Swapnil Pimpale (spimpale)
Romit Kudtarkar (rkudtark)

15-440/640 Project 4 Report

Collaboration Statement: All work submitted by us is our own.

Introduction

This report contains information about the design and implementation of K-means clustering program that we have written for Project 4. It also has the performance analysis of the K-means algorithm when run on 2D data points and DNA strands. Our implementation supports both sequential and parallel mode of execution.

Overview of Design

Our clustering program implements a simple K-means clustering algorithm, which can be run on either 2D data points or on DNA strands.

Distance calculation:

Distances between 2D data points are measured via Euclidean distance, and distances between DNA strands are measured by the number of locations in each DNA strand where the bases differ.

Centroid Recalculation:

Cluster centroids for 2D data points are recalculated by averaging all the data points belonging to that particular cluster, while cluster centroids for DNA strands are recalculated by using the most common base for each position in the strand among all the strands belonging to that particular cluster.

Initialization:

The initial centroids for both 2D points and DNA strands are chosen randomly. If the selection of these random initial centroids is bad the K-means algorithm could take very long to converge or might not converge at all. Hence both the clustering algorithms are run for exactly 100 iterations before termination.

DNAStrandGenerator

Our DNAStrandGenerator takes in parameters specifying the following:

- Number of strands per cluster
- Number of clusters
- Length of DNA Strand

Strand generation works as follows:

1. For each of the k clusters, a ***'base'*** strand is generated. The base strand is essentially the ***'true cluster centroid'*** for the cluster, and all other points that are generated for that cluster are generated using the base strand as a ***'seed'***. Our generator defines a threshold distance between base strands to be at least as large as approximately half the length of a single DNA strand. This means that base strands will differ from one another by a value that is at least as large as the threshold distance. This is to ensure that clusters are sufficiently far away from one another. If the number of clusters specified is too large (relative to the length of the DNA strand), it is possible for the generator to run in an infinite loop searching for a base strand that is sufficiently different from other base strands. To avoid this, set the length of the DNA strand to be a large number relative to the total number of clusters specified.
2. Once all the base strands are generated, each base strand is used to generate strands for the cluster the base strand belongs to. Our generator ensures that each strand that is generated for a cluster differs from it's base strand by a value that is less than half the threshold distance defined between base strands. This ensures that the generated clusters do not *'intersect'*, i.e. it ensures that a strand that is generated from a base strand will always have a smaller distance between it and the base strand that generated it rather than any other base strand. Once again, it is possible for an infinite loop to occur if the number of points per cluster specified is too large(relative to the length of the DNA strand). This is because it is possible for the generator to generate all possible combinations of a DNA strand for a particular length and still not have generated the number of points per cluster specified by the user. To avoid this, set the length of the DNA strand to be sufficiently large.
3. Once all strands are successfully generated, they are written to the file */DNA_DataGenerator/cluster.csv*

MPI Communication Protocol

Implementation of the K-means algorithm for clustering objects in a data set involves assigning a set of objects into clusters/groups such that the degree of similarity between is strong between members of the same cluster and weak between members of

different clusters. Similarity between two objects can also be thought of as the distance between these two objects.

We designed the parallel mode of the algorithm to be a ***master-slave*** configuration. In our design, Process-0, the process with rank 0 acts as the master and processes with higher ranks are slave processes. The master performs the following tasks:

1. Construct an initial list of k centroids. These centroids are chosen randomly from the available data set
2. Send the centroid list to all the slaves.
3. Receive intermediate/final result from the slaves. The data set is divided evenly between all the available slave processes. Every slave process works on it's portion of data set. The intermediate/final result is a mapping of all the objects in that slave's range to the centroids they are closest to.
4. Recalculate the centroid locations based on the results received from the slaves.
5. Repeat steps 2, 3, 4 for the number of iterations specified.
6. At the end of all the iterations send a completion message to all the slave processes for them to execute MPI.Finalize()

The slave processes perform the following tasks:

1. Construct the portion of data set to work on by reading from the cluster.csv.
2. Receive an MPI message from the master
    a. If this message has *COMPLETION_TAG*, perform *MPI_Finalize()* and exit the loop.
    b. Else if this message has *CENTROID_TAG*, then goto step 3
3. For every data object in it's portion calculate its distance from all the centroids and determine the centroid it is nearest to. Assign the data object to this centroid.
4. Send the result back to the master

MPI Features Used

We have made use of the following MPI features in our implementation:

- **Send / Receive**: We have used the blocking versions of MPI send and receive routines. This ensure in-order delivery of both send and receive messages on the master and slave sides.
- **Tags:** We have used different tags for different types of messages. For ex., *CENTROID_TAG* when sending over a list of centroids from master to slaves, *RESULT_TAG* when sending over the result from slaves to the master, etc. This provides a clean way of distinguishing between different types of messages.

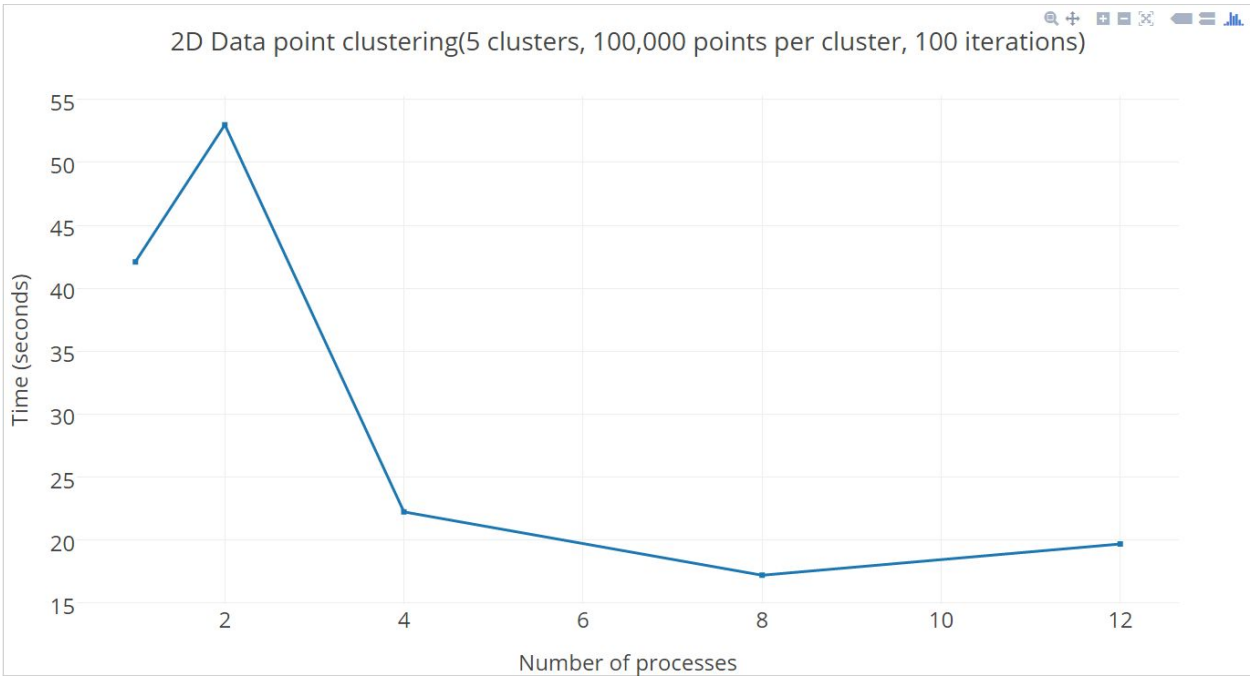
- **MPI.OBJECT:** We wrap up the buffer to be sent across in an array of *Object []* and then send it.

Experimentation and Analysis

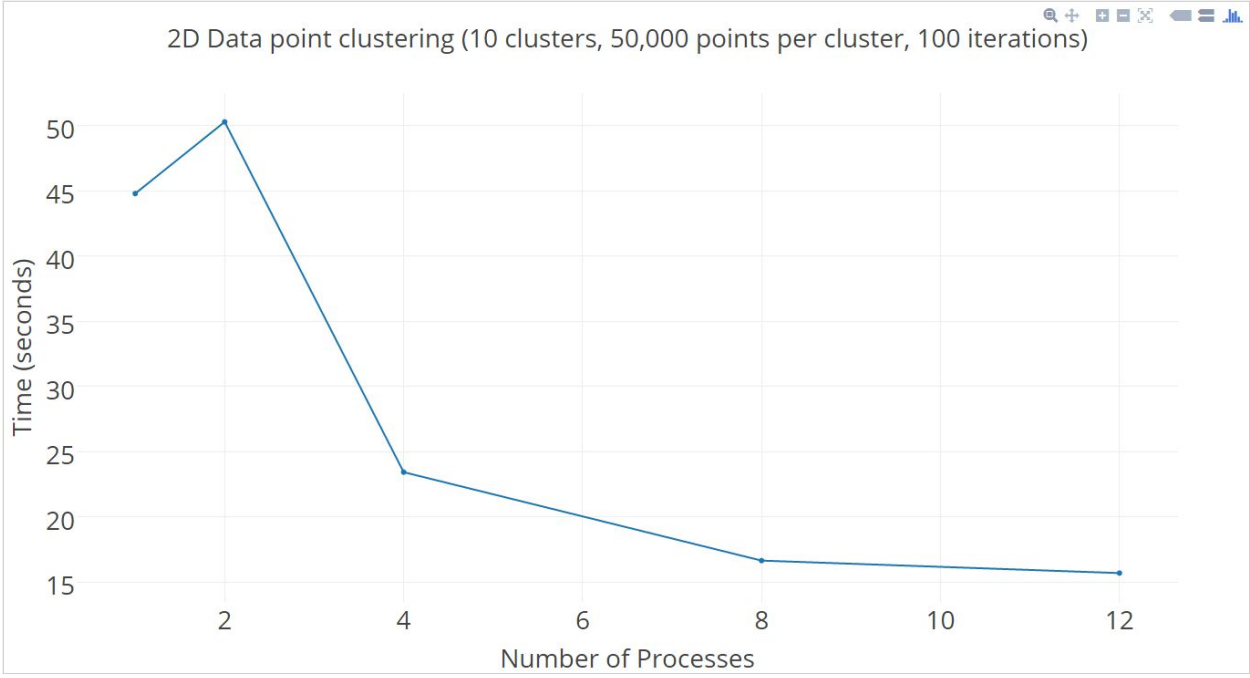
Note that the number of iterations for the parallel mode of algorithm is set of 100 for both 2D data points and DNA strands. All the below numbers are in seconds. We perform 3 runs for each mode in each configuration and plot the graphs using the average of the values from the 3 runs.

2D data points

- 5 clusters, 100,000 points per cluster
    - Sequential
        - Run-1 : 42.377
        - Run-2 : 42.149
        - Run-3 : 41.765
    - Parallel, number of processes = 2
        - Run-1 : 51.530
        - Run-2 : 54.219
        - Run-3 : 53.219
    - Parallel, number of processes = 4
        - Run-1 : 21.242
        - Run-2 : 22.830
        - Run-3 : 22.623
    - Parallel, number of processes = 8
        - Run-1 : 15.913
        - Run-2 : 16.017
        - Run-3 : 19.670
    - Parallel, number of processes = 12
        - Run-1 : 22.130
        - Run-2 : 19.153
        - Run-3 : 17.763



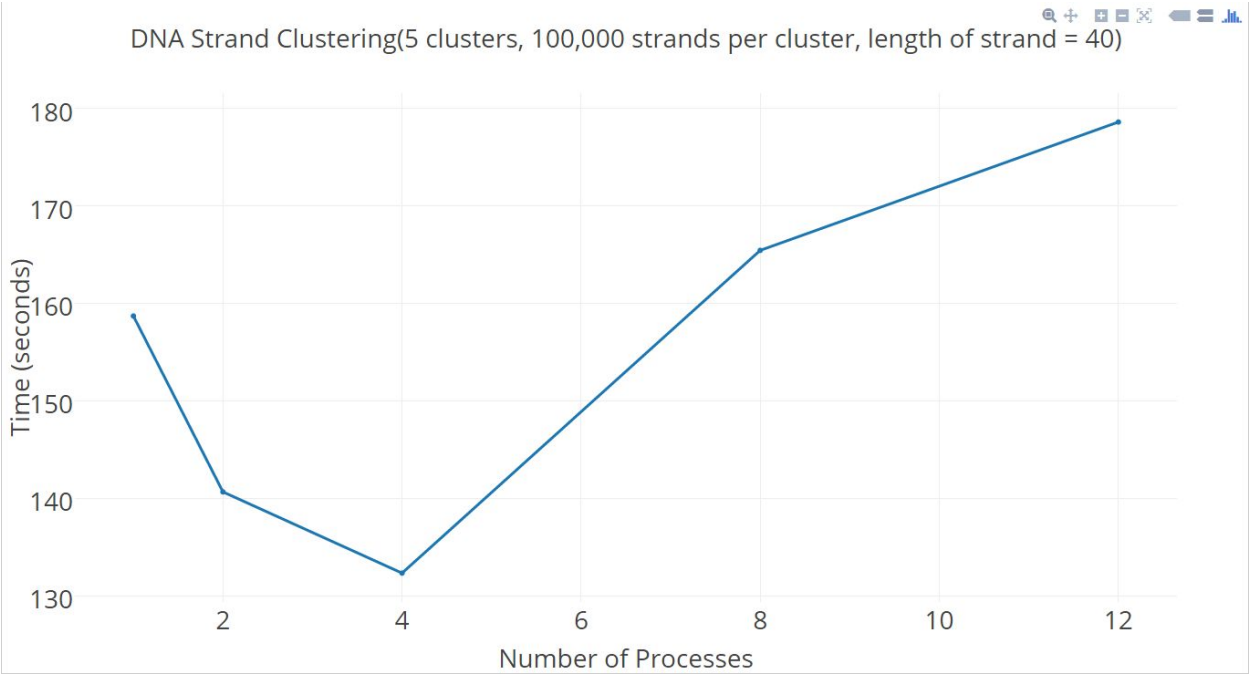
- 10 clusters, 50,000 points per cluster
  - Sequential
    - Run-1 : 44.528
    - Run-2 : 46.629
    - Run-3 : 43.169
  - Parallel, number of processes = 2
    - Run-1 : 49.027
    - Run-2 : 49.637
    - Run-3 : 52.143
  - Parallel, number of processes = 4
    - Run-1 : 20.430
    - Run-2 : 25.340
    - Run-3 : 24.500
  - Parallel, number of processes = 8
    - Run-1 : 20.948
    - Run-2 : 13.444
    - Run-3 : 15.544
  - Parallel, number of processes = 12
    - Run-1 : 15.795
    - Run-2 : 15.699
    - Run-3 : 15.575



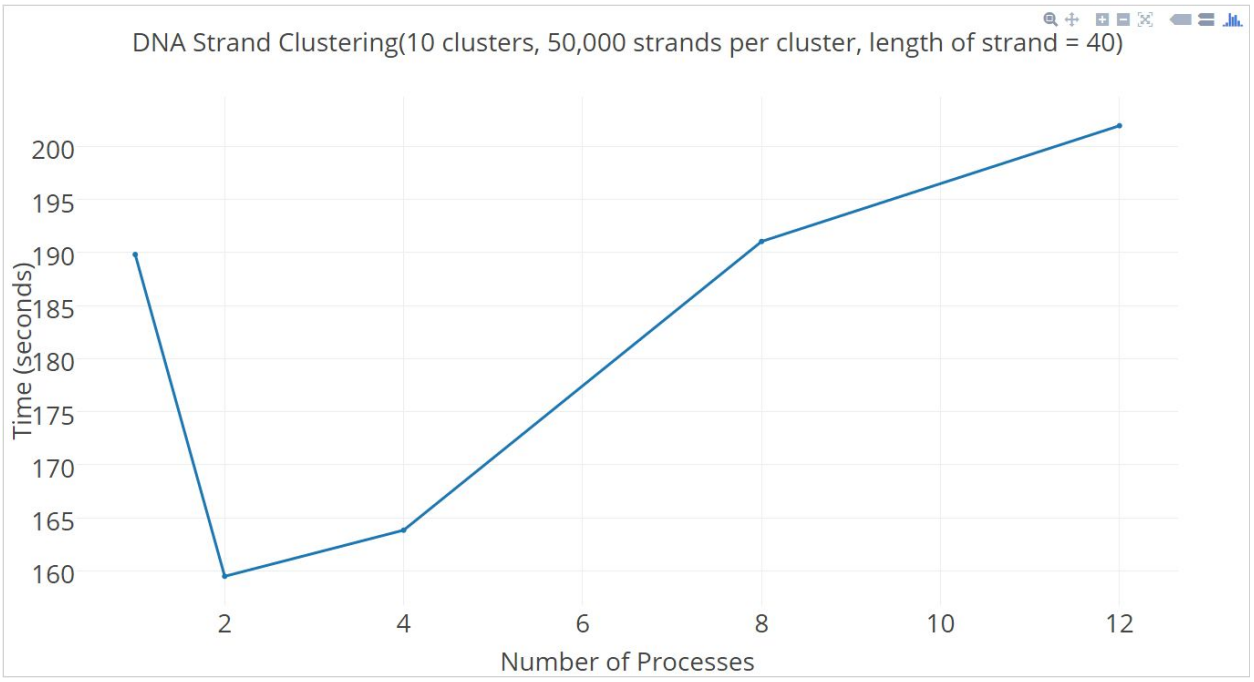
DNA strands

- 5 clusters, 100,000 DNA strands per cluster, DNA length - 40
    - Sequential
        - Run-1 : 156.917
        - Run-2 : 154.310
        - Run-3 : 164.857
    - Parallel, number of processes = 2
        - Run-1 : 132.436
        - Run-2 : 146.213
        - Run-3 : 143.287
    - Parallel, number of processes = 4
        - Run-1 : 140.183
        - Run-2 : 125.396
        - Run-3 : 131.355
    - Parallel, number of processes = 8
        - Run-1 : 170.087
        - Run-2 : 145.621
        - Run-3 : 180.592
    - Parallel, number of processes = 12
        - Run-1 : 160.025
        - Run-2 : 172

        - Run-3 : 203.79



- 10 clusters, 50,000 DNA strands per cluster, DNA length - 40
    - Sequential
        - Run-1 : 190.419
        - Run-2 : 185.729
        - Run-3 : 193.269
    - Parallel, number of processes = 2
        - Run-1 : 159.683
        - Run-2 : 163.472
        - Run-3 : 155.305
    - Parallel, number of processes = 4
        - Run-1 : 168.029
        - Run-2 : 163.621
        - Run-3 : 159.851
    - Parallel, number of processes = 8
        - Run-1 : 190.608
        - Run-2 : 194.291
        - Run-3 : 188.204
    - Parallel, number of processes = 12
        - Run-1 : 199.684
        - Run-2 : 204.761
        - Run-3 : 201.378



Analysis:

From all the above graphs we can infer that the run time for the sequential algorithm remains more or less constant for a particular configuration. For the parallel algorithm the run time changes as the number of processes in the MPI environment is changed.

For the parallel algorithm, it can be seen that the run time with 2 processes is almost the same as that for sequential. This is expected because in this case Process-0 (master) will schedule all the distance computation work on Process-1. This means that we essentially have only one process doing all the work which is the same as sequential mode.

When the number of processes is increased to 4, we see the run time decrease in all the above cases. This is because we now have multiple processes performing computation concurrently on different portions of the data set.

As we go on increasing the number of processes in the MPI environment, we see that at some point the run time again starts increasing for the same workload. This happens because of the following reasons:

- A node can only run as many parallel processes as the number of cores/processors it has. So there are only so many processes that can be run parallelly on a single node.
- As the number of processes increases the overhead of MPI communication exceeds the cost benefit received by parallelizing the computation.

For the tests that we carried out, we see that the **sweet spot** for 2D data set is achieved at 8/12 processes and that for the DNA Strand data set is achieved at 4 processes. Thus we can see that this spot is dependent on the type of the data set we are running K-means on and will differ from one data set to another data set.

UserGuide: Instructions for compiling and running our framework

1. Copy the *'src'* folder into your working directory, and change directory into the *'src'* folder.
2. There will be 3 scripts present in the directory: *compileClusteringJar.sh*, *generate2DPoints.sh*, and *generateDNAStrands.sh* . Run the *'chmod 700'* command on all these scripts.
3. Edit the *generate2DPoints.sh* and *generateDNAStrands.sh* scripts, and run them to generate the desired number of clusters and points per cluster. Note that *generateDNAStrands.sh* also allows you to edit the length of the DNA Strand. If the generateDNAStrands script appears to running infinitely, please increase the length of the DNAStrand and run the script again. The reason for this behavior is explained in the 'DNAStrandGenerator' section of our report.
4. Run the *compileClusteringJar.sh* script to compile the program
5. Edit the *machines.txt* file to specify which nodes you wish to run on.
6. To run the program sequentially use the following command:

mpirun -np 1 -machinefile machines.txt java -cp ./clusteringsrc Driver <k> sequential <type>

where <*k*> is the number of clusters and <*type*> is either *'point'* or *'dna'*. For example, to perform sequential clustering on the generated 2D data points using 5 clusters, run:

mpirun -np 1 -machinefile machines.txt java -cp ./clusteringsrc Driver 5 sequential points

7. To run the program parallely, use the following command:

mpirun -np <numberOfProcesses> -machinefile machines.txt java -cp ./clusteringsrc Driver <k> parallel <type>

where <*numberOfProcesses*> is the number of processes you wish to run, <*k*> is the number of clusters, <*type*> is either 'point' or 'dna'. For example, to perform parallel clustering on the generated DNAStrands using 5 clusters and 4 processes, run:

mpirun -np 4 -machinefile machines.txt java -cp ./clusteringsrc Driver 5 parallel dna

Note that the value of <numberOfProcesses> should be greater than or equal to 2, due to the 'master-slave' communication protocol used by our parallel clustering algorithms. This is explained in greater detail in the *'Overview of Design'* section of our report.

7. After a successful run, the program will output a list of *k* centroids for the given data set.

Side Notes:

Note that the clustering algorithms may not return the number of clusters specified by the user. This can occur if no points are assigned to a particular cluster centroid.