UNIVERSITÀ DEGLI STUDI FIRENZE

K-Means Algorithm

Implementation of sequential and parallel version using OpenMP

Marco Trambusti - 7135350

23-11-2024

Introduction

Purpose of the Assignement

- Implement the K-Means algorithm in C++ in both sequential and parallelized versions using OpenMP.
- Dataset: 300147 2D points related to age and total spending of supermarket customers.
- Performance comparison between sequential and parallel version using speedup and efficiency .

K-Means Algorithm

Def.: The K-Means algorithm is a widely used clustering method for partitioning a data set into K distinct clusters

Steps:

1. **Initialization**: Random selection of K initial cluster centers.

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

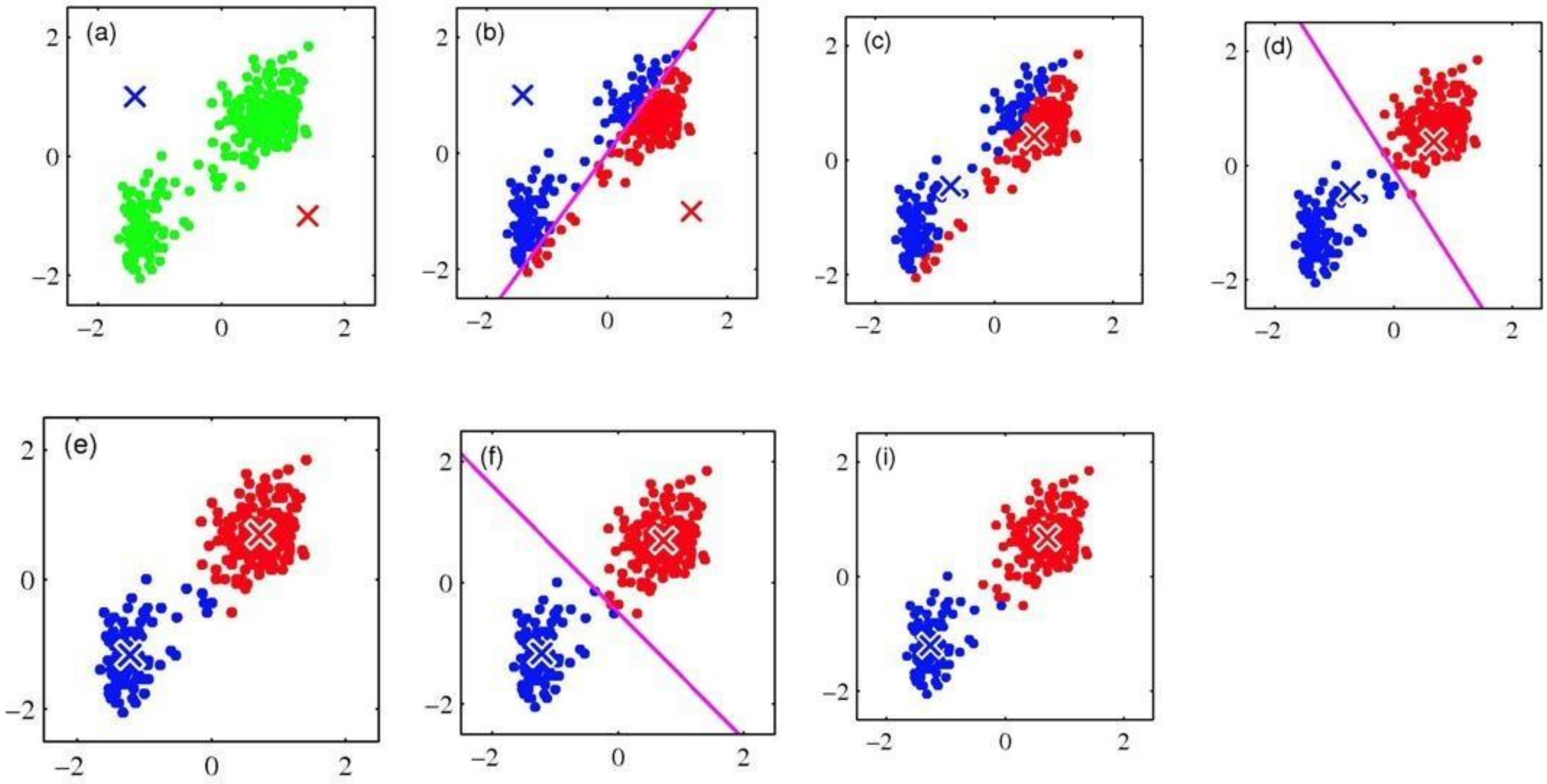
3. **Centroid Update:** Centroids are recalculated as the mean of the points assigned to each cluster.

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

Stopping criterion used: maximum number of iterations.

Algorithm Parameters

- **Number of clusters (K):**
  - Specified a priori
  - Defined in an indicative manner (alternatively could have been used methods like Elbow, Silhouette or Gap Static)
- **Initialization of centroids:**
  - Randomly among the data points
- **Choice of distance metric:**
  - squared Euclidean distance
- **Maximum number of iterations**



Example of K-Means

Sequential Implementation

- The project is in C++
- Chosen OpenMP for the parallel implementation.
- 2D points points are readed from a csv file and normalized
- Clusters are initialized randomly;

The points in the dataset are represented using a struct **Point**, composed by:

  - x, y coordinates (double)
  - cluster (int);
  - minDist (double)
  - distance(Point p) (method)

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.

Sequential Implementation

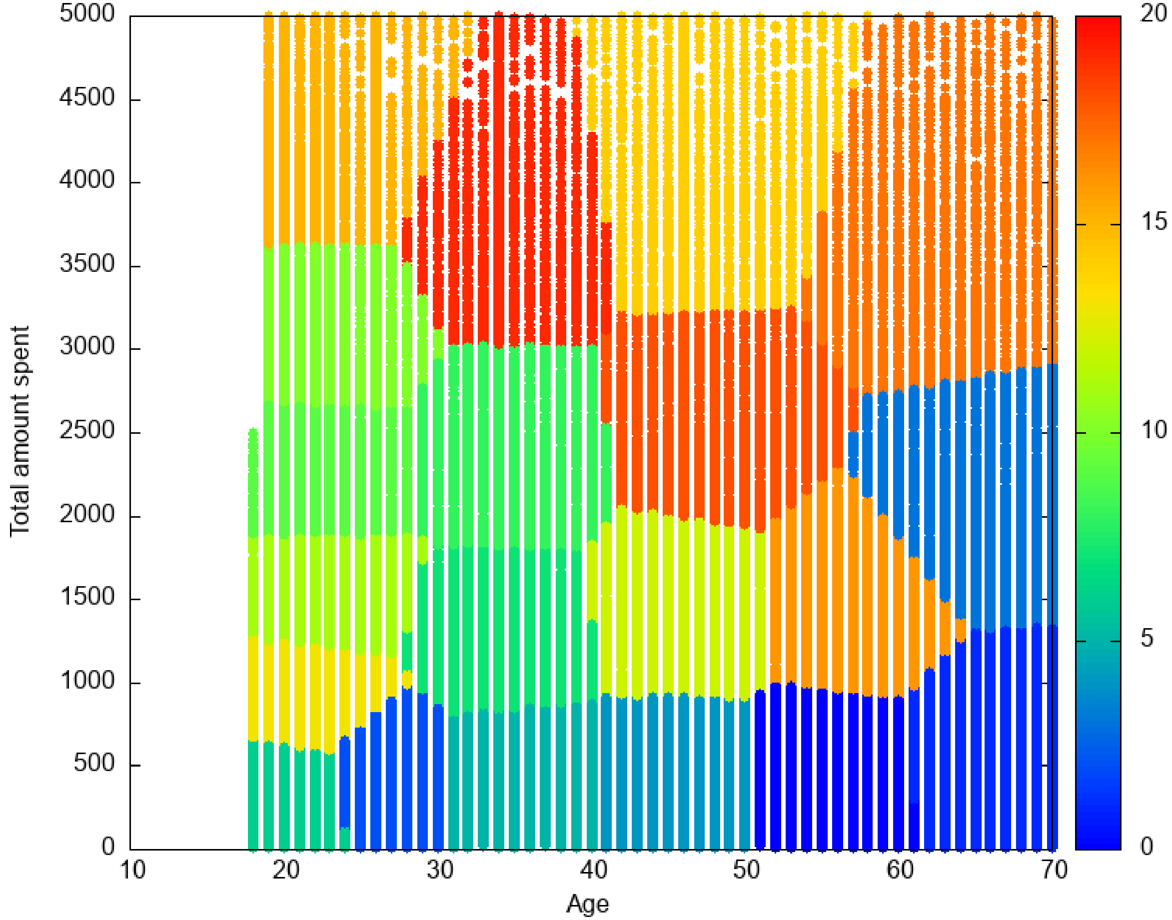
Kmeans method: Cluster assigment part

```
void kmeans(std::vector<Point>& points, std::vector<Point> centroids,
    int k, int epochs) {
    for (int epoch = 0; epoch < epochs; ++epoch) {
        int nPoints[k]= {0};
        double sumX[k] = {0.0};
        double sumY[k] = {0.0};
        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();
        }
        ...
```

Sequential Implementation

Kmeans method: Centroid Update 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;
    }
}
```



Example of results plotted using GnuPlotted

Parallel Implementation

Cluster Assignment

- Cluster assignment step is independent for each data point
  - Ideal candidate for parallelization.
- Using OpenMP, it is possible to distribute the calculation parallelizing the outer loop.
- Updating the sum of coordinates and the number of points relative to the cluster, adds complexity due to data sharing,
  - Optimized using reductions

```
#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) {
            ...
```

Centroids Update

Parallelized using a parallel loop to update centroids.

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

Results and Performance Evaluation

Setup:

- Tests conducted on an Intel Core i7-6700K processor (4 cores and 8 threads). Due to Hyper-Threading technology, the processor is capable of handling up to eight instruction streams simultaneously.
- Data sets of various sizes with a commensurate number of clusters and epochs.
- The results are averaged across five runs.

Results and Performance Evaluation

Duration:

Average execution time of each configuration.

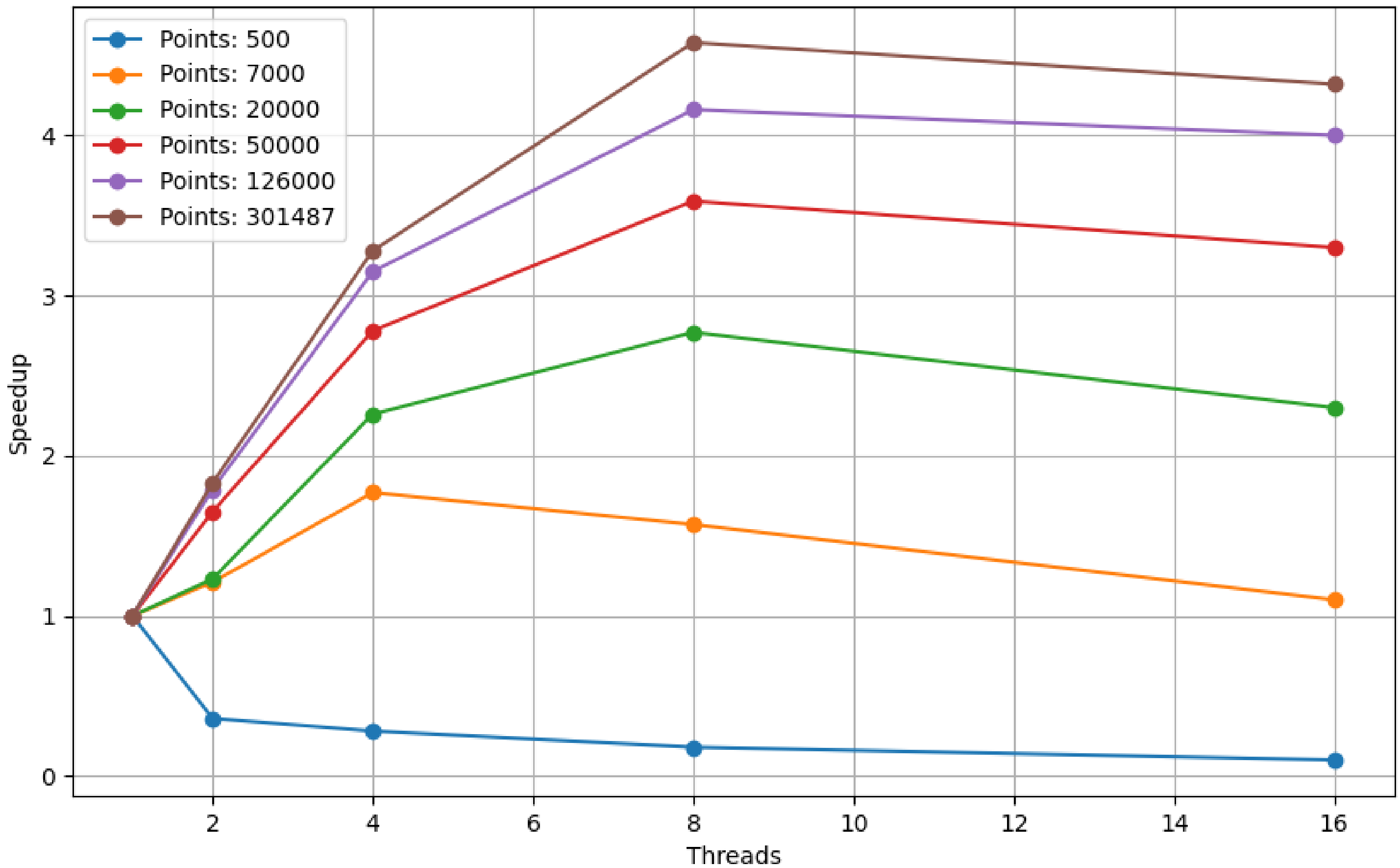
- operations on larger data sets take significantly longer.

Computational complexity and resource demands escalate with larger input.
Causes:

- Increased memory usage
- higher I/O operations
- more intensive processing requirements

Results and Performance Evaluation

Speedup: Ratio of sequential to parallel execution time.



Speedup:

Small data set (500 points):

- Performance does not improve with more threads
- overhead introduced by parallelism.

Medium data set (7000 points):

- Speedup increasing up to 4 threads with a sublinear trend.
- Decrease in speedup with 8 and 16 threads

Large data set size (20000 points): Speedup increases up to 8 threads

- linear trend up to 4
- sublinear from 4 to 8 and a subsequent
- decrease with 16 threads, suggesting saturation

Very large data set (>50000 points):

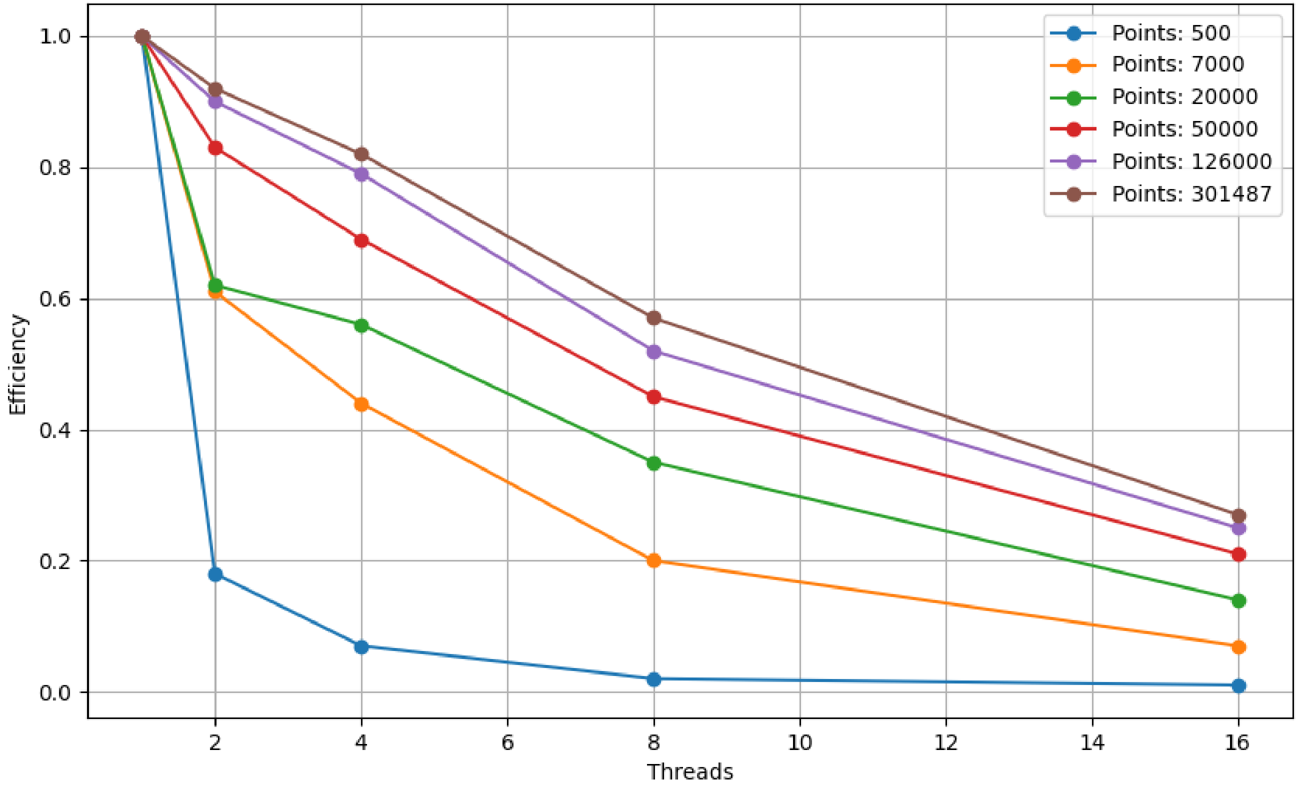
- Speedup has a linear trend up to 4 threads
- continues up to 8 threads with a sublinear trend,
- Slight decrease at 16 threads.

Beyond a certain number of threads, speedup almost stabilizes, suggesting saturation

Most tests show optimal performance gains between 4 and 8 threads

Results and Performance Evaluation

Efficiency: Ratio of Speedup to Number of Threads.



Efficiency:

Smaller datasets:

- Tends to decrease significantly as the number of threads increases, (e.g. with 500 points , reaching only 1% with 16 thread).
- thread management can become a bottleneck.

Larger datasets:

- Higher values initially
- Decreases as the number of threads increases.
- E.g. for 301487 points, the efficiency drops to 27% with 16 threads,
- Even if the speed-up is relatively high, the ability to scale is not optimal

Conclusions

- Parallelizing the K-Means algorithm proved effective, especially with larger datasets.
- Challenges in terms of efficiency.
- OpenMP simplified implementation and performance optimization.
- The larger the number of points, the more effectively threads can be utilized up to a certain point.