Parallel KMeans
Parallel Programming for Machine Learning

Niccolò Arati
niccolo.arati@edu.unifi.it

Abstract

This project aims to analyze the advantages of implementing the KMeans clustering algorithm in parallel. Specifically, we aim to observe the ***speedup****, calculated as the ratio between sequential execution time and parallel execution time, while varying different configurations of the algorithm's main parameters.*
In addition to the algorithm parameters, we will also analyze the results while varying the number of threads related to the parallel configuration.

Future Distribution Permission

1. Introduction

KMeans is an unsupervised learning clustering algorithm. Given a dataset of points and a number K of clusters, KMeans assigns each point to one of the K clusters, aiming to minimize the following objective function for each point x:

$$SSE = \sum_{i=1}^{K} \sum_{x \in C_i} ||x - m_i||^2 \tag{1}$$

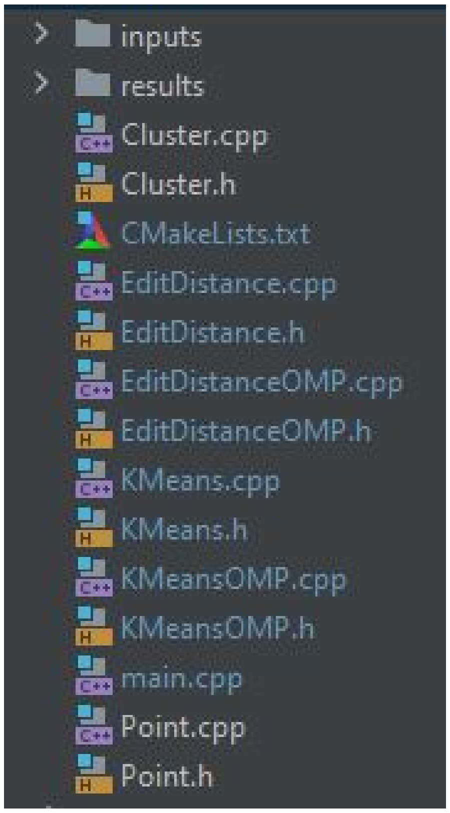
In Eq. (1), C_i represents the i-th cluster and m_i the corresponding centroid, while SSE indicates the Sum of Squared Error. The distance measure is relative to each point's dimensions.
The clustering process specified by the algorithm can be summed up into 4 main steps:

- **Initialization** of clusters: K points from the dataset are randomly chosen, and those points will represent the initial cluster centroids;
- **Assignment** of points: each point is assigned to the nearest cluster. To do this, the distance of the point from each centroid is calculated, and the point is assigned to the cluster corresponding to the centroid within the smallest distance;
- Calculation of **centroids**: the centroids are updated by calculating the mean of the coordinates of the points assigned to the cluster they represent;
- The process of point assignment and centroid recalculation is repeated until convergence is reached.

In this work, **convergence** was set using a maximum number of iterations, 35, and the percentage of points that change clusters after each iteration. If this percentage drops below 0.1%, the algorithm terminates.
The purpose of the project is to measure the speedup achieved by parallelizing the algorithm and comparing it with its sequential version. OpenMP was used for parallelization, and the programming language used is C++.
All experiments were conducted on the SSH server ”papavero.dinfo.unifi.it”.
Additionally, it was chosen to handle also points with number of dimensions greater than 2.

2. Code Structure

CLion was used as the IDE and MinGW as the compiler.
The code includes files from another project with



```
> inputs
> results
  Cluster.cpp
  Cluster.h
  CMakeLists.txt
  EditDistance.cpp
  EditDistance.h
  EditDistanceOMP.cpp
  EditDistanceOMP.h
  KMeans.cpp
  KMeans.h
  KMeansOMP.cpp
  KMeansOMP.h
  main.cpp
  Point.cpp
  Point.h
```

Figure 1. Code structure.

a common directory. The files relevant to this project are:

- **Cluster.h/cpp**: contain the declaration of the Cluster class and the definitions of its methods;
- **KMeans.h/cpp**: contain the declaration of the KMeans class and the definitions of its methods;
- **KMeansOMP.h/cpp**: contain the declaration of the KMeansOMP class and the definitions of its methods with parallelized implementation;
- **Point.h/cpp**: contain the declaration of the Point class and the definitions of its methods;
- **main.cpp**: contains the functions related to the tests and other auxiliary functions.

We also consider the folder **inputs**, it contains 4 CSV files. The information memorized in them will be used to initialize the points for the tests to be clustered with KMeans.

2.1. Class Point

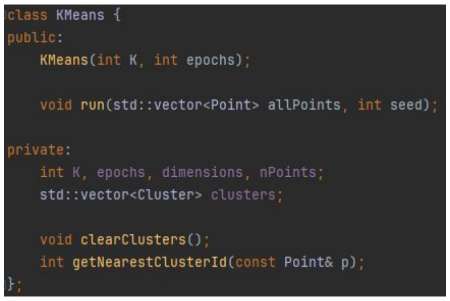
The Point class represents the points on which the algorithm will be executed.
The attributes are: pointId, which specifies a unique identifier for the point; clusterId, which specifies the identifier of the cluster to which the point is assigned; dimensions, indicating the number of coordinates of the point; and values, which is a vector where the coordinates are stored.
There are also getter and setter methods for the various attributes, and two private methods are defined: linetoVecTXT and linetoVecCSV. These methods receive a line from a text or CSV file respectively, and convert the information into the coordinates of the point.
Finally, two constructors are defined, the second one is the copy constructor.

2.2. Class Cluster

The Cluster class represents the clusters on which the points will be grouped through the execution of the algorithm.
The attributes are: clusterId, which indicates a unique identifier for the cluster; centroid, which is a vector where the coordinates of the centroid are stored; and points, a vector containing the points (stored as objects of class Point) assigned to the cluster.
The methods of this class include getters and setters for clusterId, two methods to handle the centroid coordinates (to obtain them and to change them), three methods to handle the points assigned to the cluster (including a method to reset the points vector), and finally, a method that returns the dimension of the cluster (number of points assigned to it).
Finally, there is of course a constructor for the class.

2.3. Class KMeans

The KMeans class encapsulates the parameters and the methods to execute the algorithm.



```
class KMeans {
public:
    KMeans(int K, int epochs);

    void run(std::vector<Point> allPoints, int seed);

private:
    int K, epochs, dimensions, nPoints;
    std::vector<Cluster> clusters;

    void clearClusters();
    int getNearestClusterId(const Point& p);
};
```

Figure 2. Class KMeans declaration.

Its attributes include: K, indicating the number of clusters; epochs, indicating the maximum number of iterations before stopping the execution of KMeans; two attributes to store the number of points and their associated dimensions (dimensions and nPoints, assigned via the run() method); and finally clusters, an std::vector item where the K clusters are saved.
The methods are: run, which initializes and executes the algorithm, it requires as input a list of points to cluster and a seed (to eventually replicate results, as initially the centroids are randomly chosen); and two private methods, clearClusters to reset the clusters (used when points need to be reassigned to the nearest clusters) and getNearestClusterId to identify the nearest cluster to a given input point.
The constructor requires only the parameters K and epochs, initially assigning 0 to the attributes nPoints and dimensions.

2.4. Class KMeansOMP

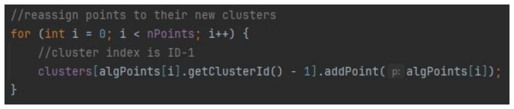
The KMeansOMP class is identical to the KMeans class. The only difference is the implementation of the run method, which is parallelized.
This implementation will be further analyzed in Sec. 3.

2.5. main.cpp

The file contains the tests related to the parallelization of the KMeans algorithm (presented in Sec. 4), along with the main function that executes the tests and the readPointsCSV function.
This last function is responsible for instantiating objects of the Point class on which the algorithm will be executed. As the name suggests, it is designed to read the values of the point dimensions from CSV files, requiring as input the file name and an integer called ”stop,” which indicates how many columns will be considered as the dimensions of the Point object.
For each row read from the file, the function initializes the corresponding Point and adds it to a list (std::vector). The list is then returned as output.

3. Parallelization with OpenMP

In the code of the KMeans class, there are several for loops that could potentially be parallelized; however, many of them have relatively few iterations. Therefore, it was decided to keep the execution sequential for those loops.



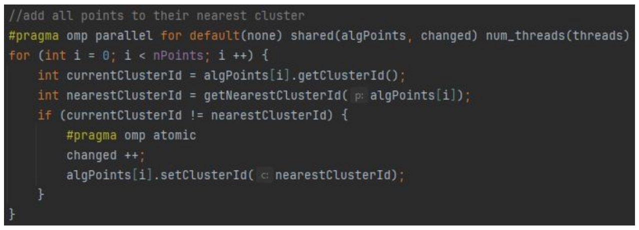
```
//reassign points to their new clusters
for (int i = 0; i < nPoints; i++) {
    //cluster index is ID-1
    clusters[algPoints[i].getClusterId() - 1].addPoint( p: algPoints[i]);
}
```

Figure 3. for cycle which memorizes points in their assigned clusters.

The only case that does not fit into this perspective is the for loop used to assign points to their new cluster, which iterates over all the points to be clustered (Fig. 3).
The reason for not parallelizing this loop is that the only instruction within the loop accesses an std::vector which, with parallelization, would be shared among all threads. Experimentally, it was found that concurrent access to this shared resource results in a heap error. One possible solution could be to use a critical directive, but this would get the parallelization pointless as it would impose that only one thread at a time executes the only instruction within the loop.
The code was therefore parallelized in two points, both related to the run method:

- Where it assigns points to their nearest cluster;
- Where it recalculates the centroid of each cluster.

3.1. Parallelization of the points assignment to their nearest cluster



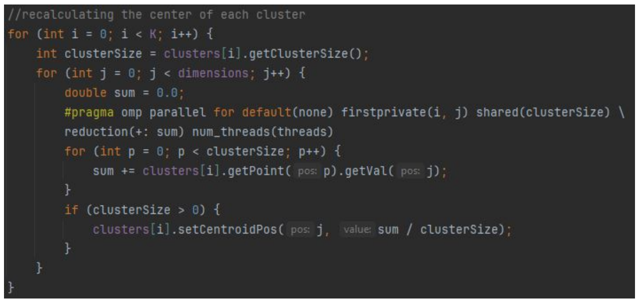
```
//add all points to their nearest cluster
#pragma omp parallel for default(none) shared(algPoints, changed) num_threads(threads)
for (int i = 0; i < nPoints; i ++) {
    int currentClusterId = algPoints[i].getClusterId();
    int nearestClusterId = getNearestClusterId( p: algPoints[i]);
    if (currentClusterId != nearestClusterId) {
        #pragma omp atomic
        changed ++;
        algPoints[i].setClusterId( c: nearestClusterId);
    }
}
```

Figure 4. Parallelization of the points assignment to their nearest cluster.

In Fig. 4, it can be observed how an **omp parallel for** section is defined specifying the shared variables algPoints, which is the list containing all the points to be clustered, and changes, which is a counter related to the number of points that are assigned to a new cluster.
To manage concurrent access to these two variables, an **omp atomic** directive was used in correspondence of the increment of the variable changes. This ensures that access to the variable is atomic, preventing potential simultaneous variations by more threads.
On the other hand, for algPoints it was not necessary to specify an **omp critical** directive, as concurrent access to the list is used to modify the clusterId attribute of each element in the list. Therefore, there will never be an attempt of simultaneous access by two or more threads to the same Point object, as indeed would have occurred in Fig. 3.
In that case, access to the cluster list is made to add Point objects to the private lists of points associated with Cluster objects, and by doing so there is the possibility of concurrently calling the addPoint method on the same Cluster object.

3.2. Parallelization of the calculation related to the new centroids coordinates

In Fig. 5, we observe three nested for loops. It was decided to insert an **omp parallel for** block in correspondence to the innermost loop, because there is a dependency between the three loops that would prevent a potential omp parallel for section at the outermost loop by specifying a collapse



```
//recalculating the center of each cluster
for (int i = 0; i < K; i++) {
    int clusterSize = clusters[i].getClusterSize();
    for (int j = 0; j < dimensions; j++) {
        double sum = 0.0;
        #pragma omp parallel for default(none) firstprivate(i, j) shared(clusterSize) \
        reduction(+: sum) num_threads(threads)
        for (int p = 0; p < clusterSize; p++) {
            sum += clusters[i].getPoint( pos: p).getVal( pos: j);
        }
        if (clusterSize > 0) {
            clusters[i].setCentroidPos( pos: j,  value: sum / clusterSize);
        }
    }
}
```

Figure 5. Parallelization of the calculation related to the new centroids coordinates.

clause.
The indices of the outer for loops are specified as firstprivate, while the cluster size (intended as the number of points belonging to the cluster, which is also the number of iterations of the parallelized for loop) is shared.
The purpose of this code section is to calculate the new centroids of the K clusters. The first for loop iterates over the clusters, the second over the dimensions (interpreted as the number of coordinates) of the centroids, and the third calculates the j-th coordinate of the cluster. To do this, it sums all the j-th coordinates of the cluster points, storing them in a variable called sum, and then divides by the number of points in the cluster.
Because of this, a reduction clause was specified for the sum variable.

4. Experiments conducted

Returning to the main.cpp file, the auxiliary functions written for the tests are as follows:

- testKMeansTime: takes as inputs the parameters to execute KMeans, K and epochs, and a list of points to be clustered. The function then measures the execution time of the algorithm and returns it as the output.

- testKMeansOMPTime: takes as inputs the parameters for KMeans, a list of points to be clustered, and the number of threads for parallelization. The computational time of the parallelized version of KMeans is measured, which is also the output of the function.

- **testKMeans**: function where the actual tests are performed, executed by the main function. The speedup is calculated using the results of the two functions described earlier, without defining a dedicated function for it.

In general, the main parameter considered for the experiments is obviously the number of threads used for parallelization, while for the others we rely on the algorithm's **complexity**:

$$\mathcal{O}(n * K * I * d) \tag{2}$$

In Eq. (2), it is indicated that the complexity depends on the number of points (n), the number of specified clusters (K), the number of iterations (I), and the number of dimensions associated with individual points (d).
We will therefore take into account all these parameters in the tests, except for the number of iterations for which choices have been made previously and will be kept as indicated in Sec. 1 (in the part concerning convergence).
The test results are saved in specific CSV files contained in the "results" folder, and the graphs are generated using some Python code (not provided).
Below are the tests conducted for the KMeans algorithm.

4.1. Test1: minimum number of points

With this initial brief test, the aim was to realize what could be the minimum number of points for which parallelization would prove to be useful. The number of points was chosen as a parameter because the parallel part of the code relates to for loops iterating over the points to be clustered. We would expect that as the number of points (and their dimensions) increases, the speedup between the sequential and parallel versions would be greater with the same number of threads used for parallelization.

Observing Tab. 1, it is clear that with 3000 points in 2 dimensions, the parallel algorithm has slightly lower performance than the sequential version, although increasing the number of clusters to be searched (from 3 to 6) tends to bring the speedup value closer to 1.

	3000 points 2D	19000 points 8D
4 threads	0.946291	2.23869
8 threads	0.948598	3.27011
4 threads	0.989534	2.64693
8 threads	0.964235	3.29542

Table 1. Table showing the values of speedup varying with the number of points to be clustered and the number of threads used for the parallel version. The difference between the first two rows and the last two (always indicated with 4 and 8 threads) is that in the last two rows K has an higher value to increase the complexity of the algorithm.

This could indicate that even with a few more points, we could achieve a positive speedup, and indeed with the second smaller dataset available, 19000 points in 8 dimensions, we already have very significant results despite the relatively small number of threads used.
The decision to not use a high number of threads was made to focus on the number of points and to evaluate better their actual impact.
We have results in line with the expectations, demonstrating the actual utility of the parallel approach. In fact, for the sequential version the increase in complexity due to the growing number of points leads to an increase in computational times, while this growth is much more contained in the parallel version. This inevitably leads to an increase in the speedup value.

4.2. Test2: changing nThreads and K

For the second test, the number of threads and the number of clusters sought by the algorithm varies, as indicated in Fig. 6.
What we expect is that as the number of threads increases, the parallel algorithm will obviously become faster than the sequential one, resulting in an higher speedup. The number of threads is varied between 2 and 16, considering multiples of 2.
Regarding the number of clusters, since they affect the complexity as K increases, the computational time of the sequential algorithm will also increase. Therefore, we would expect that, with more clusters and the same number of threads,

parallelization becomes more impactful.

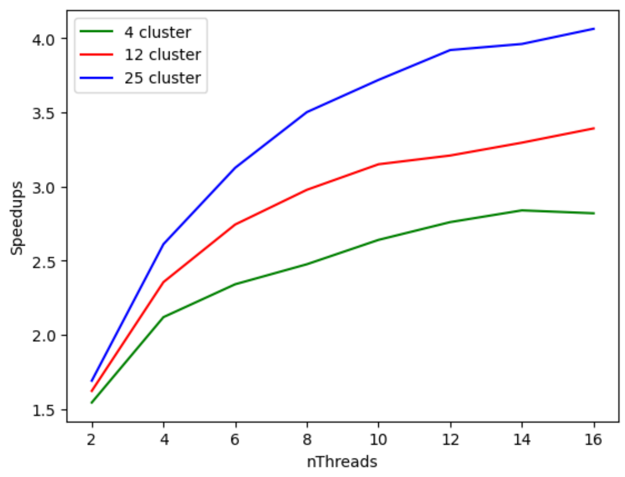


Figure 6. Graph showing the values of speedup varying with nThreads. The 3 lines refer to different values of K, with green representing 4 clusters, red representing 12, and blue representing 25. The algorithm was executed on 500,000 points with 5 dimensions.

It is observed that for all 3 values of K, the increasing number of threads used for parallelization also increases the speedup, as expected, although the growth is not equal for all 3 lines.
It is also noticeable that with a greater number of clusters, the increasing number of threads leads to a higher speedup value, especially when compared with the executions associated with a lower value of K. For example, with 16 threads and 4 clusters, the speedup is 2.81982, while with 16 threads and 25 clusters the speedup is 4.06303.
During testing we stopped at a maximum number of threads equal to 16, but this can obviously be increased. However, the growth of the speedup will not always be constant. We expect that after a certain number of threads, the CPU will become saturated, and therefore we will observe an attenuation of the trend observed in the graphs (i.e. the speedup will remain unchanged as the number of threads increases, because the time spent managing the threads will become greater than the time gained from dividing the work among the threads).

4.3. Test3: varying K

In this final test, the number of clusters K generated by the KMeans algorithm is varied, with a fixed number of threads for the parallel version. The considered values of K are 4, 10, 20, and 30. We would expect that, since K positively influences the complexity of the algorithm, increasing K would also increase the speedup obtained with the parallelized version of KMeans.

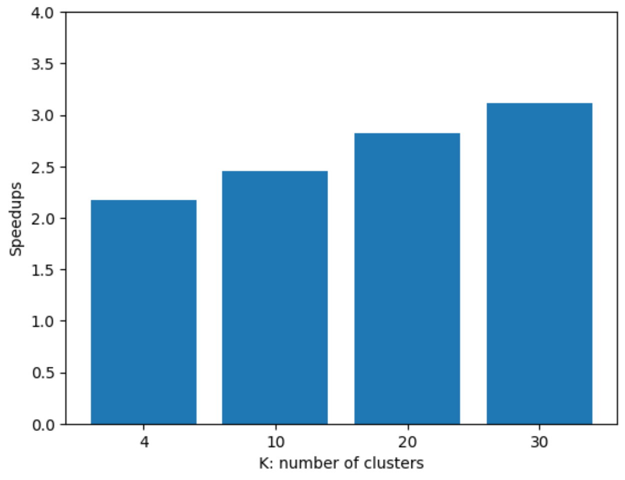


Figure 7. Speedup by varying K with a fixed number of threads set to 8. The algorithm was executed on 4 million points with 3 dimensions.

Again, the number of threads was kept low to isolate the actual impact of the number of clusters on performance.
The results shown in Fig. 7 are in line with expectations. We notice that as K increases, the speedup also becomes higher, indicating that the parallelized version of KMeans handles the increase in algorithm complexity better (with the same parameters as the sequential version).

4.4. Final considerations on number of points

In addition to the first test (Sec. 4.1), we can make general considerations about the parameter indicating the number of points on which we execute the algorithm by simply comparing the results of the other two tests, as they are performed on an increasingly greater number of points (and their associated dimensions).

	Speedup with 8 threads
3000 points 2D	0.948598
19000 points 8D	3.27011
500000 points 5D	2.47619
4mln points 3D	2.17211

Table 2. ”Table with the values of speedup by varying the number of points to be clustered, with a number of threads set to 8.”

The configurations most similar across the various tests have been selected and shown in Tab. 2. Aside from the number of points and dimensions, only the number of clusters K varies slightly (3 for the first row, 5 for the second, and 4 for the last two).

We observe that as the algorithm’s complexity increases due to the number of points and dimensions, initially with the same configuration relative to other parameters, we observe an increase in speedup. However, later on, while the speedup remains positive it decreases.

This is likely due to a slowdown in the heap, which cannot handle that particular number of values (third and fourth rows). Looking at the results of the third test (Sec. 4.3), we can infer another factor leading to this conclusion, namely the value of K. By distributing the points across more clusters (25), we observe a speedup of 3.11685, which is a result comparable to the best configuration presented in the table analyzed in this section.

Another factor that influences this is undoubtedly the number of threads used. By looking at the results of the second test (Sec. 4.2), we see that with 16 threads the results are better (3.50519) than those analyzed in Tab. 2.

The presented results serve as insight for managing the arising complexity due to an increasing number of points and dimensions.