MPI KMeans Clustering

Series/Parallel

REPORT – LAB 4
Distributed Systems – 15640
Summer 2014

Team Members
Pranav Saxena

Vaibhav Suresh Kumar

Design

We decided to implement this project partly in Python and partly in Java. The sequential K means 2D data point implementation has been done in Python, Parallel K means Data Point implementation and the data set generator has been written in python as well. Java has been used to implement the Kmeans DNA Sequential and Kmeans DNA Parallel version .

Design Decisions -

1) **Logical Abstraction of a Data Point** - We abstracted a 2D data point as a separate class which encapsulates the

2) **Logical Abstraction of a Data Point Cluster** - We created a separate class for a cluster which extends the data point class and is responsible for updating and fetching the centroid coordinates.

3) **K Means Sequential 2D Data Point Algorithm** - Here we first randomly choose the distinct centroid coordinates from a random sample. Next for each randomly generated data point, we find the minimum distance to the centroid and put it in that centroid's cluster. Once the first iteration is done, we find the mean of the all the data points within one centroid and update the centroid's coordinates. This is done for all the centroids and this updation sometimes results in shifting of data points from one cluster to another.We have a threshold value which the user can set which decides the extent to which centroid position updation has to be entertained. If the updation to the centroid coordinate remains less than the threshold, the program converges. This would lead to a situation when the clusters have become stable and thus no more changes occur.

4) **K Means Parallel 2D Data Point Algorithm** - This has been implemented in Python and uses the mpi4py library to support all the interprocess communication using MPI. We have followed a master-slave relationship for implementing this algorithm. The master is assigned the RANK 0 and maintains the list of all centroid and sends it all the respective machines which are responsible for running K means algorithm on their respective chunks of the entire data. First the master divides the randomly generated data into chunks and distributes it to the different machines. Once that has been received by each slave machine, the list of centroids randomly sampled on the master machine is sent to each of the slave machines. As soon as each slave machine receives the list of centroids, it starts calculating the distance to each of the centroids and assigns it that

centroid to which it has the minimum distance i.e running sequential K means algorithm on its local chunk. By the end of one iteration, each data point has a cluster associated to it. Next , we calculate the mean values for each centroid on each of the slave machines. These new mean values are relayed back to the master machine which updates its local list of centroids with these new values by further taking a mean of the means (**similar to the median of Medians approaches to find the best possible pivot in Algorithms**) calculated by individual slave machine. Next the master sends it back again to each of the slave machines to calculate the distances. This process keeps on happening until the program converges according to a initially set threshold value. FInally all the slave machines will send the data to the master and the master will finally write all the data in one file.

Note - We have considered Euclidean distance to calculate distance between the data points.

PSEUDO CODE
****

```
def main():
   comm = MPI.COMM_WORLD
   size = comm.Get_size()
   machinerank = comm.Get_rank()
   procname =MPI.Get_processor_name()
   writerInput = csv.writer(open("input", "w"))

   chunkSize =   int(Num_clusters / (size-1))
    centroidList = []
   centroidList = SplitList(randomPoints, chunkSize)
   temp = copy.deepcopy(centroidList)
   x=[]
   globalPointMap = {}
   clusterContainer=[]
   temp.reverse()
   ''' get initial centroids '''
   if machinerank == 0:      # RANK = 0 - MASTER MACHINE
       randomsample = random.sample(randomPoints,num_clusters)
       for i in range(1,size):
             # send centroids to each slave machine
             comm.send(randomsample,dest=i,tag=11)
```

```
            print "sending centroids to all slaves"
    else:
       randomsample = comm.recv(source=0,tag = 11)
       print "centroids received at slave" + str(randomsample)  # random sample
sent to each slave machine

    if machinerank == 0:
           send chunk to each machine
          centroidList = comm.recv(source = i, tag = 11)
          recalculate Mean ( centroidList)
 for i in range( 1, size):           # send the updated centroid list to each slave machine
         comm.send(centroidList, dest = i , tag = 12)
   else:
       # Receive chunk from master
       x=comm.recv(source =0,tag =11)
      # Receive centroid list from master
     converge = False
      while not Converge : {
            Kmeans (x)     # Run K means on local chunk

           updateMean(x)   # find the mean of the data points belonging to each cluster in
one slave machine

      comm. Send( centroidList, 0, tag = 11)   # each process sends the updated list back
to the master
      updatedCentroidList = comm.recv(source =0, tag =12)
      if( maxUpdate < threshold)
           break
      }

if machineRank == 0:
           getAlldata( for each slave machine)
           writeData ( file)
```

****

5) DNA Sequential K means Clustering Algorithm -

This has been implemented in Java but it receives its data set from a random generator implemented in Python. The data set has been generated using gaussian distribution by first randomly choosing a centroid string and then generating the other strands by simply generating all permutations by removing one character, then randomly putting another one. This sort of data helps in reaching the convergence at a faster rate in the sequential version of the program. The algorithm is slightly different from the sequential 2D Data point algorithm . Here we define the similarity between two strings as number of unmatched characters which has been implemented in Python using a simple zip function and character comparison. Further, the difference in the algorithm comes when we have to update the centroids when running K means.

Algorithm to update centroid -

We compare each string with every other string and calculate its similarity distance. The string which has the maximum similarity in the dataset is chosen as the next centroid. This is fairly reasonable approach here as we are able to reach the convergence at a much faster rate.

6) DNA Parallel K means Clustering Algorithm -

The algorithm to implement DNA Parallel is sort of based on a similar strategy as 2D DataPoint Parallel approach . We end up calculating mean of means (median of medians) to get the updated centroid position. After the first iteration, we just use the strategy in the sequential version to find the maximum similarity string in every slave machine and assign that as the centroid and send it to the master. Once the master receives this updated centroid from each slave machine, it runs the maxStringSimilarity logic again on this list and find the final updated centroid and send it back to all the slave machines to run the next iteration until convergence happens or a maximum number of iterations set internally are reached.

This approach works well here as there is almost no ties in selection of the centroid and leads to an equal distribution of cluster sizes most of the times.

Pseudo Code -

```
public static void main() {

MPI.Init(args);
DNAParallel dna = new DNAParallel();
LinkedList<String> centroidList = new LinkedList<String>();
LinkedList<String> entireDataSet = new LinkedList<String>();

LinkedList<LinkedList<String>> computedResult = new LinkedList<LinkedList<String>>();
if(dna.getRank() == 0){

File file = new File ("randomData");
BufferedReader in = new BufferedReader(new FileReader(file));
String line = "";
String arg[];


divideintoChunks(data )

if machineRank == 0:
sendCentroidsToSlaveMachines(

  MPI.send(centroidList, dest = i , tag = 11);
)

 KMeansatSlaveMachines();  // run K means at local machine

MPI.send(updatedCentroidList);   // send updated centroid list to master


if machinerank == 0:   // Master receives centroid from each machine
         MPI.recv( updatedCentroidList)
         caclulateFinalMean(updatedCentroidList)  // Reevaluate mean of means at Master program
        MPI.send( updatedCentroidList, dest = i , tag = 11)   // send the updated centroidList
}

MPI.Finalize()
```

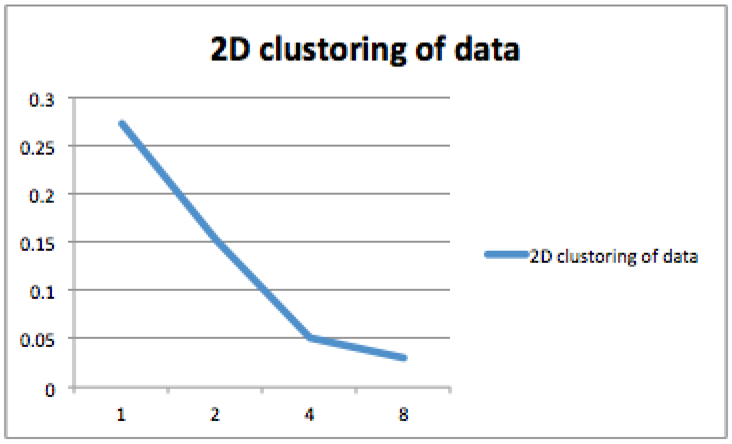
****

EXPERIMENTATION AND ANALYSIS

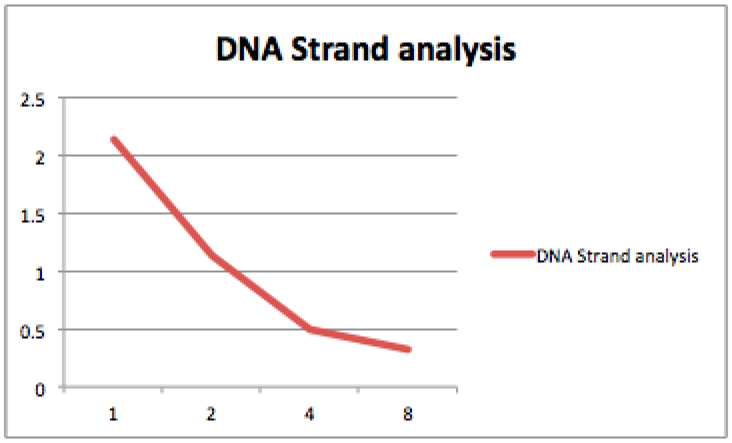
We tested our code with test inputs of various sizes. We changed the number of processors ,the number of clusters, and the number of points per cluster. For the number of processors, we used 2 4 or 8 processors. For clusters, we used 2, 3, 4, 8, 12, or 25 clusters. we used 10, 50, 100, 1000, 5000, and 10000 points per cluster.

Version	Number of Clusters/ Number of processors	DataPoints per Cluster	Program Run time(in sec)
2D & series	4/1	100	0.272 seconds
DNA & Series	4/1	20	2.131 seconds
2D & Parallel	4/2	20	0.1515 seconds
DNA & Parallel	4/2	100	1.13 seconds
2D & Series	4/4	1000	0.05 seconds
2D & Parallel	4/4	1000	0.5 seconds
DNA & Series	4/8	1000	0.03 seconds
DNA & Parallel	4/8	1000	0.313 seconds

2D series an parallel:



Dna Strand Analysis:



FINDINGS -

We observed that since the data generated for DNA was by just permuting the string which acted as the centroid, it took far fewer iterations to converge compared to the 2D Data point. But in order to observe some benefits of parallel processing, we tested our code with very long strand lengths (order of magnitude > 100 , 1000) and lesser number of clusters which took approximately 2.5 seconds to converge.

We also observed that increasing the processors helps to a limit but after that the communication overhead between more number of processors increase a lot which inhibits the performance and increases the response time.

INSTRUCTIONS TO RUN THE CODE

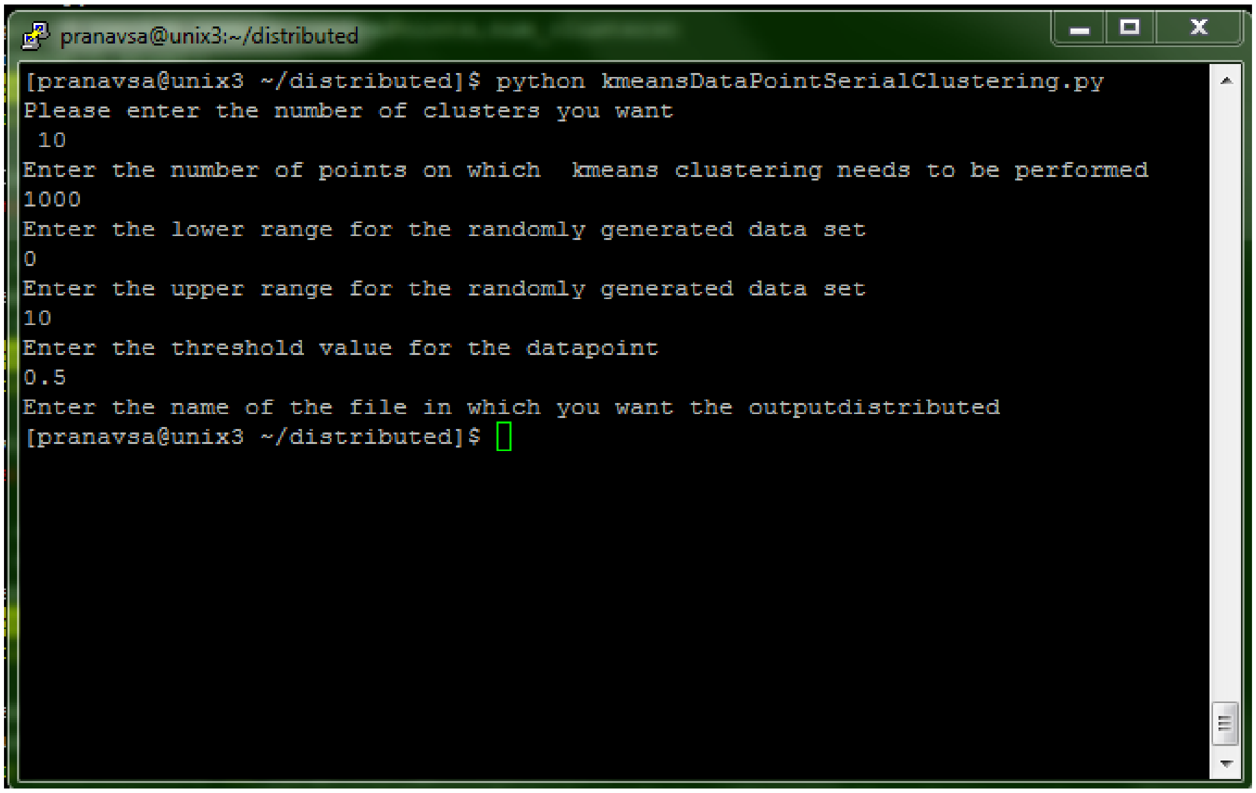
All the screenshots below are of AFS machines and the code works fine on them. We used GHC33, GHC35, GHC37 to run the code using MPI and didn't have any issues with sshing without a password.

1) To run the series version of 2D Data Point -

Type the following command -

```
$ mpirun  kmeansDataPointSerialClustering.py
```

You will see the following screen and enter the details as shown in the screenshot.



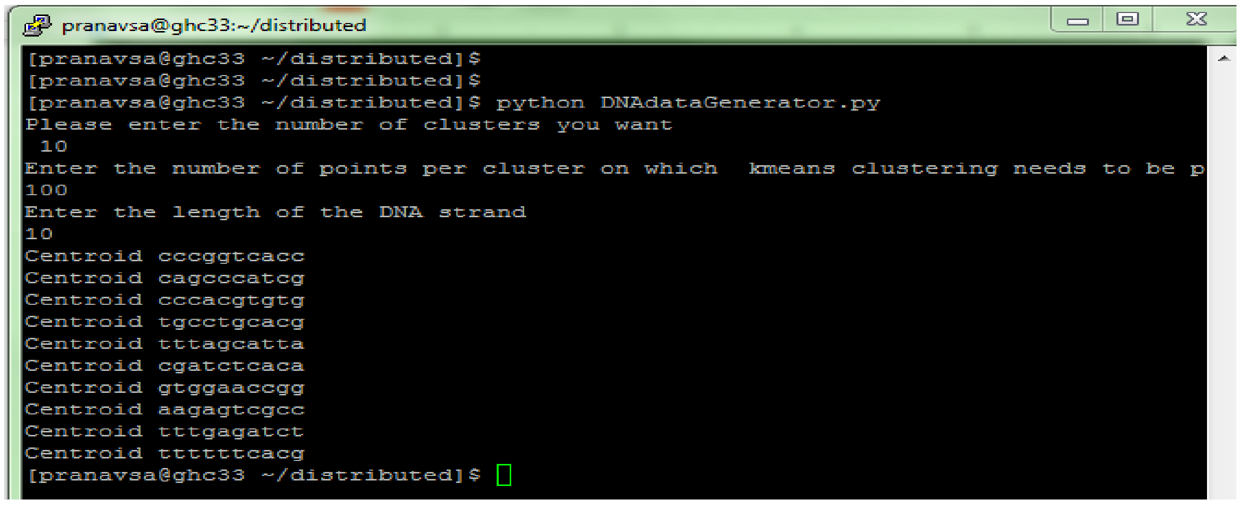
```
[pranavsa@unix3 ~/distributed]$ python kmeansDataPointSerialClustering.py
Please enter the number of clusters you want
 10
Enter the number of points on which  kmeans clustering needs to be performed
1000
Enter the lower range for the randomly generated data set
0
Enter the upper range for the randomly generated data set
10
Enter the threshold value for the datapoint
0.5
Enter the name of the file in which you want the outputdistributed
[pranavsa@unix3 ~/distributed]$
```

1) **To Run the series version of the DNA Series version -**

We would need to use the Python program to generate the data and Java program to run the K means algorithm on the data file generated.

a) Generate the data using

$ python DNAdataGenerator.py



```
[pranavsa@ghc33 ~/distributed]$
[pranavsa@ghc33 ~/distributed]$
[pranavsa@ghc33 ~/distributed]$ python DNAdataGenerator.py
Please enter the number of clusters you want
 10
Enter the number of points per cluster on which  kmeans clustering needs to be p
100
Enter the length of the DNA strand
10
Centroid cccggtcacc
Centroid cagcccatcg
Centroid cccacgtgtg
Centroid tgcctgcacg
Centroid tttagcatta
Centroid cgatctcaca
Centroid gtggaaccgg
Centroid aagagtcgcc
Centroid tttgagatct
Centroid ttttttcacg
[pranavsa@ghc33 ~/distributed]$
```

NOTE - The output data file should be generated in the same directory as the python and the java programs.

b) Next, compile and run the JAVA program on the output data file.

```
$javac DNASerial.java
```

```
$ java DNASerial
```

The output will be generated in **DNASerialOutput.txt** file.

2) **TO RUN THE PARALLEL VERSION OF THE CODE**

a) 2D DATA POINT - This has been implemented in Python

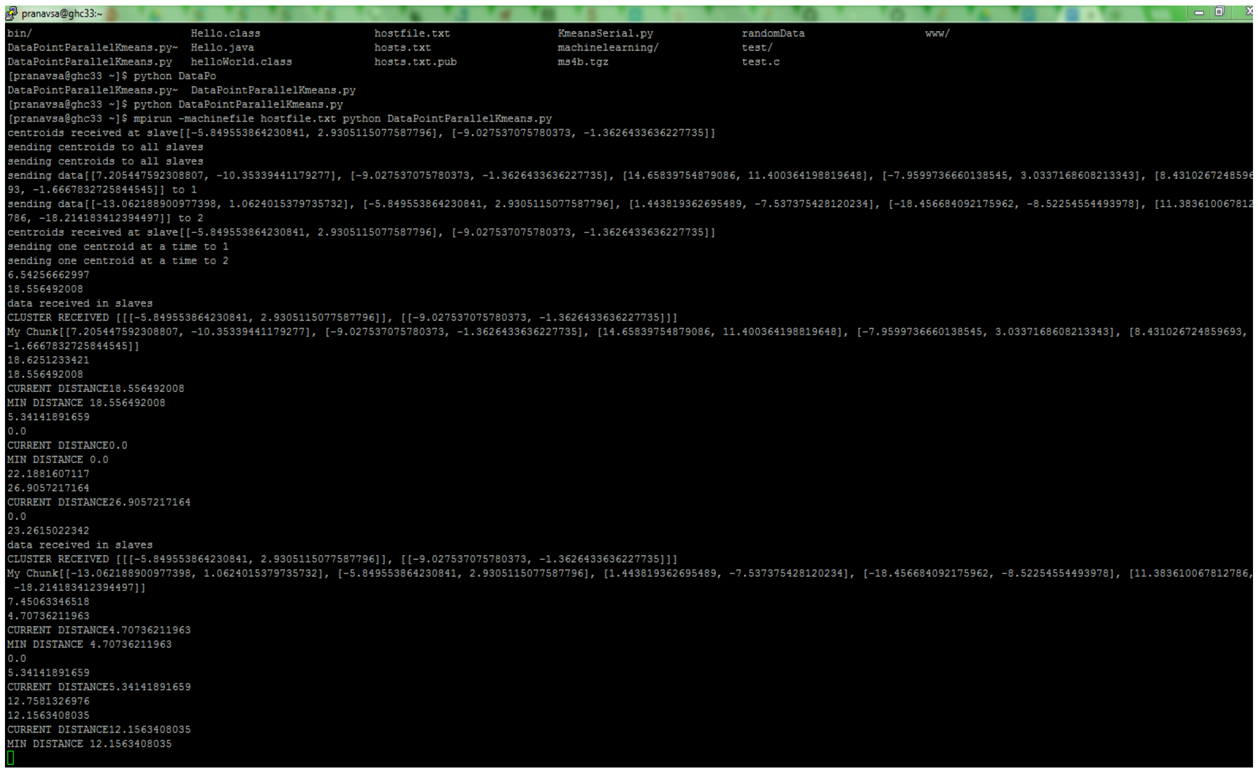
This has been implemented in python will the algorithm described in the report earlier.

To run the program, type this -

```
$  mpirun -np <No of processors >   -machinefile hostfile.txt python
DataPointParallelKmeans.py
```

We have provided the hostfile.txt which contains 3 different GHC machine addresses (ghc33, ghc35, ghc37) . We have been able to run our code on different machines. Also we have hardcoded the configuration (in order to overcome EOF error on ASF machine as it was not accepting input.)

totalPoints = 100   num_clusters = 2   threshold = 20



```
bin/                          Hello.class                   hostfile.txt                  KmeansSerial.py               randomData                    www/
DataPointParallelKmeans.py~   Hello.java                    hosts.txt                     machinelearning/              test/
DataPointParallelKmeans.py    helloWorld.class              hosts.txt.pub                 ms4b.tgz                      test.c
[pranavsa@ghc33 ~]$ python DataPo
DataPointParallelKmeans.py~  DataPointParallelKmeans.py
[pranavsa@ghc33 ~]$ python DataPointParallelKmeans.py
[pranavsa@ghc33 ~]$ mpirun -machinefile hostfile.txt python DataPointParallelKmeans.py
centroids received at slave[[-5.849553864230841, 2.9305115077587796], [-9.027537075780373, -1.3626433636227735]]
sending centroids to all slaves
sending centroids to all slaves
sending data[[7.205447592308807, -10.35339441179277], [-9.027537075780373, -1.3626433636227735], [14.65839754879086, 11.400364198819648], [-7.9599736660138545, 3.0337168608213343], [8.4310267248596
93, -1.6667832725844545]] to 1
sending data[[-13.062188900977398, 1.0624015379735732], [-5.849553864230841, 2.9305115077587796], [1.443819362695489, -7.537375428120234], [-18.456684092175962, -8.52254554493978], [11.383610067812
786, -18.214183412394497]] to 2
centroids received at slave[[-5.849553864230841, 2.9305115077587796], [-9.027537075780373, -1.3626433636227735]]
sending one centroid at a time to 1
sending one centroid at a time to 2
6.54256662997
18.556492008
data received in slaves
CLUSTER RECEIVED [[[-5.849553864230841, 2.9305115077587796]], [[-9.027537075780373, -1.3626433636227735]]]
My Chunk[[7.205447592308807, -10.35339441179277], [-9.027537075780373, -1.3626433636227735], [14.65839754879086, 11.400364198819648], [-7.9599736660138545, 3.0337168608213343], [8.431026724859693,
-1.6667832725844545]]
18.6251233421
18.556492008
CURRENT DISTANCE18.556492008
MIN DISTANCE 18.556492008
5.34141891659
0.0
CURRENT DISTANCE0.0
MIN DISTANCE 0.0
22.1881607117
26.9057217164
CURRENT DISTANCE26.9057217164
0.0
23.2615022342
data received in slaves
CLUSTER RECEIVED [[[-5.849553864230841, 2.9305115077587796]], [[-9.027537075780373, -1.3626433636227735]]]
My Chunk[[-13.062188900977398, 1.0624015379735732], [-5.849553864230841, 2.9305115077587796], [1.443819362695489, -7.537375428120234], [-18.456684092175962, -8.52254554493978], [11.383610067812786,
 -18.214183412394497]]
7.45063346518
4.70736211963
CURRENT DISTANCE4.70736211963
MIN DISTANCE 4.70736211963
0.0
5.34141891659
CURRENT DISTANCE5.34141891659
12.7581326976
12.1563408035
CURRENT DISTANCE12.1563408035
MIN DISTANCE 12.1563408035
```

The convergence happens and we are able to get the final set.

b) DNA PARALLEL Kmeans - This has been implemented in JAVA

First generate the data using the Python Random generator.
$ python DNAdataGenerator.py

To compile run the DNA Parallel code, please type this command -

$ mpijavac DNAParallel.java
$mpirun -np <No of processors >   -machinefile hostfile.txt java DNAParallel