K-Means Clustering with Open-MP

Davide Del Bimbo
davide.delbimbo@edu.unifi.it

Abstract

K-Means is a widely used clustering algorithm for data segmentation and analysis. This report introduces a parallelized implementation of the K-Means algorithm, developed in C++ and utilizing OpenMP. The application is designed to supports the clustering of multidimensional data, whether sourced from input datasets or generated randomly. It shows the advantages of parallel execution by optimizing hardware utilization through the use of certain tricks, such as SoA (Structure of Arrays) architecture for representing points and centroids.
This report conducts a comprehensive performance analysis, including speed-up and structure comparison on an Intel i7-10750H processor with 6 cores. The study includes a detailed comparison between the sequential and parallel versions of the K-Means algorithm, showing the efficacy and benefits of parallelization.

1. Introduction

This report provides an examination of the K-Means algorithm, offering insights into its theoretical foundation and implementation principles. Subsequently, it introduces a parallelization strategy with OpenMP, a parallel programming framework designed for multiprocessing on shared memory, to minimize the overall execution time and enable faster and more efficient clustering of large multidimensional datasets.

2. K-Means

The K-Means algorithm is a well-known unsupervised learning technique used in machine learning and data analysis to group data points into K distinct groups or **clusters**, in order to identify hidden patterns in the data and partition them into homogeneous groups.
The main goal of K-Means is to assign each data point to one of the clusters such that data points within the same cluster are most similar to each other, while data points belonging to different clusters are most dissimilar to each other [1].

Mathematically, the K-Means algorithm divides a dataset $\mathcal{X} = \{x_1, .., x_N\}$ of N samples into K disjoint clusters $\mathcal{C} = \{\mathcal{C}_1, ..., \mathcal{C}_K\}$, where each cluster $\mathcal{C}_j$ is described by the mean μ_j of the samples belonging to it. These means are commonly called **centroids** of clusters.
Therefore, the goal of K-Means is to minimize the within-cluster sum-of-squares criterion, called **inertia**. Formally, given a dataset of observations $\mathcal{X} = \{x_1, \ldots, x_N\}$, the goal is to find the centroids μ_j that minimize the sum-of-squares of the distances between each data point x_i and the centroid μ_j of the cluster to which it is assigned:

$$\sum_{i=0}^{N} \min_{\mu_j \in \mathcal{C}} \|x_i - \mu_j\|^2$$

where the centroid μ_j of the cluster $\mathcal{C}_j$ is defined as the mean of the data points belonging to the cluster [2]. That is:

$$\mu_j = \frac{1}{|\mathcal{C}_j|} \sum_{x \in \mathcal{C}_j} x$$

2.1. K-Means algorithm

The implementation of the K-Means algorithm can be summarized in four steps and showed in figure 1:

1 **Initialization**: given a dataset $X = \{x_1, ..., x_N\}$ of N multidimensional samples, choose K random initial cluster centroids from the dataset. These centroids will be the representatives of the clusters.

2 **Assignments**: assign each data point to the centroid of the nearest cluster based on a distance metric, usually Euclidean distance.

3 **Updates**: update cluster centroids by calculating the mean of all data points assigned to each cluster.

4 **Repeat**: repeat steps 2 and 3 until convergence is met. Convergence occurs when cluster centroid updates no longer change significantly or when a specified maximum number of iterations is reached.

In 1 is shown the pseudocode of the K-Means algorithm [3].

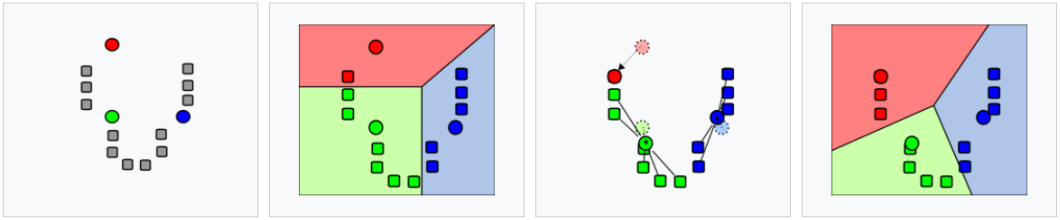

Figure 1. K-Means steps.

Running i iterations of the K-Means algorithm requires a computational complexity of $O(N \times D \times K \times i)$, for N D-dimensional points to be partitioned into K clusters.

The K-Means algorithm converges fast, but doesn't guarantee convergence to the global optimum. Convergence can be evaluated by various criteria, including no centroid shifts, no changes in point assignment or reaching the minimum sum of distances.

In this paper, we adopted the criterion that a centroid isn't considered shifted if the difference between its position before and after the update is less than $\varepsilon = 10^{-6}$. However, since these conditions may not be met, an additional termination criterion was introduced based on the maximum number of iterations allowed.

Algorithm 1 K-Means.

Input:

- Dataset $\mathcal{X} = \{x_1, ..., x_N\}$ of samples to be clustered.
- K number of desired clusters.

Output:

- Set $\mathcal{C}$ of K clusters.

1: Assign initial values for $\mu_1, ..., \mu_K$.
2: **repeat**
3: Assign each sample x_i to the closest cluster.
4: Update the mean for each cluster.
5: **until** convergence criteria is met.

3. Sequential implementation

The sequential implementation of the K-Means algorithm in C++ is shown in 1.

To achieve this goal, we developed a `Point` class that represents a multidimensional data point. Each point is characterized by a vector of double representing its coordinates in space, a unique identifier and a label identifying the cluster to which it has been assigned.

Similarly, we developed a `Centroid` class that represents a multidimensional centroid of a cluster. Each centroid is characterized by a vector of double representing its coordinates in space and a unique identifier of the cluster it represents.

The `KMeans` class is designed to execute the K-Means algorithm sequentially. It offers various methods and parameters to configure the execution. In particular, points can be initialized in two different ways: generating N random points from a uniform distribution (specifying the number of points N and the dimensions D) or importing a dataset from an external file (specifying the path to the file containing the dataset). Instead, centroids are initialized picking the coordinates of K distinct random points from the dataset (by specifying the desired number of K clusters). The method `KMeansIteration` represents a sequential iteration of the algorithm. This method is divided into two `for` loops. In the first loop, we calculate the distance between each point $i = 1, .., N$ and all clusters, assigning the point to the nearest cluster. Then, we store the coordinates of the point and increase the cluster size to allow the next update. In the second loop, we update the coordinates of each cluster based on the assignments.

The convergence criterion is satisfied when the coordinates of all centroids don't change within a certain error threshold, e.g. $\varepsilon = 10^{-6}$.

Listing 1 K-Means sequential iteration.

```
bool KMeansIteration() {
    // Variables to store cluster informations.
    vector<vector<double>> sum(K,
    ↪  vector<double>(dimensions, 0)).
    vector<int> size(K, 0);

    // Convergence flag (true at the beginning).
    bool converged = true;
    for(int i=0; i<N; i++) {
        double minDist = DBL_MAX;
        int minClusterId = -1;

        for(int j=0; j<K; j++) {
            double dist = distance(points[i],
            ↪  centroids[j]);

            if(dist < minDist) {
                minDist = dist;
                minClusterId = j;
            }
        }

        // Assign point to closest cluster.
        points[i].clusterId = minClusterId;
```

```
        // Store clusters informations.
        for(int dim=0; dim<dimensions; dim++) {
            sum[minClusterId][dim] +=
            ↪ points[i].coordinates[dim];
        }
        size[minClusterId]++;
    }

    for(int j=0; j<K; j++){
        double tmpCoordinate = 0;

        for(int dim=0; dim<dimensions; dim++) {
            // Store previous informations.
            tmpCoordinate =
            ↪ centroids[j].coordinates[dim];

            // Update centroid.
            centroids[j].coordinates[dim] =
            ↪ sum[j][dim] / size[j];

            // Check for convergence.
            if (fabs(tmpCoordinate -
            ↪ centroids[j].coordinates[dim]) >
            ↪ EPSILON)
                converged = false;
        }
    }
    return converged;
}
```

The `distance` method shown in 2 allows us to calculate the Euclidean distance between a `Point` and a `Centroid` in a multidimensional space.

Listing 2 K-Means sequential distance.

```
const double distance(const Point &p, const
↪ Centroid &c) {
    double sum = 0;
    for(int dim=0; dim<dimensions; dim++) {
        sum += (c.coordinates[dim] -
        ↪ p.coordinates[dim]) *
        ↪ (c.coordinates[dim] -
        ↪ p.coordinates[dim]);
    }
    return sqrt(sum);
}
```

The `KMeans` class also provides a `run` method to execute the entire algorithm. Specifically, this method creates an instance of the points and centroids from specified inputs. Then, it runs sequential iterations until the convergence condition is reached or until a specified maximum number of iterations is reached. In addition, if necessary, it can generate a plot of the results for each iteration in order to create a GIF animation documenting the evolution of the execution. Finally, it calculates the total running time of the algorithm and stores useful information in a text file for reference purposes.

4. Parallel implementation

In order to optimize the K-Means algorithm shown in 3, were implemented different levels of parallelism. First, was implemented a low-level of parallelism exploiting the hardware features, by transforming the AoS (Array of Structures) architecture to SoA (Structure of Arrays). This is because the SoA architecture turns out to be more cache-friendly. Furthermore, since the points are in a multidimensional space, a linearized solution was adopted to represent the matrices of these structures.

Therefore, we created a `Points` class, which represents the structure of N points in D dimensions. It consists of an array of linearized coordinates, where each value represents the coordinate of a specific point. Specifically, the d-th coordinate of the i-th point is stored at position $i + N * d$ of the array.

Similarly, we created a `Centroids` class, which represents the structure of K centroids in D dimensions. It consists of an array of linearized coordinates, where each value represents the coordinate of a specific centroid. The d-th coordinate of the j-th centroid is stored at position $j + K * d$ of the array.

Regarding high-level parallelism, we used OpenMP to parallelize the code and allow simultaneous execution on multiple threads. We observe that K-Means falls into the category of "embarrassingly parallel" problems. In fact, each point can be assigned to the nearest centroid independently of the others. The most expensive cost of sequential implementation is associated with the first loop responsible for iterating over each point and assigning it to a cluster. This operation involves a computational cost of $O(N \times K \times D)$ at each iteration i. In fact, it require to compute the distance from each point to all centroids and assign it to the nearest cluster. This operation is particularly expensive when N becomes very large. Thus, the main goal of the parallel implementation concerns reducing the time required by the algorithm to execute the first loop, distributing the workload over multiple processing units.
For this purpose, we applied a `parallel for` directive to the first outer loop, adopting a `static` policy to evenly divide dataset processing among the threads. In addition, we specified the `default(none)` clause to explicitly declare variables outside the loop as `shared` or `private`.
Once a point has been assigned to the nearest cluster, the coordinate and cluster size update operations must be performed safely in order to avoid "race conditions". In fact, these variables are shared among all threads. Therefore, we included the `atomic` directive before performing these critical operations. We note that the `converged` flag doesn't need to be modified safely. In fact, it is sufficient that during a single iteration no thread sets it to `false`.
Furthermore, we note that second loop depends only on centroids, with a cost of $O(K \times D)$ at each iteration i, where

typically $K \ll N$. Hence, this operation doesn't require parallelization. On the contrary, the introduction of parallelism may involve overhead costs, which may actually decrease performance.

Listing 3 K-Means parallel iteration.

```
bool KMeansIteration() {
    // Variables to store cluster informations.
    double sum[K * dimensions] = {0};
    int size[K] = {0};

    // Convergence flag (true at the beginning).
    bool converged = true;
    #pragma omp parallel for schedule(static)
    ↪ default(none) shared(points, centroids,
    ↪ sum, size, converged)
    for(int i=0; i<N; i++) {
        double minDist = DBL_MAX;
        int minClusterId = -1;

        for(int j=0; j<K; j++) {
            double dist = distance(i, j);

            if(dist < minDist) {
                minDist = dist;
                minClusterId = j;
            }
        }

        // Assign point to closest cluster.
        points.clustersIds[i] = minClusterId;

        // Store clusters informations safely.
        for(int dim=0; dim<dimensions; dim++) {
            #pragma omp atomic
            sum[minClusterId + K * dim] +=
            ↪ points.coordinates[i + N * dim];
        }
        #pragma omp atomic
        size[minClusterId]++;
    }

    for(int j=0; j<K ; j++){
        double tmpCoordinates[dimensions] = {0};

        for(int dim=0; dim<dimensions; dim++) {
            // Store previous informations.
            tmpCoordinates[dim] =
            ↪ centroids.coordinates[j + K *
            ↪ dim];

            // Update centroid.
            centroids.coordinates[j + K * dim] =
            ↪ sum[j + K * dim] / size[j];

            // Check for convergence.
            if (fabs(tmpCoordinates[dim] -
            ↪ centroids.coordinates[j + K *
            ↪ dim]) > EPSILON)
                converged = false;
        }
    }
    return converged;
}
```

We observe that, we can exploit vectorization to compute distance, since this operation has no dependencies between the data (e.g. loop-carried dependency). This optimization could improve the performance of our code, especially when working with high-dimensional data. To enable vectorization, we can use the `simd` directive offered by OpenMP. This directive allows us to vectorize a nested loop by splitting it into chunks that fit a SIMD vector register. In addition, using the `reduction` clause allows us to perform a reduction on the variable `sum`. This clause instantiates the variable `sum` as private and applies the `+` reduction operator at the end of the construct.
In 4 is shown the implementation of the method `distance`, which exploits SIMD parallelism to calculate the distance between a point i and a centroid j.

Listing 4 K-Means parallel distance.

```
const double distance(const int pointId, const
↪ int centroidId) {
    double sum = 0;
    #pragma omp simd reduction(+:sum)
    for (int dim = 0; dim < dimensions; dim++) {
        sum += (centroids.coordinates[centroidId
        ↪ + K * dim] -
        ↪ points.coordinates[pointId + N *
        ↪ dim]) *
        ↪ (centroids.coordinates[centroidId + K
        ↪ * dim] - points.coordinates[pointId +
        ↪ N * dim]);
    }

    return sqrt(sum);
}
```

5. Correctness

To verify the correctness of the algorithm, can be visualized a plotted representation of the results obtained from a clustering. For this purpose, we used the Gnuplot library [4] to generate the cluster result plot at the i-th iteration and ImageMagick [5] to create a GIF animation showing the entire algorithm execution process. We note that, the execution time of the GIF matches the actual time of the algorithm execution.
Figure 2 shows the plotted visualization of the first iteration of the clustering process. Figure 3 shows the last iteration of the clustering process, representing the convergence result.

6. Performance

We now perform a detailed performance analysis of the K-Means algorithm on an Intel i7-10750H processor with 6 cores and Hyper-Threading support. The performance analysis includes a study of speedup as the number of generated points, number of clusters and data dimensions change. In addition, we made a comparison between sequential code,

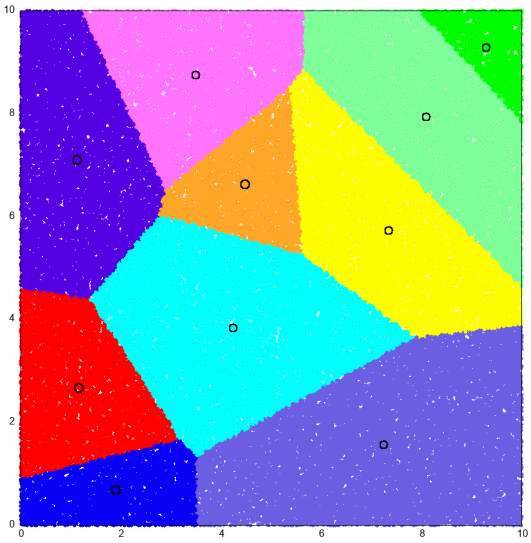

Figure 2. First iteration of the clustering process on $N = 100000$ randomly generated data points and $K = 10$ clusters in $D = 2$ dimensions. The algorithm was executed in parallel using a total of 6 threads.

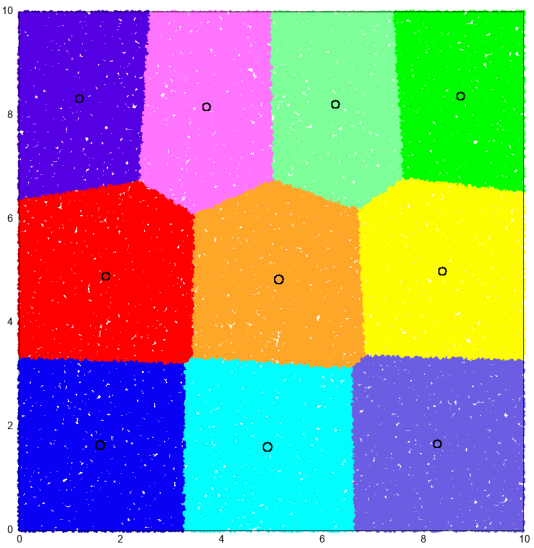

Figure 3. Final iteration (i.e. 87-th iteration) of the clustering process, representing the convergence result. The process took a total execution time of 1.34 seconds.

implemented with an Array of Structures (AoS) architecture, and parallel code, implemented with a Structure of Arrays (SoA) architecture.

6.1. Speedup

A detailed speedup analysis of the implementation was conducted. For each analysis, the algorithm was run repeatedly for 10 iterations, using 2, 4, 6, 8, 10, 12 threads and considering the average execution time.

The first check aims to evaluate the speedup of the parallel code in relation to the number of cores and the size of the dataset. For this purpose, five different datasets of 2-dimensional points with cardinality of 1000, 10000, 100000, 1000000 and 10000000 respectively were generated and organized into 10 clusters.
The results of this experiment are shown in Table 1.
Figure 4 provides a visual representation of the speedup.
The maximum speedup recorded is 5.48. The results are in accordance with expectations, showing an increase in speedup proportional to the number of threads, with the maximum increase recorded in the context of the largest dataset.

Points \ Threads	2	4	6	8	10	12
1000	1.35	1.94	2.17	2.17	1.85	2.25
10000	1.64	2.66	3.27	3.83	4.32	3.92
100000	1.63	2.81	3.49	3.94	4.88	5.21
1000000	1.63	2.81	3.55	4.07	4.92	5.24
10000000	1.63	2.82	3.62	4.24	5.00	**5.48**

Table 1. Speedup for different number of points.

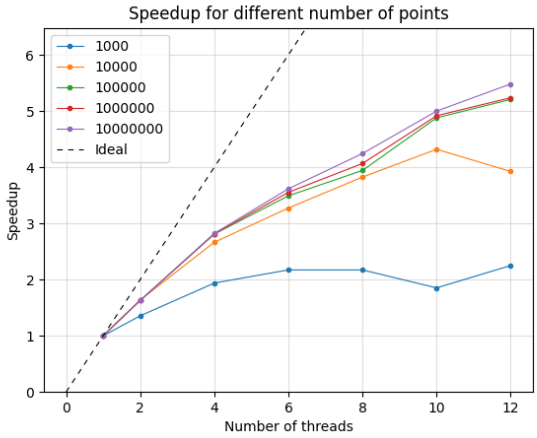

Figure 4. Speedup graph for different number of points.

We next examined the speedup of the parallel code in relation to the number of cores and the number of clusters. For this purpose, we generated a fixed-size dataset consisting of 10000 points in 2 dimensions, organized into 2, 5, 10, 50 and 100 clusters.
The results of this experiment are shown in Table 2.
Figure 5 provides a visual representation of the speedup.
The maximum speedup recorded is 6.52. As might be expected, speedup increases with the number of cores. A similar explanation to that given above can be applied to justify these results. In fact, the results show an increase of speedup proportionally to the number of clusters, with the maximum increase recorded in the context of the largest number of clusters.

Points \ Threads	2	4	6	8	10	12
2	0.92	1.02	1.02	1.00	0.96	0.93
5	1.41	1.99	2.43	2.86	3.24	3.18
10	1.57	2.69	3.18	4.01	4.93	5.32
50	1.80	3.43	4.08	5.04	6.28	6.49
100	1.79	3.44	4.56	5.26	6.27	**6.52**

Table 2. Speedup for different number of clusters.

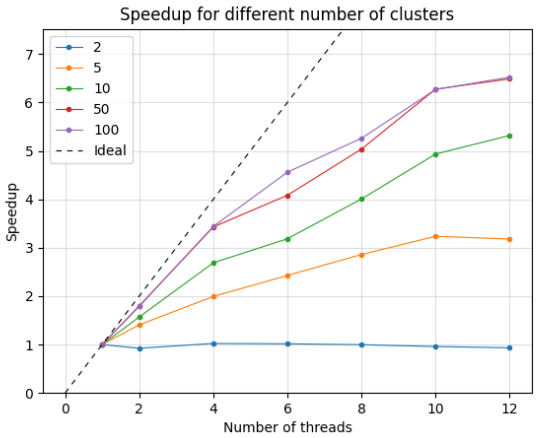

Figure 5. Speedup graph for different number of clusters.

Finally, we examine the speedup of the parallel code in relation to the number of cores and the spatial dimensions of the points. For this purpose, we generated a fixed-size dataset consisting of 10000 points organized into 10 clusters, with dimensions of 2, 3, 5, 10 and 20.
The results of this experiment are shown in Table 3.
Figure 6 provides a visual representation of the speedup.
Again, we observe an increase in speedup proportional to the number of cores. However, the computational complexity associated with distance calculation increases with the dimensions of the points, leading to worse performance for high-dimensional datasets with the same number of threads. Nevertheless, figure 7 shows that the use of vectorization provides a slight performance improvement when dealing with high-dimensionality datasets. This suggests that despite increasing complexity, vectorization optimization helps to mitigate performance decrement, providing an added benefit when dealing with high-dimensionality data.

Points \ Threads	2	4	6	8	10	12
2	1.63	2.67	3.34	3.86	4.78	**5.11**
3	1.67	2.68	3.30	3.74	4.61	5.02
5	1.61	2.48	3.10	3.60	4.45	4.89
10	1.59	2.35	2.95	3.47	4.31	4.65
20	1.69	2.32	2.87	3.45	4.23	4.55

Table 3. Speedup for different dimensions.

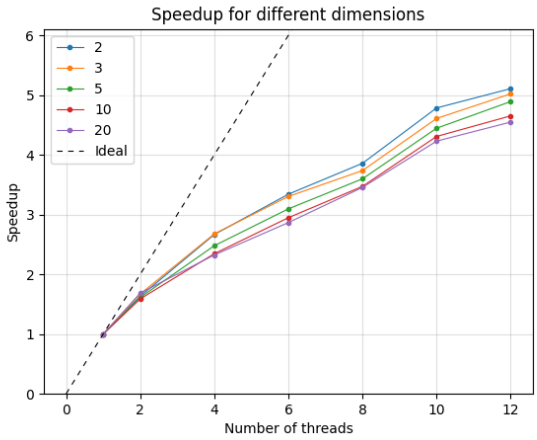

Figure 6. Speedup graph for different dimensions.

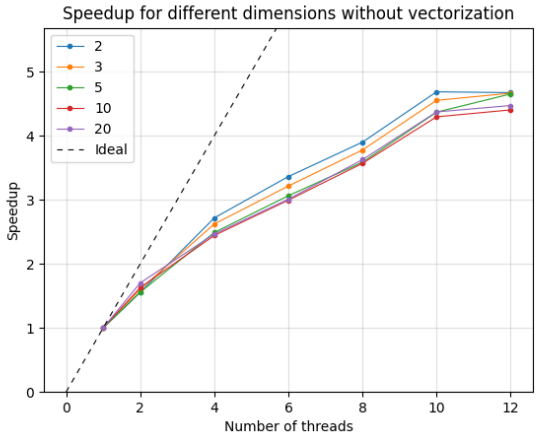

Figure 7. Speedup graph for different dimensions without vectorization.

6.2. AoS vs SoA

Now, we proceed with a comparison of execution times between the sequential implementation based on the AoS (Array of Structures) architecture and the parallel implementation with a single thread based on the SoA (Structure of Arrays) architecture. AoS and SoA architectures represent different approaches in storing and organizing data in a program. At the hardware level, the use of SoA offers significant advantages in terms of parallelism and efficient memory access. The SoA architecture allows temporal and spatial locality of data in memory, facilitating more efficient access to contiguous data. In addition, SoA could potentially be more efficient because of the ability to vectorization and be more cache-friendly.

To make this comparison, we ran sequential and parallel code (with a single thread) using 5 different datasets with sizes of 1000, 10000, 100000, 1000000 and 10000000 two-

dimensional points, organized into 10 clusters.
Figure 8 shows the comparison of execution times. The results show that the SoA architecture in this application leads to a more efficient solution. Interestingly, this improvement seems independent of the size of the dataset.

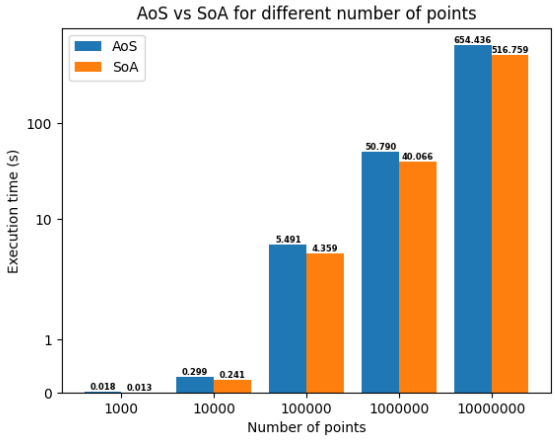

Figure 8. Comparison of AoS and SoA execution times for different dataset sizes.

7. Conclusions

We have seen that the K-Means algorithm has been implemented in various ways, using a sequential and parallel approach. Adopting a parallel approach with OpenMP has proven to be particularly advantageous with an increasing number of variables involved, enabling a significant improvement in computational performance.
In addition, the application of optimization techniques, such as vectorizing and organizing the data in SoA architecture, helped to achieve small improvements over the sequential implementation. These arrangements helped to optimize the overall speedup of the process.

References

[1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. 1

[2] Wikipedia contributors, "K-means clustering — Wikipedia, the free encyclopedia," 2023. [Online; accessed 10-November-2023]. 1

[3] M. F. Saputra, T. Widiyaningtyas, and A. Wibawa, "Illiteracy classification using k means-naïve bayes algorithm," *JOIV : International Journal on Informatics Visualization*, vol. 2, p. 153, 05 2018. 1

[4] T. Williams, C. Kelley, and many others, "Gnuplot 4.6: an interactive plotting program." http://gnuplot.sourceforge.net/, April 2013. 4

[5] ImageMagick Studio LLC, "Imagemagick." https://imagemagick.org. 4