Part 1 - Laplace Solver - MPI

Ferran Mirabent Rubinat (1528268) - Marti Lorente Quintana (1444955)

November 2025

Abstract

This report describes the parallelization strategy, implementation details, experimental methodology, and performance analysis of a Laplace equation solver parallelized using MPI. The solver implements the Jacobi iteration method on a 2D grid with domain decomposition. Two communication strategies are compared: blocking and non-blocking MPI operations. The implementation is analyzed through strong and weak scaling experiments across different processor counts and compared against a previous OpenMP implementation.

Introduction

The sequential Laplace solver code was provided from previous lab assignments, and the goal of this assignment is to parallelize the heat diffusion process using the Message Passing Interface (MPI). The simulation runs for a fixed number of iterations (100), computing the average of neighboring cells using the Jacobi iteration method until convergence.
The main challenges addressed in this work are:

- Correct distribution of the matrix across MPI processes
- Heat propagation across subdomain boundaries using halo exchanges
- Efficient convergence checking across distributed processes
- Comparison between blocking and non-blocking communication strategies
- Performance analysis across different scales and problem sizes

Parallelization Strategy

Domain Decomposition

The simulation uses a 1D row-wise decomposition. Given a global grid of size $rows \times columns$, the total number of rows is divided evenly among MPI ranks:

- Each rank owns $chunk = rows/size$ real rows
- Two additional rows are allocated for halo/ghost layers
- The domain portion handled by each rank is: Rank r: $[r \cdot chunk, (r+1) \cdot chunk - 1]$

This approach simplifies communication and minimizes boundary complexity.

Halo Exchange for Heat Propagation

Heat diffusion requires values from neighboring rows. To ensure correct propagation across process boundaries, each iteration performs:

- Send the first real row to the upper neighbor and receive into the top halo row
- Send the last real row to the lower neighbor and receive into the bottom halo row

Note that the first process (rank 0) only has one halo row (bottom), and the last process only has one halo row (top), as they lack neighbors on one side. Interior processes maintain two halo rows for bidirectional communication.
This ensures seamless heat propagation across MPI subdomains.

Blocking Communication

The initial implementation uses `MPI_Sendrecv` to exchange boundary rows with neighboring processes in a single blocking call:

```
MPI_Sendrecv(&matrix[1][0], columns, MPI_FLOAT, rank-1, TAG,
             &matrix[0][0], columns, MPI_FLOAT, rank-1, TAG,
             MPI_COMM_WORLD, &status);
```

Convergence checking uses `MPI_Allreduce` to compute the global error by summing local errors across all processes. The blocking approach guarantees communication completion before computation continues, but introduces synchronization points where processes may wait idle.

Non-Blocking Communication

To reduce synchronization overhead and enable potential computation-communication overlap, a non-blocking version was implemented using:

- `MPI_Irecv/MPI_Isend`: Non-blocking routines that initiate communication and return immediately
- `MPI_Waitall`: Ensures all pending communications complete before using received data

The non-blocking approach aims to reduce idle time and improve scalability, though the current implementation posts and waits immediately, limiting actual overlap potential.

Global Residual and Convergence

Each local domain computes a residual value based on the difference between iterations. A global maximum residual is computed via:

```
MPI_Allreduce(&local_residual, &global_residual, 1,
              MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
```

The simulation terminates after 100 iterations.

Reasoning Behind the Parallelization Strategy

The 1D row-wise decomposition was chosen for several reasons:

- The stencil for heat diffusion requires only vertical neighbor communication
- Communication volume is minimized (only two halo rows exchanged per iteration)
- Implementation is simple and avoids complex 2D Cartesian communicators
- It ensures balanced load as long as rows are evenly divisible

This strategy provides a good balance of implementation simplicity and computational scalability.

Performance Results

Experimental Methodology

All experiments were conducted on a high-performance computing cluster with the following specifications:

- Hardware: 1-8 nodes, 12 processors per node
- Software: OpenMPI, TAU profiling toolkit
- Measurement: Execution times measured using TAU, communication overhead profiled

Two scaling studies were performed:
Strong Scaling: Fixed problem size (24000 × 24000 matrix, 100 iterations) with 12, 24, 48, and 96 processors.
Weak Scaling: Problem size scales with processors (200 rows/processor): 2400×2400, 4800×4800, 9600×9600, 19200×19200.

Comparison with OpenMP

For baseline comparison, single-node performance (12 threads/processors, 4096×4096 matrix, 100 iterations) was measured:

Implementation	**Execution Time (s)**
OpenMP (12 threads)	0.4101
MPI Blocking (12 procs)	3.3192
MPI Non-Blocking (12 procs)	3.5428

Table 1: Single-node performance comparison (4096×4096 matrix).

OpenMP significantly outperforms MPI on a single node, achieving approximately 8× faster execution. This is expected because shared-memory parallelism (OpenMP) avoids communication overhead entirely, while MPI incurs message-passing costs even within a single node.
Note: At this small problem size, blocking MPI slightly outperforms non-blocking (3.32s vs 3.54s), likely because the overhead of managing asynchronous operations exceeds any overlap benefits. However, as shown in subsequent sections, non-blocking communication demonstrates clear advantages at larger scales and problem sizes.

Execution Time Analysis

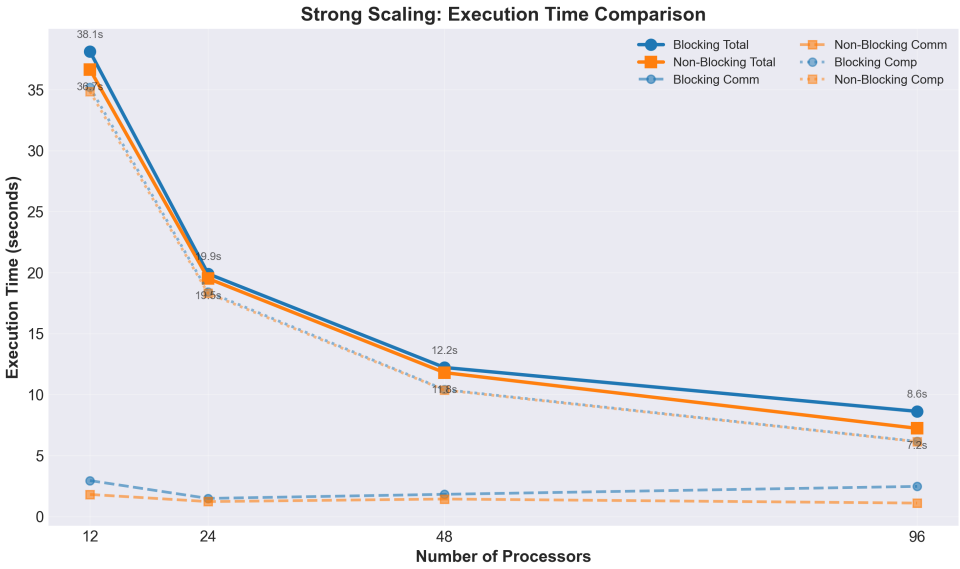

Figure 1: Strong scaling execution time breakdown. Solid lines show total time, dashed lines show communication time, dotted lines show computation time.

Procs	Blocking			Non-Blocking		
	Total (s)	Comm (s)	Comm %	Total (s)	Comm (s)	Comm %
12	38.13	2.94	7.7	36.65	1.80	4.9
24	19.90	1.48	7.4	19.51	1.22	6.2
48	12.22	1.81	14.8	11.79	1.43	12.1
96	8.62	2.46	28.6	7.22	1.09	15.1

Table 2: Strong scaling execution time breakdown (24000×24000 matrix, 100 iterations).

Key Observations:

- Non-blocking communication consistently achieves lower total execution time at all scales
- At 96 processors, non-blocking reduces total time by 16% (7.22s vs 8.62s)
- Communication overhead grows significantly with scale for blocking (7.7% to 28.6%)

- Non-blocking maintains lower communication overhead at all scales, reaching only 15.1% at 96 processors—nearly 50% reduction compared to blocking

Speedup and Efficiency

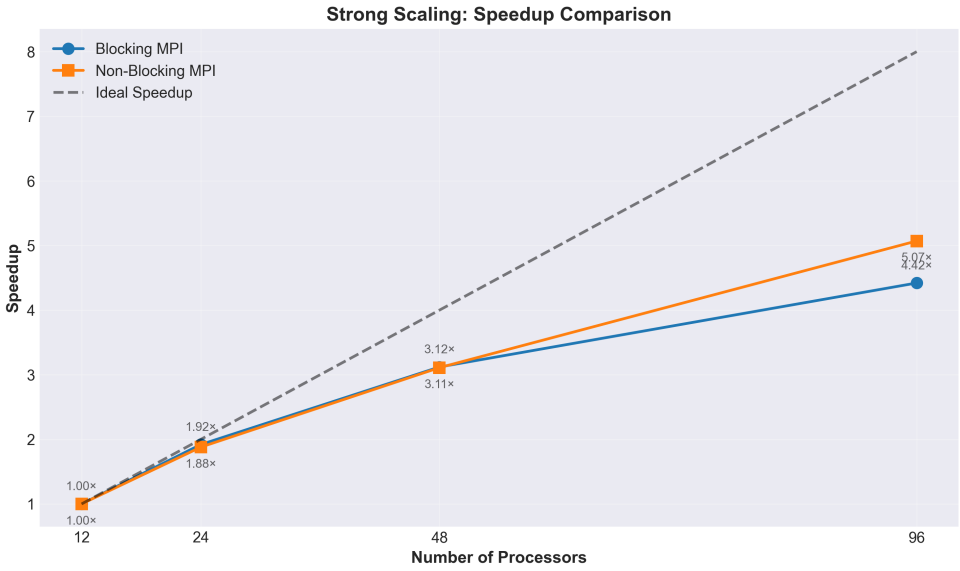

Figure 2: Strong scaling speedup comparison against ideal linear speedup.

Procs	Blocking		Non-Blocking	
	Speedup	Efficiency (%)	Speedup	Efficiency (%)
12	1.00	100.0	1.00	100.0
24	1.92	95.8	1.88	93.9
48	3.12	78.0	3.11	77.7
96	4.42	55.3	5.07	63.4

Table 3: Strong scaling speedup and efficiency.

Analysis:

- Non-blocking achieves 5.07× speedup at 96 processors compared to blocking's 4.42×—a 15% improvement
- Efficiency remains above 63% for non-blocking at 96 processors versus 55% for blocking

- Near-linear scaling is maintained up to 48 processors for both implementations
- Non-blocking demonstrates superior scalability characteristics at larger process counts

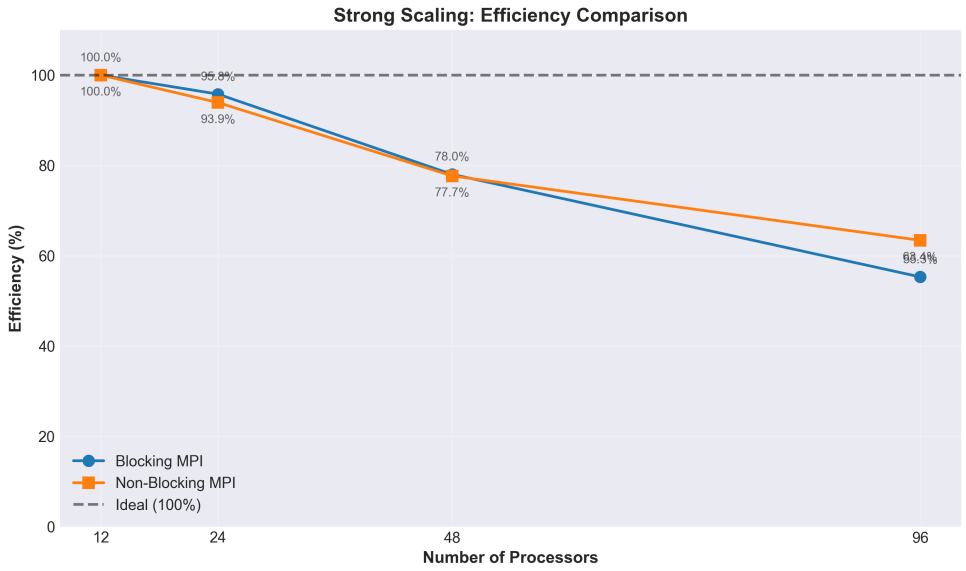

Figure 3: Strong scaling efficiency comparison showing better efficiency retention for non-blocking communication.

Weak Scaling Analysis

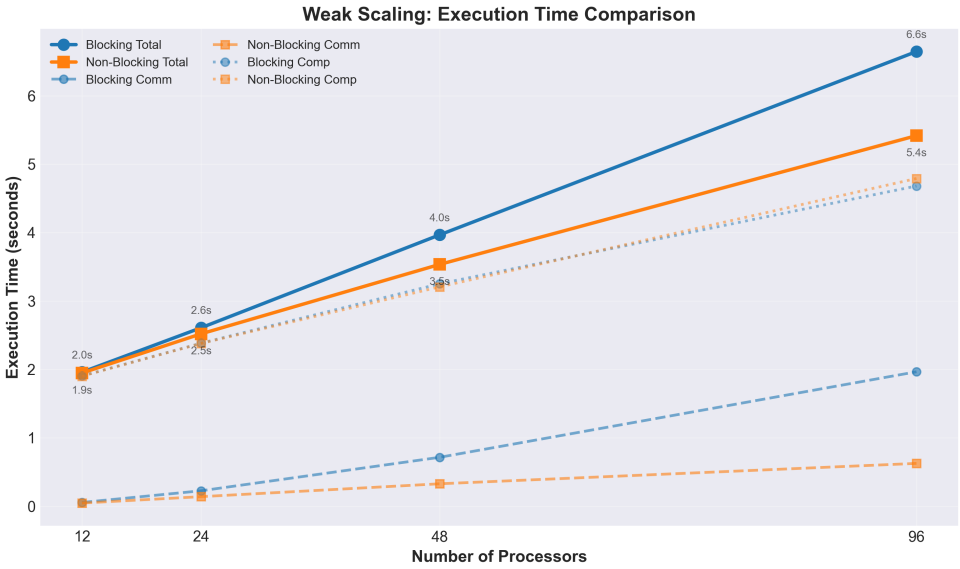

Figure 4: Weak scaling execution time. Problem size scales with processor count (200 rows/processor). Ideal weak scaling would maintain constant execution time.

Procs	Matrix Size	Blocking			Non-Blocking		
		Total (s)	Comm (s)	Comm %	Total (s)	Comm (s)	Comm %
12	2400×2400	1.96	0.06	2.8	1.95	0.05	2.4
24	4800×4800	2.61	0.23	8.7	2.52	0.14	5.6
48	9600×9600	3.97	0.72	18.0	3.53	0.33	9.3
96	19200×19200	6.65	1.97	29.6	5.42	0.63	11.6

Table 4: Weak scaling execution time breakdown (200 rows/processor, 100 iterations).

Critical Finding: Non-blocking communication demonstrates significantly better weak scaling characteristics. At 96 processors:

- Blocking: Execution time increased 3.39× versus baseline, with 29.6% communication overhead
- Non-blocking: Execution time increased only 2.78×, with 11.6% communication overhead

- Communication overhead for non-blocking is less than half that of blocking at large scale

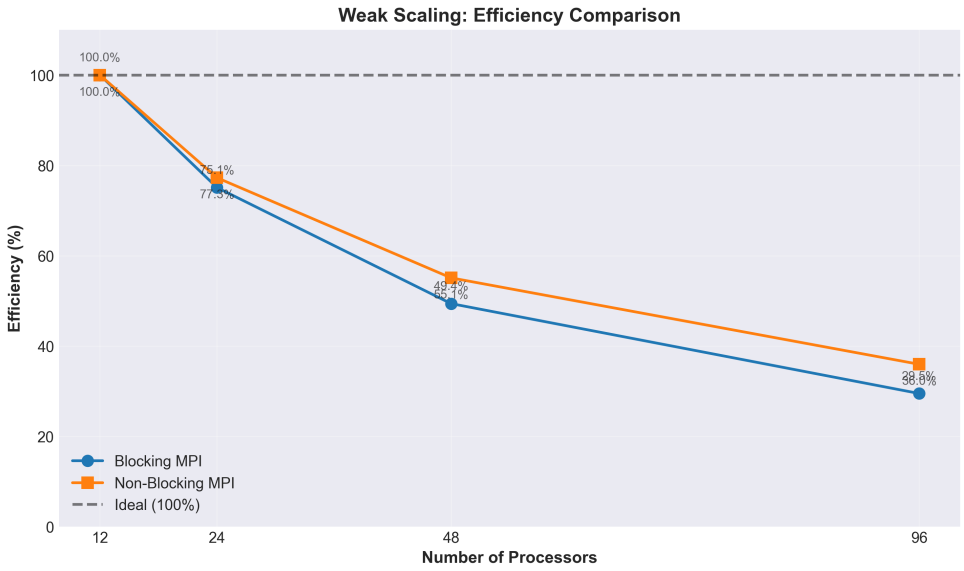

Figure 5: Weak scaling efficiency comparison demonstrating superior scalability of non-blocking communication.

Processors	Blocking Eff. (%)	Non-Blocking Eff. (%)
12	100.0	100.0
24	75.1	77.3
48	49.4	55.1
96	29.5	36.0

Table 5: Weak scaling efficiency showing 6.5 percentage point advantage for non-blocking at 96 processors.

TAU Profiling Summary

TAU profiling revealed important differences between blocking and non-blocking implementations:

Blocking Implementation:

- Significant load imbalance detected in `MPI_Sendrecv` (up to 3.9s variance)
- `MPI_Allreduce` shows up to 2.2s variance across processes

- Communication overhead grows from 7.7% to 28.6% at largest scale

Non-Blocking Implementation:

- Reduced load imbalance across all MPI operations
- 8-node (96 processor) configuration showed zero load imbalance warnings
- Communication overhead remains below 16% even at 96 processors
- Better process independence reduces synchronization bottlenecks

Scalability Summary

Non-blocking communication provides consistent and measurable improvements:

- **Strong scaling:** 15% better speedup at 96 processors (5.07× vs 4.42×)
- **Weak scaling:** 22% better efficiency retention at 96 processors (36.0% vs 29.5%)
- **Communication overhead:** Approximately halved in percentage terms at large scale (15.1% vs 28.6%)
- **Load balance:** Superior balance characteristics, particularly at larger scales

The benefits of non-blocking communication are most pronounced when scaling across multiple nodes where communication latency becomes significant.

Conclusions

This study demonstrates successful MPI parallelization of a Laplace equation solver with the following key findings:

1. Parallelization Success: Both implementations achieve reasonable scaling characteristics. Non-blocking communication reaches 5.07× speedup at 96 processors (63.4% efficiency) compared to blocking's 4.42× speedup (55.3% efficiency) in strong scaling tests.

2. OpenMP vs MPI Trade-offs: OpenMP significantly outperforms MPI on single-node configurations (0.41s vs 3.3s for small problems), demonstrating the inherent advantage of shared-memory parallelism when inter-node communication is unnecessary. However, MPI enables scaling beyond a single node, which is essential for larger problems that exceed single-node memory or computational capacity.

3. Problem Size Dependency: Performance critically depends on computation-to-communication ratio. Small problems (4096×4096) show minimal benefit from non-blocking due to low computation time. Larger problems (24000×24000) consistently demonstrate non-blocking advantages, with communication overhead reduced from 28.6% to 15.1% at 96 processors.

4. Non-Blocking Advantages: Non-blocking communication provides consistent improvements particularly at larger scales:

- 15% better speedup at 96 processors
- 6.5 percentage points better weak scaling efficiency
- Reduced load imbalance and synchronization bottlenecks
- Lower communication overhead percentage at all multi-node configurations

5. Scalability Characteristics: Both implementations show degrading efficiency as scale increases, primarily due to growing communication overhead. This is expected in distributed-memory systems and represents a fundamental challenge for fine-grained stencil computations. Non-blocking communication partially mitigates this effect through reduced synchronization overhead.