Implementation and Improvement of Parallel K-means Algorithm in Big data using MPI

Yedong Liu
Illinois institute of Technology
3300 South Federal Street
Chicago, IL, 60616, USA
yliu275@hawk.iit.edu

***Abstract*— With the development of computer technology in various fields, in various formats (pictures, music, video and documents, etc.), huge amounts of data are generated and stored in the database. Mining data from the ocean of useful knowledge and apply this knowledge to the development of human society has become a focus for Computer Science industry. But too much data also raises one problem, too much data, too little information, so finding the useful information costs too much price. Distributed Data Mining technology provides a solution to this problem. A key step towards distributed data mining system is to import the traditional data mining algorithms to distributed computing platform. After decades of development, distributed data mining emerged in a variety of algorithms, they are faced with different demands and different tasks. These algorithms improve over the years, so that it can adapt to the new distributed environment, and this is an important step in the distributed data mining.**

***Keywords* — Data Mining, K-means Algorithm, Parallelization, MPI.**

I. Introduction

In recent years, the Message Processing Interface has been widely used in all over the world and after years of development, it has become more mature. The implementation of distributed data mining system using MPI which is a great interface gives us the chance to dig deep into the distributed data programming. Therefore, this project focuses on data mining research about K-means algorithm implementation and its improvement.

K-means clustering is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining. K-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells. [1]

The problem is computationally difficult (NP-hard); however, there are efficient heuristic algorithms that are commonly employed and converge quickly to a local optimum. These are usually similar to the expectation-maximization algorithm for mixtures of Gaussian distributions via an iterative refinement approach employed by both algorithms. Additionally, they both use cluster centers to model the data; however, k-means clustering tends to find clusters of comparable spatial extent, while the expectation-maximization mechanism allows clusters to have different shapes. [1]

The algorithm has a loose relationship to the k-nearest neighbor classifier, a popular machine learning technique for classification that is often confused with k-means because of the k in the name. One can apply the 1-nearest neighbor classifier on the cluster centers obtained by k-means to classify new data into the existing clusters. This is known as nearest centroid classifier or Rocchio algorithm. [1]

II. Approaches

2.1 General Idea

Big data is a new and hot field in our world right now. It features big data set, multi dimensional and most importantly, requires to process this huge data set fast and efficiently as well as low memory cost.

K-means algorithm is a very commonly seen and used algorithm in data mining and many other fields. Considering that the data size nowadays is becoming bigger and bigger, for example, 1 million or even 1 billion data set, my plan was to set the data set very big(up to 1 million data set), and also test the performance for different dimensions of the data set and different numbers of cluster centers for each data set. I will also try to learn and improve the now existing methods and make comparison for my original code, new code and the serial code for MPI.

For the data set part, the properties of each point in a graph are coordinates(x,y or z). The number of coordintaes depends on the number of dimensions. For example, two dimensions requires two coordinates and three dimensions requires three coordinates. Same thing goes to the cluster centers, cluster centers have the same properties with the data points. Every data point has a nearst cluster center and of course belongs to that particular cluster. The position of each cluster centers will change during each iteration in order to form new clusters and make the clusters more reasonable for this data set. The cluster to which each data point belongs may or may not change during each iteration. If the position of the cluster centers are not changed anymore, we can end our iteration and the clustering process is now compeleted.

The most time cosuming part is two calculating processes during each iteration. That is, calculating the distance between all cluster centers and each data point, another one is calculating the total distance of one cluster center to determine whether the position of this cluster center is changed or not during this iteration. Obviously this it the part we can parallel and save our time.

2.2 Serial Version of K-means clustering algorithm

For the serial version of the K-means clustering algorithm, I chose the traditional iterative method, that is, calculating the distance between all cluster centers and each data point to determine which cluster it belongs to, then calculate the new position of those cluster centers in each iteration. Finally terminate the iteration when the position of the cluster centers are not changed anymore.

See figures below:

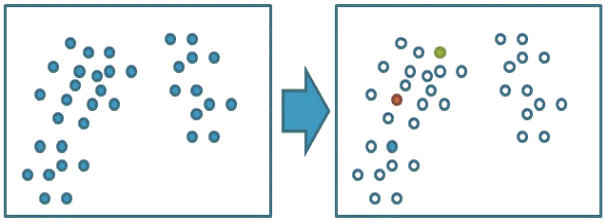

Figure 2.2.1 Load data and choose initial cluster centers

For the serial code, initially, load the data from file and allocate the memory location for future use. Then pick k data points as initial cluster centers, in this example, I picked three initial cluster centers. Considering the real life scenario, all data set and the position of all initial centers will be randomly generated in my code. And after this we can go to the iteration part:

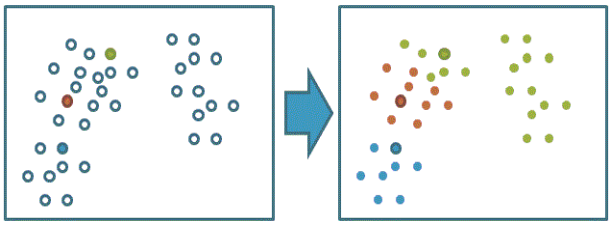

Figure 2 2.2 Find the cluster center for each data point

During each iteration, each data point will calculate the distance between itself and all cluster centers and choose the nearest of the three as its cluster center. Thus all data point will have a cluster center after this process.

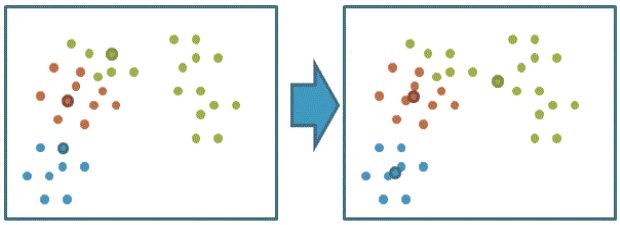

Figure 2 2.3 Recalculate the new cluster centers

After each data point find its cluster center, we can recalculate the new position for all three cluster centers according to the previously formed new clusters, and make sure that all three cluster centers will be the pivot point of each cluster.

Now we can repeat step two and step three until the position of all three cluster centers are not changed anymore then we can terminate our iteration and finish the clustering process. We can simply use the total distance of each cluster center to all its data points to check whether the position is changed or not, if the distance remain constant in the previous two iterations, obviously the position is not changed, otherwise it is changed.

2.3 Two versions of parallel K-means algorithm

For the parallel or simply MPI version of K-means algorithm, my coding process was a little bit tricky. I coded two versions in total. For the first version, I followed the traditional way that many people have already done before on the internet. But some problem occurred and I tried to find out the cause and fix it. After I finally fix all the bugs, I found the performance of my first version code was not good at all, so after the analyze I decided to improve my code and try another different approach. The second version of my code was doing well and it is the version I attached in my folder. I will talk about both of my versions below.

For the first version. My design followed the traditional master-worker model. Process 0 or rank 0 will initialize, load data and pick the starting cluster centers before entering the iteration. Then during each iteration, the process 0 will evenly partition the data set to all processes including process 0 together with the three cluster centers. Then every process needs to calculate the distance between each data point in its own data set and all three cluster centers to determine which cluster center it belongs to. Finally, every other process send the data back to process 0 and leave the process 0 to determine whether to terminate the iteration or not.

After finish my version 1.0, I soon found that there were some fatal deadlocks which caused the code not to terminate. Especially when the data size is extremly big like 100 thousand. I started to look into my design. Soon I found that the workload for each process is not evenly partitioned. The process 0 has a much heavier workload than any other process. The process 0 not only does the same calculating and finding which cluster every data point belongs to, but also calculating the total distance to determine whether the iteration should be terminated or not. When the data size is small, the calculating of Euclidean distance is fast and simple, so the process 0 needs approximately the same time with other processes to finish its work during each iteration. Thus other processes do not need to wait process 0 to finish. But when the data size is becoming big enough, or when the dimension or cluster centers are becoming extrememly large, process 0 needs much more time calculating those square root equation, as a result, other processes cannot get the new data needed in order to start the new round of iteration, that is the deadlock.

To solve the problem, I tried to add more MPI Barriers inside each iteration, make sure all processes will be synchronized at the end of each iteration. This did eliminated the deadlocks, but it cost a lot. The performance of my code dropped dramatically after I made more synchronizations, especially when the iteration number is big enough. In every iteration, other processes wasted too much time waiting for process 0, and the total wasted time sums up to be a significant number when the iteration numbers piles up. In my testing, the serial code needs about 16 seconds to handle a data set of 1 million data, and the MPI code needs around 13 seconds. The speedup is a little bit more than 1, this is not the speedup I want and I tried another approach.

In my second version or version 2.0, I came up with an idea that is to evenly partition the workload to all processes including process 0.

In my new design, I let process 0 do less work than before and let every other process do a little bit more work than the origianl version. This time, process 0 will partition all the data set to other processes and itself will not do any calculation for finding the cluster for each data point in the whole dataset. Process 0 will only gather the data and calculate the new cluster center and determine whether to end loop or not in each iteration. For other process, the workload is heavier. But as the number of processes increases, the "more" work for each processor will become less. For example, if right now we have 4 processors, each processors needs to handle 100 dataset, the total dataset is 4*100=400 dataset. After I applied my changes, every processor needs to handle 400/3=133 dataset, 33% more workload than the original one. For 8 processors, the workload increase is (8*100)/(8-1)/100=14%. This is a acceptable workload increase rate for us. Because in real life, the processors sometime will be even bigger than 8, thus the workload increase rate will be lower and lower, so my change will not affect the performance too much.

I coded my version 2.0, the benefit of my second version of code is that I can now remove some unnecessary MPI Barriers to provide me a better performance. The new speedup is now more than 2. I will show in the below.

III. CODE, COMPILE AND EXECUTE

In my code, I can input three arguments: cluster numbers, dimension and data size for both serial code and MPI code. I will use cluster numbers = 3, dimension = 3 and data size = 1 million as my default input.

The compile and execution are shown below.

3.1 Serial Code compile and Execution

For Serial code, simply compile with "gcc kmeans_serial.c –lm –o kmeans_serial", and execute with "./kmeans_serial". The result is shown below:

Figure 3.1.1 Compile and Execute for Serial code

As shown above, the code will take three numbers as arguments, cluster numbers, dimension and data size. Here I tested the code with 3 cluster numbers, 3 dimensions and data set of 15. The code will show the detail of each iteration if the data set is smaller or equal to 50, this is convenient for us to check the correctness.

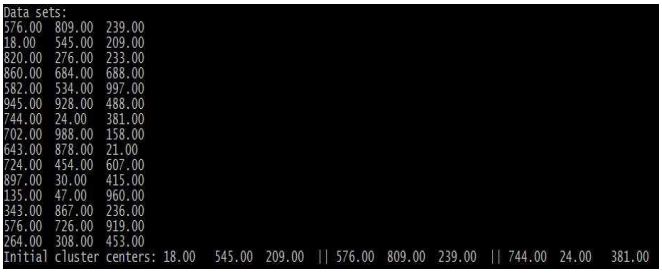

Figure 3.1.2 Data set and Initial Cluster Centers

As shown in Figure 3.1.2, after the code started running, the data set and the initial randomly selected cluster centers will be shown.

Figure 3.1.3 Information in each iteration

The information of to which cluster every data point belongs and the new position of all cluster centers will be shown every time an iteration ends.

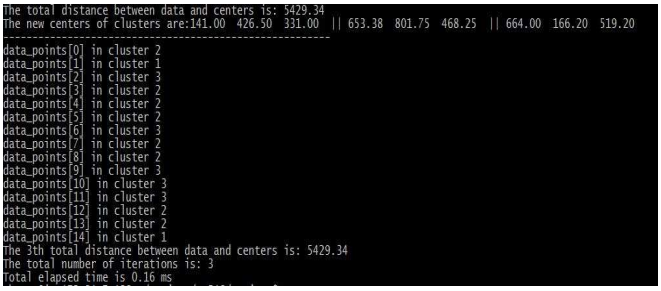

Figure 3.1.4 Iteration termination and total time

If the position of all cluster centers are not changed in the previous two iterations, the iteraion will end and show the total elapsed time.

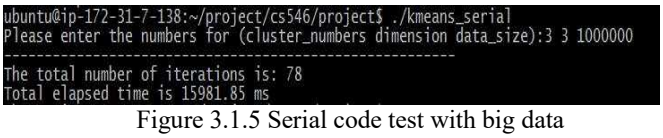

Figure 3.1.5 Serial code test with big data

If the data set is bigger than 50, the code will directly show the result.

3.2 MPI code compile and Execution

Similar to the serial code, MPI code also took three arguments: cluster numbers, dimension and data size. So compile with ”mpicc kmeans_mpi.c –lm –o kmeans_mpi”, and execute with “mpiexec –n 8 ./kmeans_mpi”. Here I use 8 processors as an example.

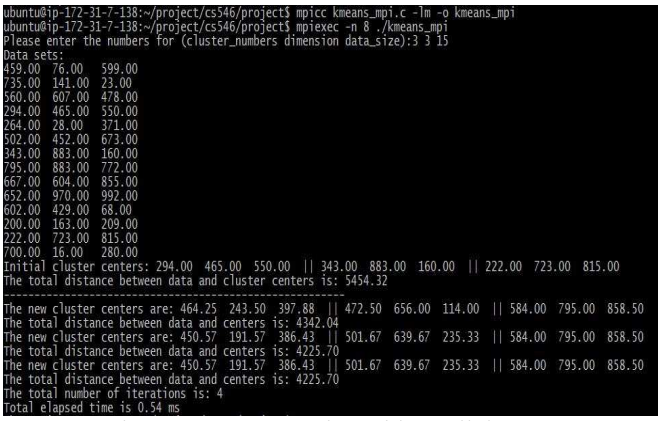

Figure 3.2.1 MPI testing with small data set

As shown in Figure 3.2.1, the MPI code will also print the data set and initial cluster centers when the data set is less than or equal to 50. The MPI code will directly show the new position of the cluster centers and total distance between all cluster centers and its data points in each iteration. Also, after the iteration terminated, the number of iterations and total elapsed time will be displayed.

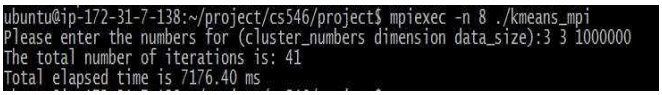

Figure 3.2.2 MPI test with big data

Similar to the serial code testing with big data, the MPI code will also directly display the total elapsed time when the dataset is bigger than 50.

IV. OUTOUT AND ANALYSIS

In this part, I will test performance between the serial code and the MPI code when handling big data. I will also test the performance of my MPI code for different number of processors, different dimensions and different cluster centers as well as the analysis.

4.1 Performance

I choosed the data set growing from 100k all the way up to 1 million and here is the result.(MPI with 3 cluster centers and 8 processors)

	Serial	MPI (8 processes, 3 clusters)
100k	1153.12	801.18
200k	1968.01	1545.04
300k	2699.00	2324.75
400k	3447.94	3046.58
500k	5025.76	3502.27
600k	6430.96	4024.48
700k	8416.22	5121.11
800k	11967.87	5846.59
900k	14208.60	6531.51
1000k	16537.62	8825.86

Table 4.1.1 Performance of Serial and MPI

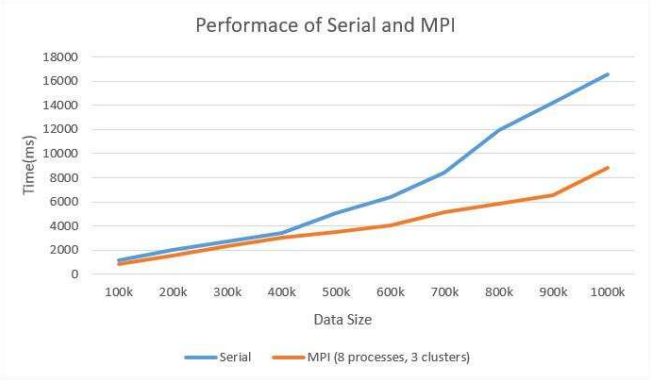

Figure 4.1.1 Performance of Serial and MPI

As shown in the Figure 4.1.1, when the data size is small than 400k, the difference between the Serial code and the MPI code is not big. After the 400k mark, the gap between the two is growing larger and larger as the data size becomes larger. The total speedup of the MPI to Serial is 2.11 when the data size is 1 million.

When the data size is small, in each iteration, the communication cost and overheads contribute to the delay of the performance. Because every time each process will send back tens of thousands of data set back to process 0, this is a huge cost, when the total data size is small, this cost is significant to the total time.

But when the data size increases, the communication cost becaomes insignificant as the calculation consumes more and more time. At this time, the advantage of the parallel code emerges and the MPI line follows the linear growth pattern which is good when we want to handle some even bigger data set.

4.2 Cluster Numbers

My default cluster numbers for testing is 3, and we can do more testing about cluster numbers. But I will reduce the data size because this will be more convinient for our testing to get the result more quickly.

	MPI with 10 clusters	MPI with 100 clusters
10k	183.09	1081.75
20k	227.96	2574.19
30k	669.94	5877.11
40k	887.58	8281.18
50k	890.20	9306.97
60k	903.26	10841.83
70k	1028.97	11340.05
80k	1218.81	12878.66
90k	1399.19	13881.41
100k	1799.41	15729.62

Table 4.2.1 Performance for different clusters

Figure 4.2.1 Performance for different clusters

As we can seen in Table 4.2.1, the performnce of MPI code with different cluster numbers is very good. The time needed to handle 100 cluster centers is almost 10 times than the time needed to handle 10 cluster centers. Also both line follow the linear growth which is a good sign of great scalability.

4.3 Dimensions

Similar to the cluster numbers, I will test my MPI code with 10 dimensions and 100 dimensions. I will also reduce the data set to get a quicker result.

	MPI with 10 dimension	MPI with 100 dimension
10k	149.75	1837.64
20k	387.17	2945.04
30k	479.85	5524.75
40k	841.82	8046.58
50k	937.39	10502.27
60k	976.94	12024.48
70k	1024.41	13121.11
80k	1286.31	15846.59
90k	1570.88	18438.06
100k	1902.31	23381.70

Table 4.3.1 Performance for different dimensions

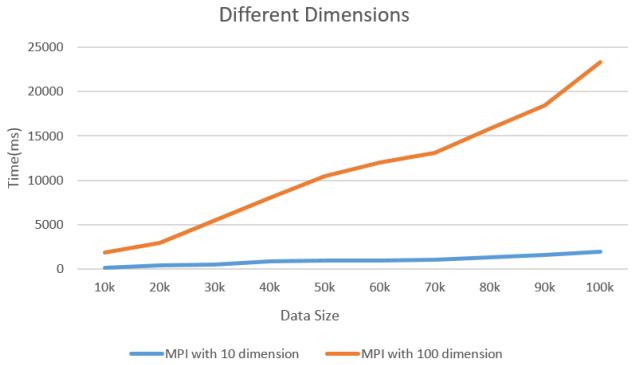

Figure 4.3.1 Performance for different dimensions

The performance of different dimensions is also positive. Both line follow linear growth and the time needed for 100 dimensions is approximately 10 times than the time needed for 10 dimensions.

4.4 Processors

I will test from 100k to 1m data set using from 4 processors to 8 processors.

	4	5	6	7	8
100k	974.87	891.54	879.28	853.4	801.18
200k	1943.94	1897.62	1796.19	1622.47	1545.04
300k	2613.29	2587.05	2432.47	2402.43	2324.75
400k	3476.94	3307.91	3258.24	3143.72	3046.58
500k	4659.14	4486.25	4325.36	4222.85	3502.27
600k	4762.47	4664.95	4593.02	4505.4	4024.48
700k	5722.87	5677.74	5302.9	5203.73	5121.11
800k	6995.95	6913.02	6008.03	6000.76	5846.59
900k	7602.81	7390.12	7007.49	6972.69	6531.51
1m	9263.85	9009.93	8990.15	8875	8825.86

Table 4.4.1 Performance for different processors

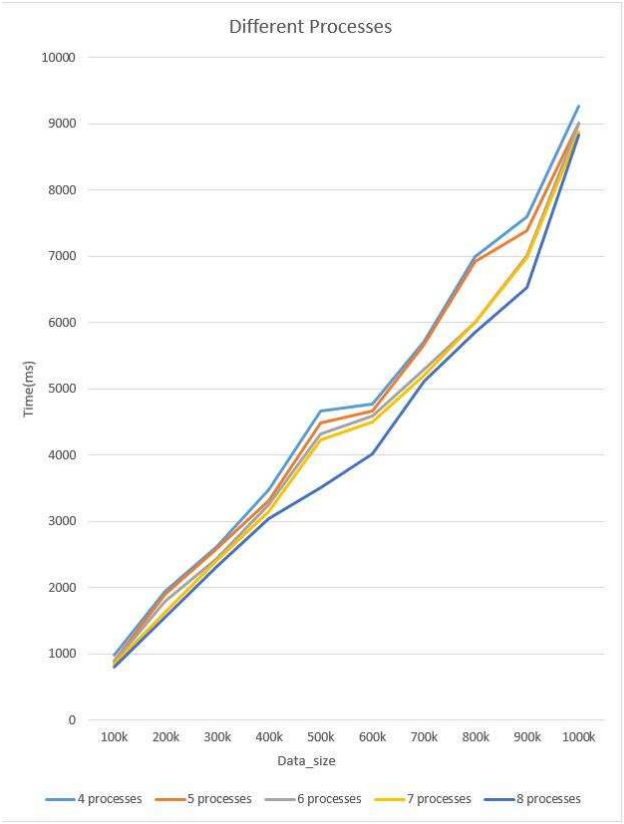

Figure 4.4.1 Performance for different processors

We can see that in Table 4.4.1 and in Figure 4.4.1, the time needed to handle the same amount of data will slightly decrease when the number of processors increases. But the gap between 4 processors and 8 processors is not that big, and I tried to find out the reason for that.

As I have mentioned above in section 2.3, the more amount of workload that each processor will take during one iteration will decrease as the number of processors increases. The percentage of "more" workload will decrease from 33% all the way down to 14% if the number of processors grows from four to eight.

Then, another factor is that, in my coding, the dataset is generated randomly every time when the code is executed. So the floating point calculation time will also make a difference when the data size is really big. The difference in each iteration piles up to contribute to the slight amount of time decrease when processor increases.

REFERENCES

[1] https://en.wikipedia.org/wiki/K-means_clustering