REPORT
(19111063)

Overview:

First, the files are divided among the no of processes available. Then the program reads the data file using a custom MPI datatype and MPI collective read call (self (aggregator)). The k-means algorithm will run over all the files available to a process. I have already performed some of the tests over multiple K (no of centroids) and found that optimal value of K life b/w 15-25, so, I have taken 20. And finally, the clustering info will get printed in the output files using simple write.

Pseudocode:

K-Means starts by randomly defining k centroids but by using kmeans ++ initializer the centroids get uniformly distributed over the sample space. From there, it works in iterative (repetitive) steps to perform two tasks:

1. Assign each data point to the closest corresponding centroid, using the standard Euclidean distance. In layman's terms: the straight-line distance between the data point and the centroid.
2. For each centroid, calculate the mean of the values of all the points belonging to it. The mean value becomes the new value of the centroid.

1. Kmeans++ initialization

```
void kmeans++init()
{
   //Initializing first centroid
   c[0] = allPoints[rand()%len];

   for(j = 1; j < K ; j++)
   {
      init nextCenter //eg. c[1]

      for(i = 0; i < len ; i++)
      {
         findOutTheFarthestPoint
      }
   c[j] = nextCenter;
   }
}
```

2. Kmeans++ Algo

```
void kmeans()
{
      for(iterate for max 500 rounds) //or till new cluster centers are same as old ones
          {
          //Initializing centres for all points
           for (i = 0; i < len; i++)
                    {
                     Assign points to the centroids
                    }
          //for each cluster
          for(i = 0; i < K; i++)
                    {
                              Find new cluster centers for each clusters/centroids
                              Assign new centroid values
                    }
          }
}
```

3. Main Function

```
int main(int argc, char *argv[])
{

   for(i =1; i <= np ; i++)
   {
      Assign files to each process/ Divide among the process
          e.g 16/1,16/2,...
   }

   for(j = arr[myrank] ; j<arr[myrank+1]; j++)
   {
          // Custom data type to read and handle cluster data
      MPI_Datatype pointtype;
      MPI_Type_contiguous(4,MPI_DOUBLE,&pointtype);
      MPI_Type_commit(&pointtype);
          //MPI file pointer
      MPI_File fh;
      MPI_Offset filesize;

      //Timing the preprocessing of each time step
      double preTime = MPI_Wtime();

      MPI_File_open(MPI_COMM_SELF,strcat(filename1,lName[j])
,MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
      MPI_File_get_size(fh,&filesize);
      MPI_File_read(fh, allpoints, filesize,pointtype , &status);
```

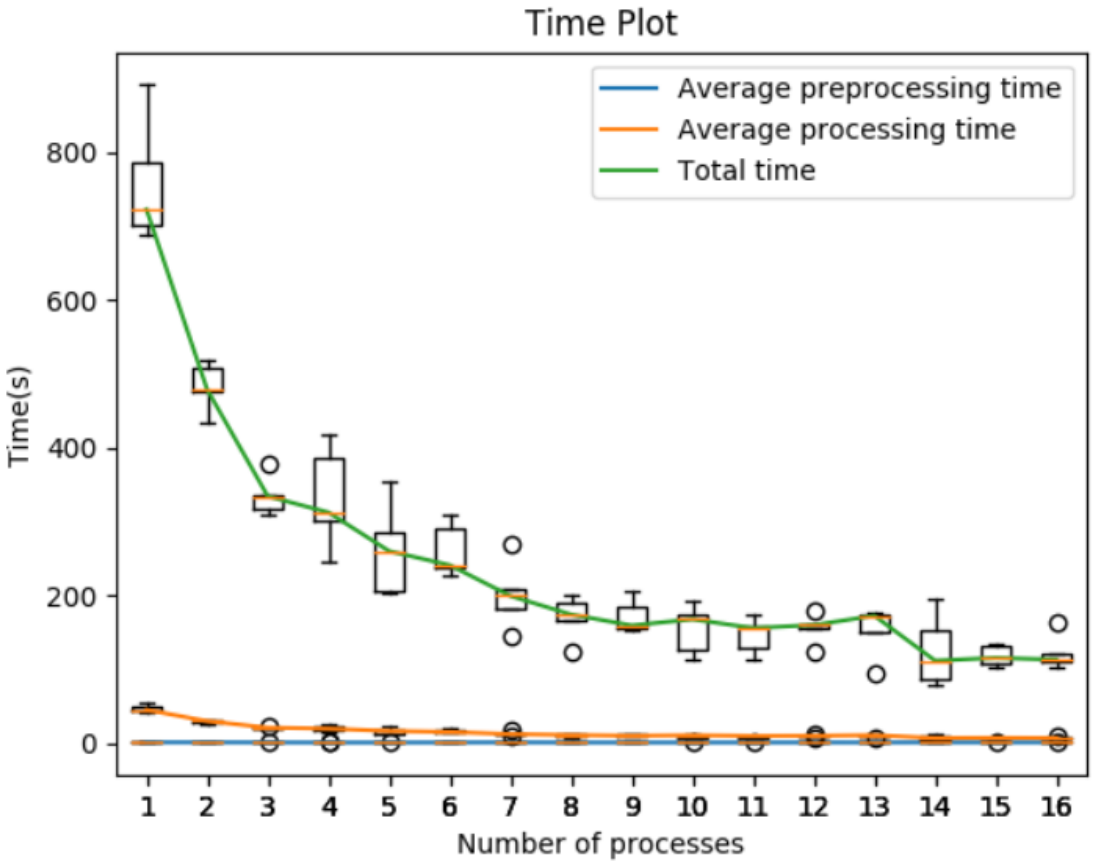
```
    //number of points actually read
    MPI_Get_count( &status, pointtype, &count );

    preTime = MPI_Wtime()-preTime;
    sumPreTime+=preTime;
    //############Clustering################

    //Clustering Timmer
    double clustering_time = MPI_Wtime();

    //Initializing the cluster-centres using kmean++ algorithm
    kmplus(allcpoints,count,centr,K);

    //Kmeans call
    kmeans(allcpoints,count,centr,K,clusterSize);
    clustering_time = MPI_Wtime()-clustering_time;
    sumClusTime+=clustering_time;

    for(c = 0 ; c < K ; c++)
    {
       concat all the outputs of (Centroids)
    }
  }

  totalTime = MPI_Wtime() - totalTime;
  //########## Time calculation and all ###############
  //Reducing out the time that we need


MPI_Reduce(&sumPreTime,&avgPreProTime,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_W
ORLD);
MPI_Reduce(&sumClusTime,&avgClusTime,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WO
RLD);
  MPI_Reduce(&totalTime,&maxtTime,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);

  //Gathering lenths since file is not evenly distributed
  MPI_Gather(&mylen, 1, MPI_INT, recvcounts, 1, MPI_INT,root, MPI_COMM_WORLD);

  if (myrank == root) {
        Prepare receive buffer and the offsets since the
  }

  //Since we have calculated the receive buffer, counts, and displacements. So now we can
gather the strings
// Gattering all the info of each file from different process based on the metadata provided by
the gather just seen before.
```

```
 MPI_Gatherv(str, mylen, MPI_CHAR,tString, recvcounts, displs, MPI_CHAR,root,
MPI_COMM_WORLD);

   //Adding timings to the total string performed by the root
   if(myrank == root)
   {
         // Average prepocess time
      sprintf(t1,"Average time to preprocess: %lf",avgPreProTime);
         //Average Clustering Time
      sprintf(t1,"Average time to process: %lf\n",avgClusTime);
         //Average Max Time
      sprintf(t1,"Total time: %lf\n",maxtTime);
      printf("%lf %lf %lf\n",avgPreProTime,avgClusTime,maxtTime);
   }

   //Final File Write performed by the root
   if(myrank == root)
   {
      fprintf(fp,"%s",clusterData);
   }
   MPI_Finalize();
}
```

4. Data Decomposition

I handle the decomposition of the data by dividing the files among the given number of processes as uniform as possible eg. For 16 files and 3 processes = distribution will be [6, 5, 5]. Making each process as its own aggregator each reads the file and stores the point in the temp buffer.

5. Observations



Plot CSE DATA 1

Processing time:

a). CSE

For the performed run on the CSE cluster, I took ppn 4 as every CPU was having min 4 physical cores. Overall the time for clustering decreases as the number of processes increases which divides the workload and reduces the processing and overall time. Main thing to notice is that after assigning 6 or more processes the rate of decreasing the processing time reduces.

b). HPC

For the performed run on the HPC cluster, I took ppn 8 as every Node is having min 8 CPU (since each socket is having 4 cores). Same as earlier, the time for clustering decreases as the number of processes increases which divides the workload and decreases the processing time. Assigning 6 or more processes the rate of decreasing the processing time reduces.

Because of the higher ppn and exclusive availability of the node to my job, the program get executed faster over here. But, sometimes if we get lucky and no other jobs were

running on the CSE cluster, due to the higher core clocks of the 8th gen coffee lake architecture as compared to Xeon (Nehalem arch) (which prefers more cores than speed) we can get our jobs faster than HPC 2010, but it has very few chance of doing that.

Scalability:
As per my architecture, which divided the files among the process and then perform the main algo, 6-8 processes are enough to handle, but we can still divide the tasks by like performing k-means on completely different process but it will increase the communication time and when data will be small, it will not be worth to do so.

Preprocessing time:
As per the plots, the preprocessing time for every run is almost same and it's not depending on almost anything. So, it's working under O(1).