Sequential vs Parallel K-Means Clustering Algorithm

Davide Tarlini
E-mail address
davide.tarlini@edu.unifi.com

1. Introduction and Problem Statement

K-Means algorithm is one of the most widely used clustering algorithms. It is an unsupervised learning algorithm that partitions a dataset into a predefined number of groups, or clusters, based on the similarity of data points. The goal of K-Means is to find groups of points such that the points within each group are as similar as possible, while points in different groups are as dissimilar as possible. More formally, we describe the problem of K-Means clustering in the following way: given a set X of n data points $x_1, ..., x_n$, with $x_i \in \mathbb{R}^d$, the goal is to find a partition of X in a desired number k of clusters $C_1, ..., C_k$ such as to minimize the *within cluster sum of square distances* (WCSS):

$$\arg\min_{C} \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} ||\mathbf{x} - \boldsymbol{\mu}_i||^2 \tag{1}$$

1.1. K-Means Algorithm

The K-Means clustering problem as defined by the objective function above is an NP-hard problem, and many algorithms to find approximate solutions have been designed. A commonly used approach, and the one taken as reference in this report is based on an iterative refinement of the clustering by alternating between two steps:

1. Assignment Step: assign each data point to the cluster with the nearest centroid, that is the one with the smallest squared euclidean distance.

$$\begin{aligned} C_i^{(t)} = \{x_p : ||x_p - m_i^{(t)}||^2 \\ \leq ||x_p - m_j^{(t)}||^2 \ \forall j, 1 \leq j \leq k\} \end{aligned} \tag{2}$$

2. Update Step: recalculate the values of the centroids of the newly formed clusters.

$$m_i^{(t+1)} = \frac{1}{\left|C_i^{(t)}\right|} \sum_{x_j \in C_i^{(t)}} x_j \tag{3}$$

To kickstart the method we select an initial set of centroids (means) $m_1^{(0)}, ..., m_k^{(0)}$. The two steps defined above are repeated until a chosen convergence criteria has been met.

1.2. Objective of the project

The aim of this project was to study experimentally the performances of different implementations of the K-Means clustering algorithm, especially the possible gains due to parallelization, vectorization and other implementation choices and the usage of an implicit threading framework, in this case OpenMP.

2. Implementations

A simple sequential implementation follows very closely the steps delineated in equations 2 and 3.

Algorithm 1 K-Means Clustering (Sequential)

```
Require: Number of clusters $k$, points $P$, init centroids $C$
 1: $\varepsilon \leftarrow 0.1, T \leftarrow 100$
 2: for $t = 1$ to $T$ do
 3:     Initialize $C'_j \leftarrow \vec{0}$, $S_j \leftarrow 0$ for all $j$
                                                ▷ Assignment Step
 4:     for $i = 1$ to $n$ do
 5:         $j^* \leftarrow \arg\min_j dist(p_i, C_j)$
 6:         $C'_{j^*} \leftarrow C'_{j^*} + p_i$
 7:         $S_{j^*} \leftarrow S_{j^*} + 1$
 8:     end for
                                                ▷ Update Step
 9:     for $j = 1$ to $k$ do
10:         if $S_j > 0$ then
11:             $C'_j \leftarrow C'_j / S_j$
12:         end if
13:     end for
14:     $movement \leftarrow \sum_{j=1}^{k} dist(C'_j, C_j)$
15:     $C \leftarrow C'$
16:     if $movement \leq \varepsilon$ then
17:         break
18:     end if
19: end for
20: return $C$
```

C' is a vector containing the new centroids computed at every iterations, and S is a vector keeping track of each cluster size. The algorithm ends either after T iterations of cluster assignments and updates or when the total change in the new centroids with respect to the old ones is less than an user-defined ϵ. A first modification we could try is the introduction of some vectorization. By encoding each point as an array of coordinates we could try to explicitly tell our processor to vectorize the computation of the distances between points. The (squared) euclidean distance functions used for the non-vectorized code is shown in Algorithm 2.The vectorized counterpart needs only the usage of an OpenMP directive,*#pragma omp simd reduction(+= sum)*, before the loop.

Algorithm 2 Squared Euclidean Distance

```
1: function DIST($p_1, p_2$)
2:     $sum \leftarrow 0$
3:     for $i = 1$ to $d$ do
4:         $diff \leftarrow p_1[i] - p_2[i]$
5:         $sum \leftarrow sum + diff^2$
6:     end for
7:     return $sum$
8: end function
```

Algorithm 3 Parallel K-Means (Critical Region)

```
Require: Number of clusters $k$, points $P$, init centroids $C$
1: $\varepsilon \leftarrow 0.1$, $T \leftarrow 100$
2: $assignments[i] \leftarrow 0$ for all $i$
3: for $t = 1$ to $T$ do
4:     Initialize $C'_j \leftarrow \vec{0}$, $S_j \leftarrow 0$ for all $j$
                                        ▷ Assignment Step
5:     for all $i \in \{1, \ldots, n\}$ in parallel do
6:         $j^* \leftarrow \arg\min_j dist(p_i, C_j)$
7:         $assignments[i] \leftarrow j^*$
8:     end for
                                        ▷ Thread-local accumulation
9:     Each thread $t$ maintains a $prvC_{t,j}$ and $prvSizes_{t,j}$ for each cluster $j$
10:    for all $i \in \{1, \ldots, n\}$ in parallel do
11:        $j \leftarrow assignments[i]$
12:        $prvC_{t,j} \leftarrow prvC_{t,j} + p_i$
13:        $prvSizes_{t,j} \leftarrow prvSizes_{t,j} + 1$
14:    end for
                                        ▷ Merge thread-local centroids
15:    begin critical section
16:    for $j = 1$ to $k$ do
17:        $C'_j \leftarrow \sum_t prvC_{t,j}$
18:        $S_j \leftarrow \sum_t prvSizes_{t,j}$
19:    end for
20:    end critical section
                                        ▷ Update and check convergence
21:    $movement \leftarrow \sum_{j=1}^{k} dist(C'_j, C_j)$
22:    $C \leftarrow C'$
23:    if $movement \leq \varepsilon$ then
24:        break
25:    end if
26: end for
27: return $C$
```

2.1. Parallelization opportunities

The k-means algorithm offers significant parallelization opportunities. A very natural route to follow is to parallelize the assignment step over the points and then to separately parallelize the computation of the new centroids.

Algorithm 3 shows this parallelization. Notice how the new centroids are computed: we allocate two private vectors for each thread, $threadC_{t,j}$ and $threadSizes_{t,j}$, in wich each thread separately accumulates the informations of the assigned points. A critical region is then entered by each thread to finally accumulate all the local results. A possible alternative implementation of this second step of the algorithm is possible in wich we do not require a critical section for the accumulation step. The idea is shown in Algorithm 4.

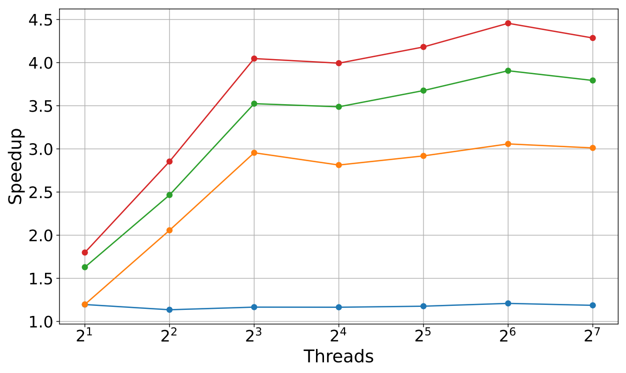
Algorithm 4 Parallel K-Means (No Critical Region)

```
Require: number of clusters $k$, points $P$, init centroids $C$
1: $\varepsilon \leftarrow 0.1$, $T \leftarrow 100$
2: $assignments[i] \leftarrow 0$ for all $i$
3: for $t = 1$ to $T$ do
4:     $C'_j \leftarrow \vec{0}$, $S_j \leftarrow 0$ for all $j$
                                              ▷ Assignment Step
5:     for all $i \in \{1, \dots, n\}$ in parallel do
6:         $j^* \leftarrow \arg\min_j dist(p_i, C_j)$
7:         $assignments[i] \leftarrow j^*$
8:     end for
                  ▷ Thread-local accumulation without private arrays
9:     A shared array that contains one $thC_{t,j}$ and $thSizes_{t,j}$ for each thread $t$ and cluster $j$
10:    for all $i \in \{1, \dots, n\}$ in parallel do
11:        $j \leftarrow assignments[i]$
12:        $thC_{t,j} \mathrel{+}= p_i$
13:        $thSizes_{t,j}++$
14:    end for
                  ▷ Merge thread-local centroids sequentially
15:    for $t = 1$ to $Total\_Threads$ do
16:        for $j = 1$ to $k$ do
17:            $C'_j \leftarrow \sum_t thC_{t,j}$
18:            $S_j \leftarrow \sum_t thSizes_{t,j}$
19:        end for
20:    end for
21:    $movement \leftarrow \sum_{j=1}^{k} dist(p_i, C_j)$
22:    $C \leftarrow C'$
23:    if $movement \le \varepsilon$ then break
24:
25:        return $C$
```

Instead of allocating private centroids arrays and cluster sizes arrays we instantiate them in the sequential part of the code and then we parallelize the update of new centroids and let each thread access the ”local” arrays via their id, wich is always associated to a thread in a parallel region. The final aggregation of the local arrays is performed sequentially without needs of synchronizing some threads. After having performed some experiment, an implementation of algorithm 4 with *dist_SIMD* has also been tested.

3. Experiments and Results

The different implementations have been tested in clustering 8 dimensional points. The generation process consisted in the uniformly random creation of k independent 8 dimensional gaussians from wich points have been sampled for each run of the experiment. Different configurations of total number of points to be clustered and thread allowed to be spawned by OpenMP where tested. For each configurations 10 runs where executed. The experiments have



[H]

Figure 1. Mean speedups of the methods for datasets of 10^6 points. Red = parallel + no crit + simd; Green = parallel + no crit; Yellow = parallel + crit; Blue = sequential + simd

been conducted on a machine with a quadcore intel CPU i7-7700HQ.

3.1. Results

As expected parallelizing the code leads to some speedup. Also, the elimination of the critical section between the threads in the accumulation phase of the centroids seems to be a good implementation choice. The usage of the simd and reduction directives of OpenMP, especially for the computation of distances between points, adds further to the performance. As we can see in figure 1 and 4, the three parallel versions have speed ups that perform qualitatively similarly along the x-axis. Also, the bigger the set of points, the higher the maximum speedup obtained. It is also interesting to note, by loking at figures 1,3 and 4, how the increase in number of points leads to an increase in the best performing thread configurations by moving from 2^3 at 10^4 points to 2^5 at 10^5 points and what seems to be possibly the start of the ascending phase of the speedups at 2^7 threads for the scenarios with 10^6 points.

At 1000 points we are already into the dataset sizes in wich there is no increase in performance given the implemented strategies. Indeed, all the parallel versions of the algorithm perform worse as soon as we start to increase the threads even lightly.

4. Conclusions

K-Means is an algorithm that offers quite good opportunities for parallelization and seems to also benefit from the explicit vectorization of some of the internal computations. OpenMP is a very powerfull framework for the implicit parallelization and vectorization of C++ code, especially considering its easiness of use.

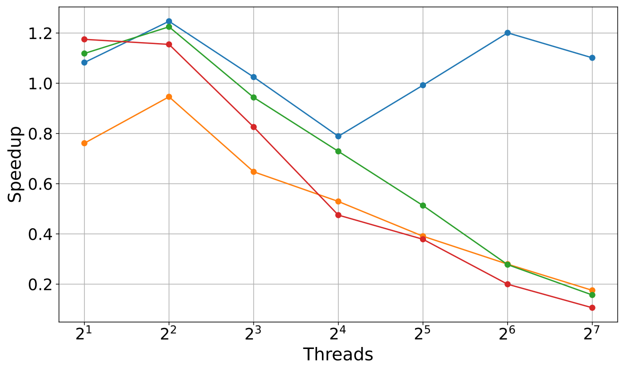


Figure 2. Mean speedups of the methods for datasets of 1000 points. Red = parallel + no crit + simd; Green = parallel + no crit; Yellow = parallel + crit; Blue = sequential + simd

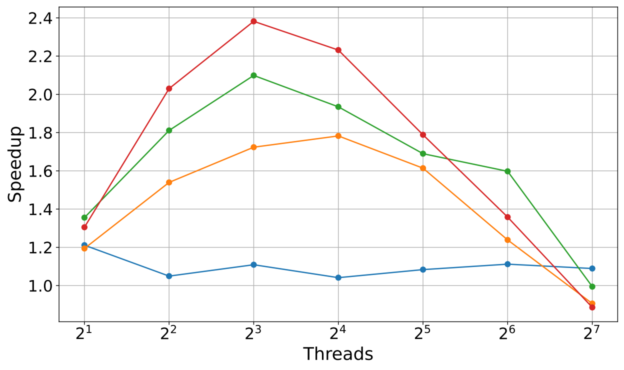


Figure 3. Mean speedups of the methods for datasets of 10^4 points. Red = parallel + no crit + simd; Green = parallel + no crit; Yellow = parallel + crit; Blue = sequential + simd

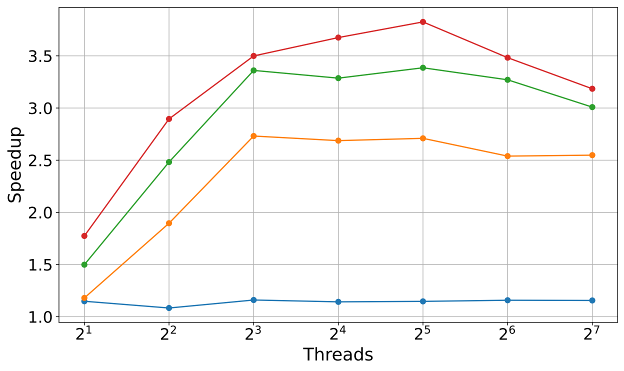


Figure 4. Mean speedups of the methods for datasets of 10^5 points. Red = parallel + no crit + simd; Green = parallel + no crit; Yellow = parallel + crit; Blue = sequential + simd