COL380 Lab 1

Aditya Jain
2016CS10335

February 2019

1 Introduction

K-Means algorithm is a widely known and used algorithm for the clustering problem. Despite being simple it is very powerful. It also finds its usage in unsupervised machine learning problems. In this Lab assignment this algorithm was implemented in both sequentially and parallel paradigm. Parallel implementation was further done in two different ways , one using pthreads and one using openMP to understand both the methods. All codes were written in C++ and the results were visualized using python.

2 Sequential Implementation

The sequential algorithm which was given in the problem statement was implemented with random initializations. The following are the highlights of the implementation:

- A class of K-Means with data points, N and K its members
- created private method *updateClass* for assigning clusters to data points based on minimum euclidean distance metric
- another private method *updateCentroids* made to reassign centroids after new assignment of the clusters
- Ran the above mentioned methods until either convergence or a minimum number of iterations for better comparison between implementations

Convergence is defined when the change in the centroids is less than 3% of the sum of all the coordinates of all the centroids. Typically convergence came in under 5 to 6 iterations for the given test cases so to better compare time the algorithm was ran for at least a total of 25 iterations. So for stopping the iterations it should have converged and the number of iterations should be more than 25.

3 Parallel Design

The parallel implementation designs of the two methods are explained below.

3.1 pThread

Since pThread is low-level API for working with threads, the sequential code had to be modified a lot. The following changes were made :

- The *updateClass* method was parallelized by making it a void * type function and running the given number of threads on it. Each thread was given equal numbers of data points of the iterations
- The run function was called in the main function and not as a method of the class kMeans as done before for sequential code
- same stopping strategy was used as in sequential part
- mutexe locks were also implemented at first but due to their heavy toll on the running time they were removed later
- the *updateCentroid* method was run sequentially since it had a critical section and using locks slowed the whole program and also since it had only O(k*3) computations as compared to O(N*K*3) of the *updateClass* .

3.2 openMP

openMP is a high-level API and was very easy to implement on our sequential code :

- the *updateClass* method was parallelized using the *pragma omp parallel for* command of the API
- the number of threads were set at the starting of the program and each thread was distributed equal number of data points for computation and allocation of the new centroids in the method updateClass
- stopping and convergence conditions were not changed as from the above parts for better comparison of the speedup and efficiency of the program
- No kind of other changes were made in the sequential code

4 Observations

For observing the performance and correctness of the implementations speedup and efficiency were measured on same initialization and stopping conditions for all the codes.
Initialization: The first k centroids were chosen using rand() function without

N \ T	2	3	5	6	10	16	32
5000 (3)	1.44	1.80	1.66	1.75	1.53	1.26	0.75
10000 (4)	1.56	1.99	2.07	2.28	1.71	1.98	1.57
50000 (4)	1.63	2.21	2.10	2.49	2.10	1.94	2.24
75000 (5)	1.69	2.35	2.27	2.31	2.59	2.62	2.44
100000 (6)	1.78	2.43	2.43	2.69	2.66	3.01	3.03
500000 (7)	1.85	2.58	2.53	2.77	2.82	2.73	3.05
1000000 (10)	1.87	2.64	2.72	2.98	3.23	3.45	3.51

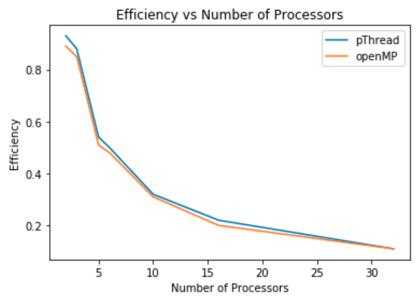
Table 1: SpeedUp for pThread ,N : ProblemSize, T: threads, (k) denotes k blobs

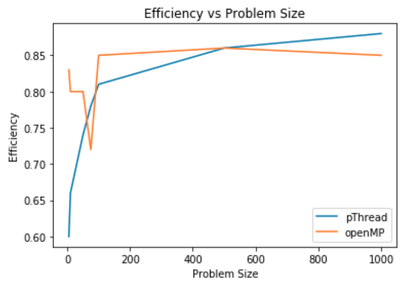
N \ T	2	3	5	6	10	16	32
5000 (3)	1.88	2.5	2.56	2.89	2.73	2.12	1.81
10000 (4)	1.4	2.4	2.39	2.86	2.73	2.67	2.38
50000 (4)	1.77	2.4	2.44	2.52	2.7	2.84	2.76
75000 (5)	1.77	2.16	2.48	2.81	2.58	3.23	3.19
100000 (6)	1.76	2.55	2.56	2.66	2.7	3.53	3.42
500000 (7)	1.79	2.57	2.6	2.86	3.06	3.08	3.15
1000000 (10)	1.77	2.56	2.56	2.88	3.13	3.2	3.37

Table 2: SpeedUp for openMP ,N : ProblemSize, T: threads, (k) denotes k blobs

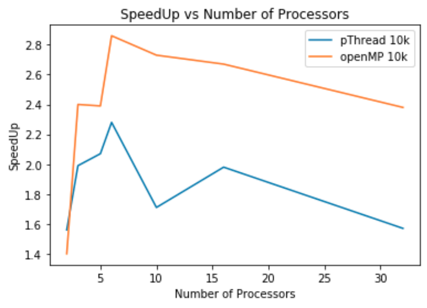
any seeding which gave same initialization to all the codes as was confirmed after printing them.
Stopping: to stop the kmeans loop both the convergence should be achieved and the number of iterations should be over a minimum threshold . The threshold condition was added because sometimes the algorithm converged in less than 5 iterations which gave a very high variance in the execution time. The threshold used in the implementations is currently set to be 25.
The following four tables shows the variations of the speedup and efficiency with problem size and number of threads. Datasets were created for this purpose using the given datapoint generators. Same datasets were used for evaluation of both pthread and openmp codes.

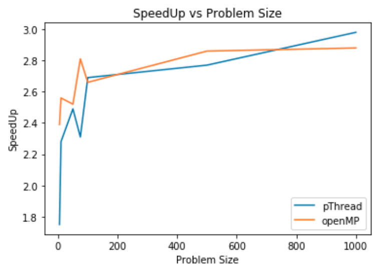
The running time of serial is $O(iterations * N * K)$ and of parallel is $O(iterations * N * K/numThreads)$. So speedup ideally should be of order $O(numThreads)$. And the efficiency is of the order $O(1)$. It can be seen that the results obtained are not ideal and vary too much from these.





The problem size is in thousands.The above graph showans that efficiency decreases as the number of processors are increased. Also how it generally increases with increase in the problem size. It happens because of the overhead which gets added when using an extra thread.The first efficiency graph is taken for a problem size of 10k datapoints, and the second graph for three threads. Below are the graphs of speedup for better visualization:





First graph of speedup used the 10k datapoints dataset and the next one used six threads . These choices were completely arbitrary to show the variation of speedup with the varying parameter only.

5 Conclusion

In this Lab assignment different APIs (pThread and openMP) for the implementation of parallel systems were used on K-Means algorithm. From the results that were obtained after exhaustive experiments on different datasets and different number of threads we saw how the performance varies across them as measured using speedup and efficiency. Max speedup comes with large problem size and more processors which indicates how powerful and usefull parallel systems can be for the computations of bigger datasets. But more threads take a toll at the efficiency of the system as shown in the graph 1. Also more threads do not always mean higher speedup as it also depends on the problem size which can be observed by table. We also confirms that Amdahl's law is followed as no speedup is more than the threshold provided by the Amdahl's law.