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:

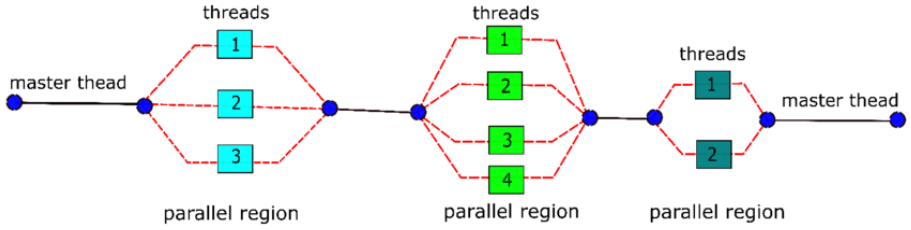
OpenMP is an open-source implementation of MPI, an Application Programming Interface (API) that supports multi-platform shared-memory multiprocessing programming, jointly defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared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

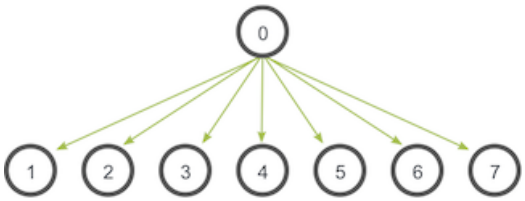
Sending and receiving are the two foundational concepts of MPI, which the process of communicating data follows a standard pattern, they are called "point-to-point" communication that is able to send and receive data between 2 specific processors. The data will be sent from sender rank to receiver rank 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)

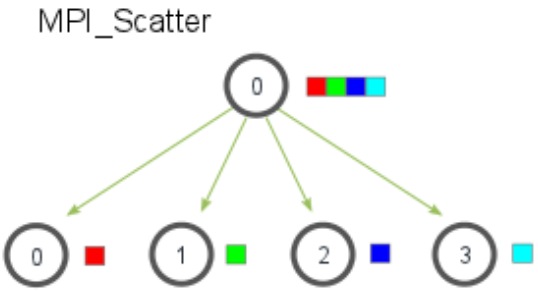
MPI_Receive(data, count, MPI_Datatype, source, tag, comm, status)

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.



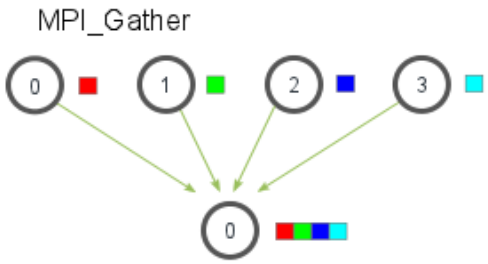
C/C++: **MPI_Bcast(data, count, MPI_Datatype, root, comm)**

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)***

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)***

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)

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)***

- 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, $\mathbf{K < N}$

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 Euclidean distance between the point that is the cluster center and the point in the dataset based on the Euclidean algorithm:

$$S_i^{(t)} = \{x_p : \| x_p - m_i^{(t)} \| 2 \leq \| x_p - m_j^{(t)} \|^2\} , \forall j, 1 \leq j \leq k$$

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. Choose first samples associate with number of clusters from the dataset and initialize these samples as centroids. Initialize a centroid matrix.
3. Scatter all data points divided by number of processes to each process.
4. Broadcast the centroid matrix to each process.
5. In each process there is part of data set and centroid matrix. So, it is able to carry out calculating the Euclidean distance matrix and determine clustering vectors in each process.
6. Calculate which clusters were faced and with what frequency in each process by counting number of data points and doing sum value of data points in each centroid that belongs to. This step is necessary for furthering recalculation of centroid means in each process.
7. Make a sum operation for the counts and the sums of the data points from each process into main process by using Reduce method.
8. In main process, update new centroids by calculating the mean of the sum and the count, and calculate the difference in the Euclidean distance between the previous centroid's position and the updated centroid's position.
9. Broadcast the results of new centroids and distance in step 8 to all processes.
10. The program will finish when old centroids equal to new centroids that means there is no new centroids updated or the distance result equal 0, if it is not, repeat from step 5 to step 9.

For OpenMP:

1. The Master thread read a data set.
2. Choose first samples associate with number of clusters 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 the 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. Wait for all threads reach at a point by barrier region to calculate the difference in the Euclidean distance between the previous centroid's position and the updated centroid's position in thread 0.
8. Step 4 - 7 keep on continuing until the iteration reaches MAX_ITERATION by your definition or until there is no change in the centroid's position after the calculation in step 7.
9. The program will finish when there are no new centroids updated or out of condition in step 8.

IV. Implementation.

1. Decomposition and Partition:

Data decomposition: Split input data by dividing all data by the number of processes (MPI) or threads (OpenMP) in each time running program. In detail, splitting the index of data in OpenMP implementation and splitting the amount of data in MPI implementation, which help the program will access the data independently in each process or thread during parallel executions. Then the output data will be composited to main process or thread.

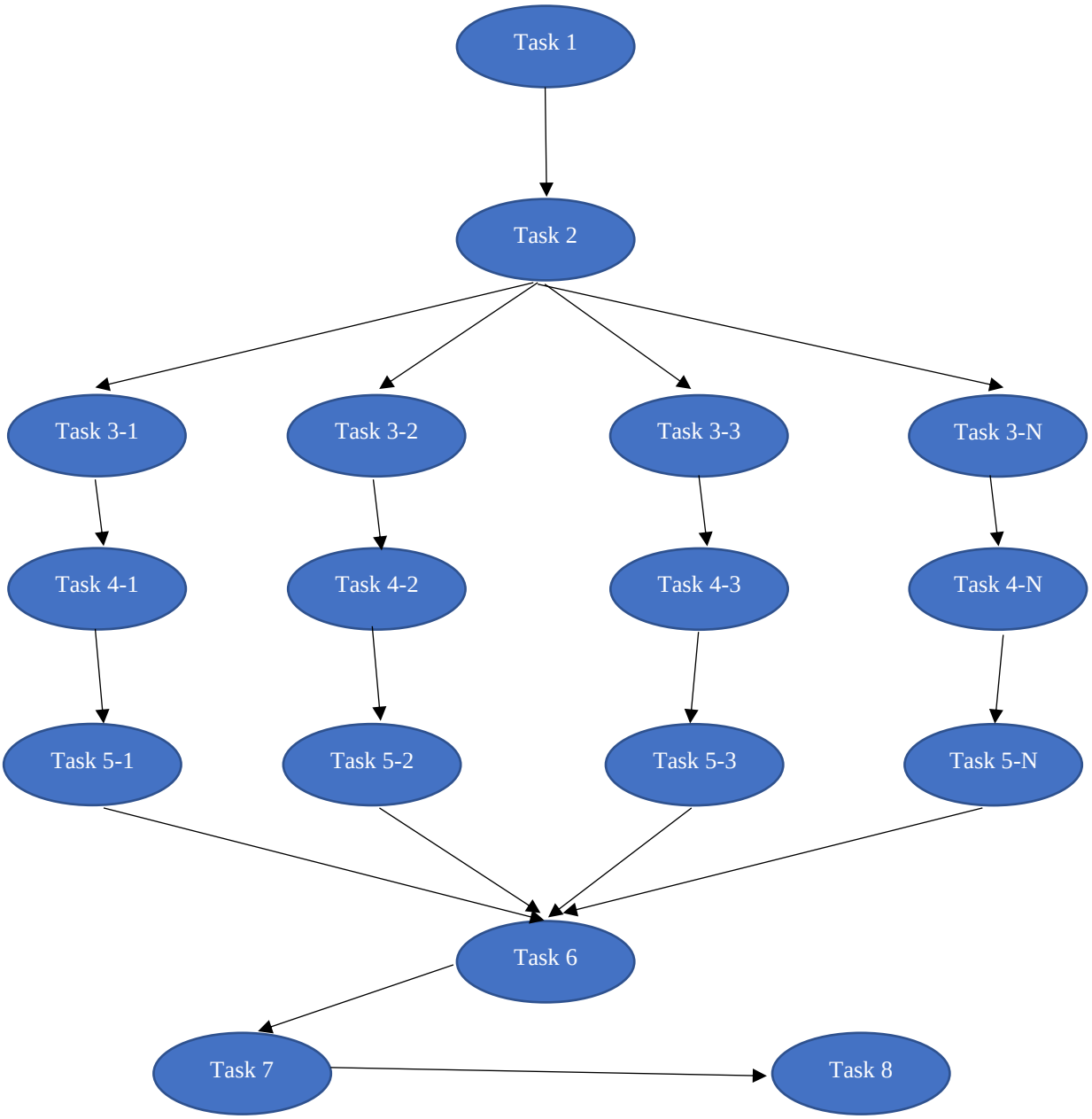
Control flow decomposition: Using recursive decomposition technique for executing parallel tasks to solve problem.

The problem is split into some sub tasks which are both concurrency and sequence. There are 8 tasks in whole program being executed to give out the final result, in which the number of concurrent tasks can be run and the maximum number of tasks running at an executing point is 3 for MPI implementation, but it is 4 for OpenMP implementation, because the different way of interacting with data between them. So, the number of these tasks is degree of concurrency of decomposition and maximum degree.

Partitioning based on data decomposition and task decomposition, the number of partitioning depends on the number of processes or threads that created during executing program. In addition, the tasks in concurrency will be executed the same in every process or thread, and the data is different from each of them, so the concurrent tasks will be replicated as the number of sub-tasks respectively to the number of processes or threads.

MPI:

- Tasks in Concurrency:
  +3rd Task: Calculate the distance between centroid points and each data point to decide which centroid the data point will belongs to.
  + 4th Task: Count the number of data points that belong to centroid.
  + 5th Task: Count the sum value of data points that belong to centroid.
- Tasks in Sequence:
  + 1st Task: Split data based on division of the amount of data to the number of processes.
  + 2nd Task: Scatter and broadcast data to each process before doing parallel.
  + 6th Task: Sum operation by making Reduce method for counts and sums in each process to main process.
  + 7th Task: Update new centroids by the mean of counts and sums value.
  + 8th Task: Broadcast new centroids and distance value to each process.

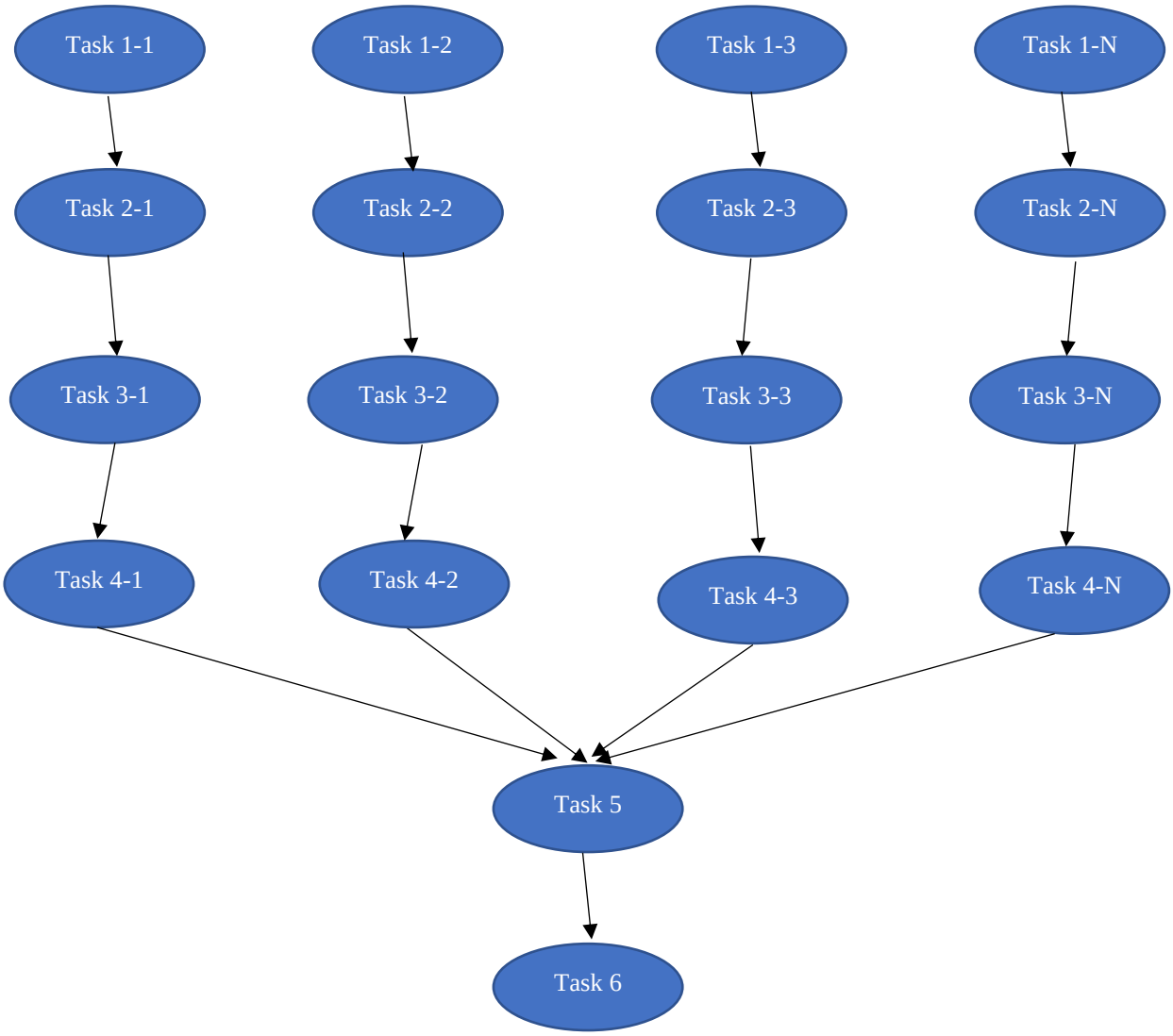


MPI: Critical Path Length of Task Decomposition

(* Note: N - number of processes)

OpenMP:

- Tasks in Concurrency:
  + 1st Task: Split data's index based on division of the amount of data to the number of threads.
  + 2nd Task: Calculate the distance between centroid points and each data point to decide which centroid the data point will belongs to.
  + 3rd Task: Count the number of data points that belong to centroid.
  + 4th Task: Count the sum value of data points that belong to centroid.
- Tasks in Sequence:
  + 5th Task: Calculate the mean value of counts and sums value.
  + 6th Task: Update new centroids and calculate the distance between old and new centroids by the mean value.



OpenMP: Critical Path Length of Task Decomposition

(* Note: N - number of threads)

2. Communication:

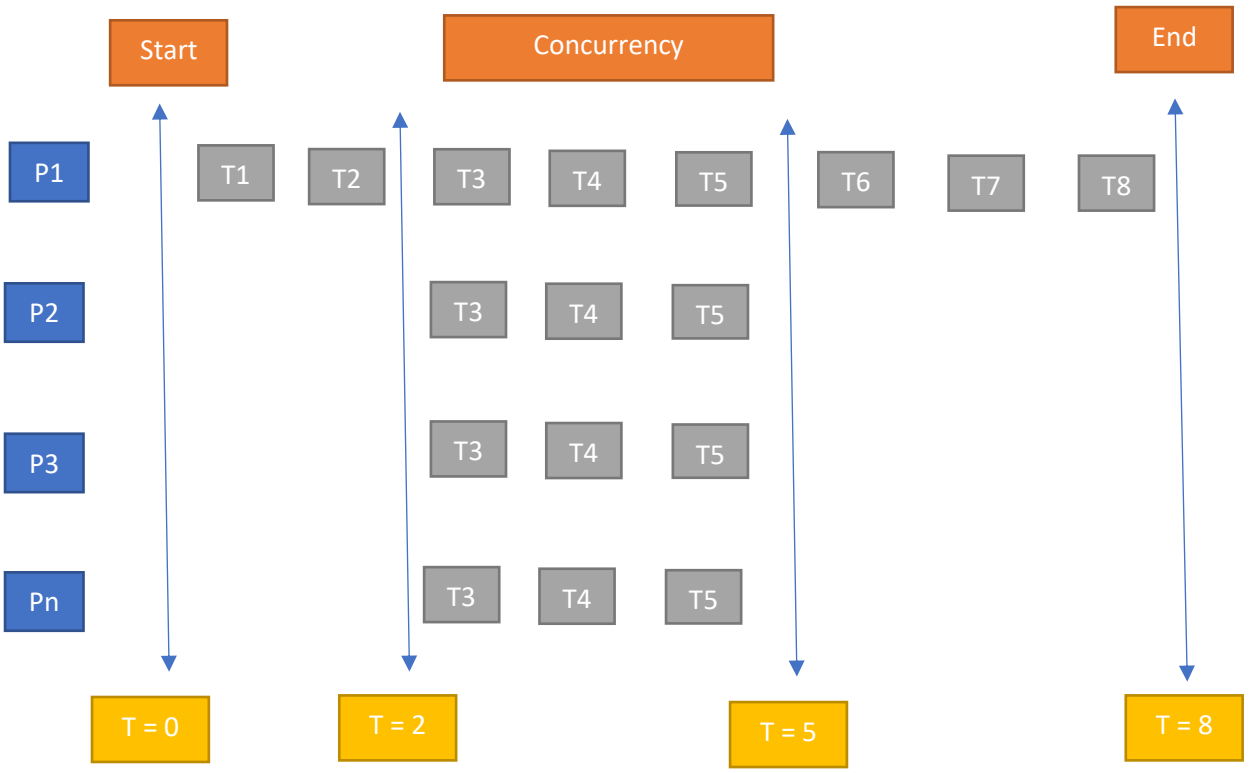
MPI:

I used collective communication techniques which are scatter, broadcast, gather, reduce method to transfer, update or share data among processes for this project. Broadcast helps send the same data (initial centroids and updated centroids) from root process to all processes. And scatter involves to send different chunks data from root process to all processes in a communicator. In contrast, gather is an inverse of scatter method that collects elements from many processes to one single process.

Reduce method is used for sum operation purpose by collecting the data from all processes then calculating a sum and return result to root process only.

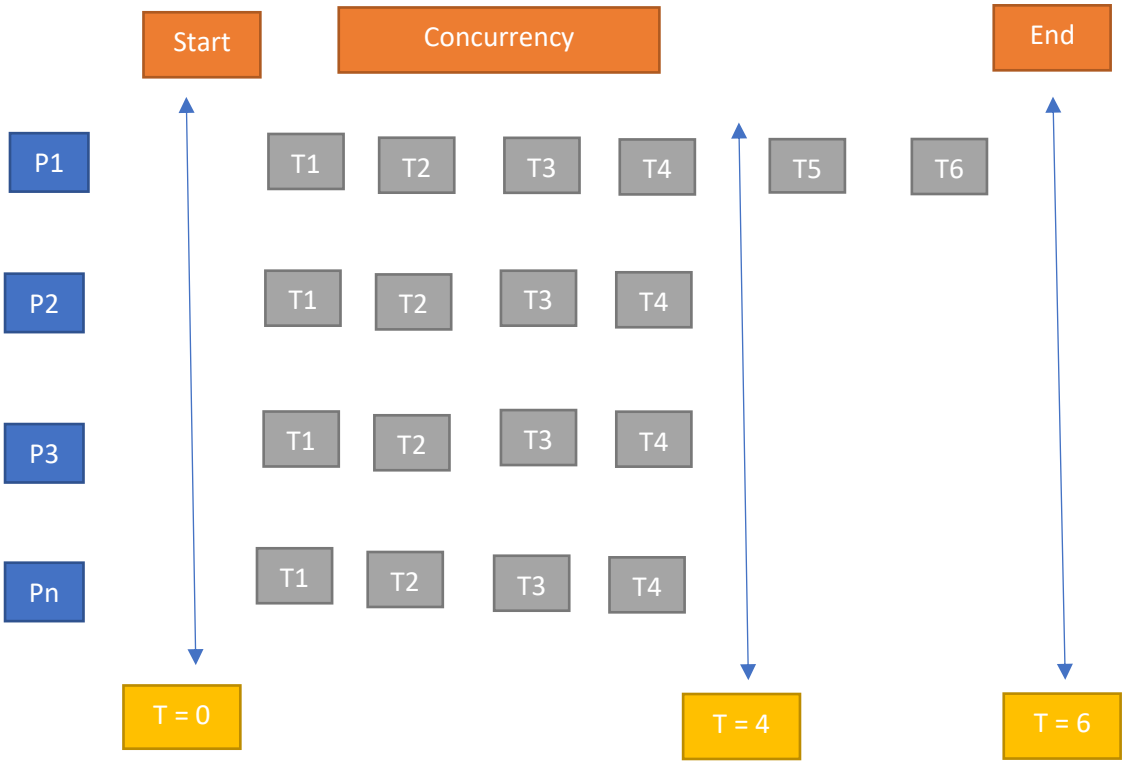
3. Mapping:

Because of basing on data partitioning, there is dynamic mapping technique being used in this project. With the same concurrent tasks running on every process or thread, the replicated concurrent sub-tasks will be assigned when executing program with parameter of the number of processes or threads. The tasks will be mapped to process or thread at runtime.



MPI Mapping the task to process

(* Note: n – number of processes)



OpenMP Mapping the task to thread

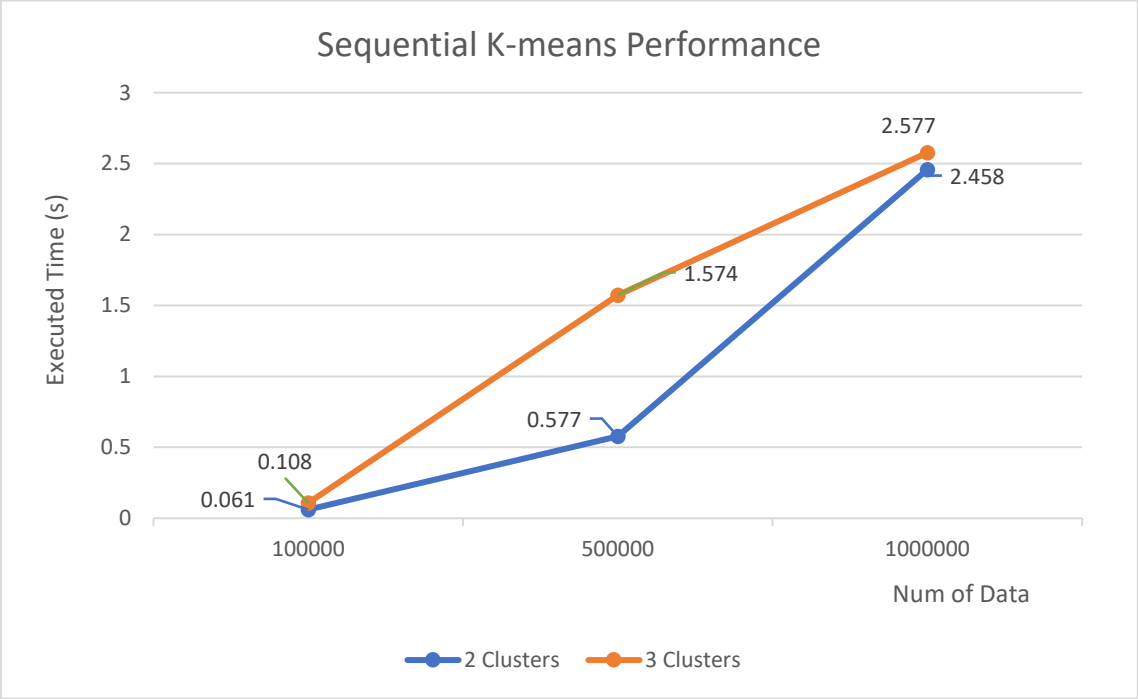
(* Note: n – number of threads)

IV. Result of MPI and OpenMP Implementation:

In academy purpose, I used a ranges of different data numbers which are 100k, 500k, 1000k records of random numbers in 3 features to make testing 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.



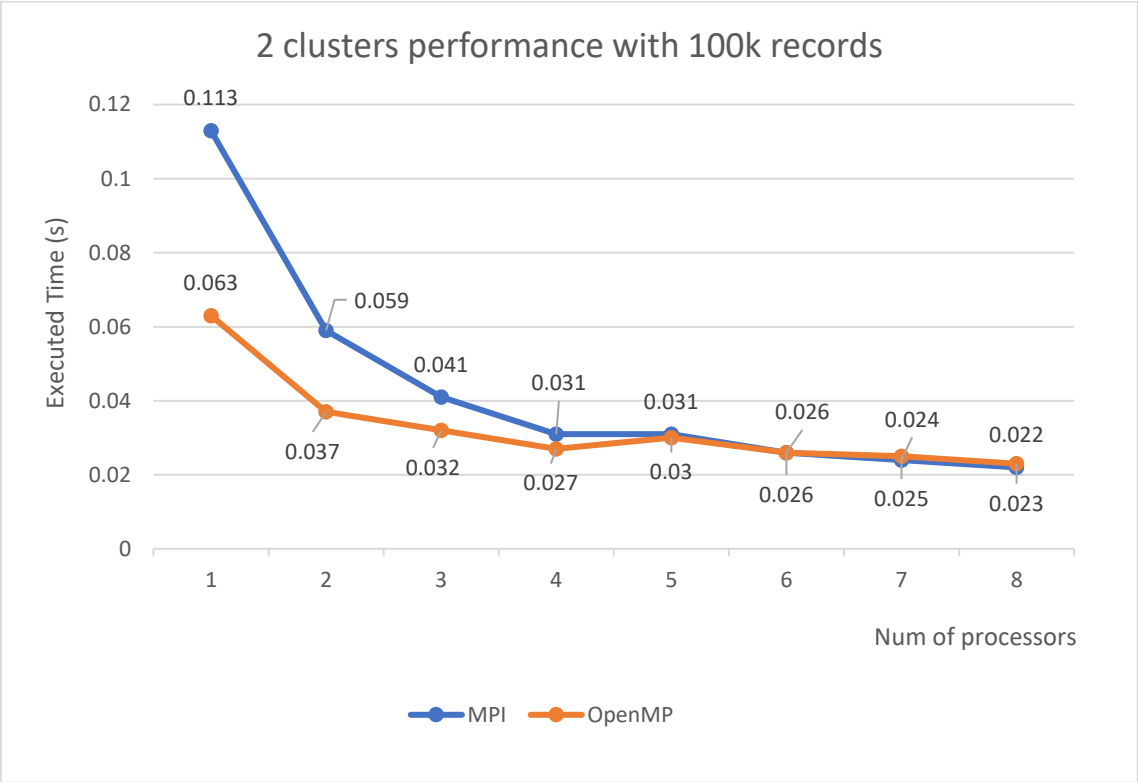
Looking at the graph of algorithm testing result we can give out an overall picture that, time of execution of algorithm is steadily increasing when both number of clusters and number of dataset records are increasing in sequential mode, in specific, it is up from 0.061 seconds to 2.458 seconds in approximate for 2 clusters and 0.108 seconds to 2.577 seconds in approximate for 3 clusters with different data and the amount of data from 100k to 1000k.

Applying Amdahl Law in parallelism, we can calculate the speed-up to measure the performance increase when increasing the processors or cores, and can find out efficiency that express the ability of exploiting the resources, which measures the work rate per processor or core. Following the result of program, we can use some Amdahl Law's formulas to find out speed-up and efficiency value:

$$f_s = T_s / (T_s + T_p) \quad \textit{(Serial fraction)}$$

$$S_n = T_1 / T_n = n / (n * f_s + (1 - f_s)) \quad \textit{(Speed up)}$$

$$E_n = S_n / n = 1 / (n * f_s + (1 - f_s)) \quad \textit{(Efficiency)}$$



MPI Speed-up and Efficiency:

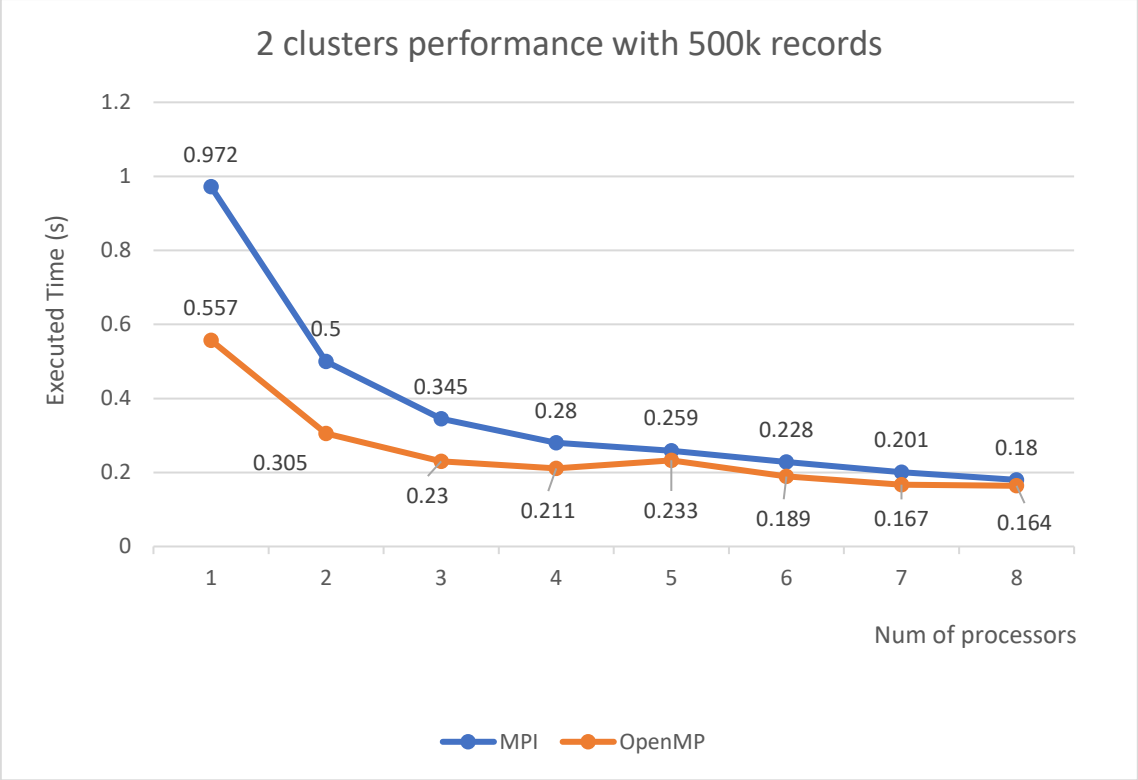
$S_2 = 1.915$ $S_3 = 2.75$ $S_4 = 3.645$ $S_5 = 3.645$
$S_6 = 4.346$ $S_7 = 4.708$ $S_8 = 5.136$

$E_2 = 0.957$ $E_3 = 0.918$ $E_4 = 0.911$ $E_5 = 0.729$
$E_6 = 0.724$ $E_7 = 0.672$ $E_8 = 0.642$

OpenMP Speed-up and Efficiency:

$S_2 = 1.702$ $S_3 = 1.968$ $S_4 = 2.333$ $S_5 = 2.1$
$S_6 = 2.423$ $S_7 = 2.52$ $S_8 = 2.739$

$E_2 = 0.851$ $E_3 = 0.656$ $E_4 = 0.583$ $E_5 = 0.42$
$E_6 = 0.403$ $E_7 = 0.36$ $E_8 = 0.342$



MPI Speed-up and Efficiency:

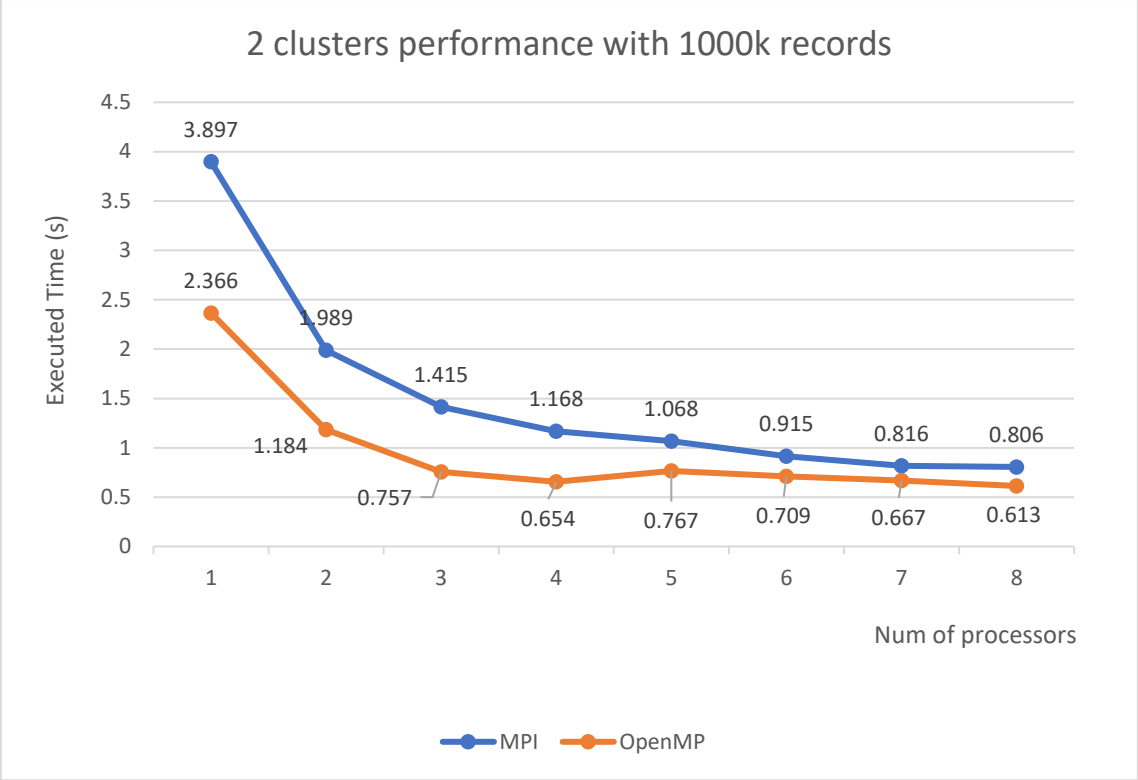
$S_2 = 1.944$ $S_3 = 2.817$ $S_4 = 3.471$ $S_5 = 3.752$
$S_6 = 4.263$ $S_7 = 4.835$ $S_8 = 5.4$

$E_2 = 0.972$ $E_3 = 0.939$ $E_4 = 0.867$ $E_5 = 0.750$
$E_6 = 0.710$ $E_7 = 0.690$ $E_8 = 0.675$

OpenMP Speed-up and Efficiency:

$S_2 = 1.826$ $S_3 = 2.421$ $S_4 = 2.639$ $S_5 = 2.390$
$S_6 = 2.947$ $S_7 = 3.335$ $S_8 = 3.396$

$E_2 = 0.913$ $E_3 = 0.807$ $E_4 = 0.659$ $E_5 = 0.478$
$E_6 = 0.491$ $E_7 = 0.476$ $E_8 = 0.424$



MPI Speed-up and Efficiency:

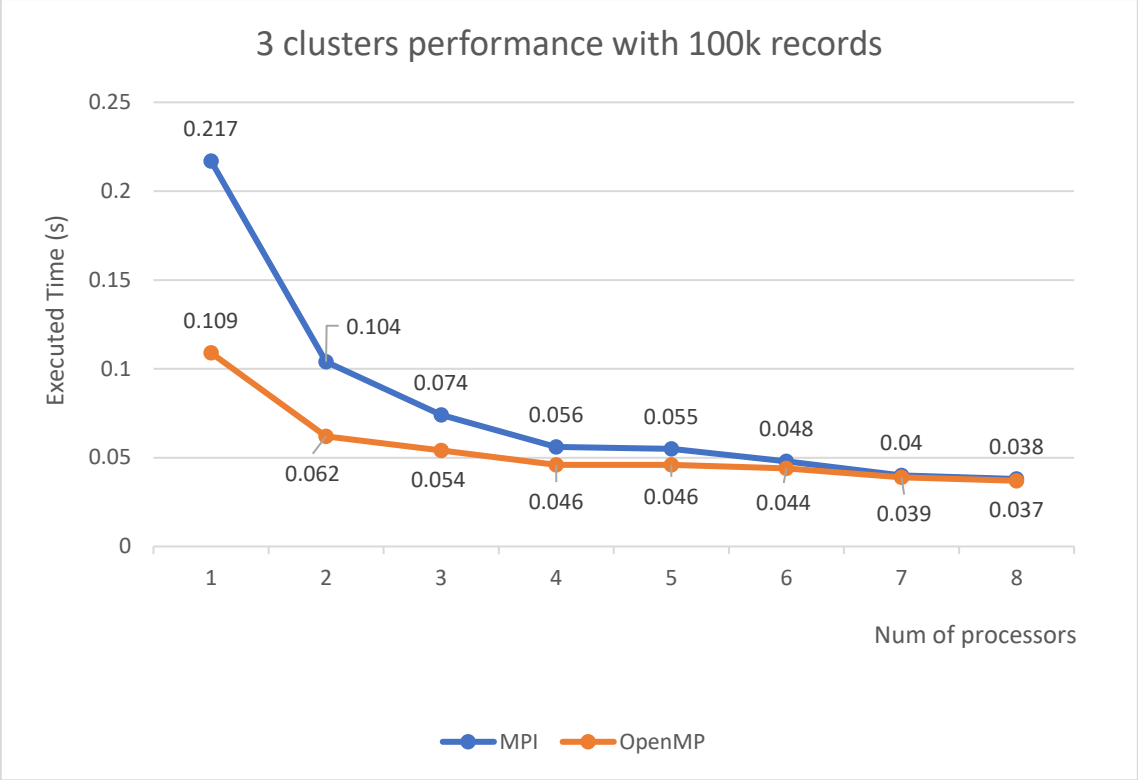
$S_2 = 1.959$ $S_3 = 2.754$ $S_4 = 3.336$ $S_5 = 3.648$
$S_6 = 4.259$ $S_7 = 4.775$ $S_8 = 4.834$

$E_2 = 0.979$ $E_3 = 0.918$ $E_4 = 0.834$ $E_5 = 0.729$
$E_6 = 0.709$ $E_7 = 0.682$ $E_8 = 0.604$

OpenMP Speed-up and Efficiency:

$S_2 = 1.998$ $S_3 = 3.125$ $S_4 = 3.617$ $S_5 = 3.084$
$S_6 = 3.337$ $S_7 = 3.547$ $S_8 = 3.759$

$E_2 = 0.999$ $E_3 = 1.041$ $E_4 = 0.904$ $E_5 = 0.616$
$E_6 = 0.556$ $E_7 = 0.506$ $E_8 = 0.482$



MPI Speed-up and Efficiency:

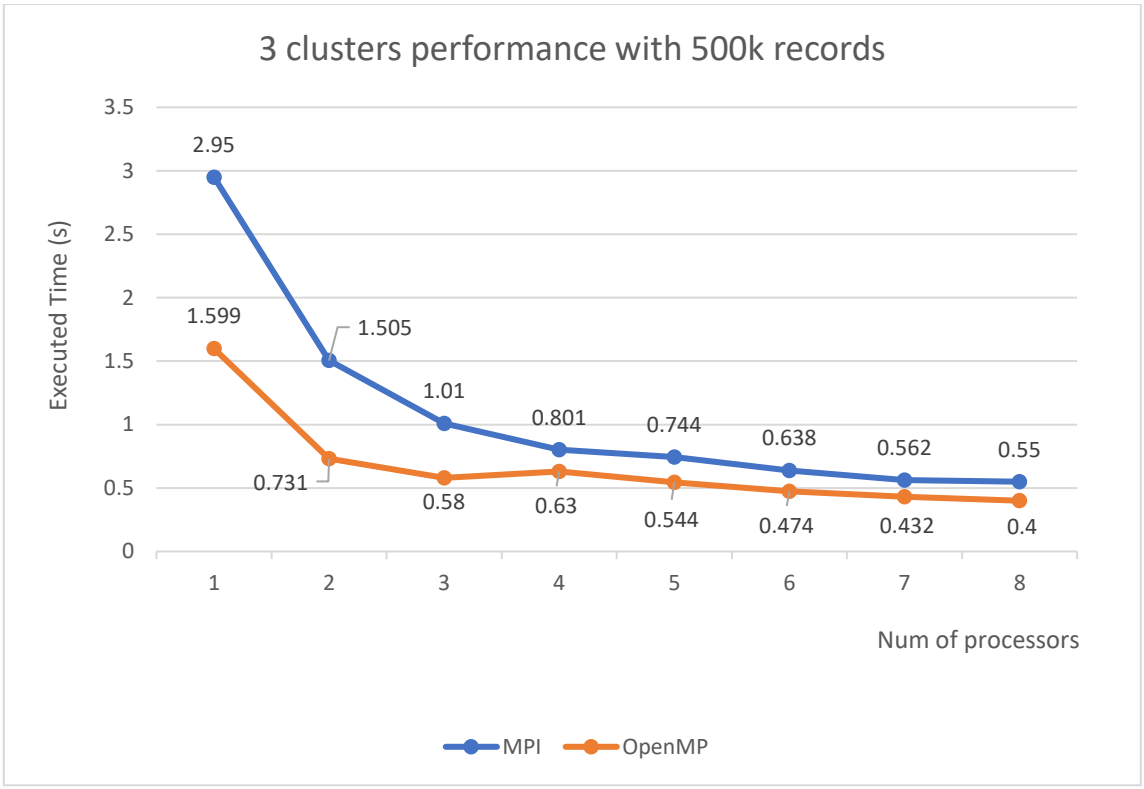
$S_2 = 2.086$ $S_3 = 2.932$ $S_4 = 3.875$ $S_5 = 3.945$
$S_6 = 4.520$ $S_7 = 5.425$ $S_8 = 5.710$

$E_2 = 1.043$ $E_3 = 0.977$ $E_4 = 0.968$ $E_5 = 0.789$
$E_6 = 0.753$ $E_7 = 0.775$ $E_8 = 0.713$

OpenMP Speed-up and Efficiency:

$S_2 = 1.758$ $S_3 = 2.018$ $S_4 = 2.369$ $S_5 = 2.369$
$S_6 = 2.477$ $S_7 = 2.794$ $S_8 = 2.945$

$E_2 = 0.879$ $E_3 = 0.672$ $E_4 = 0.592$ $E_5 = 0.473$
$E_6 = 0.412$ $E_7 = 0.399$ $E_8 = 0.368$



MPI Speed-up and Efficiency:

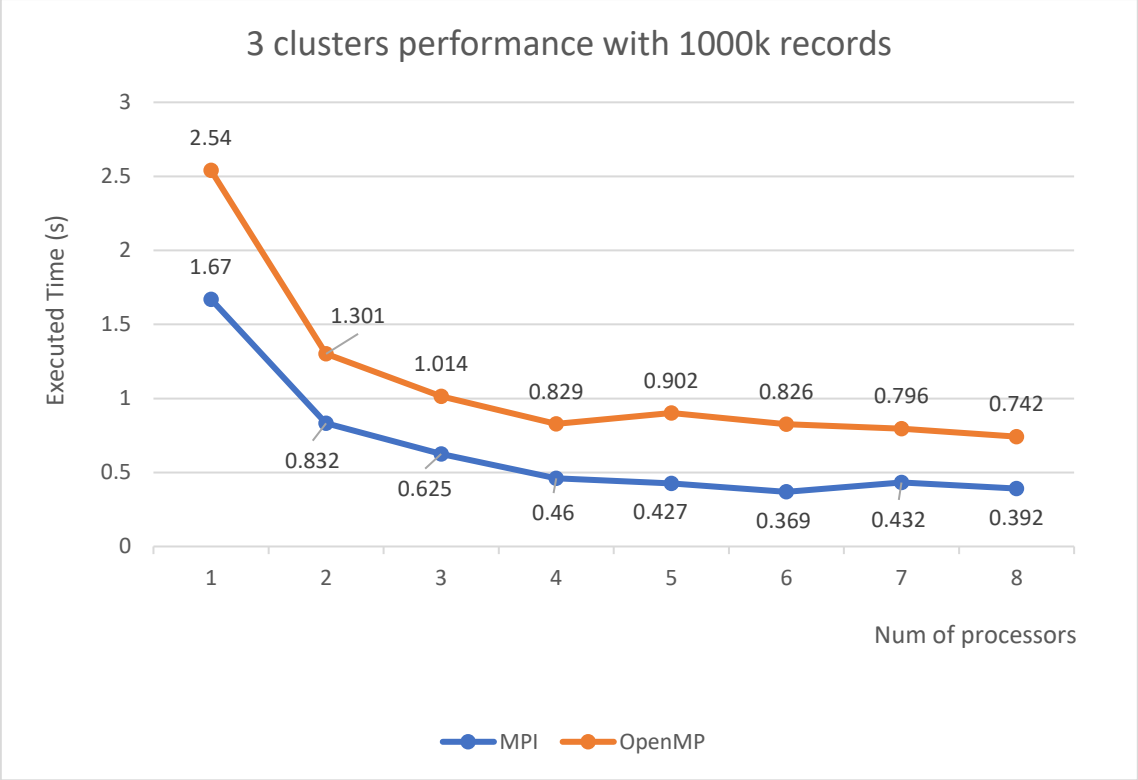
$S_2 = 1.960$ $S_3 = 2.920$ $S_4 = 3.682$ $S_5 = 3.965$
$S_6 = 4.623$ $S_7 = 5.249$ $S_8 = 5.363$

$E_2 = 0.980$ $E_3 = 0.973$ $E_4 = 0.920$ $E_5 = 0.793$
$E_6 = 0.770$ $E_7 = 0.749$ $E_8 = 0.670$

OpenMP Speed-up and Efficiency:

$S_2 = 2.187$ $S_3 = 2.756$ $S_4 = 2.538$ $S_5 = 2.939$
$S_6 = 3.373$ $S_7 = 3.701$ $S_8 = 3.997$

$E_2 = 1.093$ $E_3 = 0.918$ $E_4 = 0.634$ $E_5 = 0.587$
$E_6 = 0.562$ $E_7 = 0.528$ $E_8 = 0.499$



MPI Speed-up and Efficiency:

$S_2 = 2.007$ $S_3 = 2.672$ $S_4 = 3.630$ $S_5 = 3.911$
$S_6 = 4.525$ $S_7 = 3.865$ $S_8 = 4.260$

$E_2 = 1.003$ $E_3 = 0.890$ $E_4 = 0.907$ $E_5 = 0.782$
$E_6 = 0.754$ $E_7 = 0.552$ $E_8 = 0.532$

OpenMP Speed-up and Efficiency:

$S_2 = 1.952$ $S_3 = 2.504$ $S_4 = 3.063$ $S_5 = 2.815$
$S_6 = 3.075$ $S_7 = 3.190$ $S_8 = 3.423$

$E_2 = 0.976$ $E_3 = 0.834$ $E_4 = 0.765$ $E_5 = 0.563$
$E_6 = 0.512$ $E_7 = 0.455$ $E_8 = 0.427$

Using MPI and OpenMP get significantly effective results by decreasing trend. OpenMP implementation got a better result than MPI one in all test, excepting the test with 1000k records for 3 clusters finding. Both MPI and OpenMP implementation have a good result of efficiency when using 2 – 4 processors or threads, which it is almost approximate more than 85 percent and 50 percent. That means saving about a half of time in execution when using 2 processors or threads and a quarter of time in using 4 processors or threads. The efficiency reached to 40 percent and upper when using 8 processors and threads in 6 cases of test with 3 kind of data amount and 2 kind of 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