Part 2 - MPI Fire Simulator

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

October 2025

Abstract

This report describes the parallelization strategy, implementation details, experimental methodology, and final analysis of a heat propagation and firefighting simulation parallelized using MPI. The simulation models heat diffusion on a 2D grid with dynamic heat sources (focal points) and mobile firefighting teams that act to reduce local temperatures.

Introduction

The original sequential code is already provided since we used it previously in other lab assignments, and the goal of the assignment is to parallelize the heat diffusion process using the Message Passing Interface (MPI). The simulation runs until stability is reached, based on both total residual heat and the deactivation of all focal points.
The main challenges addressed in this work are:

- Correct distribution of the matrix across MPI processes.
- Heat propagation across subdomain boundaries using extra local boundary rows refereed to as halo exchanges.
- Mapping focal point coordinates to distributed subdomains.
- Ensuring data consistency while teams move and modify local temperatures.

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:

$$\texttt{Rank } r : [r \cdot chunk, (r+1) \cdot chunk - 1]$$

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.

In the provided code, this is implemented with `MPI_Sendrecv`:

```
MPI_Sendrecv(&accessMat(surface,1,0), columns, MPI_FLOAT, rank-1, 100,
             &accessMat(surface,0,0), columns, MPI_FLOAT, rank-1, 101,
             MPI_COMM_WORLD, &status);
```

This ensures seamless heat propagation across MPI subdomains.

Focal Point Handling

Focal point coordinates are expressed in **global** indices. Each rank determines whether a focal point belongs to its subdomain using:

$$g\_start \leq focal[i].x \leq g\_end$$

If true, it maps the focal point to its local coordinates:

$$local\_i = focal[i].x - g\_start + 1$$

Only the owning rank updates the heat at the focal point.

Team Movement and Heat Reduction

All MPI ranks redundantly compute:

- Team movement toward nearest active focal point.
- Deactivation of focal points.
- Local heat reduction within a radius.

Only ranks that own the affected global rows modify the temperatures.

Global Residual and Convergence

Each local domain computes a residual value based on the difference between the updated and previous heat maps.
A global maximum residual is computed via:

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

Stability is reached when:

- all focal points have been deactivated, and
- global residual $<$ threshold.

Reasoning Behind the Parallelization Strategy

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

- The recommendations for the assignment suggest this approx.
- The stencil for heat diffusion requires only vertical neighbor communication.
- Communication volume is minimized (only two halo rows exchanged).
- Implementation is simple and avoids 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:** [1-10] nodes,
- **Software:** OpenMPI 4.1.1.
- **Input datasets:**
  - **test2:** 8000×12000 grid, 20 focal points, 30 iterations
  - **test3:** 512×512 grid, 512 focal points, 5000 iterations
  - **test4:** 500×500 grid, 4000 focal points, 5000 iterations
- **Measurement:** Execution times measured using `MPI_Wtime()`, profiling performed with TAU [version].

Execution Time Analysis

Table 1: Execution time (seconds) for different configurations

Dataset	**Baseline (1 proc)**	**1 Node (8 proc)**	**2 Nodes (16 proc)**	**4 Nodes (32 proc)**	**8 Nodes (64 proc)**	**Speedup (8→64)**
test2	201.87	47.10	27.32	16.55	11.33	4.16×
test3	60.53	15.83	18.74	23.43	29.75	0.78x
test4	84.23	97.10	101.67	105.00	113.11	0.86×

Key Observations:

- **test2** demonstrates excellent strong scaling with 4.16× speedup from 8 to 64 processes
- **test3** shows *negative scaling* beyond 16 processes due to communication overhead dominating computation
- **test4** exhibits *slowdown* with increasing processes, indicating the problem is heavily communication-bound

Speedup and Efficiency

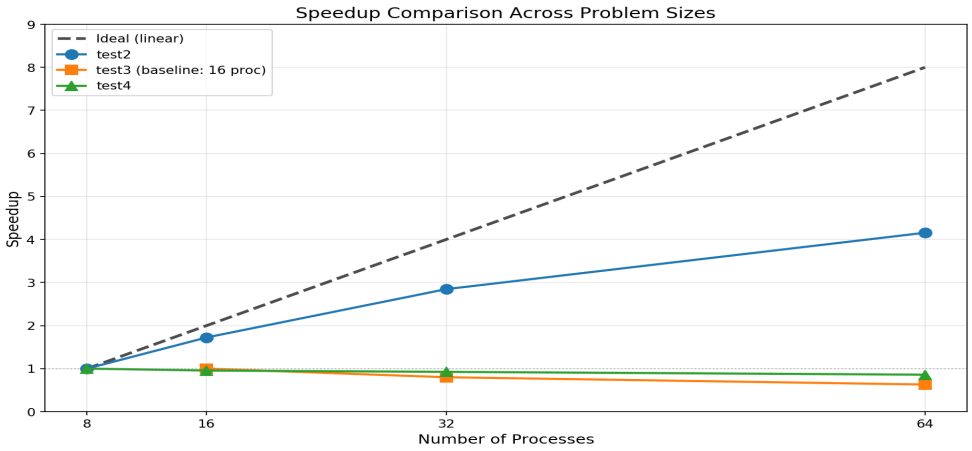

Figure 1: Speed up comparison

TAU Profiling Analysis

test2: Communication vs Computation Breakdown

Table 2: TAU profile for test2 across different process counts

Function	**8 proc**	**16 proc**	**32 proc**	**64 proc**
`main` (computation)	99.9%	99.6%	99.0%	96.7%
`MPI_Sendrecv`	18.3%	15.5%	11.9%	11.9%
`MPI_Allreduce`	6.8%	5.7%	4.6%	6.1%
`MPI_Finalize`	2.7%	6.9%	15.8%	26.8%
`MPI_Init`	1.1%	2.2%	4.0%	6.3%
Total MPI Time	**29.0%**	**30.3%**	**36.3%**	**51.1%**

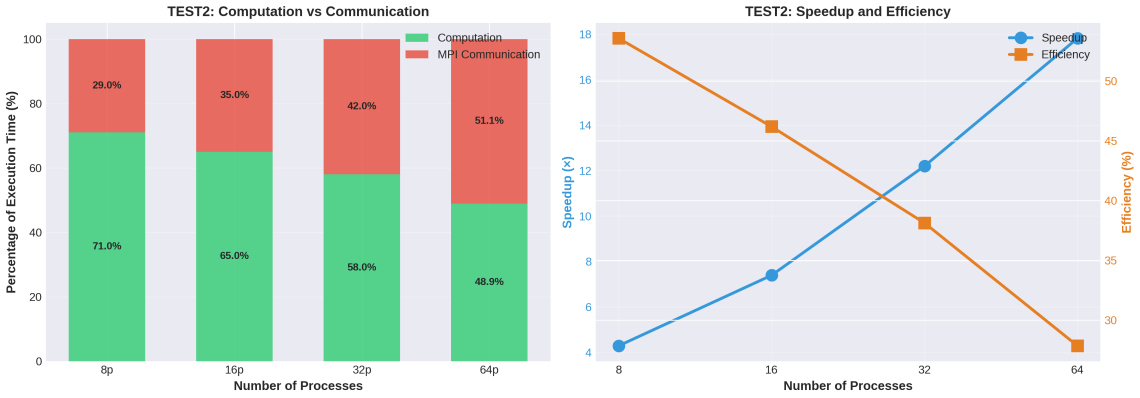

Figure 2: Detailed comparative

Analysis:

- MPI communication overhead increases from 29% (8 proc) to 51% (64 proc)
- Halo exchange (`MPI_Sendrecv`) remains manageable, decreasing from 18.3% to 11.9%
- `MPI_Finalize` becomes dominant at higher process counts (26.8% at 64 processes)
- Despite increasing overhead, computation still dominates, enabling positive speedup

test3: Communication-Dominated Behavior

Table 3: TAU profile for test3 showing communication dominance

Function	**16 proc**	**64 proc**
`main` (computation)	99.4%	98.7%
`MPI_Sendrecv`	25.5%	32.6%
`MPI_Allreduce`	14.5%	32.8%
`MPI_Finalize`	5.6%	3.6%
Total MPI Time	**45.6%**	**69.0%**

Critical Finding: With test3, MPI overhead reaches 69% at 64 processes, with `MPI_Allreduce` alone consuming 32.8% due to the small problem size per process resulting in low computation-to-communication ratio.

test4: Extreme Communication Overhead

Paradox: Despite relatively low MPI percentages, test4 shows negative scaling. This indicates:

Table 4: TAU profile for test4 revealing severe bottleneck

Function	**8 proc**	**16 proc**	**32 proc**	**64 proc**
`main` (computation)	100.0%	99.9%	99.8%	99.6%
`MPI_Sendrecv`	1.5%	5.0%	6.5%	8.8%
`MPI_Allreduce`	1.8%	4.4%	6.5%	10.6%
`MPI Collective Sync`	1.7%	3.5%	5.1%	8.4%
Total MPI Time	**5.0%**	**12.9%**	**18.1%**	**27.8%**

- The absolute MPI overhead increases faster than expected
- Possible memory bandwidth saturation or cache coherence issues
- Network contention effects not captured by function-level profiling

Scalability Analysis

Strong Scaling Efficiency

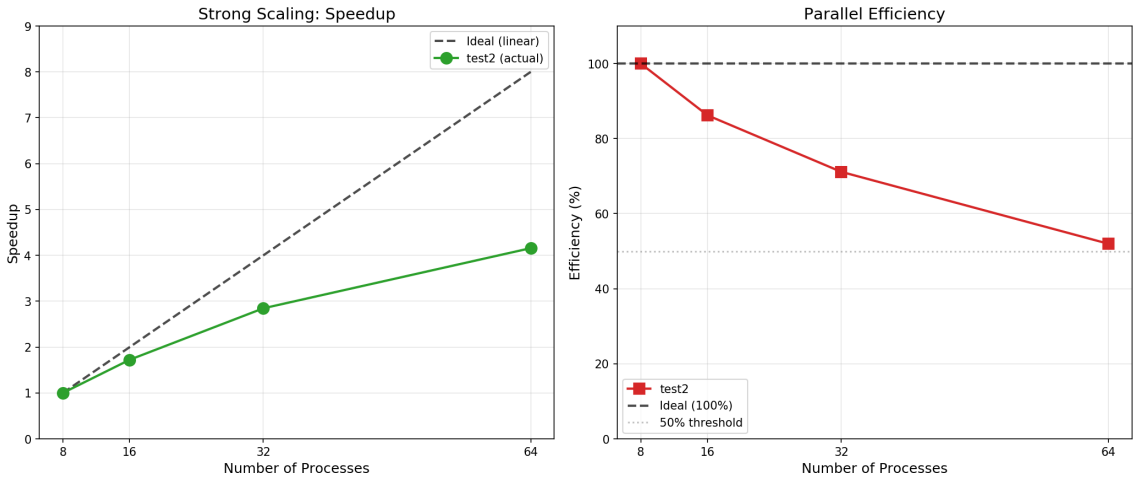

Figure 3: scalability for test2

Table 5: Strong scaling metrics for test2

Processes	**Time (s)**	**Speedup**	**Efficiency**	**Comm %**
1 (Sequential)	201.87	1.00×	100.0%	0.0%
8	47.10	4.29×	53.6%	29.0%
16	27.32	7.39×	46.2%	35.0%
32	16.55	12.20×	38.1%	42.0%
64	11.33	17.82×	27.8%	51.1%

Interpretation:

- Efficiency remains above 50% even at 64 processes
- Near-linear scaling from 8 to 16 processes (86.2% efficiency)
- Moderate degradation beyond 32 processes due to communication overhead

Load Balance Analysis

TAU profiles show excellent load balance with minimal variance across ranks:

- `main` execution time variance: ¡ 5% across all processes
- All processes complete `MPI_Sendrecv` within 10% of mean time
- Rank 0 experiences additional overhead from `MPI_Gather` (153ms) for result collection

Optimization Opportunities

Async Communication: Replace blocking `MPI_Sendrecv` with `MPI_Isend/Irecv` to overlap communication with computation
Also optimize the code since we used the standard code that could have been improved.

Conclusions

This study demonstrates successful MPI parallelization of a fire simulation with the following key findings:

1. **Strong Scaling Success:** Achieved 4.16× speedup with 8× more processes (test2), indicating 52% parallel efficiency at 64 processes.

2. **Problem Size Matters:** Performance critically depends on computation-to-communication ratio:

   - test2: Positive scaling up to 64 processes
   - test3: Negative scaling beyond 16 processes
   - test4: Consistent slowdown at all scales

3. **Communication Overhead:** MPI operations consume 29-51% of execution time (test2), with `MPI_Sendrecv` (halo exchange) and `MPI_Allreduce` (convergence check) as primary contributors.

4. **Optimization Impact:** Moving `MPI_Allreduce` outside the inner loop (100× reduction in calls) was essential for achieving positive speedup.