HPC Assignment Report

1st Daniele Spalazzi
Quantum Engineering Master Degree
Politecnico of Turin
Turin, Italy
s346877@studenti.polito.it

2nd Simone Sticca
Quantum Engineering Master Degree
Politecnico of Turin
Turin, Italy
s347328@studenti.polito.it

3rd Enrico Bozzetto
Quantum Engineering Master Degree
Politecnico of Turin
Turin, Italy
s339473@studenti.polito.it

I. Introduction

This report presents solution and experimental results of the assignments for the 2024/2025 edition of the course on High Performance Computing of the Master's Degree program in Quantum Engineering at the Politecnico of Turin. The assignment comprises three different exercises which require the use of three different parallelization techniques and strategies to be solved.

The first exercise consists in performing matrix multiplications as systolic arrays. The operation indeed can be organized as a set of fixed operations in independent processing elements connected between them and this structure is known as a systolic array. The problem requires to develop, using MPI, the structure to compute the matrix multiplication of two input square matrices of different dimensions and then analyzing the performances with the different inputs and changing the number of threads.

The second exercise concerns the multi-dimensional data processing. This is an operation which is used in several fields; in particular, in 2-/n-dimensional applications stencils/filters are applied to all input elements of an image to calculate or process features. The problem requires to use CUDA to preprocess and apply a specific type of stencil/filter on any 4K,8K or 16K input image and then analyzing the performances with the different inputs and changing the number of threads.

The third exercise concerns the simulation of heat diffusion in a 2D grid. This simulation can be performed in several ways and one of them is the finite difference method which approximate derivatives with difference equations. The problem requires to analyze the heat diffusion in a metal plate represented by a 1024x1024 grid with two different initial temperature conditions and laws of propagation (case A and case B), using OpenMP. Then, as with the other problems, the performance analysis as the number of threads varies is requested.

II. First exercise

A. Introduction on the exercise

Systolic arrays are a class of architectures designed to efficiently implement algorithms such as large general matrix multiplication, where no sparse matrix consideration can be done. The rapid growth of artificial intelligence models has increased the interest in this kind of architecture. This evident if we look at the increase in the parameter size of Deep Neural Networks (DNN) and, in particular, large language models like ChatGPT. The dimensions of the input and weight matrices to model the artificial neuron behavior for GPT-3.5 are up to 175 billion matrix elements.

The classic Von Neumann architecture with central processing units and memory does not scale well for matrix multiplication problems. The bottleneck of this architecture is memory due to its latency. In matrix multiplication, the number of memory accesses is very large because of the high repetitiveness of the processes with different data, and memory latency is therefore a crucial aspect. For this reason, in artificial intelligence problems, they started to massively use GPUs that have thousands of processing units that can separately work. This improved performance but because they still follow the Von Neumann paradigm memory latency remains a bottleneck.

Tensor Processing Units (TPUs) have been proposed to solve problems of this kind by physically implementing systolic architecture. This architecture is made of many processing elements (PEs) that exchange data at every clock cycle; this allows limiting the memory bottleneck because data are instead passed through localized arrays of PEs in parallel that can perform the multiplication and addition operations simultaneously on different units. In a matrix problem, we need to map the dimensions onto the physical structure. The fundamental dimensions for this architecture are the rows, columns, and time. These represent the spatial and temporal dimensions of computation: rows and columns map matrix dimensions across PEs, while time corresponds to the pipelined propagation of data.

The algorithm can be chosen to flow in different ways using the convention for DNN with $OutputMatrix = WeightMatrix \times InputMatrix$:

- weight stationary: where the weight matrix is pre-filled in the systolic array whereas the input matrix propagates pipelined at each clock cycle;
- input stationary: similar to the preceding one but in this case the systolic array is pre-filled with the input matrix values;
- output stationary: both weight matrix and input matrix are pipelined in the systolic array whereas each PE stores the value of the local result, updating it.

In summary systolic arrays provide a scalable and energy-efficient alternative to classical Von Neumann architectures for

matrix multiplication. The choice of dataflow (weight stationary, input stationary, or output stationary) determines how data is mapped to the array and directly impacts performance and energy efficiency.

The proposed problem is to implement square matrix multiplication systolic algorithm for a commodity high-performance computing (HPC) cluster using message passing interface protocol. In this context, the processors in the cluster represent the PEs of the systolic array, where matrix elements are exchanged through message passing to enable the distributed execution of the multiplication. To perform the multiplication $C = A \times B$ the elements of the rows of A are send step by step to the PEs whereas the elements of the column of B are sent step by step to the PEs; at each step the local value of C is updated.

The experiments need to be performed with input matrices of order 500, 1000 and 2000 and varying the number of PEs used. The algorithm needs to read the input matrices from csv files and store the resulting output matrix in a CSV file.

B. Introduction to open MPI

Open MPI is an open-standard library for C, C++, and Fortran that implements the Message Passing Interface (MPI) and is suited for scientific applications where computation is parallelized across different nodes. It is used in distributed-memory computers, where one processor cannot directly access the memory of another processor; instead, communication is performed through a communication network. It supports both point-to-point and collective communications, and also allows synchronization. Using MPI, each processor executes the same code, which is organized to enable cooperation among the different processors thanks to the initialization of a communicator.

C. Summary of the analysis

The assignment required determining the execution performance and the usage of resources for each matrix size. These values depend on the number of processes used. Execution performance is evaluated in terms of execution time, varying both the matrix dimensions and the number of processors.

For the experimental environment a program was developed to randomly initialize 10 pairs of matrices A and B for the three dimensions. Matrix multiplication is performed for each of these 10 matrix pairs using the canonical serial $O(n^3)$ algorithm to generate reference C matrices for result verification. The execution times of the serial algorithm are also recorded to compute the average reference execution time to compare this result with the systolic algorithm from which is expected a speedup respect to the reference.

The experiment is performed for each combination of matrix order and number of processes running 10 iterations with the previously initialized A and B matrices. Statistics on the execution time are gathered to compute the average execution time.

The resource usage considered includes the number of processors used and the memory consumed. Memory usage is manually evaluated by observing the allocated memory as the number of processors changes.

For the 500×500 matrices the number of processors is varied from 1 to 72 in 1 step in a single node to determine the best trade-off between performance and resource usage. Then a further experiment is to distribute the workload across different nodes to understand how does this change the execution time.

For the 1000×1000 and 2000×2000 matrices, the number of processors in the node is varied from 4 to 72 in steps of 4. This allows evaluation of performance trends for the same total number of processors changing the dimensions.

A Python script was created to gather the data and generate graphs of the execution time.

D. Code explanation

In this section the main codes used for the simulations are explained.

The core of the project is `systolicMatricesMultiplication.c` (Algorithm 1), which performs matrix multiplication in a systolic manner for different initialization values and also supports non-square PE organizations.

In the simple case where the PEs are organized in a $n \times n$ array and the matrices have order n then, PE receives one M value from the left and one N value from the top, computes $R \leftarrow R + M \cdot N$, and then sends $P = M$ to the PE on its right and $Q = N$ to the PE below.

In the more general case, each PE depending on its position and the current step memorize three local matrices of the same dimension: $matrixM$ for the row entries of A, $matrixN$ for the column entries of B, and one matrix for the local elements of C. At each step, the PE receives a column M from the left and a row N from the top and sends the last column of $matrixM$ to the right and the last row of $matrixN$ downward. Then, is performed a shift of $matrixM$ to the right with the new M inserted as first column, and a shift of $matrixN$ downward with the new N inserted as first row. Finally, an element-element multiplication of $matrixM$ and $matrixM$ is added into the local C.

The rows elements of A and the column elements of B are sent in a delayed way starting with the elements in position $(0, 0)$. The total number of steps that need to be performed are $3n - 2$, in this way the last elements of A ($a_{n,n}$) and of B ($b_{n,n}$) reach the PE in the bottom right corner.

A key aspect to consider is load balancing among the PEs, i.e., the local matrix dimensions and the number of elements exchanged in each direction. Considering a PE with coordinates (i, j), a systolic array with r rows and c columns, and input matrices of order n, the dimensions of the local matrices are (lri, lcj), where

A key aspect to take into consideration is load balancing among the PEs, i.e., the local matrices dimensions and the number of elements exchanged in each directions. Considering a PE with coordinates (i, j), a systolic arrays with r rows and c column, and input matrices with order n, then dimensions of the local matrices are (lri, lcj), where if $i < n\%r$ then $lri =$

$n/r + 1$ else $lri = n/r$ and if $j < n\%c$ then $lcj = n/c + 1$ else $lcj = n/c$.

In this way, all PEs with coordinates lower than the remainder of the division of n by the array size in that dimension receive one additional element. Thus, the matrices are perfectly distributed across the PEs in a balanced manner, with at most a difference of one element in each direction. Consequently, all PEs in the same row handle the same number of column elements, while all PEs in the same column handle the same number of row elements.

`systolicMatricesMultiplication.c` takes the matrix order and repetition number as input. These values are needed to load the right matrices A and B from the input files, which are named with the convention `csv/InputI_dim_repetition`. For B, since the columns are shifted during the systolic process, it is generally easier to load the transpose instead of the original.

First the communication environment is set up. After that, a Cartesian communicator is created using the virtual topology. This makes it easier to organize the sends and receives between the different processing elements. Each process in the array is identified by a rank, which is used later on.

The initialization continues by loading the A and B matrices from the input files. This step is done only by the rank 0 process. Since this part is sequential and not part of the systolic algorithm, it is not counted in the execution time.

From this point the systolic algorithm starts, together with the execution time analysis. One of the first things is to evaluate the dimensions of the local matrices. To know what data needs to be sent we first need to know the top-left coordinates of C in a PE. This is done with `getBaseMatrixJCoord()`. Then the rank 0 process, which has A and B in memory, sends entire rows of A to the first column of the systolic array and entire columns of B to the first row. The choice of which rows and columns to send is again based on `getBaseMatrixJCoord()`. After that, all local matrices are allocated with the right dimensions: $localMMatrix$, $localNMatrix$, $localC$, M, N, P, and Q. This basically concludes the initialization, and from there the systolic steps are executed.

A first version of the algorithm, called `systolicMatricesMultiplicationV0.c`, had many useless send and receive operations. The systolic algorithm itself works in a kind of wave behaviour: at the beginning only the top-left PE has data to compute, while at the end only the bottom-right PE does. The first version performed sends and receives from every PE for each of the $3n-2$ steps, but many of these were unnecessary. The second version, which is the one used for the experiments, improves this by sending and receiving only when really needed. This is done using the function `getDimensionToSendRecv()`, which returns the size of the P data to send and the row of the first element, as well as the size of the Q data to send and the column of the first element. Then the ranks of the neighbouring PEs are determined with the MPI shift functions. If there is something to send down or right (and something to receive), the send and receive operations are carried out. To reduce the chance of deadlocks, when a PE is fully active it is more convenient to use `MPI_Sendrecv`. After this, $localMMatrix$ and $localNMatrix$ are shifted, the new row and column are added, $localC$ is updated, and the step counter is increased. In the final part of the algorithm, the C matrix is reconstructed: all PEs that are not rank 0 send their local C to rank 0, which uses the coordinates of the sending PE to place the values in the right location.

Then a barrier is called and the execution time measurement stops. The time is printed, and the resulting matrix is saved in a csv file following the convention `results/referenceOutputC_dim_repetition`. At the end, the allocated memory is freed and the communication environment is destroyed.

To automate the experiments, other programs are needed. `initializeMatrices.c` takes the order of the matrices and the repetition number, and generates random A and B matrices saved into csv files. `calculateReferenceMatrix.c` does something similar: it takes the order and repetition number, reads the input files, and computes the C matrix using the serial algorithm. The result is stored in a file named `referenceOutputC_dim_repetition`. `checkResults` is used to check whether the C matrices obtained with the systolic algorithm are correct, by comparing them with the reference matrices from the previous program.

`submitReferenceMatrix.sh` receives the dimensions as parameters. It iterates over them and for each one calls `referenceMatrix.slurm`, passing the dimension and the number of iterations. Then it calls `initializeMatrices.c` and `calculateReferenceMatrix.c` iteratively, saving the execution times and computing the average. This part is needed to set up the experimental environment.

Finally, the experiments are run with `submitExperiment.sh` (or the multi-node version). This script again takes the dimensions and number of processes, and for each combination calls `systolicRun.slurm` (or the multi-node version), which takes the dimension, number of processes, and number of repetitions. It then calls `systolicMatricesMultiplication.c` and `checkResult.c` iteratively. The outcome of the check and the execution times are written into a csv file named `results/systolic_dim_processes_rep`. Different dimensions and process counts produce different files, to avoid race conditions when many processes try to write at the same time.

E. Analysis of the results

The presented experiment refers to three different matrix orders: 500, 1000 and 2000. All the results of the experiments presented here were obtained on the partition called `cpu_sapphire`.

The experiments with matrix order 500 can be divided into three categories: experiments on a single node, on two nodes, or on a number of nodes equal to the number of processes.

Starting with the single node case, the number of processes is varied with a unitary step from 1 to 72 processes and with a single core per process. The experiment is repeated 10 times to gather information about the average execution time. The complete results are presented in Table VIII, where the average execution time, the speedup compared to the reference execution time (0.3786042 s), the speedup compared to the single process case, and the efficiency are shown.

The speedup with respect to the reference is measured by $S_{ref}(p) = T_{ref}/T_p$ and allows us to understand when and how much the systolic algorithm is more convenient than the sequential one.

The normal speedup instead is measured by $S(p) = T_1/T_p$ and shows the time advantage of using p processes. Efficiency is a parameter that shows how efficiently the resources are used: $E(p) = S(p)/p$. From Table VIII we can notice that for $p > 57$ the efficiency drops and both $S_{ref}(p)$ and $S(p)$ drop, and eventually the execution times become longer than the reference.

The execution time statistics are also presented in a box-plot in Figure 1. Box-plots (or whisker plots) are very useful to represent more statistical information: for each processor number the middle line represents the median, the box represents the borders of the first and third quartiles where 50% of the results are distributed, the lower whisker is evaluated as $Q_1 - 1.5 \times (Q_3 - Q_1)$, the upper whisker as $Q_3 + 1.5 \times (Q_3 - Q_1)$, and finally the outliers, i.e., values that are not distributed between the whiskers, are individually represented with circles. In red the reference average execution time is also represented.

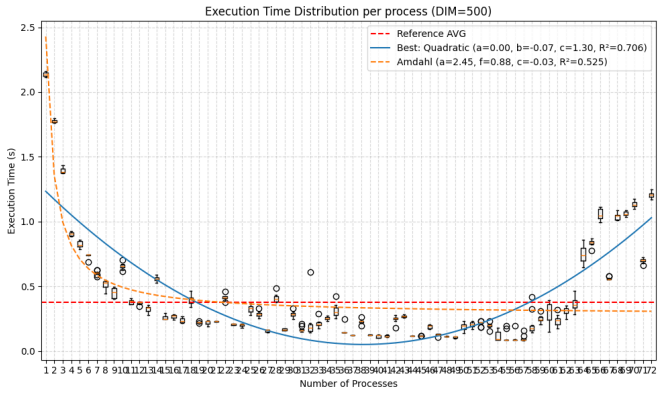


Fig. 1: Box-plot of execution time per process with matrix order 500 in a single node

The Python code that generated this graph also evaluated the best-fit function for these distributions, truncating the analysis at the quadratic function. Considering the whole distribution of values in the experiment, the best fit is quadratic with $R^2 = 0.706$, which is not a perfect fit; but if we truncate at 61 processes we obtain a behavior similar to Amdahl's law with $R^2 = 0.879$, $T(1) = 2.48$, $f = 0.98$, $overhead = 0.11$ (even if in this case the best fit would be the exponential one). Amdahl's law states that "the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used" and is described by Equation 1.

$$T(p) = T(1)\left[(1-f) + \frac{f}{p}\right] + overhead(p) \qquad (1)$$

Intuitively this tells us that initially with a low number of processes the improvement given by the parallelizable part dominates, but increasing the number of processes it is not possible to improve beyond the time required for the serial part. In our case moreover the overhead at a certain point increases, making the execution time increase with the number of processes. This is common behavior for MPI programs because the control overhead due to message exchanges can be very onerous.

Considering the experiments with matrix order 500 on two nodes, the number of processes was varied from 4 to 72 with a step size of 4. The collected results are the same as in the preceding case and are reported in Table XI. In this case we can notice instead that initially the speedup and the efficiency for a small number of processes are lower, but increasing the number of nodes there is no increasing trend in the average execution time as in the single node case.

A final experiment for matrix order 500 is to allocate a number of nodes equal to the number of processes, but in this case apart from specific values of processes that are 2, 4, 5 and 6, SLURM returned the following error:

```
An ORTE daemon has
    ↪ unexpectedly failed
    ↪ after launch and before
communicating back to mpirun.
    ↪ This could be caused by
    ↪ a number
of factors, including an
    ↪ inability to create a
    ↪ connection back
to mpirun due to a lack of
    ↪ common network
    ↪ interfaces and/or no
route found between them.
    ↪ Please check network
    ↪ connectivity
(including firewalls and
    ↪ network routing
    ↪ requirements).
```

The results in this case are shown in Table XII, but too few values are available to draw any conclusion.

The average values for the three experiments with matrix order 500 are all plotted in Figure 2, where it is possible to notice that in the two-node case the average times are greater than in the single-node case but are decreasing also for numbers of processes greater than 61.

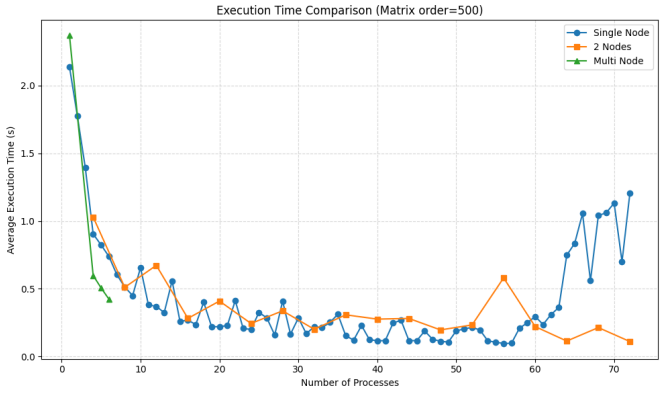


Fig. 2: Comparison of single-node and two-node experiments with matrix order 500

The experiments performed with matrix orders 1000 and 2000 are both run with processes from 4 to 72 with a step of 4, but including also the single-process case. In both cases the experiments are performed on a single node, with 10 repetitions to gather statistics on the execution time.

The results for the 1000 matrix order case are presented in Table IX, where the reference average execution time is 2.791806 s. Looking at these results we can notice that the maximum speedup with respect to the reference, obtained with 60 processes with a value of 4.225456, is greater than the speedup that can be obtained with the 500 matrix order case, and also the efficiency around 50% is a good value. These results are also represented in the box-plot in Figure 3, where it is possible to notice that the variance in this case is much smaller than in the 500 case and also the outliers are very limited. Moreover in this case the best fit is almost perfectly in accordance with Amdahl's law, in fact $R^2 = 0.982$, $T(1) = 21.45$, $f = 0.9$ and $overhead = -1.07$.

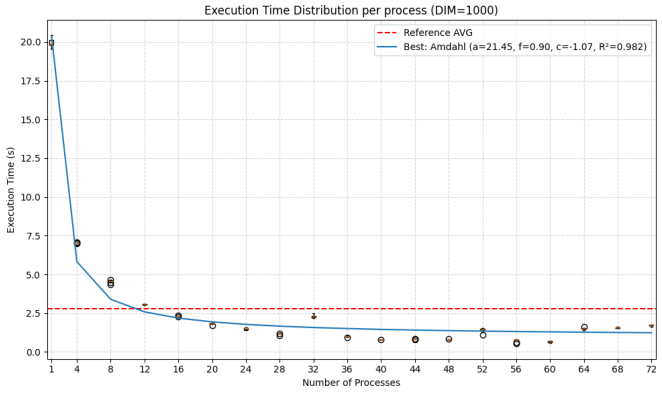


Fig. 3: Box-plot of execution time per process with matrix order 1000 in a single node

The results for the 2000 matrix order case are presented in Table X, where the reference average execution time is 22.214955 s. These results show a trend: in this case the speedup is even greater than in the preceding case (4.850642 for 68 processes), and the efficiency values are all around

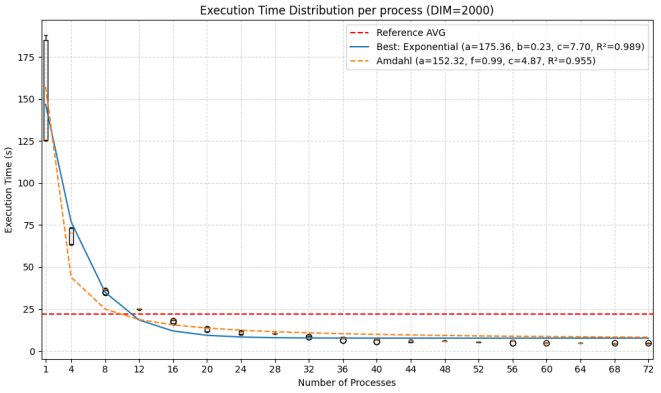


Fig. 4: Box-plot of execution time per process with matrix order 2000 in a single node

50% even for 72 processes. It seems in fact that the average execution times are not yet at saturation in this case. This is confirmed also by looking at the box-plot in Figure 4, where we can notice that the variance is even smaller and the outliers are closer to the median value. The trend is continuously decreasing. The best fit is in fact exponential and not Amdahl, even if the fit with Amdahl gives good results with $R^2 = 0.955$, $T(1) = 152.32$, $f = 0.99$ and $overhead = 4.87$.

Another fundamental aspect to take into consideration is the memory usage and how it changes with the matrix order and the number of processes. It is possible to manually evaluate this cost by considering the allocated memory for each process and the variables used; for this analysis, it is possible to ignore the variables and memory usage that are constant and independent of the order or number of processes, such as the auxiliary variables used by rank 0 process for the initialization.

To perform the calculation, it is important to point out that the rank 0 process has a different memory usage because it stores matrices A, B, and C. Additionally, the processes in the first row and column of the systolic array have different memory occupation because they store entire rows of A and columns of B.

The result obtained with these considerations is $64 \times n^2 + 480 \times p_x \times p_y + 16 \times n \times (p_x + p_y)$ bytes, where p_x and p_y are the number of rows and columns of the systolic array, and n is the matrix order. We can notice that $p_x \times p_y$ is equal to the number of processes used. The memory usage behavior is quadratic with n, with $p_x \times p_y$, and with $n \times (p_x + p_y)$.

F. Conclusion

The results for the 500 matrix order case show an increase in the number of resources in the single node case. This can be caused by limitations in the node and not by a real increase of control overhead given by the MPI protocol used for the systolic algorithm. If this was the case, in fact, we should have noticed also an increase in times for the dual node case that instead was not visible, as clearly shown in Figure 2.

The best trade-off in the single node case could be evaluated as the minimum point of execution time which is 0.096078 in correspondence of 57 processes with a speedup with respect to reference of 3.940588. Also, if we consider the efficiency and we limit ourselves with the trade-off to an efficiency of around 40%, the best is still this point or maybe 49 processes if we consider that we need to have a value of efficiency greater than 40%.

The trade-off for the double node case cannot be obtained clearly because the step size for the processes evaluation is too large to obtain a clear value and moreover in this case the execution times have not properly arrived at saturation of resources. In fact, the lowest average time is the last with 72 processes.

Comparing the single node case and the double node case, as already pointed out when showing the results, it is possible to notice that the execution times of the double node case are greater than the single node case and this can be easily explained by the time overhead for communication between different nodes with message exchange.

Comparing instead the behaviour of the algorithm on different matrix sizes on a single node with the same number of processors, it is possible to notice from the graph in Figure 5 that for a small number of processes, the increase in execution time with matrix size grows faster than linear (closer to the cubic complexity of the algorithm), whereas for a higher number of processes the scaling appears closer to linear because the additional parallelism compensates part of the computational growth. From the structure of the algorithm this can be easily explained: to initialize the communication and the systolic behaviour some time is required, but if the number of processes is too small, this startup cost remains significant relative to the total computation. Instead, when we increase the number of used processes, the behaviour approaches the expected polynomial growth in n (approximately cubic), with the advantage of parallelism counterbalancing the initialization costs.

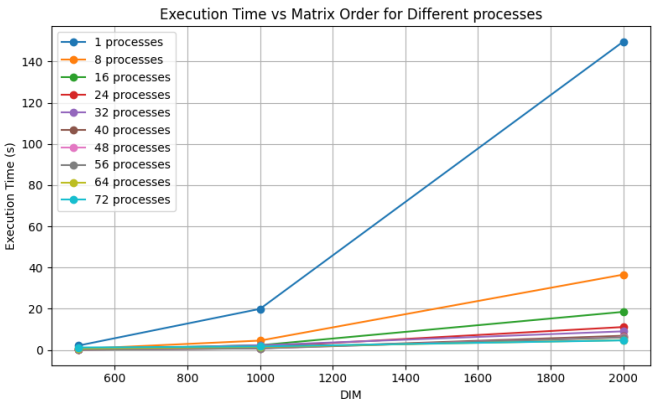


Fig. 5: Execution time varying the matrix order for different numbers of processes.

Finally, another interesting comparison is about the Amdahl's law parameters obtained. In the 500 case we look at the value obtained with a lower number of processes. In particular, looking at the $T(1)$ parameter, this matches well with the real $T(1)$ value, but the more interesting point is the fact that the value of f increases with the matrix order. This means that increasing the dimension of the problem increases the percentage of the algorithm that is parallel, showing a greater advantage in parallelization. This is also confirmed by the fact that the speedup and the efficiency increased with the dimension of the problem.

This perfectly matches the expected behaviour: MPI programs that are not communication intensive typically achieve higher efficiency as the problem size grows because the ratio between computation and communication becomes more favorable. In other words, larger problems are able to better amortize the fixed communication and synchronization overheads, making the parallel fraction dominate the execution time. As a result, the scaling trends observed across matrix orders 500, 1000, and 2000 provide strong evidence that the systolic algorithm benefits significantly from increasing problem size. Overall, the experiments confirm that while small problems suffer from communication and initialization costs, larger problems align more closely with Amdahl's law predictions and highlight the real potential of MPI-based parallelization for computationally intensive workloads.

In conclusion, these experiments highlight that the potential advantages of a real systolic array implemented in dedicated hardware lie in the reduction of memory latency and the communication bottlenecks caused by message exchange. Moreover, compared to the proposed solution, dedicated hardware would not need all the buffers required to implement the systolic behaviour, which cause an increase in memory usage with the number of processing units used. This explains why architectures such as Google's TPU, which are based on systolic arrays, achieve enormous performance benefits in large dense matrix multiplications, and why Google has invested heavily in this technology.

III. Second exercise

A. Introduction on the exercise

In today's computational applications such as computer vision, scientific simulations, and neural networks, operation on huge multi-dimensional data sets is a fundamental requirement. A common approach to obtaining meaningful information or enhancing the quality of data is through stencil or filtering operations. These compute weighted averages over local neighborhoods of data points (for instance, pixels in an image) that can remove noise, emphasize features, or enable further analysis.

The exercise suggested in this project is the use of a two-dimensional stencil filter to work on digital images. Particularly, a predetermined filter W is applied on a per-pixel basis of an image to denoise and preserve the original signal components. The project also entailed demonstrating performance scalability using higher-resolution images (4K, 8K, and 16K) and verifying the filter itself in removing noise that was artificially added.

Graphics Processing Units (GPUs) suit this type of load well. The stencil operation is parallel to the extent that the computation of every output pixel is dependent only upon its local neighborhood and can be computed independently from other pixels. CUDA, which is a parallel programming model by NVIDIA, gives explicit control over GPU threads and memory hierarchies in order to release enormous parallelism and achieve significant performance benefit compared to CPU execution.

Besides, the High-Performance Computing (HPC) configuration used in this project, with the scheduling of tasks by SLURM, facilitates multiple GPU resources to be allocated and fine-grained monitoring of resource utilization. Utilizing both CUDA and HPC resources makes it possible to process very large images efficiently, the effect of different thread configurations to be examined, and the dimensionality between parallelism, runtime, and resource utilization to be experimented.

B. Background

Filtering is a widely used technique in image processing and multi-dimensional data analysis. Its purpose is to manipulate or enhance images by emphasizing relevant features or reducing undesired noise. A common class of filters are stencil operations, which operate by replacing each input element with a weighted combination of its neighbors. This approach is particularly effective for smoothing, denoising, and feature extraction.

The matrix **W** defines the stencil used in this work.

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

This filter computes the weighted average of a pixel and its immediate neighbors with the formula:

$$g(x,y) = \frac{1}{16} \sum_{j=-1}^{1} \sum_{i=-1}^{1} W(i,j) \cdot f(x+i, y+j) \tag{3}$$

where $f(x,y)$ is the input image and $g(x,y)$ is the filtered output. The normalization factor $\frac{1}{16}$ guarantees that the final pixel intensity is within valid ranges. For pixels along the boundary of the image, where the neighbor pixels will not be available, zero-padding is possible. The use of this filter creates an output that is blurred, effectively eliminating high-frequency noise but preserving the overall structure of shapes and edges. For color images, the same stencil is independently applied across the three RGB bands, then the outputs reassembled into a final image. This paper assesses both the filtering process's precision and its computational effectiveness with regard to high-resolution images. The calculations grow quadratically as the dimensions of the picture grow from 4K (3,840 × 2,160 pixels) through the 16K (15,360 × 8,640 pixels) dimension, so GPU acceleration is vital.

C. Methodology

The assignment deployment was accomplished in multiple steps, each addressing a specific part of the workflow from image preprocessing to GPU-accelerated filtering.

1) Installation and configuration: The initial step involved the installation of the software environment. As the task involved image decomposition and reassembly, OpenCV library was installed and configured in the HPC environment. OpenCV offers efficient functionality to load images, decompose them into their RGB components, and perform basic operations, which made integration with CUDA easier.

2) CPU-Based Preprocessing (C++ Implementation): To verify the workflow and to provide input data for the GPU, a C++ implementation was devised. The program performs a simple resizing of the input image in a way that different resolutions (4K, 8K, 16K) were created. The C++ code is ran before executing the CUDA kernel. The code in "resize.cpp".

3) GPU-Based Filtering (CUDA Implementation): The central piece of the project is the CUDA computation of the stencil filter **W**. The filter is convolved with each pixel of the input image by the CUDA kernel, who computes the weighted sum of its local neighborhood, according to Equation 3. The operation is performed independently across the three RGB channels and results recombined into a final filtered image.

Different arrangements of threads and blocks were attempted in order to examine the effect of parallelism on execution time and on resource usage. The CUDA code was built and executed on the HPC cluster with the help of SLURM job scheduling, thus allowing systematic testing of performance at different image sizes and levels of GPU parallelization.

The application of the stencil filter is done with the usage of a CUDA program (decomposeAndFilter.cu), where we define the kernel that will run on our GPUs.

4) Execution on HPC with SLURM: The last step of the workflow was executing the programs on the HPC cluster. For reproducibility and automating the process, two different SLURM job submission scripts were written:

run_cuda_size.slurm job script: runs the C++ program for resizing and preprocessing images applying the CUDA stencil filter to each resolution.

run_cuda_noise.slurm job script: runs the CUDA-enabled stencil filter with the number of GPUs and CPU cores per task on 4K images with different noise (50%, 75% and 90%).

Both scripts create output and error log files, which allowed for monitoring of the execution time and resource usage. Having separated the jobs, it was easy to test and check the two components of the project independently, as well as distribute the required resources into each run.

D. Results

The experiments were performed in an attempt to evaluate the performance as well as the efficiency of the CUDA implementation of the stencil filter. To provide an advanced analysis, several parameters and metrics were considered:

- Elapsed time: the time required for carrying out the filtering operation, measured in a bid to gauge the computing effectiveness at different sizes of images and setups.
- Resource utilization: memory utilization and GPU utilization metrics, retrieved from the HPC job monitor system, to estimate kernel scaling with problem size.
- Threads and blocks: several settings were employed to investigate the effects of parallelization granularity on performance.
- Number of devices: in addition to single-device execution, multi-device runs were performed to evaluate device-wide scalability.
- Peak Signal-to-Noise Ratio (PSNR): a quantitative estimator of image quality, used to compare the filtered output with the original image. More effective noise reduction and higher similarity with the ground truth is indicated by a high PSNR.

Elapsed time, consumption of resources, and parallelization parameters provide hints about the implementation efficiency from a performance point of view, while PSNR provides a measure of the filtering quality. Together, they permit a discussion of the computational vs. image quality trade-off.

For the experiments I used the the "pexels-christian-hetz.jpg" image provided on Portale della didattica.

1) CUDA preprocessing: Thanks to SLURM is possible to set the number of GPUs to use, choosing between different partitions. What I did is to run different SLURM jobs with different attributes, obtaining so different results. The results are reported in Tables XIII, XIV, XV and XVI

From the tables is possible to see that the main difference is between the execution times. They vary using different threads and GPUs, improving or getting worse.

Here the graphs, done with Python:

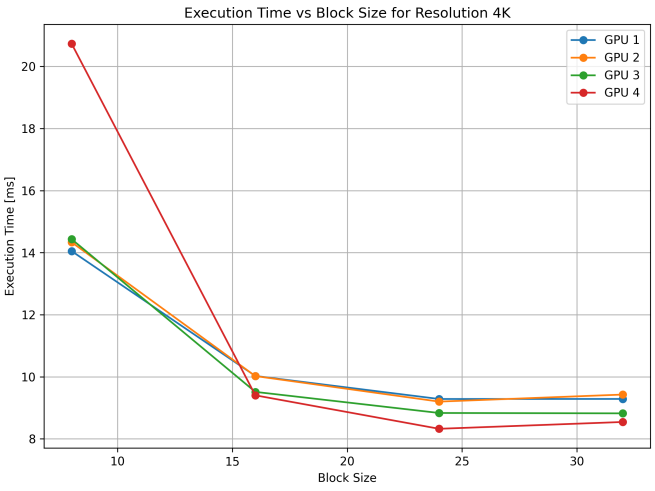


Fig. 6: Execution times of resized_4K image

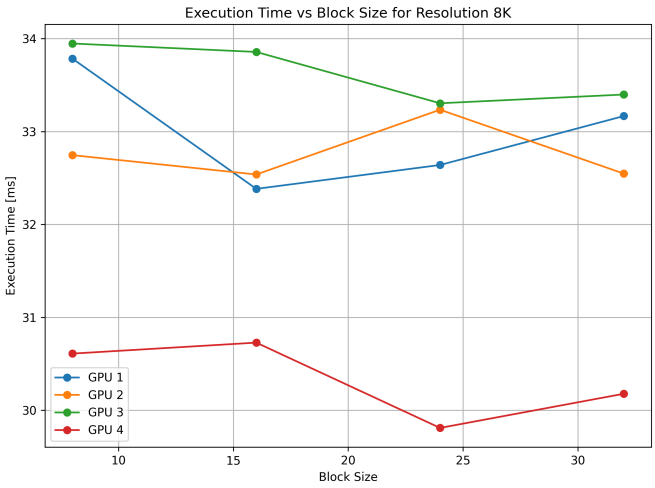


Fig. 7: Execution times of resized_8K image

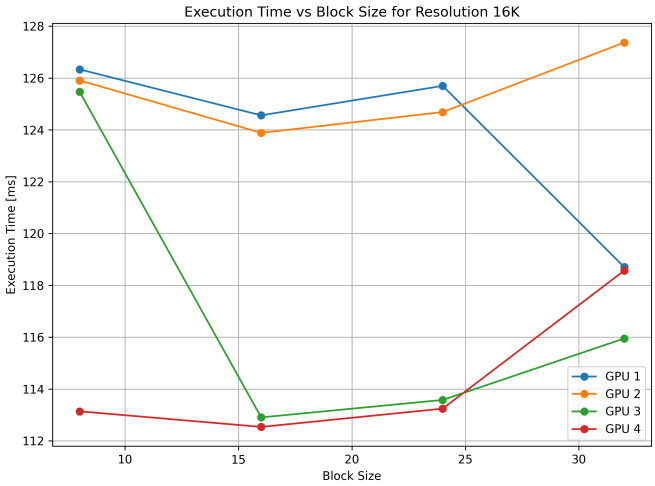


Fig. 8: Execution times of resized_16K image

2) CUDA processing on noise: In this step, after a first application of the stencil filter on the different resolution images, I applied and ran a CUDA kernel to blur an input noisy image, with the purpose of restoring its visual quality and verifying the parallel execution effectiveness. The noise was introduced through online tools on the 4K resolution image in different percentages (50%, 75% and 90%). The results are reported in Tables XVII, XVIII, XIX and XX

In this job, similar to the first one, always the execution time is the more oscillating value. Another value that is possible to observe is the filtered PSNR that is less in the image with the 50% of noise and increase in the images where the noise percentage is 75% and 90%. So in the first image the quality is not increased, instead in the other two the quality is increased. As in the first job here are reported the graphs with the execution times:

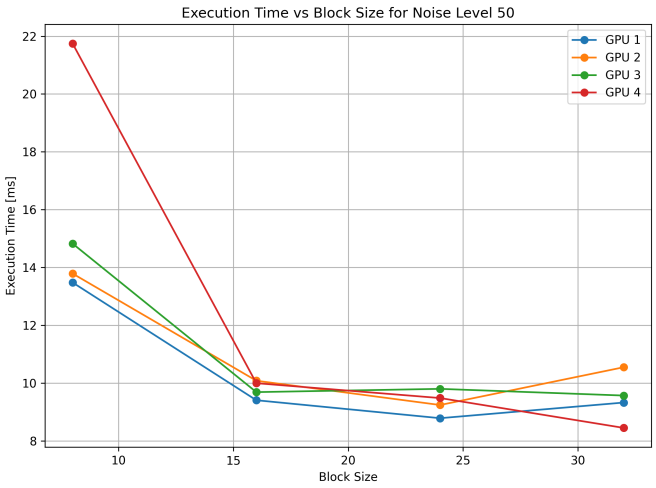


Fig. 9: Execution times of image_noise50 image

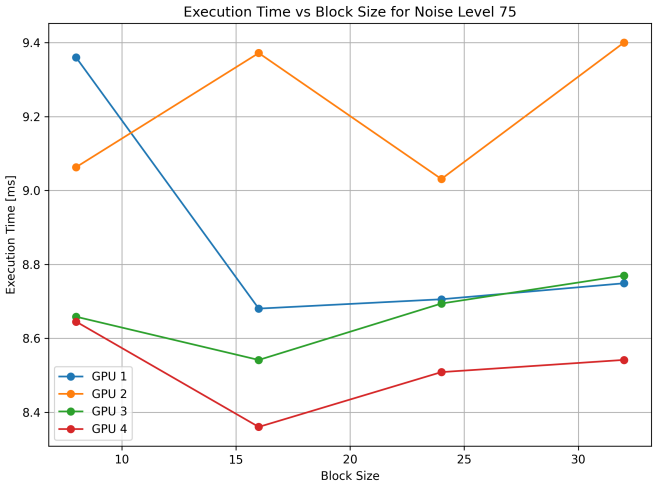


Fig. 10: Execution times of image_noise75 image

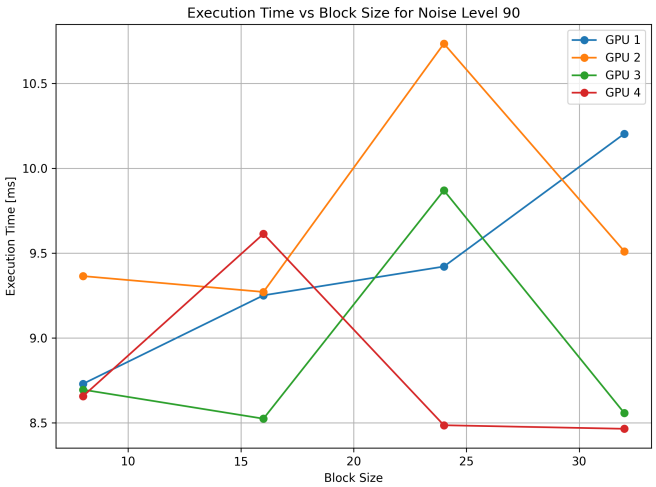


Fig. 11: Execution times of image_noise90 image

As is possible to see from the graphs, using a large number of GPUs and threads is not always the better solution. For example, seeing the Figure 9, using only one GPU and 8x8, 16x16 and 24x24 block size the execution time is less than other configurations.

E. Conclusion

In conclusion, the application of the stencil filter with CUDA was run successfully and tested on noisy images. The kernel performed as expected using different configurations of GPUs to confirm the efficiency of the parallel solution. However, the improvement in image quality was relatively small, as the PSNR values indicate, because the chosen filter reduces noise only partially. These results suggest that while CUDA provides excellent benefits in execution time and scalability, more advanced filtering techniques or adaptive kernels may be required to achieve significant improvements in image improvement.

Here are reported some images, before and after the application of the stencil filter:

Fig. 12: Pexels-Christian-Heitz in 4K with 90% of noise

Fig. 13: Filtered Pexels-Christian-Heitz with 90% of noise

At human eyes the improvement in Figure 13 is imperceptible, but the stencil filter works perfectly.

IV. Third Exercise

A. Introduction on the exercise

Heat diffusion is a process by which thermal energy spreads through a material due to a temperature gradient. It is more

efficient in some materials with respect to some others depending on the microscopic characteristics of them. In particular in metals heat diffusion is particularly efficient because of the presence of lots of free electrons which can transfer energy between them through collisions simplifying the diffusion of heat. In this work we concentrate only on the heat diffusion in metals.

This process can be modeled to study how the temperature changes in a material through different methods. One of these methods is the finite difference method. This method consists in approximating derivatives with difference equations using truncated Taylor expansion. In this method spatial domain (or time one) is discretized and the values of the solution at the end points are approximated by solving algebraic equations containing finite differences and values from nearby points. A piece of a metal can then be represented by a NxN grid (or by a cube in 3D) where each cell holds a temperature value. If there is an initial gradient of temperature in the grid, heat will diffuse according to the characteristics of the material. Over time the temperature of each cell is iteratively updated until the change between one step and the following one falls below a minimum threshold value or a maximum number of iterations is reached.

The exercise requires to use a 1024x1024 grid that represents a metal plate. Then two different initial conditions and thermal propagation rules are proposed:

- The first input condition consists in having half of the plate at 250.0°C and the other half at the earth's average temperature (15.0°C). The propagation law instead is that the temperature of every cell is given by the average of its four neighbors.
- The second input condition is that the central quarter of the cell has a temperature of 540.0°C and the rest of the plate has the earth's average temperature (15.0°C). The propagation law in this case is anisotropic since the heat flows differently in the horizontal and vertical axis following the rule:

$$\begin{aligned} cell'(i,j) = w_x * (cell(i,j-1) + cell(i,j+1)) + \\ + w_y * (cell(i-1,j) + cell(i+1,j)) \end{aligned} \tag{4}$$

  where w_x and w_y respectively are the weights of horizontal and vertical diffusion with values 0.3 and 0.2.

B. Introduction to OpenMP

The simulation is performed with OpenMP. OpenMP which stands for Open Multi Processing is an Application Programming Interface for shared-memory multiple-thread applications. It allows to write parallel programs in C, C++ and Fortran for shared memory architectures. It separates a program in sequential and parallel parts and it spreads the operations of the parallel parts between different threads providing also sinchronization between them. It uses a fork-join parallelization algorithm. There is a master node which executes the sequential parts of the code and coordinates the work of the other nodes. OpenMP is able to execute the code with different number of threads whose number can be set by the user.

C. Summary of the analysis

In this section a brief explanation of how the analysis has been made is reported. First of all the requests of the exercise are to evaluate the impact of the number of threads in the performance execution in the two cases and to determine and analyze the execution performance and the temperature diffusion for the range of up to 10,000 iterations.

To analyze different situations and simulations the following numbers of maximum iterations have been chosen: 50, 500, 500, 10000 and then two extra simulations have been performed with 20000 and 50000 maximum iterations. The number of threads for every simulation changes between 1 and 11. The first ten cases are used to obtain the data about the performance while the last case (11 threads) is used only to save the grid temperature values in a file using an appropriate sampling period to represent then the diffusion graphically.

For all the six simulations both the case A and B are simulated. All the 12 simulations are performed with the number of threads varying between 1 and 11. The execution times varying the number of threads between 1 and 10, are taken and then they are represented in a graph to analyze the impact of the number of threads in the performance. Then in the last step, when there are 11 threads, the grid is saved 10 times during the simulation. The sampling period used is indeed $N/10$ where N is the number of max iterations. The grids are saved and then represented in appropriate graphs. Both the heatmaps and execution times graphs are obtained using two Python codes. The first one represents execution times as the number of threads varies and calculate the best fit function for the trend. The second one represents the heat distribution given all the grid values.

At the end other three cases have been studied. In these three cases the boundary conditions have been changed with respect to the initial case to understand which are the most efficient ones. These three simulations have been performed only with 10000 maximum iterations and the same data as before have been taken.

D. Code Explanation

In this section an explanation of the OpenMP code used for the simulations is provided.

The code is written in C language and the parallelization is performed thanks to OpenMP. At the beginning the variables are defined, such as the temperature, the dimension of the grid and the number of threads. Here also the threshold is defined which is set at 0.001 in all the cases. This value has been chosen because it is small enough in such a way the simulations always arrive to the maximum number of iterations. This is done because the analysis consists in comparing the performances as the number of threads changes and this is easier if the whole simulation is performed each time.

Then the memory space is created for the grid and for a copy of it which is necessary when parallel operations are performed

to avoid race conditions. These are situations that happen if a copy of the grid is not created. It can happen indeed that, working in parallel, one thread reads a value when another one is still writing it or it can happen that two threads write the same value at the same time. It is necessary then to create a copy of the grid where all the operations will be performed and then the new grid will be copied in the original one.

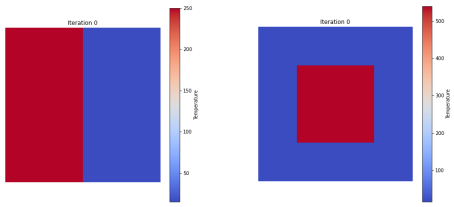
Then a .csv file is open to save the grid, which is saved only when the number of threads is 11 at the beginning and then every N/10 iterations where N is the maximum number of them. The inital conditions of the grid are set and then the parallel parts begin. In the first one the propagation law with its boundary conditions is defined. In the case A a generic cell of the grid is updated equal to the mean of its four neighbours. Since the propagation takes place only in the horizontal and vertical directions this is considered to be true also for the edges and the angles. The angles then will have only two neighbours while the edges' cells three and the new temperature will be the mean of their values. These are the boundary conditions set which will be used for all the simulations (except for the extra three with different B.C. at the end). In the case B instead a generic cell is updated following the rule 4. Also in this case the propagation is only vertical and horizontal so the angles will be influenced only by two cells and the edges by three.

Then another parallel part is present which copies the new grid modified by the propagation at every step in the original one. Grid values are saved in the .csv file every sampling period and execution times are saved in another .csv file. At the end the memory spaces occupied by the grids are freed and the .csv files are closed.

E. Main analysis

In this section the main part of the analysis will be performed reporting the data taken in tables and representing them in appropriate graphs.

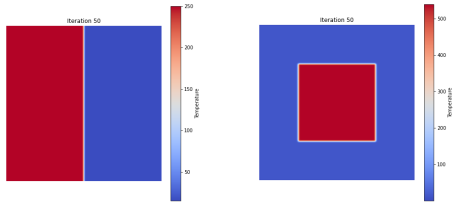
To begin the two initial configurations of the grid are included here.



(a) 0 it., A (b) 0 it., B

The grid configurations are represented by heatmaps where colour changes from blue (colder parts) to red (hotter parts).

The first case is the case with a maximum of 50 iterations. It is possible to see as expected a little variation in the grid after only 50 iterations. The last configuration of the two cases A and B is here reported (the configuration in the middle are unhelpful so they are not present).



(a) 50 it., A (b) 50 it., B

This is then a visual and intuitive way to see how the heat spreads. In this case the changing is minimal beacuse the number of iterations is of course very small.

After that the execution times changing the number of threads are reported, first in a table where also the speedup and efficiency are calculated and secondly the times are represented in a graph to see their trend. The data refer to the case A (data for the case B are almost identical). Speedup is a measure of how much the time has been reduced with respect to the case with one thread and it is measured by $S(p) = T_1/T_p$ where p is the number of threads. Efficiency is a parameter that takes into account also the number of threads used and its expression is: $E(p) = S(p)/p$.

TABLE I: Performances with 50 iterations max.

Threads	**Time [s]**	**Speedup** $S(p)$	**Efficiency** $E(p)$
1	0.231	1.00	1.00
2	0.116	1.99	0.99
3	0.078	2.96	0.99
4	0.062	3.72	0.93
5	0.049	4.71	0.94
6	0.042	5.50	0.92
7	0.036	6.42	0.92
8	0.035	6.6	0.83
9	0.032	7.22	0.80
10	0.029	7.96	0.80

From this table it is possible to see the following things. First of all the speedup is increased for every step which means that there is no exceed of resources of server during the simulation. Then concerning the efficiency it is possible to see that the highest values are for 2 and 3 threads. This was an expected result since it is a behaviour that usually happens in HPC. Indeed when a small number of threads is used, first of all resources are well distributed, costs for scheduling, barriers and synchronization are small since the number of threads is small and at the end the resources used are not too much so there is no the risk to pass over the memory bound imposed by the server. However the efficiency values remain pretty high also in the last cases since they are all above 0.80.

Now the graph which represents the execution times as the number of threads changes is reported. It is obtained with a Python code using the values reported in the table I.

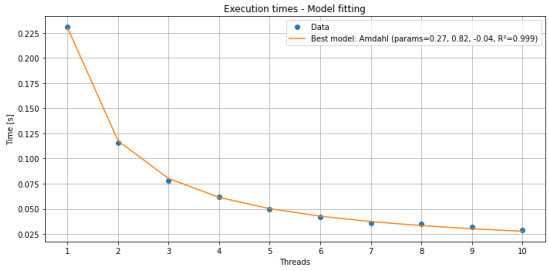


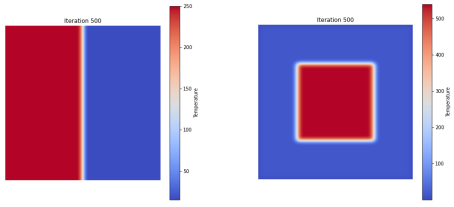
Fig. 16: Execution times with 50 iterations

The trend in times as a function of the number of threads is not linear, but decreases with asymptotic flattening. At first, the reduction seems exponential because parallelizable work dominates, but once a certain threshold is exceeded, memory becomes the bottleneck and times no longer improve significantly. This behavior is described by Amdahl's law that was defined in the first exercise. This means that the trend of execution times will never be linear since some parts of the code are sequential and then they cannot be accelerated. The law that describes the performance improvements is Equation 1

Where $T(p)$ is the execution time for p threads, $(1-f)$ is the serial fraction of the code, f the parallel part which is improved linearly with the number of threads and $overhead(p)$ is the increasing costs due to synchronization between threads. The trend obtained is then a classical trend obtained in these type of problems.

The Python code implemented to represent this graph is also able to determine the most probable function which describes the trend of the time values, using the R^2 coefficient. It is possible to see that the best fitting obtained is the Amdahl's one with a R^2 coefficient very close to 1. The trend is compared with the function: $T(p) = a * [(1-f) + f/x] + c$, where $a*f/x$ represents the parallel part, $a*(1-f)$ the serial one and c approximates the overhead.
The analysis goes on with the next case where the maximum number of iterations is 500. Here the simulation is similar to the previous one but in this case execution times are of course higher.

First of all the heatmaps of the two cases after 500 iterations are reported.



(a) 500 it., A (b) 500 it., B

It is possible to see a little difference between the previous case because of course higher the number of iterations, bigger is the spread of the heat in the grid.

After that the graph with execution times for the case A is reported. In this case the table with times, efficiency and speedup is not included since it is very similar to the previous case (except for the longer times).

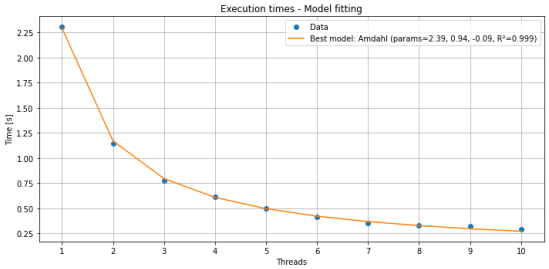
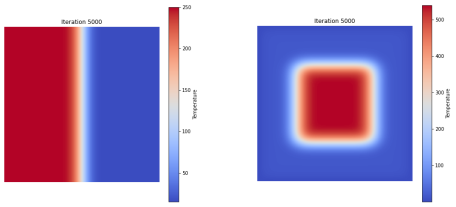


Fig. 18: Execution times with 500 iterations

In this case of course times are higher since the number of iterations is larger. It is possible to see that all the execution times are almost 10 times the times in the previous case where the number of iterations was 10 times smaller. It can therefor be deduced, that times grows linearly with the maximum number of iterations (this hypothesis will be confirmed in the following simulations). In this case, as in the previous one, the best fit is calculated and it follows as expected the Amdahl's law with a R^2 coefficient very close to 1.
The analysis goes on with the next case where the maximum number of iterations is 5000. Here the simulation is similar to the previous one but in this case execution times are of course higher. First of all the heatmaps of the two cases after 5000 iterations are reported.



(a) 5000 it., A (b) 5000 it., B

It is possible to see that heat begins to spread more because the number of iterations is increasing. In the case A indeed the border between the two regions tends to the mean temperature of the two and the strip involved is larger with respect to the case of 500 iterations. The same happens in the case B where heat spreads more around the edges of the central square of the grid.

After that the graph with execution times for the case A is reported. In this case the table with times, efficiency and speedup is not included since it is very similar to the first case (except for the longer times).

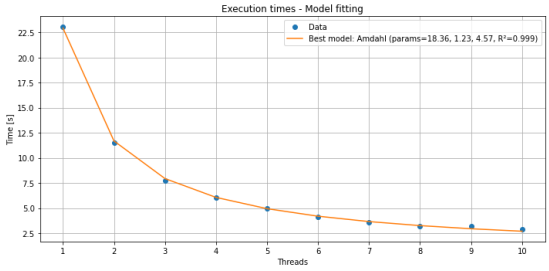
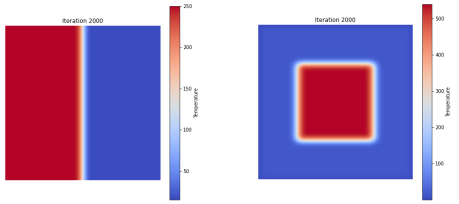


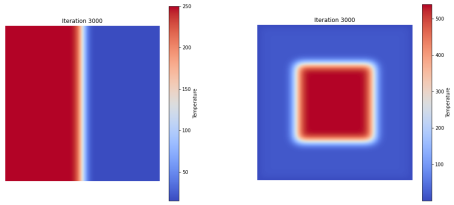
Fig. 20: Execution times with 5000 iterations

It is possible to see that all the execution times are almost 10 times the times in the previous case where the number of iterations was 10 times smaller. This is another confirmation of the hypothesis made before. In this case, as in the previous one, the best fit is calculated and it follows as expected the Amdahl's law with a R^2 coefficient close to 1.
Now data for the case of 10000 iterations maximum are provided. In this case different steps of the heat diffusion are reported to have a general representation of it. The heatmaps at 2000, 3000, 8000 and 10000 iterations are reported for both the cases.



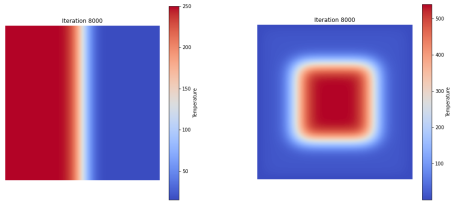
(a) 2000 it., A

(b) 2000 it., B



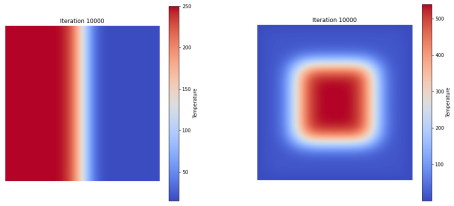
(c) 3000 it., A

(d) 3000 it., B



(e) 8000 it., A

(f) 8000 it., B



(g) 10000 it., A

(h) 10000 it., B

Fig. 21: Heatmaps at different steps

The figures 21 clearly show the spread of the heat in the grid. It is easy to notice that increasing the number of iterations lets the heat diffusing more and more. It is possible to notice indeed that at the beginning the heat spreads only in a central strip in the case A and only around the edges of the square in the case B. When the number of iterations increases instead the heat begins to spread all over the grid and in the region around the central strip in the case A and around the edges of the central square in the case B, the temperature tends to uniform between the two different ones. As it is possible to see 10000 iterations are not sufficient to see a complete distribution of the heat in all the grid, that's why also 20000 and 50000 iterations have been simulated (data are reported in the next section).

After having visualizing the heat diffusion through the grid, the analysis goes on considering the performances changing the number of threads. In this case, as in the first case of 50 iterations, times are reported in a table with also efficiency and speedup to see the differences with respect to the first case. Data for case A are reported (data for case B are almost identical).

TABLE II: Performances with 10000 iterations max.

Threads	**Time [s]**	**Speedup** $S(p)$	**Efficiency** $E(p)$
1	46.060547	1.00	1.00
2	22.962975	2.01	1.00
3	15.729801	2.93	0.98
4	12.112103	3.80	0.95
5	9.906704	4.65	0.93
6	8.265296	5.57	0.93
7	7.228200	6.37	0.91
8	6.785887	6.79	0.85
9	6.408788	7.19	0.80
10	5.760955	8.00	0.80

From this table it is possible to see the following things. First of all the speedup is increased for every step which means that there is no exceed of resources of server during the simulation. Then concerning the efficiency it is possible to see that the highest value is for 2 threads. This was an expected result as was explained before. However the efficiency value remain high since they are all above 0.80. It is possible to say then that a simulation with 10000 iterations is still advantageous since it does not overcome the resources available.

The graph of execution times as function of number of threads in the case of 10000 iterations is here reported.

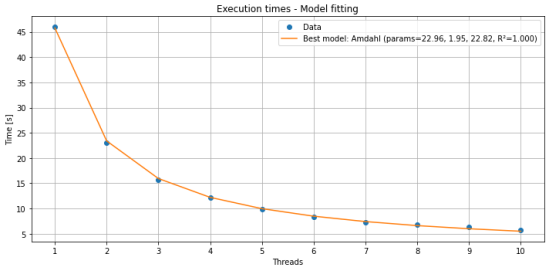


Fig. 22: Execution times with 10000 iterations

In this case of course times are higher since the number of iterations is larger. It is possible to see that all the execution

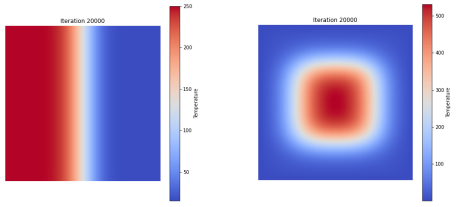
times are 200 times the execution times in the first case where the number of iterations was 50 (200 times smaller). This confirms the fact, already noticed before, that execution times grows linearly with the maximum number of iterations. In this case, as in the previous one, the best fit is calculated and it follows as expected the Amdahl's law with a perfect $R^2 = 1$.

In this section the first main analysis was completed. Firstly the case with a maximum of 50 iterations was presented. It was possible to see a tiny diffusion of the heat due to the very small number of iterations. Execution times (with correspondent efficiency and speedup) were reported for every threads number setup and it was seen a monotonic decreasing trend which assured that resources used were not too many. The trend was then seen to have Amdahl's like behavior, typical for this type of parallel problems. The same analysis was done for the case of 500 and 5000 iterations and the same results were obtained. At the end the case with 10000 iterations was treated, firstly reporting different heatmaps at different steps to visualize the heat distribution. Then execution times were reported and their trend was analyzed, confirming also in this case the Amdahl's law behavior. The hypothesis of linear dependence between execution times and number of iterations was proposed and confirmed by the data.

F. Extra analysis

In this section data of two different extra analysis are reported. The first analysis consists in simulating with a number max of iterations equal to 20000 and 50000. This is done for two main reasons: the first is to see if the simulation remains convenient or if the resources of the cluster are excedeed and the second reason is to see how the heat propagation continues after 10000 iterations. The second analysis instead consists in changing the boundary conditions of the propagation and simulating with 10000 iterations to see which are the most convenient conditions.

Firstly the case of 20000 iterations is considered. The heatmaps for the two cases after 20000 iterations are here reported.



(a) 20000 it., A (b) 20000 it., B

The two heatmaps are similar to the case of 10000 iterations but it is easy to see that the heat spreading is larger in both the cases. Now data about performances for the case A are reported.

Here some interesting things can be noticed. First of all it is possible to see that efficiencies are very low with respect to the other cases, they have high values only for 2,3 and 4 threads but then they reduce a lot showing that maybe the resources

TABLE III: Performances with 20000 iterations max.

Threads	**Time [s]**	**Speedup** $S(p)$	**Efficiency** $E(p)$
1	137.191757	1.0	1.0
2	70.538574	1.94	0.97
3	49.765141	2.76	0.92
4	38.011452	3.61	0.9
5	33.269192	4.12	0.82
6	37.727711	3.64	0.61
7	32.467579	4.23	0.6
8	69.815292	1.97	0.25
9	46.328915	2.96	0.33
10	44.088146	3.11	0.31

of the cluster are excedeed, making the simulation no more effcient. This thing is confirmed if times and speedup values are observed. Indeed times decreases only until 5 threads and after that they don't follow more a monotonic decreasing trend as in the prevoius cases. This demonstrates the fact that the resources of the cluster are not sufficient for this simulation with 20000 iterations. Or better they are sufficient but only if the number of threads is kept under 5.

If data are represented in a graphic way indeed it is notable that the trend is not as 'clean' as before.

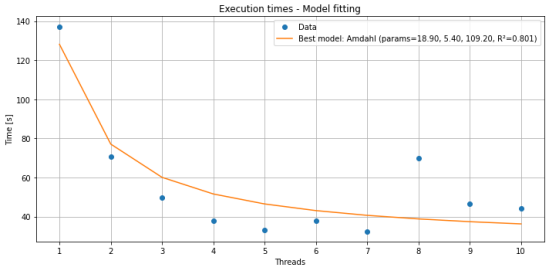
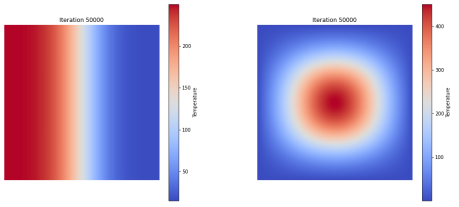


Fig. 24: Execution times with 20000 iterations

The trend is like Amdahl's one only at the beginning, while after the 5th simulation it changes and it is no more analyzeable. The trend results in fact in an Amdahl's function but the coefficient $R^2 = 0.801$ which is a lot smaller then the previous cases and not sufficient to assure that the behavior is actually this.

Then the 50000 iterations simulation have been performed to confirm the idea that the resources had been excedeed. Firstly the two heatmaps are reported to see where the difussion has arrived.



(a) 50000 it., A (b) 50000 it., B

Here the difference is big with respect to the previous case. Indeed in the case A the central line has become broader and the heat has spread in more than half of the grid. In the case B instead the central square is no more visible since the diffusion

has rounded the corners. In this case the diffusion of the heat is higher since the propagation law of the heat in the case B is more aggressive and since the initial gradient of temperature was stronger.

After that data concerning the performances are reported.

TABLE IV: Performances with 50000 iterations max.

Threads	Time [s]	Speedup $S(p)$	Efficiency $E(p)$
1	317.965881	1.0	1.0
2	160.914536	1.98	0.99
3	111.545364	2.85	0.95
4	81.161545	3.92	0.98
5	67.697472	4.7	0.94
6	58.566612	5.43	0.9
7	53.474903	5.95	0.85
8	77.747314	4.09	0.51
9	103.857422	3.06	0.34
10	103.643288	3.07	0.31

First of all the efficiency values are better than the 20000 iterations case. They are indeed very high until 5 threads, they can be considered good for 6 and 7 threads and then they become very low. Times are reduced until 7 threads and then the trend changes. However data are better than the previous case. This is probably done to the fact that a medium problem like 20000 iterations is bigger than an easy one but smaller then a big one so it can have parallelization costs which are higher than calculations. It is possible then that it places in an unfavorable zone where the number of operations is not small but not even high enough to balance parallelization costs. 50000 iterations case instead is much bigger so in this case there are many operations more so parallelization costs become less important. This can be seen also in the graph.

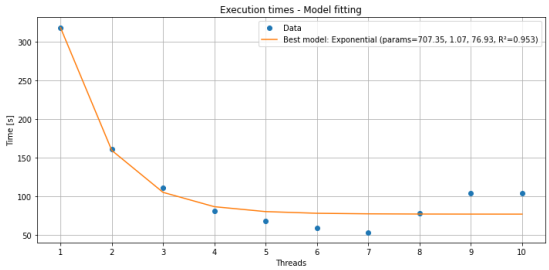


Fig. 26: Execution times with 50000 iterations

The trend is clearly Amdahl's type until the 7 simulation. Then times grows instead of decreasing due to an exceding in the resources usage. The trend is however better because the coeffient $R^2 = 0.953$ which is much higher. If the hypothesis of 5 per cent is used for example, which consists in accepting the trend only if $R^2 > 0.95$ the hypothesis of Amdahl's law can be accepted.

The last analysis consists in using other boundary conditions to see if the performances improves or not. Three new boundary conditions are used: Dirichlet, Von Neumann and periodic boundary conditions. The first ones are conditions which set the edges at certain constant values. In this case this means that edges cells are fixed at the initial values and they are not changed. In the code this means changing the previous conditions and propagation law with:

```
for (int x=1; x<N-2; x++)
for (int y=1; y<N-2; y++)
new_grid[x][y]=0.25f*(grid[x-1][y]+
grid[x+1][y]+grid[x][y-1]+grid[x][y+1]);
```

The second conditions are the Von Neumann ones: in this case the missing neighbor is reflected so every cell has always 4 neighbors. So the code must be modified inserting:

```
int xm = (x==0)   ? 0     : x-1;
int xp = (x==N-1) ? N-1   : x+1;
int ym = (y==0)   ? 0     : y-1;
int yp = (y==N-1) ? N-1   : y+1;
new_grid[x][y] = 0.25f*(grid[xm][y]+
grid[xp][y]+grid[x][ym]+grid[x][yp]);
```

The last boundary conditions are the periodic ones because the two opposite edges of the grid are considered to be in contact, so the neighbor out of the edge is the opposite edge. These conditions are also called Torus conditions since the grid is simulated as it was the torus geometric figure. The code has been modified then in this way:

```
int xm = (x+N-1)%N,;
xp=(x+1)%N;
ym=(y+N-1)%N;
yp=(y+1)%N;
new_grid[x][y] = 0.25f*(grid[xm][y]+
grid[xp][y]+grid[x][ym]+grid[x][yp]);
```

Data for the three different conditions for the case A and in the 10000 iterations simulation are reported.

TABLE V: Performances with 10000 iterations max., Dirichlet B.C.

Threads	Time [s]	Speedup $S(p)$	Efficiency $E(p)$
1	33.585903	1.0	1.0
2	17.106684	1.96	0.98
3	11.642487	2.88	0.96
4	8.669334	3.87	0.97
5	7.493926	4.48	0.90
6	6.366282	5.28	0.88
7	5.742070	5.85	0.84
8	5.055539	6.64	0.83
9	9.545355	3.52	0.39
10	8.746332	3.84	0.38

TABLE VI: Performances with 10000 iterations max., Neumann B.C.

Threads	Time [s]	Speedup $S(p)$	Efficiency $E(p)$
1	62.205605	1.00	1.00
2	32.771236	1.90	0.95
3	23.469122	2.65	0.88
4	17.596888	3.54	0.89
5	14.885243	4.18	0.84
6	13.591388	4.57	0.76
7	12.763610	4.87	0.70
8	10.935977	5.69	0.71
9	16.005981	3.88	0.43
10	15.154691	4.11	0.41

TABLE VII: Performances with 10000 iterations max., periodic B.C.

Threads	Time [s]	Speedup $S(p)$	Efficiency $E(p)$
1	38.289772	1.00	1.00
2	19.363874	1.98	0.99
3	13.218386	2.90	0.97
4	10.093752	3.79	0.95
5	8.408437	4.55	0.91
6	7.180584	5.33	0.89
7	6.243115	6.13	0.88
8	5.661868	6.76	0.85
9	10.519311	3.64	0.40
10	10.375998	3.69	0.37

This data must be compared with data in table II. First of all it is easy to see that Neumann boundary conditions have worse execution times with respect to the initial conditions, while Dirichlet and periodic conditions have better execution times with respect to the initial conditions. In particular Dirichlet conditions seem to be the most efficient. This result was pretty expected for the following reasons: first of all Neumann conditions are pretty expensive because they must perform integer division operations and copy all the edges values. The problem is that the work is done only on the edges cells which are few with respect to the whole grid. Work is then not well parallelized and the overhead is high. This results in the fact that execution times are the highest ones.

Then there are periodic conditions which seem more efficient than the first ones. This can seem strange since these conditions are the ones which require more calculations because every cell (not only edges) is upgraded using integer division. The work in fact in this case is larger but better distributed since all the cells are involved. That's why times better than the initial case are obtained.

Lastly Dirichlet boundary conditions times are the best between the four cases. This was expected since these boundary conditions are the most easiest ones in terms of calculations. Indeed they simply remove the edges cells so less calculations are performed during the simulations.

It can be seen to conclude that all the three new cases have execution times which decrease only until 8 threads simulation while the first initial simulation had all the time decreasing. This is probably due to the fact that the new server (the first simulation was done with the old Legion server) has different configuration and maybe different resources available. Indeed the initial case was then re-simulated in the new server and the same behavior was obtained. It is possible to conclude then that Dirichlet boundary conditions are the most efficient ones.

G. Conclusions

The addressed problem was the simulation of heat diffusion in a metal represented by a 1024x1024 grid, using OpenMP in a HPC server provided by Politecnico of Turin. At the beginning a first introduction on the exercise and one on OpenMP were proposed. After that the analysis made and the code used were explained. The main analysis followed where for different configurations of number of iterations and threads graphs and data were reported. Performance analysis was done using as parameter the execution times, speedup and efficiency. The trend of execution times was seen to have Amdahl's like behavior, typical for this type of parallel problems and the hypothesis of linear dependence between execution times and number of iterations was proposed and confirmed by the data. The distribution of heat was shown with appropriate heatmaps. After that two extra analysis have been included. The first where the maximum number of iterations has been increased and the second changing the boundary conditions of the propagation law. The first analysis was used to show better the heat propagation in a visual way and to understand the cluster resources limits, which were overcome. The second extra analysis instead highlighted the most efficient boundary conditions, which turned out to be the Dirichlet ones.

Algorithm 1: Systolic Matrix Multiplication with MPI

Input: $matrixOrder$, $repetitionNumber$
Output: integer for error evaluation

```
Initialize MPI environment
Determine rank of the process and the size i.e. the number of processes.
Evaluate the dimensions of the systolic array
Create 2D Cartesian grid of processes (the systolic array)
Get process coordinates in the grid
if rank == 0 then
    Read matrices A and B from files
    Initialize output filename
Synchronization barrier
Start timer for the performance analysis
Compute remainder of the matrixOrder divided by the systolic array dimension in both directions
Compute local sub-matrix sizes for the current process
Compute the top right element coordinate of each local sub-matrix
if rank == 0 then
    for each process in first row except rank 0 do
        Send corresponding columns of B to process
    for each process in first column except rank 0 do
        Send corresponding rows of A to process
if process in first column and not rank 0 then
    Receive local rows of A and initialize localA
if process in first row and not rank 0 then
    Receive local columns of B and initialize localBTranspose
if rank == 0 then
    Initialize localA and localBTranspose with own rows and columns
Allocate memory for localC, Mmatrix, Nmatrix, M, N, P and Q
Initialize localC, Mmatrix, and Nmatrix with zeros
```

```
for step = 0 to 3n − 3 do
    if process in first column then
        Extract the element of localA for this step to place into new M column
    if process in first row then
        Extract the element of localBTranspose for this step to place into new N row
    Determine dimensions and indices to send/receive for horizontal and vertical shifts
    if the column element to receive are non zero then
        if the column element to send are non zero then
            Perform a SendRecv to the right and from the left
        else
            Perform a Recv from the left
    else
        if the column element to send are non zero then
            Perform a Send to the right
    if the row element to receive are non zero then
        if the row element to send are non zero then
            Perform a SendRecv downward and upward
        else
            Perform a Recv upward
    else
        if the row element to send are non zero then
            Perform a Send downward
    Update local M matrix by shifting columns right
    Insert received new M column at leftmost column
    Update local N matrix by shifting rows down
    Insert received new N row at top row
    Update localC[i, j] += matrixM[i, j] · matrixN[i, j]
if rank != 0 then
    Send local C to root process
else
    Receive local C from all other processes
    Merge local C into global C using base coordinates
Barrier synchronization
Stop timer
if rank == 0 then
    Save global C to output file
    Print execution time
Free all allocated memory
Finalize MPI
```

TABLE VIII: Execution results for matrices of order 500 on a single node

Processes	**Average time [s]**	**Speedup to reference**	**Speedup** $S(p)$	**Efficency [%]** $E(p)$
1	2.137366	0.177136	1.000000	100.000000
2	1.774326	0.213379	1.204607	60.230357
3	1.394527	0.271493	1.532681	51.089371
4	0.904285	0.418678	2.363599	59.089970
5	0.823819	0.459572	2.594461	51.889214
6	0.738002	0.513013	2.896153	48.269214
7	0.604033	0.626794	3.538492	50.549879
8	0.513102	0.737873	4.165575	52.069685
9	0.449453	0.842367	4.755485	52.838721
10	0.656050	0.577097	3.257932	32.579316
11	0.380592	0.994778	5.615903	51.053662
12	0.370251	1.022560	5.772743	48.106188
13	0.323331	1.170949	6.610456	50.849661
14	0.558739	0.677604	3.825335	27.323824
15	0.258837	1.462711	8.257566	55.050438
16	0.268089	1.412233	7.972596	49.828723
17	0.235820	1.605478	9.063537	53.314923
18	0.400220	0.945991	5.340482	29.669343
19	0.220521	1.716865	9.692360	51.012421
20	0.219421	1.725470	9.740941	48.704704
21	0.228768	1.654970	9.342941	44.490197
22	0.412833	0.917087	5.177310	23.533228
23	0.206919	1.829722	10.329482	44.910789
24	0.199314	1.899536	10.723612	44.681717
25	0.325401	1.163501	6.568408	26.273634
26	0.284763	1.329540	7.505766	28.868332
27	0.158435	2.389655	13.490517	49.964877
28	0.408770	0.926203	5.228774	18.674193
29	0.165123	2.292863	12.944092	44.634800
30	0.284447	1.331018	7.514107	25.047024
31	0.168597	2.245615	12.677359	40.894707
32	0.219015	1.728671	9.759011	30.496911
33	0.212860	1.778655	10.041191	30.427853
34	0.251441	1.505740	8.500481	25.001414
35	0.311506	1.215399	6.861394	19.603982
36	0.152898	2.476186	13.979023	38.830618
37	0.121000	3.128971	17.664240	47.741190
38	0.228564	1.656444	9.351260	24.608578
39	0.124680	3.036600	17.142772	43.955827
40	0.115707	3.272097	18.472243	46.180608
41	0.117025	3.235234	18.264136	44.546673
42	0.246599	1.535302	8.667371	20.636599
43	0.267955	1.412938	7.976580	18.550185
44	0.117359	3.226029	18.212173	41.391301
45	0.116679	3.244836	18.318343	40.707430
46	0.188165	2.012090	11.359023	24.693528
47	0.127070	2.979505	16.820449	35.788189
48	0.112131	3.376448	19.061347	39.711140
49	0.106546	3.553428	20.060462	40.939718
50	0.189799	1.994766	11.261220	22.522440
51	0.202937	1.865625	10.532170	20.651314

Continued on next page

TABLE VIII – continued from previous page

Processes	**Average time [s]**	**Speedup to reference**	**Speedup** $S(p)$	**Efficency [%]** $E(p)$
52	0.215354	1.758058	9.924910	19.086365
53	0.195358	1.938004	10.940776	20.642974
54	0.115179	3.287100	18.556939	34.364702
55	0.105366	3.593233	20.285178	36.882141
56	0.096078	3.940588	22.246131	39.725234
57	0.097617	3.878454	21.895361	38.412914
58	0.208370	1.816984	10.257571	17.685468
59	0.248387	1.524252	8.604990	14.584729
60	0.295421	1.281574	7.234978	12.058297
61	0.236129	1.603380	9.051696	14.838845
62	0.309286	1.224124	6.910653	11.146214
63	0.362737	1.043744	5.892336	9.352915
64	0.747466	0.506517	2.859484	4.467945
65	0.835047	0.453393	2.559576	3.937809
66	1.054948	0.358884	2.026039	3.069757
67	0.559230	0.677010	3.821983	5.704452
68	1.040735	0.363786	2.053709	3.020160
69	1.060908	0.356868	2.014658	2.919794
70	1.132601	0.334279	1.887130	2.695901
71	0.699121	0.541543	3.057218	4.305941
72	1.207055	0.313659	1.770728	2.459344

TABLE IX: Execution results for matrices of order 1000 on a single node

Processes	**Average time [s]**	**Speedup to reference**	**Speedup** $S(p)$	**Efficency [%]** $E(p)$
1	19.944782	0.139977	1	100
4	7.044567	0.396306	2.831229	70.780728
8	4.543562	0.614453	4.389680	54.871000
12	3.057979	0.912958	6.522211	54.351755
16	2.305361	1.211006	8.651479	54.071746
20	1.831222	1.524559	10.891516	54.457579
24	1.494872	1.867589	13.342134	55.592225
28	1.185724	2.354516	16.820760	60.074143
32	2.285887	1.221323	8.725184	27.266199
36	1.012476	2.757405	19.699017	54.719493
40	0.806987	3.459542	24.715116	61.787790
44	0.833332	3.350173	23.933777	54.394948
48	0.802954	3.476918	24.839250	51.748437
52	1.410156	1.979785	14.143672	27.199369
56	0.757231	3.686863	26.339106	47.034118
60	0.660711	4.225456	30.186839	50.311398
64	1.499041	1.862394	13.305025	20.789102
68	1.572360	1.775552	12.684617	18.653849
72	1.719254	1.623848	11.600837	16.112273

TABLE X: Execution results for matrices of order 2000 on a single node

Processes	**Average time [s]**	**Speedup to reference**	**Speedup** $S(p)$	**Efficency [%]** $E(p)$
1	149.614186	0.148482	1.000000	100.000000

Continued on next page

TABLE X – continued from previous page

Processes	**Average time [s]**	**Speedup to reference**	**Speedup** $S(p)$	**Efficency [%]** $E(p)$
4	68.817684	0.322809	2.174066	54.351650
8	36.564962	0.607548	4.091736	51.146705
12	24.986576	0.889076	5.987783	49.898188
16	18.489170	1.201512	8.091990	50.574939
20	13.237481	1.678186	11.302315	56.511577
24	11.125187	1.996816	13.448240	56.034335
28	10.408718	2.134264	14.373930	51.335464
32	9.038353	2.457854	16.553258	51.728932
36	7.804439	2.846451	19.170396	53.251100
40	6.890707	3.223901	21.712459	54.281148
44	6.050050	3.671863	24.729412	56.203210
48	6.148157	3.613271	24.334804	50.697508
52	5.512849	4.029669	27.139179	52.190729
56	6.101848	3.640693	24.519488	43.784799
60	4.744519	4.682235	31.534108	52.556847
64	4.772613	4.654674	31.348485	48.982007
68	4.579797	4.850642	32.668303	48.041622
72	4.646871	4.780626	32.196758	44.717719

TABLE XI: Execution results for matrices of order 500 on two nodes

Processes	**Average time [s]**	**Speedup to reference**	**Speedup** $S(p)$	**Efficency [%]** $E(p)$
1	2.137366	0.177136	1.000000	100.000000
4	1.029445	0.367775	2.076232	51.905797
8	0.510534	0.741584	4.186528	52.331602
12	0.672671	0.562837	3.177431	26.478588
16	0.281154	1.346607	7.602111	47.513192
20	0.409117	0.925419	5.224343	26.121714
24	0.243582	1.554320	8.774732	36.561385
28	0.337058	1.123260	6.341233	22.647262
32	0.200968	1.883903	10.635355	33.235484
36	0.307424	1.231539	6.952511	19.312531
40	0.275803	1.372736	7.749620	19.374049
44	0.280813	1.348244	7.611353	17.298529
48	0.196089	1.930779	10.899990	22.708313
52	0.232870	1.625820	9.178377	17.650726
56	0.580260	0.652473	3.683463	6.577613
60	0.220639	1.715943	9.687159	16.145265
64	0.113910	3.323725	18.763704	29.318287
68	0.213346	1.774603	10.018313	14.732813
72	0.110271	3.433391	19.382813	26.920573

TABLE XII: Execution results for matrices of order 500 with processes equal to nodes

Processes	**Average time [s]**	**Speedup to reference**	**Speedup** $S(p)$	**Efficency [%]** $E(p)$
1	2.370053	0.159745	1.000000	100.000000
4	0.598059	0.633055	3.573841	89.346026
5	0.506532	0.747443	4.219605	84.392108
6	0.422924	0.895206	5.053784	84.229732

TABLE XIII: 1 GPU usage

Image	Blocks	Times [ms]	Mem[MB]	PSNR	PSNR Filtered
resized_4K	8	14.0502	287	361.202	24.109
resized_4K	16	10.0261	287	361.202	24.109
resized_4K	24	9.28355	287	361.202	24.109
resized_4K	32	9.28502	287	361.202	24.109
resized_8K	8	33.7841	335	361.202	25.5588
resized_8K	16	32.3833	335	361.202	25.5588
resized_8K	24	32.6401	335	361.202	25.5588
resized_8K	32	33.1669	335	361.202	25.5588
resized_16K	8	126.333	527	361.202	26.2738
resized_16K	16	124.559	527	361.202	26.2738
resized_16K	24	125.698	527	361.202	26.2738
resized_16K	32	118.705	527	361.202	26.2738

TABLE XIV: 2 GPUs usage

Image	Blocks	Times [ms]	Mem[MB]	PSNR	PSNR Filtered
resized_4K	8	14.3364	287	361.202	24.109
resized_4K	16	10.0215	287	361.202	24.109
resized_4K	24	9.19891	287	361.202	24.109
resized_4K	32	9.42624	287	361.202	24.109
resized_8K	8	32.746	335	361.202	25.5588
resized_8K	16	32.538	335	361.202	25.5588
resized_8K	24	33.2349	335	361.202	25.5588
resized_8K	32	32.5473	335	361.202	25.5588
resized_16K	8	125.906	527	361.202	26.2738
resized_16K	16	123.884	527	361.202	26.2738
resized_16K	24	124.687	527	361.202	26.2738
resized_16K	32	127.37	527	361.202	26.2738

TABLE XV: 3 GPUs usage

Image	Blocks	Times [ms]	Mem[MB]	PSNR	PSNR Filtered
resized_4K	8	14.4354	287	361.202	24.109
resized_4K	16	9.51229	287	361.202	24.109
resized_4K	24	8.83261	287	361.202	24.109
resized_4K	32	8.82051	287	361.202	24.109
resized_8K	8	33.9471	335	361.202	25.5588
resized_8K	16	33.8565	335	361.202	25.5588
resized_8K	24	33.3043	335	361.202	25.5588
resized_8K	32	33.3982	335	361.202	25.5588
resized_16K	8	125.466	527	361.202	26.2738
resized_16K	16	112.903	527	361.202	26.2738
resized_16K	24	113.576	527	361.202	26.2738
resized_16K	32	115.949	527	361.202	26.2738

TABLE XVI: 4 GPUs usage

Image	Blocks	Times [ms]	Mem[MB]	PSNR	PSNR Filtered
resized_4K	8	20.7366	287	361.202	24.109
resized_4K	16	9.40166	287	361.202	24.109
resized_4K	24	8.32298	287	361.202	24.109
resized_4K	32	8.54259	287	361.202	24.109
resized_8K	8	30.6107	335	361.202	25.5588
resized_8K	16	30.7287	335	361.202	25.5588
resized_8K	24	29.8109	335	361.202	25.5588
resized_8K	32	30.1786	335	361.202	25.5588
resized_16K	8	113.137	527	361.202	26.2738
resized_16K	16	112.535	527	361.202	26.2738
resized_16K	24	113.241	527	361.202	26.2738
resized_16K	32	118.568	527	361.202	26.2738

TABLE XVII: 1 GPU usage

Image	Blocks	Times [ms]	Mem[MB]	PSNR	PSNR Filtered
image_noise50	8	13.4778	287	26.3601	23.5988
image_noise50	16	9.40445	287	26.3601	23.5988
image_noise50	24	8.78307	287	26.3601	23.5988
image_noise50	32	9.32413	287	26.3601	23.5988
image_noise75	8	9.36006	287	21.2626	22.5861
image_noise75	16	8.68032	287	21.2626	22.5861
image_noise75	24	8.70554	287	21.2626	22.5861
image_noise75	32	8.74883	287	21.2626	22.5861
image_noise90	8	8.72899	287	18.9769	21.71
image_noise90	16	9.25069	287	18.9769	21.71
image_noise90	24	9.42064	287	18.9769	21.71
image_noise90	32	10.2035	287	18.9769	21.71

TABLE XVIII: 2 GPUs usage

Image	Blocks	Times [ms]	Mem[MB]	PSNR	PSNR Filtered
image_noise50	8	13.4778	287	26.3601	23.5988
image_noise50	16	10.0793	287	26.3601	23.5988
image_noise50	24	9.23907	287	26.3601	23.5988
image_noise50	32	10.547	287	26.3601	23.5988
image_noise75	8	9.06304	287	21.2626	22.5861
image_noise75	16	9.37197	287	21.2626	22.5861
image_noise75	24	9.03104	287	21.2626	22.5861
image_noise75	32	9.40042	287	21.2626	22.5861
image_noise90	8	9.36416	287	18.9769	21.71
image_noise90	16	9.27078	287	18.9769	21.71
image_noise90	24	10.7347	287	18.9769	21.71
image_noise90	32	9.51034	287	18.9769	21.71

TABLE XIX: 3 GPUs usage

Image	Blocks	Times [ms]	Mem[MB]	PSNR	PSNR Filtered
image_noise50	8	14.8231	287	26.3601	23.5988
image_noise50	16	9.68525	287	26.3601	23.5988
image_noise50	24	9.79747	287	26.3601	23.5988
image_noise50	32	9.56621	287	26.3601	23.5988
image_noise75	8	8.65834	287	21.2626	22.5861
image_noise75	16	8.54131	287	21.2626	22.5861
image_noise75	24	8.69402	287	21.2626	22.5861
image_noise75	32	8.76986	287	21.2626	22.5861
image_noise90	8	8.69459	287	18.9769	21.71
image_noise90	16	8.52358	287	18.9769	21.71
image_noise90	24	9.8695	287	18.9769	21.71
image_noise90	32	8.55638	287	18.9769	21.71

TABLE XX: 4 GPUs usage

Image	Blocks	Times [ms]	Mem[MB]	PSNR	PSNR Filtered
image_noise50	8	21.7461	287	26.3601	23.5988
image_noise50	16	9.99421	287	26.3601	23.5988
image_noise50	24	9.47958	287	26.3601	23.5988
image_noise50	32	8.44611	287	26.3601	23.5988
image_noise75	8	8.64506	287	21.2626	22.5861
image_noise75	16	8.36006	287	21.2626	22.5861
image_noise75	24	8.50842	287	21.2626	22.5861
image_noise75	32	8.54144	287	21.2626	22.5861
image_noise90	8	8.65626	287	18.9769	21.71
image_noise90	16	9.61296	287	18.9769	21.71
image_noise90	24	8.48496	287	18.9769	21.71
image_noise90	32	8.464	287	18.9769	21.71