Dhirubhai Ambani Institute of Information and Communication Technology

High Performance Computing - CS301

Parallel Implementation of K-means clustering algorithm

Made By:-
Ronak Nahata - 201701416
Ronak Jain - 201701419
Jay Dev - 201701425

Introduction

K-means algorithm is an iterative approach to partition the given dataset into K different subsets or clusters, where each data point belongs to only one cluster. It assigns data points into a cluster such that the sum of squared distances between data points is minimum. The lesser the variation we have inside the cluster, the more the homogenous clusters we get.

Serial Algorithm and Complexity Analysis

The general algorithm we follow,

1. Provide the number of clusters K
2. Initialize K random centroids of the clusters from the dataset
3. Keep iterating the following steps until we get no change to the centroid data point.
    a. Find the minimum distance from each data point to the centroids.
    b. Assign the data point to the closest centroid.

c. Compute the centroid of each cluster by taking the mean of data points in that cluster.

This approach is known as Expectation-Maximization. Where the E-step is assigning different data points to the closest cluster in each iteration. Whereas M-step is calculating the mean of data points in the cluster after each iteration.

Scope of parallelism

The clustering algorithm requires massive computations, with distance between each data point and each centroid being calculated. Since calculation of the appropriate centroid for each data point is independent of the others, the algorithm provides a good scope for parallelism.

However, there is a bottleneck. The threads need to communicate among themselves to keep the centroid values updated, as more than one thread might try to access the same centroid point. In that case, it is imperative to ensure that both threads do not try to modify the centroid at the same time, as it might result in corrupted values and false sharing.

Parallelization strategies

For parallelization of the *k-means* algorithm, a **data-parallelism** approach was adopted. The *N* points were equally split among the number of threads (in case of an imperfect data split among threads, the remainder points are allotted to the last thread)

Key features of the algorithm:

- **Initialization:** The first *K* data points are chosen as the initial centroids

- **Data-parallelism:** Each thread assigned *N/num_threads* number of points

- **Thread function:** Each thread runs a loop (with a *max_iter* value of 100), in every iteration of which it computes the closest cluster centroid for every point assigned to it, and then assigns the point to that cluster. After every point is assigned to a cluster, the global (shared) cluster centroids are updated with the values computed for their coordinates from the points that were assigned to that cluster. This is again an instance of *data-parallelism*.

- **Stopping-condition:** The *L2-norm* is computed for every cluster centroid coordinates by comparing against corresponding values in the previous iteration. The norms are then summed and compared against the *threshold* value, chosen to be *1e-6*. All threads break from the iterations loop as soon as the *delta* value goes below the *threshold*.

Further Approach Analysis:

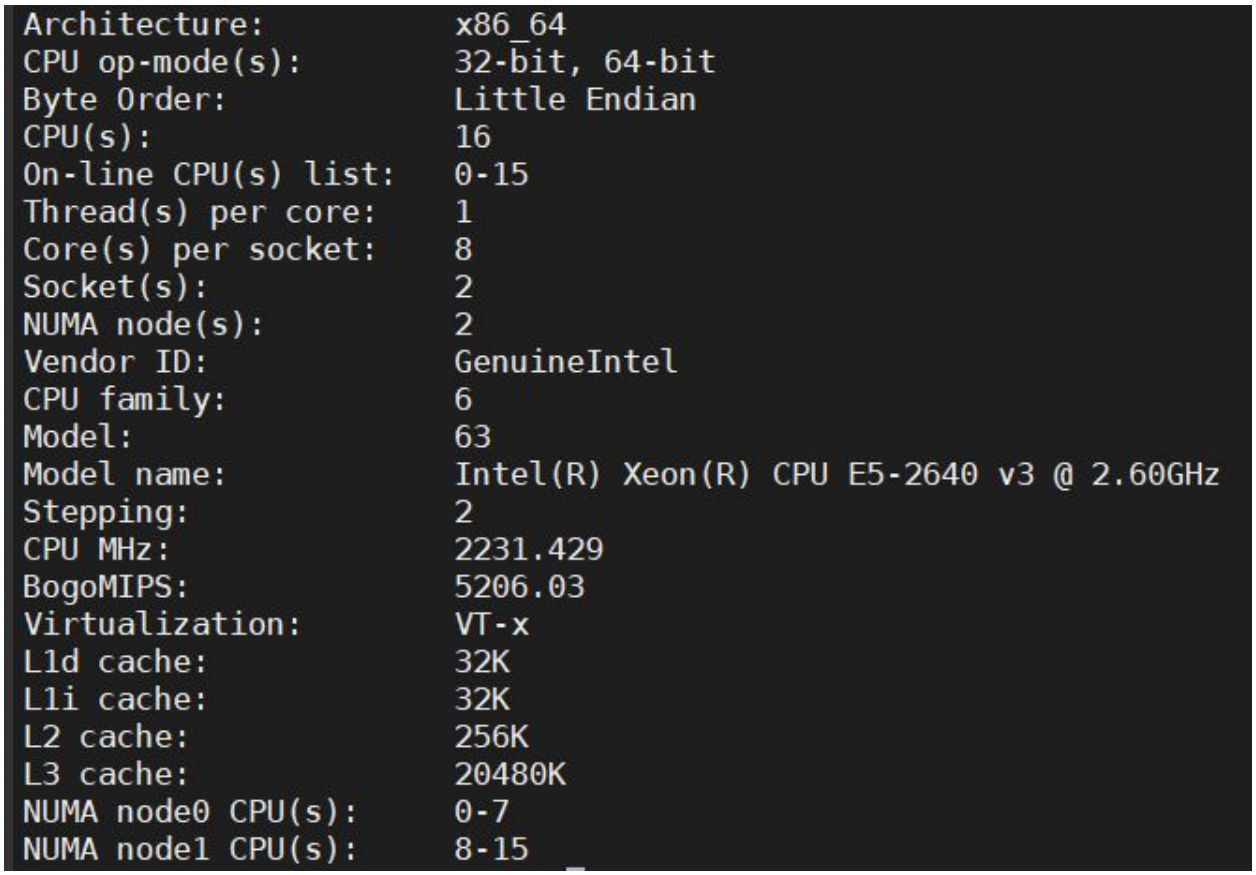
Mutual-exclusion and Critical section: The first *thread-synchronization* strategy which ensures that updates to the shared cluster *centroid* array are undertaken by each thread only is undertaken using *pragma omp critical* construct. This is essential to avoid data race due to different threads attempting to modify the shared array.

Barriers: After every iteration, barriers are used for thread synchronization so that all threads view the same updated value of the *centroids* array. Two instances of barriers are used. The first one for synchronization of the *centroids* update, and the second one following the first to synchronize the value of *delta,* again a shared variable among all threads.The *pragma omp barrier* construct is used for progress synchronization.

False-sharing: To minimize false sharing, shared variable updates for the *centroids* array are minimized. Moreover, local variables are used wherever feasible (for instance in *delta* value computations, variable temp is used to store calculated delta value which is then updated at the end of thread execution).

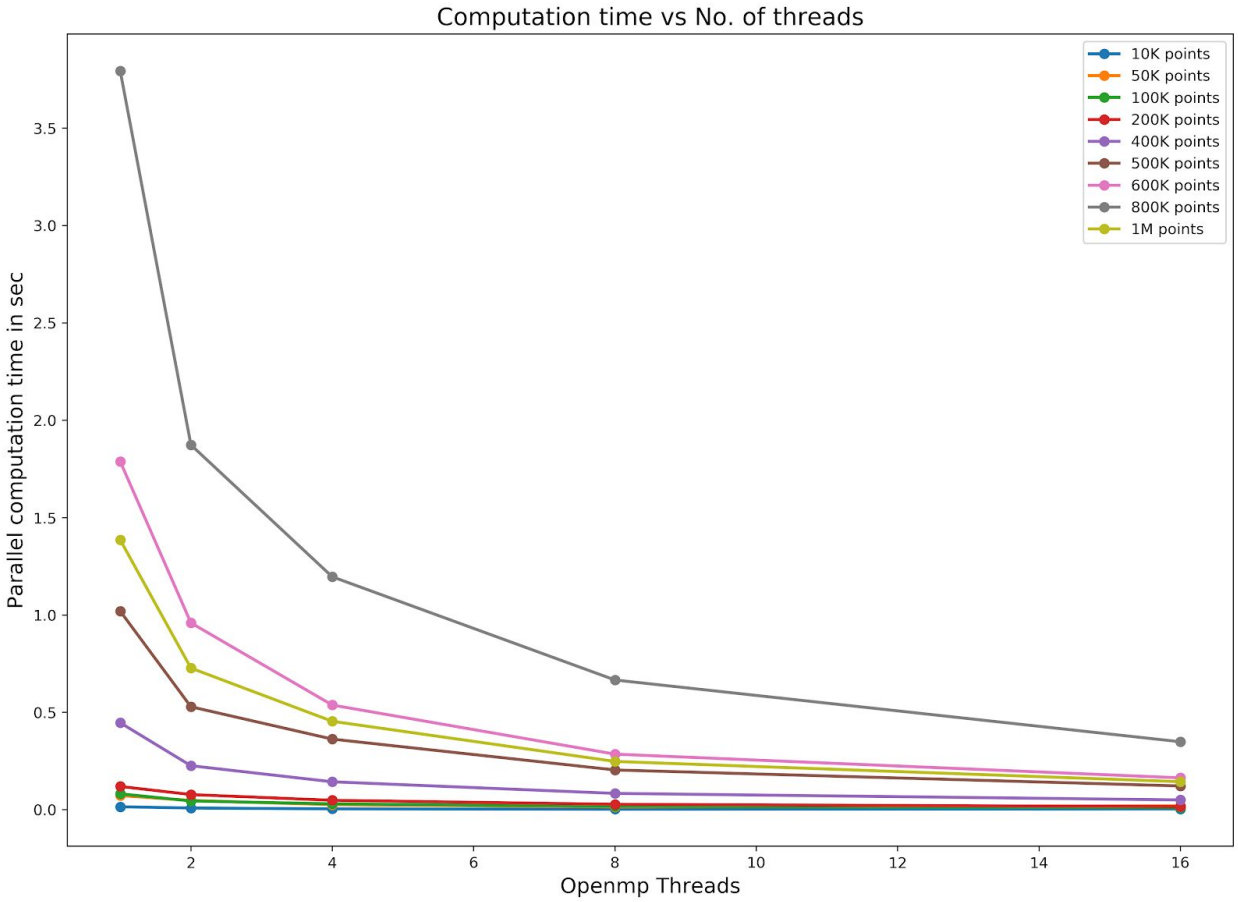
Load-balancing: The nature of parallelization probably does not require an explicit load-balancing strategy since its an **SPMD**-based parallelization scheme. One obvious source of unbalanced load would be the case of a slightly larger number of points (bounded by *num_threads*) processed by the last thread.

Hardware Details:

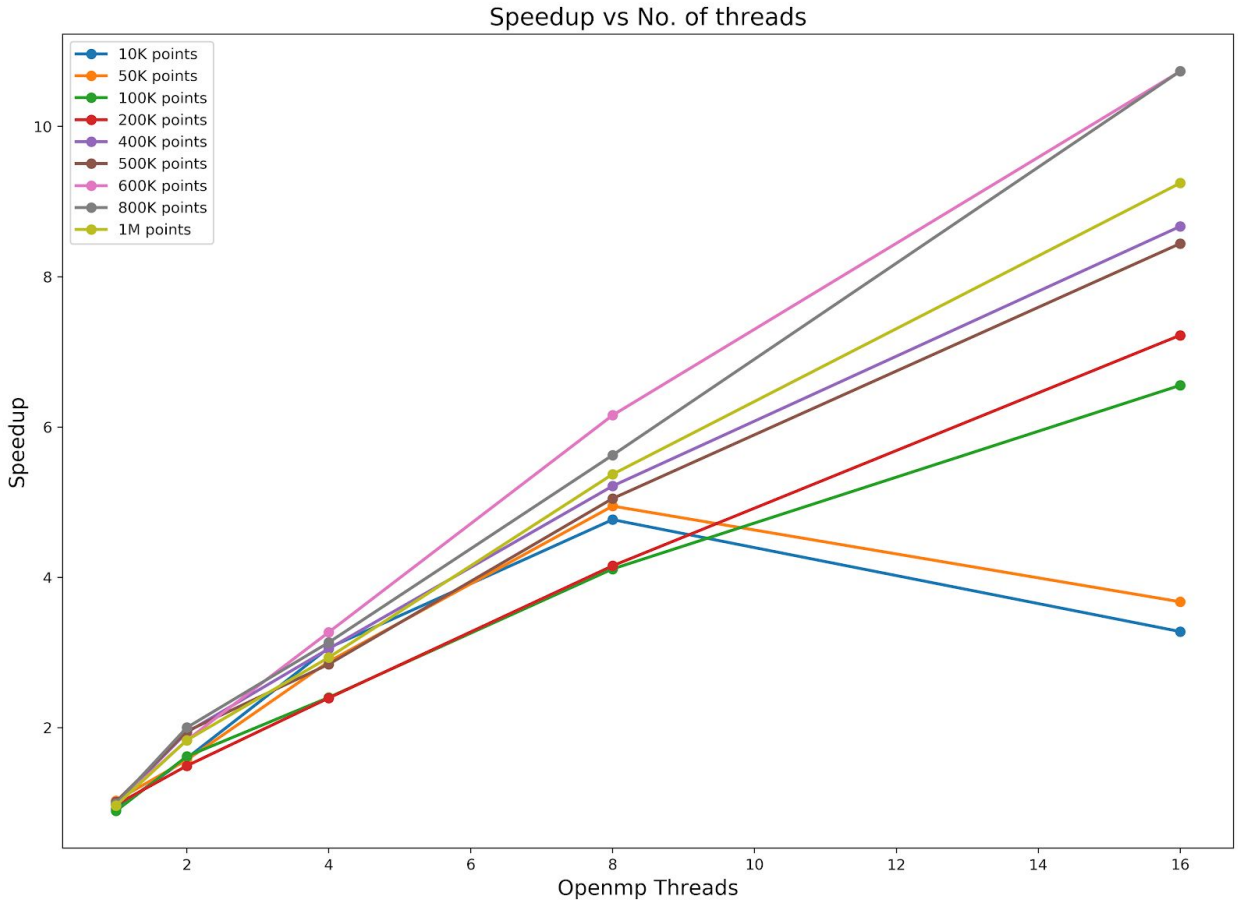


```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                16
On-line CPU(s) list:   0-15
Thread(s) per core:    1
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
Stepping:              2
CPU MHz:               2231.429
BogoMIPS:              5206.03
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
NUMA node0 CPU(s):     0-7
NUMA node1 CPU(s):     8-15
```

Results and Discussions:



As expected from any multithreaded programming method, the computation time of the algorithm decreases as the number of threads increases so much so that for 16 threads, the overall computation time for various datasets is close enough to zero. The difference between computation time of different datasets on a fixed number of threads is different because of the unequal distribution of points for each of the respective threads.

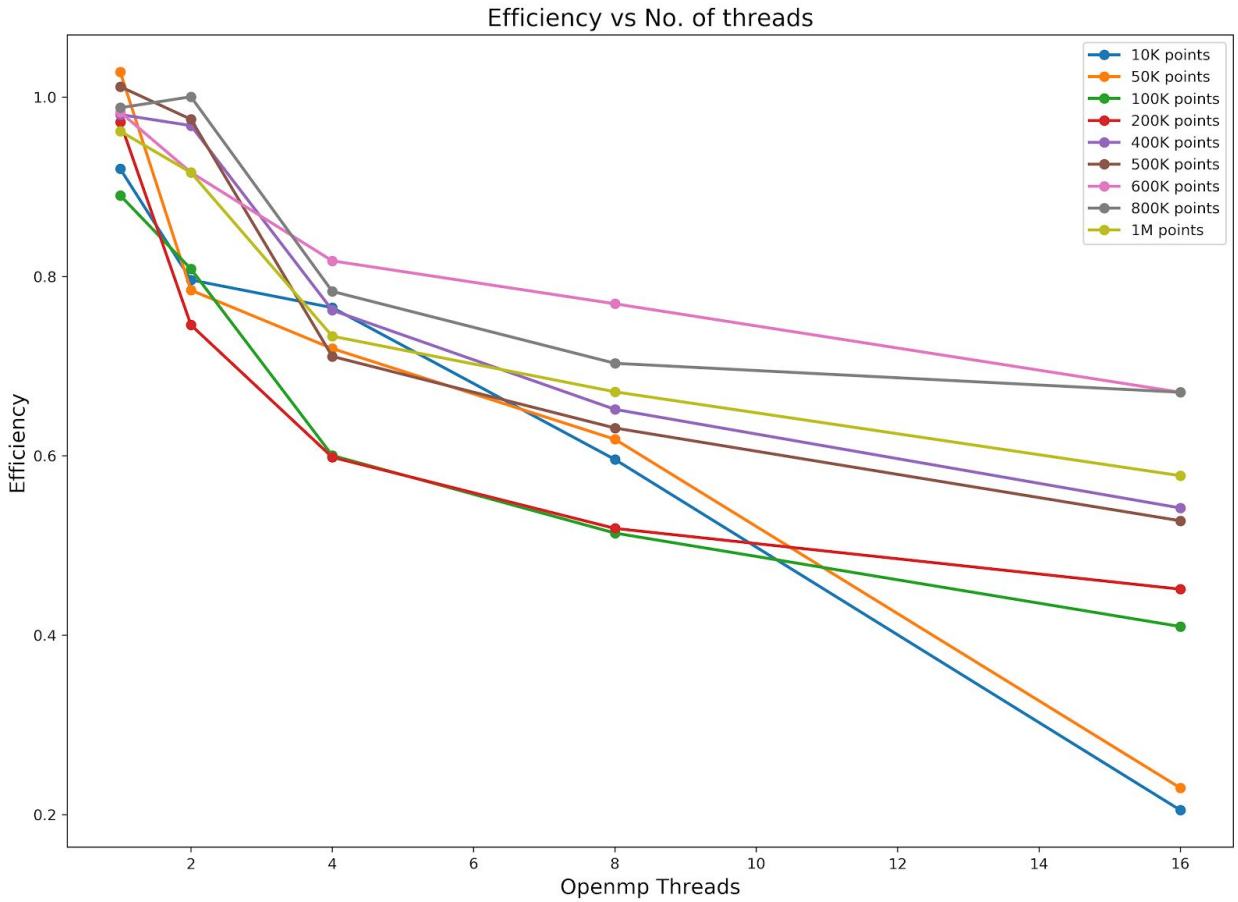


Theoretically, if the problem size is too low, the speedup decreases as the number of cores increases. This is because the cost of overheads will be far more than actual cost of computation, thereby giving low speedup. But for larger problem sizes, the overhead cost is negligible and the work could be distributed among many processors, thus the speedup increases as the number of processors increases. The **best speedups** are obtained for the larger-sized threads (sizes 800,000 and 1 million). The worst performance is for the dataset-sized 10,000.

Linear speedup is not achievable, in general, because of contention for shared resources, the time required to communicate between processors and between processes, and the inability to structure the

software so that an arbitrary number of processors can be kept usefully busy.

Superlinear speedups are observed in some runs at single-threaded run due to out-of-control system execution properties and/or latencies.



The efficiency curves decrease for every instance run with an increase in the number of threads. This is again due to the diminishing returns property on including an extra thread - a consequence of the **Amdahl's Law.**

The efficiency increases if the problem size is increased keeping the number of processing elements constant. And this fact is clearly visible for 16 threads.

The efficiency at *num_threads = 1* goes above 1.0 in certain runs due to superlinear speedups perhaps due to out-of-control system latencies during particular executions.

The efficiency curves demonstrate **best efficiency at the larger-sized datasets** (800,000 and 1 million) and the **worst-performance at smaller-sized ones** (10000).

Future Scope of Improvement:

The given algorithm is coarse grain, specially for large data inputs. In such cases, it might be a good idea to implement the given code on a distributed memory system. Also, MPI provides special libraries for parallel data input, which can be a huge advantage in terms of speedup.

Also we have stuck to just the clustering of 3 dimensional data points. We can attempt to cluster multi dimensional data points along with implementing dimensionality reduction.We can try to implement this using GPU's but this would be quite challenging since the cost of overheads of GPU's is paid only if we are working with hundreds of threads at the same time.

References:

1. Inderjit S. Dhillon, Dharmendra S. Modha, in ‘A Data Clustering Algorithm On Distributed Memory Multiprocessors’
2. https://medium.com/@dilekamadushan/introduction-to-k-means-clustering-7c0ebc997e00