Suhas Kamath

DS 221: INTRODUCTION TO SCALABLE SYSTEMS

November 7, 2025

Parallel Programming - OpenMP Assignment

THEORY & METHODOLOGY

K-means clustering is an iterative unsupervised learning algorithm that segregates a set of data points into K clusters. [1] [2] The diagram below shows how the algorithm iterates:

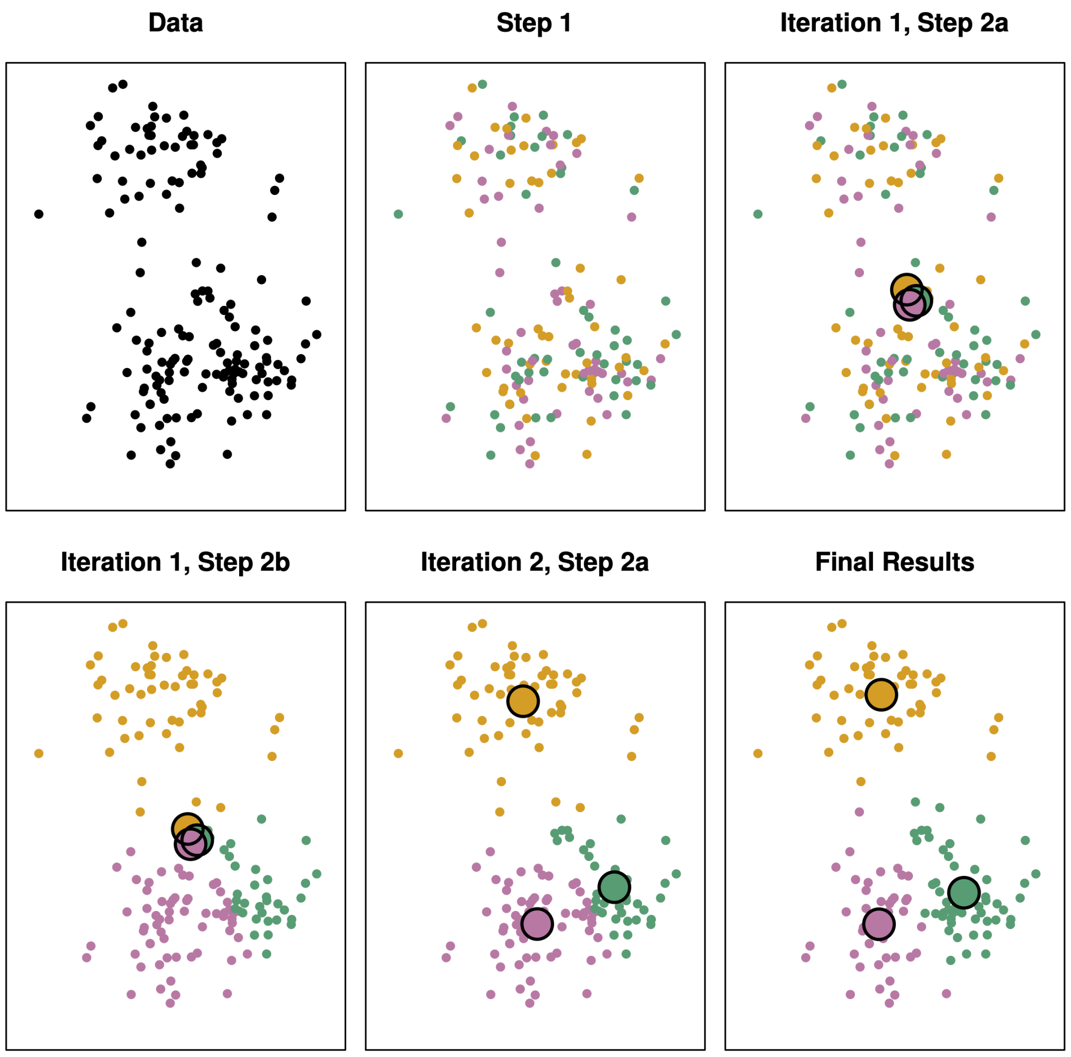

FIGURE 1. K-means iterative clustering algorithm. [3]

The basic (sequential) algorithm for K-means clustering is as follows:

1) **Initialization:**
   - Choose the number of clusters, K. We use $K = 20$, as specified in the question.
   - Initialize the K cluster centroids, $C_1, C_2, \ldots, C_K$. We simply use the first K data points as the centroids for simplicity.[1]

[1]We can also use `kmeans++` algorithm, but such an algorithm will only increase the complexity. Our aim is check the time saved by using OpenMP, not obtain a perfect clustering algorithm. [4]

2) **Assignment Step (Cluster Assignment):**
   - For each data point P_i in the dataset:
   - Calculate its distance to each of the K centroids. We use Squared Euclidean distance:

$$d(P_i, C_j) = \|P_i - C_j\|^2$$

   - Assign the data point P_i to the cluster of its nearest centroid.
3) **Update Step (Centroid Recalculation):**
   - For each cluster j (from $j = 1$ to K):
   - Recalculate the position of its centroid C_j to be the arithmetic mean of all data points P_i currently assigned to it.
4) **Check for Convergence:**
   - After the update step, we check if the algorithm has converged. Convergence is reached if the cluster assignments (the sets S_j) did not change from the previous iteration.[2]
   - If not converged, return to Step 3 (Assignment Step).

We can parallelise a few aspects of this algorithm:

1) The assignment step, wherein we assign each point to its nearest centroid, can be fully parallelised.
2) The accumulation step, where the per-thread local sums are merged into global, can be parallelised.
3) To check for convergence, we calculate the distance of each new centroid from its previous position. This can also be parallelised using OpenMP's `reduce` feature.

Experimental Setup

Both the sequential and parallel programs were executed on the teaching cluster. The teaching cluster has the following specifications:

- **CPU:** Intel Xeon Gold 5318Y 24-core 2.1GHz
- **RAM:** 128 GB RAM
- **Storage (HDF5):** 2×12 TB SATA HDD, 1.92TB PCIe NVME M.2 Enterprise SSD
- **Operating System:** Cent OS
- **Job scheduling software:** SLURM

The program was first executed sequentially. After this, the program was executed parallelly using OpenMP. For testing the benefits of OpenMP $4, 8, 16, 32$ threads were used. To obtain more accurate readings, each experiment was repeated 5 times, and the execution time recorded.

The execution times (of each experiment) is then fed into Python, where the average corresponding to each thread count is calculated and output. The Python program also produces a graph using `matplotlib`.

[2] To accommodate for floating point errors, we assume convergence when the change in centroid positions between iterations is below a small threshold, ϵ

Results

As mentioned above, we calculated the average time over 5 experiments per number of threads. These averages are displayed in the below table:

Table 1. Table containing the average execution time against number of threads.

Threads	**Average Time (ms)**
1	27.713
4	8.922
8	5.710
16	4.667
32	3.691

The graph obtained by plotting the averages is as follows:

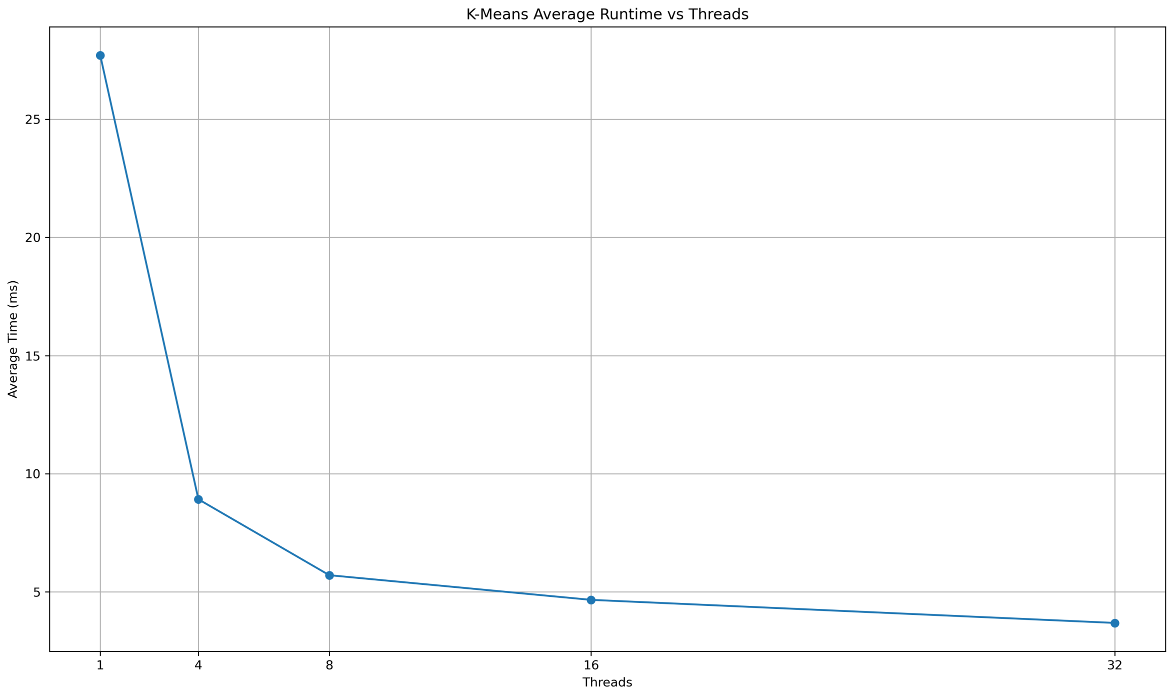

Figure 2. The graph depicting the average execution time against the number of threads.

Observations

As can be seen in the graph, when we go from a sequential program to a parallel program with 4 threads, it takes just 32% of the time of the sequential program. This correlates to a speed-up of around 3.11.

However, as we increase the number of threads, the speed-up reduces. For example, we we go from 4 threads to 8, the speedup is just 1.56 (which is about half of the speed-up we obtained when going from a sequential program to 4 threads).

This is essentially the law of diminishing returns, wherein a higher ”investment” need not guarantee the same ”return on investment”. In parallel programming, this is called as Amdahl’s law. Amdahl’s law can be stated as follows:

“The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used.” [5]

Amdahl’s law can be visualised using the following graph:

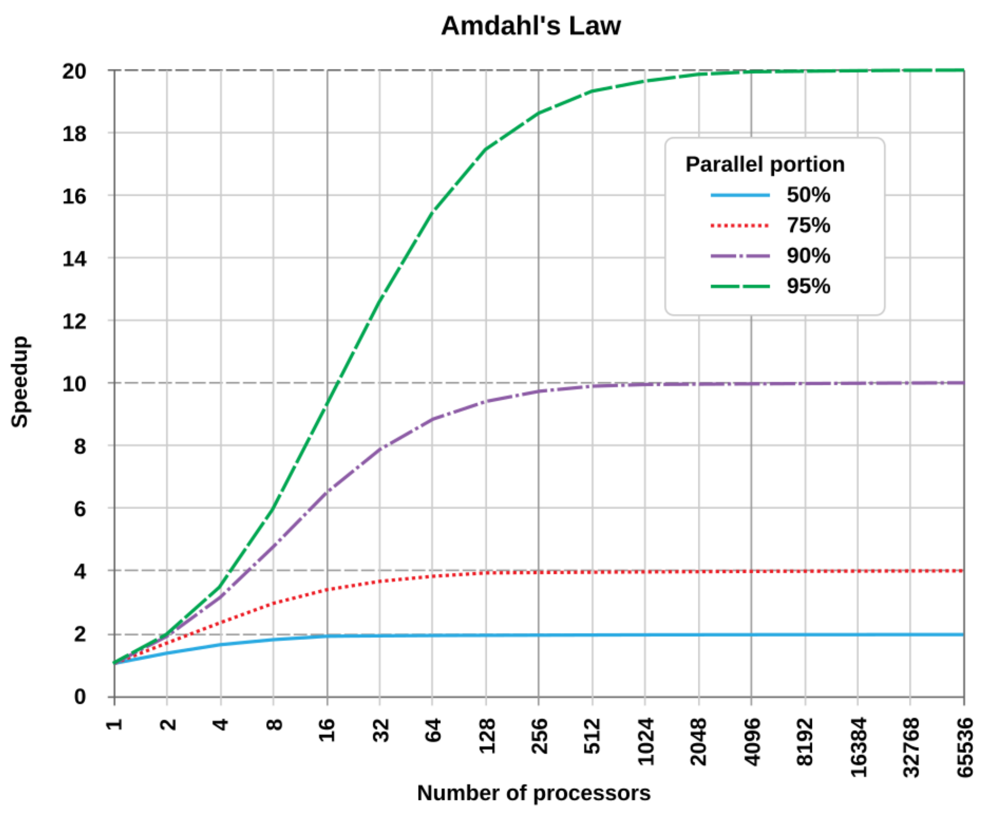

Figure 3. Visualisation of Amdahl’s law. [6]

As can be seen in the graph, after a particular threshold, adding more processors (or threads) does not make any difference in the speedup. What we have observed in our experiment corroborates Amdahl’s law.

References

[1] E. Kavlakoglu and V. Winland, "What is k-means clustering?,"

[2] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Berkeley Symposium on Mathematics Statistics and Probability, 1967*, pp. 281–297, 1967.

[3] S. Bacallado and J. Taylor, "Stats202: K-means clustering," 2022.

[4] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '07)*, pp. 1027–1035, 2007.

[5] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[6] "Svg graph illustrating amdahl's law." Wikipedia, 2008.