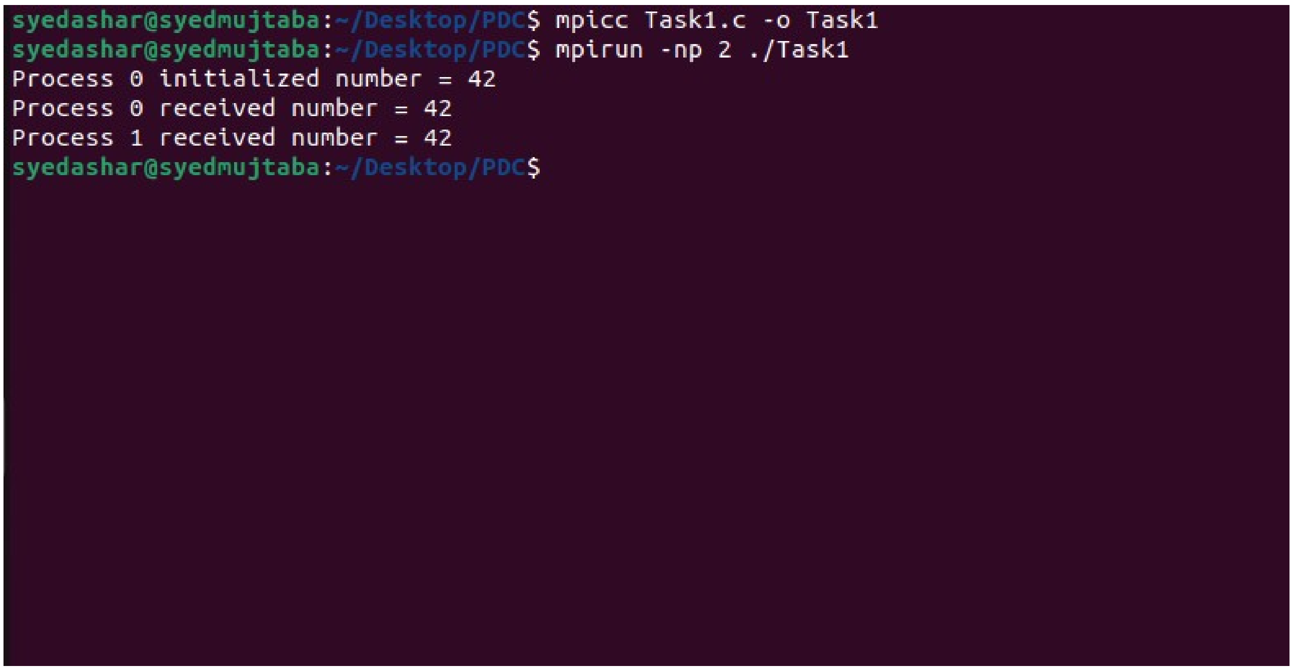
Parallel and Distributing Computing

Name **Noman Shahid**
Roll no **22P-9029**
Section **BCS-6A**

Assignment # 3

Task 1



```
syedashar@syedmujtaba:~/Desktop/PDC$ mpicc Task1.c -o Task1
syedashar@syedmujtaba:~/Desktop/PDC$ mpirun -np 2 ./Task1
Process 0 initialized number = 42
Process 0 received number = 42
Process 1 received number = 42
syedashar@syedmujtaba:~/Desktop/PDC$
```

Questions

1. What happens if a non-root process changes the value before the broadcast?

If a non-root process changes the variable *before* the broadcast, its value will be overwritten by the broadcasted value from the **root process**.

Eaxample

```
    if (rank != 0) {
  number = 999;        // This value will be overwritten
}
```

MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);

2. What constraints must be followed when calling `MPI_Bcast`?

- All processes in the communicator must call `MPI_Bcast` (otherwise, it will hang).
- The **data type and count** must match across all processes.
- The **buffer variable (e.g., `number`) must exist** on all processes.

Good

```
MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Bad

```
    // This may hang or crash
if (rank == 0)
   MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);
// Other processes must also call this line
```

3. How does `MPI_Bcast` differ from point-to-point send/receive?

Feature	**`MPI_Bcast`**	**`MPI_Send` + `MPI_Recv`**
Communication Type	One-to-many (from one to all)	One-to-one
Code Simplicity	One line for all processes	Multiple lines (send for each recv)
Performance	Optimized for broadcast internally	More overhead

Feature	`MPI_Bcast`	`MPI_Send` + `MPI_Recv`
Usage Example	Sharing config/initial data	Custom messaging between processes

Task 2

Adaptation Note:

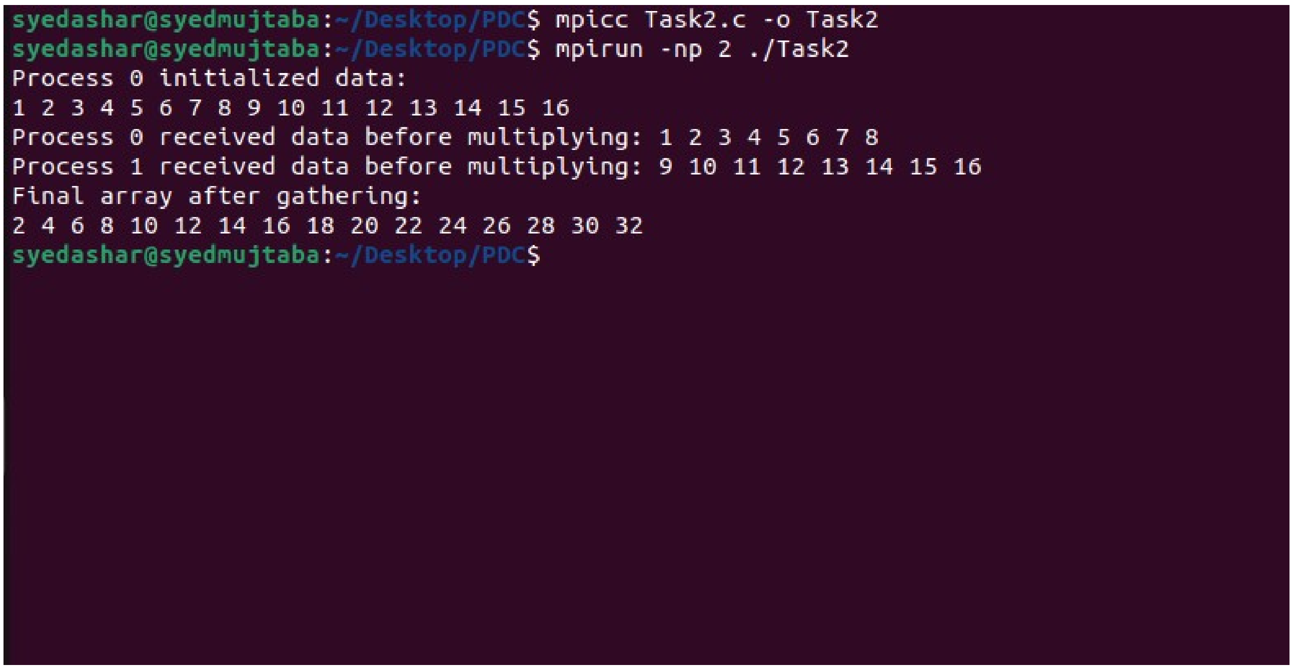
Due to hardware limitations allowing only 2 parallel processes, this implementation:

•Uses **2** processes instead of 4

•Maintains the original 16-element array size

•Divides data into 8-element chunks per process (16 ÷ 2)

Reaming Flow of the program is same



```
syedashar@syedmujtaba:~/Desktop/PDC$ mpicc Task2.c -o Task2
syedashar@syedmujtaba:~/Desktop/PDC$ mpirun -np 2 ./Task2
Process 0 initialized data:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Process 0 received data before multiplying: 1 2 3 4 5 6 7 8
Process 1 received data before multiplying: 9 10 11 12 13 14 15 16
Final array after gathering:
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
syedashar@syedmujtaba:~/Desktop/PDC$
```

Questions

a. What will happen if the number of processes is not evenly divisible by the data size?

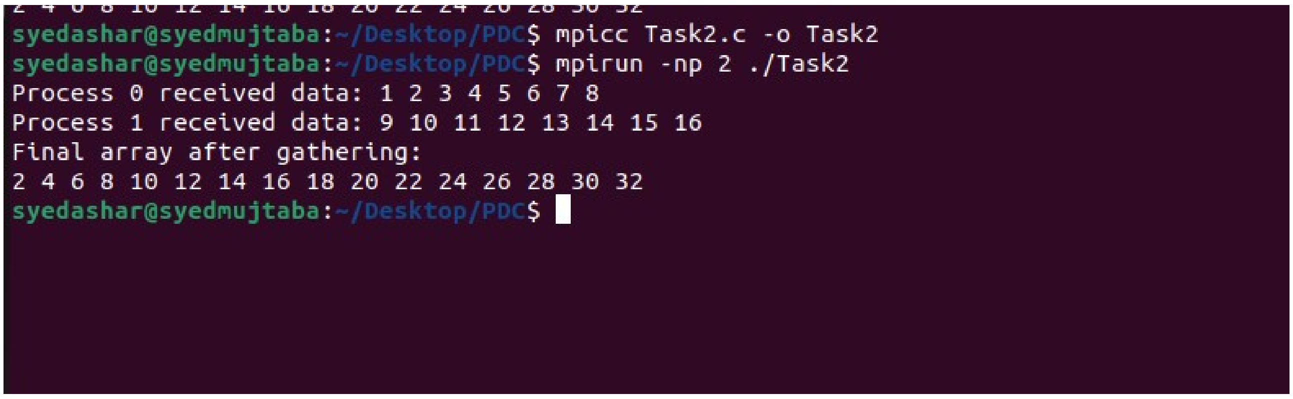
If the **number of processes** is not evenly divisible by the **data size**, the `MPI_Scatter` operation will encounter an issue. Specifically:

- `MPI_Scatter` divides the total number of elements across processes evenly. If the data size cannot be evenly divided by the number of processes, some processes may end up receiving fewer elements than others, leading to errors or undefined behavior.

For example:

- If you have 16 integers but only 3 processes, each process would normally get 5 or 6 integers. But MPI cannot automatically decide how to handle the remaining elements when the division isn't even.

b. Modify the program to handle uneven distribution using dynamic offsets (optional).

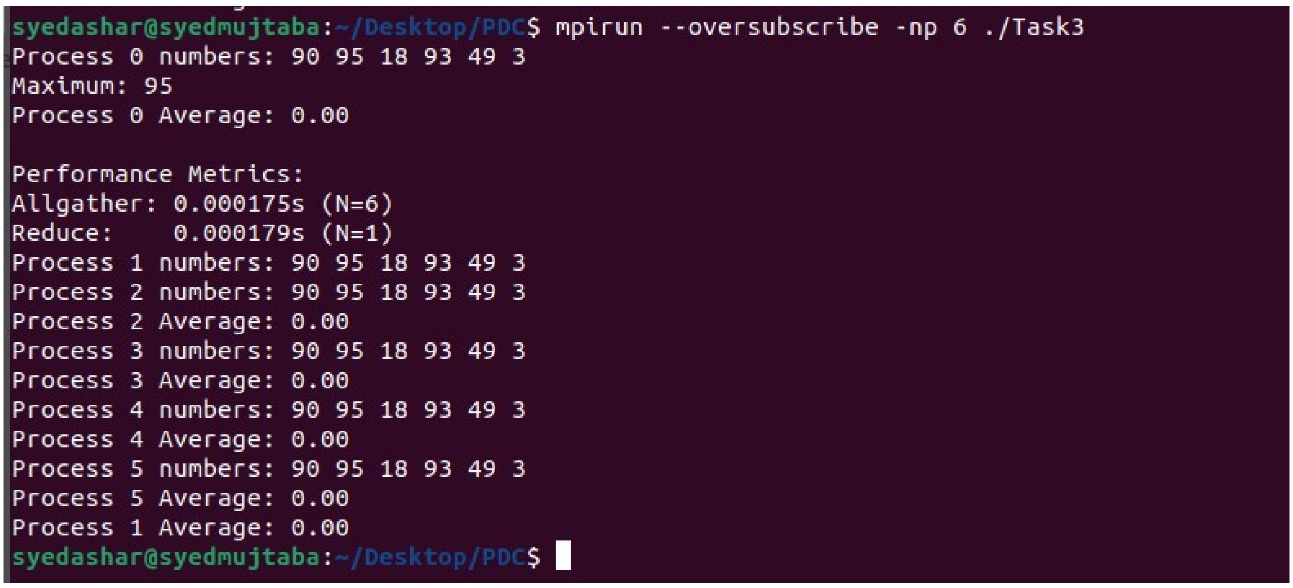


```
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
syedashar@syedmujtaba:~/Desktop/PDC$ mpicc Task2.c -o Task2
syedashar@syedmujtaba:~/Desktop/PDC$ mpirun -np 2 ./Task2
Process 0 received data: 1 2 3 4 5 6 7 8
Process 1 received data: 9 10 11 12 13 14 15 16
Final array after gathering:
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
syedashar@syedmujtaba:~/Desktop/PDC$
```

Steps to Handle Uneven Distribution:

1. **Calculate the Base Chunk Size:**
   - **Chunk size** for each process = `total_data_size / num_processes`.
   - **Extra elements** = `total_data_size % num_processes`.
2. **Distribute Extra Elements:**
   - First `extra` processes receive **1 additional element** to balance the load.
3. **Dynamic Calculation for `MPI_Scatter`:**
   - Use `chunk_size + 1` for processes that receive an extra element.
   - Adjust `sendcount` dynamically to handle uneven data distribution.

Task 3



```
syedashar@syedmujtaba:~/Desktop/PDC$ mpirun --oversubscribe -np 6 ./Task3
Process 0 numbers: 90 95 18 93 49 3
Maximum: 95
Process 0 Average: 0.00

Performance Metrics:
Allgather: 0.000175s (N=6)
Reduce:    0.000179s (N=1)
Process 1 numbers: 90 95 18 93 49 3
Process 2 numbers: 90 95 18 93 49 3
Process 2 Average: 0.00
Process 3 numbers: 90 95 18 93 49 3
Process 3 Average: 0.00
Process 4 numbers: 90 95 18 93 49 3
Process 4 Average: 0.00
Process 5 numbers: 90 95 18 93 49 3
Process 5 Average: 0.00
Process 1 Average: 0.00
syedashar@syedmujtaba:~/Desktop/PDC$
```

a. MPI_Allgather Cost:

Allgather requires all-to-all communication (O(N^2) complexity) where every process receives data from every other process, unlike Gather which only targets one process.

b. Allreduce vs Reduce+Broadcast:

Yes, MPI_Allreduce combines reduction and broadcast in one optimized operation, avoiding the synchronization overhead of separate calls.

c. Large Data Challenges:
With 1M-element arrays:

•Memory becomes constrained

•Network bandwidth limits throughput

•Synchronization overhead dominates

•Buffer management complexity increases

•Latency impacts scaling efficiency