Parallel K-Means

Chetan Mittal (2016CS10343)

February 2019

1 Introduction

Performance of sequential implementation and parallelized implementations (using pthread and OpenMP) of K-Means clustering algorithm are compared in this report. The report also mentions the design decisions taken in the implementation.

2 Sequential K-Means

2.1 Initialisation

1. Forgy method has been used to initialise k centroids. K indices are uniformly chosen from 1 to N and centroids are initialised to data points corresponding to these indices.
2. A seed of **1234** has been set to compare performance across multiple runs.

2.2 Forming Clusters

1. Each point is assigned to the nearest centroid to form a cluster. A single loop traverses over the N data points and assigns a cluster to each point.
2. An array **points_in_cluster** maintains the number of points in each cluster, is updated in each iteration of the loop (Loop1).

2.3 Computing new means

1. The sum of points in each cluster is maintained in the **centroids array**.
2. A loop traverses over the points and adds the value to the corresponding cluster (Loop2).
3. Another loop computes the new means by traversing over the K elements of the centroids array and dividing each sum by the corresponding value in points_in_cluster (Loop3).

2.4 Termination

1. Max iterations are limited to **10000**.
2. Execution terminates if there are less than **2 cluster changes**.

3 Parallel K-Means

3.1 Loops and Data

1. There are a total are 3 loops (excluding initialisation) which are repeated till convergence.

2. The for loops update the following variables :
a. clusterChanges : Loop 1
b. data_point_cluster : Loop1
c. points_in_cluster : Loop1
d. centroids : Loop2, Loop3

3.2 Parallelization strategy

1. First loop is parallelized by assigning each thread equal number of points, to compute the cluster for each point.

2. Since all threads update the same K elements of the centroids array in th second loop, parallelizing this loop did not lead to speedups (rather resulted in slowdown) due to overhead of waiting in lock (for mutual exclusion of the K elements) and communication overheads.

3. Even having a separate array of K elements for each thread did not result in any improvements, probably due to the overhead of syncronization at the end and probably because all threads still read the shared data_point_cluster. For these reasons, this loop was not parallelized in the final implementation.

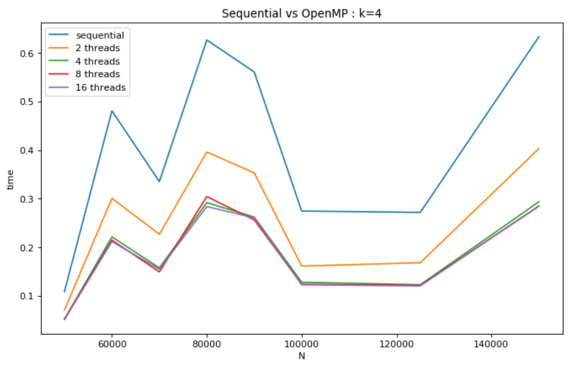
4. Since the third loop runs for K iterations (2-10), parallelizing this loop will lead to insignificant speedup (and possibly slowdown due to communication overheads).

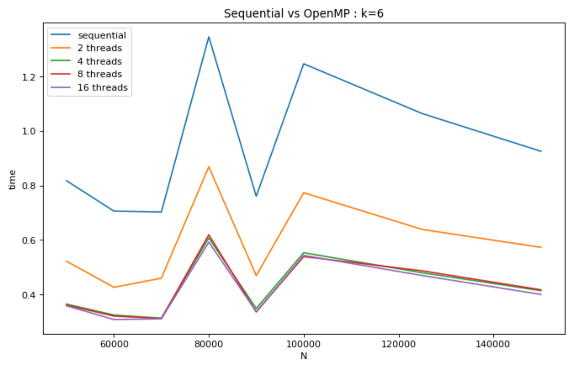
3.3 Loadbalancing

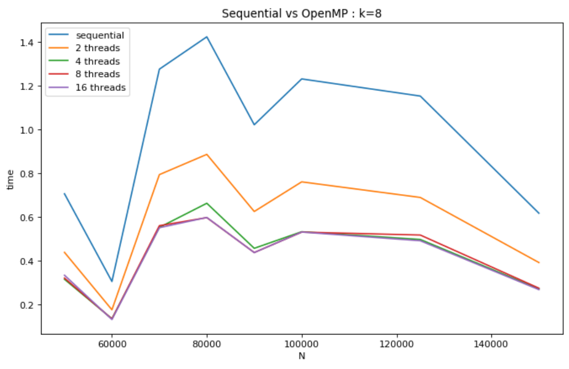
1. Since the first loop is equally divided among threads, the load corresponding to each thread is equal and no separate load balancing is required.

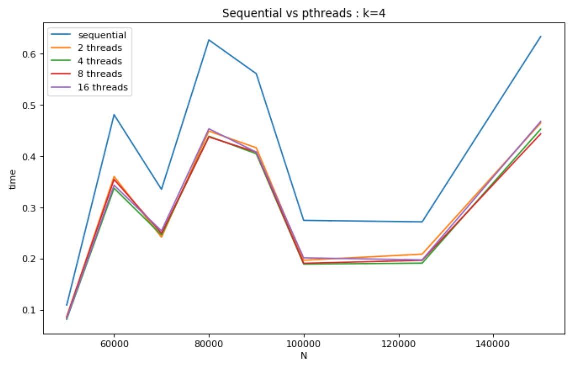
4 Speedups and Efficiency

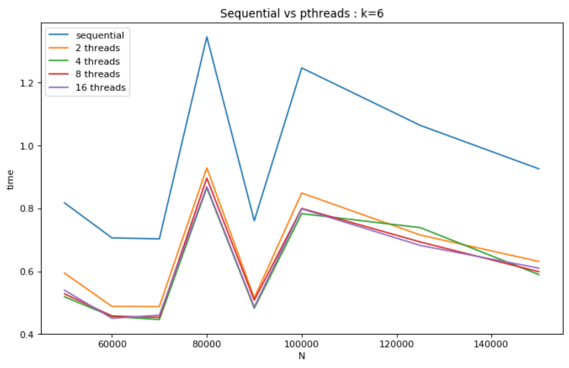
This section compares runtime, speedup and efficiency corresponding to different number of threads, different number of clusters and different input sizes.
Note : Time on changing N is highly random due to random initialisation which may affect the number of iterations it takes to converge. Only thing that should be noted is the relative time between sequential and parallel execution for each N.

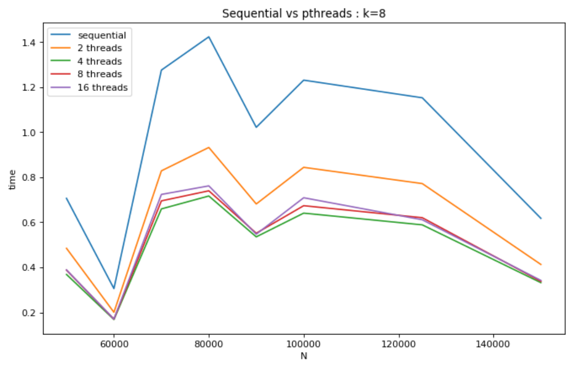

Sequential vs OpenMP : k=4
sequential
2 threads
4 threads
8 threads
16 threads
time
N

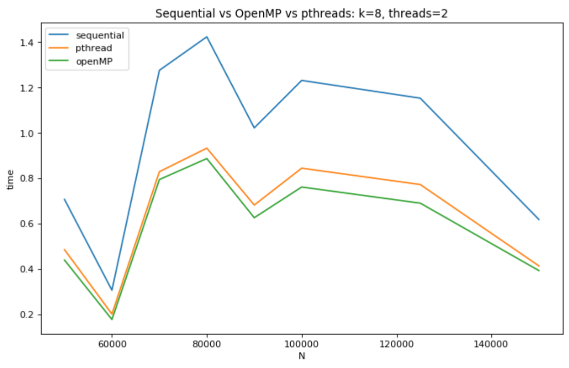

Sequential vs OpenMP : k=6
sequential
2 threads
4 threads
8 threads
16 threads
time
N

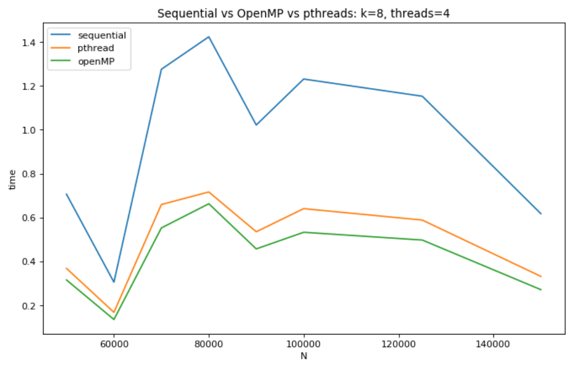

Sequential vs OpenMP : k=8
sequential
2 threads
4 threads
8 threads
16 threads
time
N

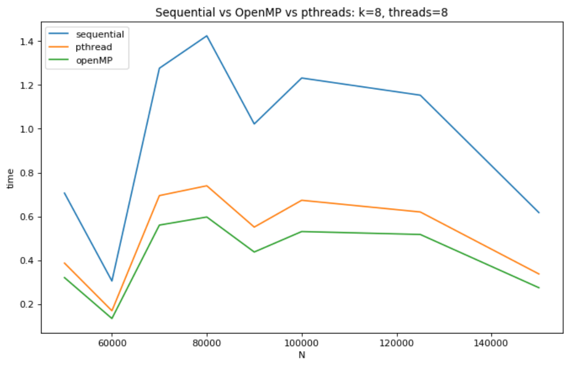

Sequential vs pthreads : k=4
sequential
2 threads
4 threads
8 threads
16 threads
time
N

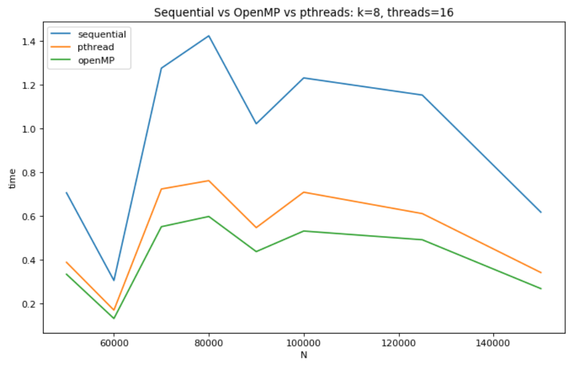

Sequential vs pthreads : k=6
sequential
2 threads
4 threads
8 threads
16 threads
time
N

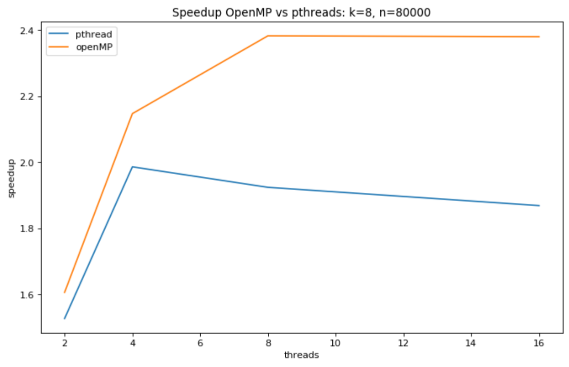

Sequential vs pthreads : k=8
sequential
2 threads
4 threads
8 threads
16 threads
time
N

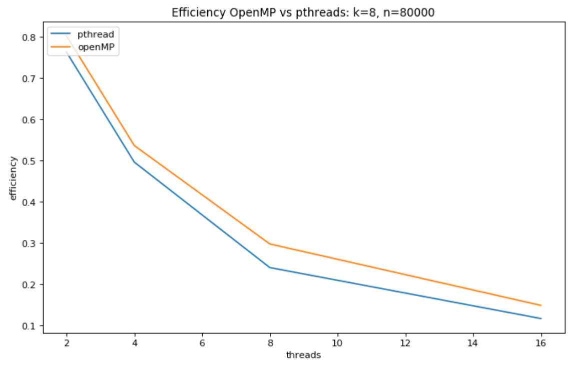

Sequential vs OpenMP vs pthreads: k=8, threads=2
sequential
pthread
openMP
time
N
0.2
0.4
0.6
0.8
1.0
1.2
1.4
60000
80000
100000
120000
140000


Sequential vs OpenMP vs pthreads: k=8, threads=4
sequential
pthread
openMP
time
N
0.2
0.4
0.6
0.8
1.0
1.2
1.4
60000
80000
100000
120000
140000


Sequential vs OpenMP vs pthreads: k=8, threads=8
sequential
pthread
openMP
time
N
0.2
0.4
0.6
0.8
1.0
1.2
1.4
60000
80000
100000
120000
140000


Sequential vs OpenMP vs pthreads: k=8, threads=16
sequential
pthread
openMP
time
N
0.2
0.4
0.6
0.8
1.0
1.2
1.4
60000
80000
100000
120000
140000


Speedup OpenMP vs pthreads: k=8, n=80000
pthread
openMP
speedup
threads
1.6
1.8
2.0
2.2
2.4
2
4
6
8
10
12
14
16


Efficiency OpenMP vs pthreads: k=8, n=80000
pthread
openMP
efficiency
threads
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
2
4
6
8
10
12
14
16