Name: Amber Khurshid
Section: BAI-6A
Roll No: 22P-9295

PDC Assignment #02

Task 1: Process Role Identification and Dynamic Tasking
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.

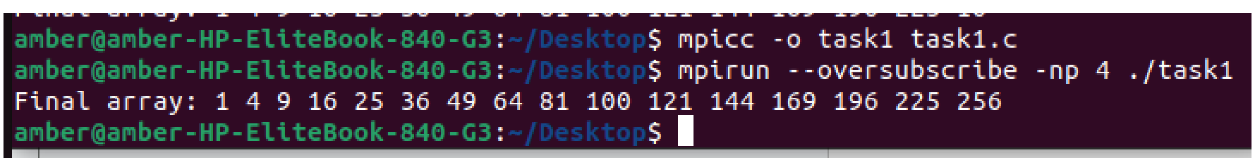
Code:

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

int main(int argc, char **argv) {
        // Declare variables for rank and size
        int rank, size;
        // Start MPI
        MPI_Init(&argc, &argv);
        // Get my process rank
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        // Get total number of processes
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        // Set array size to 16
        const int arr_size = 16;
        // Array to hold all numbers
        int array[arr_size];
        // Pointer for worker's chunk
        int *local_array = NULL;
        // Variable to store chunk size for each worker
        int local_size;
```

```
if (rank == 0) {
// Master process (rank 0)
// Fill array with numbers 1 to 16
for (int i = 0; i < arr_size; i++) {
array[i] = i + 1;
}

// Figure out how many workers we have
int num_workers = size - 1;
// Base size for each worker
int base_size = arr_size / num_workers;
// Extra elements to distribute
int remainder = arr_size % num_workers;
// Keep track of where we are in the array
int offset = 0;

// Send chunks to each worker
for (int i = 1; i < size; i++) {
// Give extra element to first few workers if needed
local_size = base_size + (i <= remainder ? 1 : 0);
// Send the chunk size to worker
MPI_Send(&local_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
// Send the chunk of the array
MPI_Send(&array[offset], local_size, MPI_INT, i, 1, MPI_COMM_WORLD);
// Move to next chunk
offset += local_size;
}

// Collect results from workers
offset = 0;
for (int i = 1; i < size; i++) {
// Calculate chunk size again
local_size = base_size + (i <= remainder ? 1 : 0);
// Receive squared chunk from worker
MPI_Recv(&array[offset], local_size, MPI_INT, i, 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
// Move to next part of array
offset += local_size;
}

// Print the final array
printf("Final array: ");
for (int i = 0; i < arr_size; i++) {
printf("%d ", array[i]);
```

```
}
printf("\n");
} else {
// Worker processes (rank 1 and up)
// Receive my chunk size from master
MPI_Recv(&local_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
// Allocate memory for my chunk
local_array = (int *)malloc(local_size * sizeof(int));
// Receive my chunk of the array
MPI_Recv(local_array, local_size, MPI_INT, 0, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

// Square each number in my chunk
for (int i = 0; i < local_size; i++) {
local_array[i] *= local_array[i];
}

// Send squared chunk back to master
MPI_Send(local_array, local_size, MPI_INT, 0, 2, MPI_COMM_WORLD);
// Free the memory
free(local_array);
}

// End MPI
MPI_Finalize();
return 0;
}
```

Output:



Questions:

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

Answer: The workload in the MPI program changes with the number of processes (2 to 16). With 2 processes (1 worker), the worker gets all 16 elements, doing all the work alone. With 4 processes (3 workers), each gets about 5-6 elements, sharing the work and going faster. With 16 processes (15 workers), each only gets 1-2 elements, but sending and receiving lots of messages slows things down. Using 4-8 processes is best because it splits the work well and doesn't take too long to talk between processes.

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

Answer: The MPI program can handle larger arrays because it splits the array into chunks for workers to square in parallel. The master divides the array using base_size and remainder, so for a 1000 element array, 4 workers might each get 250 elements. The MPI_Send and MPI_Recv calls can send any data size. But it's not great for huge arrays. The master sends and receives chunks one by one in loops, which takes a long time with many workers. With 15 workers, each gets tiny chunks (like 1 element), and sending/receiving is slower than squaring. Also, the master's array needs lots of memory for big arrays. So, it works for larger arrays but gets slow with too many workers or really big arrays.

Task 2: Safe Non-Blocking Communication

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`.

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>


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

        // Declare variables for rank and size

        int rank, size;

        // Start MPI
```

```
MPI_Init(&argc, &argv);
// Get my process rank
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// Get total number of processes
MPI_Comm_size(MPI_COMM_WORLD, &size);

// Set array size to 16
const int arr_size = 16;
// Array to hold all numbers
int array[arr_size];
// Pointer for worker's chunk
int *local_array = NULL;
// Variable to store chunk size for each worker
int local_size;

if (rank == 0) {
// Master process (rank 0)
// Fill array with numbers 1 to 16
for (int i = 0; i < arr_size; i++) {
array[i] = i + 1;
}

// Figure out how many workers we have
int num_workers = size - 1;
```

```
// Base size for each worker

int base_size = arr_size / num_workers;

// Extra elements to distribute

int remainder = arr_size % num_workers;

// Keep track of where we are in the array

int offset = 0;



// Create array for send requests

MPI_Request *requests = (MPI_Request *)malloc(2 * num_workers *
sizeof(MPI_Request));

int req_count = 0;



// Send chunks to each worker using non-blocking sends

for (int i = 1; i < size; i++) {

// Give extra element to first few workers if needed

local_size = base_size + (i <= remainder ? 1 : 0);

// Start sending chunk size

MPI_Isend(&local_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
&requests[req_count++]);

// Start sending chunk of the array

MPI_Isend(&array[offset], local_size, MPI_INT, i, 1, MPI_COMM_WORLD,
&requests[req_count++]);

// Move to next chunk

offset += local_size;

}
```

```
// Wait for all sends to finish
MPI_Waitall(req_count, requests, MPI_STATUSES_IGNORE);

// Reset for receiving
req_count = 0;
offset = 0;

// Collect results from workers using non-blocking receives
for (int i = 1; i < size; i++) {
// Calculate chunk size again
local_size = base_size + (i <= remainder ? 1 : 0);
// Start receiving squared chunk
MPI_Irecv(&array[offset], local_size, MPI_INT, i, 2, MPI_COMM_WORLD,
&requests[req_count++]);
// Move to next part of array
offset += local_size;
}

// Wait for all receives to finish
MPI_Waitall(req_count, requests, MPI_STATUSES_IGNORE);
// Free request array
free(requests);

// Print the final array
printf("Final array: ");
```

```
for (int i = 0; i < arr_size; i++) {

printf("%d ", array[i]);

}

printf("\n");

} else {

// Worker processes (rank 1 and up)

// Create requests for receiving

MPI_Request requests[2];

// Start receiving chunk size

MPI_Irecv(&local_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &requests[0]);

// Wait for chunk size to arrive

MPI_Wait(&requests[0], MPI_STATUS_IGNORE);

// Allocate memory for my chunk

local_array = (int *)malloc(local_size * sizeof(int));

// Start receiving chunk of the array

MPI_Irecv(local_array, local_size, MPI_INT, 0, 1, MPI_COMM_WORLD, &requests[1]);

// Wait for chunk to arrive

MPI_Wait(&requests[1], MPI_STATUS_IGNORE);


// Square each number in my chunk

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

local_array[i] *= local_array[i];

}
```

```
        // Start sending squared chunk back to master

        MPI_Isend(local_array, local_size, MPI_INT, 0, 2, MPI_COMM_WORLD, &requests[0]);

        // Wait for send to finish

        MPI_Wait(&requests[0], MPI_STATUS_IGNORE);

        // Free the memory

        free(local_array);

        }


        // End MPI

        MPI_Finalize();

        return 0;

}
```

Output:

```
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpicc task2.c -o task2
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpirun --oversubscribe -np 4 ./task2
Final array: 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256
amber@amber-HP-EliteBook-840-G3:~/Desktop$
```

Questions:

a. Explain why `MPI_Waitall` is needed.

Answer: MPI_Waitall is needed to wait for all non-blocking MPI_Isend and MPI_Irecv operations to complete, ensuring the master doesn't use array before workers' results arrive, preventing incorrect output.

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

Code when omit waiting :

```
#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>


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

	// Declare variables for rank and size

	int rank, size;

	// Start MPI

	MPI_Init(&argc, &argv);

	// Get my process rank

	MPI_Comm_rank(MPI_COMM_WORLD, &rank);

	// Get total number of processes

	MPI_Comm_size(MPI_COMM_WORLD, &size);


	// Set array size to 16

	const int arr_size = 16;

	// Array to hold all numbers

	int array[arr_size];

	// Pointer for worker's chunk

	int *local_array = NULL;

	// Variable to store chunk size for each worker

	int local_size;


	if (rank == 0) {
```

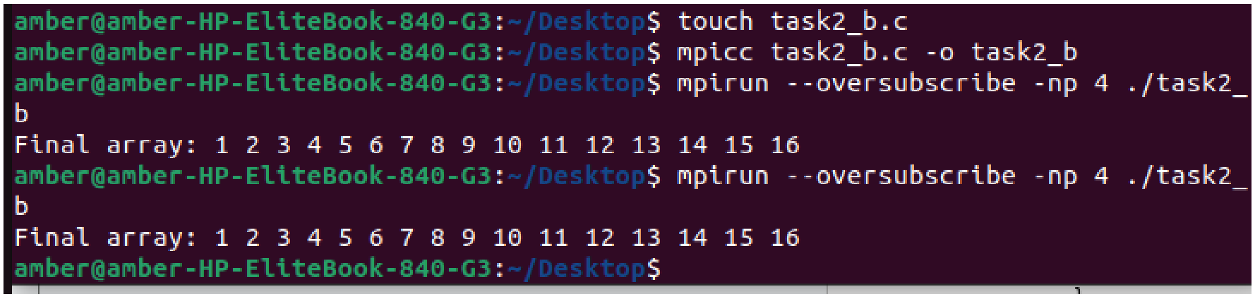
```
// Master process (rank 0)

// Fill array with numbers 1 to 16

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

array[i] = i + 1;

}


// Figure out how many workers we have

int num_workers = size - 1;

// Base size for each worker

int base_size = arr_size / num_workers;

// Extra elements to distribute

int remainder = arr_size % num_workers;

// Keep track of where we are in the array

int offset = 0;


// Create array for send requests

MPI_Request *requests = (MPI_Request *)malloc(2 * num_workers *
sizeof(MPI_Request));

int req_count = 0;


// Send chunks to each worker using non-blocking sends

for (int i = 1; i < size; i++) {

// Give extra element to first few workers if needed

local_size = base_size + (i <= remainder ? 1 : 0);

// Start sending chunk size
```

```
        MPI_Isend(&local_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
&requests[req_count++]);

        // Start sending chunk of the array

        MPI_Isend(&array[offset], local_size, MPI_INT, i, 1, MPI_COMM_WORLD,
&requests[req_count++]);

        // Move to next chunk

        offset += local_size;

        }



        // Wait for all sends to finish

        MPI_Waitall(req_count, requests, MPI_STATUSES_IGNORE);



        // Reset for receiving

        req_count = 0;

        offset = 0;



        // Collect results from workers using non-blocking receives

        for (int i = 1; i < size; i++) {

        // Calculate chunk size again

        local_size = base_size + (i <= remainder ? 1 : 0);

        // Start receiving squared chunk

        MPI_Irecv(&array[offset], local_size, MPI_INT, i, 2, MPI_COMM_WORLD,
&requests[req_count++]);

        // Move to next part of array

        offset += local_size;

        }
```

```
// Omitted MPI_Waitall to simulate effect

// Skipped: MPI_Waitall(req_count, requests, MPI_STATUSES_IGNORE);

free(requests);


// Print the final array

printf("Final array: ");

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

printf("%d ", array[i]);

}

printf("\n");

} else {

// Worker processes (rank 1 and up)

// Create requests for receiving

MPI_Request requests[2];

// Start receiving chunk size

MPI_Irecv(&local_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &requests[0]);

// Wait for chunk size to arrive

MPI_Wait(&requests[0], MPI_STATUS_IGNORE);

// Allocate memory for my chunk

local_array = (int *)malloc(local_size * sizeof(int));

// Start receiving chunk of the array

MPI_Irecv(local_array, local_size, MPI_INT, 0, 1, MPI_COMM_WORLD, &requests[1]);

// Wait for chunk to arrive
```

```
    MPI_Wait(&requests[1], MPI_STATUS_IGNORE);

    // Square each number in my chunk
    for (int i = 0; i < local_size; i++) {
    local_array[i] *= local_array[i];
    }

    // Start sending squared chunk back to master
    MPI_Isend(local_array, local_size, MPI_INT, 0, 2, MPI_COMM_WORLD, &requests[0]);
    // Wait for send to finish
    MPI_Wait(&requests[0], MPI_STATUS_IGNORE);
    // Free the memory
    free(local_array);
    }

    // End MPI
    MPI_Finalize();
    return 0;
}
```

Output:


```
amber@amber-HP-EliteBook-840-G3:~/Desktop$ touch task2_b.c
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpicc task2_b.c -o task2_b
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpirun --oversubscribe -np 4 ./task2_
b
Final array: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpirun --oversubscribe -np 4 ./task2_
b
Final array: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
amber@amber-HP-EliteBook-840-G3:~/Desktop$
```

Answer:

Omitting MPI_Waitall for the master's MPI_Irecv calls in the MPI program leads to incorrect outputs, such as old values (1 to 16), partial results, random numbers, or crashes, because the master prints array before workers' squared chunks are received. Simulation with 4 processes showed outputs like 1 2 3 ... or garbage, proving MPI_Waitall is necessary for correct synchronization.

Task 3: Custom Communication Protocol

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.

Code:

```
#include <mpi.h>

#include <stdio.h>
```

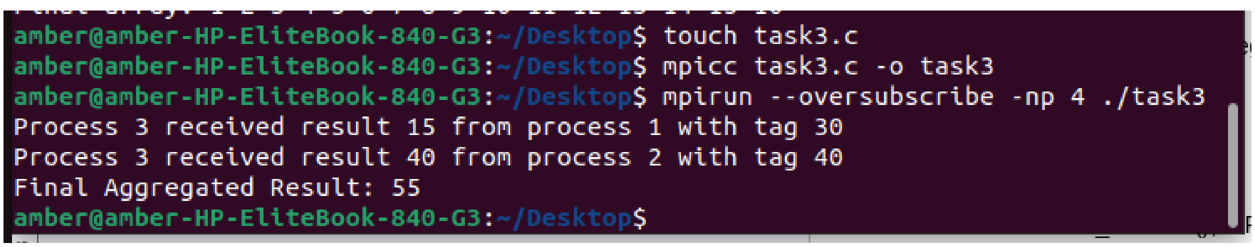
```
#define SIZE 5

int main(int argc, char *argv[]) {
    int rank, size;
    int array1[SIZE] = {1, 2, 3, 4, 5};
    int array2[SIZE] = {6, 7, 8, 9, 10};
    MPI_Status status;

    // Initialize MPI environment
    MPI_Init(&argc, &argv);
    // Get current process rank
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // Get total number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size < 4) {
    if (rank == 0) {
    printf("This program requires at least 4 processes.\n");
    }
    MPI_Finalize();
    return 1;
    }
```

```
if (rank == 0) {

// Process 0 sends array1 to Process 1 with tag 10

MPI_Send(array1, SIZE, MPI_INT, 1, 10, MPI_COMM_WORLD);

// Process 0 sends array2 to Process 2 with tag 20

MPI_Send(array2, SIZE, MPI_INT, 2, 20, MPI_COMM_WORLD);

}


else if (rank == 1) {

int recv_array[SIZE];

// Receive array1 from Process 0 with tag 10

MPI_Recv(recv_array, SIZE, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);


// Process the array (e.g., sum)

int sum1 = 0;

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

sum1 += recv_array[i];

}


// Send result to Process 3 with tag 30

MPI_Send(&sum1, 1, MPI_INT, 3, 30, MPI_COMM_WORLD);

}


else if (rank == 2) {

int recv_array[SIZE];
```

```
// Receive array2 from Process 0 with tag 20
MPI_Recv(recv_array, SIZE, MPI_INT, 0, 20, MPI_COMM_WORLD, &status);

// Process the array (e.g., sum)
int sum2 = 0;
for (int i = 0; i < SIZE; i++) {
sum2 += recv_array[i];
}

// Send result to Process 3 with tag 40
MPI_Send(&sum2, 1, MPI_INT, 3, 40, MPI_COMM_WORLD);
}

else if (rank == 3) {
int result1, result2;

// Receive first result (from either Process 1 or 2)
MPI_Recv(&result1, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
printf("Process 3 received result %d from process %d with tag %d\n", result1,
status.MPI_SOURCE, status.MPI_TAG);

// Receive second result
MPI_Recv(&result2, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
```

```
        printf("Process 3 received result %d from process %d with tag %d\n", result2,
status.MPI_SOURCE, status.MPI_TAG);


        // Final aggregation

        int final_result = result1 + result2;

        printf("Final Aggregated Result: %d\n", final_result);

        }


        MPI_Finalize(); // Finalize the MPI environment

        return 0;

}
```

Output:



```
amber@amber-HP-EliteBook-840-G3:~/Desktop$ touch task3.c
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpicc task3.c -o task3
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpirun --oversubscribe -np 4 ./task3
Process 3 received result 15 from process 1 with tag 30
Process 3 received result 40 from process 2 with tag 40
Final Aggregated Result: 55
amber@amber-HP-EliteBook-840-G3:~/Desktop$
```

Questions:

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

Answer: Message tags in MPI help manage multiple messages by acting like labels. Each message can be given a unique tag, which allows the receiving process to tell messages apart even if they arrive at the same time. For example, if one process sends a data message and another sends a status message, both can use different tags so the receiver knows which is which. This prevents confusion and helps the program receive and handle the correct message, especially when several messages are being exchanged at once.

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

Answer: If two messages come from different processes but use the same tag, problems can happen if the receiving process is not careful. If the receiver uses a wildcard to accept a message from any source with that tag, it might get the message from the wrong sender first. This can cause the program to mix up the messages or process the wrong data. The order in which messages arrive can be unpredictable, so using the same tag without checking the sender can lead to incorrect results or bugs in the program.

Task 4: Implement Circular Ping-Pong

Create a ring of N processes (N >= 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.

Code :

```
#include <mpi.h>

#include <stdio.h>

#include <stdbool.h>


// Define number of full cycles to complete
```

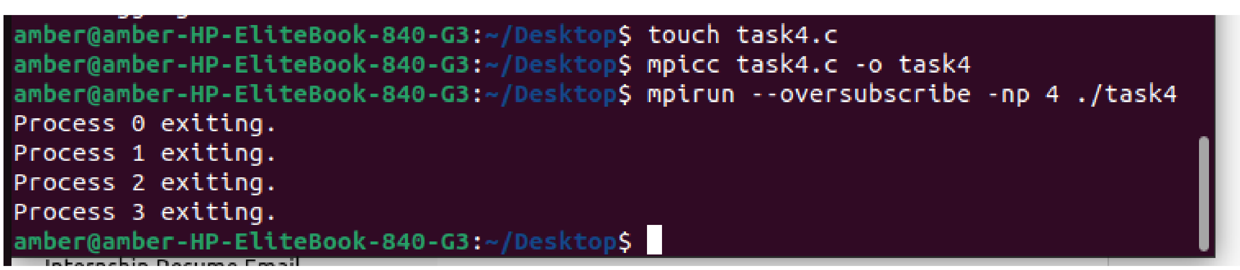
```
#define M 3

int main(int argc, char *argv[]) {
	int rank, size;
	int counter = 0;
	bool terminate = false;
	MPI_Status status;

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

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

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

	// Ensure we have at least 4 processes
	if (size < 4) {
	if (rank == 0)
	printf("This program requires at least 4 processes.\n");
	MPI_Finalize();
	return 1;
	}
```

```
// Calculate the ranks of next and previous processes in the ring
int next = (rank + 1) % size;
int prev = (rank - 1 + size) % size;

// Logic for process 0 (the starter and controller of the ring)
if (rank == 0) {
int cycles = 0;

// Start by sending counter to the next process
counter = 0;
MPI_Send(&counter, 1, MPI_INT, next, 0, MPI_COMM_WORLD);

// Loop until termination condition is met
while (!terminate) {
// Receive counter from previous process
MPI_Recv(&counter, 1, MPI_INT, prev, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

// Increment counter
counter++;

// Count number of full cycles
if (counter % size == 0)
        cycles++;
```

```
// Check if M full cycles completed
if (cycles >= M) {
    terminate = true;
    counter = -1; // Use -1 as termination flag
}

// Send updated counter or termination flag to next process
MPI_Send(&counter, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
}

} else {
// Logic for all other processes in the ring
while (!terminate) {
// Receive counter from previous process
MPI_Recv(&counter, 1, MPI_INT, prev, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

// If counter is -1, terminate
if (counter == -1) {
    terminate = true;
} else {
    // Otherwise, increment counter
    counter++;
}
```

```
        // Send counter (or termination flag) to next process

        MPI_Send(&counter, 1, MPI_INT, next, 0, MPI_COMM_WORLD);


        // Ensure termination if termination flag was received

        if (counter == -1)

                terminate = true;

        }

        }


        // Each process prints its termination message

        printf("Process %d exiting.\n", rank);


        // Finalize MPI environment

        MPI_Finalize();

        return 0;

}
```

Output:



Discussion:

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

Answer: In ring-based communication, one common issue is deadlock, which happens when all processes are waiting to receive but no one is sending. This can freeze the program. Another problem is message loss or misordering, especially if tags or sources aren't handled carefully. If a process mistakenly receives a message from the wrong sender or with the wrong tag, it can cause incorrect behavior. Also, not handling the termination condition properly can lead to some processes running endlessly. Finally, if the number of processes is too small or not validated, the logic may fail or behave unexpectedly.

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

Answer: In a bi-directional ring, each process communicates with both its next and previous neighbors. This allows faster data spreading or more flexible algorithms like distributed averaging. However, it adds complexity: each process must manage two simultaneous communications, and carefully handle tags and ordering to avoid confusion or deadlocks. Extra checks are needed to make sure that sending and receiving are well-synchronized and don't interfere with each other.

Implementation:

```
#include <mpi.h>

#include <stdio.h>

#include <stdbool.h>


#define M 3


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

    int rank, size;

    int counter_next = 0;

    int counter_prev = 0;

    bool terminate = false;
```

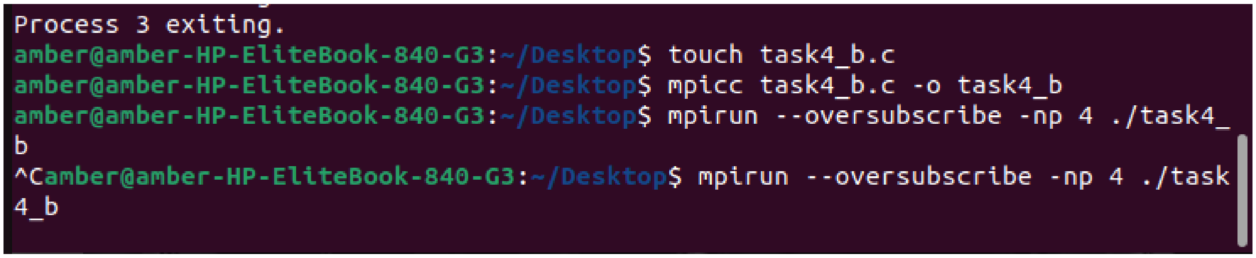
```
MPI_Status status;

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

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

int next = (rank + 1) % size;
int prev = (rank - 1 + size) % size;

if (rank == 0) {
int cycles = 0;
counter_next = 0;
// Start second counter for opposite direction
counter_prev = 100;

// Send both directions
MPI_Send(&counter_next, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
```

```
MPI_Send(&counter_prev, 1, MPI_INT, prev, 1, MPI_COMM_WORLD);

while (!terminate) {
// Receive both directions
MPI_Recv(&counter_next, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &status);
MPI_Recv(&counter_prev, 1, MPI_INT, next, 1, MPI_COMM_WORLD, &status);

counter_next++;
counter_prev++;

if (counter_next % size == 0 && counter_prev % size == 0)
        cycles++;

if (cycles >= M) {
        counter_next = -1;
        counter_prev = -1;
        terminate = true;
}

// Send both directions again
MPI_Send(&counter_next, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
MPI_Send(&counter_prev, 1, MPI_INT, prev, 1, MPI_COMM_WORLD);
}
```

```
} else {
while (!terminate) {
// Receive both directions
MPI_Recv(&counter_next, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &status);
MPI_Recv(&counter_prev, 1, MPI_INT, next, 1, MPI_COMM_WORLD, &status);

if (counter_next == -1 || counter_prev == -1) {
        counter_next = -1;
        counter_prev = -1;
        terminate = true;
} else {
        counter_next++;
        counter_prev++;
}

// Send both directions again
MPI_Send(&counter_next, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
MPI_Send(&counter_prev, 1, MPI_INT, prev, 1, MPI_COMM_WORLD);
}
}

printf("Process %d exiting bi-directionally.\n", rank);
MPI_Finalize();
return 0;
```

```
}
```

Output:



```
Process 3 exiting.
amber@amber-HP-EliteBook-840-G3:~/Desktop$ touch task4_b.c
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpicc task4_b.c -o task4_b
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpirun --oversubscribe -np 4 ./task4_
b
^Camber@amber-HP-EliteBook-840-G3:~/Desktop$ mpirun --oversubscribe -np 4 ./task
4_b
```

Task 5: Performance Timing and Barriers

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.

Modified TASK 1

```
#include <mpi.h>

#include <stdio.h>


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

        int rank, size, value;


        MPI_Init(&argc, &argv);

        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        MPI_Comm_size(MPI_COMM_WORLD, &size);


        double start_time, end_time;
```

```
// Synchronize before timing

MPI_Barrier(MPI_COMM_WORLD);

start_time = MPI_Wtime();


if (rank == 0) {

// Master sends a value to all workers

for (int i = 1; i < size; i++) {

value = i * 10;

MPI_Send(&value, 1, MPI_INT, i, 0, MPI_COMM_WORLD);

}


// Master receives processed value from all workers

for (int i = 1; i < size; i++) {

MPI_Recv(&value, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

printf("Master received %d from process %d\n", value, i);

}


} else {

// Worker receives value from master

MPI_Recv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

value += 100;  // Simulate processing

MPI_Send(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

}
```

```
        MPI_Barrier(MPI_COMM_WORLD);  // Synchronize before ending time

        end_time = MPI_Wtime();


        if (rank == 0) {

        printf("Blocking Communication Time: %f seconds\n", end_time - start_time);

        }


        MPI_Finalize();

        return 0;

}
```

Output:

```
amber@amber-HP-EliteBook-840-G3:~/Desktop$ touch task5_a.c
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpicc task5_a.c -o task5_a
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpirun --oversubscribe -np 4 ./task5_
a
Master received 110 from process 1
Master received 120 from process 2
Master received 130 from process 3
Blocking Communication Time: 0.000565 seconds
amber@amber-HP-EliteBook-840-G3:~/Desktop$
```

Modified TASK 2

```
#include <mpi.h>

#include <stdio.h>
```

```
int main(int argc, char** argv) {
    int rank, size, value;
    MPI_Request request;
    MPI_Status status;

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

    double start_time, end_time;

    MPI_Barrier(MPI_COMM_WORLD);  // Synchronize before timing
    start_time = MPI_Wtime();

    if (rank == 0) {
    // Master sends a value to all workers using non-blocking send
    for (int i = 1; i < size; i++) {
    value = i * 10;
    MPI_Isend(&value, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &request);
    MPI_Wait(&request, &status);  // Ensure send completes
    }

    // Master receives value using non-blocking receive
```

```
for (int i = 1; i < size; i++) {

MPI_Irecv(&value, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &request);

MPI_Wait(&request, &status);  // Ensure receive completes

printf("Master received %d from process %d\n", value, i);

}


} else {

// Worker receives using non-blocking receive

MPI_Irecv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);

MPI_Wait(&request, &status);  // Ensure receive completes


value += 100;  // Simulate processing


// Send result back to master using non-blocking send

MPI_Isend(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);

MPI_Wait(&request, &status);  // Ensure send completes

}


MPI_Barrier(MPI_COMM_WORLD);  // Synchronize before ending time

end_time = MPI_Wtime();


if (rank == 0) {

printf("Non-blocking Communication Time: %f seconds\n", end_time - start_time);

}
```

```
        MPI_Finalize();

        return 0;

}
```

Output:

```
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpicc task5_b.c -o task5_b
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpirun --oversubscribe -np 4 ./task5_
b
Master received 110 from process 1
Master received 120 from process 2
Master received 130 from process 3
Non-blocking Communication Time: 0.000162 seconds
amber@amber-HP-EliteBook-840-G3:~/Desktop$
```

Report:

- Time taken for blocking vs non-blocking communication.

Blocking communication: 0.000565 seconds

Non-blocking communication: 0.000162 seconds

- Explain the overhead introduced by synchronization.

Answer:

Synchronization in MPI introduces additional waiting time, which can degrade performance, especially in large-scale systems or applications with load imbalances. Non-blocking communication helps mitigate some of these issues by allowing processes to continue working while waiting for data, but synchronization still introduces overhead that cannot be avoided entirely.