COL380 Lab-1 Report

Animesh Singh (2016CS10406)

Problem Statement:

To implement K-means clustering algorithm in sequential form as well as in parallel form with the help of PThreads library and OpenMP library.

Parameters:

1. **N** -> Total number of points in the dataset. Each point has 3 dimensions of type int.
2. **K** -> Total number of clusters to be made
3. **T** -> Total number of threads to be run

Design Decisions:

- I've used a struct Point which stores its ID, cluster assigned to it and its coordinates. Similarly, chosen a structure Cluster which has its ID, coordinates of centroid, a vector of its constituent points and methods to add & remove points
- I've chosen to keep arrays of the above structures which I use primarily for the abstraction and computation in my code. Appropriate memory allocation using malloc for data structures
- According to Forgy method of initialisation, randomly chosen K out the N points and chosen as cluster centres
- Subsequently assigned each point to its nearest cluster by calculating distance from the cluster centroids using the distance formula // This function is parallelized
- Re-computation of new centroids of the clusters as the mean of the points in it
- Reiterate, assigning points according to the new cluster centres and continuing the process till no further changes are observed
- Stopping criteria is either when there are no points changing their clusters or the number of iterations of re-assignment have reached a maximum value
- Efficiency of K-Means algorithm depends on the initial cluster centres and the algorithm might get stuck at a local optimum

Parallelization Strategy:

- For parallelization in case of PThreads and OpenMP, I have parallelised the part of the code which deals with assigning nearest cluster to each point. We don't parallelise the entire code as that would require a lot of access to critical sections (requiring locks) which wouldn't yield a good speedup
- Each thread re-assigns the points to clusters & does the required calculation for N/T points. It finds the nearest centre and allocates the point to that cluster

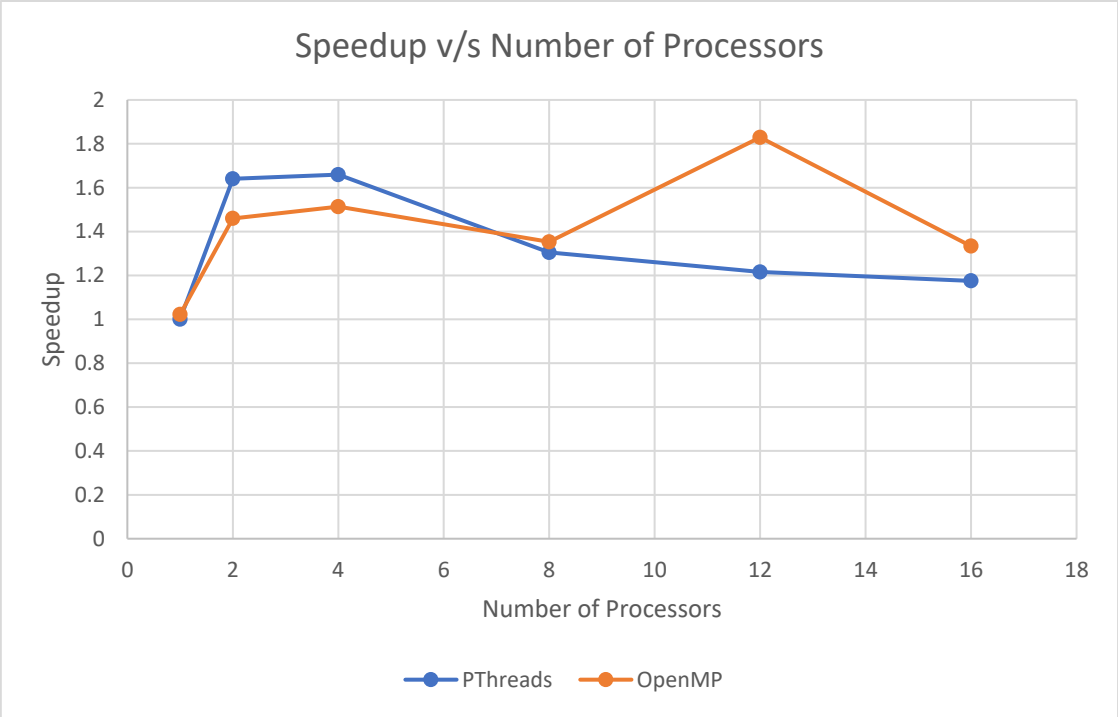
PThread	OpenMP
• For the argument passing mechanism, I've created a separate struct for passing as a void* argument to the function. A new instance of the struct is created for every thread execution (creates an overhead in the execution time) • All of the created threads are joined after the execution of the for loop is complete (have to wait for all threads to complete) • I needed to lock the pthreads using mutex_lock for the critical section of adding & deleting points from clusters (adds overhead to each thread) • PThread gave me, the programmer, a low level experience of the code by specifying the target function to be parallelised. More freedom is given, but also adds responsibility of memory management etc. for the programmer. Greater scope for error.	• No new struct is required, and the splitting of the array can be done without needing for a separate function • There was a need for securing the critical section by the #pragma omp critical directive (adding overhead) • OpenMP allows the programmers to define the code block which needs to be parallelised and involves a high level understanding of the code and parallelisation. I used it on the same loop for which a different function was made for PThreads computation

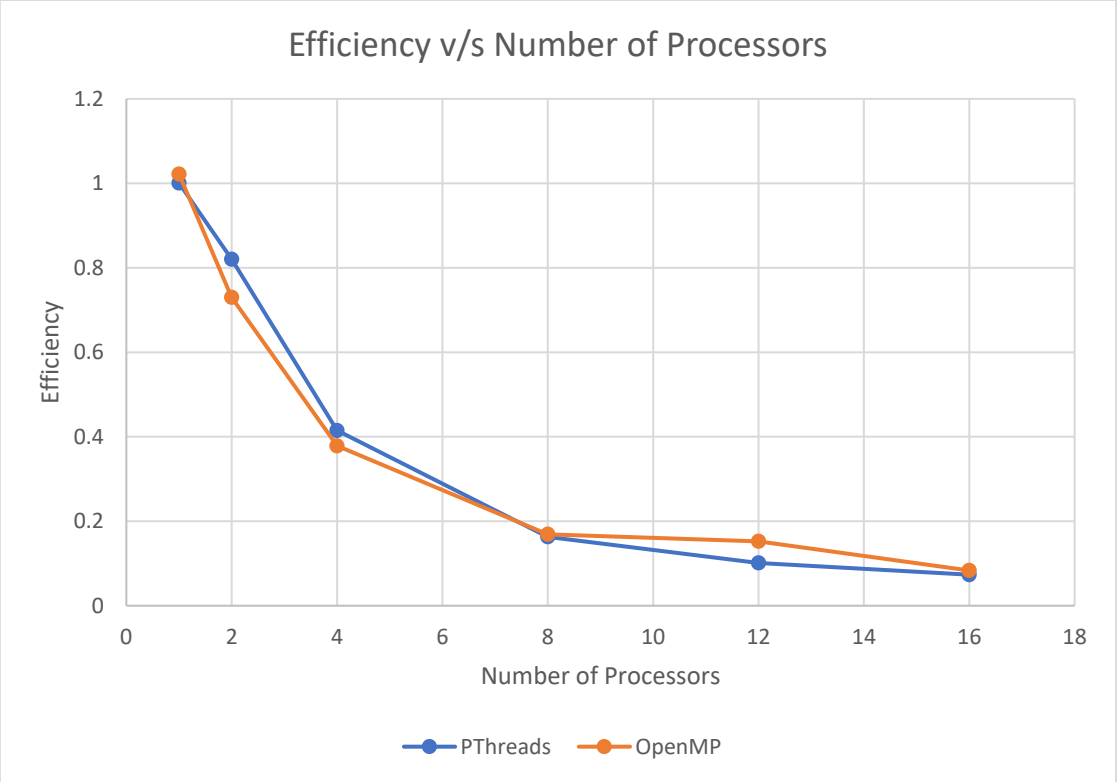
Observations & Analysis:

Performed and plotted speed-up and efficiency analysis of parallel implementations by changing the (i) input size and (ii) number of threads

- For a fixed problem size, the time taken by the sequential code is same in all the cases. A decrease in time taken is observed by OpenMP and PThreads implementations on increasing the number of threads as should have happened because of parallelization. But further increasing the number of threads leads to more running time because of increasing overheads as explained above.
- With increasing problem size, we see that generally the time taken also increases for all the implementations. We see an anomaly for the case of 1000 data points, this might have happened because the initialisation points might not be good, so the time taken for convergence was high.
- OpenMP takes a little less time than compared to the PThreads implementation of the same logic in both variations because OpenMP is better wrapped with better optimisation flags.
- Total complexity of the algorithm → O(iteration*N*K) for both the parallel implementation as well as the sequential implementation. The only difference is in the execution of re-assignment of points to clusters function which is → O(N*K) for sequential and → O(N*K/T) for parallel implementations

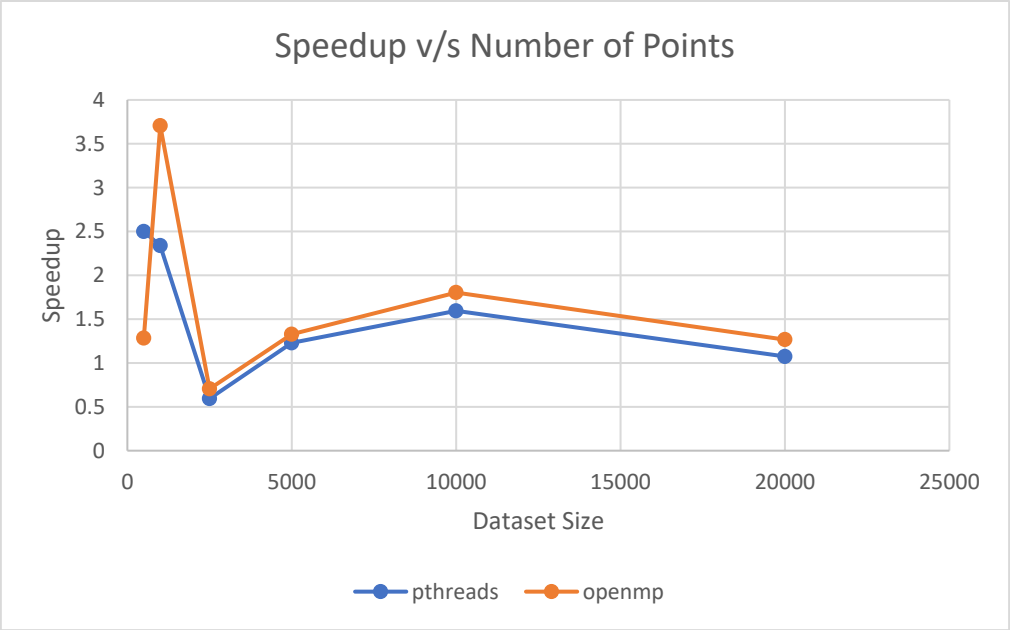
1. Varying Number of Processors (T) (constt K=4; N=5000)

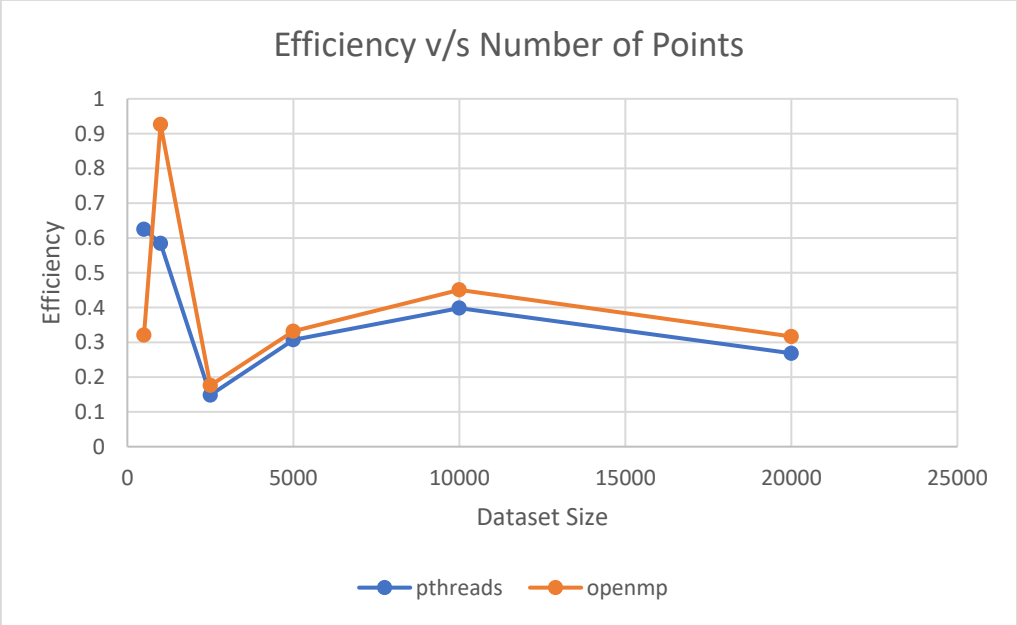




- The above two graphs show the dependence of speedup and efficiency of the parallel programs on the number of processors. On increasing the number of processors, speedup should increase with is visible in the first graph. With further increase in the number of processors, the overhead of maintaining threads leads to increase in time, hence the speedup decreases (pthreads curve nicely depicts this; openmp curve is a little skewed may be because of bad init).
- For efficiency vs number of processors, we know that on increasing the number of processors the efficiency of system decreases which is also the trend which is shown by both the graphs alike.

2. Varying Number of Points (N) (constt K=4; T=4)





- Most evident observation is that both these graphs of speedup and efficiency are exactly same. It is because efficiency equals the ratio of speedup and number of processors and the number of processors is kept constant. Hence, efficiency is same as speedup, just divided by a constt value
- Efficiency of a system must increase with increase in problem size with number of processors kept constant, as shown by the graph (20K point is an anomaly; maybe because of bad init).
- Ahmdal's Law is verified by the above curve as the speedup remains same in both the parallel implementations for same T (number of processors) and same f (fraction of parallelization)

Conclusion:

- Experimented & analysed speedups and efficiencies for various problem sizes and different number of processors with sequential and parallel implementation of K-Means algorithm.
- Parallel implementation was achieved by using the PThreads and OpenMP libraries.
- Experimental results were in coherence with the theoretical results studied in class.