Parallel K-Means Report

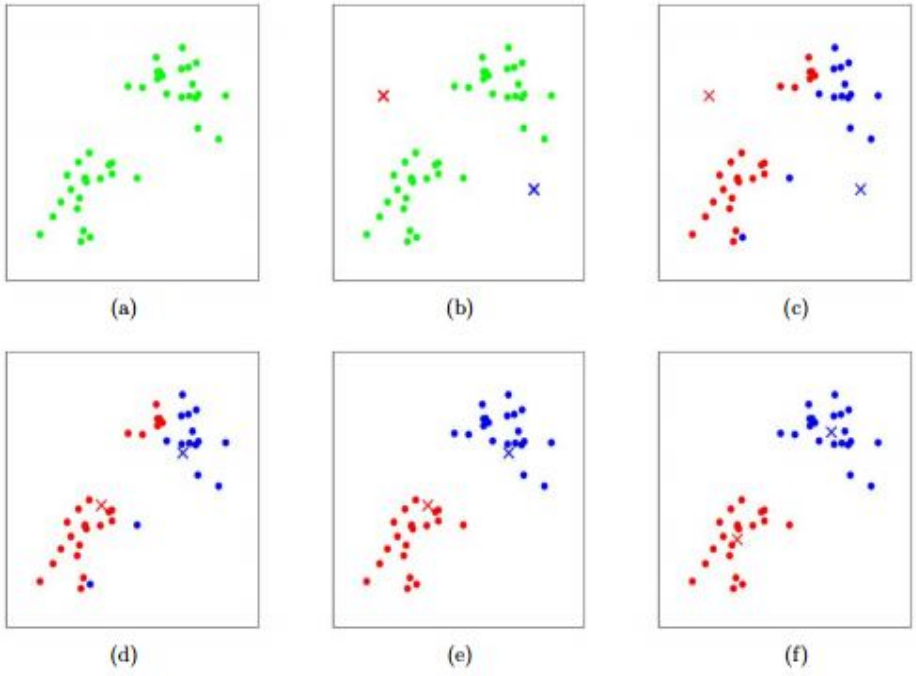
EC 526 - 05/01/2020

Tu Timmy Hoang
Andreas Francisco

Algorithm:

1. Randomly place K points into the space represented by the objects that are being clustered. These points represent initial group centroids.
2. Assign each point to the cluster that has the closest centroid.
3. When all objects have been assigned, recalculate the positions of the K centroids by computing the center of mass.
4. Repeat Steps 2 and 3 until the centroids no longer move.
5. Calculate Within Cluster Sum Of Squares
6. Repeat steps 1 - 5 ten thousand times and use the set of clusters that had the smallest within cluster sum of squares.

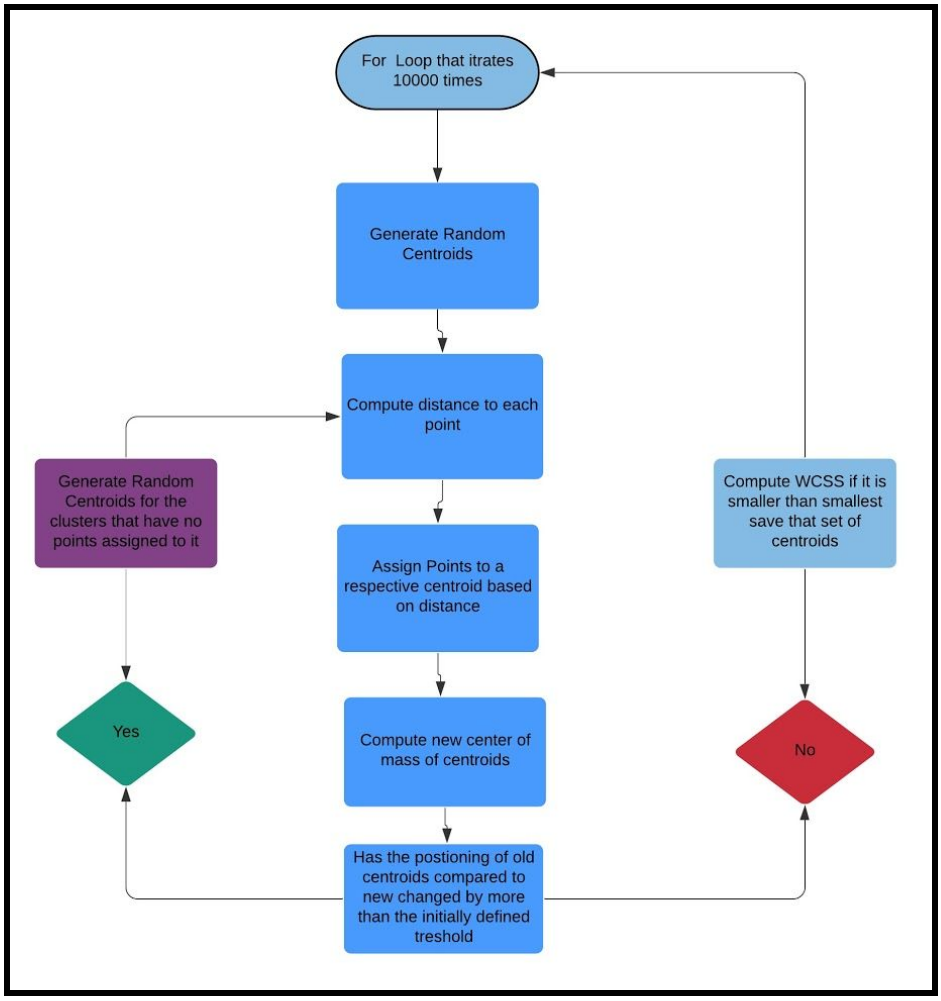
- Here is an example of how K-means works on a dataset:



- Step (a) is just an unlabeled dataset.
- Step (b) is initializing random centroids.
- Step (c) is assigning the points in the grids to the initialized centroids
- Sted (d) is calculating the new centroids by computing the new center of mass.
- Step (e) is doing step (c) but now with the new centroids.
- Step (f) is a combination of steps which can be computing another set of centroids and seeing how those new centroids don't get any new points assigned to them therefore their center of mass doesn't change and the program can halt.

Parallelization:

- Before we go into the parallelization here is a schematics of the serial code:



Open ACC:

- The steps inside the program we tried to parallelize:
    1. We initially thought to parallelize the outermost loop indicated in the chart as the first shape from the top. The iterations of the loop acted independently from each other, the only result that was needed by the end of an iteration was the WCSS, which could be stored inside an array and the centroids that produced the WCSS which could be stored inside a 3D array. Once all for loops were done you could simply parse the WCSS array to get the index that yielded the best centroids and

get the 2D centroids indexed at the same index the minimum WCSS was found in the WCSS array. This proved to be more difficult than expected and openACC could not consider the process as independent as we had theorized. We attempted the following:

a. Make the outermost for loop (while making everything inside run sequentially just to simplify acc and have it not try to do things we did not want):
    i. #pragma acc loop independent
    ii. #pragma acc parallel loop
    iii. #pragma acc kernels
    iv. #pragma acc parallel loop reduction(min:WCSS) (this was a different idea that we thought we could implement a reduction but it also didn't work out)

2. The next bottle neck of the problem were the distance point calculations represented by the following pseudo code:

Loop that iterates through each sample

Distance = Maximum Value

Loop that iterates through each centroid

temp = l2-distance of current sample to centroid

If the temp < Distance

Distance = temp

Assign in a separate struct that the current sample belongs to the centroid the inner for loop is at

End

End

This calculation is actually very similar to the outermost for loop. What we need to do in order to parallelize the inner loop is to have a pragma that implements a reduction on the distance while keeping track of the index that yielded the minimum distance. However, we again could not figure out how to do this. The outer loop was something we were able to parallelize by making the inner for loop

sequential and the outer loop parallel but it only slowed down the program after a time comparison of how long the loop took before and after the pragmas. In the end we decided to leave this calculation unparalleled.

3. Inside each for loop there are calculations of the new center of mass of centroids by getting all the points that are assigned to a centroid and computing the center of mass of those points. This was a step that worked out really well and we were able to parallelize it. Here is the pseudocode of how it was done:

    Parallelizing the new center of mass of centroids

    #pragma acc parallel loop

    Loop that iterates through samples

        Number = the centroid the current sample is at

        #pragma acc atomic update

        Centroid[number][x] += x coordinate of current sample

        Centroid[number][y] += y coordinate of current sample

    #pragma acc parallel

    Loop that iterates through new centroids

        #pragma acc atomic update

        Number = number of points that a centroid contains

        Centroid = centroid/number

4. The last calculation that we were able to parallelize was the threshold calculation to determine early stopping conditions in the algorithm. Here is the pseudocode for it:

    #pragma acc parallel loop reduction(+:thresh_met_counter)

Loop that iterates through samples

#pragma acc loop reduction(+:thresh_met_counter)

Loop that iterates through the x and y coordinates of the sample

Temp = percent changed of a centroid coordinate compared to its old copy

If temp < threshold

thresh_met_counter++; (thresh_met_counter is what determines the early stopping conditions in the program)

OpenMP:

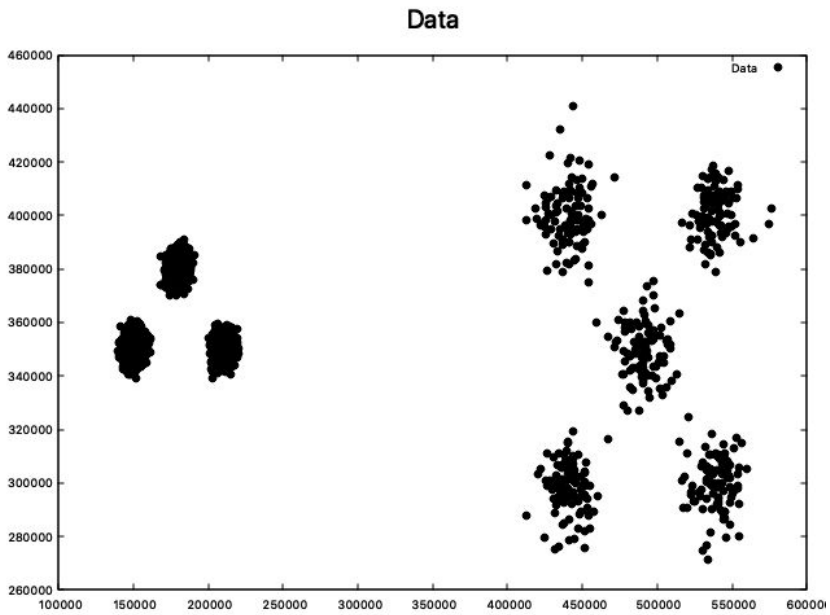
- We decided not to use an mpi. We would have needed to partition the grid into squares or divided the data into the number of threads in some way if we wanted to use an mpi. Either of these solutions don't seem to work well with the problem at hand. First because there are different densities across subsquares inside the grid of points so one of the threads might be doing 90% of the work if you choose to partition a L by L grid, into subgrids. One could partition the data but the problem now becomes having to communicate with every thread in order to calculate new centroids and do the distance point calculations. Overall you are never fully dividing the process and you may always need to communicate with every other thread. This complexity and cost of communication made us choose Open ACC rather than Openmp for tackling this problem.

Randomization Challenges:

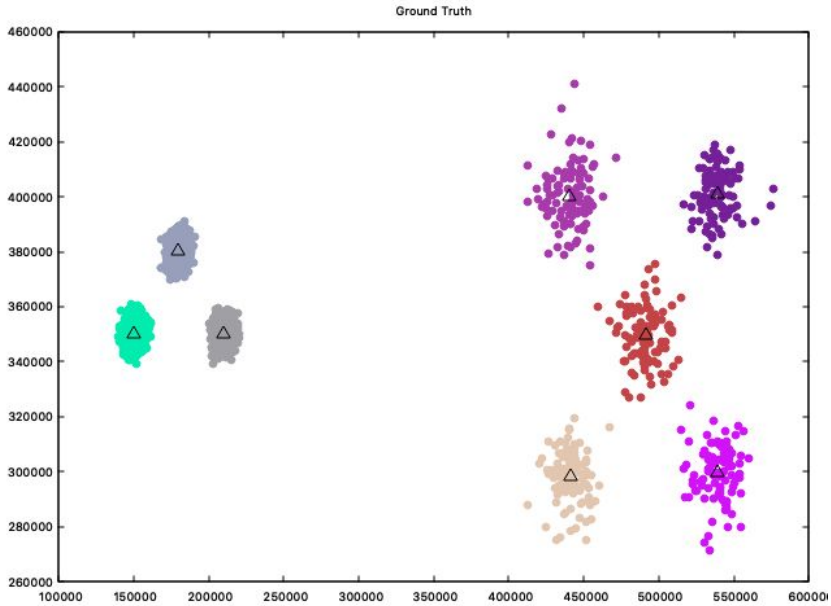
Results:

Number of centroids = 8, Number of Samples = 6500

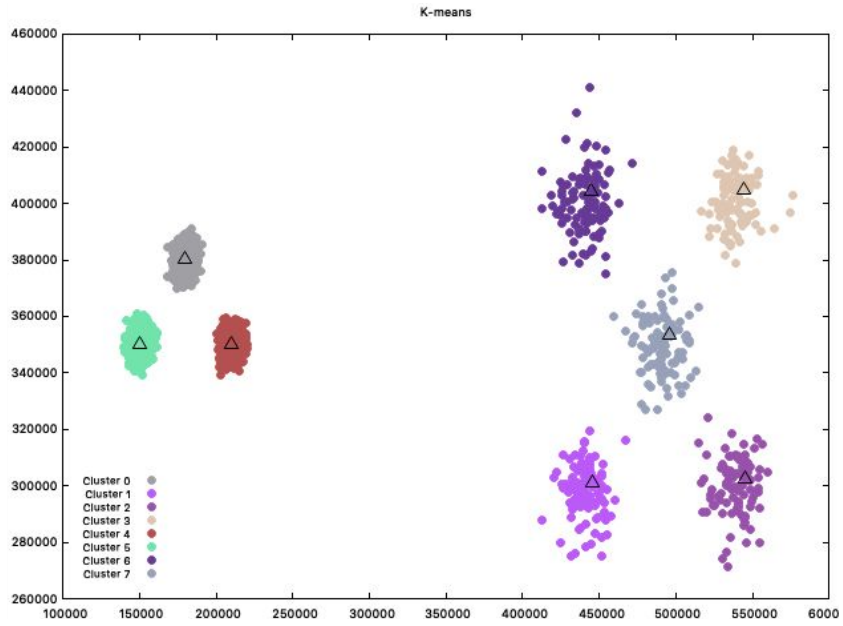
- **Unlabeled Dataset:**



- **Ground Truth Labeling Dataset**

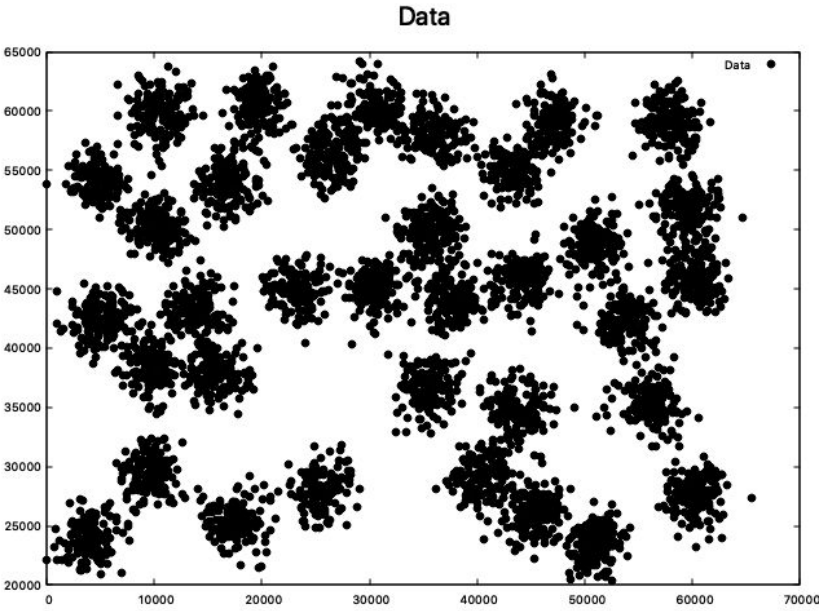


- **K-means in Parallel Labeling:**

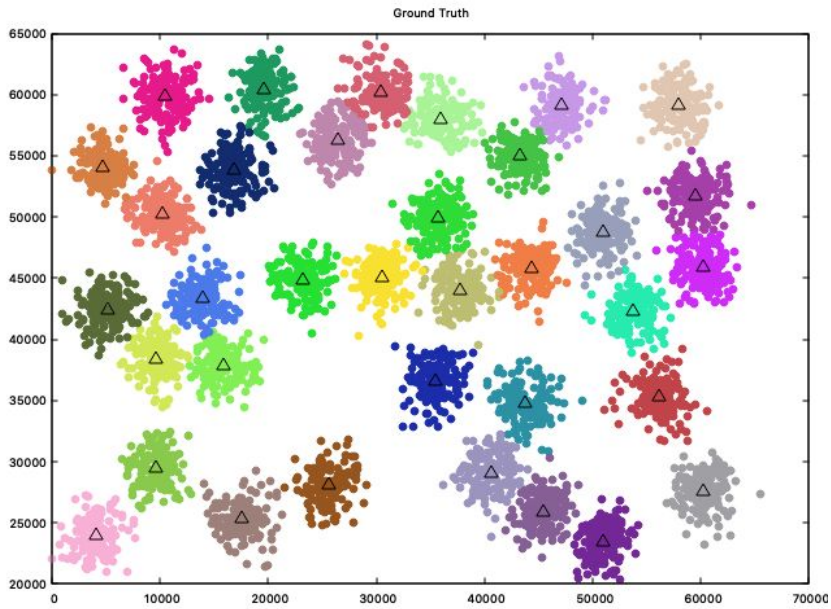


Number of centroids = 35, Number of Samples = 5250

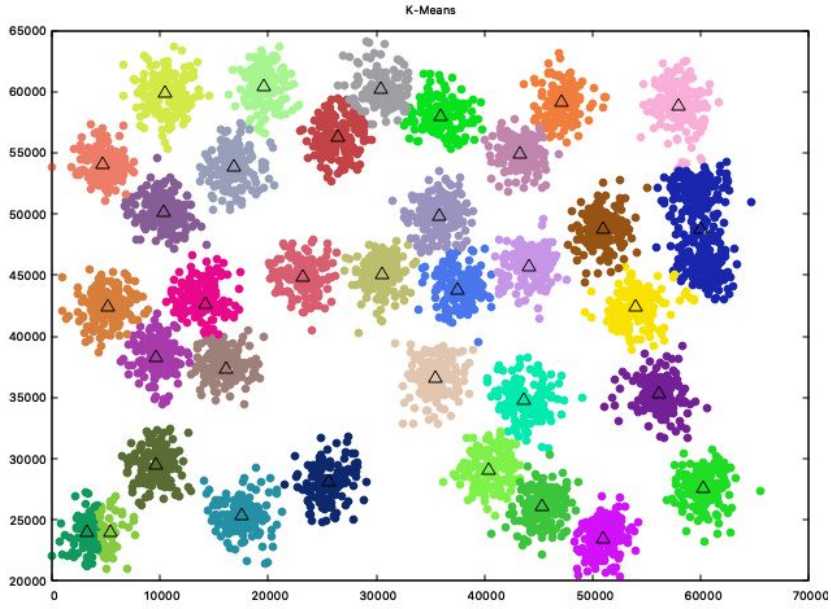
- **Unlabeled Dataset:**



- **Ground Truth Labeling**

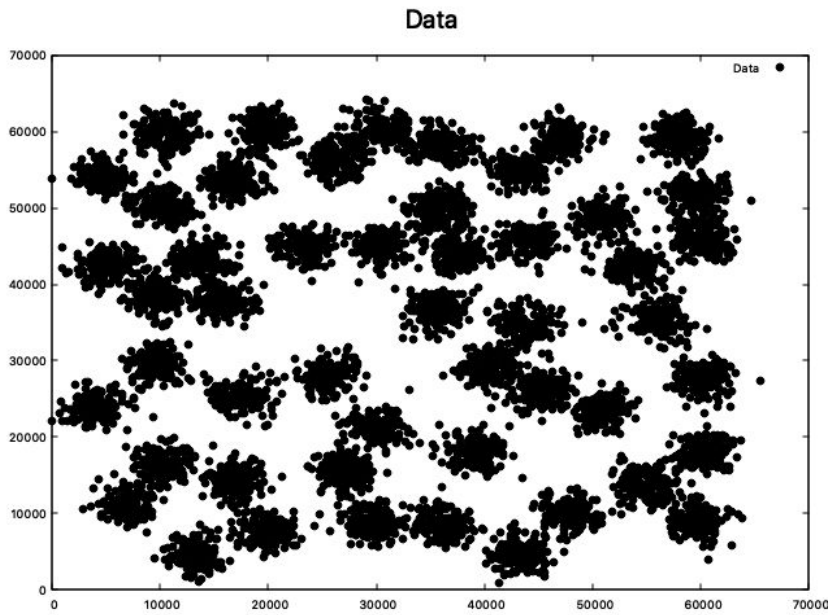


- **K-Means in parallel Labeling:**

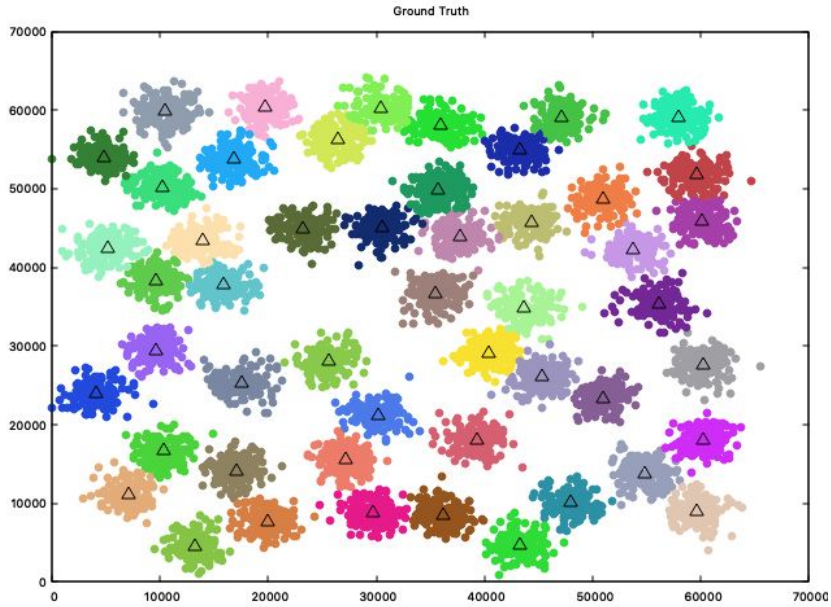


Number of centroids = 50, Number of Samples = 7500

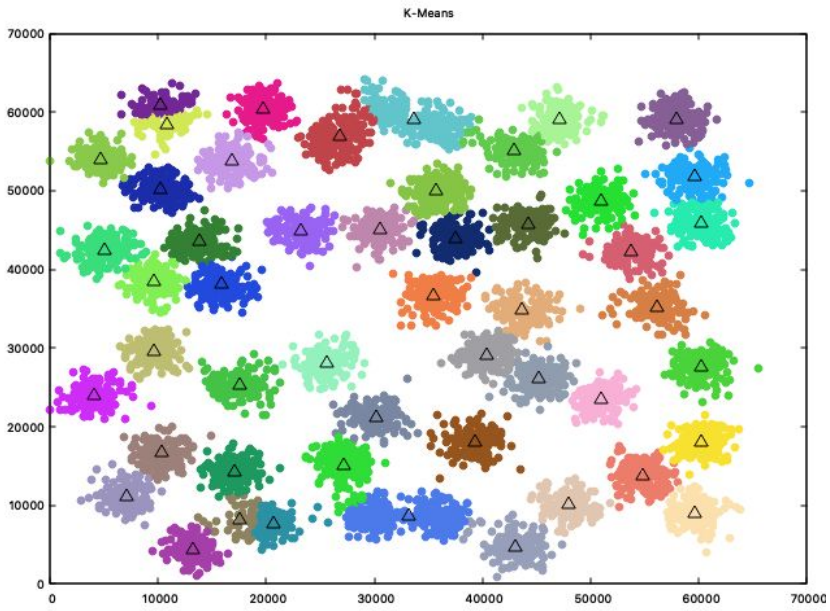
- **Unlabeled Dataset:**



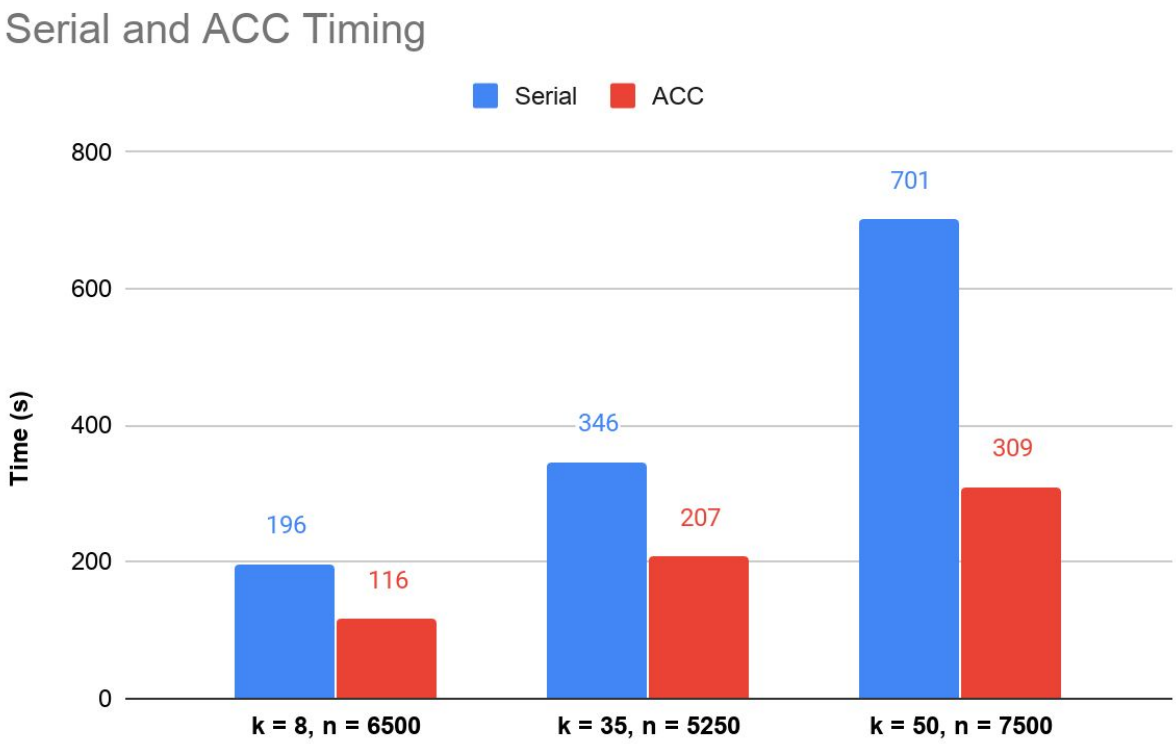
- **Ground Truth Labeling:**



- **Parallel K-Means Labeling:**



Timing Benchmarks:



Work Breakdown

Tu Timmy Hoang: Wrote serial code for K-Means clustering and parallelized portions of the algorithm using OpenACC. Also benchmarked the serial and parallel code to create the plots above.

Andreas Francisco: Attempted to write to parallelize the serial code in open MP and worked on the parallelization of open ACC code as well.