K-means clustering algorithm with OpenMP in C++

Mirko Bicchierai
E-mail address
mirko.bicchierai@edu.unifi.it

Abstract

K-Means is a popular unsupervised learning algorithm used in both data mining and machine learning. Its primary objective is to partition a dataset into a user-defined number of clusters, facilitating the discovery of patterns within the data. In recent times, datasets have grown significantly in size, comprising a vast number of observations and high dimensionality. Consequently, the demand for faster and more efficient methods to apply these algorithms has become imperative. This project involves the implementation of the K-Means algorithm in C++ in both sequential and parallel versions, and the analysis of the speedup between the two versions. All the code used in this project can be found in the repository: `https://github.com/MirkoBicchierai/Parallel_Programming_K-Means`.

1. Introduction

In this report, we delve into the details of the K-means algorithm, starting from a theoretical overview of its functioning to a practical implementation in C++. Additionally, the algorithm has been parallelized on the CPU using the OpenMP library. Furthermore, we have also analyzed the K-means++ initialization. In addition to the standard random initialization, it has also been parallelized on the CPU. As we will see, the parallel version significantly reduces the execution times of the algorithm, allowing for the analysis of very large datasets in a considerably shorter time interval.

2. Dataset Generation

For generating the datasets used in the conducted tests, we generated n points in d dimensions with a standard deviation of 4 using the *make-blobs* function from the *sklearn* library in Python.

Certainly, points could have been generated much more simply, for example, randomly in a bounded space, and the algorithm would have functioned in the same way. However, for graphical purposes and to fully demonstrate its effectiveness, this mode of generation was preferred. It directly generates k blobs, which aids in visually understanding the clustering process.

For these tests, we utilized datasets generated in this manner with varying values of:

$$n = (10^2, 10^3, 10^4, 10^5, 10^6, 10^7)$$

for each:

$$k = (3, 5, 10, 15, 20, 25, 30, 40, 50)$$

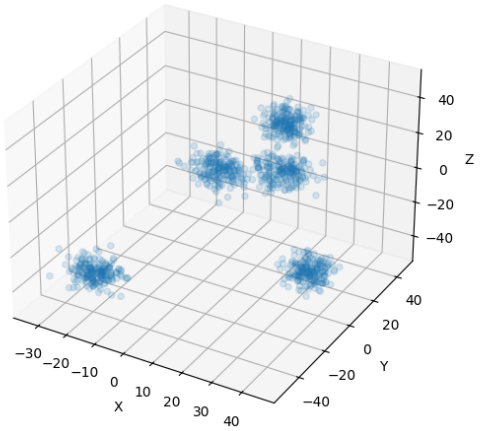

Figure 1. Example of $n = 1000$ points generated for $k = 5$.

2.1. K-means clustering algorithm

K-means clustering is a technique derived from signal processing, originally utilized for vector quantization. Its goal is to divide a dataset containing n observations into k clusters, where each observation is assigned to the cluster with the nearest mean, also known as the cluster center or centroid. These centroids serve as prototypes for their respective clusters. K-means clustering minimizes within-cluster variances, specifically squared Euclidean distances. It's worth noting that it doesn't minimize regular Euclidean distances, which would present a more complex optimization problem known as the Weber problem. In this context, while the mean optimizes squared errors, only the geometric median minimizes Euclidean distances. The problem of K-means clustering is computationally challenging, classified as NP-hard. Nevertheless, efficient heuristic algorithms exist that converge rapidly to a local optimum.

Given a set of observations $(x_1, x_2, ..., x_n)$, where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into $k(\leq n)$ sets $S = S_1, S_2, ..., S_k$ so as to minimize the within-cluster sum of squares (WCSS) (i.e. variance). Formally, the objective is to find:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2$$

$$\boldsymbol{\mu_i} = \frac{1}{|S_i|} \sum_{\mathbf{x} \in S_i} \mathbf{x}$$

The objective function in k-means is the WCSS (within cluster sum of squares), after each iteration decreases and so we have a nonnegative monotonically decreasing sequence. This guarantees that the k-means always converges, but not necessarily to the global optimum. The algorithm has converged when the assignments no longer change or equivalently, when the WCSS has become stable. The algorithm is not guaranteed to find the optimum

2.2. Algorithm

Di seguito una breve spiegazione sui passi fondamentali dell'algoritmo

- **Initialization**:
  - **Random**: Start by randomly selecting K points from the dataset. These points will act as the initial cluster centroids.
  - **K-means++**: Assigning the first centroid to the location of a randomly selected data point, and then choosing the subsequent centroids from the remaining data points based on a probability proportional to the squared distance away from a given point's nearest existing centroid.
- **Assignment**: For each data point in the dataset, calculate the distance between that point and each of the K centroids. Assign the data point to the cluster whose centroid is closest to it. This step effectively forms K clusters.
- **Update centroids**: Once all data points have been assigned to clusters, recalculate the centroids of the clusters by taking the mean of all data points assigned to each cluster.
- **Repeat**: Repeat steps 2 and 3 until convergence. Convergence occurs when the centroids no longer change significantly or when a specified number of iterations is reached.
- **Final Result**: Once convergence is achieved, the algorithm outputs the final cluster centroids and the assignment of each data point to a cluster.

The complexity is:

$$O(n \times d \times k \times iter)$$

Where n is the cardinality of the dataset, d is the number of dimension, k is the number of clusters and $iter$ is the number of iteration required to converge.

2.3. Initialization

One of the most critical aspects concerning this clustering algorithm revolves around the initialization of centroids. The solution of K-means heavily relies on the initial state of the centroids. Initializing centroids at different points can lead to different solutions of the algorithm. In most cases, the algorithm converges to suboptimal solutions for that instance of the problem. Two types of initialization are commonly used:

- **Random**
- **KMeans++**

2.3.1 Random Initialization

The random initialization of centroids is indeed the simplest approach. We randomly select k points from the dataset as centroids.

Algorithm 1 Random Initialization

```
vector<Point> randomCentroid(int k, vector<
    Point> &data) {
    vector<Point> centroids(k);
    vector<int> indexes(k);
    std::uniform_int_distribution<> distrib
        (0, data.size() - 1);
    int r;
    for (int i = 0; i < k; i++) {
        while(true) {
            r = distrib(gen);
            if(std::find(indexes.begin(),
                indexes.end(), r) == indexes
                .end())
                break;
        }
        indexes.push_back(r);
        centroids[i] = data[r];
    }
    return centroids;
}
```

2.3.2 KMeans++ Initialization

The K-means++ algorithm, in contrast, is more intricate but is highly suitable for parallelization. The concept involves iterating through each point and computing its distance relative to the closest already chosen centroid. The selection of the new centroid is based on a weighted probability distribution, where the weights correspond to the distances calculated previously. The K-means++ initialization is designed to choose initial centroids in a way that they are more representative of the data. This typically results in a solution that is, on average, closer to the global optimum compared to random initialization. Moreover, because the initial centroids are chosen more intelligently, K-means++ tends to converge more quickly than random initialization. This means that the K-means algorithm requires fewer iterations to reach a stable solution, thereby reducing the overall computation time for any clustering algorithm used subsequently.

Algorithm 2 KMeans++ Initialization

```
Point next_centroid (const std::vector<
    double> &dist, const std::vector<Point>
    &data) {
    std::discrete_distribution<> distrib(
        dist.begin(), dist.end());
    return data[distrib(gen)];
}

std::vector<Point> initialization_kmean_seq
    (const std::vector<Point> &data, const
    int k) {
    std::vector<Point> centroids;
    std::uniform_int_distribution<> distrib
        (0, data.size() - 1);
    centroids.push_back(data[distrib(gen)])
        ;
    while (centroids.size() < k) {
        vector<double> distances_glob(data.
            size(), numeric_limits<double>::
            max());
        for (size_t i = 0; i < data.size();
             i++) {
            for (const Point &centroid :
                centroids) {
                distances_glob[i] = std::
                    min(distances_glob[i],
                    euclideanDistance(data[i
                    ], centroid));
            }
        }
        centroids.push_back(next_centroid(
            distances_glob, data));
    }
    return centroids;
}
```

2.4. KMeans clustering sequential implementation

The code utilizes a Point class, which acts as a struct to store the coordinates of a point and its associated cluster ID. This class is also equipped with operator overloads to improve code readability and maintainability. The code implements the K-means algorithm for clustering points. Initially, a list of new centroids is created. Then, for each iteration up to the maximum allowed, the code assigns each point to the nearest centroid and updates the new centroids by calculating the mean of the points assigned to each centroid. The *counts* vector is used to keep track of the cardinality of the new clusters, which is then used to calculate the mean of the distances. At the end of each iteration, the new centroids replace the old ones.

Algorithm 3 KMeans clustering, sequential version

```
vector<Point> kMeans(vector<Point> &data,
    vector<Point> &centroids, int k, int
    maxIterations) {
    vector<Point> newCentroids = std::
        vector<Point>(k, Point());
    for (int iter = 0; iter < maxIterations
        ; ++iter) {
        vector<int> counts(k, 0);
        newCentroids = std::vector<Point>(k
            , Point());
        for (Point &pt: data) {
            double minDistance = distance(
                pt, centroids[0]);
            pt.actualCentroid = 0;
            for (int i = 0; i < k; i++) {
                double x = distance(pt,
                    centroids[i]);
                if (x < minDistance) {
                    minDistance = x;
                    pt.actualCentroid = i;
                }
            }
            newCentroids[pt.actualCentroid]
                 += pt;
            counts[pt.actualCentroid]++;
        }
        for (int i = 0; i < k; i++)
            newCentroids[i] /= counts[i];
        centroids = newCentroids;
    }

    return centroids;
}
```

3. Parallelization

In this project, we have proposed parallel versions of both the K-means algorithm and the K-means++ initialization algorithm. Both of these algorithms are well-suited for parallelization.

It's worth noting that K-means falls into the category of embarrassingly parallel problems. Each point can be assigned to the closest centroid independently of the others, making it an ideal candidate for parallelization.

The same can be said for the K-means++ initialization. Finding the distance of all points from the centroid being considered in a specific iteration is independent for each point, making it well-suited for parallelization.

3.1. Main strategy

The most significant expense in the sequential implementation is associated with the loop responsible for iterating through each point and assigning it to every cluster.

Therefore, the initial idea is to mitigate this cost by assigning a group of points to each available thread. The aim is to achieve a complexity of $O(\frac{n}{N} \times d \times k \times iter)$, where N represents the number of utilized CPUs by parallelizing the main loop mentioned. In order to achive that a reduction like strategy was employed. It's important to note that this cost represents the computational expense for each CPU utilized. Thus, the overall final cost remains unchanged, but we have distributed the workload across multiple CPUs.

First two new private variables were defined: *tmp_newCentroids* and *tmp_counts*. Each core operates on its chunk of points in a manner analogous to the sequential algorithm.

The schedule policy chosen is *static*. The dataset is divided into equal parts. A *nowait* clause was added since there is no need for threads to synchronize before entering the critical section. On the contrary, it could improve performance if threads finish their workload asynchronously.

The critical section remains the only overhead compared to the sequential version, primarily for

thread management and aggregating the partial results obtained from various CPUs.

Algorithm 4 KMeans clustering, parallel version

```
vector<Point> kMeans(vector<Point> &data,
    vector<Point> &centroids, int k, int
    maxIterations, int threads) {
int block = ceil(data.size()/threads);
vector<Point> newCentroids(k, Point());
vector<int> counts(k, 0);
for(int iter=0;iter<maxIterations;++iter){
    newCentroids=vector<Point>(k,Point());
    counts = vector<int>(k, 0);
    #pragma omp parallel num_threads(
        threads)
    {
        vector<int> tmp_counts(k, 0);
        vector<Point> tmp_newCentroids(k,
            Point());
        #pragma omp for nowait schedule(
            static, block)
        for (Point &pt: data) {
            double minDistance = distance(
                pt, centroids[0]);
            pt.actualCentroid = 0;
            for (int j = 1; j < k; j++) {
                double dist = distance(pt,
                    centroids[j]);
                if (dist < minDistance) {
                    minDistance = dist;
                    pt.actualCentroid = j;
                }
            }
            tmp_newCentroids[pt.
                actualCentroid] += pt;
            tmp_counts[pt.actualCentroid
                ]++;
        }
        #pragma omp critical
        {
            for (int i = 0; i < k; i++) {
                newCentroids[i] +=
                    tmp_newCentroids[i];
                counts[i] += tmp_counts[i];
            }
        }
    }
    for (int i = 0; i < k; i++)
        newCentroids[i] /= counts[i];
    centroids = newCentroids;
}
}
```

When entering the critical section, all partial results from various CPUs are aggregated. This includes the cardinalities and the sum of positions of all points in that cluster of that data chunk, stored in two global vectors, ”newCentroids” and ”counts”. At this point, we can proceed as in the sequential version by setting the centroids' positions to the mean of the positions of the points in their cluster.

3.2. Kmeans++ Initialization

As in the previous case, the objective here is to reduce the algorithm's cost by starting with the inner loop over the dataset points, bringing it down to a factor of $\frac{n}{N}$. However, this parallelism must be executed every time a new centroid is added to the list of centroids. In this case, there are no additional mechanisms for managing the running threads, so we do not have management overhead.

Algorithm 5 KMeans++ Initialization, parallel version

```
std::vector<Point> initialization_kmean_par
    (const std::vector<Point> &data, const
    int k, const int t) {
    std::vector<Point> centroids;
    centroids.reserve(k);
    std::uniform_int_distribution<> distrib
        (0, data.size() - 1);
    centroids.push_back(data[distrib(gen)])
        ;
    int chunk = ceil(data.size() / t);
    while (centroids.size() < k) {
        vector<double> distances_glob(data.
            size(), numeric_limits<double>::
            max());
        #pragma omp parallel for schedule(
            static, chunk) num_threads(t)
            for (int i = 0; i < data.size()
                ; i++) {
                for (const Point &centroid
                    : centroids) {
                    distances_glob[i] = std
                        ::min(distances_glob
                        [i],
                        euclideanDistance(
                        data[i], centroid));
                }
            }
        centroids.push_back(next_centroid(
            distances_glob, data));
    }
    return centroids;
}
```

The strategy applied is the same: divide the dataset into chunks, one for each CPU. The schedule policy chosen is *static*, as each iteration of the loop takes the same amount of time as in the previous case.

3.3. Vectorization

Additionally, we can utilize vectorization in the distance calculation. This could enhance the performance of our code, especially when dealing with high-dimensional data. Vectorization in this case would be achieved by leveraging the omp directive provided by OpenMP. Despite theoretically working, this approach does not seem to improve the performance of our algorithm in the presented case, where the dimensionality of the points analyzed is 3. Tests were also conducted for dimensions 5 and 10, but no improvements were observed from vectorization. The benefits of vectorization are likely to be seen with significantly higher dimensions.

Algorithm 6 Compute Euclidean Distance by two point

```
double euclideanDistance(const Point& p1,
    const Point& p2) {
    double dist = 0;
    #pragma omp simd
    for (int i = 0; i < DIM; i++)
        dist += (p1.coordinate[i] - p2.
            coordinate[i]) * (p1.coordinate[
            i] - p2.coordinate[i]);
    return sqrt(dist);
}
```

4. Output of the Algorithm

The correctness of the algorithm, both in its sequential and parallel forms, was verified graphically by plotting the results of the clusters generated by the algorithm in the case of 3-dimensional points (as higher dimensions are not easily representable).

In the graphs 2 and 3, the dimensionality of the datasets used is 1000 for better visualization compared to the 10^7 case. Additionally, the number of clusters is kept small to ensure proper comprehension of the graph. The "X" markers indicate the centroids found by the algorithm.

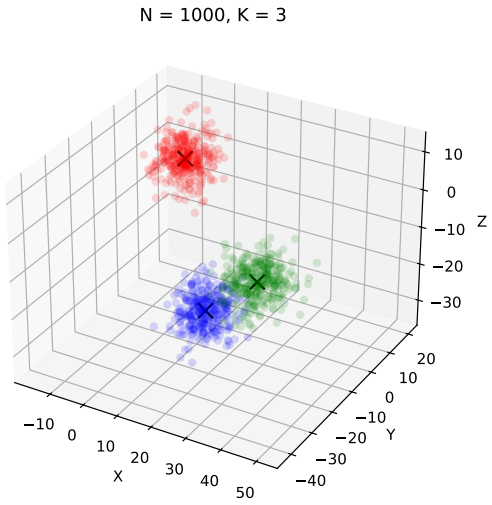

Figure 2. Example of output of the parallel version of the KMeans clustering for $n = 1000$ and $k = 3$.

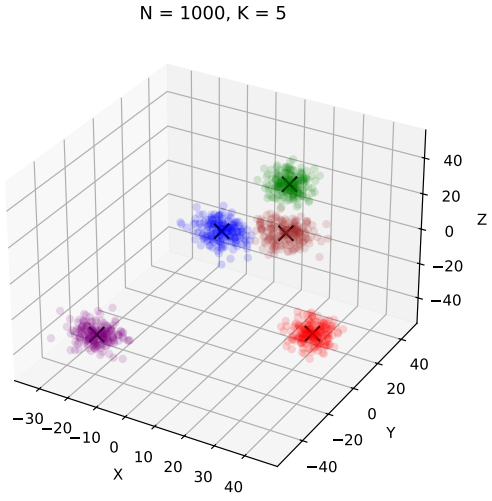

Figure 3. Example of output of the parallel version of the KMeans clustering for $n = 1000$ and $k = 5$.

These are just a few of the outputs produced by the algorithm during the conducted tests. To view all of them, please navigate to the "plots" folder within the project repository.

5. Hardware used

Experiments were performed on 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz (Mobile) (8 CPU cores and 16 threads) and NVIDIA GeForce RTX 3050 Ti (Mobile).

6. Results

The analysis of the results in terms of execution times was divided into two parts: one for the initialization of centroids with the KMeans++ algorithm and the other for the KMeans clustering algorithm.

For the termination of the clustering algorithm, there are two criteria. The first is reaching a maximum number of iterations set, and the second is when the centroids at iteration i are all in the same position as in iteration $i - 1$. However, for the analysis of the algorithm's execution times, only the first termination criterion was used, setting the maximum number of iterations to 150.

For both KMeans++ initialization and the clustering algorithm, each recorded execution time is the average of 100 executions of the same problem instance to ensure that the recorded time is as accurate as possible. Only in the case of the dataset containing 10^7 points, the average was taken over 50 executions instead of 100 because the execution time of a single instance was very high and therefore unmanageable.

6.1. Speedup analysis

In the analysis of execution times, particular focus was placed on the metric of speedup, which is the ratio between the execution time of the algorithm in sequential version and the execution time of the algorithm in parallel version. Both algorithms were tested using 1, 2, 4, 8, and 16 threads.

6.1.1 KMeans clustering

The analysis of the obtained speedup can be divided into two categories: the first based on the **number of observations** in the dataset, and the second based on the **number of clusters**.

In the first case, the number of clusters to generate was fixed and it was tested for each dataset size ranging from 10^2 to 10^7 in powers of 10. In the second case, the dataset size in terms of the number of observations was fixed, and it was tested for each number of clusters considered ($k = (3, 5, 10, 15, 20, 25, 30, 40, 50)$).

In both cases, we present some examples of graphs and tables of speedup obtained from the execution of the code. For a comprehensive view of all the results obtained, please refer to the "SpeedUp" folder within the project repository.

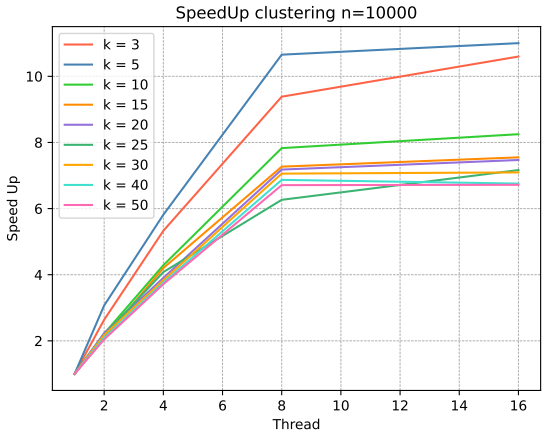

Figure 4. Speed Up graph for $n = 10000$.

Clusters / Thread	**2**	**4**	**8**	**16**
3	2.63	5.33	9.40	10.59
5	3.06	5.81	10.65	**11.00**
10	2.21	4.28	7.82	8.24
15	2.08	4.20	7.26	7.54
20	2.23	3.90	7.18	7.46
25	2.17	4.07	6.26	7.16
30	2.17	3.83	7.05	7.09
40	2.09	3.76	6.86	6.75
50	2.04	3.71	6.70	6.71

Table 1. Speed Up table for $n = 10000$.

As shown in the graph 4 and its associated table 1, concerning the speedups recorded when $n = 10000$, the highest observed value is **11.00** in the case with 5 clusters and 16 threads.

Already from these initial results, we observe a trend showing that for a smaller number of clusters, better results are obtained, which then stabilize to lower but still excellent speedup values. This is because parallelization was performed on the loop related to the number of points in the dataset, rather than the number of clusters.

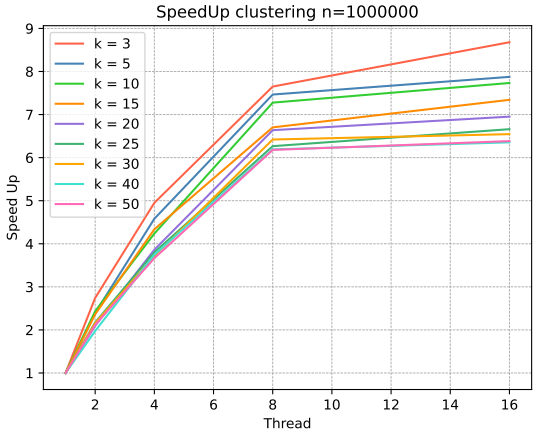

Figure 5. Speed Up graph for $n = 1000000$.

Clusters / Thread	**2**	**4**	**8**	**16**
3	2.74	4.95	7.64	**8.67**
5	2.39	4.58	7.46	7.87
10	2.43	4.23	7.27	7.32
15	2.36	4.33	6.70	7.34
20	2.09	3.86	6.63	6.95
25	2.17	3.80	6.26	6.66
30	2.15	3.71	6.42	6.54
40	1.97	3.74	6.19	6.35
50	2.11	3.66	6.17	6.38

Table 2. Speed Up table for $n = 1000000$.

As shown in the graph 5 and its associated table 2, concerning the speedups recorded when $n = 1000000$, the highest observed value is **8.67** in the case with 3 clusters and 16 threads. Similarly in this case, we observe that for a smaller number of clusters, slightly better results are obtained.

In general the results of this type of experiments are in line with what we expected, in particular the speedup increase with the number of threads, the fact that the results slightly worsen with a higher number of clusters reinforces the idea that essentially only the loop related to the number of points was parallelized, not the number of clusters. In the second experiment, a better understanding is provided of what happens as the dataset size varies, rather than the number of clusters.

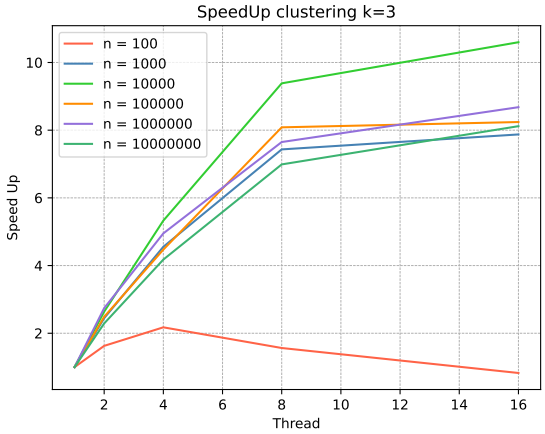

Figure 6. Speed Up graph for $K = 3$.

Clusters / Thread	**2**	**4**	**8**	**16**
100	1.62	2.17	1.56	0.82
1000	2.44	4.55	7.43	7.87
10000	2.63	5.32	9.38	**10.59**
100000	2.48	4.47	8.08	8.24
1000000	2.74	4.95	7.64	8.68
10000000	2.28	4.17	6.98	8.11

Table 3. Speed Up table for $K = 3$.

From the graphs 6, 7, and the tables 3, 4, we can immediately notice that for very low values of the number of points in the dataset, the overhead of thread management in these cases leads to disadvantages compared to the sequential version. This results in a speedup between 0 and 1, meaning that the sequential version performs better than the parallel version.

Clusters / Thread	**2**	**4**	**8**	**16**
100	2.12	3.14	2.88	1.77
1000	2.23	4.33	7.36	**7.77**
10000	2.08	4.20	7.26	7.54
100000	2.07	3.82	6.91	7.27
1000000	2.36	4.33	6.70	7.34
10000000	2.28	4.19	6.57	7.05

Table 4. Speed Up table for $K = 15$

Even in the case of these experiments, the re-

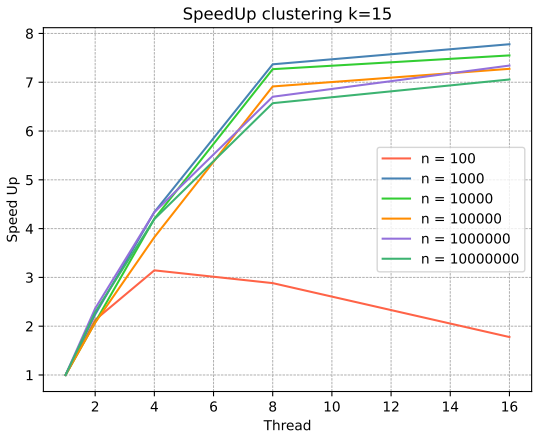

Figure 7. Speed Up graph for $K = 15$.

sults align with theoretical expectations in terms of the speedup obtained based on the number of threads used. Additionally, as the number of clusters and the number of observable data points in the dataset increase, the speedup tends to converge to values between 7 and 8.

6.1.2 KMeans++ Initialization

As in the previous case, in the analysis of KMeans for clustering, the analysis of the obtained speedup can be divided into two categories: the first based on the **number of observations** in the dataset, and the second based on the **number of clusters**. The same values of k and n were used as in the previous case.

Clusters / Thread	**2**	**4**	**8**	**16**
3	1.40	1.78	2.05	2.13
5	1.47	2.00	2.35	2.36
10	1.65	2.35	2.53	2.56
15	1.45	2.28	3.19	3.28
20	1.48	2.40	2.60	3.10
25	1.64	2.64	3.86	4.01
30	1.64	2.83	4.06	4.12
40	1.67	2.77	4.32	4.27
50	1.67	2.80	4.27	**4.32**

Table 5. Speed Up table for $n = 10000$.

As we can see from the graphs 8, 9, and the associated tables 5, 6, the speedup increases linearly with the increase in the number of centroids that the KMeans++ algorithm initializes. This result is exactly what we expect given how the parallelization was implemented in the algorithm's code, as the loop that was parallelized, related to the points in the dataset, is nested within the loop for k. Therefore, if there is speedup, it is natural to see an increase in it as k grows. Another factor to consider is that the remaining sequential part of the code, namely the call to the *next_centroid()* function, which extracts a point from the dataset using a discrete distribution with bounds computed in the parallel section, is computationally expensive. This factor significantly reduces the achieved speedup, although it is still good.

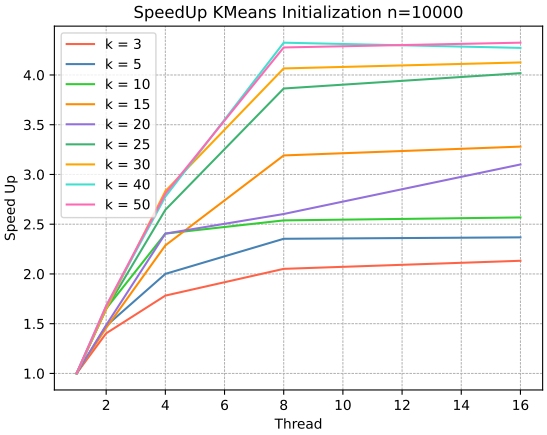

Figure 8. Speed Up graph for $n = 10000$.

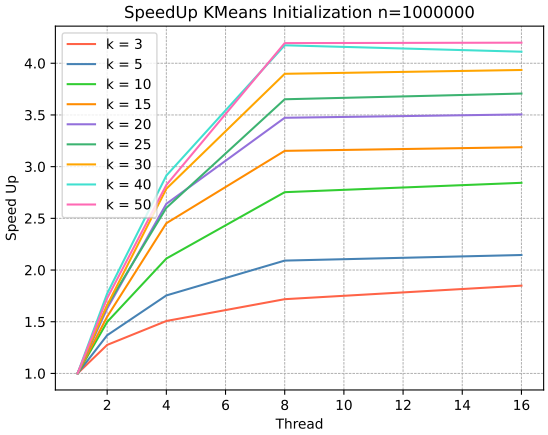

Figure 9. Speed Up graph for $n = 1000000$.

Clusters / Thread	2	4	8	16
3	1.27	1.50	1.71	1.84
5	1.36	1.75	2.09	2.14
10	1.49	2.11	2.75	2.84
15	1.54	2.45	3.15	3.18
20	1.62	2.64	3.47	3.50
25	1.65	2.60	3.65	3.70
30	1.66	2.78	3.89	3.93
40	1.77	2.91	4.17	4.11
50	1.72	2.82	4.19	**4.20**

Table 6. Speed Up table for $n = 1000000$.

As we can see from the tables provided, concerning the speedups recorded when $n = 10^4$, the highest observed value is **4.32** in the case with 50 clusters and 16 threads, In the case of $n = 10^6$, the best value of **4.20** i it is consistently observed in the case with 50 clusters and 16 threads.

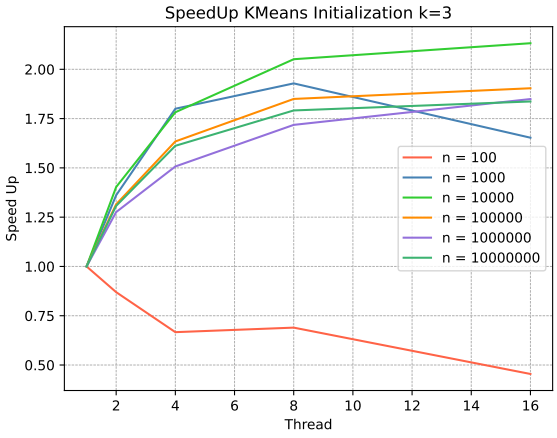

Figure 10. Speed Up graph for $K = 3$.

Dataset / Thread	2	4	8	16
100	0.86	0.66	0.68	0.45
1000	1.36	1.80	1.92	1.65
10000	1.40	1.78	2.05	**2.13**
100000	1.31	1.63	1.84	1.90
1000000	1.27	1.50	1.71	1.84
10000000	1.30	1.61	1.79	1.83

Table 7. Speed Up table for $K = 3$.

In this second type of experiment, where we fix the number of clusters and observe how the speedup varies with n, the number of points in the dataset, we can clearly see from the graphs 10,11 that for very low n, such as $n = 100$ and $n = 1000$, the overhead cost of thread management leads to a speedup of less than 1, indicating a disadvantage in using the parallel version in terms of execution time in this case. This result for n is nonetheless an expected and valid outcome.

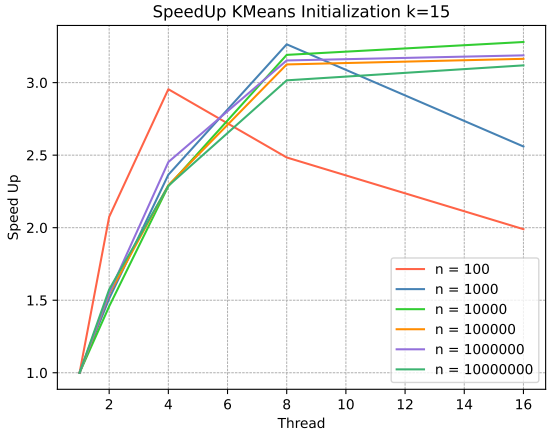

Figure 11. Speed Up graph for $K = 15$.

Dataset / Thread	2	4	8	16
100	2.07	2.95	2.48	1.98
1000	1.51	2.36	3.26	2.55
10000	1.45	2.28	3.19	**3.28**
100000	1.54	2.29	3.12	3.16
1000000	1.54	2.45	3.15	3.18
10000000	1.57	2.88	3.01	3.11

Table 8. Speed Up table for $K = 3$.

In both reported cases, excluding values for n that are too small, the speedup tends to stabilize around 3 for 16 threads, which, despite being low, considering the mentioned issues, is a good value.

7. Conclusions

In conclusion, this project thoroughly examined the K-Means algorithm and its parallelization using the OpenMP framework. We explored

its theoretical foundations, operational steps, and initialization methods, including both random and K-Means++ initializations. By leveraging parallelization, we significantly reduced execution times for large datasets.

Our experiments focused on two key aspects: the number of observations in the dataset and the number of clusters. In both cases, the results aligned well with theoretical expectations. For smaller datasets, we observed that the overhead of managing threads sometimes led to slower performance compared to the sequential version. However, as the dataset size increased, the benefits of parallelization became evident, with speedup values stabilizing between 7 and 8 for the clustering algorithm.

We also found that the number of clusters influenced the performance of the clustering algorithm, with fewer clusters resulting in better speedup values. This is because parallelization was applied to the loop over the data points, not the clusters. Conversely, for the K-Means++ initialization, a higher number of clusters resulted in greater speedup, which was expected. This difference arises because the initialization step benefits more from parallelization when there are more centroids to initialize.

Overall, the parallel K-Means algorithm demonstrated substantial performance improvements, making it a viable option for large-scale data analysis.