

# Lab4: Parallel K-Means using Spark

---

## Distributed Systems

**Presented by**

**1- Mohamed Said 57**

**2- Ahmed el bawab 8**

**3- Khalil Ismail 23**

**4/3/2020**

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 Spark 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

---

|   |              |
|---|--------------|
| <b>1- Problem Definition</b>                | <b>3</b>     |
| <b>2- Goals</b>                             | <b>4</b>     |
| <b>3- Pseudo-Code</b>                       | <b>5-6</b>   |
| <b>4- Algorithms &amp; Code description</b> | <b>7-9</b>   |
| <b>5- Implementation</b>                    | <b>10-12</b> |
| <b>6- Sample Runs</b>                       | <b>13-15</b> |
| <b>7- Results &amp; Conclusion</b>          | <b>16</b>    |
| <b>8- Challenges</b>                        | <b>17</b>    |

# 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, Parallel K-Means Spark MapReduce , focuses on the MapReduce implementation of standard K-means using Spark.

And the other is the Unparalleled (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 Spark MapReduce framework.
2. Implementing an unparallelled version of the K-Means algorithm using normal implementation.
3. Evaluating the parallel version and comparing it with the unparallelled version of K-Means using the IRIS dataset in terms of run time and clustering accuracy.

## 3 Pseudo-Code

---

### **Unparallel K-means Pseudo-Code:**

Function K-means ( K : number of clusters, D : dataset of samples) :

1. Initialize k cluster centroid randomly :  $M(1), M(2), \dots$  to  $M(k)$ .
2. Repeat Until Convergence:
  - a. For every sample  $i$  in  $D$  :
    - i.  $C(i) = \operatorname{argmin}_j (\text{ecludian\_distance}(D(i) - M(j)))$
  - b. For  $j$  from 1 to  $K$ :
    - i.  $M(j) = \text{Mean}(\text{any sample } i \text{ where } C(i) == j)$
3. Return the cluster centroids.

## 3 Pseudo-Code

---

### **Parallel K-means Pseudo-Code:**

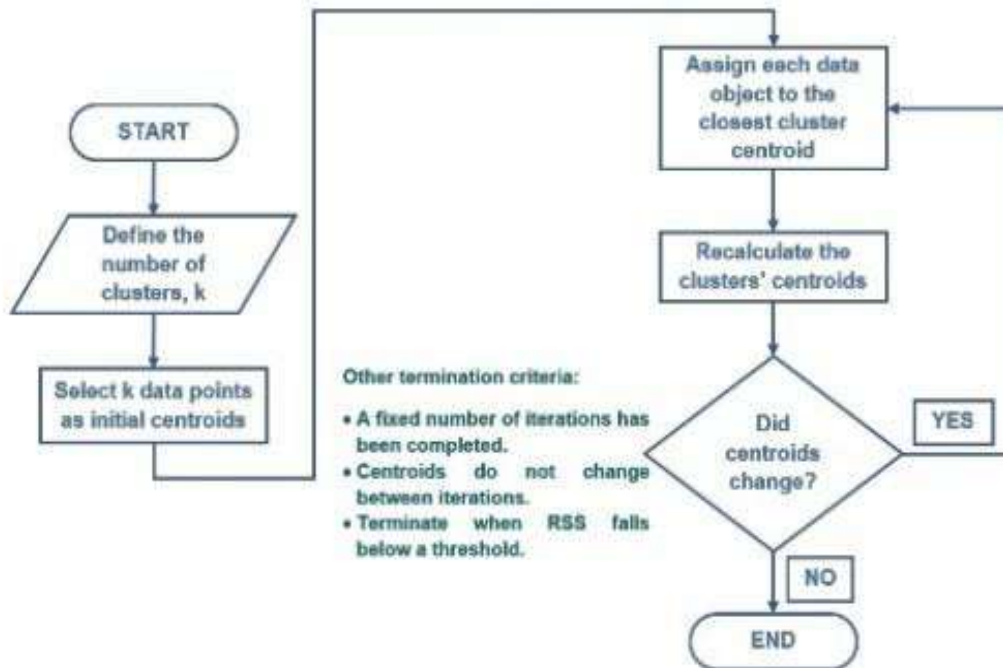
Function K-means ( K : number of clusters, D : dataset of samples) :

1. Initialize k cluster centroid randomly :  $M(1), M(2), \dots$  to  $M(k)$ .
2. Repeat Until Convergence:
  - a. Map each sample  $i$  in  $D$  to a centroid in parallel:
    - i.  $C(i) = \operatorname{argmin}_j (\text{ecludian\_distance}(D(i) - M(j)))$
    - ii. Return Pairs of  $(C(i), D(i))$
  - b. For  $j$  from 1 to  $K$ , calculate each centroid new mean in parallel by reducing pairs with same key  $C(i)$ :
    - i.  $M(j) = \text{Mean}(\text{any sample } i \text{ where } C(i) == j)$
3. Return the cluster centroids.

## 4 Algorithms & Code description

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

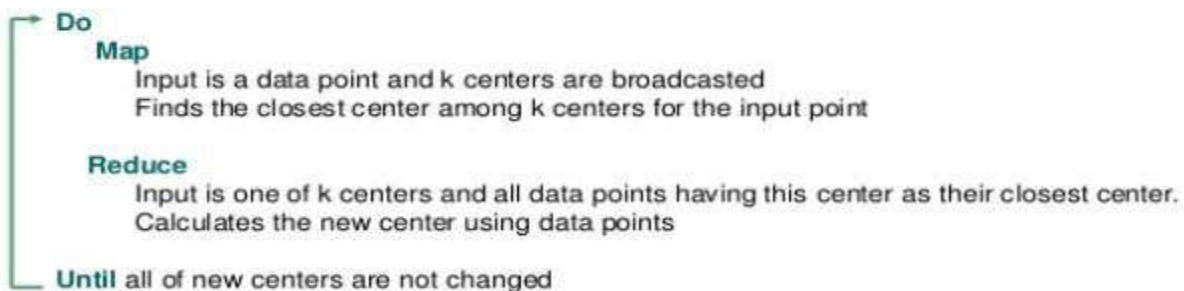
### Unparalleled K-Means:



### 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.



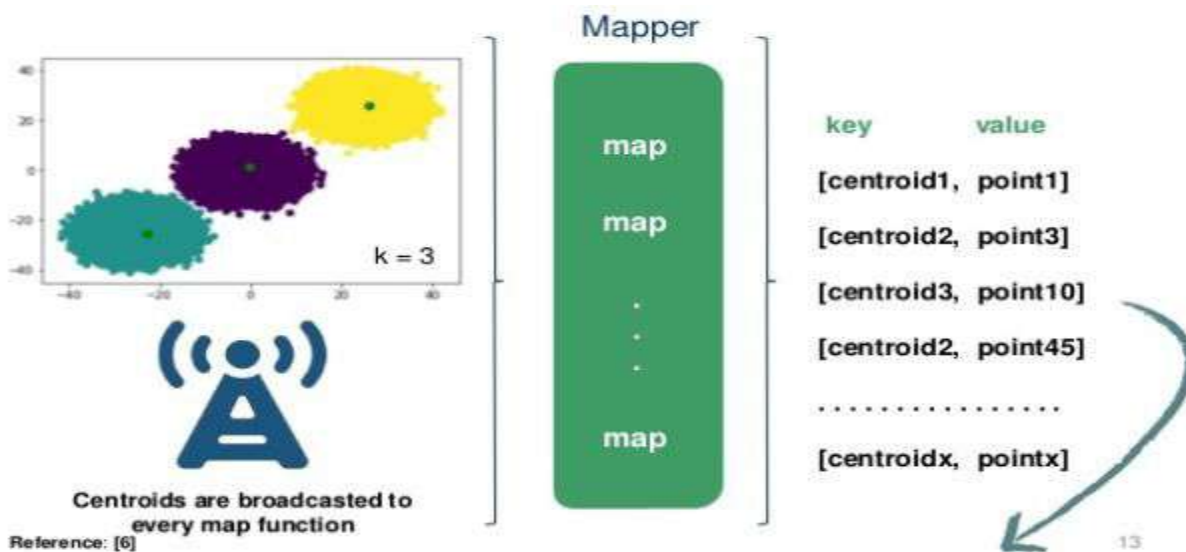
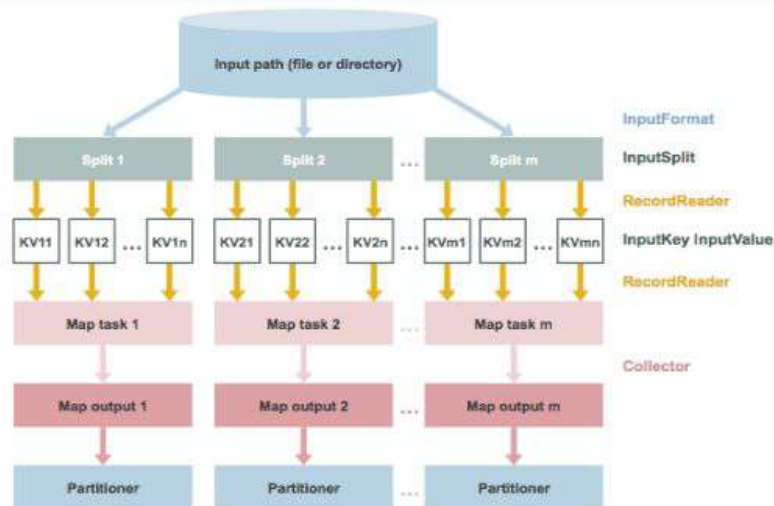
## 4 Algorithms & Code description

### Map Phase:

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



### Mapper Input Flow

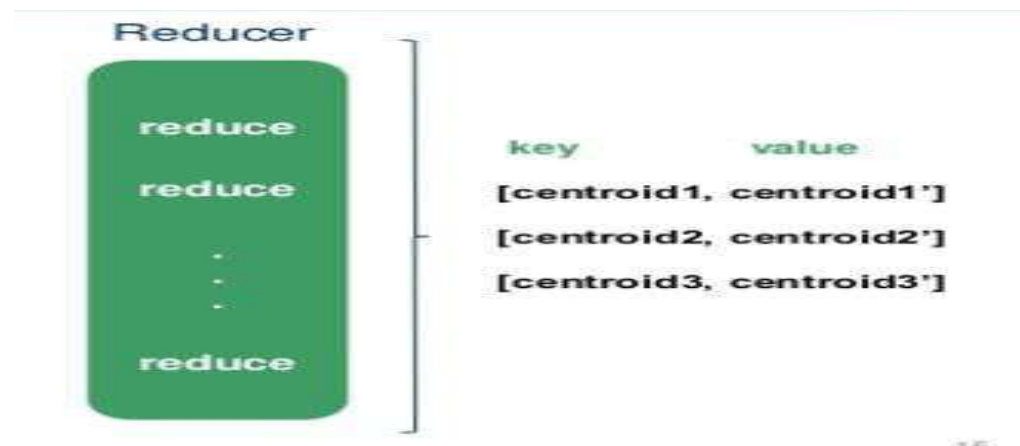




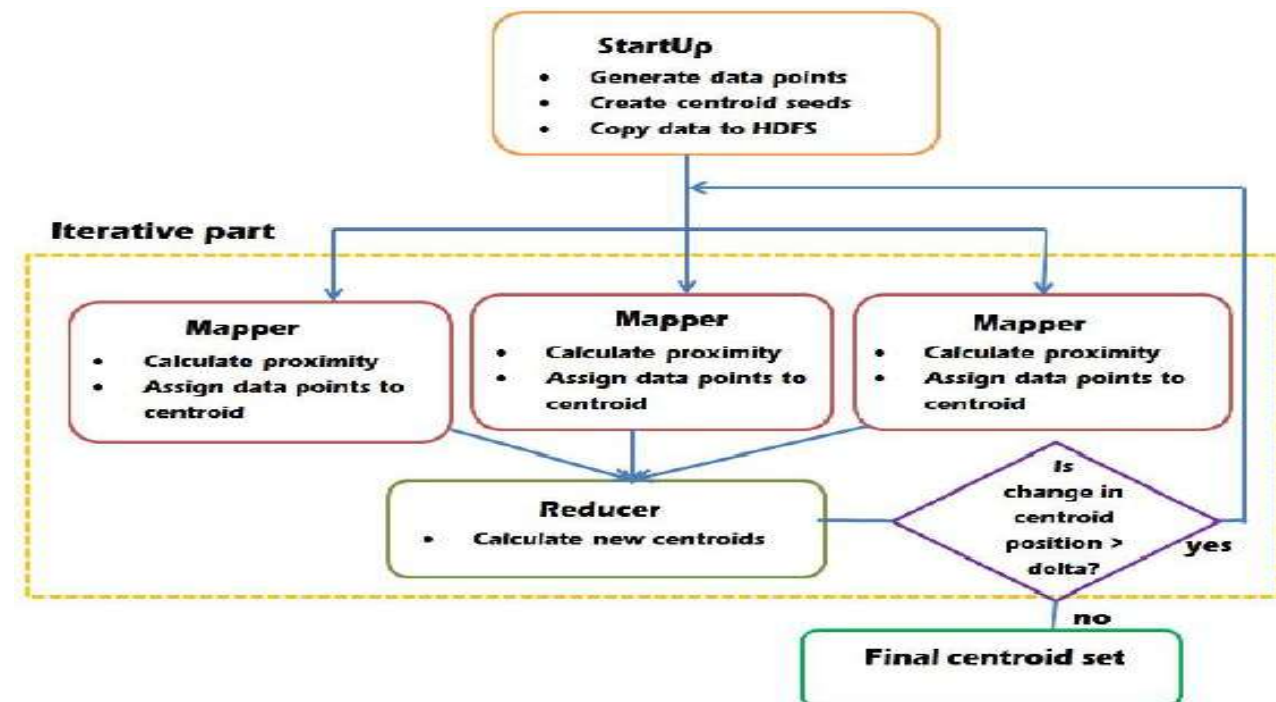
## 4 Algorithms & Code description

### 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:



# 5 Implementation

---

## **Main method:**

```
public static void main(String[] args) throws Exception {

    String path = args[0];
    int k = Integer.parseInt(args[2]);
    int maxIterations = Integer.MAX_VALUE;
    double convergenceEpsilon = 0; // default
    if (args.length > 3) {
        if (!args[3].equals("MAX")) {
            maxIterations = Integer.parseInt(args[3]);
        }
    }
    if (args.length > 4) {
        convergenceEpsilon = Double.parseDouble(args[4]);
    }

    SparkConf conf = new SparkConf().setMaster("local").setAppName("kmeans");
    context = new JavaSparkContext(conf);

    JavaRDD<Vector> data = context.textFile(path)
        .map(new Function<String, Vector>() {
            @Override
            public Vector call(String line) {
                return parseVector(line);
            }
        })
        .cache();

    context.parallelize(kmeans(data, k, convergenceEpsilon, maxIterations)).saveAsTextFile(args[1]);

    System.exit(0);
}
```

## K-means method

```
public static List<Vector> kmeans(JavaRDD<Vector> data, int k,
    double convergeDist, long maxIterations) {

    final List<Vector> centroids = data.takeSample(false, k);
    long counter = 0;
    double tempDist;
    Instant start = Instant.now();
    do {

        JavaPairRDD<Integer, Vector> closest = data
            .mapToPair(new PairFunction<Vector, Integer, Vector>() {
                @Override
                public Tuple2<Integer, Vector> call(Vector vector) {
                    return new Tuple2<Integer, Vector>(closestPoint(
                        vector, centroids), vector);
                }
            });

        JavaPairRDD<Integer, Iterable<Vector>> pointsGroup = closest.groupByKey();
        Map<Integer, Vector> newCentroids = pointsGroup.mapValues(
            new Function<Iterable<Vector>, Vector>() {
                @Override
                public Vector call(Iterable<Vector> ps) {
                    ArrayList<Vector> list = new ArrayList<Vector>();
                    if(ps != null) {
                        for(Vector e: ps) {
                            list.add(e);
                        }
                    }
                    return average(list);
                }
            }).collectAsMap();
        tempDist = 0.0;
        for (int j = 0; j < k; j++) {
            tempDist += centroids.get(j).squaredDist(newCentroids.get(j));
        }
        for (Map.Entry<Integer, Vector> t : newCentroids.entrySet()) {
            centroids.set(t.getKey(), t.getValue());
        }
        counter++;
    } while (tempDist > convergeDist && counter < maxIterations);
    Instant end = Instant.now();
    Duration timeElapsed = Duration.between(start, end);
    System.out.println("Time taken: " + timeElapsed.toMillis() + " milliseconds");
    System.out.println("Converged in " + String.valueOf(counter) + " iterations.");
    System.out.println("Final centers:");
    for (Vector c : centroids) {
        System.out.println(c);
    }
    return centroids;
}
```

---

**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  $C = \{c_1, c_2, \dots, c_k\}$
- $\delta$  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 || 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 += 1
16: end for
17: return outputlist
```

---



---

**Algorithm 3** Algorithm for Reducer

---

**Require:**

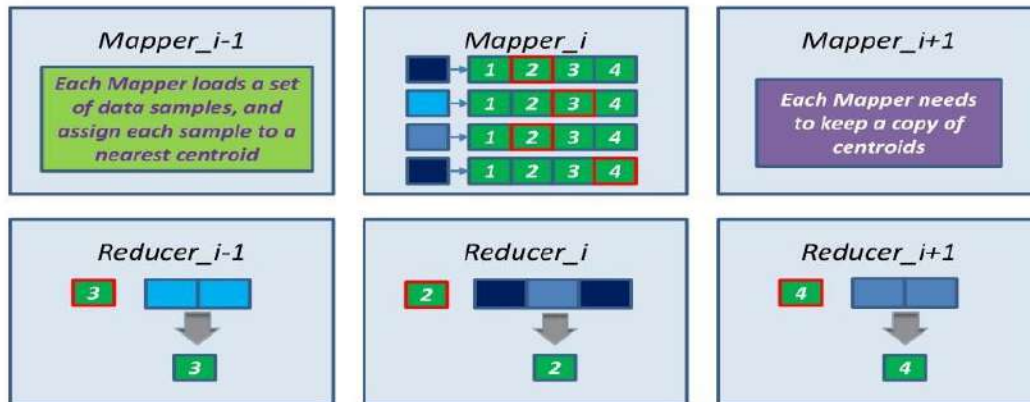
Input: (key, value) where key = bestCentroid and value = objects assigned to the centroids by the mapper.

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 << (newCentroid)
17: end for
18: return newCentroidList
```

---

## K-Means Clustering with MapReduce



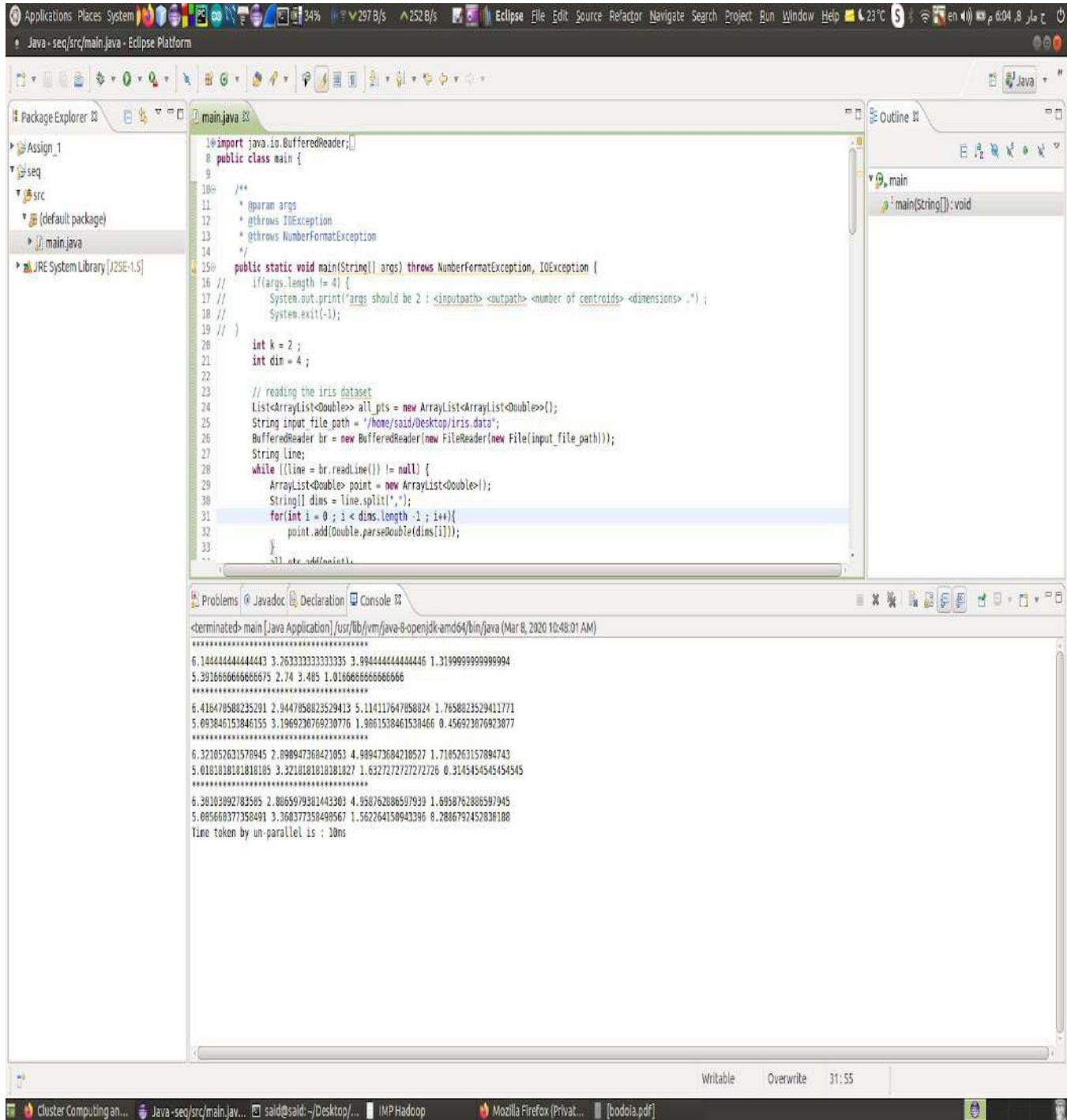
**How to set the initial centroids is very important!**  
**Usually we set the centroids using *Canopy Clustering*.**

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

26

# 6 Sample Runs

## ***Un-Parallel K-means (Sequential):***



```
import java.io.BufferedReader;
public class main {
    /**
     * @param args
     * @throws IOException
     * @throws NumberFormatException
     */
    public static void main(String[] args) throws NumberFormatException, IOException {
        if(args.length != 4) {
            System.out.print("args should be 2 : <inputpath> <outpath> <number of centroids> <dimensions> .");
            System.exit(-1);
        }
        int k = 2;
        int dim = 4;

        // reading the iris dataset
        List<ArrayList<Double>> all_pts = new ArrayList<ArrayList<Double>>();
        String input_file_path = "/home/said/Desktop/iris.data";
        BufferedReader br = new BufferedReader(new FileReader(new File(input_file_path)));
        String line;
        while ((line = br.readLine()) != null) {
            ArrayList<Double> point = new ArrayList<Double>();
            String[] dims = line.split(",");
            for(int i = 0; i < dims.length - 1; i++){
                point.add(Double.parseDouble(dims[i]));
            }
            all_pts.add(point);
        }

        // ... (rest of the code) ...

        <terminated> main [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (Mar 8, 2020 10:48:01 AM)
        *****
        6.144444444444443 3.7633333333333335 3.9944444444444446 1.3199999999999994
        5.3016666666666675 2.74 3.405 1.0166666666666666
        *****
        6.416478588235291 2.9447858823529413 5.114117647858824 1.7658823529411771
        5.093846153846155 3.1969230769230776 1.9861538461538466 0.456923076923077
        *****
        6.321852631578945 2.898947368421051 4.899473684210527 1.7105263157894743
        5.0181818181818185 3.3218181818181827 1.6327272727272726 0.3145454545454545
        *****
        6.30103092783505 2.8065979301443303 4.958762086507939 1.6958762086507945
        5.085668377358491 3.368377358490567 1.562264158943396 0.2886792452838188
        Time taken by un-parallel is : 10ms
```

# 6 Sample Runs

## ***Parallel K-means (MapReduce) Spark:***

```
Applications Places System 76% 47 B/s 70 B/s 20°C en 7:30 م
said@said: ~/Desktop/Distributed-Kmeans/Spark/target
20/04/03 19:29:52 INFO BlockManagerInfo: Added broadcast_10_piece0 in memory on localhost:45541 (size: 3.1 KB, free: 948.5 MB)
20/04/03 19:29:52 INFO SparkContext: Created broadcast 10 from broadcast at DAGScheduler.scala:874
20/04/03 19:29:52 INFO DAGScheduler: Submitting 1 missing tasks from ResultStage 9 (MapPartitionsRDD[19] at mapValues at Kmeans.java:91)
20/04/03 19:29:52 INFO TaskSchedulerImpl: Adding task set 9.0 with 1 tasks
20/04/03 19:29:52 INFO TaskSetManager: Starting task 0.0 in stage 9.0 (TID 9, localhost, PROCESS_LOCAL, 1165 bytes)
20/04/03 19:29:52 INFO Executor: Running task 0.0 in stage 9.0 (TID 9)
20/04/03 19:29:52 INFO ShuffleBlockFetcherIterator: Getting 1 non-empty blocks out of 1 blocks
20/04/03 19:29:52 INFO ShuffleBlockFetcherIterator: Started 0 remote fetches in 0 ms
20/04/03 19:29:52 INFO Executor: Finished task 0.0 in stage 9.0 (TID 9). 1287 bytes result sent to driver
20/04/03 19:29:52 INFO TaskSetManager: Finished task 0.0 in stage 9.0 (TID 9) in 9 ms on localhost (1/1)
20/04/03 19:29:52 INFO TaskSchedulerImpl: Removed TaskSet 9.0, whose tasks have all completed, from pool
20/04/03 19:29:52 INFO DAGScheduler: ResultStage 9 (collectAsMap at Kmeans.java:103) finished in 0.009 s
20/04/03 19:29:52 INFO DAGScheduler: Job 5 finished: collectAsMap at Kmeans.java:103, took 0.055587 s
Time taken: 446 milliseconds
Converged in 4 iterations.
Final centers:
(5.056666666666667, 3.2600000000000007, 1.8116666666666668, 0.38666666666666666)
(6.367777777777777, 2.9166666666666667, 5.056666666666667, 1.7400000000000007)
20/04/03 19:29:52 INFO SparkContext: Starting job: saveAsTextFile at Kmeans.java:66
20/04/03 19:29:52 INFO DAGScheduler: Got job 6 (saveAsTextFile at Kmeans.java:66) with 1 output partitions (allowLocal=false)
20/04/03 19:29:52 INFO DAGScheduler: Final stage: ResultStage 10 (saveAsTextFile at Kmeans.java:66)
20/04/03 19:29:52 INFO DAGScheduler: Parents of final stage: List()
20/04/03 19:29:52 INFO DAGScheduler: Missing parents: List()
20/04/03 19:29:52 INFO DAGScheduler: Submitting ResultStage 10 (MapPartitionsRDD[21] at saveAsTextFile at Kmeans.java:66), which has no missing parents
20/04/03 19:29:52 INFO MemoryStore: ensureFreeSpace(94640) called with curMem=213267, maxMem=994584821
20/04/03 19:29:52 INFO MemoryStore: Block broadcast_11 stored as values in memory (estimated size 92.4 KB, free 948.2 MB)
20/04/03 19:29:52 INFO MemoryStore: ensureFreeSpace(30968) called with curMem=307907, maxMem=994584821
20/04/03 19:29:52 INFO MemoryStore: Block broadcast_11_piece0 stored as bytes in memory (estimated size 30.2 KB, free 948.2 MB)
20/04/03 19:29:52 INFO BlockManagerInfo: Added broadcast_11_piece0 in memory on localhost:45541 (size: 30.2 KB, free: 948.4 MB)
20/04/03 19:29:52 INFO SparkContext: Created broadcast 11 from broadcast at DAGScheduler.scala:874
20/04/03 19:29:52 INFO DAGScheduler: Submitting 1 missing tasks from ResultStage 10 (MapPartitionsRDD[21] at saveAsTextFile at Kmeans.java:66)
20/04/03 19:29:52 INFO TaskSchedulerImpl: Adding task set 10.0 with 1 tasks
20/04/03 19:29:52 INFO TaskSetManager: Starting task 0.0 in stage 10.0 (TID 10, localhost, PROCESS_LOCAL, 1535 bytes)
20/04/03 19:29:52 INFO Executor: Running task 0.0 in stage 10.0 (TID 10)
20/04/03 19:29:52 INFO deprecation: mapred.output.dir is deprecated. Instead, use mapreduce.output.fileoutputformat.outputdir
20/04/03 19:29:52 INFO deprecation: mapred.output.key.class is deprecated. Instead, use mapreduce.job.output.key.class
20/04/03 19:29:52 INFO deprecation: mapred.output.value.class is deprecated. Instead, use mapreduce.job.output.value.class
20/04/03 19:29:52 INFO deprecation: mapred.working.dir is deprecated. Instead, use mapreduce.job.working.dir
20/04/03 19:29:52 INFO FileOutputCommitter: Saved output of task 'attempt_202004031929_0010_m_000000_10' to file:/home/said/Desktop/output4/_temporary/0/task_202004031929_0010_m_000000
20/04/03 19:29:52 INFO SparkHadoopMapRedUtil: attempt_202004031929_0010_m_000000_10: Committed
20/04/03 19:29:52 INFO Executor: Finished task 0.0 in stage 10.0 (TID 10). 620 bytes result sent to driver
20/04/03 19:29:52 INFO TaskSetManager: Finished task 0.0 in stage 10.0 (TID 10) in 152 ms on localhost (1/1)
20/04/03 19:29:52 INFO TaskSchedulerImpl: Removed TaskSet 10.0, whose tasks have all completed, from pool
20/04/03 19:29:52 INFO DAGScheduler: ResultStage 10 (saveAsTextFile at Kmeans.java:66) finished in 0.17352 s
20/04/03 19:29:52 INFO DAGScheduler: Job 6 finished: saveAsTextFile at Kmeans.java:66, took 0.177352 s
20/04/03 19:29:52 INFO SparkContext: Invoking stop() from shutdown hook
20/04/03 19:29:52 INFO SparkUI: Stopped Spark web UI at http://192.168.43.215:4040
20/04/03 19:29:52 INFO DAGScheduler: Stopping DAGScheduler
20/04/03 19:29:52 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
20/04/03 19:29:52 INFO Utils: path = /tmp/spark-34d4bf46-5d63-42f7-a577-904e298c5680/blockmgr-5393efcb-8abf-4d57-bf3f-693feb2e8283, already present as root for deletion.
20/04/03 19:29:52 INFO MemoryStore: MemoryStore cleared
20/04/03 19:29:52 INFO BlockManager: BlockManager stopped
20/04/03 19:29:52 INFO BlockManagerMaster: BlockManagerMaster stopped
20/04/03 19:29:52 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
20/04/03 19:29:52 INFO SparkContext: Successfully stopped SparkContext
20/04/03 19:29:52 INFO Utils: Shutdown hook called
20/04/03 19:29:52 INFO Utils: Deleting directory /tmp/spark-34d4bf46-5d63-42f7-a577-904e298c5680
said@said:~/Desktop/Distributed-Kmeans/Spark/target$
```

## 7 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 Spark K-Means takes **446** ms

The **Parallel** MapReduce Hadoop K-Means takes **5716** ms{Last Lab}

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



## 8 Challenges Faced

---

- Passing Feature Row per Sample in Mapper: We decided to parse each line that represents values of features per sample and separating them by the delimiter ',', then converting these values to a vector of double values.
- How to get Initial Centroid: We decided to set the initial centroid randomly picking k-samples from the initial file.
- How to pass results of each round: Pass centroids as arguments to the mapping function, and update their values with each round.
- Number of clusters: Take it as a command line argument from the user.
- Termination condition: Either terminate with a maximum number of iterations, or when change to centroids is less than or equal a certain threshold. Both maximum number of iterations and the threshold are command line arguments by the user with default values of maximum possible integer for number of iterations and a threshold of 0 for the change.