UNIVERSITÀ DEGLI STUDI FIRENZE

OpenMP K-Means Clustering

Mattia Bacci 7060637
Elaborato Mid-Term

13/01/2021

K-Means

The algorithm consist in few steps:

1. Generate N pseudo-random Point in the bidimensional space(with finite dimension).
2. Generate K pseudo-random centroids for the K Clusters.
3. Compute for each Point the euclidean distance to all the Cluster.
4. Assign every Point to the nearest Cluster.
5. Update the centroid position as the average of the positions of the points inside the Cluster.
6. Repeat from 3 to 5 until convergenze or interrupt condition is satisfied.

Parameters

- Numbers of **Point** and **Clusters.**
- **Maximum range**: the dimension of the square where the point will be generated, fixed to 100000.
- **Maximum Iteration**, fixed to 100.
- **Parallel:** active or deactivate parallelism.
- **Animation:** activate or deactivate animations.

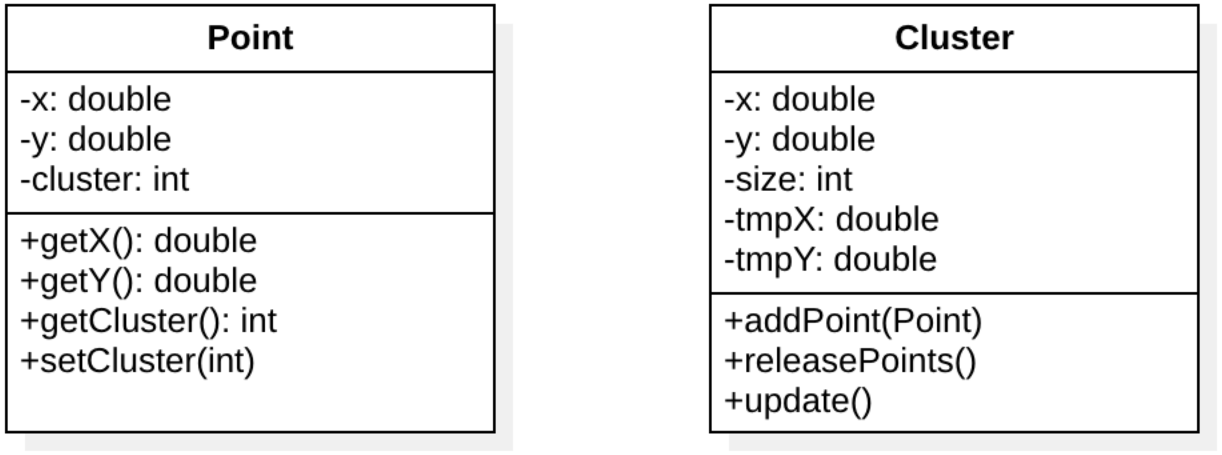
Convergence

The algorithm converge very quickly but there is **no guarantee to find the optimal solution**. We must provide the algorithm with a converge condition. There are a lot of condition to establish convergence:

- Centroids doesn’t change their position
- No Point change Cluster.
- The sum of the distances is reduced to the minimum.

First condition was chosen, a centroid doesn’t move if the differences between his position before and after the update is less than 10^{-1}

UML


Point
-x: double
-y: double
-cluster: int
+getX(): double
+getY(): double
+getCluster(): int
+setCluster(int)
Cluster
-x: double
-y: double
-size: int
-tmpX: double
-tmpY: double
+addPoint(Point)
+releasePoints()
+update()

Sequential Version

```
void kMeansIteration(std::vector<Point> &points, std::vector<Cluster> &clusters) {
    int clusterIndex;
    double minDist;

    for (int i = 0; i < POINTS_NUMBER; i++) {
        Point &point = points[i];
        clusterIndex = 0;
        minDist = distance(point, clusters[0]);

        for (int j = 1; j < CLUSTERS_NUMBER; j++) {
            Cluster &cluster = clusters[j];
            double dist = distance(point, cluster);
            if (dist < minDist) {
                minDist = dist;
                clusterIndex = j;
            }
        }
        clusters[clusterIndex].addPoint(points[i]);
        points[i].setCluster(clusterIndex);
    }
}
```

Parallel Version

```
void kMeansIterationParallel(std::vector<Point> &points, std::vector<Cluster> &clusters) {
    int clusterIndex;
    double minDist;

#pragma omp parallel default(shared) private(minDist, clusterIndex)
    {
#pragma omp for schedule(static)
        for (int i = 0; i < POINTS_NUMBER; i++) {
            Point &point = points[i];
            clusterIndex = 0;
            minDist = distance(point, clusters[0]);

            for (int j = 1; j < CLUSTERS_NUMBER; j++) {
                Cluster &cluster = clusters[j];
                double dist = distance(point, cluster);
                if (dist < minDist) {
                    minDist = dist;
                    clusterIndex = j;
                }
            }
            clusters[clusterIndex].addPoint(points[i]);
            points[i].setCluster(clusterIndex);
        }
    }
}
```

Parallel Version

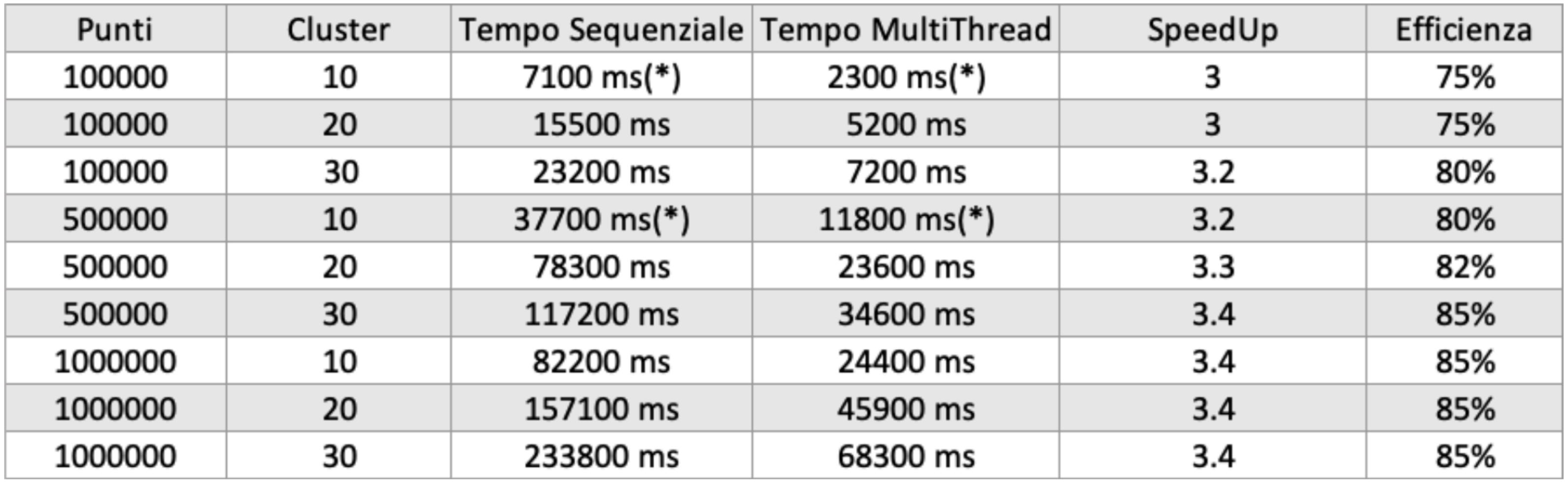
```cpp
void kMeansIterationParallel(std::vector<Point> &points, std::vector<Cluster> &clusters) {
    int clusterIndex;
    double minDist;

#pragma omp parallel default(shared) private(minDist, clusterIndex)
    {
#pragma omp for schedule(static)
        for (int i = 0; i < POINTS_NUMBER; i++) {
            Point &point = points[i];
            clusterIndex = 0;
            minDist = distance(point, clusters[0]);

            for (int j = 1; j < CLUSTERS_NUMBER; j++) {
                Cluster &cluster = clusters[j];
                double dist = distance(point, cluster);
                if (dist < minDist) {
                    minDist = dist;
                    clusterIndex = j;
                }
            }
            clusters[clusterIndex].addPoint(points[i]);
            points[i].setCluster(clusterIndex);
        }
    }
}
```

```cpp
    void addPoint(const Point &point) {
#pragma omp atomic
        tmpX += point.getX();
#pragma omp atomic
        tmpY += point.getY();
#pragma omp atomic
        size++;
    }
```

Parallel Sections

```
#pragma omp parallel
    {
#pragma omp sections
        {
#pragma omp section
            {
                points = initPoints(POINTS_NUMBER);
            }
#pragma omp section
            {
                clusters = initCluster(CLUSTERS_NUMBER);
            }
        }
    }
```

Results



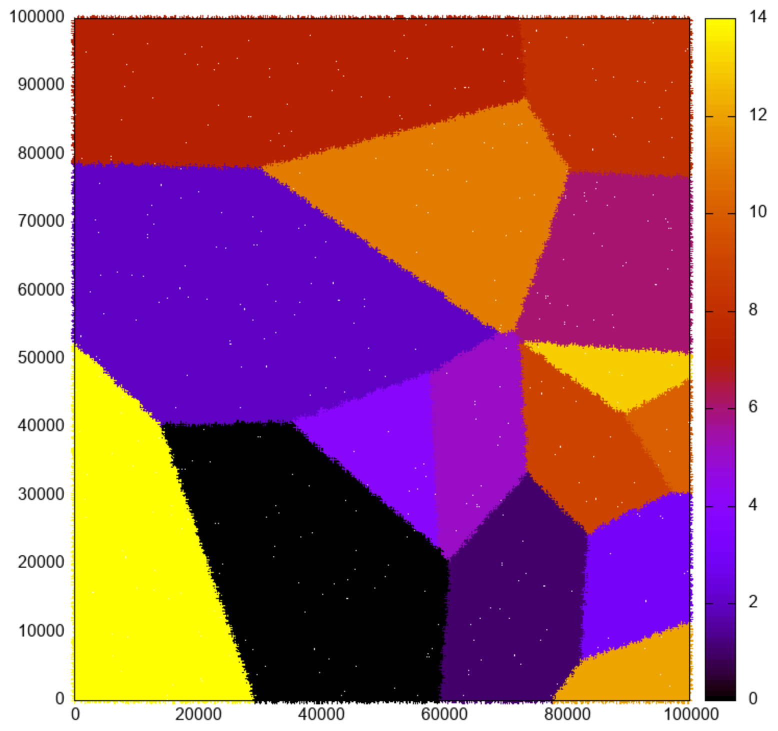
Punti	Cluster	Tempo Sequenziale	Tempo MultiThread	SpeedUp	Efficienza
100000	10	7100 ms(*)	2300 ms(*)	3	75%
100000	20	15500 ms	5200 ms	3	75%
100000	30	23200 ms	7200 ms	3.2	80%
500000	10	37700 ms(*)	11800 ms(*)	3.2	80%
500000	20	78300 ms	23600 ms	3.3	82%
500000	30	117200 ms	34600 ms	3.4	85%
1000000	10	82200 ms	24400 ms	3.4	85%
1000000	20	157100 ms	45900 ms	3.4	85%
1000000	30	233800 ms	68300 ms	3.4	85%

Figure 3. Intel Core i5 (I5-4570S) 4-Core

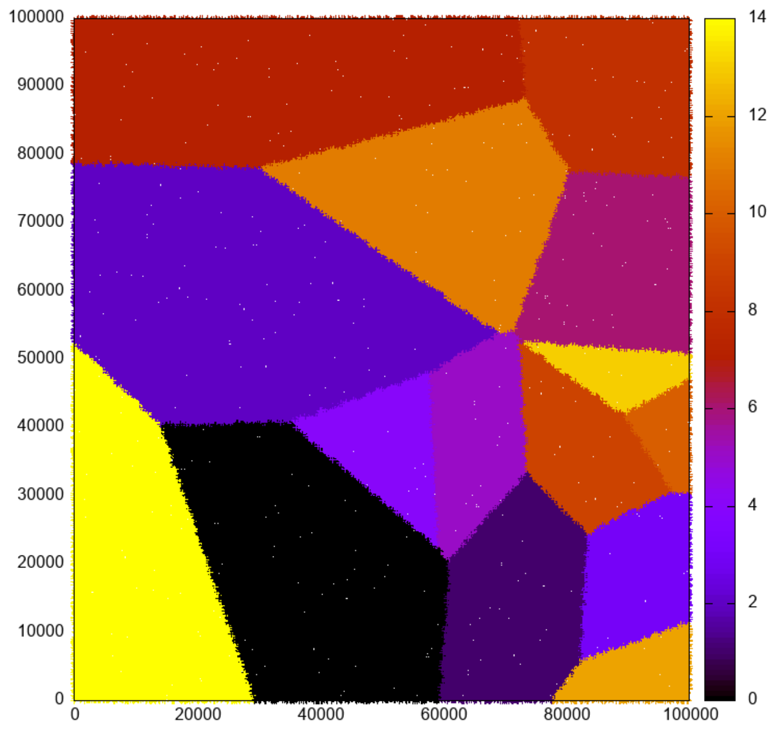
(*) – Convergence reached

GnuPlot and Animation

If the ANIMATION flag is set, the software will build a gif with its run time equal to the time needed by the algorithm.



Parallel – 7s



Sequential – 24s

*see images folder if the animation is not playing