

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

Introduction

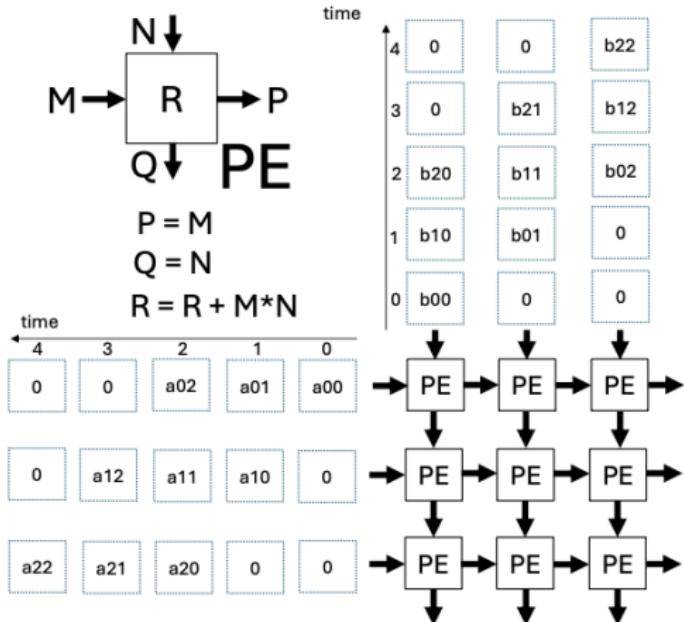
- Design and implementation of a systolic array architecture for matrix multiplication
- Distributed realization using MPI processing elements
- Performance evaluation and scalability analysis
- Discussion of results and trade-offs

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

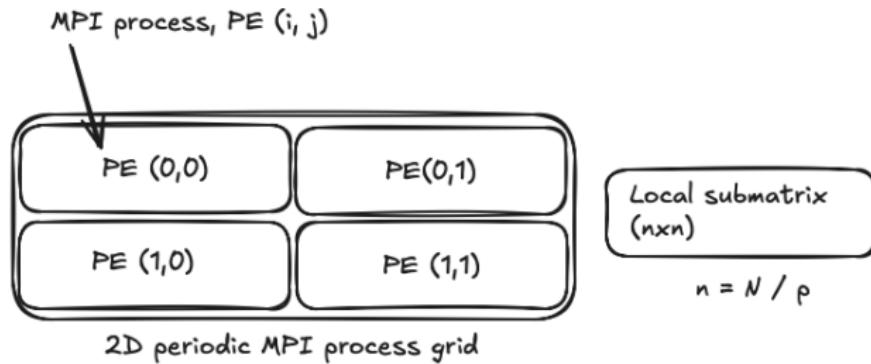
Methodology - Processing elements and Systolic Behavior

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 processes 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 PEs are 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);
```

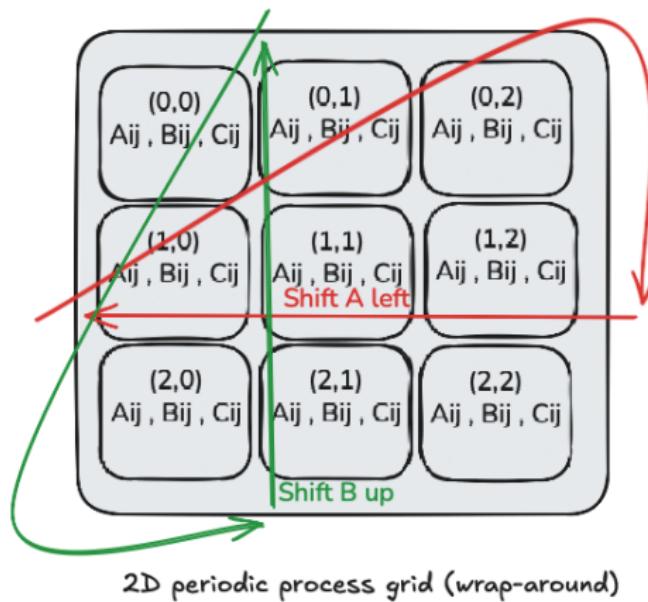
Result gathering after the computation

```
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: 3.40 s.

1 Node		
Processes	Time (s)	Speed-Up
4	0.28	12.14
16	0.15	22.67
25	0.15	22.67

2 Nodes		
Processes	Time (s)	Speed-Up
4	0.31	10.97
16	0.45	7.56
25	0.63	5.40

Results and Analysis for the 1000x1000 Matrix

Sequential execution time: 4.88 s.

1 Node		
Processes	Time (s)	Speed-Up
4	2.19	2.23
16	0.67	7.28
25	0.65	7.51

2 Nodes		
Processes	Time (s)	Speed-Up
4	2.10	2.32
16	1.08	4.52
25	1.36	3.59
64	2.33	2.09

Results and Analysis for the 2000x2000 Matrix

Sequential execution time: 40.38 s.

1 Node		
Processes	Time (s)	Speed-Up
4	15.95	2.53
16	3.83	10.54
25	3.42	11.81

2 Nodes		
Processes	Time (s)	Speed-Up
4	15.35	2.63
16	4.58	8.82
25	4.10	9.85
64	4.24	9.52

Conclusions

- 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