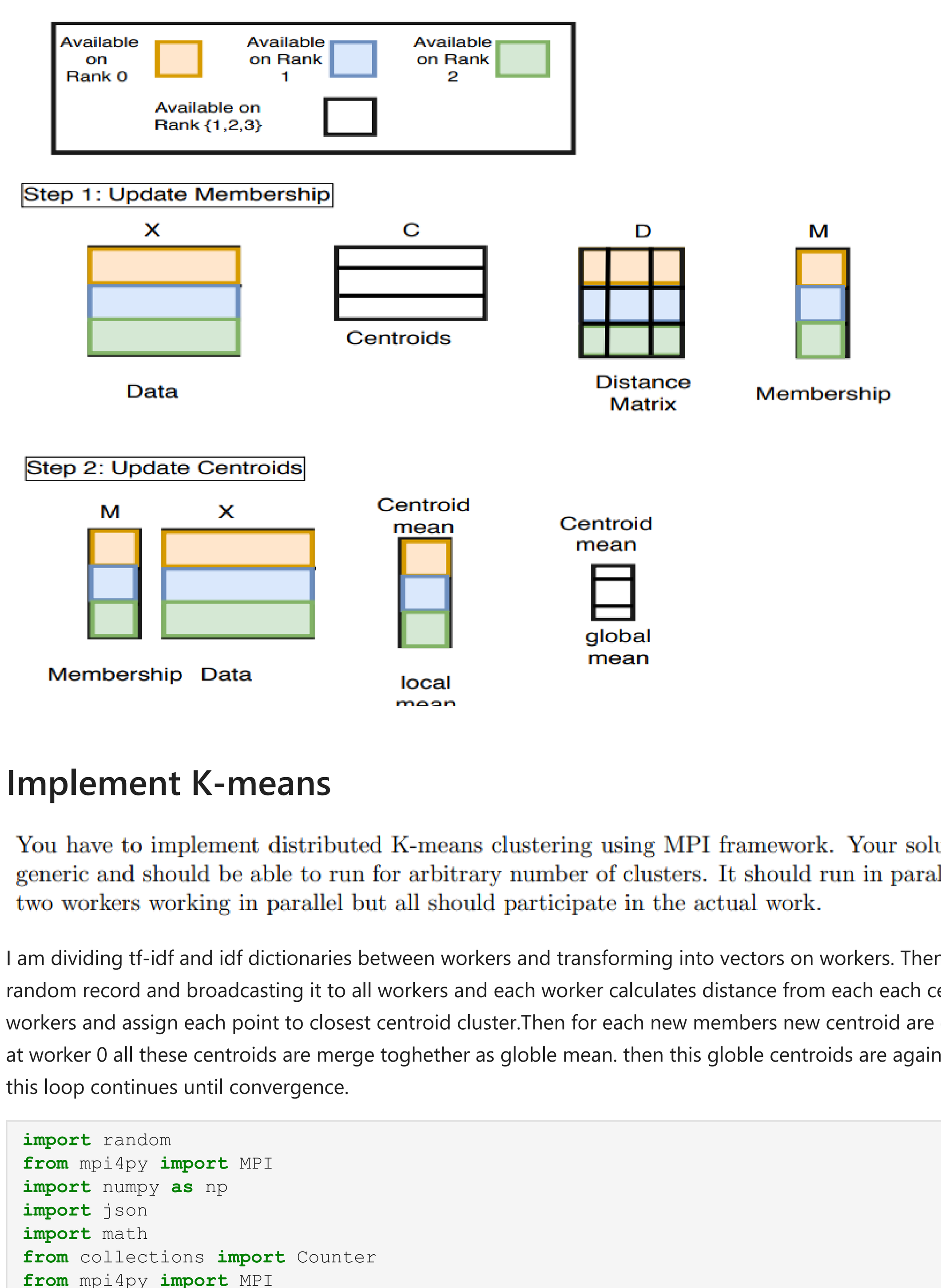
Lab Course: Distributed Data Analytics
Exercise Sheet 3

Syed Wasif Murtaza Jafri- 311226

Distributed K-means Clustering

The k-means algorithm clusters the data instances into k clusters by using euclidean distance between data instances and the centroids of the clusters. The detail description of the algorithm is listed on slides 1-10 `https://www.ismll.uni-hildesheim.de/lehre/bd-16s/exercises/bd-02-lec.pdf`. However, in this exercise sheet you will implement a distributed version of the K-means. Figure below explains a strategy to implement a distributed K-means.
Suppose you are given a Dataset $X \in \mathbb{R}^{M \times N}$ and a random initial centroids $C \in \mathbb{R}^{M \times K}$, where M are the number of features of a Data instance, N are the number of Data instances and K the number of clusters (K **is a hyperparameter**). Lets assume you want to implement a distributed version with 3 workers. (**Note** your solution should be generic and should work with any number of workers.) In the figure below three workers are given colors i.e. Rank 0 = orange, Rank 1 = blue, and Rank 2 = green. If a data is represented in white color this means it must be available on all the workers. The algorithm progress as

- Initialize K centroids
- Divide Data instances among P workers (X shown in figure, different colors represent parts at different workers)
- Unitill converge
  - step 1
    - calculate distance of each Data instance from each centroid using the euclidean distance. (populate Distrance matrix shown in the figure)
    - Assign membership of each data instance using the minimum distance in the distance matrix.
  - step 2
    - Each worker calculates the new centroids (local means) using the new membership of data instance.
    - collect updated centroids (local mean) information from each worker and find the global centroids (global mean).
    - redistribute new centroids of clusters to each worker.



Implement K-means

You have to implement distributed K-means clustering using MPI framework. Your solution should be generic and should be able to run for arbitrary number of clusters. It should run in parallel i.e. not just two workers working in parallel but all should participate in the actual work.

I am dividing tf-idf and idf dictionaries between workers and transforming into vectors on workers. Then iniitailization centroid with random record and broadcasting it to all workers and each worker calculates distance from each each centroid of every point in that workers and assign each point to closest centroid cluster.Then for each new members new centroid are calculated by taking mean and then at worker 0 all these centroids are merge toghether as globle mean. then this globle centroids are again broadcasted to all workers and this loop continues until convergence.

In [ ]:

```python
import random
from mpi4py import MPI
import numpy as np
import json
import math
from collections import Counter
from mpi4py import MPI
import numpy as np
np.set_printoptions(threshold=10000000)
np.random.seed(115)

comm = MPI.COMM_WORLD  # setting up the MPI communicator
total_worker = comm.Get_size()  # getting number of workers
rank = comm.Get_rank()  # storing rank of each worker
start_time = MPI.Wtime()  # time variable to find time at the end
master = 0  # master with id=0
converge = 0 # for checking convergence condition
iteration = 0 # to run epoxs
max_iter = 500 # total number of iteration
def randomCentroid(data, k):
    randomRows = np.random.randint(data.shape[0], k)
    centroid = data[randomRows]

    return centroid

def initailizeRandomCentroid(tfIdfDict, cleanIdfDict,k):
    tfIdfDictKeys = random.sample(list(tfIdfDict),k)
    tfIdfDictRand = {k:tfIdfDict[k] for k in tfIdfDictKeys}
    centroid = transformData(tfIdfDictRand,cleanIdfDict)

    return centroid


def euclidean_distance(dataPoint, centroid):
    '''returns euclidean distance between two instances'''
    return np.sqrt(np.sum((np.array(dataPoint) - np.array(centroid)) ** 2))


def calculateDistanceMatrix(tfIdfVectorsChunk, centroids):
    '''returns distance matrix which contains distances of each points from each centroids,
    Every row of distance matrix represents an instance and each columns represents centroids'''

    rows = tfIdfVectorsChunk.shape[0]
    k = centroids.shape[0]
    distanceMatrix = np.zeros((rows, k))
    for r, record in enumerate(tfIdfVectorsChunk):
        for c, centroid in enumerate(centroids):
            distanceMatrix[r][c] = euclidean_distance(record, centroid)
    return distanceMatrix

def assignMembership (distanceMatrix):
    '''In distance matrix, for every row instance get minimum value in that row which tells which is closet cer
    Assign that column index as cluster of that row.Return this matrix of assignments'''
    rows= distanceMatrix.shape[0]
    membershipVector = []
    for i in range(rows):
            membershipVector.append([distanceMatrix[i].argmin()])

    return np.array(membershipVector)


def transformData(tfIdfDict, cleanIdfDict):
    '''
    Transforn tf-idf dictionary to vector form. where every row represents on document and
    there is column represent on token word in the corpus. Each entry in this matrix is tf-idf of
    that document for a perticular word.

    '''
    cols = len(cleanIdfDict)  # tokens as M features
    rows = len(tfIdfDict)  # number of Documents as N rows in dataset

    tf_idf_mat = np.zeros((rows, cols))
    r = 0  # for iterating over rows
    for doc in tfIdfDict:
        c = 0 # for iterating over columns
        for token in cleanIdfDict:

            # if there is a token in doc, then place tf-idf of token else zero value will remain
            if (tfIdfDict[doc].get(token) != None):
                tf_idf_mat[r][c] = tfIdfDict[doc].get(token)

            c += 1

        r += 1


    return tf_idf_mat


if rank == master:  # Condition of checking master to distribute data
    # reading tf-idf scores stored in last exercise
    tfIdfDict = dict((json.load(open("tf_Idf.txt")).items()))

    # reading idf files so that we know how many terms are there in total
    # and these are then converted into feature
    # columns M
    idfDict = dict((json.load(open("IDF.txt")).items()))


    # removing tokens with idf=0 because their tf-idf will be 0 and wont help in clustering.
    cleanIdfDict = {key: val for key, val in idfDict.items() if val != 0}

    chunk_size = len(tfIdfDict) // total_worker  # each chunk size processed by worker

    k = 2
    # initializing centroid randomly
    #centroids = np.random.uniform(0.5,1,size=(k,len(cleanIdfDict)))
    centroids = initailizeRandomCentroid(tfIdfDict,cleanIdfDict,k)

    # splitting tf-idf dictionary into chunks
    tfIdfDictChunkList = []
    for i in range(1, total_worker+1):  # for loop to send data
        # slicing tf dictionary for sending to workers
        tfIdfDictChunk = {'document' + str(k): tfIdfDict['document' + str(k)] for k in
                          range(((i - 1) * chunk_size) + 1, (i * chunk_size) + 1)
                          if 'document' + str(k) in tfIdfDict}

        tfIdfDictChunkList.append(tfIdfDictChunk)
    lossList =[]
else:
    tfIdfDictChunkList = None
    cleanIdfDict = None
    tran_tf_idfs = None
    centroids = None


# everyprocess recieves one chunk  tf-idf dictionary
tfIdfDictChunk = comm.scatter(tfIdfDictChunkList,root=0)

# broadcasting Idf dictionary for column features
idfDict = comm.bcast(cleanIdfDict, root=0)

# transforming dictionary chunk into vector
tfIdfVectorsChunk = transformData(tfIdfDictChunk, idfDict)

# broadcasting initial centroids
centroids = comm.bcast(centroids, root=0)



while (iteration < max_iter and converge == 0):

    new_Centroids = centroids.copy()

    # distance matrix of shape(numberOfPoints,number of columns) where each rows tell the distance from each ce
    distanceMatrix = calculateDistanceMatrix(tfIdfVectorsChunk, new_Centroids)

    # membership vector of shape(numberOfPoints,1) where each row tells which is closest centroid
    memberShipVectorChunk = assignMembership(distanceMatrix)
    totalloss= np.zeros(1)
    loss = 0
    for i in range(len(memberShipVectorChunk)):
        loss += euclidean_distance(tfIdfVectorsChunk[i],new_Centroids[memberShipVectorChunk[i]])
    comm.Reduce(loss,totalloss,op=MPI.SUM, root=0)
    numberOfClusters , features = centroids.shape

    '''# for each cluster, finding member points index and getting those points
    and adding each feature values of those member points and
    dividing with total number of members to get the local mean of that cluster'''
    for i in range(numberOfClusters):

        centroidKIndex = np.where(memberShipVectorChunk== i) # gives indexes of cluster member

        if len(np.array(centroidKIndex)[0]) !=0:
            new_Centroids[i]= np.sum(tfIdfVectorsChunk[np.array(centroidKIndex)[0],:], axis=0,keepdims=True)/le

    # gathering local means at master worker
    localMeansList = comm.gather(new_Centroids, root=0)

    if rank == master:

        # calculating globle centroid by taking mean of local centroids returns by each worker
        updatedCentroids = np.empty([numberOfClusters,features])
        centroidDistance = 0
        for k in range(numberOfClusters):
            i = 0
            for localMean in localMeansList:
                if i == 0:
                    globleMeanNp = localMean[k,:].reshape(1,features)
                else:
                    globleMeanNp = np.concatenate((globleMeanNp,  localMean[k,:].reshape(1,features)), axis=0)

                i+=1

            updatedCentroids[k] = np.sum(globleMeanNp, axis=0,keepdims=True)/(total_worker)

            # calculating distance between new centroids and last iteration centroids
            centroidDistance += euclidean_distance(updatedCentroids[k],centroids[k])

        if(centroidDistance == 0): # checking if centroids are not moving, then it means it is converged and co
            converge = 1
        print('------------Iteration '+str(iteration)+'---------')
        print('Distance Btw new and old centroids:',centroidDistance)
        print('totalLoss:',totalloss)
        lossList.append(totalloss[0])
    else:
        updatedCentroids = None
        centroids = None
    # updating centroids
    centroids = updatedCentroids
    # broadcasting new centroids
    centroids = comm.bcast(centroids, root=0)
    iteration = iteration + 1
    # broadcasting convergence variable so that all workers come out of loop
    converge = comm.bcast(converge, root=0)


if rank == 0:

    clusteringTime =   MPI.Wtime()-start_time
    print('Total Time :', clusteringTime)
    mode = 'w'  if total_worker==1 else 'a'
    if(total_worker==1):
        with open('loss.txt', mode) as f:
            f.write(lossList + '\n')
    with open('Timefork'+str(numberOfClusters)+'.txt', mode) as f:
        f.write(str(total_worker) +','+str(clusteringTime) + '\n')
```

```
PS C:\Users\wasif\DDA Lab\Exercise03\Jafri_311226_Exercise3_code> mpiexec -n 4 python .\Exercise1Clustering.py
------------Iteration 0---------
Distance Btw new and old centroids: 1.258059762818474
totalLoss: [9478.2671128]
------------Iteration 1---------
Distance Btw new and old centroids: 0.18142732889662108
totalLoss: [5965.23604668]
totalLoss: [5964.39123203]
------------Iteration 7---------
Distance Btw new and old centroids: 0.0010528962842819354
totalLoss: [5964.38887764]
------------Iteration 8---------
Distance Btw new and old centroids: 0.00026322407107048516
totalLoss: [5964.38839]
------------Iteration 9---------
Distance Btw new and old centroids: 6.580601776762205e-05
totalLoss: [5964.38827441]
------------Iteration 10---------
Distance Btw new and old centroids: 1.6451504441907532e-05
totalLoss: [5964.3882459]
------------Iteration 11---------
Distance Btw new and old centroids: 4.11287611047412e-06
totalLoss: [5964.3882388]
Total Time : 108.25868139999875
```

Loss function is defined as:

$$\mathcal{L} = \sum_{i=1}^{m} \sum_{k=1}^{K} 1\{c_i = k\} ||x_i - \mu_k||^2$$
$$= \sum_{i=1}^{m} ||x^i - \mu_{c^i}||^2$$

As seen from results, total loss is decreasing with every iteration.

Performance Analysis

You have to do a performance analysis and plot a speedup graph. First you will run your experiments with varying number of clusters i.e. $P = \{1, 2, 4, 6, 8\}$. To plot the speedup graph please follow the lecture slides 15 (`https://www.ismll.uni-hildesheim.de/lehre/bd-16s/exercises/bd-02-lec.pdf`).

In [6]:

```python
import pandas as pd
worker_cluster = {'k:1':[78.7704605999752,44.058279899996705,30.054899000009755,26.390722500014817,25.79847229
                  'k:2': [79.25233600000502,47.68594880000455,34.423739499994554,30.0705275000073,28.65797420000
                  'P:4': [84.02475839998806,60.09628399999929,46.92675479999161,40.57284479998634,35.58097149999
                  'P:6': [87.92940929997712,67.69387079999433,54.03667519998271,45.67440719998558,44.0670922999
                  'P:8': [92.11813620000612,78.6590625999961,62.42019859998254,53.60897220001789,53.9892783999
     }

time = pd.DataFrame.from_dict(worker_cluster, orient = 'index')
time.columns = ['p:1', 'p:2', 'p:3', 'p:4','p:5', 'p:6', 'p:7', 'p:8']
time2 = time.style.set_properties(**{
    'background-color': 'white',
    'font-size': '16pt',
})
time2
```

Out[6]:

	p:1	p:2	p:3	p:4	p:5	p:6	p:7
k:1	78.770461	44.058280	30.054899	26.390723	25.798472	24.888756	25.317337
k:2	79.252336	47.685949	34.423739	30.070528	28.657974	28.670496	39.960211
P:4	84.024758	60.096284	46.926755	40.572845	35.580971	40.086169	54.849322
P:6	87.929409	67.693871	54.036675	45.674407	44.067092	43.912240	67.351899
P:8	92.118136	78.659063	62.420199	53.608972	53.989278	54.953496	90.249572

In [ ]:

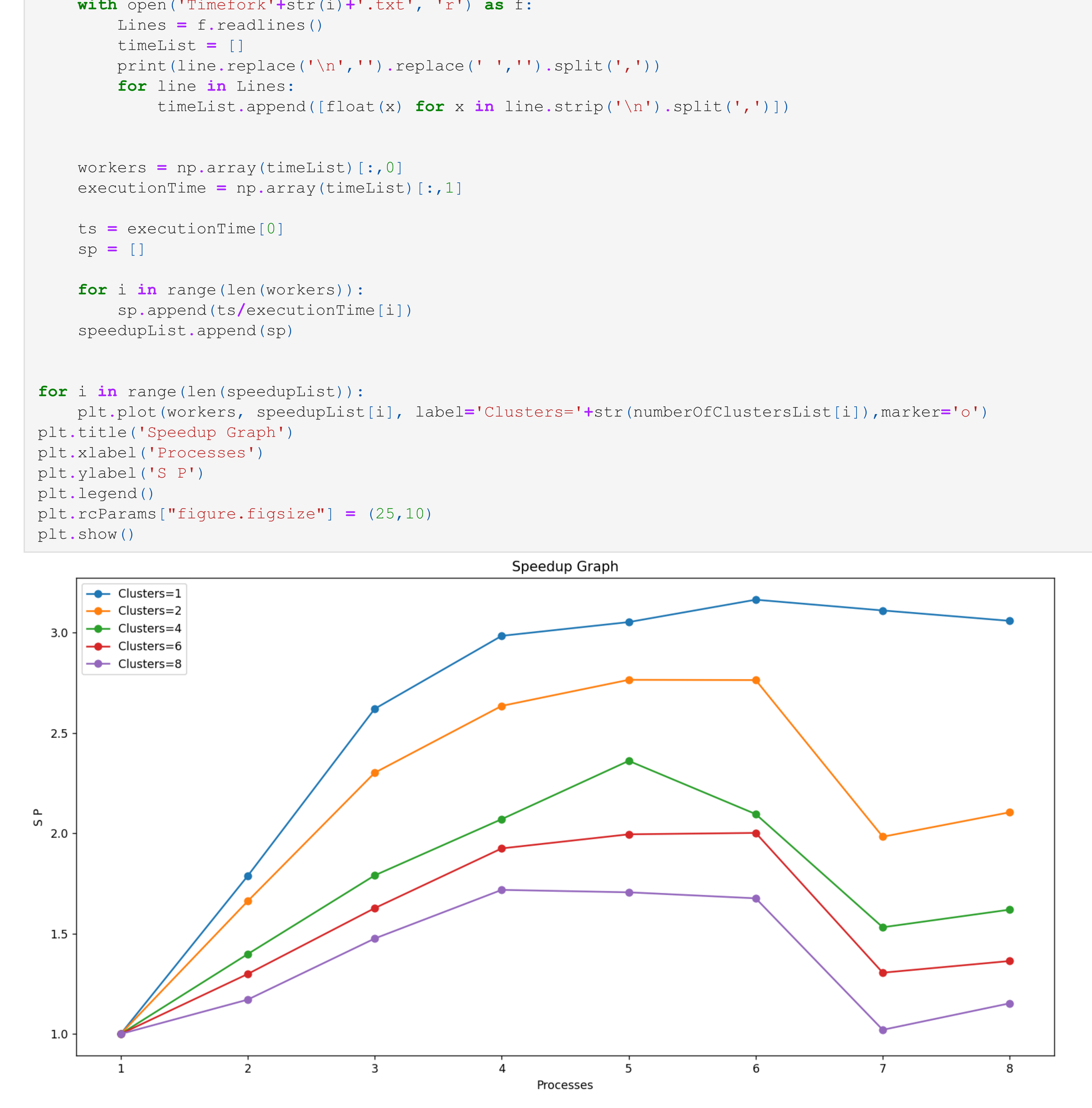
```python
import numpy as np
from matplotlib import pyplot as plt
numberOfClustersList = [2]
speedupList = []
for i in numberOfClustersList:
    with open('Timefork'+str(i)+'.txt', 'r') as f:
        Lines = f.readlines()
        timeList = []
        print(line.replace('\n','').replace(' ','').split(','))
        for line in Lines:
            timeList.append([float(x) for x in line.strip('\n').split(',')])


    workers = np.array(timeList)[:,0]
    executionTime = np.array(timeList)[:,1]

    ts = executionTime[0]
    sp = []

    for i in range(len(workers)):
        sp.append(ts/executionTime[i])
    speedupList.append(sp)


for i in range(len(speedupList)):
    plt.plot(workers, speedupList[i], label='Clusters='+str(numberOfClustersList[i]),marker='o')
plt.title('Speedup Graph')
plt.xlabel('Processes')
plt.ylabel('S P')
plt.legend()
plt.rcParams["figure.figsize"] = (25,10)
plt.show()
```



Common speedup is acheive when running for different worker and cluster k configurations.