PC-2023/2024 Parallel k-means

Christian Mancini
christian.mancini1@edu.unifi.it
firstauthor@i1.org

Abstract

In this report, we present the implementation of the classic K-means clustering algorithm in Python, focusing on enhancing its performance through parallelization techniques. We utilized Joblib [1] and Numba [2] to leverage multi-core CPU capabilities, resulting in significant speed improvements. The implementation with Joblib achieves a speedup of approximately 4 times compared to the standard implementation, while the Numba-optimized version demonstrates an impressive speedup of over 20 times.

Future Distribution Permission

1. Introduction

K-means is a widely used clustering algorithm that partitions a dataset into K distinct clusters based on feature similarity. It operates by iteratively assigning data points to the nearest cluster centroid and then updating the centroids based on the assigned points. K-means is considered an embarrassingly parallel problem because the assignment of data points to clusters and the computation of centroids can be performed independently for each data point. This means that multiple data points can be processed simultaneously without the need for inter-process communication or synchronization. As a result, K-means is well-suited for parallelization, allowing for significant performance improvements when implemented on multi-core processors.

2. Parallelization strategy

We utilized both Joblib and Numba as parallelization strategies to enhance performance and overcome limitations imposed by the Global Interpreter Lock (GIL) in Python. Joblib is a library that facilitates the parallel execution of tasks by creating separate processes. This means that each task runs in its own Python interpreter. By using Joblib, we can efficiently distribute the computation of distance calculations across multiple CPU cores. On the other hand, Numba is a Just-In-Time (JIT) compiler that translates a subset of Python and NumPy code into optimized machine code at runtime. By using Numba's @njit decorator with the parallel=True option, we can compile the distance computation function to run in parallel without being constrained by the GIL. To compare the performance of the K-means algorithm using Joblib and Numba, we implemented two distinct functions for computing the labels of data points based on their distances to the centroids. The performance metrics were evaluated based on the execution time of each approach when fitting the model to various datasets.

```python
@njit(parallel=True)
def compute_label_numba(point, centroids):
        k = centroids.shape[0]
        distances = np.empty(k)
        for i in prange(k):
                distances[i] = np.linalg.norm(
                    point - centroids[i])
        return np.argmin(distances)
```

Listing 1. numba implementation

```python
def compute_label_joblib(point, centroids):
        k = centroids.shape[0]
        distances = np.empty(k)
        for i in range(k):
                distances[i] = np.linalg.norm(
                    point - centroids[i])
        return np.argmin(distances)
```
Listing 2. Serial and Joblib Jmplementation

```
new_labels = Parallel(n_jobs=n_jobs)(delayed(
    compute_label_joblib)(self.points[i], self.
    centroids) for i in range(self.points.shape
    [0]))
```
Listing 3. Joblip parallel call

3. Results

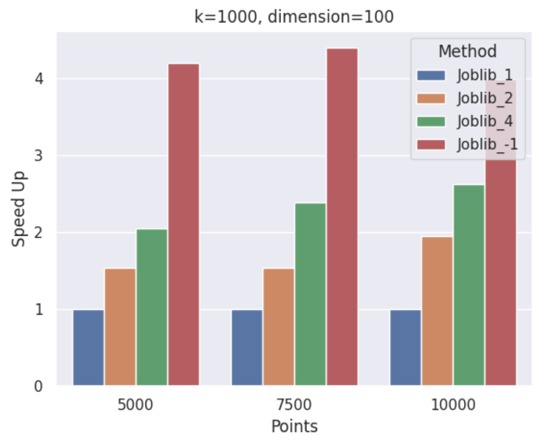
Below is the table 1 showing the speed-ups achieved with different methods for the k-means algorithm. Additionally, two graphs are presented that compare the performance between Joblib and Numba.

Table 1. Speed Up of K-Means

Points	Method	Speed Up	
5000	Numba	3.5288	22.79
5000	Joblib_1	80.4298	1.00
5000	Joblib_2	52.1189	1.54
5000	Joblib_4	39.1091	2.06
5000	Joblib_-1	19.1136	4.21
7500	Numba	5.8069	26.09
7500	Joblib_1	151.4988	1.00
7500	Joblib_2	98.7612	1.53
7500	Joblib_4	63.2441	2.40
7500	Joblib_-1	34.4307	4.40
10000	Numba	7.7299	28.83
10000	Joblib_1	222.8223	1.00
10000	Joblib_2	114.4596	1.95
10000	Joblib_4	84.9274	2.62
10000	Joblib_-1	55.8627	3.99

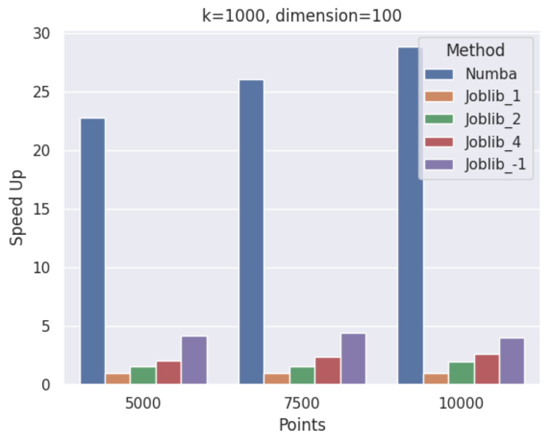
In Figure 1, we can see the results of Joblib. The parallel gain is not linear with the number of cores, but we still achieve a good speed-up.

Figure 1. Joblib speed up



In Figure 2, we can observe the significant increase in speed-up achieved with Numba compared to Joblib. This improvement is more than linear due to the fact that Numba is compiled, eliminating the overhead associated with the Global Interpreter Lock (GIL) in Python.

Figure 2. Numba and Joblib speed up



References

[1] J. Developers. Joblib: Lightweight pipelining in python, n.d. Available at: `https://joblib.readthedocs.io/en/latest/`.

[2] N. Developers. Numba: A just-in-time compiler for python, n.d. Available at: `https://numba.pydata.org/`.