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

PDC ASSIGNMENT 03

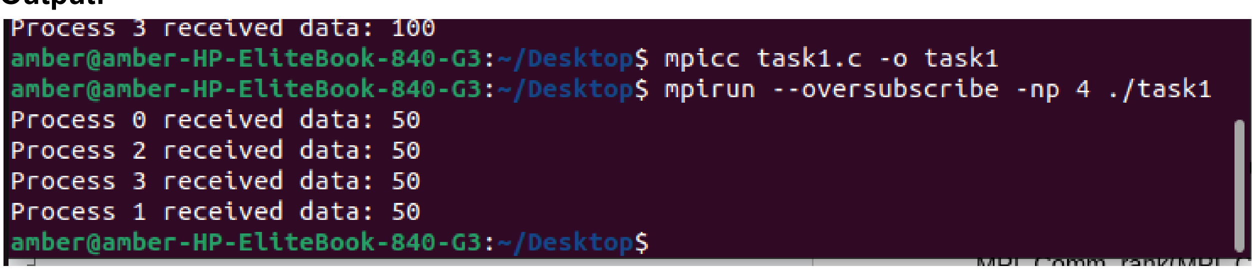
Task 1: Basic Understanding of Broadcast
Write an MPI program where:
- Process 0 initializes an integer variable.
- Use MPI_Bcast to broadcast this variable to all other processes.
- Every process should print the received value.

Code:

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

int main(int argc, char** argv) {
MPI_Init(NULL, NULL);
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int data;
if (world_rank == 0) {
// Only process 0 sets the data
data = 50;
}
// Broadcast data from process 0 to all other processes
MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
// Now all processes have the same value of data
printf("Process %d received data: %d\n", world_rank, data);
MPI_Finalize();
return 0;}
```

Output:



Questions:
1. What happens if a non-root process changes the value before the broadcast?

Answer: If a non-root process changes the value before the broadcast, its change is ignored. MPI_Bcast uses the root process's (rank 0) data, overwriting the data variable in all processes.

2. What constraints must be followed when calling MPI_Bcast?

Answer: All processes in the communicator must call MPI_Bcast with identical parameters (buffer, count, datatype, root, communicator), except the non-root buffer content is ignored. The root must have valid data, and the call must occur between MPI_Init and MPI_Finalize.

3. How does MPI_Bcast differ from point-to-point send/receive operations?

Answer: MPI_Bcast is a collective operation that sends data from one root process to all processes in a communicator simultaneously, while point-to-point operations like MPI_Send and MPI_Recv involve direct communication between only two processes.

Task 2: Data Scattering and Gathering (Intermediate)
Write a program that:
- Initializes an array of 16 integers in process 0.
- Use MPI_Scatter to send 4 integers to each of 4 processes.
- Each process multiplies its chunk by 2.
- Use MPI_Gather to collect the updated data back to process 0.
- Process 0 should print the final array.

Code:

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


int main(int argc, char** argv) {
	MPI_Init(NULL, NULL);

	int world_rank, world_size;
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Creating buffers for sending and receiving data
int sendbuf[16];

// receive buffer for 4 integers per process
int recvbuf[4];

// buffer to collect all results in process 0
int recv_all[16];

// Setting up initial values in process 0
if (world_rank == 0) {
for (int i = 0; i < 16; i++) {

sendbuf[i] = i + 1;
}
}

// Distributing data from process 0 to all processes
MPI_Scatter(sendbuf, 4, MPI_INT, recvbuf, 4, MPI_INT, 0, MPI_COMM_WORLD);

// Displaying the values  process received
for (int i = 0; i < 4; i++) {
printf("Process %d received value %d: %d\n", world_rank, i, recvbuf[i]);
}

// Multiplying received chunk by 2 in each process
for (int i = 0; i < 4; i++) {

recvbuf[i] *= 2;
}

// Collecting updated data back to process 0
MPI_Gather(recvbuf, 4, MPI_INT, recv_all, 4, MPI_INT, 0, MPI_COMM_WORLD);

// Printing  final array from process 0
if (world_rank == 0) {
// Displaying  final array header
printf("Final array on process 0:\n");
```

```
        for (int i = 0; i < 16; i++) {
        // Printing each element of  array
        printf("%d ", recv_all[i]);
        }


        printf("\n");
        }


        MPI_Finalize();


        return 0;
}
```

Output:

```
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpicc task2.c -o task2
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpirun --oversubscribe -np 4 ./task2
Process 2 received value 0: 9
Process 2 received value 1: 10
Process 2 received value 2: 11
Process 2 received value 3: 12
Process 0 received value 0: 1
Process 0 received value 1: 2
Process 1 received value 0: 5
Process 1 received value 1: 6
Process 1 received value 2: 7
Process 1 received value 3: 8
Process 3 received value 0: 13
Process 3 received value 1: 14
Process 3 received value 2: 15
Process 3 received value 3: 16
Process 0 received value 2: 3
Process 0 received value 3: 4
Final array on process 0:
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
amber@amber-HP-EliteBook-840-G3:~/Desktop$
```

Follow-up Questions:

a. What will happen if the number of processes is not evenly divisible by the data size?

Answer: If the number of processes does not evenly divide the data size (e.g., 16 integers with 5 processes), MPI_Scatter will fail or produce undefined behavior due to non-integer chunk sizes, and MPI_Gather will incorrectly collect data, leading to runtime errors or incomplete results.

Task 3: Distributed Reduction and All-Gather (Advanced)
Write a program using 6 processes where:
- Each process generates a random number between 1 and 100.
- Use MPI_Allgather to collect all numbers into an array on every process.
- Use MPI_Reduce to compute the maximum value among all numbers, with process 0 printing it.
- Then, use MPI_Allreduce to compute the average value and display it from all processes.

Code:

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


#include <time.h>

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

    int process_id, total_procs;
    int random_value;

    // Array to gather values from all processes (assuming 6 processes)
    int gathered_values[6];

    // Variables to store the maximum value (from reduce) and the sum (from allreduce)
    int highest_value, total_sum;

    // Variable to hold the calculated average value
    float avg_value;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &process_id);

    // total number of processes in the communicator
    MPI_Comm_size(MPI_COMM_WORLD, &total_procs);
```

```
	//  the random number generator with a different value for each process
	srand(time(NULL) + process_id);

	// Generating a random number between 1 and 100
	random_value = (rand() % 100) + 1;

	// Print the number generated by this process
	printf("Process %d generated: %d\n", process_id, random_value);

	// Collecting random values from all processes into gathered_values array on each
process
	MPI_Allgather(&random_value, 1, MPI_INT, gathered_values, 1, MPI_INT,
MPI_COMM_WORLD);

	// Find the maximum value among all processes using MPI_Reduce and store it at
process 0
	MPI_Reduce(&random_value, &highest_value, 1, MPI_INT, MPI_MAX, 0,
MPI_COMM_WORLD);

	// Compute the sum of all values across processes using MPI_Allreduce
	MPI_Allreduce(&random_value, &total_sum, 1, MPI_INT, MPI_SUM,
MPI_COMM_WORLD);

	// Calculate the average
	avg_value = total_sum / (float)total_procs;

	if (process_id == 0)
	printf("\nMaximum number: %d\n", highest_value);


	printf("Process %d: Average number: %.2f\n", process_id, avg_value);


	MPI_Finalize();

	return 0;
}
```

Output:

```
--------------------------------------------------------------------------
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpicc -o task3 task3.c
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpirun --oversubscribe -np 6 ./task3
Process 1 generated: 10
Process 2 generated: 10
Process 0 generated: 79
Process 4 generated: 62
Process 3 generated: 45
Process 5 generated: 92
Process 1: Average number: 49.67

Maximum number: 92
Process 0: Average number: 49.67
Process 3: Average number: 49.67
Process 5: Average number: 49.67
Process 4: Average number: 49.67
Process 2: Average number: 49.67
amber@amber-HP-EliteBook-840-G3:~/Desktop$
```

Enhancements:
- Time the MPI_Allgather and MPI_Reduce phases using MPI_Wtime.
- Print both raw timing and number of operations performed.

Code:

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

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

    int process_id, total_procs;
    int random_value;
    int gathered_values[6];
    int highest_value, total_sum;
    float avg_value;

    double start_time, end_time, duration_allgather, duration_reduce;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_id);
    MPI_Comm_size(MPI_COMM_WORLD, &total_procs);

    srand(time(NULL) + process_id);
    random_value = (rand() % 100) + 1;
```

```
    printf("Process %d generated: %d\n", process_id, random_value);

    // Timing MPI_Allgather
    start_time = MPI_Wtime();
    MPI_Allgather(&random_value, 1, MPI_INT, gathered_values, 1, MPI_INT,
MPI_COMM_WORLD);
    end_time = MPI_Wtime();
    duration_allgather = end_time - start_time;

    // Timing MPI_Reduce
    start_time = MPI_Wtime();
    MPI_Reduce(&random_value, &highest_value, 1, MPI_INT, MPI_MAX, 0,
MPI_COMM_WORLD);
    end_time = MPI_Wtime();
    duration_reduce = end_time - start_time;

    // Total sum using Allreduce
    MPI_Allreduce(&random_value, &total_sum, 1, MPI_INT, MPI_SUM,
MPI_COMM_WORLD);
    avg_value = total_sum / (float)total_procs;

    if (process_id == 0) {
    printf("\nSummary (Process 0) \n");
    printf("Maximum number: %d\n", highest_value);
    printf("MPI_Allgather: %.6f seconds (%d operations)\n", duration_allgather, total_procs);
    printf("MPI_Reduce   : %.6f seconds (%d operations)\n", duration_reduce, total_procs);
    printf("\n\n");
    }

    printf("Process %d: Average number: %.2f\n", process_id, avg_value);

    MPI_Finalize();
    return 0;
}
```

Output:

```
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpicc -o task3_b task3_b.c
amber@amber-HP-EliteBook-840-G3:~/Desktop$ mpirun --oversubscribe -np 6 ./task3_
b
Process 5 generated: 36
Process 2 generated: 96
Process 1 generated: 34
Process 3 generated: 67
Process 0 generated: 26
Process 4 generated: 42
Process 5: Average number: 50.17
Process 1: Average number: 50.17
Process 3: Average number: 50.17
Process 4: Average number: 50.17

Summary (Process 0)
Maximum number: 96
MPI_Allgather: 0.000262 seconds (6 operations)
MPI_Reduce   : 0.000066 seconds (6 operations)


Process 0: Average number: 50.17
Process 2: Average number: 50.17
amber@amber-HP-EliteBook-840-G3:~/Desktop$
```

Questions:
a. Why is MPI_Allgather more expensive than MPI_Gather?

Answer: Because MPI_Allgather sends data to all processes, while MPI_Gather sends data only to the root. This means more communication and synchronization in Allgather, making it more expensive.

b. Can MPI_Allreduce be used as a substitute for Reduce+Broadcast? Explain.

Answer: Yes. MPI_Allreduce combines the reduction and broadcast in one call, returning the result to all processes, making it more efficient than doing Reduce followed by Broadcast separately

c. What challenges arise if the data is large (e.g., arrays of size 1 million)?

Answer: With large data (e.g., 1 million elements), `MPI_Allgather` and `MPI_Allreduce` cause high memory usage on each process, increased communication time, and may lead to network congestion or buffer overflows if the system can't efficiently handle large messages.