CS633: Assignment 2

Baldip Singh Bijlani(17807203), Paramveer Raol(170459)

March 29, 2021

1 Introduction

1.1 Optimization Strategies

We experimented with the following strategies for the Bcast, Reduce, Gather:

1.1.1 Node level optimization only

- In this case, we formed a communicator for all ranks belonging to the same node. This was done by calling MPI_Comm_split using the colour as the number of the processor, for example getting 25 from csews25. Then for each such group, there was a leader assigned(the process with the rank 0, with respect to the communicator created above) ie. for node 1 there is leader 1, node 2 there is leader 2, etc.
- Then a separate communicator was formed which consisted of these node level leader only which is used for inter-node communication. This was done by calling MPI_Comm_split on the MPI_COMM_WORLD with the color as 0 for the node leaders, and the color as MPI_UNDEFINED for the non-leaders, so that they are not a part of this group.

1.1.2 Group-level optimization only

- In this case, we formed a communicator for all the ranks belonging to the same group (group here is wrt to topology). Then for each group level communicator one group level leader was decided for eg for group 1 there is a leader 1 and so on.
- Then a separate communicator was formed which consisted of the group leaders only which is used for inter-group communication.

1.1.3 Node plus Group level optimization

- In this case, the first level of communicator consisted of all the ranks belonging to the same node. Then each of such a node-level communication group has a leader.
- A new communicator at the level of the group was formed which is basically consisted of node-level leaders only. This communicator is used for intra-group communications.

Each of the group-level communicators also had a leader which is one of the members of the node level leader of the same group.

- Then these group-level leaders were used to create a communicator which is then used for inter-group communication.

1.1.4 Rank placements optimization

- In this method instead of creating multiple communicators a single communicator is created which has the following properties:

  1. All the processes belonging to the node have contiguous ranks in the new communicator.
  2. All the processes belonging to the same group are such that they occupy a contiguous set of ranks.
  3. For example, if there are 8 process, 2 each on node1, node2, node3 and node 4. Also assume that nodes 1 and 2 are in the same group, and node 3 and node 4 are in another group. Now assume that the ranks given to the processes were as follows: node 1 process 1, node 3 process 1, node 2 process 1, node 4 process 1, node 1 process 2, node 3 process 2, node 2 process 2, node 4 process 2(1, 3, 2, 4, 1, 3, 2, 4). Then the new ranks in our communicator will be as follows: 1, 1, 2, 2, 3, 3, 4, 4.
  4. The intuition behind this is that in the algorithms such as binomial tree, in the final steps all the communication, will be intra-node. We will expand more on this in further sections.

- And then the collective call is called on the new communicator.

1.2 Optimization in the implementations

An implementation optimization worth noting is where we assigned the key of the root for any collective as (-1)*root_rank while calling the Comm_Split. This will make the rank of the root process to be 0 in any newly made subcommunicator. This optimization helps in the following manner as explained by an example of optimizing bcast.

Let's say that while optimizing bcast we have made a sub-comminicator for each of the processes in a single node. For example there are 4 nodes with 4 processes each. Then there are 4 sub-communicators with 4 processes each. Now in each of the 4 node-level subcommunicators, a process is made the node leader(The process with rank 0 with respect to the newly made node level subcommunicator). And another subcommunicator is made with all these node-leaders, i.e. a subcommunicator containing 4 processes each of which is a node leader of a node-level subcommunicator. Let us call this interNodeLeaderSubcommunicator. Now the optimization for bcast is that:

1. Send the buffer from the root process to it's node leader.

2. Send the rank of the root's node leader with respect to the interNodeLeaderSubcommunicator, to the root.

3. Broadcast this rank from the root to all the process.

4. Now all the process have the rank of the root's node leader with respect to the interNodeLeaderSubcommunicator. Now we broadcast the buffer from the root's node leader on the interNodeLeaderSubcommunicator(this is why we needed the rank broadcasted on the previous step as for calling broadcast you need the rank of the root process).

5. Finally we call broadcast from the node leaders in their respective node-level subcommunicators.

One can see in addition to the broadcast at the node-leader level and at the intra-node level done on step 4 and 5 respectively, we had to do 2 additional sends and an additional bcast on steps 1,2 and 3 respectively. Now if we perform the optimization written in the beginning of the section, the root process will be the node leader of it's node, as it will have the rank 0 as it's key will be negative while the key of the other process in the MPI_Comm_split call for making the node level subcommicators, will be non-negative. Also it's rank in the interNodeLeaderSubcommunicator will be 0 for the same reason mentioned above. So now we do not need to do step 1, above as the root itself is the node leader. Also step 2 and step 3 are no longer necessary as we can assume the rank of the root's node leader(the root itself after this optimization) with respect to the interNodeLeaderSubcommunicator to be 0.

An optimization with a similar flavour can be used in optimizing any collective with a root process(bcast, gather, scatter, etc.). Please note that in addition to removing the unnecessary communication calls this also makes the code less complex to understand and much shorter.

1.3 Hostfiles effect Execution Times

We also experimented with different host files. Basically in the normal host file generated by the helper function provided, all the contagious ranks are placed on the same node and the nodes belonging to the same group also have their ranks placed in such a manner that most of the the send/recvs in the algorithms for collectives are done on the intranode/intragroup level making them faster. While if the process placement was unfavourable it will slow down these communications. To explain this better will present and example. Consider the following two cases of the placements of ranks for 8 process, 4 running on node 2 and 4 running on node 1. In the first case ranks 0-3(inclusive) are placed on the node 1 and ranks 4-7(inclusive) are placed on node 2. In the second case ranks 0,3,5,6 are placed on node 1 and ranks 1,2,4,7 are placed on node 2. Now consider the execution of the binomial tree algorithm for bcast on both of these placements. See the figures below. Here in the placement 1, only the send in the step 1 is on a different node, rest are on the same node, and hence very fast. Whereas on placement 2, all the sends are internode, and hence slower. This is validated by our experiments, wherein we generate two hostfiles enforcing the placments mentioned below and the on the first placement bcast works approximately 3 times faster

the root rank and the data size being the same. A similar logic can be extended to varios other algorithms like recursive halving distance doubling for reduce, Vander Geijn for bcast, etc. Also this logic can be extended to placements of the ranks accross the groups. Hence we feel that the optimization theme of changing the ranks by creating a new communicator presented in the section 1.1.4 should help according to us.

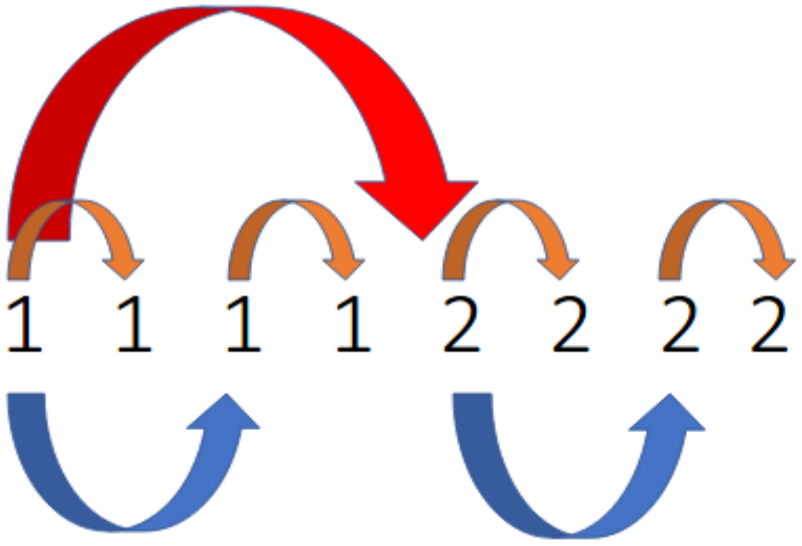

Figure 1: Placement 1, send only on the first step is to different node, rest are intra node

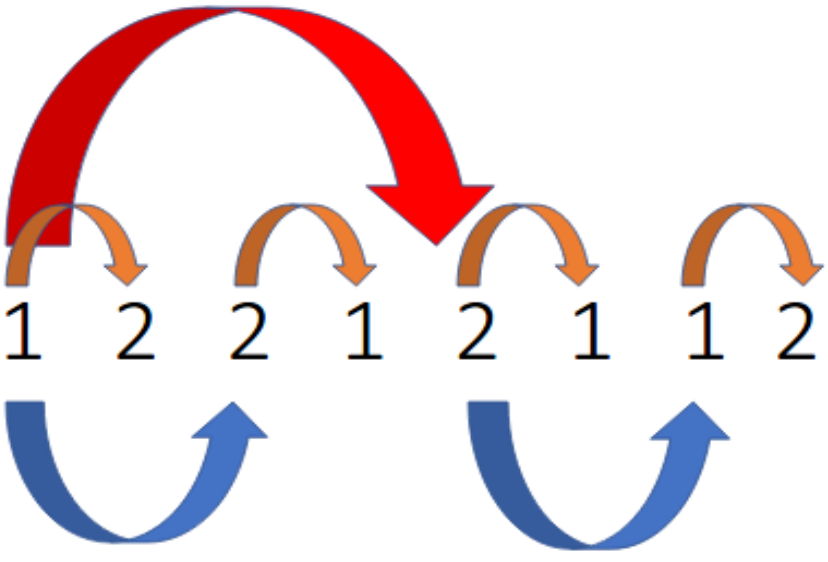

Figure 2: Placement 2, sends on all steps are to different nodes

So we looked at the cases where every process is placed in the most randomized manner possible:

For eg for the host file

```
csews1:2
csews31:2
csews70:2
```

The corresponding host file that we created was

```
csews1:1
csews31:1
csews70:1
csews1:1
csews70:1
csews31:1
```

For these type of host files, the topology-aware algorithm either changed the ranks(in 4) or did communications by exploiting the topology levels(1,2 and 3) hence ensuring that the communications were done in the most optimized manner possible. However, the default topology agnostic collective calls indeed performed worse as compared to the optimizations. But since the ppn is often given as the correct parameter ie. csews1:1, csews:1 is usually written as csews1:2 by the used we did not use these host files for our results. Instead, we just randomized the lines of the host-files generated which is quite a practical case when one does not know the topology of the network. Basically, the transformations that we made while generating results just shuffled the lines of the host file that is generated hence ensuring that the node in the same groups is not necessarily contiguous in the host file ie.

```
csews1:4
csews2:4
csews31:4
csews32:4
```

was randomly shuffled to :

```
csews2:4
csews31:4
csews1:4
csews32:4
```

Note : The type of host files described above(where the ppn can be non-one but lines are shuffled) were used to generate the results.

2 Bcast

Before going forward remember that the root will always be the leader at the topmost level ie. it will have 0 rank in any sub-group(due to implementation optimization mentioned in

section 1.2). A brief algorithm for various optimizations in the case of Bcast is provided below:

1. The node-level optimization pseudo code will be: **This implementation is in the function MPI_Bcast_opt_node_only in the file our_bcast.c**

   (a) First subcommicators are made at 2 levels as mentioned in section 1.1.1

   (b) Broadcast the buffer to the node leaders from the root using the subcommunicator of the node leaders.

   (c) Now the leaders have the buffer. Broadcast this buffer from the node leaders to the process on the same node using the intra node subcommunicators.

   We think that this optimization will help as it will limit the inter node communication to only one step, which is the broadcast between the node leaders. After which an intranode broadcast will be fast.

2. The group-level optimization pseudo code will be: **This implementation is in the function MPI_Bcast_opt_group_only in the file our_bcast.c**

   (a) First subcommicators are made at 2 levels as mentioned in section 1.1.2.

   (b) Broadcast the buffer to the group leaders from the root using the subcommunicator of the group leaders.

   (c) Now the group leaders have the buffer. Broadcast this buffer from the group leaders to the process on the same group using the intra group subcommunicators.

   We believe that optimization should for a reason similar the one mentioned in the node level optimization.

3. The node + group level optimization pseudo code will be. Please note that here the root of the broadcast will be both a node leader and a group leader due to optimization mentioned in section 1.2.**This implementation is in the function MPI_Bcast_opt_node_group_only in the file our_bcast.c**

   (a) Make the subcommunicators as mentioned in section 1.1.3.

   (b) Broadcast the buffer to the group leaders from the root using the subcommunicator of the group leaders.

   (c) Then each group leader will broadcast the buffer to the node leaders using the subcommunicators of the node leaders.

   (d) Then finally the node leaders will broadcast the buffer to the node ranks in the intra-node level communicator.

   We believe that this optimization will help as compared to previous two since this will minimize both inter-node and inter-group communication.

4. The rank optimization will be.**This implementation is in the function MPI_Bcast_opt_rank_only in the file our_bcast.c**

(a) Form a new group with following key given to the Comm_split, (10000*group_number + 100*node_number - comm_world_rank) .

(b) Note that the color is the same for all ranks in Comm_split, and node_number is 1 for csews1 and so on. The subtraction of comm_world rank is just done for distinct keys it is a small number and ensures that the rank level optimization constraints are satisfied for the new communicator formed.

(c) Then the broadcast is called from the root to all other processes using the new communicator.

5. A pipelined optimization. **this implementation is in the file Assignment2_final/optbcast_with_pipeline.c**

   (a) In this optimization we using a non blocking bcast both at the inter-group level and at the intra-group level, and did multiple bcast of the data by dividing it into chunks so that the broadcasts overlap at the inter-group and intra-group level. We tried turning on async progress, but this version performed considerably worse than the above 4, so we dropped it from further consideration.

We ran all the methods for all the configurations ie. different data size, ppn, nodes and then after observing the best method for each set of configuration, (A configuration is of form (processes, ppn, data size)) for multiple iterations. Based on the these results we decide that we should use the bcast in following manner:

```
if(ppn==1)
    if(numProcs/ppn > 13 and data_size >= 2048KB) )
        MPI_Bcast_opt_group_only
    else
        MPI_Bcast_opt_rank_only
else
    if(data_size < 2048KB)
        MPI_Bcast_opt_node_only
    else
        MPI_Bcast_opt_node_group
```

The results for this are:

1. We observed that there was always speed up in case of ppn==1 and for data size=2048KB.

2. There was slowdown just in case of ppn==8 and data-sizes 16KB and 256KB that is just for 4 cases.

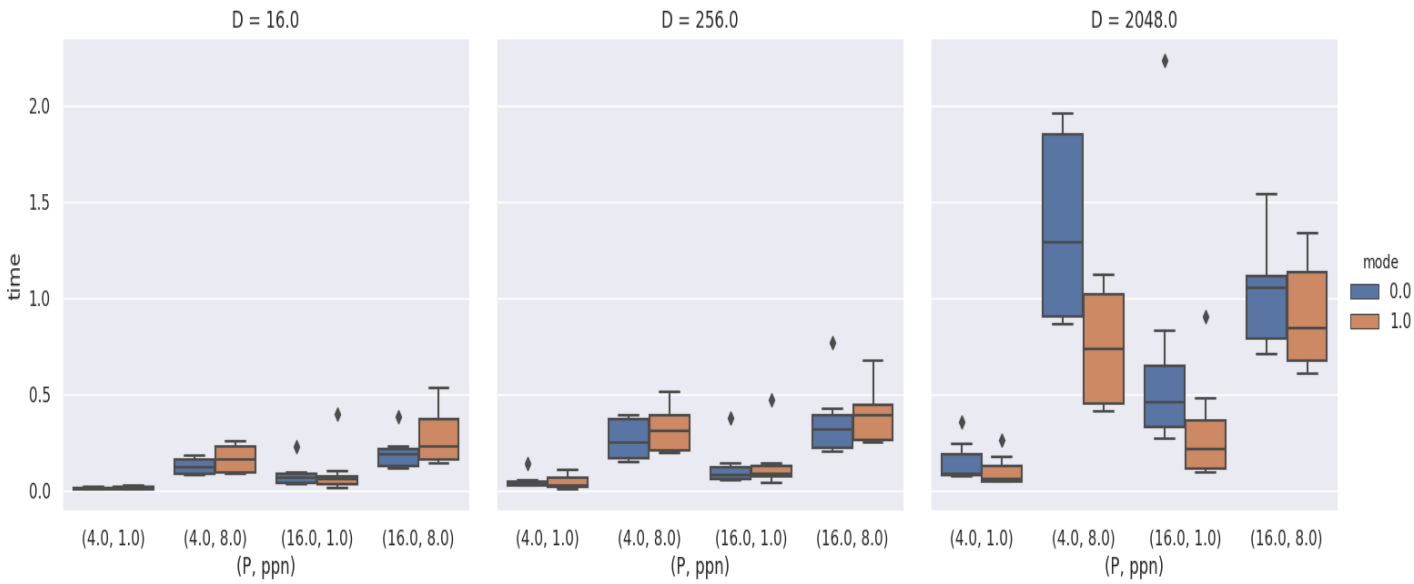

Figure 3: Bcast Results

3 Reduce

A brief idea of algorithms used for various levels of optimizations in the case of Reduce is provided below:

1. The node-level optimization pseudo code will be: **this implementation is in the file**
   our_reduce.c in the function MPI_Reduce_opt_node_only
   (a) Reduce all the processes in the same rank and set the root as the node level group's leader.
   (b) Use the reduced buffer of the node leaders, which is further reduced amongst themselves using the inter-node communicator (of node leaders), and set the root as the actual root.
   (c) From construction root will always be the node leader.
2. The group-level optimization pseudo code will be: **this implementation is in the file**
   our_reduce.c in the function MPI_Reduce_opt_group_only
   (a) Reduce all the processes in the same group and set the root as the group leader.
   (b) Use the reduced buffer of the group leaders, which is further reduced amongst themselves using the inter-group communicator (of group leaders) and set the root as the actual root.
   (c) From construction root will always be the group leader.

3. The group+node level optimization pseudo code will be: **this implementation is in the file**
   our_reduce.c in the function MPI_Reduce_opt_node_group
   (a) Reduce all the processes in the same rank and set the root as the node level group's leader.
   (b) Now use these reduced buffers for further reduction at the group level composed of node leaders and set the root at the corresponding group leader.
   (c) Now reduce the buffer received at the group leaders and reduce them using the inter-group communicator, and set the root as the actual root.
   (d) From construction root will always be the group leader and a node leader.
4. Rank optimization will be: **this implementation is in the file**
   our_reduce.c in the function MPI_Reduce_opt_rank_only
   (a) form a new group with the following key given to the Comm_split, (10000*group_number + 100*node_number - comm_world_rank) (except the actual root whose key is set to 0).
   (b) Note that the color is the same for all ranks in Comm_split, and node_number is 1 for csews1 and so on. The subtraction of comm_world rank is just done for distinct keys it is a small number and ensures that the rank level optimization constraints are satisfied for the new communicator formed.
   (c) Then the reduction is called from the root to all other processes using the new communicator.

We ran all the methods for all the configurations ie. different data size, ppn, nodes and then after observing the best method for each set of configuration, (A configuration is of form (processes, ppn, data size)) for multiple iterations. Based on the trailing results empirically we decide that we should use the bcast in following manner:

```
if(ppn > 1){
    if( data_size < 2048KB)
        MPI_Reduce_opt_node_only
    else
        MPI_Reduce_opt_node_group
}
else{
    if( data_size < 2048KB)
        MPI_Reduce_opt_rank_only
    else
        MPI_Reduce_opt_group_only
}
```

The reasoning for this was empirically which collective is performing best in which range of data sizes, number of processes , and ppn. The results are as follows: We can observe

that for less data size , there is almost equal performance to the default reduce. Whereas for the large and medium data size there is a clear speedup.

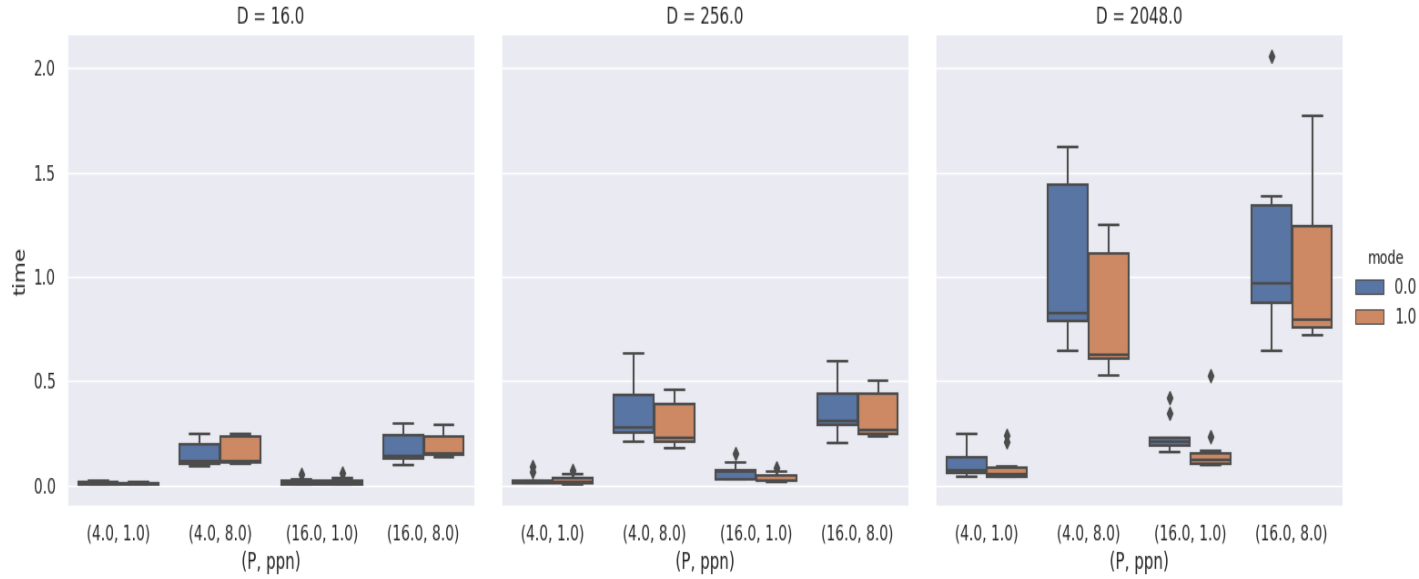

Figure 4: Reduce

4 Gather

A brief description of the algorithm used for various optimization levels is provided below:

1. The node-level optimization:
   (a) First gather all the data at the node-leader for the ranks in the node level communicator.
   (b) Then used the inter-node level communicator to gather all the data collected by the node leaders at the root.
   (c) Gather of rank positions of the buffer was also done so now un-shuffle the buffer's data using the rank positions gathered.
2. The group-level optimization:
   (a) First gather all the data at the group-leader for the ranks in the group level communicator.
   (b) Then used the inter-group level communicator to gather all the data collected by the group leaders at the root.
   (c) Gather of rank positions of the buffer was also done so now un-shuffle the buffer's data using the rank positions gathered.

3. The node+group level optimization:

   (a) First gather all the data at the node-leader for the ranks in the node level communicator.

   (b) Now gather the buffers at the node leaders at the group level to the group leaders.

   (c) Now gather the data from the group leaders to the root and unshuffle the data at the root.

4. Rank optimization will be:

   (a) form a new group with the following key given to the Comm_split, (10000*group_number + 100*node_number - comm_world_rank) (except the actual root whose key is set to 0).

   (b) Note that the color is the same for all ranks in Comm_split, and node_number is 1 for csews1 and so on. The subtraction of comm_world rank is just done for distinct keys it is a small number and ensures that the rank level optimization constraints are satisfied for the new communicator formed.

   (c) Then the gather is called from the root to all other processes using the new communicator. After that, the de-shuffling of the buffer is done.

Note: We have also written implementations where the number of nodes in a group is variable however after reading the post that we could take ppn as the input we assumed that the nodes per group could also be assumed constant as otherwise extra gathers of vector form have to be done to get data in various groups.
We ran all the methods for all the configurations ie. different data sizes, ppn, nodes, and then after observing the best method for each set of configurations, (A configuration is of the form (processes, ppn, data size)) for multiple iterations. Based on the following results we decide that we should use the gather in the following manner:

```
if(numProcs/ppn > 13)
    MPI_Gather_opt_group_only
else
    MPI_Gather_opt_rank_only
```

The results are as follows:

1. For the processes with ppn==8 there was either comparable performance or a consistent speed up. So was the case for all the cases of largest data size ie 2048KB.

2. So the only configurations where there seems to be a bit slowdown were of ppn==1 and data size less than 2048KB which implies the slowdown occurs for just 4 cases.

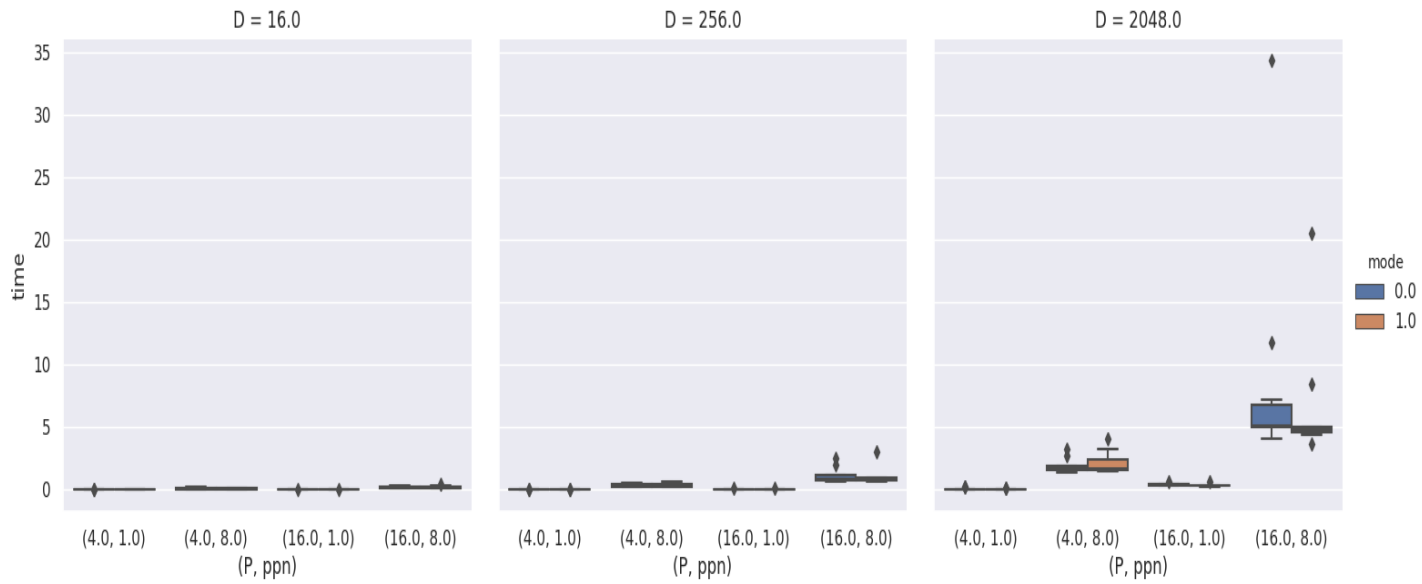

Figure 5: Gather

5 Alltoallv

A brief description of the algorithm used for various optimization levels is provided below:

1. **Implementation using gathervs, alltoall and scattervs.this implementation is in the file Assignment2/our_alltoallv.c**. The idea of this algorithm is as follows that first gather the data to be sent of every process at it's node leader. Then we do an alltoallv only among the node leaders of the data accumulated in the previous step. Now after this step each leader will have the data that is to be received by all the processes on it's node. Now we will scatter the data the all the process in the node from the node leader.

   (a) First, we create a rearranged communicator with all the process, so that the ranks are such that the processes belonging to the same node are contiguous, as mentioned in the section 1.1.4. Call this comm_world_new. Creating this new communicator is helpful for implementation, as now we can assume that processes belonging to the same nodes are placed contiguously.

   (b) Then we create subcommunicators for processes on each node by splitting the above communicator. We designate a node leader process for every node and make a subcommunicator out of the node leaders. This is akin to what is mentioned in the section 1.1.2, with the slight difference that instead of splitting MPI_COMM_World we are splitting comm_world_new.

   (c) Gather the sendcounts array of every process on its node leader process by a gather on the intra-node subcommicator

   (d) Gather the recvcounts array of every process on its node leader process by a gather

(e) Now for every process p, we gather the data that every other process i has to send to p at i's node leader. This will be done using a sequence of gatherv's(one for every process p) called using the intra node level communicator, the counts and displacements of which are found using the sendcounts array of every process in that node which had been sent to the node leader in the previous process.

(f) At the end of the previous step every node leader will have an buffer, which contains the data that has to be sent from every process in the node, to every other process in the global communicator. Let us say that there are 5 processes in total, and this node has ranks 0,1,2, and the 0 is the node leader. Then 0 will now have a buffer, which will look as follows:
$0->0; 1->0; 2->0; 0->1; 1->1; 2->1; 0->2; 1->2; 2->2; 0->3; 1->3; 2->3; 0->4; 1->4; 2->4$, where $i->j$ represents the data to be sent to the j^{th} process by the i^{th} process. Also assume that ranks 3, and 4 are in a different node with 3 being the node leader, then the buffer on 3 will look like: $3->0; 4->0; 3->1; 4->1; 3->2; 4->2; 3->3; 4->3; 3->4; 4->4$

(g) Now we call an alltoallv in the subcommunicators of the node leaders, to accumulate the data to be received by the process in the node of that node leader. The send buffer will be the ones described in the previous step. The counts and displacement for the send and recv buffers can be found using the sendcount and recvcount arrays at the node leaders. Basically the sendCount[i] will become the count of the data to be sent from the process of this node to the processes in the node of the node leader with rank i. Recvcounts for this all to allv call can be found similarly.

(h) At the end of the alltoallv call in the previous step, in the representative example process 0 will have a buffer looking as follows: $0->0; 1->0; 2->0; 0->1; 1->1; 2->1; 0->2; 1->2; 2->2; 3->0; 4->0; 3->1; 4->1; 3->2; 4->2$, and process 3 will have a buffer looking as follows:$0->3; 1->3; 2->3; 0->4; 1->4; 2->4; 3->3; 4->3; 3->4; 4->4$.

(i) Now, as you can see that the receive buffer in the previous step at the node leader contains the data that was to be received by the processes on it's node by all the processes. Now we will scatter the data to the corresponding node process-wise, that is first the data sent by process 0 is scattered to the corresponding process, i.e. $0->0$ is sent to 0, $0->1$ is sent to 1 $0->2$ is sent to 2(Note that all these sends are done simultaneously via a scatter call). This is done for all 0,1,2,3,4. The calculation of displacements is a bit non-trivial. The reader can refer to lines 240-249 of the file our_alltoallv.c in the submission folder.

(j) By some simple bookeeping no rearrangment is required by giving recvCount and recvBuf address intelligenty in the scatter call.

2. Implementing Alltoallv as a sequence of Gathervs for every process(**this implementation is in the file Assignment2/multiple_gathers_all_to_allv.c**)

(a) First, we create a rearranged communicator with all the process, so that the

ranks are such that the processes belonging to the same node are contiguous, as mentioned in the section 1.1.4. Call this comm_world_new. Creating this new communicator is helpful for implementation, as now we can assume that processes belonging to the same nodes are placed contiguously.

(b) Then we create subcommunicators for processes on each node by splitting the above communicator. We designate a node leader process for every node and make a subcommunicator out of the node leaders. This is akin to what is mentioned in the section 1.1.2, with the slight difference that instead of splitting MPI_COMM_World we are splitting comm_world_new.

(c) Gather the sendcounts array of every process on its node leader process by a gather on the intra-node subcommicator

(d) Gather the recvcounts array of every process on its node leader process by a gather

(e) for every process p do the following:

i. Gather the data that every process was going to send to the process p on its node leader using gatherv . Now since we have sent the sendcounts for every process to the node leader, we can find the displacements, counts to be used in this gatherv can be found on the node-leader process.

ii. Gather the data from all the node leaders in the node leader of process p using gatherv on the subcommunicator of the node leaders. The counts of the data coming to this node leader from each node, can be found from the recv counts of the process p.

iii. Now all the data from process p is on it's node leader. We now send it to the process p.

(f) When the loop terminated every process has all the data that it needed to receive. But this data is assuming that the processes are in order, as a final step we need to rearrange this data, using a map from the old ranks to the new ranks of every process, which is shared using MPI_allgather.

Note: Our Implementation has not assumed that the process belonging to the same nodes will be given contiguous ranks, and neither do we assume that every node will have the same number of processes.
We ran all the methods for all the configurations ie. different data sizes, ppn, nodes, and then after observing the best method for each set of configurations, (A configuration is of the form (processes, ppn, data size)) for multiple iterations.

5.1 Observations

1. We observed that the first optimization beats the second one in most of the runs. This can be attributed to the fact that in the second optimizations we are calling an inter-node gather v for each of the process, while in the first on the inter-node level, only a single alltoallv is called albeit for more data.

2. when ppn is 1,2 or 3, The default all to all beats both the cases as our optimizations are done only on the node level, not on the group level, and hence is only profitable when there are many processes on a node.

3. Lastly we observe that when there are a lot of processes and the data size D is 2048 KB or 256KB, the malloc functions fails in the optimized implementation, since in the node leader the data from all the process is accumulated which can go over 1.5 GB, due which either these implementation runs very slow(since the processes has a lot of data) or they crash.

4. When ppn is high(greather than or equal to 4) and the data is in a moderate range like 2KB - 500KB our implementation conclusively beats the default all to all. At an average it is approximately 3 times faster.

Based on the trailing observations we decide that we should use the alltoallv in the following manner:

```
if(ppn < 4){
   standard_alltoallv
}
else{
    if(count* numProcs < 2000000){
    //each leader does not have a lot of data
     my_optimized_alltoallv(first implementation above)
    else{
     standard_alltoallv
    }
}
```

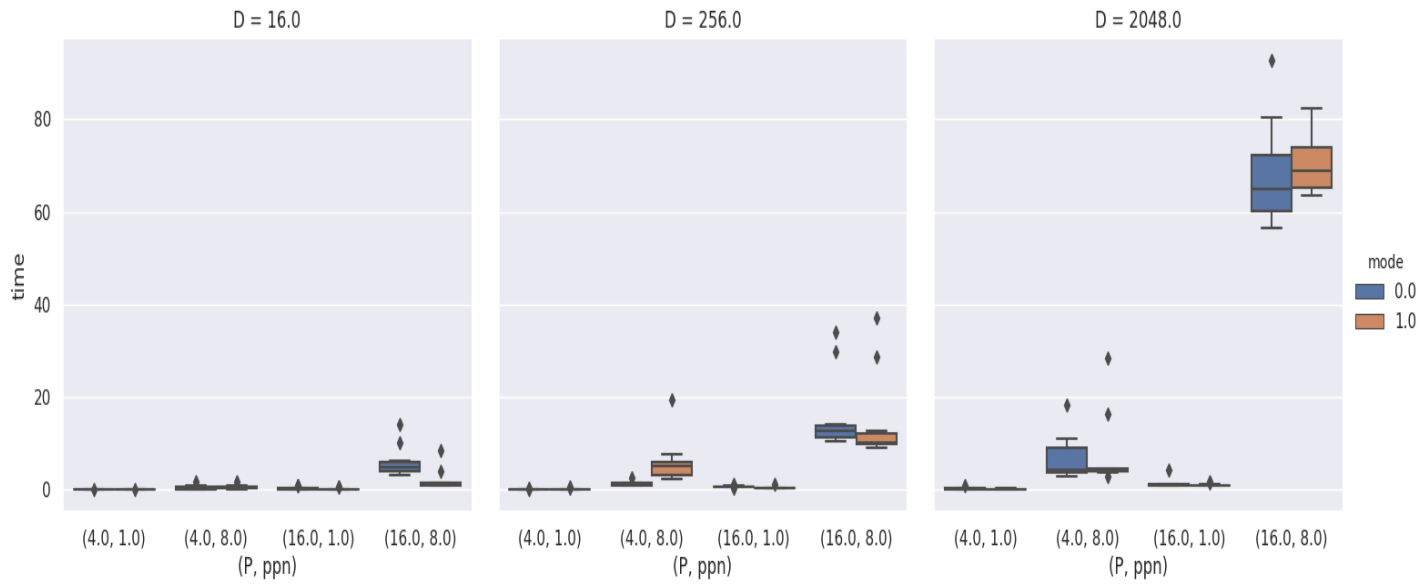

Figure 6: Alltoallv

6 Issues faced

The primary issue was that in all to all the way we had implemented all the data of the process of a node was being accumulated at the node-leader due to which even malloc was failing and returning null. We faced some out of memory errors as well.

7 Correctness Check

For checking the correctness of the collectives, the MPI standard collective and our own topology aware collective were ran on the same data and after the execution the values of the resultant data were matched on the correct set of nodes. That is root node in case of Gather, Reduce and all the node in case of Bcast(execpt the root node) and Alltoallv. This test was done for several times on all the configuration ie the node, ppn, data size etc.

8 Execution Instructions

1. Run the script to generate the results using the command **python3 run.py**. A new file named **data** will be generated after the script is completed. This will take quite a large amount of time to execute. The output in the *data* file is of form

```
bcast normal avg time
bcast optimized avg time
reduce normal avg time
```

```
reduce optimized avg time
gather normal avg time
gather optimized avg time
alltoallv normal avg time
alltoallv optimized avg time
```

for each configuration so total number of lines in the *data* file will be 8*120 = 960.

2. Then to generate the plots run the command **python3 plot.py**. The new files generated will be **"plot_Bcast.jpg", "plot_Reduce.jpg", "plot_Gather.jpg", "plot_Alltoallv.jpg"**.