K-Means Clustering

Comparison between a sequential
and parallel implementation

Beatrice Paoli

Firenze – 2/2/2023

The Algorithm

K-Means Clustering

The K-means algorithm is a parametrized clustering technique that allows to partition a dataset of points or observations into K clusters.
The algorithm is composed by few steps, with two main phases alternated between each other: the **Assignment** phase and the **Update** phase

Algorithm 1 K-Means Clustering

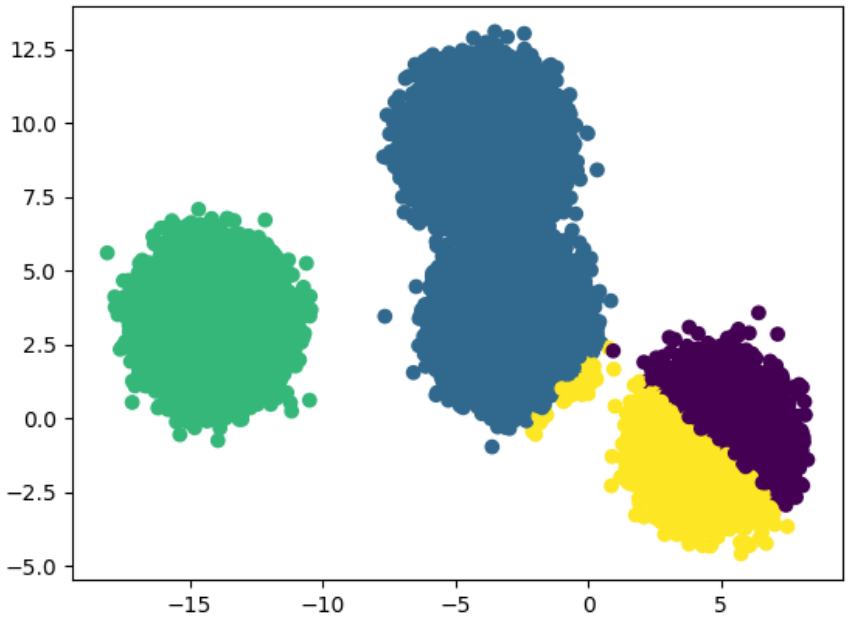
Require: K = number of clusters to create

```
Select K points as initial centroids of the clusters
while Centroids keep changing do
    for each point p do
        Assign p to the cluster with the closest centroid
    end for
    for each cluster c do
        Update the centroid of c
    end for
end while
return clusters
```

The Algorithm

K-Means Clustering

The algorithm converges after few steps to a **local minimum**, depending on the choice of the initial centroids.
In the implementation, the starting centroids are chosen **randomly** from the dataset in input. Therefore, different runs of the algorithm on the same dataset can yield different results.



Implementation

Classes: Point

The AoS (Array of Structure) approach was used.

```
class Point {
public:
    Point(): x(0), y(0), clusterId(-1) {};
    Point(float x, float y): x(x), y(y), clusterId(-1) {};

    float dist(const Point &p) const;

    float x;
    float y;
    int clusterId;
};
```

Implementation

Classes: Cluster

The AoS (Array of Structure) approach was used.

```
class Cluster {
public:
    explicit Cluster(int id, Point centroid);

    void addPoint(Point point);
    void updateCentroid();

    int id;
    Point centroid;

private:
    float tempSumX;
    float tempSumY;
    int size;
};
```

```
void Cluster::addPoint(Point point) {
    this->tempSumX += point.x;
    this->tempSumY += point.y;
    this->size++;
}
```

```
void Cluster::updateCentroid() {
    this->centroid.x = this->tempSumX / this->size;
    this->centroid.y = this->tempSumY / this->size;
    this->tempSumX = 0;
    this->tempSumY = 0;
    this->size = 0;
}
```

Implementation

Function

```
std::vector<Cluster> kMeansClustering(int k, std::vector<Point> &points, int maxIters) {

    std::vector<Cluster> clusters = randomInit(k, points);
    int iter = 0;
    bool updateStopped;
    const int pointsSize = points.size();
    const int clusterSize = clusters.size();

    do {
        iter++;
        updateStopped = true;

        // Assignment
        for (int i = 0; i < pointsSize; i++) {...}

        // Update
        for (int i = 0; i < clusterSize; i++) {
            clusters[i].updateCentroid();
        }
    } while (!updateStopped && iter < maxIters);

    return clusters;
}
```

Implementation

Initialization

```
std::vector<Cluster> randomInit(int k, std::vector<Point> &points) {
    std::vector<Cluster> clusters;
    clusters.reserve(k);

//    std::random_device rd;
//    srand(rd());
    srand(0);
    for (int i = 0; i < k; i++) {
        clusters.emplace_back(i, points[rand() % points.size()]);
    }

    return clusters;
}
```

Implementation

Assignment Step

```
// Assignment
for (int i = 0; i < pointsSize; i++) {
    Point *point = &points[i];

    float currentMinDist = std::numeric_limits<float>::max();
    int currentMinClusterId = -1;

    for (int j = 0; j < clusterSize; j++) {
        Cluster *cluster = &clusters[j];
        float dist = point->dist(cluster->centroid);
        if (dist < currentMinDist) {
            currentMinDist = dist;
            currentMinClusterId = cluster->id;
        }
    }

    if (point->clusterId != currentMinClusterId) {
        point->clusterId = currentMinClusterId;
        updateStopped = false;
    }
    clusters[currentMinClusterId].addPoint(*point);
}
```

Parallelization with OpenMP

Assignment Step Parallelization

To achieve parallelization, it was necessary to use only a few pragma statements.

The assignment step is embarrassingly parallel since the search and assignment of each point to a cluster is independent from one another. So, a `#pragma omp parallel for` statement was used on the points loop to use threads to parallelize the computation.
Since the workload for each thread is similar the default static scheduler is used.

```
        // Assignment
#pragma omp parallel for default(none) firstprivate(pointsSize, clusterSize) shared(updateStopped, points, clusters)
        for (int i = 0; i < pointsSize; i++) {
            Point *point = &points[i];

            float currentMinDist = std::numeric_limits<float>::max();
            int currentMinClusterId = -1;

            for (int j = 0; j < clusterSize; j++) {
```

Parallelization with OpenMP

Assignment Step Parallelization

However, each thread accesses the shared clusters to call `addPoint()` and update the internal cluster variable. Without any synchronization mechanism, this introduces a **race condition** on the algorithm that leads to incorrect results.

The method `addPoint()` only performs three sums over the variables `tempSumX`, `tempSumY` and `size`, so three `#pragma omp atomic` were used to ensure the correct update of the variables and avoid the race condition.

```
void Cluster::addPoint(Point point) {
#pragma omp atomic
    this->tempSumX += point.x;
#pragma omp atomic
    this->tempSumY += point.y;
#pragma omp atomic
    this->size++;
}
```

Speedup Analysis

Definition

In order to compare the performance of a sequential algorithm with its parallel version we can use the concept of **speedup**.

Speedup is measured as the ratio between the execution time of the sequential algorithm and the parallel one.

$$S = \frac{t_s}{t_p}$$

Ideally, we should look for *perfect speedup* or even *linear speedup*, meaning S should be equal or similar to the number of processors used to perform the parallel algorithm.

Speedup Analysis

Speedup results with different K and N

The experiments were performed on a Intel Core i7-1165G7 with 8 logical cores.

Since the algorithm has the hyperparameter K for the number of clusters, different experiments were performed with varying K and fixed N (number of points), and viceversa.

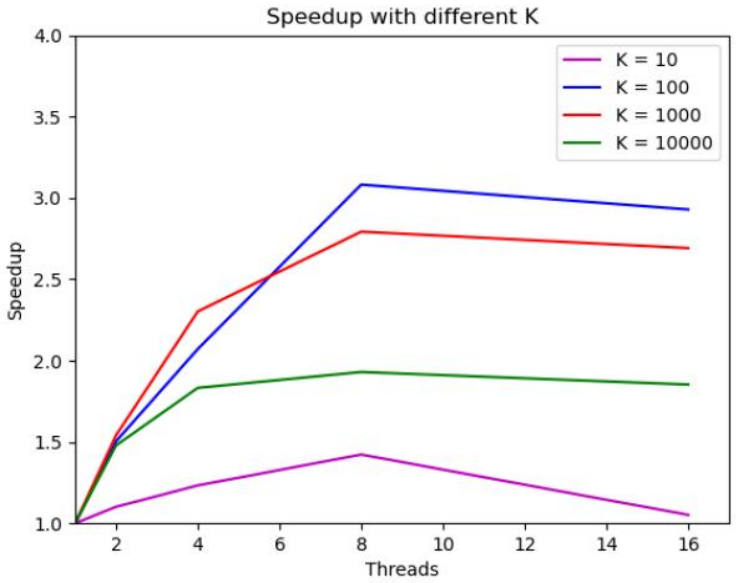


Figure 1. Speedup K Means Clustering con N = 100000.

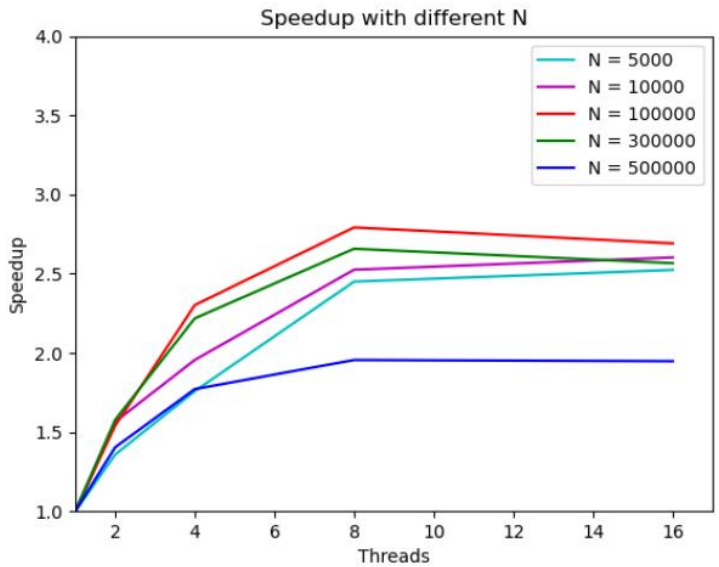


Figure 2. Speedup K Means Clustering con K = 1000.

Speedup Analysis

Execution Times

Threads	Execution Time (s)	Speedup
1	0.213	1
2	0.194	1.101
4	0.173	1.232
8	0.150	1.422
16	0.203	1.052

Table 1. Execution Times for K = 10 and N = 100000.

Threads	Execution Time (s)	Speedup
1	1.458	1
2	0.970	1.504
4	0.704	2.071
8	0.473	3.082
16	0.498	2.929

Table 2. Execution Times for K = 100 and N = 100000.

Threads	Execution Time (s)	Speedup
1	12.106	1
2	7.860	1.540
4	5.259	2.302
8	4.335	2.792
16	4.498	2.691

Table 3. Execution Times for K = 1000 and N = 100000.

Threads	Execution Time (s)	Speedup
1	126.525	1
2	85.591	1.478
4	69.055	1.832
8	65.570	1.930
16	68.286	1.853

Table 4. Execution Times for K = 10000 and N = 100000.

Speedup Analysis

Execution Times

Threads	Execution Time (s)	Speedup
1	0.610	1
2	0.449	1.358
4	0.347	1.759
8	0.249	2.450
16	0.242	2.524

Table 5. Execution Times for K = 1000 and N = 5000.

Threads	Execution Time (s)	Speedup
1	1.293	1
2	0.827	1.565
4	0.661	1.955
8	0.512	2.525
16	0.497	2.602

Table 6. Execution Times for K = 1000 and N = 10000.

Threads	Execution Time (s)	Speedup
1	38.045	1
2	24.122	1.577
4	17.152	2.218
8	14.315	2.658
16	14.824	2.567

Table 7. Execution Times for K = 1000 and N = 300000.

Threads	Execution Time (s)	Speedup
1	63.696	1
2	45.375	1.404
4	35.962	1.771
8	32.577	1.955
16	32.705	1.948

Table 8. Execution Times for K = 1000 and N = 500000.