Parallelising K-means Algorithm

with OpenMP

Ernesto Palchetti

21 luglio 2025

Table of Contents

1 Introduction

2 K-means

3 Serial Implementation

4 Parallel Implementation

5 Experiments

6 Conclusions

Section 1

Introduction

Our Goal

The goal is to implement two version of the **K-Means** algorithm.

Our Goal

The goal is to implement two version of the **K-Means** algorithm.

- **serial** implementation in C++

Our Goal

The goal is to implement two version of the **K-Means** algorithm.

- **serial** implementation in C++
- **parallel** implementation in C++ with OpenMP

Our Goal

The goal is to implement two version of the **K-Means** algorithm.

- **serial** implementation in C++
- **parallel** implementation in C++ with OpenMP

The aim is to compare them in terms of **speedup**.

$$\text{speedup} := \frac{\text{serial time}}{\text{parallel time with} p \text{threads}} \tag{1}$$

Amdahl's Law

Robey and Zamora [2, p. 11-12]

For fixed-size problems

$$SpeedUp(N) = \frac{1}{S + \frac{P}{N}}, \qquad P + S = 1. \tag{2}$$

Gustafson and Barsis's Law

Robey and Zamora [2, p. 12-13]

$$SpeedUp(N) = N - S(N-1), \tag{3}$$

when the size of the problem grows proportionally to the number of processors.

Section 2

K-means

K-means Algorithm

- **Clustering** algorithm

K-means Algorithm

- **Clustering** algorithm
- **Quantitative** variables

K-means Algorithm

- **Clustering** algorithm
- **Quantitative** variables
- **distance-based** clustering

K-means Algorithm

- **Clustering** algorithm
- **Quantitative** variables
- **distance-based** clustering
- k centroids

K-means Algorithm

- **Clustering** algorithm
- **Quantitative** variables
- **distance-based** clustering
- k centroids
- **Convergence** criterion

Centroids Initialization

We select k random points of the dataset

$$\boldsymbol{c}_j = \boldsymbol{x}_{i_j}, \quad i_1, ..., i_k \in \{1, ..., N\} \tag{4}$$

Centroids Initialization

We select k random points of the dataset

$$\boldsymbol{c}_j = \boldsymbol{x}_{i_j}, \quad i_1, ..., i_k \in \{1, ..., N\} \tag{4}$$

Other option: we can assign random points to clusters and skip the first step, or set them according to previous analysis.

Compute distances

We compute the Euclidean distance between every point and every centroid.

$$d(\boldsymbol{x}_i, \boldsymbol{c}_j) = \sqrt{\sum_{l=1}^{D} \left(\boldsymbol{x}_{i\,l} - \boldsymbol{c}_{j\,l}\right)^2} \tag{5}$$

Compute distances

We compute the Euclidean distance between every point and every centroid.

$$d(\boldsymbol{x}_i, \boldsymbol{c}_j) = \sqrt{\sum_{l=1}^{D} \left(\boldsymbol{x}_{i\,l} - \boldsymbol{c}_{j\,l}\right)^2} \tag{5}$$

We can use different distances (see Chapter 14 of Hastie, Tibshirani, and Friedman (2009) [1]).

Cluster assignment

We assign a point to the cluster represented by the closest centroid.

$$\boldsymbol{x}_i \in \mathcal{C}_j \iff j = \operatorname*{arg\,min}_{t=1,\ldots k} d(\boldsymbol{x}_i - \boldsymbol{c}_t) \tag{6}$$

New Centroids

New centroids are computed as the average point of every cluster

$$\boldsymbol{c}_j = \frac{1}{|\{i : \boldsymbol{x}_i \in \boldsymbol{c}_j\}|} \sum_{i=1}^{N} \boldsymbol{x}_i \mathbb{I}_{[\boldsymbol{x}_i \in \boldsymbol{c}_j]} \tag{7}$$

Section 3

Serial Implementation

Centroids Selection

```
int p;
for (int i = 0; i < k; ++i) {
    p=distrib(gen);
    C[i][0]=v[p][0];
    C[i][1]=v[p][1];
}
```

Choose k centroids by randomly selecting them from the dataset.

Number of iterations

```
int soglia=0.01*N;
//int soglia=0.01*N;
while (change > soglia){
```

The algorithm stops when the number of points that change classification after one step is less than 1% of N,

Reset counters

```
for (j=0; j<k; ++j) {
    sums[j][0]=0.0;
    sums[j][1]=0.0;
    contatori[j]=0;
}
change = 0;
```

At every step, we reset to 0 every counter and partial sum.

Distances

```
for (i=0;i<N;i++) { // loop on every point
    dmin=distanza_punto_punto(v[i][0],C[0][0],v[i][1],C[0][1]);
    kmin=0;
        d=distanza_punto_punto(v[i][0],C[j][0],v[i][1],C[j][1]);
        if (d<dmin) {
            dmin=d;
            kmin=j;
        }

    }
```

We compute all distances, storing the index of the closest centroid.

Classification

```
    if (ass[i]!=kmin) {
        change++;
        ass[i]=kmin;
    }
    sums[kmin][0]+=v[i][0];
    sums[kmin][1]+=v[i][1];
    contatori[kmin]++;
}
```

We assign the point to the cluster of the closest centroid. Then we update change counter, the counter of points in the relative cluster, and a sum containing all points in that cluster.

Computing new centroids

```
for (j=0;j<k;j++) {
    C[j][0]=sums[j][0]/contatori[j];
    C[j][1]=sums[j][1]/contatori[j];

}
```

We compute new centroids by dividing the sums by the counters.

Section 4

Parallel Implementation

Centroids Selection

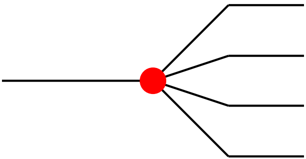
```
int p;
for (int i = 0; i < k; ++i) {
    p=distrib(gen);
    C[i][0]=v[p][0];
    C[i][1]=v[p][1];
}
int change=N;
int soglia=0.01*N;
```

Choose k centroids by randomly selecting them from the dataset in a serial way. Initialize the shared threshold and counter.

Creating threads and private variables

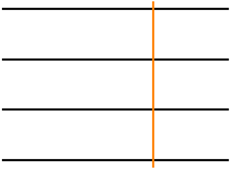
Robey and Zamora [2] Chapter 7

```
#pragma omp parallel private(d,dmin,kmin) shared(change)
        num_threads(threads_number)
    int changet=0;
    vector<int> contatorit(k);
    vector<vector<double>> sumst(k,vector<double>(2));
```



Create threads_number threads and define private counters and sums.

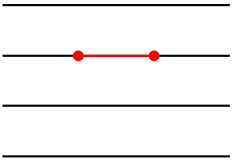
```
while (change > soglia) {
    changet=0;
    for (int j=0;j<k;j++) {
        contatorit[j]=0;
        sumst[j][0]=0.0;
        sumst[j][1]=0.0;
        }
#pragma omp barrier
```



Start `while` loop and initialize private counters. Make sure every thread enters the loop.

Resetting shared counter

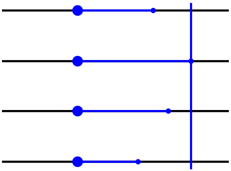
```
#pragma omp single
    change = 0;
```



A single core changes the value of the counter.

Resetting shared counters

```
#pragma omp for
    for (int j=0; j<k;j++) {
        sums[j][0]=0.0;
        sums[j][1]=0.0;
        contatori[j]=0;
    }
```



Every thread resets a portion of shared counters (no risk of race conditions)

Parallel Loop over points

```
#pragma omp for nowait
for (int i=0;i<N;i++) {
    dmin=distanza_punto_punto(v[i][0],C[0][0],v[i][1],C[0][1
    kmin=0;
    for (int j=1;j<k;j++) {
        d=distanza_punto_punto(v[i][0],C[j][0],v[i][1],C[j][
        if (d<dmin) {  minimum
            dmin=d;
            kmin=j;
        }
    }
```

Every thread computes distances between points and centroids, equally splitting points to treat.

Parallel Loop over points

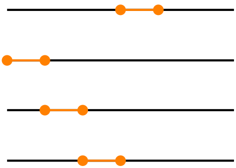
```
if (ass[i]!=kmin) {
        changet++;
        ass[i]=kmin;
    }
    sumst[kmin][0]+=v[i][0];
    sumst[kmin][1]+=v[i][1];
    contatorit[kmin]++;  points
}
```

If the point is assigned to a different cluster, the counter is updated. Then, the private (no race condition) counter and sum for the relative cluster are updated.

Assignment

```
#pragma omp critical
    {
    change+=changet;
    for (int j=0;j<k;j++) {
    sums[j][0]+=sumst[j][0];
    sums[j][1]+=sumst[j][1];
    contatori[j]+=contatorit[j];
    }
    }
```

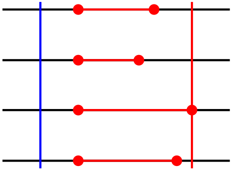
In a critical section, every thread updates shared sums and counters one at a time, avoiding a race condition.



Re-center clusters

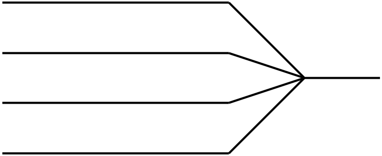
```
#pragma omp barrier
#pragma omp for
    #pragma omp for
    for (int i=0;i<k;i++) {
        C[i][0]=sums[i][0]/contatori[i];
        C[i][1]=sums[i][1]/contatori[i];
    }
```

After a barrier to ensure that every counter is ready, centroids are computed in a parallel for.



Join Threads

At the end of the `parallel` section, threads join.



Section 5

Experiments

Tests

The experiments consist of two phases:

Tests

The experiments consist of two phases:

- I applied the two algorithm on a Kaggle dataset of 21600 2D points;

Tests

The experiments consist of two phases:

- I applied the two algorithm on a Kaggle dataset of 21600 2D points;
- I applied the two algorithms on 9 randomly generated datasets with different values of N and k.

Tests

The experiments consist of two phases:

- I applied the two algorithm on a Kaggle dataset of 21600 2D points;
- I applied the two algorithms on 9 randomly generated datasets with different values of N and k.

For both, I computed **times** for serial and parallel algorithms, and I evaluated the **speedup**, checking that the resulting clusters are the same. The analysis is performed in RStudio.

Kaggle Example I

For this experiment, I applied both algorithms, setting a varying number of threads

$$n_{threads} = 2^p, \qquad p = 0, 1, ..., 8. \tag{8}$$

I repeated every test $N_{rep} = 100$ times.

Kaggle Example II

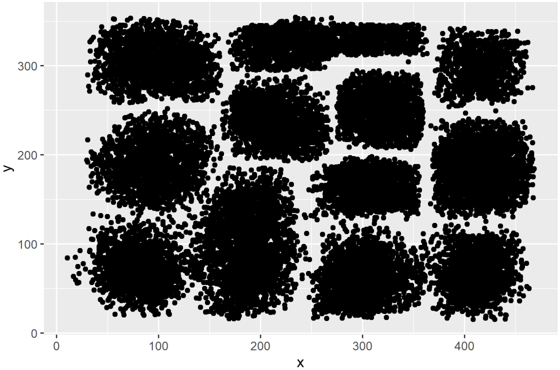


Figura 1: Input data of the experiment.

Kaggle Example III

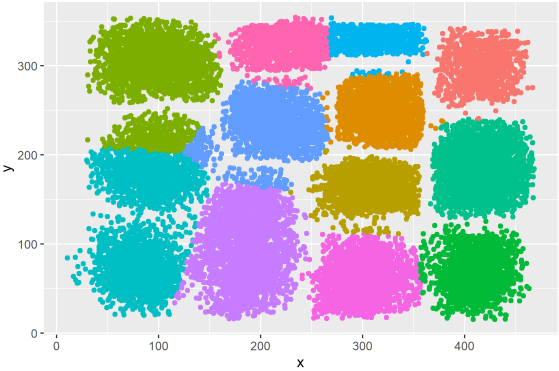


Figura 2: Output clusters of the experiment.

Kaggle Example IV

Threads	Mean Speedup	Lower	Upper
1	1.00	0.988	1.01
2	1.70	1.70	1.73
4	2.76	2.74	2.79
8	2.99	2.96	3.01
16	2.79	2.77	2.81
64	2.43	2.42	2.45
128	1.99	1.97	2.00
256	1.41	1.40	1.41

Tabella 1: Speedups for the Kaggle dataset.

Kaggle Example V

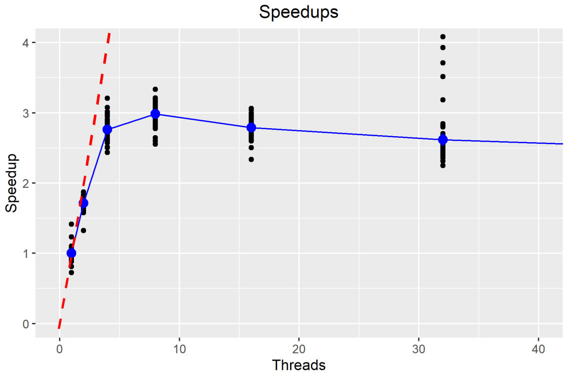


Figura 3: Speedups of the experiments of the Kaggle dataset.

Random Datasets I

I repeated 10 tests for every combination of $N = 10^4,\ 10^5$, and 10^6, and $k = 10,\ 20$ and 40.

Random Datasets II

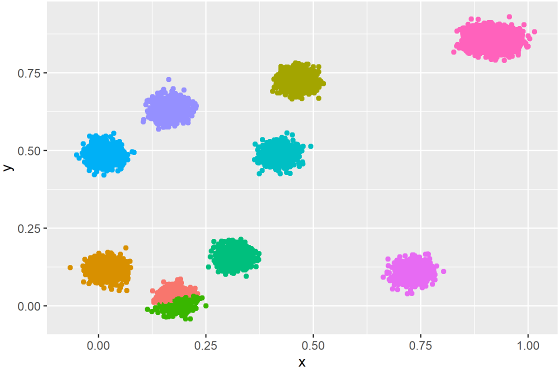


Figura 4: Randomly generated dataset and relative clusters for $k = 10$ and $N = 10^4$.

Random Datasets III

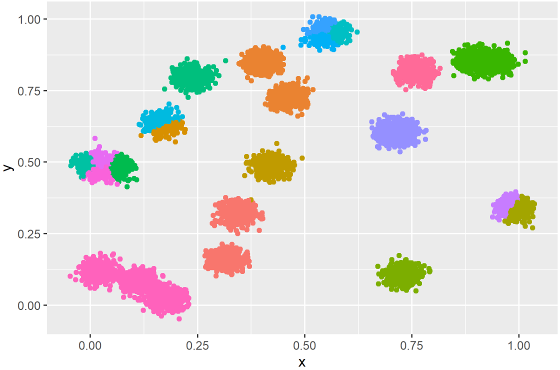


Figura 5: Randomly generated dataset and relative clusters for $k = 20$ and $N = 10^4$.

Random Datasets IV

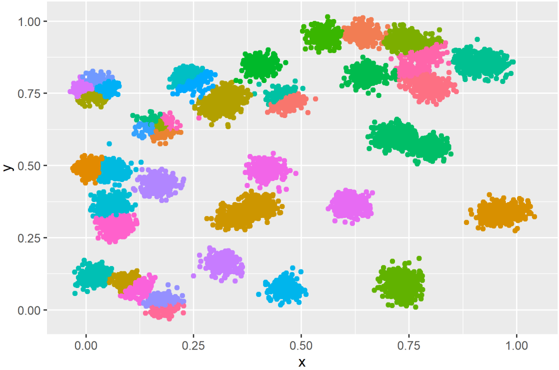


Figura 6: Randomly generated dataset and relative clusters for $k = 40$ and $N = 10^4$.

Random Datasets V

Threads	$N = 10^4$	$N = 10^5$	$N = 10^6$
1	0.974	1.00	0.994
2	1.76	1.44	1.70
4	2.21	2.07	2.49
8	2.65	3.38	3.00
16	2.69	3.30	2.92
32	2.47	3.24	2.96
64	1.92	3.29	3.00
128	1.36	2.96	3.04
256	0.768	2.59	3.25

Tabella 2: Speedups for $K = 10$.

Random Datasets VI

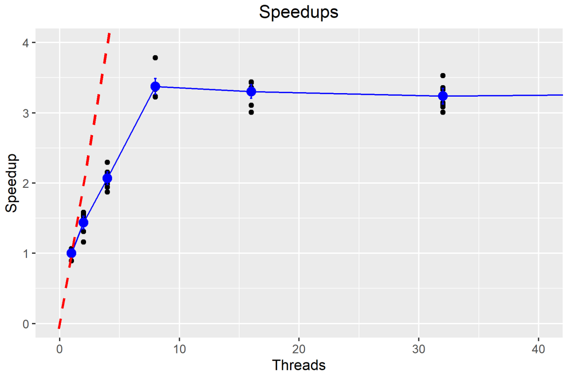


Figura 7: Speedups for $N = 10^5$ for $k = 10$.

Random Datasets VII

Threads	$N = 10^4$	$N = 10^5$	$N = 10^6$
1	1.01	1.00	0.982
2	1.84	1.73	1.71
4	2.70	2.57	2.52
8	3.01	3.18	2.95
16	2.55	3.58	2.89
32	2.63	3.02	2.90
64	2.18	3.03	2.92
128	1.64	2.87	2.91
256	1.22	2.63	2.88

Tabella 3: Speedups for $K = 20$.

Random Datasets VIII

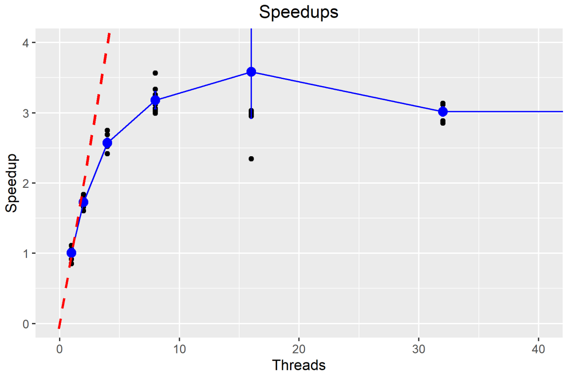


Figura 8: Speedups for $N = 10^5$ for $k = 20$.

Random Datasets IX

Threads	$N = 10^4$	$N = 10^5$	$N = 10^6$
1	1.01	1.00	1.01
2	1.70	1.73	1.71
4	2.08	2.58	2.54
8	3.33	2.98	2.92
16	3.37	2.90	2.91
32	3.09	2.94	2.91
64	2.97	2.93	2.90
128	2.41	2.88	2.90
256	1.85	2.77	2.89

Tabella 4: Speedups for $K = 40$.

Random Datasets X

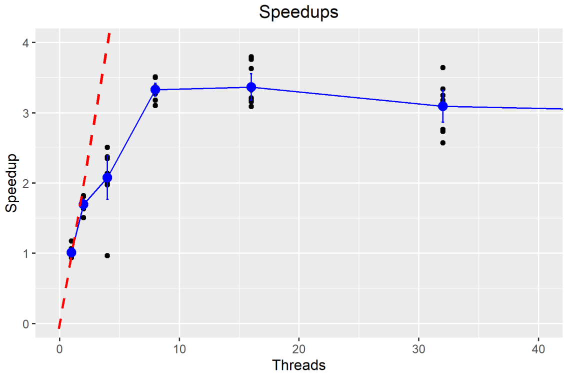


Figura 9: Speedups for $N = 10^4$ for $k = 40$.

Section 6

Conclusions

Final considerations

- The speedup I obtained is good (not excellent);

Final considerations

- The speedup I obtained is good (not excellent);
- it increases with the dimensions of the dataset;

Final considerations

- The speedup I obtained is good (not excellent);
- it increases with the dimensions of the dataset;
- The management cost of threads has to be balanced.

Final considerations

- The speedup I obtained is good (not excellent);
- it increases with the dimensions of the dataset;
- The management cost of threads has to be balanced.

I can reach better results with a better distribution of data in memory using the **first touch** principle.

References

T. Hastie, R. Tibshirani, and J. Friedman.
The elements of statistical learning: data mining, inference, and prediction.
Springer series in statistics. New York, 2nd ed edition, 2009.

Robert Robey and Y. Zamora.
Parallel and High Performance Computing.
Manning Publications Co. LLC, New York, 2021.