FACULTY OF ENGINEERING

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

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 unparalleled version of the K-Means algorithm using normal implementation.

3. Evaluating the parallel version and comparing it with the unparalleled version of K-Means using the IRIS dataset in terms of run time and clustering accuracy.

3 Pseudo-Code

<u>Unparallel K-means Pseudo-Code:</u>

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

1. Initialize k cluster centroid randomly : M(1), M(2), ... to M(k).

2. Repeat Until Convergence:

a. For every sample i in D :

    i. C(i) = argmin j (ecludian_distance(D(i) - M(j))

b. For j from 1 to K:

    i.M(j) = Mean(any sample i 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), ... to M(k).

2. Repeat Until Convergence:

a. Map each sample i in D to a centroid in parallel:

    i.C(i) = argmin j (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):

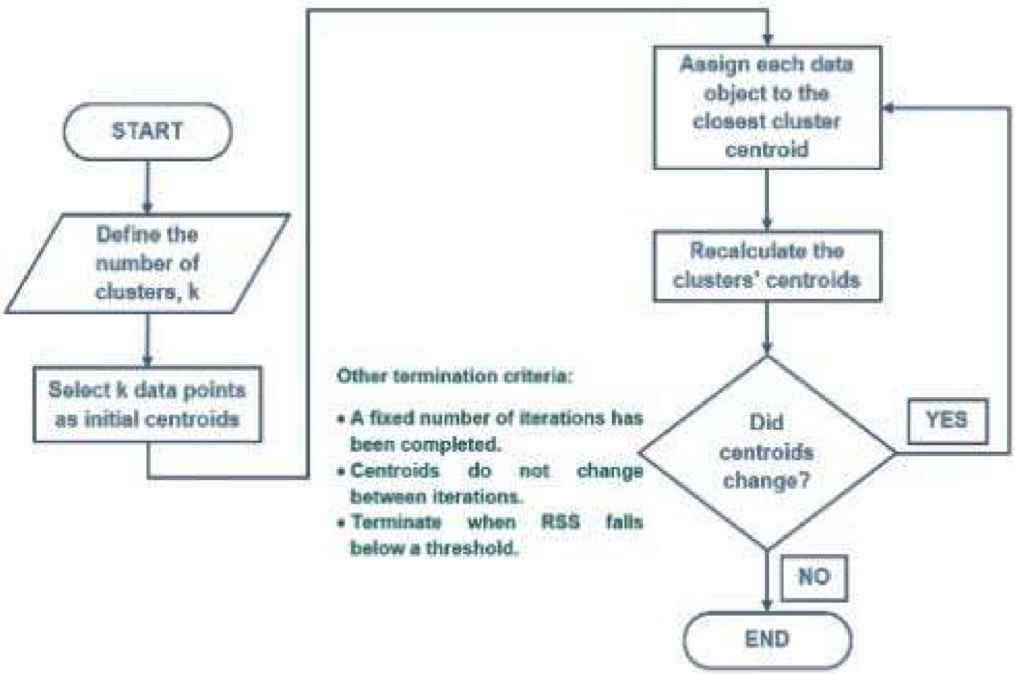
    i.M(j) = Mean(any sample i where C(i) == j)

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.

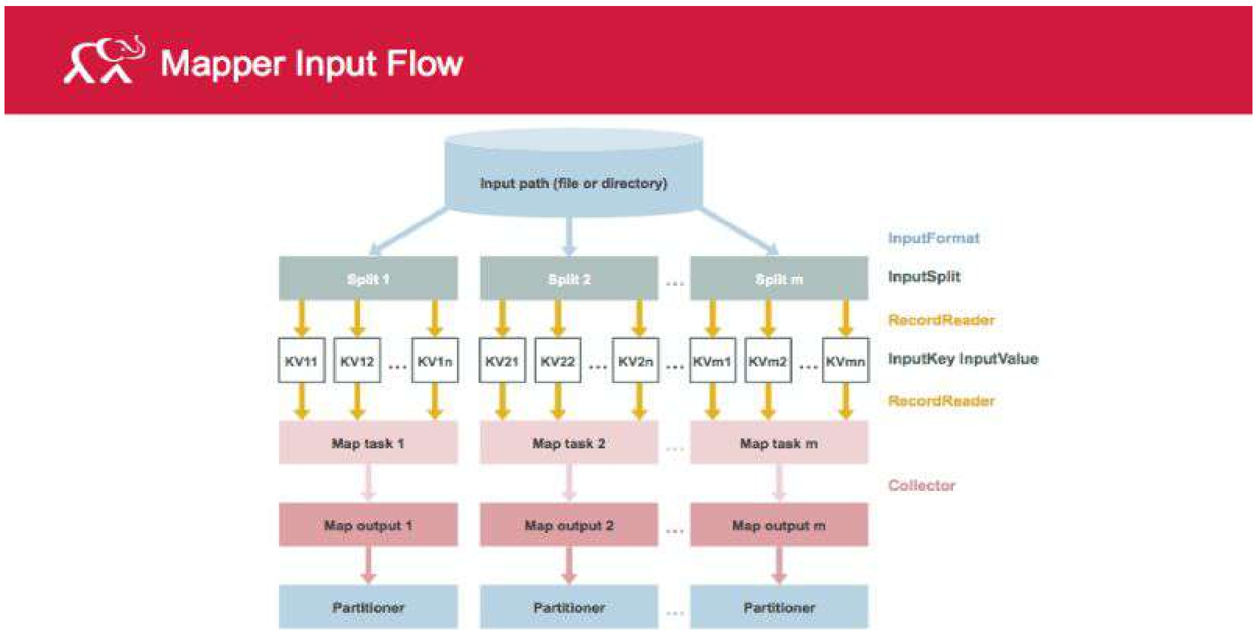
We will follow the following map and reduce procedure.

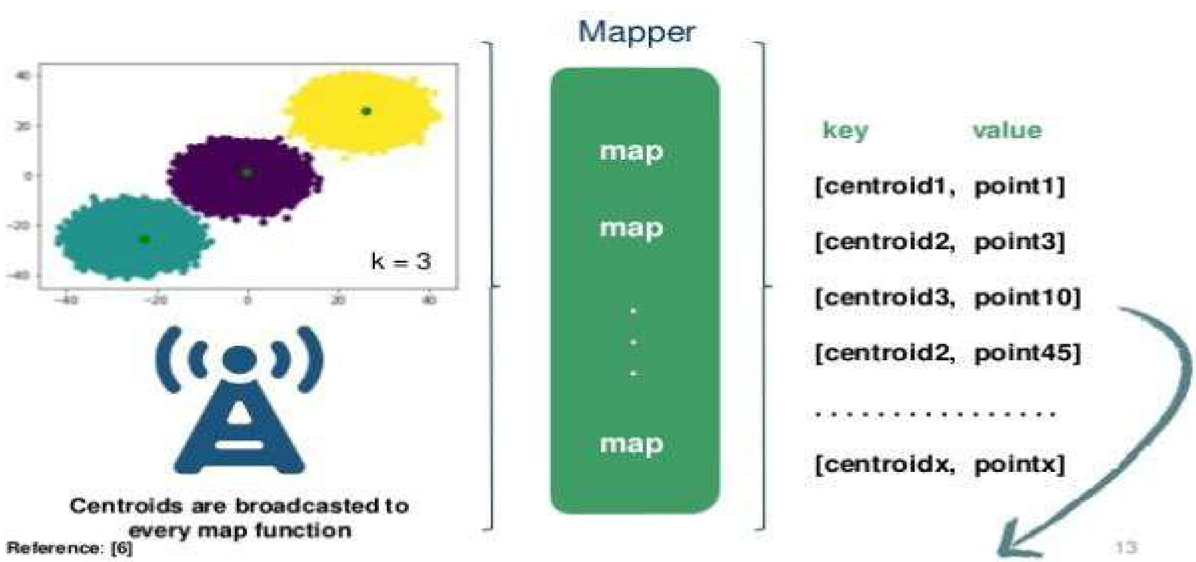
```
Do
    Map
        Input is a data point and k centers are broadcasted
        Finds the closest center among k centers for the input point

    Reduce
        Input is one of k centers and all data points having this center as their closest center.
        Calculates the new center using data points

Until all of new centers are not changed
```

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.

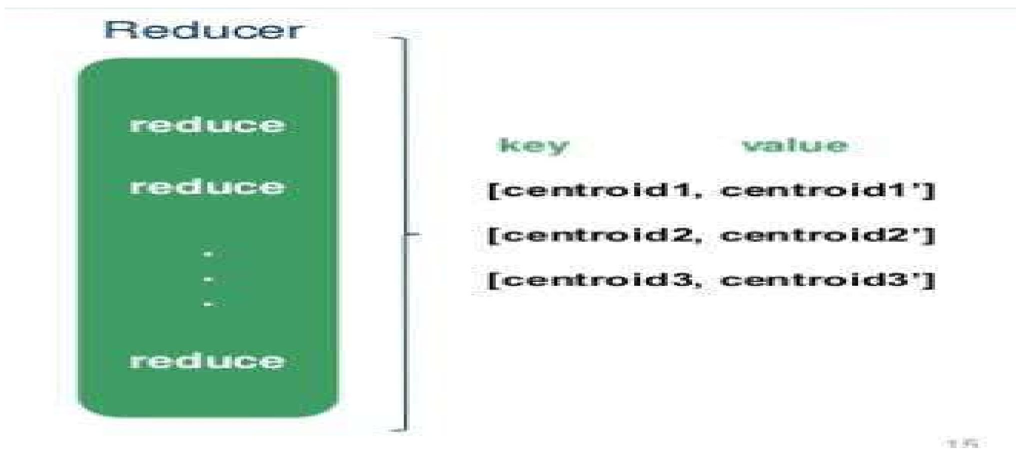




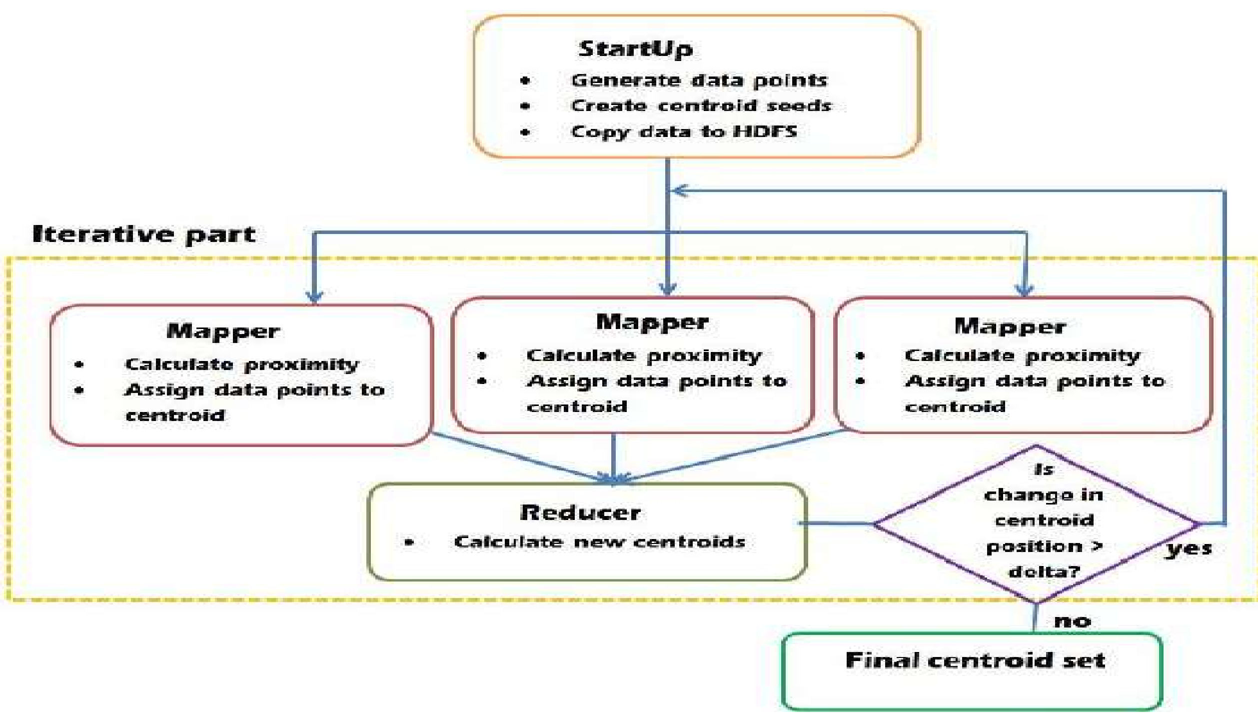
4 Algorithms & Code description

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 convergenceEpslon = 0; // default
    if (args.length > 3) {
        if (!args[3].equals("MAX")) {
            maxIterations = Integer.parseInt(args[3]);
        }
    }
    if (args.length > 4) {
        convergenceEpslon = 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, convergenceEpslon, 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= $\{x_1, x_2, \ldots, x_n\}$
- k-number of clusters where $k < n$
- initial set of centroids $C = \{c_1, c_2, \ldots, c_k\}$
- δ convergence delta

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

$load\,(X, C)$
$current\_centroids \Leftarrow C$
$initialise$ numIter, timetoConverge, finalClusters
$startTime \Leftarrow currentTime()$
$C' \Leftarrow perform\ MapReduce$
$new\_centroids \Leftarrow C'$
$numIter \Leftarrow 1$
while $change(new\_centroids, current\_centroids) > \delta$ **do**
    $current\_centroids \Leftarrow new\_centroids$
    $C' \Leftarrow$ perform MapReduce
    $new\_centroids \Leftarrow C'$
    $numIter \Leftarrow numIter + 1$
end while
$endTime \Leftarrow currentTime()$
$timetoConverge \Leftarrow (endTime - startTime)$
$perform\ outlierRemoval$
$finalClusters \Leftarrow perform\ finalClustering$
$writeStatistics$
return $current\_centroids, numIter, finalClusters, timetoConverge$

Algorithm 2 Algorithm for Mapper

Require:

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

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.

$M_1 \Leftarrow \{x_1, x_2, \ldots, x_m\}$
$current\_centroids \Leftarrow C$
$distance\,(p, q) = \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2}$ where p_i (or q_i) is the coordinate of p (or q) in dimension i
for all $x_i \in M_1$ such that $1 \leq i \leq m$ **do**
    $bestCentroid \Leftarrow null$
    $minDist \Leftarrow \infty$
    for all $c \in current\_centroids$ **do**
        $dist \Leftarrow distance(x_i, c)$
        if $bestCentroid = null \mid\mid dist < minDist$ **then**
            $minDist \Leftarrow dist$
            $bestCentroid \Leftarrow c$
        end if
    end for
    $outputlist << (bestCentroid, x_i)$
    i +=1
end for
return $outputlist$

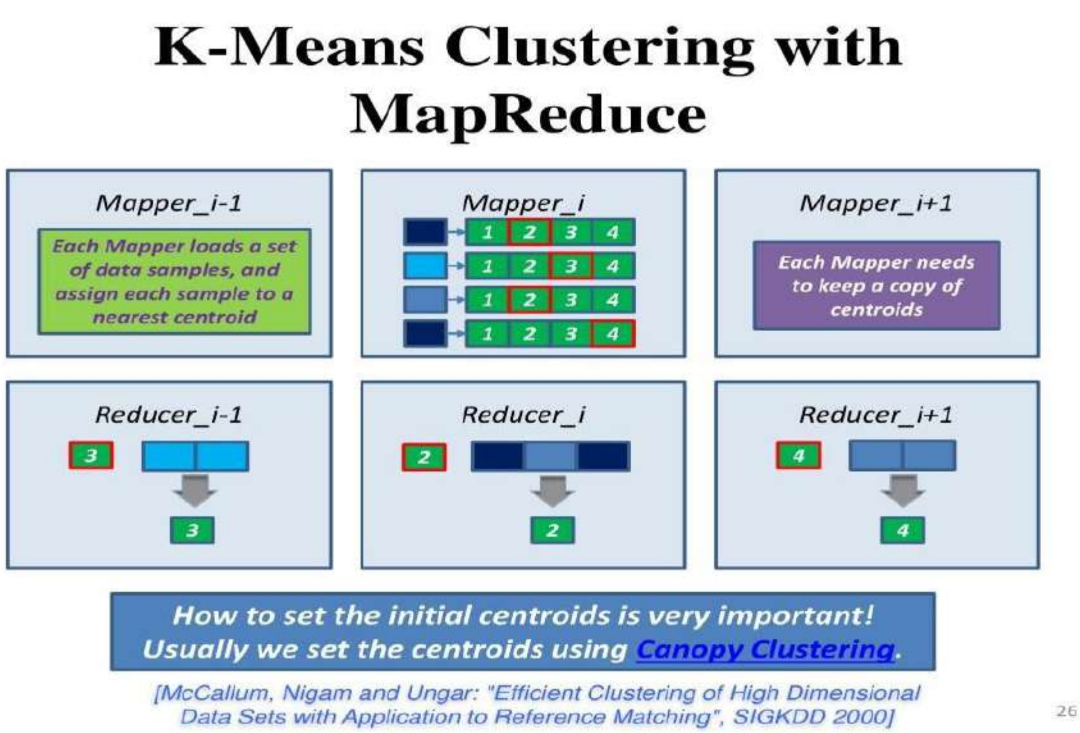
Algorithm 3 Algorithm for Reducer

Require:

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

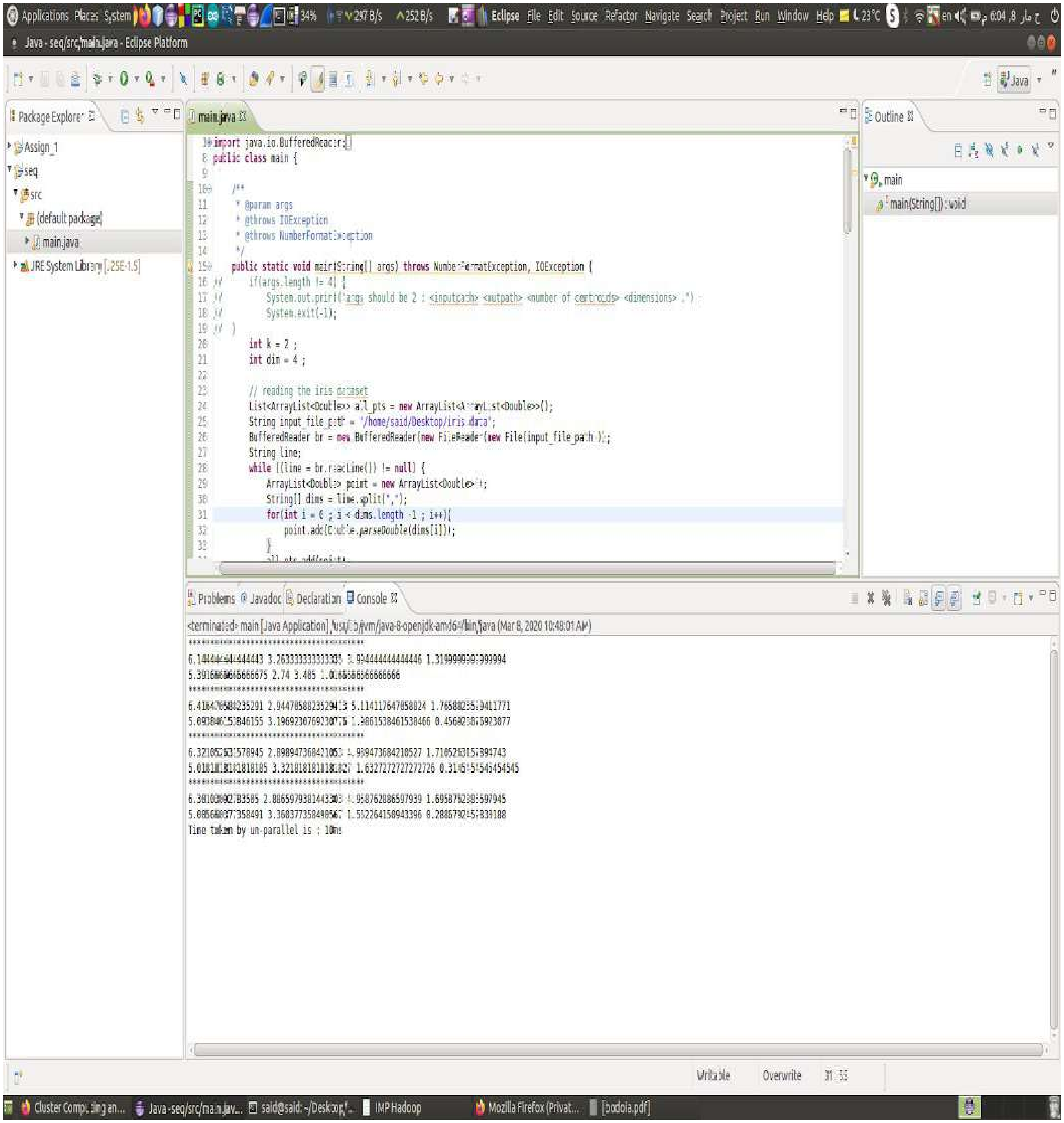
Output: (key, value) where key = oldcentroid and value = newBestCentroid which is the new centroid value calculated for that bestCentroid.

```
outputlist ⇐ outputlists from mappers
v ⇐ {}
newCentroidList ⇐ null
for all y ∈ outputlist do
    centroid ⇐ y.key
    object ⇐ y.value
    v[centroid] ⇐ object
end for
for all centroid ∈ v do
    newCentroid, sumofObjects, numofObjects ⇐ null
    for all object ∈ v[centroid] do
        sumofObjects += object
        numofObjects += 1
    end for
    newCentroid ⇐ (sumofObjects ÷ numofObjects)
    newCentroidList << (newCentroid)
end for
return newCentroidList
```



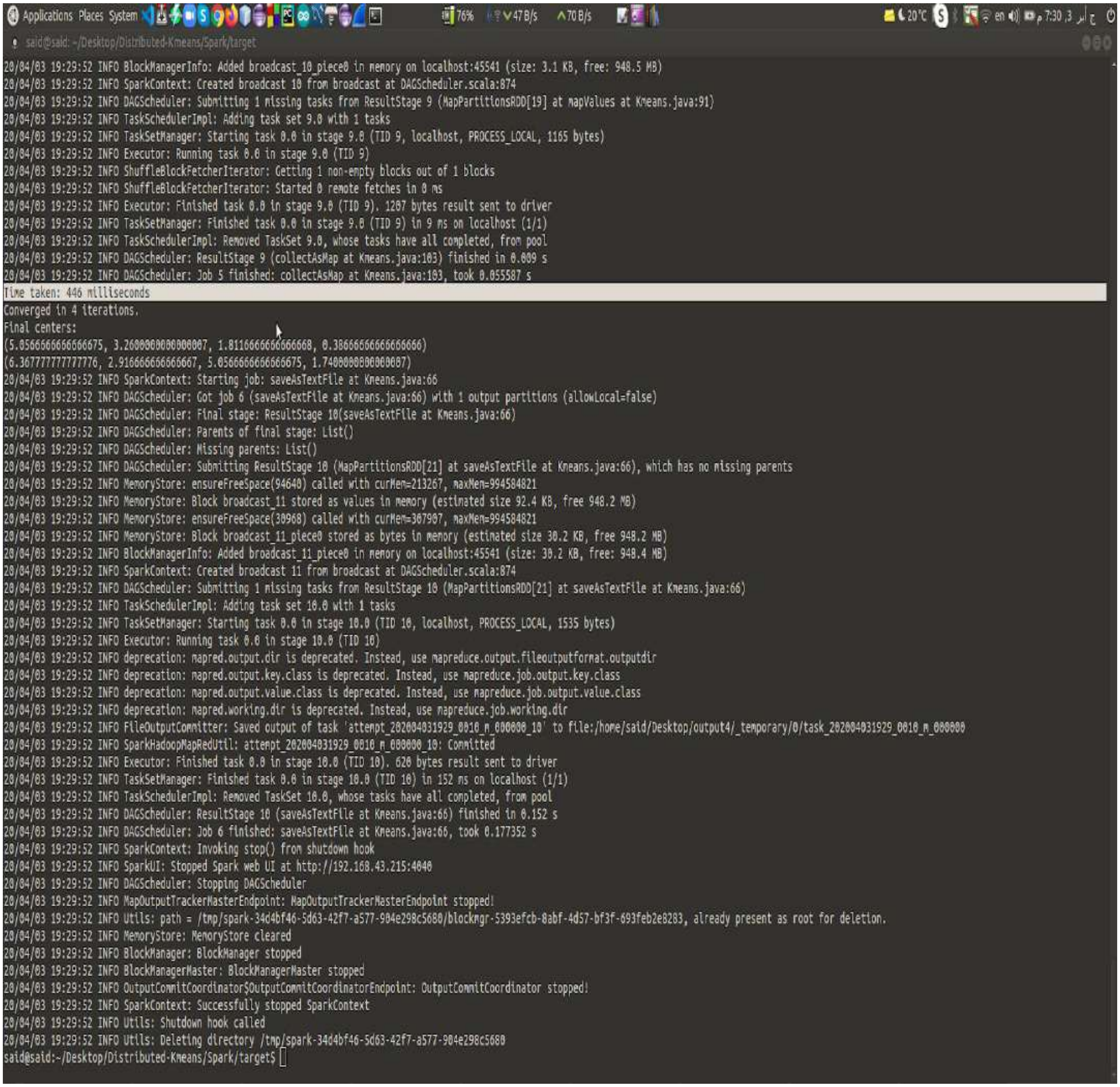
6 Sample Runs

<u>Un-Parallel K-means (Sequential):</u>



6 Sample Runs

<u>Parallel K-means (MapReduce) Spark:</u>



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 centroides 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.