CS633: Assignment 3 (Fall Semester 2019-20)

Yugesh Kothari (170830)

Clustering Moving particles in 3D space

K-Means clustering

For the purposes of this assignment I have adopted and used the K-Means clustering algorithm, that, given a number of clusters #K, repeatedly assigns points to these clusters based on their distance from the centroid and recomputes the centroid based on the assignment. The idea behind this is that, after a reasonable number of iterations, the centroids for these clusters start to converge.

Pseudo Code

To implement the K-Means algorithm parallely, I decided to begin with a set of #K centroids (as guess centroids), divide the dataset across processes, and apply the algorithm described above.

The code at a high level, does the following :

```
dataset = read_from_file; //at each process
centroids = initialize_centroids(); //at root process
MPI_Bcast(centroids);
dataset.split_across_processes();
while(threshold.not_reached()){
   newCentroids, cluster_size = perform_kmeans(dataset.slice, centroids);
   MPI_AllReduce(newCentroids);
   MPI_AllReduce(cluster_size);
   if(newCentroids~centroids){
      break;
   }
}
print_cluster_data();
```

A couple of heuristics adopted are -

1. To speed-up the *perform_kmeans* computation process, I do not divide the *newCentroids* with the *cluster_size* at each process individually, rather I accumulate them all at all processes first and them Reduce it (MPI_AllReduce followed by newCentroid:=newCentroid/cluster_size). The advantage of this is that it is easier to merge the individual partial k-means, which would otherwise require division followed by multiplication and then another division.
2. The initial centroid guess is calculated at the root process and broadcasted out : it is simpler than doing an AllReduce for the initial guess.
3. Data is decomposed uniformly (in contiguous chunks) among all nodes - the last process might have an uneven number to balance edge cases. An alternate approach would have been to distribute the uneven number of data-points across all processes.
4. The optimal value of #K for the clustering was determined experimentally by trying different values, looking at the loss and picking a #K based on the knee in the graph (as discussed on piazza).
    a. The following table contains the values of #K adopted for different time_sets.

Time Step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dataset1	17	17	18	18	18	19	19	19	20	20	21	22	22	23	23	22	22
dataset2	6	6	6	7	4	3	6	6	7	8	8	8	8	7	7	7	-

Observations and Design Decisions

1. Given the size of the binary files were not very huge, reading the entire dataset at each node was not a huge overhead; the advantage on the other hand was that the initial centroid guess for the clusters were consistent.
2. Since the root process had to read the entire dataset anyway (to compute the initial guess), it made sense to allow other processes to read the data too, since it would take nearly the same time anyway. Hence, doing an MPI_Scatter call was avoided (which would have added additional network bandwidth).
3. Linux (Ubuntu) does not display the correct file size (rather rounds it off to the nearest block size) unless explicitly asked for it using a system call like *stat*. This caused some confusion initially as I was getting a lot of zero-ed points in my dataset (due to incorrect computation of number of points).
4. As the number of Processes increased, the design decisions taken started to reflect - the pre-processing time crossed the actual time of computation. Therefore, a balance between the two should always be found.
5. HPC2010 does not support initialization in for loops (kind of surprising - I would never have thought of it).
6. The approach does not scale very well, because as the number of processes increases the pre-processing time exceeds the computation time.