce. This section gives an overview of the MapReduce programming model and details the implementation of the DKMeans algorithm. The forth section suggests the canopy algorithm in order to improve the DKMeans. The fifth section introduces the DKMeans evaluation. Finally the sixth details the project progress and the ongoing work.

## II. The sequential K-means clustering algorithm

### A. K-Means overview

The K-Means clustering algorithm groups the data into predefined number of $k$ clusters. The similarity is high between elements within the same cluster (inter-clusters). Therefore, members of a given cluster $i$ exhibit a high level of dissimilarity with members of different cluster $j$.

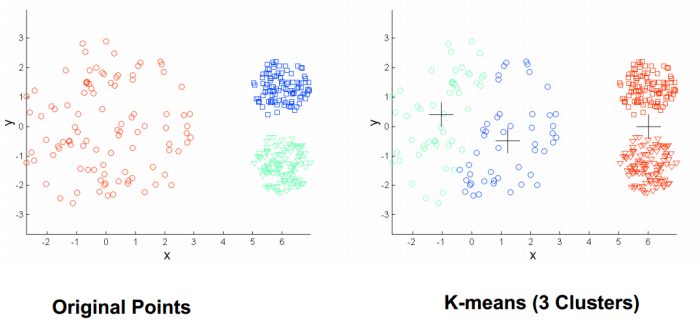



Fig. 1. Clustering 2D elements into 3 clusters [8].

The K-Means clustering algorithm takes as input (1) $k$ which is the number of clusters, and (2) $n$ elements to be clustered. The output is $n$ elements grouped into $k$ clusters. Cluster similarity is measured with regards to the average between the elements and the cluster centers[8].

First, the algorithm selects randomly $K$ from the the data-set ($K < n$). These $K$ elements are the cluster centers for the first iteration. Then, the distance between each cluster center and the data-set elements is calculated. Each element, therefore, selects the cluster that has the smallest distance to it. After clustering all the elements the new clusters are calculated by measuring the average within each cluster. Second, the algorithm performs many iterations that recalculates the cluster centers. The algorithm will stop in two cases: (1) when no changes are performed on the cluster centers or (2) when the number of iteration reaches the predefined maximum number of iterations. The pseudo-code of the sequential K-means clustering algorithm is presented in Algorithm 1.

Figre 1 presents the $n$ elements before and after the operations of clustering. In this example $K = 3$.

### B. Application scenario: Wireless sensor networks

A wireless sensor networks (WSN) consists of a set of sensor nodes spread spatially. WSN are used for many applications such as monitoring the environment. However, these sensors have generally small sizes, thus small batteries. So, an energy efficient communication is needed in order to save power. Researchers have proposed clustering as an effecient solution that reduces the power consumption by transmitting aggregated data through the cluster centers to the sink node[10].

In this context, the K-Means clustering algorithm can be applied to cluster the sensor nodes. However, the number of

**Algorithm 1** K-Means algorithm

**Require:** $iter = 0$;
**Require:** $Maxiter = 500$;
Select randomly $K$ cluster centers from the $n$ elements of the data-set;
**repeat**
  **for all** $n$ in the data-set **do**
    **for all** $K$ cluster centers **do**
      $dis = ComputeDis$(value of $n$ ,value of $K$ );
      **if** $dis < minDis$ **then**
        $minDis = dis$;
        $cCenter$ =value of $K$ ;
      **end if**
    **end for**
    $cCenter$ is the cluster of $n^th$ value ;
  **end for**
  $iter++$
**until** No changes **or** $iter > Maxiter$

sensors is generally very large which slows the computations. Moreover, it makes the time complexity of this algorithm high. The time complexity of this algorithm is equal to O( n * K * I * d ), where n = number of elements, K = number of clusters, I = number of iterations, d = the euclidean distance time computation[10].

As a solution to speedup the computation, the following section presents the implementation of the distributed K-Means algorithm using Hadoop MapReduce.

## III. Distributed K-Means algorithm using MapReduce

We choose to implement the distributed K-Means algorithm (DKMeans) using the Java programming language. We use the Hadoop MapReduce framework [2]. This section is divided as follow: the first subsection presents the MapRduce literature, and the second subsection details the DKMeans implementation.

### A. MapReduce literature

*1) MapReduce overview:* MapReduce is a programming model that offers the commodity of writing distributed algorithms across a distributed cluster of many computing nodes. This model was first introduced by Google in 2003 in order to simplify indexing web pages[5]. MapReduce is a very powerful programming model because programmers do not have to deal with the complex details of parallelizing the computations, distributing data, and scheduling the execution. Such tasks are hidden and automatically managed by the MapReduce framework[5], [4].

MapReduce is widely used. In fact, more than ten thousand distinct MapReduce programs have been implemented at Google. Moreover, an average of one hundred thousand MapReduce jobs are executed on Google's clusters every day[4].

*2) The MapReduce programming model:* The MapRduce programming model is expressed as a *map*, optionally *combine*, and *reduce*functions. The input and the output of all these functions are based on *key/value* pairs[4], [7].

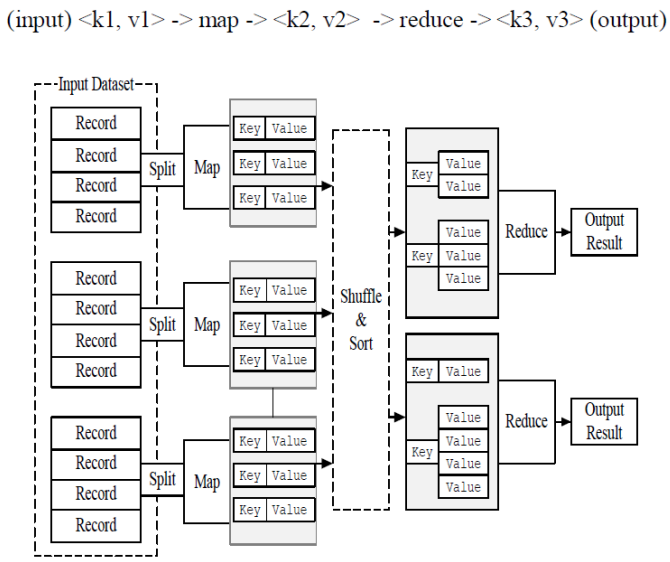



Fig. 2. MapReduce Model Abstraction [7].

SeqFile.txt

```
4, 1
8, 3
9, 4
8, 77
6, 2
8, 22
99, 1
2, 6
```

Fig. 3. Sensor Coordinates file.

Figure 2 presents the MapReduce abstraction model. First, the split divide the data-set into a set of mappers. In each mapper, the map function transforms independently the input splits into *key/value* pairs. Then, the shuffling and the sort phase will move the outputs to the reducers. In the reduce phase, the *key/value* pairs are aggregated according to the *reduce* function.

The architecture consists also of an optimal phase called combine phase. The combine phase aims at optimizing the network bandwidth. This latter is not presented in figure 2. It can be added just after the map phase and before the shuffling phase. The code of the combine function is generally similar to the *reduce* function.

### B. Distributed K-Means (DKMeans) implementation

We choose to implement the DKMeans algorithm on a large scale (WSN). We consider a WSN consisting of n static nodes. Nodes are placed in an area "A". The wireless sensor nodes are presented with coordinates of 2 dimensions *(x, y)*. The sensor nodes coordinates are stored in a sequential file where each line presents a single node coordinates. Figure 3 is a sample of the input data. The goal now is to define the *map*, *combine* and *reduce* functions [11].

*1) Map function:* **Input:** ($Key$ = the name of the sequential file containing all the sensor coordinates as shown in

figure3. $Value =$ corresponds to the value of these coordinates in this file). **Output:** a list of ($key =$ cluster index. $Value =$ string containing the coordinates of the sensor nodes belonging to this cluster).

The *map* function reads the cluster centers value from a sequential file stored on the MapReduce HDFS file system [2]. These cluster centers are placed into $centers$ array of k length. The euclidean distance between each sensor node $j$ and the cluster center $center[i]$ is calculated. The sensor node will be assigned to the cluster center that will have the minimum euclidean distance to it. Algorithm 2 presents a pseudo-code of the *map* function.

**Algorithm 2** map(key, value)

```
Construct the sample instance from value;
minDis = Double.MAX − VALUE;
index = −1;
for i = 0 to centers.length do
    dis = ComputeDis(instance, center[i]);
    if dis < minDis then
        minDis = dis;
        index = i;
    end if
end for
key′ = index;
Construct value′ as a string containing the sensor node coordinates;
return < key′, value′ > pair;
```

*2) Combine function:* **Input:** the output of the *map* function.
**Output:** a list of ($key =$ cluster index. $Value =$ a string composed of the number of nodes belonging to this cluster, and the sum of all their euclidean distances). the *combine* function performs partial sums at the mapper level in order to send an aggregated results to the reducer. This eliminate overhead at the network level. Algorithm 3 presents a pseudo-code of the *combine* function.

**Algorithm 3** combine(key, value)

```
Initialize one array V to record the number of nodes belonging to the same cluster, and the sum of their euclidean;
num = 0;
while key = V[clusterIndex] do
    Construct the sample of instance from Value;
    Add the the euclidean distance of instance to the array V;
    num++;
end while
key′ = key;
Construct value′ as a string comprised of sum values of different dimensions and num;
return < key′, value′ > pair;
```

*3) Reduce function:* **Input:** the output of the *combine* function. **Output:** a list of ($key =$ cluster index. $Value =$ the value of the new cluster index.

The *reduce* function calculates the new cluster center values. To do so, it calculates the average of all the partial sums belonging to the same cluster. The output of the *reduce* function is recorded to the the HDFS file system used in the first iteration to read the first cluster centers. This is in order to be visible to all the mappers in the next iteration. Algorithm 4 presents the pseudo-code of the *reduce* function.

**Algorithm 4** reduce(key, value)

```
Initialize one array V to record the number of nodes belonging to the same cluster, and the sum of their euclidean;
num = 0;
while key = V[clusterIndex] do
    Construct the sample of instance from Value;
    Add the the euclidean distance of instance to the array;
    num++;
end while
Calculate the average within each cluster
key′ = key;
Construct value′ as a string comprised of the new cluster center coordinates;
return < key′, value′ > pair;
```

*4) Putting all together:* The presented functions (*map*, *combine*, and *reduce*) are executed in just one job. As K-Means is an iterative algorithm, this single job has to be executed many times. So, the output of the reducer presenting the new cluster centers is always recorded into HDFS file. Then in the next iteration, the mapper will read the cluster centers from this HDFS file.

## IV. DKMeans enhancement: Canopy algorithm

The DKMeans algorithm is very sensitive to the first choice of $K$ random cluster centers. In fact, a bad selection of the first random cluster centers may affect the performance of the algorithm. In this case, we can apply the canopy algorithm before performing the DKMeans algorithm. In fact, the canopy algorithm can be used as a preprocessing step for the K-Means algorithm to define the first cluster centers rather than selecting them randomly [9].

## V. Implementation

The implementation of the algorithm can be divided into 4 parts: (1) Coordinates generator, (2) The sequential K-Means implementation, (3) The MapReduce implementation, and (4) The Clusters separator.

### A. Coordinates generator

Coordinates generator is a script written in the C language in order to generate the input file containing the sensor coordinates. the coordinates number and the file name have to be specified for this script. Then, after running this script, a new file is created containing the specified number of coordinates each line in the format x,y. This file serves as input for both the sequential and the MapReduce algorithms.The figure 3 presents a sample of this file.

### B. Sequential implementation

The sequential implementation is written in the Java language as detailed in the second section of this report. The input is the sensor coordinates file that has been generated by the *Coordinates generator* script. However, The output is a file containing the final centroids. The number of centoids $K$ is a parameter has to be set in the code. A bunch of classes were needed and this is the list of the most important ones:

- **Coordinate**: every sensor coordinate is expressed in the code as a coordinate.
- **EuclideanDistance**: is the method that calculates the euclidean distance of two sensor coordinates.
- **CalculClusterIndex**: is the method that calculate the cluster index that has been used for the calculation of the centroids.

### C. MapReduce implementation

It is written in Java language where the Map, Reduce, and Combine functions are implemented as it has been detailed in the third section of this report. This is the most important part of the code as it is the distributed implementation. Moreover, implementing this part consolidated what has been understood in the third section of this report.

The implementation of this part was typically as detailed in the 3rd section in this report. However, we added a small function inside the mapper called *setup*. This function is called once at the beginning of the each job execution just before starting the map function. This function has been used in order to read the HDFS file containing the initial centroids. Reading the initial centroids at the level of this function has a very good performance impact. In fact, it reduces the execution time as this HDFS file will be read just once for all the map functions in every job.

### D. Cluster separator

It is written in the Java language as well. It takes as input: (1) the file generated by the *Coordinates generator* script, and (2) the new centroids generated by either the sequential or the distributed implementation. As output it will give $K$ files, and each file contains the coordinates of each cluster separately. This step is performed in order to (1) be sure about the result of the sequential and distributed implementation, and (2) to have the possibility to plot the final results after the clustering operation.

### E. Verification of the accuracy of the implemented algorithms

After the implementation, we used the cluster separator in order to be sure that both algorithms are functioning right. So, I plotted the sensor coordinates before and after clustering. Figure 4 is a plot of the generated coordinates before and after clustering.

## VI. Results and evaluation

The evaluation of the MapReduce algorithm is divided into 2 parts. The first part compares the performance of the MapReduce execution with the sequential execution. However, the second part evaluates the MapReduce implementation with regards to the Speedup, Scalability, and Fault-tolerance.

TABLE I. Laptop characteristics

| Characteristics | Power |
|---|---|
| Processor | Intel(R) Core(TM)i7-2640M CPU@2.8GHz |
| cores | 8 |
| Installed RAM | 8.00 GB |
| System type | 64-bit Operating System |

TABLE II. A single node characteristics

| Characteristics | Power |
|---|---|
| API Name | c1.xlarge |
| Processor/ Compute units (CU) | 20 (8 core x 2.5 unit) |
| Installed RAM | 7.0 GB |
| System type | 64-bit Operating System |
| I/O Performance | High / 1000 Mbps |

### A. Evaluation of the Sequential implementation vs. the MapReduce implementation

In order to compare both the sequential and the distributed implementation, we executed the both implementations each experiment with the same input file, and the same initial first centroids. This is in order to have the same number of iterations in both the sequential and distributed execution. Moreover, we contrasted their execution time while varying the input file size and the number of nodes in the distributed execution. The input file size vary from 500M to 8GB, and the number of nodes allocated for this experiments varies from 2 to 16 nodes.

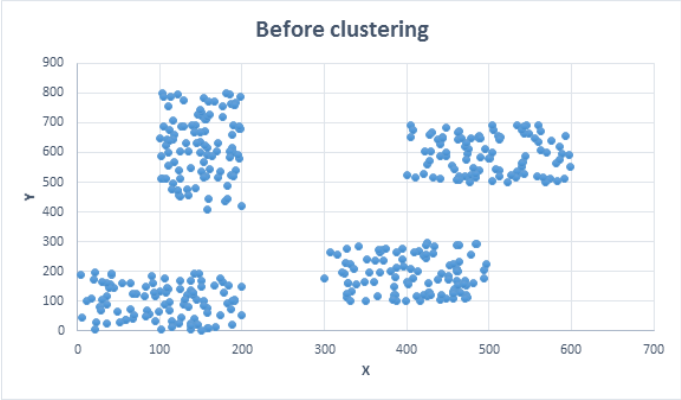
*1) The sequential settings:* To run the sequential K-Means algorithm, we used a laptop with medium power. The table I lists its main characteristics[6].

*2) The distributed settings:* The cloud computing platform is richer. In fact, we have the possibility to choose from a wide range of computing power [1]. This range varies from small nodes having 0.6 GB as RAM and 2 compute units (CU) till nodes having 244 GB as RAM and 104 CU. A compute unit (CU) is equivalent to CPU capacity of a 1.0-1.2 Ghz 2007 Opteron or 2007 Xeon processor. We tried to choose nodes that are not far away from the laptop configuration. this is in order to have valid comparison between the sequential and the distributed execution. Moreover, we tried also to choose nodes that have good I/O performance. This is because we noticed that if the I/O performance is low, the performance of the MapReduce execution is considerably affected even if the nodes are powerful. Table II lists the main characteristics of a single node that has been used in the next experiments.

*3) Comparison between the sequential and the MapReduce execution:* Figure 5 plots the sequential and the distributed execution time while varying the input file size. The plots interpretation can be divided into 3 parts where the distributed execution is performed by:

*a) **2 nodes:** The sequential execution time is considerably less than the MapReduce execution time for all the input file sizes. This is because of the overhead that has been issued during the MapReduce execution. In fact, the overhead is triggered by (1) Reading the HDFS file at the beginning of each job, (2) the input file partition into blocks to feed the mappers, (3) Writing to the HDFS file in the reducer phase, and (4) the iterative jobs: initializing each job takes time and trigger overhead. However, this overhead does not exist in the sequential execution as (1)there is no coordination between mapper, and (2) the centroids are read directly from*


(a) Before clustering.




(b) After clustering

Fig. 4. Clustering 2D point into 4 clusters using K-Means and DKMeans

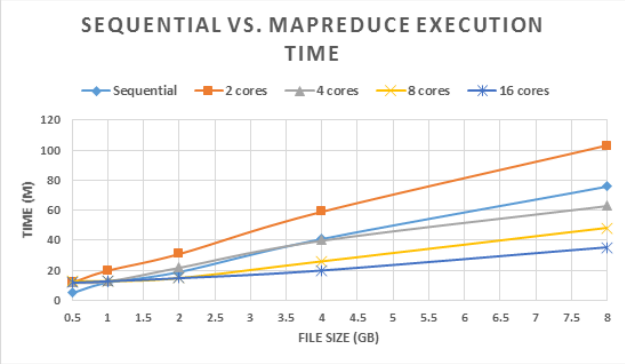



Fig. 5. Comparison between the execution time of the sequential and MapReduce implementation

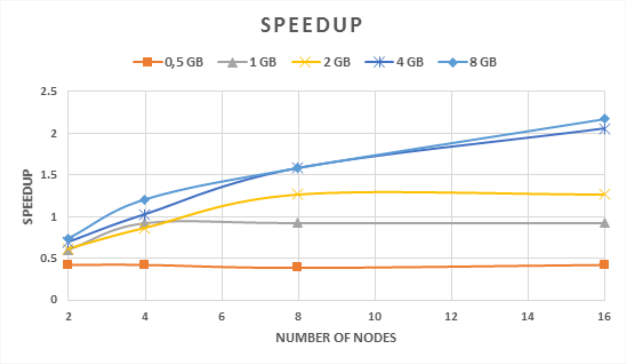



Fig. 6. SpeedUp

*the memory. So, as a conclusion, there is need for more number of nodes that can cooperate while the computation, so, the overhead will be insignificant:*

*b)* ***4 nodes:*** *The sequential execution time is less than the MapReduce execution time when the input file size is less than 4 GB. However, when the file size is more than 4 GB, the MapReduce execution outperforms the sequential execution. In fact, as the file gets bigger and bigger, the MapReduce execution outperforms more and more the sequential execution. This is show that the MapReduce implementation is more efficient than the sequential implementation when it comes to bigger data.:*

*c)* ***8 nodes and above:*** *the MapReduce outperforms considerably the sequential execution. In fact, the MapReduce execution starts slightly to outperform the sequential execution when the input file size is 2 GBe. Then, as much as the input file size gets bigger and bigger, the MapReduce execution will outperform increasingly the sequential execution. This consolidates the fact that MapReduce is more efficient when processing big data. :*

### *B. The distributed K-Means evaluation*

The performance evaluation of the distributed K-Means algorithm consists of three main experiments: Speedup, Scalability, and Fault-tolerance.

*1) Speedup:* The speedup measures how much the distributed algorithm is faster than the sequential algorithm. The speedup is defined by the following formula: Speedup = Sequential execution time/parallel execution time. In this evaluation we measured the speedup while varying the number of nodes for different file sizes. The figure 6 presents the speedup of the MapReduce execution.

When the input file size is less than 2GB, we notice, at a certain level, that increasing the number of nodes will not speedup the execution. This is because the files' sizes are small, and we allocated more than the needed resources. However, when it comes to bigger files more than 4 GB, the speedup increases as much as the number of nodes increases. This because the files are bigger, and increasing the nodes number will involve more mapper that will help in reducing the execution time.

Normally, a perfect parallel algorithm demonstrates linear speedup. However, linear speedup is difficult to achieve because the communication cost increases when the number of nodes become larger. This is because larger nodes number will trigger more overhead.

*2) Scalability:* Scalability is the ability to process a growing amount of data by adding more hardware commodity. In this experiment, we kept the input file size equal to 8GB while we increase the number of the nodes from 2 to 16. The figure

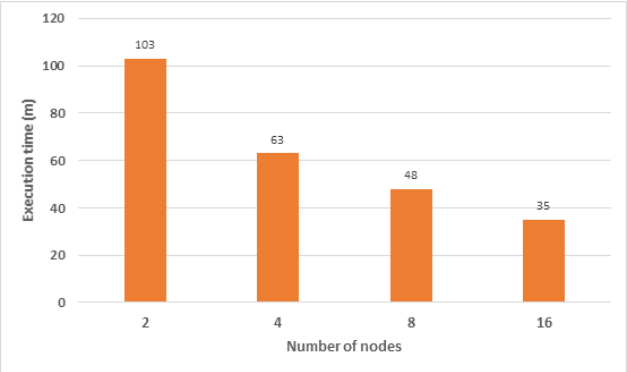



Fig. 7. Scalability by fixing the input file size and varying the number of nodes

7 presents this evaluation scenario.

It is obvious when we increase the number of nodes, the execution time decreases. In fact, increasing the number of nodes from 2 to 4, decreases the execution time by about 40% which is an important percentage. However, increasing the number of nodes from 4 to 8 or from 8 to 16, decreases the execution time by about only 25%. So, increasing continuously the number of the nodes does not decrease significantly the execution time. This is again because of the overhead that has been created by the involved computational nodes. In fact, more involved nodes triggers more overhead in term of nodes coordination and scheduling. Moreover, the overhead can be explained by the fact that the K-Means is an iterative algorithm that needs to run many jobs following each other. So, the fact of running many jobs where each job has to read and write from to the HDFS file creates a lot of overhead.

*3) Fault-tolerance:* The fault-tolerance has been tested through AWS EC2 nodes. This is by terminating one of the executing nodes while running the distributed K-Means algorithm. We noticed that upon this sudden termination, the cluster re-sized itself and continued the computation without stopping. Moreover, the computation results were accurate. This ensures that the MapReduce framework is fault-tolerant.

## VII. Conclusion

In this project, we went first through an understanding of the K-Means algorithm. Then implementing it in the sequential fashion. Then, we went through the Hadoop MapReduce framework where we did understand its concept. To apply the K-Means into the MapReduce programming language, we did some changes in the sequential algorithm to get the distributed MapReduce algorithm. Then, we did implement the distributed K-Means in the Java language. We executed this code on the Amazon Web Service platform.

In order to compare the sequential execution with the distributed execution, we used different input file sizes raging from 500M to 8GB. The experimental results show that definitively the MapReduce execution outperforms the sequential execution when it uses an important input file size, and involves many nodes in the execution. However, when it comes to small input files, the sequential execution outperforms the distributed execution. This is absolutely legitimate because the overhead in this case is more than the computations.

## Acknowledgment

I want to give a special thanks to Dr.Tamer Elsayed who helps a lot to cope many difficulties in this project. I want to thank Dr. Aiman Erbad who gave me the opportunity to work on AWS.

## References

[1] Ec2 nodes configuration. http://www.ec2instances.info/.

[2] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11:21, 2007.

[3] Amazon Elastic Compute Cloud. Amazon web services. *Retrieved November*, 9:2011, 2011.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[6] Paul Doe. Intel core i7-2640m processor specification. http://ark.intel.com/products/53464/Intel-Core-i7-2640M-Processor-4M-Cache-up-to-3_50-GHz.

[7] Suhel Hammoud. Mapreduce network enabled algorithms for classification based on association rules. 2011.

[8] Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining: concepts and techniques*. Morgan kaufmann, 2006.

[9] Amresh Kumar, Saikat Mukherjee, et al. Verification and validation of mapreduce program model for parallel k-means algorithm on hadoop cluster. *International Journal of Computer Applications*, 72, 2013.

[10] P Sasikumar and Sibaram Khara. K-means clustering in wireless sensor networks. In *Computational Intelligence and Communication Networks (CICN), 2012 Fourth International Conference on*, pages 140–144. IEEE, 2012.

[11] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on mapreduce. In *Cloud Computing*, pages 674–679. Springer, 2009.

## Appendix

This report gives a brief explanation of the *Map* and *Reduce* functions that have been implemented for the distributed KMeans Algorithm. Then, it presents a tutorial showing the execution of the the jar file of this distributed implementation.

### A. Map function

As have been described in the report, the *Map* function has to read the initial centroids from the HDFS file. So, reading the HDFS file has to be performed at the level of the setup function in the Mapper class.Figure8 shows the implementation of the setup function in the map function.

Reading the initial centroids at the setup function allows the HDFS file to be read just **once** at the beginning of the job. In fact, if reading the file is performed at the level of the *Map* function not in the setup function, the performance of the algorithm will dropped significantly. This is because the HDFS file will be read in every Map task.

Figure 9 show the remaining of the implementation of the *Map* function. In this function we calculate the distance between every point and the centroids. Then emitting the centroids index and the point coordinates to the reducer.

```
public class Map extends Mapper<LongWritable, Text, IntWritable, Text> {
                                Input type      Output type
    //Declaration and Initialization of the initial clusters
    public static Coordinates[] centroid = new Coordinates[Constant.NClusters];
    @Override
      protected void setup(Context context) throws IOException, InterruptedException {
        super.setup(context);
                Setup function to read the InitialCentroids.txt file at the begining of the jo
            //Read the centroids from the HDFS file
            FileSystem fs = FileSystem.get(new Configuration());
            Path path = new Path("/user/hadoop/InitialCentroids.txt");
            if (!fs.exists(path)) {
                System.out.println("The InitialCentroids.txt file does not exist.");
                return;
            }

            //Variable to read on it the line content
            String sCurrentLine;

            //Read the HDFS file containing the centroids
            BufferedReader br = new BufferedReader(new InputStreamReader(fs.open(path)));
            int i=0;
            while (i<Constant.NClusters && (sCurrentLine = br.readLine().trim()) != null) {
                centroid[i] = new Coordinates(sCurrentLine);
                i++;
            }
            //Close the File
            if (br != null)
                br.close();

    }
```

Fig. 8. Implementation of the setup function inside the Map function.

```
    public void map(LongWritable key, Text value, Context context) throws IOException{

        //Read the coordinates sensor point, and store the value on the Coordinates variable
        Coordinates point = new Coordinates(value.toString());

        //distance is a variable issued from the Distance class to store
        //the calculated distances between var and each centroid
        Distance distance = new Distance();
        //System.out.println("after creating distance");
        for(int p=0; p<Constant.NClusters; p++){
            distance.dist[p]= SequentialKMeans.EuclideanDistance(centroid[p], point);
            //System.out.println(distance.dist[p]);
        }

        //Calculate the nearest centroid for the point, and store its index in the index variable
        int index=distance.CalculClusterIndex();

        //Preparing the emission of the results to the reducer
        String toSend = new String();
        toSend=value.toString() + ",1";

        System.out.println("key: " + index + "    value:" + toSend);
        //Send the results to the Reducer
        //The result would have the form key=clusterindex value= sensor coordinates in a string form
        try {
            context.write(new IntWritable(index), new Text(toSend));
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

Fig. 9. Implementation of the Map function.

## B. Reduce function

Although in the head of the Reducer, we had to specify the output type of the Reduce function, this algorithm does not emit any output. However, it has to write the results into the HDFS file in order to be used in the next iteration. Moreover, the results of this function are the new centroids that serves as input for the next job iteration. Figure 10 shows the implementation of the reduce function.

In this step, we upload all the necessary files into the Amazon S3 buckets. So, the necessary files are: InitialCentroids.txt, NewCentroids.txt, KMeansMapReduce.jar, and the Sensorcoordinates.txt. In fact, by using the Amazon S3 service:

1) We create an S3 bucket named *aws.test.dhoha*
2) We create 4 folders named data, log, job, and CentroidsInitialization.
3) the jar will be uploaded into the job folder.
4) The SensorCoordinates.txt file is uploaded into the data folder.

```
public class Reduce extends Reducer< IntWritable, Text, IntWritable, Text> {
                                    Input type      Output type
    public void reduce(IntWritable key, Iterable<Text> values, Context context)throws IOException, InterruptedException {
        //ClustersCoordinatesSums sums = new ClustersCoordinatesSums();

        double sumX=0.0, sumY=0.0;
        int sumF=0;
        Coordinates centroid = new Coordinates(0,0);

        //Read the coordinates point belonging to one cluster
        for(Text val: values){
            String currentLine = val.toString();
           CoordinatesSums Partialsum = new CoordinatesSums(currentLine);
           sumX+=Partialsum.X;
           sumY+=Partialsum.Y;
           sumF+=Partialsum.F;
        }

        centroid.x = (int) (sumX/sumF);
        centroid.y = (int) (sumY/sumF);
                                Calculate the new centroids

        //Write again the new centroid to the HDFS file
        FileSystem fs = FileSystem.get(new Configuration());
        Path path = new Path("/user/hadoop/NewCentroids.txt");  Path of the HDFS file of the NewCentroids. txt file

        BufferedWriter meansWriter = new BufferedWriter(new OutputStreamWriter(fs.append(path)));

        meansWriter.write(centroid.x + "," + centroid.y+"\n");   Write the output to the HDFS file
        System.out.println(centroid.x + "," + centroid.y+"\n");
        System.out.println("\n");
        meansWriter.close();

    }
}
```

Fig. 10. Implementation of the Reduce function.

5) Both InitialCentroids.txt and NewCentroids.txt are uploaded into the CentroidsInitialization folder.

Figure 11 shows how to upload the files.

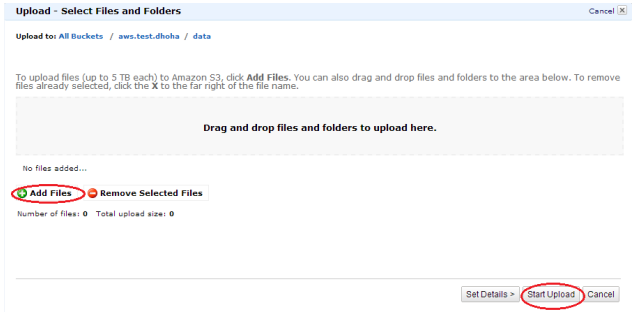



Fig. 11. Uploading the files into the Amazon S3 bucket.

## C. Create cluster

The cluster is create by using the *Elastic MapReduce* service of the AWS. So, in the *Elastic MapReduce* service on AWS, we click on the create cluster button. Figure 12 is the first step to a create the Hadoop MapReduce cluster.

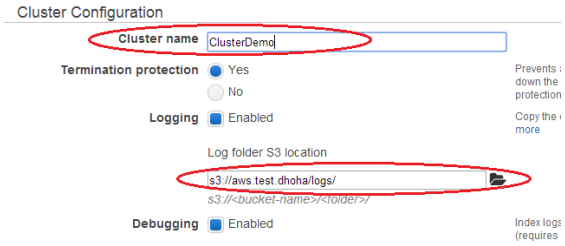



Fig. 12. Create a Hadoop MapReduce cluster

## D. Cluster configuration

After clicking on the *Create cluster* button, we will be directed to the cluster configuration page. In this page we have to set:

- The cluster name.
- The log file path.
- The cores type and number.
- The key pair.

In order to create a key pair, consult this web page http://docs.aws.amazon.com/gettingstarted/latest/wah/getting-started-create-key-pair.html. All the details are there for the creation of the key pair.

Then after specifying all the settings, we have to click the *create cluster* button. Figure 12 and 13 shows where to set the configuration.

Notice that specifying the number of cores is very important. In fact, cores in this concept means the number of nodes that can be involved in the computation. We have also to specify the private key. This is in order to be able to ssh the master node later on. To ssh the master nodes is very easy, the following page has all the details about accessing the Amazon instances by putty: https://linuxacademy.com/blog/linux/connect-to-amazon-ec2-using-putty-private-key-on-windows/. Figure 13 shows how to setup the cores and the private key.

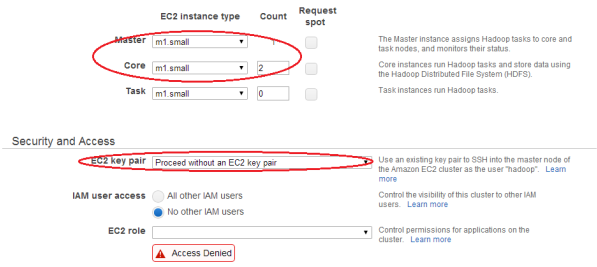



Fig. 13. Cluster configuration

After that click on the created cluster button, the cluster now is provisioning the cores in order to be ready.

Now, we have to transfer the InitialCentroids.txt and New-Centroids.txt file from the Amazon S3 to the HDFS platform. In order to do so, we have to access the master instance by putty and by using the already created private key. The same private key has to be specified in both cluster creation, and master node access. These are the command that should be used:

1) hadoop distcp s3://aws.test.dhoha/CentroidsInitialization/InitialCentroids.txt InitialCentroids.txt
2) hadoop distcp s3://aws.test.dhoha/CentroidsInitialization/NewCentroids.txt NewCentroids.txt

The figure 14 shows the command that has been used to transfer the initial centroids file from the S3 bucket to the HDFS platform.

After uploading all the necessary files, we can now execute the implemented distributed K-Means algorithm. We have to go back to cluster already created where we have to add a step to execute the Jar file. So, once back to the cluster click on

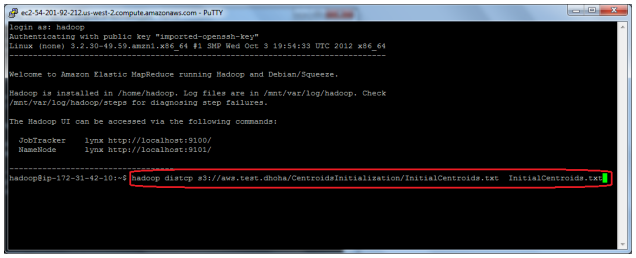



Fig. 14. Uploading the HDFS files.

step. Figure 15 shows how to create step. In fact, in this step we have to specify:

- The step name (it does not have to be unique).
- Jar file location: s3n://aws.test.dhoha/job/KMeansMapReduce.jar
- The coordinates points input file: s3n://aws.test.dhoha/data/SensorCoordinates1G.txt

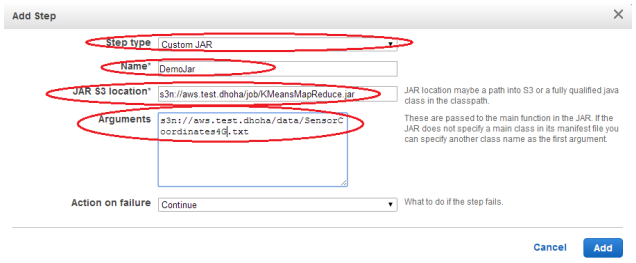



Fig. 15. Create step to execute the Jar file.

On Clicking on the *create step* button , a form is displayed showing the execution of the jar file. Figure 16 shows how to fill this form.

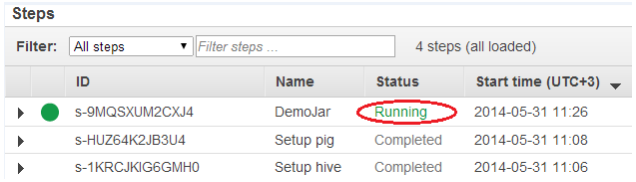



Fig. 16. Running the job.

After the job completes, the results will be available on the InitialCentroids.txt. We can access them by the following command: "hadoop fs -cat InitialCentroids.txt". Figure 17 shows the new centroids in the InitialCentroids.txt file after the execution of MApReduce K-Means algorithm.

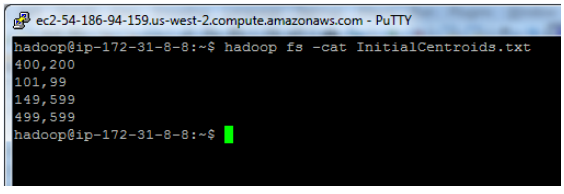



Fig. 17. The new centroids after execution of the MapReduce K-Means algorithm.