Parallelization of K-Means Algorithm

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

Abstract

This project aims to implement, apply, and evaluate the performance of a parallel version of the K-means clustering algorithm programmed in C++. The focus is on wisely assigning tasks and the data involved in the algorithm to different threads, and managing the overhead costs generated by the fork and join of them. This goal is reached with the OpenMP framework, and the evaluation is based on the reached speedup. *The coherence of the parallel algorithm with the same clusters as the serial one, and the analysis and plots are made by importing outputs in R Studio.*

Future Distribution Permission

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

1. Introduction

This section will cover the preliminary information we need before turning to the details of the implementation and the analysis of performance. As we can observe, the main characters of this work are the K-means algorithm and the parallelization of a program with OpenMP. Finally, a little part of this Section is dedicated to the metric I adopt to evaluate the performance of the code, the *speedup*.

1.1. K-means

The K-means algorithm is a clustering strategy based on the computation of k centroids, that is, k points that represent k different classes. The assignment of one training point to the cluster is performed by simply computing distances (Euclidean in our case) between that point and every centroid, and by choosing the class of the closest centroid. Focusing on the choice of centroids, we have many possibilities, but I chose to randomly select k training points. Then, once we obtained k clusters, we compute the new centroids as the average point of every cluster. For the convergence condition, we have several possibilities as well. The idea I follow is to continue with a clustering step until the number of points that change cluster is 0.

Formally, considering data as $X = (x_i)_{i=1}^{N}$ with $x_i \in \mathbf{R}^D$ representing quantitative variables, and given the centroids $C_1, ..., c_k \in \mathbf{R}^D$, the algorithm consists of the following two steps:

1. Compute, for every $i = 1, ..., N$ and $j = 1, ..., k$,

$$d(x_i, C_j) = \sqrt{\sum_{l=1}^{D} \left(x_{il} - C_{jl}\right)^2}. \quad (1)$$

   The class $k_i \in 1, ..., k$ is chosen with

$$k_i = \underset{j=1,...,k}{\arg\min}\, d(x_i, C_j). \quad (2)$$

2. New centroids are computed, for $j = 1, ..., k$, with

$$C_j = \frac{1}{N_j} \sum_{i:k_i=j} x_i, \quad (3)$$

where N_j is the number of points assigned to the class j.

More details and other versions can be found in Chapters 13-14 of Hastie, Tibshirani, and Friedman (2009) [1].

1.2. OpenMP directives

This Subsection is not a review of all the directives and the clauses, but only a little summary of the one I used in the code.

Parallelizing the algorithm means using multiple threads to achieve a faster version of it. This goal is reached in OpenMP, and in this work, starting with the generation of several threads with the directive

```
#pragma omp parallel
```

and specifying the required number of threads and the sharedness of the variables between different threads.

Then tasks are assigned to threads with other directives. I mostly used parallelization of for loops, with

```
#pragma omp for
```

This directive assigns indexes of the loop to different threads and, after every thread has finished its work, it sets a *barrier*, a point where all threads wait for the others to finish. The `nowait` clause disables this barrier.

Sometimes we need to ensure synchronization, and we can manually set a barrier with

```
#pragma omp barrier
```

When we need only one thread to execute a section, we can force it with

```
#pragma omp single
```

that makes the code be executed by the first thread that reaches that point.

Finally, I have a *critical* section, defined by

```
#pragma omp critical
```

This part of the code is executed by one thread at a time, avoiding different threads from accessing the same memory location (usually for shared variables) at the same time and ensuring a computation free from the so-called *race condition*.

Those and more directives and clauses can be found in Chapter 7 of Robey and Zamora (2021) [2].

1.3. Speedup

The metric I used for the evaluation of performance is *speedup*, or, more precisely, the *parallel speedup* or *serial-to-parallel speedup* of Robey and Zamora (2021) [2, p. 31]. In Robey and Zamora (2021) [2, p. 9], the speedup is defined as the reduction of the run time of a program and is computed, as we can see at page 11, with the run time of the serial version divided by the run time of the parallel one. In case we are dealing with fixed-size data, Amdahl's Law holds, limiting the value of the speedup because of a non-parallelizable fraction of the algorithm.

2. The code

In this Section, I refer to the code of this GitHub repo, to explain the programming choices I made in the programming phase. The first Subsection explains the serial code, following steps introduced in Section 1.1. A second Subsection focuses on the choices in the parallel version and on the differences from the serial one.

2.1. K-means serial algorithm

The serial version of the algorithm strictly reflects the structure of the one in Section 1.1. Given k as a fixed parameter, the dimension of the dataset N, and v, a matrix containing N two-dimensional points, the first part generates k indexes $p_1, ..., p_k$ between 0 and $N-1$ and sets $v[p_i]$, $i = 1, ..., k$ as starting centroids. Then, after initializing a counter to

```
change=N;
```

a `while` loop starts with the condition

```
while(change=>0)
```

The counter `change` will contain the number of points that change cluster after a step of the algorithm. I started with a less restrictive condition, but I found that one to be reasonable.

Then, we find a for loop on the training points and another one for the centroids. Inside this double loop, I compute the distance in Equation (1). During the same loop, the minimum distance (for a fixed training point) is stored together with its position between 0 and k (the relative class). Then I assign this class to the point with an additional vector of dimension N, updating the number of points that change cluster if it happens, and I update a vector of partial summation and a vector of counters.

Once the double loop is finished, we need to compute Equation (3), so we need the sum, divided per component, of the points of every cluster and the number of points in it. This is made by the previous code, inside the double loop. In a separate for loop over clusters, I compute both the components of new centroids by just dividing the sums by the counters.

2.2. K-means parallel algorithm

This version of the algorithm obviously presents some changes from the serial one, but maintains the same structure.

I declared centroids outside the parallel section, together with the declaration of some variable that will be shared between every thread. Then I generate threads only once, before the while loop, to reduce the overhead of the fork and join. I declared private counters and sums, similar to those of the serial algorithm. This privatization manages a handmade reduction to reduce the use of shared counters and partial summations that can result in a race condition.

Every thread enters the while loop and resets (sets if it is the first step) to 0 the local counters and sums, then I set a barrier to ensure that the following code would be executed only when every thread has fulfilled the while condition. Then, I reset to 0 the shared counter of points that change the cluster inside a single section (only one thread executes this), and I reset the shared counters in a parallel for loop. Even though k (the number of clusters) is much smaller than N (the dataset dimension), k can be chosen arbitrarily big, and I pay no costs if I use a parallel loop instead of a serial one, so I decided to parallelize it.

The main corpus of the algorithm is constituted by a parallel for (with nowait clause to avoid synchronization) that works like the serial algorithm but updates only private counters and partial sums. Then, a critical section guarantees that, once a thread has scanned its portion of data, it updates the shared counters and sums accessing the shared variables without race conditions. After this chunk, I set a barrier to ensure that every shared variable is correctly updated before computing the new centroids.

Another parallel for computes k new centroids, one for every cluster, like in the serial version.

3. The experiment

The experiment consists of a repeated test of the algorithm on a dataset to evaluate the speedup with different numbers of threads. Then, the results are evaluated with R. In this Section, I explain the structure of the experiment and report results and plots.

3.1. Repetition of tests

The project runs $Nrep = 100$ times a chunk of code, where I set the number of threads to 2^p, $p = 0, 1, 2, ..., 8$, resulting in 900 total tests. Here, I computed the run time of completing the serial algorithm. Then, I performed the parallel algorithm computing the relative time. The speedup is computed as the division of those two times. At the end of the process, I have 900 speedups corresponding to 100 speedups for every number of threads between 1 and 256.

To test the speedup behaviour with an increasing number of points, I repeated the experiment with 9 datasets generated with Python, with a combination of $k = 10, 20$ and 40 clusters and $N = 10^4, 10^5$ and 10^6, but with only ten repetitions for every dataset. With these datasets, I set a different convergence criterion, stopping the algorithm when the number of points that change cluster is less than 1% of N.

3.2. Data

The first dataset is sourced from Kaggle, with enough data to ensure a run time long enough. This consists of 21600 two-dimensional points naturally organized into 12 clusters, as shown in Figure 1. I arranged them in a *csv* file of two columns. The output of the program is made of every input point, followed by the class in which they are assigned by the serial algorithm and by the parallel algorithm. Then I collected the speedup data in a different *csv* file.

Randomly generated datasets are generated with a Python code choosing k and N, then imported through a *csv* file. One of these is shown in Figure 3,

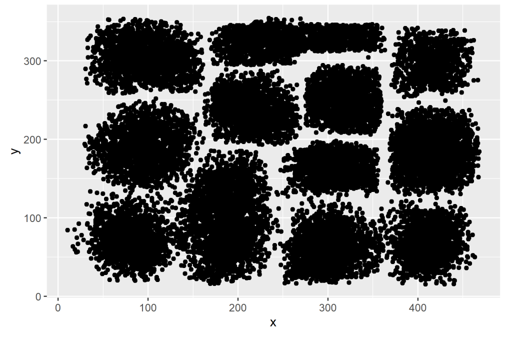


Figure 1. Input data of the experiment.

3.3. Analysis

The analysis is performed in R. A first test checks that the assignment of the serial algorithm is the same as the parallel one. The clusters are shown in Figure 2. The speedup analysis is then performed graphically in Figure 4 and more detailed in Table 1. As we can easily see in Figure 4, I didn't obtain a linear speedup, and this effect is more evident with a growing number of threads.

For the randomly generated datasets, I reported

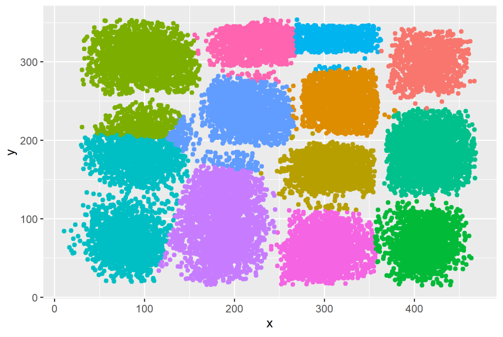


Figure 2. Output clusters of the experiment.

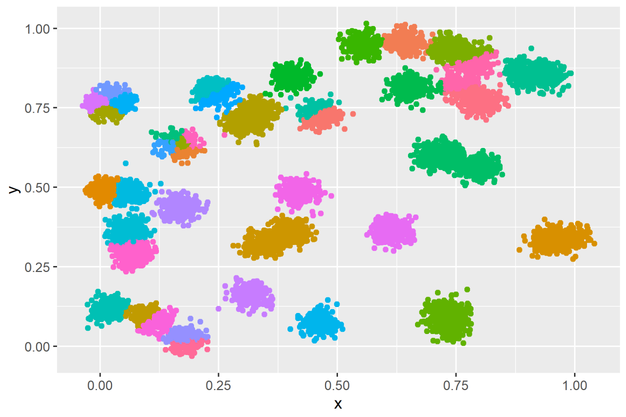


Figure 3. Randomly generated dataset and relative clusters for $k = 40$ and $N = 10^4$.

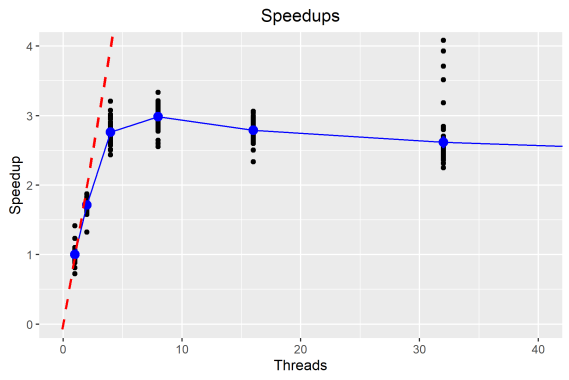
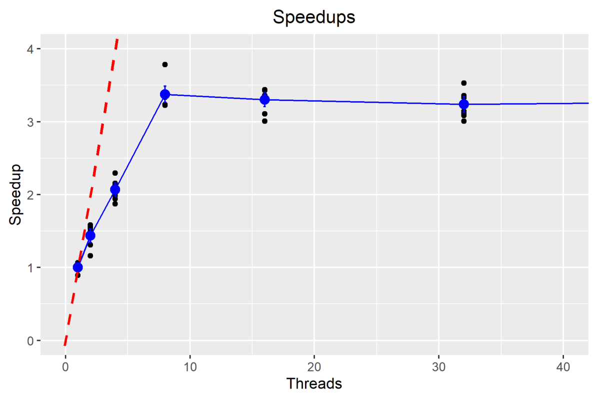


Figure 4. Speedups of the experiments of the Kaggle dataset. The dashed line shows the linear speedup, and the blue broken line shows the obtained speedup. The big blue point represents the average speedup given the number of threads.

results in Table 2, 3, and 4, respectively for $k = 10$, 20, and 40. I obtained the best results for $k = 10$ and 20 and $N = 10^5$ and for $k = 40$ and $N = 10^4$. The first two cases are shown in Figure 5.



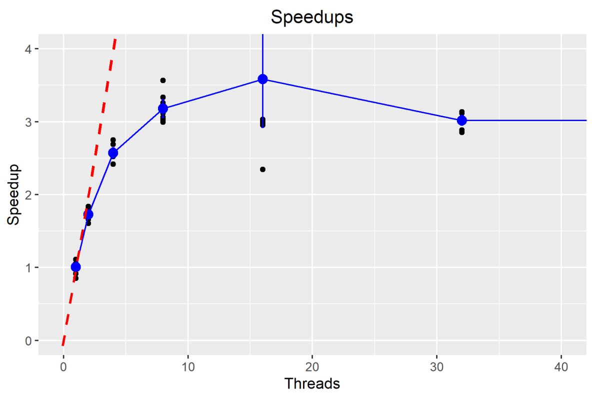


Figure 5. Speedups for $N = 10^5$ for $k = 10$ (left) and $k = 20$ (right).

Threads	Mean Speedup	Lower	Upper
1	1.00	0.988	1.01
2	1.70	1.70	1.73
4	2.76	2.74	2.79
8	2.99	2.96	3.01
16	2.79	2.77	2.81
64	2.43	2.42	2.45
128	1.99	1.97	2.00
256	1.41	1.40	1.41

Table 1. Speedups. The first column contains the number of threads, the second the average speedup over 100 tests with the same number of threads. The third and fourth are respectively the lower and upper bounds of a 95% confidence interval for the mean in column two.

Threads	$N = 10^4$	$N = 10^5$	$N = 10^6$
1	0.974	1.00	0.994
2	1.76	1.44	1.70
4	2.21	2.07	2.49
8	2.65	3.38	3.00
16	2.69	3.30	2.92
32	2.47	3.24	2.96
64	1.92	3.29	3.00
128	1.36	2.96	3.04
256	0.768	2.59	3.25

Table 2. Speedups for $K = 10$.

Threads	$N = 10^4$	$N = 10^5$	$N = 10^6$
1	1.01	1.00	0.982
2	1.84	1.73	1.71
4	2.70	2.57	2.52
8	3.01	3.18	2.95
16	2.55	3.58	2.89
32	2.63	3.02	2.90
64	2.18	3.03	2.92
128	1.64	2.87	2.91
256	1.22	2.63	2.88

Table 3. Speedups for $K = 20$.

Threads	$N = 10^4$	$N = 10^5$	$N = 10^6$
1	1.01	1.00	1.01
2	1.70	1.73	1.71
4	2.08	2.58	2.54
8	3.33	2.98	2.92
16	3.37	2.90	2.91
32	3.09	2.94	2.91
64	2.97	2.93	2.90
128	2.41	2.88	2.90
256	1.85	2.77	2.89

Table 4. Speedups for $K = 40$.

4. Conclusion and comments

The speedup I obtained is good (not excellent), and it increases with the dimensions of the dataset. I minimized the overhead of the fork and join of threads, defining them only once, but then, the cost of managing shared variables is still high, and that doesn't guarantee better performance.

Another aspect that deserves more attention is the algorithm that charges data. I tried to use a parallelized version of this, but I obtained a worsening of performance due to the sharedness of the source file. I decided to maintain the serial algorithm for this part because the impact on the total time was too low, but a parallelized version of it should guarantee a better distribution of data in memory. In a NUMA system like that one, the thread can upload its portion of data, and that would be the same data it works on during the algorithm. This aspect can lead to a better result in terms of speedup.

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. 2

[2] R. Robey and Y. Zamora. *Parallel and High Performance Computing*. Manning Publications Co. LLC, New York, 2021. 2