Parallel Computing

Assignment 3

Shreya Sharma (CS19111087)

Pseudo Code for Parallel K-Means Clustering using MPI

```
Function Classify ( means, items, I ,K )
{
        Calculates the euclidean distance of a data point with index I stored in an array whose address is
        stored in the pointer argument items from all K means.

        Returns the index of that cluster whose distance is smallest from the data point
}

Function CalculateMeans ( items, k, rank, count )
{
        // Argument "count" stores the number of  datapoints for the process who calls CalculateMeans
        function

        If ( rank==0 )
        {
                Initializes means to initial datapoints (passed in items) of process with rank 0.
                Initializes clusterSizes (datapoints in a cluster) and sums (sum of datapoints in a cluster)
                with 0.
        }

        While ( iterations for convergence )
        {
                Initializes 1-D array clusterSizes of size k (datapoints in a cluster) and sums of size k*3
                (sum of datapoints in a cluster, 3 is for data point dimension) with 0.
                Process with rank 0 broadcasts means to all process.

                For ( all items  )
                {
                        Process with rank "rank" calls function Classify to get index "ind" of the cluster to
                        which it belongs and increment the no of datapoints in cluster with index "ind"
                        by 1.
                        It then adds the datapoints coordinates to the array sums of all datapoints
                        coordinates in the cluster with index "ind".
                }
        Process 0 get sum of all datapoints and number of data points in a cluster by using MPI_Reduce
        in 1-D array Gsums ( size k )  and GclusterSizes array respectively.

        If ( rank is equal to 0 )
        {
                Calculates means by dividing Sum of coordinates of all datapoints by sum of total number
                of datapoints in a cluster (Gsums / GclusterSizes).

                Update means in array “means”  (size k*3, k is for number of clusters and 3 is for number
                of coordinates in datapoint)
        }

        Process 0 broadcasts updates means to all processes.
}

Function Main ( Argument1 to identify for which dataset function is called, Argument2 for number of
Process )
{
        //already stores k values for both data sets

        All processes opens file and calculate file size in variable filesize.
        If ( rank is equal 0)
        {
                Reads the datafile in an 1-D array buf.
                Skips the Ids and stores coordinates of datapoint in an 1-D array buf2.
        }
        All process calculates proc_size ( filesize with id removed )

        Process 0 scatters the datapoints stored in array buf2 to all processes in an 1-D array items.
        Calculates number of datapoints belonging to a process and stores in variable “count”.

        All processes calls function CalculateMeans ( items, k, rank, count )
}
```

K-Means

The algorithm works as follows:

1. First, it initializes means to some random number between minimum and maximum of data points.
2. It then categorizes data to its closest mean by calculating distance from all means and updating means by taking averages of data points contained in that mean.
3. Repeat Step1 and Step2 for convergence. At the end, Data points get clustered in groups.

Elbow Method is used to find out the number of clusters.

Data Decomposition Technique

Every process opens and calculates the size of data file and stores it in a variable filesize. Variable filesize is then divided by 32 to get number of data points. Only Process 0 reads the datafile in an array. Process 0 then loops over the array with filesize as counter and store the coordinates in a separate array.

Every Process then calculates Proc_size by dividing by 4(no of data points in a line) and multiply by 3(dimensions) to get no of points excluding the ids. A process will get (Proc_size)/(number of Process) data points.

Process 0 then scatters the data to every process in an 1-D array “items” of size (Proc_size)/(number of Process).

Observations

- Pre-processing time (data distribution data) overlaps with processing time.
- As data is read by only one process and scatters it to others that’s why time is increased a little as compared to if MPI_File_Read_at is used. This will not add time to scatter and loops to it.
- As number of processes are increased, total time decreases as load get distributed among more number of processes.
- As number of processes increases above 13 in cse, time tends to get saturate. In hpc, time decreases more than cse.
- In hpc, time for data2 is taking around 7 sec and in cse, it is taking upto 35 sec.
- Processing time is more when number of processes is 1 but as processes is increased by only 2, it almost equals pre-processing time.
- I have tried using MPI_File_Read_At and giving offset, and skipping ids while looping but not able to remove ids. After printing data it was not starting from ids, may be, it is possible after setting view of the file.

Compile/Run

- **For cse**: bash run.sh

- **For hpc**:

  make hpc

  qsub sub.sh