Assignment no 2

Name: **Muhammad Abdullah**
Roll no: **22P-9371**
Section: **BCS-6B**
Submitted to**: Dr. Ali Sayyed**

Task 1: Process Role Identification and Dynamic Tasking

Solution:

Source Code:

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

#define MAX_ARRAY_SIZE 16   // Total number of elements in the array
#define MAX_SEGMENT_SIZE 16 // Maximum size of the segment each worker can handle

int main(int argc, char *argv[])
{
   int rank, size;
   // Initialize the MPI environment
   MPI_Init(&argc, &argv);
   // Get the rank (ID) of the current process
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   // Get the total number of processes
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   // Ensure no more than 16 processes are used
   if (size > 16)
   {
       if (rank == 0)
           printf("Error: Maximum 16 processes allowed.\n");
       MPI_Finalize();
       return 1;
   }
   int array_size = MAX_ARRAY_SIZE;              // Total elements in the main array
   int array[MAX_ARRAY_SIZE];                    // Original array (used only by master)
   int result[MAX_ARRAY_SIZE];                   // Final result array to store squares
   int segment[MAX_SEGMENT_SIZE];                // Buffer for each worker's segment
   int segment_size = array_size / (size - 1); // Base segment size for each worker
```

```
    int remainder = array_size % (size - 1);    // Extra elements to distribute evenly

    if (rank == 0)
    { // Master process
        // Initialize the array with values 1 to 16
        for (int i = 0; i < array_size; i++)
        {
            array[i] = i + 1;
        }
        // Distribute segments of the array to worker processes
        int offset = 0;
        for (int i = 1; i < size; i++)
        {
            int send_size = segment_size;

            // Distribute remaining elements evenly to the first few workers
            if (i <= remainder)
            {
                send_size += 1;
            }
            // Send the size of the segment
            MPI_Send(&send_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            // Send the actual segment of the array
            MPI_Send(&array[offset], send_size, MPI_INT, i, 0, MPI_COMM_WORLD);

            offset += send_size;
        }
        // Collect results from workers
        offset = 0;
        for (int i = 1; i < size; i++)
        {
            int recv_size = segment_size;
            if (i <= remainder)
            {
                recv_size += 1;
            }
            // Receive the squared segment from each worker
            MPI_Recv(&result[offset], recv_size, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

            offset += recv_size;
        }
        // Print the final squared array
        printf("Final squared array: ");
        for (int i = 0; i < array_size; i++)
        {
            printf("%d ", result[i]);
        }
        printf("\n");
```

```
    }
    else
    { // Worker processes
        int recv_size;
        // Receive the size of the segment from the master
        MPI_Recv(&recv_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // Receive the actual segment from the master
        MPI_Recv(segment, recv_size, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // Compute the square of each element in the segment
        for (int i = 0; i < recv_size; i++)
        {
            segment[i] = segment[i] * segment[i];
        }
        // Send the squared segment back to the master
        MPI_Send(segment, recv_size, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    // Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

Output:

```
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_Assignment#2$ mpirun -np 4 --hostfile myhostfile ./task1
Final squared array: 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256
```

a. How is workload distribution affected by the number of processes?
Ans: The workload is divided among size - 1 worker processes, with each handling approximately array_size / (size - 1) elements. If there's a remainder, some workers process one extra element. More processes reduce per-process workload but increase communication overhead.

b. Can this design scale for larger arrays? Why or why not?
Ans: It can scale for larger arrays by increasing array_size and MAX_ARRAY_SIZE, but scalability is limited by the overhead of point-to-point communication (MPI_Send/MPI_Recv) and the 16-process limit. Collective operations like MPI_Scatter/MPI_Gather would be more efficient for large arrays.

Task 2: Safe Non-Blocking Communication

Solution:

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

```
#define MAX_ARRAY_SIZE 16   // Total number of elements in the array
#define MAX_SEGMENT_SIZE 16 // Maximum size a worker can receive

int main(int argc, char *argv[])
{
   int rank, size;
   // Initialize MPI environment
   MPI_Init(&argc, &argv);
   // Get the rank (ID) of this process
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   // Get the total number of processes
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   // Check if number of processes exceeds the limit
   if (size > 16)
   {
       if (rank == 0)
           printf("Error: Maximum 16 processes allowed.\n");
       MPI_Finalize();
       return 1;
   }
   int array_size = MAX_ARRAY_SIZE;              // Number of elements in the array
   int array[MAX_ARRAY_SIZE];                    // Array to be processed (only used by
master)
   int result[MAX_ARRAY_SIZE];                   // Array to store squared results
   int segment[MAX_SEGMENT_SIZE];                // Buffer for segment each worker will
process
   int segment_size = array_size / (size - 1); // Basic segment size (excluding master)
   int remainder = array_size % (size - 1);    // Remainder to be distributed among first
few workers
   if (rank == 0)
   { // Master process
       // Initialize the array with values from 1 to 16
       for (int i = 0; i < array_size; i++)
       {
           array[i] = i + 1;
       }
       // Array to hold MPI request objects for sends (size + data) to each worker
       MPI_Request requests[2 * (size - 1)];
       int req_index = 0;
       int offset = 0; // Keeps track of where we are in the array
       // Distribute array segments using non-blocking sends
       for (int i = 1; i < size; i++)
       {
           int send_size = segment_size;
           // Distribute remainder elements to first few workers
           if (i <= remainder)
           {
               send_size += 1;
```

```
            }
            // Non-blocking send of segment size to worker i
            MPI_Isend(&send_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &requests[req_index++]);

            // Non-blocking send of actual segment data to worker i
            MPI_Isend(&array[offset], send_size, MPI_INT, i, 0, MPI_COMM_WORLD,
&requests[req_index++]);

            offset += send_size;
        }
        // Wait for all non-blocking sends to complete
        MPI_Waitall(req_index, requests, MPI_STATUSES_IGNORE);
        // Reset index for receiving results
        req_index = 0;
        offset = 0;
        // Reuse the same requests array to receive results from workers
        for (int i = 1; i < size; i++)
        {
            int recv_size = segment_size;
            if (i <= remainder)
            {
                recv_size += 1;
            }
            // Non-blocking receive from worker i
            MPI_Irecv(&result[offset], recv_size, MPI_INT, i, 0, MPI_COMM_WORLD,
&requests[req_index++]);

            offset += recv_size;
        }
        // Wait for all results to be received
        MPI_Waitall(req_index, requests, MPI_STATUSES_IGNORE);

        // Print the final array containing squared values
        printf("Final squared array: ");
        for (int i = 0; i < array_size; i++)
        {
            printf("%d ", result[i]);
        }
        printf("\n");
    }
    else
    {                            // Worker processes
        MPI_Request requests[2]; // One for size, one for data
        int recv_size;
        // Non-blocking receive of segment size from master
        MPI_Irecv(&recv_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &requests[0]);
        // Non-blocking receive of segment data from master
        // Note: we allocate full buffer but only process recv_size
        MPI_Irecv(segment, MAX_SEGMENT_SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD, &requests[1]);
```

```
        // Wait until both receives are done
        MPI_Waitall(2, requests, MPI_STATUSES_IGNORE);
        // Square each element in the received segment
        for (int i = 0; i < recv_size; i++)
        {
            segment[i] = segment[i] * segment[i];
        }
        // Non-blocking send of the computed segment back to master
        MPI_Request send_request;
        MPI_Isend(segment, recv_size, MPI_INT, 0, 0, MPI_COMM_WORLD, &send_request);
        // Ensure send completes before process exits
        MPI_Wait(&send_request, MPI_STATUS_IGNORE);
    }
    // Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

Output:

```
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_Assignment#2$ mpicc -o task2 task2.c
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_Assignment#2$ mpirun -np 4 --hostfile myhostfile ./task2
Final squared array: 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256
```

a. Explain why `MPI_Waitall` is needed.
Ans: MPI_Waitall ensures that all non-blocking operations (MPI_Isend and MPI_Irecv) in the request array have completed before the program proceeds. For the Master, it guarantees that all segment sizes and data are sent before receiving results, and all results are received before printing. For Workers, it ensures both segment size and data are received before computing squares. Without MPI_Waitall, the program might access incomplete or invalid data, leading to undefined behavior.

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

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

#define MAX_ARRAY_SIZE 16
#define MAX_SEGMENT_SIZE 16 // Maximum segment size for workers

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

    if (size > 16) {
```

```
        if (rank == 0) printf("Error: Maximum 16 processes allowed.\n");
        MPI_Finalize();
        return 1;
    }

    int array_size = MAX_ARRAY_SIZE;
    int array[MAX_ARRAY_SIZE];
    int result[MAX_ARRAY_SIZE];
    int segment[MAX_SEGMENT_SIZE];
    int segment_size = array_size / (size - 1); // Workers = size - 1 (master excluded)
    int remainder = array_size % (size - 1);

    if (rank == 0) { // Master
        // Initialize array
        for (int i = 0; i < array_size; i++) {
            array[i] = i + 1; // Example: 1 to 16
        }

        // Array to store send requests
        MPI_Request requests[2 * (size - 1)]; // 2 requests per worker (size + data)
        int req_index = 0;

        // Distribute segments using non-blocking sends
        int offset = 0;
        for (int i = 1; i < size; i++) {
            int send_size = segment_size;
            if (i <= remainder) {
                send_size = send_size + 1;
            }
            // Non-blocking send for segment size
            MPI_Isend(&send_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &requests[req_index++]);
            // Non-blocking send for segment data
            MPI_Isend(&array[offset], send_size, MPI_INT, i, 0, MPI_COMM_WORLD,
&requests[req_index++]);
            offset = offset + send_size;
        }

        // Omitted MPI_Waitall for sends

        // Array to store receive requests
        req_index = 0;
        offset = 0;
        for (int i = 1; i < size; i++) {
            int recv_size = segment_size;
            if (i <= remainder) {
                recv_size = recv_size + 1;
            }
            // Non-blocking receive for squared segment
```

```c
            MPI_Irecv(&result[offset], recv_size, MPI_INT, i, 0, MPI_COMM_WORLD,
&requests[req_index++]);
            offset = offset + recv_size;
        }

        // Omitted MPI_Waitall for receives

        // Print final array (likely before receives complete)
        printf("Final squared array: ");
        for (int i = 0; i < array_size; i++) {
            printf("%d ", result[i]);
        }
        printf("\n");
    } else { // Workers
        MPI_Request requests[2]; // For receiving size and data
        int recv_size;

        // Non-blocking receive for segment size
        MPI_Irecv(&recv_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &requests[0]);
        // Non-blocking receive for segment data
        MPI_Irecv(segment, MAX_SEGMENT_SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD, &requests[1]);

        // Omitted MPI_Waitall for receives

        // Compute squares (using potentially uninitialized recv_size and segment)
        for (int i = 0; i < recv_size; i++) {
            segment[i] = segment[i] * segment[i];
        }

        // Non-blocking send for squared segment
        MPI_Request send_request;
        MPI_Isend(segment, recv_size, MPI_INT, 0, 0, MPI_COMM_WORLD, &send_request);

        // Omitted MPI_Wait for send
    }

    MPI_Finalize();
    return 0;
}
```

Output:

```
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_Assignment#2$ mpicc -o task2_b task2_b.c
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_Assignment#2$ mpirun -np 4 --hostfile myhostfile ./task2_b
Final squared array: 2048 0 3801088 0 49152 0 7602176 0 7602176 0 1310720 0 49152 0 902882984 32767
```

Ans: Omitting MPI_Waitall causes the Master to print result before receives complete, leading to garbage values (e.g., 2648, 380168, 49152).

Workers access recv_size and segment before data arrives, causing incorrect computations or crashes.

The output is inconsistent, often showing zeros or random numbers due to uninitialized memory.

This demonstrates the critical need for MPI_Waitall to ensure communication completion and data integrity.

Task 3: Custom Communication Protocol

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

#define ARRAY_SIZE 8 // Number of elements in each array

int main(int argc, char *argv[])
{
   int rank, size;
   // Initialize the MPI environment
   MPI_Init(&argc, &argv);
   // Get the rank (ID) of the current process
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   // Get the total number of processes
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   // Ensure there are at least 4 processes to run this program
   if (size < 4)
   {
       if (rank == 0)
           printf("Error: At least 4 processes required.\n");
       MPI_Finalize();
       return 1;
   }
   int array1[ARRAY_SIZE];        // First array (initialized by Process 0)
   int array2[ARRAY_SIZE];        // Second array (initialized by Process 0)
   int result1[ARRAY_SIZE];       // Stores squared values from array1 (computed by Process
1)
   int result2[ARRAY_SIZE];       // Stores squared values from array2 (computed by Process
2)
   int final_result[ARRAY_SIZE]; // Final aggregated array (computed by Process 3)
   if (rank == 0)
   {
       // -------------------- Process 0: Data Distributor --------------------
       // Initialize both arrays
       for (int i = 0; i < ARRAY_SIZE; i++)
       {
           array1[i] = i + 1;       // array1: 1 to 8
           array2[i] = (i + 1) * 2; // array2: 2 to 16
```

```c
        }
        // Send array1 to Process 1 using tag 10
        MPI_Send(array1, ARRAY_SIZE, MPI_INT, 1, 10, MPI_COMM_WORLD);
        // Send array2 to Process 2 using tag 20
        MPI_Send(array2, ARRAY_SIZE, MPI_INT, 2, 20, MPI_COMM_WORLD);
    }
    else if (rank == 1)
    {
        // -------------------- Process 1: Compute squares of array1 --------------------
        MPI_Status status;
        // Receive array1 from Process 0 with tag 10
        MPI_Recv(array1, ARRAY_SIZE, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
        // Compute square of each element in array1
        for (int i = 0; i < ARRAY_SIZE; i++)
        {
            result1[i] = array1[i] * array1[i];
        }
        // Send result1 to Process 3 using tag 30
        MPI_Send(result1, ARRAY_SIZE, MPI_INT, 3, 30, MPI_COMM_WORLD);
    }
    else if (rank == 2)
    {
        // -------------------- Process 2: Compute squares of array2 --------------------
        MPI_Status status;
        // Receive array2 from Process 0 with tag 20
        MPI_Recv(array2, ARRAY_SIZE, MPI_INT, 0, 20, MPI_COMM_WORLD, &status);
        // Compute square of each element in array2
        for (int i = 0; i < ARRAY_SIZE; i++)
        {
            result2[i] = array2[i] * array2[i];
        }
        // Send result2 to Process 3 using tag 40
        MPI_Send(result2, ARRAY_SIZE, MPI_INT, 3, 40, MPI_COMM_WORLD);
    }
    else if (rank == 3)
    {
        // -------------------- Process 3: Aggregate results --------------------
        MPI_Status status;
        // Dynamically receive two arrays from Processes 1 and 2
        for (int i = 0; i < 2; i++)
        {
            // Determine which result buffer to receive into
            MPI_Recv(i == 0 ? result1 : result2, ARRAY_SIZE, MPI_INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);

            // Print info about the received message
            int source = status.MPI_SOURCE;
            int tag = status.MPI_TAG;
            printf("Process 3 received message from Process %d with tag %d\n", source, tag);
```

```
        }
        // Aggregate: sum corresponding elements of result1 and result2
        for (int i = 0; i < ARRAY_SIZE; i++)
        {
            final_result[i] = result1[i] + result2[i];
        }
        // Display the final aggregated array
        printf("Final aggregated array: ");
        for (int i = 0; i < ARRAY_SIZE; i++)
        {
            printf("%d ", final_result[i]);
        }
        printf("\n");
    }
    // Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

Output:

```
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_Assignment#2$ mpicc -o task3 task3.c
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_Assignment#2$ mpirun -np 4 --hostfile myhostfile ./task3
Process 3 received message from Process 2 with tag 40
Process 3 received message from Process 1 with tag 30
Final aggregated array: 5 20 45 80 125 180 245 320
```

a. How do message tags help in handling multiple simultaneous messages?
Ans: Message tags allow processes to differentiate between multiple messages arriving simultaneously. They act as identifiers, ensuring a process matches the correct message to its intended operation.
For example, Process 3 uses tags 30 and 40 to distinguish results from Processes 1 and 2. This prevents confusion and ensures proper message handling in complex communication patterns.

b. What can go wrong if two messages arrive with the same tag from different sources?
Ans: If two messages have the same tag, the receiver might process them in the wrong order or mix them up. For instance, if Process 3 receives messages from Processes 1 and 2 with tag 30, it may assign the wrong array to result1 or result2. This leads to incorrect aggregation, producing wrong results (e.g., summing the same array twice). MPI_Status can help identify the source, but identical tags still risk misinterpretation of message intent.

Task 4: Implement Circular Ping-Pong

Solution:

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

int main(int argc, char *argv[])
{
```

```
int rank, size;
// Initialize the MPI environment
MPI_Init(&argc, &argv);
// Get the rank (ID) of the current process
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// Get the total number of processes
MPI_Comm_size(MPI_COMM_WORLD, &size);
// Ensure there are at least 4 processes for the ring to function meaningfully
if (size < 4)
{
    if (rank == 0)
        printf("Error: At least 4 processes required.\n");
    MPI_Finalize();
    return 1;
}
const int M = 3;   // Number of complete cycles to perform around the ring
int counter = 0;   // Shared counter passed around the ring
int terminate = 0; // Flag to indicate when to terminate
MPI_Status status; // MPI status object to get message metadata

// Determine neighbors in the ring topology
int next = (rank + 1) % size;         // Next process in ring
int prev = (rank - 1 + size) % size; // Previous process in ring (wrap around)
if (rank == 0)
{
    // Only Process 0 starts the ring communication
    counter = 0;
    printf("Process 0 starting with counter: %d\n", counter);
    // Send initial counter to next process with tag 1 (active message)
    MPI_Send(&counter, 1, MPI_INT, next, 1, MPI_COMM_WORLD);
}
// Loop until termination flag is set
while (!terminate)
{
    // Receive message from previous process (could be counter or termination signal)
    MPI_Recv(&counter, 1, MPI_INT, prev, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    if (status.MPI_TAG == 0)
    {
        // If tag is 0, this is a termination signal
        terminate = 1;
        // Forward termination signal to next process
        MPI_Send(&counter, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
        break;
    }
    // If not termination, increment counter
    counter++;
    if (rank == 0)
    {
        // Process 0 checks if the desired number of cycles has been completed
```

```
            printf("Process 0 received counter: %d\n", counter);
            if (counter >= M * size)
            {
                // If M full cycles completed, initiate termination
                printf("Process 0 initiating termination after %d cycles\n", M);
                terminate = 1;
                // Send termination message to next process
                MPI_Send(&counter, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
                break;
   // Finalize MPI
   MPI_Finalize();
   return 0;
}            }
        }

        // Forward the updated counter to the next process
        MPI_Send(&counter, 1, MPI_INT, next, 1, MPI_COMM_WORLD);
   }
```

Output:

```
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_Assignment#2$ mpicc -o task4 task4.c
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_Assignment#2$ mpirun -np 4 --hostfile myhostfile ./task4
Process 0 starting with counter: 0
Process 0 received counter: 4
Process 0 received counter: 8
Process 0 received counter: 12
Process 0 initiating termination after 3 cycles
```

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

Ans:

Deadlocks can occur if processes wait for messages in the wrong order, causing a cyclic dependency. Improper termination might leave processes waiting indefinitely, especially without a clear exit signal. Message ordering issues can arise if tags or buffers are mismanaged, leading to incorrect data passing. Scalability is limited as communication latency grows linearly with the number of processes in the ring.

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

Solution:

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

int main(int argc, char *argv[]) {
   int rank, size;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```c
    if (size < 4) {
        if (rank == 0) printf("Error: At least 4 processes required.\n");
        MPI_Finalize();
        return 1;
    }

    const int M = 3; // Number of cycles
    int counter_cw = 0; // Clockwise counter
    int counter_ccw = 0; // Counterclockwise counter
    int terminate = 0; // Termination flag
    MPI_Status status[2];
    MPI_Request requests[2];

    // Define neighbors
    int next = (rank + 1) % size; // Clockwise
    int prev = (rank - 1 + size) % size; // Counterclockwise

    if (rank == 0) {
        // Start the ring in both directions
        counter_cw = 0;
        counter_ccw = 0;
        printf("Process 0 starting with clockwise counter: %d, counterclockwise counter: 
%d\n", counter_cw, counter_ccw);
        MPI_Send(&counter_cw, 1, MPI_INT, next, 1, MPI_COMM_WORLD); // Clockwise
        MPI_Send(&counter_ccw, 1, MPI_INT, prev, 2, MPI_COMM_WORLD); // Counterclockwise
    }

    int cycles_completed = 0;
    while (!terminate) {
        // Non-blocking receives for both directions
        MPI_Irecv(&counter_cw, 1, MPI_INT, prev, MPI_ANY_TAG, MPI_COMM_WORLD, &requests[0]);
        MPI_Irecv(&counter_ccw, 1, MPI_INT, next, MPI_ANY_TAG, MPI_COMM_WORLD, 
&requests[1]);
        MPI_Waitall(2, requests, status);

        // Check for termination from either direction
        if (status[0].MPI_TAG == 0 || status[1].MPI_TAG == 0) {
            terminate = 1;
        } else {
            // Increment counters
            counter_cw++;
            counter_ccw++;

            if (rank == 0) {
                cycles_completed++;
                printf("Process 0 cycle %d: clockwise counter = %d, counterclockwise counter 
= %d\n", cycles_completed, counter_cw, counter_ccw);
                if (cycles_completed >= M) {
```

```
                    printf("Process 0 initiating termination after %d cycles\n", M);
                    terminate = 1;
                }
            }
        }

        // Send updated counters or termination signal
        if (terminate) {
            MPI_Send(&counter_cw, 1, MPI_INT, next, 0, MPI_COMM_WORLD); // Terminate
clockwise
            MPI_Send(&counter_ccw, 1, MPI_INT, prev, 0, MPI_COMM_WORLD); // Terminate
counterclockwise
        } else {
            MPI_Send(&counter_cw, 1, MPI_INT, next, 1, MPI_COMM_WORLD); // Clockwise
            MPI_Send(&counter_ccw, 1, MPI_INT, prev, 2, MPI_COMM_WORLD); // Counterclockwise
        }
    }

    MPI_Finalize();
    return 0;
}
```

Ans: The bidirectional ring communication in the task4_b.c can take a lot of time or get stuck due to potential deadlocks arising from the synchronous nature of blocking MPI_Recv calls. Each process waits for messages from both clockwise and counterclockwise neighbors, and if one direction is delayed (e.g., due to network latency or mismatched timing), it can create a cyclic dependency where processes block each other. The termination signal (tag 0) might not propagate if a process is stuck, causing the program to hang indefinitely, especially with larger process counts or network delays. The fixed version with non-blocking MPI_Irecv and MPI_Waitall mitigates this by allowing concurrent message handling, ensuring smoother progress and timely termination.

Task 5: Performance Timing and Barriers

Solution:

Block:

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

#define MAX_ARRAY_SIZE 16   // Total number of elements in the array
#define MAX_SEGMENT_SIZE 16 // Maximum size of the segment each worker can handle

int main(int argc, char *argv[])
{
    int rank, size;

```

```c
// Initialize the MPI environment
MPI_Init(&argc, &argv);

// Get the rank (ID) of the current process
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// Get the total number of processes
MPI_Comm_size(MPI_COMM_WORLD, &size);

// Ensure no more than 16 processes are used
if (size > 16)
{
    if (rank == 0)
        printf("Error: Maximum 16 processes allowed.\n");
    MPI_Finalize();
    return 1;
}

int array_size = MAX_ARRAY_SIZE;             // Total elements in the main array
int array[MAX_ARRAY_SIZE];                   // Original array (used only by master)
int result[MAX_ARRAY_SIZE];                  // Final result array to store squares
int segment[MAX_SEGMENT_SIZE];               // Buffer for each worker's segment
int segment_size = array_size / (size - 1); // Base segment size for each worker
int remainder = array_size % (size - 1);     // Extra elements to distribute evenly

double start_time, end_time;
// Synchronize all processes before starting the computation
MPI_Barrier(MPI_COMM_WORLD);
start_time = MPI_Wtime();

if (rank == 0)
{ // Master process
    // Initialize the array with values 1 to 16
    for (int i = 0; i < array_size; i++)
    {
        array[i] = i + 1;
    }

    // Distribute segments of the array to worker processes
    int offset = 0;
    for (int i = 1; i < size; i++)
    {
        int send_size = segment_size;

        // Distribute remaining elements evenly to the first few workers
        if (i <= remainder)
        {
            send_size += 1;
        }
```

```
            // Send the size of the segment
            MPI_Send(&send_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD);

            // Send the actual segment of the array
            MPI_Send(&array[offset], send_size, MPI_INT, i, 0, MPI_COMM_WORLD);

            offset += send_size;
        }

        // Collect results from workers
        offset = 0;
        for (int i = 1; i < size; i++)
        {
            int recv_size = segment_size;
            if (i <= remainder)
            {
                recv_size += 1;
            }

            // Receive the squared segment from each worker
            MPI_Recv(&result[offset], recv_size, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

            offset += recv_size;
        }

        // Print the final squared array
        printf("Final squared array: ");
        for (int i = 0; i < array_size; i++)
        {
            printf("%d ", result[i]);
        }
        printf("\n");
    }
    else
    { // Worker processes
        int recv_size;

        // Receive the size of the segment from the master
        MPI_Recv(&recv_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // Receive the actual segment from the master
        MPI_Recv(segment, recv_size, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // Compute the square of each element in the segment
        for (int i = 0; i < recv_size; i++)
        {
            segment[i] = segment[i] * segment[i];
```

```
        }

        // Send the squared segment back to the master
        MPI_Send(segment, recv_size, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    // Synchronize all processes after computation
    MPI_Barrier(MPI_COMM_WORLD);
    end_time = MPI_Wtime();

    if (rank == 0)
    {
        printf("Blocking communication time: %f seconds\n", end_time - start_time);
    }

    // Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

Non Block:

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

#define MAX_ARRAY_SIZE 16   // Total number of elements in the array
#define MAX_SEGMENT_SIZE 16 // Maximum size a worker can receive

int main(int argc, char *argv[])
{
    int rank, size;

    // Initialize MPI environment
    MPI_Init(&argc, &argv);

    // Get the rank (ID) of this process
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Get the total number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Check if number of processes exceeds the limit
    if (size > 16)
    {
        if (rank == 0)
            printf("Error: Maximum 16 processes allowed.\n");
        MPI_Finalize();
        return 1;
    }
```

```
    int array_size = MAX_ARRAY_SIZE;                // Number of elements in the array
    int array[MAX_ARRAY_SIZE];                      // Array to be processed (only used by master)
    int result[MAX_ARRAY_SIZE];                     // Array to store squared results
    int segment[MAX_SEGMENT_SIZE];                  // Buffer for segment each worker will process
    int segment_size = array_size / (size - 1); // Basic segment size (excluding master)
    int remainder = array_size % (size - 1);    // Remainder to be distributed among first few workers

    double start_time, end_time;
    // Synchronize all processes before starting the computation
    MPI_Barrier(MPI_COMM_WORLD);
    start_time = MPI_Wtime();

    if (rank == 0)
    { // Master process
        // Initialize the array with values from 1 to 16
        for (int i = 0; i < array_size; i++)
        {
            array[i] = i + 1;
        }

        // Array to hold MPI request objects for sends (size + data) to each worker
        MPI_Request requests[2 * (size - 1)];
        int req_index = 0;
        int offset = 0; // Keeps track of where we are in the array

        // Distribute array segments using non-blocking sends
        for (int i = 1; i < size; i++)
        {
            int send_size = segment_size;

            // Distribute remainder elements to first few workers
            if (i <= remainder)
            {
                send_size += 1;
            }

            // Non-blocking send of segment size to worker i
            MPI_Isend(&send_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &requests[req_index++]);

            // Non-blocking send of actual segment data to worker i
            MPI_Isend(&array[offset], send_size, MPI_INT, i, 0, MPI_COMM_WORLD, &requests[req_index++]);

            offset += send_size;
        }
```

```c
        // Wait for all non-blocking sends to complete
        MPI_Waitall(req_index, requests, MPI_STATUSES_IGNORE);

        // Reset index for receiving results
        req_index = 0;
        offset = 0;

        // Reuse the same requests array to receive results from workers
        for (int i = 1; i < size; i++)
        {
            int recv_size = segment_size;
            if (i <= remainder)
            {
                recv_size += 1;
            }

            // Non-blocking receive from worker i
            MPI_Irecv(&result[offset], recv_size, MPI_INT, i, 0, MPI_COMM_WORLD,
&requests[req_index++]);

            offset += recv_size;
        }

        // Wait for all results to be received
        MPI_Waitall(req_index, requests, MPI_STATUSES_IGNORE);

        // Print the final array containing squared values
        printf("Final squared array: ");
        for (int i = 0; i < array_size; i++)
        {
            printf("%d ", result[i]);
        }
        printf("\n");
    }
    else
    {                               // Worker processes
        MPI_Request requests[2]; // One for size, one for data
        int recv_size;

        // Non-blocking receive of segment size from master
        MPI_Irecv(&recv_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &requests[0]);

        // Non-blocking receive of segment data from master
        // Note: we allocate full buffer but only process recv_size
        MPI_Irecv(segment, MAX_SEGMENT_SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD, &requests[1]);

        // Wait until both receives are done
        MPI_Waitall(2, requests, MPI_STATUSES_IGNORE);
```

```c
        // Square each element in the received segment
        for (int i = 0; i < recv_size; i++)
        {
            segment[i] = segment[i] * segment[i];
        }

        // Non-blocking send of the computed segment back to master
        MPI_Request send_request;
        MPI_Isend(segment, recv_size, MPI_INT, 0, 0, MPI_COMM_WORLD, &send_request);

        // Ensure send completes before process exits
        MPI_Wait(&send_request, MPI_STATUS_IGNORE);
    }

    // Synchronize all processes after computation
    MPI_Barrier(MPI_COMM_WORLD);
    end_time = MPI_Wtime();

    if (rank == 0) {
        printf("Non-blocking communication time: %f seconds\n", end_time - start_time);
    }

    // Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

Output:

```
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_Assignment#2$ mpicc -o task5_block task5_block.c
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_Assignment#2$ mpirun -np 4 --hostfile myhostfile ./task5_block
Final squared array: 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256
Blocking communication time: 0.000148 seconds
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_Assignment#2$ mpicc -o task5_nonBlock task5_nonBlock.c
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_Assignment#2$ mpirun -np 4 --hostfile myhostfile ./task5_nonBlock
Final squared array: 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256
Non-blocking communication time: 0.000134 seconds
```

Report:

Time Taken for Blocking vs Non-Blocking Communication

- Blocking: 0.000148 seconds
- Non-Blocking: 0.000134 seconds
- Analysis: On my system, the non-blocking communication (Task 5: task5_nonBlock.c) is slightly faster at 0.000134 seconds compared to the blocking version (Task 5:

task5_block.c) at 0.000148 seconds, a difference of about 9.5%. This aligns with expectations, as non-blocking communication (MPI_Isend, MPI_Irecv, MPI_Waitall) allows overlapping of communication and computation, reducing total execution time, especially beneficial even in small-scale setups with 4 processes.

Overhead Introduced by Synchronization

MPI_Barrier adds overhead by forcing all processes to wait for the slowest one, delaying faster processes and extending the measured time by their idle periods. In your runs, barriers before and after computation ensure synchronized MPI_Wtime measurements, but if the Master finishes collecting results later, Workers wait at the final barrier. This overhead, likely a small fraction of the 0.000148 seconds in blocking, is more noticeable in small-scale tests where computation is fast. The non-blocking version's slight edge (0.000134 seconds) suggests better workload overlap, mitigating some barrier-induced delays.