Lab Course: Distributed Data Analytics
Exercise Sheet 3

Task1: Distributed K-means Clustering

1) In K means clustering algorithm, required steps are
Step1-Initializing k centroids,
Step2-Forming k clusters by assigning the membership of the closest centroid to each data point based on euclidean distance.
Step3-Finding new centroids of new clusters.
Repeating steps 2 and 3 until convergence criterion is met.

Process followed for clustering: A function named **"k_clustering"** is created. This takes 2 arguments which are [allotted tf_idf data and broadcasted centroids] that are in sparse matrix form.

To create a **distance matrix,** Iterating through each point of tf_idf data(sparse matrix), the Euclidean distance between the point and each centroid is calculated and is appended into a list. This list is appended to a main list "d_main" for each point. This list is converted into a matrix of shape (N,k) called "d_mat" with N being the number of points in the allotted TF_IDF data and k being the number of clusters or centroids. The code for distance claculation is in Fig1.

```
def k_clustering(loc_tf_idf,gl_centroids):
    k,N=gl_centroids.shape[0],loc_tf_idf.shape[0]
    d_main=[]
    # Finding euclidean distance
    for i in range(N):
        # finding Euclidean distance between each point and each of k centroids into a list
        d=[np.sqrt((loc_tf_idf[i]-gl_centroids[j]).power(2).sum()) for j in range(k)]
        d_main.append(d)
    d_mat=np.array(d_main)
```

Figure1: Euclidena distance claculation

To find **membership**, np.argmin() is used on this "d_mat" to get an array of indices of closest centroids for each point.

To form **clusters** according to the memberships, np.nonzeros(membership==cluster number) is used. This reutuns the indices havng the given cluster number. Using these indices, clusters are formed from allotted tf_idf data.
To find **new centroids,** if the size of newly formed cluster is not zero, then the new centroid is calculated using cluster.mean(axis=0). If the new cluster have no elements, then old corresponding centroid is taken as the new centroid. The code for calculating new mebership,clusters and new centroids is in Figure 2

```
# Finidng membership
membership=(np.argmin(d_mat, axis=1)).reshape((-1,1))
cluster_l,new_centroid_l=[],[]
#clustring and finding new centroids
for m in range(k):
    cluster=loc_tf_idf[np.nonzero(membership == m)[0]]
    cluster_l.append(cluster)
    if cluster.shape[0]!=0:
        new_centroid_l.append(cluster.mean(axis=0))
    else:
        new_centroid_l.append(gl_centroids[m].todense())
return new_centroid_l,membership
```

Figure 2: calculating new mebership,clusters and new centroids

The newly formed centroids and membership array is returned by each processor to master to find new global centroid. For description convenience, the total function **"k_clustering"** is presented as 2 parts each in Figure 1 and figure 2 respectively.

2) Parllelizing K-means clustering using MPI

i)The MPI communicator is initialised and rank and number of workers p are initialised. Processor with rank=0 is the master.

ii) **Data loading and centroid initilaization:** In the master, list of raw docs are loaded from sklearn using " fetch_20newsgroups(subset='all')". The raw docs are converted to TF_IDF sparse matrix using "TfidfVectorizer()" and "fit.transform()". K number of random global centroids are picked from the "tf_idf" matrix. The code for loading data and initializing global centroids is in figure 3.

```
if __name__ == '__main__':
    comm = MPI.COMM_WORLD
    p = comm.Get_size()
    rank = comm.Get_rank()
    if rank == 0:
        # number of clusters
        k = 8
        # Data loading
        corpus = fetch_20newsgroups(subset='all')
        data=corpus['data']
        vectorizer = TfidfVectorizer()
        tf_idf = vectorizer.fit_transform(data)
        N=tf_idf.shape[0]
        np.random.seed(8)
        # Intitializing K number of centroids
        C_index=np.random.randint(N, size=k)
        gl_centroids = tf_idf[C_index]
```

Figure 3: Data loading and initializing random k global centroids from "tf_idf" matrix

iii) **Data and centroids distribution:** Instead of splitting the data in master, the **tf_idf matrix** is broad casted to all processors using **comm.bcast()**. Timer starts before this step. The total number of rows i.e “N” of tf_idf matrix is splitted as per the number of processors “p” using “np.array split”. Then the splitted **tf_idf indices** named “indices_split” is scattered among all workers including master using **com.scatter()** of collective communication. In this way, each worker can slice his chunk of data instead of master doing the slicing iterating through whole data. This saves distribution time.The k number of **global centroids** are broadcasted using **comm.bcast()**. The distribution of tf_idf, centroids,tf_idf row indices is in Figure 4

```
        #splitting indices in master
        indices_split=np.array_split(list(range(0,N)),p)
    else:
        indices_split=None
        tf_idf=None
        gl_centroids=None

    time_start = MPI.Wtime()
    # broad casting the tf_idf matrix
    tf_idf=comm.bcast(tf_idf,root=0)
    # scattering the indices
    indices_split = comm.scatter(indices_split, root=0)
    tolerance,counter=2,0
    while tolerance>1:
        counter+=1
        # broad casting the global centroids
        gl_centroids=comm.bcast(gl_centroids,root=0)
```

Figure4: Broad casting TF_IDF matrix and global centroid, scattering TF_IDF row indices

iv) **Execution and gathering:** Each worker including master receives the tf_idf row indices and global centroids and then creates their own local_tf_idf matrix based on these indices and works on it using k means clustering function i.e “k_means()”. This function returns a new local centroids and membership details of their allotted data points. All these returned results from each worker and master are gathered individually using **comm.gather().** The gathered two lists are **“new_membership_l”** & **“new_centroids_l”.** The code is in FIgure 5.

```
while tolerance>1:
    counter+=1
    # broad casting the global centroids
    gl_centroids=comm.bcast(gl_centroids,root=0)
    #calculating new centroids and memberships in each processor
    new_centroids,new_membership=k_clustering(tf_idf[indices_split],gl_centroids)
    #gatherign the results  for membership and local centroids from all workers and master
    new_membership_l=comm.gather(new_membership,root=0)
    new_centroids_l=comm.gather(new_centroids,root=0)
```

Figure5: Executing k_means clustering and gathering results.

v) Finding new global centroids: The “new_centroids_l” is a nested list of local new centroids list. Therefore in master, iterating through each cluster number, each of the centroids from each local list of “new_centroids_l” are taken out and are verticaclly stacked and mean is evaluated using using np.mean(axis=0). All such calculated k number of global centroids are again vertically stacked to form new global centroids. The old global centroids are assigned to variable “old_gl_centroids” and the new one are converted to sparse matrix and assigned to “gl_centroids”. The gathered memberships from each worker are also vertically stacked to get an array of size (N,1). The codes for finding new global centroids is in Figure 6.

```
if rank == 0:
    new_membership_=np.vstack(new_membership_l)
    new_gl_centroid = []
    # Finding new global centroids
    for cluster_num in range(k):
        #grouping centroids of specific cluster number from all centroids list
        cl_centroid_list = [l[cluster_num] for l in new_centroids_l]
        # Stacking and finding the mean of grouped centroids
        cl_centroids_vec = np.vstack(cl_centroid_list)
        new_gl_centroid.append(np.mean(cl_centroids_vec, axis=0))
    # vertically stacking the k number of new global centroid.
    new_gl_centroid = np.vstack(new_gl_centroid)
    old_centroids = gl_centroids
    gl_centroids = csr_matrix(new_gl_centroid)
```

Figure 6: The codes for finding new global centroids.

vi) Repeating until convergence: The steps of iv and v that includes broad casting global centroids, calculating local centroids by clustering, gathering and averaging the centroids to get new global centroids, again broadcasting global centroids are repeated until a convergence criteria is met. This criteria is taken from the “cluster-kmeans” algorithm provided in “ml_cluster_analysis” pdf in learnweb i.e

$$\text{until} \quad \frac{1}{K}\sum_{k=1}^{K} ||\mu_k - \mu_k^{\text{old}}|| < \epsilon$$

In master, after calculating new global centroids, as per the above formula, tolerance between old and new global centroids is calculated as “tolerance=abs(gl_centroids-old_centroids).sum()/k**”.** The maximum value of the tolerance is set to 1. If the tolerance is greater than 1, tolerance is broadcasted and then the steps iv and v are repeated using while loop. If the tolerance is less than 1, convergence is met. Timer stops here. Master writes number of processors, clusters and total time taken and number of iterations taken to “time.txt” file. Also when p=4, the resultant membership vector is loaded to “membership_{cluster number}.txt” file for reference, using pickle.load(). The convergence code and final steps after converging is presented in figure 7.

```
    # CONVERGENCE
    if tolerance <= 1:
        end = MPI.Wtime()
        total_time=round(MPI.Wtime() - time_start, 3)
        print(
            f"time taken for clustering with processors={p}  and cluster number ={k} is {total_time}")
        with open('time.txt', 'a') as f:
            f.write(f'{p} {k} {total_time} {counter}\n')
        with open('membership_2.pickle', 'wb') as data:
            pickle.dump(new_membership_, data)
else:
    tolerance = None
tolerance = comm.bcast(tolerance, root=0)
```

Figure 7 :The convergence code and final steps after converging.

3) **Crosschecking**: To cross check the resultant memberships of parallel execution, size N=1000 is taken and 8 clusters are taken and executed. Then the final new membership obtained serially i.e with one processor and parallelly with 4 processors are loaded into 2 .pickle files. These are reloaded and sum of absolute values of the differences between 2 memberships obtained serially and parallelly is found to be zero. This shows that the results are correct using parallelization. The code used for cross checking is given in figure 8a and result is figure 8b.

```
                with open('membership4.pickle', 'wb') as data:
                    pickle.dump(new_membership_, data)
#         else:
#             tolerance = None
#         tolerance = comm.bcast(tolerance, root=0)

with open('membership4.pickle', 'rb') as f:
    membership4 = pickle.load(f)
with open('membership4.pickle', 'rb') as f:
    membership1 = pickle.load(f)
sum_of_diff=np.sum(abs(membership1-membership4))
print(f"difference between serially and paralelly computed memberships is {sum_of_diff}")
```

Figure 8a: The code used for cross checking the result of membership obtained serially and parallelly.

```
/usr/bin/python3.8 /home/yalavarthi/Desktop/LAB4/KMEANS_D.py
difference between serially and paralelly computed memberships is 0
```

Figure 8b: Cross checking result

4) **Result:** Taking processors =4, and clusters=4 the final 4 centroids are printed out and number of data points assigned to each cluster is printed out in 8c

```
**************************************************
the centroid matrix of 4 number of centroids is
[[1.62653432e-03 2.20601246e-03 1.22840238e-05 ... 1.40990502e-05
  0.00000000e+00 1.56995998e-05]
 [4.32223172e-03 4.30398213e-04 1.24242621e-04 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [6.62323236e-03 1.38553583e-03 5.86673928e-05 ... 0.00000000e+00
  6.88535090e-05 0.00000000e+00]
 [1.54935505e-03 3.54774225e-03 2.47169888e-05 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]]
**************************************************
Total number of members for each cluster is
Counter({0: 6978, 3: 5526, 2: 5328, 1: 1014})
```

Figure 8c: Result showing the centroid matrix of 4 clusters and total number of members allotted for each cluster.

Task2: Performance Analysis:

1) **Experimenting with number of workers p = {1, 2, 3, 4 }. (**My pc have only 4 Logical processors. I used "–hwthread–cpus" to run mpi. This allowed me upto 4 threads.

Approach: The clustering is repeated with p=1 to 4 for different clusters [2,4,6,8]. The total time taken is observed to be increasing when processors are increased. The observed timings for each number of clusters and each number of processors is given in Figure 9a. The same is given in the table format in Figure 9b

```
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 1 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=1  and cluster number =2 is 1000.154
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 2 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=2  and cluster number =2 is 865.968
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 3 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=3  and cluster number =2 is 673.009
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 4 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=4  and cluster number =2 is 608.276
```

```
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 1 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=1  and cluster number =4 is 2038.814
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 2 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=2  and cluster number =4 is 1737.84
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 3 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=3  and cluster number =4 is 1381.552
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 4 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=4  and cluster number =4 is 1272.176
```

```
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 1 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=1  and cluster number =6 is 2338.408
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 2 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=2  and cluster number =6 is 2013.637
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 3 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=3  and cluster number =6 is 1632.761
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 4 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=4  and cluster number =6 is 1533.671
```

```
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 1 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=1  and cluster number =8 is 2877.38
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 2 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=2  and cluster number =8 is 2498.844
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 3 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=3  and cluster number =8 is 2014.345
(bigdata) yalavarthi@yalavarthi:~/Desktop/LAB4$ mpirun -np 4 --use-hwthread-cpus python KMEANS_D.py
time taken for clustering with processors=4  and cluster number =8 is 1924.4
```

Figure 9a: Recorded timings for number of clusters k= [2,4,6,8] for p={1, 2,3,4}

Clusters k	k=2	k=4	k=6	k=8
Processor p				
1	1000.154	2038.814	2338.408	2877.38
2	865.968	1737.84	2013.637	2498.844
3	673.009	1381.552	1632.761	2014.345
4	608.276	1272.176	1533.671	1924.4

Figure 9b: Table showing execution timings for number of clusters k= [2,4,6,8] for p={1, 2,3,4}

Performance Analysis: For **each cluster number,** the total time taken is observed to be **decreasing** when processors are increased from p=1 to 4. As the processors increased from 1 to 3, big saving in time is observed. This is due to the large difference in the data share per each processor and there by each worker and master sharing the calculation of euclidean distance and clustering the data.

But as the p is increased from 3 to 4, there is no big difference in timings. This may be due to small variation of data shared among processors when processors are increased only by one number from 3 to 4. If we try with 6 or 8 processors, then the timing may further improve. Overall on an average over all clusters, there is around 35% savings in time when processors are increased from p=1 to 4 showing effective parallelization. The cross checking result in figure 8b also shows the rightfullness of the parallelization.

One more point is **as the clusters are increased, on each processor**, **tme taken is increased.** This is shown in table of figure 9b and in graph of figure 9c. This is due to increase in number of centroids increasing computational overhead over each worker and master in terms of distance and membership claulations for each data point to the increased number of centroids and also due to gathering and averaging the increased number of centroids.

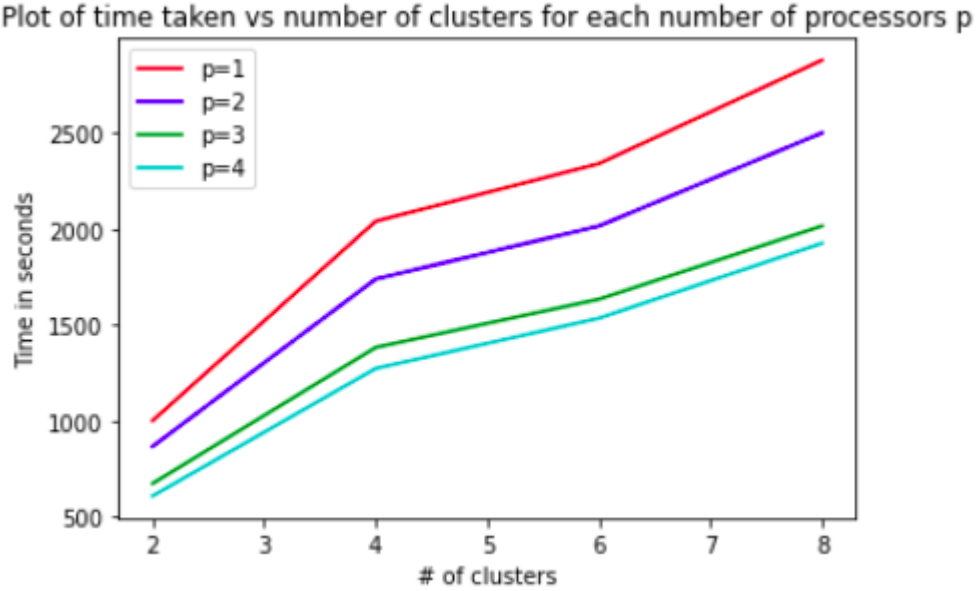


Figure 9c: Plot showing execution timings vs number of clusters [2,4,6,8] for each p={1, 2,3,4}

Speedup analysis is followed below.

Plotting speedup graph

Approach: 1) The timings recorded stored in time.txt file is loaded into a data frame named "time".

In [210…
```
time=pd.read_csv("time.txt",sep=" ",header=None ,names=['proc_p', 'clust_k', 'time(s)','iterations'])
print(" Data frame showing the timiings recorded and iterations took for converging agianst processors and clusts
time
```

```
 Data frame showing the timiings recorded and iterations took for converging agianst processors and clustsers
```

Out[210…

	proc_p	clust_k	time(s)	iterations
0	1	2	1000.154	4
1	2	2	865.968	4
2	3	2	673.009	4
3	4	2	608.276	4
4	1	4	2038.814	6
5	2	4	1737.840	6
6	3	4	1381.552	6
7	4	4	1272.176	6
8	1	6	2338.408	6
9	2	6	2013.637	6
10	3	6	1632.761	6
11	4	6	1533.671	6
12	1	8	2877.380	6
13	2	8	2498.844	6
14	3	8	2014.345	6
15	4	8	1924.400	6

- Speedup $S_p = \frac{T_s}{T_p}$
  - P = # processes
  - Ts = Best serial execution time
  - Tp = execution time on P processes
  - Sp = Speedup on P processes

Calculation of Speedup

1) The speedup "Sp" is calculated using the formula above
2) The serial execution time list is taken into "Ts_l" by filtering the timings with processor=1
3) All times T_p for each cluster against each processor from p=1 to 4 is taken by applying necessary conditions

4) In each cluster, Serial execution time Ts is divided by Tp of each processor and stored in a dictionary with keys as clusters and key values as speedup values against each processor p=1 to 4. For p=1, Sp is always 1.

In [211…
```
Ts_l=list(time.loc[(time['proc_p']==1)]["time(s)"])
Ts_l
```

Out[211…
```
[1000.154, 2038.814, 2338.408, 2877.38]
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [214…
```
cluster=[2,4,6,8]
S_p={}
for i,k in enumerate(cluster):
    T_p=np.array(time.loc[(time['clust_k']==k)]["time(s)"])
    S_p[k]=np.round(Ts_l[i]/T_p,4)
S_p
```

Out[214…
```
{2: array([1.    , 1.155 , 1.4861, 1.6442]),
 4: array([1.    , 1.1732, 1.4757, 1.6026]),
 6: array([1.    , 1.1613, 1.4322, 1.5247]),
 8: array([1.    , 1.1515, 1.4284, 1.4952])}
```

The same is shown in the table below

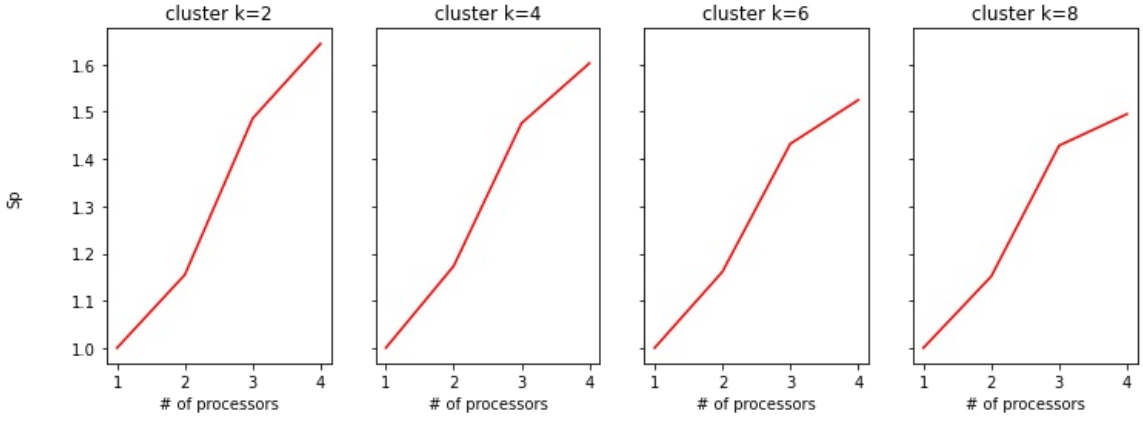
```
### Table showing the speedup Sp values for each cluster against each processor
```

Processor p	p=1	p=2	p=3	p=4
Clusters k				
2	1	1.155	1.4861	1.6442
4	1	1.1732	1.4757	1.6026
6	1	1.1613	1.4322	1.5247
8	1	1.1515	1.4284	1.4952

In [215…
```
print( "Title:              Plots of speedup vs number of processor for each cluster                ")

fig, axs = plt.subplots(1,4,sharey=True,figsize=(12,4))
axs = axs.ravel()
fig.text(0.05, 0.5, 'Sp', va='center', rotation='vertical')
for i,k in enumerate(cluster):
    axs[i].set_title(f"cluster k={k}")
    axs[i].plot([1,2,3,4], s_p[k],'r',label=f"cluster k={k}")
    axs[i].set_xlabel('# of processors')
plt.show()
```

```
Title:              Plots of speedup vs number of processor for each cluster
```

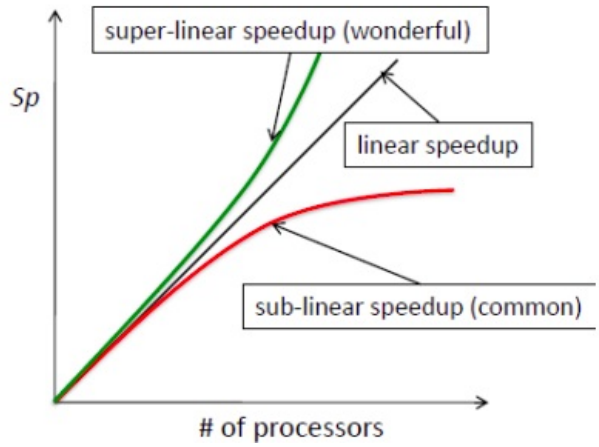


In [ ]:

Analysis on speedup graph:

1))The speed up graphs of Sp vs number of processors for each cluster of [2,4,6,8] is plotted using matplotlyb.
2) It can be seen that, overall the speedup graph is increasing showing the benefit of parallelization. The slope of the cureve is increasing upto 3rd processor. The steepness of speedup is high when processors is increased from p=2 to p=3. This may be due to great difference in distribution of workload on each worker, by disributing the data to be worked on and the process of clustering. But the slope decerased when processors increased from p= 3 to p=4. This is may be due to small variation of data shared among processors when processors are increased only by one number from 3 to 4. Hence it can be seen that, parallelizing with 4 processors may not yield much benefit than taht of with processors p=3. One can get approximately same benefit by chosing only upto 3 processors without spending cost and time on including 4 th processor. Comparing this graph with the common sub-linear spedup graph shown below, we may see more decline in the steepness of

speedup if we increase the processors from p=4 to 6 or 8.



In [ ]: