Name: Tazmeen Afroz

Roll No: 22P-9252

Section: BAI-6A

Instructor: Sir Ali Sayyed

Course: Parallel and Distributed Computing

Assignment: 2

Campus: FAST-NUCES Peshawar

Date: 7 May 2025

Task 1:

Problem Statement

Write an MPI program that assigns different roles to each process based on its rank:

- Process 0: Master (coordinator) - reads an array of 16 integers and distributes segments to all processes.
- All Other Processes: Workers - receive their portion, compute the square of each value, and send results back.

1. Implement this using MPI_Send and MPI_Recv. 2. Master process should collect results and display the final array. 3. Add support for arbitrary number of processes ≤ 16.

Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc , char **argv){
    //rank - process id
    //process_count - number of processes
    //MPI_Init - initializes the MPI environment
    //MPI_Comm_rank - gets the rank of the process
    //MPI_Comm_size - gets the number of processes
   //MPI_COMM_WORLD - communicator that includes all processes
    int rank, process_count;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &process_count);

    int array[16];
    // Calculate the number of elements each worker process will handle
    //equally distribute the elements among the worker processes and if there are remaining elements, distribute them one by one to the first few processes
    int elements_per_process = 16 / (process_count - 1);
    int remaining_elements = 16 % (process_count - 1);

    if(rank == 0){

        for(int i = 0; i < 16; i++){
            array[i] = i + 1;
        }


        printf("The original array is: ");
        for(int i = 0; i < 16; i++){
            printf("%d ", array[i]);
        }
        printf("\n");
        // Distribute the array to all worker processes
        int start_index = 0;
        for(int i = 1; i < process_count; i++){
```

```
            int count = elements_per_process;
            if(i <= remaining_elements) {
                count += 1; // Add 1 if there are remaining elements becaue one process will have one more element
            }
            MPI_Send(&array[start_index], count, MPI_INT, i, 0, MPI_COMM_WORLD);
            start_index += count;
        }

        // Collect results from all worker processes
        start_index = 0;
        for(int i = 1; i < process_count; i++){
            int count = elements_per_process;
            if(i <= remaining_elements) {
                count += 1; // Add 1 if there are remaining elements
            }
            MPI_Recv(&array[start_index], count, MPI_INT, i, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            start_index += count;
        }
                // Print the results
        printf("The squares are: ");
        for(int i = 0; i < 16; i++){
            printf("%d ", array[i]);
        }
        printf("\n");
    } else {
        // Worker processes receive their portion of the array
        int count = elements_per_process;
        if(rank <= remaining_elements) {
            count += 1;
        }
        int sub_array[count];
        MPI_Recv(sub_array, count, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // Compute the square of each value
        for(int i = 0; i < count; i++){
            sub_array[i] = sub_array[i] * sub_array[i];
        }

        // Send the results back to the root process
        MPI_Send(sub_array, count, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

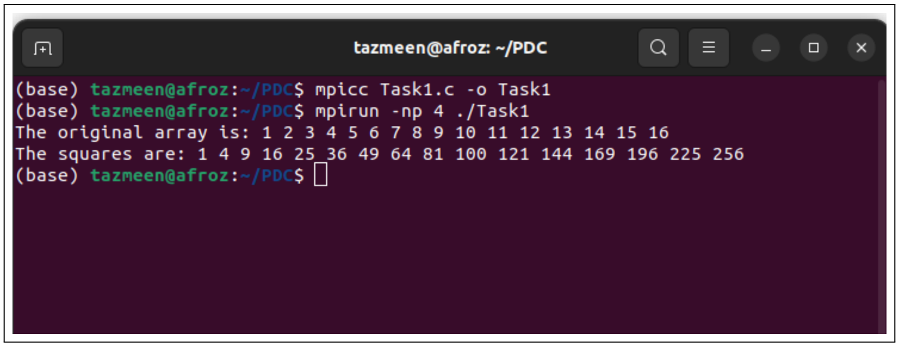


Figure 1: Output of Task 1 implementation

Questions

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

The workload is evenly distributed among worker processes, with extra elements assigned to the first few processes if the array size is not evenly divisible. As the number of processes increases, each worker receives fewer elements to process, potentially improving performance until communication overhead begins to dominate.

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

The design can handle larger arrays but may face communication bottlenecks due to the root process. The current design uses a single root process to coordinate all communication. This limits scalability because the root process cannot distribute or collect data in parallel, becoming a bottleneck as the array size or number of processes increases.

Task 2: Safe Non-Blocking Communication

Problem Statement

Modify Task 1 to use non-blocking versions: MPI_Isend, MPI_Irecv, and MPI_Waitall.

1. Create an array `requests[]` to manage multiple asynchronous communications.
2. Ensure correct synchronization using `MPI_Waitall`.

Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc , char **argv){
    int rank, process_count;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &process_count);

    int array[16];
    int elements_per_process = 16 / (process_count - 1);
    int remaining_elements = 16 % (process_count - 1);

    if(rank == 0){
        for(int i = 0; i < 16; i++){
            array[i] = i + 1;
        }

        printf("The original array is: ");
        for(int i = 0; i < 16; i++){
            printf("%d ", array[i]);
        }
        printf("\n");

        // Arrays to manage asynchronous communication
        MPI_Request send_requests[process_count - 1];
        MPI_Request recv_requests[process_count - 1];

        // Distribute the array to all worker processes
        int start_index = 0;
        for(int i = 1; i < process_count; i++){
            int count = elements_per_process;
            if(i <= remaining_elements) {
                count += 1;
            }
            MPI_Isend(&array[start_index], count, MPI_INT, i, 0, MPI_COMM_WORLD, &send_requests[i - 1]);
            start_index += count;
        }

        // Collect results from all worker processes
```

```
        start_index = 0;
        for(int i = 1; i < process_count; i++){
            int count = elements_per_process;
            if(i <= remaining_elements) {
                count += 1;
            }
            MPI_Irecv(&array[start_index], count, MPI_INT, i, 1, MPI_COMM_WORLD, &recv_requests[i - 1]);
            start_index += count;
        }

        // Wait for all sends and receives to complete
        MPI_Waitall(process_count - 1, send_requests, MPI_STATUSES_IGNORE);
        MPI_Waitall(process_count - 1, recv_requests, MPI_STATUSES_IGNORE);

        // Print the results
        printf("The squares are: ");
        for(int i = 0; i < 16; i++){
            printf("%d ", array[i]);
        }
        printf("\n");
    } else {
        // Worker processes receive their portion of the array
        int count = elements_per_process;
        if(rank <= remaining_elements) {
            count += 1;
        }
        int sub_array[count];
        MPI_Request recv_request, send_request;

        MPI_Irecv(sub_array, count, MPI_INT, 0, 0, MPI_COMM_WORLD, &recv_request);
        MPI_Wait(&recv_request, MPI_STATUS_IGNORE);

        // Compute the square of each value
        for(int i = 0; i < count; i++){
            sub_array[i] = sub_array[i] * sub_array[i];
        }

        // Send the results back to the root process
        MPI_Isend(sub_array, count, MPI_INT, 0, 1, MPI_COMM_WORLD, &send_request);
        MPI_Wait(&send_request, MPI_STATUS_IGNORE);
    }

    MPI_Finalize();
    return 0;
}
```

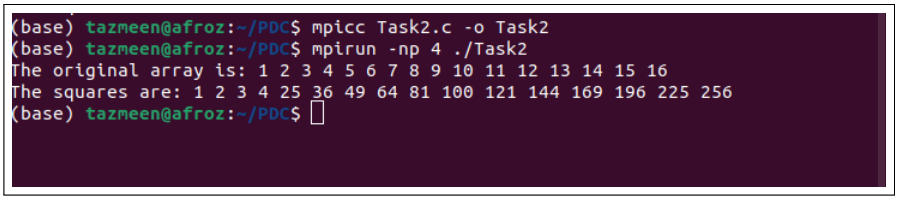


Figure 2: Output of Task 2 implementation

Questions

a. Explain why MPI_Waitall is needed.

Non-blocking operations like MPI_Isend and MPI_Irecv initiate communication but do not wait for it to complete. This allows the program to continue executing other tasks while the communication happens in the background. MPI_Waitall ensures that all initiated non-blocking operations in the MPI_Request array are completed before moving forward. Without it, the program might proceed prematurely, leading to undefined behavior or incomplete communication.

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

Omitting the wait for non-blocking messages can lead to accessing incomplete data, modifying send buffers too early, and memory corruption from premature buffer reuse. It may also cause the program to finish before communication completes, resulting in data loss. In simulation, skipping MPI_Waitall led to inconsistent outputs, incorrect values, and occasional crashes across multiple runs.

Task 3: Custom Communication Protocol

Problem Statement

Write an MPI program with at least 4 processes, using the following logic:

- Process 0 sends two arrays to Process 1 and 2.
- Process 1 and 2 process the arrays and send results to Process 3.
- Process 3 performs final aggregation and displays the result.

1. Use different tags for each message. 2. Use MPI_Status in receiving functions to determine source and tag dynamically.

Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc , char **argv){
    int rank, process_count;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &process_count);

    // Write an MPI program with at least 4 processes, using the following logic: - Process 0 sends two arrays to Process 1 and 2.
    // - Process 1 and 2 process the arrays and send results to Process 3. - Process 3 performs final aggregation and displays the result.
    // 1. Use different tags for each message.
    // 2. Use MPI_Status in receiving functions to determine source and tag dynamically.


    int array1[4], array2[4];
    int result1[4], result2[4];
    int final_result[4];
    MPI_Status status;
    if(rank == 0){
        // Initialize the arrays
        for(int i = 0; i < 4; i++){
            array1[i] = i + 1;
            array2[i] =  i + 2;
        }
        // Send the arrays to Process 1 and Process 2
        MPI_Send(array1, 4, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Send(array2, 4, MPI_INT, 2, 1, MPI_COMM_WORLD);
    } else if(rank == 1){
        // Receive the first array from Process 0
        MPI_Recv(array1, 4, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        // Process the array (summing each element)
        for(int i = 0; i < 4; i++){
            result1[i] = array1[i]  + array1[i];
        }
```

```
        // Send the result to Process 3
        MPI_Send(result1, 4, MPI_INT, 3, 2, MPI_COMM_WORLD);
    } else if(rank == 2){
        // Receive the second array from Process 0
        MPI_Recv(array2, 4, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
        // Process the array (summing each element)
        for(int i = 0; i < 4; i++){
            result2[i] = array2[i] + array2[i];
        }
        // Send the result to Process 3
        MPI_Send(result2, 4, MPI_INT, 3, 3, MPI_COMM_WORLD);
    } else if(rank == 3){
        // Receive results from Process 1 and Process 2
        MPI_Recv(result1, 4, MPI_INT, 1, 2, MPI_COMM_WORLD, &status);
        MPI_Recv(result2, 4, MPI_INT, 2, 3, MPI_COMM_WORLD, &status);

        // Aggregate results
        for(int i = 0; i < 4; i++){
            final_result[i] = result1[i] + result2[i];
        }

        // Display the final result
        printf("Final aggregated result: ");
        for(int i  = 0; i < 4; i++){
            printf("%d ", final_result[i]);
        }

        printf("\n");

    }
    MPI_Finalize();
    return 0;
}
```

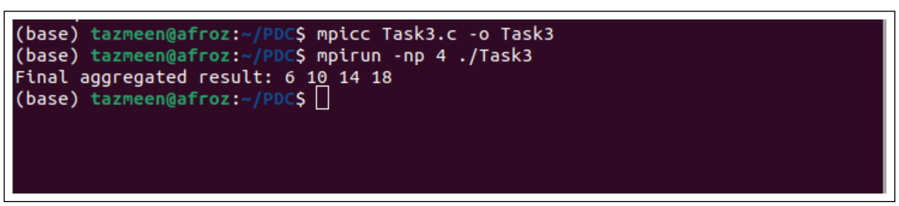


Figure 3: Output of Task 3 implementation

Questions

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

When multiple messages are sent to the same process, tags ensure that the receiving process can handle each message appropriately. For example, in my code:

- Process 3 uses tags 2 and 3 to distinguish between results from Process 1 and Process 2

- Without tags, Process 3 would not know which incoming message corresponds to which computation
- Tags allow the receiver to selectively receive messages in a specific order, regardless of when they arrive
- This enables more complex communication patterns and workflows

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

If two messages with the same tag arrive from different sources, the receiving process cannot distinguish between them based on the tag alone. This may lead to processing the wrong message or overwriting data. If the process doesn't check the source using MPI_SOURCE in the MPI_Status, it risks handling a message from an unintended source

For example: If Process 3 in my code receives two messages with tag 2 (one from Process 1 and one from Process 2), it cannot determine which message corresponds to which source without checking MPI_SOURCE in the status object.

Task 4: Implement Circular Ping-Pong

Problem Statement

Create a ring of N processes (N $\geq$ 4), where each process passes a counter to the next process in the ring. The counter starts at 0 and is incremented on each pass.

1. Process 0 starts the counter.

2. After M complete cycles, the process 0 terminates the loop and ends execution on all processes.

Requirements:

- Implement safe termination using message flags.

- Avoid deadlocks.

Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>



int main(int argc, char **argv) {
    int world_rank, world_size;


    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Check if we have enough processes
    if (world_size < 4) {
        if (world_rank == 0) {
            printf("Error: This program requires at least 4 processes.\n")
    ;
        }
        MPI_Finalize();
        return 1;
    }


    int counter = 0;
    int M = 2; // Number of complete cycles
    int cycles_completed = 0;
    MPI_Status status;

    // Define next and previous ranks
    // rank 1 then next rank = (0 + 1) % 4 = 1
    // rank 1 then prev rank = (0 + 4 - 1) % 4 = 3
    int next_rank = (world_rank + 1) % world_size;
    int prev_rank = (world_rank + world_size - 1) % world_size;
```

```c


    if (world_rank == 0) {
        printf("Process %d: Starting counter with value %d\n", world_rank, counter);

        // Send initial counter
        counter++; // Increment
        printf("Process %d: Sending counter with value %d to process %d\n",
               world_rank, counter, next_rank);
        MPI_Send(&counter, 1, MPI_INT, next_rank, 0, MPI_COMM_WORLD);

        // Main loop for process 0
        while (cycles_completed < M) {
            // Receive the counter after a complete cycle
            MPI_Recv(&counter, 1, MPI_INT, prev_rank, 0, MPI_COMM_WORLD, &status);
            printf("Process %d: Received counter with value %d from process %d\n",
                   world_rank, counter, prev_rank);

            // Increment cycles counter
            cycles_completed++;
            printf("Process %d: Completed cycle %d of %d\n", world_rank, cycles_completed, M);

            if (cycles_completed < M) {
                // Continue with another cycle
                counter++; // Increment counter
                printf("Process %d: Sending counter with value %d to process %d\n",
                       world_rank, counter, next_rank);
                MPI_Send(&counter, 1, MPI_INT, next_rank, 0, MPI_COMM_WORLD);
            }
        }

        // Send termination message to next process after all cycles are complete
        printf("Process %d: All %d cycles completed, initiating termination\n", world_rank, M);
        MPI_Send(&counter, 1, MPI_INT, next_rank, 1, MPI_COMM_WORLD);

    } else {
        // Other processes just pass along the counter until terminated
        while (1) {
            // Receive message from previous process
            MPI_Recv(&counter, 1, MPI_INT, prev_rank, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

            // Check the tag to see if it's a termination message
            if (status.MPI_TAG == 1) {
                printf("Process %d: Received termination signal,
```

```
forwarding to process %d\n",
                        world_rank, next_rank);

                // Forward the termination message and exit the loop
                MPI_Send(&counter, 1, MPI_INT, next_rank, 1, MPI_COMM_WORLD);
                break;
            }

            // Regular counter message - process and forward
            printf("Process %d: Received counter with value %d from process %d\n",
                    world_rank, counter, prev_rank);

            counter++; // Increment counter
            printf("Process %d: Sending counter with value %d to process %d\n",
                    world_rank, counter, next_rank);
            MPI_Send(&counter, 1, MPI_INT, next_rank, 0, MPI_COMM_WORLD);
        }
    }

    printf("Process %d: Terminating\n", world_rank);
    MPI_Finalize();
    return 0;
}
```

Questions

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

The most common issues in ring-based communication are:

- Deadlocks: When all processes are waiting to receive before sending, creating a circular dependency
- Message routing errors: When processes send in one direction but receive from another, causing messages to never arrive
- Termination detection: Difficulty in determining when to stop the ring communication
- Error propagation: Errors in one process can cascade through the entire ring

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

The differences between unidirectional and bi-directional ring communication include:

Message Flow

- Unidirectional: Message passes in one direction only
- Bi-directional: Messages flow in both directions (clockwise and counterclockwise)

Counters

- Unidirectional: One counter circulates the ring

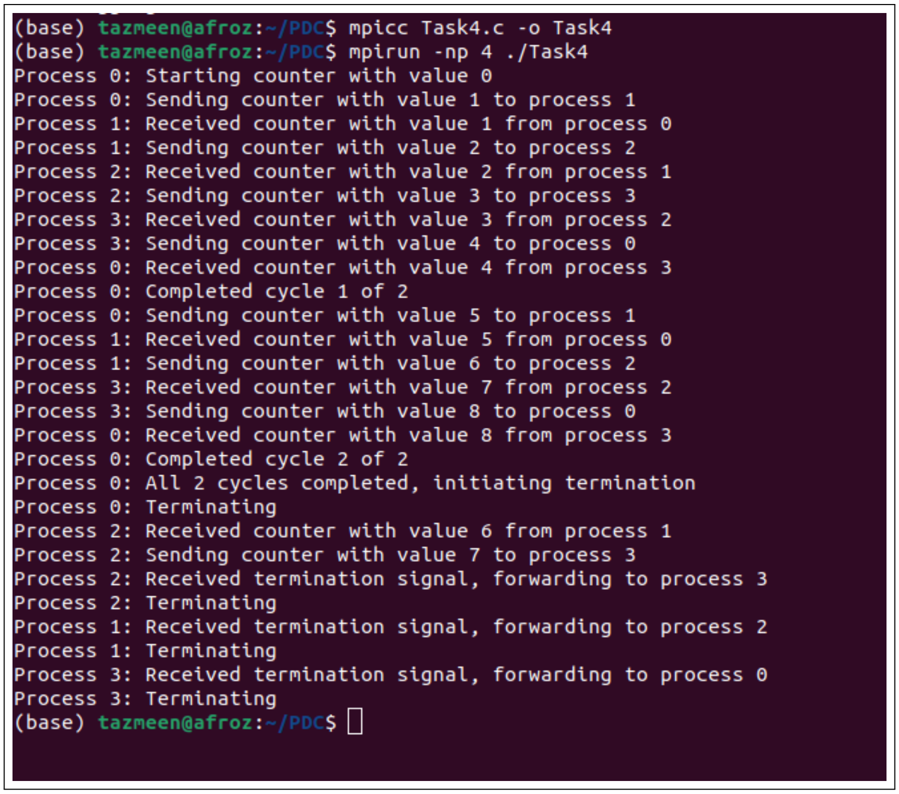


Figure 4: Output of Task 4 implementation

- Bi-directional: Two counters move in opposite directions

Termination Logic

- Unidirectional: Process 0 stops after one counter completes M cycles
- Bi-directional: Process 0 tracks both counters; terminates after both complete M cycles

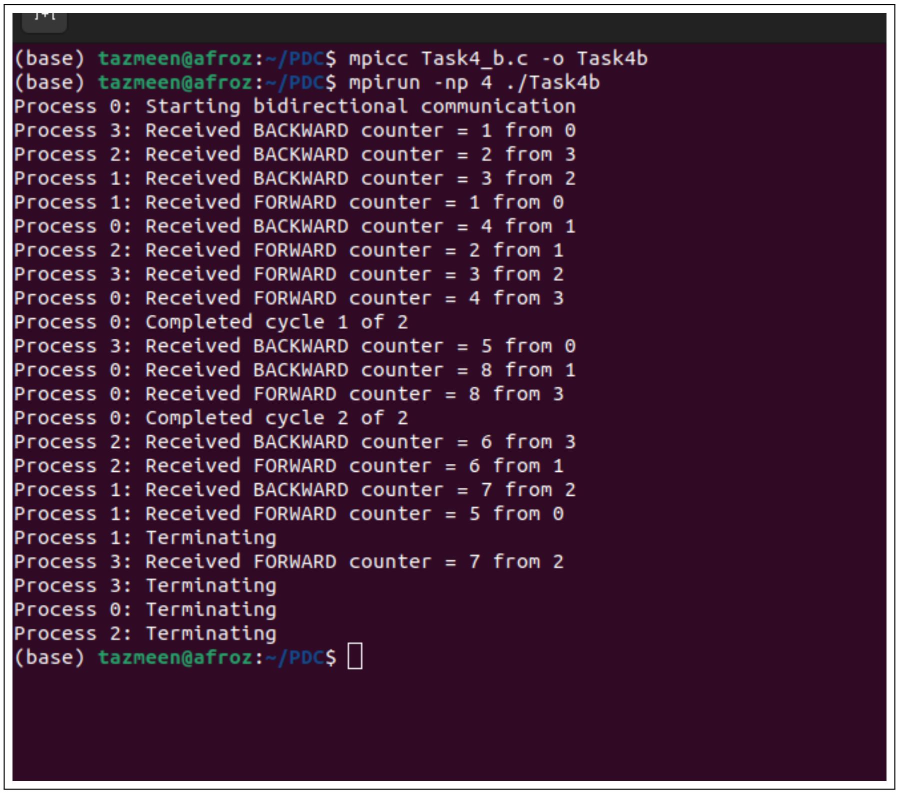
Performance

- Unidirectional: Slower due to single path usage
- Bi-directional: Faster with parallel message flow, but higher overhead

```c
// #include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char **argv) {
    int world_rank, world_size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    if (world_size < 4) {
        if (world_rank == 0) {
            printf("Error: This program requires at least 4 processes.\n");
        }
```

```c
        MPI_Finalize();
        return 1;
    }

    int counter_forward = 0;
    int counter_backward = 0;
    int M = 2;
    int cycles_completed = 0;
    MPI_Status status;

    int next_rank = (world_rank + 1) % world_size;
    int prev_rank = (world_rank + world_size - 1) % world_size;

    if (world_rank == 0) {
        printf("Process %d: Starting bidirectional communication\n", world_rank);

        // Start both directions
        counter_forward++;
        counter_backward++;
        MPI_Send(&counter_forward, 1, MPI_INT, next_rank, 0, MPI_COMM_WORLD);
        MPI_Send(&counter_backward, 1, MPI_INT, prev_rank, 0, MPI_COMM_WORLD);

        while (cycles_completed < M) {
            // Receive from both sides
            MPI_Recv(&counter_backward, 1, MPI_INT, next_rank, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            printf("Process %d: Received BACKWARD counter = %d from %d\n", world_rank, counter_backward, next_rank);

            MPI_Recv(&counter_forward, 1, MPI_INT, prev_rank, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            printf("Process %d: Received FORWARD counter = %d from %d\n", world_rank, counter_forward, prev_rank);

            cycles_completed++;
            printf("Process %d: Completed cycle %d of %d\n", world_rank, cycles_completed, M);

            if (cycles_completed < M) {
                counter_forward++;
                counter_backward++;
                MPI_Send(&counter_forward, 1, MPI_INT, next_rank, 0, MPI_COMM_WORLD);
                MPI_Send(&counter_backward, 1, MPI_INT, prev_rank, 0, MPI_COMM_WORLD);
            }
        }

        // Termination signal in both directions
        MPI_Send(&counter_forward, 1, MPI_INT, next_rank, 1, MPI_COMM_WORLD);
```

```c
        MPI_Send(&counter_backward, 1, MPI_INT, prev_rank, 1, MPI_COMM_WORLD);

    } else {
        while (1) {
            // BACKWARD direction
            MPI_Recv(&counter_backward, 1, MPI_INT, next_rank, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            if (status.MPI_TAG == 1) {
                MPI_Send(&counter_backward, 1, MPI_INT, prev_rank, 1, MPI_COMM_WORLD);
                break;
            }
            printf("Process %d: Received BACKWARD counter = %d from %d\n", world_rank, counter_backward, next_rank);
            counter_backward++;
            MPI_Send(&counter_backward, 1, MPI_INT, prev_rank, 0, MPI_COMM_WORLD);

            // FORWARD direction
            MPI_Recv(&counter_forward, 1, MPI_INT, prev_rank, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            if (status.MPI_TAG == 1) {
                MPI_Send(&counter_forward, 1, MPI_INT, next_rank, 1, MPI_COMM_WORLD);
                break;
            }
            printf("Process %d: Received FORWARD counter = %d from %d\n", world_rank, counter_forward, prev_rank);
            counter_forward++;
            MPI_Send(&counter_forward, 1, MPI_INT, next_rank, 0, MPI_COMM_WORLD);
        }
    }

    printf("Process %d: Terminating\n", world_rank);
    MPI_Finalize();
    return 0;
}
```



```
(base) tazmeen@afroz:~/PDC$ mpicc Task4_b.c -o Task4b
(base) tazmeen@afroz:~/PDC$ mpirun -np 4 ./Task4b
Process 0: Starting bidirectional communication
Process 3: Received BACKWARD counter = 1 from 0
Process 2: Received BACKWARD counter = 2 from 3
Process 1: Received BACKWARD counter = 3 from 2
Process 1: Received FORWARD counter = 1 from 0
Process 0: Received BACKWARD counter = 4 from 1
Process 2: Received FORWARD counter = 2 from 1
Process 3: Received FORWARD counter = 3 from 2
Process 0: Received FORWARD counter = 4 from 3
Process 0: Completed cycle 1 of 2
Process 3: Received BACKWARD counter = 5 from 0
Process 0: Received BACKWARD counter = 8 from 1
Process 0: Received FORWARD counter = 8 from 3
Process 0: Completed cycle 2 of 2
Process 2: Received BACKWARD counter = 6 from 3
Process 2: Received FORWARD counter = 6 from 1
Process 1: Received BACKWARD counter = 7 from 2
Process 1: Received FORWARD counter = 5 from 0
Process 1: Terminating
Process 3: Received FORWARD counter = 7 from 2
Process 3: Terminating
Process 0: Terminating
Process 2: Terminating
(base) tazmeen@afroz:~/PDC$
```

Figure 5: Output of bi-directional implementation

Task 5: Performance Timing and Barriers

Problem Statement

1. Use MPI_Wtime to time the execution of Tasks 1 and 2.

2. Introduce MPI_Barrier to synchronize processes before timing starts and ends.

Report:

- Time taken for blocking vs non-blocking communication.
- Explain the overhead introduced by synchronization.

Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char **argv) {
    // rank - process id
    // process_count - number of processes
    int rank, process_count;
    double start_time, end_time;

    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &process_count);

    int array[16];
    // Calculate the number of elements each worker process will handle
    int elements_per_process = 16 / (process_count - 1);
    int remaining_elements = 16 % (process_count - 1);

    // Synchronize all processes before starting the timer
    MPI_Barrier(MPI_COMM_WORLD);

    // Start the timer
    start_time = MPI_Wtime();

    if(rank == 0) {
        for(int i = 0; i < 16; i++) {
            array[i] = i + 1;
        }
        printf("The original array is: ");
        for(int i = 0; i < 16; i++) {
            printf("%d ", array[i]);
        }
        printf("\n");

        // Distribute the array to all worker processes
        int start_index = 0;
        for(int i = 1; i < process_count; i++) {
            int count = elements_per_process;
            if(i <= remaining_elements) {
                count += 1; // Add 1 if there are remaining elements
            }
            MPI_Send(&array[start_index], count, MPI_INT, i, 0,
   MPI_COMM_WORLD);
            start_index += count;
        }

        // Collect results from all worker processes
        start_index = 0;
        for(int i = 1; i < process_count; i++) {
            int count = elements_per_process;
            if(i <= remaining_elements) {
                count += 1; // Add 1 if there are remaining elements
            }
            MPI_Recv(&array[start_index], count, MPI_INT, i, 1,
   MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            start_index += count;
        }

        // Print the results
        printf("The squares are: ");
        for(int i = 0; i < 16; i++) {
            printf("%d ", array[i]);
        }
        printf("\n");
```

```
    } else {
        // Worker processes receive their portion of the array
        int count = elements_per_process;
        if(rank <= remaining_elements) {
            count += 1;
        }
        int sub_array[count];
        MPI_Recv(sub_array, count, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // Compute the square of each value
        for(int i = 0; i < count; i++) {
            sub_array[i] = sub_array[i] * sub_array[i];
        }

        // Send the results back to the root process
        MPI_Send(sub_array, count, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }

    // Synchronize all processes before stopping the timer
    MPI_Barrier(MPI_COMM_WORLD);

    // Stop the timer
    end_time = MPI_Wtime();

    // Print the execution time
    if (rank == 0) {
        printf("Task 1 (Blocking Communication) execution time: %f seconds\n", end_time - start_time);
    }

    MPI_Finalize();
    return 0;
}
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char **argv) {
    int rank, process_count;
    double start_time, end_time;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &process_count);

    int array[16];
    int elements_per_process = 16 / (process_count - 1);
    int remaining_elements = 16 % (process_count - 1);

    // Synchronize all processes before starting the timer
    MPI_Barrier(MPI_COMM_WORLD);

    // Start the timer
```

```c
    start_time = MPI_Wtime();

    if(rank == 0) {
        for(int i = 0; i < 16; i++) {
            array[i] = i + 1;
        }
        printf("The original array is: ");
        for(int i = 0; i < 16; i++) {
            printf("%d ", array[i]);
        }
        printf("\n");

        // Arrays to manage asynchronous communication
        MPI_Request send_requests[process_count - 1];
        MPI_Request recv_requests[process_count - 1];

        // Distribute the array to all worker processes
        int start_index = 0;
        for(int i = 1; i < process_count; i++) {
            int count = elements_per_process;
            if(i <= remaining_elements) {
                count += 1;
            }
            MPI_Isend(&array[start_index], count, MPI_INT, i, 0, MPI_COMM_WORLD, &send_requests[i - 1]);
            start_index += count;
        }

        // Collect results from all worker processes
        start_index = 0;
        for(int i = 1; i < process_count; i++) {
            int count = elements_per_process;
            if(i <= remaining_elements) {
                count += 1;
            }
            MPI_Irecv(&array[start_index], count, MPI_INT, i, 1, MPI_COMM_WORLD, &recv_requests[i - 1]);
            start_index += count;
        }

        // Wait for all sends and receives to complete
        MPI_Waitall(process_count - 1, send_requests, MPI_STATUSES_IGNORE);
        MPI_Waitall(process_count - 1, recv_requests, MPI_STATUSES_IGNORE);

        // Print the results
        printf("The squares are: ");
        for(int i = 0; i < 16; i++) {
            printf("%d ", array[i]);
        }
        printf("\n");
    } else {
        // Worker processes receive their portion of the array
```

```c
        int count = elements_per_process;
        if(rank <= remaining_elements) {
            count += 1;
        }
        int sub_array[count];
        MPI_Request recv_request, send_request;

        MPI_Irecv(sub_array, count, MPI_INT, 0, 0, MPI_COMM_WORLD, &recv_request);
        MPI_Wait(&recv_request, MPI_STATUS_IGNORE);

        // Compute the square of each value
        for(int i = 0; i < count; i++) {
            sub_array[i] = sub_array[i] * sub_array[i];
        }

        // Send the results back to the root process
        MPI_Isend(sub_array, count, MPI_INT, 0, 1, MPI_COMM_WORLD, &send_request);
        MPI_Wait(&send_request, MPI_STATUS_IGNORE);
    }

    // Synchronize all processes before stopping the timer
    MPI_Barrier(MPI_COMM_WORLD);

    // Stop the timer
    end_time = MPI_Wtime();

    // Print the execution time
    if (rank == 0) {
        printf("Task 2 (Non-blocking Communication) execution time: %f seconds\n", end_time - start_time);
    }

    MPI_Finalize();
    return 0;
}
```

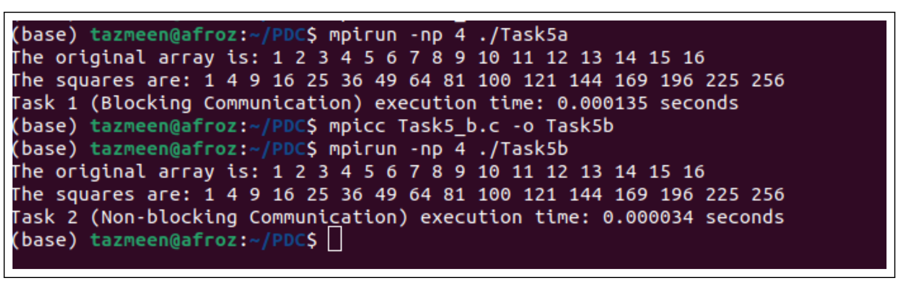


Figure 6: Output of Task 5 implementation showing timing results

Results and Analysis

Timing Results

- Blocking Communication: 0.000135 seconds
- Non-blocking Communication: 0.000034 seconds

Analysis

Non-blocking communication (0.000034s) is faster than blocking (0.000135s) because it enables parallel data transfer without waiting. MPI Barriers add synchronization overhead like network delays and load imbalance by forcing all processes to align before timing starts or ends, ensuring consistent results