Assignment 2

How to run

```
python3 run.py
```

The folder contains the files:

1. run.py : creates the hostfile using script.py and runs src.c for each configuration, generates the plots for all 4 collectives
2. script.py : creates the hostfile using information from nodefile.txt
3. nodefile.txt : contains csews cluster information required by script.py
4. Makefile : compiles src.c
5. src.c : contains the implementation of optimized collective functions based on the topology of *csews* cluster

Code

Helper Functions:

```
int get_my_node()
int get_my_group(int node)
double get_arrays(int root)
double get_comms(int root, int* is_node_leader, int* is_group_leader)
void Operation(MPI_Op op, double *itmd_buf, double *sendbuf, int count)
```

1. get_my_node() : returns which node the process runs on based on hostname
2. get_my_group() : returns group no. based on node
3. get_arrays(int root): for each process, creates the arrays node_ranks, node_leaders and group_leaders

- node_ranks: array of all ranks on the corresponding node
- node_leaders: array of node_leaders on the corresponding group
- group_leaders: array of group_leaders

4. get_comms(int root, int* is_node_leader, int* is_group_leader): creates the subcommunicators for the three levels of communication i.e. node level (*node_comm*), group level (*nl_comm*) and global (*gl_comm*) depending on whether the calling process is a node_leader or group_leader
5. Operation(MPI_Op op, double *itmd_buf, double *sendbuf, int count): given two buffers, this function applies the *op* operation on each element of the two arrays and returns the updated value in sendbuff.

Optimized Collective Functions:

```
int MPI_Bcast_optimised(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm, int
```

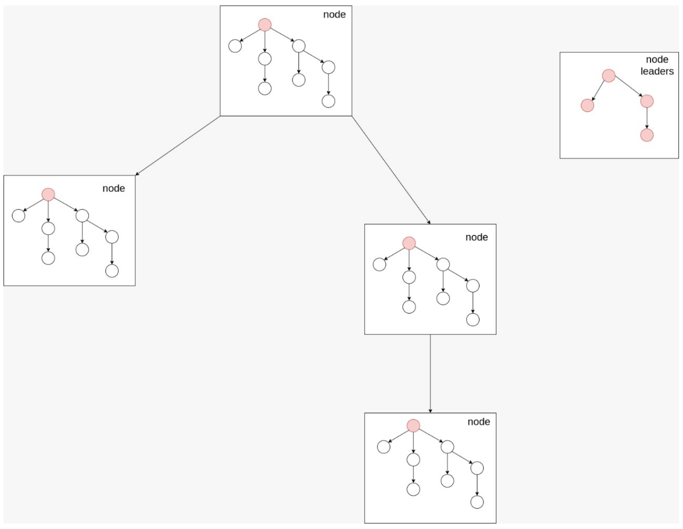
```
int MPI_Reduce_optimised(void* sendbuf, double *recvbuf, int count, MPI_Datatype datatype, MPI_Op op,
```

```
int MPI_Gather_optimised(const void *sendbuf, int scount, MPI_Datatype stype, void *recvbuf, int rcou
int MPI_Alltoallv_optimised(const void* sendbuff, const int* send_counts, const int* send_displs, MPI
```

Heirarchy and Assumptions

Heirarchy

1. The processes are in a 3 level heirarchy of group leaders, node leaders, and nodes.
2. For each process we find which node and group it belongs to, thus finding the corresponding arrays for ranks in the node(node_ranks), node_leaders in the group(node_leaders), and the overall group leaders(group_leaders)
3. We have 3 subcommunicator levels: node_comm(corresponding to node_ranks), nl_comm(corresponding to node_leaders) and gl_comm(corresponding to group_leaders)
4. This is to mitigate inter-node and inter-group communication overheads. Thus, all nodes communicate with their node leaders, all node leaders communicate with their group leaders and the group leaders communicate with the root. We maintain that the root(wherever applicable) is always the node leader and group leader for its corresponding node and group.
5. We have added a check for no of groups. If no of groups is 1, we do not create gl_comm.



Assumptions

Optimised collective functions assume the following:

1. Root is known before hand and remains the same for all the 5 iterations. (MPI_Bcast, MPI_Reduce, MPI_Gather)
2. Consequently we can reuse the communicators that have been created for nodes, node leaders and group leaders. (Please Note that this condition can be relaxed by using a single MPI_Send from root to the respective node leader.)
3. The communicator used for the collective calls is MPI_COMM_WORLD in all cases. (MPI_Bcast, MPI_Reduce, MPI_Gather, MPI_Alltoallv)
4. The send buffer is contiguous. (MPI_Alltoallv) (This assumption can be relaxed, this has been discussed in the next section.)

It shall be noted that the time for extra commmunicator creation (n_comm, nl_comm and/or gl_comm) has been included as overhead.

Optimisations

Bcast

1. Binomial Implementation: [This can be found in bcast.c]

- At each level of heirarachy we carry out the communication in binomial fashion (as depicted in the diagram).
- Thus, the root sends data to group leaders using MPI_Isend/MPI_Irecv, group leaders to node leaders and finally node leaders to all ranks in their nodes.
- This implementation relieves us of subcommunicator creation overheads.
- Significant performance improvement can be observed for small data sizes [approx. 5 times].

2. Bcast with subcommunicators: [We have used this implementation for plot generation]

- Root sends data to the group leaders in gl_comm communicator using stanard Bcast. Thereafter, group leaderes send data to the node leaders in nl_comm and node leaders send data to all processes(ranks) in node_comm, all using standard Bcast.
- This performs consistently well for all data sizes.

Reduce

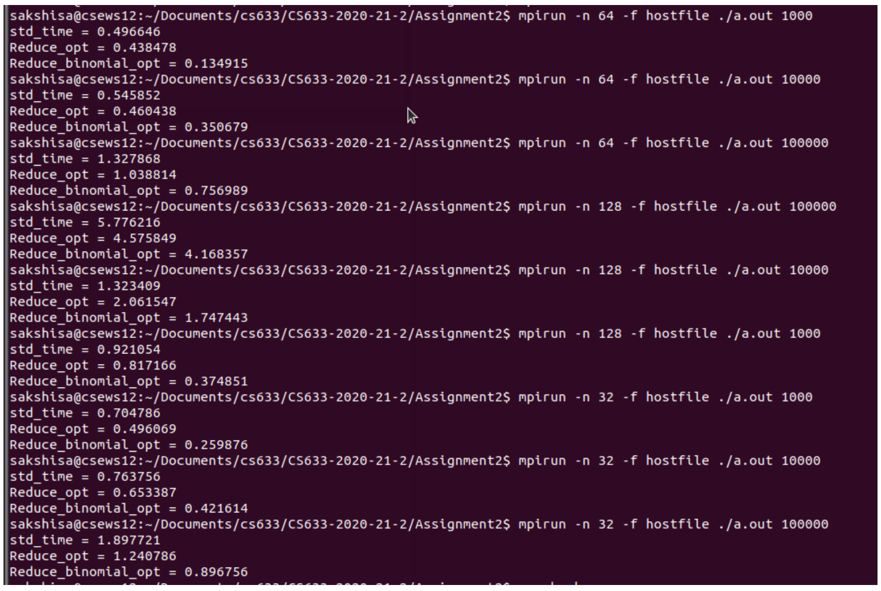
1. Binomial Implementation: [We have used this implementation for plot generation]

- Starting at node level, binomial reduce is carried out within node to find the reduced value of data at the node leaders.
- This is done by sending data to respective parent using MPI_Send/MPI_Recv. Using helper function *Operation(MPI_Op op, double* \itmd_buf, double *sendbuf, int count)*, the required reduced value is calculated and passed on to further levels. This way the node leaders get the final reduced value for a node.
- The same method is carried out at nl_comm and gl_comm levels.
- The root gets the final reduced value through group_leaders.
- This method optimises reduce significantly for small data sizes. Time reduction by a factor of approx 3 for small data sizes. Optimization is good for large data sizes as well.

2. Reduce with subcommunicators:

- Perform standard reduce at each level of communication, starting at node level using n_comm.
- Reduced data from node leaders is further reduced at group level using nl_comm. Similarly, the root gets final reduced value using gl_comm.
- The performance is comaparable to standard reduce and mostly better.

[This comaparison script can be found in reduce-compare.c]



```
sakshisa@csews12:~/Documents/cs633/CS633-2020-21-2/Assignment2$ mpirun -n 64 -f hostfile ./a.out 1000
std_time = 0.496646
Reduce_opt = 0.438478
Reduce_binomial_opt = 0.134915
sakshisa@csews12:~/Documents/cs633/CS633-2020-21-2/Assignment2$ mpirun -n 64 -f hostfile ./a.out 10000
std_time = 0.545852
Reduce_opt = 0.460438
Reduce_binomial_opt = 0.350679
sakshisa@csews12:~/Documents/cs633/CS633-2020-21-2/Assignment2$ mpirun -n 64 -f hostfile ./a.out 100000
std_time = 1.327868
Reduce_opt = 1.038814
Reduce_binomial_opt = 0.756989
sakshisa@csews12:~/Documents/cs633/CS633-2020-21-2/Assignment2$ mpirun -n 128 -f hostfile ./a.out 100000
std_time = 5.776216
Reduce_opt = 4.575849
Reduce_binomial_opt = 4.168357
sakshisa@csews12:~/Documents/cs633/CS633-2020-21-2/Assignment2$ mpirun -n 128 -f hostfile ./a.out 10000
std_time = 1.323409
Reduce_opt = 2.061547
Reduce_binomial_opt = 1.747443
sakshisa@csews12:~/Documents/cs633/CS633-2020-21-2/Assignment2$ mpirun -n 128 -f hostfile ./a.out 1000
std_time = 0.921054
Reduce_opt = 0.817166
Reduce_binomial_opt = 0.374851
sakshisa@csews12:~/Documents/cs633/CS633-2020-21-2/Assignment2$ mpirun -n 32 -f hostfile ./a.out 1000
std_time = 0.704786
Reduce_opt = 0.496069
Reduce_binomial_opt = 0.259876
sakshisa@csews12:~/Documents/cs633/CS633-2020-21-2/Assignment2$ mpirun -n 32 -f hostfile ./a.out 10000
std_time = 0.763756
Reduce_opt = 0.653387
Reduce_binomial_opt = 0.421614
sakshisa@csews12:~/Documents/cs633/CS633-2020-21-2/Assignment2$ mpirun -n 32 -f hostfile ./a.out 100000
std_time = 1.897721
Reduce_opt = 1.240786
Reduce_binomial_opt = 0.896756
```

Gather

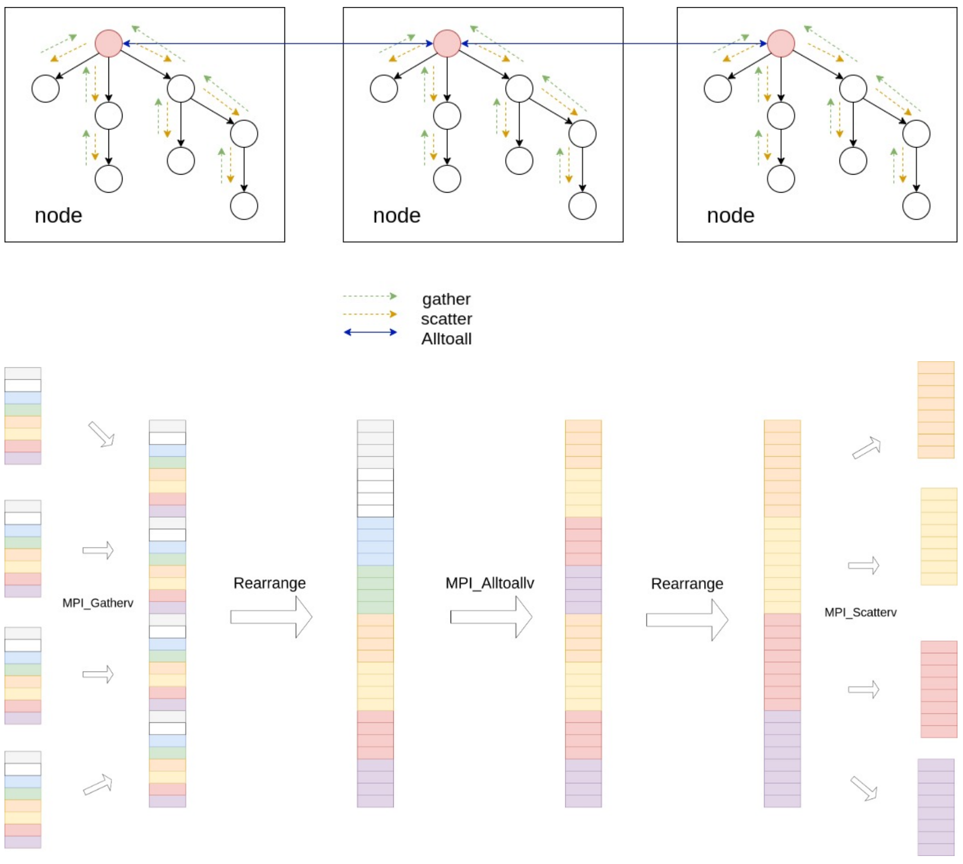
1. Gather with subcommunicators:

- Perform standard gather at each level of communication, starting at node level using n_comm.
- Gather process ranks in to comm at each level of communication to rearrange data at root.
- The performance is comparable to standard reduce.

Alltoallv

1. Alltoallv with subcommunicators:

- Starting at node level, the node leader gathers send_buff and send_counts from all ranks in the n_comm using MPI_Gatherv.
- Using send_counts information, node leader rearranges data to place data outgoing to same rank contiguously.
- Node Leaders exchange data going to their nodes using MPI_Alltoallv.
- Node Leaders again rearranges the data.
- Node Leaders sends data to respective ranks within the node using MPI_Scatterv. (see fig for better visualisation)
- All ranks then copy recieved data into recieve buffer at appropriate offsets.
- This method optimises alltoallv significantly for large number of processes (given enough processes per node.)
- Assumption 4. has been used in gather step. Non leader processes simply send the complete send buffer instead of copying data from provided offsets. This assumption can be done away with by just copying relevant parts of send buffer into a new buffer and use this buffer for MPI_Gatherv, without adding much overhead.

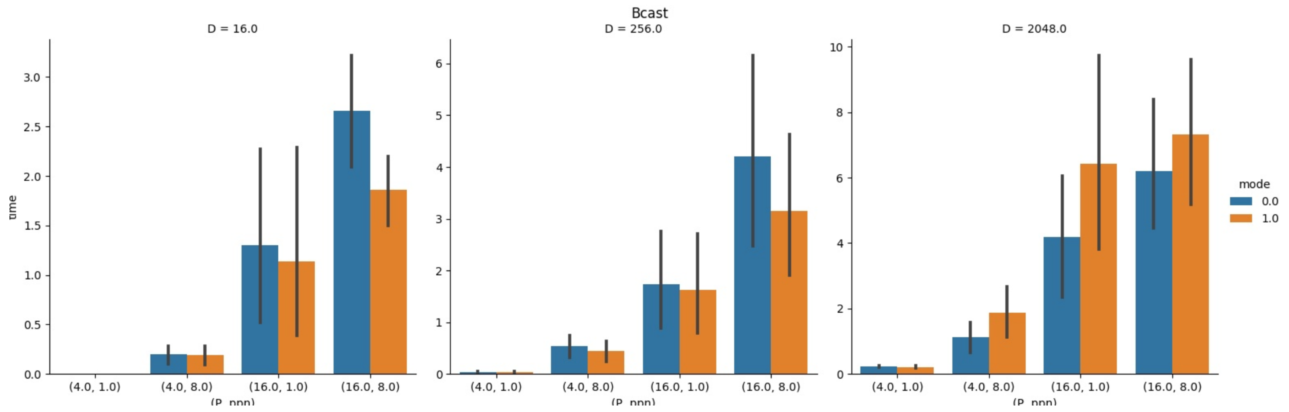


Performance Analysis

For all plots: mode 0 is optimised version of collective and mode 1 is default.

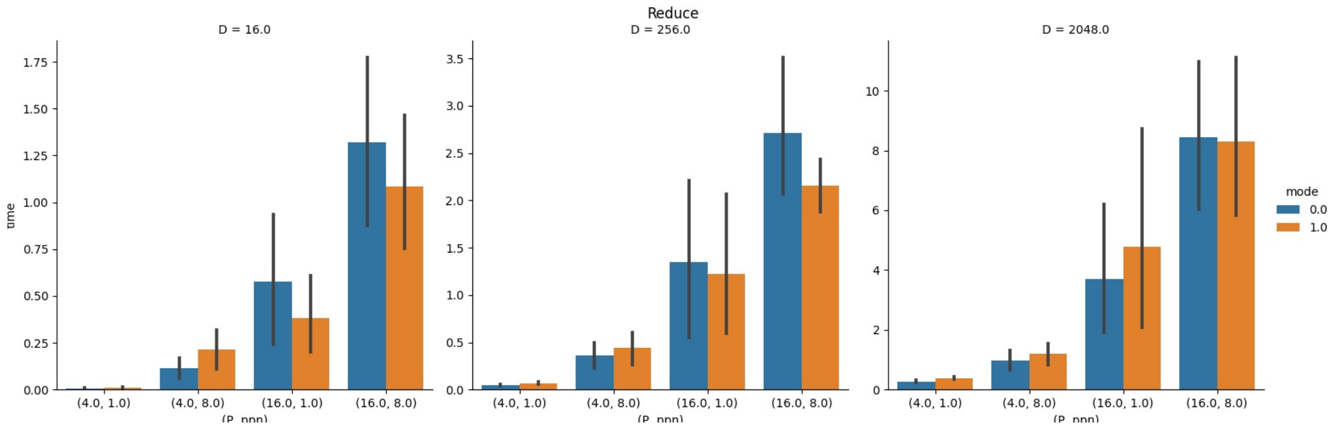
Bcast

- Performance is almost comparable for small data size. Optimisation performance improves with increase in data size.
- Performance does not change much with change in number of processes or ppn.



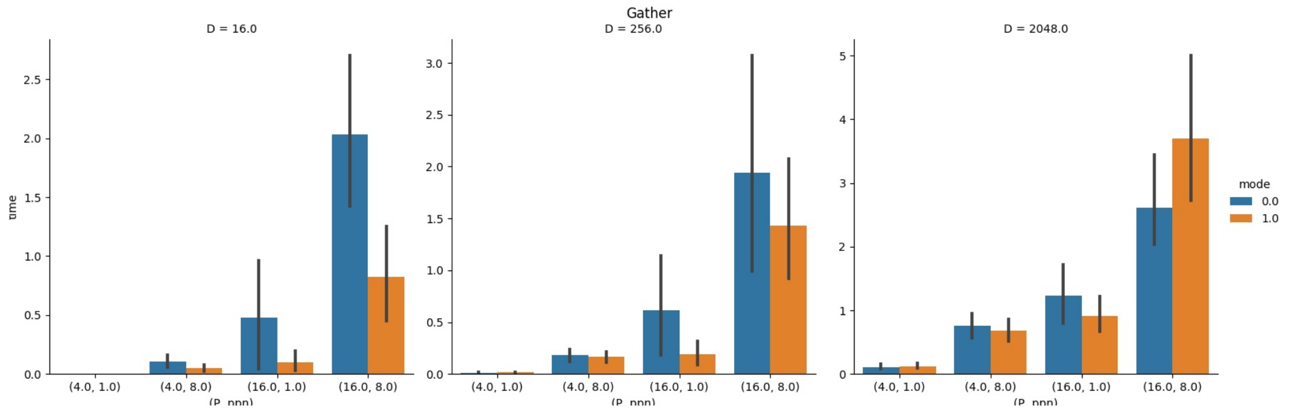
Reduce

- Performance is significantly better for small data sizes and comparable for large data sizes
- With increased no of groups, the optimized code performs slightly worse. Else, optimization can be seen.



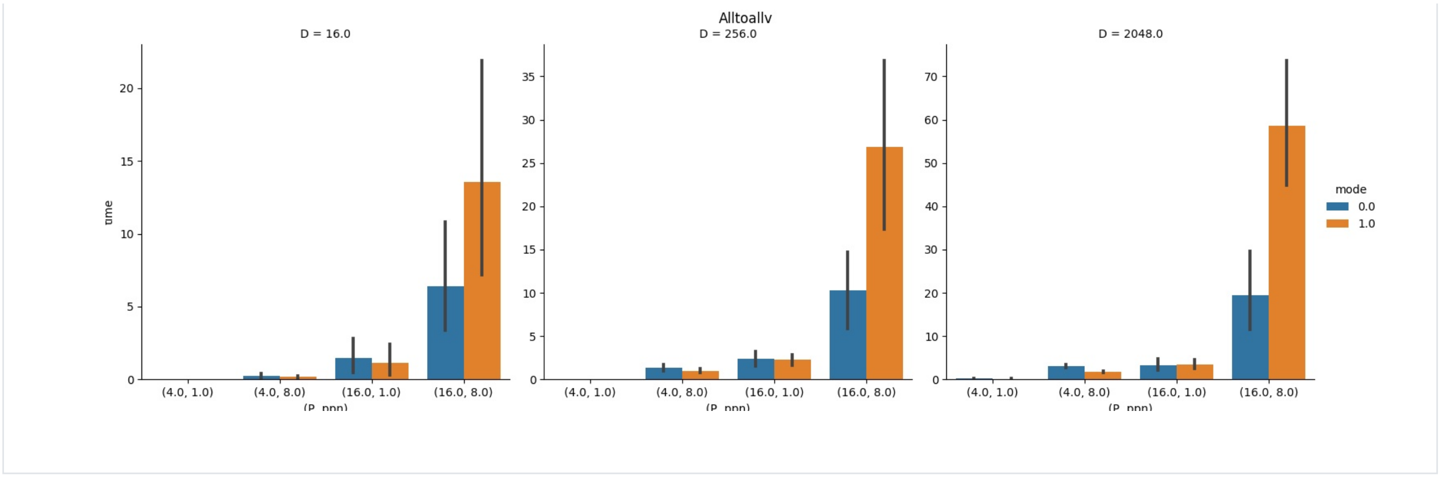
Gather

- Performance is mostly comparable. No improvement can be seen.
- Only improvement is for data size with many groups having large ppn



Alltoallv

- Performance for optimised version improves significantly with increase in number of processes.
- Performance gap increases with increase in data size.
- Performance gap significantly increases with number of processes per node.


Alltoallv
D = 16.0
D = 256.0
D = 2048.0
time
(P, ppn)
(4.0, 1.0)
(4.0, 8.0)
(16.0, 1.0)
(16.0, 8.0)
mode
0.0
1.0