TP5: Introduction to MPI Programming

Parallel and Distributed Programming

Ikram Benfellah

February 25, 2026

1 Introduction

This lab covers MPI basics through five exercises: Hello World, data sharing, ring communication, matrix-vector multiplication, and Pi calculation.

2 Exercise 1: Hello World

We implemented a basic MPI program where each process prints its rank and total process count. Only rank 0 prints a special message.

Key functions: `MPI_Init`, `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_Finalize`

2.1 What happens if MPI_Finalize is omitted?

- Resource leaks (memory, buffers)
- Processes may hang indefinitely
- MPI runtime cannot cleanup properly
- Program may not terminate cleanly

3 Exercise 2: Sharing Data

Rank 0 reads integers from terminal and broadcasts them to all processes using `MPI_Bcast`. Loop continues until negative input.

Observation: All processes receive the same value simultaneously via collective communication.

4 Exercise 3: Ring Communication

Data passes through all processes in a ring: each adds its rank and forwards to the next. Process 0 receives the final result.

Final value: $\text{input} + \sum_{i=0}^{n-1} i = \text{input} + \frac{n(n-1)}{2}$

Neighbor calculation:

- Next: $(rank + 1) \mod size$

- Previous: $(rank - 1 + size) \mod size$

5 Exercise 4: Matrix-Vector Multiplication

Parallel matrix-vector product using row distribution. Used `MPI_Scatterv` to distribute rows, `MPI_Bcast` for vector, and `MPI_Gatherv` to collect results.

Non-divisible handling: First $(N \mod P)$ processes get one extra row.

5.1 Results

P	Serial (s)	Parallel (s)	Speedup	Efficiency
1	0.000662	0.005208	0.13	12.7%
2	0.000514	0.004760	0.11	5.4%
4	0.000586	0.004928	0.12	3.0%
8	0.000555	0.009738	0.06	0.7%

Table 1: Matrix-Vector Performance (1000×1000)

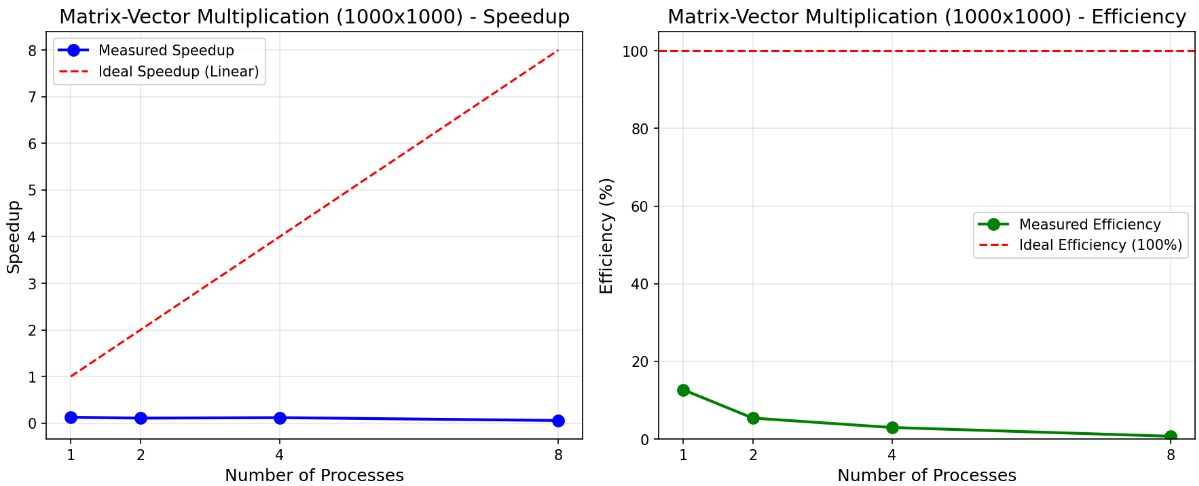

Figure 1: Matrix-Vector Speedup and Efficiency

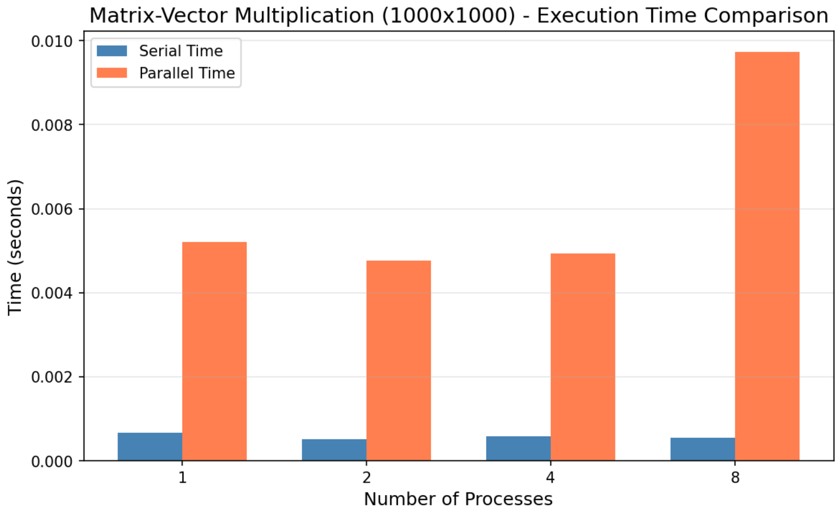

Figure 2: Matrix-Vector Execution Times

5.2 Observations

- Speedup < 1: parallel is slower than serial
- Communication overhead dominates for this small matrix size
- Efficiency drops as processes increase (more overhead, less work per process)
- Would need larger matrices to see benefits

6 Exercise 5: Pi Calculation

Computed π using numerical integration:

$$\pi = \frac{4}{N} \sum_{i=0}^{N-1} \frac{1}{1 + x_i^2}, \quad x_i = \frac{i + 0.5}{N}$$

Each process computes partial sum, then `MPI_Reduce` combines them.

6.1 Results

P	Serial (s)	Parallel (s)	Speedup	Efficiency
1	1.434	1.411	1.02	101.7%
2	1.399	0.701	2.00	99.8%
4	1.410	0.345	4.08	102.1%
8	3.894	0.500	7.79	97.3%

Table 2: Pi Calculation Performance ($N = 10^9$)

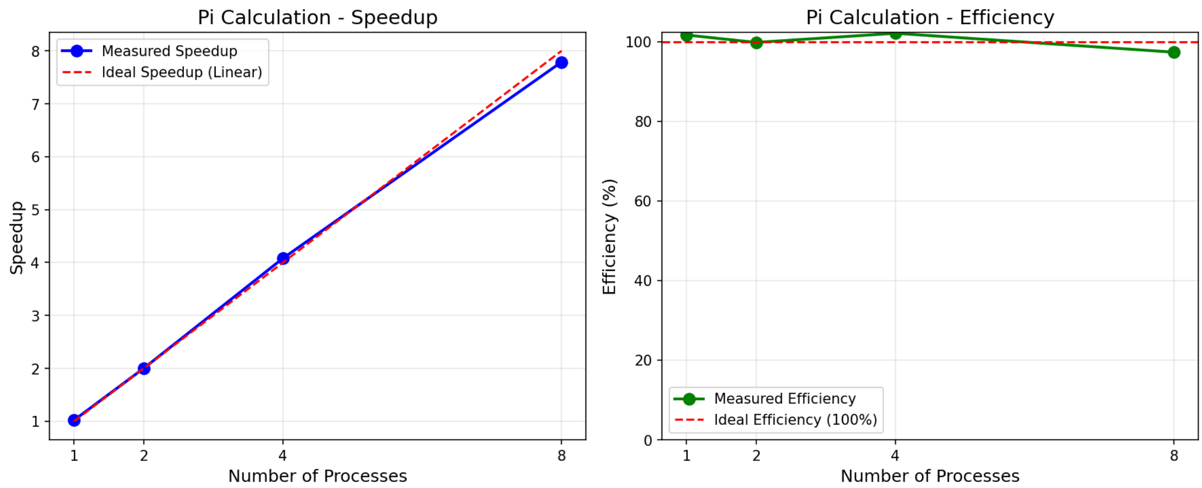

Figure 3: Pi Calculation Speedup and Efficiency

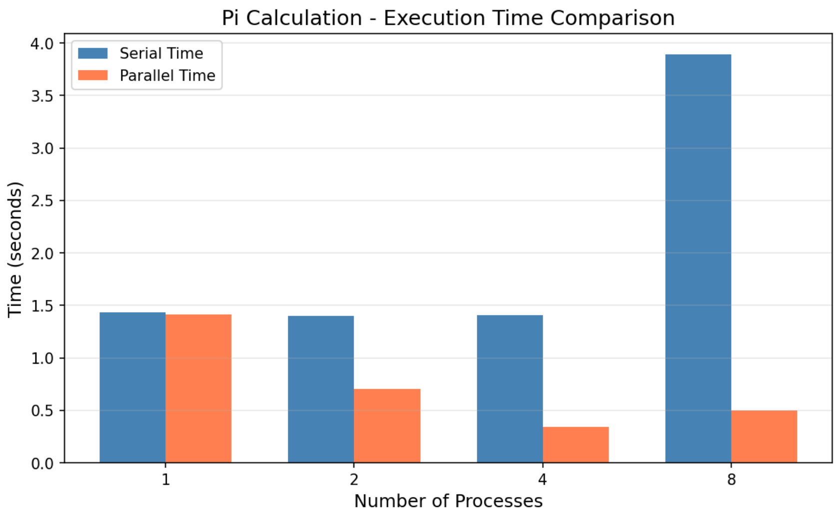

Figure 4: Pi Calculation Execution Times

6.2 Observations

- Near-linear speedup achieved (efficiency $\approx 100\%$)
- Only one `MPI_Reduce` call $\rightarrow$ minimal communication
- Embarrassingly parallel: no data dependencies between iterations
- Stark contrast with matrix-vector (communication-bound vs computation-bound)

7 Conclusion

Key takeaways:

1. Communication overhead can kill parallelization benefits for small problems

2. Embarrassingly parallel problems (Pi) scale much better than communication-heavy ones (matrix-vector)
3. Proper load balancing handles non-divisible workloads
4. Problem size matters: need enough computation to amortize communication cost