HPC: Parallel Strategies for Matrix Multiplication, Image Processing, and Heat Diffusion Simulation

1st Filippo Marangoni
s346410@studenti.polito.it

2nd Denis Fabian Filip
s338446@studenti.polito.it

3rd Walid Bou Ezz
s336649@studenti.polito.it

***Abstract*—This report examines three parallel computing tasks: matrix multiplication using MPI systolic arrays, image noise reduction with CUDA, and heat diffusion simulation via OpenMP. Performance metrics like execution time and scalability are analyzed for each task, highlighting trade-offs between parallelism and efficiency. The results demonstrate how parallel strategies optimize large-scale computations in scientific applications.**

I. Introduction

High-performance computing (HPC) relies on parallelism to solve complex problems efficiently. This report explores three HPC applications: matrix multiplication (MPI), image filtering (CUDA), and heat diffusion (OpenMP). Each task evaluates performance, scalability, and optimization techniques. The findings provide insights into parallel computing's role in accelerating scientific simulations and data processing.

II. Problem 1: Systolic Array Structures for Matrix Multiplication

A. Architectural Foundation

The implementation adopts a systolic array architecture for parallel matrix multiplication using MPI. This approach organizes processes into a logical two-dimensional grid of size $\sqrt{P} \times \sqrt{P}$, where P represents the total number of MPI processes. The matrices $\mathbf{A}$ and $\mathbf{B}$ undergo decomposition into $\sqrt{P} \times \sqrt{P}$ smaller blocks, each of dimensions $bs \times bs$, with bs calculated as $N/\sqrt{P}$. This decomposition strategy ensures that each process becomes responsible for computing a specific block of the resulting matrix $\mathbf{C}$ through coordinated data movements and localized computations.

B. Initialization and Matrix Preparation

The computation commences with rank 0 assuming responsibility for matrix discovery and initial data preparation. Through the `detect_matrix_size` function, it parses the input CSV files to determine matrix dimensions, while simultaneously verifying that N is divisible by $\sqrt{P}$ to ensure proper block alignment. Once validated, rank 0 reads the complete matrices $\mathbf{A}$ and $\mathbf{B}$ into memory using `read_full_matrix`. It then constructs specialized communication buffers: `A_rowbuf` captures $\sqrt{P}$ horizontal tiles from $\mathbf{A}$'s leftmost column for each process row, while `B_colbuf` assembles $\sqrt{P}$ vertical tiles from $\mathbf{B}$'s topmost row for each process column. These buffers serve as reservoirs that will feed data into the systolic array during computation.

C. Data Distribution Strategy

The boundary processes receive their initial data payloads through targeted MPI communication. Column 0 processes obtain their portion of `A_rowbuf` via `MPI_Recv` operations tagged with identifiers (10 + row number), while row 0 processes acquire their `B_colbuf` segments through similarly tagged messages (20 + column number). This selective distribution ensures that processes positioned at the grid's western and northern edges receive their initial data blocks without requiring direct access to the full matrices. The remaining processes initialize their local `Ablk` and `Bblk` buffers to zero, setting the stage for the systolic computation pipeline.

D. Systolic Computation Pipeline

The core computation unfolds through $3\sqrt{P} - 2$ meticulously synchronized pipeline stages. During each stage, data undergoes directional propagation: $\mathbf{A}$ blocks shift eastward (left to right) while $\mathbf{B}$ blocks migrate southward (top to bottom). Processes employ non-blocking `MPI_Isend` operations to transmit their current blocks to eastern and southern neighbors, simultaneously initiating `MPI_Irecv` calls to receive incoming blocks from western and northern neighbors. This dual communication strategy enables overlapping of data transfer and computation. Boundary processes inject fresh tiles from their pre-distributed buffers when the communication pattern would otherwise extend beyond grid edges, maintaining continuous data flow.

Upon receiving new tiles, processes promptly copy them into local buffers `Ablk` and `Bblk` using efficient `memcpy` operations. The computational kernel `dgemm_tile` then activates, executing the core matrix multiplication through its triple-nested loop structure. Crucially, valid computations occur only when the current stage index s and process coordinates satisfy the condition $k = s - my\_row - my\_col$, creating a wavefront of computational activity that progresses diagonally across the process grid. This intricate coordination ensures that each tile multiplication occurs precisely when the required data elements converge at the appropriate process.

E. Result Aggregation and Output

As the pipeline completes its final stage, the solution fragments distributed across the grid require reassembly. Rank 0 initiates the gathering process through a series of `MPI_Recv` operations (tagged 200) that collect $\mathbf{C}$ blocks from all processes. Non-root processes transmit their local `Cblk` buffers

using corresponding `MPI_Send` operations. The master process then reconstructs the complete $\mathbf{C}$ matrix in row-major order before writing it to the specified output CSV file via `write_full_matrix`. This function ensures precision by formatting values with ten decimal digits while maintaining CSV structure.

F. Performance Instrumentation

Beyond computational results, the implementation captures detailed performance metrics. Execution time measurement employs `MPI_Wtime`, with local measurements synchronized through `MPI_Barrier` and the maximum time across processes determined via `MPI_Reduce`. Resource utilization metrics, including peak resident memory (`ru_maxrss`) and CPU time, are collected using `getrusage`. Rank 0 consolidates these measurements into a concise performance summary formatted as `N=<size> P=<procs> time=<sec> cpu=<sec> memKB=<kB>`, which it appends to a user-specified statistics file. This instrumentation provides quantitative insights into the implementation's efficiency across varying problem scales and process counts.

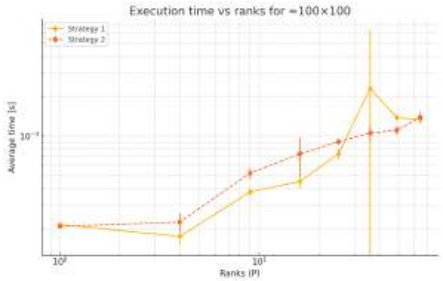
G. Performance Analysis of the MPI Systolic-Array Code

1) Experimental Methodology: Two deployment strategies were benchmarked. *Strategy 1* ("fill& spill") packs as many ranks as possible on the first node (up to 49 ranks on the `cpu_sapphire` partition) before a second node is allocated; *Strategy 2* always distributes the same number of ranks evenly across **two** nodes, forcing inter-socket traffic from the outset. For each strategy the matrix dimension N is adjusted on launch so that $N/\sqrt{P}$ is an integer (`matrix_generator` selects the nearest multiple of the tile size), guaranteeing that the $\sqrt{P}\times\sqrt{P}$ process grid receives perfectly square sub-blocks.

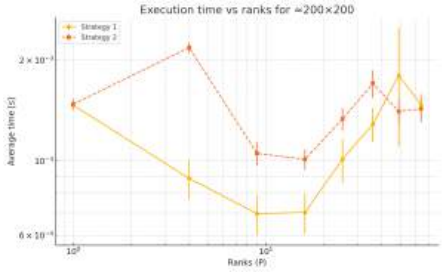
Measurements were automated through the Slurm script `matrix_multiplication.sbatch`. The header requests the `cpu_sapphire` partition, a 30-minute wall-clock limit and at most `--nodes=2`, `--ntasks=90`. Open-MPI 4.1.8 is loaded and its TCP BTL is pinned via `OMPI_MCA_pml=ob1` and `OMPI_MCA_btl=self,tcp` to bypass the cluster's default InfiniBand stack, ensuring comparable behaviour across CPU nodes. Three base sizes ($N \approx 100, 200, 500$) and six square rank counts ($P = 1, 4, 9, 16, 25, 36$) are swept; the innermost loop repeats each experiment ten times, appending a compact line `N=<size>P=<procs>time=<s>cpu=<s>memKB=<kB>` to an aggregate file. A helper `run_case` function invokes `srun--mpi=pmix` so that Slurm's PMI layer handles rank placement consistently under both strategies. All statistics analysed below therefore derive from identical source code, library versions and runtime options, differing only in the rank-to-node mapping.

2) Results and Discussion: Figure 1 plots the mean execution time (log–log scale) for each base size; error bars indicate one standard deviation over the ten repetitions.

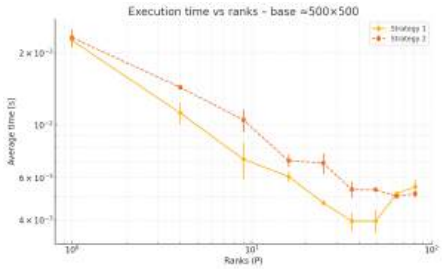
In the graphs you can see two curves compared with one another. At size 100 we can see that as soon the program



(a) $N \approx 100$



(b) $N \approx 200$



(c) $N \approx 500$

Fig. 1: Execution time vs. ranks for the two strategies.

reaches 4 threads we do not obtain an improvement in time due to the small size of the problem. The 200-element case (Fig. 1b) shows the same qualitative trend, but the knee shifts to $P = 9$—the point where each rank holds only $\approx (200/\sqrt{P})^2 = 4500$ elements and communication begins to dominate. Only when the workload grows to $N \approx 500$ (Fig. 1c) does arithmetic density outweigh network cost: Strategy 1 still outperforms its two-node counterpart, yet both strategies continue to speed up up to $P = 64$ (Strategy 1) or $P = 81$ (Strategy 2), delivering a five-fold and four-fold reduction in wall-time respectively.

The vertical bars highlight that run-to-run variability stays below $15\,\%$ for all but one configuration ($N \approx 100$, $P = 49$), confirming that ten samples suffice for stable means at these problem sizes. Scaling falls short of the linear ideal primarily because message volume grows with $\sqrt{P}$ while useful work per rank shrinks as $1/P$; consequently the *parallel efficiency* drops from $50\,\%$ at $P = 4$ to below $10\,\%$ beyond $P = 25$ on the 500-element matrix—even under the more communication-savvy Strategy 1. Additional nodes would therefore be justified only for matrices larger than those tested or for systems equipped with true low-latency fabrics (e.g.

InfiniBand HDR).

In summary, the data confirm that confining ranks to a single node whenever possible maximises memory-bandwidth utilisation and postpones the communication bottleneck. Nevertheless, at larger N the inherent $\mathcal{O}(N^3)$ arithmetic dominates, so the balanced two-node strategy narrows the gap and eventually approaches the single-node baseline.

III. Problem 2: Fundamentals of Multi-Dimensional Data Processing

Introduction This exercise focuses on using CUDA to apply a stencil filter on high-resolution images (4K, 8K, and 16K). CUDA allows us to run image processing tasks in parallel on the GPU, which can greatly improve performance. We will apply the filter to remove noise from the images and measure how different image sizes and thread configurations affect execution speed. This helps us understand how GPU parallelism can be used for efficient image processing.

Definition and Mathematical Formulation The Gaussian blur smooths an image using:

$$g_{(x,y)} = \frac{1}{16} \sum_{j=-1}^{1} \sum_{i=-1}^{1} w_{(i,j)} \cdot f_{(x+i,y+j)}$$

Where f is the input image and W is the stencil/filter matrix

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 3 & 4 & 3 \\ 1 & 2 & 1 \end{bmatrix}$$

Boundary Handling: Mirroring We decided to use Mirroring as our choice for boundary handling.

Mathematical Formulation

For an image of width N:

$$I_{\text{mirrored}}(x) = \begin{cases} I(|x|) & \text{if } x < 0 \\ I(2N - 2 - x) & \text{if } x \geq N \end{cases}$$

Visual Example

For pixels `[A B C D]`, mirrored extension:
`B A B C D C B`

Salt-and-Pepper Noise Analysis

- **Noise model**: Discrete impulse noise where corrupted pixels take either minimum (pepper, $I = 0$) or maximum (salt, $I = 255$) intensity values:

$$P(I_{noisy}) = \begin{cases} p_{pepper} & \text{if } I = 0 \\ p_{salt} & \text{if } I = 255 \\ 1 - p_{pepper} - p_{salt} & \text{otherwise} \end{cases}$$

Gaussian Blur's Effect on Noise

Noise Reduction Mechanism

This blur process makes the image look softer by reducing crisp edges and small details, thus partially removing the effect of the S&P Noise. As shown in Fig. 2

(a) Original image (b) After 90% S&P

(c) After 90% S&P + Gaussian blur

Fig. 2: Quantitative comparison of noise reduction

Limitations

- **Edge degradation**: Blurring reduces sharpness measured by gradient magnitude:
- **Residual noise**: Blurring creates gray "halos" around salt/pepper pixels

CUDA Implementation Details

Hierarchy Overview

- **Grid Organization**:
  - Calculated as $\lceil \frac{\text{width}}{\text{blockSize}} \rceil \times \lceil \frac{\text{height}}{\text{blockSize}} \rceil$ blocks per grid
  - Example: For 1024×768 image with blockSize=32:
    - $\lceil \frac{1024}{32} \rceil = 32$ blocks horizontally
    - $\lceil \frac{768}{32} \rceil = 24$ blocks vertically
    - Total grid dimension: 32×24 blocks
- **Thread Block Configuration**:
  - Base configuration: blockSize $\times$ blockSize threads per block
  - Extended configuration (for CHANNEL_THREAD version): blockSize $\times$ blockSize $\times$ 3 threads for RGB processing
  - Experimental block sizes tested: [4, 8, 16, 32]
- **Memory Hierarchy**:
  - Global memory for image I/O
  - Constant memory for Gaussian kernel weights
  - Shared memory for halo exchange optimization

Halo Exchange Mechanism

In our implementation, we employ a halo region strategy to efficiently handle boundary conditions during the Gaussian blur convolution. We allocate shared memory with dimensions $(BLOCK\_SIZE + 2) \times (BLOCK\_SIZE + 2) \times 3$ bytes, creating a one-pixel border around each block to store neighboring pixels. To handle image boundaries, we implement a mirrored padding technique through our `mirror` device function, which reflects out-of-bound indices back into valid image coordinates.

We populate the shared memory tile by having each thread load its corresponding pixel along with surrounding halo pixels. While this approach correctly processes the image boundaries, we recognize that it redundantly loads halo pixels across threads. We ensure synchronization before computation using `__syncthreads()`, guaranteeing all data is properly loaded before any thread begins processing.

Code Analysis

In this task, we first implemented a Gaussian blur using a halo region. After that, we improved the code by using threads to handle each color channel separately. This helps us measure the effect of each improvement.

The first version of the code, called `halo.cu`, has the following main steps:

- Allocate memory on the GPU for the input and output images.
- Copy the input image from the CPU (host) to the GPU (device) using `cudaMemcpy`.
- Run the kernel function called `applyGaussianBlur`.
- Copy the output image from the GPU back to the CPU.

The **kernel** is the function that runs in parallel on many GPU threads.

Inside the kernel, we store the image pixels from the block in a shared memory area called `shared_tile`.

Next, we use `__syncthreads()` to wait until all threads have finished copying the pixels. This is important to make sure all data is ready before we start the calculation.

Finally, we apply the Gaussian blur filter (stencil matrix) and save the result to the output image. (calculated on all channels of the pixel)

Second Version: `channel_thread.cu`

In the second version of the code, called `channel_thread.cu`, most steps are the same. However, now we use three times more threads. Each thread is responsible for calculating the blur of only one color channel (red, green, or blue) of a single pixel.

This version increases parallelism and helps us process the image faster using the GPU.

Profiling and Optimization

We tried three main methods for profiling:

1) **CUDA Events:** We use CUDA events to measure how long a kernel takes to run. We do this using `cudaEventRecord` and `cudaEventElapsedTime`. This gives the time in milliseconds.
2) **nsys:** `nsys` is a tool from NVIDIA that gives detailed information about the program. It shows when kernels start and stop, memory copies, and GPU usage. It helps us see how the program runs on the GPU and CPU together.
3) **ncu:** `ncu` (NVIDIA Compute Utility) is another tool from NVIDIA. It gives deep information about kernel performance, such as memory usage, cache usage, and occupancy. However, it needs `sudo` (administrator) permissions to run.

We did not have `sudo` access, so we could not use `ncu`. Because of this, we used CUDA events to measure the duration of kernel execution.

For `nsys`, we used two reports:

- **gputrace:** This report shows a timeline of GPU activity. It includes kernel launch times, memory copy events, and synchronization. It helps us see the order and duration of GPU tasks.
- **gpumemsizesum:** This report shows how much GPU memory is used by the program. It gives a summary of memory allocations and sizes, which helps us understand memory usage.

Block Size Constraints in CUDA

In CUDA, the number of threads in a block has a limit. The maximum number of threads allowed per block is **1024**.

Because of this rule:

- For `halo.cu`, the maximum block size we can use is `[32 × 32] = 1024` threads.
- For `channel_thread.cu`, each pixel needs 3 threads (one for each color channel: red, green, blue). So the maximum block size becomes `[16 × 16 × 3] = 768` threads. If we go higher, we exceed the limit.

Parallelization Results - Variants Experiment

We tested the runtime of both code versions using different block sizes. The goal was to see how the `channel_thread.cu` version compares to the original `halo.cu`.

We used a shell script called `measurement_performance.sh` (available in the project's GitHub repository) to test the performance. This script runs both kernel versions on four different image sizes:

- 4K
- 8K
- 16K
- 32K

We tested (Fig. 3) block sizes of 4, 8, 16, and 32. For each run, we measured the kernel execution time using `cudaEventElapsedTime`.

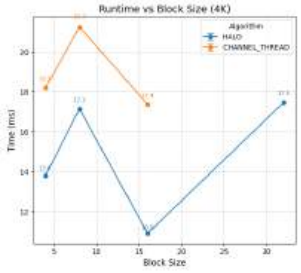
We also used `nsys` to measure the duration of memory operations:

- `cudaMemcpy HtoD` (Host to Device)
- `cudaMemcpy DtoH` (Device to Host)

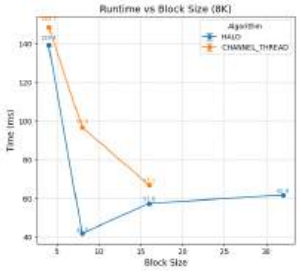
The results (Fig. 4) showed that memory copy time (HtoD and DtoH) does not change with block size or algorithm. It only depends on image resolution. Larger images take more time to copy.

Results Summary:

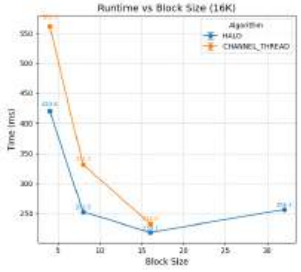
- **HALO**
  - Performs better with small block sizes.
  - At larger block sizes, performance becomes flat or worse due to memory inefficiencies.
- **CHANNEL_THREAD**
  - Slower overall, especially for small images.



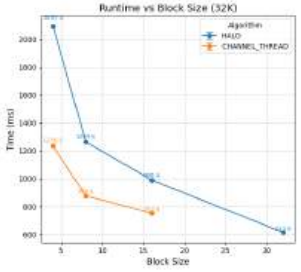
(a) 4K Image



(b) 8K Image

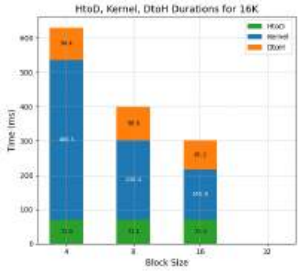


(c) 16K Image



(d) 32K Image

Fig. 3: Kernel execution time vs block size for both `halo.cu` and `channel_thread.cu` on different image sizes.



(a) 16K Image

(b) 32K Image

Fig. 4: HtoD, Kernel, DtoH vs block size for `channel_thread.cu` on 16K image memory transfer size.

– Scales better on large images due to increased thread usage.

To reduce total run time, it may also help to parallelize memory allocation functions.

Parallelization Results - Statistical Experiments

We also performed two statistical experiments to better understand how block size affects performance and speedup under different conditions.

1. Trade-off Between Parallelism and Performance (pexels-christian-heitz.jpg)

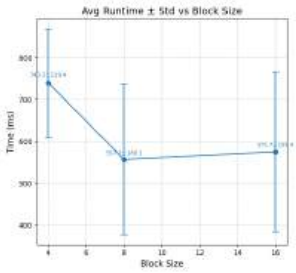
In this experiment, we analyzed how block size affects the runtime of our algorithm when using the image `pexels-christian-heitz.jpg`.

We measured:

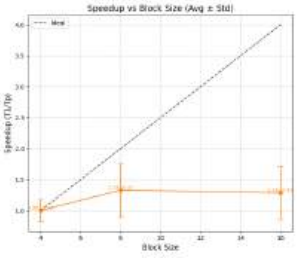
- Average kernel runtime with standard deviation.
- Speedup compared to a baseline block size.

Observation:

- Block size **8 × 8** gives the best performance. It balances the GPU overhead and the benefits of parallelism.
- Very small block sizes (like **4 × 4**) create too much overhead because many kernel launches are needed.
- Very large block sizes lead to under-utilization of threads and shared memory.



(a) Average Runtime ± Std Dev vs Block Size



(b) Speedup vs Block Size

Fig. 5: Statistical analysis for different block sizes using `pexels-christian-heitz.jpg`.

2. Speedup Across Multiple Image Resolutions

In this experiment (Fig. 6), we tested how the block size affects performance across images with different resolutions. We again measured speedup versus block size and looked for stable configurations.

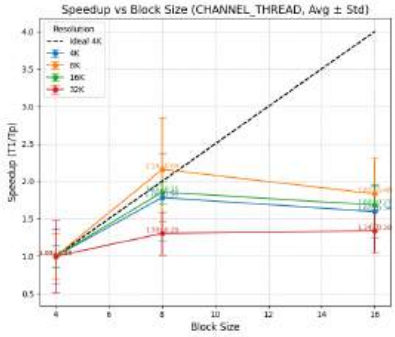


Fig. 6: Speedup vs block size for multiple image resolutions.

Block Size Usage for different resolutions:

- **4 × 4:** Highly unstable. Too many kernel launches cause overhead. Should be avoided for large images.
- **8 × 8:** The best balance of speed and stability. Works well for all image sizes.
- **16 × 16:** Works well for large images, but may show performance changes under heavy GPU load or cache issues.

Conclusion

Through our experiments, we learned how different block sizes affect GPU performance and how to choose the right configuration for better results.

A very small block size (such as `4 × 4`) creates high overhead. This happens because many small blocks need to be launched, and each launch adds extra work. It also increases memory accesses and reduces efficiency.

A very large block size can also be a problem. It may lower parallel efficiency because the GPU can launch fewer blocks. This means not all GPU cores are used well, and some threads may stay idle.

From our results, we found that a block size of **8 × 8** gives a good balance. It reduces overhead while still keeping high GPU usage. This size worked well across different image resolutions.

However, the best block size may change depending on:

- The resolution of the image.
- The type of GPU hardware used.
- How the algorithm is written.

Other Considerations (not tested in this work):

- Improving Host-to-Device (HtoD) and Device-to-Host (DtoH) memory transfer times — for example, by parallelizing them — could reduce total runtime. But this would not change kernel execution time.
- For very large images that do **not fit in GPU memory**, a possible solution is to split the image into smaller parts (slices) and process them one by one. However, this increases the total kernel time because the kernel must run separately for each slice.

IV. Problem 3: Heat Diffusion Simulation in 2D Grid

The implementation of this exercise consists of a simple finite-difference solver using a 2D grid. Each cell is updated iteratively based on its neighbours using either an isotropic or an anisotropic stencil. The process iterates until the maximum temperature change across the grid falls below a specified convergence threshold or a maximum number of iterations is reached. After developing the serial version, the simulation is parallelized using OpenMP to assess scalability with varying numbers of threads. Performance is evaluated in terms of execution time and temperature evolution for up to 10,000 iterations, sampling every 100–200 iterations to analyse heat diffusion behaviour.

The complete source code and Slurm job scripts are available in the the project's GitHub repository.

Main idea of the Serial Code. First, the program initializes all parameters, including the temperature values for each region and the weights used for the anisotropic diffusion model. The `init_plate` function sets the initial grid configuration (temperatures) based on the selected mode (isotropic or anisotropic), which is passed as a command-line argument.

The main function parses the input arguments and allocates the main grid, and a temporary grid used for iterative updates. At each iteration, the new temperature of each cell is computed using its neighbours according to the selected diffusion model. Auxiliary variables help define neighbouring cells and handle boundary conditions. During each step, the maximum temperature difference is tracked to check for convergence. After updating, the grids are swapped for the next iteration. The simulation stops when the maximum difference falls below a defined threshold or when the maximum number of iterations is reached. Optionally, the grid can be saved in binary format for further analysis.

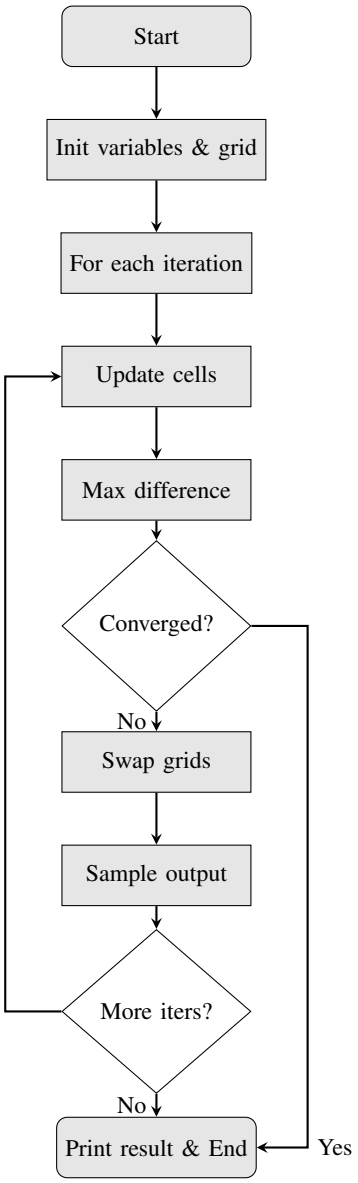


Fig. 7: Compact flow chart of the serial heat diffusion simulation.

Parallelization Strategy. The update loop is parallelized using `#pragma omp parallel for collapse(2)` to flatten the two nested loops into a single iteration space of size N^2. This improves load balancing across threads, which is crucial for large grids and high thread counts. A static schedule is used, since the workload per cell is uniform, using dynamic or guided scheduling would add unnecessary overhead in

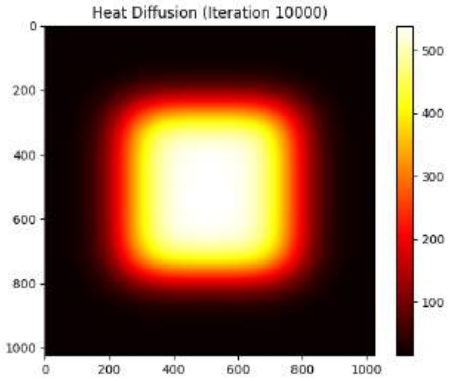


Fig. 8: Snapshot of the 2D heat diffusion simulation (Mode 1 shown). Full animations for both isotropic (Mode 0) and anisotropic (Mode 1) cases are available at our GitHub repository.

this context. A `reduction(max: max_difference)` safely computes the global maximum temperature change.The auxiliary grid `next` is swapped with the actual grid outside the parallel region to avoid unnecessary synchronization among threads. All these choices are well suited to stencil computations, which have no data dependencies within an iteration. This makes domain decomposition straightforward. Finally, only the master thread executes the `printf` statement for iteration updates, thanks to the OpenMP `#pragma omp master` directive. This prevents duplicated output, race conditions, or mixed text in the output. Since `max_difference` is already computed with a reduction, the final value is guaranteed to be correct and global, so only one thread needs to print it (the master).

Interlude. OpenMP is a shared-memory parallel programming model. This means it can only exploit the cores within a single compute node — or more precisely, within a single shared-memory domain. It does not support communication across nodes by itself; if inter-node scaling is needed, a hybrid approach combining MPI and OpenMP is typically used. In this exercise, the focus is solely on `intranode` scalability, so before running the parallel experiments, we verified the number of physical cores available on the target node (`cpu_sapphire`). To do this, we submitted a simple Slurm script (named `cpu_check.sh`) which used the lscpu command and the relevant Slurm environment variables to report CPU and task allocation details. Based on this information, the maximum number of OpenMP threads was set equal to the total number of cores per node, ensuring the experiments fully exploited the node's shared-memory resources.

Experimental Setup. Before execution, the node's core configuration was verified using `lscpu`. The target node provides 48 physical cores and supports hyper-threading, resulting in 96 logical CPUs. The Slurm file was configured to test thread counts from 1 up to 96 to analyze intra-node scalability.

Each configuration was executed five times per mode, measuring the execution time with `omp_get_wtime()`. The average was used to limit random fluctuations. The CPU usage factor was estimated using the Linux `time` command, defined as the ratio of total CPU time to elapsed real time. This factor indicates how fully the cores were kept busy.

Scalability and Speedup.

The observed variations between runs were minimal, indicating that during our tests the HPC cluster provided stable and consistent conditions.

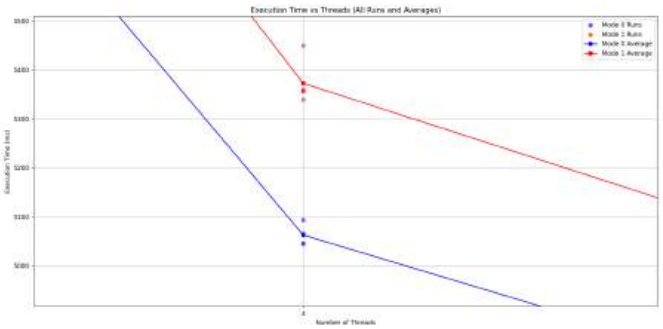


Fig. 9: Random variations observed in the case of 4 threads.

Figure 10 shows that, at first, the execution time decreases significantly as the number of threads increases, but then the improvement gradually saturates. (Analogously for mode 1).

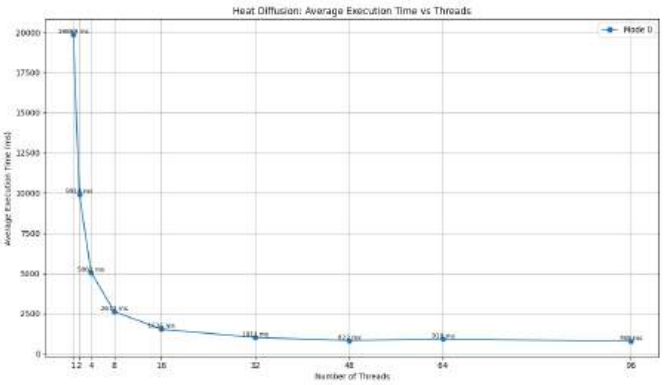


Fig. 10: Average execution time vs number of threads for mode a.

This trend becomes clearer when plotting the speedup, defined as $S_p = T_1/T_p$, where T1 is the execution time with one thread and Tp is the execution time with p. The speedup is shown in Figure 11.

Ideally, the speedup doubles every time the number of threads doubles. In practice, this behavior is visible up to about 32 threads. Beyond this point, the speedup continues to increase but with diminishing returns. For example, at 64 threads there is even a slight drop in speedup.

This deviation from ideal scaling is due to several factors:

- Increased parallel overhead (e.g., more threads competing for shared memory bandwidth.

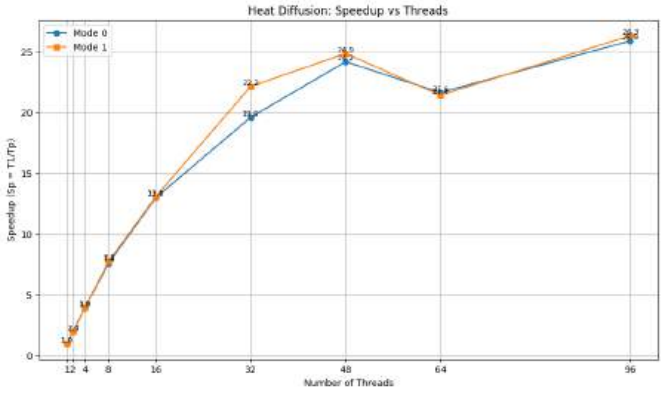


Fig. 11: Speedup vs number of threads for the two modes.

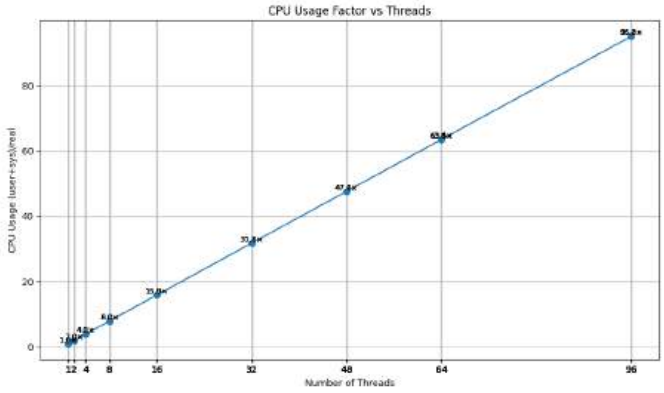


Fig. 12: CPU usage factor vs number of threads.

- Synchronization costs, which grow with the number of threads.
- Diminishing workload per thread: when the problem size is fixed, adding more threads means each thread does less work and so, overhead starts to dominate.
- NUMA effects: on multi socket nodes, cores may compete for memory.

Such effects are typical in shared-memory parallelism and highlight the practical limit of intra-node scalability for this problem size.

The drop in speedup beyond 48 threads is due to the transition from physical to logical cores: hyper-threading does not provide the same performance gains for memory-bound workloads such as stencil computations and may even introduce slight contention for bandwidth inside the same core and so performance can drop or flatten.

CPU Usage Factor, Parallel efficiency and Larger Grids.

To better understand resource usage, the Linux `time` command was used to estimate the CPU usage factor, which is defined as the ratio of total CPU time (`user + sys`) to the elapsed real (wall-clock) time. This factor indicates how fully the allocated cores were kept busy during execution. By comparing the CPU usage factor with the measured speedup, it became clear that all threads were indeed active: the usage factor closely matched the number of threads used. However, the speedup did not increase proportionally at higher thread counts. This demonstrates that the cores were busy, but parallel overheads — such as synchronization and memory bandwidth contention — limited performance gains.

This observation suggested that increasing the grid size would increase the computational workload per thread, thereby reducing the relative impact of overheads and improving parallel efficiency. Parallel efficiency is defined as the ratio of the achieved speedup to the ideal speedup, given by:

$E_p = S_p/p$.

To test this, the grid size was doubled to 2048×2048, and the experiments were repeated. The results confirmed this intuition: the speedup for the larger grid was consistently higher than for the smaller one, and the parallel efficiency improved as well. This is because each thread has more useful work to do, and so this reduces synchronization costs and the impact of memory bandwidth limits.

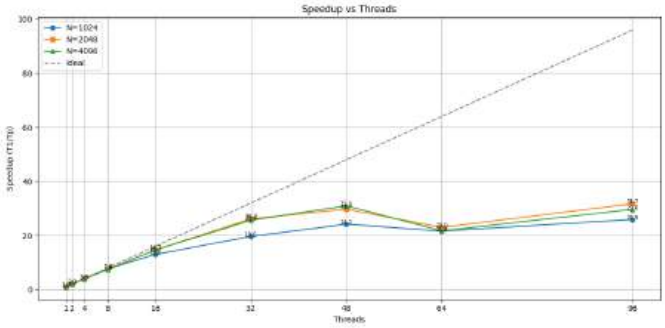


Fig. 13: Speedup vs number of threads for different grid sizes.

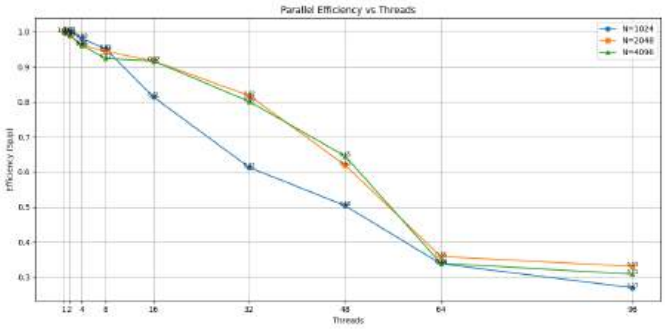


Fig. 14: Parallel efficiency vs number of threads for different grid sizes.

A practical balance must be found: too small a grid underutilizes the cores; too large a grid achieves still better scaling but at the cost of prohibitive run times. The 2048×2048 case represent a good compromise for this HPC context.

Conclusion. The experiments demonstrate that for this stencil-based heat diffusion simulation, increasing the problem size improves parallel scalability by raising the computation-to-

overhead ratio. At the same time, the results clearly highlight the practical limits of a pure shared-memory approach.