I started this assignment by spending a lot of time reading through the serial code to understand how each portion of the algorithm worked and how each class (Point, cluster, Kmean) worked together. Doing this made it easier for me to understand how and where I could get speedup.

From there, I decided to parallelize the recalculating center loops. I believed this was where I could achieve the most speedup. I parallelized each loop starting from the inner loop moving outwards. Unfortunately, this resulted in minimal speedup, in fact for < 10^4 rows, the non-optimized sequential application is faster. If sunlab could store larger files, I would have tested with a dataset that has 1 million rows. In this case, I would expect more speedup relative to the sequential.

Next, having spent time implementing the parallel versions, I realized that there was room for some optimization in the sequential version. For one, there was no need to compute the square root and instead of using pow() I could simply multiply the difference in getIDNearestCenter() by itself. These were very basic optimization techniques and likely provided no significant speedup. I also reduced memory references when possible. Additionally, in the kmeans loop, I reordered when the algorithm checks if done == true or iter >= max_iterations so that if the algorithm is done then we do not need to recalculate the center. However, the bulk of the speedup came through significantly reducing the work done by recalculating the center. Each cluster now stores the sums of all the points in vector rather than the points themselves. In this way, I was able to reduce each recalculation to O(total_values). The other optimization came from a small trick I recognized. When getting the ID of the nearest center I only perform the operation if the cluster has changed since the last operation. Each point stores an array of how far it is from each cluster. All in all, I was able to achieve ~100x speedup relative to the non-optimized sequential version for 10^5 rows.

For my most efficient implementation, I parallelized both phases of the kmeans algorithm. To do this I had to use a lock for when cluster.remove/addPoint were called. This implementation was ~500x faster than the other implementations.

I also thought it would be interesting to see how dimensions affected the runtime (Figure 2). Interestingly, the execution time was not really affected by the implementation or dimensions. I also was interested in seeing how the number of clusters in the largest dataset would affect the execution time (Figure 3). Interestingly, execution time decreased for all implementations until 20 clusters, at which point it began increasing again. 20 may be the optimal number of clusters for that dataset.

Figure 1

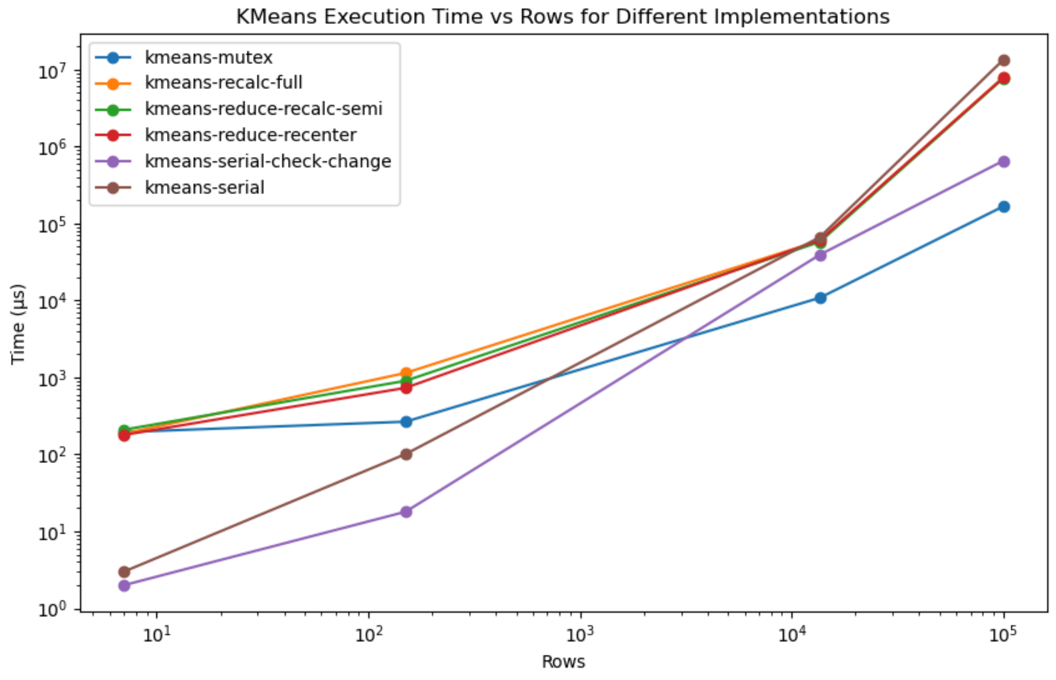


Figure 2

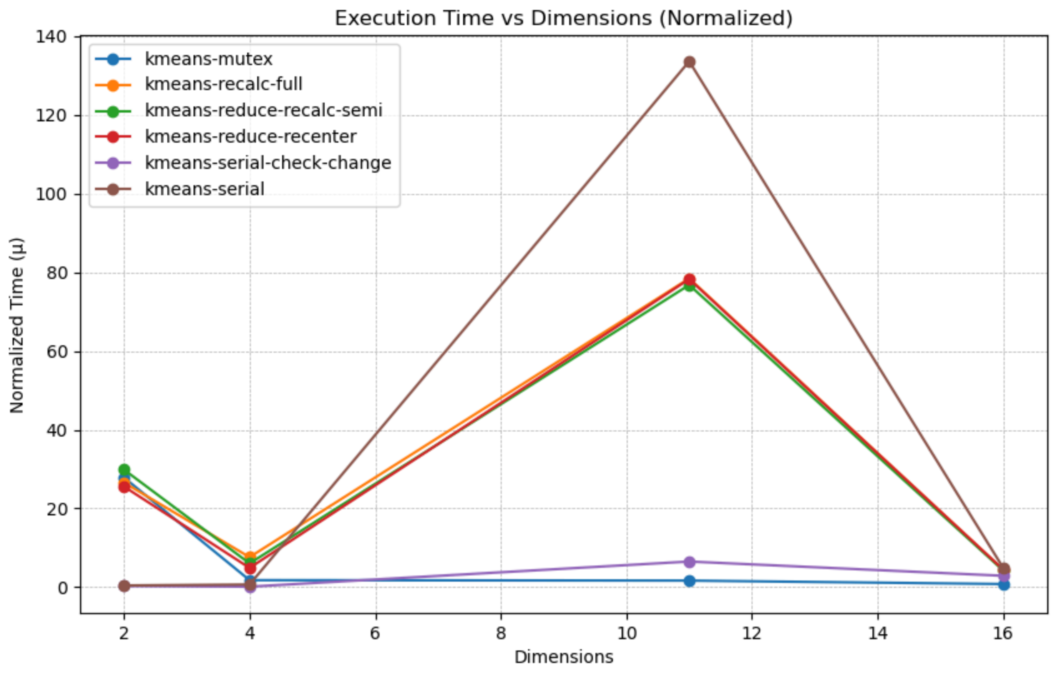
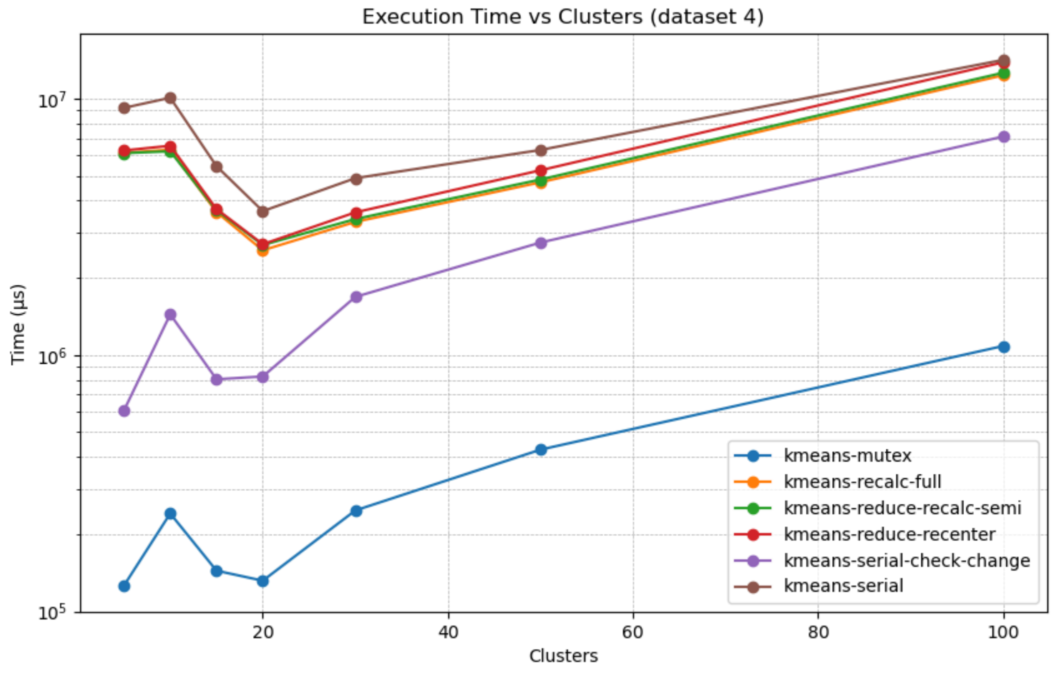


Figure 3



Execution Time vs Clusters (dataset 4)
Time (µs)
Clusters
10^7
10^6
10^5
20
40
60
80
100
kmeans-mutex
kmeans-recalc-full
kmeans-reduce-recalc-semi
kmeans-reduce-recenter
kmeans-serial-check-change
kmeans-serial