HPC Assignment Presentation

Simone Pio Maurutto
Sara Maddalena Piccinini
Francesca Scognamiglio

Politecnico di Torino

January 2026

Index

- Systolic array structures for matrix multiplication with MPI
- Fundamentals of multi-dimensional data processing with CUDA
- Heat Diffusion Simulation in 2D Grid with OpenMP

General Overview

1. Introduction
2. Methodology
3. Metrics
4. Results and Analysis
5. Conclusions and Observations

Introduction

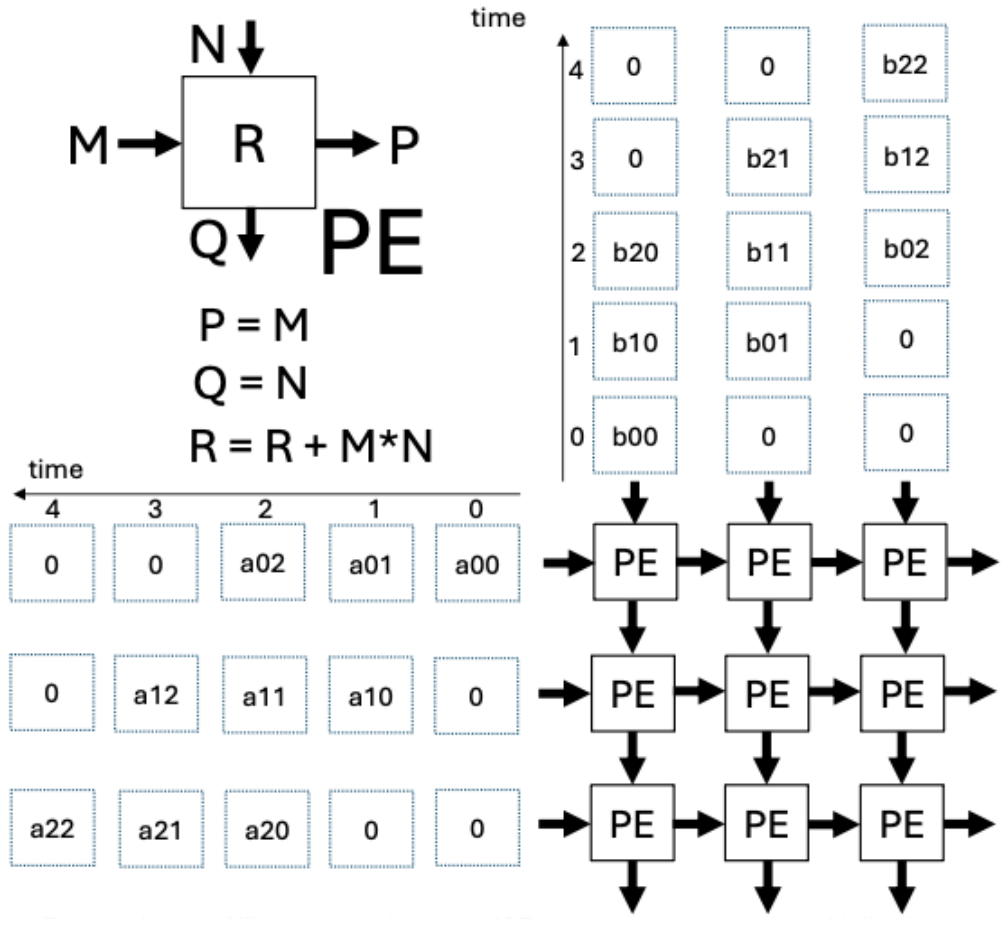
- Design and implementation of a systolic array architecture for matrix multiplication
- Distributed realization using MPI processing elements
- Performance evaluation and scalability analysis
- Discussion of results and trade-offs

Methodology – Software Workflow

1. Partition of input matrices among processing elements (PEs)
2. Cannon's algorithm initialization for block alignment
3. Cannon's algorithm computation phase
4. Gathering of local results into the output matrix

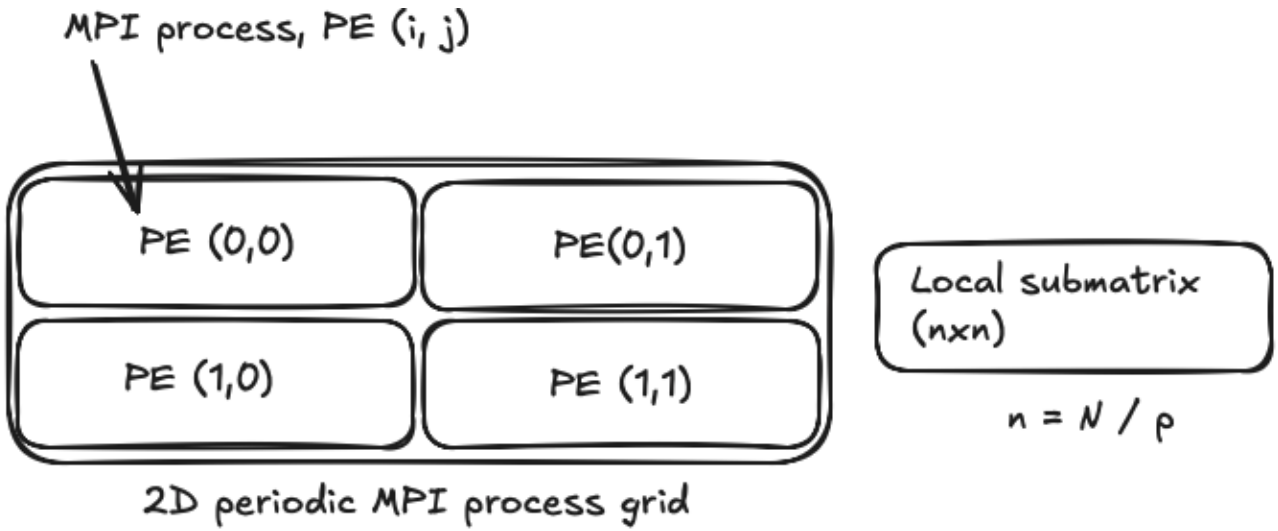
Methodology - Processing elements and Systolic Behavior

Processing Elements (PEs) are the fundamental units of the systolic array architecture. They perform local computations and propagate data across the array over time.



Methodology – Process Mapping

- Each Processing Element (PE) is mapped to one MPI process
- Block size: $n = N/p$ with N input matrices and p number of rocess per dimension
- Processes arranged in a 2D periodic grid
- Inter-process communication implemented via MPI



Methodology – Data Distribution and Initialization

- Input matrices are distributed among processing elements
- The PEsre initialized according to Cannon's algorithm requirements
- Data distribution and gathering implemented using MPI Scatter and Gather operations after the main computation

Data distribution among processing elements

```
MPI_Scatterv(A, counts, displs, blocktype2, A_block, block_size*block_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatterv(B, counts, displs, blocktype2, B_block, block_size*block_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

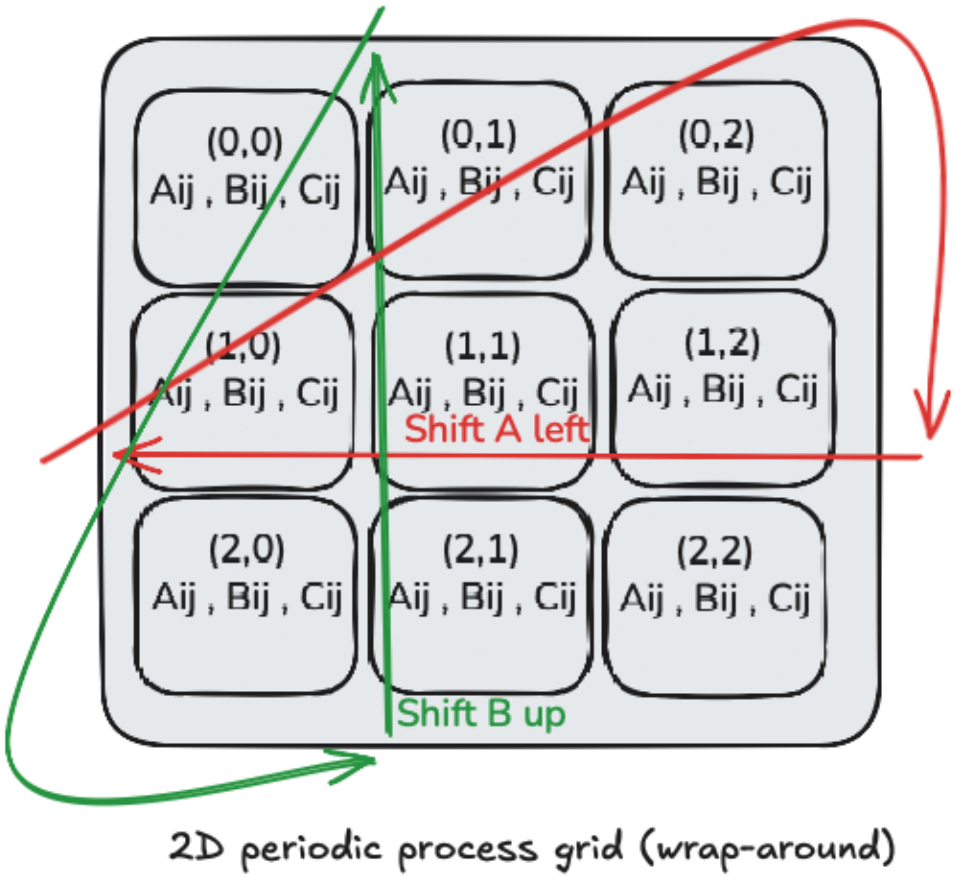
Result gathering after the computation

```
MPI_Gatherv(C_block, block_size*block_size, MPI_DOUBLE, C, counts, displs, blocktype2, 0, MPI_COMM_WORLD);
```

Methodology – Computation Phase

During each iteration, the processing elements:

1. Compute the local multiply
2. Exchange data blocks with neighboring processes (A left, B up on a 2D periodic grid)



Metrics

- Experiments were executed on one or two nodes of the **Legion HPC cluster**
- The number of MPI processes was varied according to the matrix size and the 2D process grid configuration
- Execution time was measured on the root process using **MPI_Wtime()**

Results and Analysis for the 500x500 Matrix

Sequential execution time: 3.37 s.

Table: Execution time and speed-up on 1 node (top) and on 2 nodes (bottom).

Processes	**Time (s)**	**Speed-Up**
4	0.24	14.04
16	0.13	25.92
25	0.13	25.92

Processes	**Time (s)**	**Speed-Up**
4	0.31	10.97
16	0.45	7.56
25	0.61	5.52

Results and Analysis for the 1000x1000 Matrix

Sequential execution time: 4.79 s.

Table: Execution time and speed-up on 1 node (top) and on 2 nodes (bottom).

Processes	Time (s)	Speed-Up
4	2.27	2.11
16	0.68	7.04
25	0.62	7.02

Processes	Time (s)	Speed-Up
4	2.15	2.22
16	1.05	5.56
25	1.61	2.97
64	2.33	2.05

Results and Analysis for the 2000x2000 Matrix

Sequential execution time: 38.40 s.

Table: Execution time and speed-up on 1 node (top) and on 2 nodes (bottom).

Processes	Time (s)	Speed-Up
4	15.33	2.50
16	3.91	9.82
25	3.92	9.79

Processes	Time (s)	Speed-Up
4	15.03	2.55
16	4.58	8.38
25	4.10	9.36
64	4.33	8.86

Conclusions and Observations:

- Increasing parallelism does not always translate into improved performance.
- Running the application on multiple nodes introduces additional overhead, leading to different performance compared to a single-node execution with the same number of processes.
- The most favorable speed-up values are obtained when computations are confined to a single node.

CUDA Filtering

- Image filtering with CUDA kernels
- Memory access patterns
- Speedup over CPU implementation

OpenMP Heat Diffusion

- Parallelization strategy
- Strong and weak scaling

Conclusions

- Summary of results
- Lessons learned
- Possible improvements