

Parallelization of K-Means Algorithm with Python Multiprocessing

Ernesto Palchetti
Università degli Studi di Firenze
ernesto.palchetti@edu.unifi.it

Abstract

This report aims to implement and analyse a parallel version of the K-means clustering algorithm, realized with the Python package multiprocessing. This review focuses on the algorithm, implementation details, and tests for the evaluation of the speedup.

Future Distribution Permission

The author of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses.

1. Introduction

This Section provides a little review of the steps composing the K-means algorithm, one of the most common clustering algorithms. Then, the second part of the Section introduces the multiprocessing Python Package [3, 2].

1.1. K-means algorithm

K-means is a clustering strategy with several different implementations, some of which can be found in Chapters 13-14 of Hastie, Tibshirani, and Friedman (2009) [1]. Given k (parameter of the model) centroids, the structure of the algorithm is based on two steps:

1. We assign a point to the cluster associated with the closest centroid;

2. The new centroid is computed as the average point of a cluster.

The starting values for the centroids can be chosen randomly inside the domain, from other previous analyses, or randomly selected from the dataset. The algorithm consists of repeated iterations of these two steps until a convergence criterion is reached. The one I chose for this project is to stop when the number of points that are classified in a different cluster after a step is less than 1% of the dataset size.

More formally, in the first phase, we compute the Euclidean distance (other choices are possible) between a point and every centroid, storing the position of the closest one:

$$c_i = \arg \min_{j=1,\dots,k} ||x_i - d_j||. \quad (1)$$

The new centroid is computed with

$$c_j = \frac{1}{|\mathcal{C}_j|} \sum_{x_i \in \mathcal{C}_j} x_i, \quad (2)$$

where \mathcal{C}_j is the j -th cluster.

1.2. Multiprocessing Package

The multiprocessing Python package is part of the Python Standard Library and, according to the documentation at this [link](#), provides a way to bypass the Global Interpreter Lock (*gil*) by substituting threads with processes, everyone with its own *gil*.

In this project, the `pool` class ([documentation link](#)) plays a main role. This is particularly useful in situations in which every process is created once and reused several times. The main method

I used is the `map`, a parallel version of the `map` Python method. This method makes pools perform a specified function in a synchronous (or asynchronous for `map_async`) way over a large number of different arguments.

2. Implementation

This Section goes deep into the implementation details, firstly of the serial algorithm and then of the parallel one. The code can be found on this [GitHub repository](#).

2.1. Serial Algorithm

The main component of both algorithms is the Euclidean distance between two points, so I defined a basic function that returns

```
np.sqrt(sum((p1-p2)**2))
```

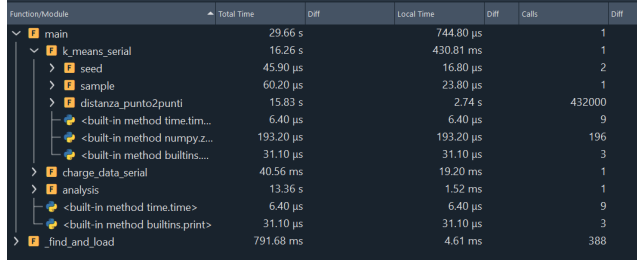
ready-to-use with points of every dimension.

The serial algorithm simply reflects the structure of the algorithm of Section 1.1. The first part selects k points from the dataset and sets them as starting centroids. Then, a `while` loop repeats the two steps of Section 1.1 until the fraction of points that are classified differently between two consecutive steps is less than 1%.

The main corpus is made by a `for` loop over the dataset. For every point, the distance between it and every centroid is computed, and the position of the closest one is stored. If the classification is different from the previous one, the counter involved in the `while` loop is updated. Then the classification is changed. Inside the same `for` loop, when a point is classified, its components are added to a counter (one for every cluster). This, together with a counter of points inside every cluster, allows us to compute the new centroids after the loop. Here, for every cluster, the centroid is computed with a simple division like

```
C[i,:]=sums[i,:]/counters[i].
```

To achieve a good parallelization, I used the Spyder profiler, obtaining results in Figure 1. As we can see, most of the time is spent inside the distance function, so we will focus on that part while parallelizing the algorithm.



Section/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	
> distancia_punto2punto	15.83 s		2.74 s		432000	
> <built-in method time.time>	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	

Figure 1. Profiling of serial algorithm.

2.2. Parallel Algorithm

The parallel version of the algorithm is based on a function that, given a point and the centroids, computes the distance between that point and each centroid and returns the index of the closest one. This operation will be performed in the main function by a pool. The code is the following

Listing 1. Distance Function

```
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
```

The first part of the algorithm is the same as the serial one. After the centroids are chosen, a pool is defined with a given number of processes with

```
pool = multiprocessing.Pool(processes=8)
```

Inside a `while` loop with the condition mentioned before, a `map` method makes the pool apply the distance function in Listing 1, returning the index of the closest centroid for every point, that is the vector of new assignment `ass_new`.

The counter of the number of points that changes cluster is computed simply with

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

where `ass` is the vector of assignment of the previous step.

Then, I used a `for` loop over N (the dataset size) indexes and, if the relative point is classified differently, I update sums and counters as the serial version. I also tried a parallel version of this part, but the management of these counters with a shared variable was too heavy, resulting in worse performance.

The computation of new centroids is then performed in parallel by the poll with another function that simply computes a division. Every process only needs to read shared counters and in different indexes, so there's no possibility of *race condition*.

3. Experiments

The experiment consists of applying the two versions of the algorithm on a randomly generated dataset.

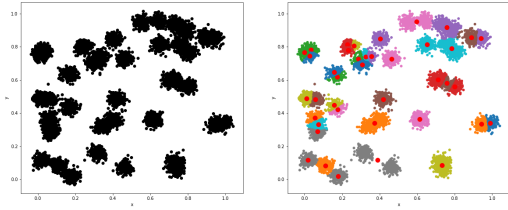


Figure 2. Data and clusters for $k = 40$ and $N = 10^4$.

3.1. Test Repetitions

The test is repeated with different values of the number of points N (10^4 , 10^5 , and 10^6), and of the number of clusters k (10, 20, and 30). For one of these, the dataset and resulting clusters are shown in Figure 2. For every combination, I changed the number of processes and I tested 1, 2, 4, 8 and 16 processes. Then, I computed the obtained speedup as the division of the time of the serial algorithm by the time of the parallel one.

3.2. Results for $k = 10$

Speedup values are shown in Table 1 and in Figure 3. As we can see, with $k = 10$ the costs of managing processes are not balanced for low values of N , and I reached the best speedup with

$N = 10^6$ and 2 or 4 processes, considering the number of cores involved.

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

Table 1. Speedups for $k = 10$.

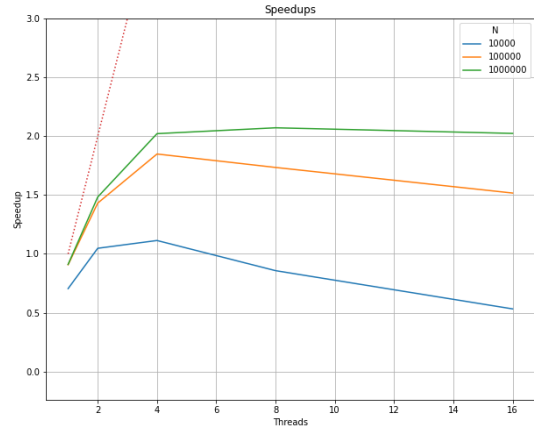


Figure 3. Speedup values for $k = 10$.

3.3. Results for $k = 20$

Speedup values are shown in Table 2 and in Figure 4. With an increased value of k , I reached better results for every value of N , and, again, the best values are obtained for two and four processes, both for $N = 10^5$ and $N = 10^6$.

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

Table 2. Speedups for $k = 20$.

3.4. Results for $k = 40$

Speedup values are shown in Table 3 and in Figure 5. With 40 clusters, speedups are still better, also for $N = 10^5$, reaching 1.72 with two cores and 2,38 with four cores.

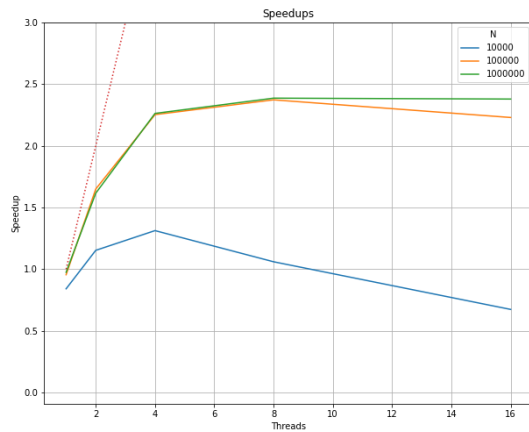


Figure 4. Speedup values for $k = 20$.

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

Table 3. Speedups for $k = 40$.

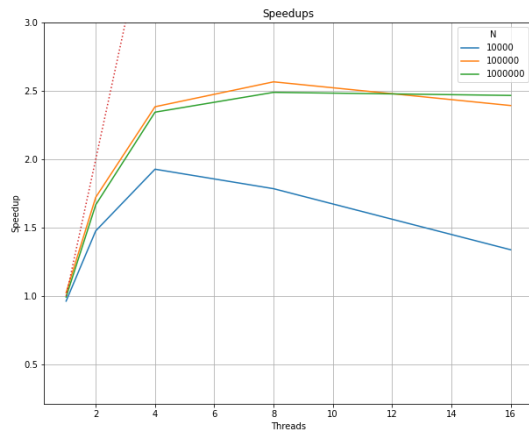


Figure 5. Speedup values for $k = 40$.

4. Conclusions

The reached speedup is far from linear, but by increasing the dataset's dimension, I obtained good values with two and four cores involved. The speedup behaviour in Figure 3, 4, and 5 shows that process costs are less impactful until I used 4

cores, then performance improvement stops and starts decreasing. The impact of the numerosity of the dataset on the speedup is essential. With a small dataset, there's no need for parallelism, especially if the costs are so high.

References

- [1] 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. 1
- [2] Python. multiprocessing — process-based parallelism. <https://docs.python.org/3/library/multiprocessing.html>. Accesso: 13 luglio 2025. 1
- [3] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. 1