Assignment no 3

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

Task 1: Basic Understanding of Broadcast

Solution:

```c
#include <stdio.h> // For standard input/output functions
#include <mpi.h>   // For MPI functions

int main(int argc, char **argv)
{
   int rank, value;
   // Initialize the MPI environment
   MPI_Init(&argc, &argv);
   // Get the rank (ID) of the current process
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   // If this is the root process (rank 0), set the value
   if (rank == 0)
   {
       value = 59;
       printf("Process 0: Initial value = %d\n", value);
   }
   else
   {
       // Other processes initialize value to 0
       value = 0;
   }
   // Broadcast the value from process 0 to all processes in
MPI_COMM_WORLD
   MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);
   // All processes (Including 0) print the received value
       printf("Process %d: Received value = %d\n", rank, value);
   // Clean up the MPI environment
```

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

Output:

```
● muhammadabdullah@muhammad-abdullah:~/Documents/pdc_3$ mpicc -o task1 task1.c
● muhammadabdullah@muhammad-abdullah:~/Documents/pdc_3$ mpirun -np 4 --hostfile hostfile ./task1
  Process 0: Initial value = 59
  Process 0: Received value = 59
  Process 1: Received value = 59
  Process 2: Received value = 59
  Process 3: Received value = 59
```

All processes received the same value Broadcasted by process 0.

1. What happens if a non-root process changes the value before the broadcast?
Ans: If a non-root process modifies value before MPI_Bcast, it has no effect on the broadcast. MPI_Bcast overwrites the buffer in non-root processes with the root's data, ensuring all processes receive the root's value (e.g., 59 from process 0).

2. What constraints must be followed when calling MPI_Bcast?
Ans: All processes in the communicator must call MPI_Bcast with matching parameters (buffer, count, datatype, root, communicator). The buffer must be allocated, and the root process's data must be valid. Calls must be synchronized to avoid deadlocks.

3. How does MPI_Bcast differ from point-to-point send/receive operations?
Ans: MPI_Bcast sends data from one root process to all processes in a communicator in a single operation, ensuring synchronization. Point-to-point operations (e.g., MPI_Send, MPI_Recv) involve one sender and one receiver, requiring explicit pairing and potentially multiple calls for multiple recipients.

Task 2: Data Scattering and Gathering (Intermediate)

Solution:

```c
#include <stdio.h> // For standard I/O functions
#include <mpi.h>   // For MPI functions

int main(int argc, char **argv)
{
    int rank, size;
    int full_array[16];  // Array in process 0 holding 16 integers
```

```c
int local_chunk[4];  // Each process will receive 4 integers
int final_array[16]; // Array to gather results back in process 0

// 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 the program runs with exactly 4 processes
if (size != 4)
{
    if (rank == 0)
    {
        printf("This program requires exactly 4 processes.\n");
    }
    MPI_Finalize(); // Exit MPI if the number of processes is incorrect
    return 1;
}

// Process 0 initializes the array with values 1 to 16
if (rank == 0)
{
    for (int i = 0; i < 16; i++)
    {
        full_array[i] = i + 1;
    }

    // Print the initialized array
    printf("Process 0: Initial array: ");
    for (int i = 0; i < 16; i++)
    {
        printf("%d ", full_array[i]);
    }
    printf("\n");
}
```

```
    // Scatter the full_array into chunks of 4 integers to each process
    MPI_Scatter(full_array,       // Send buffer (only used by root)
                4,                // Number of elements sent to each process
                MPI_INT,          // Data type of elements
                local_chunk,      // Receive buffer for each process
                4,                // Number of elements received by each
process
                MPI_INT,          // Data type of elements
                0,                // Root process (source of scatter)
                MPI_COMM_WORLD); // Communicator

    // Each process multiplies its 4-element chunk by 2
    for (int i = 0; i < 4; i++)
    {
        local_chunk[i] *= 2;
    }

    // Each process prints its modified chunk
    printf("Process %d: Local chunk after multiplication: %d %d %d %d\n",
           rank, local_chunk[0], local_chunk[1], local_chunk[2],
local_chunk[3]);

    // Gather the modified chunks back to process 0
    MPI_Gather(local_chunk,       // Send buffer
               4,                 // Number of elements to send
               MPI_INT,           // Data type of elements
               final_array,       // Receive buffer (only used by root)
               4,                 // Number of elements received from each
process
               MPI_INT,           // Data type of elements
               0,                 // Root process (destination of gather)
               MPI_COMM_WORLD); // Communicator
    // Process 0 prints the final gathered array
    if (rank == 0)
    {
        printf("Process 0: Final array after gathering: ");
        for (int i = 0; i < 16; i++)
        {
            printf("%d ", final_array[i]);
        }
```

```
        printf("\n");
    }
    // Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

Output:

```
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_3$ mpicc -o task2 task2.c
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_3$ mpirun -np 4 --hostfile hostfile ./task2
Process 0: Initial array: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Process 2: Local chunk after multiplication: 18 20 22 24
Process 3: Local chunk after multiplication: 26 28 30 32
Process 0: Local chunk after multiplication: 2 4 6 8
Process 1: Local chunk after multiplication: 10 12 14 16
Process 0: Final array after gathering: 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
```

a. What will happen if the number of processes is not evenly divisible by the data size?
Ans: If the data size (e.g., 16 integers) is not evenly divisible by the number of processes (e.g., 5 processes), MPI_Scatter and MPI_Gather will fail. These functions require the data to be split evenly among processes. For 16 integers and 5 processes, each process would need to handle 3.2 integers, which isn't possible since counts must be integers. MPI will typically throw an error or behave unpredictably, as the total data size (sendcount × number of processes) must match the source array size in MPI_Scatter.

Question b is skipped as it said Optional in manual.
But I have tried to do it below.

Solution:

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

int main(int argc, char** argv) {
    int rank, size;
    const int original_size = 16; // Fixed array size
    int chunk_size; // Number of elements per process (for divisible
portion)
    int divisible_size; // Size of the divisible portion
    int remainder; // Remaining elements
    int *full_array = NULL; // Array in process 0
```

```
    int *local_chunk = NULL; // Each process's chunk
    int *final_array = NULL; // To store gathered results

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

    // Compute the divisible portion of the array
    chunk_size = original_size / size; // Base chunk size per process
    divisible_size = chunk_size * size; // Largest size divisible by number
of processes
    remainder = original_size % size; // Remaining elements to be handled
by process 0

    if (chunk_size == 0) {
        if (rank == 0) {
            printf("Error: Too many processes (%d) for array size %d. Each
process must get at least 1 element.\n", size, original_size);
        }
        MPI_Finalize();
        return 1;
    }

    // Allocate arrays
    local_chunk = (int*)malloc(chunk_size * sizeof(int));
    if (rank == 0) {
        full_array = (int*)malloc(original_size * sizeof(int));
        final_array = (int*)malloc(original_size * sizeof(int));

        // Initialize the array with values 1 to 16
        for (int i = 0; i < original_size; i++) {
            full_array[i] = i + 1;
        }
        printf("Process 0: Initial array (size %d): ", original_size);
        for (int i = 0; i < original_size; i++) {
            printf("%d ", full_array[i]);
        }
        printf("\n");
    }
```

```c
    // Scatter the divisible portion of the array to all processes
    MPI_Scatter(full_array, chunk_size, MPI_INT, local_chunk, chunk_size,
MPI_INT, 0, MPI_COMM_WORLD);

    // Each process multiplies its chunk by 2
    for (int i = 0; i < chunk_size; i++) {
        local_chunk[i] *= 2;
    }
    printf("Process %d: Local chunk after multiplication: ", rank);
    for (int i = 0; i < chunk_size; i++) {
        printf("%d ", local_chunk[i]);
    }
    printf("\n");

    // Gather the modified chunks back to process 0
    MPI_Gather(local_chunk, chunk_size, MPI_INT, final_array, chunk_size,
MPI_INT, 0, MPI_COMM_WORLD);

    // Process 0 handles the remaining elements (if any)
    if (rank == 0 && remainder > 0) {
        printf("Process 0: Handling %d remaining elements: ", remainder);
        for (int i = divisible_size; i < original_size; i++) {
            final_array[i] = full_array[i] * 2;
            printf("%d ", final_array[i]);
        }
        printf("\n");
    }

    // Process 0 prints the final array
    if (rank == 0) {
        printf("Process 0: Final array after gathering: ");
        for (int i = 0; i < original_size; i++) {
            printf("%d ", final_array[i]);
        }
        printf("\n");
    }

    // Clean up
    free(local_chunk);
```

```
    if (rank == 0) {
        free(full_array);
        free(final_array);
    }

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

Output:

```
● muhammadabdullah@muhammad-abdullah:~/Documents/pdc_3$ mpicc -o task2_b task2_b.c
● muhammadabdullah@muhammad-abdullah:~/Documents/pdc_3$ mpirun -np 4 --hostfile hostfile ./task2_b
  Process 0: Initial array (size 16): 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
  Process 0: Local chunk after multiplication: 2 4 6 8
  Process 0: Final array after gathering: 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
  Process 1: Local chunk after multiplication: 10 12 14 16
  Process 2: Local chunk after multiplication: 18 20 22 24
  Process 3: Local chunk after multiplication: 26 28 30 32
```

Task 3: Distributed Reduction and All-Gather (Advanced)

Solution:

```
#include <stdio.h>       // For input/output functions
#include <stdlib.h>      // For malloc and rand
#include <mpi.h>         // For MPI functions
#include <time.h>        // For seeding the random number generator

int main(int argc, char** argv) {
    int rank, size;              // Rank of the process and total number of
processes
    int local_number;            // Random number generated by each process
    int *all_numbers = NULL;     // Array to store gathered numbers from all
processes
    double max_value, avg_value; // To store max and average values
    double start_time, end_time; // For timing MPI operations
    int op_count = 1;            // Number of operations per process (for
reporting)

    // Initialize MPI environment
```

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

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

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

    // Seed the random number generator uniquely for each process
    srand(time(NULL) + rank);

    // Each process generates a random number between 1 and 100
    local_number = (rand() % 100) + 1;
    printf("Process %d: Generated number = %d\n", rank, local_number);

    // Allocate memory to hold the gathered numbers from all processes
    all_numbers = (int*)malloc(size * sizeof(int));

    // ---------- MPI_Allgather ----------
    // All processes share their local number with every other process
    start_time = MPI_Wtime();  // Start timing
    MPI_Allgather(&local_number, 1, MPI_INT, all_numbers, 1, MPI_INT,
MPI_COMM_WORLD);
    end_time = MPI_Wtime();    // End timing

    // Print the time taken for MPI_Allgather and total operations
    printf("Process %d: MPI_Allgather time = %.6f seconds, operations =
%d\n",
           rank, end_time - start_time, op_count * size);

    // Only process 0 prints all gathered numbers
    if (rank == 0) {
        printf("Process 0: All gathered numbers: ");
        for (int i = 0; i < size; i++) {
            printf("%d ", all_numbers[i]);
        }
        printf("\n");
    }

    // ---------- MPI_Reduce ----------
```

```c
    // Compute the maximum value across all processes and send the result
to process 0
    start_time = MPI_Wtime();  // Start timing
    MPI_Reduce(&local_number, &max_value, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);
    end_time = MPI_Wtime();    // End timing

    // Only process 0 prints the maximum value and time taken
    if (rank == 0) {
        printf("Process 0: MPI_Reduce time = %.6f seconds, operations = %d,
Max value = %.0f\n",
               end_time - start_time, op_count, max_value);
    }

    // ---------- MPI_Allreduce ----------
    // Compute the sum of all local numbers and distribute the result to
all processes
    start_time = MPI_Wtime();  // Start timing
    MPI_Allreduce(&local_number, &avg_value, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
    end_time = MPI_Wtime();    // End timing

    // Calculate average by dividing the sum by the number of processes
    avg_value /= size;

    // Each process prints the average value and time taken
    printf("Process %d: MPI_Allreduce time = %.6f seconds, operations = %d,
Average value = %.2f\n",
           rank, end_time - start_time, op_count, avg_value);

    // Free dynamically allocated memory
    free(all_numbers);

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

Output:

```
● muhammadabdullah@muhammad-abdullah:~/Documents/pdc_3$ mpicc -o task3 task3.c
● muhammadabdullah@muhammad-abdullah:~/Documents/pdc_3$ mpirun -np 6 --oversubscribe ./task3
  Process 1: Generated number = 37
  Process 2: Generated number = 5
  Process 5: Generated number = 84
  Process 0: Generated number = 43
  Process 3: Generated number = 62
  Process 4: Generated number = 47
  Process 0: MPI_Allgather time = 0.000160 seconds, operations = 6
  Process 0: All gathered numbers: 43 37 5 62 47 84
  Process 4: MPI_Allgather time = 0.000098 seconds, operations = 6
  Process 2: MPI_Allgather time = 0.000256 seconds, operations = 6
  Process 5: MPI_Allgather time = 0.000457 seconds, operations = 6
  Process 1: MPI_Allgather time = 0.000439 seconds, operations = 6
  Process 3: MPI_Allgather time = 0.000355 seconds, operations = 6
  Process 5: MPI_Allreduce time = 0.000067 seconds, operations = 1, Average value = 0.00
  Process 0: MPI_Reduce time = 0.000293 seconds, operations = 1, Max value = 0
  Process 0: MPI_Allreduce time = 0.000029 seconds, operations = 1, Average value = 0.00
  Process 1: MPI_Allreduce time = 0.000027 seconds, operations = 1, Average value = 0.00
  Process 4: MPI_Allreduce time = 0.000270 seconds, operations = 1, Average value = 0.00
  Process 3: MPI_Allreduce time = 0.000033 seconds, operations = 1, Average value = 0.00
  Process 2: MPI_Allreduce time = 0.000073 seconds, operations = 1, Average value = 0.00
```

a. Why is MPI_Allgather more expensive than MPI_Gather?
Ans: MPI_Allgather sends data from all processes to all processes (O(p⋅n)O(p \cdot n)O(p⋅n) data movement, ppp processes, nnn data size), while MPI_Gather sends data only to the root (O(n⋅p)O(n \cdot p)O(n⋅p) to one process), reducing communication and synchronization costs.

b. Can MPI_Allreduce be used as a substitute for Reduce+Broadcast? Explain.
Ans: Yes, MPI_Allreduce combines reduction and distribution in one step, acting as a substitute for MPI_Reduce followed by MPI_Bcast. It's more efficient by reducing communication steps, though it may use more temporary memory.

c. What challenges arise if the data is large (e.g., arrays of size 1 million)?
Ans: Large data increases memory usage (each process stores the full array), communication overhead (higher latency and bandwidth usage), and synchronization delays (potential bottlenecks). Scalability worsens as all-to-all operations grow quadratically.