High performance computing project report

Parallel Agglomerative Clustering Algorithm

Authors:
Hussein AWALA
Assem SADEK

Supervisors:
Prof. Christophe Picard

February 21, 2019

Contents

1 Summary

In this report, we will introduce a parallel version for the agglomerative clustering algorithm. First, we will give a background about the hierarchical approach of clustering algorithms. Second, we will introduce the challenges faced us to apply parallelism using mpi4py and multiprocessing packages for python. Then, we will explain our approach for the problem giving its advantages and disadvantages. Finally we will give some results and its interpretation.

2 Background

Based on [1], clustering algorithms are always the kernel of pattern recognition for unsupervised learning, one of the sub-field of machine learning. It's known from its simple idea of regrouping close observations based on chosen similarity criteria. This criteria helps in identifying common observations which share the same patterns or similar features values. Normally the similarity criteria is a distance metric, like euclidean distance or Manhattan distance.
One of the common used clustering algorithms is the Hierarchical agglomerative clustering. From it's name it starts by assuming that every observation is a cluster from its own, then it hierarchically/iteratively group together the most nearest two clusters. It continue grouping until we get one single cluster, which contains all the observation points, or it stops at certain a certain number of clusters k, given previously.

Figure 1 shows a example of 20 observations, with two features each, grouped in 4 clusters based on euclidean distance.

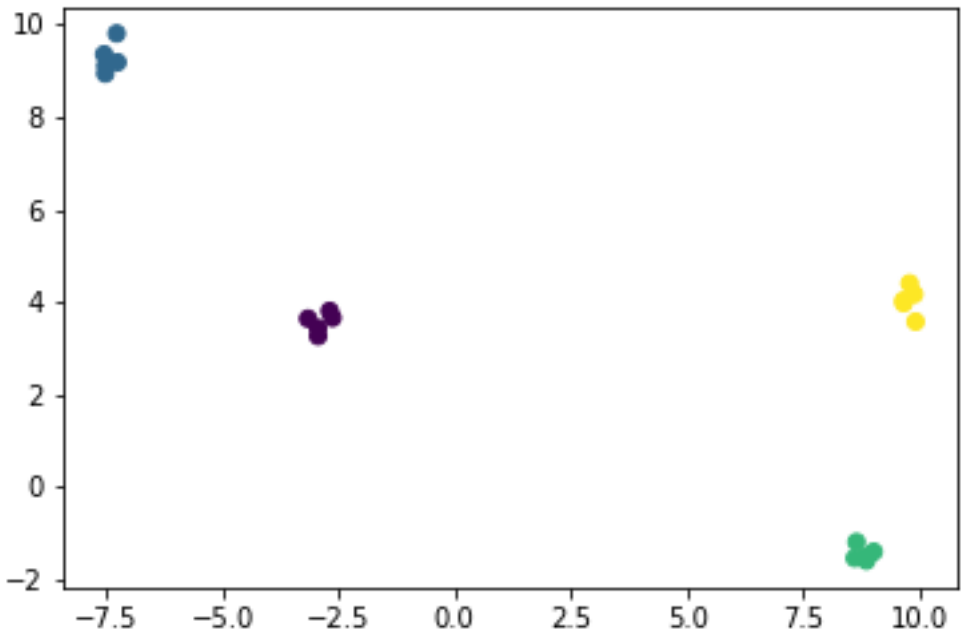

Figure 1: 20 observations grouped in 4 clusters

The pseudo-code in 1 illustrates simply and briefly the basic serial agglomerative clustering algorithm.

Algorithm 1 Hierarchical Agglomerative Clustering

```
1:  procedure HAC(Observations, k)
2:      clusters ← Observations
3:      c ← size(Clusters)
4:      while c > k do
5:          minPair ← getMinPair(clusters)
6:          clusters ← mergeMinPair(clusters, MinPair)
7:          c ← size(Clusters)
8:      end while
9:      return clusters
10: end procedure
```

3 Challenges

It's very clear that our main challenges will be focus on dealing with distributing the calculations of similarities between every two clusters. Let's take the case when we have right now (during the while loop iteration) 10 clusters. This means that we have to calculate "optimally" around 55 similarities in this iteration. We say optimally because we know that the similarity between C1 and C2 is symmetric, so in this case we will have around 55 unique similarities by removing overlapping.

Returning to the main challenge, as we see the challenge is to make the processes aware about what are the unique similarities to be calculated. We need that to avoid any overlap calculation between two processes: process A calculates similarity between C1 and C2 and process B calculates similarity between C2 and C1.

Since we will mainly use message passing paradigm. another thing we need to take into consideration is the latency. We know that the similarity between two clusters won't be simplified to a single distance between two points (this is only at the first iteration), but starting from later iterations, the similarity will be the average similarity between the points in one cluster and the other cluster or the distance between the average point in one cluster and its corresponding in the other cluster. Given this clarification of the common definition for similarity, it won't be practical at all to make every process responsible for calculating the distance between two points. If we do so, it will increase the latency, since we will have a lot of message passing during only calculating the similarity between two clusters. Imagine an example of having example 15 observations in each cluster and we have only 5 workers (processes), we will do around $\binom{15}{2} = 105$ messages passing. This is only two calculate one similarity between two clusters.

Another challenge concerning agreements, processes need to agree at every iteration about the minimum pair and the current length of the clusters and the new clusters themself.

4 Approaches

We used mainly two approaches for parallelism, one implemented using message passing paradigm by mpi4py package and one using multiprocessing package.

4.1 Using mpi4py

For the first approach, we divide the responsibility equally of calculating the similarity between a cluster X and all the other cluster, over all the workers. Looking at the example of figure 2, we have 30 clusters, we will assign for worker 0 the responsibility of calculating the dissimilarity of cluster 0 (C_0) with all the other clusters (from C_1 to C_{30}) and the same for cluster C_1 to C_5. Then worker 1 will take the next 6 responsibility of C_6 to C_{11} and so on.

In this figure, we have to note something that no one has been assigned to take the responsibility of cluster 30. This is meant to be, because we tell every worker to only calculate the similarity between a given C_i and all other C_j with $j > i$.This helps to avoid duplication in calculation of similarity and take benefit from the symmetry of the similarity function. Thus all the similarities of C_{30} are already calculated by previous clusters, so no need to do it.

Another important point, is that here we parallelize the computation and not the distribution of the data, we can notice in the figure that all the workers aware about the current clusters (the variable clusters) and the features vector of every initial

observation (the variable Data). They use this knowledge to help them calculate the similarity. In order to make every workers to be aware of the current clusters (also the length of the clusters) and the data, we broadcast at each iteration the new matrix of clusters to all processes. Thus every worker will know, for example, cluster C_3 has right now the corresponding 7 observations, it will take this into consideration while calculating similarity with this cluster.

After each "responsible" finish its task of calculating the similarities for all assigned clusters, it will choose locally the minimum, from its local knowledge and then return the minimum pair and value to the root. Thus, at the end the root will end up by N pairs (N is the number of workers) and it will do the decision to choose the global minimum which is the minimum of the received local minimum. This is so efficient during message passing, because instead of sending the calculated similarity to the root, which is huge (remember the example of $\binom{15}{2} = 105$), it will send one single pair representative of the minimum from the assigned clusters for this "responsible".

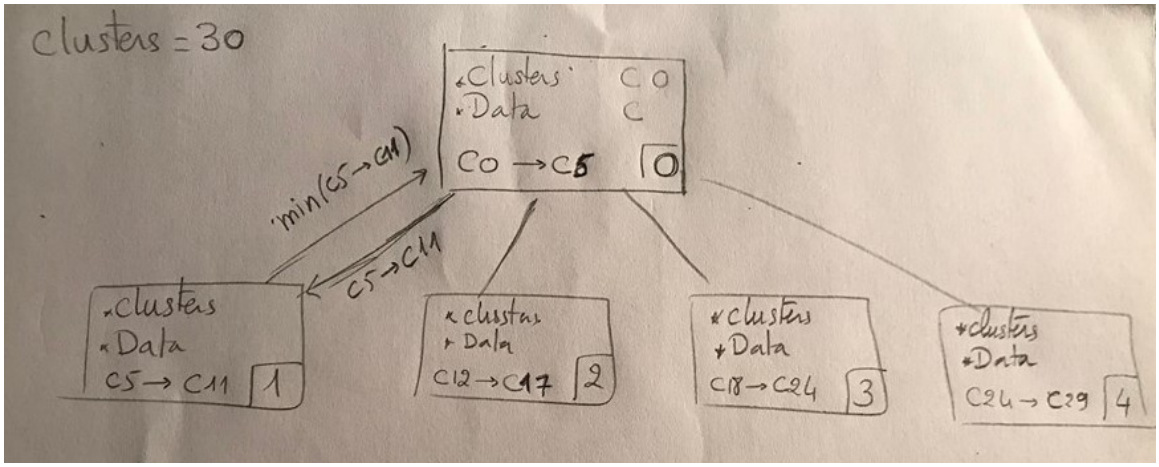

Figure 2: Message passing between workers

4.2 Using multiprocessing

As for this approach, we used the multiprocessing library, we divided the clusters into Number_Of_Cores slice, and we distribute these clusters over the CPUs or the cores available on the computer.
The library create a process on each core, can work independently of the other, each one compute the dissimilarities between the slice and all the clusters, compute the minimum of this dissimilarities, and return it with the indexes of the two clusters that minimize this dissimilarities.
And because these works are independent, we choose to run them in asynchronous apply function, to make the fully independent from each other, each one return their value, and the code add this values to the list, and after finish all the computing, the main process find the minimum between all these values.
The last step is like the other approach, we take the indexes of the two clusters wich have this minimum distance, and we merge them. We repeat this work many time until reach the desired number of clusters.

4.3 Using the workers as a service

In this approach, we changed the idea a bit, we still work with mpi4py but with different implementation, so instead of distribute the computation between all the CPUs, we divided them in two kin, the first is the naive worker, and the other is the exploiter.
Only the root is an exploiter, it create a list of all the possible combinations between the clusters, and start sending this combinations to the worker that don't know anything about the algorithm, they only receive 2 clusters ,return the distance between them, and return this distance to the root.
The root compute the minimum after each distribution, and compare it with the global minimum, and when it find a new minimum, it update the old one, with the indexes of the clusters having this minimum as a distance between them. And finally we merge the two clusters at each iteration.
For the naming of this algorithm, we observe that the worker run all the time until receiving the "STOP" command from the root, and their functions are fix and simple (receive two clusters, compute a distance and return it), and the root exploits these workers to avoid the hard computation of the distance (in the case of the big clusters and the huge number of features), and it only find the minimum between some float numbers wich are the results of these workers, that's similar to the idea of web service, that provide a certain kind of computation or provide some kind of data, and our server use it to avoid the hard computation.

5 Results

Table 1 refers to running our different versions on a clustering problem consists of 50 observations but using different number of features: 2, 100, 1000 and 100000. We focus more about comparing with respect to the number of features because the number of features is the factor that affects the time to calculate the dissimilarity function, so it's important to see how we perform when we parallelize this bottleneck, and if the price we pay for the latency during message passing is deserved or not. THe size of the data doesn't affect much, because at the end, the number of iterations is the same for all versions, since it's related to the number of initial data, so it won't help during the comparison.

For the notation, Parallel V1, corresponds to the first mentioned version with mpi4py. Parallel V2, corresponds to the one with multiprocessing. Parallel V3, corresponds to the workers as service version.

Number of cores	Number of features			
	2	100	1000	1000000
Serial (p=1)	0.393	0.412	0.591	261.14
Parallel V1 (p=1)	0.446	0.419	0.576	**223.2**
Parallel V1 (p=2)	0.344	0.359	0.456	**195.5**
Parallel V1 (p=4)	0.252	0.246	0.272	**215.7**
Parallel V1 (p=6)	0.243	0.239	0.267	**258.77**
Parallel V1 (p=8)	0.190	0.227	0.247	**260.2**
Parallel V2 (p=8)	2.2	2.3	3.5	–
Parallel V3 (p=8)	1.17	1.29	1.68	**414.2**

Table 1: Different runs with different configuration for data of size N=50, the time is given in seconds

For parallel V1, We can notice easily that when we increase the number of processes/cores, the time decrease except in the anomaly of very large features (in bold). At the beginning we didn't have an interpretation for such anomaly, but we figure out at the end that this could be related that, in fact, the calculation of similarities was not fairly distributed a we think. This is simply because when we give for example process 0 the responsibility for C_0 to C_5 and process 5 $C_{24}toC_{29}$, we can see that process 0 has more dissimilarity to calculate than process 5 (remember that we takes benefit from the symmetry to only calculate the similarity between a given C_i and all other C_j with $j > i$). So one way to solve, could be to redistribute the responsibilities differently, for example to give process 0 two clusters from the beginning and two clusters from the end, and keep going like this from the two side when we assigning responsabilities to each process.

For parallel V2, it doesn't give any promising results. This could make sense to us, since multiprocessing package is not optimized for parallelism like mpi4py. In fact multiprocessing is sometimes regarded as wrapper for much lower level of parallelism paradigm like mpi4py. Also we fail to test it on large scale data, because it gives out of memory error, since the CPU memory tries to fill the memory with a lot of coordinates for points at the same time in parallel. Multiprocessing is totally based on memory management.

For parallel V3, It doesn't give as well good results, because we have here high latency, we have a lot message passings that are being used frequently each time the user want to send a specific similiarity to be calculated (This align again with the example of $\binom{15}{2} = 105$)

6 Conclusion

Parallelization of clustering algorithms could be implemented with different aspects, it totally depends what we care to optimized, if we want to optimize similarity calculation, or data partitioning or the decision of choosing the minimum dissimilarity. We can see that the price of latency is not always on our side to get a good performance on large-scale data.

7 List of works

The work load was divided as following:

- First draft of serial version : Hussein Awala
- Final optimized serial version : Assem Sadek
- First parallel version : Assem Sadek
- Data generation and manipulation: Assem Sadek
- Parallel multiprocessing version : Hussein Awala
- Workers as a service : Hussein Awala
- Report : both of us
- Design decision in all algorithms : both of us

References

[1] Dongkuan Xu, Yingjie Tian (2015) *A Comprehensive Survey of Clustering Algorithms.*