CS633 - Parallel Computing
Assignment 2

Shivam Kumar
170669

Prateek Varshney
170494

Topology awareness based collective optimization is generally done in two ways:

1. Changing the underlying pattern of communication.
2. Changing the mapping of processes.

In our optimisations for the default MPI_Bcast, MPI_Reduce, MPI_Gather and MPI_Alltoallv collective calls, we incorporate both the above principles and try to design communication pathway routines (via creating new appropriate communicators) as well as change the MPI function calls used within each of the above collectives. Our strategies are somewhat loosely along the lines of the proposed solutions by Krishna et. al. [1]

1 Designing Topology-Aware Collective Algorithms

MPI implementations such as MPICH2 use tree-based algorithms to implement these operations and are optimized considering the conventional single-core systems use case [2]. As they are agnostic to the topology of the underlying multi-core system, there remains a scope to optimise them to lower the communication costs across clusters.

We know that the CSE Server has a simple tree topology with the intra-group distance (hops within a group) being 2 and the Inter-group distance (hops across groups) being 4. We use this information to create sub-communicators to group various processes as follows:

1. Create an intra-group communicator as to include all the processes that belong to the same leaf-switch (intra-switch)
2. Assign one process on rank 0 of each group as the group leader
3. Create an inter-group communicator to include only each of the group leaders of the individual groups.

The resulting virtual topology is shown in Figure 1. Here the red rectangular enclosed region denotes the inter-group sub communicator containing only the group leader of each group.

As the latency involved in inter-group message exchanges (4 hops) is higher than intra-group (2 hops) and intra-node (0 hops) exchanges, the cost of the entire operation is dominated by the number of inter-switch operations. Our proposed method helps minimise the number of interswitch exchanges (and therefore the number of hops required) and lowers the inter-node component of the cost of collective operations, hopefully leading to better performance for both small and large messages.

We then perform an entire collective operation (for e.g., say MPI_Gather) in the following manner:

- Each of the Group Leaders performs the intra-group operation (MPI_Gather). Note that this does not involve any inter-switch/group communication.
- The data from each of the Group Leaders is collected by the "Root" Group Leader via an inter-switch/group communication (MPI_Gather).

Note that in the general case, the actual root process calling the MPI function may not necessarily be the group leader. In such a case, an additional step of transferring data between the "Root" Group

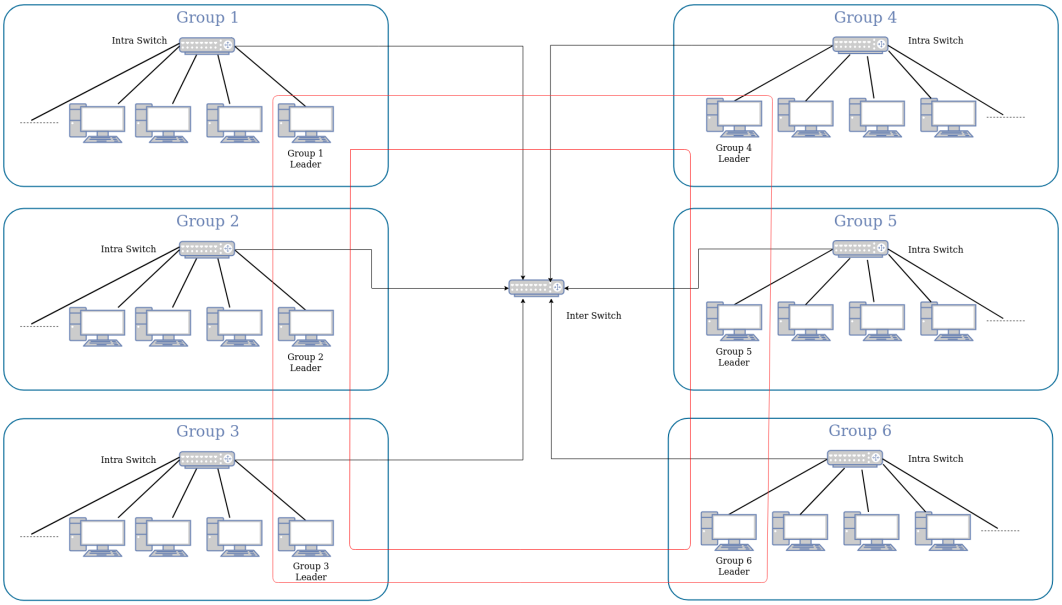

Figure 1: Leveraging the Network Topology: The Inter switch denotes the outer most level switch of the Tree Topology while the Inter switch denotes the root switch for each of the trees representing one cluster group. The 6 nodes included in the Red Rectangular region denote the Group Leaders for each of the individual groups respectively

Leader and the actual root process will be required. Since this does not involve any inter-group communication it will not be expensive (at max. 2 hops vs 4). Moreover, since we have the liberty to decide which node to be assigned as the Group Leader, we can safely omit this step by assigning the actual assigning the root process as the Leader of its respective group.

Each of the topology aware methods are discussed below:

1.1 Optimised MPI_Bcast

In this method, each of the Group Leaders and the communicators are first initialised, with the actual root of the MPI_Bcast made the Group leader of its respective group (call it Root Group). The Root Group Leader then makes an MPI_Bcast in the Group Leader only communicator (i.e. the communicator containing only the Group Leaders (at max. 6)). Thus each of the Groups (via the Group Leader) now have access to the broadcasted data. Each of the Group Leader then makes an MPI_Bcast for its own group. Thus instead of the pathway $root \xrightarrow[GlobalCommunicator]{MPI\_Bcast} node_i$, we follow the pathway $root \xrightarrow[GroupLeaderOnlyCommunicator]{MPI\_Bcast} Group\ Leader_{node_i} \xrightarrow[GroupCommunicator_{node_i}]{MPI\_Bcast} node_i$

Since this methods enables us to replace a majority of the inter-group communications (limiting them to at max. 6) with intra-group communications, there is a speedup in the overall operation. However, this speedup is overshadowed by the Intra-Group and Group Leader Only Communicators' creation time, leading to an overall decreased speed as compared to the default MPI_Bcast.

1.2 Optimised MPI_Gather

The optimised MPI_Gather Method is similar in principle to the optimised MPI_Bcast method. In this, after initialising the Group Leaders and Communicators, each of the Group Leader makes an MPI_Gather for its own group. The Root Group Leader then makes an MPI_Gatherv in the Group Leader only communicator (i.e. the communicator containing only the Group Leaders (at max. 6)). Thus instead of the pathway $root \xleftarrow[GlobalCommunicator]{MPI\_Gather} node_i$, we follow the pathway

$root \xleftarrow[GroupLeaderOnlyCommunicator]{MPI\_Gatherv} Group\ Leader_{node_i} \xleftarrow[GroupCommunicator_{node_i}]{MPI\_Gather} node_i$. Note that we use MPI_Gatherv (and not MPI_Gather) when accumulating data at the Root Group Leader since the number of process in each group (and therefore the size of data collected by each Group Leader) can be different. We also include each of the node ranks (with respect to the Global Communicator) as part of the data chunks being transferred. The placement of the gathered data chunks into the appropriate indices of the receive buffer at the Root Group Leader (and therefore the actual root) is then done using this additional rank information using a simple (linear time) traversal over the data and leveraging the rank information.

Since this methods enables us to replace a majority of the inter-group communications with intra-group communications, there is speedup in the overall operation. However, as in MPI_Bcast, this speedup is overshadowed by the Intra-Group and Group Leader Only Communicators' creation time, leading to an overall decreased speed as compared to the default MPI_Gather.

1.3 Optimised MPI_Reduce

The optimised MPI_Reduce Method is almost the same as the the optimised MPI_Gather method, except that we replace the data chunks with aggregrate (reduced) values. In this, after initialising the Group Leaders and Commuicators, each of the Group Leader then further makes an MPI_Reduce for its own group. The Root Group Leader then makes an MPI_Gather in the Group Leader only communicator (i.e. the communicator containing only the Group Leaders (at max. 6)). Thus instead of the pathway $root \xleftarrow[GlobalCommunicator]{MPI\_Reduce} node_i$, we follow the pathway $root \xleftarrow[GroupLeaderOnlyCommunicator]{MPI\_Reduce} Group\ Leader_{node_i} \xleftarrow[GroupCommunicator_{node_i}]{MPI\_Reduce} node_i$.

As before, the transmission speedup is overpowered by the Intra-Group and Group Leader Only Communicators' creation time, leading to an overall decreased speed as compared to the default MPI_Reduce.

1.4 Optimised MPI_Alltoallv

While the general approach of defining Groups, Group Leaders and making Inter and Intra Group Communicators remains the same, the code implementation for MPI_Alltoallv is quite different from the above three MPI calls since it involves much more bookkeeping of data (variable sized), their ranks and and sizes in order to obtain a "correct" output at the end on each processes' receive buffer.

We experimented with two different approaches, each of them being as follows:

1.4.1 Version 1

1. Pad each data chunk appropriately such that the data chunk to be sent to each process is of uniform size (N). This helps later on in reduced collective calls since we no longer have to bookkeep for data chunk sizes ahead of MPI calls. In order to help grouping the data appropriately at later stages, we also include the receiver rank and data chunk size as part of the transmitted data. Thus, if we were initially sending x doubles, x being variable, we will now send (N+2) doubles to each of the process.
2. Also, later on we would need the ranks of the processes present in the intragroup at the respective group leaders. To avoid a separate MPI_Gather call for ranks, we also appended the sender rank in the data sent by each process to the group leader. Therefore, the total size of the data sent by each of the processes to group leaders would be (1 + (N+2)*size) doubles.
3. Use MPI_Gather on each of the Group Leaders to accumulate the entire data needed to be sent/received by the respective group as a whole. With this single MPI_Gather call, the group leader gets the data, sendcounts and the ranks of the processes present in the group.
4. Extract away ranks of the processes present in the intragroup in a *ranks* buffer.
5. Use a MPI_AllGather call in the Group Leader Only Communicator on the rank buffer. This will help each group gather information about ranks present in other groups, which is needed to group the data appropriately.

6. Rearrange the gathered data chunks at the receiver buffer of each of the Group Leaders using the above rank information such that data chunks to be sent to same process are grouped together. Also the merged data chunks of processes of the same group are grouped together. This helps in sending different data chunks to be sent to the processes of a same group, by sending a single merged data chunk to the group leader of that group.
7. Invoke MPI_Alltoallv in the Group Leader Only Communicator. After this, each group gets the data chunks required by the final receiver buffer state of all of it's group members
8. Again rearrange the data chunks sent by different groups such that all data chunks to be sent to a single process are grouped together.
9. At each Group Leader, use MPI_Scatter to distribute and populate the data in the receive buffers of its (group) member processes.
10. Finally remove the padding embedded in the received data at each of the process, to extract away the contiguous data in the recvdata buffer and the sendcounts in sendcounts buffer.

Our Version 1 implementation for MPI_Alltoallv makes use of padding to avoid the bookkeeping required for sizes of data chunks in the MPI_Gather and MPI_Scatter operations at the Group Leaders. This leads to significant overhead time in some cases, specifically when value of D i.e. N is large. So even if the actual sendcount is less, sending and iterating over all N doubles everytime leads to very poor performance. Therefore, our this idea of using padding to reduce number of collective MPI_Calls didn't work. So instead of using padding, we made use of MPI_Gatherv and MPI_Scatterv and a clever bookkeeping strategy. The modified (and final) implementation is then as follows:

1.4.2 Version 2

1. Here, since we aren't padding the data, the leader process needs to know the sendcounts of each process in the intra-group. Also, the leader needs to know the ranks of processes present in the intragroup. So we append myrank of the process at the end of sendcounts array to merge them into a single mpi collective call
2. Use MPI_Gather in the intragroup to collect the sendcounts and ranks at the leader process of the intragroup.
3. From the sendcounts of each member process, calculate the total chunk size to be sent by each of the member process to the group leader and store it in the intra_rcounts array.
4. Use MPI_Gatherv to collect data of all member processes of a group at the leader of the group. Use intra_rcounts array created above as the receivecounts buffer at the leader process of the group.
5. Extract away ranks of the processes present in the intragroup in a *ranks* buffer.
6. Use a MPI_AllGather call in the Group Leader Only Communicator on the rank buffer. This will help each group gather information about ranks present in other groups, which is needed to group the data appropriately.
7. Rearrange the gathered data chunks at the receiver buffer of each of the Group Leaders using the above rank information such that data chunks to be sent to same process are grouped together. Also the merged data chunks of processes of the same group are grouped together. This helps in sending different data chunks to be sent to the processes of a same group, by sending a single merged data chunk to the group leader of that group.
8. Also count the total number of doubles to be sent to each process by the group. i.e. size of each of the data chunk for a process grouped above and store it in intra_totsize array. This gives us an array of sizes where each size is the size of the data chunk to be sent to an individual process by the given group.
9. Use MPI_Alltoallv call on intra_totsize buffer in Group Leader Only communicator so that each group has information about the data chunk size it will be receiving from other groups.
10. Use the above sizes calculated to construct the sendcounts and recievecounts at each of the group leader, which will be used in data communication among the groups.
11. Use MPI_Alltoallv call on the data buffer in Group leader only communicator so that each group recieves the final data it's members will be having in the end.

12. Again rearrange the data chunks sent by different groups such that all data chunks to be sent to a single process are grouped together. and also count the sizes of each chunk to be sent to individual processes.
13. Use MPI_Scatter to scatter the chunk size to be sent to each member process to the corresponding processes and store that in the recvcount buffer in the individual member processes.
14. Finally use MPI_Scatterv to communicate the individual data chunk to each of the member process.

2 Plots

Each of the MPI collective calls is done 5 times and we take consider only the average time (across the 5 times) for both the default and optimised versions as the performance metric.

The complete procedure is repeated for a total of 10 times for different configurations of P (number of nodes), ppn (Processes per node/core) and D (size of data). For MPI call, we plot the Execution Time against values of (P, ppn) for each value of D, and for both the default and optimised versions. The following plots represent the Performance of our proposed MPI call routines against the default versions:

2.1 Issue with CSE Server

We faced the following error when running the plot script on the cse server:

AttributeError: module 'seaborn' has no attribute 'catplot'

THe reason being that the seaborn version on the csews nodes is 0.8.0, while catplot is supported only by seaborn versions >= 0.11.0 (which supports catplot). As adviced by the instructor on Piazza, we had raised the issue on https://redmine.cse.iitk.ac.in/redmine, however, the issue had not been resolved by the time of submission.

It is essential that the issue be resolved for the end to end pipeline to work.

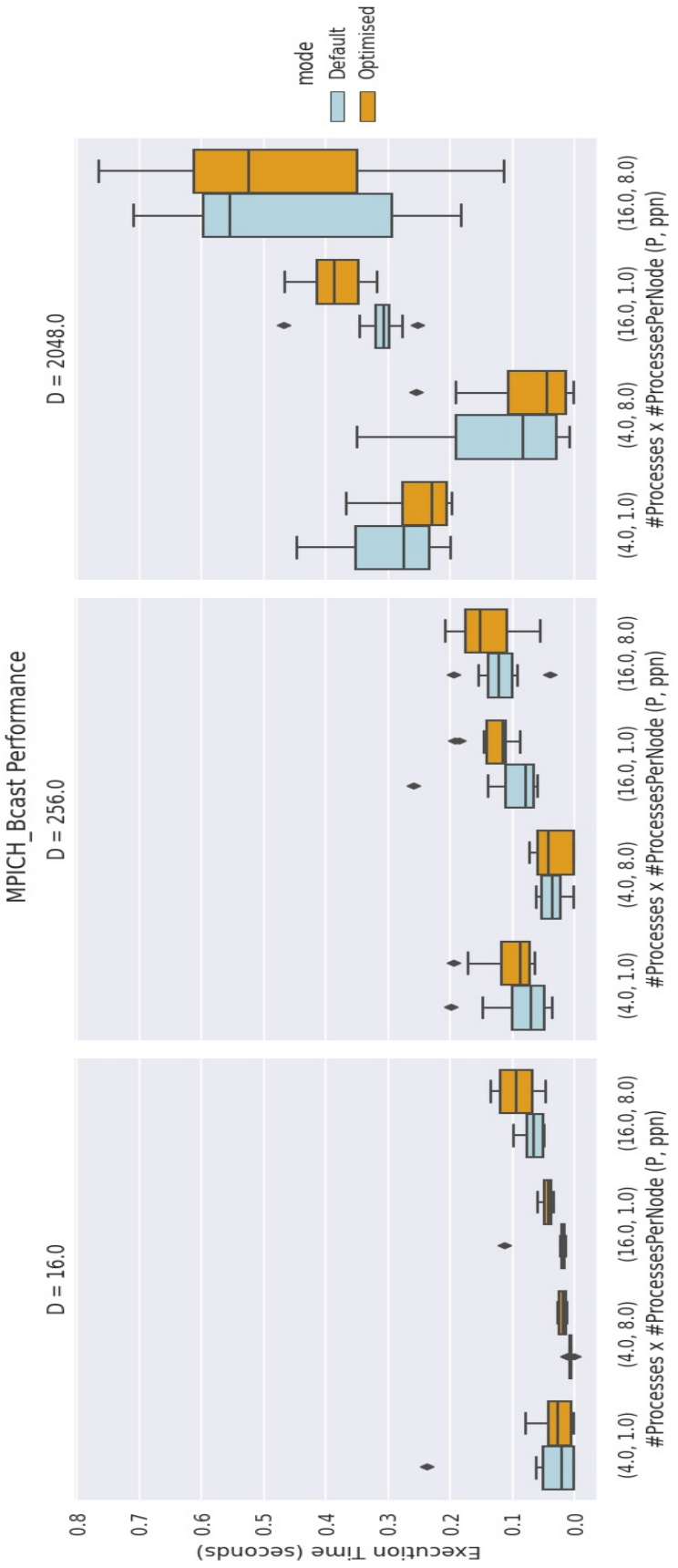

Figure 2: Plot for MPI_Bcast

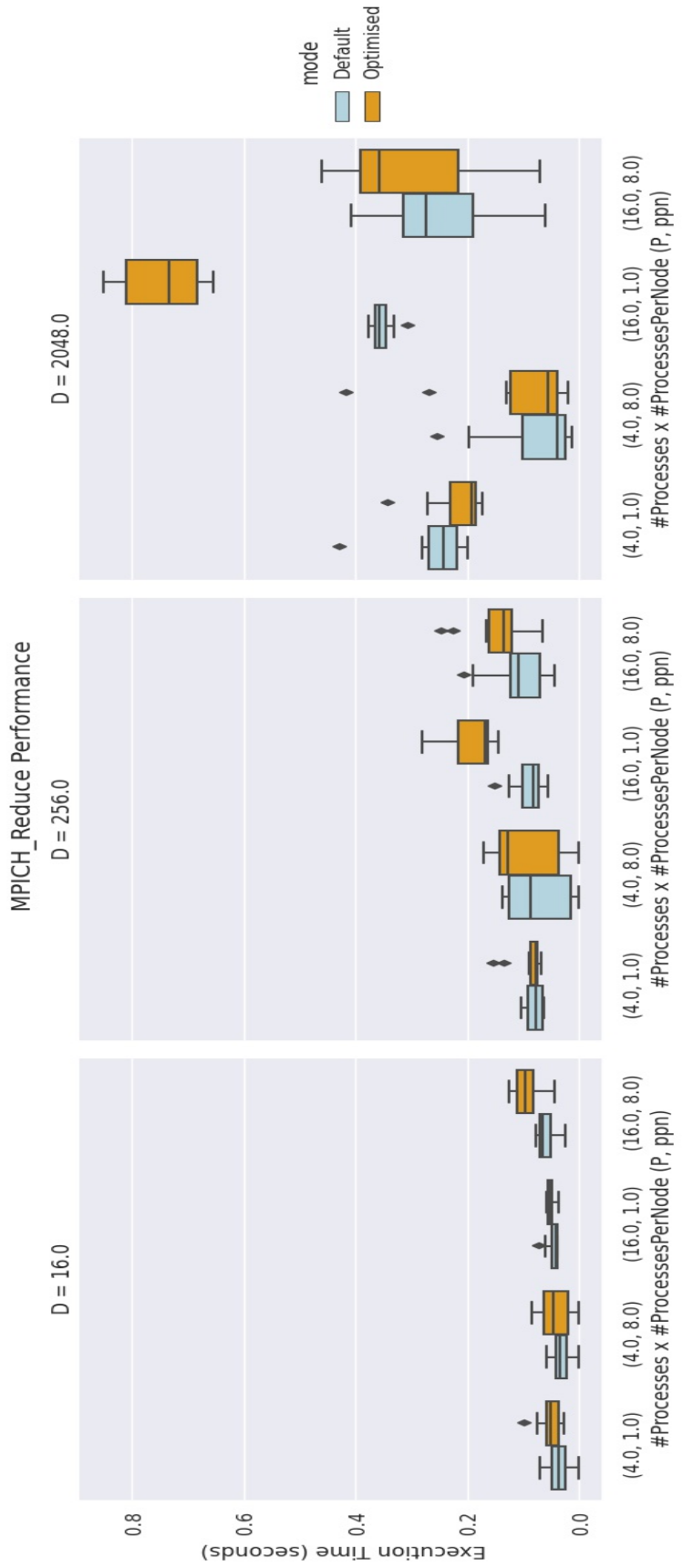

Figure 3: Plot for MPI_Reduce

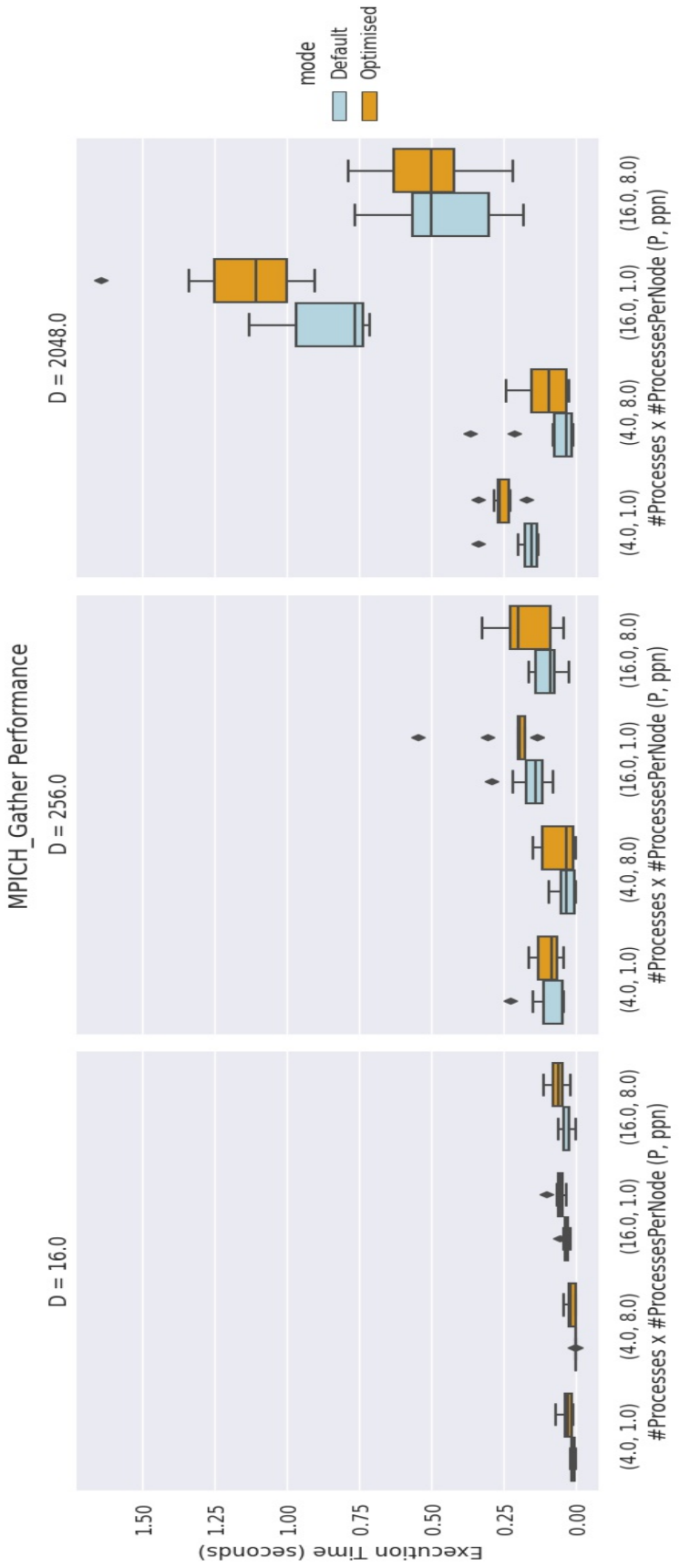

Figure 4: Plot for MPI_Gather

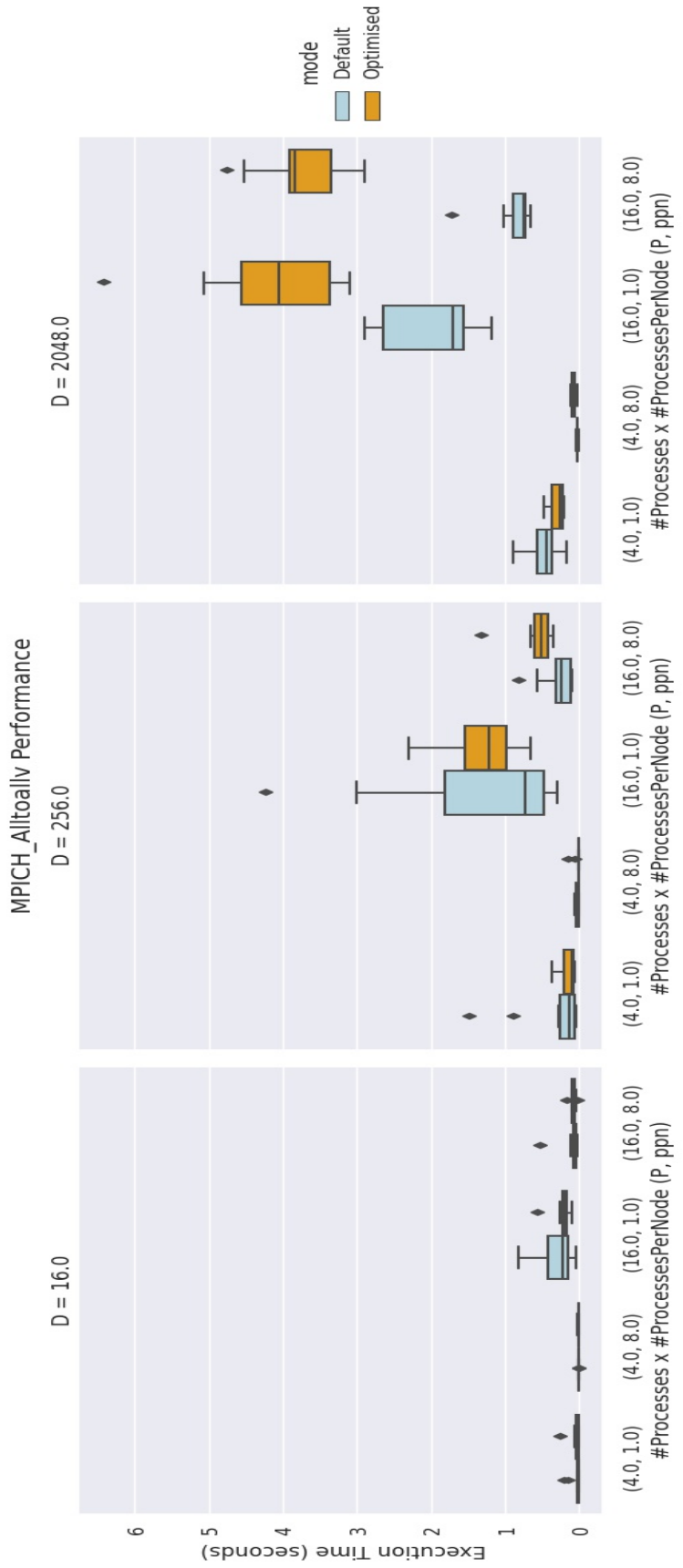

Figure 5: Plot for MPI_Alltoallv

2.2 Observations

From the plots, it can be concluded that the default versions of the MPI collective calls work a bit better than the optimised MPI collective calls, and the difference being more evident in the case of MPI_Alltoallv. As explained before, this behaviour is due to the time overhead in creating the Intra-Group and Group Leader Only Communicators. However, that if we exclude this communicator creation time from our analysis, which is a valid assumption assuming the Network Topology is not subject to change and since then the communicators would be required to be created only once (initialisation step), then our optimised collective calls seem to do much better than their default version counterparts. Note that even on including the communicator creation time, our optimised MPI_Bcast seems to outperform the default MPI_Bcast version, while the other optimised collective calls are quite competitive as compared to their default versions (since the gap in performance is not very large).

3 Submission

The directory named Assignment2 is uploaded on git, which contains the source file (src.c), Makefile, plot script(plot.py) and plots. For ease of execution, a job script is also provided, which can be executed as follows:

```
$ bash run.sh
```

The job script compiles the code, executes the complete code on the given configurations and then executes the plot script for plots. The data outputs are stored in a single file named as *data* in the order of execution specified in the assignment. A more readable form of the contents of *data* file is also printed on stdout, which is redirected into a file named as *output* in job script.

The plot script plot.py then generates the plots for each of the MPI calls, namely plot_Bcast.jpg, plot_Reduce.jpg, plot_Gather.jpg, plot_Alltoallv.jpg.

3.1 Experimental Setup

- Install the matplotlib python library by the command (if not already present):

  ```
  pip3 install matplotlib
  ```

- Install the seaborn (version >= 0.11.0) python library by the command (if not already present):

  ```
  pip3 install seaborn
  ```

- Install the pandas python library by the command (if not already present):

  ```
  pip3 install pandas
  ```

- Install the numpy python library by the command (if not already present):

  ```
  pip3 install numpy
  ```

3.2 Issues faced while Implementing the code

- As mentioned under the Plot Section, we require the seaborn version to be >= 0.11.0 for catplot to work. We have raised the issue with the server administrator to update the seaborn module installed on csewsX nodes.
- We found implementing alltoallv to be quite involving, mainly because of lots of book-keeping things to be done to send the correct data to correct place. Also there were lots of MPI Collective calls, suggesting that the performance of implemented code won't be good. To avoid that, we tried the idea of padding the blocks to make them constant sized. This helped in both neater algorithm as well as reduced number of collective calls. However, after implementing we found that this was very poor with respect to the default MPI_Alltoallv, especially for large value of D(Speedup obtained for D=2048 was 0.03), which is expected

as well since due to large value of D, the padding size will grow very large and lots of unnecessary data transfer will take plce. So, this padding idea didn't work. Finally we implemented the version2, where we didn't pad the data array, but reduced the number of MPi calls by merging several into one(See version2), which was found to be competitive with Default MPI_Alltoallv.

- Due to heavy load on cse servers, lots of times there were ssh error stopping the execution. We had to run the scripts several times to get one complete execution of run.sh.

References

[1] K. Kandalla et al. "Designing topology-aware collective communication algorithms for large scale InfiniBand clusters: Case studies with Scatter and Gather". In: *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 2010, pp. 1–8. DOI: 10.1109/IPDPSW.2010.5470853.

[2] Rajeev Thakur and William D. Gropp. "Improving the Performance of Collective Operations in MPICH". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Jack Dongarra, Domenico Laforenza, and Salvatore Orlando. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 257–267. ISBN: 978-3-540-39924-7.