

Lab2: Parallel K-Means using Hadoop

Distributed Systems

Presented by

1- Mohamed Said 57

2- Ahmed el bawab 8

3- Khalil Ismail 23

3/11/2020

A very common task in data analysis is grouping a set of unlabeled data such that all elements within a group are more similar among them than they are to the others. This falls under the field of unsupervised learning. Unsupervised learning techniques are widely used in several practical applications, With the development of information technology, data volumes processed by many applications will routinely cross the peta-scale threshold, which would in turn increase the computational requirements. Efficient parallel clustering algorithms and implementation techniques are the key to meeting the scalability and performance requirements entailed in such scientific data analyses.

The Hadoop and the MapReduce programming model represents an easy framework to process large amounts of data where you can just implement the map and reduce functions and the underlying system will automatically parallelize the computations across large-scale clusters, handling machine failures, inter-machine communications, etc.

Table of Contents

<i>1- Problem Definition</i>	<i>3</i>
<i>2- Goals</i>	<i>4</i>
<i>3- Algorithms & Code description</i>	<i>5-7</i>
<i>4- Implementation</i>	<i>8-10</i>
<i>5- Sample Runs</i>	<i>11-12</i>
<i>6- Results & Conclusion</i>	<i>13</i>

1 Problem Definition

With the development of information technology, data volumes processed by many applications will routinely cross the peta-scale threshold, which would in turn increase the computational requirements. Efficient parallel clustering algorithms and implementation techniques are the key to meeting the scalability and performance requirements entailed in such scientific data analyses.

Big data has become popular for processing, storing and managing massive volumes of data. The clustering of datasets has become a challenging issue in the field of big data analytics.

The K-means algorithm is best suited for finding similarities between entities based on distance measures with small datasets. Existing clustering algorithms require scalable solutions to manage large datasets. There are two approaches to the clustering of large datasets using MapReduce.

The first approach, K-Means Hadoop MapReduce (KM-HMR), focuses on the MapReduce implementation of standard K-means.

And the other is the sequential K-Means which is the normal implementation of the Algorithm.

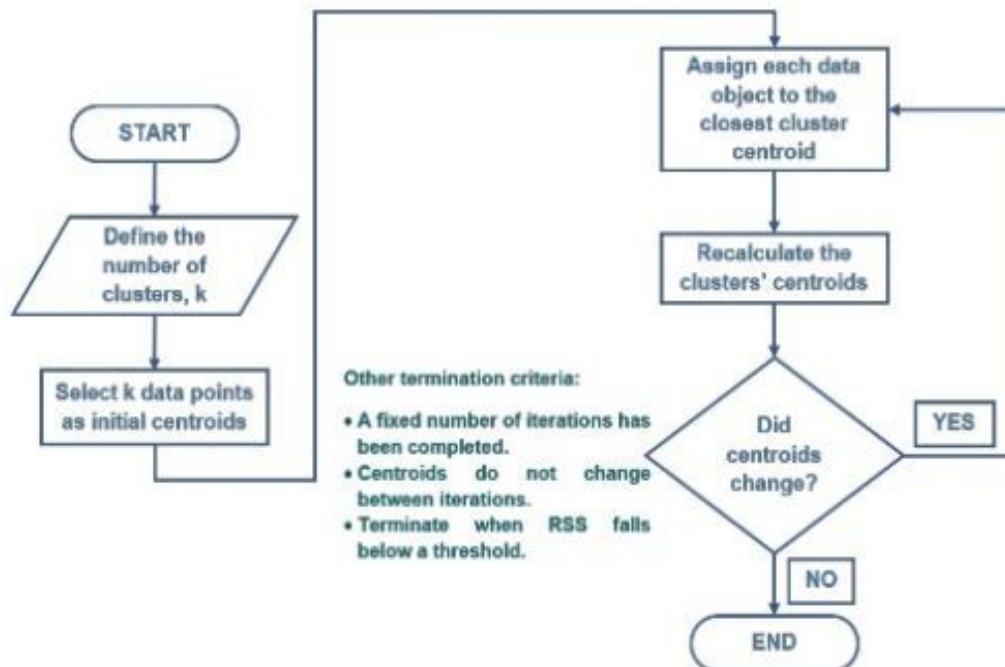
2 Goals

1. Implementing a parallel version of the K-Means algorithm using the MapReduce framework.
2. Implementing an unparallelized version of the K-Means algorithm using normal implementation.
3. Evaluating the parallel version and comparing it with the unparallelized version of K-Means using the IRIS dataset in terms of run time and clustering accuracy.

3 Algorithms & Code description

Here is a comparison between Unparalleled K-Means and Paralleled K-Means using Map-Reduce technique:

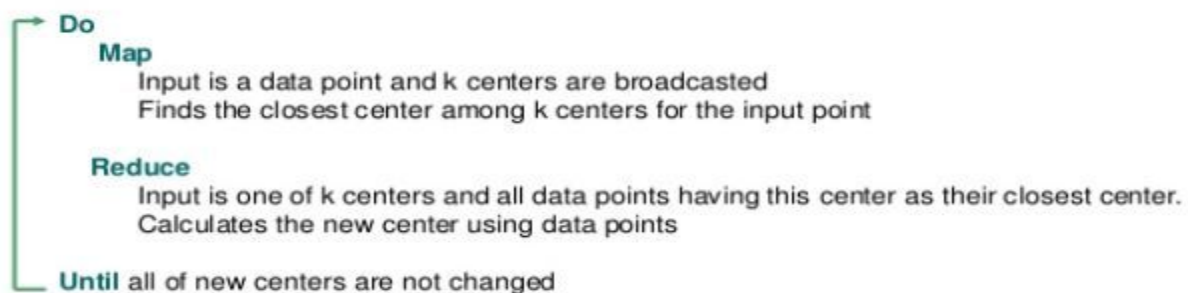
Unparalleled K-Means:



Paralleled K-Means:

We will make each iteration of the K-Means as a map-reduce phase.

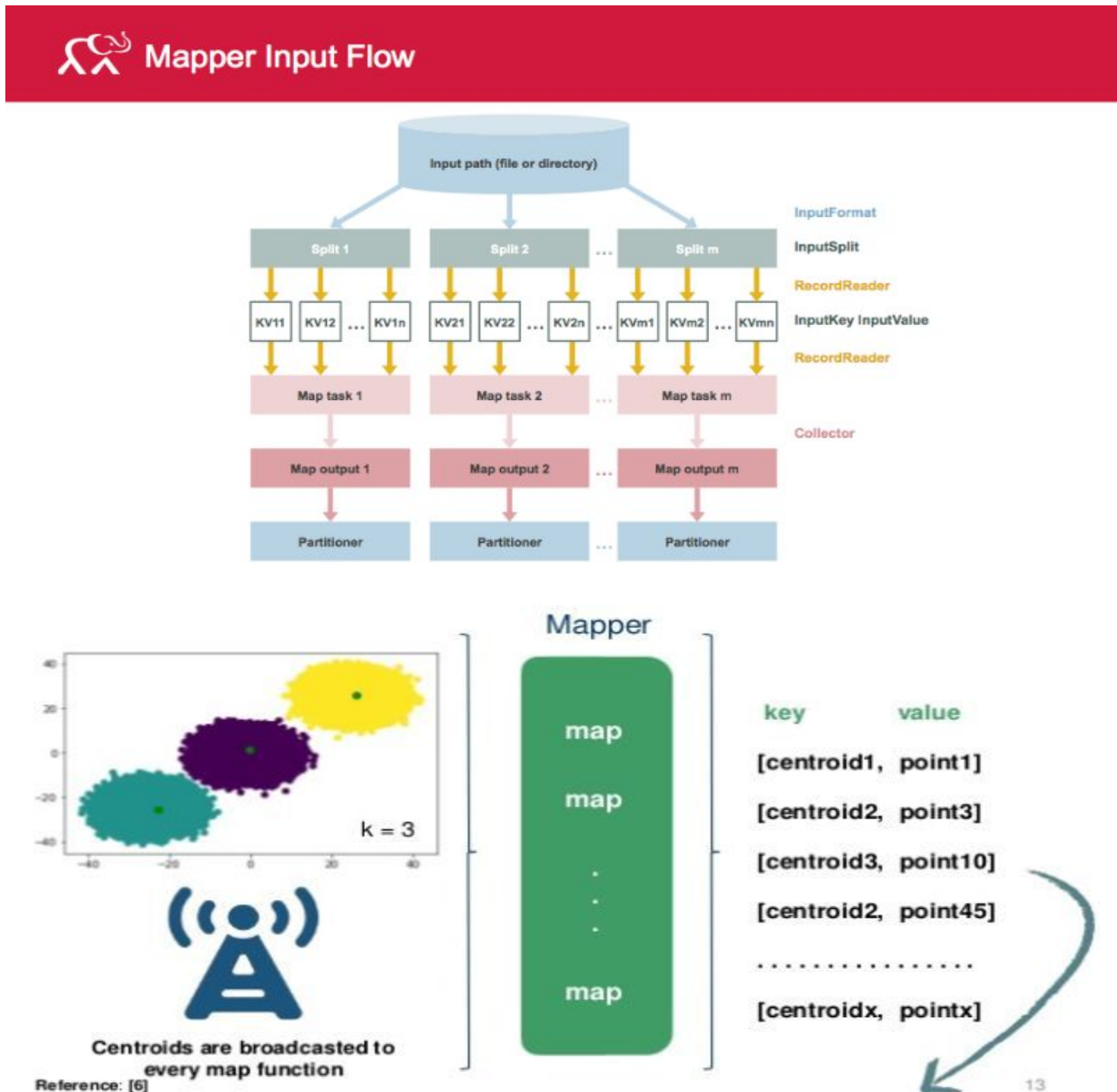
We will follow the following map and reduce procedure.



3 Algorithms & Code description

Map Phase:

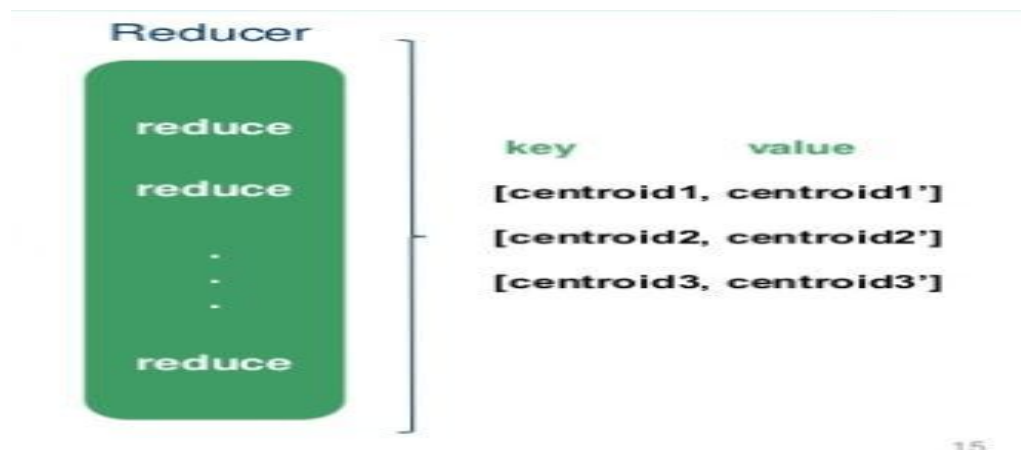
The Mapper has input of a single point then it calculates the closet cluster centroid to be assigned to it.



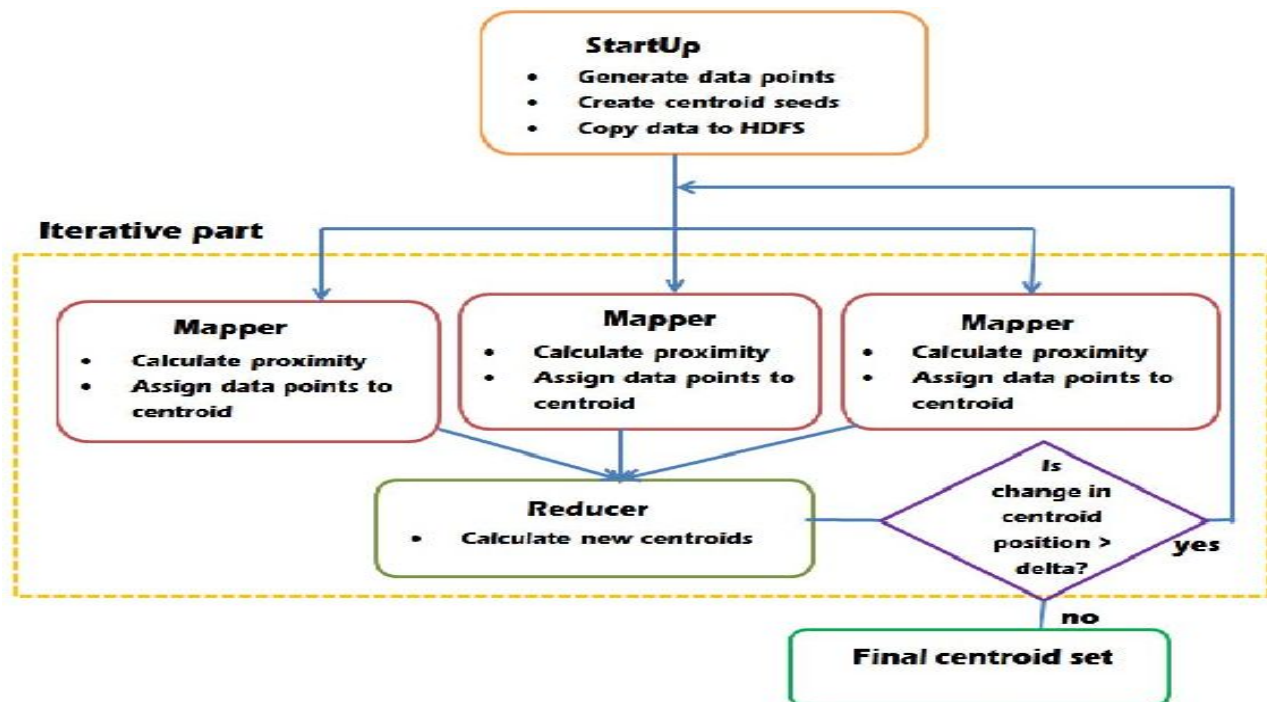
3 Algorithms & Code description

Reduce Phase:

The Reducer will take the cluster index as a key, and the list of points which are assigned to that cluster. Then it will update the cluster centroid by finding the mean of the assigned points.



Over All MapReduce K-means Using Hadoop:



4 Implementation

First, Centroids and Context (Configuration) are loaded into the Distributed Cache. This is done by overriding setup function in the Mapper and Reducer class.

Afterwards, the input data file is split and each data point is processed by one of the map functions (in Map process).

The function writes key-value pairs <Centroid, Point>, where the Centroid is the closest one to the Point.

The Reducer performs the same procedure as the Combiner, but it also checks whether centroids converged (no change) then the global Counter remains unchanged, otherwise, it is incremented.

After the one iteration is done, new centroids are saved and the program checks a condition, if the total distance (no change in centroids) value is unchanged. If then the program is finished, otherwise, the whole MapReduce process is run again with the updated centroids.

Here are some references I used to help in implementing the K-Means:

<https://www.slideshare.net/LampriniKoutsokera/implementation-of-k-means-algorithm-on-hadoop>

shorturl.at/lyJY6 shorturl.at/apPVW shorturl.at/eisQ2

Note that there are some other implementations which use combiner between the map and reduce but we don't use it here.

Algorithm 1 Algorithm for Startup

Require:

- A set of d-dimensional objects $X = \{x_1, x_2, \dots, x_n\}$
- k-number of clusters where $k < n$
- initial set of centroids $C = \{c_1, c_2, \dots, c_k\}$
- δ convergence delta

Output: a new set of centroids, number of iterations, final clusters, time taken to converge

```
1: load( $X, C$ )
2:  $current\_centroids \leftarrow C$ 
3: initialise numIter, timetoConverge, finalClusters
4:  $startTime \leftarrow currentTime()$ 
5:  $C' \leftarrow perform\ MapReduce$ 
6:  $new\_centroids \leftarrow C'$ 
7:  $numIter \leftarrow 1$ 
8: while  $change(new\_centroids, current\_centroids) > \delta$  do
9:    $current\_centroids \leftarrow new\_centroids$ 
10:   $C' \leftarrow perform\ MapReduce$ 
11:   $new\_centroids \leftarrow C'$ 
12:   $numIter \leftarrow numIter + 1$ 
13: end while
14:  $endTime \leftarrow currentTime()$ 
15:  $timetoConverge \leftarrow (endTime - startTime)$ 
16: perform outlierRemoval
17:  $finalClusters \leftarrow perform\ finalClustering$ 
18: writeStatistics
19: return  $current\_centroids, numIter, finalClusters, timetoConverge$ 
```

Algorithm 2 Algorithm for Mapper

Require:

- A subset of d-dimensional objects of $\{x_1, x_2, \dots, x_n\}$ in each mapper
- initial set of centroids $C = \{c_1, c_2, \dots, c_k\}$

Output: A list of centroids and objects assigned to each centroid separated by tab. This list is written locally one line per data point and read by the Reducer program.

```
1:  $M_1 \leftarrow \{x_1, x_2, \dots, x_m\}$ 
2:  $current\_centroids \leftarrow C$ 
3:  $distance(p, q) = \sqrt{\sum_{i=1}^d (p_i - q_i)^2}$  where  $p_i$  (or  $q_i$ ) is the coordinate of p (or q) in dimension i
4: for all  $x_i \in M_1$  such that  $1 \leq i \leq m$  do
5:    $bestCentroid \leftarrow null$ 
6:    $minDist \leftarrow \infty$ 
7:   for all  $c \in current\_centroids$  do
8:      $dist \leftarrow distance(x_i, c)$ 
9:     if  $bestCentroid = null \parallel dist < minDist$  then
10:        $minDist \leftarrow dist$ 
11:        $bestCentroid \leftarrow c$ 
12:     end if
13:   end for
14:    $outputlist \leftarrow (bestCentroid, x_i)$ 
15:    $i \leftarrow i + 1$ 
16: end for
17: return  $outputlist$ 
```

Algorithm 3 Algorithm for Reducer

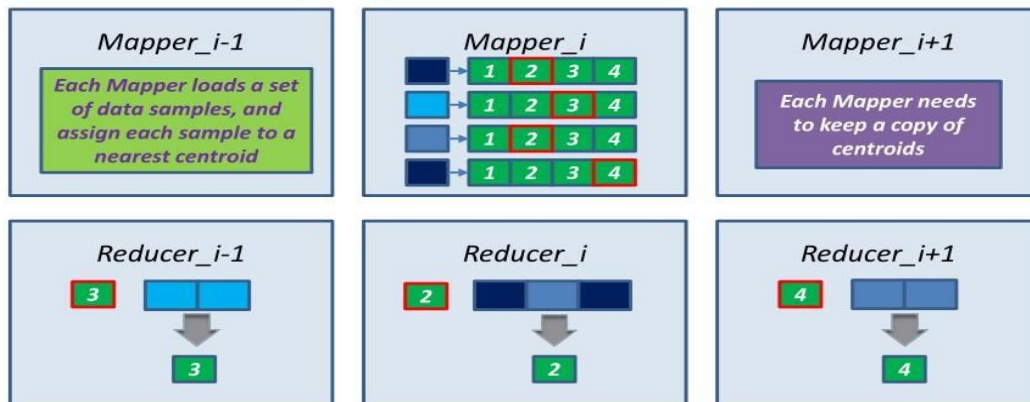
Require:

Input: (key, value) where key = bestCentroid and value = objects assigned to the centroids by the mapper.

Output: (key, value) where key = oldcentroid and value = newBestCentroid which is the new centroid value calculated for that bestCentroid.

```
1: outputlist  $\leftarrow$  outputlists from mappers
2:  $v \leftarrow \{\}$ 
3: newCentroidList  $\leftarrow$  null
4: for all  $y \in$  outputlist do
5:   centroid  $\leftarrow y.key$ 
6:   object  $\leftarrow y.value$ 
7:    $v[centroid] \leftarrow object$ 
8: end for
9: for all centroid  $\in v$  do
10:  newCentroid, sumofObjects, numofObjects  $\leftarrow$  null
11:  for all object  $\in v[centroid]$  do
12:    sumofObjects += object
13:    numofObjects += 1
14:  end for
15:  newCentroid  $\leftarrow$  (sumofObjects  $\div$  numofObjects)
16:  newCentroidList  $\leftarrow$  (newCentroid)
17: end for
18: return newCentroidList
```

K-Means Clustering with MapReduce



How to set the initial centroids is very important!
Usually we set the centroids using *Canopy Clustering*.

[McCallum, Nigam and Ungar: "Efficient Clustering of High Dimensional Data Sets with Application to Reference Matching", SIGKDD 2000]

26

5 Sample Runs

Un-Parallel K-means (Sequential):

```
1 import java.io.BufferedReader;
2 public class main {
3
4     /**
5      * @param args
6      * @throws IOException
7      * @throws NumberFormatException
8      */
9     public static void main(String[] args) throws NumberFormatException, IOException {
10         // if(args.length != 4) {
11         //     System.out.print("args should be 2 : <inputpath> <outpath> <number of centroids> <dimensions> .");
12         //     System.exit(-1);
13         // }
14
15         int k = 2;
16         int dim = 4;
17
18         // reading the iris dataset
19         List<ArrayList<Double>> all_pts = new ArrayList<ArrayList<Double>>();
20         String input_file_path = "/home/said/Desktop/iris.data";
21         BufferedReader br = new BufferedReader(new FileReader(new File(input_file_path)));
22         String line;
23         while ((line = br.readLine()) != null) {
24             ArrayList<Double> point = new ArrayList<Double>();
25             String[] dims = line.split(",");
26             for(int i = 0; i < dims.length - 1; i++){
27                 point.add(Double.parseDouble(dims[i]));
28             }
29             all_pts.add(point);
30         }
31     }
32 }
33
```

Console Output:

```
<terminated> main [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (Mar 8, 2020 10:48:01 AM)
*****
6.1444444444444443 3.2633333333333335 3.9944444444444446 1.3199999999999999
5.391666666666667 2.74 3.405 1.0166666666666666
*****
6.416470588235291 2.9447058823529413 5.114117647058824 1.7658823529411771
5.093046153846155 3.196923076923077 1.986153846153846 0.456923076923077
*****
6.321052631578945 2.890947368421053 4.909473684210527 1.7105263157894743
5.018181818181818 3.321818181818182 1.6327272727272726 0.3145454545454545
*****
6.30103092783505 2.8865979381443303 4.950762806597939 1.6950762806597945
5.005668377358491 3.368377358490567 1.562264150943396 0.2806792452830188
Time token by un-parallel is : 10ms
```

5 Sample Runs

Parallel K-means (MapReduce) Hadoop:

```
Applications Places System 27% 122 B/s 234 B/s Terminal File Edit View Search Terminal Help 23°C en 6:04 م ح مار ٢٠
sald@sald: ~/Desktop/hadoop-2.10.0

Shuffle Errors
  BAD_ID=0
  CONNECTION=0
  IO_ERROR=0
  WRONG_LENGTH=0
  WRONG_MAP=0
  WRONG_REDUCE=0

File Output Format Counters
  Bytes Written=150
20/03/08 10:46:55 INFO mapred.LocalJobRunner: Finishing task: attempt_local550050662_0006_r_000000_0
20/03/08 10:46:55 INFO mapred.LocalJobRunner: reduce task executor complete.
20/03/08 10:46:55 INFO mapreduce.Job: Job job_local550050662_0006 running in uber mode : false
20/03/08 10:46:55 INFO mapreduce.Job: map 100% reduce 100%
20/03/08 10:46:55 INFO mapreduce.Job: Job job_local550050662_0006 completed successfully
20/03/08 10:46:55 INFO mapreduce.Job: Counters: 35

File System Counters
  FILE: Number of bytes read=258150
  FILE: Number of bytes written=5996353
  FILE: Number of read operations=0
  FILE: Number of large read operations=0
  FILE: Number of write operations=0
  HDFS: Number of bytes read=54998
  HDFS: Number of bytes written=1446
  HDFS: Number of read operations=143
  HDFS: Number of large read operations=0
  HDFS: Number of write operations=64

Map-Reduce Framework
  Map input records=151
  Map output records=151
  Map output bytes=5155
  Map output materialized bytes=5463
  Input split bytes=115
  Combine input records=0
  Combine output records=0
  Reduce input groups=2
  Reduce shuffle bytes=5463
  Reduce input records=151
  Reduce output records=2
  Spilled Records=302
  Shuffled Maps =1
  Failed Shuffles=0
  Merged Map outputs=1
  GC time elapsed (ms)=4
  Total committed heap usage (bytes)=1053818880

Shuffle Errors
  BAD_ID=0
  CONNECTION=0
  IO_ERROR=0
  WRONG_LENGTH=0
  WRONG_MAP=0
  WRONG_REDUCE=0

File Input Format Counters
  Bytes Read=4551
File Output Format Counters
  Bytes Written=150

start iteration
Time taken by un-parallel is : 5716ms
sald@sald:~/Desktop/hadoop-2.10.0$
```

K-Means will finish after 14 iterations, so we will find 14 output directory.

6 Results & Conclusion

Both the parallel K-Means and unparallel K-Means versions will have the same final cluster centroids as following.

When using only 2 centroids for iris dataset:

[6.30103093 2.88659794 4.95876289 1.69587629]

[5.00566038 3.36037736 1.56226415 0.28867925]

The Sequential **Un-Parallel** K-Means takes **10** ms.

The **Parallel** MapReduce Hadoop K-Means takes **5716** ms

Notice that the same results obtained using the built-in K-Means in Sklearn.clusters package in python.