Sequential and Parallel Execution of K-Means Clustering Algorithm

([1]Isha Gupta, [2]Aishwarya R, [3]Angel Paul, [4]Ananya Muralidhar, [5]Aditi Mallya)
(USN: [1]1MS18CS049, [2]1MS18CS012, [3]1ms18cs019, [4]1ms18cs018, [5]1ms18cs010)

ABSTRACT

Pthreads is an execution model that exists independently from a language. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the pthreads API. Clustering also called the unsupervised learning method which involves grouping of data items into clusters that have high similarity but are dissimilar to data items of other clusters. The clustering method is used in many areas, for instance, computer sciences, engineering, Earth sciences, life and medical sciences, economics, social sciences, etc. K-means has been used in many clustering work because of the ease of the algorithm. K-means clustering is one of the simplest and popular unsupervised machine learning algorithms. It is a method of vector quantization, that 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. The two types of execution are sequential and parallel execution. Sequential K-means clustering is a modification to the traditional K-means method where the algorithm acts on subsets of supplied data sequentially rather than in a single batch. parallel K-mean clustering algorithm, the dataset is partitioned addicted to subparts, so less space and processing speed will be required to process subparts of the datasets. The parallel K-means algorithm is designed in such a way that each P participating node is responsible for handling n/P data points.

INTRODUCTION

K-Means Clustering is an Unsupervised Learning algorithm, which groups the unlabelled dataset into different clusters. It allows us to cluster the data into different groups and a convenient way to discover the categories of groups in the unlabelled dataset on its own without the need for any training. It is a centroid-based algorithm, where each cluster is associated with a centroid. The main aim of this algorithm is to minimize the sum of distances between the data point and their corresponding clusters. The algorithm takes the unlabelled dataset as input, divides the dataset into k-number of clusters, and repeats the process until it does not find the best clusters. The value of k should be predetermined in this algorithm.

The k-means clustering algorithm mainly performs two tasks:

- Determines the best value for K centre points or centroids by an iterative process.
- Assigns each data point to its closest k-centre. Those data points which are near to the particular k-centre, create a cluster.

Hence each cluster has data points with some commonalities, and it is away from other clusters.

Sequential K-means clustering is a modification to the traditional K-means method where the algorithm acts on subsets of supplied data sequentially rather than in a single batch. This is important in two ways:

- Clusters and their associated centroids can be discerned without loading an entire dataset into memory
- Streaming/online updates to a model can be made, letting them react to changes in the supplied data

Both traditional and sequential K-means are initialized in the same manner – using K-means++ or random assignment.

Most critical issues for Simple K-Mean clustering algorithms are the space and processing speed requirements, when the dataset is very large. To resolve these issues, the Simple K-Mean clustering algorithm is parallelized.

Three main steps of the Simple Parallel K-Mean are

- Partition
- Computation
- Compilation

In the first step at the server side, sub-datasets are produced from a given dataset. All client systems that are connected to the server receive these sub-datasets with the number of clusters, i.e., “k” and initial centroids. The said client systems calculate clusters and forward those results to the server. This process continues till there is no change in the clusters.

Parallelism techniques used are:

i)**Pthreads** is a standardised methodology for partitioning a program into subtasks that can be executed in parallel or interleaved. Pthreads gets its "P" from POSIX (Portable Operating System Interface), a set of IEEE operating system interface standards in which it is defined. Other thread models, such as Mach Threads and NT Threads, have existed and continue to exist. Pthreads are defined as a set of C language programming types and calls with a set of implicit semantics, according to programmers. Pthreads implementations are usually provided in the form of a header file that can be included in the program and a library to which the program can be linked. Pthreads can be used as a kernel-level threads library or as a user-level threads library. Most thread operations need a system call in kernel-level implementations, and the kernel scheduler schedules the threads. This technique provides a single, uniform thread model with access to system-wide resources, but at the cost of making thread operations rather costly. Most user-level thread actions, such as creation, synchronisation, and context switching, may be performed in user space without requiring kernel intervention, making them much less expensive than similar kernel thread operations. As a result, parallel programmes can be written using a large number of lightweight, user-level Pthreads, with the scheduling and load balancing handled by the threads, resulting in simple, well-structured, and architecture-independent code.

The primary goal of implementing Pthreads is to increase programme performance. A thread requires significantly less operating system overhead as compared to the cost of establishing and administering a process. Process management consumes more system resources than thread management. Threads in a process share the same address space. Inter-thread communication is more efficient and user-friendly than inter-process communication in many cases. In a number of ways, threaded applications provide possible speed improvements and practical advantages over non-threaded programmes. Multi-threaded programmes will work on a single-processor system but will immediately switch to a multiprocessor computer without having to recompile. In a multiprocessor system, the most important reason to use Pthreads is to take use of available parallelism.

ii) **OpenMP** is a library for parallel programming in the SMP (symmetric multi-processors, or shared-memory processors) model. When programming with OpenMP, all threads share memory and data. OpenMP supports C, C++ and Fortran. The OpenMP functions are included in a header file called omp.h. The fork-join approach of parallel execution is used by OpenMP. When a thread sees a parallel construct, it forms a team with itself and a small number of other threads (potentially zero). The encountering thread takes over as the new team's leader. The team's other threads are referred to as slave threads. Within the parallel construct, all team members run the code. When a thread completes its work within a parallel construct, it waits at the implicit barrier at the construct's completion. The threads can leave the barrier after all of the team members have arrived. The master thread continues to run user code after the parallel construct has ended, while the slave threads wait to be called to join other teams. Parallel sections in OpenMP can be nested within each other. If nested parallelism is deactivated, a thread meeting a parallel construct inside a parallel region creates a new team that solely consists of the experiencing thread. If nested parallelism is enabled, the new team might be made up of many threads. The OpenMP runtime library keeps track of a pool of slave threads that may be utilised in parallel areas. When a thread sees a parallel construct and wants to form a team of more than one thread, it will search the pool for idle threads and assign them to the team as slave threads. If there aren't enough idle threads in the pool, the master thread may receive fewer slave threads than it requires. The slave threads return to the pool once the team has completed executing the parallel area.

METHOD

A **struct** named **Point is created** with attributes: -

1) **X ->** X-coordinate of the point: type double
2) **Y-**>Y-coordinate of the point: type double
3) **Z ->** Z-coordinate of the point: type double
4) **Cluster ->** Cluster # to which the point belongs: type int

Following are the helper functions that are implemented along with their Descriptions:

1. **mean_recompute**-> Given N,K, the array of struct of data points and array of centroid Points recomputes the Centroid Locations by taking average of locations of the Data points in a particular Cluster
2. **addtwo**-> Given 2 points of type Point(struct), returns a new point as the sum of the Given two points, assuming that both the points belong to the same cluster
3. **euclid**-> Given 2 points of type Point(struct), returns the Euclid's distance between them
4. **assignclusters**-> Given N,K, the array of struct of data points and array of centroid Points recomputes, the nearest cluster for all points and reassigns their values of Point.cluster using euclid
5. **putback**-> puts back values of the centroids in the global vector to be returned by driver func
6. **checkClosestCluster ->** helper for assign clusters function

The algorithm is assumed to converge when the cluster values of all the points remains the same before and after an iteration of assign-clusters (). For parallelisation, the loop is parallelized where for each data point, the distances are computed from all centroids and then the index of the distance from the centroid number (from 0 to k-1) is allotted to the Point. Cluster value for each point, here this loop is executed by multiple threads where each thread updates the cluster values for each Data point. As far as Load Balancing is concerned, in the implementation of Pthreads(where P is the number of threads). each thread gets a total of N divided by P data points for centroid updation(where N is the total number of data points) and hence the Load for each thread is balanced

RESULTS

We have implemented K means clustering algorithm both sequentially and parallelly, to show parallel execution we have implemented the algorithm in both OpenMp and Pthreads that is executing in parallel using threads. We compare the three techniques and also visualize the results for a comparative study of the speedup and efficiency achieved in all the three implementations.

Sequential execution

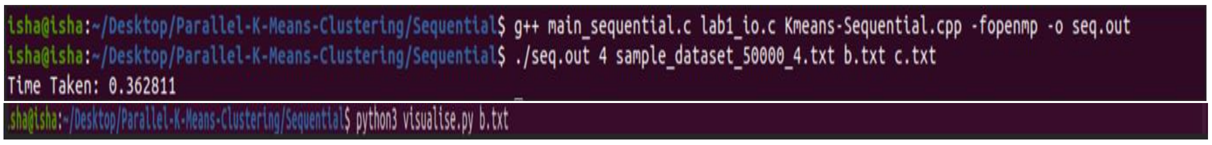
cd into Sequential folder

Compile: g++ main_sequential.c lab1_io.c Kmeans-Sequential.cpp -fopenmp -o seq.out

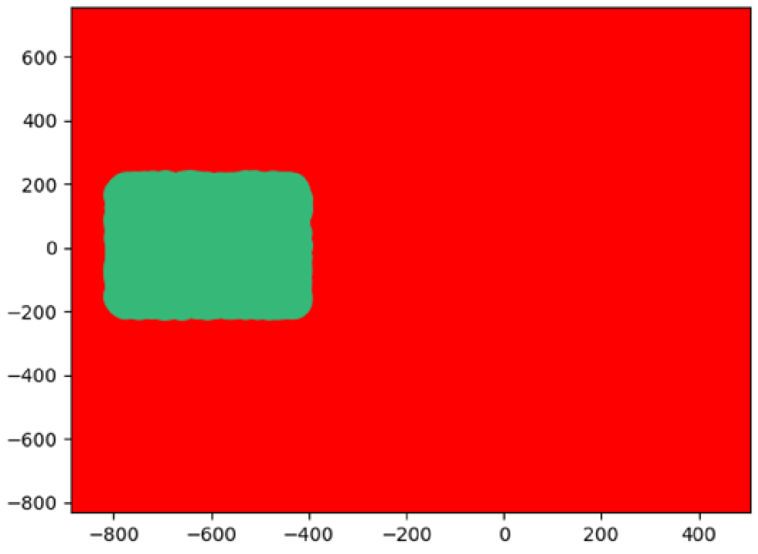
Run: ./seq.out 4 sample_dataset_50000_4.txt b.txt c.txt

4 is for the number of clusters. Can be changed.

To Visualize the results: - python visualise.py b.txt



```
isha@isha:~/Desktop/Parallel-K-Means-Clustering/Sequential$ g++ main_sequential.c lab1_io.c Kmeans-Sequential.cpp -fopenmp -o seq.out
isha@isha:~/Desktop/Parallel-K-Means-Clustering/Sequential$ ./seq.out 4 sample_dataset_50000_4.txt b.txt c.txt
Time Taken: 0.362811
isha@isha:~/Desktop/Parallel-K-Means-Clustering/Sequential$ python3 visualise.py b.txt
```



OpenMP execution

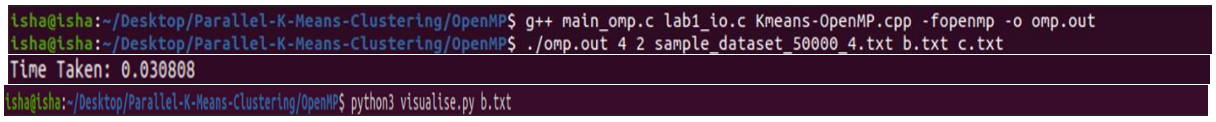
cd into OpenMP folder

Compile: g++ main_omp.c lab1_io.c Kmeans-OpenMP.cpp -fopenmp -o omp.out

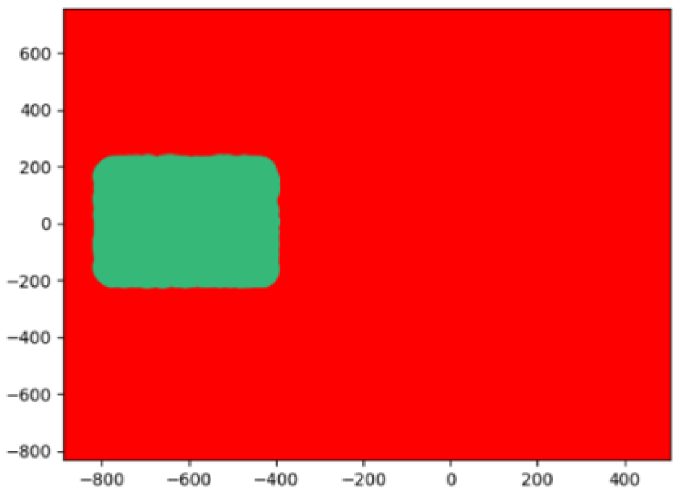
Run: ./omp.out 4 2 sample_dataset_5000_3.txt b.txt c.txt

4 is for the number of clusters and 2 is for the number of threads. Both can be changed.

To Visualize the results: - python visualise.py b.txt



```
isha@isha:~/Desktop/Parallel-K-Means-Clustering/OpenMP$ g++ main_omp.c lab1_io.c Kmeans-OpenMP.cpp -fopenmp -o omp.out
isha@isha:~/Desktop/Parallel-K-Means-Clustering/OpenMP$ ./omp.out 4 2 sample_dataset_50000_4.txt b.txt c.txt
Time Taken: 0.030808
isha@isha:~/Desktop/Parallel-K-Means-Clustering/OpenMP$ python3 visualise.py b.txt
```



P-Threads execution

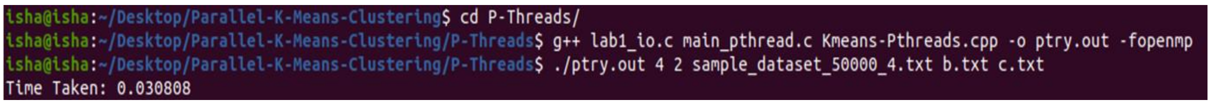
cd into P-Threads folder

Compile: g++ lab1_io.c main_pthread.c Kmeans-Pthreads.cpp -o ptry.out -fopenmp

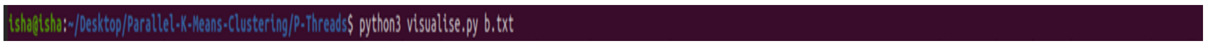
Run: ./ptry.out 4 2 sample_dataset_50000_4.txt b.txt c.txt

4 is for the number of clusters and 2 is for the number of threads. Both can be changed.

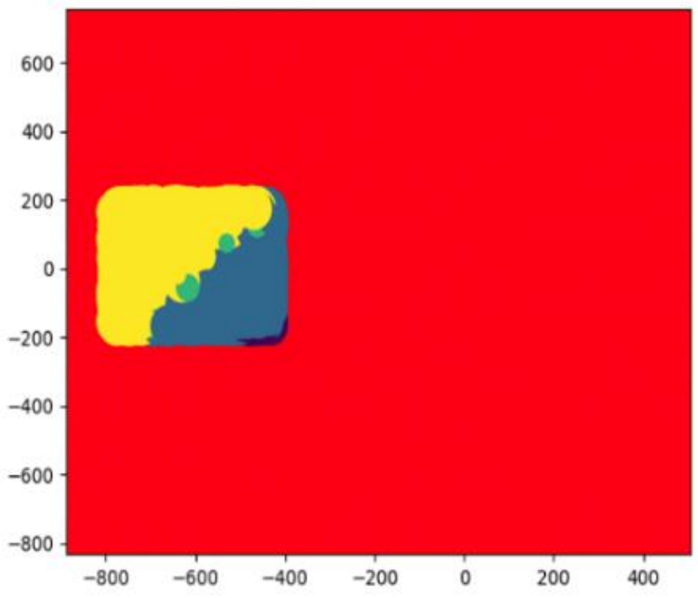
To Visualize the results: - python visualise.py b.txt



```
isha@isha:~/Desktop/Parallel-K-Means-Clustering$ cd P-Threads/
isha@isha:~/Desktop/Parallel-K-Means-Clustering/P-Threads$ g++ lab1_io.c main_pthread.c Kmeans-Pthreads.cpp -o ptry.out -fopenmp
isha@isha:~/Desktop/Parallel-K-Means-Clustering/P-Threads$ ./ptry.out 4 2 sample_dataset_50000_4.txt b.txt c.txt
Time Taken: 0.030808
```



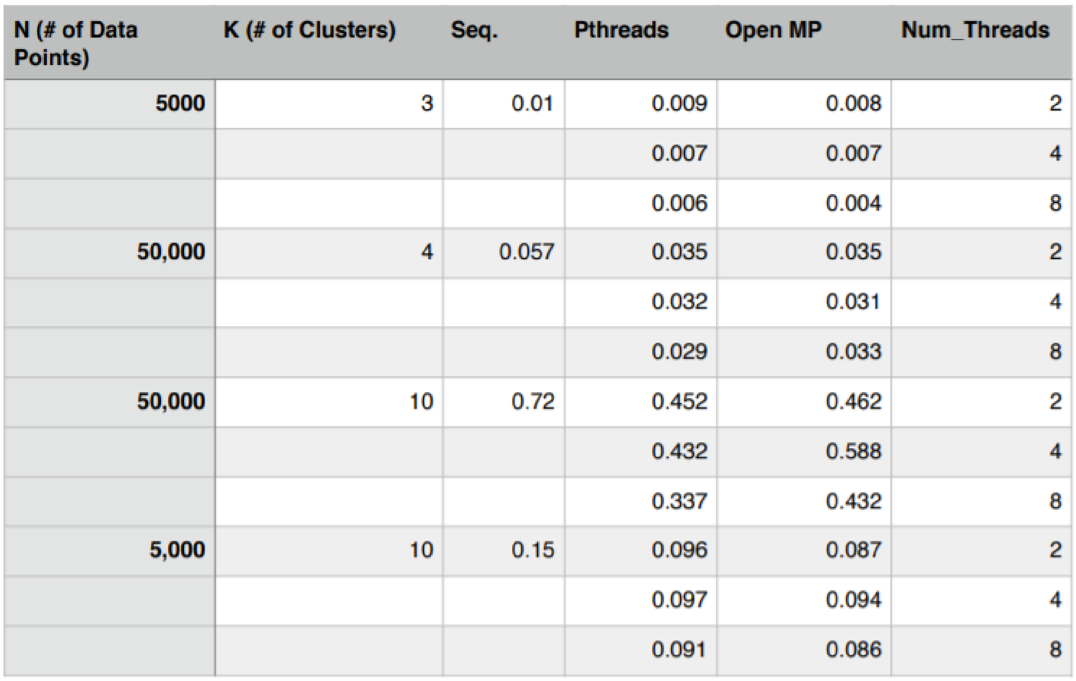
```
isha@isha:~/Desktop/Parallel-K-Means-Clustering/P-Threads$ python3 visualise.py b.txt
```



Analysis of the results:

We will now analyse the results with the help of the table shown here. This table shows the time taken by each of the technique given the number of data points, no of clusters and the number of threads.

In the first set, we have taken 5000 data points and 3 clusters, we see the sequential execution takes 0.01 seconds. For parallel execution, we execute with 2,4 and 8 threads and we can see that for 2 threads the time taken by pthreads is 0.009 seconds n time taken by OpenMp is 0.008 seconds.



N (# of Data Points)	K (# of Clusters)	Seq.	Pthreads	Open MP	Num_Threads
5000	3	0.01	0.009	0.008	2
			0.007	0.007	4
			0.006	0.004	8
50,000	4	0.057	0.035	0.035	2
			0.032	0.031	4
			0.029	0.033	8
50,000	10	0.72	0.452	0.462	2
			0.432	0.588	4
			0.337	0.432	8
5,000	10	0.15	0.096	0.087	2
			0.097	0.094	4
			0.091	0.086	8

Similarly, we perform the same on different values as shown here and to summarize we can notice the following:

- The time taken by the parallel execution techniques is less when compared to sequential execution.
- The more the number of clusters, higher is the time taken for execution for a given N, that is the number of data points.
- As the number of threads increases, the time taken for execution is less.
- Comparing the two parallel execution techniques, OpenMp and Pthreads, we can also notice that OpenMp performs better than the pthreads for smaller dataset that is when number of data points are less, but as we increase the number of data points for example, 50000 here, we can see pthread method requires less time when compared to OpenMp.

Formula for speedup and efficiency:

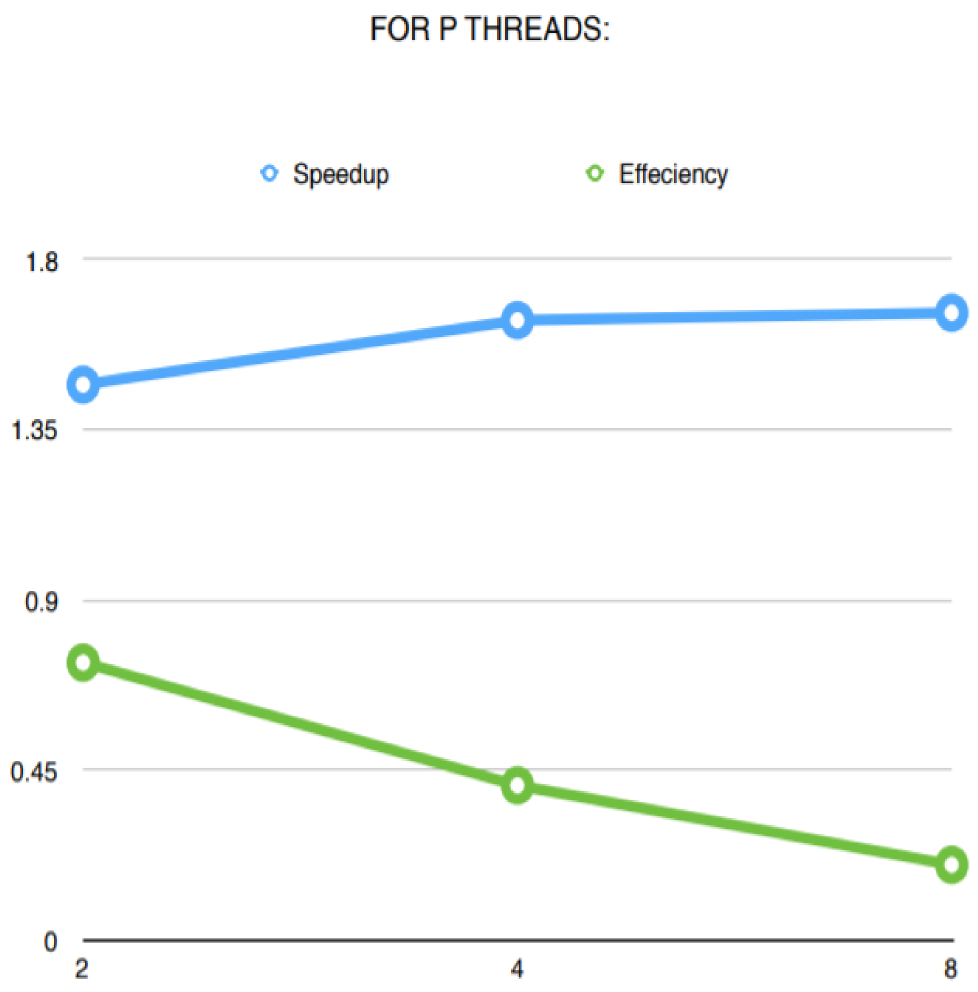
*Speed-up (S) = T (seq) / T (parallel)
*Efficiency (E) = S/ # of threads(P)

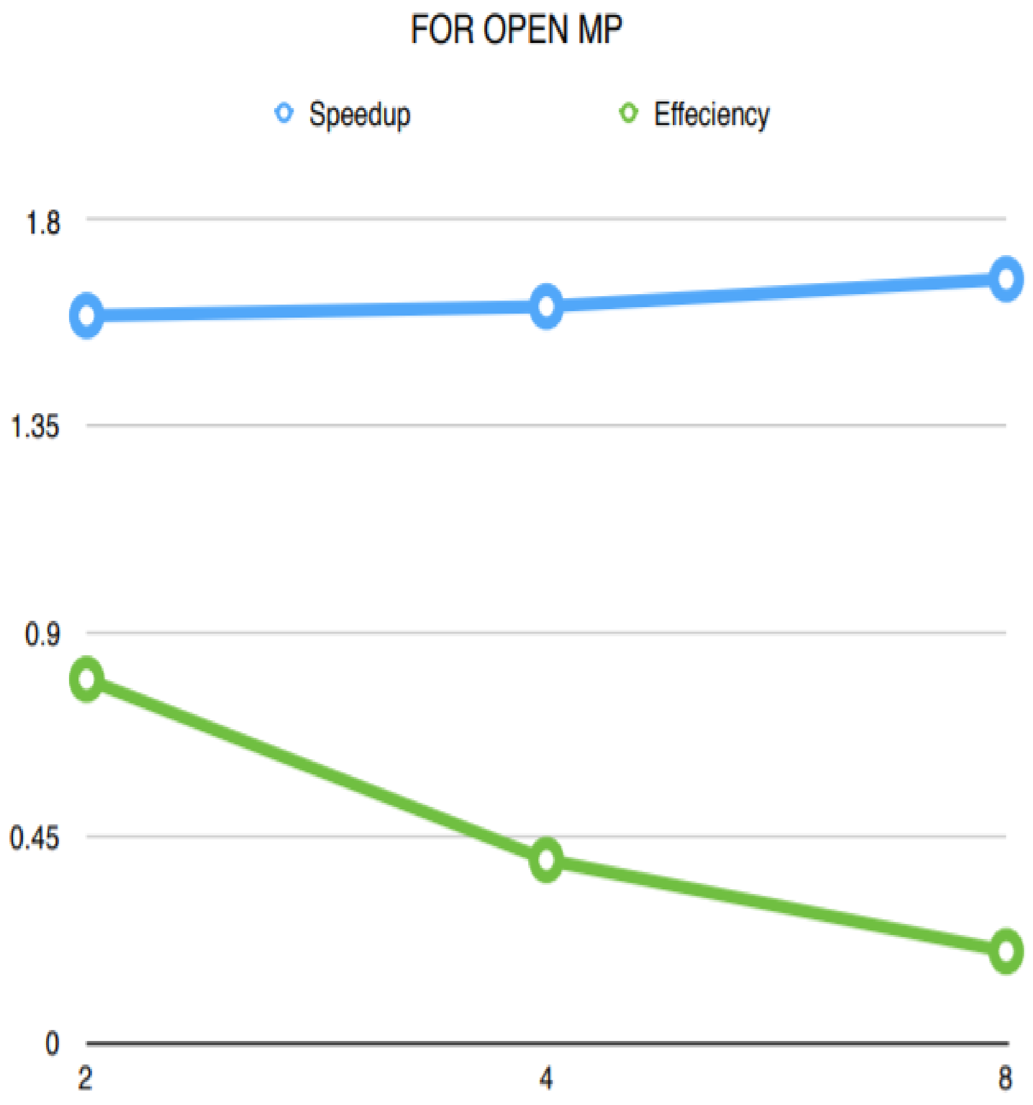
For Problem Size of 50,000 -

Speedup -> 1) open mp : 1.59(p = 2) , 1.61(p = 4), 1.67(p = 8)
2) pthreads : 1.47(p =2) , 1.64(p = 4), 1.66(p = 8)
Effeciency -> 1) open mp : 0.795(p = 2) , 0.40(p = 4), 0.20(p = 8)
2) pthreads : 0.735(p =2) , 0.41(p = 4), 0.20(p = 8)

Graphs for speedup and efficiency:

We see that the speedup increases with the number of threads for both pthreads and openmp techniques whereas the efficiency decreases with the increase in the number of threads.





FOR OPEN MP
Speedup
Effeciency
1.8
1.35
0.9
0.45
0
2
4
8

References

- ✓ https://en.wikipedia.org/wiki/K-means_clustering
- ✓ https://www.javatpoint.com/k-means-clustering-algorithm-in-machine-learning
- ✓ https://parcomp.wordpress.com/2017/04/16/k-means-in-parallel/
- ✓ https://www.alglib.net/dataanalysis/k-means-clustering.php
- ✓ https://www.worldscientific.com/doi/abs/10.1142/S0218126618501992#:~:text=The%20K-means%20clustering%20solution%20is%20fine-tuned%20by%20applying,enhance%20the%20solution%20using%20processing%20power%20of%20GPU