# Assignment 3 Report
# Parallel Computing

**Sharath HP (19111082)**
sharhp@cse.iitk.ac.in

## Program to cluster moving paticles in 3D space using MPI

**Goal**: Write a program to cluster moving particles in a 3D space. The particle locations are output of a simulation. The simulation output (per file) corresponds to a time step. Speed up program execution using MPI. The code must demonstrate scalability upto a chosen value of P (max number of processes).

**Output**:

- Average pre-processing time per time step (for a given no of processes)
- Average processing time - time to cluster - per time step (for a given no of processes)
- Total execution time

## 1 Implementation Details

### 1.1 Pre-Processing Phase

**Algorithm 1:** Data Distribution

**Result:** Processes parallely compute respective Data sets (time steps) to work on
tStepTotal = User input total time steps;
nProc = User input total # of processes;
**if** *tStepTotal % nProc == 0* **then**
    tpProc = tStepTotal / nProc;
    tOffset = myRank * tpProc;
**else**
    val = nProc - ( tStepTotal % nProc );
    **if** *myRank > (val - 1)* **then**
        tOffset = val * (tStepTotal / nProc) + ( myRank - val ) * (tStepTotal / nProc + 1 );
        tpProc = tStepTotal / nProc + 1;
    **else**
        tpProc = tStepTotal / nProc;
    **end**
**end**

For instance if there are is a total dump of 17 time steps (tStepTotal) and we have 5 processes (nProc), each will get to work on 3, 3, 3, 4, 4 time steps (tpProc corresponding to process with rank myRank) respectively. The code comments clarify this further.
The data distribution scheme is chosen this way under the assumption that the entire data dump is available to us. In a more natural setting where the data is generated on the fly, an alternate implementation of the overall algorithm is necessary. However the chosen method is found to do well, even in an online fashion (we could process the dumps in parallel in real time). For convenience, the maximum number of processes is chosen to be P = 16. It could very well be extended for larger

Preprint. Under review.

sizes of P (bigger than the total number of time steps to process), however, it would then require a slight modification to the original algorithm to handle such a case.

### 1.2 Processing Phase

**Algorithm 2:** Clustering

```
Result: All points are assigned to their respective clusters, cluster-means are computed
tStep = tOffset;
tStop = tStep + tpProc;
while tStep < tStop do
    tStep++;
    Read file corresponding to tStep;
    Cluster all data points based on k-means (k selected from a list of pre-computed optimal values*);
    thresh = minimum threshold;
    maxIterations = maximum number of iterations;
    iters = 0;
    if threshold > thresh and iters < maxIterations then
        k-means++ initialization of means;
        Assign points to cluster;
        Compute new means;
        thresh = dist (old means, new means) ;
        iters++;
    else
        Exit;
    end
end
```

The threshold and the maximum number of iterations described above is chosen based on a general standard and after validating the same over a test run on a subset of the data set. For data1 set, the optimal values of k have been pre-computed and stored (especially since we're mainly interested in comparing the behavior of the algorithm for different configurations). Whereas for data2 set it is fixed at a minimum of k = 20, owing to the huge processing times and overhead involved therein.

### 1.3 Post-Processing Phase

At the end of k-means execution, we have with us all of the cluster-means corresponding to different time steps. This is distributed across processes, hence we need to aggregate it at the root.

- Invoke **MPI_Gatherv** collective call to first get the different number of clusters for each of the different time steps (receive at root process)
- Prepare the receive buffers based on the previous values.
- Invoke **MPI_Gatherv** to now collect the actual means of the various clusters across time steps (gather at root).
- Make a couple of **MPI_Reduce** calls to collect all the timing information corresponding to various time steps. The root process can then write all of the collected information to an output file.
- For each of the above calls, in order to facilitate the transfer of data across processes, a custom data-type is created using **MPI_Type_contiguous** API and a custom operation for **MPI_Reduce** is defined as well (although not incorporating it in our current execution).

More details on the actual implementation of the entire algorithm is available and well commented in the source code. Please refer the same for further details.

## 2 Results and Observations

### 2.1 Experimental Results and Scalability

The MPI implementation of the parallel k-means algorithm outperforms that of a serial counterpart by a large margin. This can be seen comparing the execution times with P = 1 (equivalent to a serial execution set-up) and that with higher process sizes. We observe a significant drop in execution times in the latter case.
Our algorithm is found to exhibit strong scaling up to a maximum of P = 16 processes. It could very well be extended to larger process sizes, after modifying our original algorithm. This is as per natural expectation, since upon increasing the number of processes, the load on one process/ node decreases and all of the co-operating processes/ nodes read and process the various time steps in a distributed manner (parallely).
With increase in number of processes the execution time taken decreases. However as we increase the number of processes beyond a certain point, the effective decrease we observe becomes lesser and lesser i.e., the curve starts to flatten out. This is also as per expectation, as with more processes, the subsequent communication overhead starts to become significant. Also by Amdahl's law, for a given fixed problem size we have a theoretical bound on the speed up achievable, limited by the serial parts of program execution.
The results are consistent across multiple executions and both running environments (CSE cluster as well as HPC2010). However it is observed that the speed-up achieved is marginally higher in case of execution on the HPC2010 cluster as compared to the CSE cluster. The difference in results becomes conspicuous especially in the case of data2 set.
Although recorded results show consistency, during some dry runs, the variance in the HPC cluster observed was a bit more as compared to that of CSE cluster, this can be attributed to the fact that we get to choose process placements in the CSE cluster (we get to decide what goes in the hostfile), whereas this is not the case with HPC cluster. Even in the CSE cluster for some dry runs there was an abrupt times observed, mainly due ot the varying nature of loads on the department-wide shared cluster.
Comparing the execution times across the different data sets, it is observed that the data2 set consumes a significantly larger time for execution as opposed to data1. This is mainly due to the large size of the data dump in the time steps therein. Not only pre-processing (mainly file reading) but also processing time is significantly much higher for data2 set.

### 2.2 Challenges

During evaluation, the results of execution were found to be consistent across multiple runs at one instance of time (and day), while highly inconsistent during another. This can be attributed mainly to the varying loads (jobs/ processes therein), number of active users, job scheduling limitations (in case of HPC2010) among other conditions (no of hosts alive etc...).
Due to the large execution times of the algorithm (owing to the large data sizes) the job was terminated considerable number of times on the HPC2010 cluster. Also the hosts were becoming inadvertently down at times on the CSE cluster, this especially hampered the entire execution. Due to this multiple executions and different configurations in that was a common solution, which made collating the results and the overall execution step a bit more complicated.
It is to be noted however that the plots obtained as a result were drawn as box plots as can be seen from the plot.png files in their respective folders. For some strange reason, there is no apparent way to widen the gaps between individual box-plots (plotted with pandas library, via dataframes - it keeps a constant factor based on the total no of box plots ie., gap between 1 and 2 is same as that between 2 and 4 is same as that between 4 and 8 etc).
Due to the extreme times encountered during execution, the code was executed for a comparable set of process sizes (1, 2, 4, 8 and 16) only. Because of the limitation of the box plots mentioned above, a separate image has been drawn on plotL.png collecting the same results (except the box plot isn't shown, although results are still averaged out and accurate). While the image plot.png containing the box plot coherently depicts the strong scaling observed, plotL.png shows a more accurate (true to scale) representation of the same.