15-440/640 Distributed Systems

Lab 4 Report: MPI: Clustering and DNA

Team:
Abhishek Bhowmick (abhowmi1),
Neil Rajesh Dhruva (ndhruva)

Date:
12/05/2014

Contents

Problem Definition

Clustering is a task of grouping objects that are more similar to each other as compared to other objects in the dataset. Hence, the objects with the closest 'distance' to each other are put in one group. The goal of this project is derived from the idea of grouping similar objects using a distance metric in a parallel computation setting. The two main objectives of the project include:

1. Understand and implement a parallel computation algorithm for clustering in large datasets using the OpenMPI framework.
2. To analyze the performance of a parallel algorithm, and compare it with a sequential implementation of the same, for varying dataset sizes and varying degrees of parallelism.

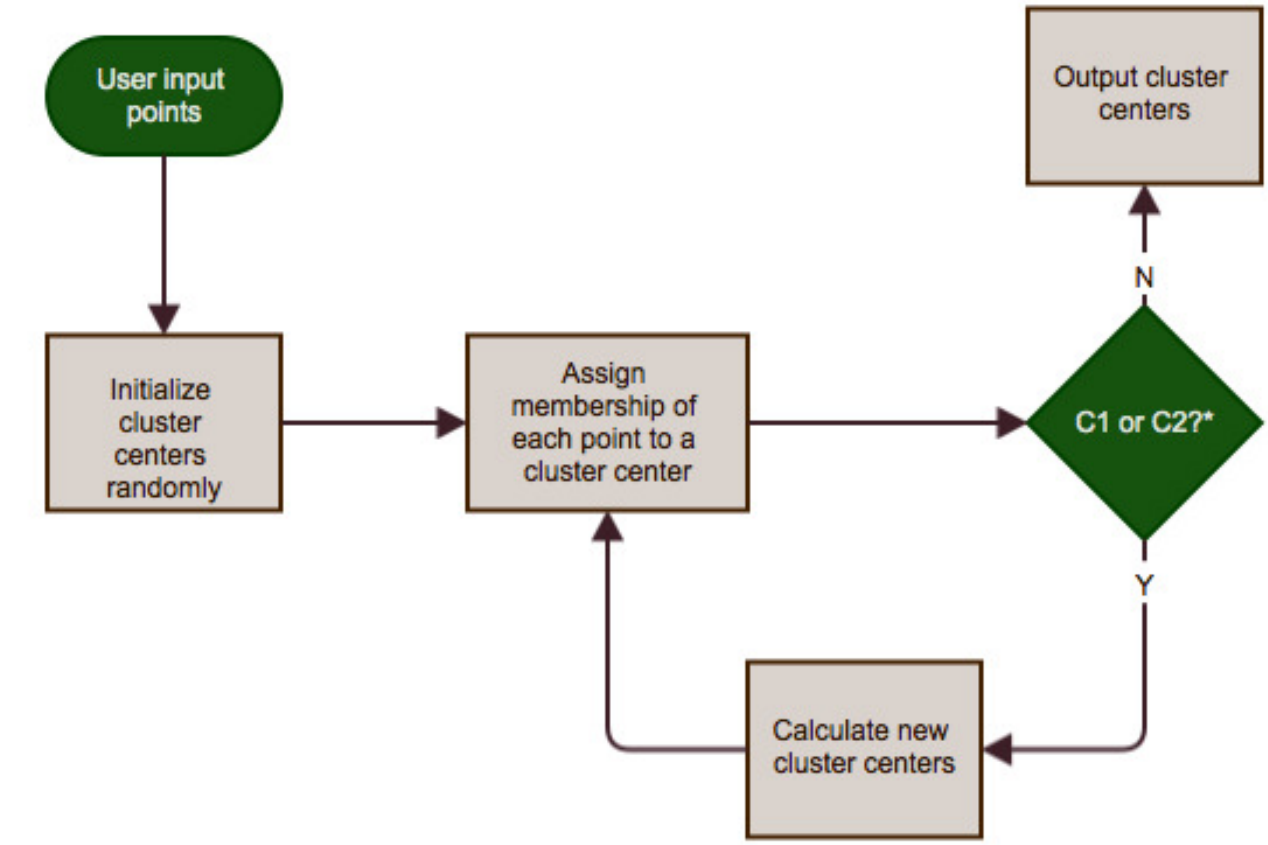
Solution Overview

For this project, we have created parallel algorithms for clustering points in 2D space using the K-means clustering technique, and generalized it for use in the DNA clustering application. In addition, we have provided implementations for generating random datasets of user-defined sizes for both these problems.

For solving the clustering problem using K-means, we use the Euclidean distance between data points to find the data center closest to a given point. The point is then a 'member' of the cluster centered at that data center. In an iterative manner, all points closest to a given cluster center are identified, and the cluster center is recalculated as an average of these points. This calculation is parallelized by distributing the dataset equally among all participating nodes, calculating the membership of points locally on those nodes, and recalculating cluster centers using OpenMPI for communication between the nodes.

This idea is then extended to calculate cluster centers in a dataset containing DNA strands (randomly generated using a script provided with the code). The closest center to each strand is calculated using the string edit distance (assuming DNA strands are represented as strings). The centroid for a set of DNA strands is computed by setting each position of the centroid strand to the most occurring character(base) at the corresponding position for all strands in the cluster. Using OpenMPI for communication, K-means is parallelized for better performance across several nodes.

Clustering Overview



*Condition 1 (C1): Number of membership changes/Total number of points < Threshold
Condition 2 (C2): Number of iterations > Maximum iterations

Figure 1: Clustering Overview

This is the basic clustering algorithm that starts with data input from the user. K clusters centers are initialed randomly to begin the algorithm, and the algorithm iterates till either C1 or C2 is satisfied.

OpenMPI Implementation Overview

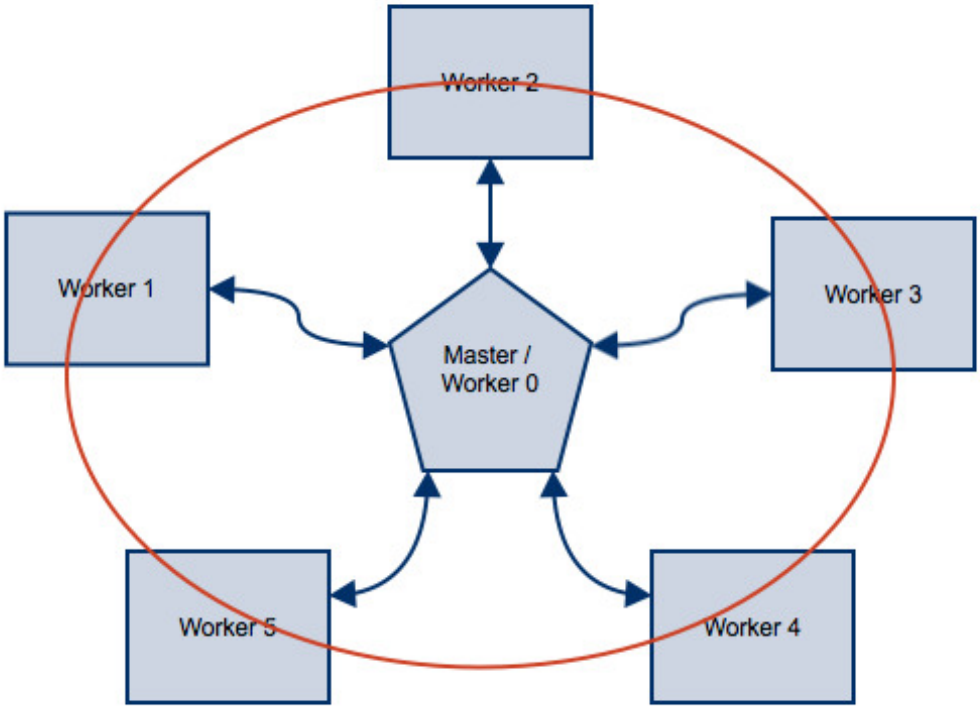


Figure 2: OpenMPI Implementation Overview

The figure implies the following:

1. The master is responsible for reading the initial data and broadcasting parts of it to the worker nodes.
2. The master and the worker nodes then run the clustering algorithm locally, and combine their results after every iteration to calculate new cluster centers.
3. The circle implies that for every iteration, and for new cluster center calculations, all nodes communicate with each other.

Implementation

The implementation has been divided into two parts, one each for the corresponding applications of 2D data points and DNA strands. The major differences between these two implementations lie in the calculation of cluster centers from the points belonging to the cluster and the calculation of distance between two points.

Both implementations are divided into the following phases:

1. Dataset generation: Dataset generation for the 2D dataset is done using the starter script provided with the handout. We have written a similar script to generate the dataset for DNA strands. We take in two parameters similar to the provided script (number of clusters and number of points per cluster). First, we randomly generate a number of points equal to the number of clusters and indicate these as centroids. For each of these centroids, we generate a number of points equal to the value of the second user supplied parameter. These points are generated by mutating each position of the centroid strand (with probability 0.2). All the points are then shuffled and accumulated to build our dataset. The result is a set of points that possesses with good underlying clusters (we also save the list of generated centroids to validate our solution against). We represent the DNA points as comma-separated strands, where each base has value ‘a’, ‘c, ‘t’ or ‘g’.

2. User input: Instructions to invoke the data generator scripts and run the sequential and MPI implementations of clustering are provided in the README file. We mainly require, from the user, the input file, number of points in the dataset, number of clusters to be generated, and the dimensions of the dataset (especially in the case of DNA strands). We assume, as mentioned in the handout, that the DNA strands are all of equal length.

3. Centroid Calculation:

    a. *2D Dataset:* For the 2D dataset, we represent each input point as an array of double values. Hence, for calculating

centroids, we sum up all points that are members of a particular cluster, and take the average of these points by dividing the sum with the total number of points belonging to that center. This gives us the new center for that cluster.

b. *DNA Strands Dataset:* For the DNA strands dataset, we use a different approach for centroid calculation. The distance function for this dataset is defined by the number of different bases between the corresponding positions of two DNA strands. Hence, a new center (centroid) is calculated based on the maximum frequency of a base at a given position in the strand from among all strands belonging to a cluster.

4. Sequential KMeans: For sequential Kmeans calculation, the input is read from user file, and an initial set of centroids (randomly selected from the dataset), equal in number to the user supplied cluster number parameter, is generated. This is followed by the membership assignment phase, where each point in the dataset is assigned to its closest center. Now, the new centers are calculated based on the centroid calculation techniques mentioned in point 3 above. Finally, we make use of two stopping conditions, whichever is satisfied first:
   a. We use the ratio of number of points that change membership in an iteration, to the total number of points. If this ratio is low, then the centroid calculation is stopped.
   b. We also use a maximum iterations parameter. If the number of iterations it takes to converge to the actual centers reaches this maximum limit, then the process is stopped even if the threshold in (a) is not met.

5. Distributed KMeans: For distributed Kmeans calculation, we use the OpenMPI platform. We run our Kmeans calculations in parallel on several processors using this platform. The input is read from the user input file by the master in a manner similar to sequential Kmeans. However, in addition, the master then distributed equal number of points to all the processors (worker nodes) involved, including itself. Additionally, initial cluster centers are randomly chosen from the dataset provided by the user and are 'broadcasted' to the worker nodes by the master.

On each worker node, of which the master is also a part now, the membership of each point assigned to that node is calculated by calculating the closest cluster centers. In addition, each node calculates the sum of all points belonging to a cluster for the 2D dataset, and the maximum frequency of all strands in the DNA dataset locally. Once all nodes finish, the 'all reduce' functionality provided by OpenMPI is used to get the global sum for each cluster (for the 2D dataset), and the global frequency count for each cluster in the DNA dataset. Each node also adds up the number of points belonging to each cluster center locally, and this sum is then added using the all reduce function to get a global sum of the total number of members per cluster.

Each node now has information on the total sum or frequency count for each cluster, and the total number of members in the cluster. Hence, each node can individually calculate the new cluster centers and proceed with the next iteration on the points assigned to it.

The stopping conditions are enforced by keeping track of the number of membership changes that take place in each iteration, globally, using the all reduce function. Hence, each node undergoes the same number of iterations, and exits the loop simultaneously when a stopping condition is met.

Analysis

For analyzing the performance of our distributed Kmeans implementation against the sequential implementation, we test for increasing number of data points for varying number of cluster centers, and increasing the number of parallel nodes executing the algorithm. Since our algorithm selected random initial cluster centers, we ran the algorithm for each triple of (#points, #clusters, #processors) multiple times (typically, 3) and took the average run time for each such triple. Hence, it is a good approximation of the total time taken for each such triple. We ran our tasks on a total of 5 machines.

For the **2D dataset**, the following tables (time in seconds) show the run time in seconds (obtained through the time command in Unix). The graphs show the trends observed in the data:

Num clusters = 5

Num points	Seq	P=2	P=4	P=6	P=8	P=12
100000	3.552	4.039	4.098	4.725	4.946	4.631
500000	8.403	6.746	6.185	6.35	6.272	5.943
1000000	13.233	9.608	6.956	6.695	6.186	6.901
2000000	21.809	15.658	10.01	8.681	8.831	8.39

Num Clusters = 10

Num points	Seq	P=2	P=4	P=6	P=8	P=12
100000	11.079	10.01	9.471	10.322	10.181	12.69
500000	26.613	16.322	12.782	14.198	12.164	11.923
1000000	48.103	28.283	19.827	16.169	16.832	16.904
2000000	55.882	36.784	24.333	20.18	17.497	21.249

Num Clusters = 15

Num points	Seq	P=2	P=4	P=6	P=8	P=12
100000	21.261	15.539	17.982	17.435	19.357	17.48
500000	33.224	26.744	24.111	19.247	20.405	18.666
1000000	85.805	38.148	43.525	26.483	23.462	24.253
2000000	110.79	55.124	40.683	35.954	30.954	33.172

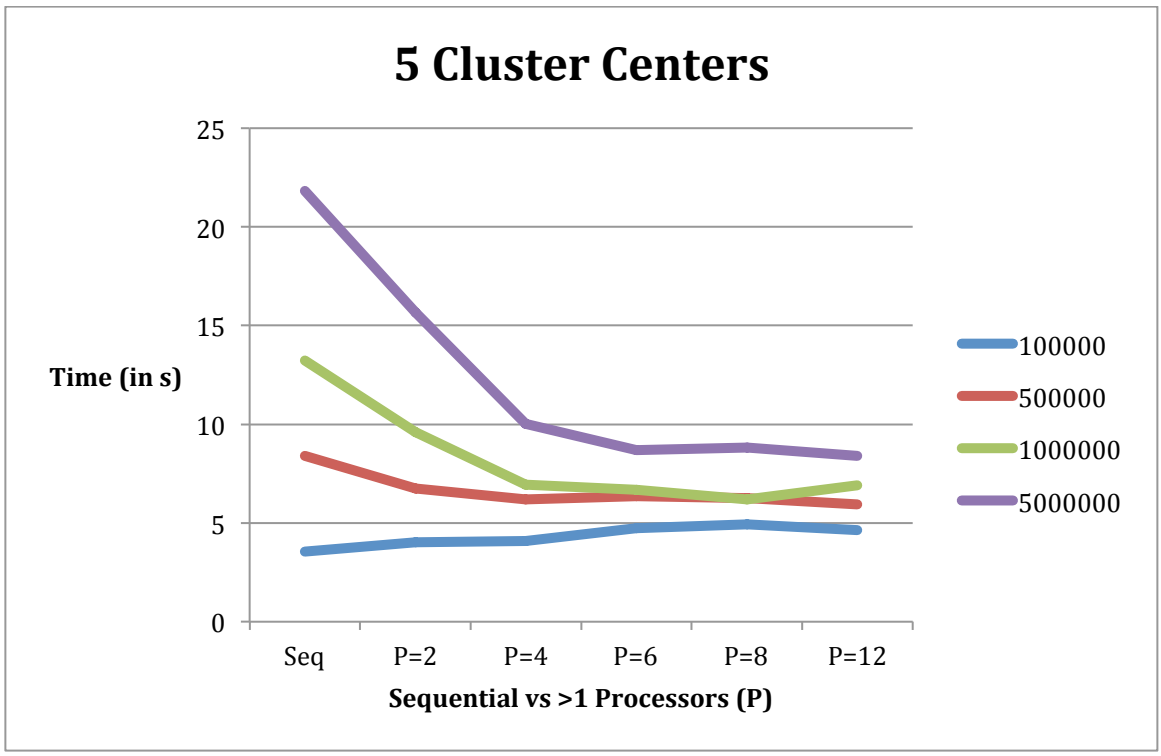


Figure 3: 5 Clusters

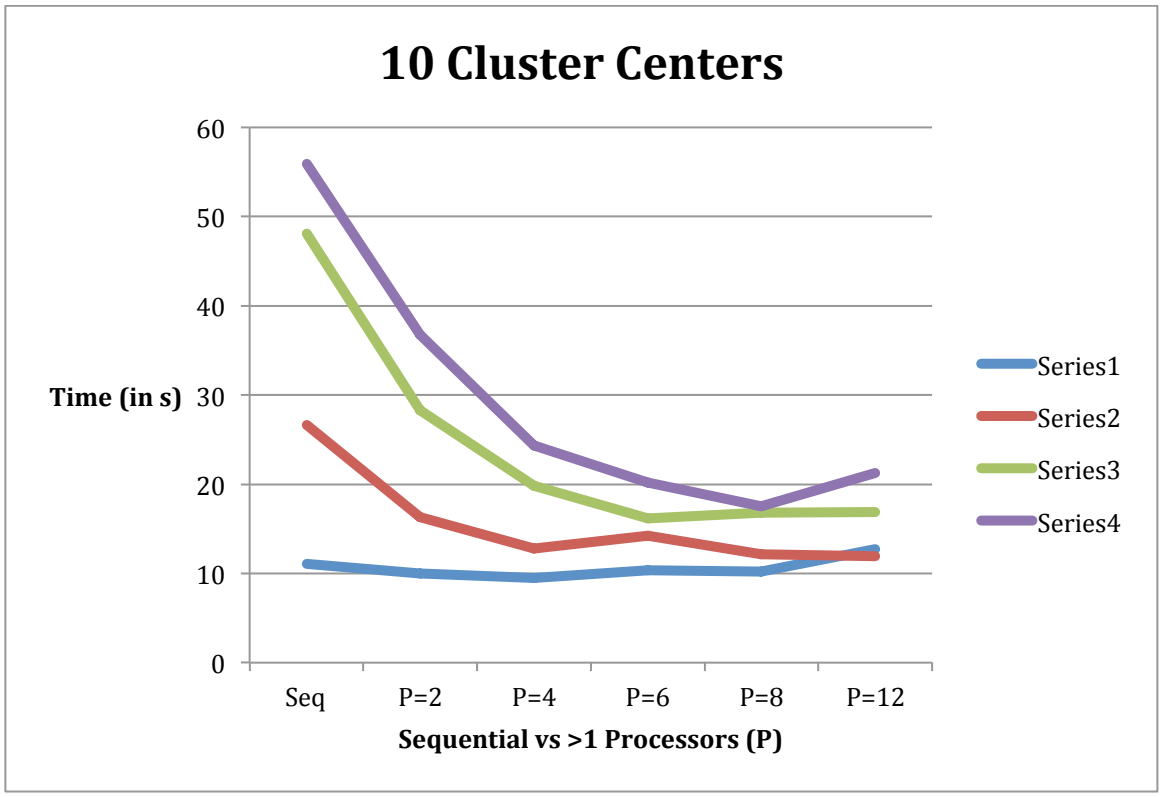


Figure 4: 10 Clusters

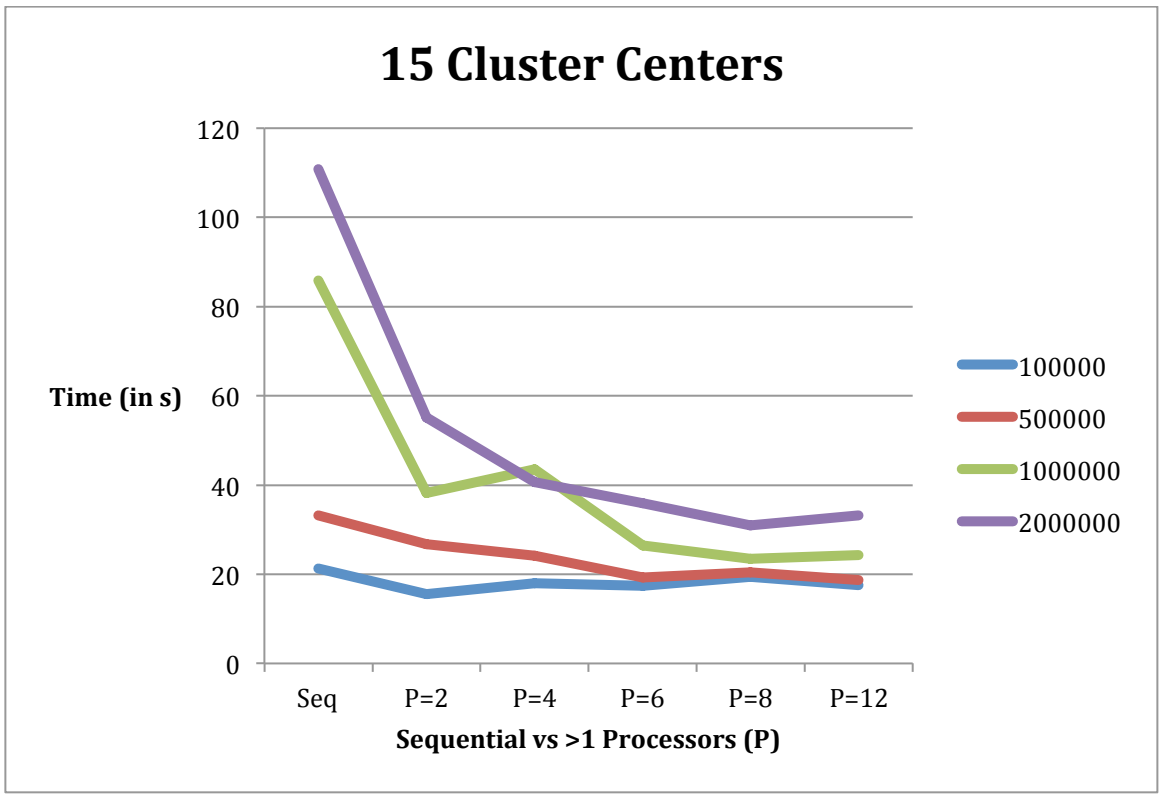


Figure 5: 15 Clusters

For the **DNA dataset**, the following tables (time in seconds) and charts show the trends:

Num Clusters = 5

Num points	Seq	P=2	P=4	P=6	P=8	P=12
100000	3.873	4.335	4.935	5.461	5.244	5.58
500000	11.16	10.731	6.94	10.72	9.727	6.583
1000000	9.052	8.773	6.963	9.407	8.012	8.313
5000000	30.373	25.772	21.056	21.601	20.888	21.357

Num Clusters = 10

Num points	Seq	P=2	P=4	P=6	P=8	P=12
100000	8.771	10.129	9.494	10.737	10.774	11.046
500000	13.046	11.758	12.218	12.142	12.88	13.405
1000000	17.874	15.212	16.348	13.429	14.621	17.493
5000000	54.701	38.8	41.514	33.821	32.97	33.601

Num Clusters = 15

Num points	Seq	P=2	P=4	P=6	P=8	P=12
100000	14.999	14.826	14.537	15.392	15.836	15.336
500000	21.247	17.776	17.671	17.289	17.71	18.44
1000000	26.537	24.729	20.992	20.983	20.038	19.28
5000000	76.42	58.812	46.886	54.812	42.111	39.126

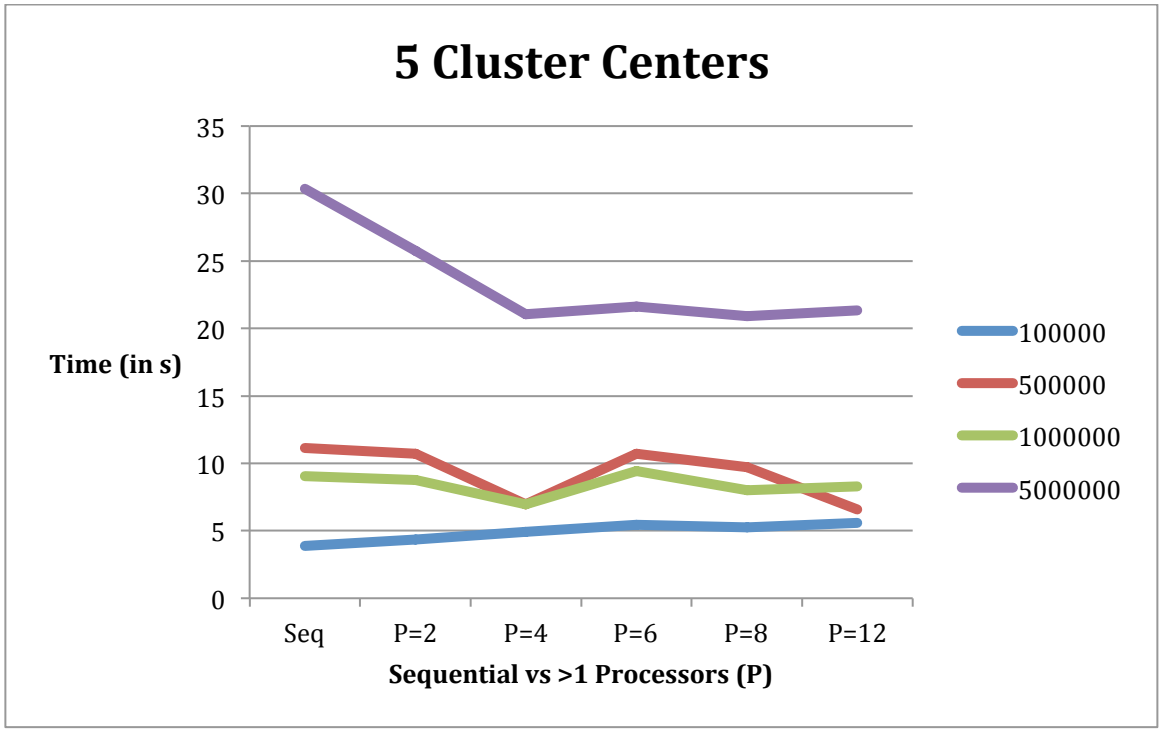


Figure 6: 5 Clusters (DNA)

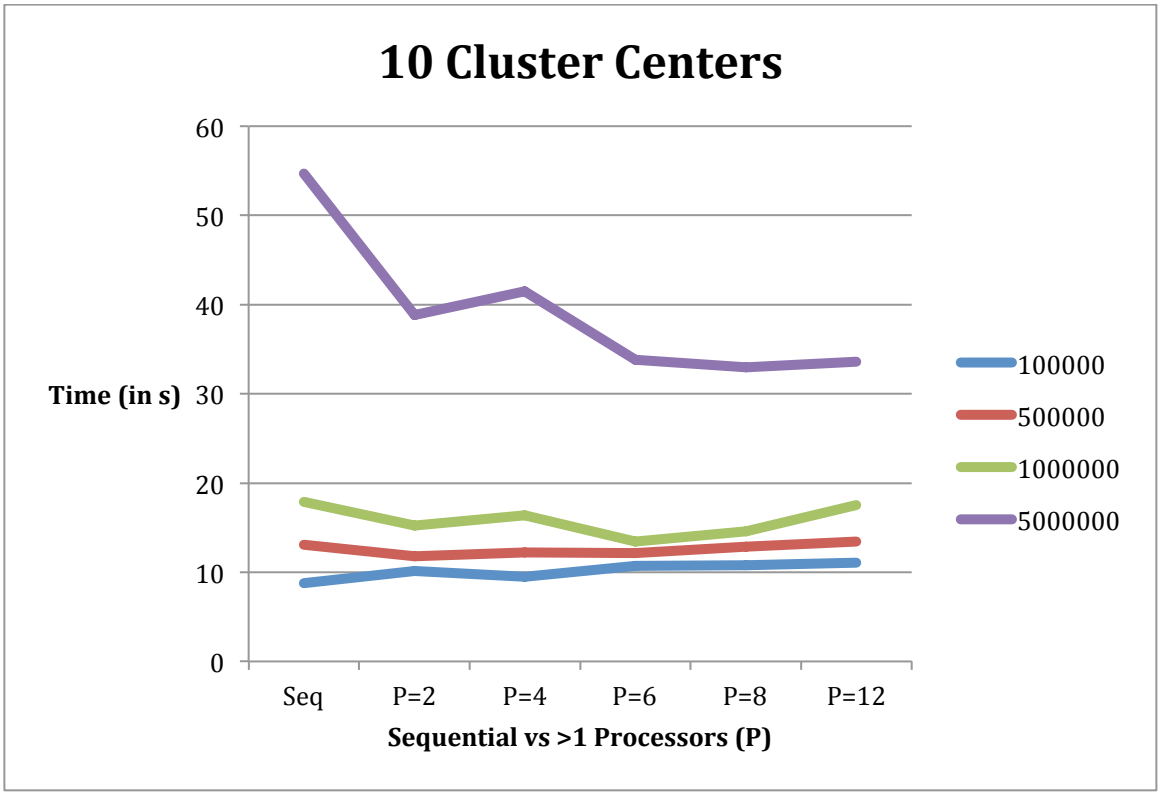


Figure 7: 10 Clusters (DNA)

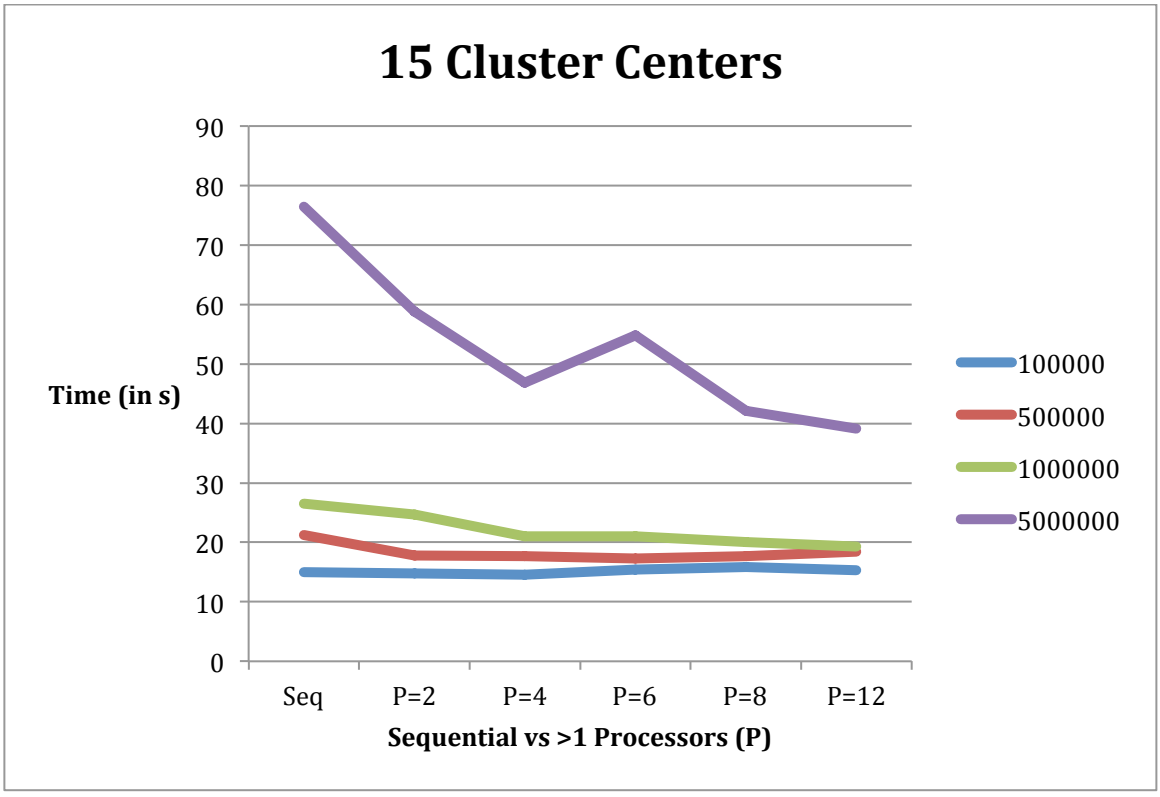


Figure 8: 15 Clusters (DNA)

Observations:

The tables and charts illustrate a clear trend of improved performance due to an increase in the number of processors. However, we observed the following details in the tables:

1. Due to the randomized initialization of cluster centers, we will get varying convergence rates. Some of the observations, like in Figure 5, P=8, and the line corresponding to 500000, we get a small increase in time which is unlikely. This is possibly a consequence of the random initialization of cluster centers.
2. Nonetheless, the general trend in almost all cases is that the amount of time taken for the entire process of IO and clustering decreases with increase in the number of parallel clustering jobs for a given dataset. We see that we achieve an increase of up to 4 times when using between 4-8 machines. But, we observe marginal diminishing returns after P=4, and the speedup does not scale with the number of worker nodes.
3. In fact, we observed that in many cases, that the time goes up for P=12, as compared to P=8. The reason for this could be the fact that our datasets are not large enough for the need to run them on 12 worker nodes. In this case, the communication time between the nodes is higher, overshadowing the benefits of parallelization.
4. This leads us to believe that, for the size of our 2D datasets, P=8 is kind of a sweet spot, and that we cannot expect much better performance with increase in P beyond 8.
5. For DNA datasets, we hit a sweet spot at P=4, followed by a slight increase and then a gradual decrease in execution time (explained in 1).
6. For DNA Strand datasets, we start observing the interesting trends over the number of processors only for the dataset of size 50 million. This leads us to believe that given the size of our DNA strands and the amount of variation among our points, parallelization will help in cases for input datasets that are more than 50 million in size.
7. In addition, we also observed that the more the number of clusters, the more the time it takes to converge. This is evident from the increase in the time for each execution for #Clusters=15 as compared 10, and the same for 10 compared to 5.
8. Though not evident form the dataset, we observed speedups due to reading a file from the cache. The first time a large input file is

read, the IO time is quite large. However it significantly decreases on reading the same file immediately again.

Conclusion

We implemented the KMeans clustering algorithm on 2 datasets: a dataset of 2D points, and a dataset containing DNA strands. We implemented the algorithm in both, a sequential manner, as well as a parallel execution algorithm. By testing the parallel execution algorithm using OpenMPI, and comparing it to the sequential implementation, we observed a significant improvement in the total run-time of the KMeans clustering algorithm.

We achieved an improvement of as much as 4 times in the parallel execution case, as compared to the sequential execution. However, our algorithm was not able to gain significant improvement on the datasets we used beyond using 8 worker nodes in parallel. We believe, however, that with larger datasets, we will be able to obtain better performance gains with increased number of parallel workers. Another interesting interaction to observe would be the effect of parallelization on DNA datasets with different amounts of variation and strand sizes.