Introduction to Parallel and Distributed Programming **Shreshth Tuli**
COL380 - Lab Assignment 1 2016CS10680
Supervisor - Prof. Subodh Sharma Date: 7/02/2019

Introduction

The K-Means clustering algorithm partitions an unlabelled set of points into k groups. The goal of the algorithm is to minimise $\sum_{i=1}^{k} \frac{1}{S_i} \sum_{x,y \in S_i} ||x - y||^2$. To achieve this an iterative algorithm is used where we first initialize means and then in each iteration update points and means. In this assignment serial and parallel implementations were developed and tested for different problem sizes and number of threads. Four parallelisation strategies were used:

1. pThread based parallelisation where the *cluster update* was parallelised referred as **pThread**
2. pThread based parallelisation where both *cluster update* and *means update* were parallelised, referred as **pThread2**
3. OpenMP based parallelisation using compiler directives, referred as **omp**

We refer to the sequential implementation as **seq**.

1 Implementation

We now present various implementation details, design decisions and optimization strategies used in different implementations.

1.1 Sequential

- Each cluster (1 to k) was given a *clusterID* and goal of the algorithm was to provide the best clusterID's to all points
- The code was divided into two main functions for updating clusterID's of all points and updating the position of the k means.
- The means were initialized using a deterministic algorithm of selecting first k points as means.
- In a loop both update clusterID and update means' position was called successively. The loop terminated after a specific number of iterations to maintain homogeneity in the execution time across different runs of the code and easier comparison.

1.2 pThread

- In the first level of pThread based parallelisation, the updating clusterID's was multi-threaded.
- All the points were partitioned into equal size sets where the number of such sets was equal to the number of threads.
- Each thread evaluates for it's set of points, the closest mean and updates the clusterID's
- To prevent thread creation overhead at each iteration the threads were created only once at the start of the program and a *work_ done[]* shared array of booleans was maintained to propagate messages to and from threads when the clusterIDs need to be updated.
- In each iteration, in the main loop, we set *work_ done[j]* to *false* for all threads *j*. Once this is done, the threads are indicated to update clusterIDs. The functions set their shared variable *work_ done[tid]* corresponding to their thread ID : *tid*.

- To wait for the threads to complete their work, we check continuously in the main loop if all *work_ done[j]* have been set to *true*. If any one of them is *false* we conitnue to wait.
- As their is one writer of *false* (the main loop) and one writer of *true* (the function) for a shared variable *work_ done[j]* the property of mutual exclusion holds.

1.3 pThread2

- Like in the earlier implementation here too the function of updating the clusterIDs has been multi-threaded, but the updating means function too has been multi-threaded.
- To prevent too much locking and synchronization in the update mean function, a similar *work_ done2[]* array has been declared.
- As the finding clusterIDs and updating means need to be done sequentially as they can not be overlapped, the same threads perform both jobs one after the other.
- Due to high overhead of so much synchronization across threads and k ranging between 2 to 10 which means k « n for small size data this strategy performs really poorly compared to the **pThread** implementation.

1.4 OpenMP

- For the OpenMP implementation we have used the compiler directive of *pragma omp parallel for*. This has been used in both updating clusterIDs and updating means.
- As we shall see later, due to optimized implementation this works slightly faster that the **pThread** approach.
- To prevent over writing of values *shared() and private()* have been used to distinguish between shared and local variables.

2 Experiments

To test the impact and efficiency of parallelisation in all the approaches we performed tests/experiments on the completion time of each implementation.

2.1 Test setup

The tests were run on the following setup:

- Intel i7 7700K quad core processor hyper threaded to 8 virtual cores
- Cache line size = 64 Bytes
- Level 1 cache = 4 x 32 KB 8-way set associative instruction caches, 4 x 32 KB 8-way set associative data caches
- Level 2 cache = 4 x 256 KB 4-way set associative caches
- Level 3 cache = 8 MB 16-way set associative shared cache

2.2 Observations and results

The graphs in Figure 1 - 3 show the comparison between times of different parallelized implementations. We can conclude the following:

- We first observe that all execution times grow linearly with problem size = number of data points.
- The execution time also increases as the number of cluster = K increases.

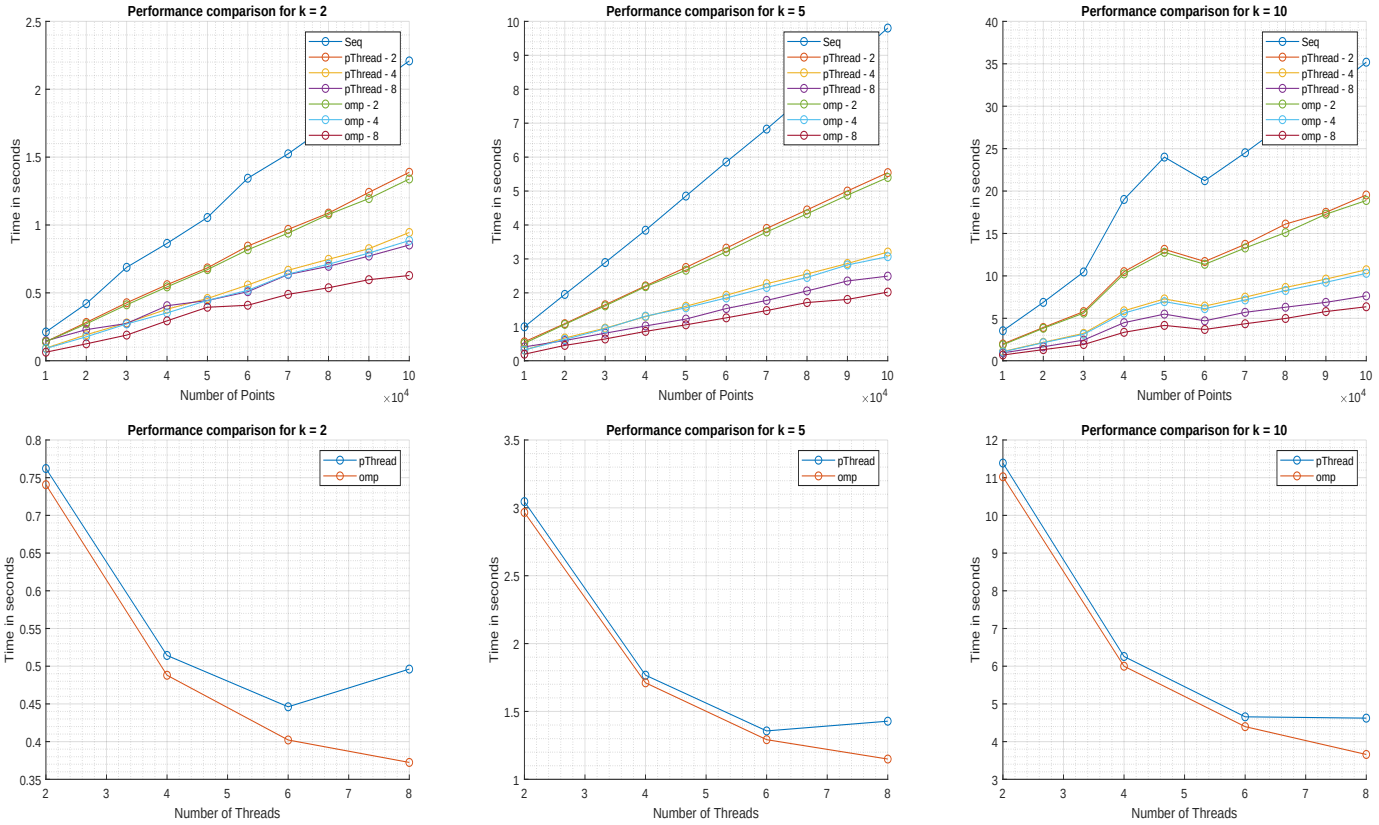

Figure 1: Graphs showing time taken by different implementations for different k

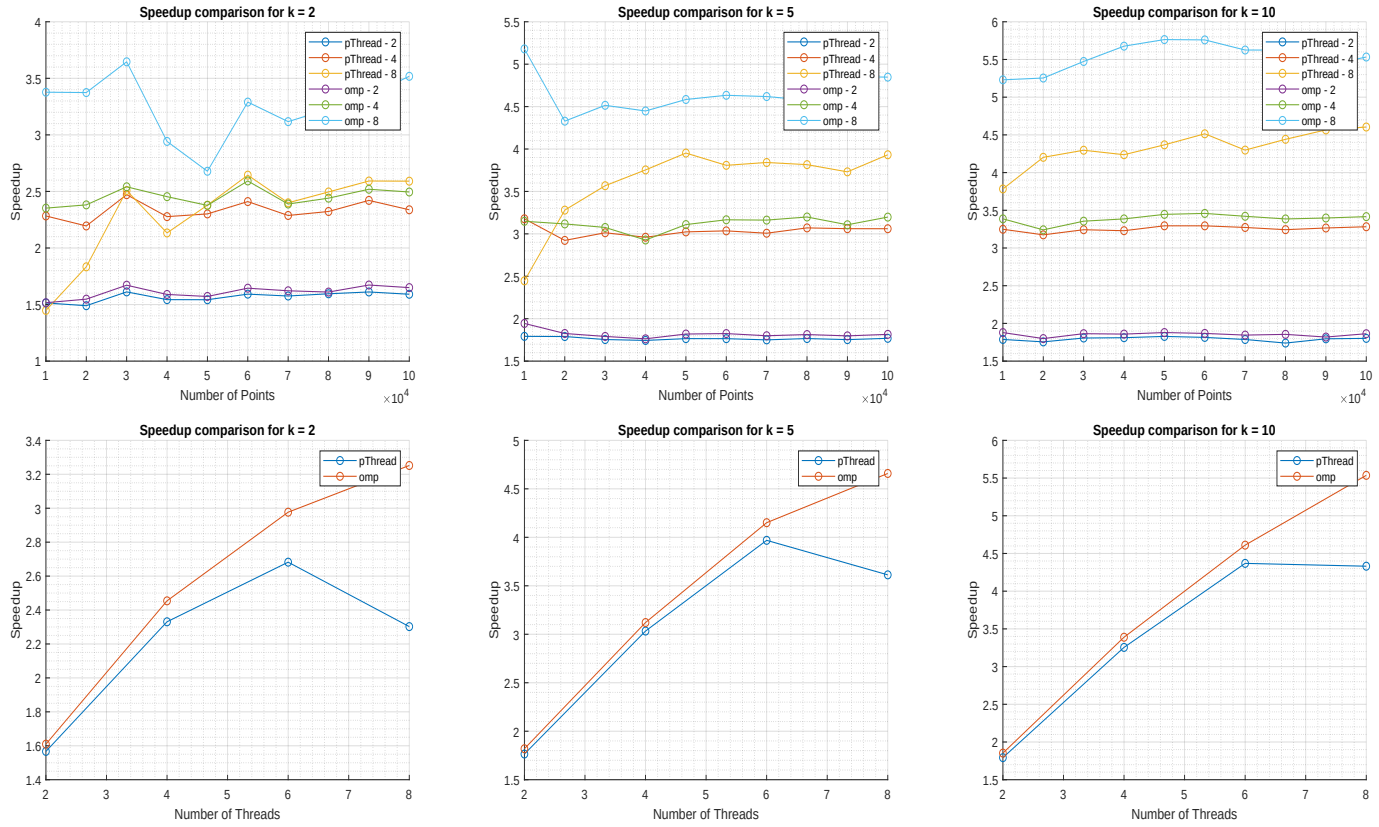

Figure 2: Graphs showing Speedup of different implementations for different k

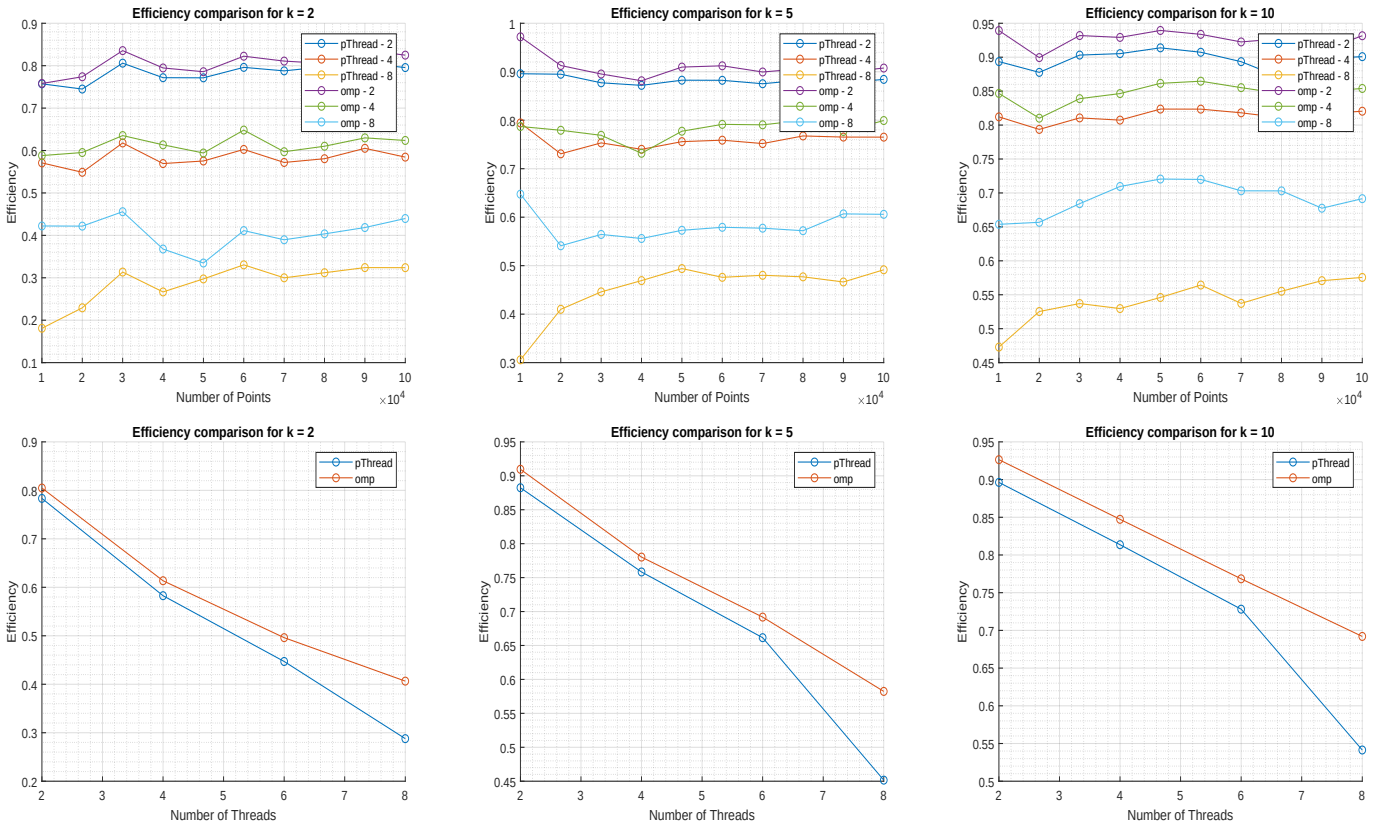

Figure 3: Graphs showing Efficiency of different implementations for different k

- We also see that the speedup in both pThread case is higher for 4 threads ≈ 2.5 for k $= 2$ and ≈ 1.7 for two threads for k $= 2$. As k increases, the speedup also increases.
- Efficiency decreases as number of threads increases which is reasonable from theory as well. With problem size, efficiency remain nearly constant. So for fraction f of code be serial:

$$\begin{aligned} Efficiency &= \frac{T_{seq}}{p \times T_p} \\ &= \frac{O(n)}{p \times O(fn + \frac{(1-f)n}{p})} \\ &= O(\frac{1}{p}) \end{aligned}$$

- We see that even efficiency increases as k increases.

Factor increases	Time	Speedup	Efficiency
Problem size	↑	=	=
k	↑	↑	↑
Number of threads	↓	↑	↓

Table 1: Summary of observations