K-Means

Filippo Taiti
filippo.taiti@edu.unifi.it

February 6, 2026

Abstract

K-Means is an unsupervised learning algorithm based on the concept of clusters (groups of elements). The goal of the algorithm is to partition the data into K distinct groups based on their similarity, so that the elements associated with a given cluster are close to each other, while the K groups must be distant. It is essential to define the concept of a cluster's centroid, i.e., its coordinates, defined as the average of the coordinates of all points belonging to that cluster.
In the project under consideration, K = 50 and a number of iterations equal to 30 have been set. Strong scaling and weak scaling tests will be carried out, and in the case of the parallel version, the three thread scheduling modes will be tested: static, guided, and dynamic.
As will be shown later, the best average speed-up achieved was 7.6x.

1 Introduction

The main objective of K-means is to group a set of data into K distinct clusters, so that the elements within each cluster are as similar as possible to each other, while the clusters are as different as possible from each other. To achieve this result, K-Means seeks to minimize the distance between the points and their respective clusters. In particular, it seeks to find the optimal coordinates of the central points of each group, called centroids. Each point belongs to one and only one cluster. Therefore, there is a continuous update regarding the cluster closest to each point and the coordinates of the centroids. This is the most expensive part of the algorithm.

2 Algorithm Description

The algorithm follows the following iterative process:

1) Selection of initial centroids: given a fixed number K and a certain set of data, k points are randomly selected from the set. These are the K initial centroids. At this stage, it is important to ensure that the K centroids selected are distant from each other, so it is important to have an algorithm/procedure that ensures this. One of the best-known algorithms to use at this stage is K-Means++, which seeks to ensure a significant distance between the K centroids. In particular, in the case under consideration, this algorithm was used for the initial selection.

2) Assignment: each element of the dataset is assigned to the nearest centroid. To define the concept of "distance," a metric must be defined, which, in the most common and well-known

case, as well as the one under consideration, is the Euclidean distance.

3) Update: For each cluster, the position of the centroid is recalculated by averaging the coordinates of the points associated with it.

4) The second and third steps are repeated until the coordinates of the centroids stop changing (convergence) or for a maximum number of iterations (the case under consideration).

The sections that can be parallelized are determining the closest centroid to each element in the dataset and updating the centroid coordinates.

3 Parallel Implementation

Parallelization was performed as follows: A parallel region was defined using the #pragma omp parallel directive with default(none) in order to control variable visibility. Two for loops were defined within the parallel region using the #pragma omp for directive. The first is used to determine the centroids belonging to each element of the dataset and to determine how to update the coordinates of the centroids in the next for loop. In fact, there is another nested loop, which is not parallelized. In this first for loop, a reduction was used through the #pragma omp reduction directive. The second and last parallel for loop is used to update the coordinates of the centroids.

The dataset was represented using SoA.

4 Experimental Setup

The code was written in CLion, while testing was performed on the terminal using the *perf* tool on Linux to analyze elements such as the program's impact on the cache and the number of instructions per cycle. Ubuntu 24.04 was used as the operating system, clang++ was used as the compiler on CLion, and gcc was used on the terminal. The flags set in the CMakeLists.txt file are as follows: -O3, -march=native, -flto, -funroll-loops.

The dataset used consists of 100,000 points, generated through the script on the Moodle page of the course, in the PDF file describing K-Means. This dataset is contained in a .csv file. Although each individual element generated can be considered a vector with 30 elements, only the first 2 elements were taken in order to be able to show the centroids (initial and final) graphically. The tests will be carried out considering K=50 and 30 iterations (actually 32, but the first 2 will not be used to determine the time statistics). The *chrono* library was used to measure the wall clock time, while the *time* library was used to measure the CPU time. The graphical representation was made possible by the *matplotlib* library.

The CPU used is an AMD Ryzen 7 9700x. It has 8 cores (16 threads), 32 MB of L3 cache, 8 MB of L2 cache, and 640 KB of L1 cache. The RAM used has a capacity of 48 GB DDR5.

5 Result and Analysis

After ensuring that the sequential version of the algorithm was implemented correctly, the final coordinates of the centroids of the two versions were compared to determine the correctness of the parallel version. Therefore, if they coincide, the correctness is confirmed.

5.1 Test

Weak scaling tests were performed by varying the thread scheduling mode (static, dynamic, guided) and chunk size (64, 512, 1024), both with and without significant compiler optimizations. In this section, for each chunk size parameter value, two graphs will be shown: the first showing the speed-up (wall-clock time), the second showing the CPU time. In both cases, the representation is based on the ratio between sequential execution time and parallel execution time. Therefore, the expected results are an increase in speedup and a decrease in CPU time.
The time results will be reported in milliseconds.
In addition, a weak scaling test was also performed using schedule(guided, 256), for which an explanatory graph will be shown for both speed-up and CPU time.

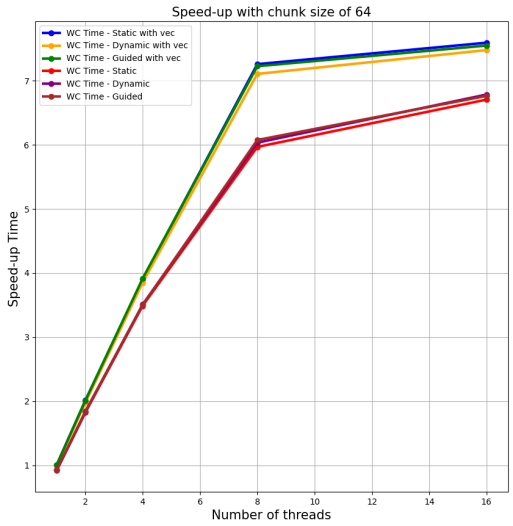

Figure 1: Speed-up with chunk size of 64.

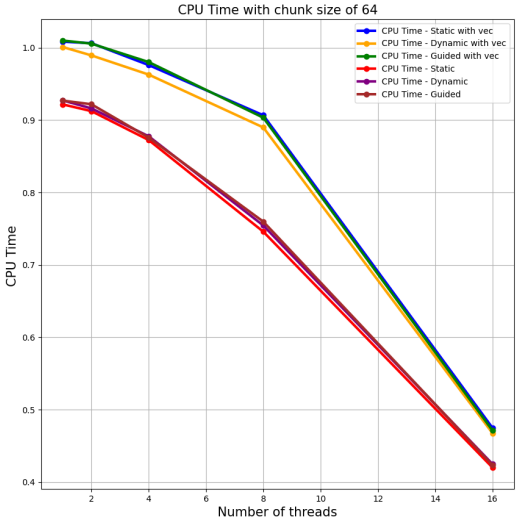

Figure 2: CPU Time with chunk size of 64.

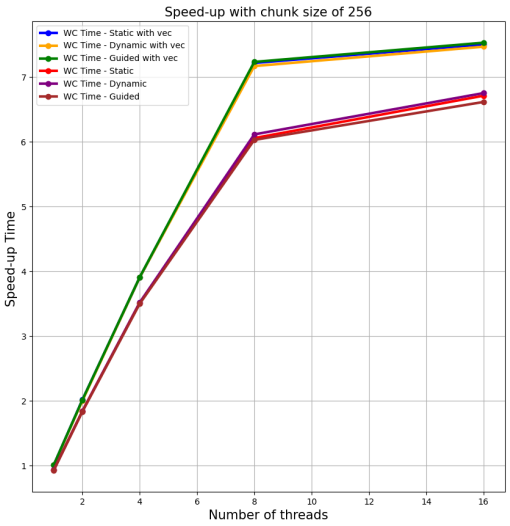

Figure 3: Speed-up with chunk size of 256.

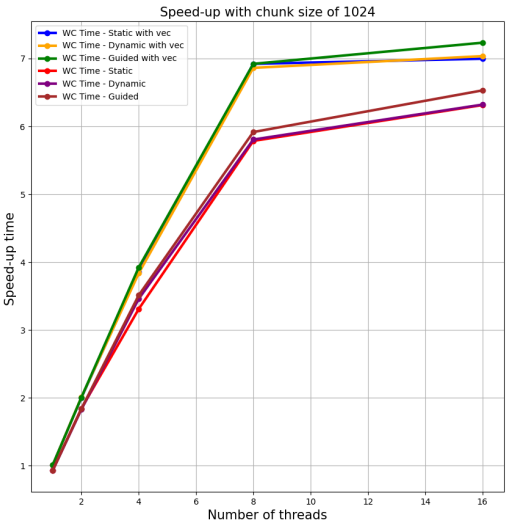

Figure 5: Speed-up with chunk size of 1024.

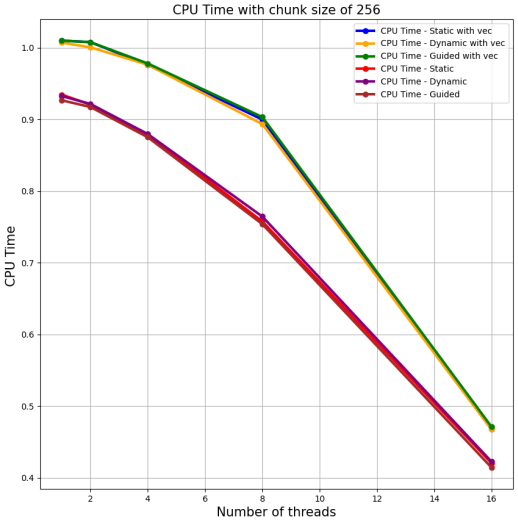

Figure 4: CPU Time with chunk size of 256.

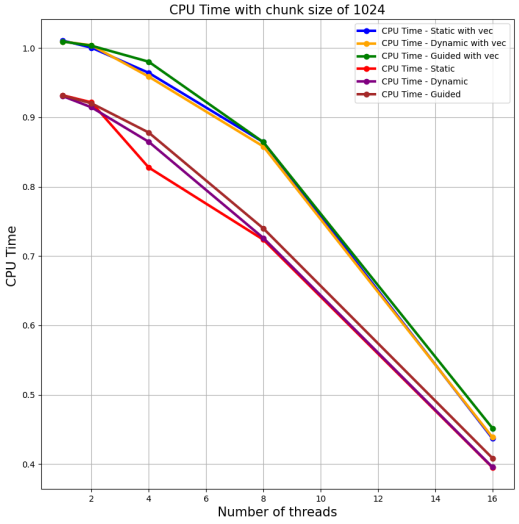

Figure 6: CPU-Time with chunk size of 1024.

The results observed coincide with those expected: it can be seen that a smaller chunk size leads to greater speed-up.
Furthermore, it should be noted that, regardless of the chunk size, the performance of dynamic scheduling is lower than that of the other two types: this is in line with what is expected from K-Means, as the workload is balanced.
Furthermore, looking at the CPU time graphs, it can be seen that, proportionally, if vectorization and all the necessary optimizations are not used, there is a more rapid decrease in the ratio between CPU time in the sequential case and CPU time in the parallel version. This is due to the fact that without optimizations, the number of instructions to be executed to achieve the same result is greater.
Another thing to note is how the speed-up increases: as long as the threads do not have to share computing units with each other, the improvement is perfectly linear, then it becomes less steep. This is in line with expectations. In conclusion, the graphs show that the best speed-up achieved (7.6x) is obtained using the necessary optimizations and static thread scheduling with a chunk size of 64.
This last graph shows that if the dataset is increased proportionally to the number of threads, there is a perfectly linear improvement up to 8 threads, then after that the speed-up increases less steeply, but this is normal because different threads have to share the same computing units.

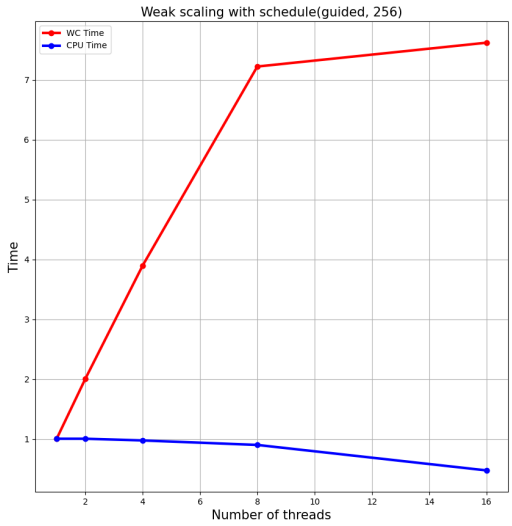

Figure 7: Weak scaling with schedule(guided, 256).

	Min	Max	Mean	Std
Seq vec	64.32	64.40	64.31	0.05
Seq	806.14	817.24	810.09	3.48
Best	8.34	8.56	8.47	0.05

Table 1: WC Time sequential version and best case of parallel version (vec) (speed-up 7.6x).

The table shows a significant difference in standard deviation between the sequential version with optimizations and without. This is due to the fact that without optimizations, the program runs for longer and is therefore subject to system interrupts, for example, and generally executes more instructions. This affects stability.

6 Limitations

Ideally, we would expect that if N threads are used, there will be an N speed-up factor, but

in reality this is not the case for several reasons: the first is that not everything can be parallelized; in fact, the individual iterations of the algorithm are not parallelized; the second is that, as mentioned above, when the number of threads used exceeds the number of physical cores, the speed-up improvement is no longer perfectly linear. As for the scalability of the problem itself, several factors must be considered, such as the size of the dataset, the number of centroids (clusters), and the dimensionality of the individual elements of the dataset. Scalability deteriorates linearly with respect to these factors.
The bottlenecks are therefore the size of the RAM, the cache, and the number of CPU cores.

7 Future work suggestion

To solve the algorithm's scalability problem, it would be interesting to implement a version of the algorithm based on mini-batches rather than the entire data set. This would allow the use of data sets that do not fit entirely into RAM.