Lab Assignment 1 (COL380)

Ankit Solanki
2016CS50401

February 10, 2019

1 K-Means Clustering

Implementing a sequential and parallel (using Pthreads and OpenMP) K-means clustering algorithm.

1.1 Algorithm

- Initialize K centers of clusters from the given set of data-points.
- For every point calculate which cluster it is nearer to, and assign it that cluster.
- Calculate mean point (new center) of every cluster.
- Do step-2, until stopping criteria seems fulfilled.

Stopping Criteria:
- If maximum iterations (a predefined constant) are over.
- Or if cluster centers are changing by a very little amount that it is statistically insignificant to difference them (must have been converged.)

1.2 Parallelization Strategy

- Initialize K centers of clusters.
- Divide data-points into equi-parition of number of threads: Assign each thread its own set of points
- Each thread would calculate which centroid its data-points are closest to, and assign them.
- Each thread would calculate new mean of its data-points, and write it into globally allocated non-shared space.
- Calculate new centroids reading from results are threads (threads are destroyed here.)

1.2.1 Using Pthreads and OpenMP

- Both uses different syntax but similar logic (specified in documentation with attached code).

- Both uses similar algorithm (since no shared variables are used; Locking doesn't come in context.)

1.3 Design Choices

- Initialize K centers of clusters as starting K points of data-points, so that it doesn't vary when comparing speedup or efficiency of parallel and sequential algorithms. Had I not done this, number of iterations would vary in the algorithms, making it hard to compare.
- Alternatively, could have randomly initialize with a common seed variable.
- No two threads are writing in the shared memory address thus no chances of data race condition. And hence easier implementation (without lock.)
- Threads are used to manipulate their own set of points. And after their destruction, global manipulation has been done to work out on their results.

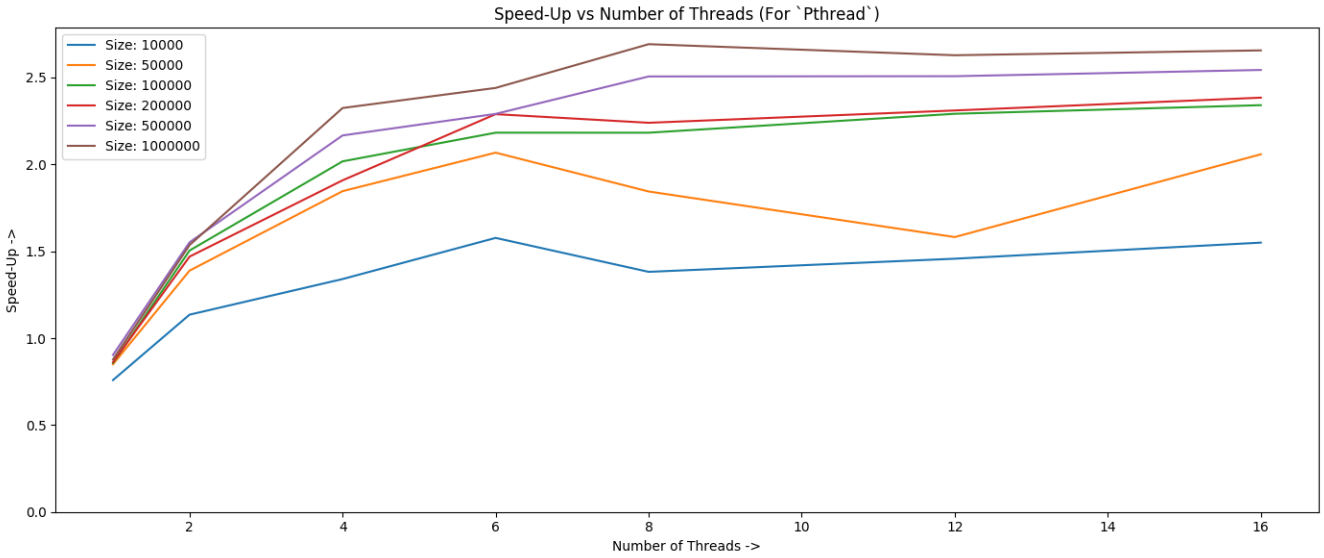
1.3.1 Load-Balancing

- Every threads has been given equal (almost equal for last thread) amount of data-points. Thus, the load is balanced and no thread does significantly more amount of work.

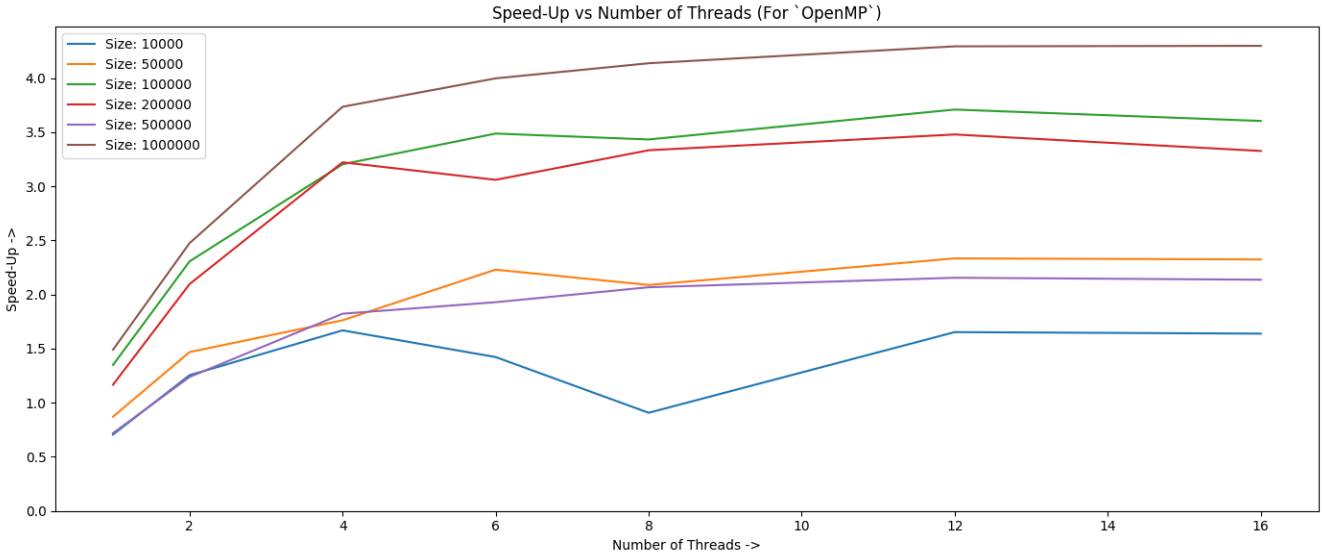
1.4 Graphs

Threads and data points are discretized as [1, 2, 4, 6, 8, 12, 16] and [10k, 50k, 100k, 200k, 500k, 1m] respectively.

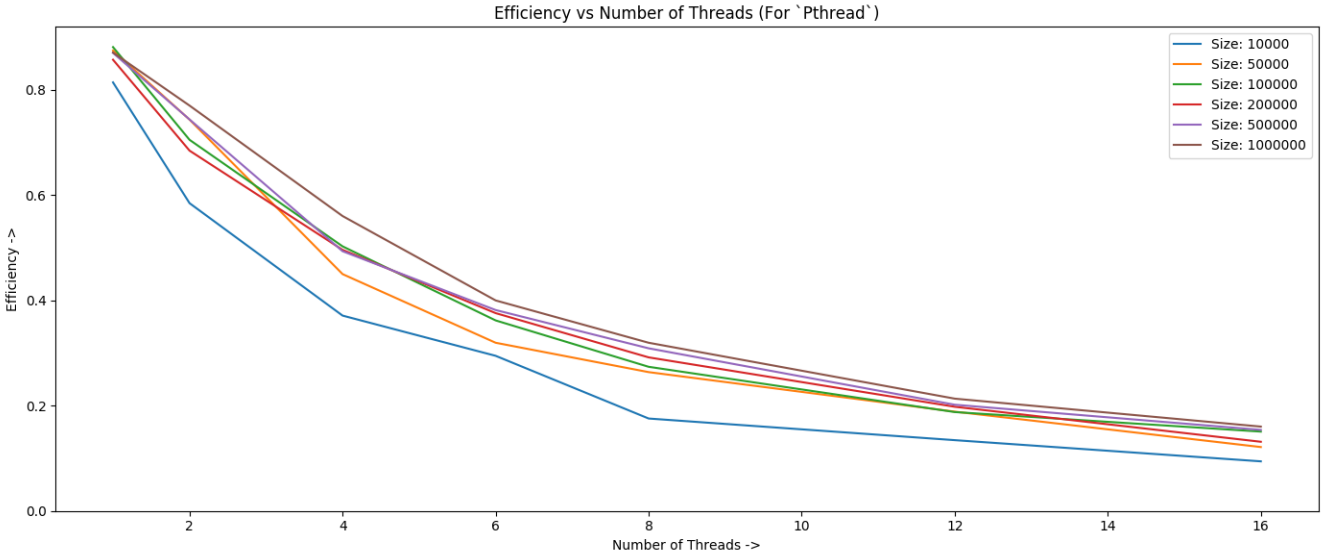
1.4.1 Speed-Up Vs Number of Threads (Pthreads)



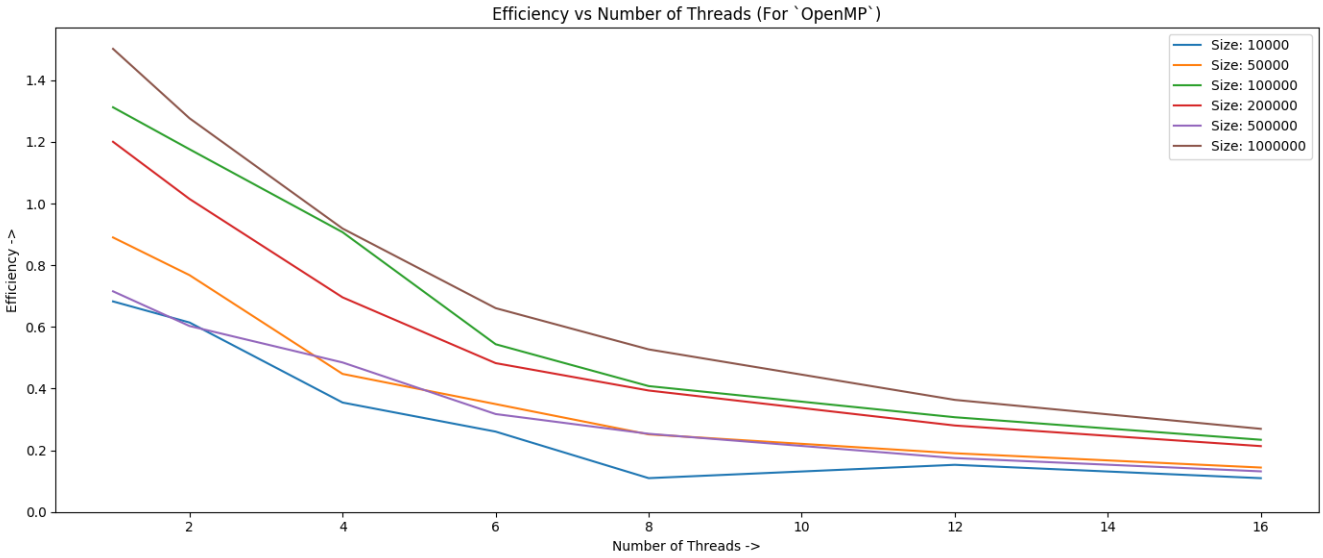
1.4.2 Speed-Up Vs Number of Threads (OpenMP)



1.4.3 Efficiency Vs Number of Threads (Pthreads)



1.4.4 Efficiency Vs Number of Threads (OpenMP)



1.5 Graph Analysis

- We observe almost **same speedup and efficiency graphs with OpenMP and Pthreads**, which we should have, given that both are libraries that helps in managing threads for any given program.
- Speedup increases till p=8 for most cases and then decreases afterward. It happens because of the **communication overheads** introduced when number of threads increase.
- My system has 8 cores, so after that speedup slows down and doesn't change much.
- Sometimes for some data-set, speedup may vary unexpectedly for different runs; As the actual time given by processor to this process may be different due to CPU scheduling and other possible events.
- **At same number of threads, efficiency for bigger problem size is more**. Which was expected as: There is higher fraction of parallel part for higher problem size.
- **Finally**: We perform data parallelism by dividing the data points into different parts for p threads. And the results show that soomthly.