CS633: Parallel Computing Assignment 2

Aditya Jain (20111004) | Mohit Kumar (20111034)
adityaj20@iitk.ac.in | mohitkumar20@iitk.ac.in

Indian Institute of Technology, Kanpur— March 26, 2021

1 Implementation Details

In our lab, star topology is followed where there are four groups. Intra-group distance is 2 and inter-group distance is 4. To obtain the speedup, we made use of the topology awareness in our programs. We created two communicators named- *Local* and *Master*.

- **Local Communicator** - It contains all the ranks that are hosted/launched on the same group of processor.
- **Master Communicator** - One rank of each local communicator is assigned as master rank. This communicator contains these master ranks.

In our MPI_Init we added the code for creation of these two communicators. So once these two communicators are created, we can call our newly created optimized collective calls as many times without including the time of their creation.

The logic of our algorithm is that all ranks that are hosted on the same group of network topology, first communicates their result to the master rank of that processor and then all master rank communicates among themselves. Master ranks then propogate the final results to all the nodes in its local communicator. We are getting a speedup because the new algorithm is generating less internode messages as only leader nodes are communicating internodes whereas in the original MPI algorithms all nodes are communicating internodes. Also, there are less number of messages receiving at a time to the central switch of star topology, hence reducing the bottleneck at the switch.

2 Bcast Optimized algorithm

Syntax of the function
*void new_bcast(double *data, int root);*

data - element that is to be broadcasted
root - rank of the process which is broadcasting the data.

Algorithm

1. The process with rank equal to root will first send its data through MPI_Send() to leader process of the group that root process belongs to.

2. After receiving the data from root process, leader process will broadcast the data among all the leaders present in the communicator *Master* through MPI_Bcast() over the Master communicator. So after this broadcast all leader processes will have the data element with them.

3. All leader processes will boradcast the data among the processes of their respective groups using MPI_Bcast() over the local communicator.

Analysis

Since we are creating two communicators at the time of MPI_Init, overhead of our algorithm is added only one time. So our algorithm gives better performance when collective is called multiple times. Here we have taken to cases for analysis.

- Case 1: Collective is called only once
- Case 2: Collective is called 5 times

Case 1
Here we are comparing our algorithm with the default algorithm where Bcast call is made only once (i.e. only one collective call among all the collectives) in the entire program.

$$speedup = \frac{t_{default}}{t_c + t_{opti}}$$

where,
$t_{default}$ = time of default MPI_Bcast() call.
t_c = time to create the Master and Local communicators.
t_{opti} = time for optimized broadcast algorithm.

Data Size	**#Nodes : ppn**			
	4 : 1	4 : 8	16 : 1	16 : 8
16KB	0.35	0.15	0.36	9.33
256KB	0.91	0.88	0.96	2.70
2048KB	1.13	1.71	1.84	0.68

Table 1: Speedup for Bcast Case 1

From Table 1, we can notice that there are speedup in some cases and in some there is not. The overhead of creating the two communicators and a MPI_Send() call is affecting the performance in the cases where there are few number of nodes and data size is less.

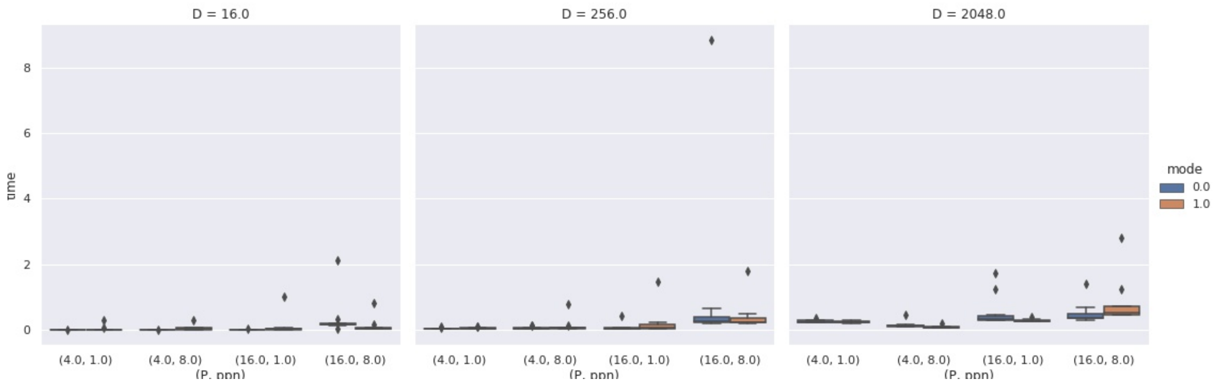

Figure 1: Bcast Case 1 plot

Case 2
Here we are performing broacast 5 times. This case gives better performance as the time for the creation of communicator is added only once and rest of the 4 times only time of collective call is included to calculate the speedup.

$$speedup = \frac{5 * t_{default}}{t_c + 5 * t_{opti}}$$

Data Size	**#Nodes : ppn**			
	4 : 1	4 : 8	16 : 1	16 : 8
16KB	0.96	1.28	1.65	1.83
256KB	1.09	1.40	1.39	1.48
2048KB	1.24	2.25	1.03	1.49

Table 2: Speedup for Bcast Case 2

From Table 2, we can conclude that our new algorithm is working for most of the configurations. It is not only working when number of processes is equal to 4 and data size is equal to 16 KB.

So we edited our new algorithm to perform default MPI_Bcast when data size is equal to 16KB and number of pocesses is equal to 4.

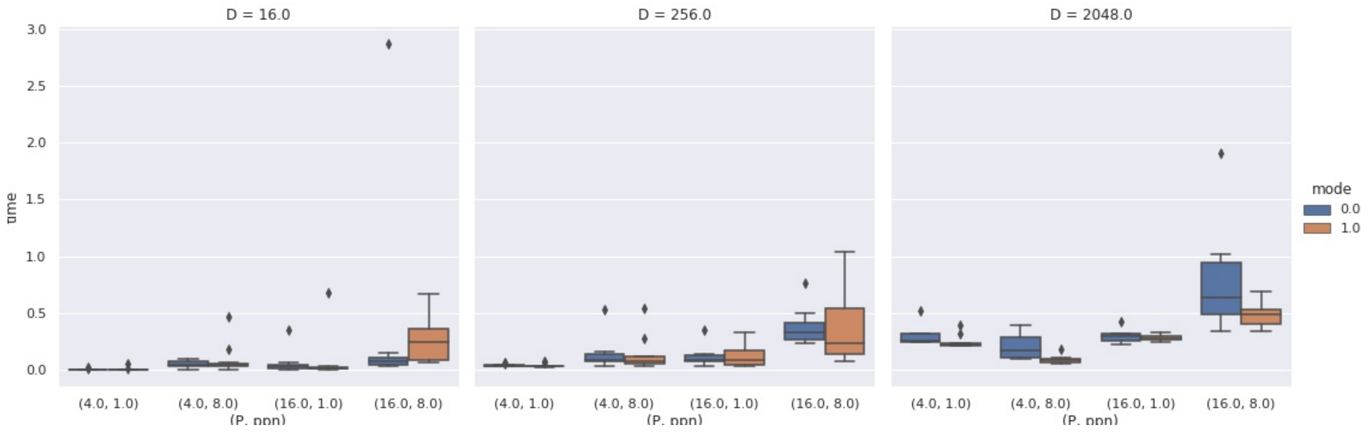

Figure 2: Bcast Case 2 plot

3 Reduce Optimized algorithm

Syntax of the function
*void new_reduce(double *data, double *retdata, int root_rank);*

data - element on which operation is needed to be performed
retdata - element which will have the result of the reduce operation.
root_rank - rank of the process which will have the result of the operation.

Algorithm

1. All leader processes will perform MPI_Reduce() among the processes of their respective groups over the Local communicator. If operation used is MPI_MAX then after this reduce among the leaders, all leaders will have the maximum value among their groups.

2. The leader processes will then perfrom MPI_Reduce() among all the leaders present in the Master communicator. The leader of the group in which the root_rank process exist will be the root of this MPI_Reduce() call.

3. The leader process of the group in which root_rank exists will now send the maximum value to the root_rank process through MPI_Send().

Analysis

Since we are creating two communicators at the time of MPI_Init, overhead of our algorithm is added only one time. So our algorithm gives better performance when collective is called multiple times. Here we have taken to cases for analysis.

- Case 1: Collective is called once
- Case 2: Collective is called 5 times

Case 1

$$speedup = \frac{t_{default}}{t_c + t_{opti}}$$

where,
$t_{default}$ = time of default MPI_Reduce() call.
t_c = time to create the Master and Local communicators.
t_{opti} = time for optimized broadcast algorithm.

Data Size	**#Nodes : ppn**			
	4 : 1	4 : 8	16 : 1	16 : 8
16KB	1.69	0.40	0.28	0.58
256KB	1.02	0.53	0.74	0.69
2048KB	1.17	0.88	1.13	1.09

Table 3: Speedup for Case 1

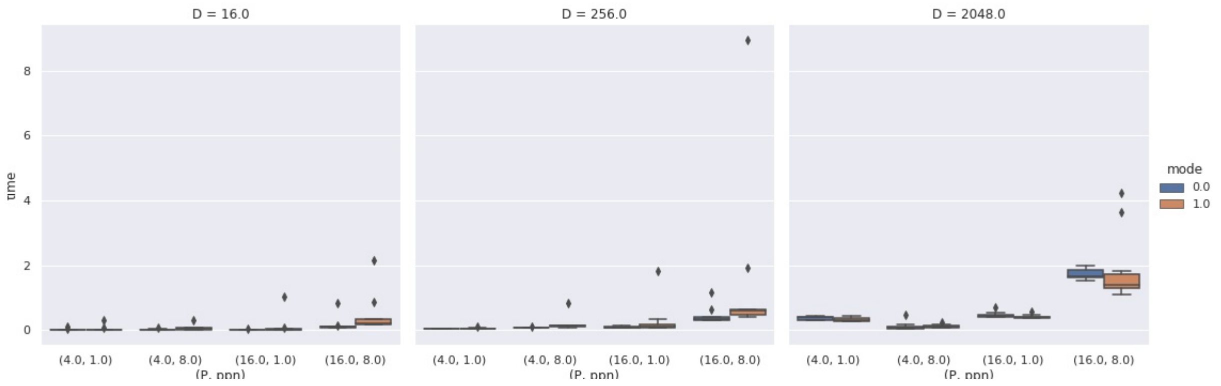

Figure 3: Optimized Reduce Case 1 plot

From Table 3, we can notice that there are speedup in some cases and in some there is not. The overhead of creating the two communicators and a MPI_Send() call is affecting the performance in the cases where there are few number of nodes and data size is less.

Case 2
This case gives better performance as the time for the creation of communicator is added only once and rest of the 4 times only time of collective call is included to calculate the speedup.

$$speedup = \frac{5 * t_{default}}{t_c + 5 * t_{opti}}$$

Data Size	**#Nodes : ppn**			
	4 : 1	4 : 8	16 : 1	16 : 8
16KB	2.49	1.40	1.12	1.01
256KB	1.15	1.03	1.04	1.00
2048KB	1.03	1.11	1.21	1.41

Table 4: Speedup for Case 2

From Table 4, we can conclude that our new algorithm is working for most of the configurations.

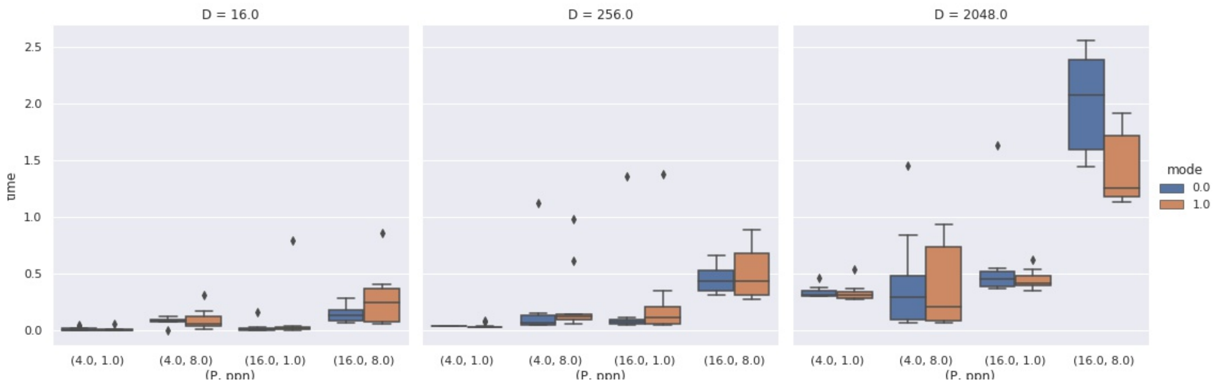

Figure 4: Optimized Reduce Case 2 plot

4 Gather Optimized algorithm

Syntax of the function
*void new_gather(double *data, double *recvdata, int root_rank);*

data - element which is sent by each process for gather.
recvdata - var will receive the data from all the other processes.
root_rank - rank of the process which will gather/receive the elements from all the other processes.

Algorithm

1. All leader processes will perform MPI_Gather() among the processes of their respective groups over the Local communicator. After this gather among the leaders, all leaders will have the data element representing the data of its group.
2. The leader processes will then perfrom MPI_Gather() among all the leaders present in the Master communicator. The leader of the group in which the root_rank process exist will be the root of this MPI_Gather() call.
3. The leader process of the group in which root_rank exists will now send the recvdata obtained from the last step to the root_rank process through MPI_Send().

Analysis

Since we are creating two communicators at the time of MPI_Init, overhead of our algorithm is added only one time. So our algorithm gives better performance when collective is called multiple times. Here we have taken to cases for analysis.

- Case 1: Collective is called once
- Case 2: Collective is called 5 times

Case 1

$$speedup = \frac{t_{default}}{t_c + t_{opti}}$$

where,
$t_{default}$ = time of default MPI_Gather() call.
t_c = time to create the Master and Local communicators.
t_{opti} = time for optimized broadcast algorithm.

Data Size	**#Nodes : ppn**			
	4 : 1	4 : 8	16 : 1	16 : 8
16KB	1.33	1.43	1.51	0.96
256KB	0.75	0.68	0.85	0.74
2048KB	1.03	1.14	1.74	1.56

Table 5: Speedup for Case 1

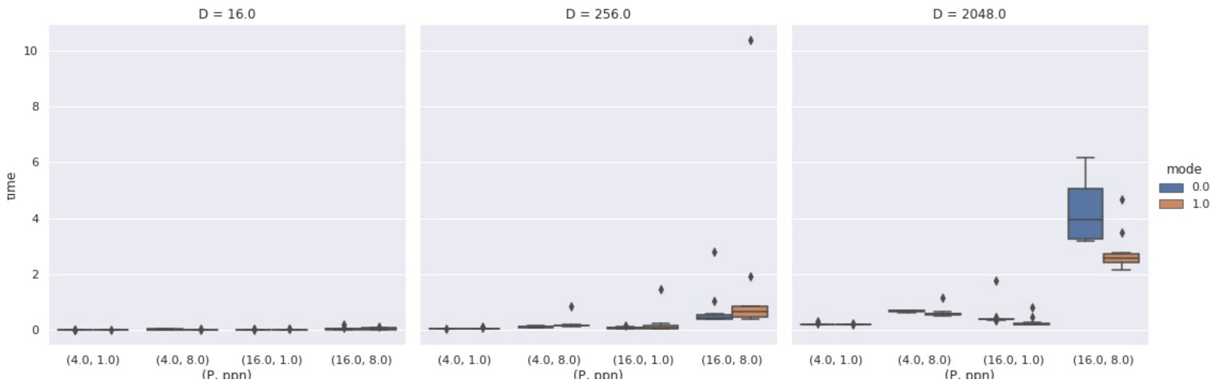

Figure 5: Optimized Gather Case 1 plot

From Table 3, we can notice that there are speedup in some cases and in some there is not. The overhead of creating the two communicators and a MPI_Send() call is affecting the performance in the cases where there are few number of nodes and data size is less.

Case 2
This case gives better performance as the time for the creation of communicator is added only once and rest of the 4 times only time of collective call is included to calculate the speedup.

$$speedup = \frac{5 * t_{default}}{t_c + 5 * t_{opti}}$$

Data Size	**#Nodes : ppn**			
	4 : 1	4 : 8	16 : 1	16 : 8
16KB	2.05	1.18	3.32	1.56
256KB	2.02	0.64	1.10	1.08
2048KB	0.97	1.21	1.51	1.50

Table 6: Speedup for Case 2

From Table 6, we can conclude that our new algorithm is working for most of the configurations.

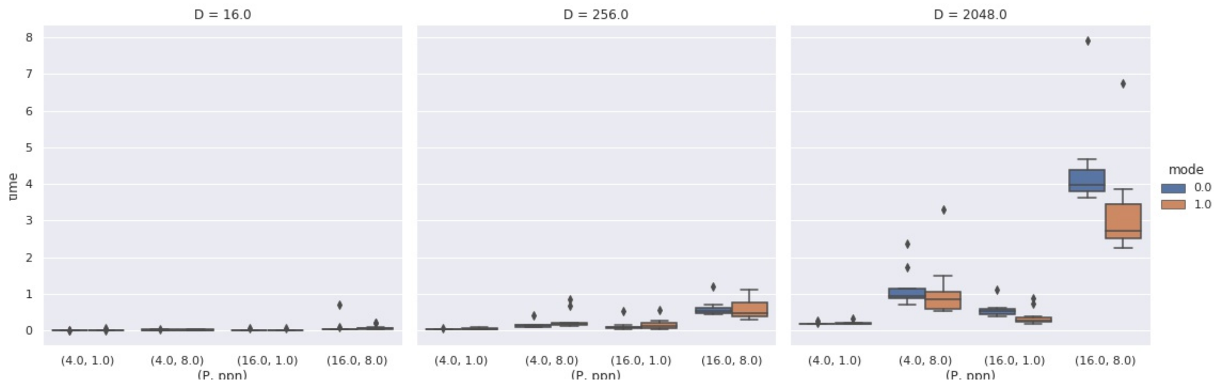

Figure 6: Optimized Gather Case 2 plot