K-Means Parallelization in OpenMP Assignment

Marco Trambusti
Matricola - 7135350
marco.trambusti1@edu.unifi.it

Abstract

The purpose of the assignment is to implement K-Means in C++ first in a sequential version and then parallelize it through the use of OpenMP, in order to compare and analyze the two versions. The analysis conducted showed a significant improvement in the parallel version compared with the serial version, with performance differences evident depending on the size of the datasets. In summary, while parallelization can offer significant performance advantages, it also presents challenges in terms of efficiency.

1. Introduction

The purpose of the mid-term paper is to implement the K-Means algorithm, utilized for clustering points, in C++ first in a sequential version and then parallelize it through the use of OpenMP, in order to compare and analyze the two versions . The dataset used is a dataset of 300147 values related to age and total spent of supermarket customers. The dataset is staggered into subsets of sizes that gradually increase to full size in order to compare performance with different numbers of points. At the end, analysis and performance comparisons of the two versions were performed. To evaluate the performance, the speedup obtained by switching from the sequential to the parallel version was measured considering also the increase in the number of threads allocated from 2 to 16.

1.1. The K-Means algorithm

The K-Means algorithm is a widely used clustering method for partitioning a data set into K distinct clusters.Formally it can be defined as the task of finding a partition $S = \S_1, S_2, ..., S_k\}$ where S satisfies:

$$\arg\min \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|^2 \qquad (1)$$

Although it is relatively simple to implement, its computational cost can become a limiting factor, especially with large volumes of data.In this context, parallelization of the algorithm is an effective strategy to improve its performance.
The K-Means algorithm consists of a few basic steps:

1. **Initialization**: Random selection of K initial centers for clusters.

2. **Cluster assignment**: Each data point is assigned to the nearest centroid.

3. **Updating centroids**: Centroids are recalculated as the average of the points assigned to each cluster.

4. **Repetition**: The assignment and update steps are repeated until the centroids no longer change significantly or a maximum number of iterations is reached.

1.2. Parameters

The algorithm depends on certain parameters that need to be specified. First, the number K of clusters must be known and chosen a priori; In the project code, values of K have been defined in an indicative manner based on the point ranges used; alternatively, more complex solutions can be adopted to choose the best K given the points as Elbow Method, Silhouette Method or Gap Statistic. However, it is not among the objectives of the paper to use the optimal number of clusters so approximate values were used. After that, the way the centroids are defined and how they are initialized must be chosen : in the case of the paper, where the points lie in a two-dimensional Euclidean space, the centroids are initialized randomly among the points in the analyzed set , after which the centroid is in turn recalculated as a point in the space defined by the average of the points belonging to that cluster. Another important detail is the choice of distance: several solutions are possible; a square Euclidean distance is used in this project. The last parameter is the choice of the maximum number of iterations, which could vary from the nature and number of points and clusters; therefore, the number of iterations varies according to the saglions of points that are used. Another stopping criterion that can be defined is convergence; however, this was not implemented since it is not in the scope of the paper to obtain the most accurate clusters.

1.3. Considerations

Is an algorithm that converges very quickly, although it does not guarantee finding the global optimum. The quality of the final result depends mainly on the number and initial position of the clusters. Also, since we have initialized the clusters randomly, it is not certain that repeating the algorithm 2 times on the same points will give the same result.

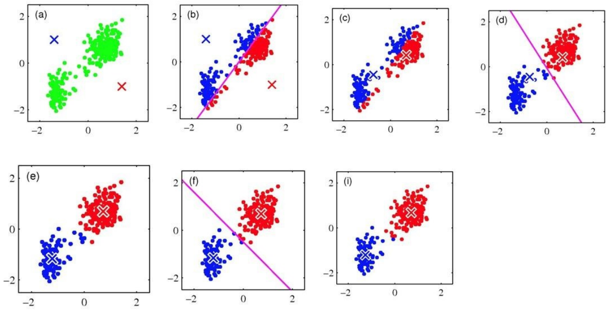

Figure 1. Example of K-Means

2. Implementation

2.1. Representation of Points

The points in the dataset are represented using a struct **Point**

```
struct Point {
   double x, y;
   int cluster;
   double minDist;

   Point() : x(0.0), y(0.0), cluster(-1),
    minDist(std::numeric_limits<double>::max()) {}

   Point(double x, double y) : x(x), y(y), cluster(-1),
    minDist(std::numeric_limits<double>::max()) {}

   [[nodiscard]] double distance(Point p) const {
    return (p.x - x) * (p.x - x) + (p.y - y) * (p.y - y);
   }
};
```

In the first lines we define the coordinates of a point, as well as the cluster to which it belongs and the distance from that cluster. Initially, the point does not belong to any cluster, and was arbitrarily set to -1. As a result, minDist was set to the maximum possible value.

```
double minDist = std::numeric_limits<double>::max();
```

We also define a method, which calculates the Euclidean distance (squared) between this point and the other point

```
[[nodiscard]] double distance(Point p) const {
        return (p.x - x) * (p.x - x) + (p.y - y) * (p.y - y);
}
```

Regardless of the versions the initial steps read the points from the csv file and normalize them via the function `readAndNormalizeCSV`. After that, you will initialize the clusters randomly among the set of points via the function `init_centroids`

2.2. K-means Sequential

The algorithm is divided into two main parts, **assignment to cluster** and **updating centroids** these steps will be repeated for the number of iterations specified a priori.
For the cluster assignment part, the code is constituted as follows:

```
void KMeansHelper::kmeans_sequential(std::vector<Point>& points,
std::vector<Point>& centroids, int k, int epochs) {
    for (int epoch = 0; epoch < epochs; ++epoch) {
        std::vector<float> sumX(k, 0.0f), sumY(k, 0.0f);
        std::vector<int> nPoints(k, 0);
        // Assign points to clusters
        for (auto& point : points) {
            for (int i = 0; i < k; ++i) {
                double dist = point.distance(centroids[i]);
                if (dist < point.minDist) {
                    point.minDist = dist;
                    point.cluster = i;
                }
            }
            //append data to centroids
            int clusterId = point.cluster;
            nPoints[clusterId]++;
            sumX[clusterId] += point.x;
            sumY[clusterId] += point.y;
            // reset distance
            point.minDist = std::numeric_limits<double>::max();
        }
        ...
```

As can be guessed from the code for each point in the dataset the squared Euclidean distance is calculated and the minimum distance values and cluster index are updated, the clusters are initialized into an array of k points passed as an argument to the function. This part of the code encapsulates a critical point regarding the complexity of the algorithm, considering the execution of the algorithm for datasets with a tens or hundreds of thousands of points and with tens of clusters.
After deriving the cluster to which the point belongs, the values related to the sum of the number and coordinates of the points for the cluster identified as the closest (`nPoints,sumX,sumY`) are updated and the distance of the point is reset to the maximum possible value, in view of the next epoch.

As for the part about updating centroids and carried out in the following portion of the code:

```
    for (int i = 0; i < k; ++i) {
        if (nPoints[i] > 0) {
            double newX = sumX[i] / nPoints[i];
            double newY = sumY[i] / nPoints[i];
            centroids[i].x = newX;
            centroids[i].y = newY;
        }
    }
```

Where the new centoids are calculated as the average of the coordinates of the points belonging to the same cluster.

Using GnuPlot, the result of the K-Means algorithm for the maximum number of points (301487) is:

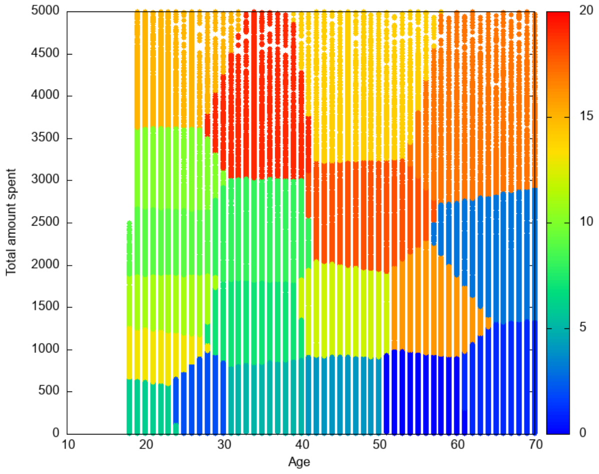

Figure 2. Result of K-Means with 301487 points

3. Parallelization

3.1. Parallelization Points

- **Cluster Assignment**: The cluster assignment step is independent for each data point, making it an ideal candidate for parallelization. Using OpenMP, it is possible to distribute the calculation of the distance between points and centroids over multiple threads, parallelizing the outermost loop.
  One part that adds complexity is updating the sum of coordinates and the number of points relative to the cluster, due to data sharing, can be optimized by using a reduction to sum the contributions of the various threads safely into the relevant arrays, since OpenMP version 4.5 and later reductions per array are supported.
  The previous code will change as follows:

```
#pragma omp parallel default(none) shared(points, centroids,
nPoints, sumX, sumY,k)
  if(omp_get_max_threads() > 1){
    // Assign points to the nearest centroid
    #pragma omp for reduction(+:nPoints,sumX,sumY)
    for (auto& point : points) {
        for (int i = 0; i < k; ++i) {
            ...
```

- **Updating Centroids**: This step can also be optimized using a parallel loop.

```
// Update centroids
#pragma omp for
for (int i = 0; i < k; ++i) {
    if (nPoints[i] > 0) {
        ...
    }
}
```

In this way there are no changes from the parallel code but by exploiting OpenMP it was possible to parallelize the code. Therefore, in the `#pragma omp parallel` directive, an `if(omp_get_max_threads() > 1)` check was added so that the sequential version is executed when the number of threads is 1.

4. Results and Performance Evaluation

4.1. Setup

The tests were conducted on a machine with an Intel Core i7-6700K processor, comprising four cores and eight threads. Due to Hyper-Threading technology, the processor is capable of handling up to eight instruction streams simultaneously. The tests were conducted on data sets of varying sizes with differing numbers of clusters and epochs, contingent on the size of the data set. The objective of the tests is to compare the sequential version, corresponding to the version with a single thread, and the parallel version with a greater number of threads (2, 4, 8, 16). By collating the elapsed times for the disparate configurations, it was possible to ascertain the ***speedup*** and ***efficiency*** parameters:

$$S_p = t_s/t_p; E = S_p/n_{threads}$$

These tests are used to evaluate the performance of parallel versions with a greater number of threads than their sequential counterparts. The results are averaged across five runs.

4.2. Results and Evaluation

The analysis conducted on the runtime results showed a significant improvement in the parallel version compared with the serial version, with performance differences evident depending on the size of the datasets.

Analysis of results

4.2.1 Duration

Represents the average execution time of each configuration. We note that durations increase differently in relation to input data.

Durations tend to increase exponentially for larger data sizes, showing that operations on larger data sets take significantly longer. This pattern suggests that the computational complexity and resource demands escalate with larger inputs, potentially due to factors such as increased memory usage, higher I/O operations, and more intensive processing requirements

Points	Threads	Duration (s)	Speedup	Efficiency
500	1	0.0010	1.00	1.00
500	2	0.0028	0.36	0.18
500	4	0.0036	0.28	0.07
500	8	0.0056	0.18	0.02
500	16	0.0098	0.10	0.01
7000	1	0.0332	1.00	1.00
7000	2	0.0274	1.21	0.61
7000	4	0.0188	1.77	0.44
7000	8	0.0212	1.57	0.20
7000	16	0.0302	1.10	0.07
20000	1	0.1538	1.00	1.00
20000	2	0.1246	1.23	0.62
20000	4	0.0682	2.26	0.56
20000	8	0.0556	2.77	0.35
20000	16	0.0670	2.30	0.14
50000	1	0.6664	1.00	1.00
50000	2	0.4032	1.65	0.83
50000	4	0.2400	2.78	0.69
50000	8	0.1856	3.59	0.45
50000	16	0.2022	3.30	0.21
126000	1	2.3644	1.00	1.00
126000	2	1.3204	1.79	0.90
126000	4	0.7508	3.15	0.79
126000	8	0.5682	4.16	0.52
126000	16	0.5918	4.00	0.25
301487	1	10.8920	1.00	1.00
301487	2	5.9396	1.83	0.92
301487	4	3.3216	3.28	0.82
301487	8	2.3772	4.58	0.57
301487	16	2.5194	4.32	0.27

Table 1. Performance Data

4.2.2 Speedup

1. **Small data set (500 points)**: Performance does not improve with more threads because of the overhead introduced by parallelism. Speedup decreases dramatically as the number of threads increases, highlighting an overhead in handling.

2. **Medium data set (7000 points)**: An increase in speedup is observed up to 4 threads with a sublinear trend, followed by a decrease in speedup with 8 and 16 threads, indicating a limited advantage at a small number of threads.

3. **Large data set size (20000 points)**: Speedup increases up to 8 threads, with a linear trend up to 4 threads and sublinear from 4 to 8 and a subsequent decrease with 16 threads, suggesting saturation.

4. **Very large data set (50000, 126000 and 301487 points)**: The increase in speedup has a linear trend up to 4 threads and continues up to 8 threads with a sublinear trend, confirming the presence of 4 physical cores and 8 logical processors, with a slight decrease at 16 threads. In this case, the trend shows that the performance improvement stabilizes beyond a certain number of threads.

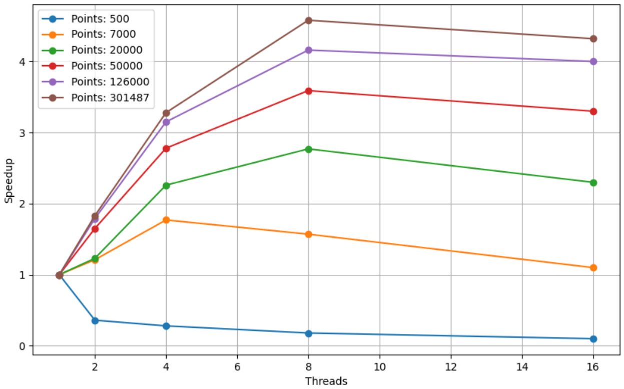

Figure 3. Graph related to speedup by point group and thread

Thus, a clear pattern is observed: the greater the number of points, the greater the ability to exploit threads effectively up to a certain point. Beyond a certain number of threads, speedup stabilizes, suggesting saturation in parallelization operations . This is indicative of Amdahl's law, according to which the advantages of parallelism decrease as the number of threads increases due to the limit determined by the non-parallelisable part of the code. This trend may also due in part to the physical characteristics of the machine on which the experiments were conducted in addition to the parallelism itself.

Most tests show optimal performance gains between 4 and 8 threads, suggesting that to work efficiently on a given workload, it is ideal to find that balance point.

4.2.3 Efficiency

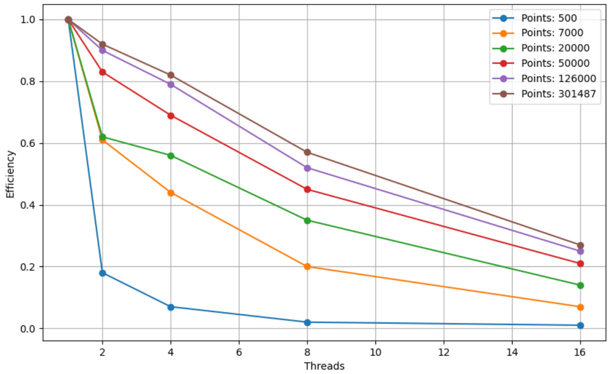

Figure 4. Graph related to efficiency by point group and thread

Efficiency tends to decrease significantly as the number of threads increases, especially for smaller datasets (such as 500 points , reaching only 1% with 16 thread), where thread management can become a bottleneck.
Efficiency is higher for larger datasets initially, but decreases as the number of threads increases. For the largest instances (e.g. 301487), the efficiency drops to 27% with 16 threads, showing that even if the speed-up is relatively high, the ability to scale is not optimal.

5. Conclusions

Parallelization of the K-Means algorithm has proven to be effective, especially with larger datasets. The use of OpenMP has simplified the implementation making performance optimization easier. In summary, the analysis highlights that while parallelization can offer significant performance advantages, it also presents challenges in terms of efficiency.
This suggests that clustering algorithms such as K-Means can benefit significantly from parallelism, paving the way for possible future developments and applications in big data contexts.