Tal Derei

CSE 375: Assignment 2
K-means

Design Framework:

The basis for redesigning a parallel version of the k-means algorithm involves understanding the issues surrounding the code block inside the if-statement (lines 231-239). **[1]** The API calls to addPoint() and removePoint() are not thread-safe, and require necessary synchronization primitives like locks or atomics. This method involving synchronization primitives like locks performed slower in testing, as the naive approach would be to simply lock each function call to be accessible by only one thread. **[2]** addPoint() and removePoint() are extraneous function calls -- i.e. why do you constantly need to add/remove points from a cluster every iteration. I chose the latter approach, involving fine-tuning the logic for calculating the centroid values since the clusters don't contain points anymore. Rather than associating *clusters with points,* we're now only associating *points with clusters* with 'points[i].setCluster(id_nearest_center)' for each iteration. So at a high-level, I'm logically just associating the points with the cluster (without ever adding/removing points to the cluster), and then just rewriting the logic (lines 243-257 in serial implementation) for calculating the centroids.

In the parallel implementation (lines 198-202), there's a vector 'centroid_counts' representing a counter for the number of points in each cluster, and an N-dimensional matrix called 'cluster_matrix' which holds the summation of each cluster. Rewriting the logic for calculating the centroids involves: **[1]** computing the cluster counts, **[2]** calculating the summation of values for each cluster, **[3]** calculate centroid values. After all the iterations, **[4]** finally adding the final points to the cluster.

In regards to the operations themselves, associating each point to nearest centroid represents a map operation, and recalculating the center of each cluster is a reduction operation. With TBB, I parallelize the for loop that calls the getIDNearestCenter() function with parallel_for. But when I add parallel_reduce on the outer for-loops inside the getIDNearestCenter() function as well, the execution times are slower than if I just do a single parallel_for on the outer for loop that calls getIDNearestCenter(). There seems to be some parallelization overhead. Additionally, in the serial code, the logic involving recalculating the center of each cluster can be parallelized with TBB's parallel_reduce or OpenMP's #pragma omp parallel for reduction(+ : cluster_matrix[points[i].getCluster()][j]). In testing, it seemed like the iterations over the number of points in each cluster might be small enough that there's more thread-overhead/contention than actual speedup.

Additionally, some SIMD OpenMP pragmas were used inside the getIDNearestCenter() function where I thought the compiler could take advantage of the for-loop execution.

Performance Results:

The performance statistics were tested against a cluster size K = 13, and dimensions = 7 for 5K/10K/100K datasets. The cluster sizes were **fine-tuned** to K = 13 according to a custom python script which **[1]** visualized the cluster sizes, **[2]** used the *elbow-method* to determine the optimal number of clusters by minimizing the cluster variance.

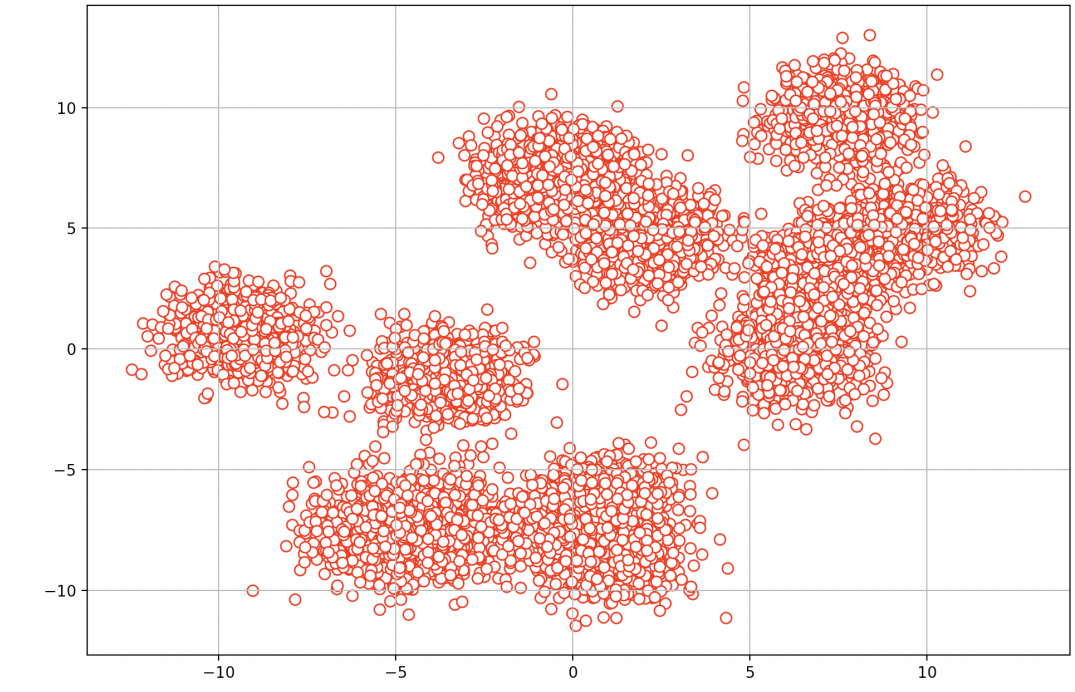

Figure 1: Clusters in 2D-Space

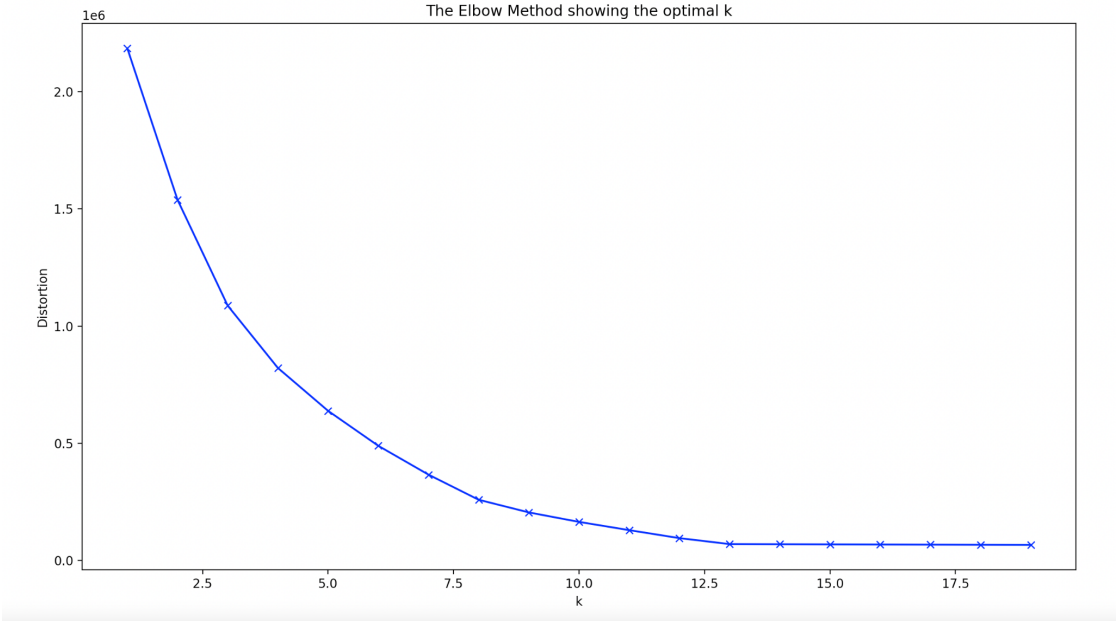

Figure 2: Elbow Method For Calculating the Optimal Cluster Size

The plot below illustrates ~6x speedup for 5K data points, ~8x speedup for 10K data points, and ~14x speedup for 100k data points. This result is interesting because as the size of the data structure increases, the sequential execution times increase exponentially, while the threading overhead decreases. The overhead for smaller data structures results in conservative performance gains.

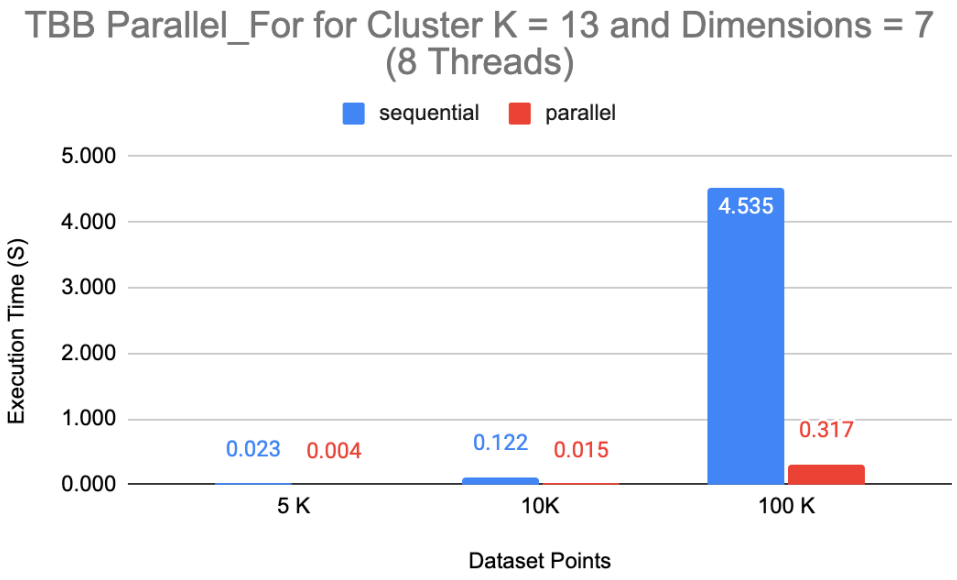

Figure 3: Dataset Points vs. Execution Time for Sequential/Parallel Runtimes

In the plot below, of particular interest was to analyze how the execution times change on the same data structure (100 K points and K = 7) with a varying number of dimensions/attributes.

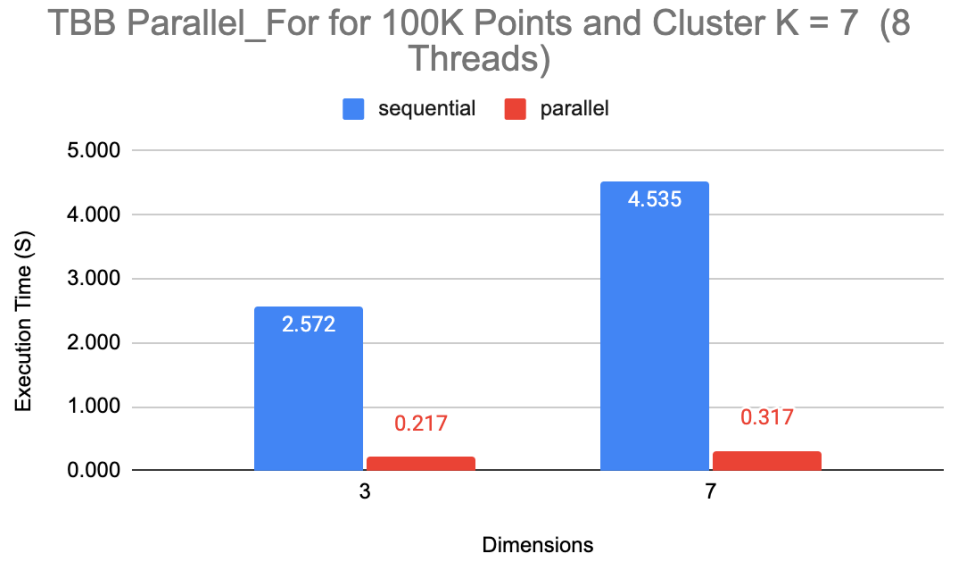

Figure 4: Dimensions vs. Execution Time for Sequential/Parallel Runtimes

(continued) As the number of dimensions increased, the sequential execution increased linearly almost, but the parallel execution only took slightly longer. 3-dimensions achieved an ~12x speedup while 7-dimensions achieved an ~13x speedup. Doubling the number of dimensions only results in slightly increased performance gains on the same data set with the same cluster sizes and points.

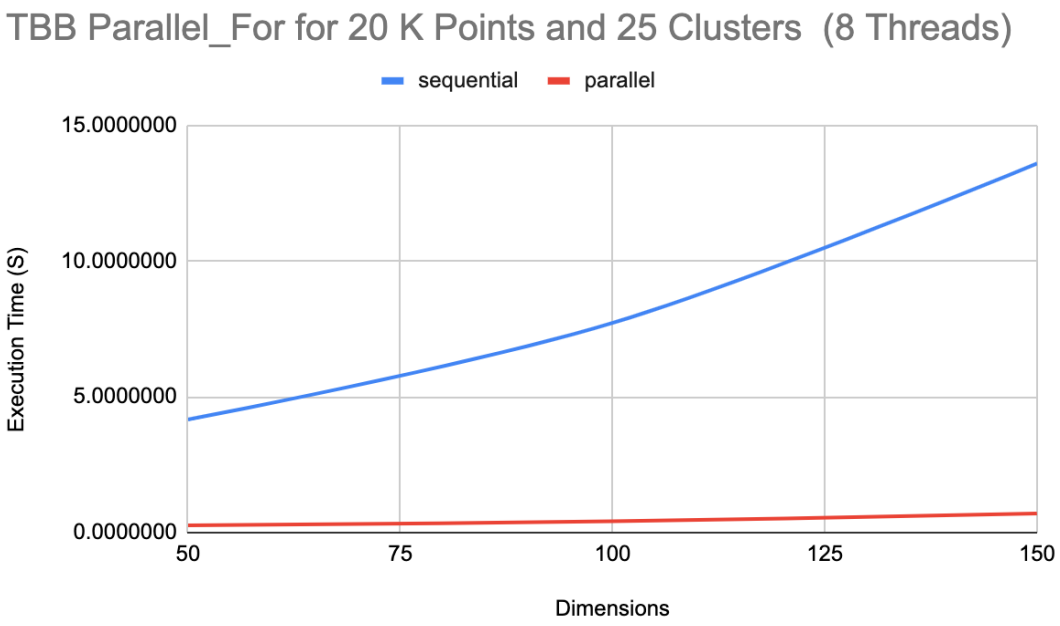

Figure 5: Dimensions vs. Execution Time for Sequential/Parallel Runtimes

In the plot above, I was experimenting with a larger number of dimensions for a larger data set. The results indicate that while the performance degradation is increasing for the sequential execution, the parallel execution is near constant, implying that I haven't reached the saturation point.