Check

May 4, 2018

1 Distributed Data Analytics - Exercise 3

Krithika Murugesan - 277537

Complex Data Lab : K-means clustering in a distributed setting

2 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.

K-means is an unsupervised learning algorithm, it assumes initial cluster centers and groups by data points with nearest cluster. After clusters are formed, new cluster centers are computed by taking the mean of data points in a particular cluster. And this process is repeated till the data points are assigned to a fixed cluster, no matter how many iterations we perform. That is we have found the exact patterns in the data based on the distance metric. Practically there are a large number of constraints including memory and processor capabilities. Therefore a subset is used for the learning here and also the loop is terminated after a 500 iterations, as it is not feasible to wait till the convergence of the algorithm sadly. The parallelization of the code implementation is explained in the upcoming sections

The process with rank 0 is considered the master process, it sends the data to the different processes and synchronizes the results. The results from previous lab are not used as input for this as it in itself is computationally heavy, scikit learn is used to reduce the pre-processing time involved.

```
In [ ]: def kMeans(data,centroid,k):
            '''The function called recursively to compute KNN'''
            comm.Barrier()
            if rank == 0:
                '''Root process'''
                dataDistribute(data,centroid)
                updatedCentroid,updatedCluster = dataCollect(k)
                ##print("BRACE FOR IMPACT")

            else :
                '''Workers'''
                updatedCluster,updatedCentroid = None,None
                partFile = comm.recv(source = 0, tag =1)
```

```
            centroid = comm.recv(source = 0, tag =2)
            #print("At rank ", rank, "shape is ",partFile.shape)
            #print("At rank ", rank, "centroid ", centroid.shape[0])
            dist = distance(partFile,centroid)
            #print("At rank ", rank, "dist ", dist.shape)
            clusters = np.unique(dist)
            #print("Clusters ",np.unique(dist))
            newSum = concatenate(partFile,dist,k)
            comm.send(newSum, dest = 0, tag =3)
            comm.send(dist,dest = 0, tag = 4)
            comm.send(clusters, dest = 0, tag = 5)
            #print("Data send from worker ", rank)

        comm.Barrier()
        #Broadcasting the values to be returned or else they end up null in main function
        updatedCluster = comm.bcast(updatedCluster, root = 0)
        updatedCentroid = comm.bcast(updatedCentroid, root = 0)
        comm.Barrier()
        return(np.concatenate(updatedCluster,axis = 0),updatedCentroid)
        comm.Barrier()
```

The root process gets the data from the main function and distributes the data among the available processes, while it also sends the complete centroid array to every process, enabling each process to compute the distances from points to each centroid easier, this distance calculation is the major bottleneck in the whole algorithm. We try to split this among the workers and use dot product to calculate the euclidean distance, reducing any latencies invloved.

Also, the sparse matrix is used as it is more efficient than the dense counterpart, moreover converting this sparse matrix to dense matrix will require more memory space and even threw a memory error in my case.

dataDistribute Function

This function encapsulates all the send statements from the root process, the matrix is split row-wise based on the number of workers and send to the respective processes.Tags are used to ensure proper sending and reception of messages when this function is recursively called. The centroid is initialized to random data points in the given data by "initialCentroid" function, this centroid is broadcast to all processes

```
In [ ]: def dataDistribute(data,centroid):
            '''All the send functions in root process'''
            rows = data.shape[0]
            for i in range(1,size):
                startIndex = int((i-1)*(rows/(size-1)))
                endIndex = int(((rows/(size-1))*i))
                ##print("Index",startIndex,endIndex)
                #May cause index out of range due to index
                #starting from 0, but we are counting from 1
                if endIndex > rows:
                    endIndex = int(rows)
                comm.send(data[startIndex:endIndex,:],dest = i,tag =1)
                comm.send(centroid, dest = i,tag =2)
```

The sent data is received by the respective workers in the else part of the code, and it is passed to distance function which computes the distances between the data points in the split and the current centroid using dot product(more efficient), later the index with minimum distance is calculated, this index is taken as the cluster. So the cluster numbers start from 0 to k-1. This membership matrix is returned to each process.

```
In [ ]: def distance(a,b):
            '''Calculate the euclidean distance and return the closest cluster center'''
            dist = np.sqrt(a.dot(b.T))
            cluster = np.argmin(dist, axis=1)
            return(np.array(cluster))
```

Cluster lists are returned to the worker, then the worker calls concatenate function to group the data based on clusters to update the centroid. There may be centroids with no data, for such clusters the current coordinates are saved, for the rest cluster wise sum is computed.The sum and the count of data points in each cluster for the worker is sent to the root process

```
In [ ]: def concatenate(partFile,dist,k):
            '''Group the documents based on the clustering done for the first time'''
            newSum = np.array([])
            clusters = np.unique(dist)
            partFile.tocsr()
            for each in range(0,k):
                if each in clusters:
                    points = scipy.sparse.vstack([partFile[j]
                                for j in range(partFile.shape[0]) if dist[j] == each])
                else:
                    points = partFile[each,:]
                newSum = scipy.sparse.vstack((newSum,np.array(points.sum(axis = 0))))
            newSum = (newSum.tocsr())[1:,]
            return(newSum)
```

The dataCollect is a function in the root process, where it colects all the data from the workers, once the sum and cluster lengths from each worker is received, it calls the globalMean function where the centroids are updated

```
In [ ]: def dataCollect(k):
            '''All the receive functions in root process'''
            updatedCentroid,updatedCluster,meanCluster = np.array([]),[],[]
            for i in range(1,size):
                #print("data received from ",i,")
                updatedCentroid = scipy.sparse.vstack
                           ((updatedCentroid,comm.recv(source = i, tag = 3)))
                updatedCluster.extend(comm.recv(source = i, tag = 4))
                meanCluster.extend(comm.recv(source =i, tag =5))
                updatedCentroid = (updatedCentroid.tocsr())[1:,]
                finalSum = globalMean(updatedCentroid,updatedCluster,k)
            #print("At source ",updatedCentroid.shape, finalSum.shape)
            return (finalSum,updatedCluster)
```

The global mean function is similar to concatenate function, where we sum all the worker specific cluster sums and lengths are merged and the global mean is calculated. This updated mean is returned to the root process, from where it is returned to the main function

```
In [ ]: def globalMean(partFile,dist,k):
        '''Each worker returns the local sum and number of elements in each cluster,
              merging and computing global mean'''
            newSum = np.array([])
            actualCluster = np.unique(dist)
            partFile.tocsr()
            #print("Actual ",actualCluster)
            #print("Update centroid ", partFile.shape)
            for each in range(0,k):
                if each in actualCluster:
                    for j in range(partFile.shape[0]):
                        if dist[j] == each:
                            points = scipy.sparse.vstack(partFile[j])
                    dnr = dist.count(each)
                else:
                    dnr =1
                    points = partFile[each]
                newSum = scipy.sparse.vstack((newSum,np.array(points.sum(axis = 0))))/dnr

            newSum = (newSum.tocsr())[1:,]
            return(newSum)
```

Main function, initializes the centroids and calls the kMeans function, after one update of the centroids, it keeps the clusterlist as default comparison for next iteration. In a while loop, the main function checks if the membership matrix(i.e. if the clusters are the same).Untill the clusters are same, the centroids and so the clusters are updated in the kMeans function, which are used as input arguments for the next iteration. This is ensured as the data is in order. The implementation ran for more than 1000 iterations, so the main function breaks out of the loop after 500 such iterations due to limitations in time. The results are compared

```
In [ ]: #Setting the communicators
        comm = MPI.COMM_WORLD
        size = comm.Get_size()
        rank = comm.Get_rank()

        def main():
            data = readData()
            flag = True
            cnt = 1
            k = 20
            #Random array to check compare the clusters after every iteration
            default = np.array([k+1]*data.shape[0])
            comm.Barrier()
            #Getting initial centroids
```

```
        centroid = initialCentroid(data,k)
        #Start of parallel processing
        comm.Barrier()
        tic = time.time()
        #Calling Kmeans function for first time to get intial clusters
        updatedCluster,updatedCentroid = kMeans(data,centroid,k)
        print("Centroid at first ",np.unique(updatedCluster))
        comm.Barrier()
        #Looping until membership remains the same
        while flag == True:
            #Checking for membership
            if cnt == 500:
                print("Algorithm did not converge in 1000 iterations")
                print("Time taken : ", time.time()-tic)
                break
            if (updatedCluster==default).all():
                print("Algorithm Converged")
                flag = False
            else:
                comm.Barrier()
                #When condition fails recursively calling the kMeans function
                updatedClusterNew,updatedCentroidNew =
                                    kMeans(data,updatedCentroid,k)
                comm.Barrier()
                print("Time taken : ", time.time()-tic)
                cnt = cnt+1
            #Updating cluster memberships and centroids
            default = updatedClusterNew
            updatedCentroid = updatedCentroidNew


    if __name__ == "__main__":

        '''Main function'''

        main()
```

Even considering only few categories, the algorithm did not converge in feasible time, the times given below are the time for 500 iterations of the whole process with different number of processors working in parallel

```
In [1]: import pandas as pd
        import numpy as np
        #Number of process
        p = [2,4,6,8]
        #Execution time with p processes
        tp = [1002.3742477893829,931.8633739948273,875.6735428910354,944.4562938461038]
        parallel = pd.DataFrame({'process': p,'tp': tp})
        display(parallel)
```

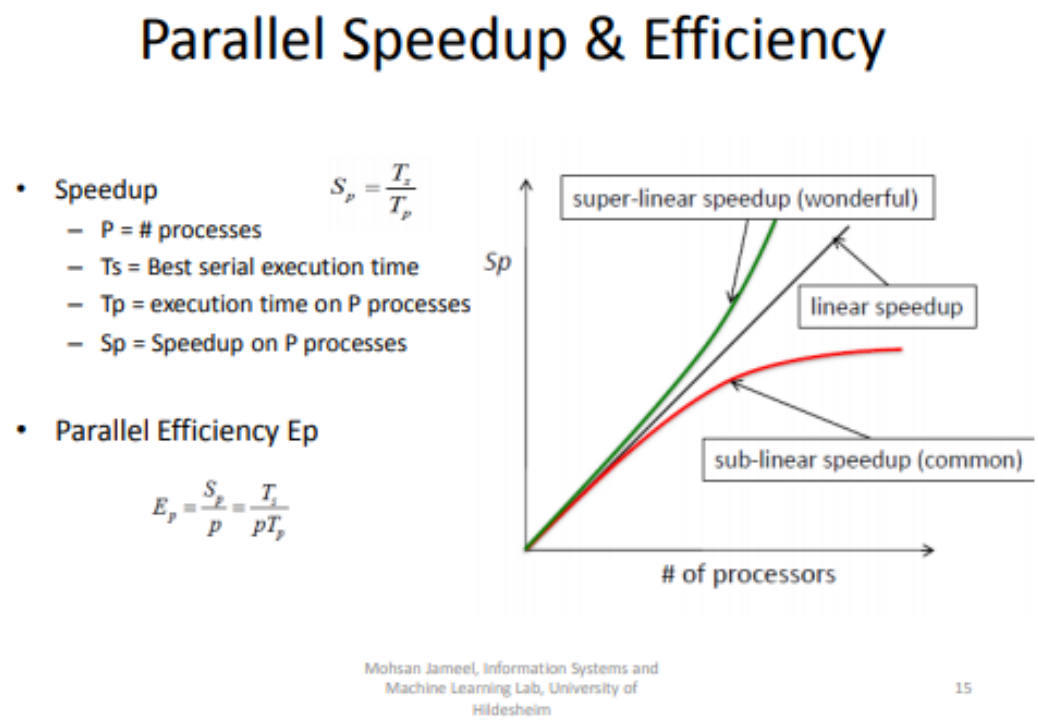
```
   process           tp
0        2  1002.374248
1        4   931.863374
2        6   875.673543
3        8   944.456294
```

3 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 [2]:
```
#Efficiency.png
from IPython.display import Image
Image(filename='Efficiency.png')
```

Out[2]:



For computing the parallel speedup and efficiency graph, we have already obtained the execution time in parallel process as shown in previous block. To find the speed up for each process, we have to get the best serial execution time. Serial execution is when each process waits for the

next process to finish its task. This can be implemented by having a root process that sends data to one worker, waits for the processed data and sends the data to the next worker. This is same as sending data to one worker over and over again. This is done by modifying only the code at root process with all the same functions, using one for loop for both send and receive instead of two seperate loops. This has been repeated five times and average has been taken. The modified code block is given below

```
In [ ]: def kMeans(data,centroid,k):
            '''The function called recursively to compute KNN'''
            comm.Barrier()
            if rank == 0:
                '''Root process'''
                for i in range(1,size):
                    '''All the send functions in root process'''
                    rows = data.shape[0]
                    startIndex = int((i-1)*(rows/(size-1)))
                    endIndex = int(((rows/(size-1))*i))
                    ##print("Index",startIndex,endIndex)
                   if endIndex > rows:
                        endIndex = int(rows)
                    comm.send(data[startIndex:endIndex,:],dest = i,tag =1)
                    comm.send(centroid, dest = i,tag =2)
                    ##print("BRACE FOR IMPACT")
                    '''All the receive functions in root process'''
                    updatedCentroid,updatedCluster,meanCluster = np.array([]),[],[]
                    #print("data received from ",i)
                    updatedCentroid = scipy.sparse.vstack
                                      ((updatedCentroid,comm.recv(source = i, tag = 3)))

                    updatedCluster.extend(comm.recv(source = i, tag = 4))
                    meanCluster.extend(comm.recv(source =i, tag =5))
                    updatedCentroid = (updatedCentroid.tocsr())[1:,]
                    finalSum = globalMean(updatedCentroid,updatedCluster,k)

            else :
                '''Workers'''
                updatedCluster,updatedCentroid = None,None
                partFile = comm.recv(source = 0, tag =1)
                centroid = comm.recv(source = 0, tag =2)
                #print("At rank ", rank, "shape is ",partFile.shape)
                #print("At rank ", rank, "centroid ", centroid.shape[0])
                dist = distance(partFile,centroid)
                #print("At rank ", rank, "dist ", dist.shape)
                clusters = np.unique(dist)
                #print("Clusters ",np.unique(dist))
                newSum = concatenate(partFile,dist,k)
                comm.send(newSum, dest = 0, tag =3)
                comm.send(dist,dest = 0, tag = 4)
```

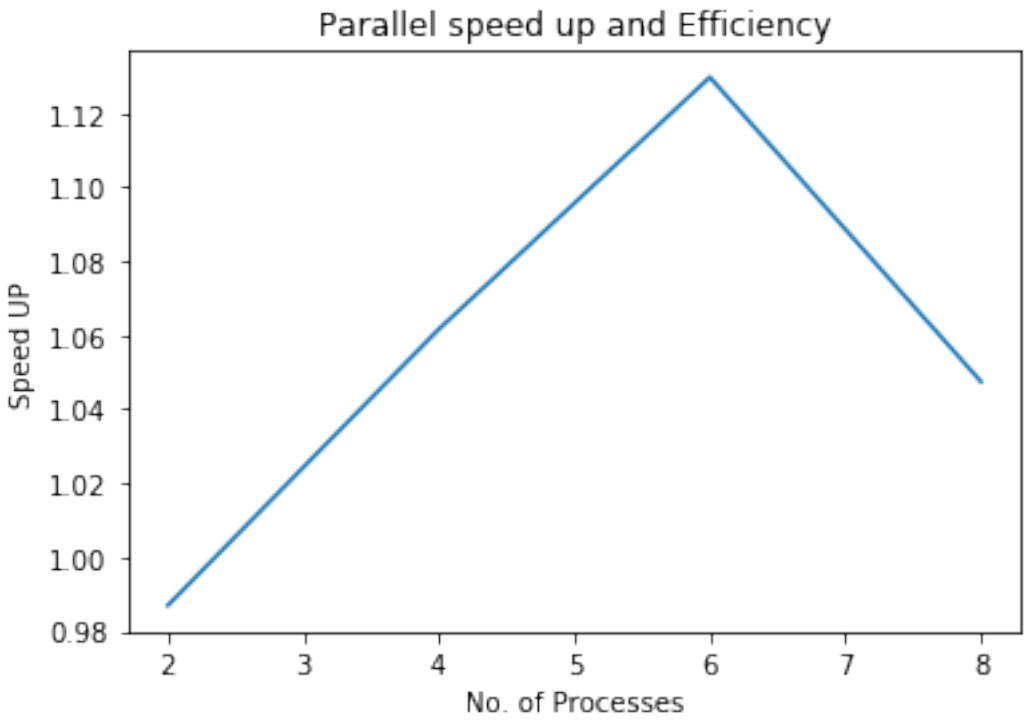
```
                comm.send(clusters, dest = 0, tag = 5)
                print("Data send from worker ", rank)

            comm.Barrier()
            #Broadcasting the values to be returned or else they end up null in main function
            updatedCluster = comm.bcast(updatedCluster, root = 0)
            updatedCentroid = comm.bcast(updatedCentroid, root = 0)
            comm.Barrier()
            return(np.concatenate(updatedCluster,axis = 0),updatedCentroid)
            comm.Barrier()
```

In [3]:
```
#Serial programming time
ts = [1024.1546782536498, 998.1648253647183, 989.2827452639603,1121.1567254378299,1005
serial = pd.DataFrame({'serialTime': ts})
display(serial)
#Minimum value is the best serial time
Ts = min(ts)
print("Best serial time ", Ts)
```

```
    serialTime
0  1024.154678
1   998.164825
2   989.282745
3  1121.156725
4  1005.547298
```

```
Best serial time  989.2827452639604
```

In [4]:
```
#Speed up on p processes
sp = [Ts/x for x in tp]
efficiency = [i/j for i,j in zip(sp,p)]
#Plotting the graph
import matplotlib.pyplot as plt
plt.plot(p,sp)
plt.xlabel("No. of Processes")
plt.ylabel("Speed UP")
plt.title("Parallel speed up and Efficiency")
plt.show()
eff = pd.DataFrame({'process' : p,'speedup': sp,'efficiency':efficiency})
display(eff)
```



	efficiency	process	speedup
0	0.493470	2	0.986940
1	0.265404	4	1.061618
2	0.188290	6	1.129739
3	0.130933	8	1.047463

It can be seen that the results are showing sub-linear speed up as depicted in the given literature. Also, the physical avaiability of processors is only 4, more points would make the graph more curved.

Note : k is a hyper parameter, which can be determined by various methods, elbow method being one where the graph between different values of k and the percentage of varaince, a point where there is a sharp dip is there is considered as optimal value of k. Due to the background knowledge, we take k = 2 the same as number of news groups in the data