SIT315 M1T2

K-means sequential program

My program reads a csv file `sample_large` populated with 100 randomly generated coordinates. It then performs k-means clustering on the points to cluster them to 5 best fitting clusters. The number of cluster `k` and the number of iterations or `epochs` have been set fixed.

- k = 5
- epoch = 100

The programs the k-means clustering 10 times and outputs the average results.

Result of sequential program

Average execution time over 10 runs: 4265 µs

It is to be noted that the sample data (csv file) has been generated by ChatGPT.

Parallelizing the program

Decomposition part

Since only the time for K-means clustering is measured, we will parallelized only the function `kMeansClustering(...)` . This function can be split into smaller sections and we can analyse this sections to keep in sequential and which to parallelize. Below are the sections I have identified.

- **Initializing random centroids**

  Keep in sequential. It has minimal cost since it only runs for a k=5 times and it is simple loop. Parallelizing would add unnecessary overheads.

- **Declaring and initialization of variables and pointers**

  Keep in sequential. One-off steps. The cost is negligible. Better just to keep them before the parallel region.

- **main loop/epochs**

Parallelized in inner loops. Epochs is set to 100. Although it is a large number of iterations, each iteration depend on the previous one. Therefore we cannot run the iterations in parallel. That being said we will start the parallel region here with these shared variables. `shared(points, centroids, nPointsA, sumXA, sumYA, k, n)`

- **Assigning centroids to points**

  Parallelized. Each point compute their centroids independently of other points. This section is said to be embarrassingly parallel. Since there is a large number of points, we can split group of points between threads to run in parallel. Here we will use static scheduling.

  The inner loop of choosing which of the 5 centroid to be for one point can be left in sequential since it has minimal cost.

- **Computing new centroids among clusters**

  Parallelized. Each cluster computes their new centre coordinates independently of other clusters. This task is embarrassingly parallel and can be parallelized over clusters. However, since the many threads may write to `SumXA` , `SumYA` and `nPoints` . We have to use reduction here to prevent race conditions. Hence, each thread is assigned a private copy of the above variables and the sum of each copy will be written to the original variable afterwards.

We will use **OpenMP** library to parallelize the program since it is simpler to write than pthreads.

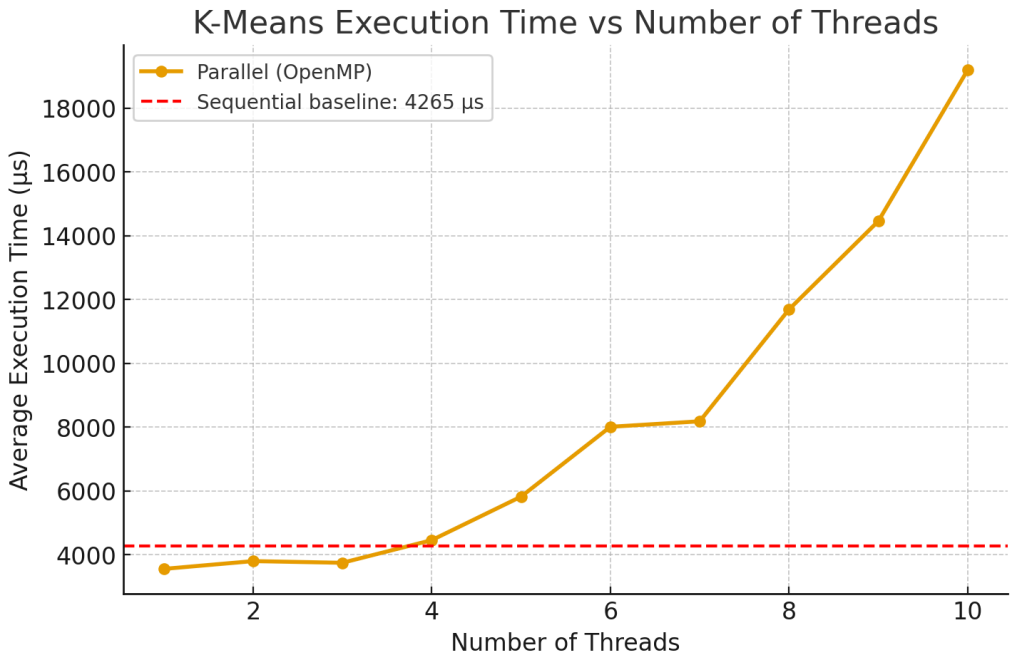
Evaluating the result of parallel program

I have experimented with different number of threads (1- 10 since my system has 10 cores) and here are the results.

- The average execution time for **1 thread** is **3551 µs**
- The average execution time for **2 threads** is **3795 µs**
- The average execution time for **3 threads** is **3743 µs**
- The average execution time for **4 threads** is **4452 µs**
- The average execution time for **5 threads** is **5816 µs**
- The average execution time for **6 threads** is **8006 µs**

- The average execution time for **7 threads** is **8178 µs**
- The average execution time for **8 threads** is **11686 µs**
- The average execution time for **9 threads** is **14454 µs**
- The average execution time for **10 threads** is **19192 µs**

Graph of K-Means Execution Time vs Number of Threads



Graph generated by ChatGPT

Conclusion

We can see from the graph, that the omp program is only slightly faster than the sequential program for small number of threads (1-3 threads). As from 4 threads, increasing the number of threads, increases the overheads significantly, hence making the performance worse.

This behaviour can be theoretically explained by the small problem size (n = 100 points): The dataset is too small to benefit from heavy parallelism. The overhead of managing threads outweighs the work each thread performs. To see meaningful improvements, a much larger dataset (e.g., thousands or millions of points) would be required so that computation dominates over parallel overhead.