



**Name:** Tazmeen Afroz

**Roll No:** 22P-9252

**Section:** BAI-6A

**Instructor:** Sir Ali Sayyed

**Course:** Parallel and Distributed Computing

**Assignment:** 3

**Campus:** FAST-NUCES Peshawar

**Date:** 13 May 2025

# Task 1: Basic Understanding of Broadcast

## Problem Statement

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.

## Implementation

```
1
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <mpi.h>
6
7 int main(int argc, char** argv){
8
9     int rank, size, data;
10    MPI_Init(&argc, &argv);
11    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12    MPI_Comm_size(MPI_COMM_WORLD, &size);
13
14    if(rank == 0){
15        data = 17;
16        printf("Process %d broadcasting data: %d\n", rank, data);
17    }
18
19
20    // Broadcast the data from process 0 to all other processes
21    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
22
23
24    printf("Process %d received data: %d\n", rank, data);
25
26    MPI_Finalize();
27
28    return 0;
29 }
```

```
(base) tazmeen@afroz:~/PDC/Assignment_03_codes$ mpicc Task1.c -o task1
(base) tazmeen@afroz:~/PDC/Assignment_03_codes$ mpirun -np 4 task1
Process 0 broadcasting data: 17
Process 0 received data: 17
Process 1 received data: 17
Process 2 received data: 17
Process 3 received data: 17
(base) tazmeen@afroz:~/PDC/Assignment_03_codes$
```

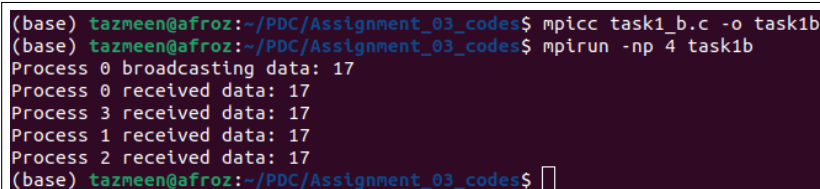
Figure 1: Output of Task 1 implementation

## Questions

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

When a non-root process changes the value before the broadcast, this modification will be overwritten by the value from the root process (Process 0). The MPI\_Bcast operation ensures that all processes receive the exact value sent by the root process, regardless of any local modifications.

```
1
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <mpi.h>
6
7 int main(int argc, char** argv){
8
9     int rank, size, data;
10    MPI_Init(&argc, &argv);
11    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12    MPI_Comm_size(MPI_COMM_WORLD, &size);
13
14    if(rank == 0){
15        data = 17;
16        printf("Process %d broadcasting data: %d\n", rank, data);
17    }
18    else{
19        data = 0;
20    }
21
22
23    // Broadcast the data from process 0 to all other processes
24    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
25
26
27    printf("Process %d received data: %d\n", rank, data);
28
29    MPI_Finalize();
30
31    return 0;
32 }
```

A terminal window showing the execution of an MPI program. The prompt is (base) tazmeen@afroz:~/PDC/Assignment\_03\_codes\$. The user runs mpicc task1\_b.c -o task1b, then mpirun -np 4 task1b. The output shows Process 0 broadcasting data: 17, and all four processes (0, 1, 2, 3) receiving data: 17. The prompt returns to (base) tazmeen@afroz:~/PDC/Assignment\_03\_codes\$.

```
(base) tazmeen@afroz:~/PDC/Assignment_03_codes$ mpicc task1_b.c -o task1b
(base) tazmeen@afroz:~/PDC/Assignment_03_codes$ mpirun -np 4 task1b
Process 0 broadcasting data: 17
Process 0 received data: 17
Process 3 received data: 17
Process 1 received data: 17
Process 2 received data: 17
(base) tazmeen@afroz:~/PDC/Assignment_03_codes$
```

Figure 2: Output demonstrating value overwriting during broadcast

Even though non-root processes initially set `data` to 0, the broadcast from Process 0 will replace this value with 17 across all processes.

## 2. What constraints must be followed when calling MPI\_Bcast?

When calling MPI\_Bcast, several critical constraints must be followed:

- **All processes in the communicator must call MPI\_Bcast** at the same time, in the same program execution flow. Mismatch in who calls it or when will hang the program.
- **The data type, buffer, count, and root must match across all processes.**
  - For example, if root uses MPI\_INT, everyone must use MPI\_INT.
  - Mismatched arguments cause errors or hangs.
- **Arguments must match:**
  - **buffer:** Pointer to data buffer
  - **count:** Number of elements to broadcast
  - **datatype:** MPI datatype of the elements
  - **root:** Rank of the process sending the data
  - **comm:** Communication communicator (usually MPI\_COMM\_WORLD)
- **All processes must participate, even the root.**
  - Root doesn't skip it, it uses the value it already has and joins the call.

## 3. How does MPI\_Bcast differ from point-to-point send/receive operations?

MPI\_Bcast is a collective communication operation that simultaneously sends data from one process (root) to all other processes in a single, efficient function call. In contrast, point-to-point operations (MPI\_Send/MPI\_Recv) require explicit, individual send and receive calls for each process. While point-to-point communications offer more flexibility in data transfer, MPI\_Bcast provides a more streamlined and optimized approach for distributing the same data to all processes, reducing programming complexity and potentially improving communication performance.

## Task 2: Data Scattering and Gathering (Intermediate)

### Problem Statement

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.

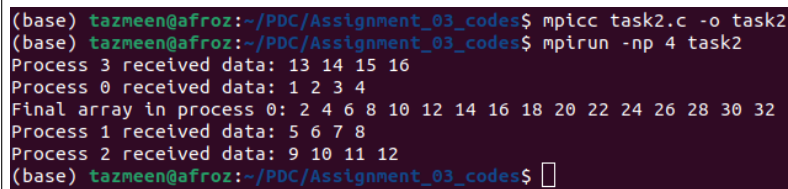
### Implementation

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4
5 int main(int argc, char **argv){
6     int rank, size;
7     int data[16];
8     int recv_data[4];
9     int gathered_data[16];
10    MPI_Init(&argc, &argv);
11    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12    MPI_Comm_size(MPI_COMM_WORLD, &size);
13
14    if(rank == 0){
15        // Initialize the array
16        for(int i = 0; i < 16; i++){
17            data[i] = i + 1;
18        }
19    }
20
21    // Scatter the data to all processes
22    MPI_Scatter(data, 4, MPI_INT, recv_data, 4, MPI_INT, 0, MPI_COMM_WORLD);
23
24    // output the received data
25    printf("Process %d received data: ", rank);
26    for(int i = 0; i < 4; i++){
27        printf("%d ", recv_data[i]);
28    }
29    printf("\n");
30    // Each process multiplies its chunk by 2
31    for(int i = 0; i < 4; i++){
32        recv_data[i] *= 2;
33    }
34    // Gather the updated data back to process 0
35    MPI_Gather(recv_data, 4, MPI_INT, gathered_data, 4, MPI_INT, 0, MPI_COMM_WORLD);
36    // Process 0 prints the final array
37    if(rank == 0){
38        printf("Final array in process 0: ");
```

```

38     for(int i = 0; i < 16; i++){
39         printf("%d ", gathered_data[i]);
40     }
41     printf("\n");
42 }
43 MPI_Finalize();
44 return 0;
45
46
47
48 }

```



```

(base) tazmeen@afroz:~/PDC/Assignment_03_codes$ mpicc task2.c -o task2
(base) tazmeen@afroz:~/PDC/Assignment_03_codes$ mpirun -np 4 task2
Process 3 received data: 13 14 15 16
Process 0 received data: 1 2 3 4
Final array in process 0: 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
Process 1 received data: 5 6 7 8
Process 2 received data: 9 10 11 12
(base) tazmeen@afroz:~/PDC/Assignment_03_codes$ 

```

Figure 3: Output of Task 2 implementation

## Questions

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

If the number of processes is not evenly divisible by the data size, the program will not work as expected. Some processes may not receive any data, while others may receive more than their share. This can lead to incorrect results and potential errors in the program.

b. Modified program to handle uneven distribution using dynamic offsets

```

1
2 #include <stdio.h>
3 #include <mpi.h>
4
5 int main(int argc, char **argv) {
6     int rank, size;
7     int data[17]; // uneven array
8     int recv_data[5]; // Maximum any process will receive is 5 in this
    case
9     int counts[4]; // Number of elements each process gets
10    int displs[4]; // Starting index in data[] for each process
11
12    MPI_Init(&argc, &argv);
13    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14    MPI_Comm_size(MPI_COMM_WORLD, &size);
15
16    int total_elements = 17;
17    int base = total_elements / size; // 17 / 4 = 4
18    int rem = total_elements % size; // 17 % 4 = 1

```

```

19
20 // Calculate counts for each process
21 for (int i = 0; i < size; i++) {
22     if (i < rem) {
23         counts[i] = base + 1; // Process 0 gets 5 elements as 0 < 1
24     } else {
25         counts[i] = base;      // Other processes get 4 elements each
26     }
27 }
28
29 //
30 displs[0] = 0;
31 for (int i = 1; i < size; i++) {
32     displs[i] = displs[i - 1] + counts[i - 1]; // Calculate starting
index for each process //dis[1] = 0 +5 = 5 , dis[2] = 5 + 4 = 9, dis[3]
= 9 + 4 = 13
33 }
34
35 // Initialize data in rank 0
36 if (rank == 0) {
37     for (int i = 0; i < total_elements; i++) {
38         data[i] = i + 1;
39     }
40
41     printf("Original data in rank 0: ");
42     for (int i = 0; i < total_elements; i++) {
43         printf("%d ", data[i]);
44     }
45     printf("\n");
46 }
47
48 // Scatterv is used to scatter uneven data
49 MPI_Scatterv(data, counts, displs, MPI_INT, recv_data, counts[rank],
MPI_INT, 0, MPI_COMM_WORLD);
50
51 // Print what each process received
52 printf("Process %d received: ", rank);
53 for (int i = 0; i < counts[rank]; i++) {
54     printf("%d ", recv_data[i]);
55 }
56 printf("\n");
57
58 // Multiply received data by 2
59 for (int i = 0; i < counts[rank]; i++) {
60     recv_data[i] *= 2;
61 }
62
63 // Print modified data in each process
64 printf("Process %d after multiplication: ", rank);
65 for (int i = 0; i < counts[rank]; i++) {
66     printf("%d ", recv_data[i]);
67 }
68 printf("\n");
69

```

```

70 // Gather all data back to rank 0
71 MPI_Gatherv(recv_data, counts[rank], MPI_INT, data, counts, displs,
MPI_INT, 0, MPI_COMM_WORLD);
72
73 // Print final result in rank 0
74 if (rank == 0) {
75     printf("Final array in process 0: ");
76     for (int i = 0; i < total_elements; i++) {
77         printf("%d ", data[i]);
78     }
79     printf("\n");
80 }
81
82 MPI_Finalize();
83 return 0;
84 }

```

```

(base) tazmeen@afroz:~/PDC/Assignment_03_codes$ mpicc task2b.c -o task2b
(base) tazmeen@afroz:~/PDC/Assignment_03_codes$ mpirun -np 4 task2b
Original data in rank 0: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Process 0 received: 1 2 3 4 5
Process 0 after multiplication: 2 4 6 8 10
Process 1 received: 6 7 8 9
Process 1 after multiplication: 12 14 16 18
Process 2 received: 10 11 12 13
Process 2 after multiplication: 20 22 24 26
Process 3 received: 14 15 16 17
Process 3 after multiplication: 28 30 32 34
Final array in process 0: 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34
(base) tazmeen@afroz:~/PDC/Assignment_03_codes$ 

```

Figure 4: Output of Task 2 modified implementation



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

### Problem Statement

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.

### Implementation

```
1 #include <stdio.h>
2 #include <mpi.h>
3 #include <stdlib.h>
4
5 int main(int argc, char **argv) {
6     int rank, size, random_number, gathered_numbers[6], maximum, sum,
    average;
7     double start_time, end_time, allgather_time, reduce_time,
    allreduce_time;
8     int count = 0;
9
10
11     MPI_Init(&argc, &argv);
12     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13     MPI_Comm_size(MPI_COMM_WORLD, &size);
14
15     if (size != 6) {
16         if (rank == 0)
17             printf("This program requires exactly 6 processes.\n");
18         MPI_Finalize();
19         return 1;
20     }
21
22     // Generate a random number between 1 and 100
23     srand(rank + 1); // Different seed for each process
24     random_number = rand() % 100 + 1;
25     printf("Process %d generated number: %d\n", rank, random_number);
26
27     // Timing MPI_Allgather
28     start_time = MPI_Wtime();
29     // all gather the random numbers and store in gathered_numbers , all
    processes will have the same gathered_numbers
30     MPI_Allgather(&random_number, 1, MPI_INT, gathered_numbers, 1, MPI_INT
    , MPI_COMM_WORLD);
31     end_time = MPI_Wtime();
32     allgather_time = end_time - start_time;
```

```

33
34 // Print gathered numbers
35 printf("Process %d gathered numbers: ", rank);
36 for (int i = 0; i < size; i++) {
37     printf("%d ", gathered_numbers[i]);
38 }
39 printf("\n");
40
41
42 start_time = MPI_Wtime();
43 // find the max no from all processes and store in maximum
44 MPI_Reduce(&random_number, &maximum, 1, MPI_INT, MPI_MAX, 0,
MPI_COMM_WORLD);
45 end_time = MPI_Wtime();
46 reduce_time = end_time - start_time;
47
48 // Print maximum from process 0
49 if (rank == 0) {
50     printf("Maximum number is: %d\n", maximum);
51 }
52
53 // Timing MPI_Allreduce to calculate sum and then compute average
54 start_time = MPI_Wtime();
55 MPI_Allreduce(&random_number, &sum, 1, MPI_INT, MPI_SUM,
MPI_COMM_WORLD);
56 end_time = MPI_Wtime();
57 allreduce_time = end_time - start_time;
58
59 average = sum / size;
60
61 // Output average and timings
62 printf("Process %d average is: %d\n", rank, average);
63 printf("Process %d Allgather time: %.6f seconds\n", rank,
allgather_time);
64 printf("Process %d Reduce time: %.6f seconds\n", rank, reduce_time);
65 printf("Process %d Allreduce time: %.6f seconds\n", rank,
allreduce_time);
66
67 // Count of operations done
68 printf("Process %d performed %d operations\n", rank, count);
69
70 MPI_Finalize();
71 return 0;
72 }

```

## Questions

### a. Why is MPI\_Allgather more expensive than MPI\_Gather?

MPI\_Allgather is more expensive than MPI\_Gather because:

- MPI\_Gather collects data from all processes to a single root process. MPI\_Allgather collects data from all processes and distributes the complete result to all processes. This requires additional communication steps (essentially a gather followed by a broad-

```

(base) tazmeen@afroz:~/PDC/Assignment_03_codes$ mpicc task3.c -o task3
(base) tazmeen@afroz:~/PDC/Assignment_03_codes$ mpirun -np 6 task3
Process 1 generated number: 91
Process 2 generated number: 47
Process 3 generated number: 2
Process 5 generated number: 42
Process 0 generated number: 84
Process 4 generated number: 76
Process 1 gathered numbers: 84 91 47 2 76 42
Process 2 gathered numbers: 84 91 47 2 76 42
Process 0 gathered numbers: 84 91 47 2 76 42
Maximum number is: 91
Process 3 gathered numbers: 84 91 47 2 76 42
Process 4 gathered numbers: 84 91 47 2 76 42
Process 5 gathered numbers: 84 91 47 2 76 42
Process 0 average is: 57
Process 0 Allgather time: 0.000179 seconds
Process 0 Reduce time: 0.000020 seconds
Process 0 Allreduce time: 0.000034 seconds
Process 5 average is: 57
Process 5 Allgather time: 0.000202 seconds
Process 5 Reduce time: 0.000007 seconds
Process 5 Allreduce time: 0.000045 seconds
Process 5 performed 0 operations
Process 4 average is: 57
Process 4 Allgather time: 0.000105 seconds
Process 4 Reduce time: 0.000004 seconds
Process 4 Allreduce time: 0.000039 seconds
Process 4 performed 0 operations
Process 3 average is: 57
Process 3 Allgather time: 0.000209 seconds
Process 3 Reduce time: 0.000005 seconds
Process 3 Allreduce time: 0.000040 seconds
Process 3 performed 0 operations
Process 1 average is: 57
Process 1 Allgather time: 0.000201 seconds
Process 1 Reduce time: 0.000022 seconds
Process 1 Allreduce time: 0.000045 seconds
Process 1 performed 0 operations
Process 2 average is: 57
Process 2 Allgather time: 0.000194 seconds
Process 2 Reduce time: 0.000022 seconds
Process 2 Allreduce time: 0.000030 seconds
Process 2 performed 0 operations
Process 0 performed 0 operations

```

Figure 5: Output of Task 3 implementation

cast). The communication complexity increases from  $O(n)$  to  $O(n \log n)$  in optimal implementations.

**b. Can MPI\_Allreduce be used as a substitute for Reduce+Broadcast? Explain.**

Yes, MPI\_Allreduce can substitute for Reduce+Broadcast:

- MPI\_Allreduce performs a reduction operation and distributes the result to all processes. This is equivalent to MPI\_Reduce followed by MPI\_Bcast. MPI\_Allreduce is typically more efficient as it can optimize the communication pattern. It reduces latency by avoiding the sequential nature of doing reduce then broadcast.

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

Challenges with large data (arrays of size 1 million):

- Memory constraints: Each process needs enough memory to store the complete result

- Network bandwidth becomes a bottleneck for collective operations
- Communication time dominates computation time
- Load imbalance may occur if processes don't complete local computations simultaneously