Assignment HPC

1st Carlo Mattioli
Politecnico di Torino
Turin, Italy
s349351@studenti.polito.it

2nd Fabio Calabrese
Politecnico di Torino
Turin, Italy
s343467@studenti.polito.it

3rd Michele Merla
Politecnico di Torino
Turin, Italy
s343500@studenti.polito.it

***Abstract*—This report presents a comparative analysis of three high-performance computing (HPC) paradigms—MPI, CUDA, and OpenMP—applied to real-world problems: matrix multiplication, image filtering, and heat diffusion, respectively. The first task implements a systolic array structure with MPI using Cannon's algorithm to parallelize matrix multiplication, showing scalability limits due to inter-process communication overhead. The second task leverages CUDA for stencil-based image filtering on high-resolution images, achieving significant speed-ups, especially with 16×16 and 32×8 thread block configurations. The third task uses OpenMP to simulate heat diffusion in isotropic and anisotropic materials, demonstrating strong scalability up to one thread per core. Results highlight that optimal parallel performance is achieved through careful workload partitioning, minimizing communication latency and balancing thread scheduling. The experiments were conducted on the Legion cluster, and performance was evaluated through speed-up and execution time across various workloads and configurations.**

I. Systolic Array Structures For Array Multiplication

A. Overview and Objectives

In modern scientific applications, the structure of an algorithm can be organized as a distributed application to speed up its performance execution. The goal of the first exercise is to implement matrix multiplication through the systolic array architecture. This approach involves decomposing the multiplication process in independent, fixed operations executed by multiple "processing elements" (PEs). These PEs are interconnected among them and communicate using MPI (Message Passing Interface) to form a synchronized and distributed network. This form of computation pipeline should be designed for matrices 500×500, 1000×1000 and 2000×2000.

B. Methodology

PE characterization

PEs are the operation units of this process. They are organized to execute the same operations on different values and propagate their results into other PEs for subsequent operations in time. The number of PEs is given by the number of processes involved in the computation. Their size depend on the square root of the number of processes and the matrix size, through the relation:

$$PE_{size} = \frac{N}{p} \tag{1}$$

where N is the matrix size, $p = \sqrt{size}$, the square root of the total processes. N must always be divisible for p and p must be a perfect square. Given a certain PE, it has to receive the values from the PE above him and that on its left. It then computes the multiplication with these values. Each of the processes is identified with its own "rank", so that every PE has its own identification. The most important process is that of rank $= 0$.

Cannon's algorithm

The systolic array procedure can be obtained with the Cannon's algorithm [3]. Every process is assigned to a PE. The PEs are organized in a 2D periodic cartesian grid. First the input blocks are shifted among the rows and the columns, as requested by the Cannon algorithm's initialization. The shifting is obtained through MPI communication. Then, the process implement the computation as:

```
for (int k = 0; k < p; k++) {
        C = mul_matrix(A,B)
    }
```

The processes then pass the two input blocks to their neighbors (from left to right and from top to bottom). The periodicity of the 2D grid enables the implementation of a systolic array architecture without requiring additional operations.

MPI implementation

The communication among the PEs is implemented using MPI functions. The matrices A and B are divided between the processes through the function:

```
MPI_Scatterv()
```

To avoid deadlocks, the processing elements (PEs) are divided into even and odd groups. Finally, after the computations, they are gathered with the function:

```
MPI_Gatherv()
```

Finally, they are written in the matrix C.

C. Metrics

In order to evaluate the execution performance of the MPI implementation, multiple runs of the parallel program were performed, varying the number of processes and compute nodes to evaluate scalability, efficiency and speed up. The runs were executed on the Legion cluster, in which each node is equipped with 32 cores [4]. Every process can be associated to a core and the number of processes must respect the criteria seen before. So, for every matrix dimension, performances were evaluated for different number of nodes and processes:

- Matrix size 500×500:
  - 1 node: 4, 16, 25 tasks;
  - 2 nodes: 4, 16, 25 tasks.
- Matrix size 1000×1000:
  - 1 node: 4, 16, 25 tasks;
  - 2 nodes: 4, 16, 25, 64 tasks.
- Matrix size 2000×2000:
  - 1 node: 4, 16, 25 tasks;
  - 2 nodes: 4, 16, 25, 64 tasks.

The execution time was measured using the function

```
MPI_Wtime()
```

called right after the MPI initialization and right before the MPI finalization. This time is identified as $t_{parallel}$. The sequential time indicates the execution of the program made only by one single process, usually the task with $rank = 0$. This time, with the $t_{parallel}$ is useful to define the speed-up, given as $S = \frac{t_{sequential}}{t_{parallel}}$.

D. Results Analysis

Matrix size 500×500

For matrix size 500×500 the sequential time is $t_{sequential} = 3.40\ s$.

1 node:

Processes	Time (s)	Speed-up
4	0.28	12.14
16	0.15	22.67
25	0.15	22.67

TABLE I: Performance results with varying number of processes for matrix 500×500 and one node

2 nodes:

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

TABLE II: Performance results with varying number of processes for matrix 500×500 and two nodes

In this case, the highest speed-up values were obtained in the case of single-node. This because the communication latency and bandwidth constraints in single-node are significantly reduced compared to the multi-node executions. Within a single node, processes can communicate via shared memory or high-speed interconnections, which results in faster data exchange. In multi-node, communication must pass through network interfaces that introduce higher latency and lower bandwidth, increasing the overhead. This communication overhead can limit scalability and reduce the overall speedup. There is also a theoretical discrepancy between the implementation with 16 processes or 25 processes. Ideally, the execution with 25 processes should be the best one, but in both the single and multiple-node execution the performance improvement is negligible. The actual computation required for each PE is not so big and it could take less time than the overhead introduced by dividing the work among the processes and managing their communication. The best solutions are achieved with 25 or 16 processes for the single-node implementation and 16 processes for the multi-node implementation.

Matrix size 1000×1000

In this case the sequential time is $t_{sequential} = 4.88\ s$.

1 node:

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

TABLE III: Performance results with varying number of processes for matrix 1000×1000 and one nodes

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

TABLE IV: Performance results with varying number of processes for matrix 1000×1000 and two nodes

From the tables it is possible to observe a similar behavior with respect to the previous situation, in which the largest speed-up values are within the single-node MPI implementation. Because of the dimension of the matrix, a calculation with 64 processes in two nodes was tried. It is important to highlight that the worst speed-up in the multi-node implementation is obtained with 64 processes. This is linked to the overhead time due to the network implementation among the different processes, which becomes disproportionate to the actual computation time for matrices of this size. The best solutions are achieved with 25 processes for the single-node implementation and 16 processes for the multi-node implementation.

Matrix size 2000×2000

The sequential time is larger than the other case, as expected: $t_{sequential} = 40.38\ s$.

1 node:

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

TABLE V: Performance results with varying number of processes for matrix 2000×2000 and one node

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

TABLE VI: Performance results with varying number of processes for matrix 2000×2000 and two nodes

For this matrices size, it is possible to appreciate the double node 64 processors implementation. The computational workload of every task is large enough to justify the cost in time of splitting the work among the processes and managing their communication. However, the best solution for multi-node implementation is with 16 processes, while for single-node is 25 nodes.

E. Conclusions

The division of execution among a large number of processors does not necessarily lead to the best performance. Achieving optimal parallel efficiency requires balancing the computational load per process with the communication and synchronization overhead introduced by inter-process coordination. When the workload assigned to each process becomes too small, communication costs may dominate, resulting in reduced overall performance. A notable case is the use of 25 processes in a multi-node configuration. While this setup provides the best performance in single-node executions across all tested matrix sizes, it proves less efficient in multi-node scenarios. This inefficiency arises because 25 is not divisible by 2, leading to an unbalanced distribution of processes (13 on one node and 12 on the other). Such imbalance causes idle times, as one node must wait for the other to complete its tasks. Furthermore, increased communication between nodes over the network introduces additional latency and overhead, further limiting performance gains.

II. CUDA IMAGE FILTERING

A. Overview and Objectives

In many domains such as computer vision, astronomy, and neural network preprocessing, stencil-based image filtering plays a crucial role in enhancing image quality and removing unwanted noise. This section presents a GPU-accelerated implementation of a 3×3 stencil filter using NVIDIA CUDA. The filter, commonly used to perform smoothing and denoising operations, is applied independently to each color channel (R, G, B) of an input image. The goal is to assess the performance, scalability, and effectiveness of the CUDA kernel across multiple resolutions and image noise at 50%, 75% and 90%. The images used for testing include a set of high-definition photographs with and without noise, provided by the course instructor.

B. Methodology

In this implementation, the input images are loaded and processed in the RGB color space using OpenCV. Each color channel (Red, Green, Blue) is handled separately by splitting the image matrix into three distinct 2D arrays. This approach allows for independent filtering and manipulation of each channel.

CUDA Kernel Design

The core image filtering operation is implemented as a CUDA kernel designed to apply a weighted 3×3 convolution filter over each pixel, excluding the border pixels to avoid boundary issues. The kernel computes the weighted sum of the current pixel's neighborhood using a predefined weight matrix and writes the normalized result to the output image. The pseudocode is the following:

```
for each thread (row, col) in the image
    (excluding borders):
    sum = 0
    for i in [-1, 0, 1]:
        for j in [-1, 0, 1]:
            sum += weight[i+1][j+1] *
                input[row + i, col + j]
    output[row, col] = clamp(sum /
        normalization_factor, 0, 255)
```

The kernel employs thread indices computed from block and thread IDs to map each CUDA thread to a specific pixel position, ensuring parallel computation across the image.

To identify the optimal CUDA configuration, we perform a sweep of different thread block dimensions, including 16×16, 32×8 and 32×32, each configuration is evaluated in terms of execution time and computational efficiency.

To cover the entire image, the CUDA grid is sized based on the image dimensions and the block size.

For a block of 16×16 threads, the grid dimensions are computed as:

```
dim3 block(16, 16);
dim3 grid((COLS + block.x - 1) / block.x,
    (ROWS + block.y - 1) / block.y);
```

This formula ensures that the grid launches enough blocks to process all pixels by rounding up the division.

C. Metrics

To evaluate the parallel performance of our CUDA implementation, we adopt two standard performance metrics:

Speedup (S)

The ratio between the sequential CPU execution time and the parallel GPU execution time. It represents the performance gain achieved through GPU acceleration.

$$S = \frac{T_{sequential}}{T_{parallel}}$$

where:

$$T_{parallel} = T_{red} + T_{blue} + T_{green}$$

Throughput (TP)

Defined as the number of pixels processed per millisecond. Since the CUDA kernel is designed with one thread per pixel, the total number of threads equals the image size in pixels.

$$TP = \frac{\text{Image Size}}{\text{Execution Time}}$$

D. Results

Tables VII–X report the measured speed-up (S) and throughput (T, in MPix/ms) for different thread block configurations (16×16, 32×8, and 32×32) across images with varying noise levels (from none to 90%).

Overall, it is observed that the 16×16 and 32×8 configurations generally achieve better performance compared to the 32×32 configuration, which consistently shows lower performance across almost all tests. This behavior can be attributed to the computational overhead and reduced GPU resource efficiency associated with larger thread blocks, which may lead to diminished effective parallelism and increased latency.

We then present the detailed data tables, followed by a graphical analysis of execution times as a function of the number of threads for each configuration, limited to the noise-free images.

TABLE VII: Speed-up (S) and Throughput (T, MPix/ms) for Different Block Configurations (No Noise) with Image Dimensions

Image	Dimensions	16×16		32×8		32×32	
		S	T	S	T	S	T
climate_change.jpg	4435×3021	198.47	1.71	210.81	1.82	113.21	0.98
forests_cloudy_sky.jpg	3507×5264	480.98	4.15	482.06	4.15	353.04	3.05
harbor_bay_sunset.jpg	4897×3083	200.91	1.73	215.42	1.86	114.81	1.00
heart_shaped_island.jpg	5333×7999	403.91	3.48	468.37	4.04	257.13	2.22
Lake_forest.jpg	5739×3826	526.86	4.56	570.47	4.94	344.13	2.98
pexels_christian_heitz.jpg	21888×14592	200.10	1.72	320.06	2.75	155.76	1.34
Planet_earth.jpg	8154×4570	323.86	2.79	334.10	2.88	182.62	1.57
Winding_road.jpg	3840×2160	373.54	3.23	334.49	2.89	214.77	1.87

TABLE VIII: Speed-up (S) and Throughput (T, MPix/ms) for Different Block Configurations (Noise = 50%)

Image	Dimensions	16×16		32×8		32×32	
		S	T	S	T	S	T
climate_change.jpg	4435×3021	255.63	2.16	213.42	1.81	115.80	0.98
forests_cloudy_sky.jpg	3507×5264	338.58	2.92	355.96	3.07	202.26	1.75
harbor_bay_sunset.jpg	4897×3083	365.15	3.15	390.24	3.37	227.39	1.98
heart_shaped_island.jpg	5333×7999	270.27	2.31	234.19	2.00	119.50	1.02
Lake_forest.jpg	5739×3826	214.23	1.85	286.67	2.47	124.09	1.07
Planet_earth.jpg	8154×4570	216.10	1.86	231.01	1.98	118.72	1.02
Winding_road.jpg	3840×2160	438.36	3.79	446.51	3.86	322.15	2.80

TABLE IX: Speed-up (S) and Throughput (T, MPix/ms) for Different Block Configurations (Noise = 75%)

Image	Dimensions	16×16		32×8		32×32	
		S	T	S	T	S	T
climate_change.jpg	4435×3021	256.87	2.17	214.00	1.81	116.23	0.99
forests_cloudy_sky.jpg	3507×5264	478.84	4.13	476.72	4.11	351.88	3.04
harbor_bay_sunset.jpg	4897×3083	201.37	1.74	215.42	1.86	115.03	1.00
heart_shaped_island.jpg	5333×7999	270.12	2.31	294.64	2.52	152.95	1.31
Lake_forest.jpg	5739×3826	214.05	1.84	286.03	2.47	148.75	1.29
Planet_earth.jpg	8154×4570	275.73	2.37	289.72	2.49	151.91	1.31
Winding_road.jpg	3840×2160	184.41	1.58	306.96	2.62	191.32	1.65

TABLE X: Speed-up (S) and Throughput (T, MPix/ms) for Different Block Configurations (Noise = 90%)

Image	Dimensions	16×16		32×8		32×32	
		S	T	S	T	S	T
climate_change.jpg	4435×3021	256.04	2.17	214.29	1.81	148.32	1.26
forests_cloudy_sky.jpg	3507×5264	477.78	4.12	477.78	4.12	350.73	3.03
harbor_bay_sunset.jpg	4897×3083	478.26	4.12	214.90	1.86	343.08	2.98
heart_shaped_island.jpg	5333×7999	210.19	1.80	233.72	2.00	119.24	1.02
Lake_forest.jpg	5739×3826	349.73	3.01	335.08	2.89	183.12	1.58
Planet_earth.jpg	8154×4570	216.20	1.86	231.01	1.98	118.69	1.02
Winding_road.jpg	3840×2160	444.95	3.80	451.16	3.86	324.41	2.79

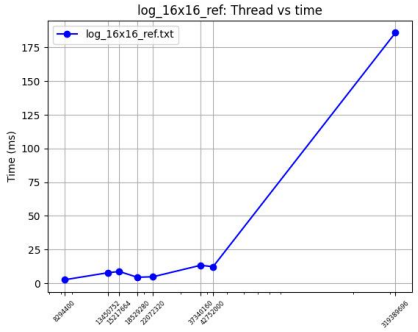


Fig. 1: Execution time vs. number of threads for the 16×16 block configuration.

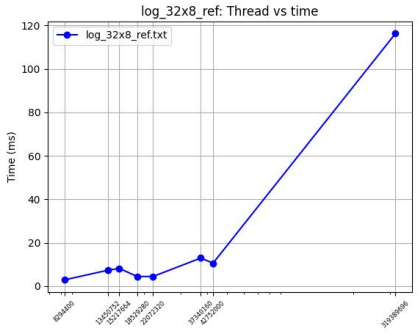


Fig. 2: Execution time vs. number of threads for the 32×8 block configuration.

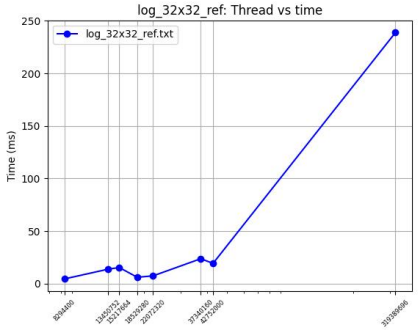


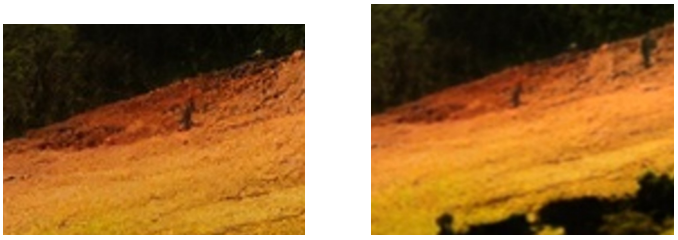
Fig. 3: Execution time vs. number of threads for the 32×32 block configuration.

Figures 1–3 depict the execution time as a function of the number of threads for the 16×16, 32×8, and 32×32 block configurations, respectively. Note that the number of threads corresponds directly to the image size in pixels. In general, the 32×32 configuration exhibits worse performance compared to the other two, likely due to higher overhead and less efficient occupancy. The 16×16 and 32×8 configurations show relatively similar trends, with the 32×8 slightly outperforming in specific cases—most notably for the *pexels-christian-heitz* image.

Results - Ultra–High–Resolution Image pexels-christian-heitz

Figure 4 presents a visual comparison, showing two zoomed-in regions (top and bottom rows) of the original high-resolution image alongside their filtered counterparts.

The filter visibly reduces fine-grain noise without significantly altering edges or larger visual structures. This effect is especially noticeable in textured regions like foliage and builidng. The bar chart in Figure 5 confirms the pattern



(a) Zoom 1 – Original (b) Zoom 1 – Filtered

(c) Zoom 2 – Original (d) Zoom 2 – Filtered

Fig. 4: Comparison between original and filtered zoomed-in regions of *pexels-christian-heitz*.

already observed in the Table VII: the 32×8 and 16×16 block configurations deliver markedly lower execution times than 32×32. For this ultra-high-resolution workload, larger blocks incur extra overhead. Consequently, 32×8 offers the best time-to-solution, closely followed by 16×16, while 32×32 remains the worst performer.

E. Analysis with Noisy Input

To assess the visual effectiveness of the 3×3 filter under different levels of corruption, we analyze *climate change.jpg* image (Image dimension = 4435x3021) with additive noise applied at 50%, 75%, and 90%.

Figure 6 presents three groups of image pairs: the left column contains the noisy input (Unfiltered), while the right

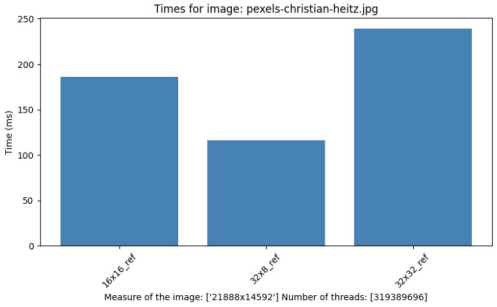
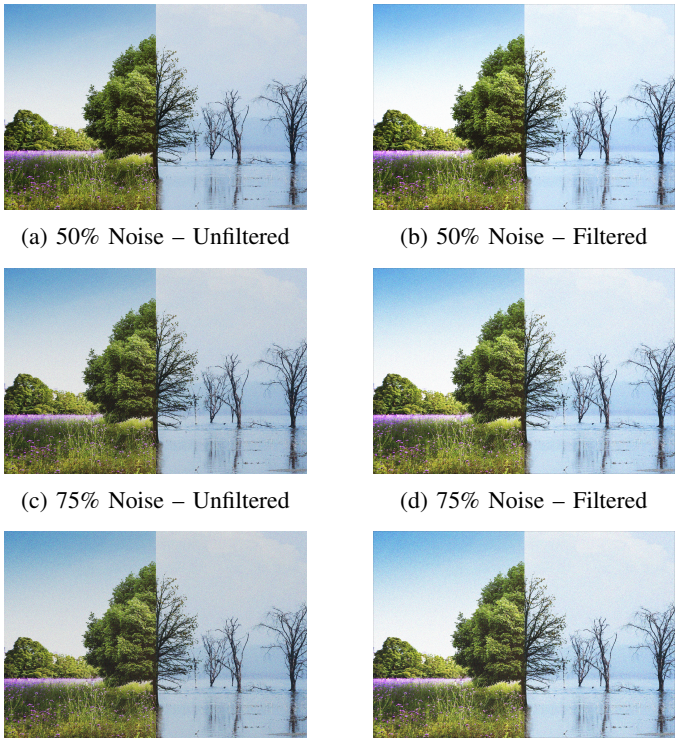


Fig. 5: Kernel execution time vs. block size for the *pexels-christian-heitz* image.

column shows the output after filtering (Filtered). Even at high corruption levels (90%), the filter manages to attenuate a substantial amount of noise while preserving most edge details and large structures. Figure 7 also reveals a significant increase in execution time for noisy images when using the 32×32 block configuration. Conversely, smaller block configurations (16×16 and 32×8) maintain more consistent performance.



(a) 50% Noise – Unfiltered (b) 50% Noise – Filtered

(c) 75% Noise – Unfiltered (d) 75% Noise – Filtered

(e) 90% Noise – Unfiltered (f) 90% Noise – Filtered

Fig. 6: Comparison between unfiltered and filtered image of *climate change.jpg* with varying levels of additive noise (50%, 75%, 90%).

F. Conclusions

The results demonstrate the filter's capability to effectively reduce noise introduced in images. From a hardware perspective, using block sizes of 16×16 or 32×8 is recommended due

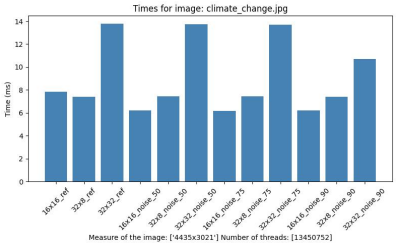


Fig. 7: Kernel execution time vs. block size and noise for the *climate change.jpg* image.

to their superior performance compared to 32×32 blocks. This is especially relevant when dealing with large-scale images or images affected by noise, where computational efficiency is critical.

III. Heat diffusion using OpenMP

The objective is to implement a program using OpenMP to simulate heat propagation in a metal plate under predefined initial conditions. The simulation models thermal diffusion according to specific physical laws, considering either isotropic or anisotropic behavior of the material.

A. Methodology

Isotropic Case

The heat diffusion simulation is parallelized using OpenMP in three main parts:

1) Grid initialization

```
#pragma omp parallel for private(i,j)
    schedule(static,10)
```

Each thread independently initializes a portion of the grid. The static schedule assigns blocks of rows to each thread, improving both speed and cache usage.

2) Stencil update

```
#pragma omp parallel for private(i,j)
    schedule(dynamic,4) reduction(max:max_diff)
```

Each internal cell is updated using the average of its four neighbors. The reduction(`max:max_diff`) ensures that the maximum temperature change is correctly computed in parallel. The dynamic schedule helps balance the workload as some areas converge faster than others.

3) Copy of updated values

```
#pragma omp parallel for private(i,j)
    schedule(static,10)
```

Updated values are copied back to the original grid. Since all cells can be copied independently, a static schedule is sufficient.

To avoid read-write conflicts during the update step, two matrices are used: one to read current values and one to store the updated values. This approach guarantees correctness and enables safe parallel execution. Boundary conditions are applied to prevent conflicts at the edges. The matrices are allocated as contiguous $N \times N$ blocks using `malloc`, which improves memory locality. The simulation stops when the maximum temperature change is below 0.01 °C or after 10,000 iterations.

Anisotropic Case

The program structure is identical to the isotropic version; the only change is the update rule applied to each internal cell:

```
matrix[i][j] = WY*(matrix[i-1][j] + matrix[i+1][j])
    + WX*(matrix[i][j-1] + matrix[i][j+1]);
```

Here WX = 0.3 weights horizontal conductivity, while WY = 0.2 weights vertical conductivity, modelling an anisotropic metal. The grid is initialised with a hot 512×512 square (25% of the area) centred on a cold background, and the convergence threshold is relaxed to 0.015 °C. A guided schedule, with chunk size equal to 10, is chosen for the stencil loop to better balance the workload as the hot region shrinks. All other aspects remain unchanged.

Chunk size

The chunk size was varied in the range of 4 to 20 for both isotropic and anisotropic cases using 24 cores and 24 threads. The results show that performance differences across chunk sizes in this range are negligible, indicating that the chunk size has minimal influence under these conditions.

B. Metrics

To evaluate the impact of parallelism on performance, a series of experiments was conducted on a single compute node of the Legion cluster.

For each test, the number of OpenMP threads was varied from 4 to 128. The number of allocated physical cores was varied too, ranging from 4 to 24. For each run the computational workload was kept constant. Execution time was measured using `omp_get_wtime()` at the beginning and end of the main simulation loop. Initial conditions and convergence parameters were kept consistent across all runs.

This setup enabled the analysis of how performance scales with the number of threads, and to observe the performance degradation due to oversubscription beyond 32 threads.

Cores vs. Threads

Throughout the experiments the job was executed on one Legion node. The number of cores refers to the physical cores reserved through SLURM (up to 24 on the chosen node), whereas the number of threads (`OMP_NUM_THREADS`) specifies how many OpenMP threads are launched. When threads exceed cores, the runtime must schedule multiple threads on the same core (multi-thread interleaving), which may introduce context-switching overhead and cache contention.

C. Results

Isotropic Case

To save space and avoid redundancy, we decided not to include all execution results for every configuration of reserved cores. Instead, we report the execution time as a function of

thread count for a representative case: the allocation of 24 cores.

The behaviour shown below is consistent across all tested core counts. Performance improves when increasing the number of threads up to the number of physical cores, then worsens when oversubscription occurs (i.e. more threads than cores).

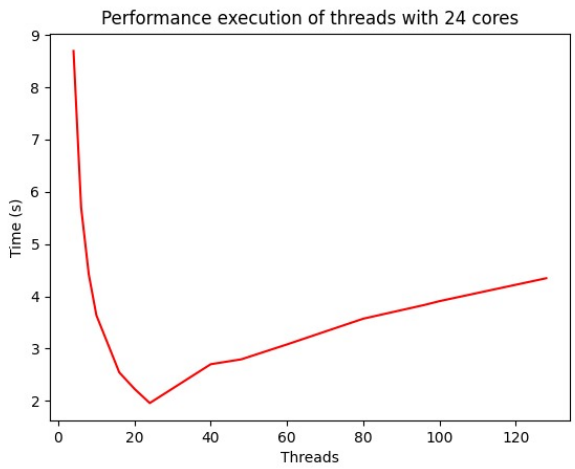


Fig. 8: Isotropic case: plot of the execution times as a function of the number of threads in the case of 24 allocated physical cores

Speedup: The sequential implementation required $T_{\text{seq}} = 32.108\,\text{s}$. Speed-up is defined as $S = T_{\text{seq}}/T_{\text{par}}$, where T_{par} is the execution time for the parallel implementation.

Observations

With 4 cores, the implementation achieves a speed-up of nearly 4×, with the best result observed at 10 threads (slight oversubscription).

TABLE XI: Best runtime and speed-up for each core reservation

Cores	Best #Threads	Time [s]	Speed-up
4	10	8.17	3.93
12	10	4.05	7.93
24	24	1.96	16.40

- With 12 cores, performance nearly doubles again, reaching a speed-up close to 8×. The best time is still observed with 10 threads.
- With 24 cores, the maximum speed-up reaches 16.4×, achieved using 24 threads.
- In all cases, using significantly more threads than reserved cores (e.g. $> 2\times$) leads to longer execution times, confirming the negative impact of oversubscription.

Overall, the parallel implementation demonstrates solid scalability up to one thread per core. Beyond that point, performance gains are limited due to memory contention and thread scheduling overhead.

Anisotropic Case

As for the isotropic case, the plot showing the trend of the execution time as a function of the number of threads is shown. The behaviour is approximately the same for the other cases.

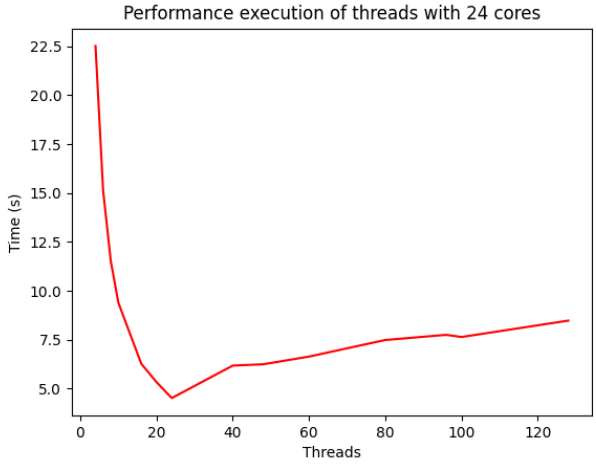


Fig. 9: Anisotropic case: plot of the execution times as a function of the number of threads in the case of 24 allocated physical cores

Speedup: In the anisotropic case, the sequential execution required $T_{\text{seq}} = 101.48\,\text{s}$. This value is significantly higher than the isotropic case due to the higher initial temperature and the different convergence behavior of the simulation.

TABLE XII: Best runtime and speed-up for the anisotropic case

Cores	Best #Threads	Time [s]	Speed-up
4	4	22.66	4.48
12	10	11.64	8.71
24	24	4.51	22.49

Observations

- With only 4 cores, speed-up is moderate ($\approx 4.5\times$) and performance degrades rapidly when using more threads due to oversubscription.
- Reserving 12 cores allows a substantial gain, almost doubling speed-up compared to the 4-core case. The optimal performance is again achieved with 10 threads.
- With 24 cores, the best result is obtained when threads match cores. The speed-up exceeds 22×, demonstrating excellent scalability.
- Overall, the performance behaviour closely mirrors that of the isotropic case, though the absolute times are longer due to the physical setup and tolerance.

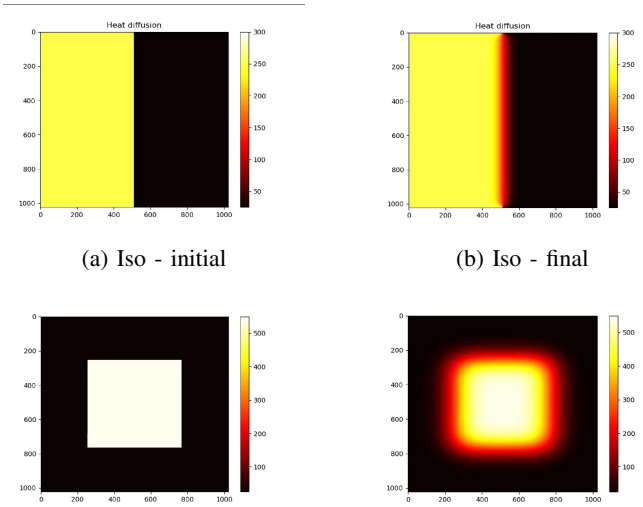
Comparison Between Isotropic and Anisotropic Cases

The two simulations differ in initial conditions and thermal conductivity. The anisotropic model introduces direction-

dependent weights (WX, WY) that alter the diffusion behaviour, requiring more iterations to converge.

- The anisotropic sequential time is more than three times that of the isotropic one (101.48 s vs. 32.108 s).
- Despite this, the anisotropic case benefits more from parallelism: with 24 cores, speed-up exceeds 22×, compared to 16.4× in the isotropic case.
- In both cases, oversubscription (threads $>$ cores) reduces performance, especially with many threads.
- The optimal number of threads remains close to the number of physical cores, with best performance generally observed with one thread per core or slightly less.

This comparison confirms that both implementations scale well on a single node, and the structure of the code—identical except for the update rule—exhibits consistent parallel behaviour.



(a) Iso - initial (b) Iso - final

(c) Aniso - initial (d) Aniso - final

Fig. 10: Temperature fields before and after convergence.

D. Conclusions

The experiments confirm that both the isotropic and anisotropic implementations benefit greatly from thread–level parallelism on a single Legion node. With one thread per core, the isotropic simulation achieves a 16.4× speed-up, while the anisotropic one reaches 22.5×.

Performance improves as more threads are used, up to the number of physical cores. This happens because the main operation (grid update) is very simple and repetitive, but it involves many memory accesses. As a result, after a certain point, adding more threads does not help, since the bottleneck moves towards the memory where many accesses are performed. Moreover, oversubscription and therefore the management of the distribution of the chunk among the data compensate the gain obtained from multi-threading.

- *Limitations*: Results refer to a single node and a fixed grid size (1024^2). Larger problems or multi-node execution may expose different bottlenecks (e.g. memory bandwidth or network latency).
- *Future Work*: A possible improvement could be to extend the code to a hybrid MPI+OpenMP model to run on multiple nodes, and/or porting the kernels to GPU with CUDA.

In conclusion, the OpenMP parallelisation delivers substantial speed-ups with minimal code complexity. Optimal performance is obtained with one thread per reserved core; increasing the thread count beyond that threshold is counterproductive on the tested architecture.

References

[1] IEEE, *IEEE templates*, 2024. [Online]. Available: https://www.ieee.org/conferences/publishing/templates.html

[2] Computational resources were provided by HPC@POLITO (http://hpc.polito.it)

[3] Lu, Hsin-Chen & Su, Liang-Ying & Huang, Shih-Hsu. (2024). Highly Fault-Tolerant Systolic-Array-Based Matrix Multiplication. Electronics. 13. 1780. 10.3390/electronics13091780.

[4] HPC documentation: https://www.hpc.polito.it/docs/guide-slurm-it.pdf

[5] E Sanchez & J. E. R. Condia, High Performance Computing (01HEKUU), course lectures, Politecnico di Torino, A.Y. 2024–2025.

[6] Github repo: https://github.com/Merlino2706/HPC_Assignment.git