Projet LSC

Ellouze Skander

March 2020

1 Introduction

1.1 The K-Means Algorithm

The K-Means Algorithm is widely used in the field of data science, it allows to classify a set of points in K different clusters. In practice, the algorithm makes it possible to generate K centroids (special points), the algorithm thus associates each point with the cluster of the nearest centroid. A small example is illustrated in the figure below:

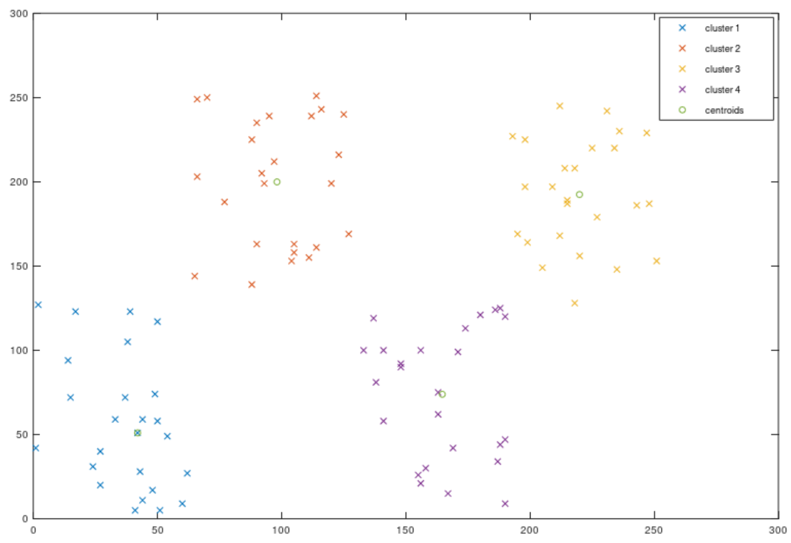

Figure 1: K Means with 40 points and 4 Clusters

The algorithm has many applications in the machine learning field and in many other fields. Here are some examples :

- Unsupervised learning and Cluster analysis : Separating a dataset into K groups with similar features.

- Market segmentation : Segment customers into groups with similar preferences to better target advertising.
- Image Compression : One of the oldest application of K-means, the colors used in the image (usually 256 * 256 * 256 available colours) are replaced by only K colors (which corresponds to the centroids computed by Kmeans).

1.2 Tools Used During The Project

The parallelization of the algorithm will be implemented using C language and the MPI Library. The performance measures and the tests will be carried out on PDC (Center for High Performance Computing, KTH University).

2 KMeans : Serial Program

2.1 Description of the Problem

The algorithms works in any dimension but for the sake of simplicity, during this project, we will only use a 2D data (so that we can plot the result into an image). $x_i \in R^2$.

Here are some notations that will be used throughout this article :

- x_i is the i-th point of the dataset.
- c_j is the j-th centroid generated by KMeans.
- C_j is the j-th cluster generated by KMeans : C_j consists of the points for which c_j is the closest centroid.

The algorithm aims to generate $(c_j)_{j\in\{1,..,K\}}$ such that :

$$J = \sum_{j\in\{1,..,K\}} \sum_{x_i\in C_j} \|x_i - c_j\|^2 \; is\ minimal$$

J is called the cost function.

This problem is an NP-problem, but the K-Means Algorithm gives usually a satisfying solution. In fact this solution depends on the initialization and that's why in practice, the algorithm is executed many times and the best result is then selected.

2.2 The Algorithm

The algorithm starts by generating random centroids (Usually K points from the dataset are chosen randomly to be the initial centroids.). Then, it assigns each point to the cluster of the nearest centroid and compute the new centroids (the new centroids are equal to the mean of the points of each cluster). These two last steps are repeated until convergence of J (In practice we consider J has converged when $|J_{step+1} - J_{step}| \leq \epsilon$)

Algorithm 1: KMeans Serial Algorithm

```
Result: K Centroids/Clusters
Randomly choose K points as the initial centroids : $c_j$
while $|J_{step+1} - J_{step}| \leq \epsilon$ do
    Assign each point x to the cluster corresponding to the nearest
      centroid : $\forall x,\ x \in C_k$ such that $k = argmin_j(\|x - c_j\|^2)$
    Compute new centroids : $c_j = \frac{1}{|C_j|} \sum_{x \in C_j} x$
    Compute the new value of J : $J = \sum_{j \in \{1,..,K\}} \sum_{x_i \in C_j} \|x_i - c_j\|^2$
end
Return the centroids and the clusters.
```

This algorithm has a running time : $T_s = O(N_{steps}NK)$, where N_{steps} denotes the number of steps before convergence.

3 KMeans : Parallel Algortihm

3.1 Idea:

- Data distribution : We will use a linear distribution. Indeed, the choice of distribution does not matter in this algorithm since it absolutely does not change the result or the performance.
- Initialization : each process randomly select K centroids. This is different from the serial algorithm since in total there will be $P \times K$ centroids initially.
- Computing new centroids : This will be done by recursive doubling. This part will be detailed further.
- Computing the new value of J : This can easily be done using recursive doubling by summing the local costs in each process.

Here are some notations that will be used in this section :

- P denotes the total number of processes.
- p denotes the rank of the process. By convention, the rank of the first process is zero and therefore $p \in \{0, .., P-1\}$.
- We will use superscript p to denote a variable associated to process p. For example the j-th centroid of the process p will be denoted as c_j^p.

3.2 Parallel Algorithm

3.2.1 The Algorithm

In the algorithm below, we suppose that the data have been already distributed.

Algorithm 2: KMeans Parallel Algorithm

```
Result: K centroid/clusters
Randomly choose K points as the initial centroids : $c_j$
while $|J_{step+1} - J_{step}| \leq \epsilon$ do
    Assign each point x to the cluster corresponding to the nearest
      centroid : $\forall x \in D_p, x \in C_k$ such that $k = argmin_j(\|x - c_j\|^2)$
    Compute the local new centroids and their weights :
      $m_j = |C_j|$ & $c_j = \frac{1}{m_j}\sum_{x \in C_j} x$
    Compute the global new centroids with recursive doubling.
    Compute the local cost : $J_{local} = \sum_{j \in \{1,..,K\}} \sum_{x_i \in C_j} \|x_i - c_j\|^2$
    Compute the global cost with recursive doubling :
      $J = \sum_{0p \leq P-1} J_{local}$
end
```

Note that, in practice we will use the predefined function **MPI_Reduce** to compute the global cost.

3.2.2 Computing new centroids with recursive doubling

To simplify the problem, let's suppose we have 2 processes. Each of these has already computed its new local centroids $c_j^p = \frac{1}{m_p}\sum_{x_i \in C_j^p} x_i$. The global centroids should be $c_j = \frac{1}{m}\sum_{x_i \in C_j} x_i = \frac{1}{m^0+m^1}\sum_{x_i \in C_j^0 \cup C_j^1} x_i = \frac{1}{m^0+m^1}(\sum_{x_i \in C_j^0} x_i + \sum_{x_i \in C_j^1} x_i) \Rightarrow c_j = \frac{m_j^1 c_j^1 + m_j^0 c_j^0}{m_j^1 + m_j^0}$. Therefore, in order to compute the global centroids, each process have to send (m_j^p, c_j^p). This result can be generalized for any number of processes. The algorithm below shows a recursive doubling version to compute the new centroids.

Algorithm 3: Recursive doubling for new centroids

```
Result: New centroids
Computing local centroids and their weights $(m_j, c_j)$
if $p \geq 2^D$ then
    send($(m_j, c_j)$,bitflip(p,D))
end
if $p < P - 2^D$ then
    receive($(m', c')$,bitflip(p,D))
    $c_j = \frac{m_j * c_j + m' * c'}{m_j + m'}$
    $m_j = m_j + m'$
end
if $p < 2^D$ then
    for $d = 0 : D - 1$ do
        send($(m_j, c_j)$,bitflip(p,d))
        receive($(m', c')$,bitflip(p,d))
        $c_j = \frac{m_j * c_j + m' * c'}{m_j + m'}$
        $m_j = m_j + m'$
    end
end
if $p < P - 2^D$ then
    send($(m_j, c_j)$,bitflip(p,D))
end
if $p \geq 2^D$ then
    receive($(m', c')$,bitflip(p,D))
    $c_j = c'$
end
```

3.3 Theoretical Performance

It is possible to compute the execution time for this algorithm :

$$T_s = t_{comm} + t_{comp}$$

- Computation Time (For a single step):
  - Assigning points to clusters : $t = 8 * \frac{N}{P} K t_a$
  - Compute the local centroids : $t = 10 * \frac{N}{P} K t_a$
  - Compute the local cost : $t = \frac{N}{P} t_a$
  - Compute the global centroids(Recursive doubling): $t = 12 * log(P) K t_a$
  - Compute the global cost(Recursive doubling): $t = log(P) t_a$
- Communication Time (For a single step):
  - Compute the global centroids recursive doubling : $t = log(P)(t_{st} + 2K t_{data})$

– Compute the global cost recursive doubling : $t = log(P)(t_{st} + t_{data})$

$$\Rightarrow T_P = N_{steps}((18\frac{N}{P}K + \frac{N}{P} + (1+12K)log(P))t_a + 2log(P)t_{st} + (2K+1)log(P)t_{data})$$

$$\Rightarrow T_P = O(N_{steps}(\frac{N}{P}K + Klog(P)))\ if\ 1 \ll N, P$$

The Speedup is then (supposing that N_{steps} is the same regardless of the number of processes :

$$S_P = \frac{(18K+1)Nt_a}{(18\frac{N}{P}K + \frac{N}{P} + (1+12K)log(P))t_a + 2log(P)t_{st} + (2K+1)log(P)t_{data}}$$

$$S_P = P\frac{1}{1 + \frac{(1+12K)Plog(P)}{(18K+1)N} + \frac{2Plog(P)}{(18K+1)N}\frac{t_{st}}{t_a} + \frac{(18K+1)Plog(P)}{(18K+1)N}\frac{t_{data}}{t_a}}$$

$$S_p = \frac{P}{1 + A\ P\ log(P)}$$

We can then draw some remarks :

- If P is fixed, increasing N will increase the Speedup.
- If N is fixed, the speedup will degrade if P gets extremely large (communication overhead).

Note that : in this case we supposed that the serial algorithm is totally parallelizable, in practice that doesn't hold and $S_P = \frac{P}{(1-f)+fP+Alog(P)P}$, where f denotes the fraction of the computation which cannot be divided into concurrent tasks.

3.4 First example : Correctness of the Algorithm

In order to verify the correctness of the implementation, the algorithm is executed for N = 1000, K=4 and P=4. The algorithm takes as input a dataset with 4 distinct regions. The figure 2 shows the output of the algorithm. This figure shows that the dataset has been successfully split into 4 clusters that seem rather well chosen.

Note that : the result may seem not perfect, but this is either due to a not perfect initialization or a big threshold ϵ.

$\Rightarrow$ Our parallel implementation is correct and we can now test our performance for larger N.

3.5 Experimental Results

In this section, some test will be carried out in order to show the effect of some parameters. During all these tests, the Kmeans algorithm is preceded by a random dataset generation. The dataset generation will generate numbers in

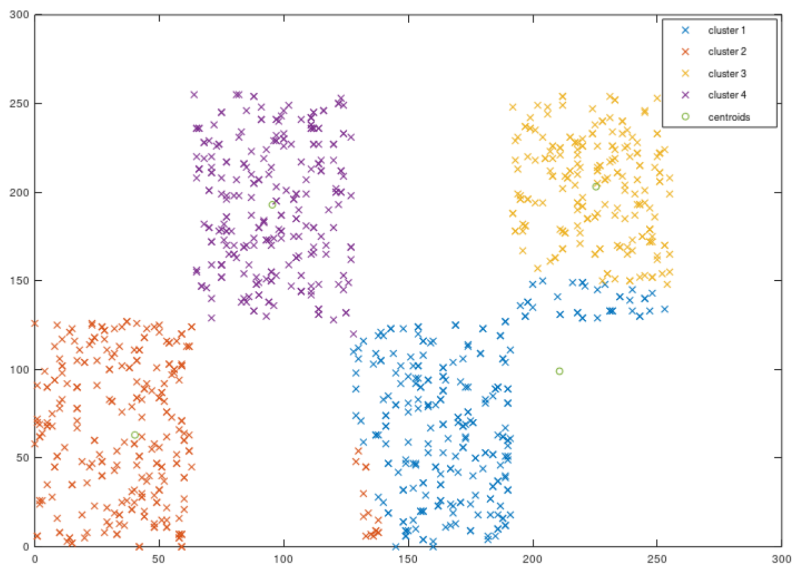

Figure 2: Parallel K Means N=1000 and K=4

four regions as shown in the figure 2. This step has a complexity of O(N/P) and may be neglected under some conditions.

The next parts will expose two tests: the first one for a small number of N and the second one for a very large number of N.

3.5.1 Test 1 : Effect of Communication overhead

During this test we will consider N=1000 and $P \in [1..48]$. In order to measure the execution time and since the PDC platform doesn't hold execution time of some ms, the program will be executed 5 10^5 and we will consider the execution time as $T_P = \frac{T_{P,500000}}{500000}$, where $T_{P,500000}$ denotes the execution time for 500 000 executions. The Array below shows the results :

P	Serial	1	2	3	4	8	12	16	20
$T_{P,500000}$(s)	-	188	98	89	73	64	67	63	69
$T_P(\mu s)$	$T_s^* = 340$	376	198	174	150	144	148	148	150
S_P	-	0.90	1.71	1.95	2.26	2.36	2.29	2.29	2.26

24	28	32	36	40	44	48
81	104	102	169	127	164	125
166	226	248	314	234	286	286
2.04	1.50	1.37	1.08	1.45	1.18	1.18

The figure 3 shows the Speedup curve. At the beginning (for low values of P) the Speedup increases and reaches the value of 2.3 before plateauing (this will be discussed in the next test). For larger P ($P > 30$), We notice that the Speedup drops to 1.18 (which is very small for a parallel program): This confirms the theoretical result and we may conclude then that in order for the algorithm to be efficient, it's essential that $P \ll N$.

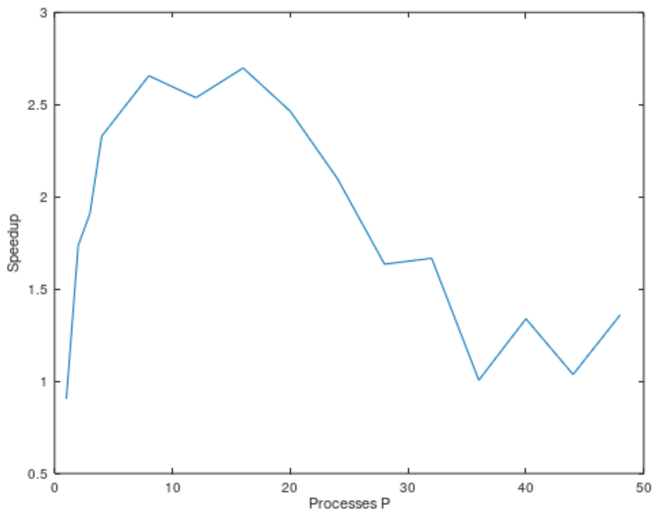

Figure 3: $S_P = f(P)$ using N=1000 and K=2

3.5.2 Test 2 : Very large inputs

During this test, the algorithm is executed with very large inputs : $N = 10^8$ and K=4 and for $P \in [1..48]$. The execution time for each P is shown in the next table.

P	Serial	2	4	6	8	12	16	20	24	28	32	36	40	44
T_P(s)	$T_s = 71$	35	28	25	13	12	9	11	6	7	6	6	6	6
S_P	-	2.0286	2.5357	2.8400	5.4615	5.9167	7.8889	6.4545	11.8333	10.1429	11.8333	11.8333	11.8333	11.8333

The figure shows the Speedup curve. We can notice that the curve accelerates quickly at the beginning but gradually begins to stagnate. This is not surprising, since the the communication is negligible and the workload is fixed. We may conclude then that for a fixed number N, it is useless to exceed a certain threshold value of P (in the case of our experiment it's ~ 36) otherwise the efficiency and the performance will be affected.

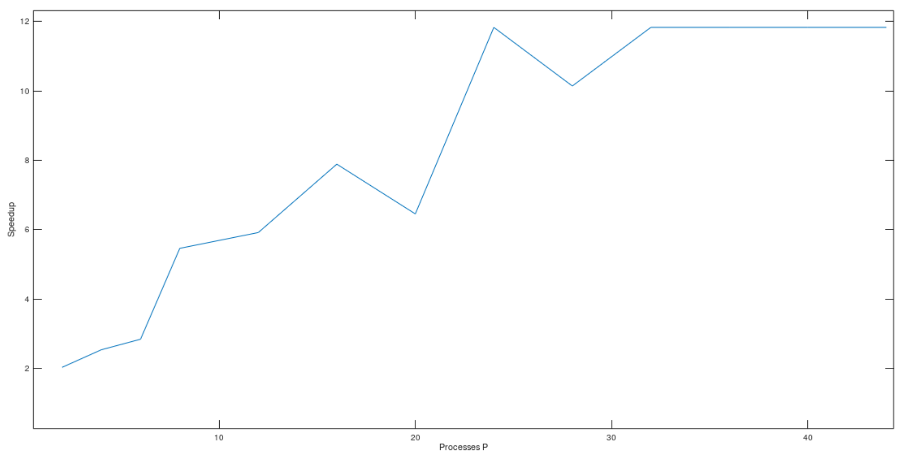

Figure 4: $S_P = f(P)$ using $N = 10^8$ and K=4

4 Reducing the Communication Overhead

In all this part we will focus on the situation where p and N are of the same order of magnitude (or in the case of a very bad network). We will therefore try to limit the communication time. Here are some suggested solutions:

4.1 Merge the two recursive doubling

Instead of performing two recursive doubling (one for the cost and the other for the new centroids), these two operations are merged together into only one recursive doubling but in this case the algorithm will compute the cost of the previous iteration instead of the current one. In order to see the impact of this change on the effect of communication, we perform a test under the same conditions as test 1 of the previous section. the results are grouped in the table below :

P	Serial	1	2	4	6	8	12	16
$T_{P,500000}$(s)	-	200	108	76	68	60	61	58
$T_P(\mu s)$	T_s^* =340	400	216	152	136	120	122	116
S_P	-	0.850	1.574	2.236	2.500	2.833	2.786	2.931

P	20	24	28	32	36	38	40	44	48
$T_{P,500000}$(s)	59	68	88	84	120	117	96	105	112
$T_P(\mu s)$	118	136	176	168	240	234	192	210	224
S_P	2.881	2.500	1.931	2.023	1.416	1.452	1.770	1.619	1.517

4.2 Gather results every two iterations instead of after each iteration

To further reduce the communication time, we can limit the recursive doubling to one iteration out of two: thus one iteration out of two the program will only calculate the local centroids. A test under the same conditions as test 1 of the previous section is carried out. The results are grouped in the table below :

P	Serial	1	2	4	6	8	12	16	20	24	28	32	36	38	40	44	46
$T_{P,500000}(\text{s})$	-	192	105	59	48	40	33	29	30	27	36	33	48	49	49	43	42
$T_P(\mu s)$	$T_s^* = 340$	384	210	118	96	80	66	58	60	54	72	66	96	98	98	86	84
S_P	-	0.885	1.619	2.881	3.541	4.250	5.151	5.862	5.666	6.296	4.722	5.151	3.541	3.469	3.469	3.953	4.047

4.3 Results

The figure 5 shows the Speedup curve for the algorithm before modification, after the first modification and after the second modification.

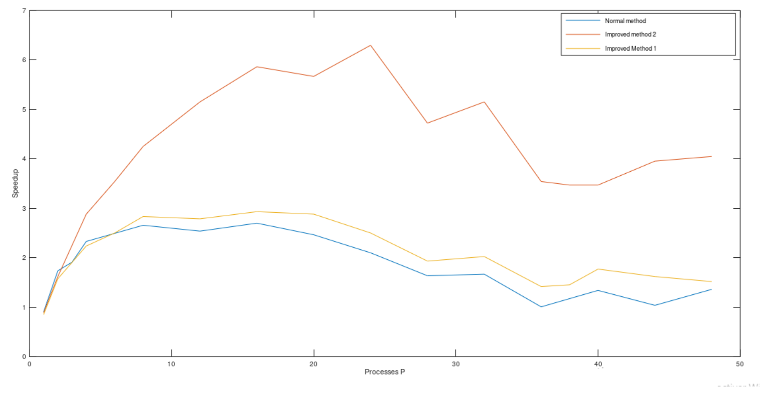

Figure 5: Comparison between Speedup Curves for normal method, improved method 1 & improved method 2

Overall, all the curves have the same shape: an increasing part and a decreasing part. But we note that the modifications made it possible to increase the speedup and to dampen the fall.

But be careful with these results. Indeed these two methods are only efficient for a small input N ($N \leq 100P$). Otherwise, the result will be quite the opposite. For instance, the first modification will add an iteration to the program (because the algorithm is computing the cost of the previous iteration), if N is very large the computation time of this iteration which is of the order of $O(\frac{N}{P}K)$ will be much greater than that of the communication $O(log(P)K)$. The second modification have the same problem and may even have worse performance since it may add more than just one iteration and it may therefore slow down the program.

5 Conclusion

The K-means algorithm may be parallelized in an efficient way satisfying therefore the requirements of Big Data. However, the performance may be affected in the case of a poor network and that's why we can have some methods to improve performance in this specific case (Merging and reducing the communication operations).

Many other variants of this algorithms can be addressed using the strategy presented in this report such as: K-medians, K-medoids & K-means++.