2D HEAT TRANSFER SIMULATION

Parallel Programming

Layla Maassarani (201865) – Noumat Khatib (220966) – Ali Joumaa (211058) – Farouk Tannir (201153)

Contents

Introduction

The growth of computational demands in scientific and engineering applications has made parallel computing an essential approach for solving large-scale numerical problems efficiently. Among these problems, heat transfer simulations are widely used to model physical phenomena in fields such as thermal engineering, materials science, and environmental modeling. However, as spatial resolution and simulation time increase, sequential implementations become computationally expensive and inefficient.

Within this project, a parallel solution for a two-dimensional transient heat equation based on the Message Passing Interface (MPI) technique was implemented. A discretized domain based on a finite difference solution with smaller domains was assigned to be processed on separate MPI processes. As a result, domain decomposition enables more efficient simultaneous computation based on several MPI processes.

To make it correct and consistent across subdomains, halo or ghost cells were incorporated for temp values exchanged between neighbor processes at every time step. Non-blocking MPI calls were incorporated for overlapping communications with computation. All internal and boundary cells were updated based on the heat equation discretized and within imposed boundary conditions.

The project illustrates that techniques of parallel programming can be very useful and efficient for solution of numerical simulations. It emphasizes domain decomposition, message passing concepts, and designs with consideration for performance.

Mathematical Model and Discretization

The physical phenomenon studied in this project is two-dimensional transient heat conduction, which describes the temporal evolution of temperature within a material due to thermal diffusion. The governing equation is given by the 2D heat equation:

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)$$

where $T(x, y, t)$ represents the temperature field as a function of space and time, and α is the thermal diffusivity constant of the material.

T is a function of space and time. It will vary based on 2 different metrics: space, in which we will be progressing in a 2D dimension grid, and time, which is the goal of our simulation: calculate the propagation of heat with each passing unit of time.

Based on that, our final function T will take 3 parameters:

- x and y that represent geo-spatial variations
- t for the temporal propagation

A point on a grid will have (x, y) coordinates. A point on a grid will have its temperature vary with each passing unit of time t.

Numerical Analysis

To solve this issue numerically, we need to find the closest equivalence to the problem.

We will conduct our analysis in a spatial domain.

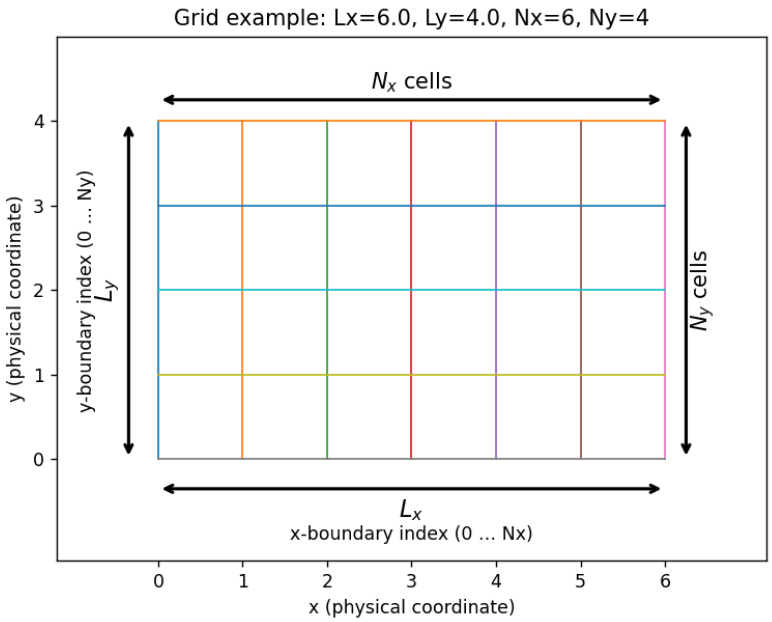
Consider the following:

$Spatial\ Domain: \Omega = \{(x, y); 0 \leq x \leq L_x \text{ and } 0 \leq y \leq L_y$ knowing that L_x and L_y are the dimensions of the grid (supposed rectangular)

$Temporal\ Domain: 0 \leq t \leq T_{final}$

$\alpha: constant\ thermal\ diffusion\ between\ each\ block\ during\ time\ t$

Grids in Space



Where L_x represents the longitude of the grid and L_y the latitude.

N_x represents the number on the x-axis for our grid and N_y represents the number on the y-axis.

Let $\Delta x = \frac{L_x}{N_x - 1}$ and $\Delta y = \frac{L_y}{N_y - 1}$; Δx and Δy represent the steps inside of the grid in the spatial domain. We note:

$$x_i = i\Delta x; \quad i = 0, 1, \dots, \mathrm{N_x} - 1$$

$$\mathrm{y}_i = i\Delta \mathrm{y}; \quad i = 0, 1, \dots, \mathrm{N_y} - 1$$

$\mathrm{x_i}$ and $\mathrm{y_i}$ represent the discrete positions of the x and y coordinates according to each step. Since we are using finite differences, it is essential to change the perspective of our problem from a continuous form to a discrete one.

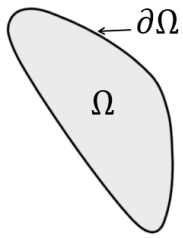
Let $\mathrm{T_{i,j}^n} = \mathrm{T}(\mathrm{x_i}, \mathrm{y_j}, \mathrm{t_n})$; $\mathrm{t_n} = \mathrm{n}\Delta\mathrm{t}$

Initial Condition

At $\mathrm{t} = 0$, the temperature is known everywhere: $T(x, y, 0) = \mathrm{T_0}(x, y), \forall (x, y) \in \Omega$

Boundary Conditions

Let $\partial\Omega$ be the boundary of the grid.



We need to determine the conditions applied to boundaries beforehand since we are dealing with a boundary value problem. A boundary value problem is a differential equation (or system of differential equations) to be solved in a domain on whose boundary (a set of conditions) is known. Boundary conditions are essential to establishing because without them, a problem might diverge.

Dirichlet Boundary Conditions

This condition specifies the value that the unknown function needs to take on along the boundary of the domain. In our case, for the LaPlace equation that we have, our problem is defined as follows:

$$\Delta\varphi(\underline{\mathrm{x}}) = 0 \quad \forall \underline{\mathrm{x}} \in \Omega$$

$$\varphi(\underline{x}) = f(\underline{x}) \quad \forall \underline{x} \in \partial\Omega$$

where φ is the unknown function, $\underline{x}$ is the independent variable (in our case, the spatial coordinates), Ω is the function domain, $\partial\Omega$ is the boundary of the domain, and f is a given scalar function defined on $\partial\Omega$. In the context of heat transfer, if we look at the steady state of the heat equation

$$\frac{\partial T}{\partial t} = \alpha \Delta T,$$

then $\frac{\partial T}{\partial t} = 0$ and the PDE reduces to Laplace's equation $\Delta T = 0$ inside the domain. Dirichlet conditions then prescribe the temperature on the edges:

$$T(x, y, t) = g(x, y, t) \; for (x, y) \in \partial\Omega_D.$$

In our project, we choose g to be constant on each edge (for example $T_{\text{left}}, T_{\text{right}}, T_{\text{top}}, T_{\text{bottom}}$), so each wall is kept at a fixed temperature during the evolution. The interior temperature changes over time, but the boundary values remain imposed.

$Left\ side\ of\ the\ grid: T(0, y, t) = T_{leeft}$

$Right\ side\ of\ the\ grid: T(L_x, y, t) = T_{right}$

$Bottom\ side\ of\ the\ grid: T(x, 0, t) = T_{bottom}$

$Top\ side\ of\ the\ grid: T\big(x, L_y, t\big) = T_{top}$

Neumann Boundary Conditions

When imposed on an ordinary or a partial differential equation, it specifies the values that the derivative of a solution is going to take on the boundary of the domain. In our case, the boundary value problem with Neumann. is written as:

$$\Delta\varphi\big(\underline{x}\big) = 0 \quad \forall \underline{x} \in \Omega$$

$$\frac{\partial\varphi(\underline{x})}{\partial n} = f\big(\underline{x}\big) \quad \forall \underline{x} \in \partial\Omega$$

with $\frac{\partial\varphi(\underline{x})}{\partial \mathrm{n}}$ being a normal derivative in the direction of the boundary. In simple terms, a normal derivative is a perpendicular derivative to the direction of the boundary edge and points outwards of the domain. In analogy with our problem, considering an edge is a boundary limit point giving the temperature value at this point at a time *t*, the normal derivative of the temperature is directly related to the heat flux through the boundary via Fourier's law: $\mathrm{q_n} = -\mathrm{k}\frac{\partial \mathrm{T}}{\partial \mathrm{n}}$.

Prescribing $\frac{\partial T}{\partial n}$ therefore means prescribing the heat flux across the wall. A very important special case is the insulated wall, where no heat passes through the boundary:

$$\frac{\partial T}{\partial n} = 0 \Rightarrow q_n = 0.$$

In that case, the wall does not exchange heat with the environment: the temperature at the boundary can still evolve in time, but the net heat flux through the boundary is zero.

Partial Differential Equations (PDEs)

We will part from this initial equation:

$$\frac{\partial T}{\partial t}(x,y,t) = \alpha\left(\frac{\partial^2 T}{\partial x^2}(x,y,t) + \frac{\partial^2 T}{\partial y^2}(x,y,t)\right) \; with\; T: Laplacian\; of\; T$$

The element on the left represents the temporal diffusion (variation with time).

The second element represents the spatial diffusion.

Spatial discretization – Finite differences

Using finite difference method in Numerical Analysis, we will derive a function to be represented on our local system.

We will begin with Taylor expansion.

General Formula

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \cdots$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \cdots$$

First Derivative Approximation using forward difference

Using the first Taylor expansion formula:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + O(h^3)$$

$$f'(x) = \frac{f(x+h) - f(x) - \frac{h^2}{2}f''(x) + O(h^3)}{h}$$

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(x) + O(h^2)$$

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad error\; of\; first\; order$$

First Derivative Approximation using backward difference

Using the second Taylor expansion formula:

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) + O(h^3)$$

$$f'(x) = \frac{-f(x-h) + f(x) + \frac{h^2}{2}f''(x) + O(h^3)}{h}$$

$$f'(x) = \frac{-f(x+h) + f(x)}{h} + \frac{h}{2}f''(x) + O(h^2)$$

$$f'(x) \approx \frac{f(x) - f(x-h)}{h} \quad error\ of\ first\ order$$

Central Difference using the Two Taylor Formulas

Apply subtraction between the two Taylor Formulas

$$-\begin{cases} f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5) \\ f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5) \end{cases}$$

We obtain the following result:

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{h^3}{3}f'''(x) + O(h^5)$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{3}f'''(x) + O(h^4) \quad \text{knowing that } \frac{h^2}{3}f'''(x) + O(h^4) \text{ is a much smaller error rate}$$

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \text{ with an error rate of order 2 (smaller error)}$$

→ better method of calculation

Going back to the main equation: $\frac{\partial T}{\partial t}(x,y,t) = \alpha\left(\frac{\partial^2 T}{\partial x^2}(x,y,t) + \frac{\partial^2 T}{\partial y^2}(x,y,t)\right)$

We are required to find the derivative of the second order to obtain the approximation of this solution. To do so, we need to add the previous two Taylor expansions:

$$+\begin{cases} f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5) \\ f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5) \end{cases}$$

$$f(x+h)+f(x-h)=2f(x)+\ h^2f''(x)+\frac{h^4}{12}f^{(4)}(x)+O(h^6)$$

$$f''(x)=\frac{-2f(x)+f(x-h)+f(x+h)}{h^2}+\frac{h^2}{12}f^{(4)}(x)+O(h^5)$$

$$f''(x)\approx\frac{f(x+h)-2f(x)+f(x-h)}{h^2}\ \text{ error of order 2}$$

Synchronization with our problem

Spatial Partial Derivatives

Based on the formula that we have obtained for the derivative of the second order, we can deduce the value of the partial differentials of x and y of the order 2.

Noting that we have the following spatial-temporal function:
$T(x,y,t), for\ a\ specific\ point\ (x_i,y_j,t_n), we\ note\ T(x_i,y_j,t_n)=T_{i,j}^n$

Based on the previous second order derivative function that we have concluded, we have the following:

$$at\ a\ specific\ time, and\ at\ a\ point\ (x_i,y_j), with\ varying\ x$$

$$\frac{\partial^2T}{\partial x^2}(x_i,y_j,t_n)\approx\frac{T_{i+1,j}^n+T_{i-1,j}^n-2T_{i,j}^n}{\Delta x^2}$$

With the same analogy, for the partial second derivative, we note

$$at\ a\ specific\ time, and\ at\ a\ point\ (x_i,y_j), with\ varying\ y$$

$$\frac{\partial^2T}{\partial y^2}(x_i,y_j,t_n)\approx\frac{T_{i,j+1}^n+T_{i,j-1}^n-2T_{i,j}^n}{\Delta y^2}$$

Temporal Partial Derivatives

Here, we use the first Taylor expansions to find the first partial derivative. We have the option to use either the forward or backward differences:

$$\frac{\partial T}{\partial t}(x_i,y_j,t_n)\approx\frac{T_{i,j}^{n-1}-T_{i,j}^n}{\Delta t}\approx\frac{T_{i,j}^{n+1}-T_{i,j}^n}{\Delta t}$$

Final Heat Equation

With proper substitution in $\frac{\partial T}{\partial t}=\ \alpha\ (\frac{\partial^2T}{\partial x^2}+\frac{\partial^2T}{\partial y^2})$ with our approximations, we obtain:

$$\frac{T_{i,j}^{n+1}-T_{i,j}^n}{\Delta t}=\alpha\left[\frac{T_{i+1,j}^n-2T_{i,j}^n+T_{i-1,j}^n}{\Delta x^2}+\frac{T_{i,j+1}^n-2T_{i,j}^n+T_{i,j-1}^n}{\Delta y^2}\right]$$

$$T_{i,j}^{n+1} = T_{i,j}^{n} + \alpha\Delta t\left[\frac{T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}}{\Delta x^2} + \frac{T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}}{\Delta y^2}\right]$$

Solution Convergence

As we have seen, the values of the derivatives are not exactly equal to the differences we have obtained. We always have a marge of errors that we cannot escape. We have made our analysis to obtain the optimal error possible, which is a second order error.

However, this is not enough to guarantee that our solution will converge. Since in ∞, our solution might explode if no attention or limitations are imposed on the error rate, we need to determine them. For that, we need to determine the Stability Conditions.

Stability Conditions

Let

$$\lambda_x = \frac{\alpha\Delta t}{\Delta x^2} \text{ and } \lambda_y = \frac{\alpha\Delta t}{\Delta y^2}$$

Our equation becomes:

$$T_{i,j}^{n+1} = T_{i,j}^{n} + \lambda_x\left(T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}\right) + \lambda_y\left(T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}\right)$$

$$T_{i,j}^{n+1} = \left(1 - 2\left(\lambda_x + \lambda_y\right)\right)T_{i,j}^{n} + \lambda_x\left(T_{i+1,j}^{n} + T_{i-1,j}^{n}\right) + \lambda_y\left(T_{i,j+1}^{n} + T_{i,j-1}^{n}\right)$$

Von Neumann's error

Let $e_{i,j}^{n}$ be the difference between the exact solution (continuous differential) and approximate solution (discrete finite differences).

Using Fourier's decomposition, and according to Von Neumann, the error can be represented as follows:

$$e_{i,j}^{n} = G^{n}e^{i\left(k_x x_i + k_y y_j\right)}$$

k_x, k_y being the spatial frequency

$$x_i = i\Delta x_i \quad i = 0, 1, \ldots, N_x - 1$$

$$x_j = j\Delta x_j \quad j = 0, 1, \ldots, N_j - 1$$

G: amplification factor

The only factor that we can control is the amplification factor. If $|G| > 1$, our solution explodes and diverges.

$$|G| \leq 1 \quad \forall k_x, k_y, \left(x_i, y_j\right)$$

Let $\theta = k_x x_i + k_y y_j$,

$$e_{i,j}^{n} = G^n e^{i\theta}$$

$$e_{i,j}^{n+1} = G^{n+1} e^{i\theta}$$

$$e_{i,j}^{n+1} = G e_{i,j}^{n}$$

Because of the linearity of our problem, we obtain:

$\mathrm{T}_{\mathrm{i,j}}^{\mathrm{n+1}} = \mathrm{GT}_{\mathrm{i,j}}^{\mathrm{n}} \Rightarrow\ T_{\mathrm{i,j}}^{\mathrm{n}} = \mathrm{G}^{\mathrm{n}}\mathrm{e}^{\mathrm{i}\theta}$

Knowing that $x_{i+1} = x_i + \Delta x,\ x_{i-1} = x_i - \Delta x,\ y_{j+1} = y_j + \Delta y,\ \ y_{j+1} = y_i + \Delta y$, we can write the temporal finite differences as follows:

$$T_{i+1,j}^{n} = G^n \mathrm{e}^{i\theta} e^{ik_x \Delta x}$$

$$T_{i-1,j}^{n} = G^n \mathrm{e}^{i\theta} e^{-ik_x \Delta x}$$

$$T_{i,j+1}^{n} = G^n \mathrm{e}^{i\theta} e^{ik_\mathrm{y} \Delta \mathrm{y}}$$

$$T_{i,j-1}^{n} = G^n \mathrm{e}^{i\theta} e^{-i\mathrm{y}\Delta \mathrm{y}}$$

$$T_{i,j}^{n+1} = G^{n+1} \mathrm{e}^{i\theta}$$

Now we replace these values in

$$T_{i,j}^{n+1} = T_{i,j}^{n} + \lambda_x\left(T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}\right) + \lambda_y\left(T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}\right)$$

We obtain the following:

$$G^{n+1}\mathrm{e}^{i\theta} = G^n \mathrm{e}^{i\theta} + \lambda_x\left(G^n \mathrm{e}^{i\theta} e^{ik_x \Delta x} - 2G^n \mathrm{e}^{i\theta} + G^n \mathrm{e}^{i\theta} e^{-ik_x \Delta x}\right) + \lambda_y\left(G^n \mathrm{e}^{i\theta} e^{ik_\mathrm{y} \Delta \mathrm{y}} - 2G^n \mathrm{e}^{i\theta} + G^n \mathrm{e}^{i\theta} e^{-i\mathrm{y}\Delta \mathrm{y}}\right)$$

We divide by $G^n \mathrm{e}^{i\theta}$

$$G = 1 + \lambda_x\left(e^{ik_x \Delta x} - 2 + e^{-ik_x \Delta x}\right) + \lambda_y\left(e^{ik_\mathrm{y} \Delta \mathrm{y}} - 2 + e^{-i\mathrm{y}\Delta \mathrm{y}}\right)$$

Knowing that $e^{i\theta} + e^{-i\theta} = 2\cos(\theta)$ and $sin^2(\theta) = \frac{1-\cos(2\theta)}{2}$

$$G = 1 + \lambda_x(2\cos(k_x \Delta x) - 2) + \lambda_y\left(2\cos(k_\mathrm{y}\Delta \mathrm{y}) - 2\right)$$

$$G = 1 - 4\lambda_\mathrm{x}\sin^2\left(\frac{\mathrm{k_x}\Delta x}{2}\right) - 4\lambda_\mathrm{y}\sin^2\left(\frac{\mathrm{k_y}\Delta y}{2}\right)$$

Let $\mathrm{A} = \sin^2\left(\frac{\mathrm{k_x}\Delta x}{2}\right)$ and $\mathrm{B} = \sin^2\left(\frac{\mathrm{k_y}\Delta y}{2}\right)$

$$\mathrm{G} = 1 - 4\left(\lambda_\mathrm{x}\mathrm{A} + \lambda_\mathrm{y}\mathrm{B}\right)$$

Knowing that $0 \leq A \leq 1$ and $0 \leq B \leq 1$

$$\Rightarrow 0 \leq \lambda_x A + \lambda_y B \leq \lambda_x + \lambda_y$$

$$\Leftrightarrow 0 \leq 4(\lambda_x A + \lambda_y B) \leq 4(\lambda_x + \lambda_y)$$

$$\Leftrightarrow -4(\lambda_x + \lambda_y) \leq -4(\lambda_x A + \lambda_y B) \leq 0$$

$$\Leftrightarrow 1 - 4(\lambda_x + \lambda_y) \leq 1 - 4(\lambda_x A + \lambda_y B) \leq 1$$

$$\Rightarrow G \in \left[1 - 4(\lambda_x + \lambda_y), 1\right]$$

Since $|G| \leq 1 \Rightarrow -1 \leq G \leq 1 \Rightarrow 1 - 4(\lambda_x + \lambda_y) \geq -1 \Rightarrow 4(\lambda_x + \lambda_y) \leq 2 \Rightarrow (\lambda_x + \lambda_y) \leq \frac{1}{2}$

$$\alpha \Delta t \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right) \leq \frac{1}{2}$$

We conclude that for our solution to converge properly, we need to make sure that those conditions are met.

Interpretation of the conditions

Δt and (x, y) are inversly proportional.

- We need to make sure not to have a large Δt that can breach this limit $\nRightarrow$ we allow too much heat in a specific interval (a large interval)
- The smaller (x, y) are, the more we have make Δt in order to adher to the limit. A smaller set of coordinates means a thinner grid.

$\Rightarrow$ We need to find a balance between Δt and (x, y) to adhere to the convergence conditions. This enters the scope of the data to feed our program. Our initial goal is to tailor the program to our needs. For that, $\alpha \Delta t \left(\frac{1}{x^2} + \frac{1}{y^2}\right) \leq \frac{1}{2}$ will be one of our conditions (checks) implemented to simulate heat dissipation.

Sequential Code

Our first approach to this problem was to implement it in a sequential way: the first intuitive form of thinking. From this code, we continued to the parallel implementation.

General Conditions

Based on our mathematical analysis initial conditions, we need to make sure that our algorithm follows the limitations that we have previously stated:

- $N_x, N_y, L_x, L_y, \alpha, T_{final}$ and Δt are all positive values

- **The structure of the grid:** In each axis of points, we need to have at least 3 points in order to form a grid, since the number of blocks in each grid is found using: $Nb_{blocks} = (N_x - 1) \times (N_y - 1)$. If we had 2 points, we would have a total number of 1 block, hence not creating a grid.
- Read the grid in a discrete manner, adhering to the finite difference principle. We will accept our input in the form of consecutive values of $T^0_{i,j}$; $i, j \in \{0, 1, \dots, N_{x,y} - 1\}$
- **Stability conditions:** based on the equation: $\alpha \Delta t \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right) \leq \frac{1}{2}$

  We have the option to either check Δt or (x, y). Since we will have an input of all T-values of our function at t=0, we have an already fixed number of coordinates. Based on that we will fix the Δx and Δy and check Δt.

$$\Rightarrow \Delta t \leq \frac{1}{\alpha(\Delta x^2 + \Delta y^2)}$$

- **The Dirichlet and Neumann Conditions:** in the sequential pseudo-code, we have only implemented the Dirichlet conditions with the function *apply_dirichlet* as follows:
  - $for\ x = 0, T(0, y_j, t_n) = T^0_{0,j}$
  - $for\ x = L_x, T(L_x, y_j, t_n) = T^0_{N_{x-1},j}$
  - $for\ y = 0, T(x_i, 0, t_n) = T^0_{i,0}$
  - $for\ y = L_y, T(x_i, L_y, t_n) = T^0_{i,y}$

  These values will not change with time.

  The Neumann Conditions will be explained in later sections.

Pseudo-code

The following section shows our first pseudo-code implementation based on the Dirichlet BCs. Note that in the final version of our project, we have added the Neumann BCs (to be explained later).

Read the input_file, output_prefix from command line

Read the variables from the input file:

Nx, Ny, alpha, T_final, dt_in

// Nx and Ny represent the size of the x and y axis on the grid

// alpha is the constant coefficient of heat

// T_final is the last unit of time to be considered in our simulation (for it not to run infinitely)

//dt_in represents the advancements in time unit (delta t)

IF Nx < 3 OR Ny < 3: // first condition to verify

  stop with error

allocate T[Nx][Ny] and Tnew[Nx][Ny]

read the input file and save values

dx = Lx / (Nx - 1) // delta x

dy = Ly / (Ny - 1) // delta y

dt_max = 0.5 / (alpha * (1/dx^2 + 1/dy^2)) // dt_max represents delta t max, which is the stability condititon
// for the problem to converge

```
IF dt_in <= 0 OR dt_in > dt_max:
   dt = 0.9 * dt_max
ELSE:
   dt = dt_in
Nt = floor(T_final / dt)
IF Nt <= 0:
   STOP with error
// Save boundary values (Dirichlet boundaries fixed in time)
CREATE left[Ny], right[Ny], bottom[Nx], top[Nx]
FOR j = 0 to Ny-1:
   left[j]  = T[0][j]
   right[j] = T[Nx-1][j]
FOR i = 0 to Nx-1:
   bottom[i] = T[i][0]
   top[i]    = T[i][Ny-1]
// Time integration loop
FOR n = 1 to Nt:
   // enforce Dirichlet BC on current field
   APPLY_DIRICHLET(T, left, right, bottom, top) // apply the Dirichlet conditions
   // update interior points using explicit finite differences
   FOR i = 1 to Nx-2:
      FOR j = 1 to Ny-2:
         laplacian =
            (T[i+1][j] - 2*T[i][j] + T[i-1][j]) / dx^2
          + (T[i][j+1] - 2*T[i][j] + T[i][j-1]) / dy^2 // apply the finite difference method that we have obtained
         Tnew[i][j] = T[i][j] + alpha * dt * laplacian
   // enforce Dirichlet BC on the new field as well
   APPLY_DIRICHLET(Tnew, left, right, bottom, top)
   // prepare for next step
   SWAP(T, Tnew)
END FOR
// Write final field to CSV
```

```
FUNCTION APPLY_DIRICHLET(T, left, right, bottom, top) // Function to impose Dirichlet Boundaries


Nx = number of rows in T

Ny = number of columns in T


FOR j = 0 to Ny-1:

   T[0][j] = left[j]

   T[Nx-1][j] = right[j]


FOR i = 0 to Nx-1:

   T[i][0] = bottom[i]

   T[i][Ny-1] = top[i]


END FUNCTION
```

With this algorithm in mind, the sequential program was implemented. The code details are in the GitHub repository along with the details of execution.

Interpretation

After the sequential analysis we realize that we have issues concerning:

- **Time complexity:** the explicit scheme updates every interior cell at every time step. So the runtime scales like O($N_t . N_x . N_y$), which becomes very large as soon as we increase resolution, making it prone to explode with grid size.
- **Memory and cache pressure:** two full 2D grids (T and T_new) of size $N_x \times N_y$ are stored. For large meshes, this increases RAM usage and reduces cache efficiency, slowing down the loop.

For these reasons, implementing an alternative approach that reduces time complexity and memory pressure is ideal. A method that does this is parallelization.

Parallelization

In this section, we are going to identify the parallelizable code in our previous sequential method, define conditions to properly implement the new solution, write the pseudo-code, explain some features in the parallel code and compare it with the previous one: both theoretically and practically.

Parallelizable sections

In our sequential code we have the following loop:

```
for (int n = 0; n < Nt; ++n) {
   for (int i = 1; i < Nx - 1; ++i) {
      for (int j = 1; j < Ny - 1; ++j) {
         // Laplacian + update
      }
   }
}
```

We notice the following when analyzing our code:

- Each cell update is local: *T_new[i][j]* uses only 4 neighbors + itself (*T[i+1][j], T[i][j+1], T[i][jj]*). That makes it perfect for domain decomposition (split the grid into blocks).
- Independence inside a timestep: for a fixed time step n, each cell update only reads from T and writes to *T_new*. So different subdomains can compute in parallel, as long as they have the neighbor border values.
- The only dependency across processes is at subdomain boundaries. That's why MPI halo (ghost) cells are used: to exchange boundary rows/columns.

Halo Cells

The need for halo cells is crucial in this context. To prove it, we should remember the previous finite differences found in the previous section:

$$\frac{\partial^2 T}{\partial x^2}(x_i, y_j, t_n) \approx \frac{T^n_{i+1,j} + T^n_{i-1,j} - 2T^n_{i,j}}{\Delta x^2}$$

$$\frac{\partial^2 T}{\partial y^2}(x_i, y_j, t_n) \approx \frac{T^n_{i,j+1} + T^n_{i,j-1} - 2T^n_{i,j}}{\Delta y^2}$$

Based on these equations, we notice that we have 5 different instances of T used in each time step: $T^n_{i,j}, T^n_{i-1,j}, T^n_{i+1,j}, T^n_{i,j-1}, T^n_{i,j+1}$.

The calculations will be done smoothly in the center of the gird (space) that we have, but for the boundaries, this will be another issue.

For example, to calculate $T^{n,1}_{i-1,j}\ for\ i = 0$ in process p ($T^{n,1}_{i-1,j}$ is an internal block inside the global grid), we will be out of boundary in this section.

Even though we have the Dirichlet and Neumann conditions, these only apply on the global boundaries of the system knowing that the boundaries in each communicator of MPI are internal components in the global grid. In the context of MPI, the messages communicated between communicators will not include the full necessary information about the neighboring cells, therefore a process might get interrupted only to fetch this instance. For this reason, metrics and conditions should be stated to unify these cells.

After domain decomposition, a stencil at a local node (i, j) needs to access the data of the blocks that are immediately next to it from the top, bottom, left and right, which may lay on an adjacent process. In each process we will have at least one case of halo cells. For that, halo cells exchange information within an interface between two subdomains that represent the processes.

Let A, B be processes that have halo cells in between. We impose:

$$T^{(A)}_{ghost} = T^{(B)}_{boundary}\ , \qquad T^{(B)}_{ghost} = T^{(A)}_{boundary}$$

With that, each process applies stencil locally.

General Conditions

All the previous conditions mentioned in the sequential code are applied here, along with the following ones:

- **Dirichlet and Neumann Conditions:** in this code, we have given the user the option to either use the Dirichlet or Neumann conditions. As previously explained in the sequential section, the same conditions apply in the parallel code.
  For Neumann option, we want to achieve zero heat flux through physical walls, adhering to this condition:
  $$\frac{\partial T}{\partial n} = 0\ \ on\ \partial\Omega$$
  Let's take the example of the left wall ($x = 0$). The normal derivative following the x-direction is: $\frac{\partial T}{\partial x}\Big|_{x=0} = 0$. Using the previous forward finite difference, we have the following formula:

$$\left.\frac{\partial T}{\partial x}\right|_{x=0} \approx \frac{T_{1,j}^n - T_{0,j}^n}{\Delta x}$$
$$\Rightarrow \frac{T_{1,j}^n - T_{0,j}^n}{\Delta x} = 0$$
$$\Rightarrow T_{1,j}^n - T_{0,j}^n = 0$$
$$\Rightarrow T_{1,j}^n = T_{0,j}^n$$

What this means is that the temperature on the left wall equals the temperature inside the domain directly.

The same concept is applied to all the other boundaries (right, top, bottom).

We will have:

- Left ($x = 0$): $T_{1,j}^n = T_{0,j}^n$
- Right ($x = N_x$): $T_{N_x-1,j}^n = T_{N_x-2,j}^n$
- Bottom ($y = 0$): $T_{x,1}^n = T_{x,0}^n$
- Top ($y = N_y$): $T_{i,N_y-1}^n = T_{i,N_y-2}^n$

- **Halo conditions between subdomains:** $T_{ghost}^{(A)} = T_{boundary}^{(B)}$, $T_{ghost}^{(B)} = T_{boundary}^{(A)}$

Pseudo-code

```
Algorithm ParallelHeat2D_MPIOpenMP(input_file, output_prefix)


Initialize MPI
 MPI_Init()
 Get world communicator: comm = MPI_COMM_WORLD
 Get rank and size: rank, size
 Save input and output paths from arguments
 // Create 2D Cartesian process grid
 dims[2] = {0, 0}
   MPI_Dims_create(size, 2, dims)      // chooses dims[0]×dims[1]
 periods[2] = {0, 0}                 // non-periodic
 MPI_Cart_create(comm, 2, dims, periods, reorder=1, cart_comm)
 Get cart_rank and coordinates:
   MPI_Comm_rank(cart_comm, cart_rank)
   MPI_Cart_coords(cart_comm, cart_rank, 2, coords)
```

Find 4 neighbors:

MPI_Cart_shift(cart_comm, dir=0, disp=1, nbr_left, nbr_right)

MPI_Cart_shift(cart_comm, dir=1, disp=1, nbr_down, nbr_up)

Read global problem data on rank 0

If cart_rank == 0:

open input_file

read Nx, Ny, Lx, Ly, alpha, T_final, dt_in, bc_str // like sequential, with additional bc_str that specifies the //boundary method used

validate alpha > 0, Lx > 0, Ly > 0, T_final > 0 // conditions

convert bc_str to uppercase

if bc_str == "NEUMANN": bc_type = 1 else bc_type = 0

allocate global_T[Nx * Ny]

for j = 0..Ny-1:

for i = 0..Nx-1:

read global_T[j*Nx + i] // initial temperature field

close file

Broadcast global parameters

MPI_Bcast(Nx, Ny, Lx, Ly, alpha, T_final, dt_in, bc_type) from rank 0

Check minimum grid size:

if Nx < 3 or Ny < 3: MPI_Abort // conditions

Local domain decomposition (block decomposition with remainder)

For each process: //define the internal grids and calculate their size for each process

local_nx = local_size(Nx, coords[0], dims[0])

local_ny = local_size(Ny, coords[1], dims[1])

global_i_start = global_start(Nx, coords[0], dims[0])

global_j_start = global_start(Ny, coords[1], dims[1])

where:

local_size(N, coord, dim):

base = N / dim

rem = N % dim

return base + 1 if coord < rem, else base

global_start(N, coord, dim):

base = N / dim, rem = N % dim

if coord < rem:

return coord * (base + 1)

else:

return rem*(base+1) + (coord - rem)*base

Compute space steps and stable time step

dx = Lx / (Nx - 1)

dy = Ly / (Ny - 1)

denom = $1/dx^2 + 1/dy^2$

dt_max = 0.5 / (alpha * denom) // explicit stability condition

if dt_in <= 0 or dt_in > dt_max:

dt = 0.9 * dt_max // choose safe dt

else:

dt = dt_in

Nt = floor(T_final / dt)

if Nt <= 0: MPI_Abort // MPI_Abort immediately terminates processes in case of error

// it is used instead of MPI_Finalize since this method only tells the specific process //running it to stop, the rest will continue normally. In the case of abort, we want the //whole program to stop and savely exit all nodes.

Allocate local arrays with halo layers

Ny_with_halo = local_ny + 2

Nx_with_halo = local_nx + 2

Allocate:

T [Nx_with_halo * Ny_with_halo] // current time step

Tnew [Nx_with_halo * Ny_with_halo] // next time step

local_block[local_nx * local_ny] // interior without halos

```
Scatter initial condition to all processes
  If cart_rank == 0:
      For p = 0..size-1:
        get coords c[2] of process p
        ln_x = local_size(Nx, c[0], dims[0])
        ln_y = local_size(Ny, c[1], dims[1])
        gi0  = global_start(Nx, c[0], dims[0])
        gj0  = global_start(Ny, c[1], dims[1])
        allocate sendbuf[ln_x * ln_y]
        for lj = 0..ln_y-1:
          for li = 0..ln_x-1:
            gi = gi0 + li
            gj = gj0 + lj
            sendbuf[lj*ln_x + li] = global_T[gj*Nx + gi]
        if p == 0:
            local_block = sendbuf
        else:
            MPI_Send(sendbuf, dest=p, tag=100)
    else:
      MPI_Recv(local_block, source=0, tag=100)

  Copy local_block into interior of T (offset by 1 for halos):
      for lj = 0..local_ny-1:
        for li = 0..local_nx-1:
          I = li + 1
          J = lj + 1
          T[I, J] = local_block[lj*local_nx + li]
```

```
Allocate halo communication buffers
   send_left, recv_left   [local_ny] // to send information between the processes about halo cells
      send_right, recv_right [local_ny]
      send_down, recv_down   [local_nx]
      send_up,   recv_up     [local_nx]
Time-stepping loop
   For n = 0..Nt-1:
    Start non-blocking halo exchanges
       // Left–right neighbors
       if nbr_left ≠ MPI_PROC_NULL:
          for j = 1..local_ny:
             send_left[j-1] = T[1, j]            // first interior column
          MPI_Irecv(recv_left,  source=nbr_left,  tag=10, reqs++) // send information between the processes
          MPI_Isend(send_left,  dest=nbr_left,    tag=11, reqs++)
       if nbr_right ≠ MPI_PROC_NULL:
          for j = 1..local_ny:
             send_right[j-1] = T[local_nx, j]     // last interior column
          MPI_Irecv(recv_right, source=nbr_right, tag=11, reqs++)
          MPI_Isend(send_right, dest=nbr_right,   tag=10, reqs++)
       // Down–up neighbors
       if nbr_down ≠ MPI_PROC_NULL:
          for i = 1..local_nx:
             send_down[i-1] = T[i, 1]            // first interior row
          MPI_Irecv(recv_down, source=nbr_down,  tag=20, reqs++)
          MPI_Isend(send_down, dest=nbr_down,    tag=21, reqs++)
       if nbr_up ≠ MPI_PROC_NULL:
          for i = 1..local_nx:
             send_up[i-1] = T[i, local_ny]       // last interior row
          MPI_Irecv(recv_up,   source=nbr_up,    tag=21, reqs++)
          MPI_Isend(send_up,   dest=nbr_up,      tag=20, reqs++)
```

```
Wait for all halo communications to complete
        MPI_Waitall(req_count, reqs) // synchronize
    Fill halo cells with received values
        if nbr_left ≠ MPI_PROC_NULL:
            for j = 1..local_ny:
                T[0, j] = recv_left[j-1]          // left ghost column
        if nbr_right ≠ MPI_PROC_NULL:
            for j = 1..local_ny:
                T[local_nx+1, j] = recv_right[j-1]   // right ghost column
        if nbr_down ≠ MPI_PROC_NULL:
            for i = 1..local_nx:
                T[i, 0] = recv_down[i-1]          // bottom ghost row
        if nbr_up ≠ MPI_PROC_NULL:
            for i = 1..local_nx:
                T[i, local_ny+1] = recv_up[i-1]     // top ghost row
    Apply Neumann BC at physical walls (if selected)
        if bc_type == 1:
            apply_neumann_ghosts(T, local_nx, local_ny, Ny_with_halo, coords, dims)
        // This enforces ∂T/∂n = 0 on global boundaries via:
        // left wall:  T[0, j]= T[1, j]
        // right wall: T[local_nx+1, j] = T[local_nx, j]
        // bottom: T[i, 0]= T[i, 1]
        // top: T[i, local_ny+1] = T[i, local_ny]
    Update all local cells using explicit scheme
        for i = 1..local_nx:
          gi = global_i_start + (i-1)       // global x-index
          for j = 1..local_ny:
            gj = global_j_start + (j-1)     // global y-index
            Tij = T[i, j]
```

```
// Dirichlet BC: keep fixed values on global boundary nodes
            if bc_type == 0 and (gi == 0 or gi == Nx-1 or gj == 0 or gj == Ny-1):
               Tnew[i, j] = Tij
               continue
            Tip1j = T[i+1, j]
            Tim1j = T[i-1, j]
            Tijp1 = T[i, j+1]
            Tijm1 = T[i, j-1]
            lap = (Tip1j - 2*Tij + Tim1j)/dx^2 + (Tijp1 - 2*Tij + Tijm1)/dy^2
            Tnew[i, j] = Tij + alpha * dt * lap
     Swap time levels
         swap(T, Tnew)
Gather final result on rank 0
   Extract interior back to local_block
       for lj = 0..local_ny-1:
        for li = 0..local_nx-1:
         I = li + 1
         J = lj + 1
         local_block[lj*local_nx + li] = T[I, J]
   If cart_rank == 0:
        allocate global_final[Nx * Ny]
        // Copy own block
        for lj = 0..local_ny-1:
         for li = 0..local_nx-1:
           gi = global_i_start + li
           gj = global_j_start + lj
           global_final[gj*Nx + gi] = local_block[lj*local_nx + li]
```

```
// Receive and insert blocks from other ranks
        for p = 1..size-1:
          get coords c[2] of p
          ln_x = local_size(Nx, c[0], dims[0])
          ln_y = local_size(Ny, c[1], dims[1])
          gi0  = global_start(Nx, c[0], dims[0])
          gj0  = global_start(Ny, c[1], dims[1])
          allocate recv_block[ln_x * ln_y]
          MPI_Recv(recv_block, source=p, tag=200)
          for lj = 0..ln_y-1:
            for li = 0..ln_x-1:
              gi = gi0 + li
              gj = gj0 + lj
              global_final[gj*Nx + gi] = recv_block[lj*ln_x + li]
        // Write CSV file
        open (output_pref + "_final.csv")
        for j = 0..Ny-1:
          for i = 0..Nx-1:
            write global_final[j*Nx + i] separated by ','
          newline
        close file
    Else (cart_rank ≠ 0):
        MPI_Send(local_block, dest=0, tag=200)
Finalize
    MPI_Comm_free(cart_comm)
    MPI_Finalize() // exit MPI correctly
```

Interpretation

With the new parallel code, we have improved the following issues:

- **Time Complexity:** With MPI and a 2D domain decomposition, the global work is still proportional to $O(N_t \times N_x \times N_y)$, but this work is split across P processes.

  Each process only updates a local subdomain of size $\approx O\left(\frac{N_x.N_y}{P}\right)$, so the per-process computational cost becomes:

  $$O\left(\frac{N_t N_x N_y}{P}\right) + communication\ cost$$

  As long as the communication overhead stays moderate, the wall-clock runtime decreases roughly like $\frac{1}{P}$. This allows us to handle much larger grids and longer simulations than in the purely sequential case.
- **Memory and cache usage:** In the parallel code, each process only stores its local sub-grid and its halo cells: two arrays of size $\approx \left(local_{nx} \times\ local_{ny}\right) \approx \frac{N_x \cdot N_y}{P}$ instead of the full $N_x \times N_y$ domain.

  This reduces the memory footprint per process by a factor $\approx \frac{1}{P}$, and the smaller local arrays fit better in cache. As a result, memory pressure and cache misses are reduced, improving the effective performance of the stencil updates compared to the sequential implementation.

To back up our statements, we are going to test both sequential and parallel codes with different datasets.

Testing

We have tested both sequential and parallel codes with different data. The tests have been run on the same Windows Machine (12th Gen Intel(R) Core(TM) i9-12900H (2.50 GHz) – x64-based processor). We have used the command ***Measure-Command***. It generates the time needed to finish executing the full program run.

We have classified our tests based on three variables:

- Executed code (parallel / sequential)
- Number of processes (parallel)
- Grid Dimensions (parallel / sequential)

For measurements, we will use the Speedup Ratio (S) and Efficiency Measurement (E) given by the following formulas:

$$S(P) = \frac{T_{seq}}{T_p} \quad and \quad E(P) = \frac{S(P)}{P}$$

Small Grid (200 × 200) – Sequential / Parallel

Code	Grid	MPI procs	Time (s)	Speedup % Sequential
Sequential	200 × 200	1	10.52	1x (baseline)
Parallel	200 × 200	1	6.42	1.64x
Parallel	200 × 200	4	2.05	5.13x

In small grids, the gain in speed is modest.

Medium and Large Grids – Sequential / Parallel

Case	Grid	Code / MPI procs	Time (s)	Speedup % Sequential
Weak P1	256 × 256	Sequential – 1 Parallel – 4	34.93 5.38	- 6.49x
Weak P4	512 × 512	Sequential – 1 Parallel – 4	558.98 94.22	- 5.93x
Weak P9	768 × 768	Sequential – 1 Parallel – 4	2749.28 600.28	- 4.58x

- As the grid grows, both versions get slower, but MPI with 4 processes always gives a 4 to 6 speedup over the sequential code.
- Speedup decreases slightly when the problem becomes big (768^2), which is normal because communication and halo exchanges become more significant.

Strong scaling on 768 × 768 – Parallel

Grid	MPI procs	Time (s)	Speedup % MPI-1	Parallel efficiency vs MPI-1	Speedup % Sequential
768 × 768	1	1413.18	1x	1	1.95
768 × 768	4	600.28	2.35x	0.59	4.58
768 × 768	6	304.19	4.64x	0.77	9.04
768 × 768	8	363.83	3.88x	0.49	7.56

On the 768 × 768 case, the MPI code shows good strong scaling up to 6 processes, but when increasing to 8 processes the runtime increases again (304s → 364s), illustrating that using more processes is not always beneficial and may degrade performance due to communication overhead and load imbalance.

Heatmaps

For each test, a heatmap was generated. They can be found in the plots directory. Note that both sequential and parallel codes generated almost the same functions, so we will show the parallel heatmaps here only. The other plots are available in the GitHub repository.

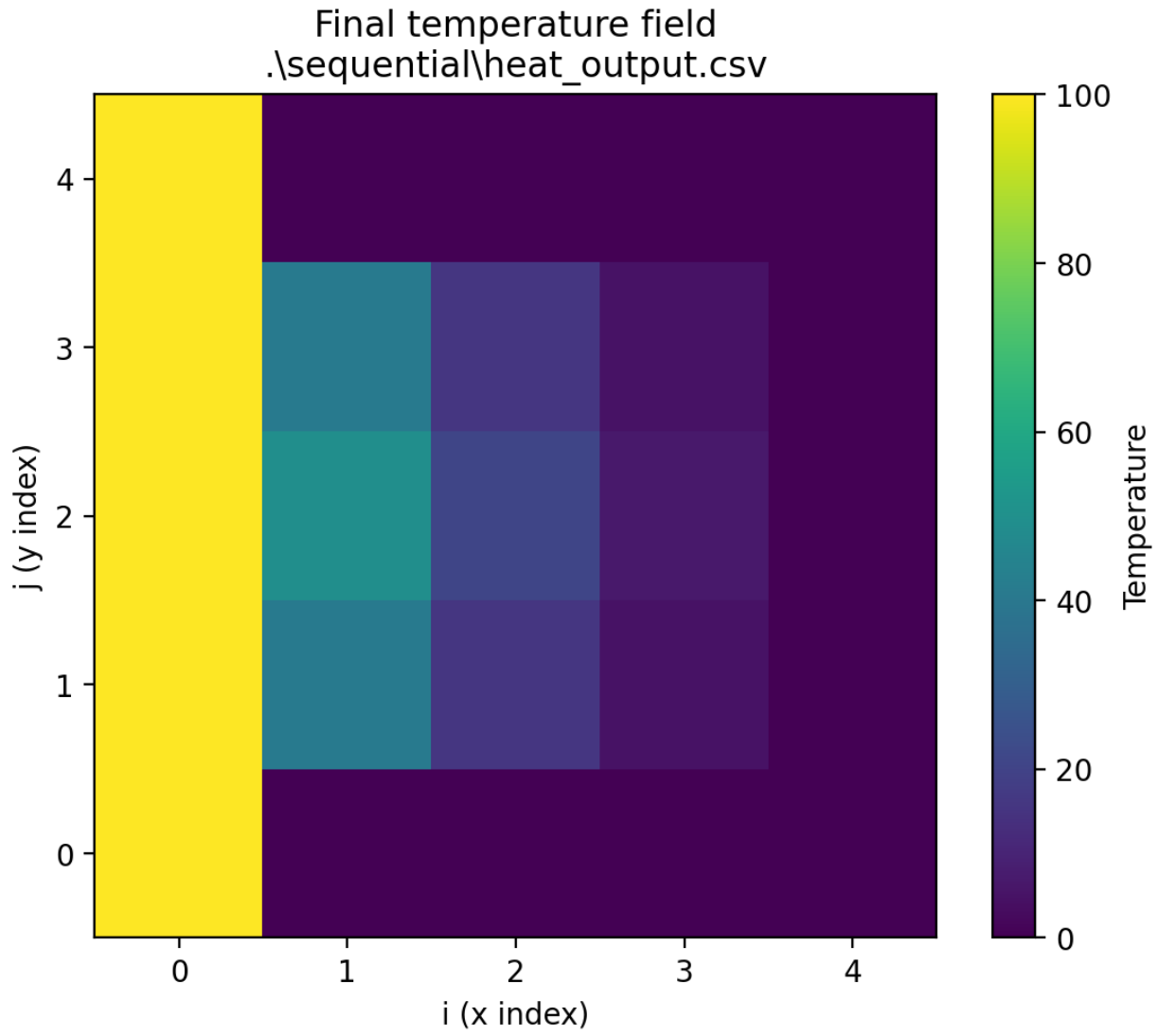


Figure 1: Heat Dissipation Simulation with 5 x 5 dimensions

Visualizing Dirichlet and Neumann Boundary Conditions

The following section is a comparison between the Dirichlet and Neumann BCs based on the plots generated on a 768 × 768 grid.

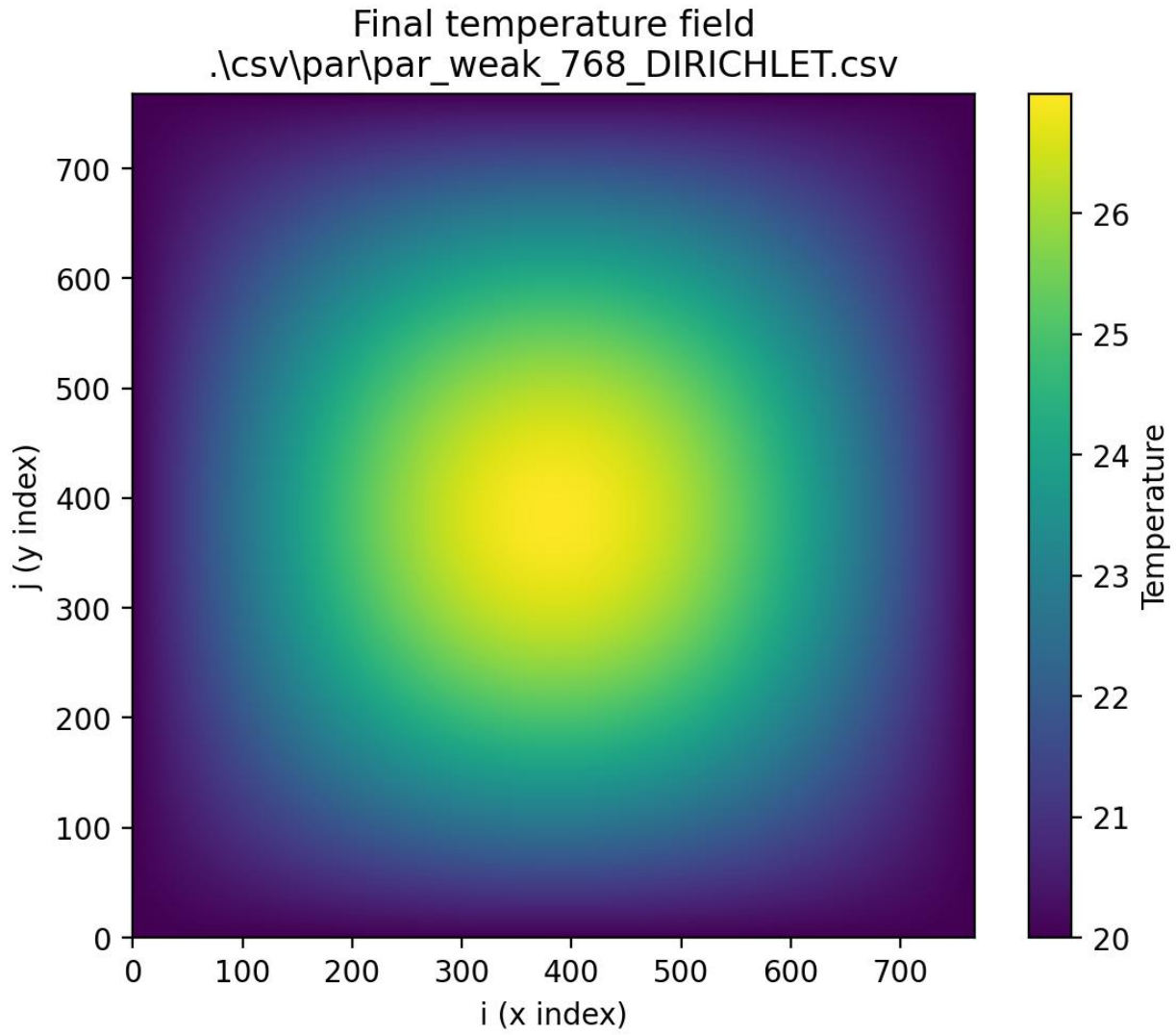


Figure 2: Heat Dissipation Simulation with Dirichlet BCs on 768x768

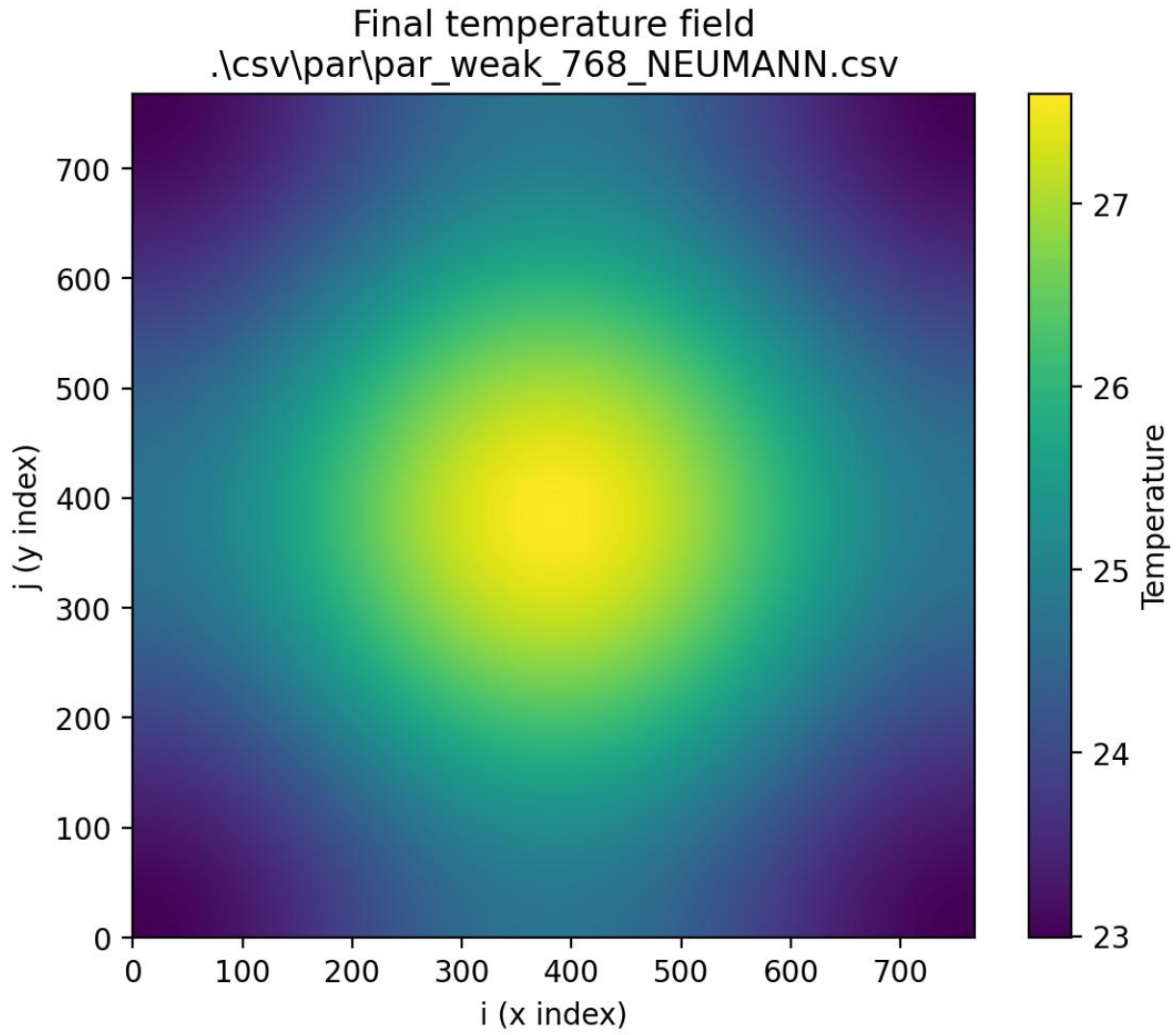


Figure 3: Heat Dissipation Simulation with Neumann BCs on 768x768

We can notice with Dirichlet that the outer ring is darker than in Neumann's, indicating a more stable temperature, which reflects back on the definitions of the Dirichlet and Neumann's BCs, with Dirichlet defining a constant temperature on the edges, whereas Neumann stating that temperatures are free to evolve with time, as long as no heat gets dissipated outside the walls.

The temperature in the middle of Neumann's model is slightly higher than the one in Dirichlet, which is justifiable since Neumann's goal is to keep heat from spreading outside the walls (Neumann stores more heat).

Scraped Design Alternatives

Initially, we considered designing the solver using an object-oriented and more functional style (classes for grids and boundary conditions, higher-level update functions shared by the sequential and parallel versions). However, this would have introduced additional abstraction layers, more boilerplate code, and indirections (constructors, virtual methods, callbacks) that complicate debugging, especially in an MPI context. The focus of the project is on numerical correctness and parallel performance rather than on software architecture, so this extra complexity was not justified. We therefore opted for a simpler, primarily procedural design that keeps the data structures and control flow explicit, making it easier to validate the numerical scheme, reason about communication patterns, and compare the sequential and parallel implementations.

Conclusion

To conclude, we implemented and validated a 2D heat equation solver in both sequential and MPI-based parallel versions, supporting Dirichlet and Neumann boundary conditions. The sequential code served as a reference for correctness and highlighted the $O(N_t N_x N_y)$ cost and memory limits on large grids. The parallel version, based on domain decomposition and halo exchanges, clearly improved runtime for medium and large meshes, especially under weak scaling, while also revealing the communication overhead and loss of efficiency on small problem sizes. Overall, the study shows that parallelization is essential to handle finer discretizations and longer simulations, but it must be balanced against MPI overhead and hardware constraints to be truly beneficial.