# Parallelising K-means Algorithm

## with Multiprocessing

Ernesto Palchetti

21 luglio 2025

# Table of Contents

# Section 1

# Introduction

# Intro

This project aims to implement and compare a **serial** and a **parallel** version of the K-means algorithm, implemented in **Python** and with the package *multiprocessing*.

# Intro

This project aims to implement and compare a **serial** and a **parallel** version of the K-means algorithm, implemented in **Python** and with the package *multiprocessing*.
The metric of comparison is the **speedup**, that is

$$S_p = \frac{t_s}{t_p}. \tag{1}$$

# Section 2

# K-means

# Introduction to the algorithm

Hastie, Tibshirani and Friedman (2009)[Hastie et al., 2009]

K-means is one of the most common **clustering** algorithm.
It is characterized by a parameter $k$ (the number of clusters), a set of $k$ centroids, and a **distance-based** clustering criterion.

# Notation

The dataset is

$$S = (\boldsymbol{x}_i)_{i=1}^{N}\,, \tag{2}$$

(3)

(4)

# Notation

The dataset is

$$S = (\boldsymbol{x}_i)_{i=1}^{N}, \tag{2}$$

$$\boldsymbol{x}_i \in \mathbb{R}^D. \tag{3}$$

$$\tag{4}$$

# Notation

The dataset is

$$S = (\boldsymbol{x}_i)_{i=1}^{N}, \tag{2}$$

$$\boldsymbol{x}_i \in \mathbb{R}^D. \tag{3}$$

$$\boldsymbol{c}_j \in \mathbb{R}^D, \quad j = 1, ..., k. \tag{4}$$

# Algorithm

K-Means consists of two steps:

(5)

(6)

# Algorithm

K-Means consists of two steps:

1. Compute distances for $i = 1, ..., N,\ j = 1, ..., k$

(5)

(6)

# Algorithm

K-Means consists of two steps:

1. Compute distances for $i = 1, ..., N,\ j = 1, ..., k$

$$d(\boldsymbol{x}_i, \boldsymbol{c}_j) = \|\boldsymbol{x}_i - \boldsymbol{c}_j\|_2 \tag{5}$$

$$\tag{6}$$

# Algorithm

K-Means consists of two steps:

1. Compute distances for $i = 1, ..., N,\ j = 1, ..., k$

$$d(\boldsymbol{x}_i, \boldsymbol{c}_j) = \|\boldsymbol{x}_i - \boldsymbol{c}_j\|_2 \tag{5}$$

$$\boldsymbol{x}_i \in \mathcal{C}_j \ \Leftrightarrow\ j = \underset{t=1,...,k}{\arg\min}\, d(\boldsymbol{x}_i, \boldsymbol{c}_t) \tag{6}$$

# Algorithm

K-Means consists of two steps:

1. Compute distances for $i = 1, ..., N,\ j = 1, ..., k$

$$d(\boldsymbol{x}_i, \boldsymbol{c}_j) = \|\boldsymbol{x}_i - \boldsymbol{c}_j\|_2 \tag{5}$$

$$\boldsymbol{x}_i \in \mathcal{C}_j \ \Leftrightarrow\ j = \underset{t=1,...,k}{\arg\min}\, d(\boldsymbol{x}_i, \boldsymbol{c}_t) \tag{6}$$

2. Compute new centroids as the average of the cluster

# Algorithm

K-Means consists of two steps:

1. Compute distances for $i = 1, ..., N,\ j = 1, ..., k$

$$d(\boldsymbol{x}_i, \boldsymbol{c}_j) = \|\boldsymbol{x}_i - \boldsymbol{c}_j\|_2 \tag{5}$$

$$\boldsymbol{x}_i \in \mathcal{C}_j \ \Leftrightarrow\ j = \arg\min_{t=1,...,k} d(\boldsymbol{x}_i, \boldsymbol{c}_t) \tag{6}$$

2. Compute new centroids as the average of the cluster

$$\boldsymbol{c}_j = \frac{1}{|\mathcal{C}_j|} \sum_{\boldsymbol{x}_i \in \mathcal{C}_j} \boldsymbol{x}_i \tag{7}$$

# Centroids selection

Centroids can be

- Randomly selected from the dataset;
- Randomly assign points to the clusters;
- Chosen with previous analysis.

# Centroids selection

Centroids can be

- Randomly selected from the dataset;
- Randomly assign points to the clusters;
- Chosen with previous analysis.

**This choice affect the convergence of the algorithm.**

# Convergence criteria

Several criteria can be chosen.

- No big variation in cluster assignments;
- No big distance between old and new centroids;
- Specific number of iterations;
- Several repetitions of previous steps.

## Section 3

# Serial Implementation

# Distance between two points

```
def distance_point_point(p1,p2):
    return np.sqrt(sum((p1-p2)**2))
```

This function simply computes the Euclidean distance between two points.

$$d(p_1, p_2) = \left( \sum_{i=1}^{D} (p_{1i} - p_{2i})^2 \right)^{\frac{1}{2}}.$$

# Distance between point and centroids

```
def distance_pont2points(p,C):
    k=len(C)
    d_min=np.inf
    k_min=k
    for j,c in enumerate(C):
        d=distance_point_point(p,c)
        if d<d_min:
            d_min=d
            k_min=j
    return k_min
```

This function returns the index of the closest centroid.

$$k_{min} = \arg\min_{j=1,\ldots,k} d(x_i, c_j).$$

# Centroid selection

```
ps=random.sample(range(0,N),k)
for j, p in enumerate(ps):
    #print(p)
    C[j,:]=data[p,:]
```

We select $k$ points, sampling without replacement from the dataset.

# Number of iterations

```
change=N
while change>N//100:
    change=0
```

We initialize a counter for points that change cluster after a step of the algorithm. This is initialized with the value $N$. The `while` loop continues until the counter is less than $\frac{N}{100}$. We reset the counter to 0 inside the loop.

# Number of iterations I

```
for i,dato in enumerate(data):
    kmin=distance_pont2points(dato, C)
    if ass[i]!= kmin:
        change+=1
        if flag:
            sums[ass[i],:]-=dato
            counters[ass[i]]-=1
        sums[kmin,:]+=dato
        counters[kmin]+=1
        ass[i]=kmin
```

The red code updates the counter of points that change cluster and assigns the new cluster to the point, if it is different.

# Number of iterations II

```
for i,dato in enumerate(data):
    kmin=distance_pont2points(dato, C)
    if ass[i]!= kmin:
        change+=1
        if flag:
            sums[ass[i],:]-=dato
            counters[ass[i]]-=1
        sums[kmin,:]+=dato
        counters[kmin]+=1
        ass[i]=kmin
```

The blue code updates sums and counters of clusters, by summing point coordinates to the relative sum and subtracting it from the previous one (if it is not the first step). The same for the counter.

# Re-center clusters

```
for i in range(k):
    C[i,:]=sums[i,:]/counters[i]
```

We compute the new centroids by dividing the sum of points in a cluster by the number of points in it.

$$c_j = \frac{1}{|\mathcal{C}_i|} \sum_{x_i \in \mathcal{C}_j} x_i,$$

# Profiling serial algorithm

| Function/Module | Total Time | Diff | Local Time | Diff | Calls | Diff |
|---|---|---|---|---|---|---|
| main | 29.66 s | | 744.80 µs | | 1 | |
| k_means_serial | 16.26 s | | 430.81 ms | | 1 | |
| seed | 45.90 µs | | 16.80 µs | | 2 | |
| sample | 60.20 µs | | 23.80 µs | | 1 | |
| distanza_punto2punti | 15.83 s | | 2.74 s | | 432000 | |
| <built-in method time.tim... | 6.40 µs | | 6.40 µs | | 9 | |
| <built-in method numpy.z... | 193.20 µs | | 193.20 µs | | 196 | |
| <built-in method builtins.... | 31.10 µs | | 31.10 µs | | 3 | |
| charge_data_serial | 40.56 ms | | 19.20 ms | | 1 | |
| analysis | 13.36 s | | 1.52 ms | | 1 | |
| <built-in method time.time> | 6.40 µs | | 6.40 µs | | 9 | |
| <built-in method builtins.print> | 31.10 µs | | 31.10 µs | | 3 | |
| _find_and_load | 791.68 ms | | 4.61 ms | | 388 | |

Figura 1: Spyder profiler output.

## Section 4

# Parallel Implementation

# Centroids Selection

```
ps=random.sample(range(0,N),k)
for j, p in enumerate(ps):
    C[j,:]=data[p,:]
```

Choose $k$ centroids by randomly selecting them from the dataset in a serial way.

# Pool generation

[Python, ]

```
pool = multiprocessing.Pool(processes=pool_size)
```

We define a pool of workers of `pool_size` dimension. This creates ready-to-use processes that can be used several times without joining them every time.

# Number of Iterations

```
change=N
while  change>N//100:
    change=0
```

We initialize the counter of points that change cluster with value $N$. The `while` loop stops if it becomes lower than $\frac{N}{100}$ and we set it to 0 just inside.

# Clustering assigment I

[Python, ]

```
ass_new=pool.map(
    partial(distance_pont2points,C=C),data)
```

The assignment of points in clusters is made with a `pool.map()` method. This method makes the processes of the pool apply the function `distance_pont2points` to every data point in a synchronous way. The output is the new assignment.

# Clustering assigment II

[Python, ]

```
ass_new=pool.map(
    partial(distance_pont2points,C=C),data)
```

The `partial` function of the `functool` package makes the function depend on only one argument, which is essential for `pool.map` method.

# Update counter

```
change=sum(ass!=ass_new)
```

The `change` counter is updated by summing the number of `True` in the boolean vector of comparison between the two assignments, the old and the new one.

# Update counters and sums

```
for i in range(N):
    if ass[i]!=ass_new[i]:
        sums[ass_new[i],:]+=data[i,:]
        counters[ass_new[i]]+=1
        if flag:
            counters[ass[i]]-=1
            sums[ass[i],:]-=data[i,:]
        ass[i]=ass_new[i]
```

This part is similar to the serial one. In a serial way, we update sums and counters for every cluster if a point is classified differently.

# New centroids

```
C=np.array(pool.map(partial(recenter,sums,counters),ran

def recenter(sums,counters,i):
    return sums[i,:]/counters[i]
```

New centroids are computed with another `pool.map` method applying a function that simply computes a division.

# Join pool

```
pool.close()
pool.join()
```

At the end of the while loop.
Processes are no more useful, and this code closes and joins them.

## Section 5

# Experiments

# Test

I applied the two algorithms with

# Test

I applied the two algorithms with

- **9** randomly generated datasets;

# Test

I applied the two algorithms with

- **9** randomly generated datasets;
- **3** values of $N$: $10^4$, $10^5$ and $10^6$;

# Test

I applied the two algorithms with

- **9** randomly generated datasets;
- **3** values of $N$: $10^4$, $10^5$ and $10^6$;
- **3** values of $k$: 10, 20, and 40;

# Test

I applied the two algorithms with

- **9** randomly generated datasets;
- **3** values of $N$: $10^4$, $10^5$ and $10^6$;
- **3** values of $k$: 10, 20, and 40;
- **5** values of process' numbers: 1, 2, 4, 8, and 16.

# Datasets

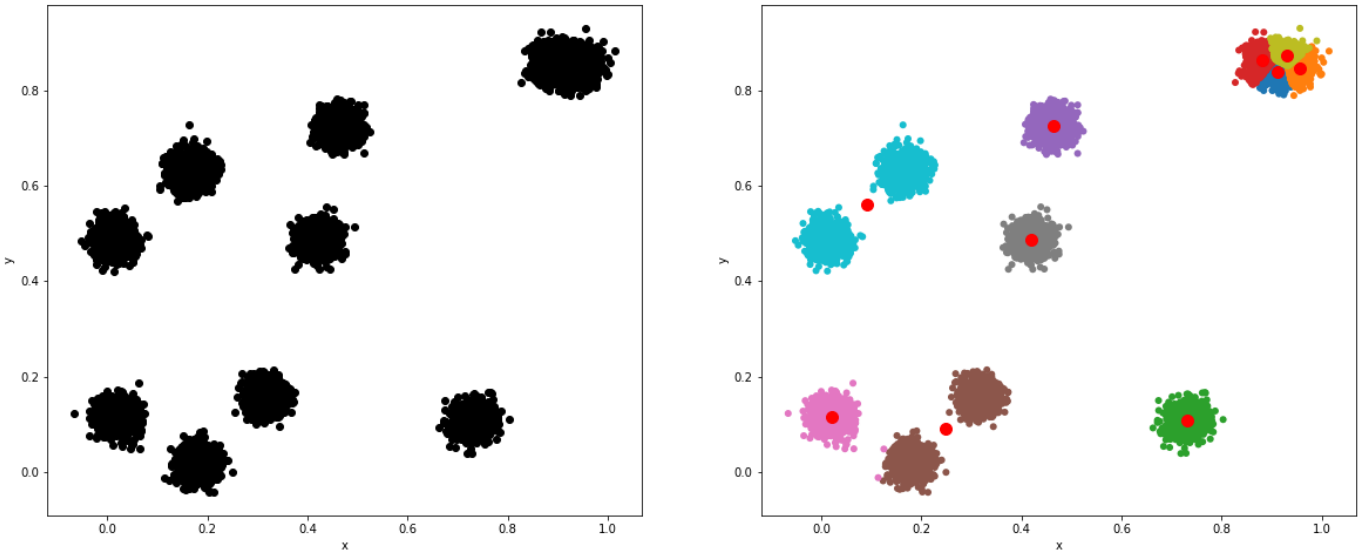



Figura 2: Dataset and cluster for $k = 10$ and $N = 10^4$.

# Datasets

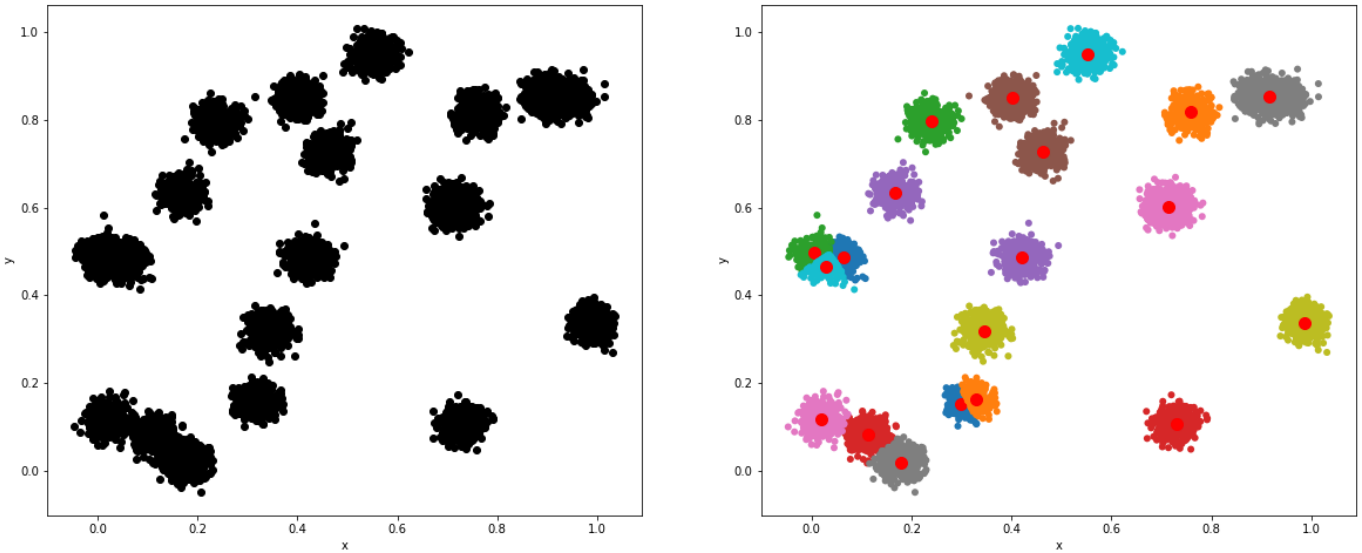



Figura 2: Dataset and cluster for $k = 20$ and $N = 10^4$.

# Datasets

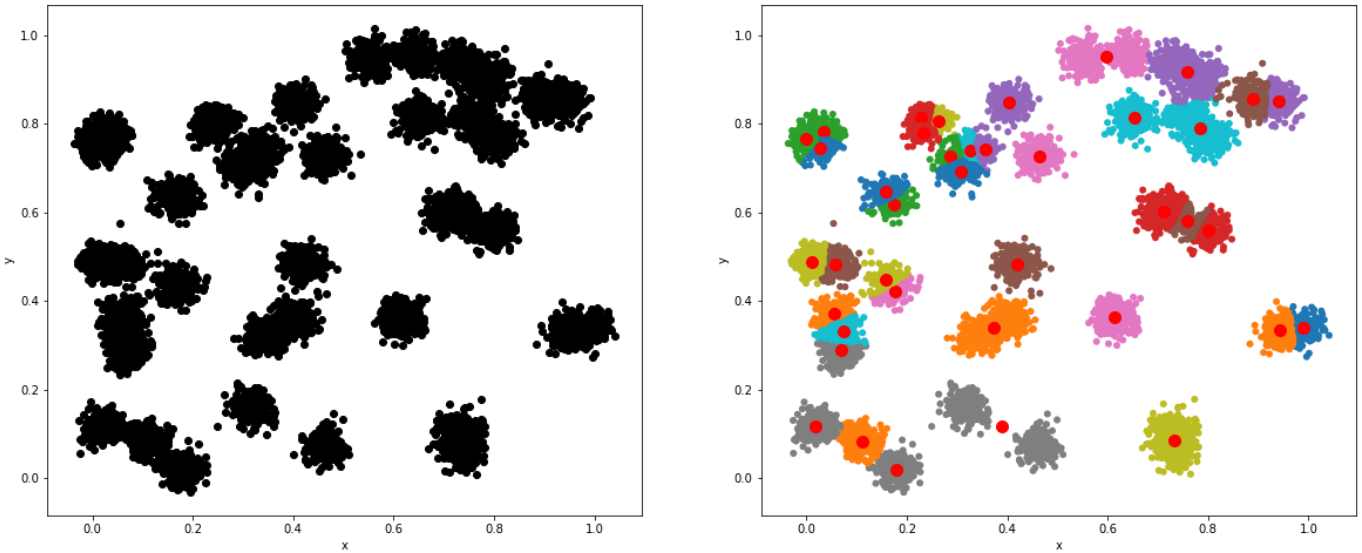
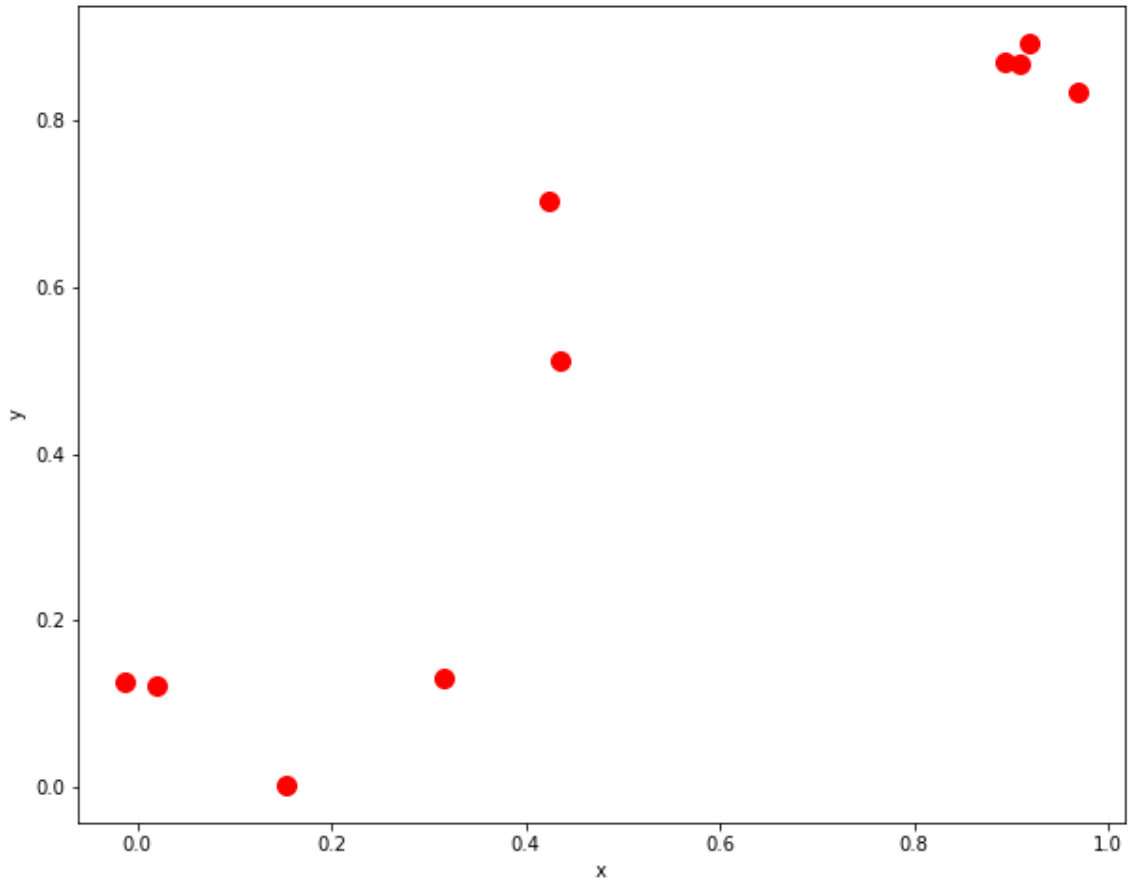



Figura 2: Dataset and cluster for $k = 40$ and $N = 10^4$.

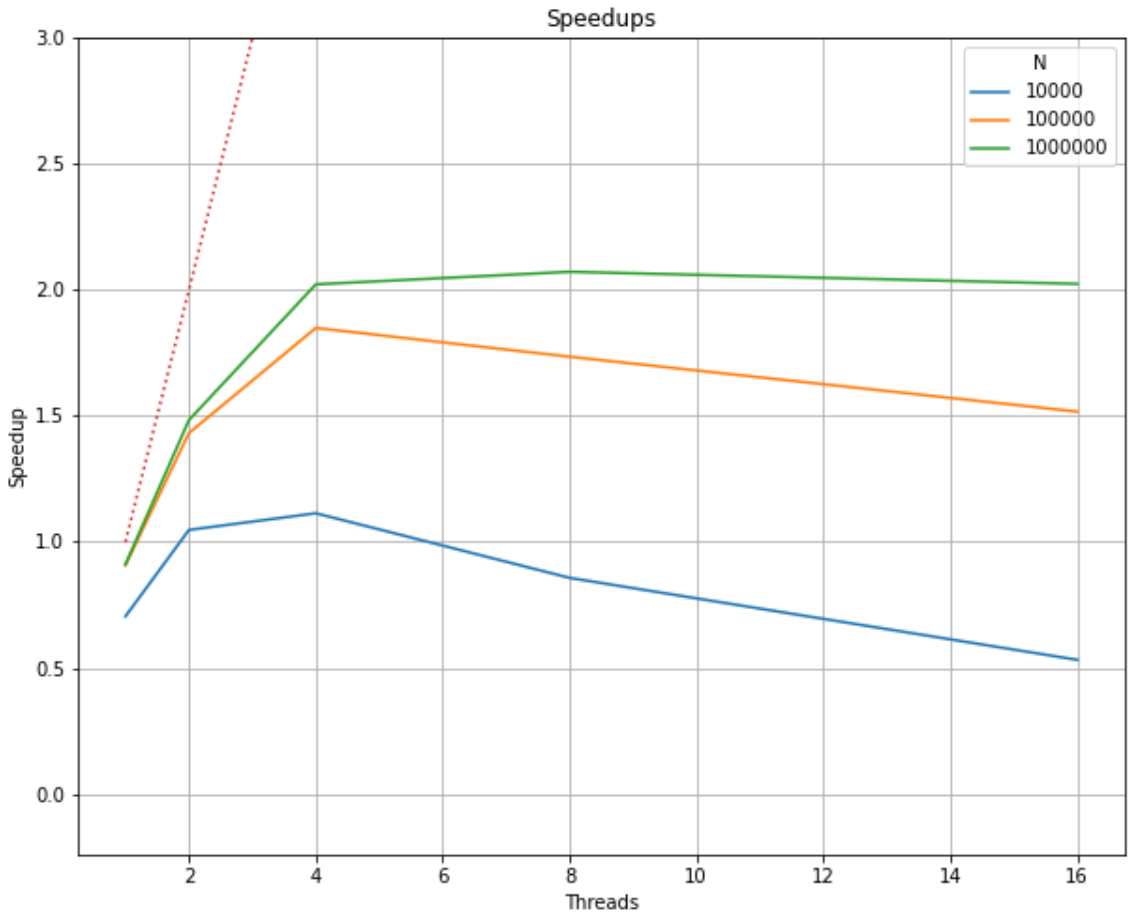
# Clustering Process



# Results for $k = 10$ I

| Processes | Speedup $N = 10^4$ | Speedup $N = 10^5$ | Speedup $N = 10^6$ |
|---|---|---|---|
| 1 | 0.70440 | 0.90754 | 0.91036 |
| 2 | 1.04726 | 1.43153 | 1.48321 |
| 4 | 1.11342 | 1.84884 | 2.02187 |
| 8 | 0.85709 | 1.73453 | 2.07164 |
| 16 | 0.53186 | 1.51680 | 2.02388 |

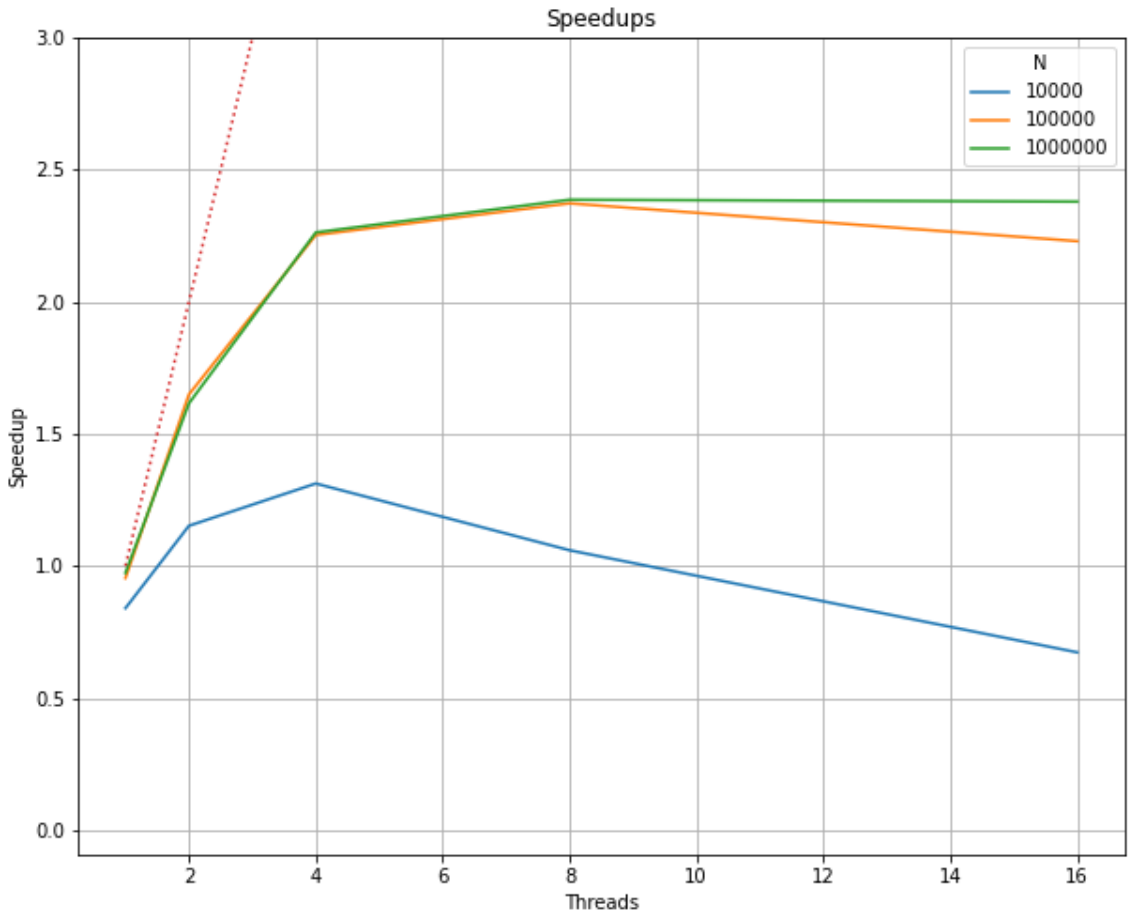
# Results for $k = 10$ II



# Results for $k = 20$ I

| Processes | Speedup $N = 10^4$ | Speedup $N = 10^5$ | Speedup $N = 10^6$ |
|---|---|---|---|
| 1 | 0.84010 | 0.95420 | 0.97390 |
| 2 | 1.15221 | 1.64954 | 1.61574 |
| 4 | 1.31224 | 2.25325 | 2.26319 |
| 8 | 1.05950 | 2.37325 | 2.38699 |
| 16 | 0.67283 | 2.23004 | 2.37975 |

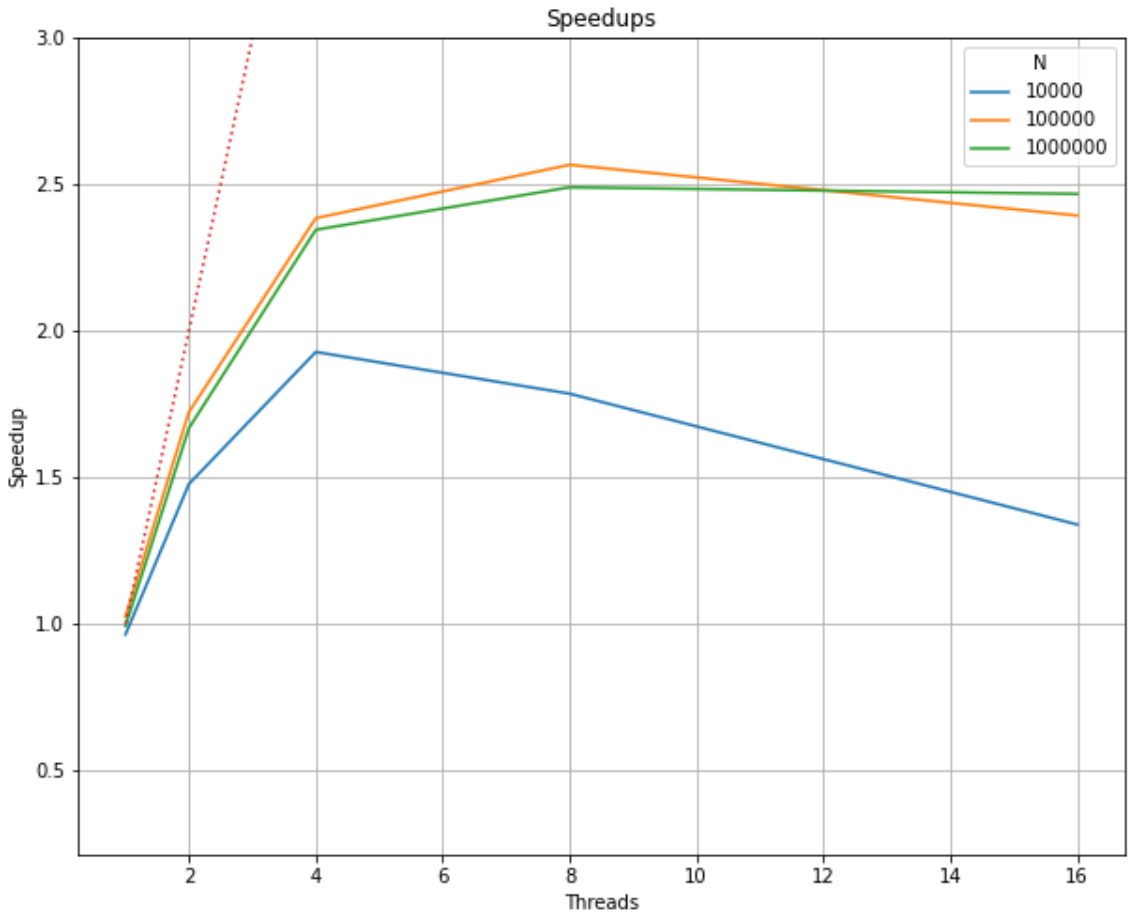
# Results for $k = 20$ II



# Results for $k = 40$ I

| Processes | Speedup $N = 10^4$ | Speedup $N = 10^5$ | Speedup $N = 10^6$ |
|---|---|---|---|
| 1 | 0.96216 | 1.02302 | 0.99209 |
| 2 | 1.47628 | 1.72194 | 1.66696 |
| 4 | 1.92731 | 2.38432 | 2.34458 |
| 8 | 1.78475 | 2.56659 | 2.48961 |
| 16 | 1.33792 | 2.39313 | 2.46708 |

# Results for $k = 40$ II



## Section 6

# Conclusions

# Comment on results

The **speedup** is not linear and its behaviour gets better with the dimension of the dataset $N$ and the number of clusters $k$.

# Comment on results

The **speedup** is not linear and its behaviour gets better with the dimension of the dataset $N$ and the number of clusters $k$.
The **cost** of creating **processes** is quite high and is balanced only with high dimensions of the dataset and a small number of processes.

# References


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


Python.
multiprocessing — process-based parallelism.
`https://docs.python.org/3/library/multiprocessing.html`.
Accesso: 13 luglio 2025.