HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY



# PROJECT REPORT

## 2D HEAT EQUATION (CUDA + MPI)

**Course:** Parallel and Distributed Programming

**Supervisor:** Dr. Vu Van Thieu

| *Authors:* | Student ID: |
|---|---|
| Nguyen Duc Binh | 20225475 |
| Tran An Khanh | 20225447 |
| Nguyen Nhu Giap | 20225441 |
| Pham Minh Hieu | 20220062 |
| Truong Tuan Vinh | 20225464 |

Hanoi, June 2025

# ABSTRACT

This project presents parallel implementations of the 2D Heat Equation using both CPU-based and GPU-based strategies. We develop three solutions: two MPI-based versions using row-wise and column-wise domain decomposition, and a CUDA-based GPU implementation. All solutions use an explicit finite difference scheme to model heat diffusion over time. Each approach is evaluated for performance in terms of execution time and scalability. Our results compare how communication patterns, data layout, and hardware architecture affect computational efficiency. The project demonstrates the advantages and limitations of different parallelization strategies and highlights practical considerations for implementing PDE solvers on modern computing systems.

# Contents

# 1 Introduction

The 2D Heat Equation is a widely studied partial differential equation (PDE) that models the diffusion of heat across a two-dimensional surface over time. It serves as a fundamental example in numerical simulation and scientific computing due to its mathematical simplicity and practical relevance in physics, engineering, and environmental science.

Solving the heat equation over a fine spatial grid and for a large number of time steps requires significant computational effort. As the resolution increases, so does the demand for memory and processing power, making sequential solutions impractical for large-scale problems. To address this challenge, parallel programming techniques can be employed to distribute the workload across multiple processing units, reducing execution time and improving scalability.

In this project, we explore and implement multiple parallel solutions to the 2D Heat Equation using both distributed and shared memory models. Specifically, we develop three versions of the solver:

- An MPI-based implementation using **row-wise domain decomposition**,
- An MPI-based implementation using **column-wise decomposition**, and
- A **CUDA-based implementation** leveraging GPU parallelism.

All three versions use an *explicit finite difference scheme*, which updates each grid point based on its immediate neighbors from the previous time step. This scheme is well-suited for parallelization due to its regular data access pattern and local dependencies.

The primary goal of this work is to analyze how different parallelization strategies affect performance, scalability, and computational efficiency. By comparing MPI and CUDA approaches, we aim to highlight the trade-offs between CPU-based distributed computation and GPU-based acceleration. This project contributes practical insights into how problem decomposition, communication patterns, and hardware architecture influence the design and effectiveness of parallel numerical solvers.

# 2 Problem Formulation

## 2.1 PDE formulation

The two-dimensional Heat Equation describes the evolution of temperature over time across a rectangular domain. In continuous form, the equation is given by: [1]

$$\frac{\partial u}{\partial t} = c\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right), \quad 0 \leq x, y \leq 1,\ t \geq 0$$

Here, $u(t, x, y)$ is the temperature at location $(x, y)$ at time $t$, and $c$ is the diffusion coefficient, assumed to be constant across the entire domain.

### 2.2 Initial and boundary conditions

To make the problem well-defined, we specify both an initial condition and fixed (Dirichlet) boundary conditions:

- **Initial condition:** $u(0, x, y) = f(x, y)$, defines the temperature at all points inside the domain at $t = 0$
- **Boundary conditions:**

$$u(t, 0, y) = \alpha_0(y), \quad u(t, 1, y) = \alpha_1(y)$$

$$u(t, x, 0) = \beta_0(x), \quad u(t, x, 1) = \beta_1(x)$$

## 3 Numerical Algorithm

### 3.1 Derivation of update formula

We apply a forward finite difference in time and central differences in space to approximate the heat equation. Let $u_{i,j}^k$ represent the temperature at grid point $(i, j)$ at time step $k$. Then the derivatives are approximated as:

$$\frac{\partial u}{\partial t} \approx \frac{u_{i,j}^{k+1} - u_{i,j}^k}{\Delta t}, \quad \frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1,j}^k - 2u_{i,j}^k + u_{i-1,j}^k}{(\Delta s)^2}, \quad \frac{\partial^2 u}{\partial y^2} \approx \frac{u_{i,j+1}^k - 2u_{i,j}^k + u_{i,j-1}^k}{(\Delta s)^2}$$

Substituting into the PDE yields the explicit update rule:

$$u_{i,j}^{k+1} = u_{i,j}^k + \frac{c\Delta t}{(\Delta s)^2} \left( u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k - 4u_{i,j}^k \right)$$

### 3.2 Stability condition

The explicit scheme is conditionally stable. To ensure the numerical solution remains bounded and accurate, the time step must satisfy the CFL (Courant–Friedrichs–Lewy) condition:

$$\Delta t \leq \frac{(\Delta s)^2}{4c}$$

Violating this condition can result in numerical instability and incorrect results.

### 3.3 Grid and timestep definitions

We discretize the unit square domain into a grid of $(n + 2) \times (n + 2)$ points, where the indices $i = 0$ and $i = n + 1$, as well as $j = 0$ and $j = n + 1$, represent fixed boundary values. The interior points $(1 \leq i, j \leq n)$ are updated at every time step.

The spatial step is:

$$\Delta s = \frac{1}{n+1}$$

The time step $\Delta t$ is selected based on the stability condition above. The simulation proceeds iteratively for a fixed number of time steps, updating the temperature field using the explicit formula at each step.

# 4 Solution Methods

This section presents two solution methods for solving the 2D Heat Equation: the explicit Finite Difference Method and the Jacobi Iterative Method. Each method is implemented using multiple programming strategies including sequential, MPI, and CUDA. The choice of parallelization affects how the computational domain is divided, how data dependencies are handled, and how communication and memory are managed across processing units.

## 4.1 Finite Difference Method

The Finite Difference Method (FDM) provides an explicit numerical scheme to approximate the solution of the two-dimensional Heat Equation. The continuous equation is:

$$\frac{\partial u}{\partial t} = c\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right), \quad 0 \le x, y \le 1,\ t \ge 0$$

To apply the method, the domain $[0,1]\times[0,1]$ is discretized into a uniform grid with spatial resolution $\Delta s = \frac{1}{N+1}$, where $N$ is the number of interior grid points in one dimension. Time is also discretized into steps of size $\Delta t$.

Each grid point $(x_i, y_j)$ is represented by indices $i, j$, and the solution at time step $k$ is denoted by $u^k_{i,j}$. The second-order spatial derivatives are approximated by central differences, and the time derivative is approximated using a forward difference. This leads to the update rule:

$$dU = u^k_{i+1,j} + u^k_{i-1,j} + u^k_{i,j+1} + u^k_{i,j-1} - 4u^k_{i,j}$$

$$u^{k+1}_{i,j} = u^k_{i,j} + \frac{c\Delta t}{(\Delta s)^2} dU$$

$$u^{k+1}_{i,j} = u^k_{i,j} + \frac{c\Delta t}{(\Delta s)^2}\left(u^k_{i+1,j} + u^k_{i-1,j} + u^k_{i,j+1} + u^k_{i,j-1} - 4u^k_{i,j}\right)$$

This formula is applied to all interior points, while boundary values remain fixed according to the Dirichlet boundary conditions.

The method is conditionally stable. To avoid numerical divergence, the time step $\Delta t$ must satisfy the following stability constraint:

$$\Delta t \leq \frac{(\Delta s)^2}{4c}$$

This ensures that the influence of each neighboring value is proportionally small enough to maintain the integrity of the numerical solution.

The Finite Difference Method is well-suited for time-marching problems like the Heat Equation due to its simple stencil and locality of data access. The explicit nature of the scheme also makes it easy to parallelize, as each update depends only on values from the previous time step.

## 4.2 Jacobi Method

The Jacobi Method is an iterative algorithm used to find a steady-state solution from the discretization of partial differential equations such as the 2D Heat Equation. Unlike the explicit Finite Difference Method, the Jacobi Method is derived from a steady-state or time-independent formulation, but can also be used in iterative solvers for transient problems.

In the context of the 2D Heat Equation, the domain is discretized into a grid of $(N+2) \times (N+2)$ points, including boundary values. Let $u_{i,j}^k$ denote the approximation at iteration $k$ and grid point $(i, j)$. The update rule is defined as the average of the four neighboring values from the previous iteration:

$$u_{i,j}^{k+1} = \frac{1}{4}\left(u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k\right)$$

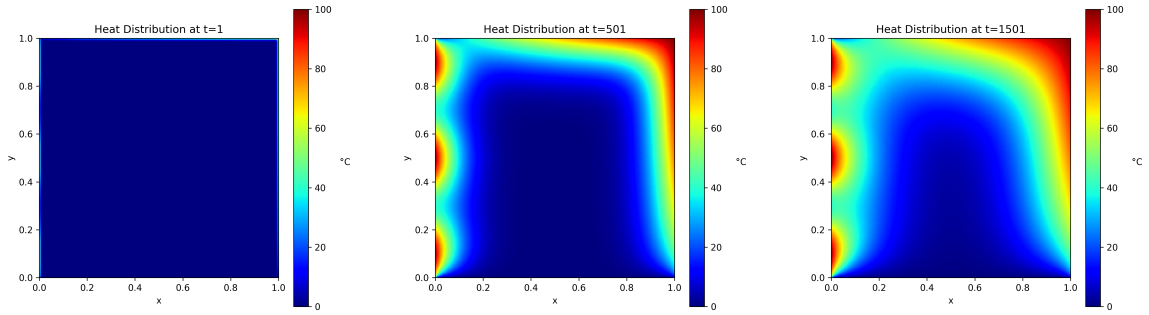
This update is applied to all interior grid points $1 \leq i, j \leq N$, while the values at the boundaries are fixed and determined by Dirichlet conditions.

Convergence of the Jacobi Method is not guaranteed in all cases, but for diagonally dominant matrices such as those arising from the 2D Laplace operator, convergence is typically achieved over a sufficient number of iterations. A convergence criterion may be defined as:

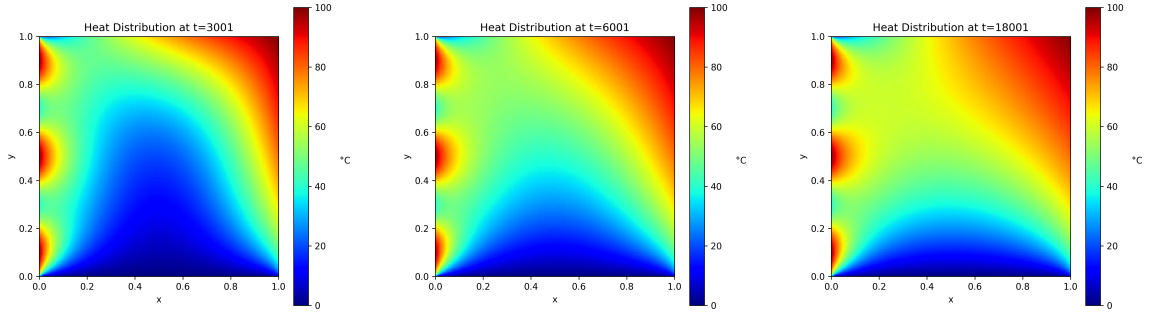
$$\max_{i,j} \left|u_{i,j}^{k+1} - u_{i,j}^k\right| < \epsilon$$

for a chosen tolerance $\epsilon$, or alternatively, a fixed number of iterations may be performed.

The Jacobi Method is naturally parallelizable due to the independence of updates at each grid point, since all values used in iteration $k+1$ are read from iteration $k$.




(a) t=1 (b) t=501 (c) t=1501




(d) t=3001 (e) t=6001 (f) t=18001

Figure 1: Visualization of the 2D heat equation following the finite difference method. t is the time step in the simulation. $u(0, y, t) = 100\sqrt{y}$, $u(x, 1, t) = 100(0.7 + 0.3 \sin 5\pi x)$, $u(1, y, t) = 0$, $u(x, 0, t) = 100\sqrt[3]{x}$.

# 5 Programming Methods

## 5.1 MPI Row-wise Decomposition

In this approach, the 2D domain is divided along the **row axis**. With $P$ MPI processes, each handles a block of $N_c = \frac{N}{P}$ rows, resulting in a local subdomain of size $N_c \times N$.

### 5.1.1 Domain Decomposition and Data Distribution

The root process initializes the full grid and distributes it among processes using `MPI_Scatter`. Each process receives its slice into a local buffer and prepares a separate array for storing updated values.

### 5.1.2 Boundary Communication

Each process must exchange its top and bottom rows with neighbors to compute values near the internal boundaries. This is performed using `MPI_Sendrecv`. Processes at the physical boundaries use fixed Dirichlet values in place of missing neighbors. Ghost rows are updated before each computation step.

### 5.1.3 Local Temperature Update

Interior points are updated using the explicit finite difference formula:

$$u_{i,j}^{k+1} = u_{i,j}^{k} + \frac{c\Delta t}{(\Delta s)^2}\left(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} - 4u_{i,j}^{k}\right)$$

Ghost row values are used when accessing neighbors outside the subdomain bounds. The update is applied independently to all interior points.

### 5.1.4 Iteration and Synchronization

Each time step involves:

- Exchanging ghost rows.
- Computing new values
- Copying the updated array

### 5.1.5 Final Gathering

Once all steps are completed, each process sends its local grid back to the root using `MPI_Gather`, which reassembles and outputs the global result.

## 5.2 MPI Column-wise Decomposition

In this approach, the 2D domain is divided along the **column axis**. Each of the $P$ MPI processes is assigned $N_c = \frac{N}{P}$ columns, resulting in a local subdomain of size $N \times N_c$.

### 5.2.1 Domain Decomposition and Data Distribution

Because the data is stored in row-major format, vertical blocks require derived MPI datatypes. These types allow processes to scatter and gather column blocks using non-contiguous memory layout. Each process stores its portion of the domain in local buffers for the current and updated values.

### 5.2.2 Boundary Communication

Each process exchanges its left and right boundary columns with neighboring processes. Communication is done using a custom column-type MPI datatype. At the left and right physical boundaries, fixed Dirichlet values are substituted for missing neighbors.

### 5.2.3 Local Temperature Update

The finite difference stencil is the same as in the row-wise method, but index access is adapted to the column-major subdomains. Each thread uses local neighbors for rows and ghost columns for left/right access.

$$u_{i,j}^{k+1} = u_{i,j}^{k} + \frac{c\Delta t}{(\Delta s)^2}\left(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} - 4u_{i,j}^{k}\right)$$

#### 5.2.4 Iteration and Synchronization

Each iteration includes:

- Exchanging ghost columns
- Updating interior values
- Swapping or copying arrays

The iteration loop structure is identical to the row-wise method.

#### 5.2.5 Final Gathering

At the end, each process sends its vertical block back to the root using the same MPI derived datatype. The global matrix is reassembled for output.

### 5.3 CUDA-based GPU Parallelization

In this approach, we use NVIDIA CUDA to accelerate the 2D Heat Equation solver by mapping each grid point to a separate GPU thread. The explicit finite difference scheme is executed in parallel on the device using a 2D grid of threads.

#### 5.3.1 Memory Layout and Grid Configuration

The temperature field is stored in global memory as two 1D arrays: one for the current step and one for the next. These arrays are allocated on the GPU and initialized from host memory. Each thread computes its grid coordinates from block and thread indices.

#### 5.3.2 Boundary Handling

Threads corresponding to boundary points return immediately to preserve fixed Dirichlet boundary values. Only interior grid points are updated during each kernel call.

#### 5.3.3 Temperature Update Formula

Each thread applies the following update rule, using its four direct neighbors:

$$u_{i,j}^{k+1} = u_{i,j}^{k} + \frac{c\Delta t}{(\Delta s)^2}\left(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} - 4u_{i,j}^{k}\right)$$

Values are loaded directly from the global memory arrays, and results are written to a separate output array.

#### 5.3.4 Iteration Strategy

The simulation runs for a fixed number of steps. The host launches the CUDA kernel in each step and swaps the device arrays afterward. This ensures that the new values become the input for the next step.

#### 5.3.5 Result Collection

After the final iteration, the computed temperature field is copied back from the device to the host for output or further analysis.

# 6 Experimental Setup

## 6.1 Programs

We coded a total of 8 programs:

- Finite Difference – sequential
- Finite Difference – MPI row
- Finite Difference – MPI column
- Finite Difference – CUDA
- Jacobi – sequential
- Jacobi – MPI row
- Jacobi – MPI column
- Jacobi - CUDA

## 6.2 Computational Resources

All experiments were conducted on a machine with the following hardware specifications:

- **Sequential Programming & MPI:** Executed on the CPU – **AMD Ryzen 7 5800H**.
- **CUDA Implementation:** Executed on the GPU – **NVIDIA RTX 4070**.

# 7 Results and Analysis

| Computation Time (seconds) | | | | | |
|---|---|---|---|---|---|
| | N=10 | N=20 | N=50 | N=250 | N=1250 |
| p=1 | 0.019 | 0.058 | 0.272 | 5.997 | 144.518 |
| p=2 | 0.030 | 0.050 | 0.182 | 3.865 | 102.512 |
| p=5 | 0.027 | 0.049 | 0.104 | 1.834 | 45.080 |
| p=10 | 0.033 | 0.043 | 0.085 | 1.197 | 30.864 |

Table 1: Computation time for varying grid sizes and process counts

| Calculated Speedup | | | | | |
|---|---|---|---|---|---|
| | N=10 | N=20 | N=50 | N=250 | N=1250 |
| $S_{1,2}$ | 0.633 | 1.160 | 1.495 | 1.552 | 1.410 |
| $S_{1,5}$ | 0.704 | 1.184 | 2.615 | 3.270 | 3.206 |
| $S_{1,10}$ | 0.576 | 1.349 | 3.200 | 5.010 | 4.682 |

Table 2: Calculated speedup of Finite Difference method using MPI column-wise decomposition

Table 2 presents the computation times (in seconds) for solving the Finite Difference Method using MPI with varying numbers of grid points ($N$) and processes ($p$). The parameter $N$ denotes the number of grid points in the computational domain, reflecting the size of the problem. The parameter $p$ represents the number of MPI processes used to perform the computation in parallel. As expected, for small problem sizes (e.g., $N = 10$), increasing the number of processes does not significantly reduce computation time and may even introduce overhead due to inter-process communication. However, for larger problem sizes (e.g., $N = 1250$), increasing the number of processes leads to a noticeable reduction in computation time, illustrating the effectiveness of parallelization using MPI.

Table 3 shows the calculated speedup obtained by using multiple processes, denoted by $S_{i,j}$, where $S_{i,j}$ represents the speedup achieved when increasing the number of processes from $i$ to $j$. These values are computed as the ratio of the computation time using $i$ processes to the time using $j$ processes. For instance, $S_{1,5}$ corresponds to the speedup when using 5 processes compared to a single process. The results demonstrate that the speedup becomes more pronounced as the problem size increases. For example, $S_{1,10}$ reaches a value of 4.682 for $N = 1250$, indicating a significant improvement in performance when using 10 processes instead of 1. This illustrates that parallelization via MPI becomes increasingly beneficial for larger-scale problems.

| Computation Time (seconds) | | | | | |
|---|---|---|---|---|---|
| | N=10 | N=20 | N=50 | N=250 | N=1250 |
| p=1 | 0.019 | 0.058 | 0.272 | 5.997 | 144.518 |
| p=2 | 0.023 | 0.036 | 0.121 | 2.337 | 58.823 |
| p=5 | 0.027 | 0.030 | 0.094 | 1.124 | 27.583 |
| p=10 | 0.027 | 0.035 | 0.067 | 0.917 | 23.308 |

Table 3: Computation time of Finite Difference method using MPI column-wise decomposition

| Calculated Speedup | | | | | |
|---|---|---|---|---|---|
| | N=10 | N=20 | N=50 | N=250 | N=1250 |
| $S_{1,2}$ | 0.826 | 1.611 | 2.248 | 2.566 | 2.457 |
| $S_{1,5}$ | 0.704 | 1.933 | 2.894 | 5.335 | 5.239 |
| $S_{1,10}$ | 0.704 | 1.657 | 4.060 | 6.540 | 6.200 |

Table 4: Calculated speedup of Finite Difference method using MPI row-wise decomposition

| Computation Time (seconds) | | | | | |
|---|---|---|---|---|---|
| | N=10 | N=20 | N=50 | N=250 | N=1250 |
| p=1 | 0.016 | 0.040 | 0.197 | 3.638 | 88.588 |
| p=2 | 0.029 | 0.041 | 0.126 | 2.410 | 73.210 |
| p=5 | 0.027 | 0.036 | 0.081 | 1.303 | 28.967 |
| p=10 | 0.028 | 0.040 | 0.072 | 1.041 | 23.931 |

Table 5: Computation time for the column-wise MPI implementation of the Jacobi method

| Calculated Speedup | | | | | |
|---|---|---|---|---|---|
| | N=10 | N=20 | N=50 | N=250 | N=1250 |
| $S_{1,2}$ | 0.552 | 0.976 | 1.563 | 1.510 | 1.210 |
| $S_{1,5}$ | 0.593 | 1.111 | 2.432 | 2.792 | 3.058 |
| $S_{1,10}$ | 0.571 | 1.000 | 2.736 | 3.495 | 3.702 |

Table 6: Calculated speedup of Jacobi method using MPI column-wise decomposition

| Computation Time (seconds) | | | | | |
|---|---|---|---|---|---|
| | N=10 | N=20 | N=50 | N=250 | N=1250 |
| p=1 | 0.016 | 0.040 | 0.197 | 3.638 | 88.588 |
| p=2 | 0.026 | 0.034 | 0.115 | 2.472 | 59.709 |
| p=5 | 0.025 | 0.029 | 0.080 | 1.069 | 28.469 |
| p=10 | 0.029 | 0.034 | 0.066 | 0.900 | 24.586 |

Table 7: Computation time of Jacobi method using MPI row-wise decomposition

| Calculated Speedup | | | | | |
|---|---|---|---|---|---|
| | N=10 | N=20 | N=50 | N=250 | N=1250 |
| $S_{1,2}$ | 0.615 | 1.176 | 1.713 | 1.472 | 1.484 |
| $S_{1,5}$ | 0.640 | 1.379 | 2.463 | 3.403 | 3.112 |
| $S_{1,10}$ | 0.552 | 1.176 | 2.985 | 4.042 | 3.603 |

Table 8: Calculated speedup of Jacobi method using MPI row-wise decomposition

| Computation Time (seconds) | | | | | |
|---|---|---|---|---|---|
| | N=10 | N=20 | N=50 | N=250 | N=1250 |
| Sequential | 0.019 | 0.058 | 0.272 | 5.997 | 144.518 |
| BlockSize=2 | 1.111 | 1.071 | 1.072 | 1.564 | 13.287 |
| BlockSize=4 | 0.940 | 1.047 | 0.960 | 1.218 | 4.214 |
| BlockSize=16 | 1.030 | 1.016 | 1.035 | 1.137 | 2.873 |

Table 9: Computation time for the CUDA Finite Difference method with varying block sizes.

| Computation Time (seconds) | | | | | |
|---|---|---|---|---|---|
| | N=10 | N=20 | N=50 | N=250 | N=1250 |
| Sequential | 0.019 | 0.058 | 0.272 | 5.997 | 144.518 |
| BlockSize=2 | 0.999 | 0.995 | 1.041 | 1.595 | 13.316 |
| BlockSize=4 | 1.119 | 1.065 | 1.065 | 1.314 | 4.285 |
| BlockSize=16 | 1.142 | 1.118 | 1.051 | 1.163 | 2.847 |

Table 10: Computation time and calculated speedup for the 2D heat equation using the CUDA Finite Difference method with shared memory and varying block sizes.

**Performance on Small Problem Sizes ($\mathbf{N \leq 50}$).** For small values of $N$ (e.g., $N \leq 50$), the sequential program often outperforms parallel models, including both MPI and CUDA-based methods. This is mainly due to the overhead time associated with parallel execution. In

MPI, this overhead comes from the communication cost of transferring data between processes. In CUDA, it arises from memory transfer time between the host and device. As a result, the sequential approach is more efficient for smaller datasets where the communication or memory transfer overhead dominates the computation time.

**Performance on Large Problem Sizes (N $\geq$ 250).** As $N$ increases (e.g., $N \geq 250$), the overhead incurred by parallel models becomes negligible compared to the performance gain from parallel computation. In these cases, both MPI and CUDA-based models outperform the sequential model significantly due to their ability to parallelize computations across multiple processors or GPU threads.

**Comparison Between MPI Row-based and Column-based Methods.** MPI methods perform more effectively when they utilize a larger number of processes and when $N$ is sufficiently large. The MPI Row-based method generally outperforms the MPI Column-based method. This is due to the overhead involved in the MPI Column-based approach, which requires additional intermediate data structures for execution. In contrast, the MPI Row-based approach only involves communication of data along rows, eliminating unnecessary processing time and reducing overhead.

**Comparison Between CUDA Finite Difference and CUDA Jacobi Methods.** CUDA performance improves with larger block sizes, as more GPU threads can be utilized efficiently. When $N = 1250$, CUDA significantly outperforms both MPI and sequential models, achieving up to 15x speedup over MPI and up to 100x speedup over the sequential implementation. The CUDA Jacobi method performs slightly better than the CUDA Finite Difference method. This is attributed to the fact that the Finite Difference method involves more complex computations per iteration, leading to increased execution time.

# 8 Conclusion

## 8.1 Insights

This project demonstrates that parallelization significantly improves the computational efficiency of solving the 2D heat equation. CUDA offers a clear advantage over MPI in terms of computational time, especially for larger grid sizes. Furthermore, within MPI implementations, increasing the number of processes leads to improved performance, with row-wise decomposition consistently outperforming column-wise decomposition. These results highlight the importance of choosing the right parallelization strategy based on the problem size and system architecture.

## 8.2 Future Work

Future work can explore hybrid parallelization approaches that combine MPI and CUDA to leverage both distributed and shared memory models. Additionally, dynamic load balancing and

adaptive decomposition strategies could further enhance performance for non-uniform grids or time-dependent problems. Investigating performance on heterogeneous architectures, including multi-GPU and cloud-based systems, will also be essential for scaling parallel solutions to more complex simulations of the heat equation.

# References

[1] V. Horak and P. Gruber, "Parallel numerical solution of 2-d heat equation," in *Parallel Numerics '05* (M. Vajtersic, R. Trobec, P. Zinterhof, and A. Uhl, eds.), (Salzburg, Austria), pp. 47–56, University of Salzburg, 2005. Chapter 3: Differential Equations.

∎