The parallelization of k-means using MPI for clustering XY-coordinate points involves four steps:

1. First, the input data is read by the root process. The reason that the data is not read in parallel on all the processes is because an MPI system makes no guarantees about the homogeneity of the file systems on the processes in the cluster.
2. Then, the root process scatters the input data across all the processes, such that each process has access only to the points on which it is personally performing calculations.
3. Then, the root initializes random means and broadcasts these means to the cluster.
4. Next, the following is repeated until convergence:
    - Each process assigns their local points to clusters and calculates a sum of the points in each cluster as well as a point count for each cluster.
    - Then, the processes perform Allreduce on their partial sums and cluster sizes so that they each have all of the sums and counts for each cluster
    - Lastly, each process calculates the means by dividing the sums by the counts, and if any means are different than previously, repeat.

Pseudo code for (4):

```
Changed <- False
while changed do:
      changed <- False
      sums <- zeros[k]
      counts <- zeros[k]
      for each point p in local_points do:
            k <- closest cluster to p
            sums[k] <- sums[k] + p
            count[k] <- counts[k] + 1
      end
      counts <- reduce counts to all processes
      sum <- reduce sums to all processes
      for each int k in clusters do:
            new_means[k] <- sums[k] / counts[k]
            if new_means[k] != means[k] do:
                  means[k] <- new_means[k]
                  changed <- True
      end
end
```

Parallel algorithm for DNA strands:

Basic algorithm

- DNA strands are represented as a string with the same length
- Distance between two DNA strands F(S1, S2) is defined as the number of positions on which the characters are different between the two strings.
- Mean: we calculate the mean by going through each position in the string, and count how many times each base appears in all strings, get the base which appears the most times for that position. Concantenate the most frequent bases for each position will give us the mean.

The parallelization of k-means using MPI for clustering DNA strands involves:

1. First, the root process read the data and scatter the data to all processes. The root process also picks the initial centroids and broadcast the initial centroids to all other processes.

2. Each process now have K centroids and part of the data, they are going to calculate the distance between each data they got with the centroids and assign them to the clusters accordingly.
3. To calcualte the centroids for each cluster, we go through each position in the string. For each position, each process count how many times each base appears in the data points they got. And then reduce their partial count to the root. The root will have the total counts for each base in this position. The root will pick the one with the maximum count and use that as the base for this position.
   This repeats until we go through all the positions got get a new mean.
4. The root process compare the new mean with the old mean and broadcast if it's changed or not to all other processes.
   If none of the centroids in K clusters have changed, the program will terminate. Otherwise, the root process will broadcast the new mean. And repeat from step 2.

Pseudo code:

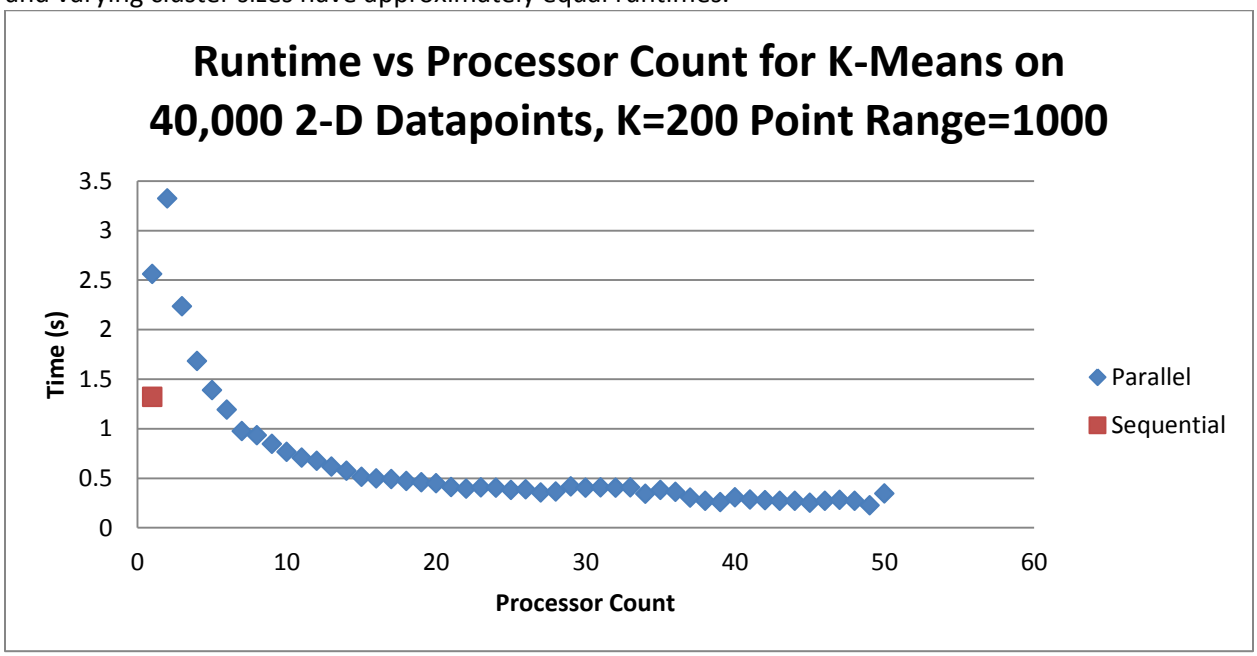
```
Root process
If (!rank):
      Read data
      MPI_Scatterv data
      Pick Initial Centroids
      MPI_Broadcast centroids
repeat while changed:
      changed <- False
      for data in local_data_set do:
            for centroid in centroids:
                  Distance(local_data_set, centroid)
            pick the centroid which is the closest
            assign the data to this cluster
     for each cluster:
            for each position in DAN strand do:
                  for each data in local_data_set:
                        if this position == 'A': countA++;
                        if this position == 'G': countG++;
                        if this position == 'C': countC++;
                        if this position == 'T': countT++;
                  MPI_Reduce countA to 0
                  MPI_Reduce countG to 0
                  MPI_Reduce countC to 0
                  MPI_Reduce countT to 0
                  If (!rank):
                        Pick from {A,G,C,T} which has the highest count
                        Newmean[this position] <- the pick
            If (the mean changed):
                  Changed <- True
      MPI_Broadcast change
      MPI_Broadcast new means
```

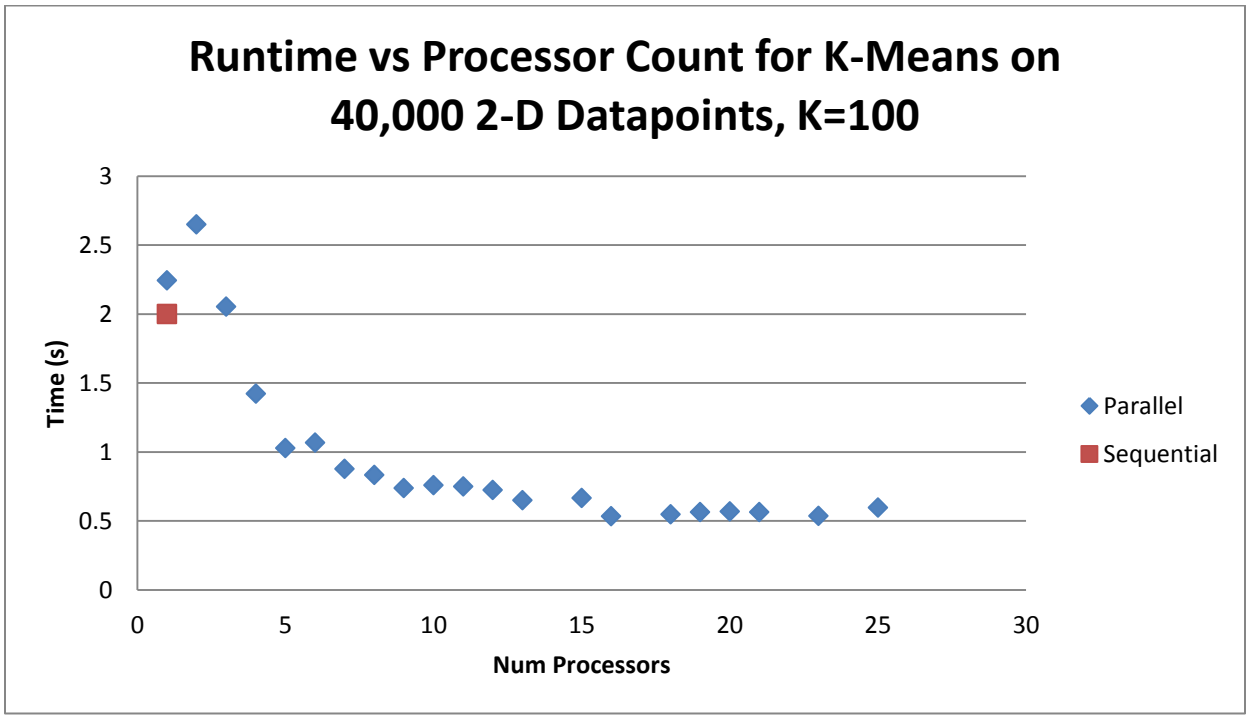
Results for 2-D Data Points:

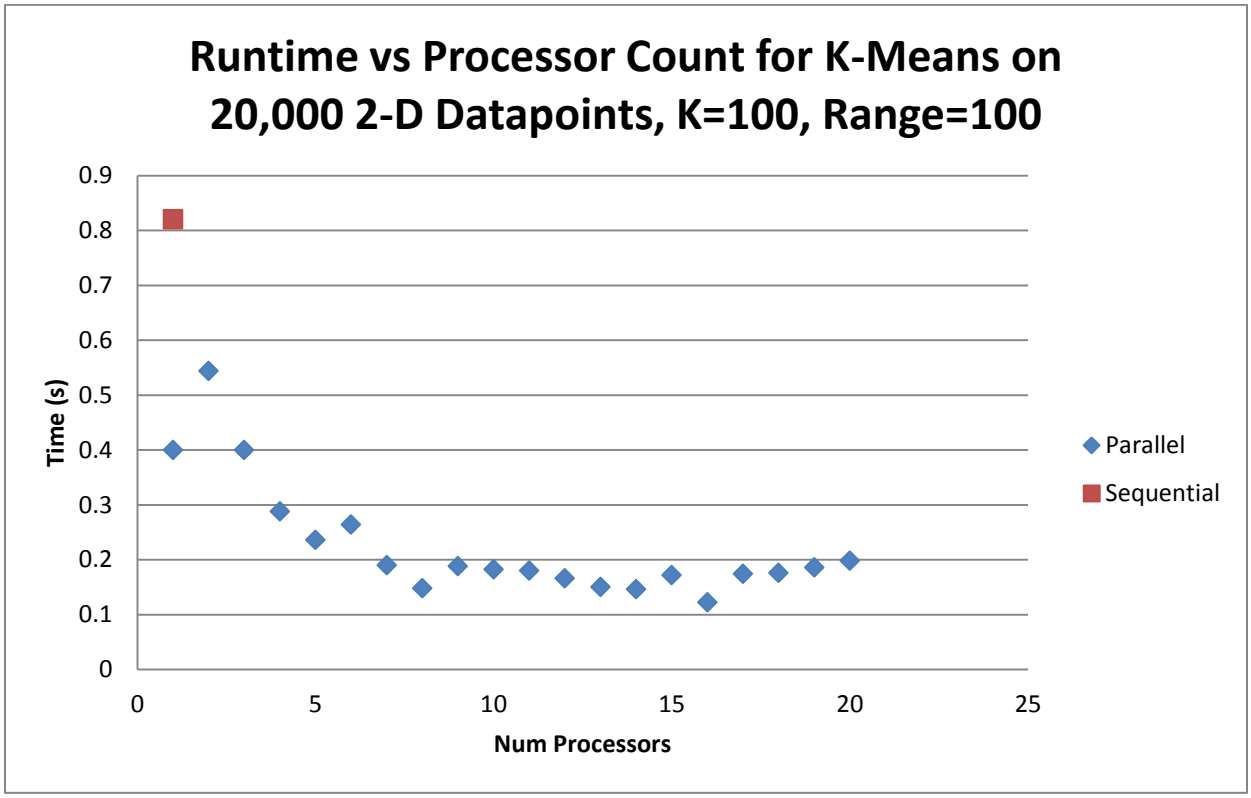
Given 200 clusters each with 200 coordinate points and a deterministic seed, the MPI-based program took between 3.5 and 0.25 seconds. The sequential version took a little bit less than 1.5 seconds and ran faster than the MPI version running 6 or less processes. The speedup of 50 processes compared to 1 was

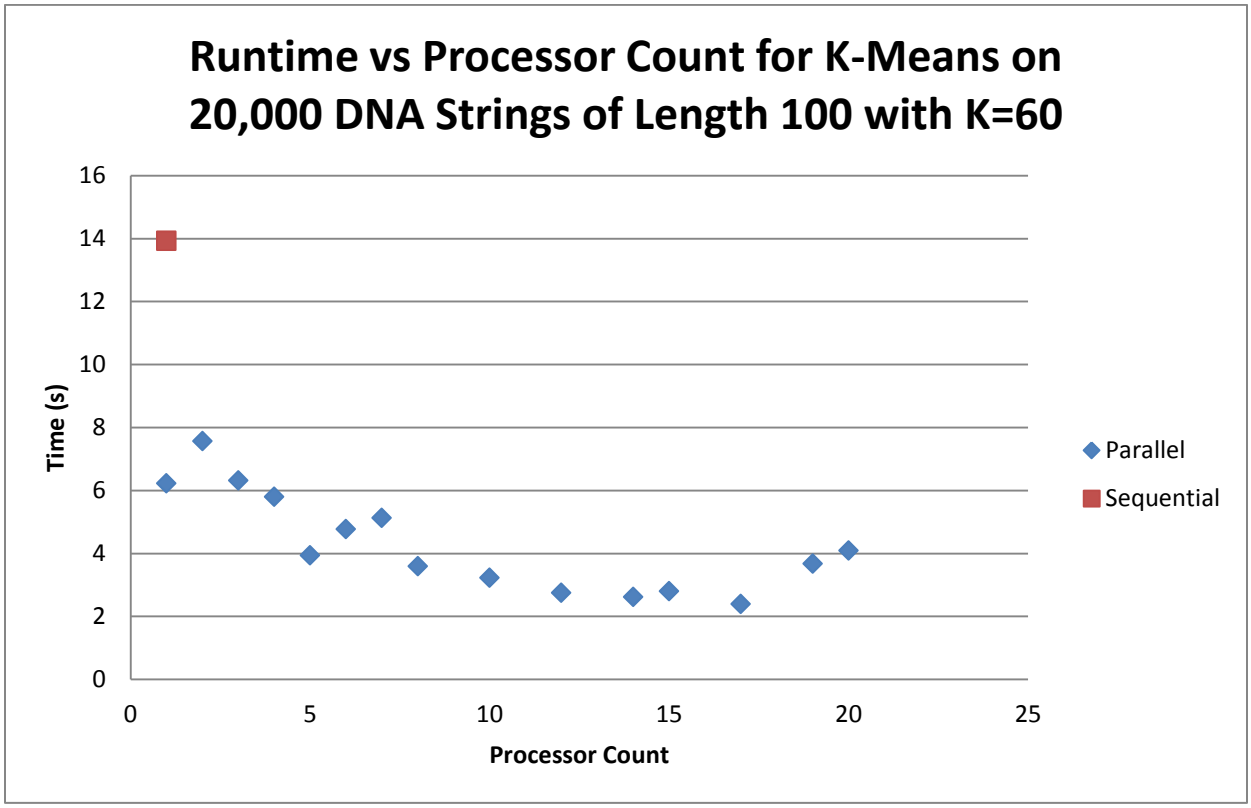
only about 5 times; runtimes experienced pretty dramatic diminishing returns after about 10 processes, which could run the program in around 0.75 seconds.

Because the runtime of the sequential version was not very much, it's likely that a large factor in the low speedup was due to communication overhead. Interestingly, every set of data tested had a bump in speed at np=2, but went down deterministically afterward. I believe this has to do with the fact that 2 processors can't make up for the communication overhead quite enough. One other thing to note is that cluster size seems to have a smaller effect on runtime than expected. The plots with 40,000 data points and varying cluster sizes have approximately equal runtimes.






Runtime vs Processor Count for K-Means on 20,000 2-D Datapoints, K=100, Range=100
Time (s)
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
5
10
15
20
25
Num Processors
Parallel
Sequential


Runtime vs Processor Count for K-Means on 20,000 DNA Strings of Length 100 with K=60
Time (s)
16
14
12
10
8
6
4
2
0
0
5
10
15
20
25
Processor Count
Parallel
Sequential

Readme:

To time the program, we comment out a lot of ouput. To see the result and check the correctness of the program, you can take the comment out.

- Sequential kmeans:
  cd kmeans/seq_kmeans
  make
  ./seq_kmeans [-p|-d] <input_file> <K_clusters> <tolerance>
  -p: for testing 2-D data points
  -d: for testing dna strands
  input_file: the input file name
  K_clusters: how many clusters to get
  tolerance: the algorithm will stop after no more than tolerance number of data points change their cluster assignments. Normally should set it to 0.

- MPI Kmeans:
  cd kmeans/mpi_kmeans
  make
  ./mpigo <numprocs> -k <clustersize> -n <numpoints> -f <inputfilename> <-p | -d>

- DNA cluster generator (in DataGeneratorScripts/randomclustergen/):
  Python generatednadata.py –c <numclusters> -p <numpoints> -o <outputname> -v <strand len>