15-440/15-640 Project 4 Report
Clustering using MPI

Team: Lixun Mao (lmao), Druhin Sagar Goel (dsgoel)

1 | Algorithms for K-Means Parallelization

1.1 | 2-D Data Points

The master begins by selecting K random centroids from the data set. It then sends each of the slaves these centroids. Upon receving the centroids, each slave determines the partition of the data set it has to work on using the getRange function (which divides the data set into partitions based on the total number of slaves and the rank of the slave in question). It then iterates over the points in its partition and using the Euclidean distance measure, finds the closest cluster centroid for each point and adds the point to that cluster, maintaining this information in two-dimensional list. Then for each cluster in the two-dimensional list, the slave calculates the new centroid as the average or mean of all the points in that cluster. The slave then sends to the master the newly calcluated centroids as well as the number of points assigned to each cluster. The master then gathers all this information from the slaves, and uses it to compute the final centroids for the current iteration. After this computation, the master checks if the new set of centroids is within a threshold distance from the old set of centroids, and if so, returns the new set of centroids. Otherwise, it moves onto another iteration and sends the slaves the new set of centroids.

1.2 | DNA Data Points

The master begins by selecting K random centroids from the data set. It then sends each of the slaves these centroids. Upon receving the centroids, each slave determines the partition of the data set it has to work on using the getRange function (which divides the data set into partitions based on the total number of slaves and the rank of the slave in question). It then iterates over the DNAs in its partition and label the closest cluster centroid index for each DNA based on the “Similarity” (here “Similarity” can be defined as a function F(. , .) of the number of bases in a strand subtracted from the number of changes required to turn one strand into the other). Slave will also record how much different label desicion made comparing to the last time. Here we use the most common base for each position in the strand among all the strands of the corresponding cluster to generate new centroids. So the slave will calculate the frequency of base for each position in the strand given its partial dataset. The slave then sends to the master the counts as well as the number of different labels assigned to each cluster comparing to the last iteration. The master then gathers all this information from the slaves, and compute the final centroids for the current iteration. Also the master will gather the total difference. And this total difference is used to decide whether the algorithm converge or not. If the total difference smaller than then threshold then, returns

the final set of centroids. Otherwise, it moves onto another iteration and sends the slaves the new set of centroids.

2 | How to run

2.1 | Compiling and Building

We use Ant to compile our project and so before compiling, you must make sure that you have Ant installed on your machine(s). We have provided the build file. It is called **"build.xml"**. To compile, navigate to the directory, through command line, where this file is stored. Type the following command into command line:

ant -f build.xml

After compilation, a folder called bin will be created which will contain all the class files related to our project. Before running , make sure you copy the directory under bin called main to the machine on which you want to start running the program.

2.2 | Running

After you login in to master ghc machine and make sure you navigate to the directory containing the **main** directory.

For 2-D Data Points

To generate the data set for 2-D points, type the following into the command line:

python generaterawdata.py -c <cluster> -p <num_points> -o <output>

where:

cluster is the number of clusters you want to create,

num_points is the number of points per cluster

output is the name of the file where you want the data set to reside

To run the sequential version of K-Means for 2-D data points, type the following into command line:

java main/SequentialDataPoint <K> <dataset> <output>

where:

K is the number of clusters you want to create,

dataset is the path to file containing the dataset and
output is the path to the file where you want to write the results of the program to.

To run the parallel version of K-Means for 2-D data points, type the following into command line:
mpirun -np <processor_number> -machinefile <machines> java main/ParallelDataPoint <K> <dataset> <output>
where:
processor_number is the number of processes you want to divide the task into
machines is the path to a file containing the names of the machines you want to run the program on
K is the number of clusters you want to create,
dataset is the path to file containing the dataset and
output is the path to the file where you want to write the results of the program to.

For DNA Data Points
To generate the data set for 2-D points, type the following into the command line:
java main/DNAGenerator <DNA length> <DNA numbers> <output>
where:
DNA length is the length of one strand of DNA,
DNA numbers is the size of the data set to be generated,
output is the path to the file where you want to write the results of the program to.

To run the sequential version of K-Means for DNA data points, type the following into command line:
java main/seqDNA <cluster numbers> <DNA length> <input dataset> <output file>
where:
cluster numbers is the number of clusters to be created,
DNA length is the length of one strand of DNA,
input dataset is the path to the file containing the data set and
output file is the path to the file where you want to write the results of the program to.

To run the parallel version of K-Means for DNA data points, type the following into command line:

mpirun -np <processor_number> -machinefile <machines> java main/parDNA <cluster numbers> <DNA length> <input dataset> <output file>

where:

processor_number is the number of processes you want to divide the task into,

machines is the path to a file containing the names of the machines you want to run the program on,

cluster numbers is the number of clusters to be created,

DNA length is the length of one strand of DNA,

input dataset is the path to the file containing the data set and

output file is the path to the file where you want to write the results of the program to.

NOTE ON machines file: Firstly, make sure the GHC machines in the machines.txt file are in your ∼/.ssh/known_hosts file. Also making sure that you can login in any machine without typing the password. If it require the password please try other machines because we find a lot of GHC machines has this problems and we can not solve that. If adding more than 2 hosts in the machines file doesn't work (gives you "Host key verification failed" error even though the hosts are in your known hosts file), then go to the file ∼/.ssh/config (create it if it doesn't exist) and add the following to it:

```
Host ghc*
   StrictHostKeyChecking no
   UserKnownHostsFile=/dev/null
```

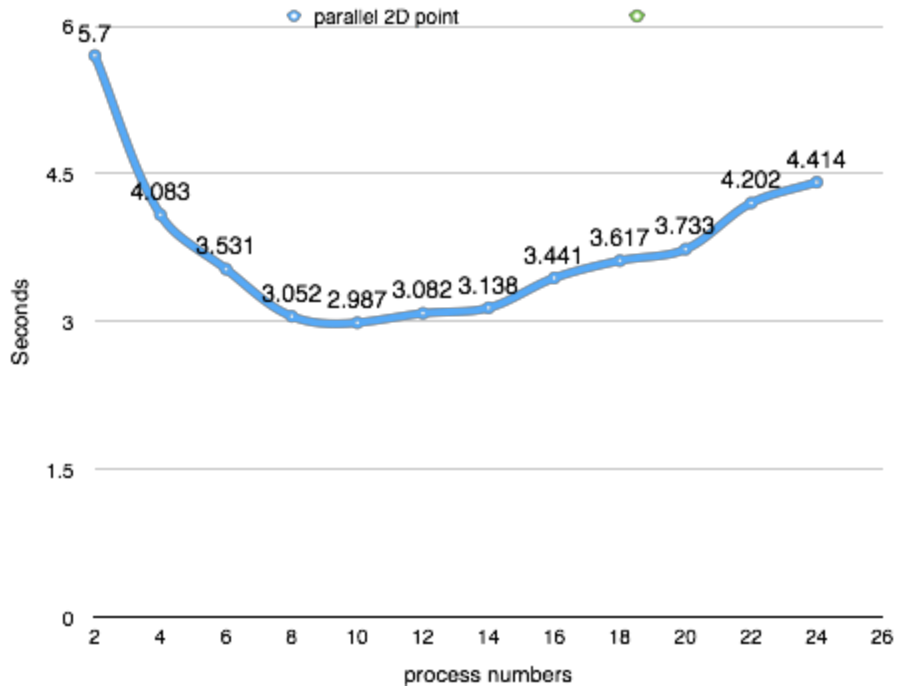
As a helper, we've provided a file called **machines.txt** with the deliverables which gives an idea of how the machines file should look like. Also, make sure you run the program on a machine that is different from the ones in your machines file.

3 | Experimentation and Analysis

Our implementations only run the algorithm on slave nodes. The master node just does the job of assigning partitions of points to clusters, and combining the results that the slaves send back. The times taken by the sequential version for the two data sets is given in table below. Both data sets had 100,000 data points and DNA strands(DNA length 50).

Dataset	K	Time (in seconds)
2D	5	5.73
DNA	5	41.86

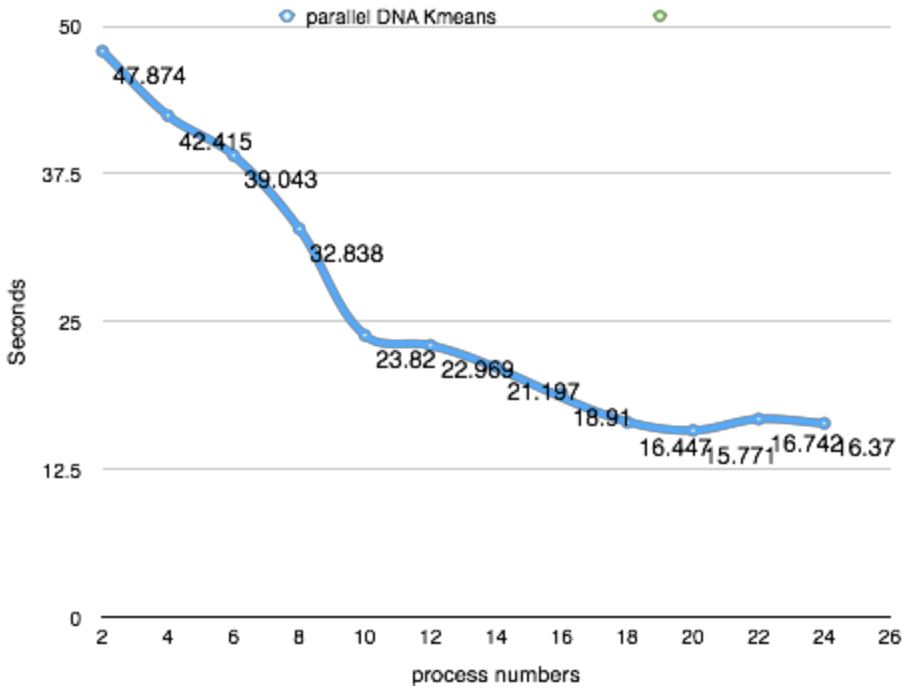
The figure below shows the graph of time taken by the whole system across different number of processes for MPI-bases K-Means for 2D data set. This was run with the same data set, and same set of initial centroids. Number of points in data set was 100,000 and k = 5 (20,000 points per cluster).



We can clearly see that intially as the number of processors increases, the time taken decreases. Initially, increasing the number of processors affects time taken a lot. But then the

change in time with number of processors decreases, and eventually at 10-12 processors we see the time going back up. The least time taken for this data set seems to be at 10 processors. We see that the time taken for the MPI-based solution with 2 processors is actually as fast (or as slow) as the sequential solution. This shows that it is expensive to do the setup, and that time spent in communication of large amounts of data is significant.

The figure below shows the graph of time taken by the whole system across different number of processes for MPI-bases K-Means for DNA data set. This was run with the same data set, and same set of initial centroids. Number of points in data set was 100,000 and k = 5 (each DNA has 50 length).



We can see a similar trend in the time taken over number of processors used. The time taken decreases globally, until we reach 20-22 processors. Beyond that the time taken starts to increase. So the least time taken for this data set seems to be at 20 processors. And after that the total time actually fixed because the I/O time and communication time dominates. Unlike the 2D data points, the time taken does not increase much beyond a certain point. This is

because with the DNA, the amount of communication required is much more compared to that of the 2D data points and so the communication time is the deciding factor in the run time of the program.

Both graphs see a minimum point. But increasing the number of processors beyond that increases the time taken. Beyond the minimum point we have diminishing returns. The proportion of points given to each processor, and the time spent in startup and communication overhead are in perfect balance when we have the number of processors at the minimum point. But if we increase the number of processors, the time spent in dealing with relatively lesser number of points per processor, is actually less compared to the time spent in communicating and starting up. In other words, the communication overhead dominates beyond the sweet spot, and increasing the number of processors only increases the time taken.