

# Parallel and Distributing Computing

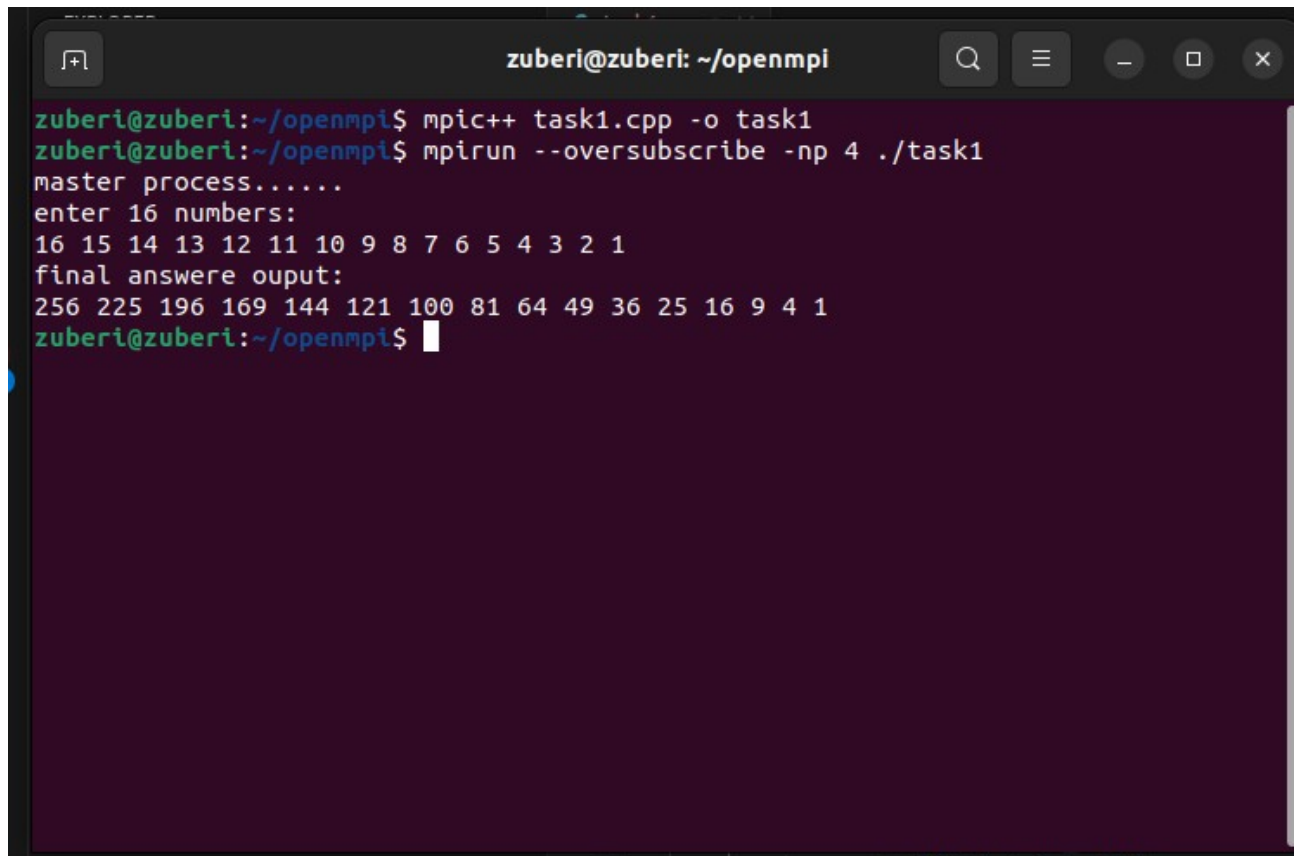
## Assignment no. 2

Name Noman Shahid

Roll No. 22P\_9029

Section BCS\_6A

## Task 1

A terminal window titled 'zuberi@zuberi: ~/openmpi' showing the execution of an MPI program. The user enters 'mpic++ task1.cpp -o task1' to compile and 'mpirun --oversubscribe -np 4 ./task1' to run with 4 processes. The program output shows a master process receiving 16 numbers (16 down to 1) and printing a final output of 16 numbers (256 down to 1).

```
zuberi@zuberi:~/openmpi$ mpic++ task1.cpp -o task1
zuberi@zuberi:~/openmpi$ mpirun --oversubscribe -np 4 ./task1
master process.....
enter 16 numbers:
16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
final answer output:
256 225 196 169 144 121 100 81 64 49 36 25 16 9 4 1
zuberi@zuberi:~/openmpi$
```

a. How is workload distribution affected by the number of processes?

Ans:

workload is splitted as even as possible between worker processes (size - 1). every worker gets around:

$\text{floor}(16 / (\text{size} - 1))$  elements

if there's any leftover (remainder), the first few processes takes one extra element. this helps to keep all workers busy.

still, sometimes the work isn't shared totally equally, like when 16 isn't divisible by the number of workers exactly. in that case, some get more than others.

**b. Can this design scale for larger arrays? Why or why not?**

Ans:

This design can scale with bigger arrays but it has limits. as the array gets bigger, the work is still split by the master, and if there are too many workers, some might not get enough work. this can lead to wasted time. also, if the array size isn't exactly divisible by (size - 1), some workers get more work than others, causing imbalance. so, it scales but not perfectly without better load balancing or optimizations.

## Task 2

```
zuberi@zuberi:~/openmpi$ mpic++ task1.cpp -o task1
zuberi@zuberi:~/openmpi$ mpirun --oversubscribe -np 4 ./task1

16 integers
[process 1] received 4 elements: 5 6 7 8
[process 1] computed squares: 25 36 49 64
[process 1] sent back 4 elements
[process 2] received 4 elements: 9 10 11 12
[process 2] computed squares: 81 100 121 144
[process 2] sent back 4 elements
[process 3] received 4 elements: 13 14 15 16
[process 3] computed squares: 169 196 225 256
[process 3] sent back 4 elements
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

master distributing data
[master] sent 4 elements to process1: 5 6 7 8
[master] sent 4 elements to process2: 9 10 11 12
[master] sent 4 elements to process3: 13 14 15 16

[master] all data distribution complete!

master collecting results

[master] all results received!

final result
1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256
zuberi@zuberi:~/openmpi$
```

**Questions:**

**a. Explain why MPI\_Wait all is needed.**

Ans:

MPI\_Waitall ensures all non-blocking operations complete before proceeding. It:

- Guarantees data safety (prevents using incomplete buffers)
- Maintains correct execution order
- Frees system resources tied to pending requests

**b. What happens if you omit waiting for non-blocking messages? Simulate it and report**

**Ans:**

Without waiting:

- Data corruption: Buffers may be overwritten before transfers finish
- Undefined behavior: Processes might read garbage/unreceived data
- Program hangs: Pending operations can block indefinitely

## Task 3

```
zuberi@zuberi:~/openmpi$ mpic++ task1.cpp -o task1
zuberi@zuberi:~/openmpi$ mpirun --oversubscribe -np 4 ./task1
process3 received squared data from process1with tag3000
process3 received cubed data from process2with tag4000
final result sat process3:
squares:1 4 9 16 25
process0 sent array1 to process1 and array2 to process2
process1 received data from process0with tag1000
process1 sent squared data to process3
process2 received data from process0with tag2000
process2 sent cubed data to process3
cubes:1000 8000 27000 64000 125000
zuberi@zuberi:~/openmpi$
```

**Questions:**

**a. How do message tags help in handling multiple simultaneous messages?**

**Ans:**

Tags let processes **identify and differentiate** messages. They help match sends and receives correctly, even when multiple messages are in transit. This prevents confusion and supports organized communication.

**Example:**

If process 0 sends array A with tag 100 and array B with tag 200 to process 1, process 1 can selectively receive A first, then B, even if B arrives first physically.

**b. What can go wrong if two messages arrive with the same tag from different sources?**

**Ans:**

If two messages with the **same tag** come from **different sources**, and the receiver uses `MPI_ANY_SOURCE`, it may receive them in **unintended order** or **misinterpret** data. This can cause logic errors or buffer issues.

**Risks:**

- You might **mix up data** if you're assuming one message came from a specific process.
- It can lead to **logic errors**, especially if the payloads differ in size or meaning.

## Task 4

```
zuberi@zuberi: ~/openmpi
zuberi@zuberi:~/openmpi$ mpic++ task1.cpp -o task1
zuberi@zuberi:~/openmpi$ mpirun --oversubscribe -np 4 ./task1
process 3: passed counter 3
process 1: passed counter 1
process 2: passed counter 2
process 2: passed counter 6
process 2: received termination
process 0: terminated after 2 cycles
process 3: passed counter 7
process 3: received termination
process 1: passed counter 5
process 1: received termination
zuberi@zuberi:~/openmpi$
```

## Questions

a. What are common pitfalls in ring-based communication?

Ans:

**Deadlocks:** If processes do not send or receive messages in a proper sequence, the communication might get stuck, and the processes may wait indefinitely. For example, if the message passing does not follow the correct order, some processes could wait for others to send messages, causing a deadlock.

**Example:** In the given code, if Process 0 does not send a message after receiving it, Process 1 may wait forever.

b. What would be different if communication was bi-directional? Implement and test.

Ans:

If communication were bi-directional (i.e., processes communicate with both their neighbors), it would eliminate some of the constraints imposed by a unidirectional ring.

**For Example :**

- **Two-way synchronization:** Processes could send and receive data in both directions, potentially reducing the overall time needed for data exchange.

## Task 5

```
zuberi@zuberi: ~/openmpi
zuberi@zuberi:~/openmpi$ mpic++ task1.cpp -o task1
zuberi@zuberi:~/openmpi$ mpirun --oversubscribe -np 4 ./task1
Process 0: Starting blocking communication
Process 0: Blocking communication time: 0.000141 seconds
Process 0: Starting non-blocking communication
Process 0: Non-blocking communication time: 0.000106 seconds
zuberi@zuberi:~/openmpi$
```

## A

Communication Type	Time (seconds)
--------------------	----------------

Blocking	0.000141
----------	----------

Non-blocking	0.000106
--------------	----------

### Key Observations:

Non-blocking communication is 24.8% faster than blocking in this test.

- The speedup comes from overlapping computation and communication in non-blocking mode.

## B

### **Synchronization Overhead (Short):**

- Barriers: Force all processes to wait for the slowest one.
- Blocking: Sequential sends/receives create idle wait time.
- Non-blocking: MPI\_Wait/MPI\_Waita still require coordination.
- MPI Internals: Message tracking and buffer management add latency.

*Result:* Even non-blocking has overhead, but less than blocking due to reduced idle time.