Lab Course: Distributed Data Analytics
Exercise Sheet 4
Group 2 - Monday

Submitted by: Sruthy Annie Santhosh, 312213

Topic:Complex Data Lab: : K-means clustering in a Distributed Setting

- My System

  Hardware and Software SetUp:

Processor	M1 chip
Operating System	MacOS Monterey
Number of Cores	8
RAM	8 GB
Python version	3.8.8

  The code editor used in this sheet : VSCode and Jupyter Notebook (graph)
  The output images attached are screenshots from the terminal.

- Exercise 1: <u>Implementing K Means</u>

  In this exercise we have to implement the K means clustering technique using MPI framework. I have used Collective communication here i.e, Scatter, Bcast and Gather methods are used for parallelism.

1. First the data is read and pre-processed. We had to work with Twenty NewsGroups corpus which contains multiple folders having multiple documents. As mentioned in the exercise I used sklearn to read and preprocess the data. Then sklearn was used to get the TF-IDF vectorized data. The result obtained

was a sparse matrix of shape (18846, 173451). Total number of documents read - 18846.

2. The matrix is split into different parts according to the workers available using array_splits
3. The centroids for K clusters are also initialized by taking **random** k data points from data matrix. The above 3 steps are done by the **master** worker.
4. Then the data splits are send to all the workers using comm.scatter() and the centroid is broadcasted to all by comm.bcast().

```
#Master worker
if(rank == 0):
    #Fetching the data using sklearn
    newsgroups_data = fetch_20newsgroups(subset="all")
    print("Number of documents-",len(newsgroups_data.data))
    #Finding TF-IDF matrix of the given data using sklearn
    vectorizer = TfidfVectorizer(stop_words="english")
    matrix = vectorizer.fit_transform(newsgroups_data.data)

    print(matrix.shape)
    print("\n Tf idf vector-", matrix)
    print(type(matrix))


    #For each cluster
    for i in range(K):
        #Finding the initial centroids as random data points in the matrix
        centroid.extend((matrix[np.random.randint(0,matrix.shape[0])]).tod

    #Spliting the data for different processes
    splits = np.array_split(range(matrix.shape[0]),size)
    matrix = [matrix[split] for split in splits]

    print("Number of Splits-",len(splits))
    print("Initial centroids-",centroid)

#Other workers
else:
    matrix = None
    centroid = None

#Scattering the data splits to all workers
matrix = comm.scatter(matrix,root=0)
```

5. Kmeans is implemented until convergence criteria is met. If euclidean distance between the current and previous centroids is measured and their sum is taken. If the sum is really small(~1) then iteration stops. But sometimes this may not

converge to optimal solution as it **depends on the initial centroids** taken. Hence I have also taken the maximum iterations as 50.

6. Time is also calculated for the whole program. This program was run for different number of clusters { 2, 4, 5, 6 } and for different number of workers { 1, 2, 4, 6, 8, 12 }.

Equation for euclidean distance

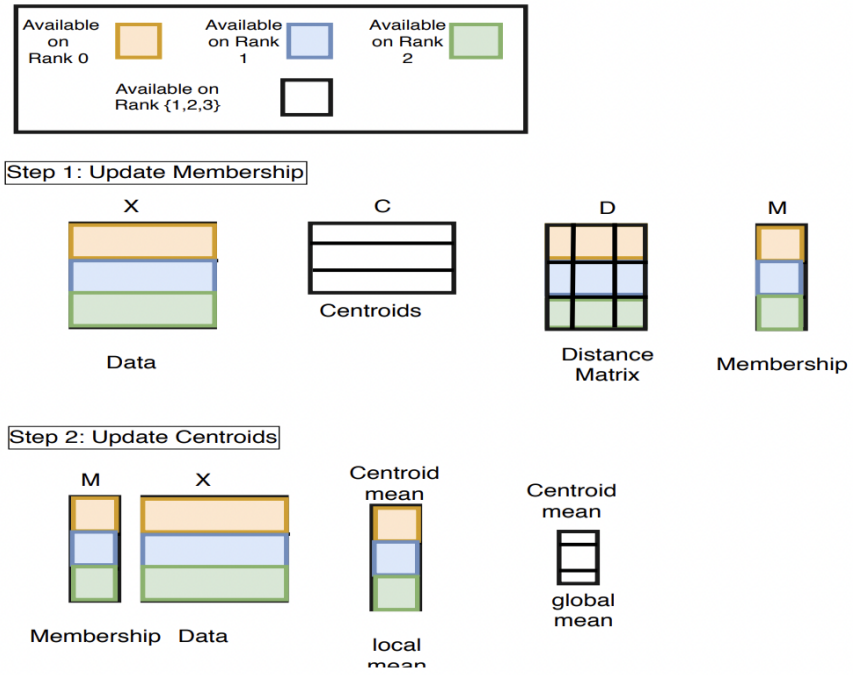
$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

p, q = two points in Euclidean n-space

q_i, p_i = Euclidean vectors, starting from the origin of the space (initial point)

n = n-space

K Means Clustering (rank =3) and cluster = 3:



<u>K-Means Clustering</u> :

- Here each worker gets a data split , while the centroids are available to all

- First the euclidean distance is calculated between the given data and the centroids. For this I converted the scipy matrix to array format and calculated the distance wrt to each centroid and stored them in an array.
- Then cluster membership is assigned by finding the minimum value at each index in the array. The cluster memberships of all the given data is found and stored in an array.

```
#Function to calculate the euclidean distance btw each point and centroids
#Assigning the cluster membership according to this measure
def euclidean(matrix,centroid):
    print("Type of the given data-", type(matrix))
    print("\nData Type of centroid" , type(centroid))
    matrix_ = matrix.A
    cluster_membership = []
    #for each datapoint
    for j in range((matrix_.shape)[0]):
        dist = []
        #for each centroid
        for i in range(K):
            #Calculating the euclidean distance
            dist.append(np.sqrt(np.sum(np.square(centroid[i]-matrix_[j]))))
        #Assigning the cluster having min distance value
        cluster_membership.append(dist.index(min(dist)))

    print("Length values of the data-", matrix_.shape)
    print("Length of cluster membership array-",len(cluster_membership))
    #Returning the array having the cluster membership values for each data point
    return cluster_membership
```

- For each cluster, a check is done to find all its members by checking the membership array values for each data. Thus elements belonging to each cluster is appended to an array.
- By taking the mean of this array, average is found which is the new local centroid for the given data for that cluster. The same is done for all clusters. Thus new local centroids are found and appended to a list.

```
#Function to implement K-Means clustering
def kmeans(centroid, matrix):
    #Call distance func
    membership_array = euclidean(matrix,centroid)

    #Calculating the Local centroid values for given data split
    new_centroid = []
    #for each cluster
    for j in range(K):
        elements = []
        #for each data point
        for i in range(matrix.shape[0]):
            #checking if the data belongs to the cluster
            if(membership_array[i] == j):
                #appending the data belonging to the given cluster
                elements.append(matrix[i,:])

        #New centroid appending the mean of all elements belonging to given cluste
        new_centroid.extend(elements[1].mean(axis=0))
    #  New local centroids for each cluster
    print("New local centroid-",new_centroid)
    #returing the new centroids and the cluster membership array
    return new_centroid, membership_array
```

- Using comm.gather, all the local centroids are gathered back. Then the master worker finds the new global centroid.

```
#While convergence criteria is not met
while(not converged and counter<50):
    #Broadcasting the centroids to all workers
    centroid = comm.bcast(centroid,root=0)

    old_centroid = centroid #saving copy of the centroid

    #Saving the old membership values after the initial run
    if(counter!=0 and rank == 0):
        old_membership = membership_full

    #Calling the K means function to calculate local centroids
    local_centroid, membership = kmeans(centroid,matrix)

    #Gathering all the local centroids and cluster memberships for data splits
    local_centroids = comm.gather(local_centroid,root = 0)
    membership_full = comm.gather(membership, root = 0)
```

- For each cluster, the local centroid found for it by different workers are selected, converted to arrays and their average is taken to get the new global centroid for that cluster. This is converted back into sparse matrix format. Thus new global centroids are found for all clusters

```
    #Finding global centroid
    #master worker
    if(rank == 0):
        #for each cluster
        for i in range(K):
            #Initializing to the shape of centroid
            sum = np.zeros((173451))
            #for each worker
            for k in range(size):
                #Getting the local centroids for the given cluster
                x = local_centroids[k][i]
                y = (np.asarray(x)).flatten()
                #Adding all the corresponding centroids to find the mean
                sum += y

            #Converting the new centroid to csr matrix form
            #dividing sum by size to get the mean centroid
             new_centroid = scipy.sparse.csr_matrix(sum/size)
             new_centroids.extend(new_centroid) #appending new centroids

        #Replacing the old with the new
        centroid = new_centroids
        print("New centroid-\n", centroid)
        counter+=1
```

- The master also checks the convergence criteria. If satisfied then program stops, else counter is updated and again iteration continues with new centroids which gets broadcasted.

```
#checking if old and new are same after first run
if(counter>2):
    #Convergence criteria, if the centroids are very near each other,
    # then sum of their differences would be very less
    if(dist(old_centroid,centroid)<=1):
            converged = True
            print("K means Clustering Converged!")
            #End Time
            print("Time taken for kmeans clustering for {} clusters and {} workers-{} "
            .format(K,size,round(MPI.Wtime()-start,4)))
    else:
            print("-----------Iterating Again-----------")
            print("Iteration number:",counter)

#maximum iterations
if(counter>=50):
    #Can occur if initial centroids are not chosen right
    print("Not converged in 50 iterations hence stopping")
    print("\n Time taken -",round(MPI.Wtime()-start,4))
    break
```

```python
#Function to calculate the euclidean distance for convergence criteria
def dist(old_centroid,centroid):

    sum = 0
    print("Old-", old_centroid)
    print("Newie-", centroid)
    #For each cluster
    for i in range(K):
        #Finding the euclidean distance btw the old and new centroid
        x=scipy.sparse.csr_matrix.sqrt(old_centroid[i].dot(centroid[i].T))

        #Finding sum of all the distance values
        sum = sum + x.data[0]
    print("Sum of Difference btw centroids-", sum)
    #Returning the sum of the distance btw each centroid
    return sum
```

MPI framework:
The reading, preprocessing, and splitting of data is done by the Master worker. The initialization of first centroids is also done by the master. Then the data splits are sent using scatter method and centroids are sent using broadcast.

Then each worker handles its given data. The master also handles it. They find the local centroids for their given data split.

Then the local centroids are gathered back and then the master finds the global centroid, which is again broadcasted if convergence is not met.

Thus the main part of the algorithm which takes the most time, which is calculating the new local centroid for each data, is handled in a parallel manner as we distribute the data among workers. Thus all workers are participating in the actual work.

Here I have observed that the time taken depends greatly on the initial random centroids chosen as sometimes many iterations may be required before convergence while other times less for the same K and same workers. Thus time can vary, but still a trend can be observed.

- Exercise 2: Performance Analysis

  Here I have tried the program for different K values and different numbers of workers. The following table shows the time taken in seconds for the same.

  Total time taken for running the whole program in seconds:

# of Clusters(K) # of Workers(P)	K = 2	K= 4	K = 5	K = 6
P = 1	87.5919	1408.3351	1573.6414	1758.4281
P = 2	46.3124	913.502	1028.9028	1157.9986
P = 4	39.6686	70.0159	920.3325	1029.9423
P = 6	36.151	110.5599	842.3061	923.8876
P = 8	35.9033	489.8684	58.7149	996.7966
P = 12	38.181	50.3597	59.4915	65.6869

Observations:

We see that the time increases as K increases i.e, for more clusters, more time is taken. This is because as K increases the for loop iterations also increase as one more centroid is there. Hence this is expected.

Also we can see that the time becomes almost half when we increase the number of workers from 1 to 2. Then there is a trend of time decreasing as the number of workers increases (almost linear) . But when we increase from 8 to 12 workers, time increases. This may be due to the communication overhead increasing as the number of workers are more.

Some exceptions can be seen, this may be due to the randomness of the initial centroid as when I run again, the time changes and convergence does occur faster. Also I checked with the sum of distance between old and new centroids as 1 for convergence, this may not happen when we increase the number of K as the sum will increase. The final sum value for cases where convergence did not happen in 50 iterations were around 1.5-3.

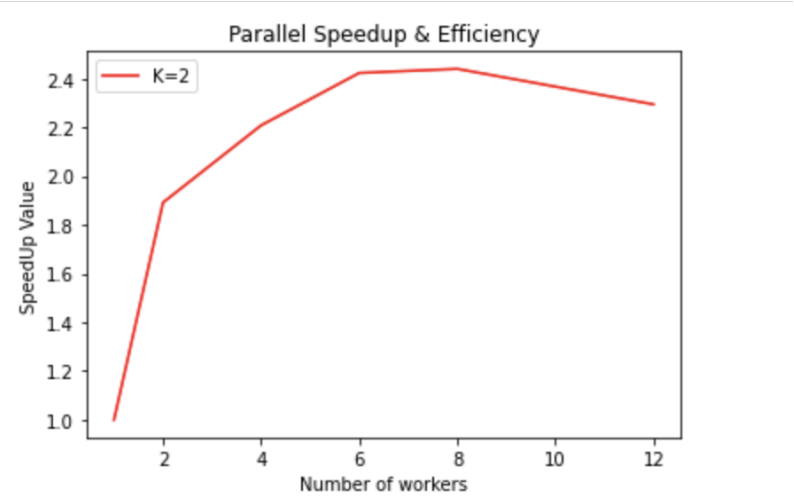
Speed Up Graph

The speedup gained from applying n workers, Speedup(n), is the ratio of the one-worker execution time to the n-worker parallel execution time:

$$Speedup(n) = T(1)/T(n).$$

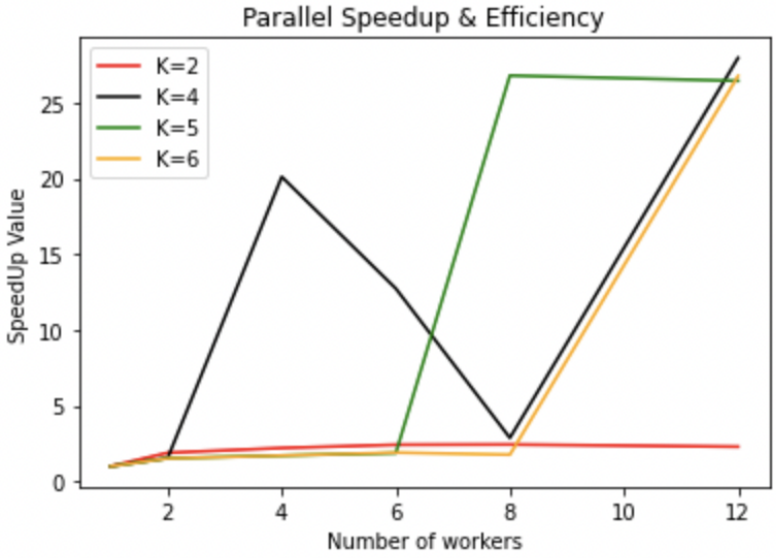
So Speedup(n) should be a number greater than 1.0, and the greater it is, the better. Normally the Speedup(n) is less than n as there will be some parts of code we cannot run parallely(like calculating the global centroid), but in some cases, it becomes greater than n which is called super linear speed up.

Speed up graph for K=2



We can see that the speed increase the most from 1 to 2 workers. Then the rate of increase is lesser and it decreases slightly from 8 to 12 workers.

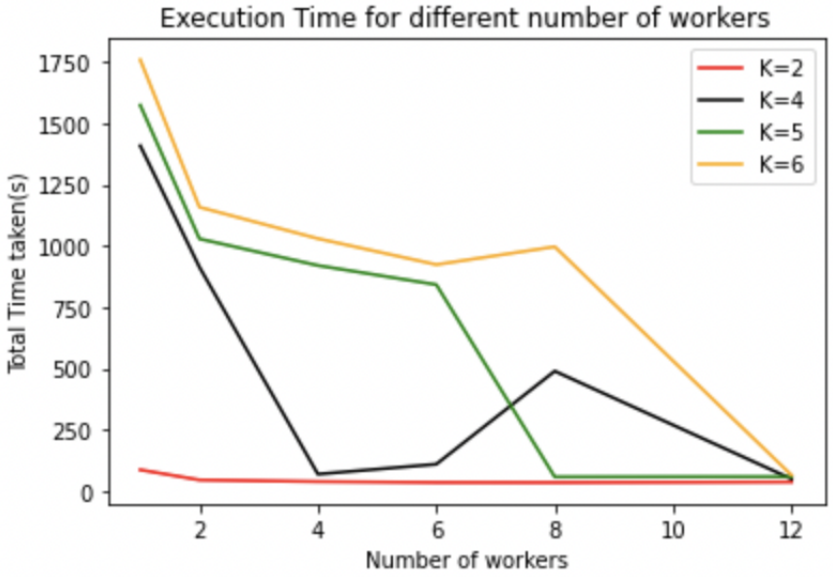
Here the below speed up graph displays the speed up with different values of K also.



We can see that, the speed has an overall increasing trend as we increase the workers. Also the speedup value increases as we increase K. K=4 shows an unexpected decrease in value at workers=8, but this maybe due to the initial centroid chosen. Overall all show increase in speedup. For K=5, the major speedvalue increase occur from workers=6 to 8.

The observations are as expected. There is no super linear increase as all parts of the code cannot be parallelised.

The below graph plots the execution time with the number of workers for each K:



Here we can see that the execution time decreases as the number of clusters decrease and the number of workers increase.

Sample Outputs:

```
<class 'scipy.sparse.csr.csr_matrix'>
Number of Splits- 1
Initial centroids- [matrix([[0., 0., 0., ..., 0., 0., 0.]]), matrix([[0.       , 0.0415238, 0.       , ...,
         0.       ]])]
Type of the given data- <class 'scipy.sparse.csr.csr_matrix'>

Data Type of centroid <class 'list'>
Length values of the data- (18846, 173451)
Length of cluster membership array- 18846
New local centroid- [matrix([[0., 0., 0., ..., 0., 0., 0.]]), matrix([[0., 0., 0., ..., 0., 0., 0.]])]
New centroid-
 [<1x173451 sparse matrix of type '<class 'numpy.float64'>'
        with 63 stored elements in Compressed Sparse Row format>, <1x173451 sparse matrix of type '<class '
        with 196 stored elements in Compressed Sparse Row format>]
Type of the given data- <class 'scipy.sparse.csr.csr_matrix'>

Data Type of centroid <class 'list'>
Length values of the data- (18846, 173451)
Length of cluster membership array- 18846
New local centroid- [matrix([[0., 0., 0., ..., 0., 0., 0.]]), matrix([[0., 0., 0., ..., 0., 0., 0.]])]
New centroid-
 [<1x173451 sparse matrix of type '<class 'numpy.float64'>'
        with 67 stored elements in Compressed Sparse Row format>, <1x173451 sparse matrix of type '<class '
        with 72 stored elements in Compressed Sparse Row format>]
Type of the given data- <class 'scipy.sparse.csr.csr_matrix'>

Data Type of centroid <class 'list'>
Length values of the data- (18846, 173451)
Length of cluster membership array- 18846
New local centroid- [matrix([[0., 0., 0., ..., 0., 0., 0.]]), matrix([[0., 0., 0., ..., 0., 0., 0.]])]
New centroid-
 [<1x173451 sparse matrix of type '<class 'numpy.float64'>'
        with 67 stored elements in Compressed Sparse Row format>, <1x173451 sparse matrix of type '<class '
        with 63 stored elements in Compressed Sparse Row format>]
Old- [<1x173451 sparse matrix of type '<class 'numpy.float64'>'
        with 67 stored elements in Compressed Sparse Row format>, <1x173451 sparse matrix of type '<class '
        with 72 stored elements in Compressed Sparse Row format>]
Newie- [<1x173451 sparse matrix of type '<class 'numpy.float64'>'
        with 67 stored elements in Compressed Sparse Row format>, <1x173451 sparse matrix of type '<class '
        with 63 stored elements in Compressed Sparse Row format>]
Sum of Difference btw centroids- 1.1399318928353284
-----------Iterating Again-----------
Iteration number: 3
Type of the given data- <class 'scipy.sparse.csr.csr_matrix'>

Data Type of centroid <class 'list'>
Length values of the data- (18846, 173451)
Length of cluster membership array- 18846
New local centroid- [matrix([[0., 0., 0., ..., 0., 0., 0.]]), matrix([[0., 0., 0., ..., 0., 0., 0.]])]
New centroid-
 [<1x173451 sparse matrix of type '<class 'numpy.float64'>'
        with 165 stored elements in Compressed Sparse Row format>, <1x173451 sparse matrix of type '<class
        with 196 stored elements in Compressed Sparse Row format>]
Old- [<1x173451 sparse matrix of type '<class 'numpy.float64'>'
        with 67 stored elements in Compressed Sparse Row format>, <1x173451 sparse matrix of type '<class '
        with 63 stored elements in Compressed Sparse Row format>]
Newie- [<1x173451 sparse matrix of type '<class 'numpy.float64'>'
        with 165 stored elements in Compressed Sparse Row format>, <1x173451 sparse matrix of type '<class
        with 196 stored elements in Compressed Sparse Row format>]
Sum of Difference btw centroids- 0.3976430739906909
K means Clustering Converged!
Time taken for kmeans clustering for 2 clusters and 1 workers-87.5919
(base) sruthysanthosh@Sruthys-MacBook-Air Sem2 %
```

```
Type of the given data- <class 'scipy.sparse.csr.csr_matrix'>

Data Type of centroid <class 'list'>
Length values of the data- (2356, 173451)
Length of cluster membership array- 2356
New local centroid- [matrix([[0., 0., 0., ..., 0., 0., 0.]]), matrix([[0., 0., 0., ..., 0., 0., 0.]])]
New centroid-
 [<1x173451 sparse matrix of type '<class 'numpy.float64'>'
        with 529 stored elements in Compressed Sparse Row format>, <1x173451 sparse matrix of type '<class
        with 1052 stored elements in Compressed Sparse Row format>]
Old- [<1x173451 sparse matrix of type '<class 'numpy.float64'>'
        with 642 stored elements in Compressed Sparse Row format>, <1x173451 sparse matrix of type '<class
        with 768 stored elements in Compressed Sparse Row format>]
Newie- [<1x173451 sparse matrix of type '<class 'numpy.float64'>'
        with 529 stored elements in Compressed Sparse Row format>, <1x173451 sparse matrix of type '<class
        with 1052 stored elements in Compressed Sparse Row format>]
Sum of Difference btw centroids- 0.4170651911554263
K means Clustering Converged!
Time taken for kmeans clustering for 2 clusters and 8 workers-35.9033
```

```
        with 218 stored elements in Compressed Sparse Row format>]
Sum of Difference btw centroids- 1.9718082129702559
----------Iterating Again----------
Iteration number: 49
Type of the given data- <class 'scipy.sparse.csr.csr_matrix'>

Data Type of centroid <class 'list'>
Length values of the data- (9423, 173451)
Length of cluster membership array- 9423
New local centroid- [matrix([[0., 0., 0., ..., 0., 0., 0.]]), matrix([[0., 0.,
, 0.        , 0.        , ..., 0.        , 0.        ,
         0.        ]])]
New centroid-
 [<1x173451 sparse matrix of type '<class 'numpy.float64'>'
        with 145 stored elements in Compressed Sparse Row format>, <1x173451 s
        with 106 stored elements in Compressed Sparse Row format>, <1x173451 s
        with 148 stored elements in Compressed Sparse Row format>, <1x173451 s
        with 347 stored elements in Compressed Sparse Row format>, <1x173451 s
        with 162 stored elements in Compressed Sparse Row format>]
Old- [<1x173451 sparse matrix of type '<class 'numpy.float64'>'
        with 149 stored elements in Compressed Sparse Row format>, <1x173451 s
        with 110 stored elements in Compressed Sparse Row format>, <1x173451 s
        with 146 stored elements in Compressed Sparse Row format>, <1x173451 s
        with 174 stored elements in Compressed Sparse Row format>, <1x173451 s
        with 216 stored elements in Compressed Sparse Row format>]
Newie- [<1x173451 sparse matrix of type '<class 'numpy.float64'>'
        with 145 stored elements in Compressed Sparse Row format>, <1x173451 s
        with 106 stored elements in Compressed Sparse Row format>, <1x173451 s
        with 148 stored elements in Compressed Sparse Row format>, <1x173451 s
        with 347 stored elements in Compressed Sparse Row format>, <1x173451 s
        with 162 stored elements in Compressed Sparse Row format>]
Sum of Difference btw centroids- 1.087618296706066
K means Clustering Converged!
Time taken for kmeans clustering for 5 clusters and 2 workers-1028.9016
```

```
Sum of Difference btw centroids- 1.09116430583568
K means Clustering Converged!
Time taken for kmeans clustering for 5 clusters and 12 workers-59.4915
```

References:

- https://hmf.enseeiht.fr/travaux/CD0001/travaux/optmfn/micp/reports/s13itml/theory.html
- https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Comm.html#mpi4py.MPI.Comm.send
- https://stackoverflow.com/questions/40336601/python-appending-array-to-an-array

- https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Gautam-Shende-Spring-2018.pdf
- https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html
- https://iq.opengenus.org/implement-document-clustering-python/
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.mean.html
- https://www.delftstack.com/howto/numpy/matrix-to-array-numpy/#:~:text=7%208%209%5D-,Use%20the%20numpy.,get%20a%20one%2Ddimensional%20array.
- https://www.ibm.com/docs/en/spss-statistics/beta?topic=analysis-k-means-cluster-convergence-criteria
- https://www.ismll.uni-hildesheim.de/lehre/bd-16s/exercises/bd-02-lec.pdf