Lab 1 Report

Message Passing Interface (MPI)

Name: Ahmad Bin Rashid

Roll No: 2023-CS-53

Instructor: Sir Waqas Ali

Course: Distributed and Parallel Computing

University of Engineering and Technology

February 24, 2026

1 Introduction

The objective of this lab is to understand distributed-memory parallel programming using the Message Passing Interface (MPI). The lab explores process-based parallelism, inter-process communication, and performance scalability using multiple MPI programs.

2 Tools and Environment

- Programming Language: C
- Parallel Framework: MPI (OpenMPI)
- Compiler: `mpicc`
- Execution Tool: `mpirun`
- Operating System: Linux

3 Programs Implemented

- Hello World using MPI
- Ping-Pong Communication Benchmark
- Parallel Counting using MPI
- Parallel Computation of π using MPI

4 Hello World (MPI)

Each MPI process prints its rank and the total number of processes. This program demonstrates basic MPI initialization, rank identification, and parallel execution.

```
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 4 hello_mpi
Hello from process 1 of 4
Hello from process 2 of 4
Hello from process 3 of 4
Hello from process 0 of 4

real    0m0.722s
user    0m0.233s
sys     0m0.375s
```

Figure 1: MPI Hello World Execution

5 Ping-Pong Communication

The ping-pong program measures message latency between two MPI processes by repeatedly sending and receiving messages.

Observed Results:

- Total Time: 0.846861 seconds
- Average Round-Trip Time: 84.686075 microseconds

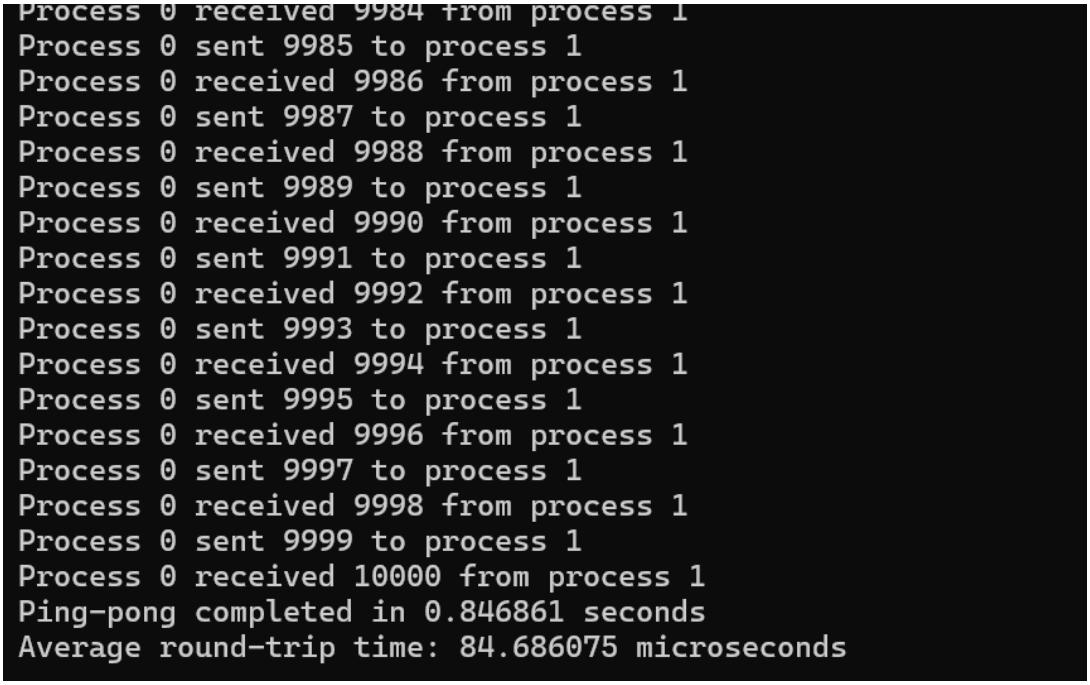

Figure 2: Ping-Pong Communication Results

6 Parallel Counting using MPI

The workload is divided among MPI processes, and each process computes a partial count. The final result is obtained using reduction.

6.1 Execution Screenshots

```
ahmad@Pakiman92:~/study/pdc lab/lab1$ mpicc -o count_mpi count_mpi.c
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 1 count_mpi
Total count = 1001084
Time with 1 processes = 0.326666 seconds

real    0m4.923s
user    0m3.548s
sys     0m1.126s
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 2 count_mpi
Total count = 999538
Time with 2 processes = 0.130052 seconds

real    0m2.293s
user    0m2.731s
sys     0m0.886s
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 3 count_mpi
Total count = 1001424
Time with 3 processes = 0.107822 seconds

real    0m2.065s
user    0m3.365s
sys     0m1.243s
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 4 count_mpi
Total count = 998284
Time with 4 processes = 0.092276 seconds

real    0m1.710s
user    0m3.712s
sys     0m1.174s
```

Figure 3: Count Program Execution Results

```
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 5 --oversubscribe count_mpi
Total count = 1000040
Time with 5 processes = 0.082485 seconds

real    0m1.895s
user    0m4.311s
sys     0m2.115s
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 6 --oversubscribe count_mpi
Total count = 1001301
Time with 6 processes = 0.074176 seconds

real    0m1.549s
user    0m4.515s
sys     0m1.914s
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 8 --oversubscribe count_mpi
Total count = 999049
Time with 8 processes = 0.095561 seconds

real    0m1.729s
user    0m6.176s
sys     0m2.640s
```

Figure 4: Count Program Execution Results

6.2 Execution Time

Processes	Execution Time (s)
1	0.326666
2	0.130052
3	0.107822
4	0.092276
5	0.082485
6	0.074176
8	0.095561

Table 1: MPI Counting Execution Time

6.3 Speedup Graph

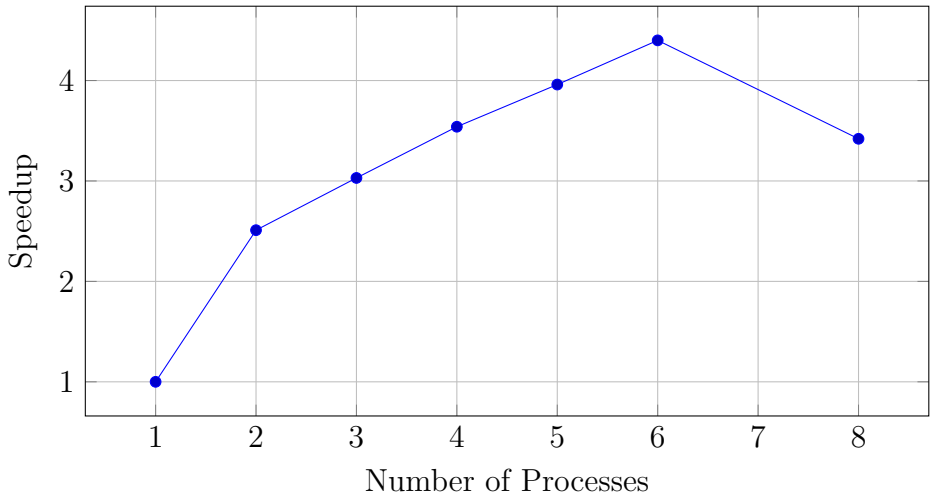

Figure 5: Speedup for MPI Counting Program

7 Parallel Computation of π

The value of π is approximated using numerical integration. The computation is distributed across MPI processes.

7.1 Execution Screenshots

```
ahmad@Pakiman92:~/study/pdc lab/lab1$ mpicc -o pi_mpi pi_mpi.c
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 1 pi_mpi
Pi = 3.1415926535899708
Time with 1 processes = 5.675158 seconds

real    0m6.745s
user    0m5.820s
sys     0m0.097s
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 2 pi_mpi
Pi = 3.1415926535899010
Time with 2 processes = 2.731667 seconds

real    0m3.582s
user    0m5.779s
sys     0m0.358s
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 3 pi_mpi
Pi = 3.1415926535899614
Time with 3 processes = 2.547047 seconds

real    0m3.262s
user    0m7.880s
sys     0m0.319s
```

Figure 6: Pi Program Execution Results

```
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 4 pi_mpi
Pi = 3.1415926535898206
Time with 4 processes = 2.072882 seconds

real    0m2.672s
user    0m8.442s
sys     0m0.400s
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 5 --oversubscribe pi_mpi
Pi = 3.1415926535895959
Time with 5 processes = 2.008539 seconds

real    0m2.953s
user    0m10.451s
sys     0m1.063s
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 6 --oversubscribe pi_mpi
Pi = 3.1415926535896825
Time with 6 processes = 1.850164 seconds

real    0m2.608s
user    0m11.251s
sys     0m1.053s
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 8 --oversubscribe pi_mpi
Pi = 3.1415926535897691
Time with 8 processes = 1.715379 seconds

real    0m2.619s
user    0m14.221s
sys     0m1.602s
ahmad@Pakiman92:~/study/pdc lab/lab1$ time mpirun -np 10 --oversubscribe pi_mpi
Pi = 3.1415926535897931
Time with 10 processes = 2.583654 seconds

real    0m3.489s
user    0m19.804s
sys     0m3.218s
```

Figure 7: Pi Program Execution Results

7.2 Execution Time

Processes	Execution Time (s)
1	5.675158
2	2.731667
3	2.547047
4	2.072882
5	2.008539
6	1.850164
8	1.715379
10	2.583654

Table 2: MPI π Computation Execution Time

7.3 Speedup Graph

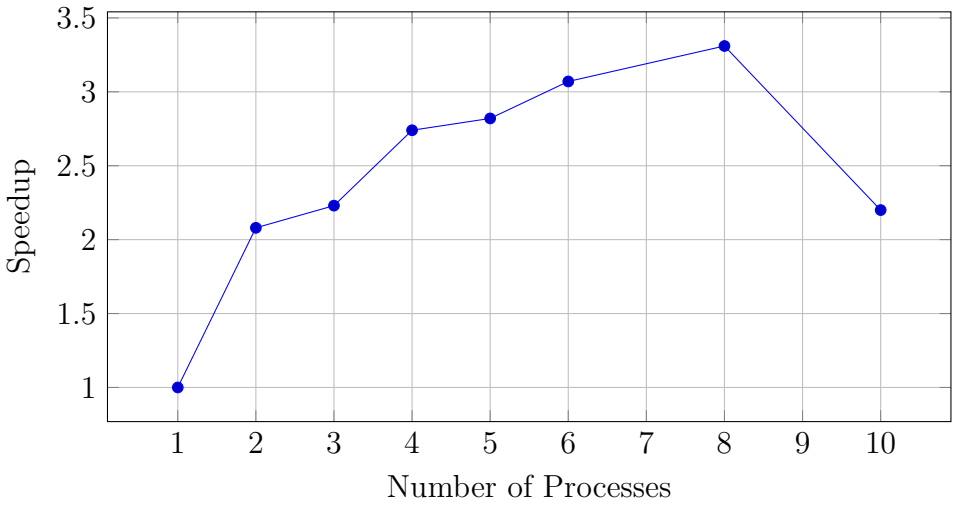

Figure 8: Speedup for MPI π Computation

8 Comparison: Pthreads vs MPI Counting Program

This section compares the Pthreads and MPI implementations of the counting program based on their measured execution times on the same machine.

8.1 Execution Time Comparison

Workers	Pthreads Time (s)	MPI Time (s)
1	0.13	0.326666
2	0.13	0.130052
3	–	0.107822
4	0.58	0.092276
5	–	0.082485
6	0.25	0.074176
8	0.24	0.095561

Table 3: Execution Time Comparison of Pthreads and MPI Counting Programs

8.2 Discussion

For a single worker, the Pthreads version performs significantly faster than the MPI version due to lower overhead and direct access to shared memory. As the number of workers increases, the MPI implementation shows better scalability and consistently lower execution times, especially from 3 to 6 processes.

The Pthreads implementation exhibits irregular performance, particularly at 4 threads, where contention and synchronization overhead dominate. MPI avoids shared-memory contention by using independent processes, resulting in more stable scaling.

At higher worker counts, oversubscription causes MPI performance to degrade, as seen at 8 processes. Overall, Pthreads are more efficient for small-scale shared-memory parallelism, while MPI provides better scalability and predictability for larger parallel configurations.

9 Self-Assessment Questions

1. **What are the main limitations of shared-memory parallelism that motivate the use of distributed memory?**
   Shared-memory systems are limited by memory bandwidth, cache coherence overhead, and contention for shared resources. They also scale poorly beyond a single machine. Distributed-memory systems avoid these bottlenecks by using private memory per process and explicit communication.

2. **Explain the SPMD model and how it is used in MPI.**
   SPMD stands for *Single Program, Multiple Data*. In MPI, all processes execute the same program but operate on different portions of the data. Each process behaves differently based on its rank, enabling parallel execution without separate programs.

3. **Why does each MPI process generate its own random data in the counting program? What are the trade-offs?**
   Having each process generate its own data avoids the overhead of distributing large arrays from a single process. This improves scalability and reduces communication cost. However, it may increase total memory usage and reduce reproducibility.

4. **What would happen if `MPI_Bcast` for `n` were omitted in the π program?**
   Without `MPI_Bcast`, different processes may use uninitialized or inconsistent values of `n`. This would lead to incorrect or undefined results. The broadcast ensures all processes perform the same amount of computation.

5. **Why is MPI_Send/MPI_Recv much slower than updating a shared variable in Pthreads?**
   MPI communication involves data copying, buffering, and protocol overhead. In contrast, updating a shared variable in Pthreads is a simple memory operation protected by a lock. The difference arises due to inter-process communication versus shared memory.

6. **What do you observe when running ping-pong with different message sizes?**
   For small messages, latency dominates and round-trip time increases slowly. As message size grows, bandwidth becomes the limiting factor and time increases linearly. This shows the transition from latency-bound to bandwidth-bound communication.

7. **According to Amdahl's Law, what is the maximum speedup with 95% parallelism on 8 processors?**
   The theoretical speedup is approximately $1/(0.05 + 0.95/8) \approx 5.93$. The measured speedup is lower due to communication overhead and load imbalance. This discrepancy highlights practical limits of parallel execution.

8. **Why might MPI show lower efficiency than Pthreads on a single machine?**
   MPI incurs overhead from message passing, data copying, and process management. Pthreads use shared memory and lightweight synchronization. On a single node, MPI communication costs are higher than thread-based access.

9. **Compare MPI_Scatter with local data generation in the counting program.**
   Using `MPI_Scatter` centralizes data generation but increases communication overhead. Local data generation improves scalability and reduces memory traffic. However, scattering provides better control over data consistency.

10. **Which programming model would you use on a 100-node cluster and why?**
    MPI is required to scale across multiple nodes in a cluster. Pthreads can be combined with MPI using a hybrid model where MPI handles inter-node communication and Pthreads exploit intra-node parallelism. This approach maximizes performance and resource utilization.

10 Reflection

Moving from shared-memory programming using Pthreads to distributed-memory programming using MPI was both challenging and insightful. In shared-memory systems, communication is implicit, which makes programming simpler but introduces risks such

as race conditions and deadlocks. MPI requires explicit communication, which initially feels more complex but results in cleaner data ownership and better isolation.

The most surprising aspect of MPI was how eliminating shared memory removed the need for locks and mutexes entirely. Instead, correctness depends on proper message ordering and collective operations. The most difficult part was understanding how to divide work evenly among processes and correctly combine partial results.

Debugging MPI programs was also more challenging due to multiple processes running independently. However, MPI programs scaled better and behaved more predictably under load. Overall, this lab improved my understanding of distributed systems and showed why MPI is widely used in high-performance computing environments.

11 Analysis and Discussion

Speedup improves with an increasing number of processes, but does not scale linearly due to communication overhead, synchronization costs, and limited hardware resources. Oversubscription results in diminishing returns and performance degradation beyond a certain point.

The π computation shows better scalability than the counting program because it has a higher computation-to-communication ratio.

12 Conclusion

This lab demonstrated the fundamentals of distributed-memory parallel programming using MPI. MPI allows true parallel execution across processes and scales better than threading for CPU-bound workloads. However, efficient parallelization requires careful consideration of communication overhead and process placement.