MESSINA UNIVERSITY

DEPARTMENT OF ENGINEERING

PARALLEL PROGRAMMING

REPORT: PARALLEL K-MEANS ALGORITHM WITH MPI AND OPENMP

Prof. Salvatore Distefano

Le Van Huy

Abstract

K-means is one of common algorithms being used in data mining and statistics, search engine applications, customer segmentation. Implementing a K-means algorithm running on parallel system is one of the ways which help us bring high performance in saving executing time as well as taking advantages of the most of the hardware power. Thus, this report is about implementation of K-means with MPI and OpenMP libraries for parallel programming purpose.

I. Introduction:

Parallel computing is a type of computing architecture in which several processors simultaneously execute multiple, refers to the process of breaking down larger problems into smaller, independent, often similar parts which can be executed simultaneously by multiple processors communicating via shared memory, the results of which are combined upon completion as part of an overall algorithm. Increasing available power of computation for faster application processing and solving problem is the primary goal of parallel computing.

II. Parallel programming:

Parallel programming is using multiple resources such as processors, memories, data to solve a problem, which breaks it down into a series of smaller steps, delivers instructions, and processors execute the solutions at the same time. It is also a form of programming that offers the same results as concurrent programming but, spending in less time and achieving with more efficiency. Many computers, such as laptops and personal desktops, use parallel programming in their hardware to ensure that tasks are quickly completed in the background.

1. Message Passing Interface (MPI):

Message Passing Interface (MPI) is a standardized and portable message-passing model designed to function on parallel computing architectures. The MPI standard defines the syntax and semantics of library routines, which are useful by being able to write portable message-passing programs in some language such as: C, C++, Fortran, Python, and so on. In addition, MPI is specifically used to allow applications to run in parallel across a number of separate computers (nodes) connected by a network, which is distributed architectures working with different memory location as known as distributed memory, and different data. MPI helps cores or memories in distributed system communicate via special send and receive routines which are message passing.

MPI is suitable for single program multiple data model that is a primary programming for large-scale parallel machines. In the other words, MPI is a communication protocol for parallel programming, which specifically used to allow applications to run in parallel across a number of separate computers connected by a network. So, using mpi4py based on python language is the approaching of parallel programming purpose in this project

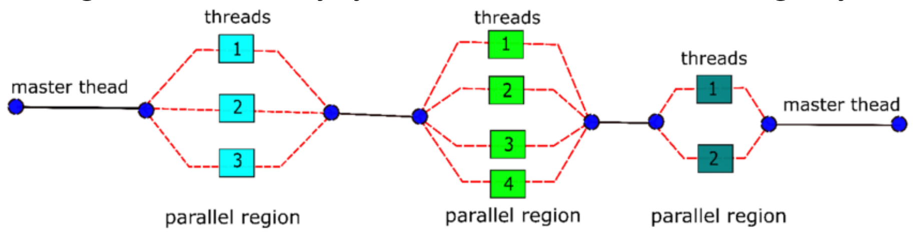
2. OpenMP:

OpenMP is an open-source implementation of MPI, an Application Programming Interface (API) that supports multi-platform shared-memory multiprocessing programming, jointly defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared memory parallel applications.

In addition, OpenMP is Fork-and-Join model, implements multithreading by creating and forking some specific sub-threads from primary thread, and the system will assign a task among them. Then the threads will run or execute concurrently the task with the runtime environment allocating threads to different processors. Specifically, it identifies parallel regions as blocks of code that may run in parallel, which allow developers to insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-time library to execute the region in parallel. Moreover, there are 2 options of shared memory between threads and private to a thread.

OpenMP is based on SPMD (Single Program Multiple Data) which means that a single program can work on multiple data by iterating over different data on the region of the memory. All tasks execute their copy of the same program simultaneously.

The fork and join mechanism started by a master thread which also served as a main entry to the whole program, the main thread will create other threads when it sees OpenMP directive to launch the desired number of threads. The forked thread will be executed concurrently by CPU's context switching. When the forked threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.



3. MPI and OpenMP Syntax:

3.1.Initialize MPI and OpenMP library in your code:

- Create number of processors (MPI):

MPI_Init(NULL, NULL); // Init MPI
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get the processor Id
MPI_Comm_size(MPI_COMM_WORLD, &size);// get number of processors

- Create number of threads (OpenMP):

omp_set_num_threads(num_threads);

In OpenMP, Compiler directives are used for various purposes like spawning a parallel region, dividing blocks of code among threads, distributing loop iterations among threads, synchronization of work among threads. Compiler directives have the following syntax:

#pragma omp construct [clause [clause]…]
{ structured_block }

In side structured block is the place to put logic code that belongs to specific parallel construct.

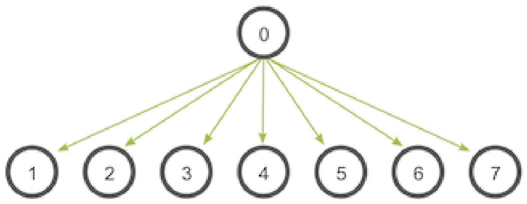
3.2.Communication Routines:

- MPI:

Sending and receiving are the two foundational concepts of MPI, which the process of communicating data follows a standard pattern, they are called "point-to-point" communication that is able to send and receive data between 2 specific processors. The data will be sent from sender rank to receiver rank by tag (rank id). After *MPI_Receive* function is called, the data is already sent to destination rank. In addition, these follow 2 concepts of communication that are blocking and non-blocking communications. Blocking communications will make a blocking send operation terminating when the message is received by the destination, and a blocking receive operation finishing when a message is received by the caller. There is a waiting state for tasks done in here. Otherwise, Non-blocking communication will create a non-blocking send operation completing when the message is sent by sender, even if it has not been received yet. And non-blocking receive operation always terminates immediately, it may not receive any message. There is no wait this communication between ranks and there is a status variable in both sending and receiving operation to query the status of message sent and received respectively.

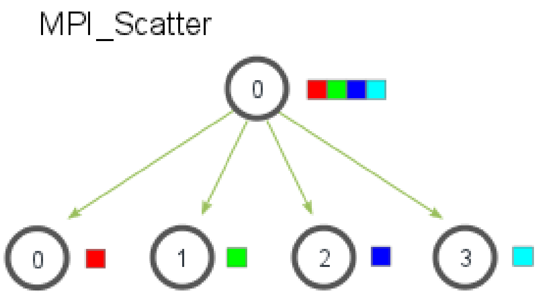
MPI_Send(data, count, MPI_Datatype, destination, tag, comm)
MPI_Receive(data, count, MPI_Datatype, source, tag, comm, status)

A broadcast is one of the standard collective communication techniques. During a broadcast, one process sends the same data to all processes in a communicator. One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes.



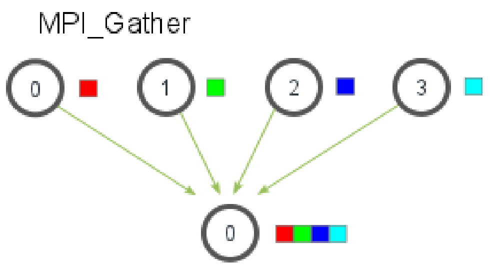
C/C++: **MPI_Bcast(data, count, MPI_Datatype, root, comm)**
Python(mpi4py): **comm.bcast(data, root = 0)**

Scatter is a collective routine that is very similar to Broadcast, it involves a designated root process sending data to all processes in a communicator. The primary difference between Broadcast and Scatter is small but important. Broadcast sends the same piece of data to all processes while Scatter sends chunks of an array to different processes.



C/C++: ***MPI_Scatter(send_data, send_count, MPI_Datatype, receive_data, receive_count, MPI_Datatype, root, comm)***
Python (mpi4py): ***comm.scatter(data, root = 0)***

Gather is the inverse of Scatter. Instead of spreading elements from one process to many processes, Gather takes elements from many processes and gathers them to one single process. This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.



C/C++: ***MPI_Gather(send_data, send_count, MPI_Datatype, receive_data, receive_count, MPI_Datatype, root, comm)***
Python (mpi4py): ***comm.scatter(data, root = 0)***

Reduce and All reduce are classic concepts from functional programming. These functions will take data from each processor and execute a MPI operation from the parameter of the function, then they will give out a output for root processor with Reduce function and for all processors. Specifically, there are some MPI operations such as: MPI_MAX to get maximum element of data, MPI_MIN to get minimum element of data, MPI_SUM to get sum of all elements, so on… For mpi4py library, we could define some custom functions for the operation of both reduce and all reduce.
C/C++: ***MPI_Reduce(send_data, receive_data, count, MPI_Datatype, MPI_Operation, root, comm)***
MPI_Allreduce(send_data, receive_data, count, MPI_Datatype, MPI_Operation, comm)
Python (mpi4py): ***comm.allreduce(data, op = operation)***
comm.reduce(send_data, receive_data, op = operation, root = 0)

Barrier is a collective communication function, which implies a synchronization point among processors, it means that all processors have to reach a point in code block before they can all start executing again.
C/C++: ***MPI_Barrier(comm)***
Python (mpi4py): ***comm.Barrier()***

- OpenMP:

A parallel region is a code block which is executed by multiple threads. This is the fundamental OpenMP parallel construct. When a thread reaches a parallel directive, it creates a group of threads and becomes the master (thread id = 0) of the group. From the beginning of the parallel region, the code block is duplicated and all threads will execute that code block. There is an implied barrier at the end of a parallel region. Only the master thread continues execution past this point.

#pragma omp parallel[clause …]
{ structured_block }

Master directive specifies a region which can be executed only by the master thread. And the other threads on the team skip this section of code. There is no implied barrier associated with this directive.

#pragma omp master
{ structured_block }

Critical directive specifies a region of code block that have to be executed by only one thread at a time. If a thread is currently executing inside a critical region and another thread reaches that critical region and attempts to execute it, it will block until the first thread exits the region.

#pragma omp critical [name]
{ structured_block }

Barrier directive synchronizes all threads. When a Barrier directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

#pragma omp barrier
{ structured_block }

OpenMP includes several run-time library routines. These routines are used for a lot of purposes such as setting and querying the number of threads, querying threads' unique identifier (thread id), querying the thread pool size and so on...

omp_get_thread_num() - Return thread id in a parallel region.
omp_set_num_threads(numbers) - Set number of threads for a parallel region.
omp_get _num_threads() - Return the number of threads for a parallel region.
omp_get_max_threads() - Return the maximum number of threads.
omp_get_wtime() - Return the current time in the epoch time format.

III. K-Means Algorithm:

K-means is an unsupervised clustering algorithm that is intended for unlabeled data. K-means clustering algorithm is a method used in analyzing cluster properties of data. It helps determine which group of data really belongs to.

The idea of the k-means clustering algorithm is to divide a set of data into different clusters. where the number of clusters is given K. Clustering work is established based on the principle: Data points in the same cluster must have the same certain properties or features. That is, between points in the same cluster must be related to each other. The k-means method is a widely used clustering technique that seeks to minimize the average squared distance between points in the same cluster. For computers, the points in a cluster will be data points that are close to each other.

Suppose we have a data set and we need to group similar data into different unknown clusters. A simple way to simulate this problem is to represent it through a geometric view. The data can be thought of as points in space, and the distance between points can be thought of as a parameter of their similarity. The closer two points are to each other, the more similar they are.

With such a geometrical view, we can rewrite the problem in the following form:
Data: dataset $\mathbf{X} \in \mathbb{R}^{\mathbf{ND}}$ consists of **N** data points with **D** dimensions
Task: Diversify the data into **K** clusters of similar data. Of course, $\mathbf{K} < \mathbf{N}$

1. Ideal of algorithm running in sequential:

1. Initialize K data points in the dataset and temporarily treat it as the center of our data clusters. $\mathbf{C^{(0)} = \{m_1^{(0)}, m_2^{(0)}, \ldots, m_k^{(0)}\}}$
2. For each data point in the dataset, it will be identified as belonging to 1 of the K nearest cluster centers by calculating the distance between the point that is the cluster center and the point in the dataset based on the Euclidean algorithm:
   $$\mathbf{S_i^{(t)} = \{x_p : \| x_p - m_i^{(t)} \| 2 \leq \| x_p - m_j^{(t)} \|^2\}, \forall j, 1 \leq j \leq k}$$
3. After all data points are centered in each cluster, we recalculate the position of the cluster center to make sure the cluster center is in the center of the cluster by calculating mean value of data points belonging to each cluster.
4. Step 2 and step 3 will be repeated until the position of the cluster center does not change or the center of all data points does not change.

2. Ideal of algorithm running in parallel:

For MPI:

1. The Master process read a data set.
2. Randomly choose samples from the dataset and initialize these samples as centroids. Initialize a centroid matrix.
3. Divide dataset into equal parts. The number of these equal parts should be equal number of processes.
4. These equal parts are scattered between each process.
5. Broadcast the centroid matrix to each process.
6. In each process there is part of data set and centroid matrix. So, it is able to carry out the step 2, 3 from sequential algorithm: calculate Euclidean distance matrix and determine clustering vectors in each process.
7. Gather the clustering vectors from each process in the same order in which the data sets were sent.
8. Calculate which clusters were faced and with what frequency in each process. This step is necessary for furthering recalculation of centroid means in each process.
9. Broadcast the result of step 8 to all processes and recalculate the means of centroid in each process how it was done in step 4 of sequential algorithm.
10. Gather the means of centroid from each process by summing them. It means for centroid which is responsible for 1st cluster it sums all means of 1st centroid from each processes and it does for each centroid.
11. Broadcast the result from previous step to all process and repeat step 5 from sequential algorithm, i.e., compare the means of old centroids with new centroids.
12. The program will finish when old centroids equal to new centroids that means there is no new centroids updated.

For OpenMP:

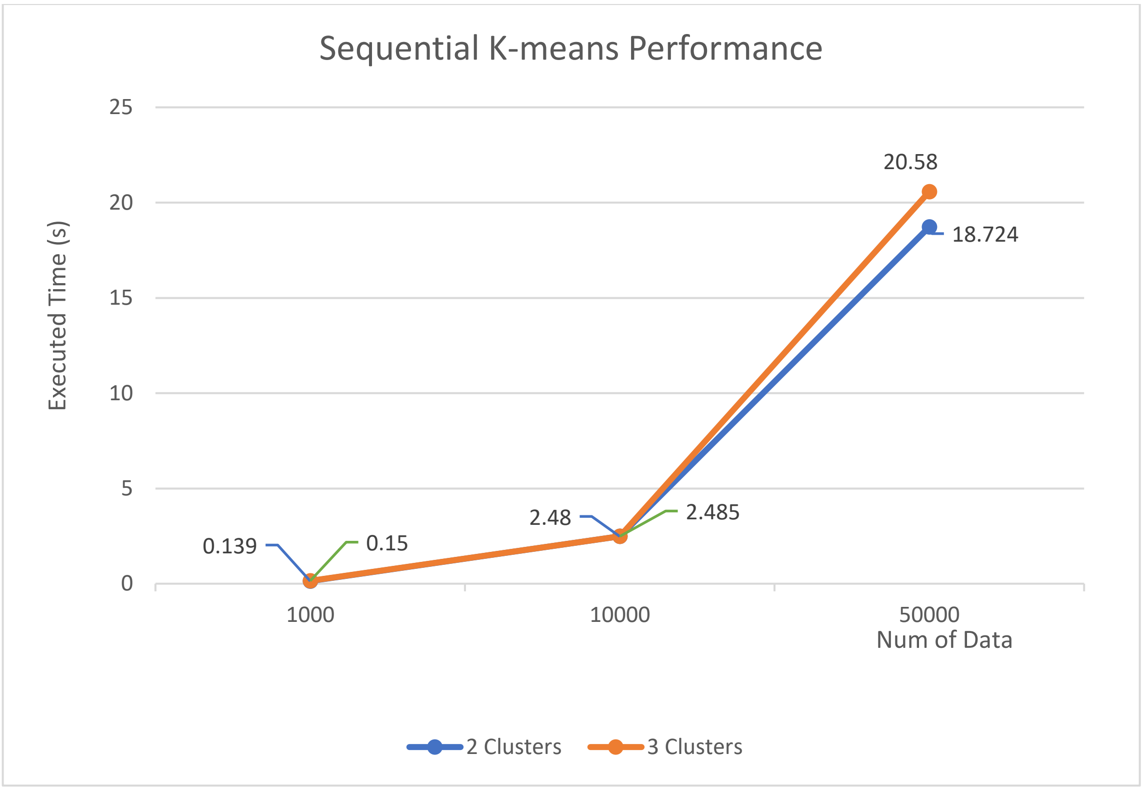
1. The Master thread read a data set.
2. Randomly choose samples from the dataset and initialize these samples as centroids. Initialize a centroid matrix.
3. Create number of ranges of index to access the dataset, which have to equal the number of threads.
4. Create a parallel region.
5. Calculate Euclidean distance between each datapoint and each centroid to assigns the data point to the cluster whose centroid is nearest to the data point in each thread.
6. Create critical region to change the position of the centroids by calculating the means of the respective data points in its cluster to update new centroids of cluster.
7. Step 4 - 6 keep on continuing until the iteration reaches MAX_ITERATION or until there is no change in the centroids position, which is calculated by the master thread after updating the centroid position by calculating the difference in the Euclidean distance between the previous centroid's position and the updated centroid's position.
8. The program will finish when there is no new centroids updated or out of condition in step 7.

IV. Result of MPI and OpenMP Implementation:

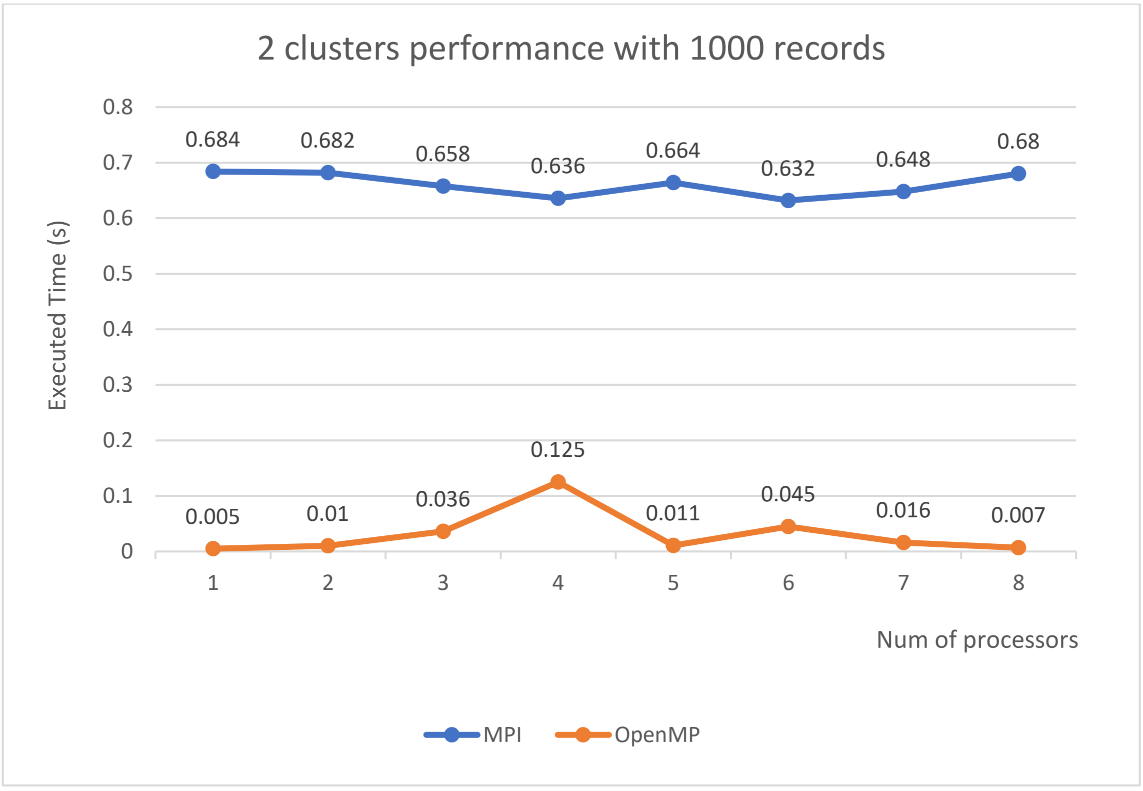
This implementation is tested with a 3D Road Network dataset which includes more than 40000 records of 3 features: Longitude, Latitude, and Altitude. In academy purpose, I used a ranges of data number which are 1000, 10000, 50000 to make a test from 2 to 3 clusters executing separately from 1 to 8 processors / threads in MPI and OpenMP integration respectively.

Test device is a laptop HP Inspiration with specification:

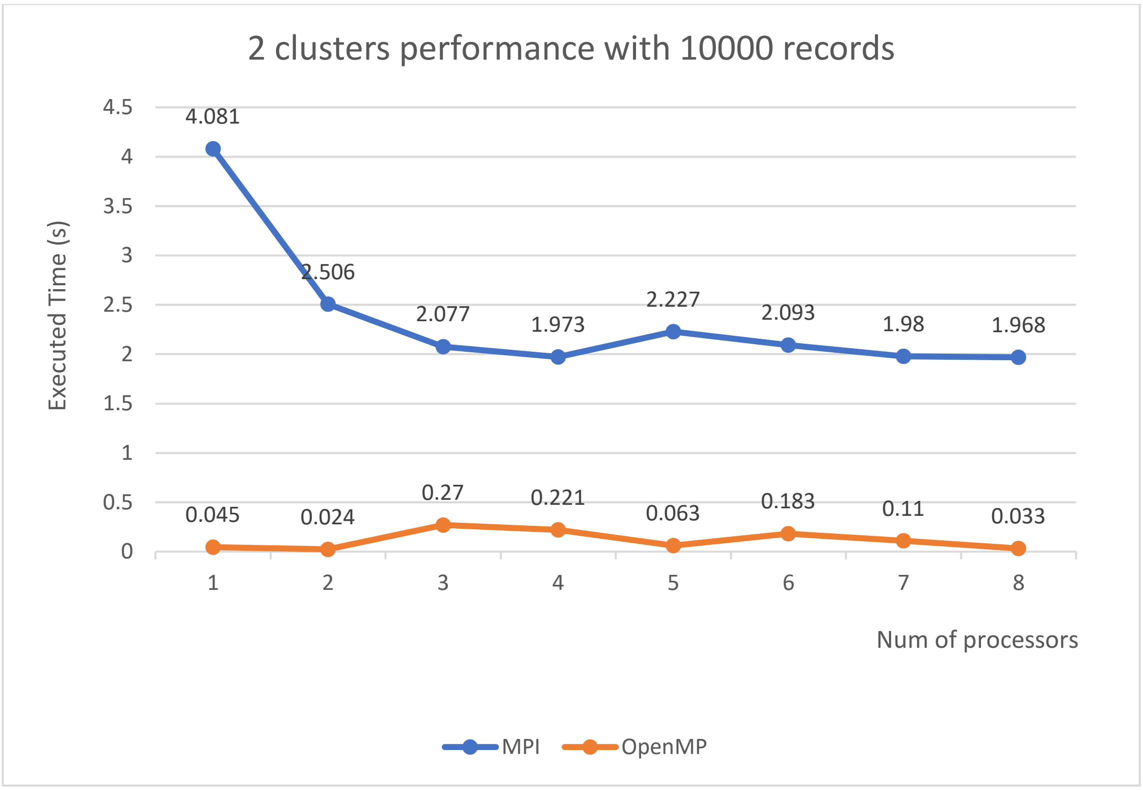
CPU: Intel(R) Core (TM) i7-8550U CPU @ 1.80GHz (4 cores, 8 threads)
RAM: 16GB.
Operation System: Windows 10 Home.

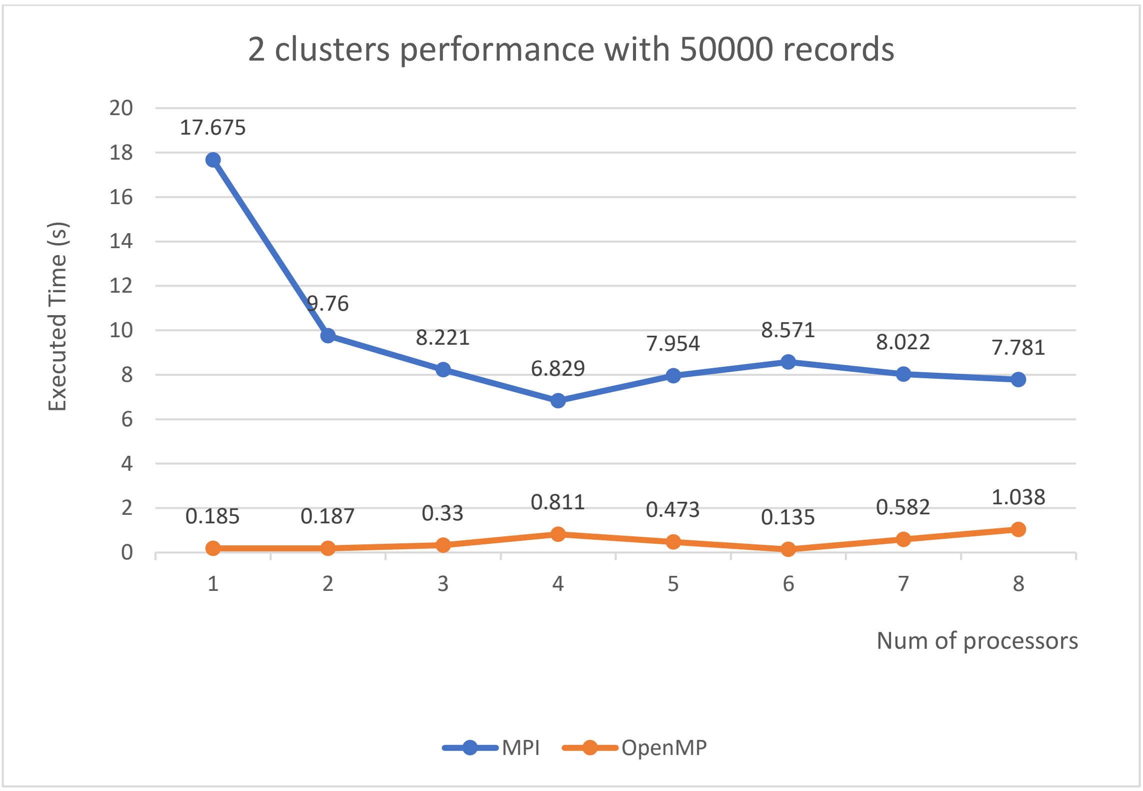


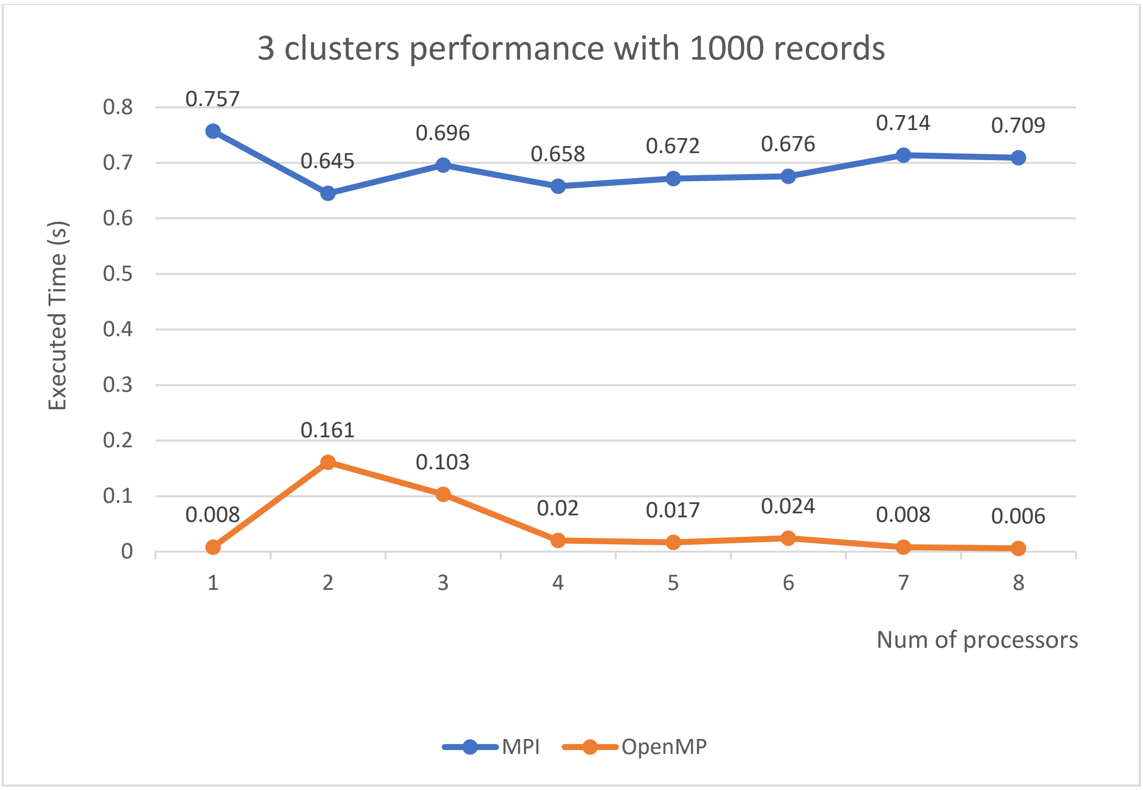
Sequential K-means Performance
Executed Time (s)
25
20
15
10
5
0
0.139
0.15
2.48
2.485
20.58
18.724
1000
10000
50000
Num of Data
2 Clusters
3 Clusters

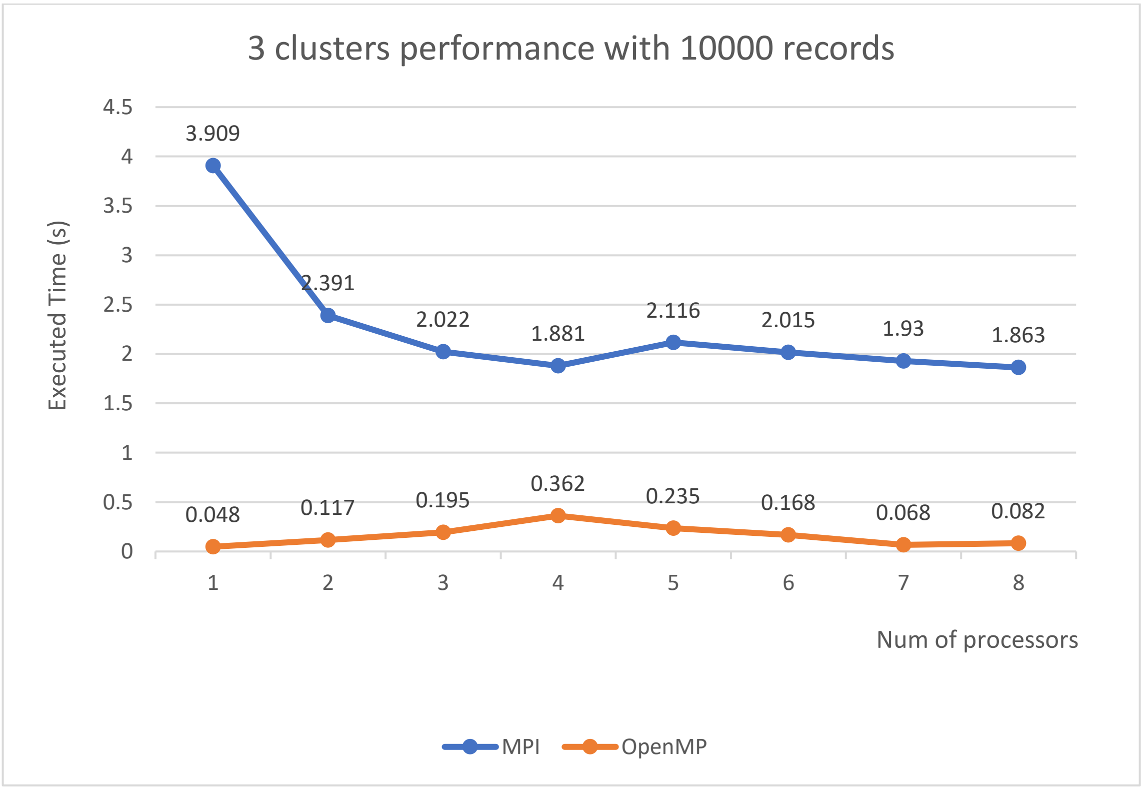


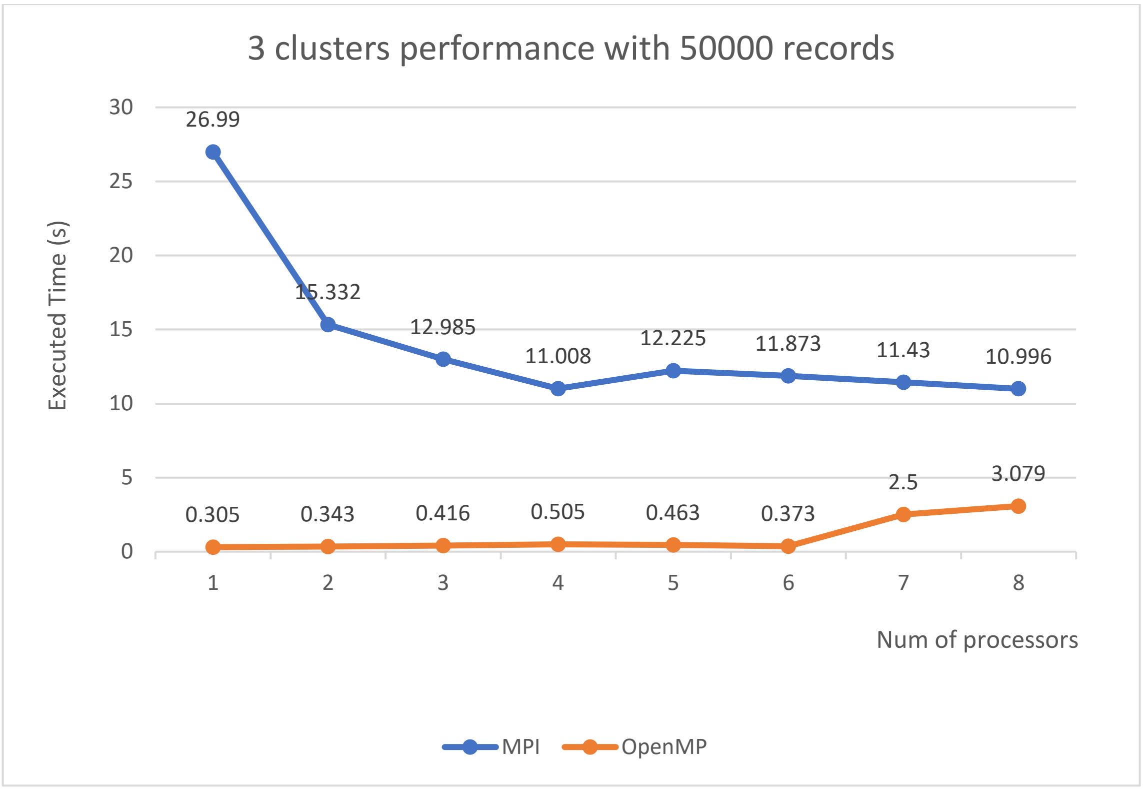
2 clusters performance with 1000 records
Executed Time (s)
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0.684
0.682
0.658
0.636
0.664
0.632
0.648
0.68
0.005
0.01
0.036
0.125
0.011
0.045
0.016
0.007
1
2
3
4
5
6
7
8
Num of processors
MPI
OpenMP


2 clusters performance with 10000 records
Executed Time (s)
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
4.081
2.506
2.077
1.973
2.227
2.093
1.98
1.968
0.045
0.024
0.27
0.221
0.063
0.183
0.11
0.033
1
2
3
4
5
6
7
8
Num of processors
MPI
OpenMP


2 clusters performance with 50000 records
Executed Time (s)
20
18
16
14
12
10
8
6
4
2
0
17.675
9.76
8.221
6.829
7.954
8.571
8.022
7.781
0.185
0.187
0.33
0.811
0.473
0.135
0.582
1.038
1
2
3
4
5
6
7
8
Num of processors
MPI
OpenMP


3 clusters performance with 1000 records
Executed Time (s)
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0.757
0.645
0.696
0.658
0.672
0.676
0.714
0.709
0.008
0.161
0.103
0.02
0.017
0.024
0.008
0.006
1
2
3
4
5
6
7
8
Num of processors
MPI
OpenMP


3 clusters performance with 10000 records
Executed Time (s)
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
3.909
2.391
2.022
1.881
2.116
2.015
1.93
1.863
0.048
0.117
0.195
0.362
0.235
0.168
0.068
0.082
1
2
3
4
5
6
7
8
Num of processors
MPI
OpenMP



Looking at the graph of algorithm testing result we can give out an overall picture that, time of execution of algorithm is steadily increasing when both number of clusters and number of dataset records are increasing in sequential mode, in specific, it is up from 0.139 seconds to 20.58 seconds in approximate for 2 clusters and 0.15 seconds to 18.724 seconds in approximate for 3 clusters with range of data from 1000 to 50000 in both.

In contrast, Using MPI and OpenMP get significantly effective results. OpenMP implementation got a better result than MPI one. In test of 1000 data for 2 and 3 clusters with range of 1 – 8 number of processors / threads there is no clearly decreasing executing time which MPI is about 0.64 to 0.757 seconds and OpenMP is about 0.005 to 0.125 seconds. The results have a significant change in data rage 10000 and 50000 records in both 2 and 3 clusters. MPI implementation method with distributed memory architecture has a steadily reduction of execution time, which is about 60 percent of decreasing in using from 1 to 8 processors. Next, In OpenMP implementation with shared memory architecture, which has some differences from trend of result. Particularly, there is a steady result of executing time when doing test in 10000 data, it is from 0.033 to 0.221 seconds for 2 clusters testing, and the lowest results returned in using 1, 2, 7 and 8 threads. For results of 3 cluster, it is about 0.033 – 0.221 seconds. Otherwise, result of testing in 50000 data, the executing time is stable when using 1 – 6 threads with approximate 0.135 seconds in 2 clusters and approximate 0.35 seconds in 3 clusters. However, there is a sign of reversal when using 7 - 8 threads, it increases to 1.038 seconds for 2 clusters and 3.079 seconds for 3 clusters.

V. Conclusion:

The goal of this project was to be familiar with general points of parallel programming and implementation of k-means algorithm to reach the boost of performance. The proposed aim was successful achieved and results of boosting were presented in both MPI and OpenMP implementation. However, these results are not optimal because of testing in virtual processors environment by only a multicore laptop with a 4 cores 8 threads, whereas MPI works best on node clusters of a big computer network such as super computer or cloud system (distributed system of hardware). Since this is a project within the scope of academic research, shortcomings and limitations are inevitable. However, applying parallelism to algorithms instead of doing sequential implementation, which saves execution time, optimizes the use of available hardware, and gives out many solutions to problems topic in life.

REFERENCES:

https://mpitutorial.com/

https://nyu-cds.github.io/python-mpi/05-collectives/

https://www.omnisci.com/technical-glossary/parallel-computing

https://en.wikipedia.org/wiki/OpenMP

https://en.wikipedia.org/wiki/Parallel_computing

https://en.wikipedia.org/wiki/Parallel_programming_model

https://www.bestcomputersciencedegrees.com/faq/what-is-parallel-programming/

https://stanford.edu/~cpiech/cs221/handouts/kmeans.html

http://personalpages.to.infn.it/~mignone/MPI/lecture1.pdf

https://hpc-forge.cineca.it/files/ScuolaCalcoloParallelo_WebDAV/public/anno-2015/24_Summer_School/openmp-slides.pdf