**Parallel Programming Lab**

**Practical No. 7**

| **Name: Aditi Mittal**  **PRN: 23610067**  **Class: TY CSE**  **Batch: T1** |
| --- |

**Title of practical:**

Introduction to MPI: simple point-to-point communication using MPI\_Send and MPI\_Recv.

# MPI vs OpenMP

### 1. Programming Model

* **MPI (Message Passing Interface)**
  + Based on **multiple processes**.
  + Each process has its **own memory space** (distributed memory).
  + Processes communicate by **sending/receiving messages**.
  + Works across **clusters (multiple machines)** as well as single multicore machines.
* **OpenMP (Open Multi-Processing)**
  + Based on **multiple threads** inside a single process.
  + Threads share the **same memory space** (shared memory).
  + Uses **pragmas/directives** (like #pragma omp parallel) in C/C++ or Fortran.
  + Works best on a **single machine with shared memory**.

### 2. Memory Model

* **MPI → Distributed Memory**
  + Each process has its own local variables.
  + Must explicitly send/receive data to share.
  + Suitable for clusters and supercomputers.
* **OpenMP → Shared Memory**
  + All threads see the same global memory.
  + Synchronization (locks, barriers) is required to avoid race conditions.
  + Works only on a single node (unless combined with MPI).

### 3. Ease of Use

* **MPI**
  + More complex: programmer must handle communication explicitly (MPI\_Send, MPI\_Recv, etc.).
  + But more **scalable** across machines.
* **OpenMP**
  + Easier: just add compiler directives (#pragma omp parallel for).
  + But limited to one machine’s memory.

**Problem Statement 1:**

Implement a simple hello world program by setting number of processes equal to 10

**Code:**

/\*

MPI INstallation on Fedora

sudo dnf install openmpi openmpi-devel

echo 'export PATH=/usr/lib64/openmpi/bin:$PATH' >> ~/.bashrc

echo 'export LD\_LIBRARY\_PATH=/usr/lib64/openmpi/lib:$LD\_LIBRARY\_PATH' >> ~/.bashrc

source ~/.bashrc

mpicc --version

mpiexec --version

To run the code:

mpic++ Q1.cpp -o Q1

mpiexec -n 10 ./Q1

\*/

/\*

Q1. Implement a simple hello world program by setting number of processes equal to 10

\*/

#include <mpi.h>

#include <iostream>

using namespace std;

int main(int argc, char\*\* argv) {

MPI\_Init(&argc, &argv);

int world\_size;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

int world\_rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

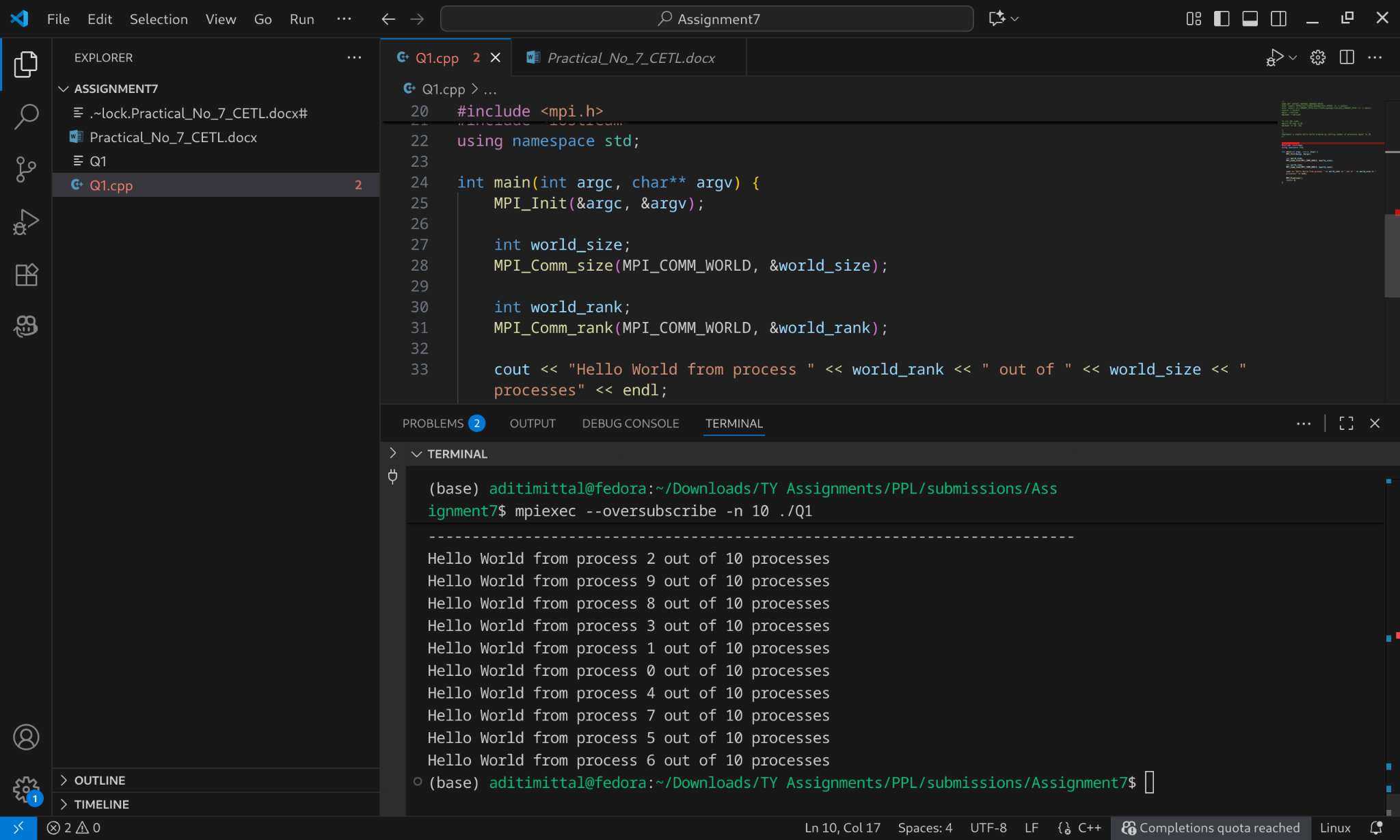
cout << "Hello World from process " << world\_rank << " out of " << world\_size << " processes" << endl;

MPI\_Finalize();

return 0;

}

**Screenshots:**

****

## What is MPI?

MPI (Message Passing Interface) is a standard for parallel programming that allows multiple processes (running on one or more computers) to communicate with each other by sending and receiving messages.

* It is not a language but a specification (standard).
* Implementations include OpenMPI, MPICH, etc.
* MPI is widely used in High-Performance Computing (HPC) for scientific simulations, engineering, data analysis, and supercomputing.

## Why MPI?

* Computers have multiple cores and clusters have many nodes.
* To use them efficiently, we need a way for processes to work together.
* Unlike multithreading (OpenMP, pthreads), MPI works well for distributed memory systems where each process has its own memory.

## Key Ideas

1. **Processes not Threads**
   * MPI uses multiple processes (not threads) → each has its own memory.
   * They communicate by passing messages.
2. **SPMD Model (Single Program, Multiple Data)**
   * All processes run the same program.
   * Each process knows its rank (ID) and can do different tasks based on that.
3. **Communicators**
   * A group of processes that can talk to each other.
   * The default is MPI\_COMM\_WORLD (all processes).

## Common MPI Functions

* **MPI\_Init →** Start MPI environment.
* MPI\_Comm\_size → Get number of processes.
* MPI\_Comm\_rank → Get process ID (rank).
* MPI\_Send / MPI\_Recv → Send and receive messages.
* MPI\_Bcast, MPI\_Reduce, MPI\_Gather, etc. → Collective communication operations.
* MPI\_Finalize → End MPI environment.

When you run with 10 processes: Each process runs the same code but prints its rank.

## Where is MPI used?

* Scientific computing (weather forecasting, molecular dynamics, physics simulations).
* Engineering (aerospace simulations, fluid dynamics).
* Big data & AI (parallel training, distributed computations).
* Any situation where we need lots of processes working together efficiently.

mpic++ Q1.cpp -o Q1 → uses the MPI compiler wrapper, which automatically links MPI libraries.

mpiexec -n 10 ./Q1 → launches 10 parallel processes of the program.

**Errors encountered:**

When running **mpiexec -n 10 ./Q1**

**Error:** There are not enough slots available in the system to satisfy the 10 slots that were requested

**Why this happened:**

* On my machine, nproc showed **8 cores**.
* By default, OpenMPI (PRRTE runtime) only allows **one MPI process per core** (1 slot = 1 core).
* When requested **10 processes**, but the runtime sees only **8 slots**, so it refuses to run.
* If MPI allowed it blindly, some processes would compete for CPU time, which could slow things down — that’s why it warns.

## What --oversubscribe Does

* --oversubscribe tells OpenMPI:  
    
    
   "I know I’m asking for more MPI processes than available cores. It’s okay, go ahead and run them."
* Essentially, it **ignores the slot limit** and lets multiple processes share the same CPU cores.
* Syntax:

mpiexec --oversubscribe -n 10 ./Q1

* Now OpenMPI will launch all 10 processes **even though you have only 8 cores**. Some cores will run more than one process (time-sharing).

The program works, but note:

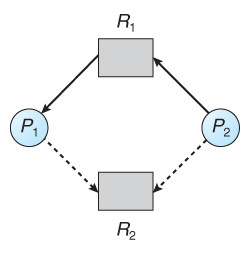
* Performance may drop slightly because processes share cores.
* On a real cluster with enough cores per node, you **usually don’t need oversubscribe**.

**Q2: Implement a MPI program to give an example of Deadlock.**

## What is a Deadlock?

A deadlock is a situation in parallel or concurrent programming where:

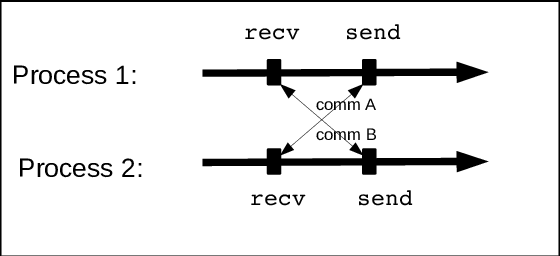
1. Two or more processes (or threads) are waiting for each other to release a resource,
2. And none of them can proceed,
3. So the program gets stuck forever.

****

### Deadlock in MPI

In MPI, a deadlock usually happens when:

1. Processes **wait for messages that never arrive**.
2. Example causes:  
   * Process 0 calls MPI\_Recv for a message from Process 1,
   * Process 1 calls MPI\_Recv for a message from Process 0 at the same time,
   * Both are now **waiting forever**.

****

**Program and screenshots**

/\*

Q2: Implement a MPI program to give an example of Deadlock.

\*/

/\*

mpic++ Q2.cpp -o Q2

mpiexec -n 2 ./Q2

\*/

#include <mpi.h>

#include <iostream>

using namespace std;

int main(int argc, char\*\* argv) {

MPI\_Init(&argc, &argv);

int rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

int msg;

if (rank == 0) {

// Process 0 waits to receive first

MPI\_Recv(&msg, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

MPI\_Send(&msg, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

} else if (rank == 1) {

// Process 1 also tries to receive first

MPI\_Recv(&msg, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

MPI\_Send(&msg, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

}

// both the processes wait to receive from each other endlessly and it causes a deadlock.

// Correct Way to communicate without deadlock:

// if (rank == 0) {

// MPI\_Send(&msg, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

// MPI\_Recv(&msg, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

// }

// else if (rank == 1) {

// MPI\_Recv(&msg, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

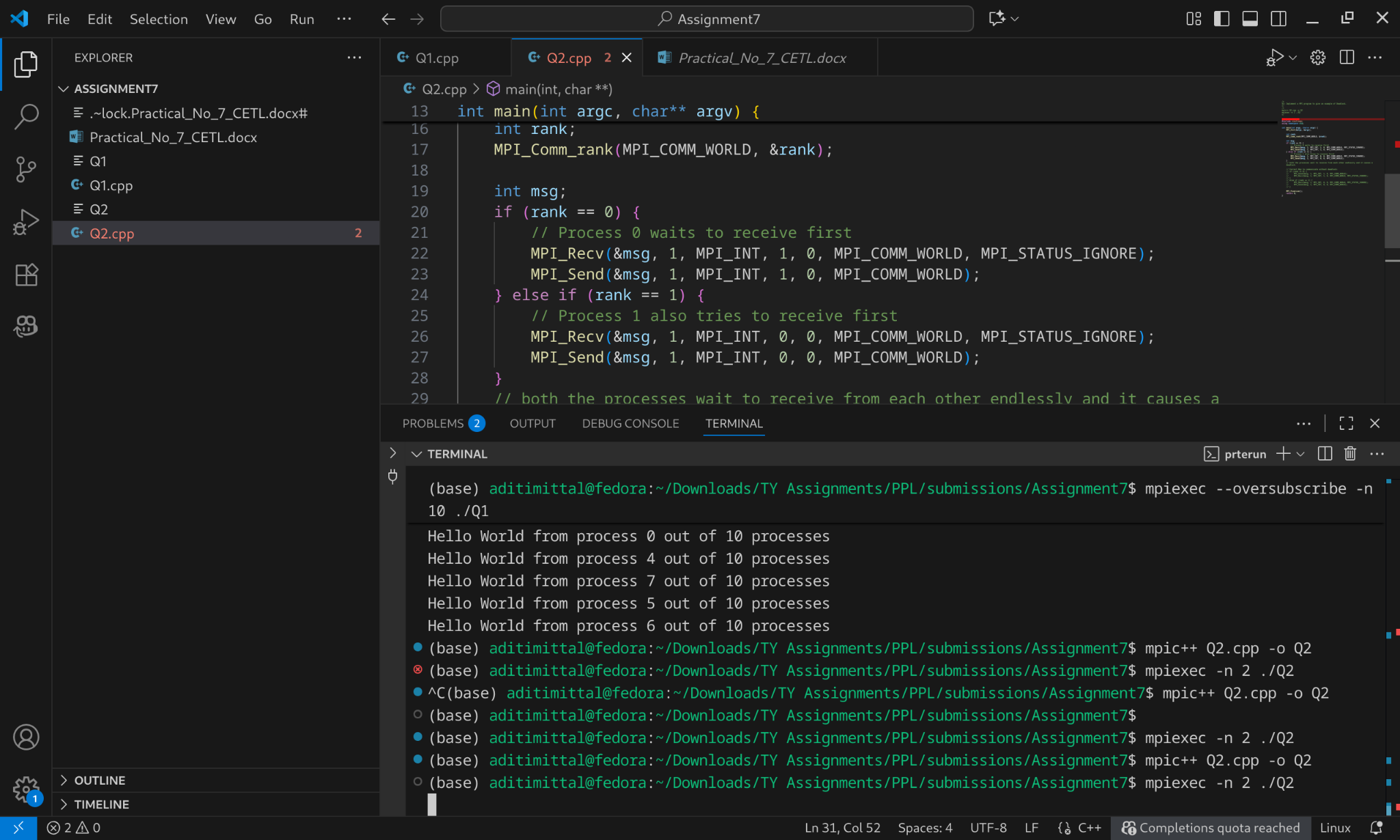
// MPI\_Send(&msg, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

// }

MPI\_Finalize();

return 0;

}

****

### Nothing prints to the console.

### Both processes appear to hang indefinitely.

### Why It Happens

1. Process 0 waits for a message from Process 1.
2. Process 1 waits for a message from Process 0.
3. Neither process can move forward because **both are stuck waiting**.
4. This is exactly what a **deadlock** is — processes cannot proceed.

If you swap Send/Recv order in one process, it **resolves the deadlock**.

### 

### Key Rules to Avoid Deadlocks in MPI

1. Match sends and receives: Every MPI\_Send should have a corresponding MPI\_Recv.
2. Avoid circular waits: Don’t have two processes waiting on each other simultaneously.
3. Use non-blocking communication: MPI\_Isend / MPI\_Irecv allows processes to continue while waiting.

Resource: <https://parallel.cs.jhu.edu/assets/slides/lec12.1.mpideadlock.pdf>

## What are MPI Send and Receive?

In **MPI (Message Passing Interface)**, processes have **separate memory spaces**. They **cannot directly access each other’s variables**.

So, to **exchange data**, MPI provides **communication functions**:

* MPI\_Send → Send data from one process to another.
* MPI\_Recv → Receive data from another process.

This is called **point-to-point communication**.

## MPI\_Send

**Syntax (C++):**

MPI\_Send(void\* data, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)

**Example:**

int x = 10;

MPI\_Send(&x, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD); // send x to process 1

## data: pointer to the data buffer to send.

## count: number of elements to send.

## datatype: type of data (MPI\_INT, MPI\_FLOAT, etc.).

## dest: rank of the receiving process.

## tag: message identifier.

## comm: communicator (MPI\_COMM\_WORLD for all processes).

## MPI\_Recv

**Syntax (C++):**

MPI\_Recv(void\* data, int count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm comm, MPI\_Status\* status)

* **data**: pointer to buffer to store received data.
* **count**: maximum number of elements to receive.
* **datatype**: type of data expected.
* **source**: rank of sending process (or MPI\_ANY\_SOURCE).
* **tag**: message tag to match (or MPI\_ANY\_TAG).
* **comm**: communicator.
* **status**: stores info about the received message (can use MPI\_STATUS\_IGNORE).

**Example:**

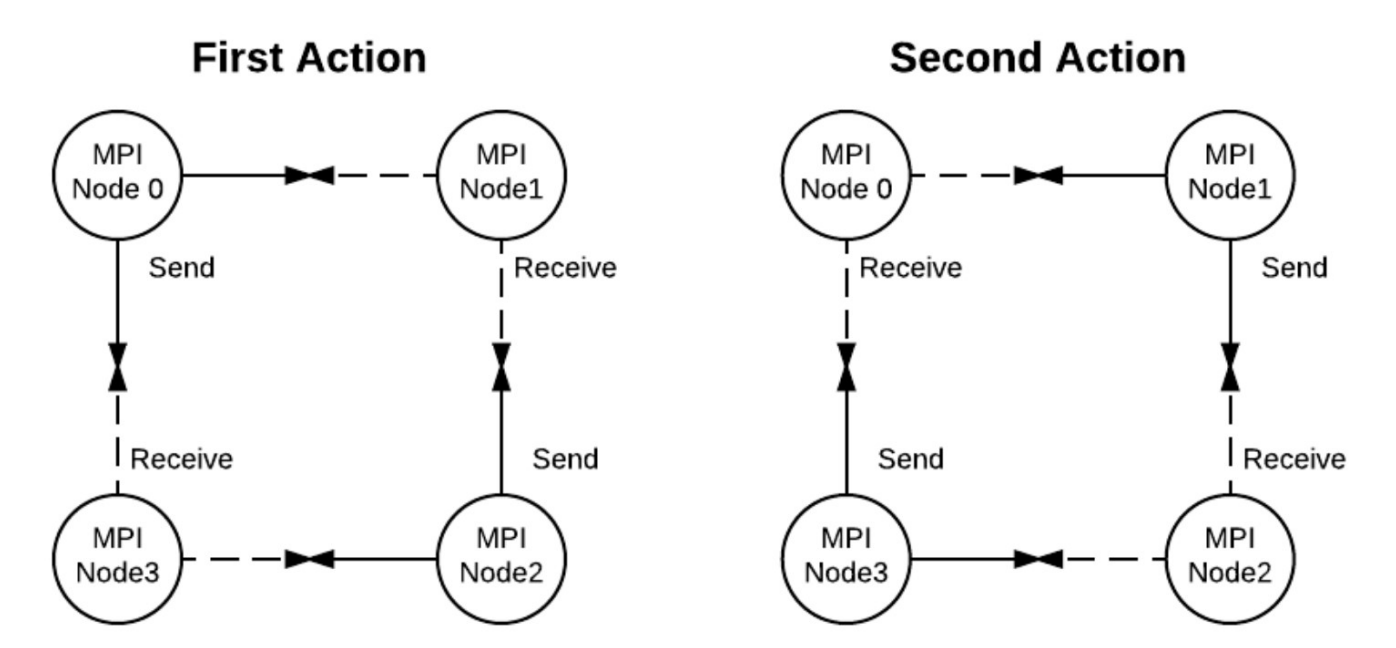
int y;

MPI\_Recv(&y, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE); // receive fr

## How Send and Recv Work Together

* **Matching is crucial**: Every MPI\_Send must have a **corresponding MPI\_Recv**.
* If both processes call MPI\_Recv first → deadlock occurs.

**Q3. Implement blocking MPI send & receive to demonstrate Nearest neighbor exchange of data in a ring topology.**

****

**Program**

/\*

Implement blocking MPI send & receive to demonstrate Nearest neighbor exchange of data in a ring topology.

\*/

/\*

mpic++ Q3.cpp -o Q3

mpiexec -n 4 ./Q3

\*/

#include <mpi.h>

#include <iostream>

#include <cstdio>

using namespace std;

int main(int argc, char\*\* argv) {

MPI\_Init(&argc, &argv);

int world\_rank, world\_size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

int token;

if (world\_rank == 0) {

token = -1;

}

int next = (world\_rank + 1) % world\_size;

int prev = (world\_rank - 1 + world\_size) % world\_size;

if (world\_rank != 0) {

MPI\_Recv(&token, 1, MPI\_INT, prev, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("Process %d received token %d from process %d\n", world\_rank, token, prev);

}

MPI\_Send(&token, 1, MPI\_INT, next, 0, MPI\_COMM\_WORLD);

if (world\_rank == 0) {

MPI\_Recv(&token, 1, MPI\_INT, world\_size - 1, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("Process %d received token %d from process %d\n", world\_rank, token, world\_size - 1);

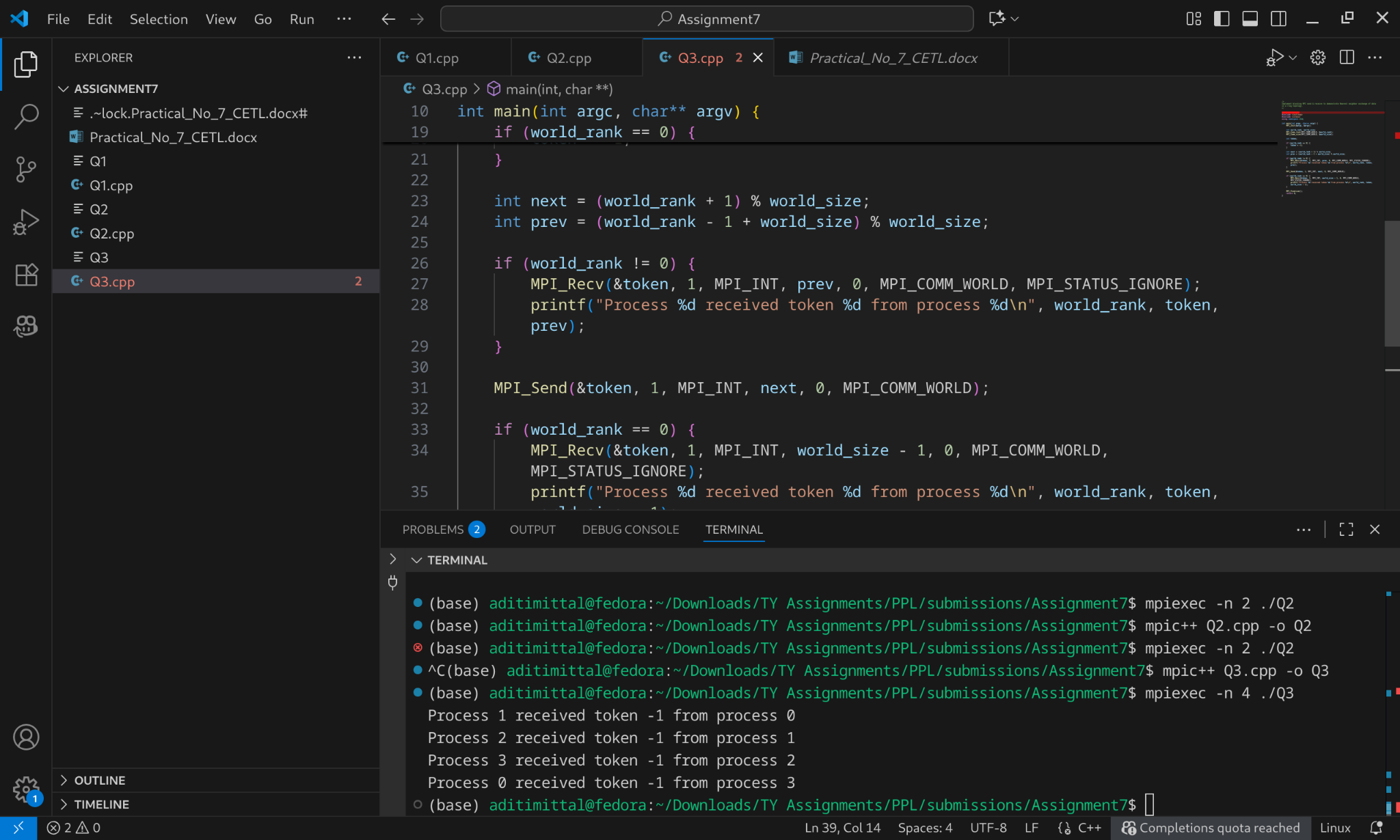
}

MPI\_Finalize();

return 0;

}

**Screenshots**

****

The ring program initializes a value from process zero, and the value is passed around every single process. The program terminates when process zero receives the value from the last process. As you can see from the program, extra care is taken to assure that it doesn’t deadlock. In other words, process zero makes sure that it has completed its first send before it tries to receive the value from the last process. All of the other processes simply call MPI\_Recv (receiving from their neighboring lower process) and then MPI\_Send (sending the value to their neighboring higher process) to pass the value along the ring. MPI\_Send and MPI\_Recv will block until the message has been transmitted.

**References:** <http://mpitutorial.com/tutorials/mpi-send-and-receive/>   
**Observations**

* The **token successfully circulated the ring**.
* The **send/receive ordering avoided deadlock** even with blocking MPI.

**Q4. Write a MPI program to find the sum of all the elements of an array A of size**

**n. Elements of an array can be divided into two equals groups. The first [n/2]**

**elements are added by the first process, P0, and last [n/2] elements the by second process, P1. The two sums then are added to get the final result.**

**Program:**

/\*

Write a MPI program to find the sum of all the elements of an array A of size

n. Elements of an array can be divided into two equals groups. The first [n/2]

elements are added by the first process, P0, and last [n/2] elements the by second process, P1. The two sums then are added to get the final result.

\*/

/\*

mpic++ Q4.cpp -o Q4

mpiexec -n 2 ./Q4

\*/

#include <mpi.h>

#include <iostream>

using namespace std;

int main(int argc, char\*\* argv) {

MPI\_Init(&argc, &argv);

int rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

int n = 10;

int A[n] = {1,2,3,4,5,6,7,8,9,10};

int local\_sum = 0;

int half = n / 2;

if (rank == 0){

for (int i = 0; i < half; i++){

local\_sum += A[i];

}

MPI\_Send(&local\_sum, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

}

else if (rank == 1){

for (int i = half; i < n; i++){

local\_sum += A[i];

}

int sum0;

MPI\_Recv(&sum0, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int total\_sum = local\_sum + sum0;

cout << "Total sum of array elements = " << total\_sum << endl;

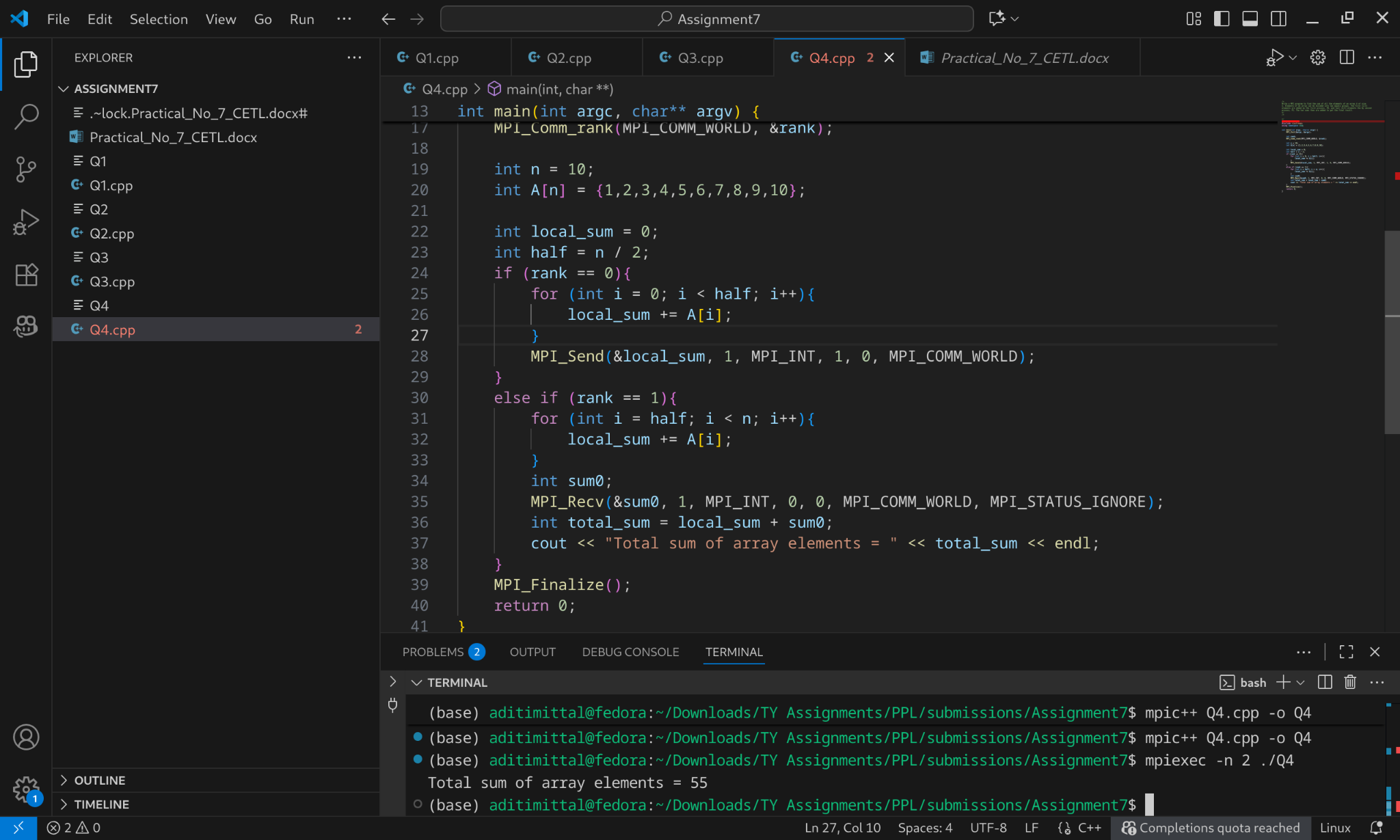
}

MPI\_Finalize();

return 0;

}

**Screenshots:**

****

### How It Works

1. Array is split into two halves.
2. Process 0 computes the sum of the first half.
3. Process 1 computes the sum of the second half.
4. Process 0 sends its partial sum to process 1.
5. Process 1 adds both sums to get the final result.