University Grenoble Alpes

High performance computing for mathematical models

# Parallel implementation of k-means clustering algorithm

Amir Asarbaev

February 13, 2018

# Contents

# Abstract

The main idea of this research work was to get acquainted with the basic concepts of parallel programming, apply parallelism in k-means clustering algorithm, and thereby estimate the boost of performance.

The idea was realized on Python(v 2.7.14) program language. It was decided increase performance of one by using `open MPI` Message Passing Interface library project. Implementation open MPI reached through the **"mpi4py"**Python package.

At the beginning of this research work it was calculated Davies–Bouldin index for different number of clusters to estimate the best one (number of clusters). It was found the best number of cluster for each data sets that were presented here. Further there were written and shown the parallel and sequential k-means algorithms in Python language using "mpi4py" package. To validate these programs (to show that written algorithms work correctly) it was carried out comparing of clustering vectors (clustering vectors it's the vector that consists of the labels were obtained after clustering) between handwritten k-means functions and k-means from "sklearn.cluster" Python library. The comparison was done using Adjust rand index from "sklearn.metrics" Python library. After it was carried out series of experiments for parallel and sequential algorithms on data sets of different sizes. The result of testing showed us that parallel algorithm has more faster run-time of code than sequential, also It was obtained the best number of processes that gives us the best run-time of code and performance boost, respectively.

# Chapter 1

# Theory

## 1.1 Formulation the problem

Let $X$ be a sample data that consists of observations $x_i$ with size $p$: $x_i = (x_i^{(1)}, \ldots, x_i^{(p)})$. The goal of clustering is to assign each observation to one of groups $\mathcal{C}_1, \ldots, \mathcal{C}_K$ in a manner that objects from one cluster $\mathcal{C}_j$ are closer to each other than to observations from other groups. With this task one aims to understand better the data structure, particularly for interpretation.

## 1.2 Evaluation of the best number of clusters

Before starting to clusterize of given data set it's necessary identify some algorith that will allow us to estimate the best number of clusters. There are a lot of such algorithms:

- Davies–Bouldin index
- AffinityPropagation
- Calinski criterion
- Bayesian information criterion
- Average silhouette method

In this research work it was implemented just Davies-Bouldin index to estimate the best number of clusters.

### 1.2.1 Davies–Bouldin index

The Davies–Bouldin index (DBI) is a metric for evaluating clustering algorithms.

Let $R_{i,j}$ be a measure of how good the clustering scheme is. This measure, by definition has to account for $M_{i,j}$ the separation between the i-th and the j-th cluster, which ideally has to be as large as possible, and $S_i$, the within cluster scatter for cluster i, which has to be as low as possible. Hence the Davies–Bouldin index is defined as the ratio of $S_i$ and $M_i$:

$$R_{i,j} = \frac{S_i + S_j}{M_{i,j}} \tag{1.1}$$

With this formulation, the lower the value, the better the separation of the clusters and the 'tightness' inside the clusters.

This is used to define Di:

$$D_i = \max_{i \neq j} R_{i,j} \tag{1.2}$$

If N is the number of clusters:

$$DB = \frac{1}{N} \sum_{i=1}^{N} D_i \tag{1.3}$$

DB is called the Davies–Bouldin index. This is dependent both on the data as well as the algorithm. Di chooses the worst-case scenario, and this value is equal to Ri,j for the most similar cluster to cluster i. There could be many variations to this formulation, like choosing the average of the cluster similarity, weighted average and so on.

These conditions constrain the index so defined to be symmetric and non-negative. Due to the way it is defined, as a function of the ratio of the within cluster scatter, to the between cluster separation, a lower value will mean that the clustering is better. It happens to be the average similarity between each cluster and its most similar one, averaged over all the clusters, where the similarity is defined as Si above. This affirms the idea that no cluster has to be similar to another, and hence the best clustering scheme essentially minimizes the Davies–Bouldin index. This index thus defined is an average over all the i clusters, and hence a good measure of deciding how many clusters actually exists in the data is to plot it against the number of clusters it is calculated over. The number i for which this value is the lowest is a good measure of the number of clusters the data could be ideally classified into

## 1.3 k-means clustering algorithm

The k-means method is a widely used clustering technique that seeks to minimize the average squared distance between points in the same cluster. Although it offers no accuracy guarantees, its simplicity and speed are very appealing in practice. By augmenting k-means with a very simple, randomized seeding technique, we obtain an algorithm that is $\theta(\log k)$ -competitive with the optimal clustering. Preliminary experiments show that our augmentation improves both the speed and the accuracy of k-means, often quite dramatically.

### 1.3.1 The algorithm

The k-means method is a simple and fast algorithm that attempts to locally improve an arbitrary k-means clustering. It works as follows. A formal definition of the algorithm is the following:

1. Arbitrarily choose k initial centers $C = \{c_1, c_2, ..., c_k\}$
2. For each $i \in \{1, ..., k\}$, set the cluster $C_i$ to be the set of points in X that are closer to $c_i$ than to any $c_j$ with $j \neq i$.
3. For each $1 \leq i \leq k$, set $c_i = \frac{1}{|Ci|} \sum_{x \in C_i} x$, i.e., the center of mass of the pointsin Ci.
4. Repeat Steps 2 and 3 until the clusters $C_i$ and the centers $c_i$ do not change anymore. The partition of X is the set of clusters $C_1, C_2, ..., C_k$.

It is standard practice to choose the initial centers uniformly at random from X . For Step 2, ties may be broken arbitrarily, as long as the method is consistent.

Note that the algorithm might encounter two possible “degenerate” situations: the first one is when no points are assigned to a center, and in this case that center is removed and we will obtain a partition with fewer than k clusters. The other degeneracy is when a point is equally close to more than one center, and in this case the tie is broken arbitrarily.

## 1.4 Parallelism with OpenMPI

As was mentioned above in order to achieve parallelism it was used "mpi4py" python library that allows us to separate tasks by processes and run the program code on several processes simultaneously.

Classifications of parallel programming models can be divided broadly into two areas: process interaction and problem decomposition.

- **Problem of interaction:**

  Process interaction relates to the mechanisms by which parallel processes are able to communicate with each other. The most common forms of interaction are shared memory and message passing, but interaction can also be implicit (invisible to the programmer). `In case of OpenMPI it was used message passing form.`

- **Problem decomposition:**

  A parallel program is composed of simultaneously executing processes. Problem decomposition relates to the way in which the constituent processes are formulated.

A task-parallel model focuses on processes, or threads of execution. These processes will often be behaviourally distinct, which emphasises the need for communication. Task parallelism is a natural way to express message-passing communication.

### 1.4.1 Message-passing communication: Master and workers

The MPI interface is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to nodes/servers/-computer instances). This research work was carried out on ordinary university computer, so the set of processes weren't mapped to nodes/servers/computer instances.

MPI programs always work with processes, but programmers commonly refer to the processes as processors. Typically, for maximum performance, each CPU (or core in a multi-core machine) will be assigned just a single process. This assignment happens at runtime through the agent that starts the MPI program, normally called mpirun or mpiexec.

MPI library functions include, but are not limited to, point-to-point rendezvous-type send/receive operations, choosing between a Cartesian or graph-like logical process topology, exchanging data between process pairs (send/receive operations), combining partial results of computations (gather and reduce operations), synchronizing nodes (barrier operation) as well as obtaining network-related information such as the number of processes in the computing session, current processor identity that a process is mapped to, neighboring processes accessible in a logical topology, and so on.

A number of important MPI functions involve communication between two specific processes. A popular example is MPI_Send, which allows one specified process to send a message to a second specified process. Point-to-point operations, as these are called, are particularly useful in patterned or irregular communication, for example, a data-parallel architecture in which each processor routinely swaps regions of data with specific other processors between calculation steps, or a `master-worker architecture` in which the `master` sends new task data to `a worker` whenever the prior task is completed.

## 1.5 Adjusted rand index

In case when was implemented some function that was written by somebody without using any library function. It's very necessary to validate this function. Related clustering algorithms, Adjusted rand index (hereinafter ARI) allows us to compare outputs of each clustering function. In the case of clustering the outputs will be vectors with labels.

Given a set S of n elements, and two groupings or partitions (e.g. clusterings) of these elements, namely $X = \{X_1, X_2, \ldots, X_r\}$ and $Y = \{Y_1, Y_2, \ldots, Y_s\}$, the overlap between X and Y can be summarized in a contingency table $[n_{ij}]$ where each entry $n_{ij}$ denotes the number of objects in common between $X_i$ and $Y_j$ $Y_j$ : $n_{ij} = |X_i \cap Y_j|$

| $X \backslash Y$ | $Y_1$ | $Y_2$ | $\ldots$ | $Y_s$ | Sums |
|---|---|---|---|---|---|
| $X_1$ | $n_{11}$ | $n_{12}$ | $\ldots$ | $n_{1s}$ | $a_1$ |
| $X_2$ | $n_{21}$ | $n_{22}$ | $\ldots$ | $n_{2s}$ | $a_2$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ | $\vdots$ |
| $X_r$ | $n_{r1}$ | $n_{r2}$ | $\ldots$ | $n_{rs}$ | $a_r$ |
| Sums | $b_1$ | $b_2$ | $\ldots$ | $b_s$ | |

$$\overbrace{ARI}^{\text{Adjusted Index}} = \frac{\overbrace{\sum_{ij}\binom{n_{ij}}{2}}^{\text{Index}} - \overbrace{[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}]/\binom{n}{2}}^{\text{Expected Index}}}{\underbrace{\frac{1}{2}[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}]}_{\text{Max Index}} - \underbrace{[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}]/\binom{n}{2}}_{\text{Expected Index}}}$$

Figure 1.1: Evaluation of the adjusted rand index.

The result of validation were presented in 3.0.2 section.

# Chapter 2

# Results and Data Analysis

In this chapter was presented result of validation (to justify that written program code works correctly), result of clustering (to justify that obtained results of clustering correspond to the problem posed in the theory of the algorithm) and performance evaluation of parallel and sequential k-means on datasets of different sizes.

It was carried out series of experiments for evaluating performance of parallel and sequential k-means algorithms. For these experiments it was chosen 2 different datasets:

- **Data set of daily electricity load consumption.** It contains 70 samples and 48 features that corresponds to measurements of electricity consumption 2 times per hour during a day (i.e. $48 = 2 \cdot 24$).
- **3D Road Network (North Jutland, Denmark) Data Set** This dataset was constructed by adding elevation information to a 2D road network in North Jutland, Denmark (covering a region of 185 x 135 $km^2$). IT contains 434874 samples and 4 features. Feature information:
  1. OSM_ID: OpenStreetMap ID for each road segment or edge in the graph.
  2. LONGITUDE: Web Mercaptor (Google format) longitude
  3. LATITUDE: Web Mercaptor (Google format) latitude
  4. ALTITUDE: Height in meters.

The first data set related electricity load consumption was used as debug data set. The second data set with elevation information was divided on two datasets of different sizes: first dataset size was $10^4$ and second was $10^5$ number of rows.

### 2.0.1 Results of clustering

Before clustering of given data sets it was calculated Davies-Bouldin index allowed us to find the best number of clusters.

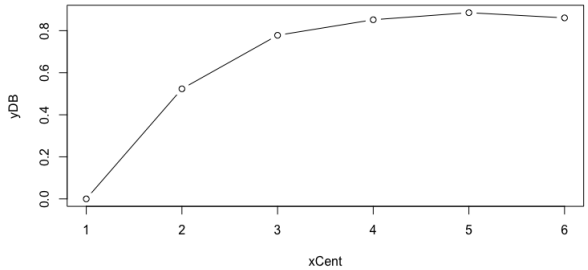


Figure 2.1: Davies-Bouldon index for debug dataset

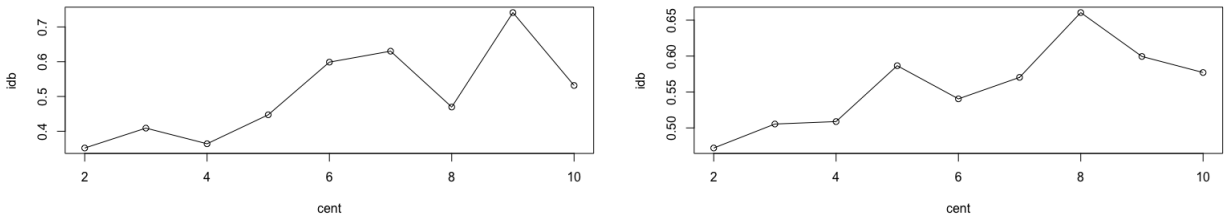


Figure 2.2: Davies-Bouldon index for debug dataset

From the definition of Davies-Bouldin index the best number of clusters corresponds to lowest DB index. In both case DBi algorithm provide **k=2** as the best number of clusters. To demonstrate correct result of parallel clustering of debug dataset it was built 1-D plot with average means of electricity consumption for each sample and displayed there found centroids (k= 2, 3, 4, 5).

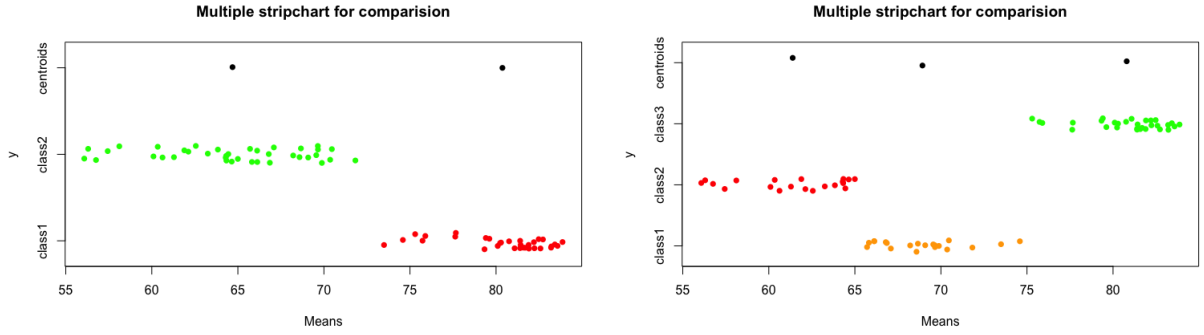


Figure 2.3: 1D plot with average means of electricity consumption for k= 2, 3

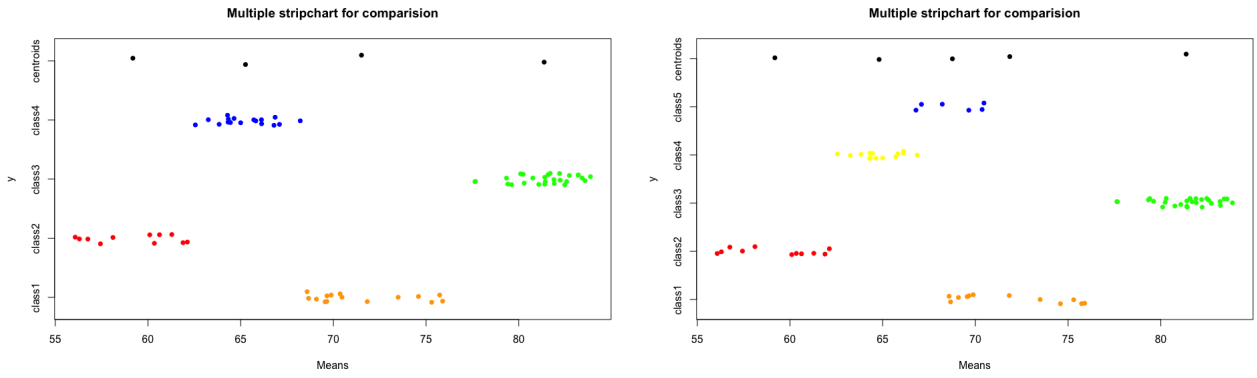


Figure 2.4: 1D plot with average means of electricity consumption for k= 4, 5

Since the main idea of k-means method is finding center of mass for each cluster,so it was not surprised that after deployment of parallel k-means method to changed signal dataset, It was divided into classes that correspond to the average values of electricity consumption. It means in the case of k = 2, we obtained 2 classes corresponding to the average value for low and high electricity consumption.

Applying parallel k-means to the prepared debug dataset, it was obtained the same tendency that was in previous results.

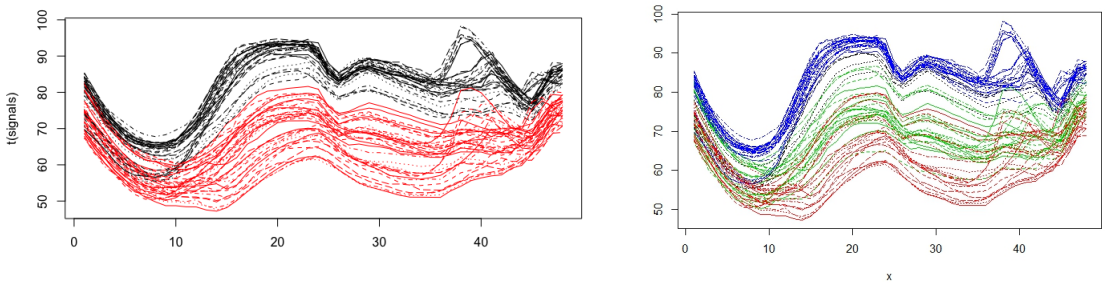


Figure 2.5: Results of k-means clustering algorithm for k = 2, 3

### 2.0.2 Validation

To confirm that written parallel and sequential program codes work correctly (in the sense of the problem of clustering)it was decided to calculate Adjusted Rand index (ARI) that allows us to understand the level of similarity between written program codes and k-means function from "sklearn.cluster" python library. ARI allowed us to compare cluster vectors by evaluating matches of labels in these vectors.

It was calculated ARI between parallel k-means and sklearn k-means for $k = 2, 3, 4$. It is equal **ARI = 100%**, which indicates absolute similarity. The same results were obtained for sequential k-means.

### 2.0.3 Parallel implementation

Before proceeding with the description of parallel implementation let's look at how the sequential algorithm works step by step to describe in further in which parts of sequential algorithm was applied parallelism and how it changes sequential algorithm.

#### Sequential algorithm

1. Choose randomly i samples from data set and identify these samples as centroids.
2. Calculate Euclidean distance between each sample and each centroid. The results were written in distance matrix. The length of dist matrix is equal length of data set and width is equal number of centroids.
3. Determine which centroid corresponds to the minimum distance from each sample. Thus, we determine which cluster each sample belongs to.
4. Recalculate means of each centroids as a ratio between the sum of the means of the samples belonging to given centroid and the number of samples belonging to the cluster of given centroid
5. compare means of new centroids and old centroids and if they are equal it was found the best means of centroids and clustering will finished, otherwise the new values are equal to the value of the old centroids and it repeating 2,3,4,5 steps until convergence (new cent = old cent).

#### Parallel algorithm

In case of parallel programming it's necessary to keep control on each process. All processes divided on MASTER(rank = 0) process and WORKERS processes (rank !=0). So, the goal of parallel programming divide the task by processes.

1. The MASTER process read a data set.
2. Randomly choose samples from this DS and initialize these samples as centroids. Initialize a centroid matrix.
3. Divide DS into equal parts. The number of these equal parts should be equal number of processes.
4. This equal parts are scattered between each process.
5. Broadcast the centroid matrix to each process.
6. In each process there is part of data set and centroid matrix. So, it is able to carry out the step 2, 3 from sequential algorithm: calculate distance matrix and determine clustering vectors in each process.
7. Gather the clustering vectors from each process in the same order in which the data sets were sent
8. Calculate which clusters were faced and with what frequency in each process. This step is necessary for furthering recalculation of centroid means in each process.
9. Broadcast the result of step 8 to all processes and recalculate the means of centroid in each process how it was done in step 4 of sequential algorithm.
10. Gather the means of centroid from each process by summing them. It means for for centroid which is responsible for 1st cluster it sums all means of 1st centroid from each processes and it does for each centroid.
11. Broadcast the result from previous step to all process and repeat step 5 from sequential algorithm, i.e compare the means of old centroids with new centroids.

**The idea of current parallel programming is divide initial data set on equal parts, scatter these parts to each process and realize sequential algorithm there. After, the parallel process will be stopped for results sharing only on steps of clustering vectors calculation and during comparison.**

**Algorithm 1** Parallel k-means clustering algorithm

**MASTER (rank = 0)**
**Inputs:**
    np = input(" number of processes ")
    k = input(" number of clusters ")
    path = input(" path to file with data set")
**Initialize:**
    df ← read_csv(path, sep(","))
    **for** $i = 1$ to k **do**
        Initial[i] ← append random sample from df (data set)
    **end for**
    df = list[$df_1$+$df_2$+...+$df_{np}$] #`divide data set into np parts, where np is number of processes.`
**ALL RANKS**
df=comm.scatter(list(df),root=0) #`scattering parts of dataset on processes`
k=comm.bcast(k, root=0) #`broadcast number of clusters from rank=0 to other ranks`
Initial=comm.bcast( initial, root = 0) # `broadcast centroid matrix from rank=0 to other ranks`
flag = True
**while** (flag == true) **do**
    **for** $j = 0$ to k **do**
        **for** $i = 0$ to len(df) **do**
            dist[i][j]= euclidean distance (initial[j],data[i])
        **end for**
    **end for**
    **for** $i = 0$ to len(dist) **do**
        clusters.append(np.argmin(dist[i]) # `identify the cluster label of each sample for each process`
    **end for**
    Q_clusts= collections.Counter (clusters) #`identify what is the labels was faced in each process and their frequency`
    totcounter = comm.allreduce(Q_clusts) #`gather Q_clusts from each process and broadcast the result back`
    cluster=comm.gather(clusters,root=0)#`gather clusters from each processes and send it in MASTER`
    **for** $m = 0$ to k **do**:
        indices=[i for i, j in enumerate(clusters) if j == m] #`identify indexes of samples that belong to 1st,2nd,...k cluster`
        cetroids[m]=$\frac{\sum df[i] \ for \ i \ in \ indices}{totcounter[m]}$ #`recalculate means of centroids in each process`
    **end for**
    centorid=comm.allreduce(centroid,MPI.SUM) #`summing all means of each centroid from each process and broadcast new centroid matrix to all processes`
    **if** centroid == initial **then**
        flag = False
    **else**
        initial = centroid
    **end if**
**end while**

### 2.0.4 Performance evaluation

In this chapter it was presented the results of carried out experiments for parallel and sequential k-means algorithm. In each experiments it was measured run-times (in seconds) of codes. As the result, there were built graphs of run-times of parallel and sequential k-means clustering algorithms against different numbers of processes and clusters. All results presented below (table 3.1, table 3.2, table 3.3).

1. **Electricity load consumption dataset(Debug dataset, 70 samples)** Below presented the table with runtime of each experiment. There were carried out experiment for number of clusters **k={2,3}** and for number of processes **-np={2,...,8}**. It was noticed the best number of performance for given data set reached on **3 processes** and it's equal **200%** for k = 2 and **255%** for k = 3. Also it was registered anomalous peak of runtime for k = 3 and -np = 5.

| Number of processes / Number of clusters | -np 2 | -np 3 | -np 4 | -np 5 | -np 6 | -np 7 | -np 8 | sequential k-means |
|---|---|---|---|---|---|---|---|---|
| k=2 | 0.0157 | 0.015 | 0.0156 | 0.022 | 0.0213 | 0.023 | 0.023 | 0.03 |
| k=3 | 0.021 | 0.020 | 0.0213 | 0.30 | 0.029 | 0.030 | 0.0316 | 0.051 |

Table 2.1: The run-times (in seconds) of parallel and sequential k-means clustering algorithm on debug dataset

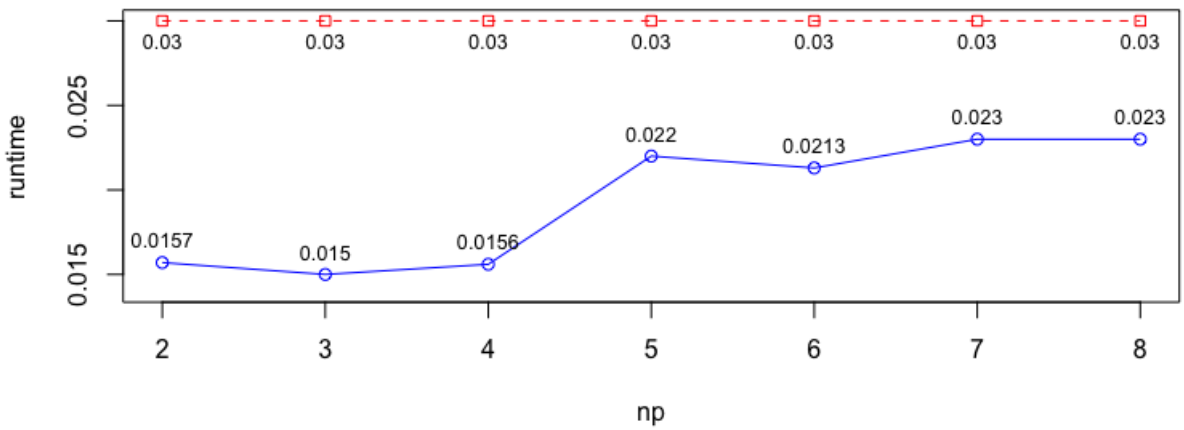


Figure 2.6: The graphs of run-times of parallel and sequential algorithms against different number of processes and clusters (k= 2)

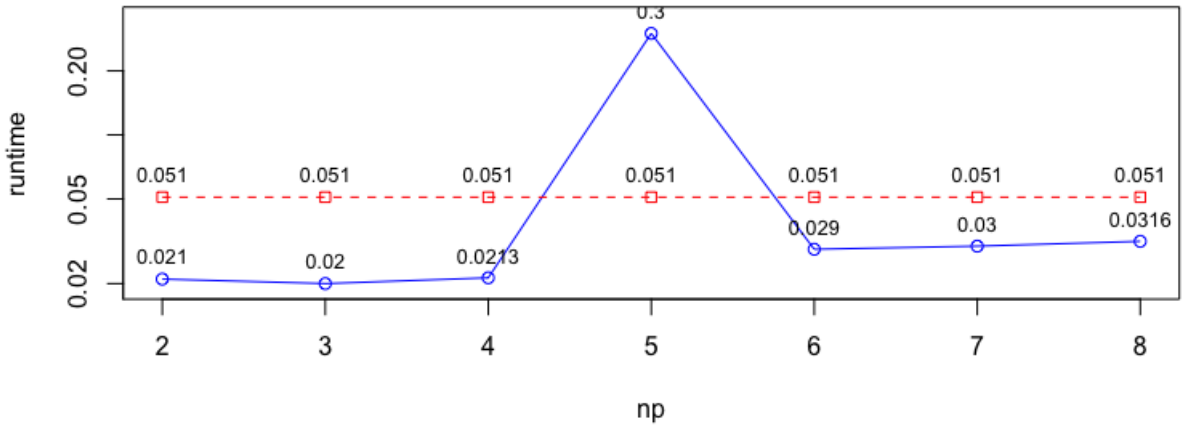


Figure 2.7: The graphs of run-times of parallel and sequential algorithms against different number of processes and clusters (k= 3)

2. **3D Road Network dataset ($10^4$ samples)**

   Below presented the table with runtime of each experiment. There were carried out experiment for number of clusters **k={2,3}** and for number of processes **-np={2,...,8}**. It was noticed the best number of performance for given data set reached on **4 processes** and it's equal **266%** for k = 2 and **280%** for k = 3. Also it was registered anomalous peak of runtime for k = 3 and -np = 5.

| Number of processes / Number of clusters | -np 2 | -np 3 | -np 4 | -np 5 | -np 6 | -np 7 | -np 8 | sequential k-means |
|---|---|---|---|---|---|---|---|---|
| k=2 | 3.515 | 2.623 | 2.148 | 3.046 | 2.81 | 2.55 | 2.62 | 5.727 |
| k=3 | 3.419 | 2.541 | 2.11 | 2.990 | 2.67 | 2.48 | 2.55 | 5.924 |

Table 2.2: The run-times (in seconds) of parallel and sequential k-means clustering algorithm on 3D Road Network dataset with $10^4$ samples

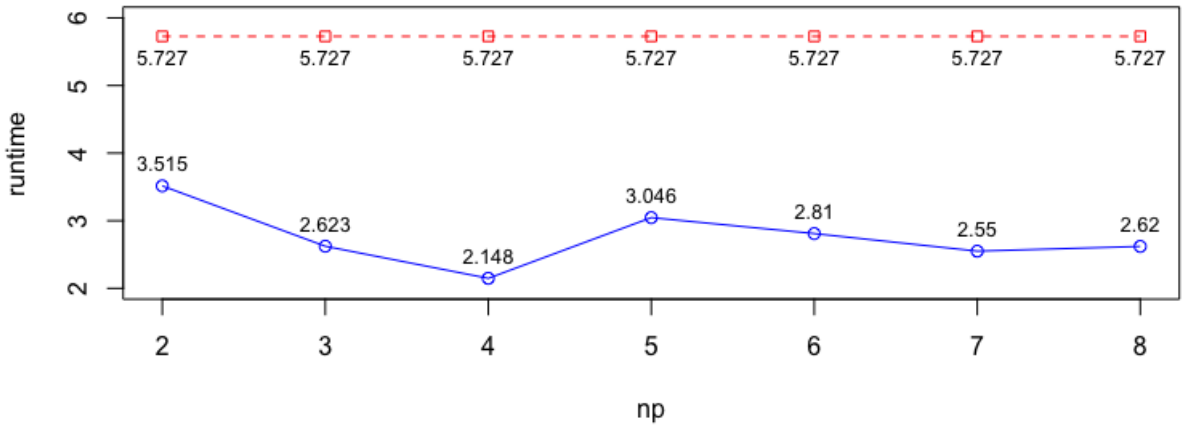


Figure 2.8: The graphs of run-times of parallel and sequential algorithms against different number of processes and clusters (k= 2)

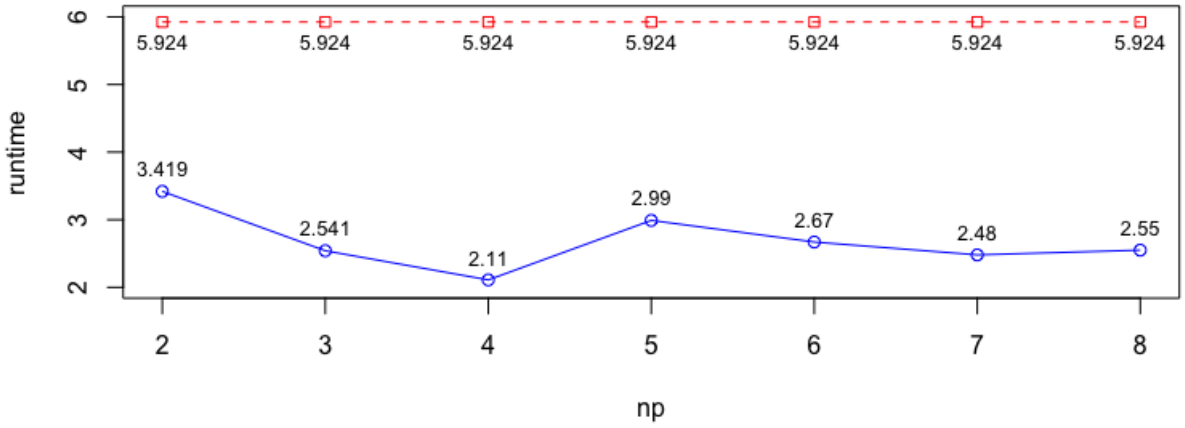


Figure 2.9: The graphs of run-times of parallel and sequential algorithms against different number of processes and clusters (k= 3)

3. **3D Road Network dataset** ($10^5$ **samples**)

   Below presented the table with runtime of each experiment. There were carried out experiment for number of clusters **k={2,3}** and for number of processes **-np={2,...,8}**. It was noticed the best number of performance for given data set reached on **8 processes** and it's equal **278%** for k = 2 and **325%** for k = 3. Also it was registered anomalous peak of runtime for k = 3 and -np = 5.

| Number of processes / Number of clusters | -np 2 | -np 3 | -np 4 | -np 5 | -np 6 | -np 7 | -np 8 | sequential k-means |
|---|---|---|---|---|---|---|---|---|
| k=2 | 24.832 | 17.111 | 14.278 | 19.166 | 16.453 | 14.973 | 14.05 | 39.082 |
| k=3 | 49.176 | 35.115 | 27.615 | 36.412 | 31.731 | 28.244 | 25.425 | 82.736 |

Table 2.3: The run-times (in seconds) of parallel and sequential k-means clustering algorithm on 3D Road Network dataset with $10^5$ samples

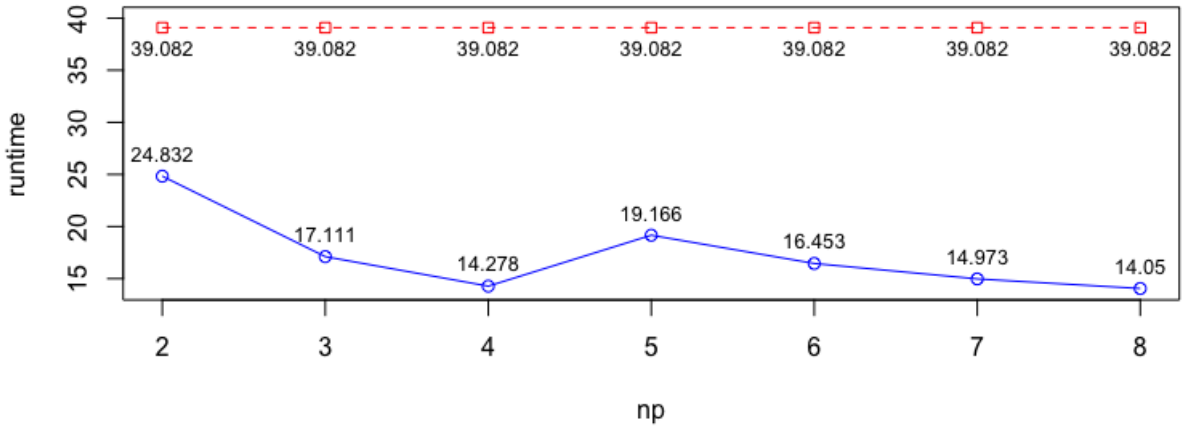


Figure 2.10: The graphs of run-times of parallel and sequential algorithms against different number of processes and clusters (k= 2)

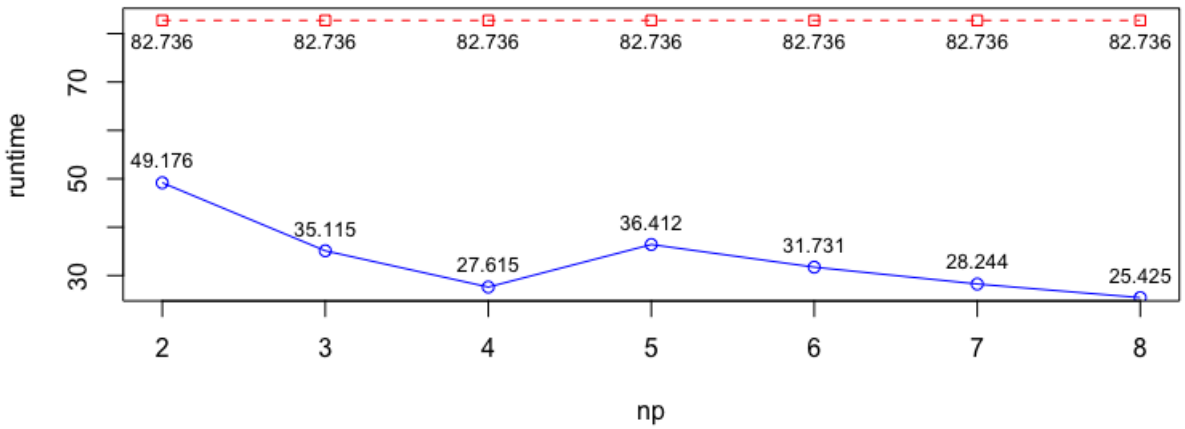


Figure 2.11: The graphs of run-times of parallel and sequential algorithms against different number of processes and clusters (k= 3)

4. **Results of experiments of parallel and sequential algorithm (the number of processes = 4, the number of clusters = 2, 3)** After it was discovered efficient number of processes there were carried out experiments for measuring runtime of algorithms (table 3.4) and provide graphs of speed up (figure 3.12).

| Number of processes / Number of clusters | -np 4 | sequential k-means | Dataset size |
|---|---|---|---|
| k=2 | 0.547 | 1.069 | $10^3$ |
| k=3 | 0.568 | 1.128 | $10^3$ |
| k=2 | 0.686 | 1.533 | $2 \cdot 10^3$ |
| k=3 | 0.796 | 1.827 | $2 \cdot 10^3$ |
| k=2 | 0.773 | 1.739 | $3 \cdot 10^3$ |
| k=3 | 0.902 | 2.206 | $3 \cdot 10^3$ |
| k=2 | 0.913 | 1.980 | $4 \cdot 10^3$ |
| k=3 | 1.145 | 2.893 | $4 \cdot 10^3$ |
| k=2 | 1.109 | 2.652 | $5 \cdot 10^3$ |
| k=3 | 1.158 | 2.875 | $5 \cdot 10^3$ |
| k=2 | 1.115 | 2.799 | $6 \cdot 10^3$ |
| k=3 | 1.534 | 4.087 | $6 \cdot 10^3$ |
| k=2 | 1.919 | 4.907 | $7 \cdot 10^3$ |
| k=3 | 1.455 | 3.701 | $7 \cdot 10^3$ |
| k=2 | 2.095 | 5.666 | $8 \cdot 10^3$ |
| k=3 | 1.975 | 5.420 | $8 \cdot 10^3$ |
| k=2 | 1.907 | 4.940 | $9 \cdot 10^3$ |
| k=3 | 2.905 | 7.598 | $9 \cdot 10^3$ |
| k=2 | 2.157 | 5.731 | $10^4$ |
| k=3 | 2.098 | 5.857 | $10^4$ |
| k=2 | 8.351 | 23.131 | $4 \cdot 10^4$ |
| k=3 | 11.283 | 33.589 | $4 \cdot 10^4$ |
| k=2 | 8.437 | 23.463 | $5 \cdot 10^4$ |
| k=3 | 12.220 | 36.457 | $5 \cdot 10^4$ |
| k=2 | 14.278 | 39.082 | $10^5$ |
| k=3 | 27.615 | 82.736 | $10^5$ |

Table 2.4: The run-times (in seconds) of parallel and sequential k-means clustering algorithm on 3D Road Network dataset for different sizes of samples

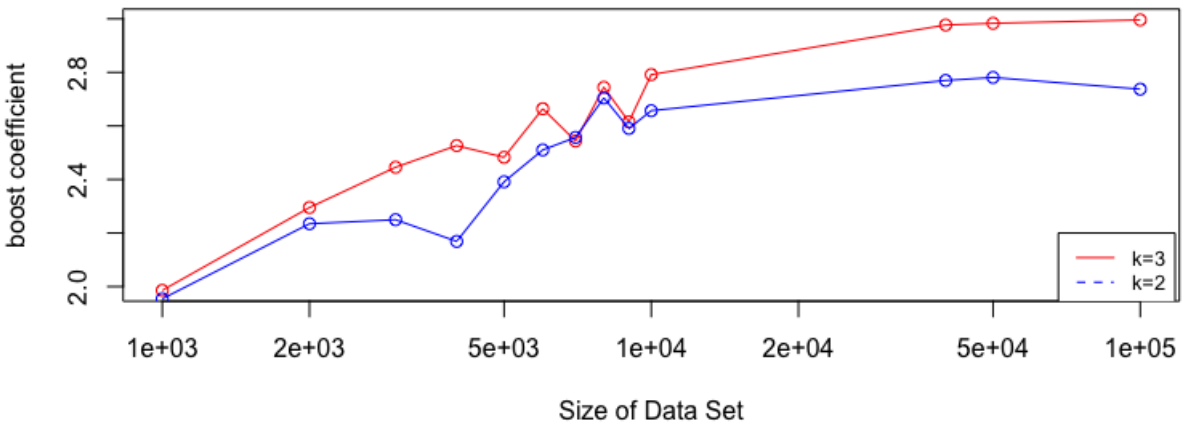


Figure 2.12: The graphs of boost coefficient against different sizes of 3D Road Network data set(k= 2, 3)

Here, boost coefficient means rate between runtime of sequential and parallel algorithms. As it was shown performance boost coefficient has logarithmic behavior and has upper bound.

# Chapter 3

# Conclusion

The goal of this project was to be familiar with general points of parallel programing and implement ones to k-means algorithm to reach the boost of performance. The proposed aim was successful achieved and results of boosting were presented. It was noticed that maximum performance boost was achieved by the number of processes that are equal the number of cores,i.e. number of processes = number of cores = 4, also in case of "large" dataset ($10^5$ samples) a good results of boosting were achieved by 8 processes. Also, it was noticed that performance boost coefficient growth has logarithmic behavior and has upper bound.

However, these results aren't optimal because for carrying out runtime experiments it was used ordinary computer, whereas openMPI directed on work of node clusters. Accordingly, the runtime will be increase because it will be spend more time on data transaction between nodes. But in order to get acquainted with the basics of parallel programming this was enough. Also, during this research work it was discovered limitations on the number of processes that could be used. MacBook Air with 2 physical cores doesn't not suitable for effective parallelism. It doesn't allow use Degraded mode (oversubscribed mode), when the number of processes exceed the number of cores. But in case of 2 cores there are only master and 1 worker. So, all experiments were carried out on ordinary university computer with 4 cores and Linux OS.

**Summing up:**

1. It was find the best number of clusters for all given datasets using Davies-Bouldin index.
2. There were written parallel and sequential k-means algorithm and they were implemented to the already found number of clusters.
3. It was carried out validation using Adjust Rand index, i.e. obtained results were compared with results of k-means library functions.
4. There were carried out experiments for parallel and sequential algorithms on given data sets of different sizes. It was done to estimate boost of performance.

# Bibliography


[1] Andrea Vattani. *k-means Requires Exponentially Many Iterations Even in the Plane*. Discrete Comput Geom (2011).

[2] David Arthur,Sergei Vassilvitskii†. *k-means++: The Advantages of Careful Seeding*. http://theory.stanford.edu/ sergei/papers/kMeansPP-soda.pdf

[3] Ishan Handa. *parallel-implementation-of-kmeans*.
https://github.com/ishanhan/parallel-implementation-of-kmeans/blob/master/mpi_kmeans.py

[4] *Parallel programming model*. https://en.wikipedia.org/wiki/Parallel_programming_model

[5] *Message Passing Interface*. https://en.wikipedia.org/wiki/Message_Passing_Interface

[6] Michael Jay Quinn. *Parallel Programming in C with Mpi and Openmp* . London : McGraw-Hill Higher Education, c2004.

[7] Christophe Picard. *Chapter 3: Parallel Patterns, MPI and some examples*. http://chamilo.grenoble-inp.fr/courses/ENSIMAGWMM9MO16/document/Resources/Lectures/3-chapter.pdf

# Chapter 4

# Appendix A

**The code of sequential k-means clustering algorithm.**

```
import math
import csv
import time
import numpy as np
import collections
from sklearn.cluster import KMeans
from sklearn.metrics.cluster import adjusted_rand_score


#=====================================================Euclidian distance=====================
def eucl_distance(point_one, point_two):                    #define function to measure Euclid
        if(len(point_one) != len(point_two)):                   #check for an error (len(1st p
                raise Exception("Error: non comparable points")   #check for an error

        sum_diff = 0.0                                                  #set variable sum_di
        for i in range(len(point_one)):                                 #iterating take diff
                diff = pow((float(point_one[i]) - float(point_two[i])), 2) #and square this
                sum_diff += diff                                            #As the result st
        final = math.sqrt(sum_diff)                                     #sqrt from sum_diff
        return final
#=====================================================Devide data set to further scattering==

global dimensions, num_clusters, num_points,dimensions,data,flag

                                                                                #turn on a
#=====================================================reading and preparing data set==========
print("Enter the number of clusters you want to make: ")
num_clusters = input()
num_clusters = int(num_clusters)
start_time = time.time()
with open('3D_spatial_network.csv','rb') as f:
    reader = csv.reader(f)
    data = list(reader)

data.pop(0)
for i in range (len(data)):
    data[i].pop(0)
data=np.array(data).astype(np.float)
```

```
data=data[0:10000]
#Print(data)
kmeans = KMeans(n_clusters=num_clusters, random_state=0).fit(data).labels_
#         print('data',[ data[i] for i in [indices] ])
#         data=np.array ([[2,10],[2,5],[8,4],[5,8],[7,5],[6,4],[1,2],[4,9]])
#=============================================================================

#================================================Initialize centroids matrix==========
initial=[]
for i in xrange(num_clusters):
    initial.append(data[i])
initial=np.vstack(initial)
#=============================================================================

num_points = len(data)                                          #number of rows
dimensions = len(data[0])                                       #number of columns
#chunks = [ [] for _ in range(size) ]

#for i, chunk in enumerate(data):
#         chunks[i % size].append(chunk)

#=============================================================================

flag= True
while flag==True:

        cluster=[]
        #print str(rank) + ': ' + str(data)
        #====================================Calculating dist matrix in each process========
        dist =np.zeros((len(data),len(initial)))


        for j in range(len(initial)):
                for i in range(len(data)):
                        dist[i][j]=np.linalg.norm(initial[j]-data[i])
        #print('dist',dist)
        #====================================Initilize lable for each sample in each process
        for i in range (len(dist)):
                cluster.append(np.argmin(dist[i])+1)                        #find column ind
        #print('clust vect',cluster)
        #====================================Calculating the number of samples in each clust
        Q_clusts=collections.Counter(cluster)
        #Q_clusts=np.array((collections.Counter(clusters).keys(),collections.Counter(cluste

        #=============================================================================
        #====================================From each worker we gather cluster vector and
        #=============================================================================
        #if rank==0:
        #         cluster=[item for sublist in cluster for item in sublist]
        #         data = [item.tolist() for item in data]
        #         data=[item for sublist in data for item in sublist]
        #         print(data)
```

```
#        print(cluster)

centroid=np.zeros((len(initial),len(initial[0])))
for k in range (1,num_clusters+1):
        indices = [i for i, j in enumerate(cluster) if j == k]
        #print('ind',indices)
        #print(k,  [data[i] for i in indices] )
        #print('sum',np.sum([data[i] for i in indices], axis=0 ))
        #print('div',np.divide((np.sum([data[i] for i in indices], axis=0)).astype(
        centroid[k-1]=np.divide((np.sum([data[i] for i in indices], axis=0)).astype


#print('centroids',centroids)
#print ('initial', initial)
#print('centroid',centroid)


if np.all(centroid==initial):
        flag=False

        print ("Execution time %s seconds" % (time.time() - start_time))
else:
        #print ('initial after', initial)
        initial= centroid

#print('final clust vect',cluster)
#print('libkms clust vect',kmeans)
print('adjusted_rand_score',adjusted_rand_score(kmeans,cluster))
```

# Chapter 5

# Appendix B

**The code of parallel k-means clustering algorithm.**

```
import math
import csv
import time
import numpy as np
import collections
from mpi4py import MPI
from sklearn.cluster import KMeans
from sklearn.metrics.cluster import adjusted_rand_score



#=====================================================Devide data set to further scattering==
def chunkIt(seq, num):
    avg = len(seq) / float(num)
    out = []
    last = 0.0

    while last < len(seq):
        out.append(seq[int(last):int(last + avg)])
        last += avg
    return out
#=====================================================Count lables for recalculating means of
def addCounter(counter1, counter2, datatype):
    for item in counter2:
        counter1[item] += counter2[item]
    return counter1

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
global dimensions, num_clusters, num_points,dimensions,data,flag
num_clusters=0


if rank==0:
        #=================================================reading and preparing data set======

        print("Enter the number of clusters you want to make: ")
```

```
        num_clusters = input()
        num_clusters = int(num_clusters)
        start_time = time.time()

        with open('3D_spatial_network.csv','rb') as f:
                reader = csv.reader(f)
                data = list(reader)

        data.pop(0)
        for i in range (len(data)):
                data[i].pop(0)
        data=data[0:10000]
        data=np.array(data).astype(np.float)

        kmeans = KMeans(n_clusters=num_clusters, random_state=0).fit(data).labels_
#         print('data',[ data[i] for i in [indices] ])
#         data=np.array ([[2,10],[2,5],[8,4],[5,8],[7,5],[6,4],[1,2],[4,9]])
        #=============================================================================

        #================================================Initialize centroids matrix=======
        initial=[]
        for i in xrange(num_clusters):
                initial.append(data[i])
        initial=np.vstack(initial)
        #=============================================================================

        num_points = len(data)                                          #number of rows
        dimensions = len(data[0])                                       #number of columns
        #chunks = [ [] for _ in range(size) ]

        #for i, chunk in enumerate(data):
        #        chunks[i % size].append(chunk)
        chunks=chunkIt(data,size)
        #=============================================================================
else:
        chunks = None
        initial = None
        data = None
        dimensions = None
        num_points = None
        cluster= None
        Q_clust= None
        num_clusters= None
        centroid=None
        kmeans= None
        start_time=None
        #=============================================================================
start_time=comm.bcast(start_time,root=0)
data=comm.scatter(chunks, root=0)
num_clusters=comm.bcast(num_clusters,root=0)
initial=comm.bcast(initial, root = 0)                                   #sen
flag= True
```

```
while flag==True:
        clusters=[]
        cluster=[]
        #print str(rank) + ': ' + str(data)
        #====================================Calculating dist matrix in each process========
        dist =np.zeros((len(data),len(initial)))


        for j in range(len(initial)):
                for i in range(len(data)):
                        dist[i][j]=np.linalg.norm(initial[j]-data[i])
        #print('rank',rank,dist)
        #====================================Initilize lable for each sample in each process
        for i in range (len(dist)):
                clusters.append(np.argmin(dist[i])+1)                        #find column in
        #print(clusters)
        #====================================Calculating the number of samples in each clust
        Q_clusts=collections.Counter(clusters)
        #Q_clusts=np.array((collections.Counter(clusters).keys(),collections.Counter(cluste
        #====================================Summing the number of samples for each cluster=
        counterSumOp = MPI.Op.Create(addCounter, commute=True)

        totcounter = comm.allreduce(Q_clusts, op=counterSumOp)
        comm.Barrier()
        #print ('Q',Q_clusts)
        #print('T',totcounter)
        #================================================================================
        #====================================From each worker we gather cluster vector and
        cluster=comm.gather(clusters, root=0)
        #data=comm.gather(data,root=0)
        comm.Barrier()
        #print('cl1',cluster)
        #================================================================================
        if rank==0:
                cluster=[item for sublist in cluster for item in sublist]
        #        data = [item.tolist() for item in data]
        #        data=[item for sublist in data for item in sublist]
        #        print(data)
        #        print(cluster)

        centroids=np.zeros((len(initial),len(initial[0])))
        for k in range (1,num_clusters+1):
                indices = [i for i, j in enumerate(clusters) if j == k]
                #print('ind',indices)
                #print(k,  [data[i] for i in indices] )
                #print('sum',np.sum([data[i] for i in indices], axis=0 ))
                #print('div',np.divide((np.sum([data[i] for i in indices], axis=0)).astype(
                centroids[k-1]=np.divide((np.sum([data[i] for i in indices], axis=0)).astype

        centroid=comm.allreduce(centroids,MPI.SUM)
        comm.Barrier()
        #print('centroids',centroids)
```

```
        #print ('initial', initial)
        #print('centroid',centroid)


        if np.all(centroid==initial):
                flag=False
                print ("Execution time %s seconds" % (time.time() - start_time))

        else:
        #        print ('initial after', initial)
                initial= centroid
        comm.Barrier()


if rank==0:
#        print('final clust vect',cluster)
#        print('libkms clust vect',kmeans)
        print('adjusted_rand_score',adjusted_rand_score(kmeans,cluster))
```