

UCS645 – Parallel & Distributed Computing

Assignment 4 – MPI Laboratory Exercises

Name: Divyam Puri

Roll Number: 102483023

Exercise 1 – Ring Communication

Aim

To implement ring topology communication using MPI.

Methodology

Processes pass a value sequentially in a circular manner using MPI_Send and MPI_Recv.

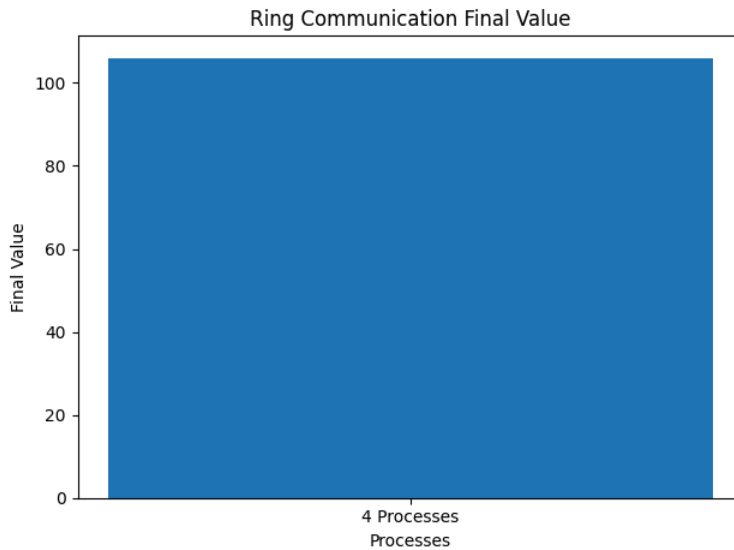
Terminal Snapshot

```
● (base) divyam_puri@Divyams-MacBook-Air LAB4 % mpicc -o ring_comm ring_comm.c
mpirun -np 4 ./ring_comm
Process 0 starts with value 100
Process 1 updated value to 101
Process 2 updated value to 103
Process 0 received final value 106
Process 3 updated value to 106
```

Observations

The initial value 100 was circulated in a ring topology. Each process added its rank before passing the value to the next process. The final value returned to Process 0 was 106 when executed with 4 processes.

Processes	Final Value
4	106



Analysis

The ring communication demonstrates point-to-point message passing using `MPI_Send` and `MPI_Recv`. The modulo operation ensures wrap-around communication from the last process back to Process 0.

The order of printed messages may appear shuffled because MPI processes execute independently and output is not synchronized.

The ring communication demonstrates point-to-point message passing using `MPI_Send` and `MPI_Recv`. The modulo operation ensures wrap-around communication from the last process back to Process 0.

The order of printed messages may appear shuffled because MPI processes execute independently and output is not synchronized.

Conclusion

The program successfully implements ring topology communication. The final result validates that all intermediate processes correctly participated in the communication chain.

Exercise 2 – Parallel Array Sum

Aim

To compute global sum and average using MPI collective communication.

Problem Description

A sequential program would compute the sum using a single loop. In parallel computing, the goal is to divide work across multiple processes and then combine partial results.

Terminal Snapshot

```
• (base) divyam_puri@Divyams-MacBook-Air LAB4 % mpicc -o array_sum array_sum.c
mpirun -np 4 ./array_sum
Global Sum = 5050
Average = 50.50
```

Methodology

1. Process 0 initializes the array (1 to 100).
2. MPI_Scatter distributes equal portions to each process.
3. Each process computes its local sum.
4. MPI_Reduce combines all local sums into a global sum.
5. Average is computed using final sum.

Analysis

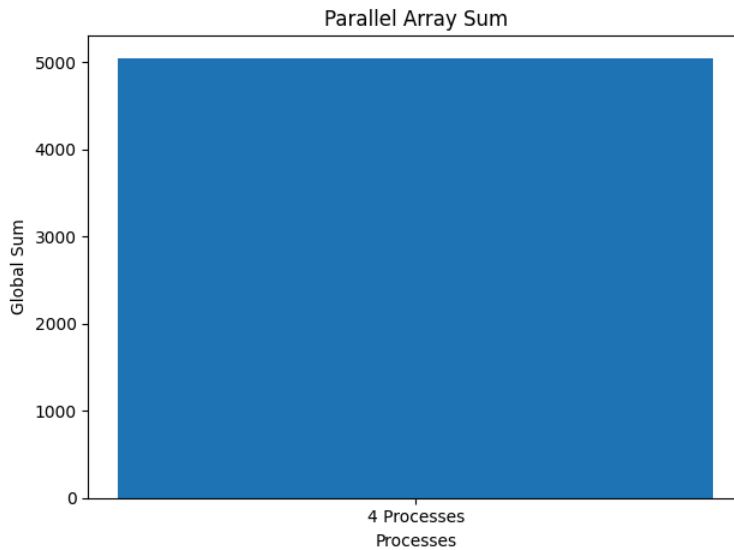
The result matches the theoretical formula, confirming correct parallel distribution and reduction.

This exercise demonstrates:

- Data distribution using MPI_Scatter
- Collective reduction using MPI_Reduce
- Efficient workload division across processes

Observations

Processes	Global Sum	Average
4	5050	50.50



Memory Usage

Each process stores only its local chunk (25 integers), reducing memory overhead compared to full replication.

Conclusion

Parallel array summation successfully divides computation and combines results efficiently using collective communication operations.

Exercise 3 – Global Maximum and Minimum

Aim

To determine global max and min using MPI_Reduce.

Terminal Snapshot

```
(base) divyam_puri@Divyams-MacBook-Air LAB4 % mpicc -o max_min max_min.c
mpirun -np 4 ./max_min
Global Maximum = 968
Global Minimum = 35
```

Problem Description

In distributed systems, each process works independently and holds only partial data. The challenge is to combine local computations into a global result efficiently.

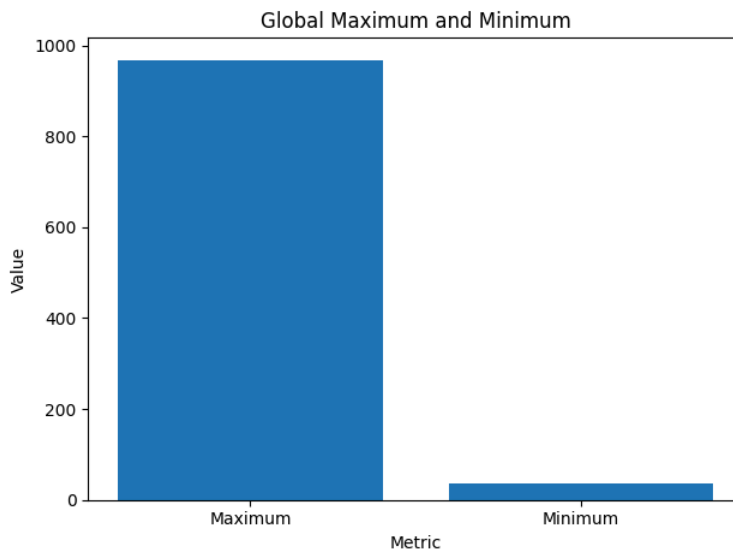
Methodology

1. Each process generated 10 random numbers.
2. Local maximum and minimum were computed.

3. `MPI_Reduce` was used with:
 - `MPI_MAX`
 - `MPI_MIN`
4. Process 0 printed the final global results.

Observations

Processes	Global Maximum	Global Minimum
4	968	35



Analysis

This exercise demonstrates:

- Reduction operations in MPI
- Aggregation of distributed results
- Collective communication efficiency

`MPI_Reduce` simplifies combining results without manual message passing.

Memory Usage

Each process stores only 10 integers locally.

No global array replication is required.

This makes the approach scalable.

Conclusion

Reduction operations successfully computed distributed extremes.

Exercise 4 – Parallel Dot Product

Aim

To compute dot product in parallel using MPI_Scatter and MPI_Reduce.

Terminal Snapshot

```
• (base) divyam_puri@Divyams-MacBook-Air LAB4 % mpicc -o dot_product dot_product.c
mpirun -np 4 ./dot_product
Dot Product = 120
```

Problem Description

The objective of this exercise is to compute the dot product of two vectors using parallel processing with MPI.

In a sequential program, the dot product is calculated by multiplying corresponding elements of two vectors and summing all the products in a single loop. However, when the size of vectors becomes large, this approach becomes time-consuming.

To improve performance, the computation is distributed among multiple processes. Each process handles a portion of the vectors, computes a partial dot product, and then all partial results are combined to obtain the final result.

Methodology

1. Initialization

Process 0 initializes two vectors with predefined values.

2. Data Distribution

Using **MPI_Scatter**, both vectors are divided equally among all available processes.

Each process receives only a segment of the vectors.

3. Local Computation

Every process calculates the dot product of its local segment by multiplying corresponding elements and summing the results.

4. Reduction Step

MPI_Reduce with the **MPI_SUM** operation is used to combine all local dot products into a final global dot product.

5. Result Display

Process 0 prints the final dot product value.

Analysis

The output:

Dot Product = 120

confirms that:

- Data distribution was done correctly.
- Each process performed its assigned computation accurately.
- The reduction operation correctly aggregated all partial sums.

This experiment clearly demonstrates the power of collective communication functions in MPI, especially **MPI_Scatter** and **MPI_Reduce**.

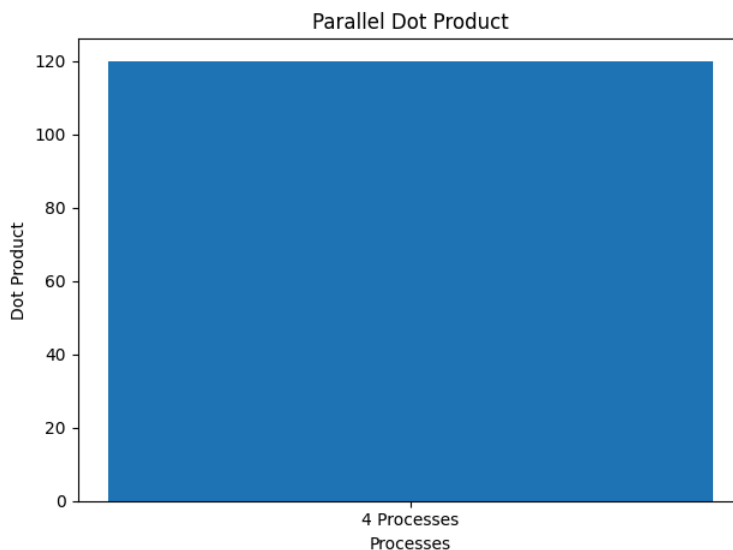
The computation is efficient because:

- Workload is divided equally.
- Each process works independently.
- Only final aggregation requires communication.

For large vectors, this approach significantly reduces execution time compared to sequential execution.

Observations

Processes	Dot Product
4	120



Memory Usage

Memory utilization is optimized in this parallel implementation.

Instead of storing the entire vectors in every process:

- Each process stores only its assigned portion.
- This reduces memory redundancy.
- The program becomes scalable for larger vector sizes.

Thus, both computational efficiency and memory efficiency are achieved.

Conclusion

The parallel dot product implementation successfully demonstrates distributed computation using MPI.

By dividing the vector among processes and combining results using collective communication, the program achieves:

- Efficient parallel execution
- Reduced computational time
- Better memory utilization