Heat equation

High performance scientific computing in Aerospace Module 1

Francesco Micucci
Personal Code 10729167
francesco.micucci@mail.polimi.it

POLITECNICO
MILANO 1863

Academic Year 2023-2024

Contents

Chapter 1

Introduction

In the following chapters, we delve into a detailed examination of the heat equation to tackle a specific aspect of the following problem.

"A satellite must visit a meteorite orbiting around the Sun. To pursue this intent, we would like to acquire an image of this object in the infrared range and compare it with a synthetic image generated by us. Through a three-dimensional model and the knowledge of the physical properties of the asteroid, we should be able to obtain a synthetic image, and through the comparison with the acquired image, we should be able to identify the landing point of the satellite on the asteroid."

In this frame, the heat equation is fundamental to reconstruct the thermal field inside the asteroid.
To achieve this goal, we will adhere to the following process:

1. Mathematical formulation of the problem
2. Approximate study of the orders of magnitude and possible simplifications
3. Discretization of the problem
4. Program implementation
5. Verification of the program

Chapter 2

Mathematical formulation of the problem

To establish the mathematical framework for our problem, we begin by describing the internal thermal dynamics of the asteroid. This process is guided by two fundamental equations of thermodynamics.
The first one corresponds to the first law of thermodynamics:

$$dE = \delta q - \delta L \tag{2.1}$$

It states that the change in the internal energy of a system is equal to the amount of heat supplied to or withdrawn from the system during the process minus the work done on or by the system during that process.
Instead, the second equation is the enthalpy equation:

$$h = E + pv \quad \rightarrow \quad dh = dE + pdv + vdp \tag{2.2}$$

Here, h represents the enthalpy, E is the internal energy, p is the pressure, and v is the volume.
Now, knowing that $\delta L = pdv$, we can rewrite (2.1) as:

$$dE = \delta q - pdv \tag{2.3}$$

and, combining the equations (2.2) and (2.3), we obtain:

$$dh = \delta q + vdp \tag{2.4}$$

At this point, we consider the pressure to be constant, so the equation simplifies to:

$$dh = \delta q \tag{2.5}$$

As of now, a variation in enthalpy on a sample volume may be due to a possible heat flow or the presence of an internal source. So, we have that:

$$\frac{\mathrm{d}h}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\int_V \rho c_p T \, dV = \int_S -\overrightarrow{q} \cdot \widehat{n} \, dS + \int_V f \, dV \tag{2.6}$$

where $\overrightarrow{q}$ is the incoming heat, $\widehat{n}$ is the normal vector to the surface, and where the last integral is due to the presence of a possible heat source.
Using the Fourier law ($\overrightarrow{q} = -k\nabla T$), we can write that:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_V \rho c_\mathrm{p} T \, dV = \int_S k\nabla T \cdot \widehat{n} \, dS + \int_V f \, dV \tag{2.7}$$

where k is the thermal conductivity.
Now, under the hypothesis of being able to exchange the integral operator and the derivative operator, and using the divergence theorem on the Equation (2.7), we get:

$$\int_V (\frac{\partial}{\partial t}(\rho c_\mathrm{p} T) - \nabla \cdot (k \nabla T) - f)\, dV = 0 \tag{2.8}$$

If the integral is equal to 0, it does not mean that the function being integrated is also equal to 0. This is only true in the case in which we work with an arbitrary volume. Since the volume of the sample is an arbitrary volume we can conclude that:

$$\frac{\partial}{\partial t}(\rho c_\mathrm{p} T) - \nabla \cdot (k \nabla T) - f = 0 \tag{2.9}$$

Dividing both sides in (2.9) by ρc_p and replacing $\frac{f}{\rho c_p}$ with $\tilde{f}$ and $\frac{k}{\rho c_p}$ with K, we obtain:

$$\frac{\partial T}{\partial t} - K \nabla^2 T = \tilde{f} \tag{2.10}$$

The equation (2.10) is expressed using dimensional variables but we would like to obtain a dimensionless equation because they are much more suitable to grasp relevant aspects of the problem. Using the chain rule with the following substitutions (where all quantities are defined by a typical quantity multiplied by the dimensionless field):

$$T = T^* \bar{T}, \quad t = t^* \bar{t}, \quad x = L\bar{x}, \quad y = L\bar{y}, \quad z = L\bar{z}$$

we obtain:

$$T^* \frac{\partial \bar{T}}{\partial \bar{t}} \frac{\partial \bar{t}}{\partial t} - T^* \frac{K}{L^2} \bar{\nabla}^2 \bar{T} = \tilde{f} \tag{2.11}$$

Since $\frac{\mathrm{d}\bar{t}}{\mathrm{d}t} = 1/t^*$, (2.11) becomes:

$$\frac{\partial \bar{T}}{\partial \bar{t}} - \frac{K t^*}{L^2} \bar{\nabla}^2 \bar{T} = \frac{\tilde{f} t^*}{T^*} \tag{2.12}$$

Chapter 3

Approximate study of the orders of magnitude and possible simplifications

In this chapter, our objective is to analyze the heat equation within a simple context. By doing so, we aim to gain a clearer understanding of the fundamental scales associated with this problem and possible simplifications to be adopted.

3.1 Problem statement

We're focusing on an infinite one-dimensional bar governed by the dimensionless heat equation:

$$\frac{\partial \bar{T}}{\partial \bar{t}} - \frac{\partial^2 \bar{T}}{\partial \bar{x}^2} = 0 \tag{3.1}$$

Here, $\bar{T}$, $\bar{t}$, and $\bar{x}$ represent the dimensionless temperature, time, and space respectively.

We observe that the meteorite's rotation occurs much more rapidly than its revolution, allowing us to treat the two phenomena separately. So, we expect the surface temperature to be a periodic function with period t_r:

$$T_c(t) = T_{max} \cos\left(\frac{2\pi t}{t_r}\right) \tag{3.2}$$

where $T_{\max}$ represents the maximum temperature.
This leads us to define the boundary conditions as:

$$T(0,t) = T_c(t) \tag{3.3}$$

Adimensionalizing, we find:

$$\bar{T}_c(\bar{t}) = \cos(2\pi\bar{t}) = \frac{1}{2}(e^{\mathrm{i}2\pi\bar{t}} + e^{-\mathrm{i}2\pi\bar{t}}) \tag{3.4}$$

with $T^* = T_{\max}$ and $t^* = t_r$.

3.2 Problem solution

Our strategy to tackle the problem involves employing the *Separation of Variables*. Hence, we look for a solution of the form:

$$\bar{T}(\bar{x},\bar{t}) = F(\bar{t})G(\bar{x}) \tag{3.5}$$

Substituting (3.5) into (3.1), we obtain:

$$\frac{F'(\bar{t})}{F(\bar{t})} = \frac{G''(\bar{x})}{G(\bar{x})} \tag{3.6}$$

Since the left side depends solely on $\bar{t}$ and the right side solely on $\bar{x}$, both sides must equal a constant, denoted as c. Thus, we have two equations:

$$\frac{F'(\bar{t})}{F(\bar{t})} = c \tag{3.7}$$

$$\frac{G''(\bar{x})}{G(\bar{x})} = c \tag{3.8}$$

From the equation (3.7), we deduce the solution form:

$$F(\bar{t}) = Ae^{c\bar{t}} \tag{3.9}$$

where A is a constant.
Next, we explore solutions for $F(\bar{t})$ that satisfy the boundary conditions (3.4) when $G(\bar{x}) = 1$. These solutions are:

$$F_1(\bar{t}) = A_1 e^{c_1 \bar{t}} \quad \text{where} \quad c_1 = \mathrm{i}2\pi \tag{3.10}$$

$$F_2(\bar{t}) = A_2 e^{c_2 \bar{t}} \quad \text{where} \quad c_2 = -\mathrm{i}2\pi \tag{3.11}$$

When $F(\bar{t}) = F_1(\bar{t})$, the equation (3.8) becomes:

$$G_1''(\bar{x}) - 2\mathrm{i}\pi G_1(\bar{x}) = 0 \tag{3.12}$$

and the corresponding solution will be:

$$G_1(\bar{x}) = D_1 e^{\sqrt{\pi}(1+\mathrm{i})\bar{x}} + E_1 e^{-\sqrt{\pi}(1+\mathrm{i})\bar{x}} \tag{3.13}$$

where $D_1 = 0$ for the solution to have physical significance (i.e., decay as $\bar{x}$ approaches infinity).
While, when $F(\bar{t}) = F_2(\bar{t})$, the equation (3.8) becomes:

$$G_2''(\bar{x}) + 2\mathrm{i}\pi G_2(\bar{x}) = 0 \tag{3.14}$$

and the corresponding solution will be:

$$G_2(\bar{x}) = D_2 e^{-\sqrt{\pi}(1-\mathrm{i})\bar{x}} + E_2 e^{\sqrt{\pi}(1-\mathrm{i})\bar{x}} \tag{3.15}$$

where $E_2 = 0$ for the solution to have physical significance (i.e., decay as $\bar{x}$ approaches infinity).

Combining these results, we formulate the general solution as:

$$\bar{T}(\bar{x}, \bar{t}) = H_1 e^{\mathrm{i}2\pi\bar{t}} e^{-\sqrt{\pi}(1+\mathrm{i})\bar{x}} + H_2 e^{-\mathrm{i}2\pi\bar{t}} e^{-\sqrt{\pi}(1-\mathrm{i})\bar{x}} \tag{3.16}$$

where $H_1 = A_1 E_1$ and $H_2 = A_2 D_2$.
Finally, we impose the boundary conditions (3.4) to determine the constants H_1 and H_2:

$$\begin{cases} \bar{T}(0, \bar{t}) = H_1 e^{\mathrm{i}2\pi\bar{t}} + H_2 e^{-\mathrm{i}2\pi\bar{t}} \\ \bar{T}(0, \bar{t}) = \bar{T}_c(\bar{t}) = \frac{1}{2} e^{\mathrm{i}2\pi\bar{t}} + \frac{1}{2} e^{-\mathrm{i}2\pi\bar{t}} \end{cases} \tag{3.17}$$

obtaining that $H_1 = H_2 = \frac{1}{2}$.
Thus, the solution to the problem is:

$$\bar{T}(\bar{x}, \bar{t}) = \frac{1}{2} e^{\mathrm{i}2\pi\bar{t}} e^{-\sqrt{\pi}(1+\mathrm{i})\bar{x}} + \frac{1}{2} e^{-\mathrm{i}2\pi\bar{t}} e^{-\sqrt{\pi}(1-\mathrm{i})\bar{x}} \tag{3.18}$$

3.3 Asteroid 162173 RYUGU

We now focus our attention on asteroid *162173 RYUGU*, renowned in the scientific community due to its exploration by the Japanese space probe *Hayabusa 2*.
It is characterized by a mean radius of 450 meters, a density ρ of $3000\,\mathrm{kg/m^3}$, a specific heat capacity c_p of $500\,\mathrm{J/(kg\ K)}$, and a thermal conductivity k of $1\,\mathrm{W/(m\ K)}$.
Utilizing this data, we aim to estimate the order of magnitude for various parameters relevant to the problem.
We begin estimating K:

$$K = \frac{k}{\rho c_p} = \frac{1}{3000 \cdot 500} = 6.67 \cdot 10^{-7}\ \frac{m^2}{s}$$

Then, we estimate L from equation (3.1), as $\frac{Kt^*}{L^2} = 1$:

$$L = \sqrt{Kt^*} = \sqrt{6.67 \cdot 10^{-7} \cdot 27720} = 0.14\ \mathrm{m}$$

where $t^* = 7,7\ \mathrm{h} = 27720\ \mathrm{s}$.
This finding indicates that implementing the spatial discretization is likely to be challenging.

Chapter 4

Problem discretization

Before diving into the code, it's crucial to introduce the discretization techniques we intend to use. In this way, we will be able to convert governing equations from their continuous form into a discrete representation suitable for numerical computation.
To pursue this intent, we'll use uniform grids to discretize both the spatial and temporal domains.

4.1 Discretization techniques

To identify the most suitable discretization technique for our case study, we will examine the strengths and weaknesses of several commonly employed methods.
For this purpose, we will consider the following discretization techniques:

- **Finite Difference Methods (FDM)**: they are both easy to implement and fast but are best suited for simple geometries. Moreover, they may not provide the required accuracy.
- **Finite Element Methods (FEM)**: they are general and applicable to complex geometries. Furthermore, they have well-established mathematics but are slow and hard to implement.
- **Finite Volume Methods (FVM)**: they are well-suited for handling complex geometries but require advanced mathematics and can be challenging to implement.
- **Spectral Methods**: they are very accurate but require a simple domain.
- **Spectral Element Methods**: they have the same pros and cons as the Finite Element Methods but are more accurate.

Tables 4.1 and 4.2 summarize the pros and cons of these techniques.

4.2 Spatial discretization

When considering spatial discretization, we've opted for Finite Difference Methods. These methods have been the preferred choice due to their ease of implementation and high speed. Consequently, during the code development phase, we'll focus on relatively simple geometries for the asteroid under investigation.

Method	Pros
Finite Difference Methods	Ease of implementation, speed
Finite Element Methods	Generality, well-established mathematics
Finite Volume Methods	Geometric flexibility
Spectral Methods	Accuracy
Spectral Element Methods	FEM pros + accuracy

Table 4.1: Pros of the different discretization techniques.

Method	Cons
Finite Difference Methods	Accuracy, simple geometries
Finite Element Methods	Slow, hard to implement
Finite Volume Methods	Complex mathematics, hard to implement
Spectral Methods	Simple domains
Spectral Element Methods	Slow, hard to implement

Table 4.2: Cons of the different discretization techniques.

Going into detail, finite difference methods (FDM) are a class of numerical techniques for solving differential equations by approximating derivatives with finite differences.
In particular, for the spatial case, the most common approximation we will use is the following:

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T(x+\Delta x) - 2T(x) + T(x-\Delta x)}{\Delta x^2}$$

This is a second-order scheme since its accuracy improves quadratically as the step size, denoted as Δx, decreases.

4.3 Temporal discretization

Finite Difference Methods stand out as the prevailing choice for temporal discretization. Furthermore, temporal discretization techniques can be divided into two main categories:

- **Single-Point methods**
- **Multipoint methods**

Our focus lies on Multipoint methods, specifically **Linear Multipoint Methods (LMM)**. Linear Multipoint Methods (LMM) for equations of the form $\dot{U} = f(t, u)$ can be elegantly represented as:

$$\sum_{i=0}^{r} \alpha_i U^{n+i} = \Delta t \sum_{i=0}^{r} \beta_i f(t_n + i\Delta t, U^{n+i}) \tag{4.1}$$

where α_i and β_i are the coefficients of the method.
Various types of LMMs exist but the two primary categories are the **Adams methods** and the **BDF methods**.

Adams methods

Characterized by:

$$\alpha_r = 1, \quad \alpha_{\text{r-1}} = -1, \quad \alpha_i = 0 \quad \forall i < r-1 \tag{4.2}$$

We have that a generic Adams method can be expressed as:

$$U^{n+r} - U^{n+r-1} = \Delta t \sum_{i=0}^{r} \beta_i f(t_n + i\Delta t, U^{n+i}) \tag{4.3}$$

BDF methods

Characterized by:

$$\beta_r = 1, \quad \beta_i = 0 \quad \forall i \neq r \tag{4.4}$$

In the subsequent chapter, we will employ either the Explicit Euler method or the Crank-Nicolson method, both falling under the category of Adams methods.

$$\frac{U^{n+1} - U^n}{\Delta t} = f(t_n, U^n) \quad \text{(Explicit Euler)} \tag{4.5}$$

$$\frac{U^{n+1} - U^n}{\Delta t} = \frac{1}{2}(f(t_n, U^n) + f(t_n + \Delta t, U^{n+1})) \quad \text{(Crank-Nicolson)} \tag{4.6}$$

It's worth noting that the stability of the Explicit Euler method depends on the chosen time step size, making it conditionally stable. In contrast, the Crank-Nicolson method is unconditionally stable.

Chapter 5

Code development

In this chapter, we'll delve into various code implementations, beginning with the simplest scenario and progressively introducing complexity to encompass more realistic cases. We'll employ the manufactured solution technique for each scenario to validate the program's functionality and assess the accuracy of the numerical solution obtained. Additionally, we'll analyze the computational time t, considering both the spatial and temporal resolution (Δx and Δt, respectively).

5.1 1D Time-Independent Heat Equation

In this section, we tackle a 1D domain under a time-independent scenario, governed by the heat equation:

$$-\frac{\mathrm{d}^2T}{\mathrm{d}x^2} = f(x) \tag{5.1}$$

Employing a finite difference approach with equally spaced points throughout our domain, we approximate the second derivative as:

$$\frac{\mathrm{d}^2T}{\mathrm{d}x^2} \approx \frac{T(x+\Delta x) - 2T(x) + T(x-\Delta x)}{\Delta x^2} \tag{5.2}$$

where Δx represents the spatial resolution. Substituting this into the heat equation yields:

$$\frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2} = -f(x_i) \tag{5.3}$$

Here, T_i denotes the temperature at the i-th point.
Now, using the manufactured solution $T(x) = \sin(x)$, we get the source term $f(x) = -\sin(x)$. Hence, Equation (5.3) is transformed into:

$$\frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2} = sin(x_i) \tag{5.4}$$

With the domain length L set to 1 and $n+2$ spatial nodes, we compute $\Delta x = \frac{L}{n+1}$. Thus, the problem transforms into the following system of equations:

$$\frac{1}{\Delta x^2}\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_{n-1} \\ T_n \end{bmatrix} = \begin{bmatrix} sin(x_1) + \frac{sin(0)}{\Delta x^2} \\ sin(x_2) \\ \vdots \\ sin(x_{n-1}) \\ sin(x_n) + \frac{sin(L)}{\Delta x^2} \end{bmatrix}$$

Recognizing that the left matrix is tridiagonal, we employ the Thomas algorithm to solve the system efficiently. The Thomas algorithm is a simplified form of Gaussian elimination that can be used to solve this type of systems of equations.
The code implementing the Thomas algorithm is shown below:

```
void thomasAlgorithm(
                double* solution,
                double* diag,
                double* upperDiag,
                double* lowerDiag,
                double* rhs,
                int dim){
    double w;

    for(int i = 1; i < dim; i++){
        w = lowerDiag[i - 1] / diag[i - 1];
        diag[i] -= w * upperDiag[i - 1];
        rhs[i] -= w * rhs[i - 1];
    }
    solution[dim - 1] = rhs[dim - 1]/diag[dim - 1];
    for(int i = dim - 2; i >= 0; i--){
        solution[i] = (rhs[i] - upperDiag[i] *
                       solution[i + 1]) / diag[i];
    }
}
```

The complete source code can be found on the GitHub repository HeatEquationSolver by francescomicucci (`https://github.com/francescomicucci/HeatEquationSolver/tree/main/`). Look for the file named `heat1D_TI.cpp` located in the subfolder '1D-TimeIndependent'.

Moreover, a summary of the results is presented in Table 5.1.

Number of spatial points n	*Error*	Computational time (s) / n
100	3.53189e-08	4.9533e-07
1000	1.13202e-10	1.58912e-07
10000	3.4015e-13	1.82419e-07
100000	2.13598e-13	1.83848e-07
1000000	2.62698e-13	8.49804e-08
10000000	4.22898e-11	7.88384e-08

Table 5.1: Results for the 1D time-independent heat equation

Looking at Figure 5.1, we observe a declining trend in error as the number of spatial

points increases (the error decreases proportionally to Δx^2). However, this decrease stabilizes once a specific threshold is surpassed. Meanwhile, looking at Figure 5.2, we observe that the computational time per point remains relatively constant as the number of spatial points increases. This happens because of the Thomas algorithm's computational complexity, which scales linearly with the number of spatial points.

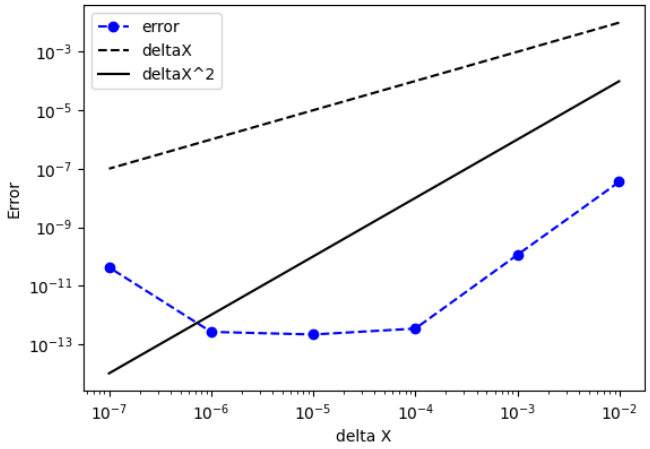


Figure 5.1: Δx vs *Error* for the 1D time-independent case

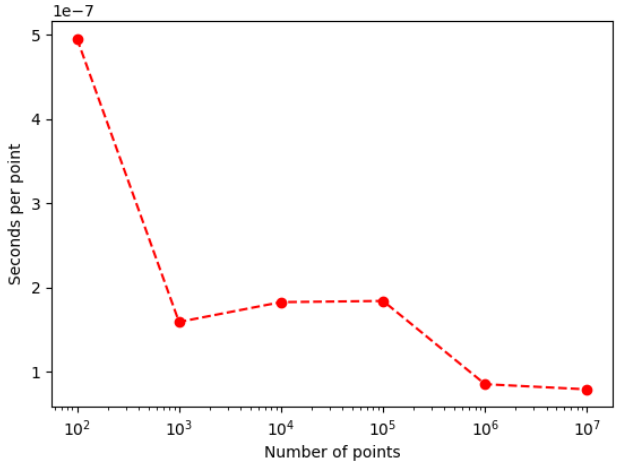


Figure 5.2: (Computational time / n) vs n for the 1D time-independent case. (It's crucial to highlight that the values along the y-axis are on the order of 1e-7)

5.2 1D Time Dependent Heat Equation

In this section, we will focus on the heat equation within a one-dimensional spatial context, integrating its temporal dependence. Therefore, the heat equation takes the following form:

$$\frac{\partial T}{\partial t} - \frac{\partial^2 T}{\partial x^2} = f(x,t) \tag{5.5}$$

In this case, we resort to finite difference methods, approximating the derivatives as follows:

$$\frac{\partial T}{\partial t} \approx \frac{T(x, t+\Delta t) - T(x,t)}{\Delta t} \tag{5.6}$$

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T(x+\Delta x, t) - 2T(x,t) + T(x-\Delta x, t)}{\Delta x^2} \tag{5.7}$$

Here, Δx and Δt represent the spatial and temporal resolutions, respectively.
For the spatial domain, we set its length L to 1 and we distribute $n_x + 2$ spatial nodes evenly within it. Consequently, we compute the spatial resolution Δx as $L/(n_x + 1)$.
Similarly, in the temporal domain, we divide the time interval using $n_t + 1$ temporal nodes, setting $T = 1$ as the total duration. Thus, the temporal resolution Δt becomes T/n_t.
When solving this problem, we'll use the Explicit Euler method for time discretization. Because of the Explicit Euler method, it's important to carefully choose values for n_x and n_t to maintain stability. Specifically, we need to ensure that Δt is not greater than $\frac{\Delta x^2}{2}$.
Now, substituting (5.6) and (5.7) into the heat equation yields:

$$\frac{T_k^{n+1} - T_k^n}{\Delta t} - \frac{T_{k+1}^n - 2T_k^n + T_{k-1}^n}{\Delta x^2} = f(x_k, t_n) \tag{5.8}$$

where T_k^n denotes the temperature at the k-th spatial node and n-th temporal node.
In this scenario, we adopt the manufactured solution $T(x,t) = \sin(x)\sin(t)$, leading to the following derivatives:

$$\frac{\partial T}{\partial t} = \sin(x)\cos(t) \quad \frac{\partial^2 T}{\partial x^2} = -\sin(x)\sin(t) \tag{5.9}$$

Substituting these derivatives into the heat equation yields:

$$f(x,t) = \sin(x)\cos(t) + \sin(x)\sin(t) = \sin(x)(\cos(t) + \sin(t)) \tag{5.10}$$

Thus, equation (5.8) becomes:

$$\frac{T_k^{n+1} - T_k^n}{\Delta t} - \frac{T_{k+1}^n - 2T_k^n + T_{k-1}^n}{\Delta x^2} = \sin(x_k)(\cos(t_n) + \sin(t_n)) \tag{5.11}$$

or, written differently:

$$T_k^{n+1} = T_k^n + \frac{\Delta t}{\Delta x^2}(T_{k+1}^n - 2T_k^n + T_{k-1}^n) + \Delta t \sin(x_k)(\cos(t_n) + \sin(t_n)) \tag{5.12}$$

In this particular case, the code was implemented in two different languages: C++ and Fortran. This enables us not only to address the heat equation within this context but also to evaluate its performance in both languages.
Using Fortran, the core of the algorithm is implemented as follows:

```
coeff = dt/(dx*dx)

DO WHILE (time < 1.d0)
    time = time + dt
    T(0) = 0.d0
    T(1:nx) = To(1:nx) * (1.d0 - 2.d0*coeff) +
```

```
            + coeff * (To(0:nx-1) + To(2:nx+1)) +
            + dt * sin(x(1:nx)) * (sin(time) + cos(time))
    T(nx+1) = sin(1.d0) * sin(time)
    To = T
END DO
```

Alternatively, the C++ implementation takes this form:

```
coeff = dt / (dx * dx);

while(time < 1.0){
    // Update time
    time = time + dt;

    // Update solution
    solution[0] = 0.0;
    for(int i = 1; i < nx + 1; i++){
        solution[i] = oldSolution[i]*(1.0-2.0*coeff) +
                    coeff*(oldSolution[i - 1] +
                    oldSolution[i + 1])+ dt*std::sin(i * dx)*
                    (std::sin(time) + std::cos(time));
    }
    solution[nx + 1] = std::sin(1.0) * std::sin(time);

    // Update oldSolution
    for(int i = 0; i < nx + 2; i++){
        oldSolution[i] = solution[i];
    }
}
```

The complete source code can be found on the GitHub repository HeatEquationSolver by francescomicucci (`https://github.com/francescomicucci/HeatEquationSolver/tree/main/`). Look for files named `heat_eq_T1D.f90` and `heat1D_TD.cpp` located in the subfolder '1D-TimeDependent'.

To evaluate the performance of the Fortran and C++ implementations, we analyzed the results presented in Table 5.2, which covers three different scenarios with varying values of n_x and n_t. The results consistently demonstrate that the Fortran implementation outperforms the C++ implementation in terms of execution time. This is attributed to Fortran's memory management and the compiler's ability to vectorize the code.
In addition, we also assessed the accuracy of both implementations. Interestingly, the Fortran implementation maintained superior accuracy despite employing the same computational steps as the C++ implementation. This observation suggests that the Fortran code is more efficient in utilizing the underlying hardware.

n_x	n_t	$time_{Fortran}$	$time_{C++}$	$error_{Fortran}$	$error_{C++}$
20	6400	1.348 ms	3.398 ms	1.80396e-06	1.08376e-05
40	6400	2.533 ms	6.693 ms	5.1194e-07	7.17799e-06
40	12800	4.993 ms	13.654 ms	3.8909e-07	3.88971e-07

Table 5.2: 1D time-dependent heat equation: Fortran vs C++

Moving ahead, we carried out two separate analyses utilizing the Fortran implementation. In the first analysis, we set n_t equal to 25600 and we allowed n_x to vary (Table 5.3). In the second analysis, we maintained n_x equal to 40 while varying n_t (Table 5.4).

As expected, the error decreases with increasing spatial and temporal nodes, while the normalized computational time (normalized in the sense that we divided the computational time by the product of n_x and n_t) remains approximately constant.

n_x	n_t	$time_{Fortran}/(n_x \cdot n_t)$	$error_{Fortran}$
10	25600	1.113e-05 ms	7.7857e-06
20	25600	9.814e-06 ms	1.5399e-06
40	25600	9.606e-06 ms	3.2759e-07
80	25600	9.067e-06 ms	9.1131e-08

Table 5.3: 1D time-dependent heat equation: Results for a predetermined value of n_t.

n_x	n_t	$time_{Fortran}/(n_x \cdot n_t)$	$error_{Fortran}$
40	6400	9.422e-06 ms	5.1194e-07
40	12800	9.761e-06 ms	3.8909e-07
40	25600	9.789e-06 ms	3.2759e-07

Table 5.4: 1D time-dependent heat equation: Results for a predetermined value of n_x.

We can conclude by saying that using Fortran for this problem proves to be both effective and scalable. Indeed, as the number of spatial and temporal nodes increases, the error diminishes, while the normalized execution time stays relatively stable.

5.3 3D Time Dependent Heat Equation

In this section, we will address the heat equation within a three-dimensional spatial context. Here, the formulation of the heat equation takes shape as follows:

$$\frac{\partial T}{\partial t} - \nabla^2 T = f \tag{5.13}$$

To solve this equation, we'll once again use finite difference methods introducing the following approximations for the derivatives:

$$\frac{\partial T}{\partial t} \approx \frac{T(x,y,z,t+\Delta t) - T(x,y,z,t)}{\Delta t} \tag{5.14}$$

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T(x+\Delta x,y,z,t) - 2T(x,y,z,t) + T(x-\Delta x,y,z,t)}{\Delta x^2} \tag{5.15}$$

$$\frac{\partial^2 T}{\partial y^2} \approx \frac{T(x,y+\Delta y,z,t) - 2T(x,y,z,t) + T(x,y-\Delta y,z,t)}{\Delta y^2} \tag{5.16}$$

$$\frac{\partial^2 T}{\partial z^2} \approx \frac{T(x,y,z+\Delta z,t) - 2T(x,y,z,t) + T(x,y,z-\Delta z,t)}{\Delta z^2} \tag{5.17}$$

In this context, Δx, Δy, and Δz denote the spatial resolutions along the x, y, and z axes respectively, while Δt stands for the temporal resolution. Furthermore, we'll adopt the assumption that $\Delta x = \Delta y = \Delta z$, indicating uniform spacing of points across all three spatial dimensions.
For the spatial domain, we set the length of each side to 1 and distribute $n + 2$ spatial

nodes evenly within this length. Consequently, we compute the spatial resolution Δx as $1/(n+1)$.
In the temporal domain, we divide the time interval using $n_t + 1$ temporal nodes, setting $T = 1$ as the total duration. Thus, the temporal resolution Δt becomes T/n_t.
When tackling this problem, we'll employ the Crank-Nicolson method for time discretization. We opted for this approach over the Explicit Euler method because the advantages of the latter aren't substantial enough to justify the constraints it imposes on Δt. The Crank-Nicolson method is a more stable and accurate method that allows for larger time steps.
Now, substituting (5.14) to (5.17) into the heat equation yields:

$$\begin{aligned}(\frac{1}{\Delta t} + \frac{3}{\Delta x^2})T^{n+1}_{i,j,k} - \frac{1}{2\Delta x^2}(T^{n+1}_{i+1,j,k} + T^{n+1}_{i-1,j,k} + T^{n+1}_{i,j+1,k} + T^{n+1}_{i,j-1,k} + T^{n+1}_{i,j,k+1} + T^{n+1}_{i,j,k-1}) = \\ (\frac{1}{\Delta t} - \frac{3}{\Delta x^2})T^{n}_{i,j,k} + \frac{1}{2\Delta x^2}(T^{n}_{i+1,j,k} + T^{n}_{i-1,j,k} + T^{n}_{i,j+1,k} + T^{n}_{i,j-1,k} + T^{n}_{i,j,k+1} + T^{n}_{i,j,k-1}) + f^{n+\frac{1}{2}}_{i,j,k}\end{aligned} \tag{5.18}$$

where $T^n_{i,j,k}$ represents the temperature at spatial coordinates (i, j, k) and time step n.
In this scenario, we adopt the manufactured solution:

$$T(x, y, z, t) = \sin(x)\sin(y)\sin(z)\sin(t) \tag{5.19}$$

Substituting this solution into the heat equation yields:

$$f(x, y, z, t) = \sin(x)\sin(y)\sin(z)(\cos(t) + 3\sin(t)) \tag{5.20}$$

Since we are currently working in the three-dimensional case, we will use the following mapping so that each point on the grid corresponds exclusively to a single position in the unknowns vector when we write the code:

$$(i, j, k) \to i + (j-1)\cdot n + (k-1)\cdot n^2 \quad i, j, k = 1, 2, ..., n \tag{5.21}$$

By using this mapping, the matrix associated with the linear system we need to solve will have a structure with multiple bands. For example, when $n = 3$, the matrix will show non-zero elements as illustrated in Figure 5.3.
Given the characteristics of the linear system matrix, employing the Thomas method is no longer viable. Thus, the initial strategy during code development was to use an iterative method to compute the numerical solution, specifically the Conjugate Gradient method. To facilitate this, we used the Eigen library for matrix declaration, vector creation, and solver functionalities.
Three key functions were developed:

- `buildMatrix`: Constructs the matrix required for solving the system.

```
void buildMatrix(SpMat &A, int n, double dx, double dt){
  double diagElem = 1.0 / dt + 3.0 / (dx * dx);
  double noDiagElem = - 1.0 / (2.0 * dx * dx);

  for(int i = 1; i <= n; i++){
    for(int j = 1; j <= n; j++){
      for(int k = 1; k <= n; k++){
        A.coeffRef(getIndex(i,j,k,n), getIndex(i,j,k,n))
```

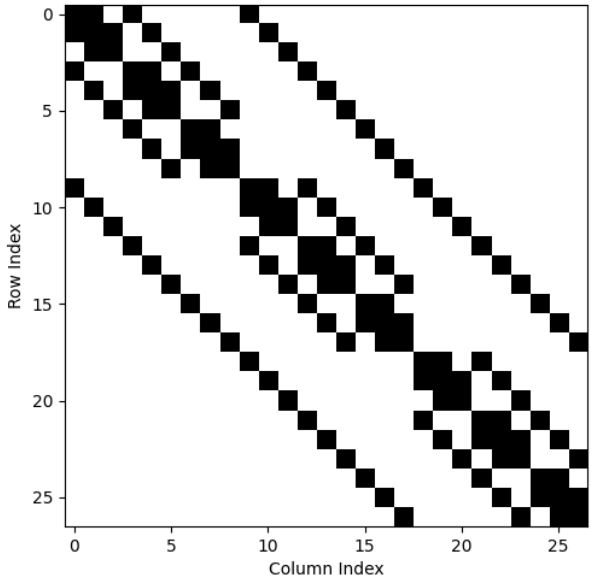


Figure 5.3: Non-zero elements of the matrix for the 3D time-dependent case with $n = 3$

```
          = diagElem;

        if(i > 1)
          A.coeffRef(getIndex(i,j,k,n), getIndex(i-1,j,k,n))
            = noDiagElem;
        if(j > 1)
          A.coeffRef(getIndex(i,j,k,n), getIndex(i,j-1,k,n))
            = noDiagElem;
        if(k > 1)
          A.coeffRef(getIndex(i,j,k,n), getIndex(i,j,k-1,n))
            = noDiagElem;

        if(i <= n - 1)
          A.coeffRef(getIndex(i,j,k,n), getIndex(i+1,j,k,n))
            = noDiagElem;
        if(j <= n - 1)
          A.coeffRef(getIndex(i,j,k,n), getIndex(i,j+1,k,n))
            = noDiagElem;
        if(k <= n - 1)
          A.coeffRef(getIndex(i,j,k,n), getIndex(i,j,k+1,n))
            = noDiagElem;
      }
    }
  }
}
```

- `buildRhs`: Constructs the right-hand side (RHS) vector.

```
void buildRhs(SpVec &b, SpVec &oldSolution, int n,
              double dx, double dt, double time){
  int index;
  for(int i = 0; i < n * n * n; i++){
    b.coeffRef(i) = 0.0;
  }
  for(int i = 1; i <= n; i++){
```

```
    for(int j = 1; j <= n; j++){
      for(int k = 1; k <= n; k++){
        index = getIndex(i, j, k, n);

        b.coeffRef(index) += 0.5 *
          (fFuntion(i, j, k, time - dt, dx) +
          fFuntion(i, j, k, time, dx));
        b.coeffRef(index) += oldSolution.coeffRef(index)
          * (1.0 / dt - 3.0 / (dx * dx));

        if(i > 1)
          b.coeffRef(index) += 0.5 * (1.0 / (dx * dx)) *
            oldSolution.coeffRef(getIndex(i - 1, j, k, n));
        if(j > 1)
          b.coeffRef(index) += 0.5 * (1.0 / (dx * dx)) *
            oldSolution.coeffRef(getIndex(i, j - 1, k, n));
        if(k > 1)
          b.coeffRef(index) += 0.5 * (1.0 / (dx * dx)) *
            oldSolution.coeffRef(getIndex(i, j, k - 1, n));

        if(i < n)
          b.coeffRef(index) += 0.5 * (1.0 / (dx * dx)) *
            oldSolution.coeffRef(getIndex(i + 1, j, k, n));
        if(j < n)
          b.coeffRef(index) += 0.5 * (1.0 / (dx * dx)) *
            oldSolution.coeffRef(getIndex(i, j + 1, k, n));
        if(k < n)
          b.coeffRef(index) += 0.5 * (1.0 / (dx * dx)) *
            oldSolution.coeffRef(getIndex(i, j, k + 1, n));

        // BCs
        if(i == n)
          b.coeffRef(index) += 0.5 * (1.0 / (dx * dx)) *
            sin(1.0)*sin(j*dx)*sin(k*dx)*(sin(time-dt)+sin(time));
        if(j == n)
          b.coeffRef(index) += 0.5 * (1.0 / (dx * dx)) *
            sin(i*dx)*sin(1.0)*sin(k*dx)*(sin(time-dt)+sin(time));
        if(k == n)
          b.coeffRef(index) += 0.5 * (1.0 / (dx * dx)) *
            sin(i*dx)*sin(j*dx)*sin(1.0)*(sin(time-dt)+sin(time));
      }
    }
  }
}
```

- `solveSystem`: Uses the Conjugate Gradient method to solve the system.

```
void solveSystem(SpMat &A, SpVec &b, SpVec &u){
    Solver solver;
    solver.compute(A);
    u = solver.solve(b);
}
```

These functions are essential for computing the numerical solution at each time step. Notably, since the matrix remains constant across all time steps, the `buildMatrix` function is invoked only once.

```
buildMatrix(A, n, dx, dt);

while(t <= 1.0){
    buildRhs(b, u, n, dx, dt, t);
```

```
    solveSystem(A, b, u);
    t += dt;
}
```

The complete source code can be found on the GitHub repository HeatEquationSolver by francescomicucci (`https://github.com/francescomicucci/HeatEquationSolver/tree/main/`). Look for the file named `heat3D_TD.cpp` located in the subfolder '3D-TimeDependent'.

Finally, we assessed the efficacy of the 3D time-dependent heat equation solver across different values of n. The findings, showcased in Table 5.5, unveil an interesting trend. While the error diminishes with an increase in the number of spatial points employed, there's a notable drawback: the time investment for constructing the right-hand side and resolving the linear systems escalates significantly.

n	n^3	$time_{buildMatrix}$	$time_{buildRHS} + time_{solveSystem}$	error
3	27	0.079722 ms	0.00821542 s	0.000123542
10	1000	67.0689 ms	1.25639 s	2.3375e-05
25	15625	24874 ms	55.7663 s	6.09001e-06

Table 5.5: 3D Time-dependent Heat Equation: Results for a predetermined value of n_t.

5.4 Douglas method

Since solving the 3D Time Dependent Heat equation can be quite time-consuming, especially when the number of spatial points is high, to get better results in terms of computational time, we have explored the so called Douglas method.
This method, developed by Douglas, is a clever trick. It lets us break down the big, complex system of equations we had before into much simpler ones.
In our specific case, we will need to solve the following three systems:

$$\begin{aligned}(1-\frac{\Delta t}{2}\partial_{xx})(T^*-T^n) &= \Delta t(\partial_{xx}+\partial_{yy}+\partial_{zz})T^n + \Delta t f^{n+1/2} \\ (1-\frac{\Delta t}{2}\partial_{yy})(T^{**}-T^n) &= T^*-T^n \\ (1-\frac{\Delta t}{2}\partial_{zz})(T^{n+1}-T^n) &= T^{**}-T^n\end{aligned} \tag{5.22}$$

where T^n represents the solution at time n.
Utilizing the Finite Difference Methods explored in the preceding section, we can deduce that the three systems of equations exhibit a tridiagonal pattern. Additionally, the matrices representing these systems are identical and have the following structure:

$$\begin{bmatrix} b & c & 0 & \cdots & 0 \\ a & b & c & \ddots & \vdots \\ 0 & a & b & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & c \\ 0 & \cdots & 0 & a & b \end{bmatrix}$$

where $a = c = -\frac{\Delta t}{2\Delta x^2}$ and $b = 1 + \frac{\Delta t}{\Delta x^2}$.
The systems differ due to variations in their respective right-hand sides. Specifically, in the initial system, the right-hand side is computed as:

$$rhs_{i,j,k} = \Delta t f_{i,j,k}^{n+\frac{1}{2}} + \frac{\Delta t}{\Delta x^2}(T_{i+1,j,k}^n + T_{i-1,j,k}^n + T_{i,j+1,k}^n + T_{i,j-1,k}^n + T_{i,j,k+1}^n + T_{i,j,k-1}^n - 6T_{i,j,k}^n)$$

Which implemented in C++ becomes:

```
for(int i = 1; i <= n; i++){
  for(int j = 1; j <= n; j++){
    for(int k = 1; k <= n; k++){
      index = getIndex(i, j, k);
      rhs[index] = dt * (fFunction(i, j, k, time, dx)
                 + fFunction(i, j, k, time + dt, dx)) / 2.0;
      rhs[index] -= 6.0 * timeStepSolution[index] * coeff;

      if(i > 1)
        {rhs[index] += coeff*timeStepSolution[getIndex(i - 1, j, k)];}
      if(i < n)
        {rhs[index] += coeff*timeStepSolution[getIndex(i + 1, j, k)];}
      else
        {rhs[index] += coeff*sin(1.0)*sin(j*dx)*sin(k*dx)*sin(time);}

      if(j > 1)
        {rhs[index] += coeff*timeStepSolution[getIndex(i, j - 1, k)];}
      if(j < n)
        {rhs[index] += coeff*timeStepSolution[getIndex(i, j + 1, k)];}
      else
        {rhs[index] += coeff*sin(1.0)*sin(i*dx)*sin(k*dx)*sin(time);}

      if(k > 1)
        {rhs[index] += coeff*timeStepSolution[getIndex(i, j, k - 1)];}
      if(k < n)
        {rhs[index] += coeff*timeStepSolution[getIndex(i, j, k + 1)];}
      else
        {rhs[index] += coeff*sin(1.0)*sin(i*dx)*sin(j*dx)*sin(time);}
    }
  }
}

// BCs on x
for(int k = 1; k < n; k++){
  for(int j = 1; j < n; j++){
    rhs[getIndex(n, j, k)] -= noDiagElem * sin(1.0) * sin(j * dx)
                    * sin(k * dx) * (sin(time + dt) - sin(time));
  }
}

// BCs on y
for(int k = 1; k < n; k++){
  for(int j = 1; j < n; j++){
    rhs[getIndex(j, n, k)] -= noDiagElem * sin(1.0) * sin(j * dx)
                    * sin(k * dx) * (sin(time + dt) - sin(time));
  }
}

// BCs on z
for(int k = 1; k < n; k++){
  for(int j = 1; j < n; j++){
    rhs[getIndex(j, k, n)] -= noDiagElem * sin(1.0) * sin(j * dx)
                    * sin(k * dx) * (sin(time + dt) - sin(time));
  }
```

```
}
```

Meanwhile, for the second and third systems, the right-hand side is the solution of the preceding system.
Because of the tridiagonal structure of the matrices, we apply again the Thomas algorithm to solve the systems.

It's crucial to observe that each system is solved in a specific direction: the first along the x axis, the second along the y axis, and the third along the z axis.

The complete source code can be found on the GitHub repository HeatEquationSolver by francescomicucci (`https://github.com/francescomicucci/HeatEquationSolver/tree/main/`). Look for the file named `heat3D_TD_Douglas.cpp` located in the subfolder '3D-TimeDependent'.

Just to make a comparison with the implementation discussed in the previous section, it's worth noting that the Douglas method is notably faster. For instance, with $n = 25$ and $n_t = 100$, the computational time reduces to approximately 0.5 seconds, whereas in the previous scenario, it was around 55 seconds.

5.5 Parallel Implementation

As our problem's complexity increases, it becomes crucial to consider parallel implementation for solving it within a reasonable timeframe. This involves splitting the problem into smaller tasks that can be tackled simultaneously, leveraging the power of multiple processors. Specifically, we will delve into three key techniques: Pipeline, 2D Pencil Decomposition, and Schur Complement. Each strategy presents its way to divide and conquer, ultimately helping us solve the problem faster.

5.5.1 Pipeline

We now explore the first technique we'll use to speed up the solving process of the 3D time-dependent heat equation: the pipeline.
The main idea behind this technique is to divide our 3D domain into blocks, each assigned to a specific processor. Naturally, the number of blocks corresponds to the number of processors engaged, with each processor possessing coordinates aligned with the block's position along the x, y, and z axes.
The pipeline technique works on the premise that each processor tackles the problem within a distinct part of the domain, resulting in computations that overlap.

Let's examine the Thomas algorithm example along the x axis. Initially, only processors located at x-coordinate 0 begin their operations, while others remain inactive due to the absence of crucial data needed to commence computation. Once this essential data arrives, these processors can initiate their calculations. It's important to note that this data is transmitted from processors with a lower x-coordinate relative to the recipient (specifically, $x_{\text{receiver}} = x_{\text{sender}} + 1$). Additionally, it's worth mentioning that the required data for computation on a new processor isn't solely produced at the end of another processor's computation, but rather during its execution. As a result, there's an overlap in computation between the two processors. Naturally, when additional missing data is required for computation, the processor awaits their transmission from the appropriate source. This process unfolds similarly along the y and z axes.

At the end of each time step, processors update the solution for the points they oversee and exchange necessary information with other processors to prepare for subsequent time steps.

An important optimization that has been done during the implementation of the pipeline method is to change the way data are stored in memory depending on which axis we are solving the problem. In this way, we aim to minimize the number of cache misses, which can significantly slow down the computation.

For which concern the implementation, we used the C++ language and the `MPI` library for communication between processors.
In terms of changes from the sequential implementation, the following additions/modifications were introduced:

- Initialization of `MPI` at the beginning and definition of the Cartesian communicator.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

MPI_Dims_create(size, ndims, dims);
MPI_Cart_create(MPI_COMM_WORLD,ndims, dims, periods, 1, &commCart);
```

- Retrieval of the current processor's coordinates and other pertinent information.

```
MPI_Cart_coords(commCart, rank, ndims, coord);

int nxSubdomain = n / dims[0];
int nySubdomain = n / dims[1];
int nzSubdomain = n / dims[2];
points = {nxSubdomain, nySubdomain, nzSubdomain};

for(int d = 0; d < ndims; d++){
    start[d] = coord[d] * points[d];
    if(coord[d] != dims[d] - 1)
        end[d] = (coord[d] + 1) * points[d] - 1;
    else
        end[d] = n - 1;
}

localSize = {end[0] - start[0] + 1,
             end[1] - start[1] + 1,
             end[2] - start[2] + 1};
```

- Adjustments to the construction of the right-hand side vector, now considering solely the points assigned to the processor.

```
void rhs(double time){
  int index;
  const int sizeX = localSize[0] + 2;
  const int sizeY = localSize[1] + 2;
  const double coeff = dt / (dx * dx);

  for(int i = start[0]; i <= end[0]; i++){
    for(int j = start[1]; j <= end[1]; j++){
      for(int k = start[2]; k <= end[2]; k++){
        index = (i - start[0] + 1) + (j - start[1] + 1) * sizeX
              + (k - start[2] + 1) * sizeX * sizeY;

        deltaSol[index] = dt * (fFunction(i, j, k, time + dt / 2));
        deltaSol[index] -= 6.0 * coeff * localSol[index];

        if(i > 0)
          deltaSol[index] += coeff * localSol[index - 1];
        if(i < n - 1)
          deltaSol[index] += coeff * localSol[index + 1];
        else
          deltaSol[index] += coeff * sin(1.0) * sin((j + 1) * dx)
                           * sin((k + 1) * dx) * sin(time);

        if(j > 0)
          deltaSol[index] += coeff * localSol[index - sizeX];
        if(j < n - 1)
          deltaSol[index] += coeff * localSol[index + sizeX];
        else
          deltaSol[index] += coeff * sin((i + 1) * dx) * sin(1.0)
                           * sin((k + 1) * dx) * sin(time);

        if(k > 0)
```

```
          deltaSol[index] += coeff * localSol[index-sizeX*sizeY];
        if(k < n - 1)
          deltaSol[index] += coeff * localSol[index+sizeX*sizeY];
        else
          deltaSol[index] += coeff * sin((i + 1) * dx)
                          * sin((j+1)*dx)*sin(1.0)*sin(time);
      }
    }
  }
}
```

- Adaptation of the Thomas algorithm to solve the system along the x, y, and z axes, including conditions for awaiting necessary data and transmitting computed data to the requiring processors, both for the forward and backward sweeps of the algorithm.

```
void thomas(const int direction){
  int index;
  int rank_source, rank_prev, rank_next;
  const int size0 = localSize[(direction + 0) % 3] + 2;
  const int size1 = localSize[(direction + 1) % 3] + 2;
  const int start0 = start[(direction + 0) % 3];
  const int start1 = start[(direction + 1) % 3];
  const int start2 = start[(direction + 2) % 3];
  const int end0 = end[(direction + 0) % 3];
  const int end1 = end[(direction + 1) % 3];
  const int end2 = end[(direction + 2) % 3];

  MPI_Cart_shift(commCart,direction,-1,&rank_source,&rank_prev);
  MPI_Cart_shift(commCart,direction,1,&rank_source,&rank_next);

  // Forward sweep
  for(int k = start2; k <= end2; k++){
    for(int j = start1; j <= end1; j++){
      index = (j - start1 + 1)*size0+(k - start2 + 1)*size0*size1;

      if(coord[direction] != 0)
        MPI_Recv(&deltaSol[index], 1, MPI_DOUBLE, rank_prev,
          getGlobalIndex(start0, j, k),commCart,MPI_STATUS_IGNORE);

      for(int i = start0; i <= end0; i++){
        if(i != 0)
          deltaSol[index+i-start0+1]-=(offDiagElem
                        *deltaSol[index+i-start0]) / diag[i - 1];
      }

      if(coord[direction] != dims[direction] - 1)
        MPI_Send(&deltaSol[index+end0-start0+1], 1, MPI_DOUBLE,
                rank_next, getGlobalIndex(end0+1, j, k), commCart);
    }
  }

  // Backward sweep
  for(int k = start2; k <= end2; k++){
    for(int j = start1; j <= end1; j++){
      index = (j-start1+1)*size0 + (k-start2+1)*size0*size1;
      if(coord[direction] == dims[direction] - 1)
        deltaSol[index + end0 - start0 + 1] /= diag[n - 1];
      else
        MPI_Recv(&deltaSol[index+end0-start0+2], 1, MPI_DOUBLE,
                 rank_next,getGlobalIndex(end0,j,k),commCart,
                 MPI_STATUS_IGNORE);
```

```
      for(int i = end0; i >= start0; i--){
        if(i != n - 1)
          deltaSol[index+i-start0+1] = (deltaSol[index+i-start0+1]
            - offDiagElem*deltaSol[index+i-start0+2]) / diag[i];
      }

      if(coord[direction] != 0)
        MPI_Send(&deltaSol[index + 1], 1, MPI_DOUBLE, rank_prev,
                 getGlobalIndex(start0 - 1, j, k), commCart);
    }
  }
}
```

- Rearrange the data in the memory depending on the axis along which we are solving the problem.

```
void rotate(const int direction){
  const int size0 = localSize[(direction + 0) % 3] + 2;
  const int size1 = localSize[(direction + 1) % 3] + 2;
  const int size2 = localSize[(direction + 2) % 3] + 2;

  vector<double> temp(size0 * size1 * size2);

  for(int i = 0; i < size0 * size1 * size2; i++)
    temp[i] = deltaSol[i];

  for(int i = 1; i <= localSize[(direction + 0) % 3]; i++){
    for(int j = 1; j <= localSize[(direction + 2) % 3]; j++){
      for(int k = 1; k <= localSize[(direction + 1) % 3]; k++){
        deltaSol[i*size2*size1+j*size1+k] =
            temp[j*size1*size0+k*size0+i];
      }
    }
  }
}
```

- Exchange of results at each time step conclusion among processors.

```
void comunication(){
  const int sizeX = localSize[0] + 2;
  const int sizeY = localSize[1] + 2;

  int rank_source, rank_prev, rank_next;

  MPI_Cart_shift(commCart, 0, -1, &rank_source, &rank_prev);
  MPI_Cart_shift(commCart, 0, 1, &rank_source, &rank_next);

  vector<double> send(localSize[1] * localSize[2]);
  vector<double> recv(localSize[1] * localSize[2]);

  // Send to prev on x axis
  for(int i = 1; i <= localSize[2]; i++){
    for(int j = 1; j <= localSize[1]; j++){
      send[(i - 1) * localSize[1] + (j - 1)] =
            localSol[i * sizeY * sizeX + j * sizeX + 1];
    }
  }
  MPI_Sendrecv(send.data(), localSize[1]*localSize[2], MPI_DOUBLE,
           rank_prev, 0, recv.data(), localSize[1] * localSize[2],
           MPI_DOUBLE, rank_next, 0, commCart, MPI_STATUS_IGNORE);
```

```
for(int i = 1; i <= localSize[2]; i++){
  for(int j = 1; j <= localSize[1]; j++){
    localSol[i * sizeY * sizeX + j * sizeX + localSize[0] + 1] =
        recv[(i - 1) * localSize[1] + (j - 1)];
  }
}

// Send to next on x axis
for(int i = 1; i <= localSize[2]; i++){
  for(int j = 1; j <= localSize[1]; j++){
    send[(i - 1) * localSize[1] + (j - 1)] =
          localSol[i * sizeY * sizeX + j * sizeX + localSize[0]];
  }
}
MPI_Sendrecv(send.data(), localSize[1]*localSize[2], MPI_DOUBLE,
             rank_next,0, recv.data(), localSize[1]*localSize[2],
             MPI_DOUBLE,rank_prev,0,commCart, MPI_STATUS_IGNORE);
for(int i = 1; i <= localSize[2]; i++){
  for(int j = 1; j <= localSize[1]; j++){
    localSol[i * sizeY * sizeX + j * sizeX] =
        recv[(i - 1) * localSize[1] + (j - 1)];
  }
}

MPI_Cart_shift(commCart, 1, -1, &rank_source, &rank_prev);
MPI_Cart_shift(commCart, 1, 1, &rank_source, &rank_next);

send.resize(localSize[0] * localSize[2]);
recv.resize(localSize[0] * localSize[2]);

// Send to prev on y axis
for(int i = 1; i <= localSize[2]; i++){
  for(int k = 1; k <= localSize[0]; k++){
    send[(i - 1) * localSize[0] + (k - 1)] =
         localSol[i * sizeY * sizeX + 1 * sizeX + k];
  }
}
MPI_Sendrecv(send.data(), localSize[2]*localSize[0], MPI_DOUBLE,
             rank_prev, 0, recv.data(),localSize[2]*localSize[0],
             MPI_DOUBLE,rank_next,0,commCart,MPI_STATUS_IGNORE);
for(int i = 1; i <= localSize[2]; i++){
  for(int k = 1; k <= localSize[0]; k++){
    localSol[i * sizeY * sizeX + (localSize[1] + 1) * sizeX + k]=
        recv[(i - 1) * localSize[0] + (k - 1)];
  }
}

// Send to next on y axis
for(int i = 1; i <= localSize[2]; i++){
  for(int k = 1; k <= localSize[0]; k++){
    send[(i - 1) * localSize[0] + (k - 1)] =
         localSol[i * sizeY * sizeX + localSize[1] * sizeX + k];
  }
}
MPI_Sendrecv(send.data(), localSize[2]*localSize[0], MPI_DOUBLE,
             rank_next,0,recv.data(),localSize[2]*localSize[0],
             MPI_DOUBLE,rank_prev,0,commCart,MPI_STATUS_IGNORE);
for(int i = 1; i <= localSize[2]; i++){
  for(int k = 1; k <= localSize[0]; k++){
    localSol[i * sizeY * sizeX + k] =
        recv[(i - 1) * localSize[0] + (k - 1)];
  }
}
```

```
  MPI_Cart_shift(commCart, 2, -1, &rank_source, &rank_prev);
  MPI_Cart_shift(commCart, 2, 1, &rank_source, &rank_next);

  send.resize(localSize[0] * localSize[1]);
  recv.resize(localSize[0] * localSize[1]);

  // Send to prev on z axis
  for(int j = 1; j <= localSize[1]; j++){
    for(int k = 1; k <= localSize[0]; k++){
      send[(j - 1) * localSize[0] + (k - 1)] =
           localSol[1 * sizeY * sizeX + j * sizeX + k];
    }
  }
  MPI_Sendrecv(send.data(), localSize[0]*localSize[1], MPI_DOUBLE,
               rank_prev, 0,recv.data(),localSize[0]*localSize[1],
               MPI_DOUBLE,rank_next,0,commCart, MPI_STATUS_IGNORE);
  for(int j = 1; j <= localSize[1]; j++){
    for(int k = 1; k <= localSize[0]; k++){
      localSol[(localSize[2] + 1) * sizeY * sizeX + j * sizeX + k]=
          recv[(j - 1) * localSize[0] + (k - 1)];
    }
  }

  // Send to next on z axis
  for(int j = 1; j <= localSize[1]; j++){
    for(int k = 1; k <= localSize[0]; k++){
      send[(j - 1) * localSize[0] + (k - 1)] =
           localSol[localSize[2] * sizeY * sizeX + j * sizeX + k];
    }
  }
  MPI_Sendrecv(send.data(), localSize[0]*localSize[1], MPI_DOUBLE,
               rank_next,0,recv.data(),localSize[0]*localSize[1],
               MPI_DOUBLE,rank_prev,0,commCart,MPI_STATUS_IGNORE);
  for(int j = 1; j <= localSize[1]; j++){
    for(int k = 1; k <= localSize[0]; k++){
      localSol[0 * sizeY * sizeX + j * sizeX + k] =
          recv[(j - 1) * localSize[0] + (k - 1)];
    }
  }
}
```

The complete source code can be found on the GitHub repository HeatEquationSolver by francescomicucci (`https://github.com/francescomicucci/HeatEquationSolver/tree/main/`). Look for the file `pipeline.cpp` located in the subfolder 'Pipeline'.

The outcomes of this parallel implementation are summarized in Table 5.6. As expected, the computational time decreases as the number of processors increases. This demonstrates the effectiveness of the pipeline method in parallelizing the solution of the 3D time-dependent heat equation, although the enhancement isn't as remarkable as we would have expected.

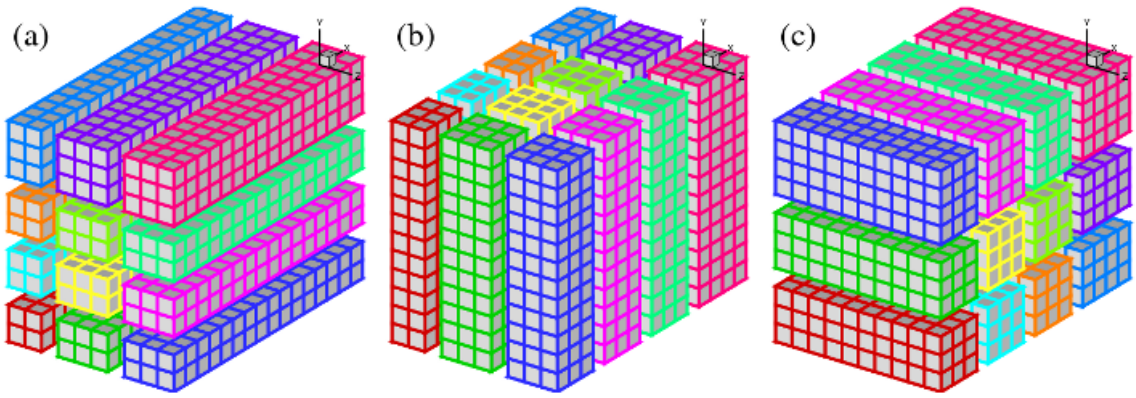
5.5.2 2D Pencil Decomposition

For our next parallelization strategy, our goal is to dynamically distribute data across processors to allow for the application of serial algorithms within local memory without any interruptions.
To implement this idea, we will employ the 2D Pencil Decomposition technique.

Number of processors	Computational time (s)
1	7.71727
2	5.66556
3	5.08347
4	4.27546

Table 5.6: Results for the 3D time-dependent heat equation using the Pipeline technique given $n = 100$ and $n_t = 100$.

This method involves splitting our 3D domain into a grid-like arrangement of elongated subdomains, referred to as 'pencils'. The orientation of these pencils dictates the type of decomposition, and in our case, we'll opt for decompositions along the axes x, y, and z. The illustration below provides a visual representation of the decomposition along these axes:



Applying this method to our problem involves a step-by-step procedure.
Initially, we segment the domain into pencils along the z axis. Each pencil is then assigned to a different processor, which utilizes the Thomas algorithm to solve the problem along the z direction. Once this phase is completed, we proceed to decompose the domain along the y axis. We then redistribute the data among processors based on this new segmentation and tackle the problem along this direction. Afterward, we replicate this procedure for the x axis.
Following these computations, we return to the initial decomposition where each processor shares its results at the boundaries with neighboring processors, as these are needed for calculations at the next time step.

For the implementation, we chose to use the Fortran language. Specifically, we utilized the `MPI` library for communication between processors and the `2decomp_fft` library for implementing the 2D Pencil Decomposition and data redistribution.
In terms of changes from the sequential implementation, the following additions/modifications were implemented:

- Initialization of `MPI` and `2decomp_fft` at the beginning.

```
CALL MPI_INIT(ierror)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size_Of_Cluster, ierror)
CALL MPI_Comm_rank(MPI_COMM_WORLD, process_Rank, ierror)

!Count number of processors per side
P_col=FLOOR(SQRT(dble(size_Of_Cluster)))
```

```
DO WHILE (MOD(size_Of_Cluster,P_col) .NE. 0)
    P_col=P_col-1
END DO
P_row=size_Of_Cluster/P_col

CALL decomp_2d_init(nx,nx,nx,P_row,P_col)
```

- Determination of the coordinates of the current processor and its neighboring processors for a 2D pencil decomposition along the z-axis.

```
!Calculate coordinate of the processor in the cart
CALL MPI_CART_COORDS(DECOMP_2D_COMM_CART_Z,process_Rank,2,        &
coords_proc,ierror)
!Find processor east and west
CALL MPI_CART_SHIFT(DECOMP_2D_COMM_CART_Z, 0, 1, source_proc_ew,  &
dest_proc_ew, ierror)
!Find processor north and south
CALL MPI_CART_SHIFT(DECOMP_2D_COMM_CART_Z, 1, 1, source_proc_ns,  &
dest_proc_ns, ierror)
```

- Calculation of the right-hand side for the current processor, considering boundary values with adjacent processors.

```
DO k = zstart(3), zend(3)
    DO j = zstart(2),zend(2)
        DO i = zstart(1),zend(1)
            dz(i,j,k) = dt * forcing_term(i,j,k,timestep + dt/2)  &
                      + dt_dx2*(-6 * told(i,j,k))
            IF(i .NE. 1) THEN
            IF(i .NE. zstart(1)) THEN
                dz(i,j,k) = dz(i,j,k) + dt_dx2*(told(i-1,j,k))
            ELSE
                dz(i,j,k) = dz(i,j,k) + dt_dx2*(tolde(j,k))
            END IF
            ELSE
            dz(i,j,k) = dz(i,j,k) + dt_dx2*(real_solution(i-1,j,k)&
                      *sin(timestep))
            END IF
            IF(i .NE. nx) THEN
            IF(i .NE. zend(1)) THEN
                dz(i,j,k) = dz(i,j,k) + dt_dx2*(told(i+1,j,k))
            ELSE
                dz(i,j,k) = dz(i,j,k) + dt_dx2*(toldw(j,k))
            END IF
            ELSE
            dz(i,j,k) = dz(i,j,k) + dt_dx2*(real_solution(i+1,j,k)&
                      *sin(timestep))
            END IF
            IF (j .NE. 1) THEN
            IF(j .NE. zstart(2)) THEN
                dz(i,j,k) = dz(i,j,k) + dt_dx2*(told(i,j-1,k))
            ELSE
                dz(i,j,k) = dz(i,j,k) + dt_dx2*(toldn(i,k))
            END IF
            ELSE
            dz(i,j,k) = dz(i,j,k) + dt_dx2*(real_solution(i,j-1,k)&
                      *sin(timestep))
            END IF
            IF (j .NE. nx) THEN
            IF(j .NE. zend(2)) THEN
                dz(i,j,k) = dz(i,j,k) + dt_dx2*(told(i,j+1,k))
```

```
            ELSE
                dz(i,j,k) = dz(i,j,k) + dt_dx2*(tolds(i,k))
            END IF
            ELSE
            dz(i,j,k) = dz(i,j,k) + dt_dx2*(real_solution(i,j+1,k)&
                      *sin(timestep))
            END IF
            IF (k .NE. 1) THEN
            dz(i,j,k) = dz(i,j,k) + dt_dx2*(told(i,j,k-1))
            ELSE
            dz(i,j,k) = dz(i,j,k) + dt_dx2*(real_solution(i,j,k-1)&
                      *sin(timestep))
            END IF
            IF (k .NE. nx) THEN
            dz(i,j,k) = dz(i,j,k) + dt_dx2*(told(i,j,k+1))
            ELSE
            dz(i,j,k) = dz(i,j,k) + dt_dx2*(real_solution(i,j,k+1)&
                      *sin(timestep))
            END IF
        END DO
    END DO
END DO
```

- Computation of the solution of the systems along the three axes and redistribution of data after solving each system.

```
!THOMAS ON Z
CALL threed_thomas_z(a,b,c,dz)
CALL transpose_z_to_y(dz, dy)

!THOMAS ON Y
CALL threed_thomas_y(a,b,c,dy)
CALL transpose_y_to_x(dy, dx)

! THOMAS ON X
CALL threed_thomas_x(a,b,c,dx)
CALL transpose_x_to_y(dx, dy)
CALL transpose_y_to_z(dy, dz)
```

- Exchange of data between processors to share boundary results.

```
! Send to west, receive from east
IF (dest_proc_ew >= 0) THEN
    btoldew = told(zend(1),zstart(2):zend(2),zstart(3):zend(3))
    CALL MPI_SEND(btoldew, size(btoldew,1)*size(btoldew,2),RTYPE, &
    dest_proc_ew, dest_proc_ew, DECOMP_2D_COMM_CART_Z,ierror)
END IF
IF (source_proc_ew >= 0) THEN
    CALL MPI_RECV(btoldew, size(btoldew,1)*size(btoldew,2),RTYPE, &
    source_proc_ew, process_Rank, DECOMP_2D_COMM_CART_Z,status,   &
    ierror)
    tolde = btoldew
END IF

! Send to east, receive from west
IF (source_proc_ew >= 0) THEN
    btoldew = told(zstart(1),zstart(2):zend(2),zstart(3):zend(3))
    CALL MPI_SEND(btoldew, size(btoldew,1)*size(btoldew,2),RTYPE, &
    source_proc_ew, source_proc_ew, DECOMP_2D_COMM_CART_Z,ierror)
END IF
IF (dest_proc_ew >= 0) THEN
```

```
    CALL MPI_RECV(btoldew, size(btoldew,1)*size(btoldew,2),RTYPE, &
    dest_proc_ew, process_Rank, DECOMP_2D_COMM_CART_Z,status,     &
    ierror)
    toldw = btoldew
END IF

! Send to south, receive from north
IF (dest_proc_ns >= 0) THEN
    btoldns = told(zstart(1):zend(1),zend(2),zstart(3):zend(3))
    CALL MPI_SEND(btoldns, size(btoldns,1)*size(btoldns,2),RTYPE, &
    dest_proc_ns, dest_proc_ns, DECOMP_2D_COMM_CART_Z,ierror)
END IF
IF (source_proc_ns >= 0) THEN
    CALL MPI_RECV(btoldns, size(btoldns,1)*size(btoldns,2),RTYPE, &
    source_proc_ns, process_Rank, DECOMP_2D_COMM_CART_Z,status,   &
    ierror)
    toldn = btoldns
END IF

! Send to north, receive from south
IF (source_proc_ns >= 0) THEN
    btoldns = told(zstart(1):zend(1),zstart(2),zstart(3):zend(3))
    CALL MPI_SEND(btoldns, size(btoldns,1)*size(btoldns,2),RTYPE, &
    source_proc_ns, source_proc_ns, DECOMP_2D_COMM_CART_Z,ierror)
END IF
IF (dest_proc_ns >= 0) THEN
    CALL MPI_RECV(btoldns, size(btoldns,1)*size(btoldns,2),RTYPE, &
    dest_proc_ns, process_Rank, DECOMP_2D_COMM_CART_Z,status,     &
    ierror)
    tolds = btoldns
END IF
```

The complete source code can be found on the GitHub repository HeatEquationSolver by francescomicucci (`https://github.com/francescomicucci/HeatEquationSolver/tree/main/`). Look for the file named `TimeDepHeat3D_parallel.f90` located in the subfolder '2DPencilDecomposition'.

The results of this parallel implementation are summarized in Table 5.7 and as expected, the computational time decreases as the number of processors increases.
Furthermore, there is a significant improvement in the computational time when moving from 1 to 2 processors, while the improvement is less significant when moving from 2 to 4 processors. This might happen because the amount of data exchanged between processors increases as they increase in number.

Number of processors	Computational time (s)
1	9.26
2	6.41
4	5.44

Table 5.7: Results for the 3D time-dependent heat equation using 2D Pencil Decomposition given $n = 100$ and $n_t = 100$.

5.5.3 Schur Complement

In tackling the Time-dependent Heat equation, our third approach involved leveraging the Schur Complement. In particular, we would like to manipulate the system of equations to distribute the calculations for different unknowns across different processors.
We will first concentrate on parallelization using two processors. However, it's worth noting that the same approach applies to any number of processors n.

We start with a linear system having the following form:

$$\begin{bmatrix} b_1 & c_1 & & & & & \\ a_2 & \ddots & \ddots & & & & \\ & \ddots & b_i & c_i & & & \\ & & a_{i+1} & b_{i+1} & c_{i+1} & & \\ & & & a_{i+2} & b_{i+2} & \ddots & \\ & & & & \ddots & \ddots & c_{n-1} \\ & & & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_i \\ x_{i+1} \\ x_{i+2} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} f_1 \\ \vdots \\ f_i \\ f_{i+1} \\ f_{i+2} \\ \vdots \\ f_n \end{bmatrix}$$

Here, we want to highlight the central row and column. This splits the matrix into two blocks - one in the top left and one in the bottom right - both of which are tridiagonal.

$$\left[\begin{array}{ccc|c|ccc} b_1 & c_1 & & & & & \\ a_2 & \ddots & \ddots & & & & \\ & \ddots & b_i & c_i & & & \\ \hline & & a_{i+1} & b_{i+1} & c_{i+1} & & \\ \hline & & & a_{i+2} & b_{i+2} & \ddots & \\ & & & & \ddots & \ddots & c_{n-1} \\ & & & & & a_n & b_n \end{array}\right]$$

In the scenario where we plan to use n processors, we'll select rows and columns to form n tridiagonal blocks, each of approximately equal size.
Next, we'll proceed by permuting the highlighted columns to the right and the highlighted rows downwards (row permutations will lead to permutations in the vector of unknowns and the vector of known terms) obtaining the following situation:

$$\left[\begin{array}{ccc|ccc|c} b_1 & c_1 & & & & & \\ a_2 & \ddots & \ddots & & & & \\ & \ddots & \ddots & & & & c_i \\ \hline & & & b_{i+2} & c_{i+2} & & a_{i+2} \\ & & & a_{i+3} & \ddots & \ddots & \\ & & & & \ddots & b_n & \\ \hline & & a_{i+1} & c_{i+1} & & & b_{i+1} \end{array}\right] \begin{bmatrix} x_1 \\ \vdots \\ x_i \\ \hline x_{i+2} \\ \vdots \\ x_n \\ \hline x_{i+1} \end{bmatrix} = \begin{bmatrix} f_1 \\ \vdots \\ f_i \\ \hline f_{i+2} \\ \vdots \\ f_n \\ \hline f_{i+1} \end{bmatrix}$$

We can represent this new system more compactly as follows:

$$\begin{bmatrix} A_1 & 0 & D_1 \\ 0 & A_2 & D_2 \\ E_1 & E_2 & F \end{bmatrix} \begin{bmatrix} \mathbf{x_1} \\ \mathbf{x_2} \\ \mathbf{x_I} \end{bmatrix} = \begin{bmatrix} \mathbf{f_1} \\ \mathbf{f_2} \\ \mathbf{f_I} \end{bmatrix}$$

By transforming the system matrix into a block-upper triangular form, we arrive at the subsequent linear system:

$$\begin{bmatrix} A_1 & 0 & D_1 \\ 0 & A_2 & D_2 \\ 0 & 0 & S \end{bmatrix} \begin{bmatrix} \mathbf{x_1} \\ \mathbf{x_2} \\ \mathbf{x_I} \end{bmatrix} = \begin{bmatrix} \mathbf{f_1} \\ \mathbf{f_2} \\ \tilde{\mathbf{f}}_\mathbf{I} \end{bmatrix}$$

where:

$$S = F - E_1 A_1^{-1} D_1 - E_2 A_2^{-1} D_2$$
$$\tilde{\mathbf{f}}_\mathbf{I} = \mathbf{f_I} - E_1 A_1^{-1} \mathbf{f_1} - E_2 A_2^{-1} \mathbf{f_2}$$

In the general case for any number of processors n, the system becomes:

$$\begin{bmatrix} A_1 & 0 & \cdots & 0 & D_1 \\ 0 & A_2 & \ddots & \vdots & D_2 \\ \vdots & \ddots & \ddots & 0 & \vdots \\ \vdots & & \ddots & A_n & D_n \\ 0 & \cdots & \cdots & 0 & S \end{bmatrix} \begin{bmatrix} \mathbf{x_1} \\ \mathbf{x_2} \\ \vdots \\ \mathbf{x_n} \\ \mathbf{x_I} \end{bmatrix} = \begin{bmatrix} \mathbf{f_1} \\ \mathbf{f_2} \\ \vdots \\ \mathbf{f_n} \\ \tilde{\mathbf{f}}_\mathbf{I} \end{bmatrix}$$

where:

$$S = F - \sum_{i=1}^{n} E_i A_i^{-1} D_i$$

$$\tilde{\mathbf{f}}_\mathbf{I} = \mathbf{f_I} - \sum_{i=1}^{n} E_i A_i^{-1} \mathbf{f_i}$$

Finally, the steps to solve the system are:

1. Solve the system $S\mathbf{x_I} = \tilde{\mathbf{f}}_\mathbf{I}$.
2. Simultaneously solve the systems $A_i \mathbf{x_i} = \mathbf{f_i} - D_i \mathbf{x_I}$ for $i = 1, 2, ..., n$ on n processors.

Other tasks that can be carried out simultaneously, besides computing $\mathbf{x_i}$, include:

- Computing $E_i A_i^{-1} D_i$ for i = 1, ..., n to form the matrix S.
- Computing $E_i A_i^{-1} \mathbf{f_i}$ for i = 1, ..., n to create the vector $\tilde{\mathbf{f}}_\mathbf{I}$.

By dividing these tasks among multiple processors, we can speed up solving the heat equation.

It appears that the most time-consuming part is manipulating the linear system. Nevertheless, there are optimizations we can employ to speed up this process which leverage the structure of the matrices D_i and E_i.
Both D_i and E_i (for $i = 1, ...,$ n) are sparse matrices. These matrices have specific dimensions: m rows by n columns for D_i and n rows by m columns for E_i. Here, n represents the number of processors we're using, while m depends on the size of the matrix A_i.

Both matrices just have two non-zero elements, located in specific positions that will be highlighted in red in the following illustration:

$$D_i = \begin{bmatrix} 0 & \dots & 0 & (1, i-1) & 0 & 0 & \dots & 0 \\ \vdots & & \vdots & 0 & \vdots & \vdots & & \vdots \\ \vdots & & \vdots & \vdots & 0 & \vdots & & \vdots \\ 0 & \dots & 0 & 0 & (m, i) & 0 & \dots & 0 \end{bmatrix}, \quad E_i = \begin{bmatrix} 0 & \dots & \dots & 0 \\ \vdots & & & \vdots \\ 0 & \dots & \dots & 0 \\ (i-1, 1) & 0 & \dots & 0 \\ 0 & \dots & 0 & (i, m) \\ 0 & \dots & \dots & 0 \\ \vdots & & & \vdots \\ 0 & \dots & \dots & 0 \end{bmatrix}$$

Given the particular structure of the matrix D_i, we deduce that the only non-zero elements of the matrix $A_i^{-1}D_i$ will be in the i-th column and the $(i-1)$-th column. This is significant because it implies that we can compute this matrix by solving only the following linear systems:

$$A_i\mathbf{x} = \mathbf{d_i} \quad \text{and} \quad A_i\mathbf{y} = \mathbf{d_{i-1}}$$

where $\mathbf{d_i}$ and $\mathbf{d_{i-1}}$ are the i-th and $(i-1)$-th columns of the matrix D_i, respectively. The two vectors $\mathbf{d_i}$ and $\mathbf{d_{i-1}}$ have the following structure:

$$\mathbf{d_i} = \begin{bmatrix} a_l \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \mathbf{d_{i-1}} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ c_k \end{bmatrix}$$

Furthermore, recognizing that the matrix A_i is tridiagonal and that the two vectors of known terms, $\mathbf{d_i}$ and $\mathbf{d_{i-1}}$, have only one non-zero element each, we can implement a tailored version of the Thomas method to solve the two systems.
Such version of the Thomas method has been implemented in C++ as follows:

```
void SchurSolver::schurSubsystemsSolver(double x[], double y[],
                                        double diag, double upperDiag,
                                        double lowerDiag, int dim)
{
    double w;
    double* b = new double[dim];
    double* d = new double[dim];

    std::fill_n(b, dim, diag);
    d[0] = lowerDiag;

    for(int i = 1; i < dim; i++){
        w = lowerDiag / b[i - 1];
        b[i] = b[i] - w * upperDiag;
        d[i] = d[i] - w * d[i-1];
    }
    x[dim - 1] = d[dim - 1] / b[dim - 1];
    y[dim - 1] = upperDiag / b[dim - 1];
    for(int i = dim - 2; i >= 0; i--){
        x[i] = (d[i] - upperDiag * x[i + 1]) / b[i];
        y[i] = (- upperDiag * y[i + 1]) / b[i];
    }
}
```

Finally, using the specific structure of matrix E_i, we can state that $E_i A_i^{-1} D_i$ will have only 4 non-zero elements in the following positions: $(i-1, i-1)$, $(i-1, i)$, $(i, i-1)$ and (i, i).

$$E_i A_i^{-1} D_i = \begin{bmatrix} 0 & \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & \dots & 0 & (i-1, i-1) & (i-1, i) & 0 & \dots & 0 \\ 0 & \dots & 0 & (i, i-1) & (i, i) & 0 & \dots & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots & \dots & 0 \end{bmatrix}$$

Knowing the positions of these non-zero elements enables us to streamline the computation, requiring fewer multiplications than in a standard matrix-by-matrix multiplication.

A further optimization consists of reusing previously computed results. Specifically, we can store the outcomes of $A_i^{-1} D_i$ and $A_i^{-1} \mathbf{f_i}$ used to calculate S and $\tilde{\mathbf{f}}_\mathbf{I}$ to avoid redundant calculations in determining the solution. In fact:

$$A_i \mathbf{x_i} = \mathbf{f_i} - D_i \mathbf{x_I} \quad \Rightarrow \quad \mathbf{x_i} = A_i^{-1} \mathbf{f_i} - A_i^{-1} D_i \mathbf{x_I}$$

All those optimizations were implemented in C++ for the 1D Time independent setting and the source code can be found on the GitHub repository HeatEquationSolver by francescomicucci (`https://github.com/francescomicucci/HeatEquationSolver/tree/main/`). Look into the folder 'SchurSolver'.