CS633 Parallel Computing
Assignment-2

Shrey Bhatt, Roll no - 20111060
Saksham Jain, Roll no - 20111053

Overview of the problem

In the given problem, the objective was to perform optimization on the given collective calls using the knowledge of CSE cluster topology. MPICH library, in its default behaviour, has its own optimization but it does not make use of the process placement or network topology, for executing the calls. So, in the given problem, our objective was to make use of the same and experiment over the given configurations to analyze the behaviour of the performance of optimized and non-optimized version of collective calls.

Optimizations performed

The principle behind the working of the optimized version of collective calls is that we have selected a node-level leader process for each node. This leader process is the process with smallest rank on each node. In some collective calls, we have also extended the idea to create a group-level leader process by combining the node-level leader processes which are in the same network group and selecting the process with smallest rank among these node-leader processes to be group-level leader. So, it is basically a 2-level hierarchy of leaders among the processes spawned. This leader and group based communication required creation of some sub-communicators at the following levels.

1. Subcommunicator to communicate among the process residing in the same nodes.
2. Subcommunicator to communicate among the node-level leader processes.
3. Subcommunicator to communicate among the node-level leader processes, but also residing in the same network group.
4. Subcommunicator to communicate among the group-level leader processes.

Lets understand how each of these collective calls are implemented and the theoretical performance advantage as well. Also, please note that, the root process was always assumed as 0 and so it will always be a group-level and node-level leader and since leaders are selected based on minimum-rank criteria, every group-level leader is also a node-level leader.

MPI_Bcast

In this collective call, initially, root node, which is also a group-level leader broadcasts the data to other group-level leaders. Then, each of these group-level leaders will broadcast the data to node-level leaders belonging to their groups and finally, all the node-level leaders will broadcast the data among all process residing in their nodes. Also, we have taken care of the case where every node is in the same group or when all the processes are in the same node. Overall, the idea is that, the count of network messages will be reduced, since in the default case, process in one node might need to communicate with many process in other nodes but in the optimized case, network messaging between nodes happen through only one process and the message size in both remain same. The same also applies to inter-group communication. Thus, there should be an observed advantage over both, propagation and transmission time as well.

MPI_Reduce

The optimization in MPI_Reduce call does not use 2-level but just a single-level hierarchy. Each of the process performs reduction under a node-level group and the result of reduction is collected at node-level leader by setting it as root. Then, these node-level leaders perform a reduction amongst themselves and the result is collected under

root (0). Also, we considered another strategy to form a 2-level hierarchy for this call, but after experimentation, the results found were not better and so we restricted to a single-level hierarchical optimization. Under this optimization, same as before, not all processes in one node will pass its data to root, but only one process needs to but only the leader will pass the data.

MPI_Gather

In this collective call too, we have implemented a 2-level hierarchy. Initially, all the node-leader gather the data from the process in same node. These node leaders pass the data to their group-level leader through gather and finally the root process gather data from all the group-level leaders. Apart from this, we have also taken care of the cases where there is a single group or where the number of process per node is 1 to minimize the number of gather calls. Here too, we are trying to optimize over inter-group and inter-node communication time but since the data of every process needs to be sent, there is no advantage over transmission, but since the overall number of messages are reduced between groups and nodes, there should b advantage over propagation time.

MPI_Alltoallv

In the Alltoallv call, we have implemented a 1-level hierarchy. Initially, every process has data in the form as demonstrated by below matrix, where each entry is a vector and row represents data of a process. Consider that there are 8 processes and 2 process per nodes.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3}...a_{1,7} & a_{1,8} \\ a_{2,1} & a_{2,2} & a_{2,3}...a_{2,7} & a_{2,8} \\ ... & & & \\ a_{8,1} & a_{8,2} & a_{8,3}...a_{8,7} & a_{8,8} \end{bmatrix}$$

Each of the process will send their data in parts (one vector at a time) to their node-level leader $p = 8$ times through Gather and so the node-level leaders will have data in the form:

$$\begin{bmatrix} a_{1,1}a_{2,1} & a_{1,2}a_{2,2} & ... & a_{1,8}a_{2,8} \\ a_{3,1}a_{4,1} & a_{3,2}a_{4,2} & ... & a_{3,8}a_{4,8} \\ a_{5,1}a_{6,1} & a_{5,2}a_{6,2} & ... & a_{5,8}a_{6,8} \\ a_{7,1}a_{8,1} & a_{7,2}a_{8,2} & ... & a_{7,8}a_{8,8} \end{bmatrix}$$

This is a p-length vector and the node-leader will group ppn-entries together so there are entries equal to number of nodes looking as shown below.

$$\begin{bmatrix} a_{1,1}a_{2,1}a_{1,2}a_{2,2} & a_{1,3}a_{2,3}a_{1,4}a_{2,4} & ... & a_{1,7}a_{2,7}a_{1,8}a_{2,8} \\ a_{3,1}a_{4,1}a_{3,2}a_{4,2} & a_{3,3}a_{4,3}a_{3,4}a_{4,4} & ... & a_{3,7}a_{4,7}a_{3,8}a_{4,8} \\ a_{5,1}a_{6,1}a_{5,2}a_{6,2} & a_{5,3}a_{6,3}a_{5,4}a_{6,4} & ... & a_{5,7}a_{6,7}a_{5,8}a_{6,8} \\ a_{7,1}a_{8,1}a_{7,2}a_{8,2} & a_{7,3}a_{8,3}a_{7,4}a_{8,4} & ... & a_{7,7}a_{8,7}a_{7,8}a_{8,8} \end{bmatrix}$$

Then, the node-leaders will perform an Alltoallv amongst themselves based on these vector-groups and so, the resultant data will be:

$$\begin{bmatrix} a_{1,1}a_{2,1}a_{1,2}a_{2,2} & a_{3,1}a_{4,1}a_{3,2}a_{4,2} & ... & a_{7,1}a_{8,1}a_{7,2}a_{8,2} \\ a_{1,3}a_{2,3}a_{1,4}a_{2,4} & a_{3,3}a_{4,3}a_{3,4}a_{4,4} & ... & a_{3,7}a_{4,7}a_{3,8}a_{4,8} \\ a_{1,5}a_{2,5}a_{1,6}a_{2,6} & a_{3,5}a_{4,5}a_{3,6}a_{4,6} & ... & a_{7,5}a_{8,5}a_{7,6}a_{8,6} \\ a_{1,7}a_{2,7}a_{1,8}a_{2,8} & a_{3,7}a_{4,7}a_{3,8}a_{4,8} & ... & a_{7,7}a_{8,7}a_{7,8}a_{8,8} \end{bmatrix}$$

Finally every node leader will scatter the data P/ppn times with each times transmitting ppn vectors to each process in their node. Hence, the final vector for any node leader say 1, will look as below.

$$\begin{bmatrix} a_{1,1}a_{2,1} & a_{3,1}a_{4,1} & ... & a_{7,1}a_{8,1} \\ a_{1,2}a_{2,2} & a_{3,2}a_{4,2} & ... & a_{7,2}a_{8,2} \end{bmatrix}$$

However note that the code was complex considering to determine the number of counts and displacement for all vector based calls. So, due to the consideration of complexity and that these scatter and gather calls could be expensive, it would be more expensive to implement 2-level hierarchy. Here too, since size of overall data to be sent remains same, there should be no visible advantage over transmission time but since the communication of some inter-node communication is translated to intra-node, we should observe some reduction in propagation time. However, there are also scatter and gather calls, that could make the communication expensive.

Code Execution, Explanation and Issues faced

Code Execution

To execute the code, simply run the file *run.py* by using the command **python3 run.py**. Also, by default, the execution will be performed 10 times. In order to change the number of executions, one just needs to edit the variable *numexecs* in the files *run.py* and *create_plots.py*.

Code Explanation

The file **run.py** is the job script, **script.py** if the script for creating hostfile using the file **nodefile.txt**. The file **create_plots.py** is used to create the plot file and **src.c** contains the main source code.

In the main function, after getting host id of each process, we create sub-communicators for optimization. The sub-communicator **node_comm** is constructed based on **hostid** as color and used for intra-node communication. Similarly, the sub-communicator **nodel_comm** is used to create communicators of node leader which have rank 0 in node_comm. We also create communicators **nlg_comm** and **gl_comm** for communication among node-leaders in same group and communication among group-level leaders respectively. This will help in optimization since intra-node communication are faster than inter-node communication. The time for sub-communicator creation comes under **optimization overhead**. After this, we randomly initialised the values for each process and make the collective call for default and optimised version using the following functions -

1. **void MPI_Bcast_default(double* buf,long nume)** - This is the function which calls default version of MPI_Bcast.

2. **void MPI_Bcast_optimized(double* Bcast_opt,long nume)** - This is the function which executes optimized version of MPI_Bcast as explained in optimizations performed section.

3. **void MPI_Reduce_default(double* inp_buf,long nume,double* recv_buf)** - This is the function which calls default version of MPI_Reduce.

4. **void MPI_Reduce_optimized(double* Reduce_opt,long nume,double* ReduceRecv_opt)** - This is the function which executes optimized version of MPI_Reduce as explained in optimizations performed section.

Similarly, **MPI_Gather_default()** and **MPI_Gather_optimized()** are used for execution of default and optimized version of MPI_Gather respectively.
MPI_Alltoallv_default() and **MPI_Alltoallv_optimized()** are used for execution of default and optimized version of MPI_Alltoallv respectively.
There are several other functions like **getHostId(char* hostn,int len), getGroupId()** and **getRandomlyAllocatedSizes(long num_pr, long nume, long* arrs)**.

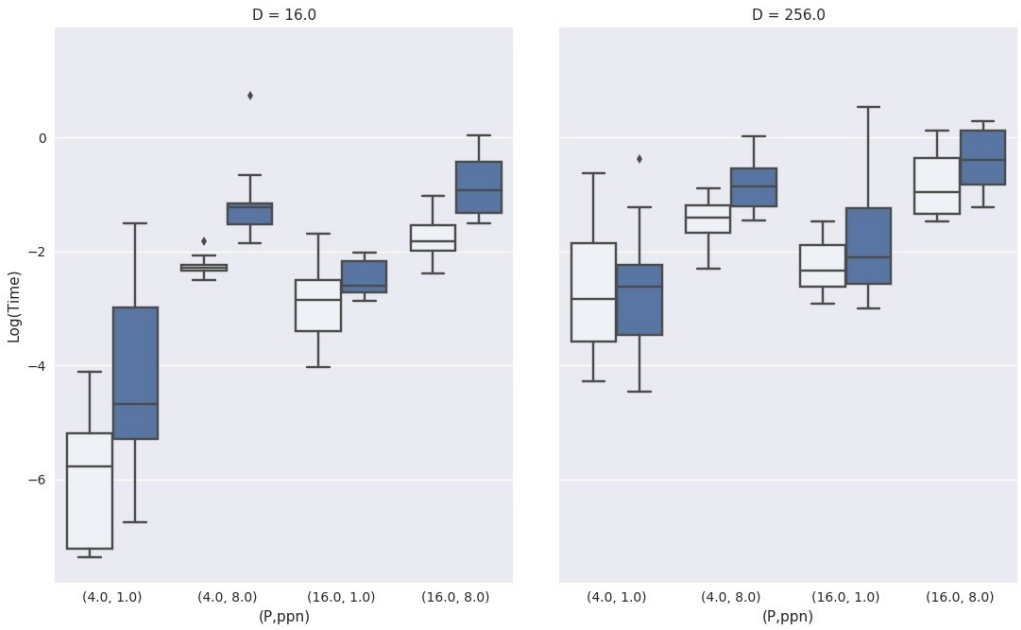
Issues faced

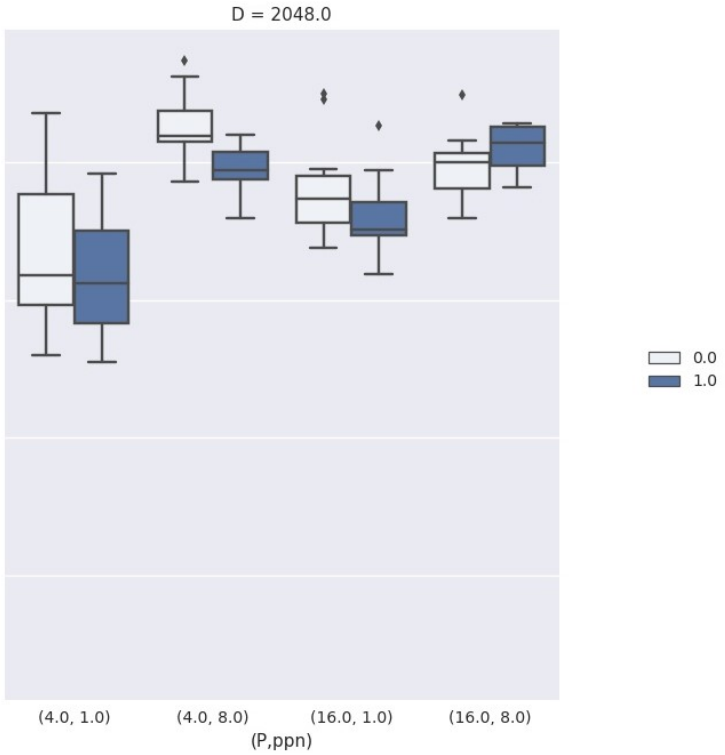
Since some nodes did not have enough memory required for holding data of multiple nodes, it sometimes caused error, especially in optimized Alltoallv case. We worked upon to handle memory management by freeing unnecessary memory and display required error messages, if encountered. Also, sometimes, the network did not respond during longer executions and so, after sometime during any point of MPI call, be it default or optimized version, process manager error was encountered.

Experimental Setup, Plot Observations and Analysis

The experiment was setup over csews cluster and 10 executions were performed for better data reliability. Although, the theoretical analysis encouraged better performance through optimization, the actual results were very different. Note that since the observed times varied over large magnitude, the obtained results were plotted over logarithmic scale. The results obtained for each case are described below.

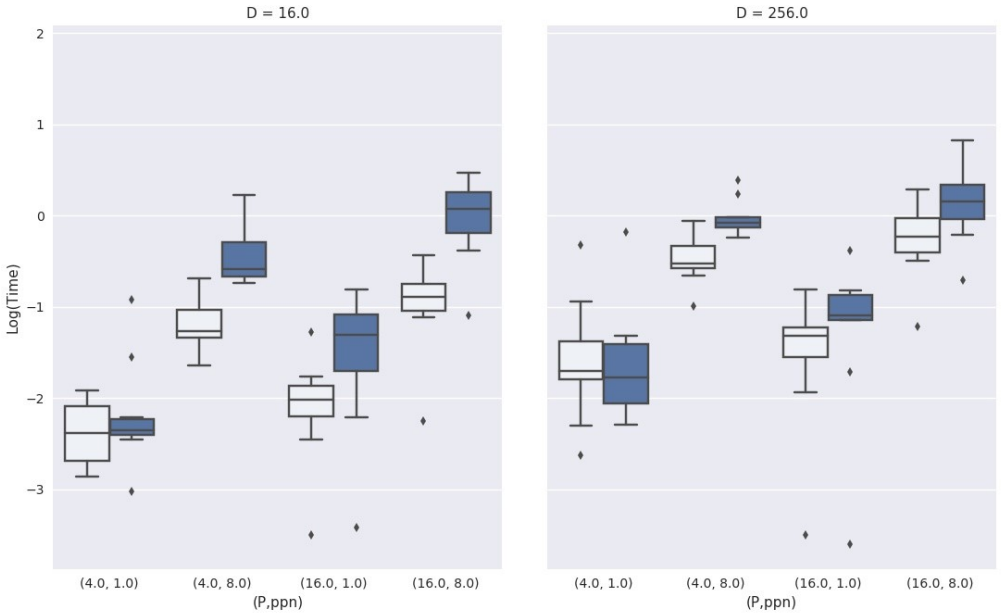
- **MPI_Bcast:** The plot (Image split due to width) describing the results of Bcast is shown below.

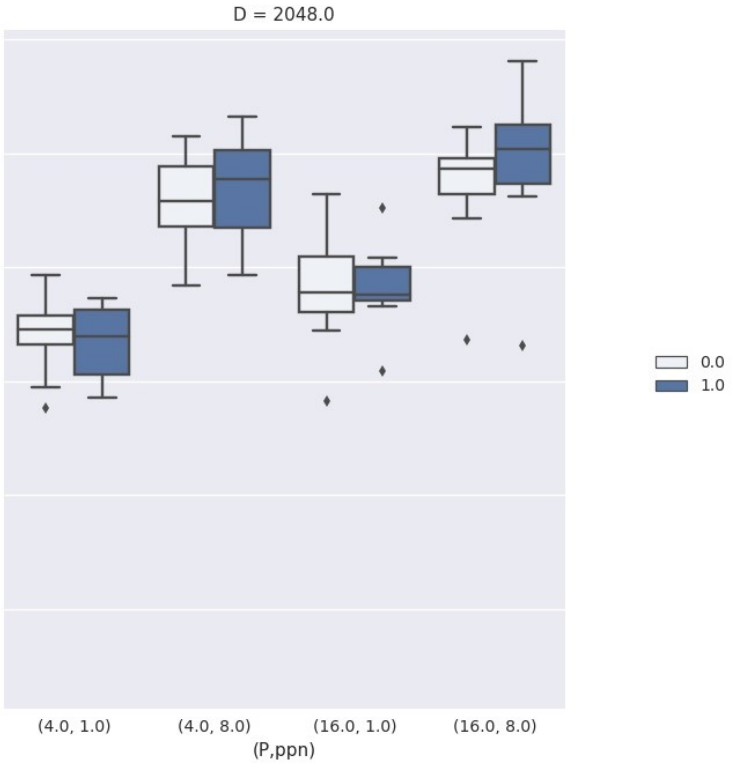




As observed here, for all cases in D=16.0, the optimized versions (in blue) perform worse than default ones (in white). Over D=256.0, the optimized version performs better than default one only at P=4,ppn=1 and for D=2048, except the last case, the optimized version perform better than the default one. One of the reason, why this is happening is because probably the overhead and intra-node communication involving numaNodes become an expensive factor which vanishes as we approach D=2048.

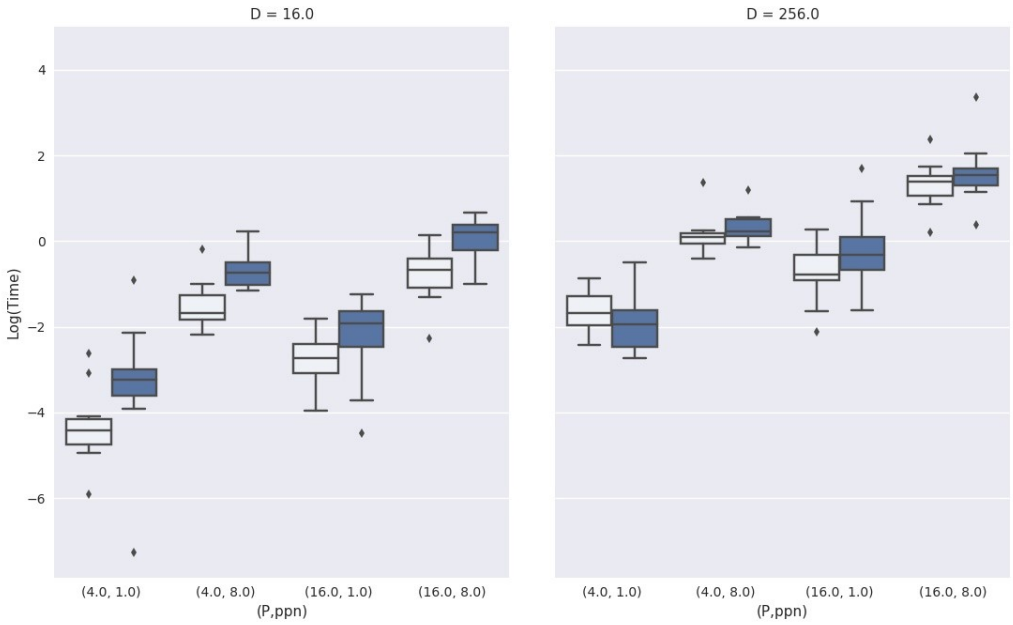
- **MPI_Reduce:** The plot describing the results of Reduce is as shown below.

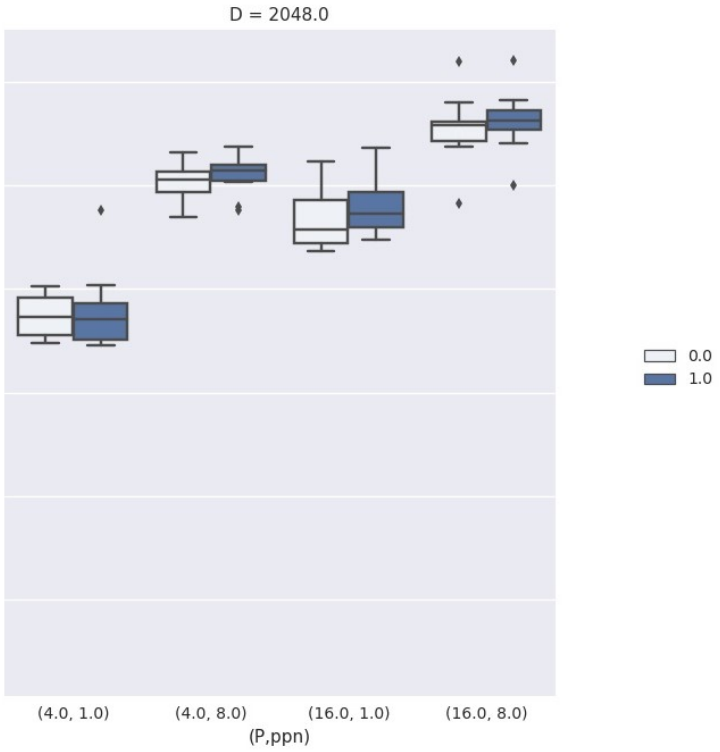




As observed here, for every D, the performance of optimized versions is better than defaults ones in cases where the number of process is low like (4,1) or (16,1) configuration. It seems like the expected advantage in propagation time has not been achieved as expected and the overhead time + node-wise computation could have made the overall computation expensive.

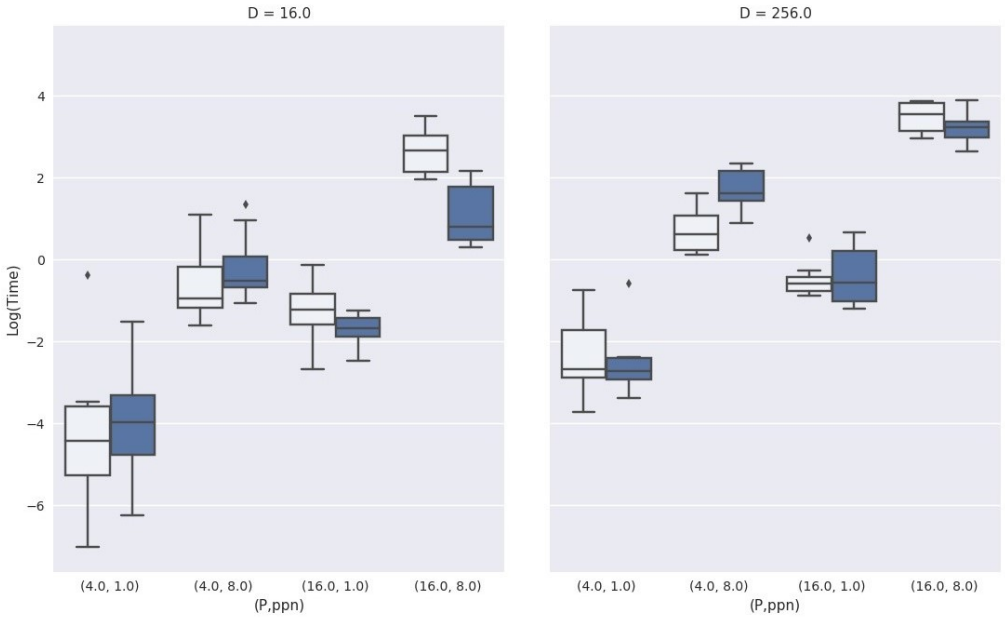
- **MPI_Gather:** The plot describing the results of Gather is as shown below.

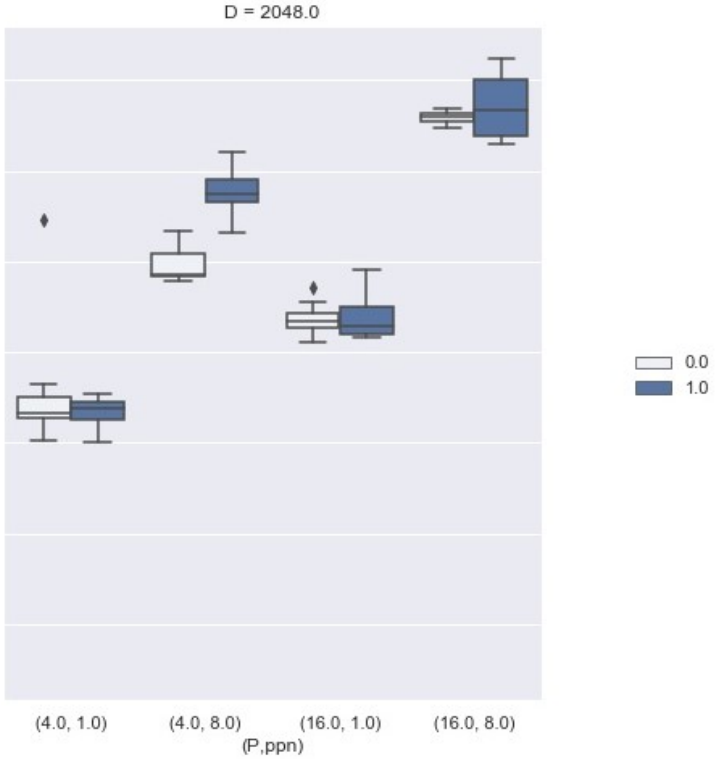




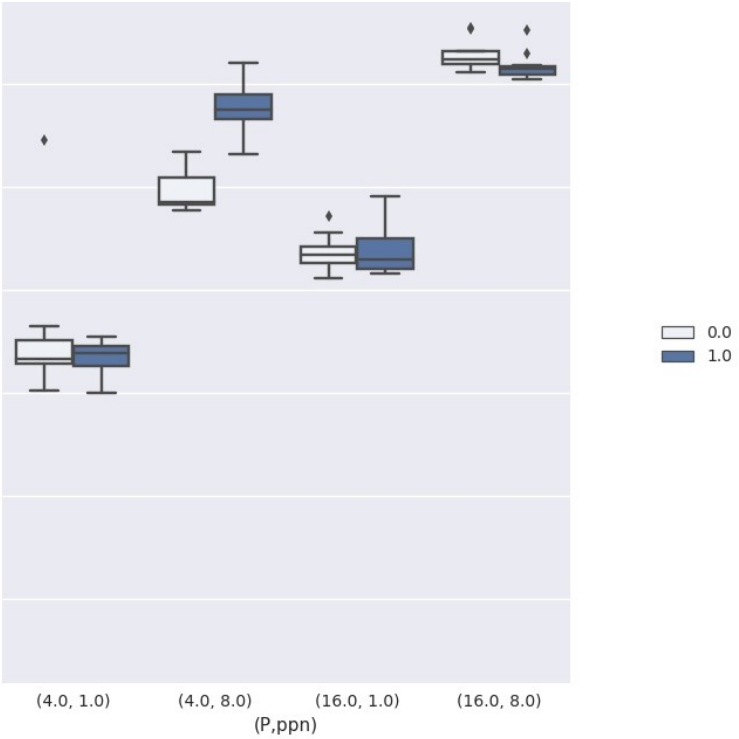
As observed, every gather performed more or less the same as that of the default one. However, for some cases like D=256, process = (4,1) and D=2048, process = (4,1), (4,8), the there was slight optimization observed while in some other cases like D=16, process = (4,1), (4,8), default one performs better. So, it can be said that the expected reduction in propagation time was not obtained as observed.

- **MPI_Alltoallv:** The plot describing the results of Alltoallv is as shown below.





As observed here, for many of the configurations, the optimized version performs better than the default ones and over large value of number of processes like 128, this reduction is easily observed. This reduction is also observed over some more cases. However, note that when we experimented with number of process=128 and D=2048, the time taken by both default and optimized version were very large and also optimized version had a very high variance covering the entire range of default time performance. Although, the medians were close, but many executions of optimized version took larger time than the default counterpart. We also experimented for number of process=128 and D=1024, since for D=2048, many times the execution used to halt due to heavy load. The plot for the same is as shown below.



As observed in the last box plots, the performance time was lower than default version. The overall performance analysis of the Alltoallv is not only contributed to the Alltoallv between node leaders in the optimized case, but also to scatterv and gatherv, due to which it seems that for many cases having high ppn and high data size, the time taken is also very high.