Parallel K-Means Clustering

A parallel implementation of Hamerly's algorithm using OpenMP and MPI in C++

Filippo Pellizzari, *University of Trento*
Parallel programming course
filippo.pellizzari@studenti.unitn.it (237822)
https://github.com/filippopellizzari1/Kmeans

***Abstract*—In this report, two k-means algorithms (Lloyd and Hamerly) are compared. Moreover, for Hamerly's version, a parallel implementation is presented and tested, using both a distributed memory (MPI) and a shared memory (OpenMP) framework in C++, to see the speedup over the standard case scaling with respect the number of points (N), the number of centroids (K) and the data dimensionality (D). Introducing a balancing strategy for the parallel execution.**

I. Introduction

A clustering algorithm is an algorithm designed with the purpose of grouping, into an arbitrary number of classes, a set of points. Such classes are created by grouping together those points that have common features. Thus, the goal of this category of algorithms is to build knowledge about the data provided as input that makes it possible to find hidden relationships among them which can be used to classify a point whose class is not known a priori.

II. Classical (or Lloyd) algorithm

A. Pseudocode

The following pseudocode describes the basic implementation (lloyd algorithm) for the k-means clustering.

Algorithm 1 Lloyd algorithm

Repeat untill converge
for each point **do**
    Calculate distance from every centroid
    Assign the point to the nearest centroid
end for
for each centroid **do**
    Calculate the new centroid by taking the mean of all points assigned to it
end for

In this project, the distance is meant as Euclidean distance.

The convergence criterion could be the point when the centroids don't move more than a certain threshold or, as it is used in this project, after a fixed number of iterations.

B. Tracking points during assignment

The first, rather small, improvement with respect to the classical algorithm can be done during the point assignment to the closest centroid. In this project, two arrays have been introduced: the first one to track the number of points belonging to a certain class and the second one to store the sum of all coordinates of the points in every class (this latter has a dimension of $K \cdot D$). In this way, during the centroids update it is not necessary to iterate through all the points as the values needed to compute the average are already present inside the two arrays. Below is the pseudocode on how the point tracking has been handled in this project.

Algorithm 2 Point tracking

After the closest centroid has been found
if Closest centroid is different from the old closest one **then**
    Decrement number of points in previous class and increment the new by one
    Remove coordinates from old class and add them to the new one
end if

III. Hamerly's Algorithm

A. Triangle inequality

Hamerly's algorithm [1] is an optimization of the standard algorithm and exploits the fact that it is not necessary to recalculate the distance of each point from the centroids each time the latter are repositioned. Using the triangle inequality it is, in fact, possible to identify some points as criticals. Only for those points it is necessary to recalculate the closest centroid, while for the others, the previously assigned centroid is still the closest one. The triangle inequality states that given 3 points A, B, C $\in \mathbb{R}^N$ then

$$|d(A,C) - d(B,C)| \leq d(A,B) \leq d(A,C) + d(B,C) \quad (1)$$

B. Upper and lower bound

When a point is assigned to a new class, it tracks the distance from the closest centroid (upper bound, ub) and the distance from the second closest one (lower bound, lb). At this point, every time the centroids are updated, the lower and upper bounds of every point have to be updated as in the following pseudocode.

Algorithm 3 upper and lower bound update

```
ub += distance moved by point.centroid
if points.centroid == centroid which moved the most then
    lb -= second max distance moved by a centroid
else
    lb -= max distance moved by a centroid
end if
```

These updates are justified by Eq. (1) because, after the centroids have moved, the point won't by any farther than ub from its centroid and won't be any closer than lb to the second closest one. In this way, critical points are the ones for which $upper\_bound > lower\_bound$

C. Time complexity

In the case of Hamerly's algorithm, time complexity required to complete the classification is:

$$D \cdot K \cdot N + mD(\phi N \cdot K + (1 - \phi)N) \tag{2}$$

Where m is the number of iterations and ϕ is the percentage of critical points with respect to the total number of points. The improvement over the standard algorithm lies in the fact that only the first time it is necessary to compute all the distances, whereas in subsequent iterations, for most points, there is no need to recompute the nearest centroid.

IV. Parallel Hamerly (OMP)

Hamerly's algorithm can highly exploit parallelization to improve its performance. Indeed, when calculating the closest centroid for each point or when updating the centroids, it is possible to partition the points (in the first case) or the centroids (in the second case) between the threads to parallelize the calculations. This is true at least for the first iteration. However, from the second one onward, partitioning the points is not as trivial because only for critical points need to be calculated the closest centroid. Because of this, simply splitting the points could lead one thread to have more criticals than another one.

A. Partitioning points between the threads (1st iteration)

If there are a total of T threads, the number of points that each thread will access is: $n = Int(N/T)$ Where Int(x) is the integer part of x. Moreover if the thread_id is lower than $N \ mod \ T$ the number of points must be incremented by one. While iterating through the points, the access pattern will be $p = points[thread\_id * n + off + j]$ where $off = N \ mod \ T$ if $thread\_id \geq N \ mod \ T$, or $off = 0$ otherwise.

B. Load balancing

After the centroids are updated, Hamerly's algorithm goes through all the points to update their lower and upper bounds. The load balancing strategy can be implemented during this phase as described by the following pseudocode.

After the balancing strategy, from the second iteration onward, the critials array can be partitioned between the threads, therefore giving each one the same workload.

Algorithm 4 Balancing strategy

```
                                          ▷ In parallel
for each point assigned to the thread do
    bounds update
    if ub > lb then
        Add the point the local critical list
    end if
end for
                                          ▷ Sequentially
create array that will contain all the criticals found
                                          ▷ In parallel
for each critical found in thread do
    add it to the global array
end for
```

V. MPI Version

In the MPI version of the parallel Hamerly's algorithm, the idea is still the same used in the OMP version, i.e. for the first iteration, split the points between the threads and from the second one onward, partition the criticals found using the balancing strategy. For the centroids update just partition them between the processes.

But, since every process has its own local storage and cannot access other's memory, the points and centroids partitioning and the communication during the load balancing phase needs to be handled using messages.

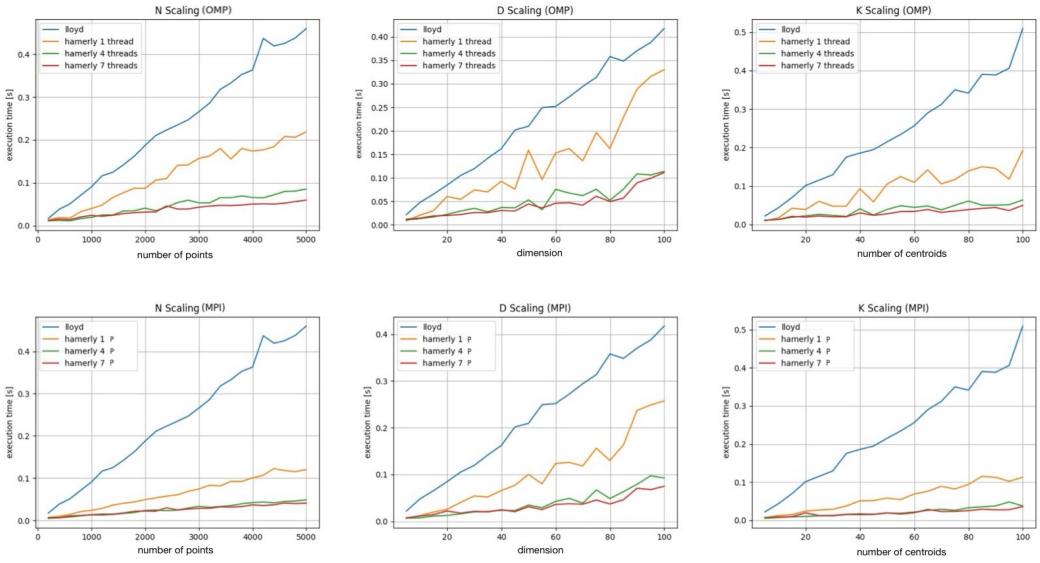
A. The main process

In this implementation, process zero is the main process and contains all the information, i.e. the array with all the points, the array with all the criticals found in the last iteration, the centroids, the array with the number of points per class and the array with the sum of all coordinates per class. Regarding the centroids, each process has also its own copy of the full array because when calculating the class of a point, the process needs to compute all the distances.

B. Scattering the data

In the beginning of each phase (points or criticals assignment, centroids update and bounds update), process zero will scatter the needed items between the processes filling a local array where each process store the assigned ones. The needed items could be points or any other value that is needed to complete the phase, such as the array with the points per class and the average of all points array, for the centroids update. At the end of the work, process zero will gather the items back and update its memory by accounting for the gathered and updated values. In the case of the centroids update, after that process zero has gathered the updated ones, it will broadcast the full centroids array to all other processes to keep their memory updated. Another value that must be shared across all processes is the number of criticals found during the last iteration because it is needed by each process when calculating the number of points it has to access during the critical assignation.

Fig. 1. Speedups results



VI. Testing

A. Scaling over N, K, D

To test how the algorithm scales over the main dimensions a random dataset has been generated for each test (N scaling, K scaling and D scaling). This allows scaling over any dimension without breaking the logic of the dataset (since it is random).

The dimension of the random datasets have been chosen to keep the product $N \cdot K \cdot D$ constant and equal to 200000. Depending on the scaling, the associated dimension should be large enough to allow an appropriate sweep.

In specific, the dimensions used are (5000, 10, 4) for the N scaling, (500, 100, 4) for the K scaling and (500, 4, 100) for the D scaling. The constant 200000 have been chosen to keep a relatively high workload but still not too big to keep the execution times small. Section VII describes what happens when changing that constant.

For every scaling test, the lloyd algorithm has been executed to fix a reference time. After that, the OMP and the MPI version have been executed with 1 to 10 threads (or processes). Every algorithm ran for a total of 20 iteration and is executed for 5 times to average the time results. The process has been repeated each time increasing the tested dimension. The results of these tests are shown in Figure 1 where the first line represent the OMP testing while in the second line are the MPI plots.

As expected the result is a straight line pattern because, no matter which dimension is scaling, Equation 2 can be reduced to one of the type y = mx + q.

B. Speedups

Even with one thread, Hamerly's algorithm shows a significant improvement in the execution time. However, increasing the number of threads results in even better performances as shown in the plots of Figure 2. To calculate the speedups has been used the following formula

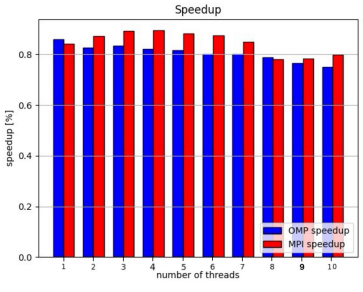
$$su = 1 - Hamerly\_time/lloyd\_time \tag{3}$$

VII. Testing with different workloads

A. workload too low

If the total complexity of the dataset ($N \cdot K \cdot D$) is too low, then increasing the level of parallelism is no longer an improvement because the overhead needed to handle the parallel execution become too large with respect to the overall load and slow down the execution. This behavior has been tested using a real dataset[1] where the goal is to classify, as either bad or good, signals coming from the ionosphere testing the number of free electrons in the latter. The total complexity of the dataset is $N \cdot K \cdot D = 351 \cdot 2 \cdot 34 = 23.869$ which is well below the tested load of 200000.

The speedup graph shows that increasing the number of threads reduces the speedup and therefore the execution times are higher when the number of threads is greater.



[1] https://archive.ics.uci.edu/dataset/52/ionosphere

Fig. 2. OMP vs MPI speedup on medium complexity dataset

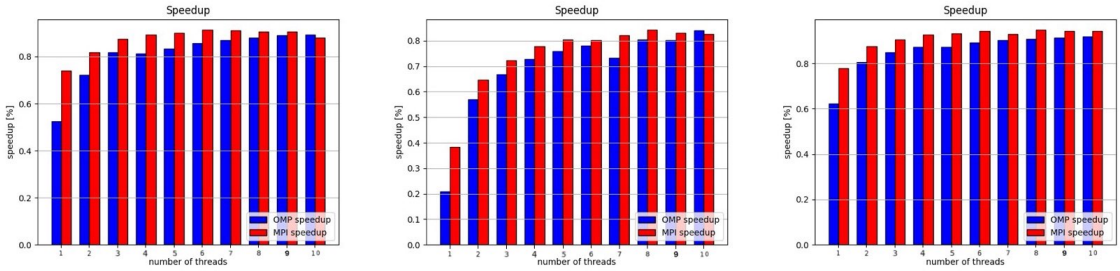
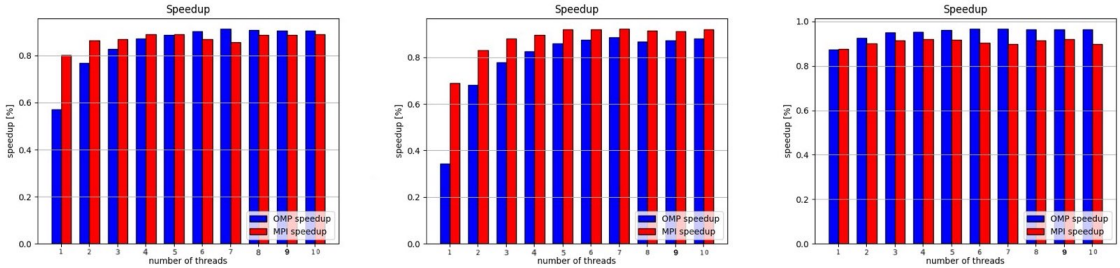


Fig. 3. OMP vs MPI speedup on large complexity dataset



B. Large workloads

Increasing the workload doesn't change the overall behavior of the parallel execution, i.e. both the OMP and the MPI improve by increasing the number of threads, with the MPI one being a little bit faster than the OMP version. This is true even for larger workload given that the number of points is not too high. In the latter case, the OMP version becomes faster than the MPI version and this happens because of process zero in the MPI version. Once it has gathered the updated critical points from the other processes it has to iterate through all of them to update the corresponding points in the points array. Therefore the more points, means the more criticals (particularly in the first phases). This doesn't happen if the number of centroids is high, because process zero can directly gather the updated centroids inside the centroids array, and neither does it occur for the dimension.

To show this behavior, Figure 3 shows three different situations: on the left there is the case of a random dataset with a complexity of (19020, 10, 2) in the center another random dataset with a complexity of (1902, 10, 20) and on the right a downloaded dataset[2] with the same dimension of the first one. In this case the total complexity is $19020 \cdot 10 \cdot 2 = 380400$ for all the cases which is almost twice the complexity of 200000 used for the previous testing.

It can be seen that, even if the total complexity of all the datasets is the same, when the number of points is high then the OMP version is slightly faster than the MPI version. But in the middle case when the number of points is lower then the behavior is the usual one with the MPI being faster than the OMP version.

[2]https://archive.ics.uci.edu/dataset/159/magic+gamma+telescope, high energy gamma particles in an atmospheric Cherenkov telescope

C. Conclusions

In every case, the Hamerly algorithm is faster than the Lloyd version even with one thread.

Increasing the number of threads improves the performace in every case if the complexity of the dataset is large enough.

Finally, increasing the complexity doesn't change the overall behavior (except for higher execution times) but if the number of points is too large than the OMP version runs faster than the MPI one, which is usually not the case for other combinations of N, K and D.

VIII. Platform and computing system description

All the tests were executed on a Windows 11 PC with an AMD Ryzen 7 5800H (3.20 GHz) processor and 16 GB of RAM. To run the MPI code has been used the Microsoft MPI library [2].

IX. Future work

This project can be further improved by implementing some enchangements. In the MPI version, process zero has to do more work than the other processes because it has to manage all the other process by scattering and gathering all the total variables. This could be balanced by partitioning all the variables between the processes even if applying this method would require a more complex strategy to handle the criticals scattering.In addition, a more advanced version of the Hamerly's algorithm optimizes also the centroid repositiong as explained by a team of researchers from the North Carolina State University who introduced this optimization in their paper [3] .

REFERENCES

[1] Partitional Clustering Algorithms by M. Emre Celebi, https://link.springer.com/chapter/10.1007/978-3-319-09259-1_2, Chapter 2.

[2] MSMPI, https://www.microsoft.com/en-US/download/details.aspx?id=57467

[3] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvath, and Todd Mytkowicz, "Yinyang K-Means: A Drop-In Replacement of the Classic K-Means with Consistent Speedup".