Assignment – 3

Group-13 Report

Ayyappa Reddy Maramreddy – R11888944

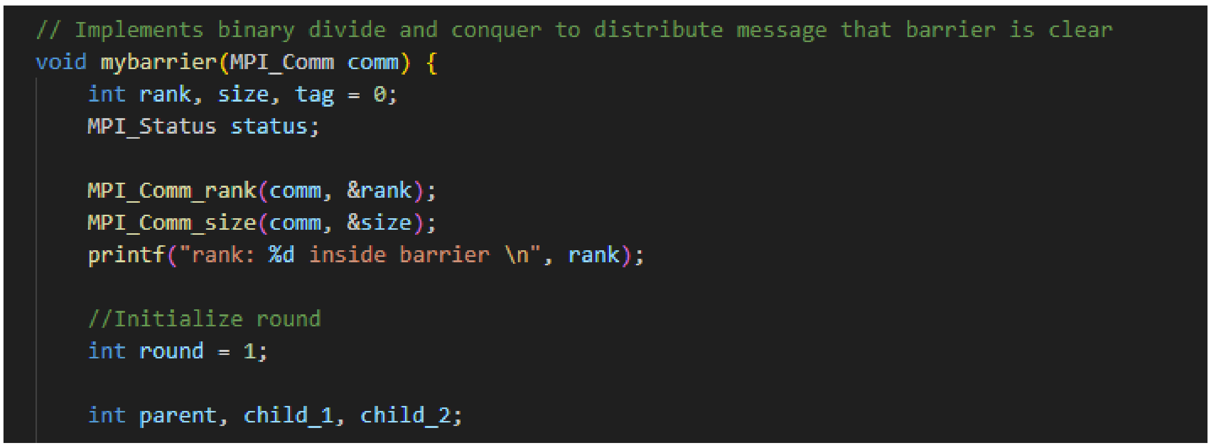
Jason Heinrich – R11913725

Rishabh Kumar – R11785189

Description:

The provided code is an MPI Program written in C that implements a custom barrier function using only MPI_Send () and MPI_Recv () communication protocols that requires no more than $4\log_2(Nodes)$ steps.

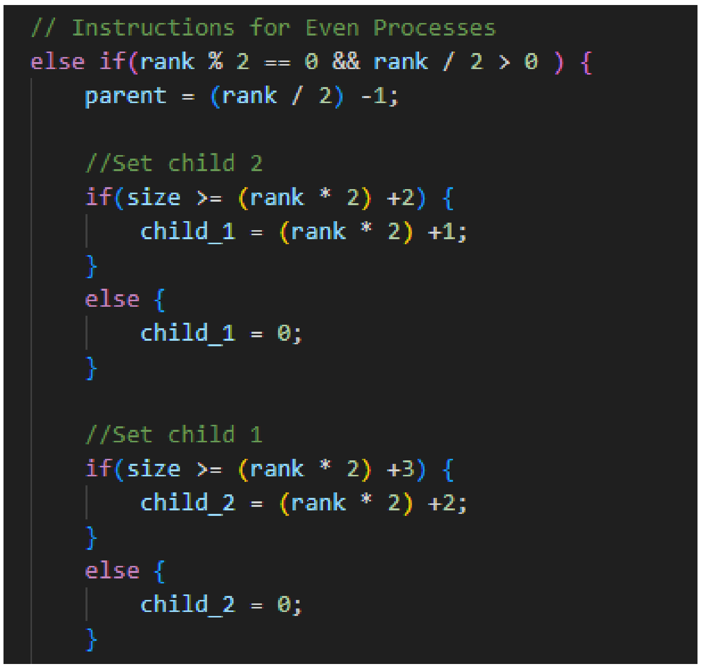
A barrier function should signal to all other active nodes that every process has made it inside the barrier function and disseminate the message that every node has arrived to all nodes so they can exit the barrier function and complete their next task. In our method of implementation, we modelled the distribution of the message off of a binary tree. In our function we first collected the rank of the node, got the size of the total number of nodes, and set the "round" variable to 1.



```c
// Implements binary divide and conquer to distribute message that barrier is clear
void mybarrier(MPI_Comm comm) {
    int rank, size, tag = 0;
    MPI_Status status;

    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
    printf("rank: %d inside barrier \n", rank);

    //Initialize round
    int round = 1;

    int parent, child_1, child_2;
```

Fig 1. Initialization of variables for each process entering barrier

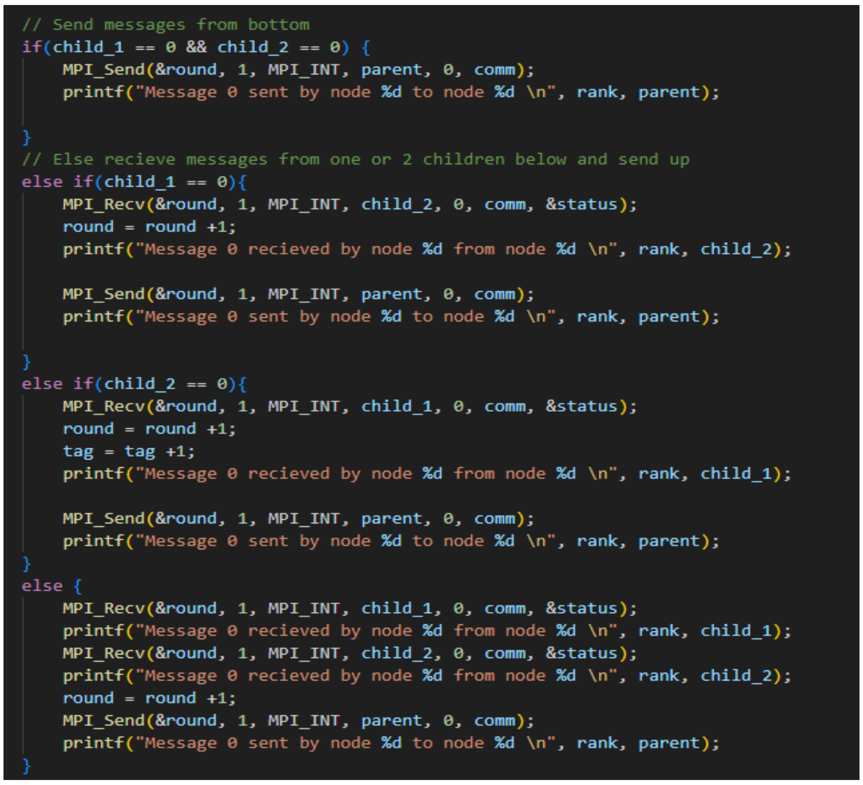
We then created 4 rank classifications. 1 for the root node at position 0, 1 for the first child at position 1, 1 for all even numbers, and 1 for all odd numbers of value 3 and larger. We then mathematically assigned each case its children and compared the node position of its children to determine if they existed. For example, the snippet below shows the child selection and comparison for even numbered processes.



```c
// Instructions for Even Processes
else if(rank % 2 == 0 && rank / 2 > 0 ) {
    parent = (rank / 2) -1;

    //Set child 2
    if(size >= (rank * 2) +2) {
        child_1 = (rank * 2) +1;
    }
    else {
        child_1 = 0;
    }

    //Set child 1
    if(size >= (rank * 2) +3) {
        child_2 = (rank * 2) +2;
    }
    else {
        child_2 = 0;
    }
```

Fig 2. Mathematical initialization of child and parent components

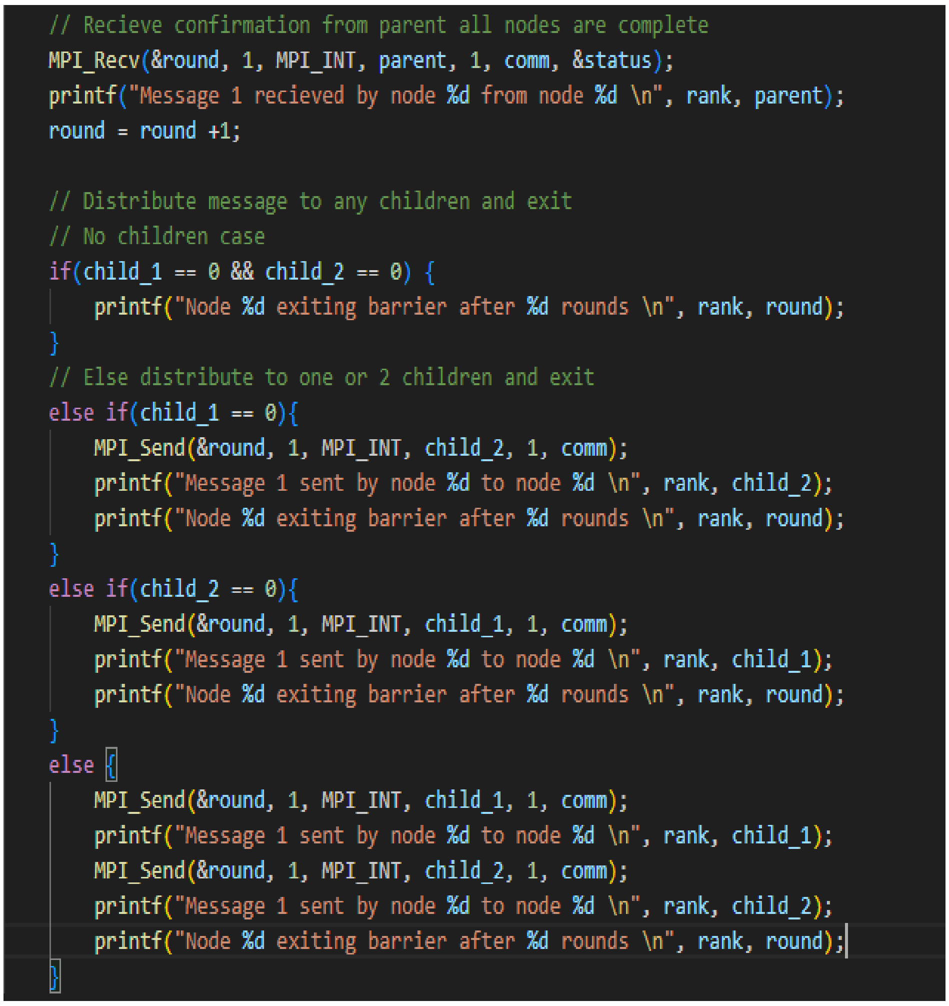
For any nodes with no children, their first operation will be to send a message to the parent node that they are in the barrier along with their round = 1 variable. All nodes with children first set up a receive for any existing children, increment the round variable and send on to their respective parent node.



```c
// Send messages from bottom
if(child_1 == 0 && child_2 == 0) {
    MPI_Send(&round, 1, MPI_INT, parent, 0, comm);
    printf("Message 0 sent by node %d to node %d \n", rank, parent);

}
// Else recieve messages from one or 2 children below and send up
else if(child_1 == 0){
    MPI_Recv(&round, 1, MPI_INT, child_2, 0, comm, &status);
    round = round +1;
    printf("Message 0 recieved by node %d from node %d \n", rank, child_2);

    MPI_Send(&round, 1, MPI_INT, parent, 0, comm);
    printf("Message 0 sent by node %d to node %d \n", rank, parent);

}
else if(child_2 == 0){
    MPI_Recv(&round, 1, MPI_INT, child_1, 0, comm, &status);
    round = round +1;
    tag = tag +1;
    printf("Message 0 recieved by node %d from node %d \n", rank, child_1);

    MPI_Send(&round, 1, MPI_INT, parent, 0, comm);
    printf("Message 0 sent by node %d to node %d \n", rank, parent);
}
else {
    MPI_Recv(&round, 1, MPI_INT, child_1, 0, comm, &status);
    printf("Message 0 recieved by node %d from node %d \n", rank, child_1);
    MPI_Recv(&round, 1, MPI_INT, child_2, 0, comm, &status);
    printf("Message 0 recieved by node %d from node %d \n", rank, child_2);
    round = round +1;
    MPI_Send(&round, 1, MPI_INT, parent, 0, comm);
    printf("Message 0 sent by node %d to node %d \n", rank, parent);
}
```

Fig 3. Logic for all non-root nodes to receive confirmations

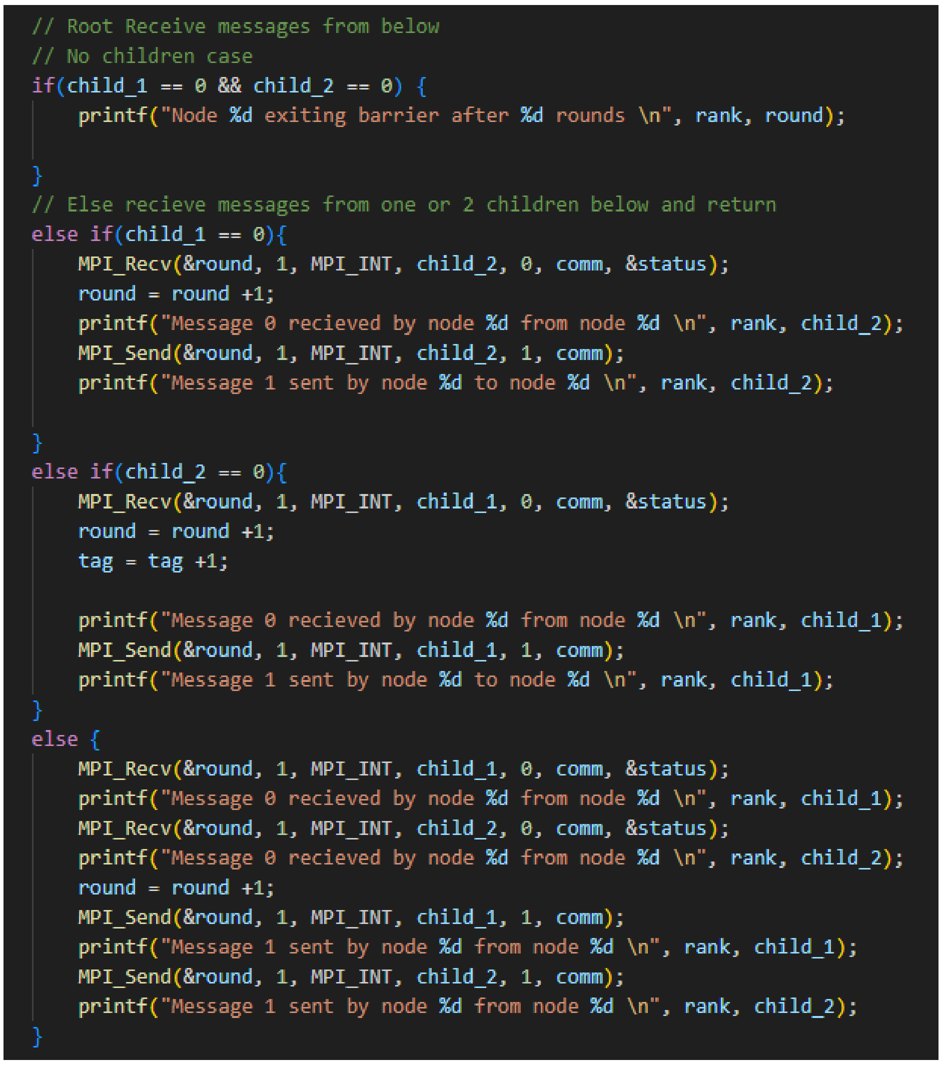
Once the node has sent the “all clear here” message to the parent it sets up a receive for the parent to return the message that all other nodes have notified the parent and been transmitted back down. Once it receives that message it them passes on to its children or exits if it has none.



```c
// Recieve confirmation from parent all nodes are complete
MPI_Recv(&round, 1, MPI_INT, parent, 1, comm, &status);
printf("Message 1 recieved by node %d from node %d \n", rank, parent);
round = round +1;

// Distribute message to any children and exit
// No children case
if(child_1 == 0 && child_2 == 0) {
    printf("Node %d exiting barrier after %d rounds \n", rank, round);
}
// Else distribute to one or 2 children and exit
else if(child_1 == 0){
    MPI_Send(&round, 1, MPI_INT, child_2, 1, comm);
    printf("Message 1 sent by node %d to node %d \n", rank, child_2);
    printf("Node %d exiting barrier after %d rounds \n", rank, round);
}
else if(child_2 == 0){
    MPI_Send(&round, 1, MPI_INT, child_1, 1, comm);
    printf("Message 1 sent by node %d to node %d \n", rank, child_1);
    printf("Node %d exiting barrier after %d rounds \n", rank, round);
}
else {
    MPI_Send(&round, 1, MPI_INT, child_1, 1, comm);
    printf("Message 1 sent by node %d to node %d \n", rank, child_1);
    MPI_Send(&round, 1, MPI_INT, child_2, 1, comm);
    printf("Message 1 sent by node %d to node %d \n", rank, child_2);
    printf("Node %d exiting barrier after %d rounds \n", rank, round);
}
```

Fig 4. Logic for all non-root nodes to receive confirmations from parent

This process is roughly the same for all nodes except the root node. Since the root node has no parent, if it has children it listens for them, increments the round and sends it right back to the children. Since the parent will be the last node to receive the bottom-up notification that all nodes have entered, it initiates the downward dissemination of the “all clear” message to its children and then will be the first to exit the barrier function and proceed.



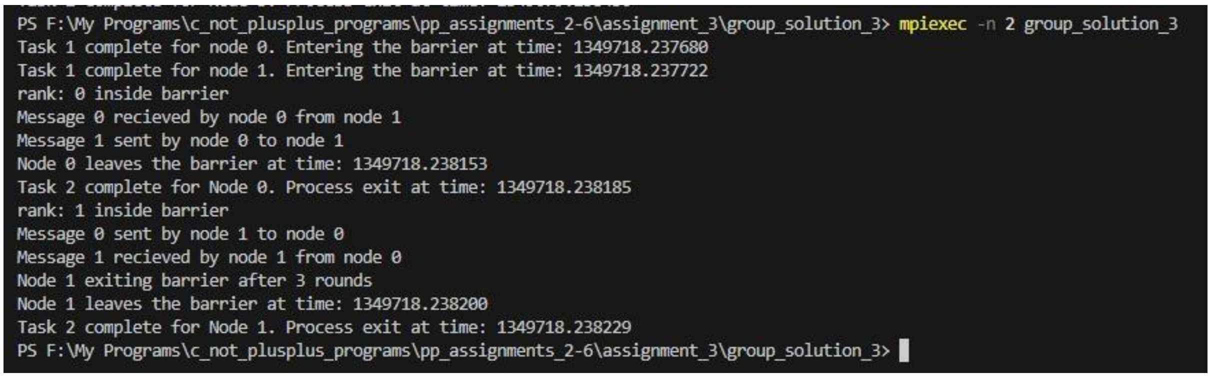
```c
// Root Receive messages from below
// No children case
if(child_1 == 0 && child_2 == 0) {
    printf("Node %d exiting barrier after %d rounds \n", rank, round);

}
// Else recieve messages from one or 2 children below and return
else if(child_1 == 0){
    MPI_Recv(&round, 1, MPI_INT, child_2, 0, comm, &status);
    round = round +1;
    printf("Message 0 recieved by node %d from node %d \n", rank, child_2);
    MPI_Send(&round, 1, MPI_INT, child_2, 1, comm);
    printf("Message 1 sent by node %d to node %d \n", rank, child_2);

}
else if(child_2 == 0){
    MPI_Recv(&round, 1, MPI_INT, child_1, 0, comm, &status);
    round = round +1;
    tag = tag +1;

    printf("Message 0 recieved by node %d from node %d \n", rank, child_1);
    MPI_Send(&round, 1, MPI_INT, child_1, 1, comm);
    printf("Message 1 sent by node %d to node %d \n", rank, child_1);
}
else {
    MPI_Recv(&round, 1, MPI_INT, child_1, 0, comm, &status);
    printf("Message 0 recieved by node %d from node %d \n", rank, child_1);
    MPI_Recv(&round, 1, MPI_INT, child_2, 0, comm, &status);
    printf("Message 0 recieved by node %d from node %d \n", rank, child_2);
    round = round +1;
    MPI_Send(&round, 1, MPI_INT, child_1, 1, comm);
    printf("Message 1 sent by node %d from node %d \n", rank, child_1);
    MPI_Send(&round, 1, MPI_INT, child_2, 1, comm);
    printf("Message 1 sent by node %d from node %d \n", rank, child_2);
}
```

Fig 5. Logic for root nodes to receive confirmations from children

Analogous to a binary search tree, dissemination of a message up or down a tree requires $\log_2(Nodes)$ steps happening in parallel. Since our implementation requires traversing up the tree to collect all entry confirmations then back down to disseminate the "all clear" message, our implementation should run in roughly $2\log_2(Nodes)$ steps. It would have been conceptually trivial to further shrink the growth rate of the steps required by modelling the algorithm after a tree with more leaves. For example, a tree with 10 leaves would require in roughly $2\log_{10}(Nodes)$ steps in parallel to accomplish the message dissemination. Using a larger leaf base like this makes sense for a distributed application. Sending 10 or even 100 messages is still fairly trivial to a machine but allows a distributed process to scale to millions or billions of nodes with only a few parallel steps. This model is not particularly taxing on any node since each node must only communicate twice with its singular parent and the children you assign it.

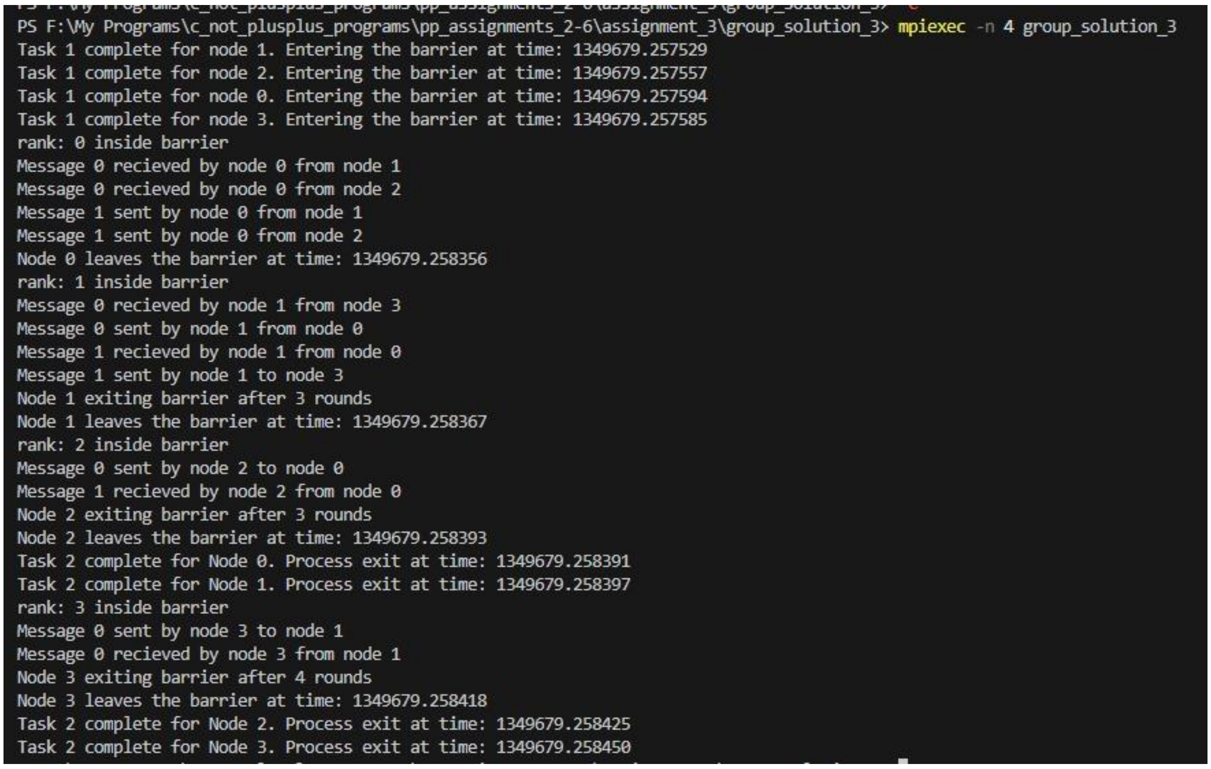
Below you can see the output from running our program. You will notice that with 5 processes, the log base 2 of 5 is 2 with a remainder of 1. We would expect a maximum of 4 rounds with this number of processes and that is what we see. You'll also note that the time of barrier entry for every node is less than time to enter the next task for any node meaning the barrier succeeded in making all processes wait until all had arrived before proceeding.

Result:



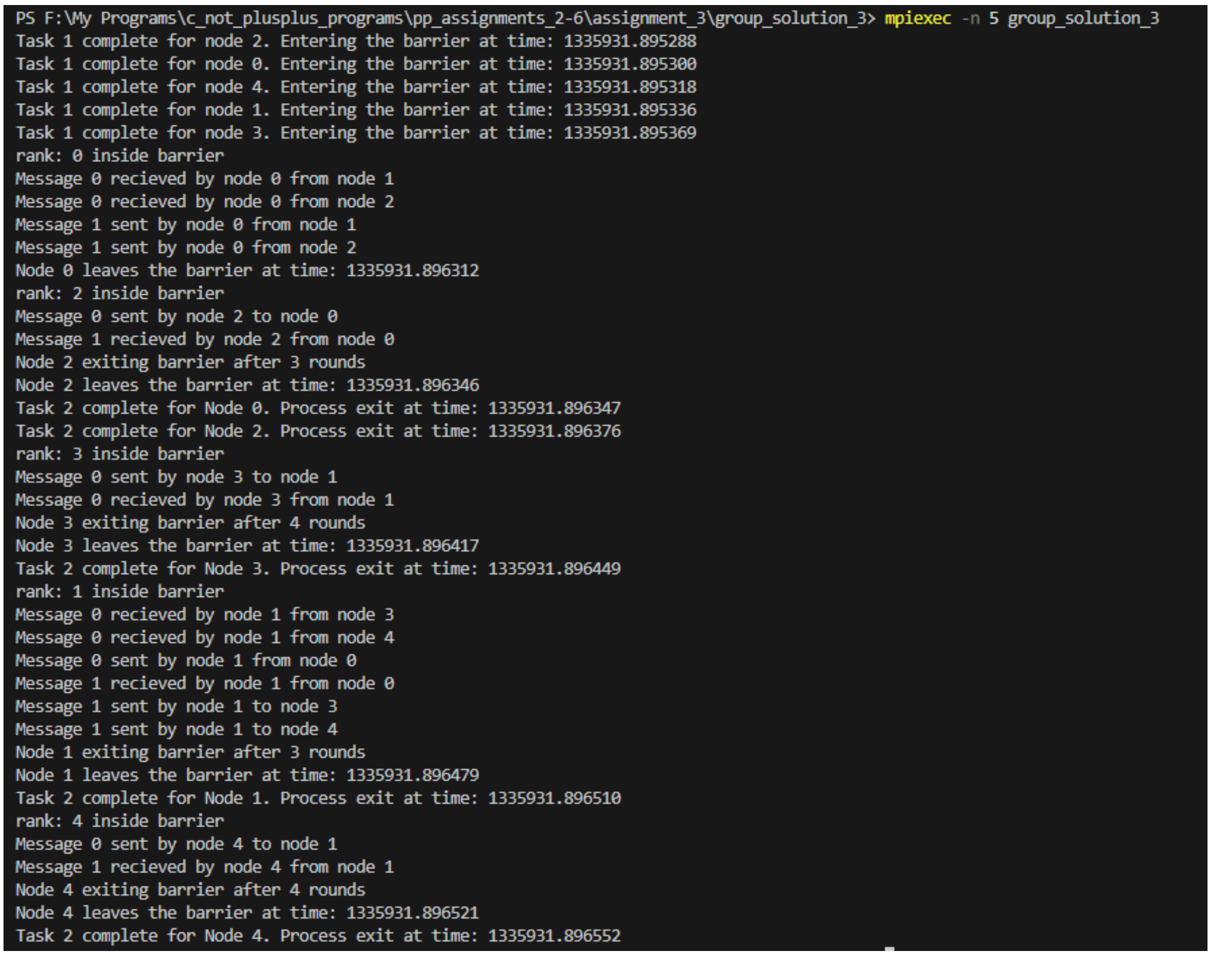
```
PS F:\My Programs\c_not_plusplus_programs\pp_assignments_2-6\assignment_3\group_solution_3> mpiexec -n 2 group_solution_3
Task 1 complete for node 0. Entering the barrier at time: 1349718.237680
Task 1 complete for node 1. Entering the barrier at time: 1349718.237722
rank: 0 inside barrier
Message 0 recieved by node 0 from node 1
Message 1 sent by node 0 to node 1
Node 0 leaves the barrier at time: 1349718.238153
Task 2 complete for Node 0. Process exit at time: 1349718.238185
rank: 1 inside barrier
Message 0 sent by node 1 to node 0
Message 1 recieved by node 1 from node 0
Node 1 exiting barrier after 3 rounds
Node 1 leaves the barrier at time: 1349718.238200
Task 2 complete for Node 1. Process exit at time: 1349718.238229
PS F:\My Programs\c_not_plusplus_programs\pp_assignments_2-6\assignment_3\group_solution_3>
```

Fig 6. Output of running our solution with 2 processes



```
PS F:\My Programs\c_not_plusplus_programs\pp_assignments_2-6\assignment_3\group_solution_3> mpiexec -n 4 group_solution_3
Task 1 complete for node 1. Entering the barrier at time: 1349679.257529
Task 1 complete for node 2. Entering the barrier at time: 1349679.257557
Task 1 complete for node 0. Entering the barrier at time: 1349679.257594
Task 1 complete for node 3. Entering the barrier at time: 1349679.257585
rank: 0 inside barrier
Message 0 recieved by node 0 from node 1
Message 0 recieved by node 0 from node 2
Message 1 sent by node 0 from node 1
Message 1 sent by node 0 from node 2
Node 0 leaves the barrier at time: 1349679.258356
rank: 1 inside barrier
Message 0 recieved by node 1 from node 3
Message 0 sent by node 1 from node 0
Message 1 recieved by node 1 from node 0
Message 1 sent by node 1 to node 3
Node 1 exiting barrier after 3 rounds
Node 1 leaves the barrier at time: 1349679.258367
rank: 2 inside barrier
Message 0 sent by node 2 to node 0
Message 1 recieved by node 2 from node 0
Node 2 exiting barrier after 3 rounds
Node 2 leaves the barrier at time: 1349679.258393
Task 2 complete for Node 0. Process exit at time: 1349679.258391
Task 2 complete for Node 1. Process exit at time: 1349679.258397
rank: 3 inside barrier
Message 0 sent by node 3 to node 1
Message 0 recieved by node 3 from node 1
Node 3 exiting barrier after 4 rounds
Node 3 leaves the barrier at time: 1349679.258418
Task 2 complete for Node 2. Process exit at time: 1349679.258425
Task 2 complete for Node 3. Process exit at time: 1349679.258450
```

Fig 6. Output of running our solution with 4 processes



```
PS F:\My Programs\c_not_plusplus_programs\pp_assignments_2-6\assignment_3\group_solution_3> mpiexec -n 5 group_solution_3
Task 1 complete for node 2. Entering the barrier at time: 1335931.895288
Task 1 complete for node 0. Entering the barrier at time: 1335931.895300
Task 1 complete for node 4. Entering the barrier at time: 1335931.895318
Task 1 complete for node 1. Entering the barrier at time: 1335931.895336
Task 1 complete for node 3. Entering the barrier at time: 1335931.895369
rank: 0 inside barrier
Message 0 recieved by node 0 from node 1
Message 0 recieved by node 0 from node 2
Message 1 sent by node 0 from node 1
Message 1 sent by node 0 from node 2
Node 0 leaves the barrier at time: 1335931.896312
rank: 2 inside barrier
Message 0 sent by node 2 to node 0
Message 1 recieved by node 2 from node 0
Node 2 exiting barrier after 3 rounds
Node 2 leaves the barrier at time: 1335931.896346
Task 2 complete for Node 0. Process exit at time: 1335931.896347
Task 2 complete for Node 2. Process exit at time: 1335931.896376
rank: 3 inside barrier
Message 0 sent by node 3 to node 1
Message 0 recieved by node 3 from node 1
Node 3 exiting barrier after 4 rounds
Node 3 leaves the barrier at time: 1335931.896417
Task 2 complete for Node 3. Process exit at time: 1335931.896449
rank: 1 inside barrier
Message 0 recieved by node 1 from node 3
Message 0 recieved by node 1 from node 4
Message 0 sent by node 1 from node 0
Message 1 recieved by node 1 from node 0
Message 1 sent by node 1 to node 3
Message 1 sent by node 1 to node 4
Node 1 exiting barrier after 3 rounds
Node 1 leaves the barrier at time: 1335931.896479
Task 2 complete for Node 1. Process exit at time: 1335931.896510
rank: 4 inside barrier
Message 0 sent by node 4 to node 1
Message 1 recieved by node 4 from node 1
Node 4 exiting barrier after 4 rounds
Node 4 leaves the barrier at time: 1335931.896521
Task 2 complete for Node 4. Process exit at time: 1335931.896552
```

Fig 6. Output of running our solution with 5 processes

Overall, Our MPI program code demonstrates the implementation of customer mybarrier () function to synchronize a parallelized process in logarithmic steps relative to process distribution size. This model and its higher-leaved variants allow for wide-spread process distribution without letting any process become a communication bottleneck.