Can You Parallelize Time?

Moving Through the Complexity Time

The Bachelor End Project Thesis by

Yara Lottering

Delft University of Technology

TU Delft

Can You Parallelize Time?

Moving Through the Complexity Time

by

Yara Lottering

Student Name	Student Number
Yara Lottering	5120837

Supervisor:	Prof. Dr. Ir. Martin B. van Gijzen
Graduation Committee:	Prof. Dr. Ir. M.B van Gijzen & Dr. Ir. A.R.P.J Vijn
Institution:	Delft University of Technology
Place:	Faculty of Electrical Engineering, **Mathematics** and Computer Science, Delft
Thesis Completion:	1st of July 2022

`Link` to Cover Image

TU Delft

Preface

This research is about creating parallel variants of algorithms that solve time-dependent partial differential equations iteratively. The main focus will lie on how these algorithms perform in time and which ones have the best potential of obtaining the fastest computation time.

In the search of finding a suitable bachelor thesis, I found myself looking particularly into the theses of the Numerical Mathematics department. As I did my 3rd year elective on Numerical Methods as well, I found this category more than interesting. My privilege went out to this thesis in particular since computing in time is a complex phenomenon and it intrigued me. As I have learned from my Applied Mathematics studies, time is such an extraordinary happening. And obviously, one of the first questions that popped into my mind was that parallel in time could possibly mean there exists a parallel universe. Till this point, unfortunately, I am still not sure.

During this research, difficulties came to the surface quite fast. As most of these issues were caused by Matlab, the programming language I coded in, creating good content for my research was a challenge. However, by predicting results and looking more into the methods without the parallelisation, some very strong conclusions could still be made.

In advance, I would like to thank my supervisor Prof. Dr. Ir. Martin van Gijzen, the highly respected director of the Applied Mathematics Master at the TU Delft, for assisting me through these eleven weeks. As more and more became clear every meeting, I would definitely say that Martin has helped me grow in this specific part of mathematics. Not only has my abstract thinking been tested, also my programming skills are freshly renewed thanks to my supervisor.

Yara Lottering
Delft, July 2022

Summary

This Bachelor End Project Thesis is about creating parallel variants of time-dependent methods. To achieve this, the time-dependent problem is formulated as a large system of linear equations that is solved iteratively. This linear system is determined from the time-dependent linear heat equation in 1D that all the analysis is about. The experiments are tested for different boundary condition combinations. Namely, for a mixture of Dirichlet, Neumann and Robin conditions. Also, three different time integration methods are used, namely Forward Euler, Backward Euler and the θ-Rule for $\theta = 0.2, 0.4, 0.6$ and $\theta = 0.8$. Since time is continuous and moves in the positive direction, time integration methods need the previous solution in order to find the current solution. Executing a calculation in parallel time is therefore a complex and tricky phenomenon.

This research considers two different iterative methods, the block Jacobi method and the block Gauss-Seidel method. Both have a lot in common but the main difference lies in the use of different parts of the matrix. Where the Jacobi method uses just the diagonal in order to approximate the inverse, the Gauss-Seidel method also uses the upper or lower triangular part of the matrix. Therefore, creating a parallel variant for the block Jacobi method is easily possible, but for the block Gauss-Seidel method it is not. In order to get to a valid conclusion, the stability of each time integration method is considered.

The block Jacobi method converges slowly to the solution since it uses the diagonal for every step instead of the whole matrix. Therefore, in this research, combinations of the block Gauss-Seidel method have been created and tested. Three different variants are discussed. The first one is the step-wise combination method which uses a block Jacobi iteration for every step size chosen, and uses the sequential Gauss-Seidel time steps for all the other iterations. The second variant is called the Red-Black Ordering and is known for its odd and even iterations. Here every odd iteration is done in parallel and similar for its even iterations. Since the equation only needs its previous solution, this method would work very well in parallel. The last variant uses the number of processors to manipulate the maximum number of iterations. Here the method converges earlier or at the number of processors chosen.

What can be concluded from the results is that the way a method converges all depends on how many Jacobi iterations take place during the computation. The Gauss-Seidel calculation only takes one iteration, so the time until the solution converges all depends on how many Jacobi iterations the method computes. Having a computer with more processors than number of Jacobi iterations performed, will also speed-up the computation. Our predictions show in particular that with only implicit time integration methods the speed-up can really be achieved.

For the block Jacobi method, this equals that the maximum number of iterations is also the number of time steps nt. The Gauss-Seidel variants are all more efficient, since less Jacobi iterations are used. For variant 1, the step-wise combination method, the maximum number of iterations equals $\frac{nt}{\text{step}}$. For the Red-Black Ordering method, every other iteration, a Jacobi step is used so the maximum number of iterations equals $\frac{nt}{2}$. The last variant, the processor-wise combination method, is a special variant of the first variant. This method has the characteristic that the maximum number of iterations equals the chosen number of processors, np and therefore has, in terms of the first variant, a 'step'-size of $\frac{nt}{np}$.

This project has been conducted with Matlab which did not fully cooperate when using the parallel time function. In order to really test, and therefore not predict, the obvious way to perform a parallel computation is to implement this program in another programming language and later on test it on the DelftBlue supercomputer. The predictions should still hold but the results will be more clearly visualised.

Contents

Preface i

Summary ii

Nomenclature iv

1 Introduction 1

2 Preliminaries 2

3 The Heat Equation 4
3.1 Semi-discretization . . . 4
3.1.1 Implementation of the Boundary Conditions . . . 5
3.1.2 The Eigenvalues of the Matrix A . . . 6
3.2 Time Integration . . . 7
3.2.1 Stability of Time Integration Methods . . . 7
3.3 All at Once System . . . 9

4 Computing Solutions with Iterative Methods 10
4.1 General . . . 10
4.2 Jacobi . . . 11
4.3 Gauss-Seidel . . . 11
4.3.1 Method 1: Step-wise . . . 12
4.3.2 Method 2: Red-Black Ordering . . . 12
4.3.3 Method 3: Processor-wise . . . 12

5 Predicting Behavioural Results 13
5.1 Goal . . . 13
5.1.1 Heat Equation Testing . . . 13
5.2 Hypothesis . . . 13
5.3 Testing on Small Problems . . . 16
5.4 Testing on Realistic Problems . . . 18
5.5 Remarks on Parallelisation . . . 22

6 Conclusion 23

7 Future Work 25

References 26

A Matlab Code Initialisation 27

B Matlab Code Methods 30

C Matlab Code Heat Equation Test 34

D Matlab Code Transformed Matrix 35

E From Sequential to Parallel 36

Nomenclature

Abbreviations

Abbreviation	Definition
D	Dirichlet
N	Neumann
R	Robin
No.	Number of
PDE	Partial Differential Equation

Symbols

Symbol	Definition
nx	No. gridpoints
nt	No. time steps
np	No. processors
L	Length of grid
T	Time range
I	The identity matrix
α	Heat transfer coefficient
λ	Eigenvalue
θ	Time integration variable

1 Introduction

Living in the 21st century, technology developments proceed very rapidly. Where at the start of this century, a touch-screen smartphone was nowhere to be found, 22 years later rumours about flying cars go through the hallways. People working in the IT, Information Technology, try to come up with faster, newer and more efficient solutions to problems every day. Usually, these solutions seem to be impossible at the start but always look to be created after all. Those who have been working on high-quality computers might have heard about parallel computing. This phenomenon has been an interesting topic for many engineers. However, parallel computing is not that easy to understand and even more difficult to actually perform. This report will therefore focus on a specific, nevertheless important, part of parallel computing, namely for solving partial differential equations.

Numerically solving time-dependent partial differential equations is normally done with the method of lines. This method shows a clear structure where initially the equation is discretized in space, called semi-discretization, and the semi-discretized equation is afterwards integrated in time with a time-integration method that suits best for that specific equation. The problem that arises during these numerical calculations is the time it takes to give a **valid** solution. This issue mostly occurs for large problems with big matrices. However, these are the exact problems one wishes to solve. Solving these problems can only be done in two steps as explained above. The need for a so-called supercomputer is crucial for huge problems. Because of the large amount of processors these supercomputers possess, the equation can be solved very fast in parallel time, which is what this report is mainly about.

During this Bachelor End Project, the focus will lie on finding algorithms to solve time-dependent partial differential equations that can be executed parallel in time. This implies distributing the calculations among x different 'processors' such that they can work in parallel among each other and the solving becomes x-times faster. The tricky part, however, is time itself. Known to the reader is that time is continuous and moves only in the positive direction ($t > 0$), meaning that it can not be discretely separated from one another. Also, time integration methods require the value of previous time-steps in order to create the next one. With this knowledge, the actual parallelization might even be impossible. The goal of this project is to find iterative methods that could solve PDE's, to discover if parallelization is even possible at all (by parallelizing the iterative methods), and if so, to integrate algorithms to solve these PDE's in parallel time.

Concerning the structure of this research, first background material will be revealed in Chapter 2. Afterwards the initialisation of the heat equation will be explained. Here the boundary of the heat rod, with their physical aspect and different time integration methods are defined. This will be done in Chapter 3. Before actually diving into the behavioural results, a thorough definition and description will be given on the different methods used to do parallel computing with. After the analysis of these methods in Chapter 4, the actual results are being given in Chapter 5. This report is ended with a full conclusion followed by discussing future work and possibilities.

2

Preliminaries

To understand the most out of the decisions made in this research, the following background material is necessary to know. In this chapter, a couple of definitions are reviewed that are used in the remainder of this thesis.

Matrix

Every matrix must be a diagonally dominant diagonalizable L-matrix in order to be used for the method of lines, which is used in the next chapter and to be transformed in a diagonal matrix, which is used further in the research. Let A be a matrix consisting of elements a_{ij} where i corresponds to the row which the element is in and similar with the column for j.

Definition.
A is an L-matrix if

(a) $a_{ij} \leq 0$, for all i, j with $i \neq j$
(b) $a_{ii} > 0$, for all i

Definition.
A is diagonally dominant if

$$|a_{ii}| \geq \sum_{\substack{j=1 \\ j \neq i}}^{N} |a_{ij}|, \quad i = 1, 2, \ldots, N$$

Definition.
A is irreducible if one can reach every row from another row in a finite amount of steps. In that case, the discrete maximum principle holds.

Definition.
The spectral radius of a matrix A equals the largest absolute eigenvalue in the spectrum of A, defined to be

$$\rho(A) = \{|\lambda| : \lambda \in \sigma(A)\}$$

Definition.
A matrix A is diagonalizable if there exists an invertible matrix S and a diagonal matrix D such that $S^{-1}AS = D$ or $A = SDS^{-1}$.
A special version of diagonalisation of a matrix is a Jordan normal decomposition where there exists a matrix J such that $A = SJS^{-1}$. This Jordan matrix has a special form, for $n = 3$, this matrix is printed below for illustration purposes.

$$J = \begin{pmatrix} \lambda_1 & 1 & 0 \\ 0 & \lambda_2 & 1 \\ 0 & 0 & \lambda_3 \end{pmatrix}$$

Heat Equation

The heat equation is a parabolic equation. Also, all the derivatives, first and second order, have been discretized using central differences in order to keep the error as low as possible.

Definition.
Parabolic equations describe transient solutions that evolve to a steady state. Solutions depend on the past, but not on the future.

Definition.
Central difference for first order derivatives equals

$$f'(x) = \frac{f_{n-1} - f_{n+1}}{2\Delta x} + O(\Delta x^2)$$

3

The Heat Equation

For the research of this Bachelor End Project, the whole analysis will be done using the linear heat equation in one dimension. The heat equation illustrates how heat flows in a rod assuming that heat only moves in the horizontal direction. It is a time-dependent equation for a process that tends to an equilibrium or stationary solution, also called parabolic. The heat equation also describes general diffusion problems. To start the process of numerically solving this equation in parallel time, one must perform the method of lines. This method is a two-step procedure where first semi-discretization takes place and after that the time integration.

3.1. Semi-discretization

Given is the linear heat equation in its simplest form

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \quad x \in (0, L), t \in (0, T] \tag{3.1}$$

with $u = 0$ on the boundary and $u(x, 0) = f(x)$. For simplicity, $f(x) = 0$ and an equidistant grid is used with grid-size $\Delta x = \frac{1}{nx+1}$.
This equation can easily be space-discretized by using finite central differences such that

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} + O(\Delta x^2)$$

with $u_0, u_N = 0$ on the boundaries. In the next few subsections of this chapter, more complex boundary conditions and the impact of them on the system will be explained.

Replacing the continuous derivatives by difference quotients yields for every grid point a linear ordinary differential equation. These equations form a system such that

$$\dot{u} = Au \tag{3.2}$$

Before time integration methods can be applied, the actual matrix A in Equation 3.2 must be determined. This matrix is a $nx \times nx$, diagonally dominant irreducible L-matrix and looks like

$$A = \frac{1}{\Delta x^2} \begin{pmatrix} -2 & 1 & 0 & & 0 \\ 1 & -2 & 1 & \dots & 0 \\ 0 & 1 & -2 & & 0 \\ & \vdots & & \ddots & \\ 0 & 0 & 0 & 1 & -2 \end{pmatrix}$$

3.1.1. Implementation of the Boundary Conditions

In the numerical mathematics as it is to be known at this point, three different boundary conditions are defined on a rod of length L:

1. Dirichlet $\rightarrow u(0) = a, u(L) = b$
2. Neumann $\rightarrow \frac{du}{dn}(0) = c, \frac{du}{dn}(L) = d$
3. Robin $\rightarrow \frac{du}{dn}(0) + \alpha u(0) = e, \frac{du}{dn}(L) + \alpha u(L) = f$

Given a 1D rod as for the linear heat equation, both boundaries can have one of the boundary conditions named above. Given these boundary conditions, the matrix A but also vector **b** can change in value. In the following paragraph will be explained in what way the system is changing. Afterwards, a more physical perspective of the boundary conditions will be given.

The difference in matrix A is determined by discretizing the boundary conditions and filling them in into the equation. The impact of the different boundary conditions will now be specified.

For Dirichlet conditions, the matrix A does not change. However, the vector **b** does, in a very simple way. Considering block matrix 3.4, which will become visible in a few paragraphs, the vector u_1, also known as the starting vector, will look like

$$u_1 = \frac{1}{\Delta x^2} \begin{pmatrix} a \\ 0 \\ \vdots \\ 0 \\ b \end{pmatrix}$$

By using central differences for the first derivatives, the matrix A will initially get different values. For the Neumann conditions, the matrix will result in

$$A = \frac{1}{\Delta x^2} \begin{pmatrix} -2 & 2 & 0 & & 0 \\ 1 & -2 & 1 & \dots & 0 \\ 0 & 1 & -2 & & 0 \\ & \vdots & & \ddots & \\ 0 & 0 & 0 & 2 & -2 \end{pmatrix}$$

Note the difference between this matrix and the matrix belonging to Equation 3.2, while especially focusing on the first and the last row. The influence of Neumann boundary conditions when using a central difference on the vector **b** is basically the same way as defined with the Dirichlet conditions, but now the value gets multiplied by 2. Visualising this gives

$$u_1 = \frac{1}{\Delta x^2} \begin{pmatrix} 2c\Delta x \\ 0 \\ \vdots \\ 0 \\ 2d\Delta x \end{pmatrix}$$

Lastly, the Robin conditions will be discussed. This is the most complex condition, both numerically and physically (which will be analysed in the next part). The matrix changes quite a bit in this case. Obviously, the changes are only made in the first and last row since that is where the boundary occurs. For $\alpha > 0$, the following matrix is created

$$A = \frac{1}{\Delta x^2} \begin{pmatrix} -(2 + 2\alpha\Delta x) & 2 & 0 & & 0 \\ 1 & -2 & 1 & \dots & 0 \\ 0 & 1 & -2 & & 0 \\ & \vdots & & \ddots & \\ 0 & 0 & 0 & 2 & -(2 + 2\alpha\Delta x) \end{pmatrix}$$

and the vector **b** is the same as for the Neumann boundary conditions, but for this specific case $c = e$ and $d = f$ of course. Note that when choosing $\alpha = 0$, this boundary condition all of a sudden turns out to be the same as the Neumann condition.

Physical Interpretation

The Dirichlet condition, also the easiest to implement as interpreted above, basically states that on that specific edge the temperature is constant and does not change. An example in daily life can be found in a refrigerator with infinitely large thermal conductivity.

For the Neumann boundary conditions, a bit more physics is applied. At the boundary, a constant heat flux is given. This means that the temperature changes constantly but with a steady speed. An important aspect of this boundary condition is that if the flux equals zero, the boundary actually represents the ideal heat insulator with the heat diffusion.

The last boundary condition that is discussed in the previous section is called the Robin boundary condition. This specific type of boundary condition is a combination, or rather called, a mix, of Dirichlet and Neumann boundary conditions. It describes the basics of Newton's law of cooling with a coefficient α representing the heat transfer. This coefficient is determined by two main properties of the material: the sharpness and the geometry between the two media. Most of the time this law is used to describe the boundary between metals and gas, as is also used in this research. For this linear heat equation, only conduction is used to transfer heat. However, heat can also be transferred by two other phenomena, namely convection or radiation. For simplicity and by staying within the scope of this research, only heat conduction will be considered.

3.1.2. The Eigenvalues of the Matrix A

In order to find a condition on the stability of the time step, one must first find a general condition on the eigenvalues of the matrix. For this condition, Gershgorin Circle Theorem is used and described below.

Let A be a $nx \times nx$ matrix, with entries a_{ij}. For $i \in \{1, \ldots, nx\}$ let R_i be the sum of the absolute values of the non-diagonal entries in the i-th row such that

$$R_i = \sum_{j \neq i} |a_{ij}|$$

Let $D(a_{ii}, R_i) \subseteq \mathbb{R}$ be a closed disc centered at a_{ii} with radius R_i. Such a disc is called a Gershgorin disc.

Theorem 1 *Every eigenvalue of A lies within at least one of the Gershgorin discs $D(a_{ii}, R_i)$.*

Proof. Let λ_j be an eigenvalue of A with corresponding eigenvector x_j. Let i be given in such a way that x_i is the largest element of the vector x. By Linear Algebra, one knows that $Ax = \lambda x$ for all eigenvalues. Now taking the i-th component of this matrix multiplication, the following is gained

$$\sum_j a_{ij} x_j = \lambda x_i$$

Rewriting and getting the diagonal element out, gives

$$\sum_{j \neq i} a_{ij} x_j = (\lambda - a_{ii}) x_i$$

Now by the Triangle Inequality and knowing the fact that $\frac{x_j}{x_i} \leq 1$ (since x_i was the absolute largest element) this results in

$$|\lambda - a_{ii}| = \left| \sum_{j \neq i} \frac{a_{ij} x_j}{x_i} \right| \leq \sum_{j \neq i} |a_{ij}| = R_i$$

And thus every eigenvalue of A lies within a Gershgorin disc.

□

Now that the theorem has been proved, it can be performed on the matrix that belongs to Equation 3.2.

Given the matrix A, one can immediately see that

$$\left|\lambda + \frac{2}{\Delta x^2}\right| \leq \frac{2}{\Delta x^2} \tag{3.3}$$

This implies that

$$-\frac{4}{\Delta x^2} \leq \lambda \leq 0$$

and the condition is found. This condition holds for Dirichlet boundary conditions and Neumann boundary conditions. A different calculation must be made for Robin boundary conditions.

$$\left|\lambda + \frac{2 + 2\alpha\Delta x}{\Delta x^2}\right| \leq \frac{2}{\Delta x^2}$$

This implies that

$$\frac{-4 - 2\alpha\Delta x}{\Delta x^2} \leq \lambda \leq -\frac{2\alpha}{\Delta x^2}$$

One can trivially see that when $\alpha = 0$, the condition is equal to the one mentioned in 3.3.

3.2. Time Integration

In the next section, different time integration methods will be discussed and analysed. To understand what time step works best for what specific method, the stability condition on the time step is proved. The stability condition is extremely important when calculating solutions for time-dependent equations. When there is no stability, the solution diverges and no analysis can be performed.

3.2.1. Stability of Time Integration Methods

To obtain the theoretical stability of the time integration methods, the amplification factor is used. This number must be smaller or equal to 1 in order for stability to occur.

Forward Euler

Let the semi-discretized equation look like

$$\dot{u} = Au$$

Then the Forward Euler method looks like

$$\frac{u^{n+1} - u^n}{\Delta t} = Au^n$$

Rewriting this equation will lead into

$$u^{n+1} = (I + \Delta tA)u^n$$

Also, the amplification factor Q can be found from this result and equals

$$Q(\Delta tA) = I + \Delta tA$$

To earn stability, the following inequality must hold for all eigenvalues λ_j for $j = 1, \ldots, m$:

$$|Q(\Delta t\lambda_j)| = |1 + \Delta t\lambda_j| \leq 1$$

For absolute stability, this equation holds as well, but then with a strict inequality. Simplifying this equation will give

$$\Delta t \leq -\frac{2}{\lambda_j}$$

for every eigenvalue λ_j. More generally will be

$$\Delta t \leq -\frac{2}{\min\{\lambda_j : \forall j\}}$$

Having found a stability condition that depends on λ is not enough to implement the system since λ is unknown. Therefore, one may use the Gershgorin Circle Theorem described at the beginning of this section. Together with the stability condition found per method, the total stability condition is formulated. Hence for the Forward Euler method the stability condition on Δt in terms of Δx, for $\alpha \geq 0$ equals

$$\Delta t \leq \frac{\Delta x^2}{2 + \alpha \Delta x}$$

Backward Euler

Let the semi-discretized equation look like

$$\dot{u} = Au$$

Then the Backward Euler method looks like

$$\frac{u^{n+1} - u^n}{\Delta t} = Au^{n+1}$$

Rewriting this equation will lead into

$$u^{n+1} = (I - \Delta t A)^{-1} u^n$$

Also, the amplification factor Q can be found from this result and equals

$$Q(\Delta t A) = (I - \Delta t A)^{-1}$$

To earn stability, the following inequality must hold for all eigenvalues λ_j for $j = 1, \ldots, m$:

$$|Q(\Delta t \lambda_j)| = \left| \frac{1}{(1 - \Delta t \lambda_j)} \right| \leq 1$$

This equation always hold for every $\Delta t > 0$ since $\lambda_j < 0$ for all $j = 1, \ldots, m$. This time integration method is then classified as unconditionally stable since every time step will work and it is not dependent of Δx.

θ-Rule

Let the semi-discretized equation look like

$$\dot{u} = Au$$

Then the so-called θ-Rule, is a weighted form of the combination of the Forward and Backward Euler methods. This method then looks like

$$\frac{u^{n+1} - u^n}{\Delta t} = A(1 - \theta)u^n + A\theta u^{n+1}$$

with $0 < \theta < 1$. Rewriting this equation will lead into

$$u^{n+1} = (I + \Delta t A(1 - \theta))(I - A\Delta t \theta)^{-1} u^n$$

Also, computing the amplification factor Q can trivially be done and equals

$$Q(\Delta t A) = (I + \Delta t A(1 - \theta))(I - A\Delta t \theta)^{-1}$$

To earn stability, the following inequality must hold for all eigenvalues λ_j for $j = 1, \ldots, m$:

$$|Q(\Delta t \lambda_j)| = \left| \frac{1 + \Delta t \lambda_j (1 - \theta)}{1 - \lambda_j \Delta t \theta} \right| \leq 1$$

For absolute stability, this equation holds as well, but then with a strict inequality. Trivially seen, is that for all $\frac{1}{2} \leq \theta < 1$ this method is unconditionally stable since $\lambda_j < 0$ thus for each $\Delta t > 0$ the condition holds. However, for $0 < \theta < \frac{1}{2}$ the stability condition equals

$$\Delta t \leq \frac{2}{\lambda_j(2\theta - 1)}$$

for every eigenvalue λ_j. More generally:

$\Delta t > 0$ for $\frac{1}{2} \leq \theta < 1$ and

$\Delta t \leq \frac{2}{\min\{\lambda_j : \forall j\}(2\theta - 1)}$ for $0 < \theta < \frac{1}{2}$

Then again, by using the Gershgorin Circle Theorem, the total stability condition of the time step Δt in terms of Δx for $\alpha \geq 0$ equals

$$\Delta t \leq \frac{\Delta x^2}{(-2 - \alpha\Delta x)(2\theta - 1)} \text{ for } 0 < \theta < \frac{1}{2}$$

3.3. All at Once System

Using the method of lines explained in the last two sections, the full discretization is being performed in two steps separate from each other. Following Wathen's idea described in his article on preconditioning for all at once PDE's, this system can also be executed all at once, which would work well for parallelisation. With this knowledge, the system can be split into different block matrices. Such a block matrix looks like

$$\mathcal{A}\mathbf{U} = \begin{pmatrix} A_0 & & & \\ A_1 & A_0 & & \\ & \ddots & \ddots & \\ & & A_1 & A_0 \end{pmatrix} \begin{pmatrix} \mathbf{u}^0 \\ \mathbf{u}^1 \\ \vdots \\ \mathbf{u}^n \end{pmatrix} = \begin{pmatrix} \mathbf{u}^0 \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{pmatrix} = \mathbf{b} \tag{3.4}$$

where the exact values of the elements of A_0 and A_1 are determined by the different time integration methods and by the space discretization. The value of the vector **b** is determined by the different boundary conditions. [4]

Forward Euler
For the Forward Euler method, the iteration equals

$$u^{n+1} = (I + \Delta t A)u^n$$

Now it is easily seen that the corresponding matrices from block matrix 3.4 are $A_0 = I$, the identity matrix, and $A_1 = -(I + \Delta t A)$.

Backward Euler
The Backward Euler method consists of an iteration that involves a fraction, namely

$$u^{n+1} = (I - \Delta t A)^{-1}u^n$$

Rewriting this equation such that the fraction disappears, shows again the values of the matrices that need to be implemented into block matrix 3.4. For this specific method, the matrices equal $A_0 = I - \Delta t A$ and $A_1 = -I$.

θ-Rule
Lastly, the implementation of the block matrix for the θ-Rule. By using the same method as for the previous time integration methods, with an iteration as follows

$$u^{n+1} = (I + \Delta t A(1 - \theta))(I - A\Delta t\theta)^{-1}u^n$$

the corresponding matrices for block matrix 3.4 equal $A_0 = I - \Delta t\theta A$ and $A_1 = -(I + \Delta t(1 - \theta)A)$.

Now that the full discretization of the system is fulfilled, one may use the methods discussed in Chapter 4 to solve the system iteratively.

4

Computing Solutions with Iterative Methods

An iterative solution method is a mathematical procedure that generates a sequence of approximations where the $i+1$-th approximation is derived from the i-th iteration. This process continues until the norm of the error, the difference between two solutions, reaches a certain number. This number, the tolerance, might be chosen randomly but usually depends on the time step dt in large problems.

4.1. General

For general iterative methods, a matrix A is split into two sub matrices, $A = M + N$. From $Ax = b$, the solution can then be found through iterating as follows

$$x^{n+1} = M^{-1}(b - Nx^k)$$

In the following two sections, specific variants of this iteration are clarified. The two sub matrices are chosen in two different ways whereas the convergence is also very different. The more the matrix M looks like the original matrix A, the faster the solution of the method converges. Therefore, if matrix $M = A$, the solution is to be done in one step since $N = 0$ then. This idea is moreover specified in the following theorem.

Theorem 2 *The iterative method converges if and only if,* $\rho(M^{-1}N) \leq 1$ *and absolute convergence is achieved when there is a strict inequality.*

In order to prove this theorem, another theorem must be considered and proven first. [5]

Theorem 3 $\rho(M^{-1}N) \leq 1$ *implies that*

$$\lim_{k\to\infty} ||A^k||_2 = 0$$

Proof. Assuming A to be diagonalizable, i.e., we assume that there exists a matrix P and a diagonal matrix D such that $A = PDP^{-1}$. The columns of P span the eigenspaces of A and the diagonal components of D are the eigenvalues of A. (In case that A is not diagonalizable an argument similar to the one that follows can be made using the Jordan decomposition of A). This implies that $A^k = PD^kP^{-1}$, which in turn implies that if $\rho(M^{-1}N) < 1$ we have that $\lim_{k\to\infty} ||A^k||_2 = 0$.

□

Now from this result, Theorem 2 is trivial since knowing that $\lim_{k\to\infty} ||A^k||_2 = 0$, one can immediately see that for taking k as large as nt the solution converges.

4.2. Jacobi

The Jacobi method was named after Carl Gustav Jacob Jacobi, a German mathematician who worked mostly in the fundamental part of the mathematics. He discovered that linear systems could also be solved by just looking at the iterations over the diagonal. His algorithm in the simplest form works as described below. [2]

Initially, the matrix A is split into three different matrices such that $A = D + L + U$ where D is the diagonal matrix, L is the strict lower-triangular matrix and U is the strict upper triangular matrix. [5] Then given a known starting solution x^0, the general method equals

$$x^{k+1} = D^{-1}(b - (L + U)x^k)$$

So comparing this to the general case, $M = D$ and $N = L+U$. For block matrices, the implementation of the block Jacobi method is defined in a slight different way [3]. Then by using the all at once method from Wathen, these Jacobi blocks then look like $D = A_0$ and $L = A_1$.

Since Jacobi basically calculates the current solution with the help of the diagonal matrix of the current row, calculating both the error and the solution can be done simultaneously. In other words, the current solution does not need the newly defined error to calculate its value. With this fact, the block Jacobi method can be implemented such that the error and the solution are calculated in parallel. It does not matter if the calculation starts with the 2nd, 5th or 100th element of the solution. This way, all the elements of the solution can be calculated at the same time, which increases the speed of every iteration and will therefore arrive earlier at the solution.

4.3. Gauss-Seidel

Similar to, but way faster than, the Jacobi method is the Gauss-Seidel method. This method was found by two mathematicians Carl Friedrich Gauss and Philip Ludwig von Seidel. For this specific method, the matrix must again be strictly diagonally dominant **or** symmetric and positive definite, meaning all eigenvalues are strictly positive. The way this algorithm works, is similar to the Jacobi method. However, Gauss and Seidel split their matrix in a different way. Using the Gauss-Seidel method, the matrix A will be split into three matrices such that $A = L+D+U$ where L is the strict lower triangular matrix, D is the diagonal matrix and U is the strict upper triangular matrix. Then, again, given a starting solution x^0, the general method equals

$$x^{k+1} = (D + L)^{-1}(b - Ux^k)$$

whereas in this case for generality $M = D + L$ and $N = U$. As known, both the Jacobi and the Gauss-Seidel method solves one equation for one variable at a time. However the difference is specified in the error. Jacobi uses only the diagonal part of the matrix to calculate the error. Gauss-Seidel uses also the lower triangular part and therefore a greater part of the matrix. Then by the theory in Section 4.1, this method converges way faster since the inverse is closer to the inverse of the exact matrix A.

Unlike the block Jacobi method, creating a parallel variant of the block Gauss-Seidel method is not that trivial. Since the block Gauss-Seidel method does not only use the diagonal to create solutions but actually uses the lower triangular part of the matrix. This is due to the fact that the block matrices are used in a different way. Here, as well as for the block Jacobi method, $D = A_0$ and $L = A_1$. But since the block Gauss-Seidel method uses the whole matrix, since $U = 0$, instead of only the diagonal, the current solution depends on the previous one. For parallelism, this gives problems. Therefore, to create a perfect Gauss-Seidel parallel variant is not possible. Nevertheless, for convergence it implies that iterating is done in one iteration. However, when combining the block Gauss-Seidel method with the block Jacobi method, a parallel variant could work. In this research, three different parallel variants are discussed. The algorithms show a combination between the usual block Gauss-Seidel method, where the error and the solution can not work simultaneously, and the block Jacobi method where those two can indeed be separated.

In the few subsections that will remain in this chapter, the characteristics of these methods will be explained.

4.3.1. Method 1: Step-wise

For the step-wise method, one takes a Jacobi iteration for every step chosen. This method calculates the solutions for a specific time step using the block Jacobi method, which could be done in parallel (theoretically speaking), then takes the sequential Gauss-Seidel for the next iteration till the (step-1)-th time step. When this part is done, the next step will again use the Jacobi iteration (in parallel). Continuing this gives the step-wise algorithm. The general idea for this algorithm is to divide the time steps in different step sizes such that every step-th iteration can be calculated in parallel time by Jacobi and the rest will just work using the sequential method of Gauss-Seidel.

4.3.2. Method 2: Red-Black Ordering

The second method that will be considered is the Red-Black Ordering block Gauss-Seidel method. In this algorithm, the iterations are split into odd and even iterations. For the use in parallel time, this algorithm would work extremely fast. Basically, all the odd iterations can be executed in parallel and all the even iterations afterwards as well. Since the block Gauss-Seidel method needs only the previous solution to calculate the current one, this method should work in parallel time. However, due to Matlab, the only analysis for this algorithm will be about the sequential variant.

4.3.3. Method 3: Processor-wise

The third and last method that will be implemented uses a processor-wise computation. This method is a special variant of the step-wise combination method and uses a step size of $\frac{nt}{np}$. Here the iterations are divided over a chosen amount of processors. So basically, the number of processors decides what the maximum number of iterations is. That equals that if there are np processors and the sequential form then needs $\leq np$ iterations to converge, this method is done in one iteration in parallel time.

5

Predicting Behavioural Results

In this chapter, the evaluation of each algorithm will be clarified. This will be done for every boundary condition mixture. For simplicity and neatness of the research, three abbreviations are used. D for Dirichlet, N for Neumann and R for Robin. The position of this letter in comparison with the other letter illustrates the left or the right boundary.

All experiments have been performed in Matlab on a 4-processor computer by using $nx = 100$ as the number of grid points.

For the remainder of this research, the behaviour and efficiency of all the methods will be discussed for the sequential forms. For a parallel implementation of the block Jacobi method in Matlab using the 'parfor' function, the reader is to be referred to Appendix E.

5.1. Goal

The goal of this research is to predict how fast an algorithm would perform in parallel time. To do this, a few things must be proven first. Initially, the algorithm should give the same answer as the exact answer. Secondly, the algorithm should converge the way that it is mathematically correct. So, the Jacobi and Gauss-Seidel characteristics must be clearly seen. At last, the algorithm must be tested for larger, and therefore more realistic, problems in order to be truly valid.

5.1.1. Heat Equation Testing

In order to confirm the right solution that is computed by the parallel variants of the block Jacobi method and the block Gauss-Seidel method, a test program has been written. With this, a solution can easily be checked and the error that the solution contains is also more clearly seen. Different boundary conditions give different results of course. An example of how such a solution is visualised in a heat rod can be found in Figure 5.1. What might stand out in this image is that the solution is found very fast. Therefore only at the very top of the image the solution was still close to the starting solution $x_0 = 0$. The image shown in Figure 5.1 has been printed for the Dirichlet conditions on both boundaries. For other boundary conditions, the solution would still look the same but the converging might take longer or follow a different path.

5.2. Hypothesis

During this research, four different algorithms have been studied. In this section, all four will be thoroughly analysed and certain expectations will be illustrated. To make a good hypothesis, theoretical mathematics is put to practice.

The Block Jacobi Method

The block Jacobi method uses just the diagonal in order to calculate the new solution. This implies that it should always need the same amount of time steps as the size of the blocks. To prove that the

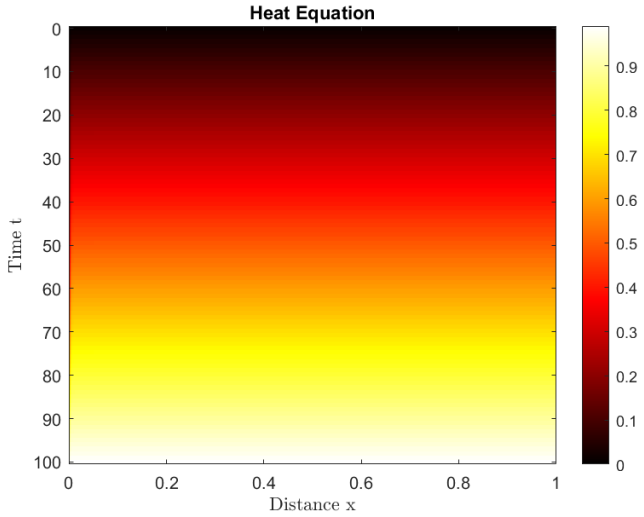


Figure 5.1: Flow of heat

matrix converges within the size of the blocks of the matrix, the Jordan normal form must be made. This form is typically known for its decomposition into the eigenvalue block matrix with ones on the upper diagonal of each block. So if $A = PJP^{-1}$, the Jordan normal form has a structure as following

$$J = \begin{pmatrix} \lambda_1 & 1 & & & & & & \\ & \lambda_1 & 1 & & & & & \\ & & \lambda_1 & 0 & & & & \\ & & & \lambda_2 & 1 & & & \\ & & & & \lambda_2 & & & \\ & & & & & \ddots & 0 & \\ & & & & & & \lambda_n & 1 \\ & & & & & & & \lambda_n \end{pmatrix} \tag{5.1}$$

This matrix is an illustration of how such a Jordan normal form would look like, the actual size of the Jacobi matrix used in the realistic research, has a different structure and is, obviously, way larger. Since, in the Jacobi iteration, the eigenvalues are zero, the main diagonal will equal zero as well. A very important characteristic of these matrices with zeros on the diagonal is that multiplying the matrix by itself will make the first upper diagonal zero and the second upper diagonal nonzero. This can continue for as long as the matrix is not equal to zero, because then the computation does not make any sense anymore. Visualising this will give that after one computation of $J \cdot J$, the matrix now looks like

$$J = \begin{pmatrix} 0 & 1 & & & & & & \\ & 0 & 1 & & & & & \\ & & 0 & & & & & \\ & & & 0 & 1 & & & \\ & & & & 0 & & & \\ & & & & & \ddots & & \\ & & & & & & 0 & 1 \\ & & & & & & & 0 \end{pmatrix} \tag{5.2}$$

Now if this action is repeatedly performed on the matrix A, one will result in

$$A^{nt} = (PJP^{-1})^{nt} = PJ^{nt}P^{-1} = P \cdot 0 \cdot P^{-1} = 0$$

Because of this fact, the expectation is that the block Jacobi method will converge earlier or at the last time step, since then the matrix is zero as well.

The Step-wise Block Combination Method

A similar hypothesis can be made on the step-wise Gauss-Seidel variant. In this variant, every step-th iteration is a Jacobi one, which, theoretically speaking, could be executed in parallel. Therefore, the maximum number of iterations that the algorithm should take in order to converge to the solution, must be at most $\frac{nt}{\text{step}}$. This can be proven, considering both the convergence of the block Jacobi method and the convergence of the block Gauss-Seidel method. As explained above, the block Jacobi method needs one iteration per time step. Therefore, it takes at most the number of time steps to get to the solution. However, the block Gauss-Seidel method is a method that is done iterating in one step, also earlier mentioned. This is, again, due to the structure of what part of the matrix the method takes. Now knowing this bit of information, one can easily predict the convergence of this step-wise algorithm. This matrix has the same structure as Jordan matrix 5.2. But now the matrix is split into step sub block matrices with eigenvectors of size $\frac{nt}{\text{step}}$.

Every step-th iteration, the Jacobi iteration is performed. Between two Jacobi iterations, a number of Gauss-Seidel iterations, step-1, are executed. Since these only take one iteration for every Jacobi iteration they follow, the number of iterations until the algorithm converges equals $\frac{nt}{\text{step}}$. Not to be mistaken with step-th iterations. This only holds, obviously, if step$^2 = nt$.

The Red-Black Ordering Method

When performing the Red-Black Ordering, the algorithm is split into two Gauss-Seidel calculations. One iteration performs the odd and the other one performs the even calculations. It is then to be expected that the maximum number of iterations that this method will take equals $\frac{nt}{2}$.

The convergence of this method works a bit different. Due to the permuting of the columns and rows symmetrically, the matrix A has a different structure, but nevertheless the same values. Therefore the matrix does not exactly look like the original matrix shown in block matrix 3.4. To calculate the transformed matrix A_T, one must find a matrix P such that $A_T = P^T AP$ and $A_T = P{-}1AP$, thus $P^{-1} = P^T$. This matrix for $nx = 3$, is shown below to illustrate the form of such a transformation matrix P.

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Here can be seen that the blocks of the block matrices will be shifted and therefore per time step dt, the blocks change order. Now, the matrix is changed to a matrix that does not only consist of a lower triangular block matrix. Therefore, the spectral radius is also different. Now the full matrix is defined such that all three sub matrices D, L and U are non-zero. So for determining the spectral radius, one must now find the largest eigenvalues in absolute value of the matrix $(D - L)^{-1}U$. Then using the theorem from Section 4.1, the following holds.
Since the eigenvalues of the matrix do not change when shifting the rows and columns, the eigenvalues of the sparse block matrix A are similar to the transformed one, A_T. In the case of Gauss-Seidel, these eigenvalues are all zero. Then again, the Jordan normal form can be made. This matrix has the same structure as matrix 5.2. But now the matrix is split into two sub block matrices with eigenvectors of size $\frac{1}{2}nt$. Then again, the same calculation can be performed on the transformed matrix A_T such that

$$A_T^{\frac{1}{2}nt} = (PAP^{-1})^{\frac{1}{2}nt} = PA^{\frac{1}{2}nt}P^{-1} = P \cdot 0P^{-1} = 0$$

And it is seen that this method converges after $\frac{1}{2}nt$ iterations.

The Processor-wise Block Combination Method

As explained briefly earlier in this research, this method works with the distribution over processors. For a number of processors chosen, the method basically divides the iterations over the different workers. Expected to see is that the solution takes a maximum iteration of np. So, for every processor 1 to np, there are $\frac{nt}{np}$ Gauss-Seidel steps that must be performed. Then only np Jacobi iterations are left to be performed. Since Gauss-Seidel does only take one iteration to get to the solution, the maximum number of iterations should be np. Another way to prove this is to verify using the step-wise combination method. For that method, the maximum number of iterations is $\frac{nt}{\text{step}}$. Now, for this method the step would be $\frac{nt}{np}$. Filling this step size in will give a maximum number of iterations of

$$\frac{nt}{\frac{nt}{np}} = np$$

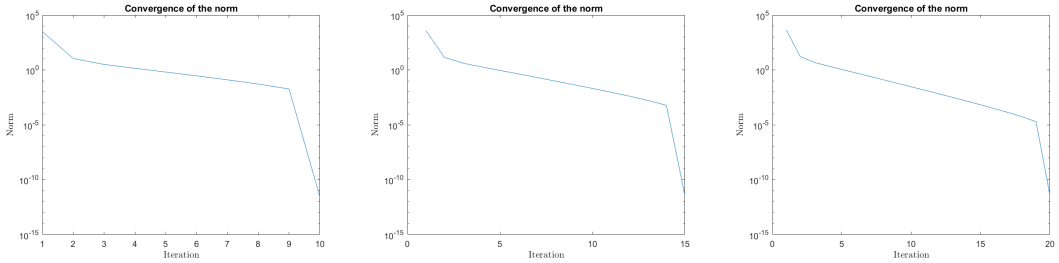
Hence the Jordan blocks as described in matrix 3.4, have size np.

5.3. Testing on Small Problems

To really validate the correct working of an algorithm, first testing on small problems, for example $nt = 10$, with extremely small tolerance, tol $= 10^{-6}$, is very important. This way, one can fully analyse the potential and notice small errors may they exist. To prove the correctness of all the four algorithms, both the exact solution and the prediction of the convergence should be made valid.

The Block Jacobi Method

As seen in last section, the block Jacobi method only converges in nt time steps. The algorithm is being executed for the time steps $nt = 10, 15, 20$. For all the three test problems, the number of iterations until the algorithm reaches the solution equals, respectively, 10, 15 and 20. Also, the exact solution is achieved and by that the hypothesis of this method is validated.
The way the norm of the error converges is displayed in Figure 5.2.



(a) $nt = 10$ **(b)** $nt = 15$ **(c)** $nt = 20$

Figure 5.2: Convergence of the block Jacobi method

The Step-wise Block Combination Method

For this method, three things should become clear. When choosing step = 1, the exact same value for the block Jacobi method should be the result. When taking step = nt, the calculation should be done in 1 iteration since this implies the block Gauss-Seidel method. Lastly, two steps between one and nt must be taken to prove that the mix does work as well. For this experiment, $nt = 20$. As expectations

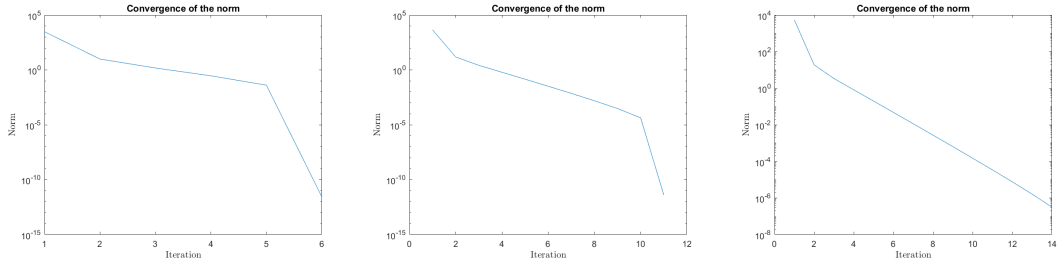
	No. iterations until the norm converges
step = 1	20
step = 4	5
step = 10	2
step = nt	1

Table 5.1: No. iterations Gauss-Seidel 1

are met, and the exact solution is equal to the analytic one as well, one can say that this algorithm works well in theory and for small problems.

The Red-Black Ordering Method

What is expected of the Red-Black Ordering, is that the algorithm always takes half the number of time steps nt for the iterations. To prove this, three different values for nt will be tested, as for the block Jacobi method. The results of this can be found in Figure 5.3.



(a) $nt = 10$ **(b)** $nt = 20$ **(c)** $nt = 30$

Figure 5.3: Convergence of the Red-Black Ordering method

Again, expectations are met except for the last experiment, $nt = 30$. The iterations stop at 13 already although it was expected to do so at 15. This implies that the tolerance should be lowered even more to really prove that the exact solution is been calculated at iteration number 15. To illustrate that this holds, the movement of the norm of the error for a tolerance of 10^{-8} can be found in Figure 5.4
From this figure can now be seen that with a tolerance of 10^{-8}, the solution really stops at 15 iterations, what is expected from the theory.

The Processor-wise Block Combination Method

As for the step-wise combination method, a few tests must be done to prove that the processor-wise algorithm also behaves as expected to. First of all, by taking $np = 1$, all iterations are distributed over one processor and therefore all iterations can be done by one Gauss-Seidel step. Hence, expected is that this computation takes only one iteration. Also, considering $np = nt$, the calculation should be done after np iterations, since this represents all Jacobi iterations. The results can be found in Table 5.2, for $nt = 20$.
As expected, the number of iterations until the norm of the error converges equals the number of processors. What is ideal about this method is that parallelizing the Jacobi iterations, which is possible as discussed, would make this method immediately faster, for even the slowest computers. The more the processors that a computer possesses, the bigger problems can be parallelized. For more information on the code itself, one may check Appendix B.

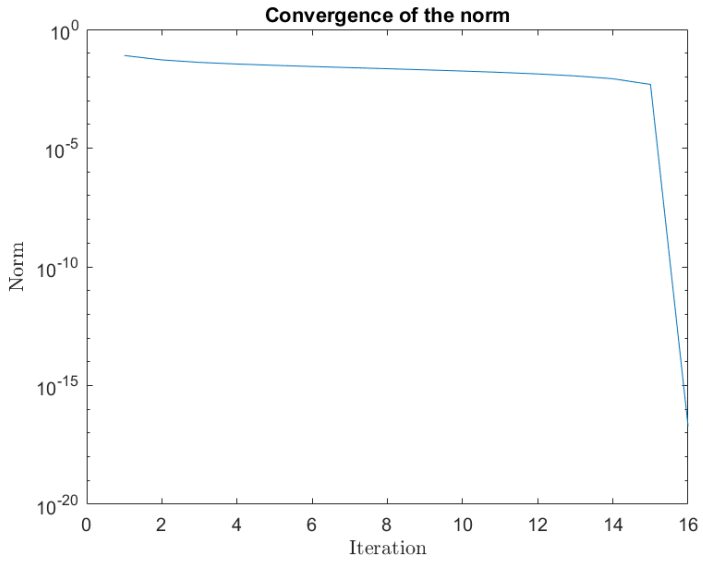


Figure 5.4: Convergence of the Red-Black Ordering method for $nt = 30$

	No. iterations until the norm converges
np = 1	1
np = 4	4
np = 10	10
np = nt	20

Table 5.2: No. iterations Gauss-Seidel 3

5.4. Testing on Realistic Problems

In reality, the actual value of the tolerance is much bigger, namely the error in the time step, dt. This could imply that the solution for which the error is accepted is finished iterating earlier than the exact answer is achieved. To create more realistic problems, the number of time steps is now set to 100. Also, since this research is still about creating fast algorithms, the maximum number of iterations that the program may take is set to 110. Although, due to the analysis that is explained in the last section, it is not expected that this maximum number of iterations is actually achieved.

The Block Jacobi Method

For creating the results that can be found in Table 5.3, a code in Matlab is used. This code can be found in Appendix B.

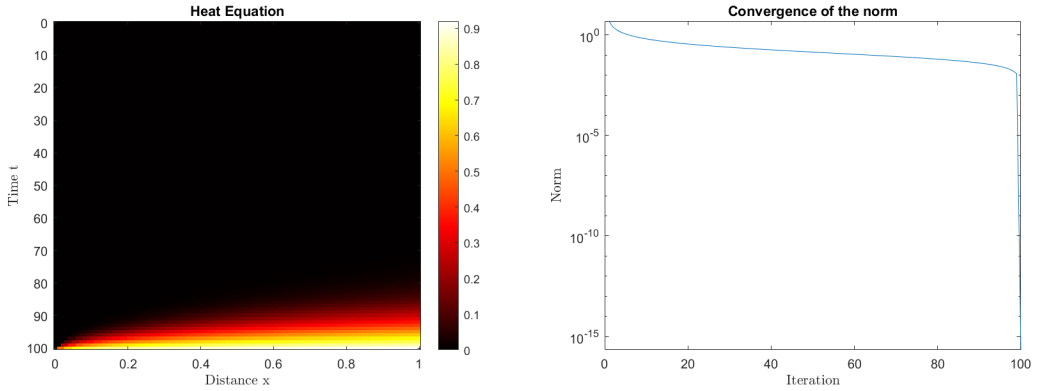
Boundary conditions		DD	DN	ND	NN	DR	RD	NR	RN	RR
Forward Euler		100	100	100	100	100	100	100	100	100
Backward Euler		10	23	21	9	15	10	47	47	8
θ-Rule	$\theta = 0.2$	100	100	100	100	100	100	100	100	100
	$\theta = 0.4$	100	100	100	100	100	100	100	100	100
	$\theta = 0.6$	29	22	29	20	20	29	46	46	20
	$\theta = 0.8$	10	22	20	8	14	10	47	47	7

Table 5.3: Block Jacobi method no. iterations until convergence

Analysing Table 5.3, most of the results are logical compared to the hypothesis of the converging rate of each time integration method. The more stable a method is, the easier it converges to the solution. For the explicit methods: Forward Euler and the θ-Rule with $\theta = 0.2, 0.4$, it can be clearly seen that the number of iterations does not continue further than the number of time steps nt. This is in accordance with the hypothesis and the results from testing on smaller problems for the block Jacobi method. Ex-

plicit methods converge slower. However, an explicit method might as well find a solution but it would take too much time to even consider choosing such a method.

Table 5.3 does show a noteworthy result for the boundary condition mix of Neumann and Robin. For all implicit methods, the number of iterations exceeds the other results with a very high number, namely more than 40 iterations. To be seen is that this also holds for all the other iterative methods that are to be discussed. There is no odd value for the boundary conditions used in this specific calculation, so there is no logical conclusion for this exceptional peak in iterations. In the discussion, Chapter 7, a recommendation for further research will be given on this boundary condition mix.



(a) Flow of heat

(b) Convergence of the norm

Figure 5.5: Forward Euler and θ-Rule for $\theta < 0.5$ for DN

Due to the stability of the time integration method, the explicit methods can not fully gain the best solution. As a result, the two figures shown in Figure 5.5 are made. In Figure 5.5a, one can see that for almost the entire time, the solution stays extremely close to zero. Only the last few iterations show that the solution is slowly moving to the exact solution as in Figure 5.1. From this can be concluded that the algorithm does work but is so slow that it makes no sense using it with time integration methods FE and $\theta < 0.5$. Also, considering Figure 5.5b, the norm is not decreasing linearly and only moves all the way down at the very end of the iterations, namely at the last time step $nt = 100$. Hence, both plots show that for these specific time integration methods, the explicit ones, give little to no value to the experiments. For different boundary conditions, as in Figure 5.6 the tolerance, tol $= dt$ can also be reached earlier than the maximum number of time steps. This holds for all the implicit methods. Since these methods are unconditionally stable, a relatively large time step could be chosen, namely $dt = 0.1$. Therefore the tolerance is also very large and the calculation is broken off earlier than expected. This does not imply that the norm of the error is already zero at this point, it implies that the numeric solution differs from the exact solution just enough to be valid for this problem. What can be concluded from this, is that if the tolerance depends on the time step, dt, the no. iterations until this tolerance is reached is always less than or equal to the maximum number of iterations, whenever this maximum is smaller than the number of time steps nt. Otherwise, the number of iterations is always less than or equal to the number of steps nt.

The Step-wise Block Combination Method

For this experiment with the step-wise block Gauss-Seidel method, Table 5.4, the step is set to 10. Changing the step-size obviously makes a change in the number of iterations until the solution converges. With step size being equal to 10, the maximum number of iterations that can be taken, following the theoretical analysis, equals $\frac{nt}{\text{step}} = \frac{100}{10} = 10$. Therefore in Table 5.4, all the explicit methods give a no.iterations of 10.

In Table 5.4 can be seen that with a step of size 10, the number of iterations are way less than for the block Jacobi method. The larger the step size, the less Jacobi steps will be taken so the more Gauss-Seidel steps are used and therefore the convergence is achieved earlier. Noting the number of iteration for the Backward Euler time integration method and for the θ-Rule with $\theta = 0.8$, the number of iterations are exactly the same. One can conclude that this algorithm works so well that the iterations

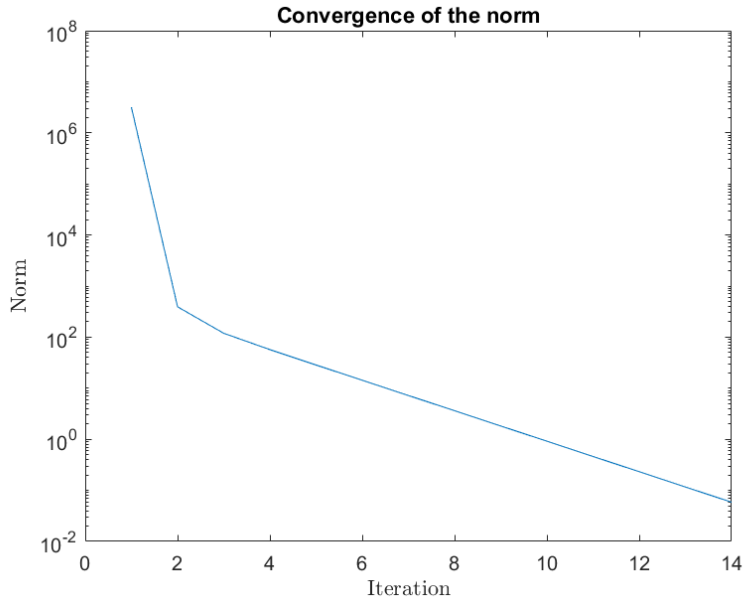


Figure 5.6: Convergence of block Jacobi method for DR and $\theta = 0.8$

Boundary conditions		DD	DN	ND	NN	DR	RD	NR	RN	RR
Forward Euler		10	10	10	10	10	10	10	10	10
Backward Euler		2	4	4	2	3	2	8	8	2
θ-Rule	$\theta = 0.2$	10	10	10	10	10	10	10	10	10
	$\theta = 0.4$	10	10	10	10	10	10	10	10	10
	$\theta = 0.6$	4	4	4	3	3	4	8	8	3
	$\theta = 0.8$	2	4	4	2	3	2	8	8	2

Table 5.4: Step-wise (10) block Gauss-Seidel method no. iterations until convergence

can not be improved more from at least $\theta = 0.8$.

The Red-Black Ordering Method

Using the theory that has been proven in the last section, the Red-Black Ordering algorithm should at most take $\frac{nt}{2}$ iterations. Now that $nt = 100$, the maximum number of iterations that this algorithm can take equals $\frac{100}{2} = 50$.

Boundary conditions		DD	DN	ND	NN	DR	RD	NR	RN	RR
Forward Euler		50	50	50	50	50	50	50	50	50
Backward Euler		6	13	12	5	8	5	28	28	12
θ-Rule	$\theta = 0.2$	50	50	50	50	50	50	50	50	50
	$\theta = 0.4$	50	50	50	50	50	50	50	50	50
	$\theta = 0.6$	14	12	14	9	9	14	27	27	9
	$\theta = 0.8$	5	12	11	4	8	5	27	27	4

Table 5.5: Red-Black Ordering block Gauss-Seidel method no. iterations until convergence

There are very straight-forward differences between the explicit and the implicit methods. A visual perspective on the convergence of the norm of the Red-Black Ordering method with Forward Euler and using the Dirichlet boundary conditions is displayed in Figure 5.7. In this figure, the convergence does not show a linear line, as expected with the explicit methods. Also, it drops rapidly at 50 iterations, since then the algorithm should stop. With this can easily be concluded that as long as $nt \leq$ max_itr, the maximum number of iterations until the norm of the error of the solution reaches the tolerance constraints equals $\frac{1}{2}nt$, as is expected in theory. Of course, since the tolerance is set equal to the time

step, dt, the implicit methods all converge earlier than $\frac{1}{2}nt$.

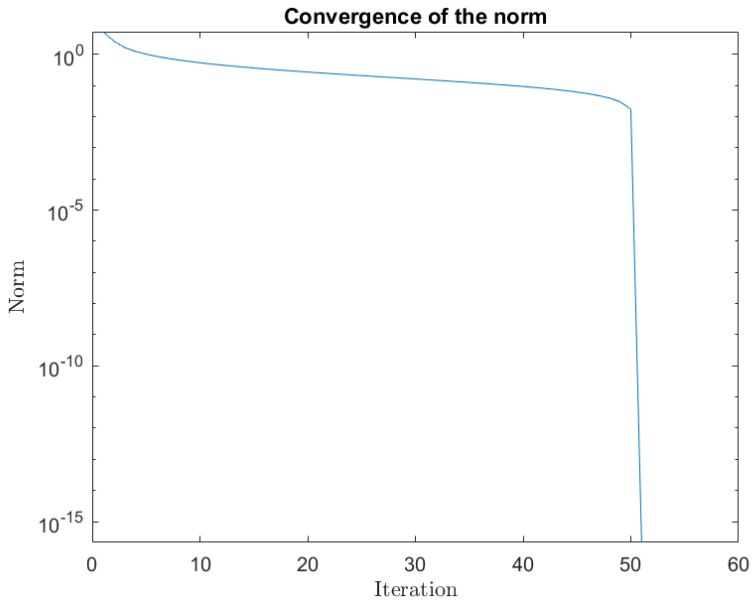


Figure 5.7: Convergence of the norm for Red-Black Ordering for DD

The Processor-wise Block Combination Method

In Table 5.6, the results for the third algorithm of the Gauss-Seidel variants can be found. For this specific experiment, $np = 20$. Respecting the theory again, the maximum number of iterations that are possible to be taken are $np = 20$.

Boundary conditions		DD	DN	ND	NN	DR	RD	NR	RN	RR
Forward Euler		20	20	20	20	20	20	20	20	20
Backward Euler		9	18	16	8	12	9	20	20	7
θ-Rule	$\theta = 0.2$	20	20	20	20	20	20	20	20	20
	$\theta = 0.4$	20	20	20	20	20	20	20	20	20
	$\theta = 0.6$	20	17	20	17	17	20	20	20	17
	$\theta = 0.8$	9	17	16	7	12	9	20	20	7

Table 5.6: Processor-wise (20) block Gauss-Seidel method no. iterations until convergence

Not surprising is that all the explicit methods give the maximum number of iterations again. However, here for the boundary condition mix of the Robin and Neumann conditions, the number of iterations before the norm reaches the tolerance is also the maximum. This is not seen earlier in the results of the other Gauss-Seidel variants. Concluding from this, is that this method might not be the very best in working as the other two Gauss-Seidel variants do.

5.5. Remarks on Parallelisation

In this last section of the chapter, some remarks on the findings will be given. Also, one may find conclusive statements that are crucial for the understanding of the whole process.

Additional Remarks

As can be seen in the results, for some Gauss-Seidel variants, the number of iterations for $\theta = 0.8$ is less than for the Backward Euler method. It is logic that how larger the θ goes, how better the computation would go, however the largest number θ can take is 1 and this implies exactly Backward Euler. This happens since the definition of the two different methods is being implemented slightly in a different manner. Nevertheless, the difference is very small and the error as well.

Furthermore, in the plots and from the code, the attentive reader might notice that the number of iterations sometimes stops at one iteration more than expected. This is due to the fact that the error is updated earlier than the solution and that therefore the iterating continues for one more iteration while the solution was already achieved. In the plots, there is a very visual 'kink' in the graph. At this 'kink', the solution is achieved and the number of iterations correspond with that value as well, though the graph does iterate one more time.

Conclusive Remarks

All the four methods work as expected and follow the general mathematical rules. Since all these algorithms are being analysed in their sequential form, the actual parallelisation has not been considered. The experiments on the parallel block Jacobi method can be found in Appendix E. However, predictions could be made for the Gauss-Seidel variants.

What can be seen from the results in this chapter, is that every algorithm depends on the Jacobi iteration. The Jacobi iteration is therefore the factor that the speed of the algorithm revolves around. First of all, in the block Jacobi method, the number of time steps nt equals the number of iterations until the solution converges. In Figure 5.2, this is clearly illustrated. For the Gauss-Seidel variants, a similar remark can be given. For the step-wise combination, the step size decides how many Jacobi iterations there are. Basically if every step, a Jacobi iteration is being executed, then in total there will be $\frac{nt}{\text{step}}$ Jacobi iterations and that is exactly how many iterations the algorithm needs to converge. The Red-Black Ordering can be seen as a split between Gauss-Seidel and Jacobi. Every **other** iteration, the algorithm performs a Gauss-Seidel iteration. Therefore, the algorithm is split between half Jacobi and half Gauss-Seidel iterations. Therefore the maximum number of iterations that this algorithm will take to converge equals $\frac{nt}{2}$. For the last variant discussed, the processor-wise block combination method the maximum number of iterations is specified by the number of processors chosen. Hence this number can never be larger than the number of processors itself, np. This is because for every processor, a Jacobi iteration takes place and that determines the maximum number of iterations.

As known, normal time integration can not be parallelized due to the continuity and the direction in which time moves. What was seen in this research is that every Jacobi and Gauss-Seidel iteration takes about the same amount of computation time as a time integration. However, these iterative methods can be used in parallelisation and are therefore extremely useful when dealing with large problems. Every Jacobi iteration that is executed by the program can be done in parallel. Therefore, a computer with x processors will make the computation x times faster. Hence, having a program with less than x Jacobi iterations, the program will execute the computation faster than the sequential form, and that is exactly what was needed to be achieved.

6

Conclusion

In order to contribute an answer to the question 'Can you parallelize time?', all the important findings will be summed up and specified in this chapter. Here, the advice will be given on using what kind of iterative method with what conditions work the best to parallelize time.
From this research, four different iterative methods have been tested and implemented to approximate the solution of the partial differential equation: the linear heat equation in 1D. The initial goal was to parallelize these methods and find the fastest computation time. However, this parallelization did not work out for the Gauss-Seidel variants. What is discussed, is how these methods perform in time. From this, predictions were made and one could assume how the parallelization would work.

For the block Jacobi method, the research has shown that this algorithm converges slowly. It takes at most as much as the number of time steps nt chosen to converge. The bigger the number is, the longer the algorithm takes. Therefore, in order to be working in parallel time, the computer must need as much processors as number of time steps chosen. Hence, for more accurate results, more processors are needed, which is very inefficient.

The block Gauss-Seidel method converges very fast due to the way the computation is made. This method uses the lower triangular matrix instead of just the diagonal and is therefore done iterating in one iteration. However, due to this choice of matrix computation, this method can not be parallelized. Therefore variants and combination of the block Gauss-Seidel and block Jacobi methods have been implemented and are created in order for parallelization to be possible. Due to the way Matlab works, these variants have not yet been tested in parallel time but are mostly predicted by looking at their convergence rate.
The first variant of combination methods works with a step size where for every step, a Jacobi iteration is used. The larger the step, the less Jacobi iterations, the faster the solution converges. This method works well in sequential time and takes a maximum of $\frac{nt}{\text{step}}$ iterations. The Red-Black Ordering method distributes the odd iterations and the even iterations separately. Therefore every other iteration is a Jacobi iteration. This method would be very fast in parallel time, however for the sequential form it takes at most $\frac{nt}{2}$ iterations. The last variant of the combination methods works with processors, where all the iterations are distributed over the processors such that per processor only one Jacobi iteration is performed. Since the Jacobi iterations determine the maximum number of iterations, this method has the characteristic that the number of iterations is always less than the number of processors np.

During this research, different boundary condition combinations have been used. Also, different time integration methods were applied. Forward Euler and the θ-Rule for $\theta < 0.5$ are explicit methods and therefore conditionally stable. In order for these methods to converge, the tolerance was very small, because it depended on the stability condition. Therefore, the results concerning explicit methods did not say much about the actual converging but more about when an iterative method is forced to stop. The implicit methods however, always converged earlier than the maximum number of iterations for that specific method. This is because the time step, dt, could be chosen way larger due to the unconditionally stability of implicit methods.

If time can be parallelized? Yes, by using Jacobi iterations in parallel time. Having a computer with more processors than number of Jacobi iterations performed, will also speed-up the computation. Our predictions show in particular that with only implicit time integration methods the speed-up can really be achieved.

7

Future Work

This report shows a thorough analysis of creating parallel variants of different iterative methods. Since there are always certain ideas that could have been worked out in a later research or follow-up experiments that could be performed, the following couple of paragraphs show optional extensions and future work:

- First of all, the use of the different boundary conditions are mostly touched upon but not thoroughly clarified. In a further research, there might be experiments on why the different boundary condition mixes give different results and how they have impact on the convergence. Especially, the Neumann and Robin combination should be analysed further to clarify the huge peak in iterations compared to the other boundary condition combinations.
- Moreover, due to time constraints, some interesting variants of the block Gauss-Seidel method were not discussed (yet). One of the rather special methods is called Asynchronous Jacobi/Gauss-Seidel. What this method describes is continuously jumping from one iteration to another. So basically, the whole process goes in parallel but not in a structured way. The advantage of this fact is that the program finds the fastest way to move through a computation, and therefore this implementation could be rather quick. [1]
- Additionally, in this research parallel computing is only performed on the 1D linear heat equation. For further research, considering the 2D linear heat equation might give different results. An even more interesting example would be to consider the 1D (or 2D) linear wave equation. Since the wave equation is a hyperbolic equation, it will always keep on fluctuating and no stationary solution will be found. Therefore parallel computing might have difficulties dealing with this equation.
- Lastly, but most importantly, it is shown that with the methods described in this thesis it is in principle possible to speed-up computations on a parallel computer. The next step is to implement the methods in a language that is suited for parallel computing and test the methods on the DelftBlue supercomputer. Then the results that are proven theoretically can really be visualised and used for larger purposes.

References

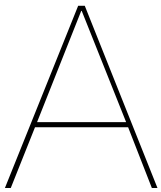
[1] I.P. Androulakis and G.V. Reklaitis. “Approaches to asynchronous decentralized decision making”. In: *Computers Chemical Engineering* 23.3 (1999), pp. 341–355. ISSN: 0098-1354. DOI: https://doi.org/10.1016/S0098-1354(98)00278-6.

[2] R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. 2nd ed. Society for Industrial and Applied Mathematics, 1994.

[3] J. Bylina and B. Bylina. “Merging Jacobi and Gauss-Seidel Methods for Solving Markov Chains on Computer Clusters”. In: (2008), pp. 263–268. DOI: 10.1109/IMCSIT.2008.4747250.

[4] A. Goddard and A. Wathen. “A note on parallel preconditioning for all-at-once evolutionary PDEs”. In: *Preprint arXiv* (2018). DOI: 1810.00615.

[5] C. Vuik and D.J.P. Lahaye. *Scientific Computing (wi4201)*. Delft Institute of Applied Mathematics, Delft, The Netherlands, 2019. URL: http://ta.twi.tudelft.nl/users/vuik/wi4201/wi4201_notes.pdf.



Matlab Code Initialisation

In this appendix, the code for the initialisation of the heat equation can be found.

```
clear all; close all;

BC = 'dirichlet';
ti = 'FE';

%% Initialisation
nx = 100;
nt = 100;
kappa = 1;
L = 1;
alpha = 1;
max_itr = 110;
e = ones(nx,1);
I = speye(nx);
x=zeros(nx,nt);
normVal=Inf;
itr=0;
u_start = zeros(nx,1);
dx = L/(nx+1);
D_L = 0;
D_R = 1;
N_L = -1;
N_R = 1;
R_L = -1;
R_R = 1;

%% Boundary conditions
% Dirichlet on both edges
if(strcmp(BC, 'dirichlet'))
    b_L = D_L/dx^2;
    b_R = D_R/dx^2;
    A = 1/dx^2*kappa*spdiags([e -2*e e], -1:1, nx, nx);
end

% Neumann left & Dirichlet right
if(strcmp(BC, 'neumanndirichlet'))
    b_L = 2*(N_L)/dx;
    b_R = D_R/dx^2;
```

```
39      A = 1/dx^2*kappa*spdiags([e -2*e e], -1:1, nx, nx);
40      A(1,2) = 2/dx^2;
41  end
42
43  % Dirichlet left & Neumann right
44  if(strcmp(BC, 'dirichletneumann'))
45      b_L = D_L/dx^2;
46      b_R = 2*N_R/dx;
47      A = 1/dx^2*kappa*spdiags([e -2*e e], -1:1, nx, nx);
48      A(nx,nx-1) = 2/dx^2;
49  end
50
51  % Neumann on both edges
52  if(strcmp(BC, 'neumann'))
53      b_L = 2*(N_L)/dx;
54      b_R = 2*(N_R)/dx;
55      A = 1/dx^2*kappa^2*spdiags([e -2*e e], -1:1, nx, nx);
56      A(1,2) = 2/dx^2;
57      A(nx,nx-1) = 2/dx^2;
58  end
59
60  % Robin left & Dirichlet right
61  if(strcmp(BC, 'robindirichlet'))
62      b_L = 2*(R_L)/dx;
63      b_R = D_R/dx^2;
64      A = 1/dx^2*kappa^2*spdiags([e -2*e e], -1:1, nx, nx);
65      A(1,2) = 2/dx^2;
66      A(1,1) = (2+2*alpha*dx)/dx^2;
67  end
68
69  % Dirichlet left & Robin right
70  if(strcmp(BC, 'dirichletrobin'))
71      b_L = D_L/dx^2;
72      b_R = 2*(R_R)/dx;
73      A = 1/dx^2*kappa^2*spdiags([e -2*e e], -1:1, nx, nx);
74      A(nx,nx-1) = 2/dx^2;
75      A(nx,nx) = -(2+2*alpha*dx)/dx^2;
76  end
77
78  % Robin on both edges
79  if(strcmp(BC, 'robin'))
80      b_L = 2*(R_L)/dx;
81      b_R = 2*(R_R)/dx;
82      A = 1/dx^2*kappa^2*spdiags([e -2*e e], -1:1, nx, nx);
83      A(1,1) = -(2+2*alpha*dx)/dx^2;
84      A(1,2) = 2/dx^2;
85      A(nx,nx) = -(2+2*alpha*dx)/dx^2;
86      A(nx,nx-1) = 2/dx^2;
87  end
88
89  % Neumann left & Robin right
90  if(strcmp(BC, 'neumannrobin'))
91      b_L = 2*(N_L)/dx;
92      b_R = 2*(R_R)/dx;
93      A = 1/dx^2*kappa^2*spdiags([e -2*e e], -1:1, nx, nx);
94      A(1,2) = 2/dx^2;
```

```
95      A(nx,nx) = -(2+2*alpha*dx)/dx^2;
96      A(nx,nx-1) = 2/dx^2;
97  end
98
99  % Robin left & Neumann right
100 if(strcmp(BC, 'robinneumann'))
101     b_L = 2*(R_L)/dx;
102     b_R = 2*(N_R)/dx;
103     A = 1/dx^2*kappa^2*spdiags([e -2*e e], -1:1, nx, nx);
104     A(1,1) = -(2+2*alpha*dx)/dx^2;
105     A(1,2) = 2/dx^2;
106     A(nx,nx-1) = 2/dx^2;
107 end
108
109 %% Time step
110 if(strcmp(ti, 'FE'))
111     dt = kappa^2*dx^2/2;
112     A0 = I;
113     A1 = -(I + dt*A);
114 end
115
116 if(strcmp(ti, 'BE'))
117     dt = 0.1;
118     A0 = I-dt*A;
119     A1 = -I;
120 end
121
122 if(strcmp(ti, 'theta'))
123     theta = 0.8;
124     if theta >= 0.5
125         dt = 0.1;
126     else
127         dt = dx^2/(-2*(2*theta-1));
128     end
129     A0 = I-dt*theta*A;
130     A1 = -(I+dt*(1-theta)*A);
131 end
132
133 %% Run
134 b = [u_start repmat(dt*[b_L;zeros([nx-2,1]);b_R],[1,nt-1])];
135 tol = dt;
```

B

Matlab Code Methods

In this appendix, the code for the different (parallel) block methods can be found.

```
clear all; close all;

%% Choose PDE
LinearHeatEquation1D;

%% Choose method
method = 'par-blk-gs3';

%% Sequential Block Methods
% Block Jacobi Method
if(strcmp(method,'blk-jac'))
    tic
    norm_r = zeros(itr,1);
    r = b;
    while (norm(r,'fro') > tol && itr < max_itr)
        r(:,1) = b(:,1) - A0*x(:,1);
        for i = 2:nt
            r(:,i) = b(:,i) - A0*x(:,i) - A1*x(:,i-1);
        end
        for i = 1:nt
            x(:,i) = x(:,i) + A0\(r(:,i));
        end
        itr = itr + 1;
        norm_r(itr) = norm(r,"fro");
    end
    toc
end

% Block Gauss-Seidel Method
if(strcmp(method,'blk-gs'))
    tic
    norm_r = zeros(max_itr,1);
    r = b;
    x(:,1) = A0\b(:,1);
    while (norm(r,'fro') > tol && itr < max_itr)
        r(:,1) = b(:,1) - A0*x(:,1);
        for i = 2:nt
            r(:,i) = b(:,i) - A0*x(:,i) - A1*x(:,i-1);
```

```
39              x(:,i) = x(:,i) + A0\(r(:,i));
40          end
41          itr = itr + 1;
42          norm_r(itr) = norm(r,"fro");
43      end
44      toc
45  end
46
47  %% (Parallel) Block Methods
48  % Parallel Block Jacobi Method
49  if(strcmp(method,'par-blk-jac'))
50      tic
51      norm_r = zeros(max_itr,1);
52      r = b;
53      while (norm(r,'fro') > tol)
54          r(:,1) = b(:,1) - A0*x(:,1);
55          parfor i = 2:nt
56              r(:,i) = b(:,i) - A0*x(:,i) - A1*x(:,i-1);
57          end
58          parfor i=1:nt
59              x(:,i) = x(:,i) + A0\(r(:,i));
60          end
61          itr = itr + 1;
62          norm_r(itr) = norm(r,"fro");
63      end
64      toc
65  end
66
67  % (Parallel) Block Gauss-Seidel Method 1
68  if(strcmp(method,'par-blk-gs1'))
69      tic
70      step = 10;
71      norm_r = zeros(max_itr,1);
72      r = b;
73      % First time step:
74      x(:,1) = A0\b(:,1);
75      r(:,1) = b(:,1) - A0*x(:,1);
76      while (norm(r,'fro') > tol && itr < max_itr)
77      % First Jacobi steps:
78          for j = 1+step:step:nt
79              r(:,j) = b(:,j) - A0*x(:,j) - A1*x(:,j-1);
80          end
81          for j = 1+step:step:nt
82              x(:,j) = x(:,j) + A0\r(:,j);
83          end
84      % Then Gauss Seidel
85          for j = 1:step:nt
86              for i = j+1:j+step-1
87              r(:,i) = b(:,i) - A0*x(:,i) - A1*x(:,i-1);
88              x(:,i) = x(:,i) + A0\r(:,i);
89              end
90          end
91          itr = itr + 1;
92          norm_r(itr) = norm(r,"fro");
93      end
94      toc
```

```
95  end
96
97  % (Parallel) Block Gauss-Seidel Method 2
98  if(strcmp(method,'par-blk-gs2'))
99      tic
100     norm_r = zeros(max_itr,1);
101     r = b;
102     while (norm(r,'fro') > tol && itr < max_itr)
103         r(:,1) = b(:,1) - A0*x(:,1);
104         x(:,1) = A0\b(:,1);
105         for i=3:2:nt
106             r(:,i) = b(:,i) - A0*x(:,i) - A1*x(:,i-1);
107             x(:,i) = x(:,i) + A0\r(:,i);
108         end
109         for i = 2:2:nt
110             r(:,i) = b(:,i) - A0*x(:,i) - A1*x(:,i-1);
111             x(:,i) = x(:,i) + A0\r(:,i);
112         end
113         itr = itr + 1;
114         norm_r(itr) = norm(r,"fro");
115     end
116     toc
117 end
118
119 % Parallel Block Gauss-Seidel Method 3
120 if(strcmp(method,'par-blk-gs3'))
121     tic
122     norm_r = zeros(max_itr,1);
123     np = 1;
124     b = reshape(b,nx,np,nt/np);
125     r = b;
126     x = zeros(nx,np,nt/np);
127     r(:,1,1) = b(:,1,1) - A0*x(:,1,1);
128     x(:,1,1) = A0\b(:,1,1);
129     while (norm(r,'fro') > tol && itr < max_itr)
130         for i = 2:np
131             r(:,i,1) = b(:,i,1) - A0*x(:,i,1) - A1*x(:,i-1,1);
132         end
133         for i = 2:np
134             x(:,i,1) = x(:,i,1) + A0\r(:,i,1);
135         end
136         for i = 1:np
137             for j = 2:nt/np
138                 r(:,i,j) = b(:,i,j) - A0*x(:,i,j) - A1*x(:,i,j-1);
139                 x(:,i,j) = x(:,i,j) + A0\r(:,i,j);
140             end
141         end
142         itr = itr + 1;
143         norm_r(itr) = norm(r,"fro");
144     end
145     toc
146 end
147
148
149 %% Figures
150 nx = linspace(0,L,100);
```

```
151 nt = [linspace(0,100,100)];
152 figure;
153 colormap hot
154 imagesc(nx,nt,x)
155 colorbar
156 xlabel('Distance x','interpreter','latex')
157 ylabel('Time t','interpreter','latex')
158 title('Heat Equation')
159
160 figure;
161 semilogy(norm_r)
162 xlabel('Iteration','interpreter','latex')
163 ylabel('Norm','interpreter','latex')
164 title('Convergence of the norm')
```

C

Matlab Code Heat Equation Test

In this appendix, the code for the testing the exact solution to the heat equation can be found.

```
clear all; close all;

%% Analytic solution
L = 1;
x = linspace(0,L,100);
t = [linspace(0,10,200)];

m = 0;
sol = pdepe(m,@heatpde,@heatic,@heatbc,x,t);
sol = transpose(sol);

colormap hot
imagesc(x,t,sol)
colorbar
xlabel('Distance x','interpreter','latex')
ylabel('Time t','interpreter','latex')
title('Heat Equation')

function [c,f,s] = heatpde(x,t,u,dudx)
c = 1;
f = dudx;
s = 0;
end

function u0 = heatic(x)
u0 = 0;
end

function [pl,ql,pr,qr] = heatbc(xl,ul,xr,ur,t) % p(x,t,u) + q(x,t)du/dx =
    0
pl = ul-1;
ql = 0;
pr = ur;
qr = 0;
end
```

D

Matlab Code Transformed Matrix

In this appendix, the code for the creating the transformation matrix P can be found.

```
clear all;

LinearHeatEquation1D;

P = zeros(nx*nx,nx*nx);
P_i = eye(nx);

for j = 1:nx:nx^2/2
    i = 2*j-1;
    P(i:i+nx-1,j:j+nx-1) = P_i;
end
for j = (nx^2/2)+1:nx:nx^2
    i = 2*(j-nx^2/2)+nx-1;
    P(i:i+nx-1,j:j+nx-1) = P_i;
end

AA = blktridiag(A0,A1,zeros(nx),nx);
A_T = transpose(P)*AA*P;
```

E

From Sequential to Parallel

Originally, this research concerned the discussion about if iterative methods could be working in parallel time. This research has been conducted but was not complete due to programming issues. In this small part, the findings that concerns parallel computing are displayed.

The 'parfor'-loop

For the Jacobi method, as explained earlier, the error and the solution line can be executed separately. Therefore creating a parallel variant is rather easy. In Matlab, a way to transform a sequential method into a parallel one, is to change the 'for'-loops into 'parfor'-loops. In order for the parallel variant to be efficient, the number of iterations should be equal or less than the number of processors on a computer. However, due to the lack of processors on the computer this research has been conducted on, the actual computing time for the Jacobi method is not really reliable. This elapsed time for the Jacobi method with time integration method Forward Euler can be found in Table E.1. As can be seen, the parallel computations take about 75 times longer than the computations with the sequential method. What the hypothesis would tell us is that the parallel computing should take 4 times less long, due to the number of processors on the computer this research is conducted on. The unfortunate fact is that Matlab is not capable of computing these solutions faster. The computation itself, however, does go in parallel but the actual results that are desired do not show. When performing this algorithm on a supercomputer for example, the results might show after all. Although this method is working in parallel time, the parallel variant is way less efficient.

Since the Jacobi method uses only the diagonal block to calculate the next solution, this method will always need at least as much processors as maximum number of iterations that is specified. Therefore, for this research, the number of processors needed to perform this calculation in fast parallel time equals 100, preferably executed on a supercomputer.

Boundary conditions	DD	DN	ND	DR	RD	NN	NR	RN	RR
Sequential	0.118	0.128	0.118	0.109	0.106	0.141	0.105	0.127	0.103
Parallel	8.5	8.8	8.6	8.3	8.4	8.7	8.3	8.9	8.2

Table E.1: Elapsed time - Jacobi method for Forward Euler

For parallelizing the Gauss-Seidel method, some actions had to be performed that were not applicable in Matlab. Creating a 'for'-loop inside of a 'parfor'-loop is not a function that Matlab is programmed to execute unfortunately. This is the reason why there are no parallel results for the Gauss-Seidel variants.

Another important note when looking at the elapsed time is that when speaking of elapsed time, this implies the time that Matlab needs to perform a certain computation. However, since a computer works with processors and these sometimes need to clear capacity, the computation time of a single computation can be a bit off every time. The results do not change whether it takes 10% longer or not, but a precise time can not be given. Therefore when elapsed time is used, always the 5th time that the

computation is being performed, the time is saved. The choice of the 5th time is because the first, second and sometimes the third time as well, the computations times really fluctuated heavily. Until the fourth time a computation is run, the results differed relatively less with each other. Therefore a save, and somewhat valid solution would be to use the 5th time a computation is run.

Also, doing computations on a complex program like Matlab, having only 4 processors on the computer, is not really helpful. As Matlab has a lot of functions that need capacity to be stored, not much will be left for doing the actual computations. Therefore, performing the parallel Jacobi algorithm on better computers might have improved the elapsed time. Likewise, Matlab does a lot of work within the program itself. This also implies that some functions work in parallel without using the 'parfor' function. Since this is how Matlab is implemented, getting rid of its internal parallel functions is not possible, or at least not within the scope of this research. All this implies that Matlab can be slower than expected due to computations that are not seen in the code itself. Slowing down a program is a large disadvantage of this fact.