Assignment 2

Mahi Agrawal (20111029) | Tamal Deep Maity (20111068)
maahi20@iitk.ac.in | tamalmaity20@iitk.ac.in

Contents

1. Running the Jobscript

To run the jobscript use the following command :-

```
python3 run.py
```

To generate the plots use the following command :-

```
python3 plot.py
```

2. Code Explanation

- We used an adaptive algorithm for this assignment depending upon three parameters, namely, P (the number of nodes), PPN (process per node), and D(data size). The total number of processes spawned is thus equal to P times PPN which we will refer to as 'size' from here on.

- We used three different ways to try and optimise the standard collectives. First, we created a method `opt_collective_nb()` where collective stands for Bcast, Gather, Reduce. In this method we used non blocking sends and receives to communicate data between different nodes. To ensure correctness, we used suitable MPI_Wait() and MPI_Waitall(). The next method we created was `opt_collective_datatype()`. In this method we used non blocking calls as above but here, we are sending chunks of 16KB of data using MPI_Type_contiguous() in a single go. By using suitable Wait and Waitall we ensure that the collectives are blocking. The third method we created was `opt_collective_comm()` which is explained in following points.

- Our code will create three different communicators(newcomm, newcomm_intra, newcomm_inter) if PPN is greater than 1 and it will create two different communicators if PPN is equal to one (newcomm_intra and newcomm_inter). For this we have two dedicated functions, namely `comm_create_g()` for PPN=8 and `comm_create_o()` for PPN=1.

- newcomm is the communicator handle for intranode communication, newcomm_intra is the communicator handle for intragroup communication, and newcomm_inter is the communicator handle for intergroup communication

- We have leaders, which will communicate with the other processes in the communicator. These leaders are decided on the basis of root node.

- We have designed our code such that one of the leaders in all the communicators should be the root node.

- In the intranode communication, there is a leader on each node, and the rank of this leader in the newcomm is **root%PPN**.

- The leader processes in newcomm (intranode communicator handle) form the subcommunicator newcomm_intra, It was difficult to find a mathematical relation between the ranks. Each rank in the newcomm_intra could not figure out which process is the new leader_intra by this process, so we broadcasted the rank of the root in newrank_intra to all the other processes.

- The same method as above was used in newcomm_inter. Here the maximum number of processes involved in communication can be 6, as there are 6 groups in the topology, so we have used non-blocking send and receive for sending the rank of the root in newrank_inter to other processes in the communicator instead of broadcasting.

- We reached at our adaptive algorithm by comparing the time taken by each of our created methods. Our adaptive algorithm is as follows:

  - If (D==16KB), use `opt_collective_nb()`

  - Else If (D==2048KB && size==128), use `opt_collective_comm()`

  - Else use `opt_collective_datatype()`

- To perform Alltoalv we had to do a bit of pre-processing as we were expected to send random size of data to different processes. For Alltoallv every process needs to have it's array of receive displacement, which denotes the index from the start of the receive buffer to save data in the receive buffer such that the data doesn't overlap. To find out the appropriate receive displacement and receive count each process needs to send its send count to other processes. For this we have used `MPI_Alltoall()` to send the send count.

- For the optimized Version of Alltoall we have used non-blocking send and receive. All the processes send the data to all the other processes and receive from all the other processes. Here we can also keep a count that if the send count is zero and the receive count is zero then the process does not need to send and receive the data respectively.

- We modified the `plot.py` script provided to us for plotting. We used barcharts with error bars for plotting. Since, the time taken for large number of processes was quite significant for larger data sizes and the time taken for small number of processes for small data sizes were extremely small in comparison, we chose to plot the time in logarithmic scale with base 10.

- To generate hostfile we modified the logic of `script.py` provided to us. We ran the list backwards starting from Group No. 6 rather than 1, so that we get less loaded nodes more often. We also randomized the groups so that if 2 groups are to be chosen, it isn't always Group No. 6 and 5, rather it can be any two groups from 1 to 6. Such generation of hostfiles is done so that we encapsulate the time variability for collectives among different groups, if any, in the course of 10 execution runs.

3. Optimizations Performed

- We created three communicators, one for intra node communication, one for intra group communication and one for inter group communication. One of the processes on each node (only for PPN=8) communicates data from all the other processes on the same node using the intra node communicator. These processes are therefore, the leaders in their respective intra node communicator. A communicator formed with these leaders, the intra group communicator, is used to communicate data to one leader in their respective communicators. These processes are therefore, the leaders in their respective intra group communicator. A communicator formed with these leaders, the inter group communicator, is used to communicate data to one leader which is the root process. By this way we reduced total number of hops since the inter group communication now needn't happen for each of the processes. The communicator creation turns out to be the overhead, as time taken to create communicator itself is more than the time taken to perform standard collectives. Although if there are multiple collectives calls to be performed on the same root node, it performs better than the standard collectives.

- Creating communicators this way helped reduce the number of hops between two processes belonging to different nodes or groups. In such an implementation, the leader of a communicator is the only point of communication for all processes belonging to that communicator.

- Initially we broadcasted the node number of each rank to all the nodes to figure out how many processes belong to the same node so that we could choose the leader. When we realise that instead of doing the `Bcast` from all processes, we can perform `Allgather`. We decided that the first process on a given node would be the leader for all the other processes on that node, except the node on which root was there. On that node we decided root node to be the leader. To broadcast the node number from all the processes to all the other processes took a lot of time. We realised that an alternative approach would be to broadcast the newrank of root to all the processes only once in the `MPI_COMM_WORLD`. We further optimized it later by setting all the leaders of other nodes to have the same rank as the root in the `newcomm`. We found that out by performing rank%PPN and this method didn't need any communication.

- Initially we broadcasted the newrank of the root in `newcomm_inter` to all the processes in the `MPI_COMM_WORLD`, because for broadcasting in the `newcomm_inter` it was needed to know the newrank of the root which was not known by all the other processes, but this broadcast took a lot of time. Inter group can only contain at maximum 4 processes, one from each group,and maximum number of groups can be 4 (as the maximum number of nodes can be 16), so instead of broadcasting we chose to do non-blocking send and receive with the source as `MPI_ANY_SOURCE`. This reduced the communication time a lot.

- Non blocking sends and receives were used to hasten the communication as just the time to create communicators turned out to be quite significant when smaller data sizes were used to perform collective calls. Since the collectives are blocking in nature, we used `MPI_Wait()` and `MPI_Waitall()` to replicate it accurately.

- We optimized it further by creating a contiguous datatype of 16KB of doubles to be communicated at a time. Since our data sizes were 16KB, 256KB and 2048 KB, we decided that 16KB would be an optimal chunk size for the data to be sent in. We saw improvement in execution time over the other two methods for a lot of configurations

this way.

- For small data sizes and less number of processes, the communicator creation time is much significant than the actual communication time. Hence, the other two methods perform better. On the other hand for large number of processes and large data size, the communicator creation time is not very significant and the benefits of reducing the number of hops outweigh that small overhead.

- From our analysis of previous assignment, we drew the conclusion that packing data together before communicating was efficient if the data to be accumalated wasn't very huge. This was our logic for using `MPI_Type_contiguous()`.

- We decided not to use communicator for our optimized version of `MPI_Alltoallv` as the data communication involved was huge. Moreover we observed that when we used communicator in `MPI_Gather` it sometimes performed good, and sometimes performed poorly. As compared to `MPI_Gather`, `MPI_Alltoallv` involves more data at leaders, which may degrade its performance on using communicators.

4. Additional Tests

- We tried to compute the hop latency and bandwidth by sending data of same size from one process, once to another process in same group and once to another process in a different group. That way we tried to compute the hop latency since we know the number of hops needed between nodes of different groups. Next time, we sent data of different sizes between the same two processes. Our aim was to find out which factor, number of hops or data size sent, was the more significant one. However, we couldn't deduce any observation from this exercise.

- We tried sending 1000 elements 100 times from one process in one group to one process in another group and compared the time taken with sending 1,00,000 elements between the same two processes. Time taken for the latter was lesser. From this we realised if the data size is small then it is better to accumulate the data and then send once rather than sending it multiple times.

- Before replacing broadcast with root%PPN to find out the newrank of the leader in `newcomm` we also replaced it with point to point communication with root being the source and all the other processes being the destination. And on several runs both took same time to run.

- We were trying to make a generic communicator such that whatever maybe the root and whatever maybe the collective, then the communicator should only be created once. But in this case it was required to fix the leaders with a certain rank. And then it was needed to transmit the data to the root, in case of gather and reduce. To collect the data from the root in case of the bcast, the transferring of the data from the leader to the root became the bottleneck, as in case of the gather the data size becomes **D*size**, which is huge. It took significant amount of time to transfer data from leader to the root as compared to again creating the communicator on the basis of root. And so we decided to stick with the approach of creating communicator on the basis of root node.

- To make few deductions we also sent 0 byte of data from one process to another process and it takes a noticeable amount of time to send 0 byte of data regardless of whether the processes are on the same node or not.

5. Problems/Issues Faced

- Uninitialized variables declared together in the same line get initialized to the same garbage value due to which there comparison was turning out to be true which we thought would be false while writing the code.

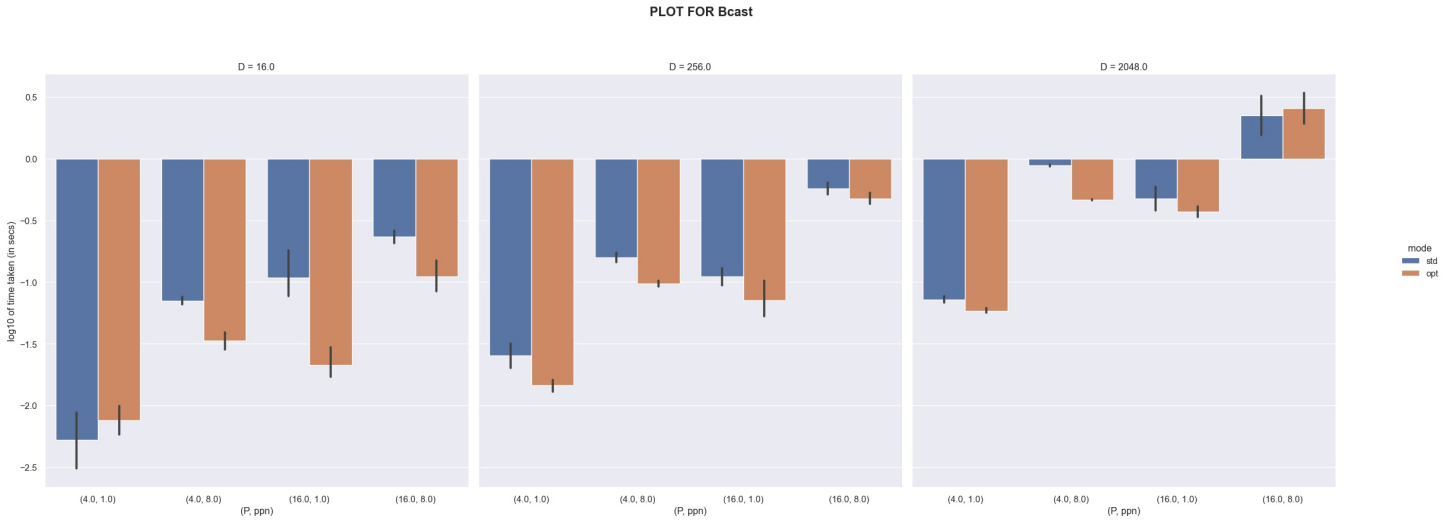
- We wanted to create communicators such that only few processes belongs to that communicator, we were creating communicator using the `MPI_COMM_Split`, which does not return till the time all the processes does not enter it, of the parent communicator of the new sub communicator. Due to this our program got into deadlock, we resolved this by using `MPI_UNDEFINED` for the key/color of the processes, which we didn't want to be the part of our new sub-communicator.

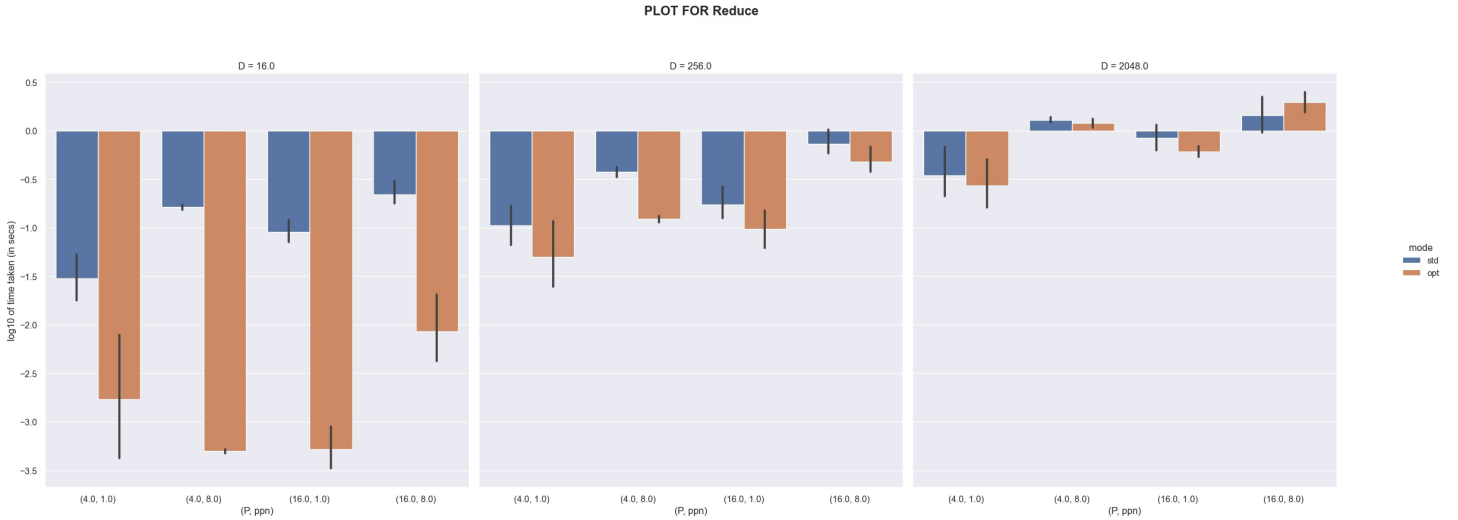
- While using Waitall, we have to use a request array. Now, say in case of Bcast, root sends data to (size-1) processes. The iterable, say i, can't be used in the Isend request array because while the request of the processes after root will not get placed continuously. However, while checking for the waitall the request array is checked for length (size-1) and that creates an issue. So to continuously place the requests for processes after root, the iterable has to be decremented by 1.

- Initially we were not using `MPI_Waitall()` for both the send and receive request in alltoallv optimized implementation. Although it worked fine for smaller number of processes and small data size.

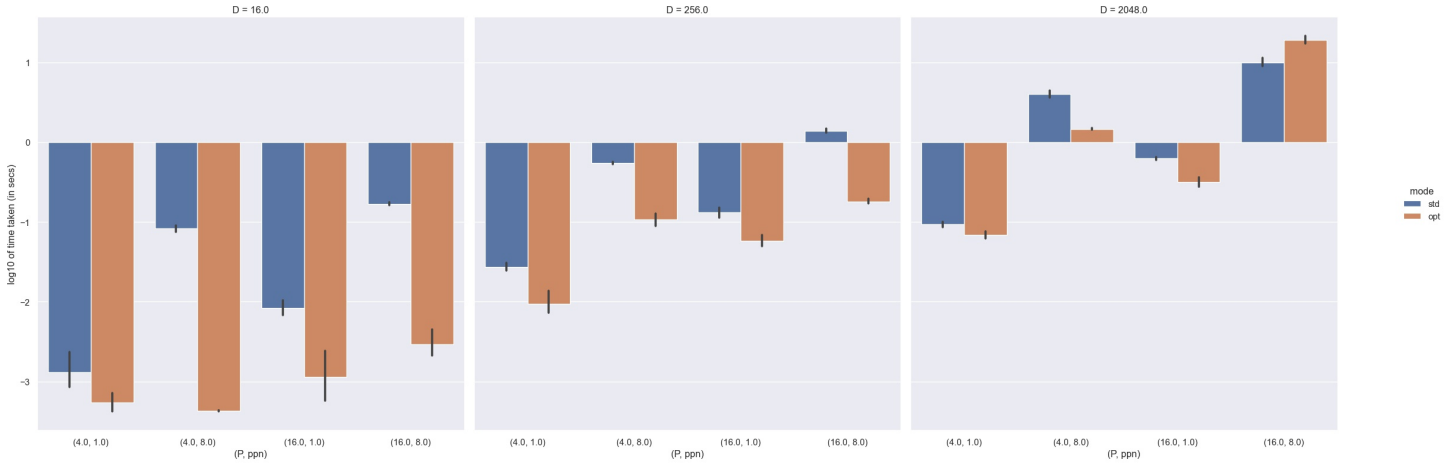
6. Plots

Since, we used log base 10 in the Y axis for plotting time, a higher negative value in the plot actually indicates a smaller positive value. So, for negative barcharts, optimization was found when the blue bars are smaller than their brown counterparts.

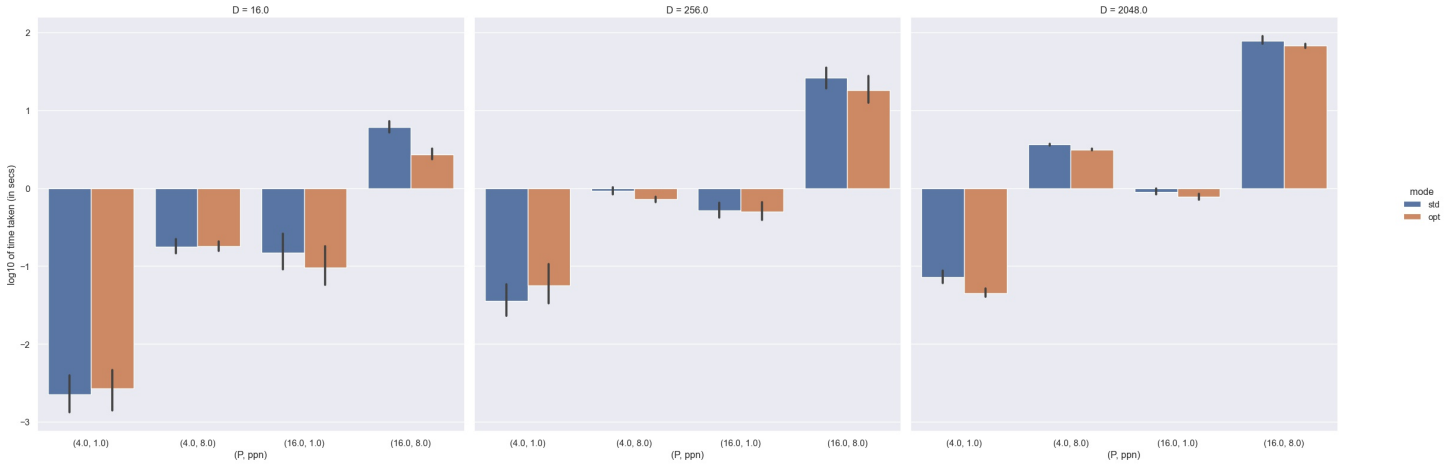




PLOT FOR Gather



PLOT FOR Alltoallv



7. Observations

- The time overhead incurred in broadcasting (or doing non blocking send/receive) the rank of the root in new communicator is due to the fact that we have considered root to be one of the input arguments. Static root allocation like say, rank 0 being root would have prevented these overheads but introduced a new overhead where all the information has to be sent back to the root from the leader of the communicator that root is in. This can be easily implemented in case of collectives like Gather or Reduce but would be harder for collectives like Alltoallv, thus increasing the overhead.

- If the collectives have to be performed on the same root, then the overhead of communicator creation will only be once, and then total time taken by our optimized collective (including the overhead of creating communicator once) will be much lesser than the total time taken by standard collectives. In contrast to this, if the root is chosen dynamically, the commnicator creation overhead has to be considered every time the root changes.

- For the following analysis of collectives let us consider there are P processes, and root wants to perform a collective operation with array of N elements.

  `MPI_Reduce()` follows Rabenseifner's algorithm. In first round of this algorithm, total of $P * \frac{N}{2}$ data is communicated, in second round $P * \frac{N}{4}$ data is communicated, similarly it goes on till each process has 1 element reduced.Then the data is gathered at root process. There has to be a synchronization between the rounds.So that the next round starts only when first round is completed. This synchronization becomes an overhead when the data size is small. In our optimized version there are no such rounds,and there is no such need of synchronization between processes, due to which `opt_reduce_nb()` performs better than the standard algorithm for smaller data size. Where as when there are large number of processes and data size is huge `opt_reduce_nb()`'s performance degrades as communication at root becomes the bottleneck.

- In standard `MPI_Gather()`, the data is transmitted in binomial fashion. For example, there are 7 processes (0 to 1), then in first round, 7,5,3,and 1 transmits the data to 6,4,2 and 0 respectively. In second round 6 and 2 send data to 4 and 0 respectively. In third round 4 sends data to 0. Here also there has to be synchronization between each round. Moreover total data sent is huge as compared to the scenario when each process is sending the data to root. Therefore `opt_gather_nb()` performs better than the standard algorithm for smaller data size. Where as when there are large number of processes and data size is huge `opt_gather_nb()`'s performance degrades as communication at root becomes the bottleneck.

- Standard `MPI_Bcast()` follows Van de Geijn et al algorithm. In this algorithm the data is scattered and then allgather is performed to collect data. In this the number of hops increases, and to implement allgather also there has to be synchronizations. Increased number of hops and synchronizations become an overhead when the data size is small. Therefore our optimized version `opt_bcast_nb()` performs slightly better than the standard version for small data size.

- We think that for similar reasons our optimized version of Alltoallv performs better than the standard collective.

- In previous assignment we observed that MPI Datatype performs better than the MPI point to point send and receive without datatype. So, we tried to optimize our `opt_bcast_nb()` for medium size messages using `opt_bcast_dataype()`. We also achieved some optimization for large sized data and small number of processes that way. But the same could not be used for large number of processes and large data size as communication at root becomes bottleneck.

- As explained in above points that `opt_bcast_nb()`,`opt_bcast_dataype()` cannot be used for huge number of processes and large datasize. We have used network topology and multi-core knowledge to optimize the standard collectives. The time taken by our optimized version is dictated by the time taken to create communicators. If time taken to create communicators is less then our optimized version `opt_collective_comm()`, performs better than the standard collective. The reason behind this is that it takes lesser number of hops to transmit the data as when compared to standard collective.
  `opt_collective_comm()` does not perform good for smaller number of processes and small data size as the time taken by standard collective is extremely small and the time taken to create the communicators is a huge overhead in those scenarios.

8. Experimental Setup

(1) **Headers included in C program:**

- stdio.h, stdlib.h, math.h, string.h, assert.h, mpi.h (MPICH version 3.3.2)

(2) **Python packages/modules used:**

- matplotlib (version 3.3.4)
- pandas (version 1.1.5)
- os
- math
- numpy (version 1.19.5)
- seaborn (version 0.11.1)