Assignment 1 Final

1. Introduction:

In assignment 1 final, the C program, ring.c, is developed to implement a parallelized MPI program for a set of tasks communicating in a ring topology. The primary goal is to demonstrate the use of MPI in a parallel programming environment while managing communication between tasks to avoid deadlocks. This report offers a comprehensive explanation of the left circular shift of data and the process of achieving an ordered output based on the task rank within the context of a ring topology.

2. The Left Circular Shift

Circular shift is a technique in which elements of an array or data structure are repositioned by moving a specific number of places, either to the left or right, while preserving their original order. When an element reaches the boundary (the beginning or the end of the array), it "loops around" to the opposite side and resumes shifting. The modulo operation plays a crucial role in ensuring that the values "loop around" within the ring topology, allowing for seamless circular shifting.

In the implementation of ring topology, each task is linked to its immediate left and right neighbours. The tasks at boundary are connected in a way that the left neighbour of the first task is the last task, and the right neighbour of the last task is the first task, thus forming a circular configuration. For instance, consider a scenario with four tasks numbered from 0 to 3. The ring topology can be visualized using the following diagram:

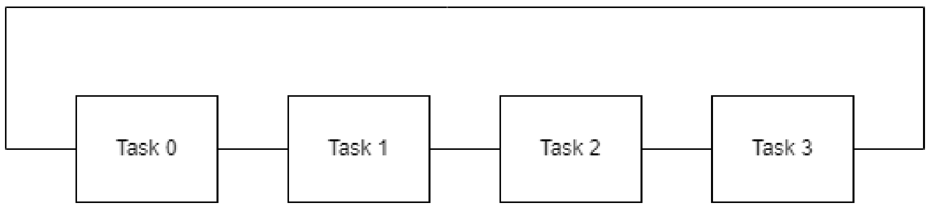


Figure 1: a ring topology with 4 tasks

When designing the left circular shift for parallel programming, it is important to account for the simultaneous execution of circular shifts by all tasks. Improper design may lead to deadlock situations. Deadlock occurs when each task in the system is waiting for other tasks to release resources or complete operations, forming a cycle of dependencies. The pseudocode for a single left circular shift is described as:

1. Send m-value to the left neighbour.
2. Receive m-value from the right neighbour.

Since each task initializes its m-value independently, the above pseudocode ensures that no deadlock occurs. Subsequently, we can move on to implement it in the C program. It is vital for a task to only communicates with its immediate neighbours. The left and right neighbours can be defined using the rank of the task and the total number of initialized tasks:

```
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int right = (world_rank + 1) % world_size;
int left = (world_rank - 1 + world_size) % world_size;
```

The modulo operation is employed to guarantee that the values "loop around" within the ring topology. By applying the modulo operation, we maintain neighbour values within the acceptable range of task ranks (0 to the total number of tasks - 1). This approach allows tasks at the ring topology boundaries to connected to their respective neighbours and avoid accessing invalid tasks. As a result, the pseudocode can be implemented as follows:

```
MPI_Send(&m, 1, MPI_INT, left, 0, MPI_COMM_WORLD);
MPI_Recv(&m, 1, MPI_INT, right, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

To perform a left circular shift of three places, we utilize a for-loop that iterates three times for each task, executing the left circular shift operation during each iteration. This can be implemented as follows:

```
for (int i = 0; i < 3; i++)
{
    MPI_Send(&m, 1, MPI_INT, left, 0, MPI_COMM_WORLD);
    MPI_Recv(&m, 1, MPI_INT, right, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

3. Strategy for Achieving Ordered Output

Each task is required to output its m-value after performing the left circular shift three times. Furthermore, these outputs must be ordered based on the rank of the task, and only point-to-point communication is allowed. To distinguish between messages sent during the circular shift and those sent for ordered output, we use different message tags. This approach helps to avoid confusion and ensures proper communication between tasks.

When tasks independently output their m-values, the order of the output is unpredictable due to MPI tasks competing for access to the shared output device. To address this issue, we designate a single task as the master, responsible for handling m-value output and ordering it based on the rank. Work partitioning plays a key role in achieving ordered output, as it helps to manage the tasks' responsibilities more effectively. In the implementation, task 0 is chosen as the master task, while the other tasks act as workers. The master task collects the m-values from the worker tasks, sorts them by rank, and then prints the m-values in order. This division of labour ensures that the program can run efficiently and avoids potential issues related to simultaneous access to shared resources or output devices.

For communication between tasks during the left circular shift, we use a message tag of 0. In contrast, a message tag of 1 is used when collecting m-values for ordered output. By utilizing different message tags, the master task can accurately identify and process messages associated with ordered output.

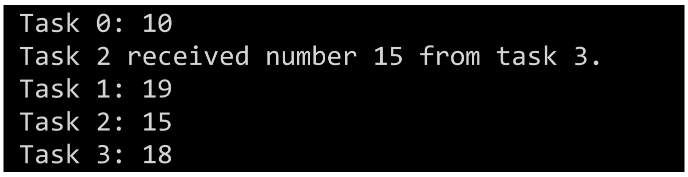
To achieve this, all worker tasks send their m-values as messages using MPI_Send() with a message tag of 1 to the master. The master then collects these messages using MPI_Receive() in a for-loop. An if-else clause is used to divide the work between the master and the workers. Only the master task handles the collection, ordering and printing while workers would only need to send the m-value to the master. The corresponding code can be implemented as follows:

```
if (world_rank == 0)
{
    for (int i = 0; i < world_size; i++) {
        if (i == 0) {
            printf("Task %d: %d\n", i, m);
        } else {
            int m_received;
            MPI_Recv(&m_received, 1, MPI_INT, i, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            printf("Task %d: %d\n", i, m_received);
        }
    }
} else {
    MPI_Send(&m, 1, MPI_INT, master, 1, MPI_COMM_WORLD);
}
```

4. Program Output:

Below is the output from an execution of the ring.c program with four tasks:

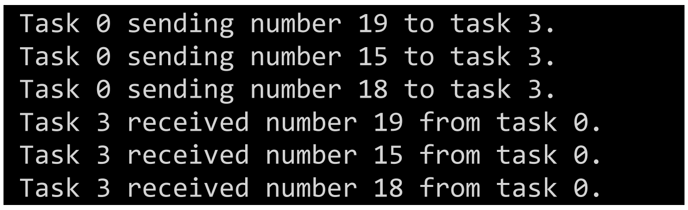
```
Task 0 sending number 19 to task 3.
Task 3 sending number 10 to task 2.
Task 3 received number 19 from task 0.
Task 3 sending number 19 to task 2.
Task 2 sending number 18 to task 1.
Task 2 received number 10 from task 3.
Task 2 sending number 10 to task 1.
Task 2 received number 19 from task 3.
Task 2 sending number 19 to task 1.
Task 0 received number 15 from task 1.
Task 0 sending number 15 to task 3.
Task 0 received number 18 from task 1.
Task 0 sending number 18 to task 3.
Task 1 sending number 15 to task 0.
Task 1 received number 18 from task 2.
Task 1 sending number 18 to task 0.
Task 1 received number 10 from task 2.
Task 1 sending number 10 to task 0.
Task 1 received number 19 from task 2.
Task 3 received number 15 from task 0.
Task 3 sending number 15 to task 2.
Task 3 received number 18 from task 0.
Task 0 received number 10 from task 1.
Final m-values after left circular shift:
```



```
Task 0: 10
Task 2 received number 15 from task 3.
Task 1: 19
Task 2: 15
Task 3: 18
```

Figure 2: Output of the ring.c with 4 tasks

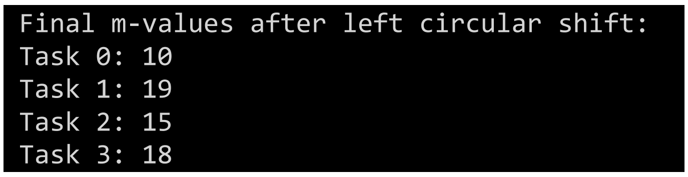
The communication record between tasks 0 and 3 demonstrates the left circular shift operation within the ring topology. Task 0 sends its m-value to task 3 (its left neighbour), and task 3 receives the m-value from task 0 (its right neighbour). This pattern aligns with the left circular shift operation, where each task sends its m-value to its left neighbour and receives an m-value from its right neighbour.



```
Task 0 sending number 19 to task 3.
Task 0 sending number 15 to task 3.
Task 0 sending number 18 to task 3.
Task 3 received number 19 from task 0.
Task 3 received number 15 from task 0.
Task 3 received number 18 from task 0.
```

Figure 3: Extracted record of communication, showing only the interactions between tasks 0 and 3.

The ordered output of m-values shows the result of the left circular shift operation performed by the tasks within the ring topology. After executing the shift three places to the left, the final m-values for each task are presented in an ordered manner based on their task ranks. The output indicates that the program successfully completed the left circular shift and achieved the ordering of the m-values based on rank.



```
Final m-values after left circular shift:
Task 0: 10
Task 1: 19
Task 2: 15
Task 3: 18
```

Figure 4: Extracted copy of the ordered output of m-values.

5. Conclusion:

In conclusion, this report has provided a detailed explanation of the left circular shift operation and the process of achieving ordered output using MPI in a parallel programming environment with a ring topology. The importance of the modulo operation for correctly connecting tasks at the boundaries of the ring is discussed, as well as the use of work partitioning and distinct message tags to ensure smooth communication and proper division of labour among tasks. The successful implementation of the left circular shift and ordered output is highlighted through the program output, demonstrating effective communication between tasks and the elimination of potential deadlocks.