Lab 1 : K-Means Clustering

Samarth Aggarwal - 2016CS10395

February 10, 2019

1 Objective

K-means clustering is an algorithm to find k partitions of an unlabelled data set. For a set of observations $(x_1, ..., x_n)$, where each observation is a d-dimensional real vector, k-means clustering aims to group the set of observations into k sets $S_1, ..., S_k$ such that:

$$argmin \sum_{i=1}^{k} \frac{1}{2|S_i|} \sum_{x,y \in S_i} ||(x-y)^2||$$

2 Algorithm

1. Initialisation :
   - Random Initialisation - Randomly initialise the k clusters to be any value within the specified bounds.
   - Forgy Initialisation - Randomly choose any k points from the input data to be initial positions of the cluster means.
2. Assign Points : For every point in the data, compute its distance from every cluster centre. Now, assign the nearest cluster to every point.
3. Recompute Cluster Means : Update the coordinates of centres of every cluster to the mean of coordinates of the points lying in that cluster.
4. Repeat steps 2 and 3 until a pre-specified maximum number of iterations are complete or the update in cluster means falls below a certain threshold.

3 Parallelisation Strategy

1. The crux of my parallelisation strategy was to identify cases of 'Data Parallelisation' and then distribute the workload effectively.
2. Random initialisation can be parallelised easily wherein different threads will be assigned the task to initialise different cluster means randomly.

3. Forgy initialisation can also be parallelised but we will have to maintain which point has already been assigned so that two different clusters are not initialised to the same value.

4. The step for assigning points to cluster can be parallelised by assigning different points to different threads. Every thread will compute the distance of a point from all clusters and assign the cluster closest to that point. Once done, this thread will be assigned the next point and so on.

5. The step for recomputing cluster centres can be parallelised by assigning each threads the task of computing mean coordinates of a different cluster.

6. This can be done by having each thread iterate over all points and adding the coordinates of only those points that belong to the given thread.

7. Another approach could be to maintain a data structure for every cluster that contains all the points that belonging to that cluster. Now each thread only needs to iterate over all points in this data structure.

4 Design Decisions

1. Experimentally, I found that random initialisation is much more prone to getting stuck in local optima than forgy initialisation. For this reason, forgy initialisation was chosen.

2. Maintenance required to ensure that different cluster means are initialised to different values will involve the use of locks. Since the number of cluster is small (between 2 to 10) so using threads will do more harm than good. Hence, I decided not to parallelise this step.

3. The parallelisation of assign points step does not require any lock hence it is extremely efficient. So this was implemented.

4. The parallelisation of recompute cluster mean by the first approach will involve every thread to iterate over all points which is very inefficient.

5. The second approach will have overheads of maintaining the proposed data structure. Moreover, it will also involve use of locks since many threads could be simultaneously updating it in the assign points step.

6. Since the use of lock inside for loop is even worse that sequential implementation, hence recompute cluster mean step was not parallelised.

5 Load Balancing Strategy

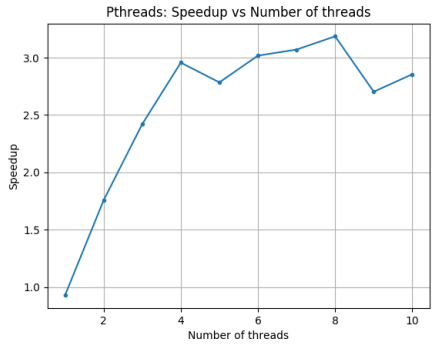
There are 2 ways to distribute n iterations on p threads.

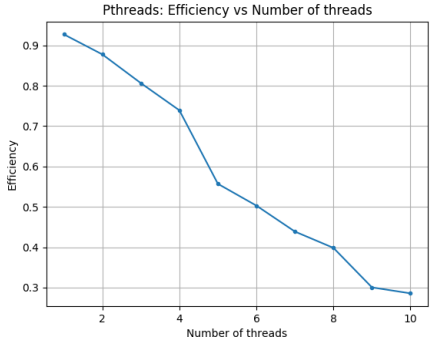
1. assign $\lfloor n/p \rfloor$ points to each thread and the leftover points to the last thread.

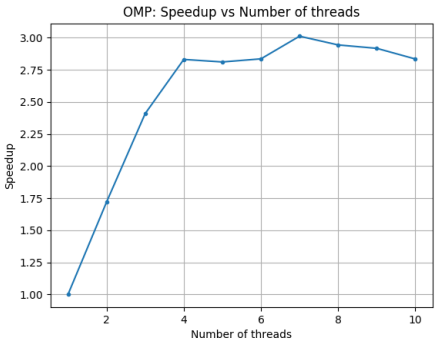
2. assign $\lfloor n/p \rfloor$ points to each thread initially and then assign the remaining n-($\lfloor n/p \rfloor$*p) points, one to each thread.

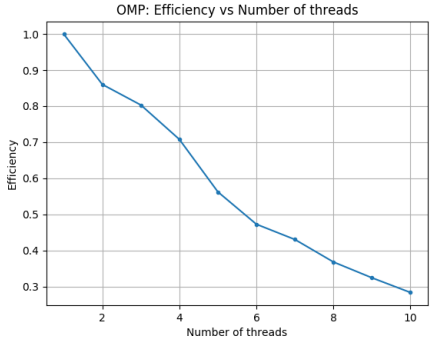
After assigning $\lfloor n/p \rfloor$ points, the number of points left will be between 0 to p-1. In case 1, all of them are assigned to a single thread hence difference in the workload is O(p). In case 2, these points are distributed over threads so difference in workload is O(1). Hence, the second strategy was chosen.

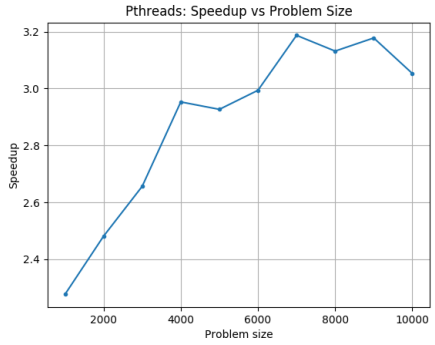
6 Observations

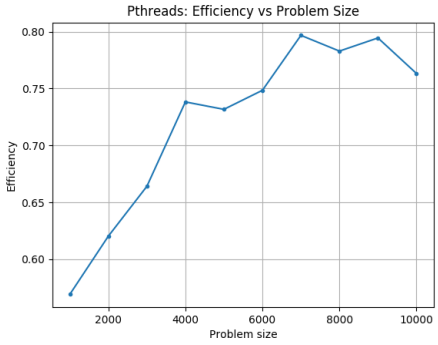


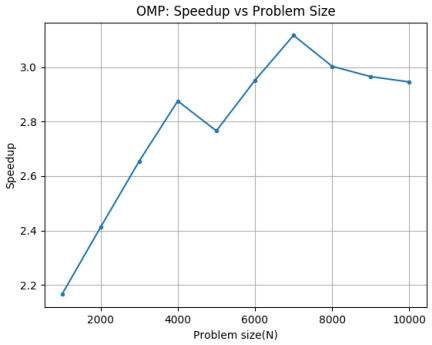


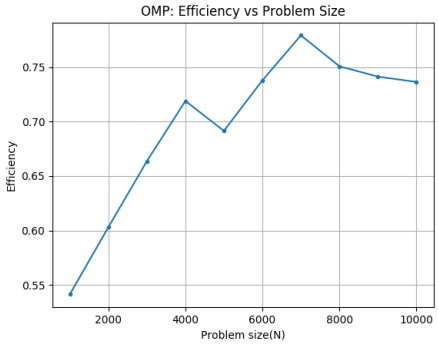












7 Calculations

By Amdahl's Law,

$$S = \frac{1}{f + \frac{1-f}{p}}$$

$$f = \frac{\frac{1}{S} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Using the above formula, at problem size of 70,000 points and 10 clusters,
Average Fraction of serial code in PThread implementation = 21.12 %
Average Fraction of serial code in OpenMP implementation = 22.31 %

8 Conclusions

1. Speedup increases with number of threads. It attains a maxima nearly around the number of cores and then begins to decrease.
2. Speedup increases with problem size. When the number of threads becomes very high, the speedup gradually saturates.
3. Efficiency varies inversely with the number of threads.
4. Efficiency increases with problem size and then saturates.
5. Within limits of experimental error, the fraction of serial code remains the same upon parallelisation using PThreads and OpenMP.