



# Addis Ababa University

Distributed Systems for AI

**Adaptive Multi-Agent Reinforcement Learning Framework for  
AI-Driven Planetarium Climate Visualization**

**Submitted by:**

Tsion Buzuayehu  
GSR/9235/17

**Submitted to:** Dr Beakal Gizachew  
**Submission Date:** December 13, 2025

# 1 MPI Parallelization (1D and 2D Geometries)

The provided serial Aliev–Panfilov cardiac simulator was extended with MPI to support distributed-memory parallelism on a 2D grid of excitation and recovery variables.

## 1D geometry (only -y set)

- Process grid:  $\text{px} = 1$ ,  $\text{py} = P$ , giving a  $1 \times P$  layout.
- The global mesh is partitioned into horizontal strips; each rank holds a contiguous block of rows.
- If  $m \bmod \text{py} \neq 0$ , the last rank in the column is assigned extra rows so that the full domain is covered even when  $P$  does not divide  $N$  evenly.
- Ghost rows are contiguous in memory. Top/bottom halos are exchanged with non-blocking MPI calls:
  - send/receive  $E_{\text{prev}}[m][1 \dots n]$  to/from the process below;
  - send/receive  $E_{\text{prev}}[1][1 \dots n]$  to/from the process above.

## 2D geometry (-x and -y set)

- Process grid:  $\text{px} \times \text{py}$ , with  $\text{rowID} = \text{rank}/\text{px}$  and  $\text{colID} = \text{rank} \bmod \text{px}$ .
- Vertical neighbors exchange ghost rows in the same contiguous manner as 1D.
- Horizontal neighbors require packing and unpacking columns into 1D buffers because columns are not contiguous:
  - right ghost column: pack  $E_{\text{prev}}[j][n]$  into `sendbuffer[j-1]` and send to the right neighbor;
  - left ghost column: pack  $E_{\text{prev}}[j][1]$  into `sendbuffer2[j-1]` and send to the left neighbor;
  - received data are written back into  $E_{\text{prev}}[j][n + 1]$  or  $E_{\text{prev}}[j][0]$ .

## Non-even division and square meshes

- The dimensions  $m$  and  $n$  are divided by  $\text{py}$  and  $\text{px}$  respectively; the remainder is given to the last row and column of ranks so that arbitrary square meshes are handled even when  $N$  is not divisible by  $\text{px}$  and  $\text{py}$ .

## Correctness across cores and geometry

For  $N = 1024$ ,  $T = 100$  and various  $P \in \{1, 2, 4, 8, 16\}$  with 1D geometry, the final maximum excitation value  $\text{Max} \approx 0.979365$  and L2 norm  $\text{L2norm} \approx 0.498186$  are identical in the first few digits, indicating consistent results across different process counts. The same Max/L2 behavior was observed in hybrid runs, indicating independence from process count and use of OpenMP to within roundoff.

## 2 OpenMP Parallelism within MPI Processes

OpenMP was added to exploit shared-memory parallelism on each node.

- The program accepts `-numthreads` from the command line and stores it in `num_threads`.
- The `simulate` function was extended with an extra parameter `int num_threads`, and `main` passes this value on every call.
- Major loops over the local subdomain are annotated with:

```
#pragma omp parallel for num_threads(num_threads) private(i)
```

or with `private(j)` for column loops, covering:

- ghost-cell updates for top/bottom and left/right boundaries;
  - the five-point stencil PDE update of  $E$ ;
  - the ODE updates for  $E$  and  $R$ .
- These changes allow hybrid configurations such as 2 MPI processes  $\times$  8 threads per process (`-np 2 -numthreads 8`), as required in part (c).

## 3 Optimizations and Code Transformations

The code preserves the numerical algorithm of the serial reference but includes performance-oriented choices.

- **Contiguous 2D arrays:** `alloc2D` allocates the 2D arrays as a single continuous memory block with row pointers, improving cache locality and simplifying MPI packing.
- **Loop structure:** ghost updates and PDE/ODE loops are simple nested `for` loops over local indices, which vectorize well and avoid unnecessary conditionals inside the inner loop.

- **Precomputation of constants:** the time step  $dt$ , diffusion coefficient  $\alpha$ , and related constants are computed once before the main time-integration loop.

No changes were made to the mathematical model or boundary conditions, ensuring that the parallel version reproduces the same solution as the serial version up to floating-point differences.

## 4 Performance Study (Gflops-based)

All performance is reported using the “Sustained Gflops Rate” printed by the program; elapsed time is recorded but not used as the main metric, in line with the assignment.

### 4.1 Strong Scaling, $N = 1024$ , $T = 100$ (MPI only, no OpenMP)

Configuration: `numthreads = 1, nocomm = 0, 1D geometry with px = 1, py = P.`

#### Command pattern

```
mpirun [--oversubscribe] -np P ./cardiac \
-n 1024 -t 100 -x 1 -y P -numthreads 1 -nocomm 0 -plotfreq 0
```

#### Measured results

P	Geometry (px × py)	Elapsed (s)	Gflops
1	1×1	73.69	9.23
2	1×2	68.95	9.87
4	1×4	81.41	8.36
8	1×8	99.59	6.83
16	1×16	126.97	5.36

#### Interpretation

$P = 2$  yields the highest Gflops (approximately 9.87), slightly outperforming  $P = 1$  and representing the best strong-scaling point on this machine. For  $P > 2$ , Gflops decreases as  $P$  increases; on this WSL laptop this is caused by oversubscription (too many MPI ranks for available cores) and increased communication/synchronization costs. Under these conditions, the “optimal” processor geometry for  $N = 1024$  among the tested MPI-only configurations is  $1 \times 2$  ( $P = 2$ ). A plot of Gflops versus  $P$  can be generated from this table.

## 4.2 Communication Overhead via `-nocomm`

The communication overhead was estimated using the indirect method by disabling MPI communication.

Configuration:  $P = 4$ , geometry  $1 \times 4$  ( $\text{px} = 1$ ,  $\text{py} = 4$ ), `numthreads = 1`. For each  $N$ , two runs were performed: with communication (`-nocomm 0`) and without communication (`-nocomm 1`).

### Command pattern

```
# With communication
mpirun --oversubscribe -np 4 ./cardiac -n N -t 100 \
-x 1 -y 4 -numthreads 1 -nocomm 0 -plotfreq 0

# Without communication
mpirun --oversubscribe -np 4 ./cardiac -n N -t 100 \
-x 1 -y 4 -numthreads 1 -nocomm 1 -plotfreq 0
```

### Measured elapsed times

$N$	$T_{\text{comm}}$ (s)	$T_{\text{nocomm}}$ (s)
1024	84.52	80.91
724	21.09	22.05
512	2.87	5.16

### Overhead formula

$$\text{overhead}(N) = \frac{T_{\text{comm}} - T_{\text{nocomm}}}{T_{\text{comm}}} \times 100\%.$$

### Interpretation

For  $N = 1024$ , the difference between  $T_{\text{comm}}$  and  $T_{\text{nocomm}}$  is small, giving a low overhead percentage. For  $N = 724$  and  $N = 512$ ,  $T_{\text{nocomm}}$  is actually larger than  $T_{\text{comm}}$ , resulting in negative overhead; disabling communication slows the code on this system. This indicates that, under this single-node WSL environment, communication is not the dominant cost at these problem sizes; cache effects and oversubscription matter more than pure message-passing time. The 25% overhead threshold mentioned in the assignment is not reached and should be explicitly stated as a consequence of the hardware and execution environment. A small plot of overhead percentage versus  $N$  can be produced from this table.

### 4.3 Hybrid MPI+OpenMP Strong Scaling (1D geometry)

The hybrid study repeats strong scaling with OpenMP threads per process.

Configuration:

- $N = 1024$ ,  $T = 100$ ;
- 1D geometry ( $\text{px} = 1$ ,  $\text{py} = P$ );
- `nocomm = 0`;
- vary  $P$  and `numthreads` so that  $P \times \text{threads} \approx 16$ .

#### Commands

```
# 8 MPI processes × 2 threads per process
mpirun --oversubscribe -np 8 ./cardiac -n 1024 -t 100 \
-x 1 -y 8 -numthreads 2 -nocomm 0 -plotfreq 0

# 4 MPI processes × 4 threads per process
mpirun --oversubscribe -np 4 ./cardiac -n 1024 -t 100 \
-x 1 -y 4 -numthreads 4 -nocomm 0 -plotfreq 0

# 2 MPI processes × 8 threads per process
mpirun --oversubscribe -np 2 ./cardiac -n 1024 -t 100 \
-x 1 -y 2 -numthreads 8 -nocomm 0 -plotfreq 0
```

#### Measured results

P	Threads	$P \times \text{threads}$	Geometry	Elapsed (s)	Gflops
8	2	16	$1 \times 8$	10085.6	0.07
4	4	16	$1 \times 4$	759.2	0.90
2	8	16	$1 \times 2$	74.0	9.20

For comparison, the MPI-only run with  $P = 16$ ,  $\text{threads} = 1$  achieved approximately 5.36 Gflops.

#### Interpretation and fastest combination

Configurations  $8 \times 2$  and  $4 \times 4$  perform very poorly (Gflops  $< 1$ ), because the environment is heavily oversubscribed: many ranks and threads contend for the same CPU cores. The  $2 \times 8$  configuration delivers about 9.20 Gflops, similar to the best MPI-only  $P = 2$  result and significantly better than MPI-only  $P = 16$ . Therefore, on this system, the fastest tested MPI+OpenMP combination is:

*2 MPI processes  $\times$  8 threads each, geometry 1 $\times$ 2.*

A bar chart of Gflops versus configuration (16 $\times$ 1, 8 $\times$ 2, 4 $\times$ 4, 2 $\times$ 8) can be used to visually compare the hybrid and MPI-only strong-scaling curves.