Sreyashi Saha Matriculation No: 1747271

Distributed K-means Clustering

A K-means clustering algorithm tries to group similar items in the form of clusters. The number of groups or clusters is represented by K.

K-means clustering algorithm works in three steps:

1) first we initialize the number of clusters.

2) Next we initialize the K centroids.

STEP1: INITIALIZE THE FIRST CENTROIDS

I have first defined a function to initialize the first k centroids by using Naive Sharding method as well as random initialisation. The sharding centroid initialization algorithm primarily depends on the composite summation value of all the attributes for a particular instance or row in a dataset. The idea is to calculate the composite value and then use it to sort the instances of the data. Once the data set is sorted, it is then divided horizontally into k shards. Finally, all the attributes from each shard will be summed and their mean will be calculated. The shard attributes mean value collection will be identified as the set of centroids that can be used for initialization.

```
# Random Method Of Centroid Initialisation
def centroid_initialisation1(data, number_of_clusters):
    # Randomly choosing 3 rows from the dataset as our centroids
    first_centroids = np.random.randint(data.shape[0], size=number_of_clusters)
    centroid_list = [] # Initialise an empty list to store the 3 random centroids
    # print('centroid rows are ', first_centroids)
    for i in first_centroids:
        centroid_list.append(np.asanyarray(data[i], dtype=float))
    return centroid_list
```

```
# Naive Sharding Method Of Centroid Initialization
def centroid_initialization(ds,k_clusters):
    n = np.shape(ds)[1]
    # print(n) # print the number of attributes in the dataset
    m = np.shape(ds)[0]
    # print(m) # print the number of rows in the dataset
    centroids = np.mat(np.zeros((k_clusters,n)))
    # Sum all elements of each row, add as col to original dataset, sort
    sum_column = np.mat(np.sum(ds, axis=1)) # Calculate the sum of each attributes and storing in a 1-d array
    Transposed_sum_column=sum_column.T
    ds = np.append(sum_column.T, ds, axis=1) # storing the sums in a separate column
    # print(ds)
    ds.sort(axis=0) # sorting the summed up values in ascending order
    # Step value for dataset sharding
    step = floor(m/k)
    # print(step)
    vectorize_func = np.vectorize(average)
    # Vectorize mean ufunc for numpy array
    # finding the average of some of the data points
    for i in range(k):
        if i == k-1:
            centroids[i:] = vectorize_func(np.sum(ds[i*step:,1:], axis=0), step)
        else:
            centroids[i:] = vectorize_func(np.sum(ds[i*step:(i+1)*step,1:], axis=0), step)
    return centroids #returning initially the 3 centroids since we have considered k(=3) clusters
```

Classical k-means clustering utilizes random centroid initialization. In order for randomization to be properly performed, the entire space being occupied by all instances in all dimensions is first to be determined. This means that all instances in the dataset must be enumerated, and a record must be kept of the minimum and maximum values of each attribute, constituting the boundaries of the various planes in n-dimensional space, n being the number of attributes in the dataset. This can be a time-consuming process; the time required to perform randomization grows exponentially with increased dataset complexity, by way of both a larger number of instances and a larger number of attributes.

Traditional k-means utilizes a randomization process for initializing these centroids -- though this is not the only approach -- but poor initialization can lead to increased numbers of required clustering iterations to reach convergence, a greater overall runtime, and a less-efficient algorithm overall.

Reference: https://www.kdnuggets.com/2017/03/naive-sharding-centroid-initialization-method.html

STEP2: Find the Euclidean Distance

```
# Calculating the Euclidean Distance
def distance(row_data,centroid):
    euclidian_distance = np.linalg.norm(row_data-centroid) #finding out the euclidean distance between the data ppoints and the centroids
    return euclidian_distance
```

K-Means procedure - which is a vector quantization method often used as a clustering method - does not explicitly use pairwise distances between data points at all. It amounts to repeatedly assigning points to the closest centroid thereby using Euclidean distance from data points to a centroid.

Given two points A and B in d dimensional space such that $A = [a_1, a_2, \cdots, a_d]$ and $B = [b_1, b_2, \cdots, b_d]$, the Euclidean distance between A and B is defined as:

$$||A - B|| = \sqrt{\sum_{i=1}^{d} (a_i - b_i)^2} \qquad (1)$$

STEP2: Find the Minimum Euclidean Distance and return its position

```
# Find the minimum distance obtained by calculating Euclidean distance for each data point w.r.t. the 3 centroids
# Return the index of the minimum distance found
def minimum_index(distance_array):
    # minimum_distance = np.min(distance_array)
    #finding the minimum distance for each instance that is calculated by computing the euclidean distance with respect to all the centroids
    Minimum_position = np.argmin(distance_array)# stores the position which has the minimum distance
    return Minimum_position+1 # to get which cluster that minimum distance belongs to
```

In this step basically I receive an array containing 3 distances computed wrt each of the centroids(3 in our case, since k=3). It then returns the position of the minimum distance from this received set. This basically helps us determine which cluster a particular instance belongs to, i.e. if we get 0 as the minimum position and 1 as output from this function it

basically means that the instance on which we computed the euclidean distance belongs to cluster 1.

STEP3: Assigning 740 instances their respective clusters

```
def assigning_clusters(data_chunks,centroids):
    array_distances = np.array([])
    eu_dist = []
    my_dict={}
    for i,row in enumerate(data_chunks):
        dist=[]
        for j in range(len(centroids)):
            # appending the euclidean distance obtained by subracting each centroids from each of the columns of each rows in the dataset
            dist.append(distance(row,centroids[j]))
            # print("The distances obtained after computing euclidean dist",d)
            eu_dist.append(dist) #appending 3 sets of distances obtained wrt 3 centroids to a list
            # Finding out the minimum distance among 3 distances obtained for 1 row
            minimum_position = minimum_index(dist)
            my_dict[i] = minimum_position # labeling the ith row according to the clusters
        array_distances = np.append(array_distances,eu_dist)
    # cluster_list = filtering_cluster(my_dict,len(centroids))
    return my_dict
```

This function takes the chunks of data and the 3 centroids as input. Inside this function we call the function to find the euclidean distance and also the function which returns us the minimum index of the Euclidean distances found. It then stores the minimum index in a dictionary and returns that. So we finally get a dictionary which has the row numbers as key and the cluster numbers as its values. So we obtained the 740 instances along with which cluster it belongs to.

STEP4: Assigning 740 instances their respective clusters

From the above step we have a dictionary with all the rows and the cluster they belong to respectively. But I want to extract the instances for each of the individual clusters which will later aid me in updating the centroids. So I defined a function which takes the dictionary I

```
# returning a list which will give me the rows obtaianed for each of the clusters
def filtered_clusters(dictA, clusters):
    resultant = []
    for i in range(clusters):
        resultant_list=[]
        for key,values in dictA.items():
            if values == i+1:
                resultant_list.append(key)

        resultant.append(resultant_list)
    return resultant
```

obtained from the above step and checks which instances match with with which cluster and returns me a list of lists. Basically it returns me the instances for each cluster as a list. So I now have a list of instances for cluster 1, cluster 2 and so on.

STEP5: Update Centroids

Now that I have all the instances for each of the clusters separately I can compute the sum for the instances belonging to each cluster. I have my list of lists from the previous step and the data chunks as input. I find the sum of all instances belonging to a particular cluster number row wise. (Note: I calculate the sum for all the attributes of those instances belonging to the same cluster)

```
# Update centroids in every iteration untilmstopping condition is met
def update_centroids(list_keys, dataset):
    updated_center = []
    for i, row in enumerate(list_keys):
        for grp in row:
            # print('Cluster No: i:', grp)
            # convert array into a df to make extraction of rows easier
            convert = pd.DataFrame(dataset)
            # print('df:', converted_df )
            df = convert.loc[convert.index[grp]]
            count = len(df)
            # print('df:', df)
            # Sum of all rows attribute wise
            sum = df.sum(axis=0)
            sum = sum.to_numpy()#convert back to numpy array
        # print(f'Sum of columns for cluster[{i}] is: {sum} \n')
        updated_center.append([count, sum])
    # print("Updated centroid:", updated_center)
    return updated_center
```

So this function returns me the total number of instances belonging to each of the clusters and the sum of those as well. So say cluster 1 has 9 instances then I will get a list of list with count as 9 and the summed up instances.

STEP6: Function to call the previous 3 functions

```
# Finding sum according to the clusters
# for eg if there are 20 rows in cluster 1 then returning the sum of all those rows
def sum_rows(dataset, centroids):
    labelled_dictionary= assigning_clusters(dataset, centroids)
    # print(labelled_dictionary)
    resultant_list = filtered_clusters(labelled_dictionary, len(centroids))
    # print(resultant_list)
    sum_to_update_centroids = update_centroids(resultant_list, dataset)
    # print(sum_to_update_centroids)
    return sum_to_update_centroids
```

For my ease I have defined another function to call the 3 functions made earlier. This function returns me the summed up instances for each cluster and using this I would calculate the local means.

STEP7: Work in Parallel

In the master worker I read the data provided. Since it is a data frame I convert into a numpy array. Here I also initialise the first 3 random centroids . I then scatter the chunks of data after splitting the data into equal number according to the number of processes. I broadcast the initial centroids so that all my process gets the centroids and perform the calculations wrt each of the centroids.

After this step I'll need to arrive to a convergence such that my after updating my global centroids I check whether it matches with the previous one or not. If it doe then the program will exit and plot a performance graph.

```
#Assigning a flag variable
flag=True
previous_global_centroid = None
while flag:

    previous_global_centroid = centroids_bcast
    # dictionary_cluster_assignment = assigning_clusters(scatter, centroids_bcast)
    # print('Each Row Assignment', dictionary_cluster_assignment, '\n')
    sum_array = sum_rows(scatter, centroids_bcast)
    print('Sum_array are  ', (sum_array), '\n')


    local_sums = []
    count = []
    for i in range(len(sum_array)):
        # print(len(sum_array))
        count.append(sum_array[i][0])
        local_sums.append(sum_array[i][1])

    print('counts are', count)
    print('localsums are', local_sums)

    count = np.array(count)
    local_sums = np.array(local_sums)#convert into array for simpler execution
```

So what I do in this while loop, I assign the centroids as the previous centroids and store it. I then call the functions accordingly to get the sums of all instances along with the count for each cluster. I then extract the count values and the sum values and store them separately to get the local means.

After this step I use the reduce function to find the total number of instances for each cluster along with the total sum of the local sums. Now to find the global average I start a nested loop to check whether I have received the reduced sums and then find the global average and store it in a list for further verification.

```
# summing up the counts obtained from each process for all clusters
total_counts = comm.reduce(count, root=root)
# print('The total number counts are ', total_counts,'\n')

# summing up the sums obtained from each process for all clusters
final_sum = comm.reduce(local_sums, root = root)
# print('Final sum', final_sum,'\n')
iter=0
k = 3

global_average = []
if total_counts is not None:
    if final_sum is not None:
        for i in range(len(total_counts)):
            global_average.append(final_sum[i]/total_counts[i])
    print('global avg is', global_average,'\n')
```

Once I get the global average I then redistribute it to the ranks to find new cluster. I continue this process until my previous global centroid matches with the newly found. I also keep a count as to how many times this process continues until my cluster instances do not change. Once it matches it will print a line stating the same and exit the loop.

```
#  Redistribute the new centroids to each cluster until it converges
centroids_bcast = comm.bcast(global_average, root=root)
# cent = np.array([centroids_bcast])
# print(cent.shape)
iter+=1
if np.array_equal(previous_global_centroid,centroids_bcast):
    print('The global centroid is same....Exit the loop')
    flag = False
else:
    continue
```

STEP8: Perfomance Analysis

When I perform K-Means on **1 processor and 3 clusters** the total time required to calculate the global centroid is 16.51 seconds. The number of iterations that is cluster has

```
global avg is [array([1.76749226e+01, 2.03529412e+01, 5.65015480e+00, 3.84829721e+00,
       2.47368421e+00, 1.59349845e+02, 2.95448916e+01, 1.43219814e+01,
       3.72631579e+01, 2.70954189e+02, 9.49907121e+01, 2.47678019e-02,
       1.34674923e+00, 4.76780186e-01, 5.75851393e-01, 8.97832817e-02,
       7.73993808e-02, 8.15882353e+01, 1.73891641e+02, 2.70773994e+01,
       6.61919505e+00]), array([2.03914729e+01, 1.87131783e+01, 6.74031008e+00, 4.03875969e+00,
       2.59302326e+00, 2.39011628e+02, 2.73875969e+01, 1.15542636e+01,
       3.65930233e+01, 2.70032624e+02, 9.45426357e+01, 6.97674419e-02,
       1.34108527e+00, 1.48062016e+00, 4.14728682e-01, 3.48837209e-02,
       1.18992248e+00, 7.53720930e+01, 1.69550388e+02, 2.62054264e+01,
       6.24418605e+00]), array([1.48616352e+01, 1.77232704e+01, 7.01886792e+00, 3.84905660e+00,
       2.61006289e+00, 3.18547170e+02, 3.34465409e+01, 1.05849057e+01,
       3.45660377e+01, 2.74944365e+02, 9.38427673e+01, 8.80503145e-02,
       1.10062893e+00, 1.37106918e+00, 7.98742138e-01, 1.00628931e-01,
       1.38364780e+00, 7.97924528e+01, 1.72666667e+02, 2.66289308e+01,
       8.64779874e+00])]
The global centroid is same....Exit the loop
total time for processes is  16.510051
Global Centroids [array([1.76749226e+01, 2.03529412e+01, 5.65015480e+00, 3.84829721e+00,
       2.47368421e+00, 1.59349845e+02, 2.95448916e+01, 1.43219814e+01,
       3.72631579e+01, 2.70954189e+02, 9.49907121e+01, 2.47678019e-02,
       1.34674923e+00, 4.76780186e-01, 5.75851393e-01, 8.97832817e-02,
       7.73993808e-02, 8.15882353e+01, 1.73891641e+02, 2.70773994e+01,
       6.61919505e+00]), array([2.03914729e+01, 1.87131783e+01, 6.74031008e+00, 4.03875969e+00,
       2.59302326e+00, 2.39011628e+02, 2.73875969e+01, 1.15542636e+01,
       3.65930233e+01, 2.70032624e+02, 9.45426357e+01, 6.97674419e-02,
       1.34108527e+00, 1.48062016e+00, 4.14728682e-01, 3.48837209e-02,
       1.18992248e+00, 7.53720930e+01, 1.69550388e+02, 2.62054264e+01,
       6.24418605e+00]), array([1.48616352e+01, 1.77232704e+01, 7.01886792e+00, 3.84905660e+00,
       2.61006289e+00, 3.18547170e+02, 3.34465409e+01, 1.05849057e+01,
       3.45660377e+01, 2.74944365e+02, 9.38427673e+01, 8.80503145e-02,
       1.10062893e+00, 1.37106918e+00, 7.98742138e-01, 1.00628931e-01,
       1.38364780e+00, 7.97924528e+01, 1.72666667e+02, 2.66289308e+01,
       8.64779874e+00])] and no of iterations 9
```

been assigned differently for 9 times until we got the same ones.

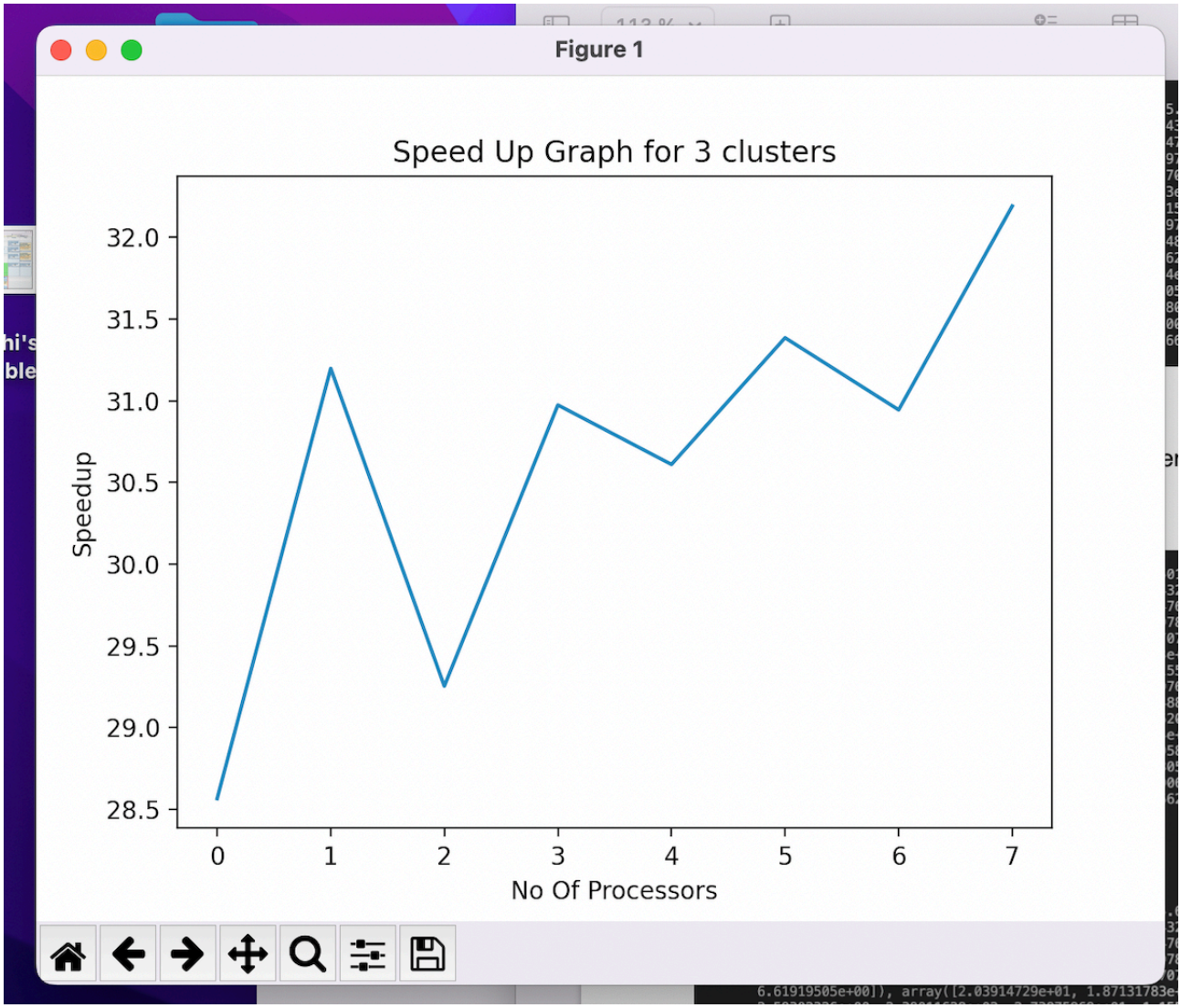
When I perform K-Means on **2 processor and 3 clusters** the total time required to calculate the global centroid is 4.21 seconds. The number of iterations that is cluster has been assigned differently for 9 times until we got the same ones.

```
global avg is [array([1.76749226e+01, 2.03529412e+01, 5.65015480e+00, 3.84829721e+00,
       2.47368421e+00, 1.59349845e+02, 2.95448916e+01, 1.43219814e+01,
       3.72631579e+01, 2.70954189e+02, 9.49907121e+01, 2.47678019e-02,
       1.34674923e+00, 4.76780186e-01, 5.75851393e-01, 8.97832817e-02,
       7.73993808e-02, 8.15882353e+01, 1.73891641e+02, 2.70773994e+01,
       6.61919505e+00]), array([2.03914729e+01, 1.87131783e+01, 6.74031008e+00, 4.03875969e+00,
       2.59302326e+00, 2.39011628e+02, 2.73875969e+01, 1.15542636e+01,
       3.65930233e+01, 2.70032624e+02, 9.45426357e+01, 6.97674419e-02,
       1.34108527e+00, 1.48062016e+00, 4.14728682e-01, 3.48837209e-02,
       1.18992248e+00, 7.53720930e+01, 1.69550388e+02, 2.62054264e+01,
       6.24418605e+00]), array([1.48616352e+01, 1.77232704e+01, 7.01886792e+00, 3.84905660e+00,
       2.61006289e+00, 3.18547170e+02, 3.34465409e+01, 1.05849057e+01,
       3.45660377e+01, 2.74944365e+02, 9.38427673e+01, 8.80503145e-02,
       1.10062893e+00, 1.37106918e+00, 7.98742138e-01, 1.00628931e-01,
       1.38364780e+00, 7.97924528e+01, 1.72666667e+02, 2.66289308e+01,
       8.64779874e+00])]
The global centroid is same....Exit the loop
The global centroid is same....Exit the loop
total time for processes is  4.211315000000001
Global Centroids [array([1.76749226e+01, 2.03529412e+01, 5.65015480e+00, 3.84829721e+00,
       2.47368421e+00, 1.59349845e+02, 2.95448916e+01, 1.43219814e+01,
       3.72631579e+01, 2.70954189e+02, 9.49907121e+01, 2.47678019e-02,
       1.34674923e+00, 4.76780186e-01, 5.75851393e-01, 8.97832817e-02,
       7.73993808e-02, 8.15882353e+01, 1.73891641e+02, 2.70773994e+01,
       6.61919505e+00]), array([2.03914729e+01, 1.87131783e+01, 6.74031008e+00, 4.03875969e+00,
       2.59302326e+00, 2.39011628e+02, 2.73875969e+01, 1.15542636e+01,
       3.65930233e+01, 2.70032624e+02, 9.45426357e+01, 6.97674419e-02,
       1.34108527e+00, 1.48062016e+00, 4.14728682e-01, 3.48837209e-02,
       1.18992248e+00, 7.53720930e+01, 1.69550388e+02, 2.62054264e+01,
       6.24418605e+00]), array([1.48616352e+01, 1.77232704e+01, 7.01886792e+00, 3.84905660e+00,
       2.61006289e+00, 3.18547170e+02, 3.34465409e+01, 1.05849057e+01,
       3.45660377e+01, 2.74944365e+02, 9.38427673e+01, 8.80503145e-02,
       1.10062893e+00, 1.37106918e+00, 7.98742138e-01, 1.00628931e-01,
       1.38364780e+00, 7.97924528e+01, 1.72666667e+02, 2.66289308e+01,
       8.64779874e+00])] and no of iterations 9
```

For 4 processors the time taken is even lesser i.e. 2.84 secs and the number of iterations remains the same.

```
global avg is [array([1.76749226e+01, 2.03529412e+01, 5.65015480e+00, 3.84829721e+00,
       2.47368421e+00, 1.59349845e+02, 2.95448916e+01, 1.43219814e+01,
       3.72631579e+01, 2.70954189e+02, 9.49907121e+01, 2.47678019e-02,
       1.34674923e+00, 4.76780186e-01, 5.75851393e-01, 8.97832817e-02,
       7.73993808e-02, 8.15882353e+01, 1.73891641e+02, 2.70773994e+01,
       6.61919505e+00]), array([2.03914729e+01, 1.87131783e+01, 6.74031008e+00, 4.03875969e+00,
       2.59302326e+00, 2.39011628e+02, 2.73875969e+01, 1.15542636e+01,
       3.65930233e+01, 2.70032624e+02, 9.45426357e+01, 6.97674419e-02,
       1.34108527e+00, 1.48062016e+00, 4.14728682e-01, 3.48837209e-02,
       1.18992248e+00, 7.53720930e+01, 1.69550388e+02, 2.62054264e+01,
       6.24418605e+00]), array([1.48616352e+01, 1.77232704e+01, 7.01886792e+00, 3.84905660e+00,
       2.61006289e+00, 3.18547170e+02, 3.34465409e+01, 1.05849057e+01,
       3.45660377e+01, 2.74944365e+02, 9.38427673e+01, 8.80503145e-02,
       1.10062893e+00, 1.37106918e+00, 7.98742138e-01, 1.00628931e-01,
       1.38364780e+00, 7.97924528e+01, 1.72666667e+02, 2.66289308e+01,
       8.64779874e+00])]
The global centroid is same....Exit the loop
The global centroid is same....Exit the loop
The global centroid is same....Exit the loop
The global centroid is same....Exit the loop
total time for processes is  2.846739
Global Centroids [array([1.76749226e+01, 2.03529412e+01, 5.65015480e+00, 3.84829721e+00,
       2.47368421e+00, 1.59349845e+02, 2.95448916e+01, 1.43219814e+01,
       3.72631579e+01, 2.70954189e+02, 9.49907121e+01, 2.47678019e-02,
       1.34674923e+00, 4.76780186e-01, 5.75851393e-01, 8.97832817e-02,
       7.73993808e-02, 8.15882353e+01, 1.73891641e+02, 2.70773994e+01,
       6.61919505e+00]), array([2.03914729e+01, 1.87131783e+01, 6.74031008e+00, 4.03875969e+00,
       2.59302326e+00, 2.39011628e+02, 2.73875969e+01, 1.15542636e+01,
       3.65930233e+01, 2.70032624e+02, 9.45426357e+01, 6.97674419e-02,
       1.34108527e+00, 1.48062016e+00, 4.14728682e-01, 3.48837209e-02,
       1.18992248e+00, 7.53720930e+01, 1.69550388e+02, 2.62054264e+01,
       6.24418605e+00]), array([1.48616352e+01, 1.77232704e+01, 7.01886792e+00, 3.84905660e+00,
       2.61006289e+00, 3.18547170e+02, 3.34465409e+01, 1.05849057e+01,
       3.45660377e+01, 2.74944365e+02, 9.38427673e+01, 8.80503145e-02,
       1.10062893e+00, 1.37106918e+00, 7.98742138e-01, 1.00628931e-01,
       1.38364780e+00, 7.97924528e+01, 1.72666667e+02, 2.66289308e+01,
       8.64779874e+00])] and no of iterations 9
```

When I increased my processor to 6 and 8 the time taken to compute also increased and I got around 3.103536 and 3.790966 seconds respectively with 6 iterations.



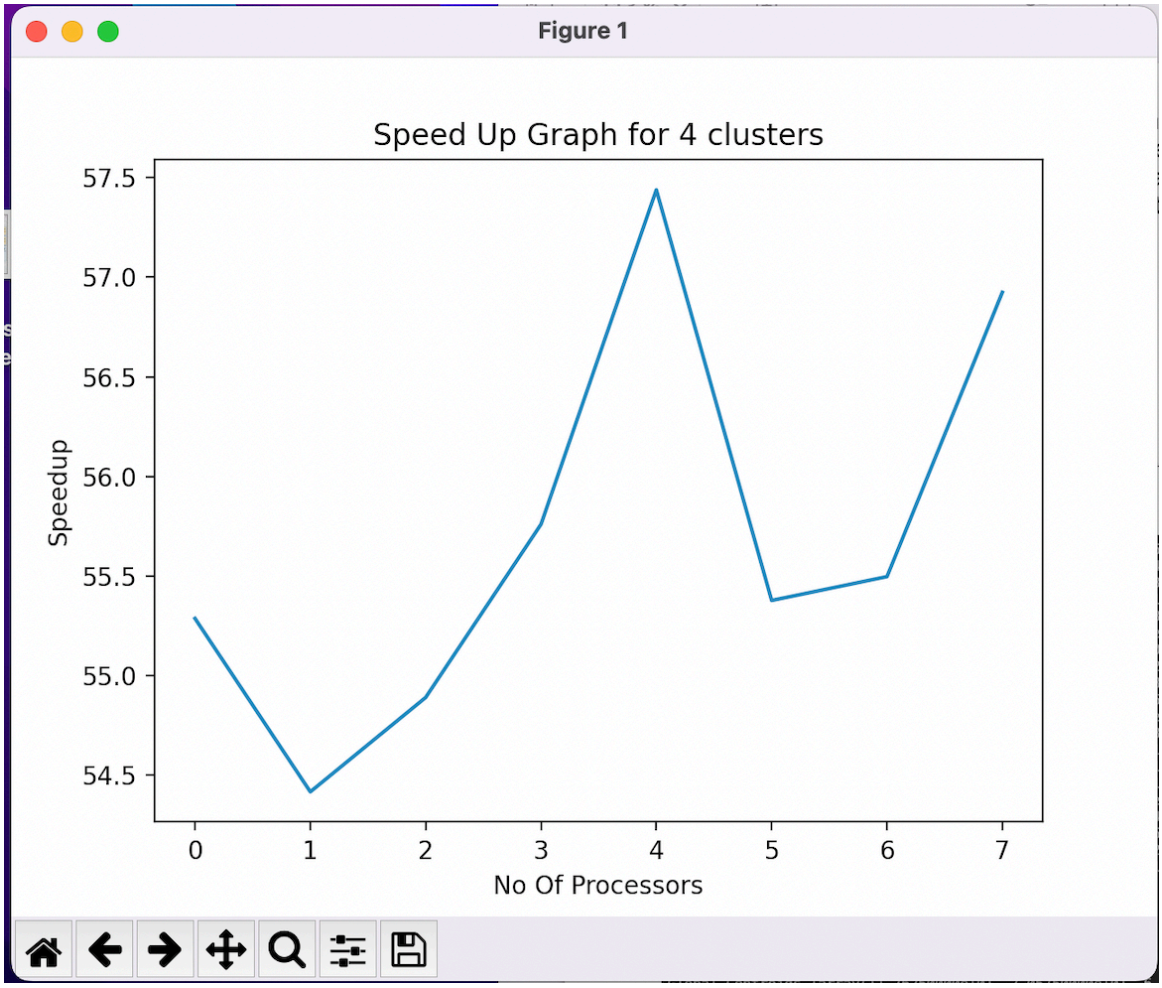
For k=4 clusters

When I run it on one core I get the following output in 51.64 seconds and 14 iterations

```
global avg is [array([1.76250000e+01, 2.05750000e+01, 6.36666667e+00, 3.89166667e+00,
       2.50000000e+00, 1.58845833e+02, 3.04833333e+01, 1.44666667e+01,
       3.75833333e+01, 2.51790317e+02, 9.51458333e+01, 3.33333333e-02,
       1.35416667e+00, 3.83333333e-01, 5.91666667e-01, 7.91666667e-02,
       7.91666667e-02, 8.18916667e+01, 1.73195833e+02, 2.73916667e+01,
       6.37083333e+00]), array([1.86819572e+01, 1.90519878e+01, 7.05504587e+00, 4.02752294e+00,
       2.57186544e+00, 2.53862385e+02, 2.94770642e+01, 1.20856269e+01,
       3.69388379e+01, 2.66612156e+02, 9.41131498e+01, 6.72782875e-02,
       1.25076453e+00, 1.50458716e+00, 5.77981651e-01, 4.58715596e-02,
       1.11620795e+00, 7.72996942e+01, 1.70168196e+02, 2.66238532e+01,
       7.04892966e+00]), array([1.51515152e+01, 1.55151515e+01, 6.90909091e+00, 3.63636364e+00,
       2.71212121e+00, 3.58590909e+02, 3.32727273e+01, 6.53030303e+00,
       3.06515152e+01, 2.69606970e+02, 9.47121212e+01, 1.21212121e-01,
       1.24242424e+00, 1.28787879e+00, 5.90909091e-01, 1.21212121e-01,
       1.93939394e+00, 7.66969697e+01, 1.74257576e+02, 2.51969697e+01,
       8.22727273e+00]), array([1.86355140e+01, 1.89532710e+01, 3.63551402e+00, 3.79439252e+00,
       2.45794393e+00, 1.77392523e+02, 2.59439252e+01, 1.34112150e+01,
       3.59906542e+01, 3.31746383e+02, 9.47102804e+01, 1.86915888e-02,
       1.30841121e+00, 7.94392523e-01, 4.67289720e-01, 1.12149533e-01,
       3.73831776e-01, 7.93738318e+01, 1.74317757e+02, 2.61495327e+01,
       6.98130841e+00])]
The global centroid is same....Exit the loop
total time for processes is  51.640524
Global Centroids [array([1.76250000e+01, 2.05750000e+01, 6.36666667e+00, 3.89166667e+00,
       2.50000000e+00, 1.58845833e+02, 3.04833333e+01, 1.44666667e+01,
       3.75833333e+01, 2.51790317e+02, 9.51458333e+01, 3.33333333e-02,
       1.35416667e+00, 3.83333333e-01, 5.91666667e-01, 7.91666667e-02,
       7.91666667e-02, 8.18916667e+01, 1.73195833e+02, 2.73916667e+01,
       6.37083333e+00]), array([1.86819572e+01, 1.90519878e+01, 7.05504587e+00, 4.02752294e+00,
       2.57186544e+00, 2.53862385e+02, 2.94770642e+01, 1.20856269e+01,
       3.69388379e+01, 2.66612156e+02, 9.41131498e+01, 6.72782875e-02,
       1.25076453e+00, 1.50458716e+00, 5.77981651e-01, 4.58715596e-02,
       1.11620795e+00, 7.72996942e+01, 1.70168196e+02, 2.66238532e+01,
       7.04892966e+00]), array([1.51515152e+01, 1.55151515e+01, 6.90909091e+00, 3.63636364e+00,
       2.71212121e+00, 3.58590909e+02, 3.32727273e+01, 6.53030303e+00,
       3.06515152e+01, 2.69606970e+02, 9.47121212e+01, 1.21212121e-01,
       1.24242424e+00, 1.28787879e+00, 5.90909091e-01, 1.21212121e-01,
       1.93939394e+00, 7.66969697e+01, 1.74257576e+02, 2.51969697e+01,
       8.22727273e+00]), array([1.86355140e+01, 1.89532710e+01, 3.63551402e+00, 3.79439252e+00,
       2.45794393e+00, 1.77392523e+02, 2.59439252e+01, 1.34112150e+01,
       3.59906542e+01, 3.31746383e+02, 9.47102804e+01, 1.86915888e-02,
       1.30841121e+00, 7.94392523e-01, 4.67289720e-01, 1.12149533e-01,
       3.73831776e-01, 7.93738318e+01, 1.74317757e+02, 2.61495327e+01,
       6.98130841e+00])] and no of iterations 14
```

With increase in the number of processors the time taken to compute decreases however the iterations remain the same. Therefore, we can conclude that the number of iterations I.e the number of times the clusters have to assigned for each instance does not depend upon the number of cores we are working with but rather with the number of clusters we have initialised.

```
global avg is [array([1.76250000e+01, 2.05750000e+01, 6.36666667e+00, 3.89166667e+00,
       2.50000000e+00, 1.58845833e+02, 3.04833333e+01, 1.44666667e+01,
       3.75833333e+01, 2.51790317e+02, 9.51458333e+01, 3.33333333e-02,
       1.35416667e+00, 3.83333333e-01, 5.91666667e-01, 7.91666667e-02,
       7.91666667e-02, 8.18916667e+01, 1.73195833e+02, 2.73916667e+01,
       6.37083333e+00]), array([1.86819572e+01, 1.90519878e+01, 7.05504587e+00, 4.02752294e+00,
       2.57186544e+00, 2.53862385e+02, 2.94770642e+01, 1.20856269e+01,
       3.69388379e+01, 2.66612156e+02, 9.41131498e+01, 6.72782875e-02,
       1.25076453e+00, 1.50458716e+00, 5.77981651e-01, 4.58715596e-02,
       1.11620795e+00, 7.72996942e+01, 1.70168196e+02, 2.66238532e+01,
       7.04892966e+00]), array([1.51515152e+01, 1.55151515e+01, 6.90909091e+00, 3.63636364e+00,
       2.71212121e+00, 3.58590909e+02, 3.32727273e+01, 6.53030303e+00,
       3.06515152e+01, 2.69606970e+02, 9.47121212e+01, 1.21212121e-01,
       1.24242424e+00, 1.28787879e+00, 5.90909091e-01, 1.21212121e-01,
       1.93939394e+00, 7.66969697e+01, 1.74257576e+02, 2.51969697e+01,
       8.22727273e+00]), array([1.86355140e+01, 1.89532710e+01, 3.63551402e+00, 3.79439252e+00,
       2.45794393e+00, 1.77392523e+02, 2.59439252e+01, 1.34112150e+01,
       3.59906542e+01, 3.31746383e+02, 9.47102804e+01, 1.86915888e-02,
       1.30841121e+00, 7.94392523e-01, 4.67289720e-01, 1.12149533e-01,
       3.73831776e-01, 7.93738318e+01, 1.74317757e+02, 2.61495327e+01,
       6.98130841e+00])]
The global centroid is same....Exit the loop
The global centroid is same....Exit the loop
The global centroid is same....Exit the loop
The global centroid is same....Exit the loop
total time for processes is  7.447089
Global Centroids [array([1.76250000e+01, 2.05750000e+01, 6.36666667e+00, 3.89166667e+00,
       2.50000000e+00, 1.58845833e+02, 3.04833333e+01, 1.44666667e+01,
       3.75833333e+01, 2.51790317e+02, 9.51458333e+01, 3.33333333e-02,
       1.35416667e+00, 3.83333333e-01, 5.91666667e-01, 7.91666667e-02,
       7.91666667e-02, 8.18916667e+01, 1.73195833e+02, 2.73916667e+01,
       6.37083333e+00]), array([1.86819572e+01, 1.90519878e+01, 7.05504587e+00, 4.02752294e+00,
       2.57186544e+00, 2.53862385e+02, 2.94770642e+01, 1.20856269e+01,
       3.69388379e+01, 2.66612156e+02, 9.41131498e+01, 6.72782875e-02,
       1.25076453e+00, 1.50458716e+00, 5.77981651e-01, 4.58715596e-02,
       1.11620795e+00, 7.72996942e+01, 1.70168196e+02, 2.66238532e+01,
       7.04892966e+00]), array([1.51515152e+01, 1.55151515e+01, 6.90909091e+00, 3.63636364e+00,
       2.71212121e+00, 3.58590909e+02, 3.32727273e+01, 6.53030303e+00,
       3.06515152e+01, 2.69606970e+02, 9.47121212e+01, 1.21212121e-01,
       1.24242424e+00, 1.28787879e+00, 5.90909091e-01, 1.21212121e-01,
       1.93939394e+00, 7.66969697e+01, 1.74257576e+02, 2.51969697e+01,
       8.22727273e+00]), array([1.86355140e+01, 1.89532710e+01, 3.63551402e+00, 3.79439252e+00,
       2.45794393e+00, 1.77392523e+02, 2.59439252e+01, 1.34112150e+01,
       3.59906542e+01, 3.31746383e+02, 9.47102804e+01, 1.86915888e-02,
       1.30841121e+00, 7.94392523e-01, 4.67289720e-01, 1.12149533e-01,
       3.73831776e-01, 7.93738318e+01, 1.74317757e+02, 2.61495327e+01,
       6.98130841e+00])] and no of iterations 14
```



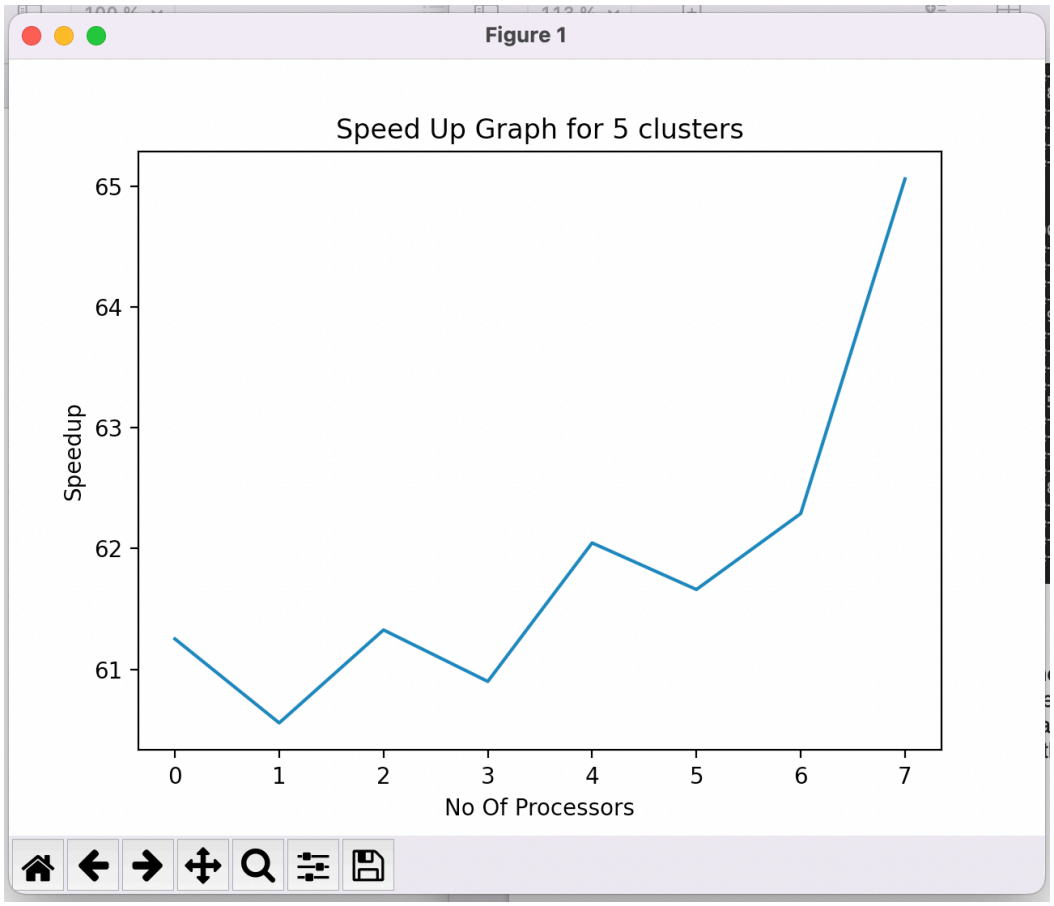


Figure 1
Speed Up Graph for 5 clusters
Speedup
65
64
63
62
61
0
1
2
3
4
5
6
7
No Of Processors

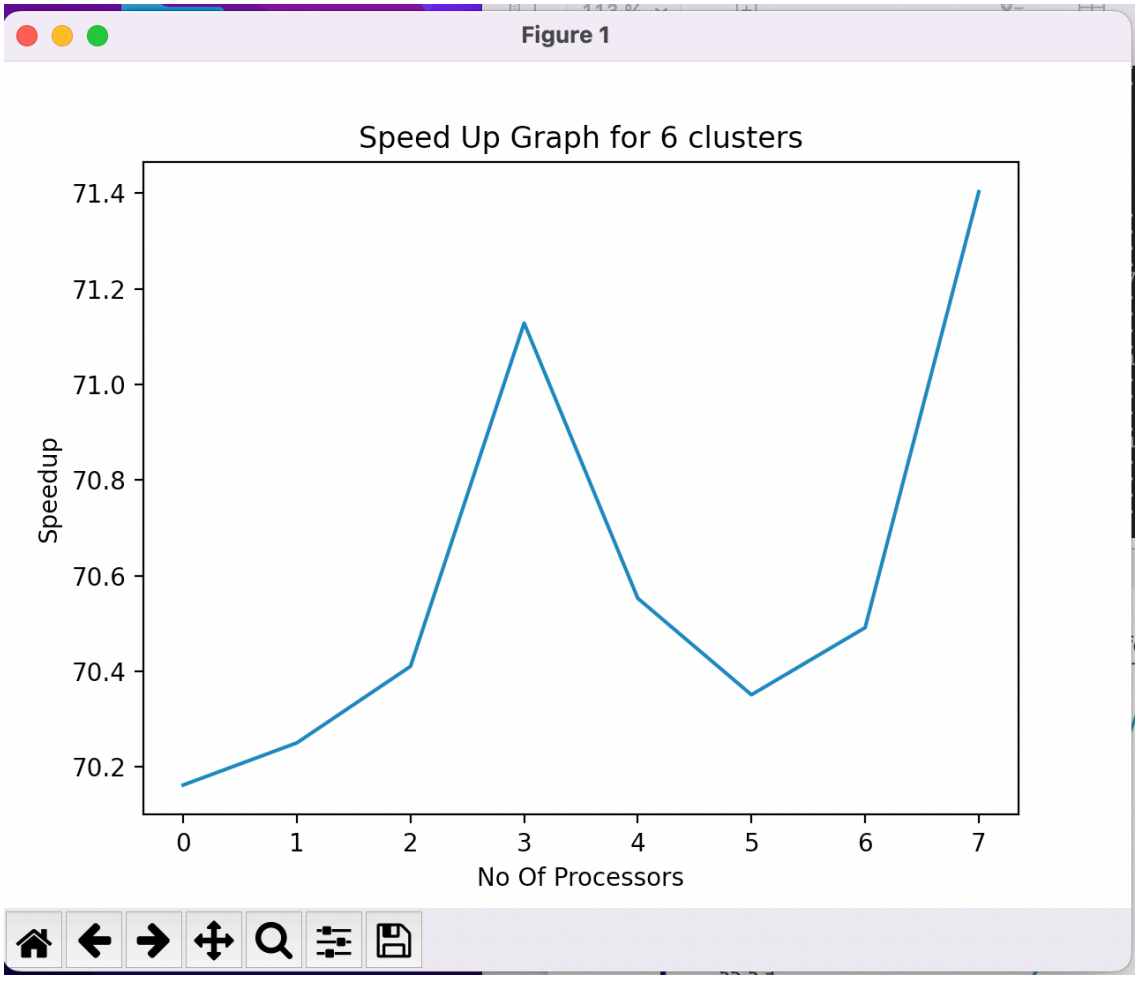


Figure 1
Speed Up Graph for 6 clusters
Speedup
71.4
71.2
71.0
70.8
70.6
70.4
70.2
0
1
2
3
4
5
6
7
No Of Processors