Parallel Implementation of K-Means Clustering with OpenMP

Alessandro Maggio

August 20, 2022

Abstract

OpenMP is a very powerful framework to scale up almost any kind of algorithm from serial programming to parallel programming. This project aims to propose a possible speedup of K-Means clustering exploiting multi-threading and evaluating the results. The same technique has been applied also to the K-Medoids variant and to the Silhouette score computation.

1 Introduction

K-Means is a popular unsupervised learning algorithm able to split a dataset of points, in an arbitrary number of dimensions, into k clusters. Each cluster is determined by its *centroid*, a point with the same dimension of to the input data. A generic sample belongs to the cluster of the nearest centroid.
Since the number of clusters k must be predefined, there are several metrics to evaluate the quality of the clustering procedure; in particular I have chosen the silhouette score to take into account both the intra-cluster similarity and inter-cluster dissimilarity.
There are many ways to parallelize the illustrated steps and the effect in terms of computational time, as we will see in next sections, can largely vary depending on the algorithm complexity and fixed overheads. All the experiments have been performed with a raw C/C++ environment in order to minimize the number of calls to black-box library functions or not thread-safe data structures.

2 Experiments description

Code is entirely written in C/C++ and no other dependencies are needed but OpenMP. The number of physical threads in both local and remote (university HPC) environments is 4, limiting the possibility to exploit further options of improvements.

2.1 Base K-Means Algorithm

The base K-Means implementation is a quite straightforward algorithm and can be schematised in few steps. We assume to have n points (in any dimension) imposing a partitioning in k clusters.
The procedure remains substantially the same in both parallel and serial implementation.

1. Initialize k centroids according to some policy.
2. For each one of the n points, compute its distance from the nearest centroid (and assign the point to the centroid's cluster).
3. For all the k clusters, compute the new centroids as the mean of points' coordinates belonging to the same cluster.
4. Repeat steps 2 and 3 until a stop criterion is satisfied.

The complexity is:

$$\mathcal{O}(n \times k \times num\_iter)$$

Where *num_iter* is the number of iterations required to converge. The implicit assumption is that computing the distance between two points needs a constant and negligible time, while the complexity is due to the large number of operations required rather that their individual cost.

2.1.1 Serial Implementation

At each iteration a pseudo-code like this one is performed:

Listing 1: Main loops pseudo-code (Step 2)

```
// 'data' and 'centroids' are array of Point
// 'data' is given
// 'centroids' is previously initialized
// Point contains its 'coordinates'(double[])  and 'cluster'(int)

forall point in data:                          // cost n
    minimum_distance = MAX;
    forall cent in centroids:                  // cost k
        dist = compute_distance(point, cent);
        if dist < minimum_distance:            // find the nearest centroid
            minimum_distance = dist;
            point.cluster = cent.cluster; // assign point to cluster

    clusters_size_list[point.cluster]++;  // update cluster dimension
    partial_cluster_sum[point.cluster] += point.coordinates // add point
```

During the two nested loops for distance computation (step 2), also the partial accumulation to prepare the mean is updated (step 3). The centroids computation is then very simple:

Listing 2: New centroids (mean computation) pseudo-code.

```
Point new_centroids[k];

for i in range(0,k):  // cost k
    new_centroids[i] = partial_cluster_sum[i] / clusters_size_list[i];
```

```
if |centroids - new_centroids| / |centroids| < eps: // stop criterion
        return new_centroids;

centroids = new_centroids;
```

The complexity of Step 3 is $\mathcal{O}(k)$, so negligible. The magnitude of eps can influence the number of iterations, so in the actual implementation I preferred a low relative threshold of 0.00001 which can fit any purpose.

2.1.2 Parallel Implementation

For the multi-thread scaling I decided to decompose the most expensive loop, the first one in the serial code, so that each thread elaborates ideally n/T input data points (where T is the number of available threads).
The new complexity is:

$$\mathcal{O}(\frac{n}{T} \times k \times num\_iter)$$

So the new code appears very similar to the old one but with per-thread partial accumulator which need to be merged before the definition of new centroids. The arrays becomes 2D to be indexed both by thread_id (`omp_get_thread_num` in OpenMP) and by cluster:

Listing 3: Parallel main loops accumulation.

```
T_clusters_size_list[thread_id][point.cluster]++;
T_partial_cluster_sum[thread_id][point.cluster] += point.coordinates;
```

Finally one more loop is required to compute the new_centroids. The complexity is $\mathcal{O}(T \times k)$ but since also this procedure has been parallelized (after an implicit barrier at the end of step 2) the result becomes $\mathcal{O}(\frac{T}{T} \times k)$ that is equal to $\mathcal{O}(k)$ as before.

Listing 4: New centroids for parallel implementation pseudo-code.

```
Point new_centroids[k];

for i in range(0,k):
    for j in range(0,T):    // collapse arrays to serial case
        new_centroids[i] += T_partial_cluster_sum[j][i];
        clusters_size_list[i] += T_clusters_size_list[j][i];

    new_centroids[i] /= clusters_size_list[i];

if |centroids - new_centroids| / |centroids| < eps:
        return new_centroids;

centroids = new_centroids;
```

2.2 K-Medoids Algorithm

K-Medoids is a variant of K-Means which imposes that the centroids must belong to the input data. As first approximation we could just run the same algorithm and return the closest input points to the centroids. This is not enough when clusters are sparse (or empty) nearby the natural centroids (such as for rings).
Although the general idea remains the same, the crucial difference is that a new centroids is computed as the point which minimize the average distance from all the other points in the same cluster. In the worst case, when clusters are completely unbalanced, the complexity becomes

$$\mathcal{O}(n^2 \times num\_iter)$$

2.2.1 Parallel Implementation

I will directly show the pseudo-code of the parallel implementation since the overall mechanism to move from serial to multi.thread is pretty much the same as before.

Listing 5: Main loop for K-Medoids.

```
forall p1 in data:                                  // cost n
    tot_dist = 0.;
    forall p2 in data:                              // cost n
        if (p1.cluster == p2.cluster):
            tot_dist += compute_distance(p1, p2);
    if (tot_dist < T_partial_min_dist[thread_id][p1.cluster]):
        T_partial_min_dist[thread_id][p1.cluster] = tot_dist;
        T_partial_new_centroids[thread_id][p1.cluster] = p1;
```

Where `T_partial_min_dist` is a 2D array which contains for every thread and for every cluster (in this order) the point with the minimum average distance from all the other points of its cluster, among the all the points processed by the same thread. So ideally the complexity becomes $\mathcal{O}(\frac{n^2}{T} \times num\_iter)$.
In particular it is worth to highlight that only the outer loop can be made parallel because in case of `collapse(2)` from OpenMP, the average distance would be computed only on a fraction of the points belonging to the same cluster of p1.
The final merge to define the centroids (like in 2) is implemented serially. The performance increasing of parallelizing this step of cost $\mathcal{O}(T \times k)$ has resulted to be close to zero (probably due to the multi-thread overhead).

2.3 Initialization

The only way to guarantee the clustering convergence is an exhaustive search but it is too expensive, for this reason the iterative methods are quite successful. In order not to fall in sub optimal solutions and to converge in the lowest number of iterations as possible, the starting positions of the centroids are relevant.
In this project I propose two options, both using centroids belonging to input data (so compatible also with K-Medoids):

- Random Initialization: k random points are picked, constant and minimum possible complexity. Not very reliable.

- Kmeans++: given a random starting point, it selects the remaining k-1 centroids in order to take them as far one from the other as possible. After the parallelization of the loop on n the complexity is $\mathcal{O}(\frac{n}{T} \times k)$.

2.4 Silhouette Score

The silhouette score is a consistent metric used to evaluate a clustering result. In this project it is implemented in its standard formulation of complexity $\mathcal{O}(n^2)$. Also in this case a parallel variant is proposed, still working on the outer loop such as in the case of K-Medoids. The final complexity is:

$$\mathcal{O}(\frac{n}{T} \times n) = \mathcal{O}(\frac{n^2}{T})$$

3 Experiment results

All the experiments have been performed on the Slurm HPC. Every run uses the Kmeans++ initialization. The data are 100'000 points in 3D for K-Means and 10'000 points in 3D for K-Medoids (much more expensive). Data are 3D to help the visualization and they have been generated using a Python script.
Finally both variants have been tested for T from 1 to 8, averaging the times of 10 computations.

Speed up table								
Algorithm	T=1	T=2	T=3	T=4	T=5	T=6	T=7	T=8
K-Means	1.0	2.02	2.86	**3.75**	3.35	3.12	3.04	3.00
K-Medoids	1.0	1.10	1.46	**2.02**	1.83	1.67	1.16	1.25
Silhouette Score	1.0	1.96	2.78	3.27	**3.34**	3.14	3.32	3.21
Kmean++	1.0	1.77	1.37	**1.48**	1.15	1.12	1.12	0.96

Table 1: Speed Up table for different T.

3.1 K-Means and K-Medoids results

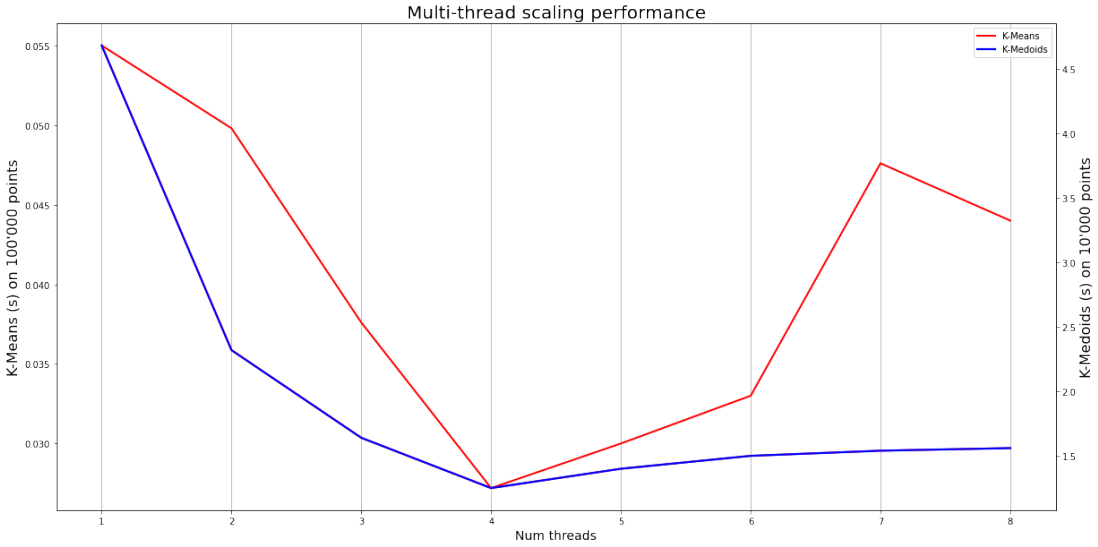

Figure 1: K-Means and K-Medoids performance increment in a multi-thread setting.

The figure shows that both the clustering methods reach their minimum computational time for T=4, the number of physical threads. After this point the overall performance is stable for K-Medoids but becomes really poor for K-Means. A possible explanation is the different complexity which leads K-Means to 'pay' more, on a relative scale, the multi-thread cost overhead.
From the Speed Up table (Table 1) we observe instead that the parallelization strategy is much more effective on K-Means (where the outer cycle is far more impacting than the inner one), reaching almost the ideal multiplier of x4. On K-Medoids, the scaling brings to an empirically measured $\mathcal{O}(\frac{n^2}{\log T})$.

3.2 Kmeans++ and Silhouette score results

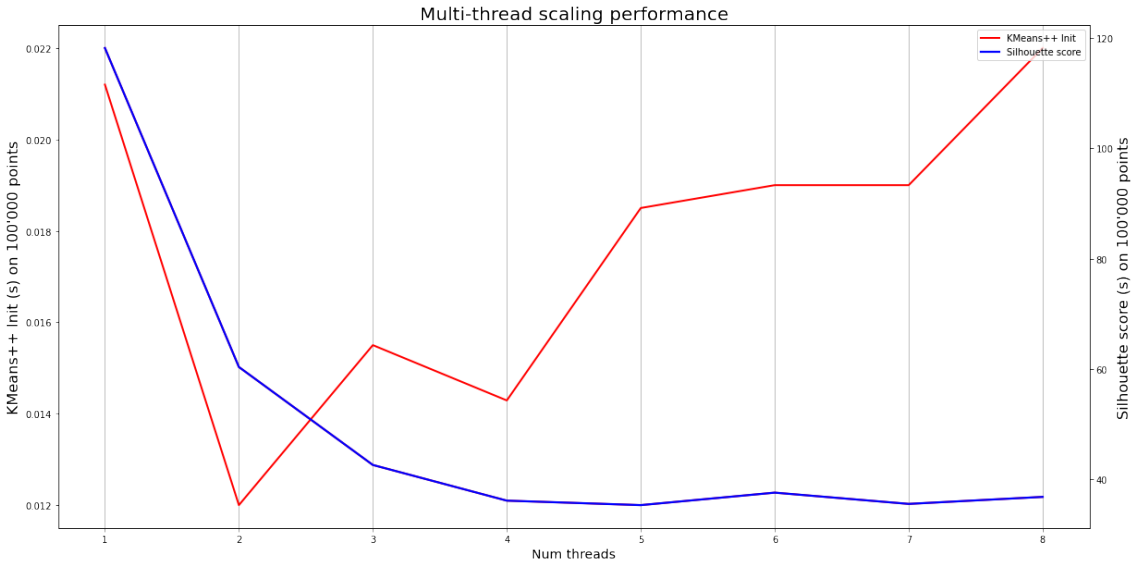

Figure 2: Silhouette score and KMeans++ Initialization performance increase in a multi-thread setting.

Also in this case we can notice a very different behaviour for the multi-thread implementation of Kmeans++ (a 'cheap' algorithm) and the Silhouette score computation (much more complex). The number of physical threads represents a clear upper bound to speed up, as we can see from the elbow in the blue line around T=4. The red line indicates an unstable trend for Kmeans++, which shares the same complexity as K-Means except for the *num_iter* factor. This missing multiplier would 'dilute' the one-time costs, highlighting the parallel gain.

4 Conclusions

To summarize, we can state that a parallel implementation boosts the performance specially for algorithm requiring a considerable computational effort. This is proven by the analogous decaying graphs of K-Medoids and Silhouette Score having a quadratic complexity. On the other side, multi-threading is not for free and it needs to be finely tuned in case of sub-quadratic algorithms in order to identify the correct trade-off.
This report does not consider the memory footprint of parallelization, which can be an interesting development for a deeper comparison.