An Implementation of the All-At-Once Method to Evolutionary PDEs

Anthony Goddard

St Hilda's College

University of Oxford

A thesis submitted for the degree of

M.Sc. in Mathematical Modelling and Scientific Computing

Trinity Term 2018

Acknowledgements

First, I would like to thank my grandfather and my mother, Basil and Sara. Without their support, in every possible sense of the word, this work would not have been possible. On the note of financial support, I would like to thank the James Pantyfedwen Foundation who were instrumental in the funding of this masters programme.

I would also like to thank Prof Andrew Wathen for sharing his endless knowledge in the area of computational mathematics during this time, his excellent advice on the initial draft of the paper and, most importantly, for giving me the opportunity to publish work alongside him on the topic of this work.

If there is a gold standard for course directors, Dr Kathryn Gillow defines it. Her advice and comments on the final drafts of the paper made a huge difference.

Finally, I would like to thank Annika Maresia for her diligent proofreading and patience throughout this year.

Abstract

Evolutionary problems are abundant in science and engineering. The objective of this work is to present a method to efficiently solve such evolutionary problems in parallel: the all-at-once method. The success of all-at-once formulations of evolutionary problems lies partly in the effectiveness of preconditioning and partly in the structure of the underlying matrices.

We present new extensions to the all-at-once method, which will allow for consideration of non-uniform temporal domains, as well as parallel implementations for both the heat equation and the wave equation.

Contents

Chapter 1

Introduction

Partial Differential Equations (PDEs) are ubiquitous in the application of mathematics to the sciences. Evolutionary PDEs are those that have a time derivative. While there is a plethora of techniques to solve PDEs, many are only feasible when the domains are simple geometries such as squares, triangles, etc. In more complex cases the mathematician is to rely on numerical techniques to solve such problems. However, numerical solutions to PDEs are not silver bullets: They are expensive to calculate. Computationally expensive problems, such as obtaining approximate solutions to PDEs, have driven research within the area of parallel computing. Parallel computing essentially boils down to distributing a computational task across multiple processes. In the ideal case the processes would seldom communicate and the speed-up would scale with the number of processes used.

There are two schools of thought when it comes to the parallel computation of the solution to evolutionary PDEs: parallelisation in space and parallelisation in time. It is a generally held viewpoint that parallelisation in time should only be considered when all parallelisation in space has been exhausted. A common method for parallelisation in space is domain decomposition. As the name suggests, this involves decomposing the problem domain into smaller subdomains and then performing the required calculations on the subdomains with the exterior of each of the subdomains communicating with the other exteriors. The subdomains are allocated to several processes in the interest of facilitating a speed-up.

An example of a parallel-in-time method is the parareal algorithm (see [10]), which involves splitting the time domain among processes and then, using a pair of integrators, iteratively integrating the temporal domain. Throughout this dissertation we are interested in the all-at-once method (see [14]). The all-at-once method is a parallel-in-time method and involves the instantiation of a monolithic system, which

is guaranteed to have a Toeplitz-like structure. It is then possible to take advantage of this structure in order to significantly reduce the cost induced to obtain an approximate solution.

1.1 The Structure of the Dissertation

In the remainder of Chapter 1 we are going to outline the idea behind Krylov subspace methods and preconditioning. Specifically, the so-called Strang circulant will be introduced. We will end Chapter 1 with a discussion on modern high performance computing. In Chapter 2 we will cover the particular details of the all-at-once method, our extension to non-uniform discretisations, and a successful application to hyperbolic problems. We will also discuss the symmetrisation of the monolithic system that results from the all-at-once method. In Chapter 3 we will go through the parallel implementation of the all-at-once method and in Chapter 4 we will present the numerical results. The outcome of Chapter 3 and Chapter 4 has been submitted for publication (see [5]).

1.2 Krylov Subspace Methods

When finding an approximate solution to a PDE using a numerical scheme, we usually arrive at a large system of linear equations which can be represented as

$$A\mathbf{x} = \mathbf{b}, \tag{1.1}$$

where $A \in \mathbb{R}^{n\times n}$. To obtain $\mathbf{x}$ we have to solve (1.1). We can solve this system using either a *direct* method or an *iterative* method. When A is a very large matrix we are hesitant to use direct methods since the usual candidates, Gaussian elimination and Cholesky factorisation, are $\mathcal{O}(n^3)$ methods (see [21]). However, when solving some physical problems, even when A is large it can have a very uniform structure with relatively few entries. In certain instances, this *sparse* structure can significantly reduce the cost of solving such a system. See Chapter 4 of [3] for an implementation of a sparse Cholesky factorisation; there are issues, however, with the fill patterns of such direct methods when applied to sparse matrices (see Section 7.5 of [1]).

In the introduction we mentioned that we would like to utilise many-core systems in our calculations to decrease the time it takes for us to arrive at a solution. Unfortunately, the direct methods mentioned above do not lend themselves to parallelisation

because they are inherently sequential, although the Basic Linear Algebra Subprograms (BLAS) library does have very fast parallel routines which significantly enhance performance. As a result, we are going to turn our attention to iterative methods, specifically Krylov subspace methods. The difference between a direct method and an iterative method is that iterative methods do not aim to solve (1.1) exactly but rather, they aim to solve (1.1) up to some tolerance. We provide a brief outline of Krylov subspace methods; for a full exposition please refer to [21].

Suppose that A of (1.1) is structured in such a way that $A\mathbf{r}$, and by extension $A(A\mathbf{r}) = A^2\mathbf{r}$, is easily calculated for any vector $\mathbf{r}$. In this case the k-th Krylov subspace, defined as

$$\mathcal{K}_k(A, \mathbf{r}) = \text{span}\{\mathbf{r}, A\mathbf{r}, ..., A^{k-1}\mathbf{r}\}, \tag{1.2}$$

is also easily calculated. This is a good representation of the Krylov subspace for demonstration purposes, but this basis is not used in practice due to the fact that $A^n\mathbf{r}$ generally tends to point in a single direction for large n. From some starting guess $\mathbf{x}_0$ we then seek iterates $\mathbf{x}_k \in \mathbf{x}_0 + \mathcal{K}_k(A, \mathbf{r}_0)$, where $\mathbf{r}_0 = A\mathbf{x}_0 - \mathbf{b}$. The first method we will discuss is the Minimal Residual method (MINRES), which was introduced in [16] in 1975. MINRES solves (1.1) by minimising the 2-norm of the residual, that is, by solving the least squares problem

$$\min_{\mathbf{x}-\mathbf{x}_0 \in \mathcal{K}_k(A,\mathbf{r}_0)} ||\mathbf{b} - A\mathbf{x}||_2. \tag{1.3}$$

The drawback of MINRES is that it can only be applied to symmetric matrices. It was not until 1986 that Saad and Schultz proposed the Generalised Minimal Residual method (GMRES), which can be used to solve (1.1) without there being any requirements on the structure of A (see [19]). Both methods rely on different algorithms to form a basis for the Krylov subspace: GMRES relies on Arnoldi's method while MINRES relies on the Lanczos algorithm. The crucial difference between the two methods is that MINRES has strong theoretical guarantees on the convergence of its iterates based solely on the matrix eigenvalues, while for GMRES we do not have such strong guarantees on convergence (see [7]).

1.3 Preconditioning

From an algebraic point of view, preconditioning is trivial. If we take (1.1) and multiply both sides by the inverse of a preconditioner P^{-1}, the system becomes

$$P^{-1}A\mathbf{x} = P^{-1}\mathbf{b}. \tag{1.4}$$

The solution of (1.4) is the same as that of (1.1). Of course, we are speaking from a "direct methods" point of view. If we apply MINRES to (1.4), then the rate at which MINRES converges is dependent on the eigenvalues of $P^{-1}A$, not those of A (see [21]). We select P such that the number of iterations required to solve (1.4) is less than that to solve (1.1). We must stress that we never form the inverse of P explicitly. However, in the iterative schemes that we use, we must still be mindful of the cost of solving

$$P\mathbf{y} = \mathbf{d} \tag{1.5}$$

once each iteration. Selecting a preconditioner and conjuring ways to apply it is more of a creative process than a mechanical one; yet there are guides. For instance, we want P to be *close* to A. Our interpretation of closeness follows that of [21]: We want the eigenvalues of $P^{-1}A$ to be close to 1 and $||P^{-1}A - \mathbb{I}||_2$ to be small. While Lecture 40 of [21] will be sufficient for our purposes, for further reading we suggest [22].

1.4 The Strang Circulant

There is one preconditioner that will be of significant importance to us: the Strang circulant. This was first proposed in [20] and was originally applied to Toeplitz matrices, as opposed to block Toeplitz matrices, which is where we will use them. A Toeplitz matrix is a matrix of the form

$$T = \begin{bmatrix} a_1 & a_{-2} & a_{-3} & \dots & a_{-(n-1)} & a_{-n} \\ a_2 & a_1 & \ddots & & & \vdots \\ a_3 & \ddots & \ddots & & & \\ \vdots & & & & \ddots & \vdots \\ a_{n-1} & & & \ddots & \ddots & a_{-2} \\ a_n & \dots & & \dots & a_2 & a_1 \end{bmatrix}. \tag{1.6}$$

Circulant matrices have the following form:

$$C = \begin{bmatrix} a_1 & a_n & a_{n-1} & \dots & a_3 & a_2 \\ a_2 & a_1 & \ddots & & & \vdots \\ a_3 & \ddots & \ddots & & & \\ \vdots & & & & \ddots & \vdots \\ a_{n-1} & & & \ddots & \ddots & a_n \\ a_n & \dots & & \dots & a_2 & a_1 \end{bmatrix}. \tag{1.7}$$

Note that both T and C have constant diagonals. As mentioned in [14], the Strang circulant results from taking the central band of T of width $n/2$ and wrapping it around to form a circulant matrix. The Strang circulant is then used to precondition a Toeplitz system. However, we will see that there are instances where the system does not necessarily need to be a Toeplitz system. A significant part of the success of this method, even though the Strang circulant is a good preconditioner, lies in the speed with which it can be applied. Circulant matrices are diagonalised by a Fourier basis and so we can write $C = U^*\Lambda U$, where $U_{j,k} = \mathrm{e}^{2(j-1)(k-1)\frac{\pi i}{n}}/\sqrt{n}$. This connection to the Fast Fourier Transform (FFT) means that matrix-vector multiplications with a circulant matrix can be performed in $\mathcal{O}(n \log n)$ operations (see [12]).

1.5 Exascale Computing

The current large-scale computing paradigm is petascale computing. Petascale systems are computing systems capable of carrying out 10^{15} calculations per second. The large-scale high performance computing systems of the next generation are referred to as exascale systems. Exascale systems are capable of carrying out 10^{18} calculations per second. In other words, exascale systems are capable of carrying out one thousand times as many calculations per second as petascale systems. The reason why there is a description of exascale systems in this work is that the paradigm shift, from petascale to exascale, requires a shift in the criteria to which we write modern parallel implementations. A report published by Falgout et al. outlines the challenges that surround the transition from exascale to petascale (see [18]).

The main issues are that the memory available per compute core of the systems will reduce in the interest of power consumption, and communication will be exaggerated due to a volumetric increase in infrastructure. We will be mindful of these limitations as we discuss the parallel implementation of the all-at-once method in Chapter 3. Sections 4.3.2 and 4.3.5 of [18] highlight the importance of efficient parallel-in-space methods and parallel-in-time methods. Furthermore, these methods can be used in combination. In [18], a major concern is the discovery of highly scalable parallel-in-time methods. Throughout the remainder of this work we aim to demonstrate that the all-at-once formulation has the capacity to scale very well.

While the jump from being able to calculate 10^{15} calculations per second to 10^{18} per second might seem meaningless to someone working outside computational mathematics, it is actually very important to those working in the field of mathematical modelling. It allows for the possibility of computing solutions to high-fidelity math-

ematical models within a reasonable time frame, which is something current high performance computing systems struggle to achieve. For instance, when calculations are executed in preparation for a weather forecast, simplifying assumptions are made that make the calculations cheaper overall. In the dawn of exascale computing some of these assumptions will no longer need to be made. There are benefits to using higher order methods which, as we will see, counter the effects of numerical dissipation present in approximate solutions obtained using lower order methods. This is simply because the communication between processes will be very expensive and as a result, according to [18], the extra cost associated with using the higher order methods will no longer be relevant. Another use of exascale computing is in the field of large-scale particle interactions. Aboria, a piece of software used in Bruna et al., is capable of large-scale reaction-diffusion simulations (see [2]).[1] Software such as Aboria would benefit significantly from exascale computing in the sense that larger numbers of particles could be considered and higher fidelity models could be implemented.

[1] `https://github.com/martinjrobins/Aboria`

Chapter 2

Formation of the Monolithic System

2.1 The Heat Equation

Consider the heat equation

$$\frac{\partial u}{\partial t} = \nabla^2 u + g \quad \text{on } \Omega \times (0, T], \tag{2.1}$$

$$u = 0 \qquad \text{for } \mathbf{x} \in \partial\Omega, \tag{2.2}$$

$$u(0, \mathbf{x}) = s(\mathbf{x}) \qquad \text{for } \mathbf{x} \in \Omega, \tag{2.3}$$

where g is a spatially dependent function, T is the length of the time interval, $\Omega \in \mathbb{R}^N$ is the spatial domain and N is the number of spatial dimensions. We are first going to discretise in space using the Finite Element Method (FEM). We present a terse description of FEM; for a full exposition see [23].

We let $H_1(\Omega)$ be the space of functions whose elements are square integrable and have square integrable derivatives. We define the trial space as

$$U = \{u \in H_1(\Omega) : u = 0 \text{ on } \partial\Omega\}. \tag{2.4}$$

There is another space that we must consider, the test space V. In the context of the problem (2.1)–(2.3), the trial space and the test space are equal. If our Dirichlet boundary condition was not homogeneous then this would not be the case. We use the notation

$$\langle u, v \rangle = \int_\Omega uv \, \mathrm{d}\Omega \tag{2.5}$$

throughout. Multiplying (2.1) by a test function $v \in V$ we obtain

$$\langle u_t, v \rangle = \langle \nabla^2 u, v \rangle + \langle g, v \rangle. \tag{2.6}$$

By integrating by parts we obtain

$$\langle u_t, v\rangle = -\langle \nabla u, \nabla v\rangle + \langle g, v\rangle, \tag{2.7}$$

where the boundary term has vanished due to the choice of the test space V. Hence we have the weak formulation

$$\text{Find } u \in U \text{ s.t } \langle u_t, v\rangle = -\langle \nabla u, \nabla v\rangle + \langle g, v\rangle \ \forall v \in V. \tag{2.8}$$

Now in order for us to convert this problem into a problem that can be dealt with computationally, we must restrict U to a finite dimensional subspace $U^h \subset U$. This is facilitated by discretising our domain into elements. Hence we have converted the weak form into the Galerkin approximation

$$\text{Find } u \in U^h \text{ s.t } \langle u_t, v\rangle = -\langle \nabla u, \nabla v\rangle + \langle g, v\rangle \ \forall v \in V^h. \tag{2.9}$$

We select

$$U^h = \{\phi \in C^0 : \phi \text{ is linear when restricted to an element}\}. \tag{2.10}$$

A convenient basis for the space of piecewise linear functions ϕ can be uniquely determined by the property $\phi_i(\mathbf{x}_j) = \delta_{ij}$, $i, j \in [1, n]$, where n is the number of nodes in the spatial discretisation of the domain. We can write u and v in terms of these piecewise linear functions:

$$u = \sum_{j=1}^{n} u_j(t)\phi_j, \quad v = \sum_{j=1}^{n} v_j(t)\phi_j. \tag{2.11}$$

By substituting (2.11) into (2.9) we obtain

$$\sum_{j=1}^{n} \langle \phi_i, \phi_j\rangle \frac{\mathrm{d}u_j(t)}{\mathrm{d}t} = -\sum_{j=1}^{n} \langle \nabla\phi_i, \nabla\phi_j\rangle u_j(t) + \langle g, \phi_i\rangle, \ i \in [1, n]. \tag{2.12}$$

By letting M, K and $\mathbf{f}$ be such that

$$M_{i,j} = \langle \phi_i, \phi_j\rangle, K_{i,j} = \langle \nabla\phi_i, \nabla\phi_j\rangle, f_i = \langle g, \phi_i\rangle, \quad i, j \in [1, n], \tag{2.13}$$

we obtain

$$M\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = -K\mathbf{u} + \mathbf{f}, \tag{2.14}$$

where M and K are the mass and stiffness matrices, respectively. For the sake of exposition we can write (2.14) as

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = q(\mathbf{u}) := M^{-1}(-K\mathbf{u} + \mathbf{f}). \tag{2.15}$$

The implicit Euler scheme for approximating first derivatives gives

$$\mathbf{u}_k \approx \mathbf{u}_{k-1} + \tau q(\mathbf{u}_k), \qquad k \in [1, \ell], \tag{2.16}$$

where ℓ is the number of time steps, $\mathbf{u}_0$ represents a projection of the initial data onto the finite element space and τ is the constant size of the time step. By substituting approximation (2.16) into (2.15) we obtain

$$(M + \tau K)\mathbf{u}_k = M\mathbf{u}_{k-1} + \tau\mathbf{f}, \qquad k \in [1, \ell]. \tag{2.17}$$

We could solve (2.17) in the conventional way, that is, we could apply either an iterative method or a direct method to (2.17) for each time step $k = 0, 1, ..., \ell$. Instead we construct a *monolithic system* by embedding all time steps in a single $n\ell$-by-$n\ell$ matrix to obtain the system

$$\mathcal{A}\mathbf{U} = \begin{bmatrix} A_0 & & & \\ A_1 & A_0 & & \\ & \ddots & \ddots & \\ & & A_1 & A_0 \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_\ell \end{bmatrix} = \begin{bmatrix} M\mathbf{u}_0 + \tau\mathbf{f} \\ \tau\mathbf{f} \\ \vdots \\ \tau\mathbf{f} \end{bmatrix} = \mathbf{b}, \tag{2.18}$$

where $A_0 = M + \tau K$ and $A_1 = -M$. The form of the matrix $\mathcal{A}$ in (2.18) is referred to as a block Toeplitz matrix. It was proved in [14] that by choosing the circulant preconditioner

$$\mathcal{P} = \begin{bmatrix} A_0 & & & A_1 \\ A_1 & A_0 & & \\ & \ddots & \ddots & \\ & & A_1 & A_0 \end{bmatrix}, \tag{2.19}$$

the matrix $\mathcal{P}^{-1}\mathcal{A}$ has eigenvalues mostly clustered at 1. The above problem will be used as our starting point to develop the all-at-once method in a more liberal framework. In [14] it is implied that the time steps must be equal, and we wish to investigate the robustness of the all-at-once formulations in the presence of non-uniform temporal discretisations. Taking a step in this direction, we can write the approximate solution of the problem (2.1)–(2.3) as

$$(M + \tau_k K)\mathbf{u}_k = M\mathbf{u}_{k-1} + \tau_k\mathbf{f}, \qquad k \in [1, \ell]. \tag{2.20}$$

This can be embedded in a monolithic system to give

$$\mathcal{B}\mathbf{U} = \begin{bmatrix} A_0^1 & & & \\ A_1 & A_0^2 & & \\ & \ddots & \ddots & \\ & & A_1 & A_0^\ell \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_\ell \end{bmatrix} = \begin{bmatrix} M\mathbf{u}_0 + \tau_1\mathbf{f} \\ \tau_2\mathbf{f} \\ \vdots \\ \tau_\ell\mathbf{f} \end{bmatrix} = \tilde{\mathbf{b}}, \tag{2.21}$$

where $A_0^i = M + \tau_i K$ and A_1 is as before. While $\mathcal{B}$ is no longer a block Toeplitz matrix, we proceed to precondition (2.21) with

$$\mathcal{Q} = \begin{bmatrix} A_0^1 & & & & A_1 \\ A_1 & A_0^2 & & & \\ & \ddots & \ddots & & \\ & & & A_1 & A_0^\ell \end{bmatrix}. \tag{2.22}$$

It would seem as though we have reached a dead end. This matrix does not have the structure that we once took advantage of, described in Section 1.3; however, we devised a method of approximating this preconditioner. First, note that $\mathcal{Q}$ can be written in the form

$$\mathcal{Q} = \mathcal{P} + \sigma \otimes K, \tag{2.23}$$

where σ is a diagonal matrix whose main diagonal consists of the perturbations about the uniform time discretisation and $\otimes$ is the Kronecker product (see [13]). To fully explain the above concept, let us consider a non-uniform temporal discretisation given by $\{t_i\}_{i=0}^{\ell}$. The time steps of this temporal discretisation are given by $\{\tau_i\}_{i=1}^{\ell} = \{t_i - t_{i-1}\}_{i=1}^{\ell}$. The time step $\tau = 1/\ell$ defines another temporal discretisation of the temporal domain $[t_0, t_\ell]$, which is uniform. The diagonal entries of σ are given by $\{\tau_i - \tau\}_{i=1}^{\ell}$. Examples of such non-uniform time discretisations are given by (2.64) and (4.3). So long as $||\sigma \otimes K|| < 1$, the inverse of (2.23) can be approximated in the following way:

$$\mathcal{Q}_i^{-1} = \sum_{j=0}^{i-1} (-1)^j \mathcal{P}^{-1}([\sigma \otimes K]\mathcal{P}^{-1})^j, \tag{2.24}$$

where i is some positive integer that we are free to choose. This type of approximation is commonly referred to as a Neumann series approximation.

2.2 The Wave Equation

In this section we derive all-at-once formulations for the wave equation. On reading a work from last year, we investigate multiple formulations in an attempt to overcome the issues that were present in [4]. The problem that we will be concentrating on is

given by

$$\frac{\partial^2 u}{\partial t^2} = \nabla^2 u + g \quad \text{on } \Omega \times (0, T], \tag{2.25}$$

$$u = 0 \qquad \text{for } \mathbf{x} \in \partial\Omega, \tag{2.26}$$

$$\frac{\partial u}{\partial t}(0, \mathbf{x}) = 0 \qquad \text{for } \mathbf{x} \in \Omega, \tag{2.27}$$

$$u(0, \mathbf{x}) = s(\mathbf{x}) \qquad \text{for } \mathbf{x} \in \Omega. \tag{2.28}$$

We consider both Neumann and Dirichlet boundary conditions in the numerical results section. As the solutions to the wave equation do not dissipate, there is no need to consider a non-uniform time-stepping strategy, although in the numerical results section we do consider various domains which, by design, must include non-uniform triangles in the triangulation. If we discretise in space using FEM we obtain

$$M\frac{\mathrm{d}^2\mathbf{u}}{\mathrm{d}t^2} = -K\mathbf{u} + M\mathbf{f}. \tag{2.29}$$

We approach the approximation of the time derivative in two ways. First we consider the central difference (CD) formula

$$\frac{\mathrm{d}^2\mathbf{u}_n}{\mathrm{d}t^2} \approx \frac{\mathbf{u}_{n-1} - 2\mathbf{u}_n + \mathbf{u}_{n+1}}{\tau^2}. \tag{2.30}$$

By substituting (2.30) into (2.29) we obtain

$$M\mathbf{u}_{k-1} + (\tau^2 K - 2M)\mathbf{u}_k + M\mathbf{u}_{k+1} = \tau^2 M\mathbf{f}, \qquad k \in [1, \ell]. \tag{2.31}$$

This can be cast into a monolithic system to give

$$\mathcal{C}_{CD}\mathbf{U} = \begin{bmatrix} A_0 & & & & & & \\ M & A_0 & M & & & & \\ & M & A_0 & M & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & M & A_0 & M & \\ & & & M & -2M & \tilde{A}_0 \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \\ \vdots \\ \mathbf{u}_{\ell-1} \\ \mathbf{u}_\ell \end{bmatrix} = \begin{bmatrix} A_0\mathbf{u}_0 \\ \tau^2 M\mathbf{f} \\ \tau^2 M\mathbf{f} \\ \vdots \\ \tau^2 M\mathbf{f} \\ \tau^2 M\mathbf{f} \end{bmatrix} = \mathbf{b}_{CD}, \tag{2.32}$$

where $A_0 = \tau^2 K - 2M$ and $\tilde{A}_0 = M + \tau^2 K$. Note that the first equation essentially says that $\mathbf{u}_0 = \mathbf{u}_1$, which encapsulates the homogeneous initial velocity profile. The modification to the last block row of (2.32) was necessary to obtain $\mathbf{u}_\ell$ correctly. The scheme that was used to approximate the second time derivative at t_ℓ is given as

(2.34). We can precondition this system with the preconditioner

$$\mathcal{R}_{CD} = \begin{bmatrix} A_0 & M & & & M \\ M & A_0 & M & & \\ & M & A_0 & \ddots & \\ & & \ddots & \ddots & M \\ M & & & M & A_0 \end{bmatrix}. \tag{2.33}$$

Note that in the statement of $\mathcal{R}_{CD}$ we have "replaced" the matrix M that was missing in the first block row of $\mathcal{C}_{CD}$ and we have ignored the complication that is present in the last block row of $\mathcal{C}_{CD}$. This was done so that the matrix $\mathcal{R}_{CD}$ has a block circulant structure which will be essential for an efficient parallel implementation, as will be discussed in Chapter 3. Anticipating that the success of the all-at-once method depends on a block lower diagonal structure, we also approximate the second time derivative using the following backward difference formula

$$\frac{\mathrm{d}^2\mathbf{u}_n}{\mathrm{d}t^2} \approx \frac{\mathbf{u}_{n-2} - 2\mathbf{u}_{n-1} + \mathbf{u}_n}{\tau^2}, \tag{2.34}$$

which will be referred to as BD2. Substituting (2.34) into (2.29) and rearranging we obtain

$$Mu_{k-2} - 2Mu_{k-1} + (M + \tau^2 K)u_k = \tau^2 M\mathbf{f}, \qquad k \in [2, \ell]. \tag{2.35}$$

As in the previous instance, this too can be compiled into a monolithic system

$$\mathcal{C}_{BD2}\mathbf{U} = \begin{bmatrix} A_0 & & & & \\ A_1 & A_0 & & & \\ A_2 & A_1 & A_0 & & \\ & \ddots & \ddots & \ddots & \\ & & A_2 & A_1 & A_0 \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \\ \vdots \\ \mathbf{u}_\ell \end{bmatrix} = \begin{bmatrix} A_0\mathbf{u}_0 \\ -M\mathbf{u}_0 + \tau^2 M\mathbf{f} \\ \tau^2 M\mathbf{f} \\ \vdots \\ \tau^2 M\mathbf{f} \end{bmatrix} = \mathbf{b}_{BD2}, \tag{2.36}$$

where $A_0 = M + \tau^2 K, A_1 = -2M$ and $A_2 = M$. We will precondition this system with the preconditioner

$$\mathcal{R}_{BD2} = \begin{bmatrix} A_0 & & & A_2 & A_1 \\ A_1 & A_0 & & & A_2 \\ A_2 & A_1 & A_0 & & \\ & \ddots & \ddots & \ddots & \\ & & A_2 & A_1 & A_0 \end{bmatrix}. \tag{2.37}$$

If it transpires that the CD formulation fails in some way, then we have to resort to BD2 which is a first order method, as opposed to CD which is a second order

method. To rectify this we consider approximating the second time derivative using a second order accurate backward difference formula

$$\frac{\mathrm{d}^2\mathbf{u}_n}{\mathrm{d}t^2} \approx \frac{-\mathbf{u}_{n-3} + 4\mathbf{u}_{n-2} - 5\mathbf{u}_{n-1} + 2\mathbf{u}_n}{\tau^2}, \tag{2.38}$$

which we will refer to as BD4. Substituting (2.38) into (2.29) we obtain

$$(2M + \tau^2 K)\mathbf{u}_k - 5M\mathbf{u}_{k-1} + 4M\mathbf{u}_{k-2} - M\mathbf{u}_{k-3} = \tau^2 M\mathbf{f}, \qquad k \in [3, \ell]. \tag{2.39}$$

We have to obtain the approximate solutions at the first few time steps using lower order methods, which will remove the Toeplitz structure of the matrix. That is, the matrix now has the following structure:

$$\mathcal{C}_{BD4} = \begin{bmatrix} B & & & & & & & & \\ C & B & & & & & & & \\ A_2 & A_1 & A_0 & & & & & & \\ A_3 & A_2 & A_1 & A_0 & & & & & \\ & A_3 & A_2 & A_1 & A_0 & & & & \\ & & \ddots & \ddots & \ddots & \ddots & & & \\ & & & & \ddots & \ddots & \ddots & \ddots & \\ & & & & & A_3 & A_2 & A_1 & A_0 \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \\ \mathbf{u}_4 \\ \mathbf{u}_5 \\ \vdots \\ \vdots \\ \mathbf{u}_\ell \end{bmatrix} = \begin{bmatrix} B\mathbf{u}_0 \\ -M\mathbf{u}_0 + \tau^2 M\mathbf{f} \\ M\mathbf{u}_0 + \tau^2 M\mathbf{f} \\ \tau^2 M\mathbf{f} \\ \tau^2 M\mathbf{f} \\ \vdots \\ \vdots \\ \tau^2 M\mathbf{f} \end{bmatrix} = \mathbf{b}_{BD4}, \tag{2.40}$$

where $B = M + \tau^2 K, C = -2M, A_0 = 2M + \tau^2 K, A_1 = -5M, A_2 = 4M, A_3 = -M$. We have used BD2 to obtain the approximate solution $\mathbf{u}_2$. We precondition this version of the wave equation system in a way similar to that of the preconditioning of the CD formulation. Previously we wrapped the Toeplitz system around to form a circulant. In this instance, we do not have a Toeplitz structure to begin with. The natural thing to do would be to pretend that $\mathcal{C}_{BD4}$ is a block Toeplitz structure and wrap it around to form the circulant

$$\mathcal{R}_{BD4} = \begin{bmatrix} A_0 & & & & & & A_3 & A_2 & A_1 \\ A_1 & A_0 & & & & & & A_3 & A_2 \\ A_2 & A_1 & A_0 & & & & & & A_3 \\ A_3 & A_2 & A_1 & A_0 & & & & & \\ & A_3 & A_2 & A_1 & A_0 & & & & \\ & & \ddots & \ddots & \ddots & \ddots & & & \\ & & & & \ddots & \ddots & \ddots & \ddots & \\ & & & & & A_3 & A_2 & A_1 & A_0 \end{bmatrix}. \tag{2.41}$$

2.3 The Convection-Diffusion Equation

The convection-diffusion equation can be stated as

$$\frac{\partial u}{\partial t} + \mathbf{V} \cdot \nabla u = \kappa \left(\nabla^2 u + \frac{f}{\tilde{k}} \right) \quad \text{on } \Omega \times (0, T], \tag{2.42}$$

$$u = 0 \qquad \text{for } \mathbf{x} \in \partial\Omega, \tag{2.43}$$

$$u(0, \mathbf{x}) = s(\mathbf{x}) \qquad \text{for } \mathbf{x} \in \Omega, \tag{2.44}$$

where $\mathbf{V}$ is the constant velocity vector, κ is the thermal diffusivity, $\tilde{k}$ is the thermal conductivity and f is a source function. By discretising in space we obtain

$$M \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} + L\mathbf{u} = \kappa(-K\mathbf{u} + \mathbf{d}), \tag{2.45}$$

where L is the convection matrix whose entries are given by

$$L_{i,j} = \langle \phi_i, \mathbf{V} \cdot \nabla \phi_j \rangle, \quad i, j \in [1, n]. \tag{2.46}$$

The entries of the vector $\mathbf{d}$ are given by

$$d_j = \frac{1}{\tilde{k}} \sum_{i=1}^{n} M_{i,j} f_j, \quad j \in [1, n], \tag{2.47}$$

where

$$f_j = \langle f, \phi_j \rangle, \quad j \in [1, n], \tag{2.48}$$

and n is the number of nodes in the spatial discretisation. Discretising in time using the implicit Euler scheme we obtain

$$(M + \tau L + \tau\kappa K)\mathbf{u}_k - M\mathbf{u}_{k-1} = \tau\kappa\mathbf{d}, \qquad k \in [1, \ell]. \tag{2.49}$$

We can cast this sequence into a monolithic system to obtain

$$\mathcal{D}\mathbf{U} = \begin{bmatrix} A_0 & & & \\ A_1 & A_0 & & \\ & \ddots & \ddots & \\ & & A_1 & A_0 \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_\ell \end{bmatrix} = \begin{bmatrix} M\mathbf{u}_0 + \tau\mathbf{d} \\ \tau\mathbf{d} \\ \vdots \\ \tau\mathbf{d} \end{bmatrix} = \mathbf{c}, \tag{2.50}$$

where $A_0 = M + \tau L + \tau\kappa K$ and $A_1 = -M$. This system can then be preconditioned with the matrix

$$\mathcal{S} = \begin{bmatrix} A_0 & & & A_1 \\ A_1 & A_0 & & \\ & \ddots & \ddots & \\ & & A_1 & A_0 \end{bmatrix}. \tag{2.51}$$

The introduction of the convection term adds a hyperbolic component to the heat equation problem. How this affects the performance of the all-at-once method, compared to the heat equation, will be of particular interest.

2.4 Extension to Higher Order in Time Methods

In this section we briefly describe how we would formulate the all-at-once method if we chose to utilise higher order methods in the approximation of the time derivative. The heat equation is considered here and its extension to the wave equation is trivial. Consider

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = q(\mathbf{u}), \tag{2.52}$$

where q is defined in (2.15). A general s-step method is given by

$$\sum_{m=0}^{s} a_m \mathbf{u}_{n+m} = \sum_{m=0}^{s} \tau b_m q(t_{n+m}, \mathbf{u}_{n+m}), \tag{2.53}$$

where a_i, b_i are coefficients that we must specify. We stipulate that $b_s \neq 0$, since we want the method to be implicit in order to address numerical stability concerns. By substituting our expression for q into (2.53) we obtain

$$\sum_{m=0}^{s} (a_m M + \tau b_m K)\mathbf{u}_{n+m} = \sum_{m=0}^{s} \tau b_m \mathbf{f}. \tag{2.54}$$

For p-step methods, we have to be mindful of the fact that the approximations to the initial $p-1$ time steps will have to be approximated by sub-p-step methods. We encountered this in a specific case when applying BD4 to the wave equation. In the more general case the system has the form

$$\mathcal{E} = \begin{bmatrix} B_1^1 & & & & & & & & \\ \vdots & \ddots & & & & & & & \\ B_1^{p-1} & \dots & B_{p-1}^{p-1} & & & & & & \\ A_{p-1} & \dots & \dots & A_0 & & & & & \\ A_p & \dots & \dots & A_1 & A_0 & & & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & & & A_p & \dots & \dots & A_1 & A_0 \end{bmatrix}. \tag{2.55}$$

However, for our preconditioner to have the properties necessary for efficient parallel implementation, we have to impose that the preconditioner has a block circulant structure. That is, the preconditioner takes the form

$$\mathcal{T} = \begin{bmatrix} A_0 & & & & & & & & \\ \vdots & \ddots & & & & & & & \\ \vdots & & \ddots & & & & & & \\ A_{p-1} & \dots & \dots & A_0 & & & & & \\ A_p & \dots & \dots & A_1 & A_0 & & & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & & & A_p & \dots & \dots & A_1 & A_0 \end{bmatrix}. \tag{2.56}$$

2.5 Symmetrisation of the Monolithic System

In some cases we can actually symmetrise the monolithic systems that we have encountered thus far, which would allow us to apply MINRES instead of GMRES. The advantage of this would be to reduce the cost per iteration of applying an iterative method; it would also allow for better convergence guarantees (see [14]). First we introduce the Hankel matrix

$$Y = \begin{bmatrix} & & & & \mathbb{I}_n \\ & & & \mathbb{I}_n & \\ & & \iddots & & \\ & \mathbb{I}_n & & & \\ \mathbb{I}_n & & & & \end{bmatrix}. \tag{2.57}$$

Multiplying the block Toeplitz matrix $\mathcal{A}$ on the left by Y results in a symmetric matrix $Y\mathcal{A}$. Therefore we can apply MINRES to solve the system $Y\mathcal{A}\mathbf{U} = Y\mathbf{b}$. This leads us to the idea of preconditioning. For us to utilise preconditioned MINRES properly we require that the preconditioner is symmetric *and* positive definite. To see this consider $\mathcal{P}$ positive definite, with diagonalisation $\mathcal{P} = X\Lambda X^T$. $\mathcal{P}$ then has a well-defined and unique positive square root

$$\mathcal{P}^{\frac{1}{2}} = X\Lambda^{\frac{1}{2}}X^T. \tag{2.58}$$

The matrix square root is discussed in detail in [9]. Here X diagonalises $\mathcal{P}$ and $\Lambda^{\frac{1}{2}}$ is the matrix that results from taking the positive entry-wise square root of the diagonal

matrix Λ, which contains the eigenvalues of $\mathcal{P}$ along the main diagonal. Therefore we can write

$$\mathcal{P}^{-1}\mathcal{A}\mathbf{x} = \mathcal{P}\mathbf{b} \tag{2.59}$$

$$\implies (\mathcal{P}^{-\frac{1}{2}}\mathcal{A}\mathcal{P}^{-\frac{1}{2}})\mathcal{P}^{\frac{1}{2}}\mathbf{x} = \mathcal{P}^{-\frac{1}{2}}\mathbf{b} \tag{2.60}$$

$$\implies (\mathcal{P}^{-\frac{1}{2}}\mathcal{A}\mathcal{P}^{-\frac{1}{2}})\tilde{\mathbf{x}} = \tilde{\mathbf{b}}. \tag{2.61}$$

The matrix appearing on the left-hand side of (2.61) is real and symmetric. The preconditioner suggested in [22] is the so-called absolute value preconditioner

$$|\mathcal{P}| = (\mathcal{P}^T\mathcal{P})^{\frac{1}{2}}. \tag{2.62}$$

This is easily computed via the FFT diagonalisation when $\mathcal{P}$ is circulant, as discussed in Chapter 1. Since $|\mathcal{P}|$ is symmetric positive definite it can be used in preconditioned MINRES to solve

$$Y\mathcal{A}\mathbf{U} = Y\mathbf{b}. \tag{2.63}$$

Now we present a novel idea that enables us to symmetrise (2.21), so long as we choose a time discretisation that is symmetric about the centre of the time interval. That is, the system is no longer a block Toeplitz system. The situation in which this might be useful is when we are interested in the endpoints of the time interval more than the intermediate times. For instance, most of the "action" in the solution of the heat equation occurs near the start of the time interval and then dissipates as time progresses. While we would also have a cluster of points at the end of the interval, the fact that we can symmetrise the system and use MINRES instead of GMRES allows a reduction in the overall cost of the method. Consider the time discretisation of the interval $[0, 1]$ as

$$t_k = \frac{1}{2}\cos\left(\frac{(\ell - k)}{\ell}\pi\right) + \frac{1}{2}, \qquad k = 0, 1, ..., \ell. \tag{2.64}$$

These nodes are the so-called Chebyshev nodes, which are an example of a "symmetric" non-uniform discretisation. Hence we have a symmetric system of equations

$$Y\mathcal{B}\mathbf{U} = Y\mathbf{b}. \tag{2.65}$$

As before, we consider the absolute value preconditioner $|\mathcal{Q}| = (\mathcal{Q}^*\mathcal{Q})^{\frac{1}{2}}$. If we evaluate this expression we are left with

$$|\mathcal{Q}| = [\mathcal{P}^*\mathcal{P} + (\sigma \otimes K)^*\mathcal{P} + \mathcal{P}^*(\sigma \otimes K) + (\sigma \otimes K)^*(\sigma \otimes K)]^{\frac{1}{2}} \tag{2.66}$$

$$\approx |\mathcal{P}| + (\sigma \otimes K). \tag{2.67}$$

We can now apply the idea that we developed in the general case to approximate the inverse of $\mathcal{P}$. That is, we can approximate the inverse of (2.67) using the Neumann series approximation to obtain

$$|\mathcal{Q}|_i^{-1} \approx \sum_{j=0}^{i-1} (-1)^j |\mathcal{P}|^{-1} ([\sigma \otimes K]|\mathcal{P}|^{-1})^j. \tag{2.68}$$

The drawback of this method is that we cannot guarantee that $|\mathcal{Q}|_i^{-1}$ is symmetric positive definite. On the assumption that we do have positive definiteness we can apply MINRES in the usual way to the system $|\mathcal{Q}|_i^{-1} Y\mathcal{B} = Y\mathbf{b}$. The system resulting from the discretisation of the wave equation can be symmetrised in an almost identical way.

2.6 Theoretical Guarantees

The following theorem is a modification of Theorem 1 in [14]. It is modified in the sense that it removes the restriction on the use of uniform time steps. We employ the notation $E_i = e_i \otimes \mathbb{I}_n$, where e_i is the i-th column of $\mathbb{I}_\ell$.

Theorem 2.6.1. *The preconditioned system $\mathcal{Q}^{-1}\mathcal{B}$ is a rank n perturbation of the identity. Furthermore, $\mathcal{Q}^{-1}\mathcal{B}$ has $(\ell - 1)n$ eigenvalues equal to 1.*

Proof. By noting that $\mathcal{Q} = \mathcal{B} + E_1 A_1 E_\ell^T$, and then by using the Sherman-Morrison-Woodbury (SMW) formula (see [6]) we obtain

$$\mathcal{Q}^{-1} = (\mathcal{B} + E_1 A_1 E_\ell^T)^{-1} = \mathcal{B}^{-1} - \mathcal{B}^{-1} E_1 (A_1 + E_\ell^T \mathcal{B}^{-1} E_1)^{-1} E_\ell^T \mathcal{B}^{-1}. \tag{2.69}$$

Hence

$$\mathcal{Q}^{-1}\mathcal{B} = \mathbb{I}_{n\ell} - \mathcal{B}^{-1} E_1 (A_1 + E_\ell^T \mathcal{B}^{-1} E_1)^{-1} E_\ell^T. \tag{2.70}$$

Since $\mathcal{B}^{-1} E_1 (A_1 + E_\ell^T \mathcal{B}^{-1} E_1)^{-1} E_\ell^T$ is of rank n, (2.70) is a rank n perturbation of the identity. The inverse of a block lower triangular matrix is also block lower triangular. In fact, in this case it can be shown by induction that the block entries of the inverse of $\mathcal{B}$ are given by

$$(\mathcal{B}^{-1})_{i,j} = \begin{cases} (-1)^{i+1} \prod_{k=i}^{j} [(A_0^k)^{-1} A_1] A_1^{-1}, & j \leq i, \\ 0, & j > i. \end{cases} \tag{2.71}$$

Letting $Z = A_1^{-1} + E_\ell^T \mathcal{B}^{-1} E_1$ we obtain

$$\mathcal{Q}^{-1}\mathcal{B} = \begin{bmatrix} \mathbb{I}_n & & & -(\mathcal{B}^{-1})_{1,1} Z^{-1} \\ & \ddots & & \vdots \\ & & \ddots & \vdots \\ & & \mathbb{I}_n & -(\mathcal{B}^{-1})_{\ell-1,1} Z^{-1} \\ & & & \mathbb{I}_n - (\mathcal{B}^{-1})_{\ell,1} Z^{-1} \end{bmatrix}. \tag{2.72}$$

Hence $\mathcal{Q}^{-1}\mathcal{B}$ has $n(\ell-1)$ eigenvalues equal to 1. □

As we are aware from [7], GMRES can have any non-decreasing convergence curve regardless of the eigenvalues. However, in practice, if the eigenvalues of a system are clustered around 1 then GMRES will usually converge to a solution in a small number of iterations. While the above theorem gives us most of the eigenvalues of the system associated with the heat equation on a non-uniform temporal domain, the following theorem sheds some light on the remaining eigenvalues. Again, this theorem is a modified version of a theorem appearing in [14], namely Theorem 2.

Theorem 2.6.2. *If $A_1^{-1}A_0^k$ has eigenvalue equal to μ_k, then $\mu_k \neq \pm 1$ and*

$$\lambda = \frac{\prod_{k=1}^{\ell} \mu_k}{(-1)^{\ell+1} + \prod_{k=1}^{\ell} \mu_k}$$

is an eigenvalue of $\mathbb{I}_n - (\mathcal{B}^{-1})_{\ell,1} Z^{-1}$.

Proof. First, note that

$$\mathbb{I}_n - (\mathcal{B}^{-1})_{\ell,1} Z^{-1} = \mathbb{I}_n - \left(\mathbb{I}_n + A_1^{-1}(-1)^{\ell+1} \left[\prod_{k=\ell}^{1} [(A_0^k)^{-1} A_1] A_1^{-1} \right]^{-1} \right)^{-1} \tag{2.73}$$

$$= \mathbb{I}_n - \left(\mathbb{I}_n + (-1)^{\ell+1} A_1^{-1} \left[\prod_{k=1}^{\ell-1} A_0^k A_1^{-1} \right] A_0^\ell \right)^{-1} \tag{2.74}$$

$$= \mathbb{I}_n - \left(\mathbb{I}_n + (-1)^{\ell+1} \left[\prod_{k=1}^{\ell} A_1^{-1} A_0^k \right] \right)^{-1}, \tag{2.75}$$

where $A_1^{-1}A_0^k = -(\mathbb{I} + \tau_k M^{-1} K)$ and both M and K are symmetric positive definite. Thus if μ_k is an eigenvalue of $A_1^{-1}A_0^k$ then $\mu_k \neq \pm 1$. In fact, $\mu_k > 1$. Hence there

exists a non-zero vector $\mathbf{x} \in \mathbb{R}^n$ such that

$$A_1^{-1}A_0^k\mathbf{x} = \mu_k\mathbf{x} \tag{2.76}$$

$$\prod_{k=1}^{\ell} A_1^{-1}A_0^k\mathbf{x} = \prod_{k=1}^{\ell} \mu_k\mathbf{x} \tag{2.77}$$

$$\left(\mathbb{I}_n + (-1)^{\ell+1}\left[\prod_{k=1}^{\ell} A_1^{-1}A_0^k\right]\right)^{-1}\mathbf{x} = \frac{1}{1+(-1)^{\ell+1}\prod_{k=1}^{\ell}\mu_k}\mathbf{x} \tag{2.78}$$

$$\left[\mathbb{I}_n - \left(\mathbb{I}_n + (-1)^{\ell+1}\left[\prod_{k=1}^{\ell} A_1^{-1}A_0^k\right]\right)^{-1}\right]\mathbf{x} = \frac{\prod_{k=1}^{\ell}\mu_k}{(-1)^{\ell+1}+\prod_{k=1}^{\ell}\mu_k}\mathbf{x}. \tag{2.79}$$

□

In [14], the original versions of the above theorems were incredibly useful. The original theorems will be of great interest to us here as they guarantee that GMRES will converge within $n+1$ iterations. Theorem 2.6.2 and Theorem 2.6.3 are not quite as useful here, as we have to approximate the preconditioner. However, if the Neumann approximation is sufficiently good then these theorems will shed some light on the convergence of GMRES when applied to the system associated with the heat equation on a non-uniform temporal domain.

The unpreconditioned systems that we considered above are mostly non-symmetric. That is, $\mathcal{A}, \mathcal{C}_{BD2}, \mathcal{C}_{BD4}$ are non-symmetric and $\mathcal{C}_{CD}$ is symmetric. However, we will be interested in knowing whether or not the preconditioned analogues of these matrices are symmetric, or at least how close they are to being symmetric. This will be discussed further in Chapter 4. The following theorem, which is a slight modification of Theorem 3 of [14], shows that the system associated with the heat equation using a non-uniform temporal discretisation is diagonalisable.

Theorem 2.6.3. *$\mathcal{Q}^{-1}\mathcal{B}$ is diagonalisable.*

Proof. From Theorem 2.6.2:

$$(\mathcal{B}^{-1})_{l,1}Z^{-1} = \left(\mathbb{I}_n + (-1)^{\ell+1}\left[\prod_{k=1}^{\ell} A_1^{-1}A_0^k\right]\right)^{-1} \tag{2.80}$$

$$= \left(\mathbb{I}_n - \prod_{k=1}^{\ell}(\mathbb{I}_n + \tau_k M^{-1}K)\right)^{-1}. \tag{2.81}$$

This matrix is diagonalisable and has real, negative eigenvalues. To see this, note that M^{-1} is symmetric positive definite, hence $M^{\frac{1}{2}}$ and $M^{-\frac{1}{2}}$ are well-defined. By writing

$$M^{-\frac{1}{2}}KM^{-\frac{1}{2}} = M^{\frac{1}{2}}MKM^{-\frac{1}{2}} \tag{2.82}$$

we can see that $M^{-1}K$ is similar to a symmetric positive definite matrix. This shows that $M^{-1}K$ is diagonalisable and has positive eigenvalues. The diagonalisability and negative definiteness of (2.81) follow trivially from this observation. Therefore $\mathbb{I}_n - (\mathcal{B}^{-1})_{\ell,1}Z^{-1}$ is diagonalisable and has eigenvalues that are real and larger than 1. Let $\mathbb{I}_n - (\mathcal{B}^{-1})_{\ell,1}Z^{-1}$ have diagonalisation VDV^{-1}. As a consequence, $\mathcal{Q}^{-1}\mathcal{B}$ has the diagonalisation $\mathcal{V}\mathcal{D}\mathcal{V}^{-1}$ where

$$\mathcal{V} = \begin{bmatrix} \mathbb{I}_n & & & V_1 \\ & \ddots & & \\ & & \mathbb{I}_n & V_{\ell-1} \\ & & & V \end{bmatrix}, \mathcal{D} = \begin{bmatrix} \mathbb{I}_n & & & \\ & \ddots & & \\ & & \mathbb{I}_n & \\ & & & D \end{bmatrix}, \tag{2.83}$$

and $V_i = (\mathcal{B}^{-1})_{i,1}Z^{-1}V(D - \mathbb{I}_n)^{-1}$. □

The above theorems help us establish a *rough* bound on the number of GMRES iterations required to solve the system associated with the heat equation when a non-uniform temporal domain is used. In the instance that we use a uniformly discretised temporal domain, then, from Theorem 1 of [14], we have an exact bound on the number of iterations required to solve the system associated with the heat equation. We would like to establish a similar bound on the system associated with the wave equation.

Theorem 2.6.4. *The preconditioned system $\mathcal{R}_{BD2}^{-1}\mathcal{C}_{BD2}$ is a rank $2n$ perturbation of the identity. Furthermore, $\mathcal{R}_{BD2}^{-1}\mathcal{C}_{BD2}$ has $(\ell-2)n$ eigenvalues equal to 1.*

Proof. In the interest of clarity we will write $\mathcal{C} = \mathcal{C}_{BD2}$ and $\mathcal{R} = \mathcal{R}_{BD2}$ throughout this proof. Note that we can write the preconditioner as

$$\mathcal{R} = \mathcal{C} + E_1A_2E_{\ell-1}^T + E_2A_2E_\ell^T + E_1A_1E_\ell^T. \tag{2.84}$$

Using the SMW formula several times we obtain, after significant rearrangement, the expression

$$\mathcal{R}^{-1}\mathcal{C} = \mathbb{I}_{nl} - Y_6 - Y_7 - Y_8, \tag{2.85}$$

where

$$Z = A_2^{-1} + E_{\ell-1}^T \mathcal{C}^{-1} E_1, \tag{2.86}$$

$$X = \mathcal{C} + E_1 A_2 E_{\ell-1}^T + E_2 A_2 E_\ell^T + E_1 A_1 E_\ell^T, \tag{2.87}$$

$$Y_1 = \mathcal{C}^{-1} - \mathcal{C}^{-1} E_1 Z^{-1} \mathcal{C}^{-1}, \tag{2.88}$$

$$Y_2 = \mathcal{C}^{-1} E_1 Z^{-1} E_{\ell-1}^T \mathcal{C}^{-1}, \tag{2.89}$$

$$Y_3 = A_1 + E_\ell^T X E_1, \tag{2.90}$$

$$Y_4 = A_2^{-1} + E_\ell^T Y_1 E_2, \tag{2.91}$$

$$Y_5 = A_2^{-1} + E_\ell^T Y_2 E_2, \tag{2.92}$$

$$Y_6 = \mathcal{C}^{-1} E_1 Z^{-1} E_{\ell-1}^T, \tag{2.93}$$

$$Y_7 = Y_1 E_2 Y_4^{-1} E_\ell^T Y_1 \mathcal{C}, \tag{2.94}$$

$$Y_8 = [Y_1 - Y_1 E_2 Y_5^{-1} E_\ell^T Y_1](E_1 Y_3^{-1} E_\ell^T)[\mathbb{I}_{nl} - \mathcal{C}^{-1} E_1 Z^{-1} E_{\ell-1}^T - Y_1 E_2 Y_4^{-1} E_\ell^T Y_1 \mathcal{C}]. \tag{2.95}$$

See Appendix A for further details. The matrices Y_6, Y_7 and Y_8 are block matrices which contain zero blocks except in the last two block columns. That is, they take the form

$$\begin{bmatrix} 0 & \dots & 0 & \times & \times \\ 0 & \dots & 0 & \vdots & \vdots \\ \vdots & \dots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & \times & \times \\ 0 & \dots & 0 & \times & \times \end{bmatrix}, \tag{2.96}$$

where 0 represents a block of zeros and $\times$ represents a block of arbitrary entries. If we evaluate the expression (2.85) then we can see that it takes the form

$$\mathcal{R}^{-1}\mathcal{C} = \begin{bmatrix} \mathbb{I}_n & & & D_1 & F_1 \\ & \ddots & & \vdots & \vdots \\ & & \mathbb{I}_n & D_{\ell-2} & F_{\ell-2} \\ & & & D_{\ell-1} & F_{\ell-1} \\ & & & D_\ell & F_\ell \end{bmatrix}, \tag{2.97}$$

where D_i and F_i are matrices. That is, (2.85) is at most a rank $2n$ perturbation of the identity. To find the eigenvalues of $\mathcal{R}^{-1}\mathcal{C}$ we consider the characteristic polynomial

$$\det(\mathcal{R}^{-1}\mathcal{C} - \lambda \mathbb{I}_{nl}) = \det \begin{bmatrix} (1-\lambda)\mathbb{I}_{n(\ell-2)} & \mathcal{F} \\ 0 & \mathcal{G} - \lambda \mathbb{I}_{2n} \end{bmatrix} \tag{2.98}$$

$$= (1-\lambda)^{n(\ell-2)} \det(\mathcal{G} - \lambda \mathbb{I}_{2n}), \tag{2.99}$$

where

$$\mathcal{G} = \begin{bmatrix} D_{\ell-1} & F_{\ell-1} \\ D_\ell & F_\ell \end{bmatrix}, \qquad \mathcal{F} = \begin{bmatrix} D_1 & F_1 \\ \vdots & \vdots \\ D_{\ell-2} & F_{\ell-2} \end{bmatrix}. \tag{2.100}$$

Hence we can see that the preconditioned system has at least $n(\ell - 2)$ eigenvalues equal to 1. □

This result tells us that the BD2 formulation of the wave equation will converge within $2n + 1$ GMRES iterations.

Chapter 3

Parallel Implementation

Throughout our implementation we keep all matrices on the master process and broadcast them when necessary. Vectors, on the other hand, are defined on all processes. To understand the reasoning for this, consider the fact that our dense block U requires $\mathcal{O}(\ell^2)$ memory and our sparse blocks (linear combinations of M, K) each require $\mathcal{O}(n)$ memory. Since there are ℓ sparse blocks, the total memory cost is $\mathcal{O}(\ell^2 + n\ell)$. Further, since a vector requires $\mathcal{O}(n\ell)$ memory, if each one of p processes has a copy of the vector, then we are using $\mathcal{O}(n\ell p)$ memory. In the complexity analysis below, we consider the situation where ℓ processes are available. Using ℓ processes would significantly increase the memory requirements of our implementation with this memory management scheme. In our case, $p \ll \ell$ and so the vector storage cost almost matches that of the matrix storage cost. Even in the case where $p \sim \ell$, the reduction in communication cost that results from this type of memory management is likely to be significant.

3.1 The Implementation

In order to see how (2.19) can be applied in parallel, we follow [14] in writing (2.19) in the form

$$\mathcal{P} = \mathbb{I}_\ell \otimes A_0 + \Sigma \otimes A_1, \tag{3.1}$$

where

$$\Sigma = \begin{bmatrix} & & & & 1 \\ 1 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & 1 & \end{bmatrix} \in \mathbb{R}^{\ell \times \ell}, \tag{3.2}$$

and $\mathbb{I}_\ell$ is the $\ell \times \ell$ identity matrix. The key property of circulant matrices is that they can be diagonalised by a Fourier basis. That is, we can write $\Sigma = U\Lambda U^*$, where $U_{k,j} = \mathrm{e}^{((k-1)(j-1)\pi i)/\ell}/\sqrt{\ell}$ and the diagonal entries of Λ are the roots of unity for the "downshift" matrix Σ. Hence, again following [14], (3.1) can be written as

$$\mathcal{P} = (U \otimes \mathbb{I}_n)[\mathbb{I}_\ell \otimes A_0 + \Lambda \otimes A_1](U^* \otimes \mathbb{I}_n). \tag{3.3}$$

Inverting $\mathcal{P}$ we obtain

$$\mathcal{P}^{-1} = (U \otimes \mathbb{I}_n)[\mathbb{I}_\ell \otimes A_0 + \Lambda \otimes A_1]^{-1}(U^* \otimes \mathbb{I}_n). \tag{3.4}$$

In code the application of the preconditioner is given as follows:

```
void ApplyPreconditioner(int mynode,int totalnodes,int N, int L,
        std::complex<double>**U,std::complex<double>**Ut,
        std::vector<std::complex<double>*>& Wblks,
        std::complex<double>* q,std::complex<double>* x)
{
    // We need to make a copy of q and conduct calculations on that.
    std::complex<double>*b = new std::complex<double>[N*L];
    SetEqualTo(mynode,totalnodes,N,L,q,b,1.0);

    // The subroutines used to calculate the
    // application of the preconditioner.
    MultiplicationByUKronIdentity(mynode,totalnodes,N,L,Ut,b,x); /*      U^*      */
    BlockTriDiagSolve_Thomas(mynode,totalnodes,N,L,Wblks,b,x);   /* ( ... )^{-1} */
    MultiplicationByUKronIdentity(mynode,totalnodes,N,L,U,b,x);  /*      U        */

    delete [] b;
}
```

The application of $(U \otimes \mathbb{I}_n)$ to a vector can be carried out in parallel. To see this, consider the explicit representation of $(U \otimes \mathbb{I}_n)$ given by

$$(U \otimes \mathbb{I}_n) = \left[\begin{array}{c|c|c} U_{11}\mathbb{I}_n & \dots & U_{1\ell}\mathbb{I}_n \\ \hline \vdots & \ddots & \vdots \\ \hline U_{\ell 1}\mathbb{I}_n & \dots & U_{\ell\ell}\mathbb{I}_n \end{array}\right]. \tag{3.5}$$

First, we broadcast each row of U to a process. Then we can evaluate

$$\mathbf{y}_i = \left[\begin{array}{c|c|c} U_{i1}\mathbb{I}_n & \dots & U_{i\ell}\mathbb{I}_n \end{array} \right] \begin{bmatrix} \mathbf{z}_1 \\ \vdots \\ \mathbf{z}_\ell \end{bmatrix} \tag{3.6}$$

on each process, where $\mathbf{z}_i \in \mathbb{R}^n$ is a chunk of the $n\ell$-vector $\mathbf{z}$ and i is an integer such that $i \in [1, \ell]$. This can be done in $\mathcal{O}(n\ell)$ operations. The local resultants of each of the calculations carried out on each process $\mathbf{y}_i$ can then be gathered to yield the final resultant of the calculation $(U \otimes \mathbb{I}_n)\mathbf{z}$. This part of the routine is given as follows:

```
void MultiplicationByUKronIdentity(int mynode, int numnodes,int N,int L,
        std::complex<double>**U,std::complex<double> *x,std::complex<double>* y)
{
    int i,j,k;
    int local_offset,rows_local;
    int *count;
    int *displacements;
    std::complex<double> prod;

    // The number of blocks each processor is dealt.
    blocks_local = L/numnodes;

    // The part of the output vector per process.
    std::complex<double> *temp = new std::complex<double>[N*rows_local];

    // Contains the rows of U that will be needed
    // by each process.
    std::complex<double>** Ulocal;

    // The offset.
    local_offset = mynode*rows_local;

    MPI_Status status;

    // Distribution of U across the processes.
    if(mynode ==0)
    {
        // This deals the matrix between processes 1 to numnodes-2.
        Ulocal = CreateMatrix(rows_local,L);
        for(i=0;i<blocks_local;i++)
        {
            Ulocal[i] = U[i];
        }
        for(i=1;i<numnodes;i++)
        {
            for(j=0;j<rows_local;j++)
            {
                // Sends rows_local rows to process i.
                MPI_Send(U[i*rows_local+j],L,MPI_DOUBLE_COMPLEX,
                        i,j,MPI_COMM_WORLD);
            }
```

```
        }
    }
    else
    {
        //This code allows other processes to obtain the chunks of data
        // sent by process 0.
        Ulocal = CreateMatrix(rows_local,L);
        for(i=0;i<rows_local;i++)
        {
                // Receives rows and stores them.
            MPI_Recv(Ulocal[i],L,MPI_DOUBLE_COMPLEX,0,i,
                MPI_COMM_WORLD,&status);
        }
    }

    // Carries out the "strip" multiplication.
    // This requires O(N*L*rows_local).
    for(i = 0;i<rows_local;i++)
    {
        for(k=0;k<L;k++)
        {
            prod = 0;
            for(j=0;j<N;j++)
            {
                prod += Ulocal[i][k]*x[k*N + j];
            }
            temp[i*N+j] = prod;
        }
    }

    // This command glues each temporary result (temp)
    // together and distributes the result among processes.
    MPI_Allgather(temp,N*rows_local,MPI_DOUBLE_COMPLEX,y,
        N*rows_local,MPI_DOUBLE_COMPLEX,MPI_COMM_WORLD);

    // Clean-up.
    delete [] temp;
    delete [] Ulocal;
}
```

The only other implementation-specific issue that we need to be concerned with is the inversion of the block tridiagonal matrix $\mathbb{I}_\ell \otimes A_0 + \Lambda \otimes A_1$. This can easily be carried out in parallel by assigning each tridiagonal block to a process and then applying the Thomas algorithm to each block, which would incur a cost of $\mathcal{O}(n)$ over ℓ processes. Therefore, the total complexity of this implementation is $\mathcal{O}(n\ell)$ over ℓ processes. The block-Thomas routine is given in code as

```
void BlockTriDiagSolve_Thomas(int mynode, int numnodes,int N,int L,
        std::vector<std::complex<double> *>&blocks,std::complex<double> *x,
        std::complex<double>* q)
{
    int i,j,k;
```

```
6       int local_offset,blocks_local;
7       int *count;
8       int *displacements;
9
10      // The number of blocks each processor is dealt.
11      blocks_local = L/numnodes;
12
13      // The part of the output vector per process.
14      std::complex<double> *z = new std::complex<double>[N*blocks_local];
15
16      // Container of local blocks.
17      std::vector<std::complex<double>*> localblocks(blocks_local);
18
19      // The offset.
20      local_offset = mynode*blocks_local;
21
22      MPI_Status status;
23
24      /* Distribute the blocks across the processes */
25      if(mynode ==0)
26      {
27          for(i=0;i<blocks_local;i++) localblocks[i] = CreateTDMatrixContiguous(N);
28          for(i=0;i<blocks_local;i++) localblocks[i] = blocks[i];
29          // This deals the matrix between processes 1 to numnodes -2
30          for(i=1;i<numnodes;i++)
31          {
32              for(j=0;j<blocks_local;j++)
33              {
34                  MPI_Send(blocks[i*blocks_local+j],3*N-2,
35                          MPI_DOUBLE_COMPLEX,i,j,MPI_COMM_WORLD);
36              }
37          }
38      }
39      else
40      {
41          /*This code allows other processes to obtain the chunks of data*/
42          /* sent by process 0. */
43          for(i=0;i<blocks_local;i++) localblocks[i] = CreateTDMatrixContiguous(N);
44          for(i=0;i<blocks_local;i++)
45          {
46              MPI_Recv(localblocks[i],3*N-2,MPI_DOUBLE_COMPLEX,
47                  0,i,MPI_COMM_WORLD,&status);
48          }
49      }
50
51
52      //ENTERING THE CALCULATION STAGE
53
54
55      for(k = 0;k<blocks_local;k++)
56      {
57          // Unpacking of contig. data structure.
58          std::complex<double> * a = new std::complex<double>[N];
59          std::complex<double> * am1 = new std::complex<double>[N-1];
```

```
60          std::complex<double> * ap1 = new std::complex<double>[N-1];
61
62          for(j=0;j<N-1;j++) am1[j] = localblocks[k][j];
63          for(j=0;j<N;j++) a[j] = localblocks[k][N-1 + j];
64          for(j=0;j<N-1;j++) ap1[j] = localblocks[k][2*N-1 + j];
65
66          // Entering the Thomas algorithm.
67          std::complex<double> *l,*u,*d,*y;
68          l = new std::complex<double>[N];
69          u = new std::complex<double>[N];
70          d = new std::complex<double>[N];
71          y = new std::complex<double>[N];
72
73          // LU
74          d[0] = a[0];
75          u[0] = ap1[0];
76          for(i=0;i<N-2;i++)
77          {
78              l[i]    = am1[i]/d[i];
79              d[i+1]  = a[i+1] - l[i]*u[i];
80              u[i+1]  = ap1[i+1];
81          }
82          l[N-2] = am1[N-2]/d[N-2];
83          d[N-1] = a[N-1] - l[N-2]*u[N-2];
84
85          // Forward substitution.
86          y[0] = q[(k+local_offset)*N];
87          for(i=1;i<N;i++)
88          {
89              y[i] = q[(k+local_offset)*N+i] - l[i-1]*y[i-1];
90          }
91
92          // Backward substitution.
93          z[k*N+N-1] = y[N-1]/d[N-1];
94          for(i=N-2;i>=0;i--)
95          {
96              z[k*N+i] = (y[i]-u[i]*z[k*N+i+1])/d[i];
97          }
98          delete [] l;
99          delete [] u;
100         delete [] d;
101         delete [] y;
102
103     }
104     // This command glues each temporary result (temp)
105     // together and distributes the result among processes.
106     MPI_Allgather(z,blocks_local*N,MPI_DOUBLE_COMPLEX,x,
107         N*blocks_local,MPI_DOUBLE_COMPLEX,MPI_COMM_WORLD);
108
109     delete [] z;
110
111 }
```

The parallel implementation of the BD2 formulation of the wave equation is very

similar. The preconditioner can now be written as

$$\mathcal{R}_{BD2} = \mathbb{I}_\ell \otimes A_0 + \Sigma \otimes A_1 + \Sigma^2 A_2. \tag{3.7}$$

Following the above formulation we can write

$$\mathcal{R}_{BD2}^{-1} = (U \otimes \mathbb{I}_n)[\mathbb{I}_\ell \otimes A_0 + \Lambda \otimes A_1 + \Lambda^2 \otimes A_2]^{-1}(U^* \otimes \mathbb{I}_n). \tag{3.8}$$

Analogous expressions for the BD4 formula and CD formula can be derived in a similar way. One thing to note is that the cost of applying the preconditioner has not changed. The formation of

$$\mathbb{I}_\ell \otimes A_0 + \Lambda \otimes A_1 + \Lambda^2 \otimes A_2 \tag{3.9}$$

occurs in the preamble of the implementation and is also a block tridiagonal matrix. The increase in computational cost of solving the wave equation system over the heat equation system comes from the application of $\mathcal{C}_{BD2}$ instead of $\mathcal{A}$: $\mathcal{C}_{BD2}$ is more dense than $\mathcal{A}$.

3.2 An Alternative Implementation

An alternative method involves the use of the FFT. In order to see how the FFT can be applied in this case, we can write

$$(U \otimes \mathbb{I}_n)\mathbf{z} = \left[\begin{array}{c|c|c} U_{11}\mathbb{I}_n & \dots & U_{1\ell}\mathbb{I}_n \\ \hline \vdots & \ddots & \vdots \\ \hline U_{\ell 1}\mathbb{I}_n & \dots & U_{\ell\ell}\mathbb{I}_n \end{array}\right] \begin{bmatrix} \mathbf{z}_1 \\ \vdots \\ \mathbf{z}_\ell \end{bmatrix} \tag{3.10}$$

$$= \left[\begin{array}{c|c|c} U & & \\ \hline & \ddots & \\ \hline & & U \end{array}\right] \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_n \end{bmatrix} \tag{3.11}$$

$$= (\mathbb{I}_n \otimes U)\mathbf{x}, \tag{3.12}$$

where $(\mathbf{x}_i)_k = (\mathbf{z}_k)_i$ and $\mathbf{x}_i \in \mathbb{R}^\ell$ are the chunks of $\mathbf{x}$, $i \in [1, n]$ and $k \in [1, \ell]$. This transformation is called a vector transpose and can be carried out in $\mathcal{O}(n)$ operations over ℓ processes. Therefore we can form

$$\mathbf{y}_i = U\mathbf{x}_i, \tag{3.13}$$

where i is an integer such that $i \in [1, n]$. According to [12], the cost of applying U to a vector using the FFT is $\mathcal{O}(\ell \log \ell)$. Before we can progress with the calculation, we will

have to apply the vector transpose again to $[\mathbf{y}_1, ..., \mathbf{y}_n]^T$. Consequently the complexity of this implementation now becomes $\mathcal{O}(\ell \log \ell + n)$ over $\max(n, \ell)$ processes. This method relies on the vector transpose being executed in parallel. This is given in the code snippet below:

```
void VecTranspose(int mynode,int numnodes,int N,int L,
        std::complex<double>*x,std::complex<double>*z)
{
    int i,j;
    std::complex<double>*y;
    int local_rows = (int)((double)L/numnodes);
    int local_offset = local_rows*mynode;
    y = new std::complex<double>[N*local_rows];
    for(j =0;j< local_rows;j++)
    {
        for(i=0;i<N;i++)
        {
            // We interpret the input vector x as a column-
            // major matrix. This map essentially transposes
            // the matrix form of x a "N-chunk" at a time.
            k = j + local_offset;
            y[j*N+i] = x[i*L+k];
        }
    }
    MPI_Allgather(y,local_rows*N,MPI_DOUBLE_COMPLEX,z,
        N*local_rows,MPI_DOUBLE_COMPLEX,MPI_COMM_WORLD);
}
```

While the complexity analysis for this method indicates that it will perform better for certain selections of n and ℓ, we have to take into account the further communications. In code, this method looks like this:

```
void ApplyPreconditionerFFT(int mynode,int totalnodes,int N, int L,
        std::vector<std::complex<double>*>& Wblks,std::complex<double>* q,
        std::complex<double>* x)
{
    std::complex<double>*b = new std::complex<double>[N*L];
    SetEqualTo(mynode,totalnodes,N,L,q,b,1.0);

    VecTranspose(mynode,totalnodes,L,N,b,x); /* +totalnodes communications */
    MultiplicationByIdentityKronU_usingFFT(mynode, totalnodes,N,L,x,b,1);
    VecTranspose(mynode,totalnodes,N,L,b,x); /* +totalnodes communications */
    BlockTriDiagSolve_Thomas(mynode,totalnodes,N,L,Wblks,b,x);
    VecTranspose(mynode,totalnodes,L,N,b,x); /* +totalnodes communications */
    MultiplicationByIdentityKronU_usingFFT(mynode, totalnodes,N,L,x,b,0);
    VecTranspose(mynode,totalnodes,N,L,b,x); /* +totalnodes communications */
    delete [] b;
}
```

As we can see, there are $\sim 4p$ extra communications over the original implementation of the application of the preconditioner. Considering that the original implementation required $\sim 6p$ communications we are almost doubling the number of communications

by using the FFT-related method. However, this method does reduce the memory footprint as we no longer need to carry around the Fourier matrices.

3.3 Application to Non-Uniform in Time All-At-Once Formulations

From an inspection of the operations we can see that the most expensive component of a GMRES iteration is the application of the preconditioner. If we consider applying the above ideology to $\mathcal{Q}_i^{-1}\mathcal{B}\mathbf{U} = \mathcal{Q}_i^{-1}\mathbf{b}$, then for us to increase i from 1 to 2 we have to take into account the cost of one extra preconditioner application, as well as communication. To see why this is consider writing (2.24) as

$$\mathcal{Q}_2^{-1} = \mathcal{P}^{-1} - \mathcal{P}^{-1}([\sigma \otimes K]\mathcal{P}^{-1}) = (\mathbb{I}_{n\ell} - \mathcal{P}^{-1}[\sigma \otimes K])\mathcal{P}^{-1}. \tag{3.14}$$

We can then store result $\mathbf{y} = \mathcal{P}^{-1}\mathbf{z}$ and carry out the matrix vector multiplication $\mathbf{x} = [\sigma \otimes K]\mathbf{y}$, followed by $\mathbf{w} = \mathcal{P}^{-1}\mathbf{x}$. Then we evaluate

$$\mathcal{Q}_2^{-1}\mathbf{z} = \mathbf{y} - \mathbf{w}. \tag{3.15}$$

That is, for us to consider the Neumann method a success, we must expect the number of GMRES iterations that are needed to solve the problem to reduce by more than half. We experimented with approximations to the Neumann approximation (3.14), such as

$$\tilde{\mathcal{Q}}_2^{-1} = (\mathbb{I}_{n\ell} - [\sigma \otimes K])\mathcal{P}^{-1}, \tag{3.16}$$

but we did not find such approximations to be more beneficial than taking $i = 1$ in (2.24). However, since $\mathcal{P}$ is in tensor format, low-rank approximations to the inverse do exist (see [15]), although the added computational cost of implementing these in parallel may induce a greater overhead than computing the inverse exactly.

3.4 Extension to GPU Programming

Our above explanation pertains to CPU programming. That is, we have developed code that can be distributed among a group of CPU-bound processes that can be executed on a many-core system. We can take this one step further and apply this method in the framework of GPU programming. The matrix-vector multiplications can be carried out rapidly and fairly simply in OpenCL. We used Parallel Cyclic

Reduction (PCR) and hybrid Thomas-PCR (see [11]) for the inversion of the block-tridiagonal matrices. Our simple implementation performed well in the application of the preconditioner $\mathcal{P}$. However, due to time constraints, we were not able to provide a robust all-at-once solver like we have done in the CPU-bound case.

3.5 Summary

In our preliminary complexity analysis we noted that the total memory cost would be $\mathcal{O}(\ell^2 + n\ell p)$ over p processes. This does not take into account the memory required for temporary storage of the distributed matrices. If we were more pedantic we would say the total memory cost is $\mathcal{O}(\ell^2 + (\ell + n)p + n\ell p)$ and the computational cost is $\mathcal{O}(n\ell^2/p)$ operations over p processes. As we increase the number of processes the computational cost decreases and the memory cost increases, which is exactly what we would expect. Throughout our experiments we are only considering $p \ll \min(n, \ell)$ and so this pedantry is uncalled for. However, in the interest of exascale computing implementations, we must take into account these observations as memory availability will be restricted.

We considered an alternative implementation that utilised the FFT, which required a significant increase in communication. On a positive note, the total memory cost for this implementation is $\mathcal{O}(n\ell p)$ over p processes. This is a significant reduction in memory cost; however, from our experiments carried out in the next chapter we found that the amount of communication throttled the performance, and this degradation in performance will only get worse in the framework of exascale computing.

Chapter 4

Numerical Results

4.1 Preface

All parallel results in this section were generated on the `nightcrawler` workstation at Oxford University. This machine is equipped with 2×18 core Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz processors, 768 GB RAM and 4600 GB in scratch disk capacity. Care was taken to access the machine when the workload was low so as to maintain consistent results. Values of n and ℓ are chosen to be multiples of 32. Values of n and ℓ equal to powers of 2 would have also been a good choice.

4.2 Uniform in Time Discretisations

In Figure 4.1 we can see that the number of iterations required to solve the system associated with the heat equation increases, roughly linearly, with the number of degrees of freedom. We do not include the results for the wave equation formulations here as the iteration counts were very large. In fact, in most instances GMRES had taken $n\ell$ iterations and was still a few orders of magnitude away from reaching the requested tolerance of 10^{-5}. In Figure 4.1 we can also see iteration counts for the convection-diffusion equation. With $V > 0$ the solution dissipates faster than the solution of the heat equation, and with $V < 0$ the solution takes longer to dissipate. These observations are quite interesting. The unpreconditioned monolithic systems derived from problems which are inherently dissipative require less GMRES iterations than those that are derived from problems which are inherently conservative. Setting aside the poor preliminary results associated with the wave equation, the results associated with the heat equation and the convection-diffusion equation are problematic.

For us to understand why this is an issue, we need to look more closely at the GMRES algorithm. The following version of the GMRES algorithm was taken from [21].

Algorithm 1 GMRES

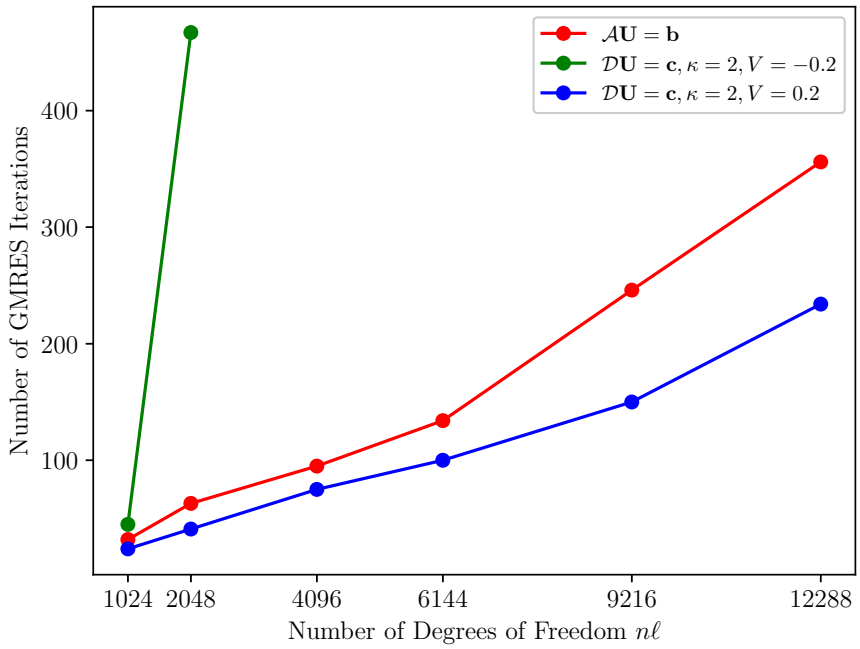
1: $\mathbf{q}_1 = \mathbf{b}/||\mathbf{b}||$
2: **for** $k = 1, 2, 3, ...$ **do**
3: $\quad \mathbf{v} = A\mathbf{q}_k$ ▷ If $A \in \mathbb{R}^{n\times n}$ (and dense) then this costs $\mathcal{O}(n^2)$.
4: $\quad$ **for** $j \in [1, k]$ **do**
5: $\quad\quad h_{jk} = q_j^* v$ ▷ $\mathcal{O}(n)$, executed k times on the k-th iteration.
6: $\quad\quad v = v - h_{jk}\mathbf{q}_j$
7: $\quad h_{k+1,k} = ||\mathbf{v}||$
8: $\quad \mathbf{q}_{k+1} = \mathbf{v}/h_{k+1,k}$
9: $\quad$ Minimisation of $||r_k||$ which yields $\mathbf{y}$. ▷ $\mathcal{O}(k^2)$
10: $\quad \mathbf{x}_k = Q_k\mathbf{y}$ ▷ $\mathcal{O}(kn)$

In the above algorithm we make notes of the complexity of the most expensive parts of the algorithm. The expense highlighted in line 9 accounts for the backward substitution which is executed when the residual is less than the user-specified tolerance, that is, when GMRES has converged. For small iteration numbers, the steps of the algorithm which scale with the iteration number k are not relevant. However, when iteration numbers get very large the time taken to complete these calculations becomes significant.

By solving a nearby problem or, more precisely, by solving the preconditioned problem, we can significantly reduce the iteration count. Solving $\mathcal{P}^{-1}\mathcal{A}\mathbf{U} = \mathcal{P}^{-1}\mathbf{b}$ with the initial condition $u_0 = x(1-x)$ and the smooth initial condition $u_0^s = \sin(2\pi x)$ results in a constant iteration count of 2 for all values of n and ℓ tested. Similar improvements hold for the convection-diffusion equation. An interesting point to make is that increasing g from 0 to 1 resulted in the GMRES iteration count for the unpreconditioned system associated with the heat equation to increase significantly, but the iteration count required to solve the preconditioned system only changed slightly in some cases. That is to say, iteration counts remained at ~ 2.

Timing results for the heat equation on a uniform temporal partition with initial condition u_0^s are given in Table 4.1. Referring to Table 4.1 we can see that increasing the number of processes from 1 to 32 results in a significant reduction in the time taken to solve the preconditioned system associated with the heat equation. The most significant speed-up is achieved when $n = 1440$ and $\ell = 1568$: The time taken

Figure 4.1: *The number of GMRES iterations required in order to solve the monolithic system* $\mathcal{A}\mathbf{U} = \mathbf{b}$ *associated with the heat equation, with* $g = 0$*, and the monolithic system* $\mathcal{D}\mathbf{U} = \mathbf{c}$ *associated with the convection-diffusion equation. Tolerance was set to* 10^{-5}*. Initial condition* $u_0 = x(1-x)$ *was used along with homogeneous Dirichlet boundary conditions.*



to solve the system reduces from ~ 35 minutes to ~ 1 minute. Observe that, for all values of n and ℓ, the time taken to solve the problem reduces by more than half as a result of increasing p from 1 to 2. We suspect that this is because half of the problem fits in local memory better than the entire problem does.

While the data presented in Table 4.1 is very useful for highlighting the speed-up achieved by distributing the calculation across multiple processes, it would be useful to see how efficiently the processes are being used. To see this, we consider the parallel efficiency of p processes defined by

$$P_{eff}^{p} = \frac{\text{Time Taken on 1 Process}}{p \times \text{Time Taken on } p \text{ Processes}}. \tag{4.1}$$

The parallel efficiency results are shown in Figure 4.2. The suspected reason for the jump in going from P_{eff}^{1} to P_{eff}^{2} is that, as noted above, the problem fits better in local memory over two processes. The parallel efficiency falls as we increase the number of processes used in the calculation, which is to be expected. This is a consequence of

Table 4.1: *Timed results (in seconds) for solving the system* $\mathcal{P}^{-1}\mathcal{A}\mathbf{U} = \mathcal{P}^{-1}\mathbf{b}$ *using GMRES with tolerance set to* 10^{-5}*. The iteration count remained at a constant value of 2 for all values of* n *and* ℓ *tested.* p *is the number of processes used in the calculations.*

		$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
	$n=320$	77.72	29.26	15.32	8.95	5.11	3.34
$\ell=768$	$n=512$	152.64	57.54	32.71	17.52	11.54	6.69
	$n=768$	245.47	97.77	50.81	30.71	16.66	9.65
	$n=320$	146.67	54.68	28.40	17.059	10.35	6.07
$\ell=1024$	$n=512$	265.22	107.07	60.86	34.13	20.40	11.75
	$n=768$	459.12	198.94	101.23	55.85	28.55	16.12
	$n=320$	325.14	124.67	63.64	39.78	22.74	13.06
$\ell=1440$	$n=512$	646.81	239.65	123.44	72.44	40.95	22.50
	$n=768$	979.85	432.46	215.77	114.99	59.80	32.41
$\ell=1568$	$n=1440$	2119.91	815.93	431.13	218.24	118.62	63.30

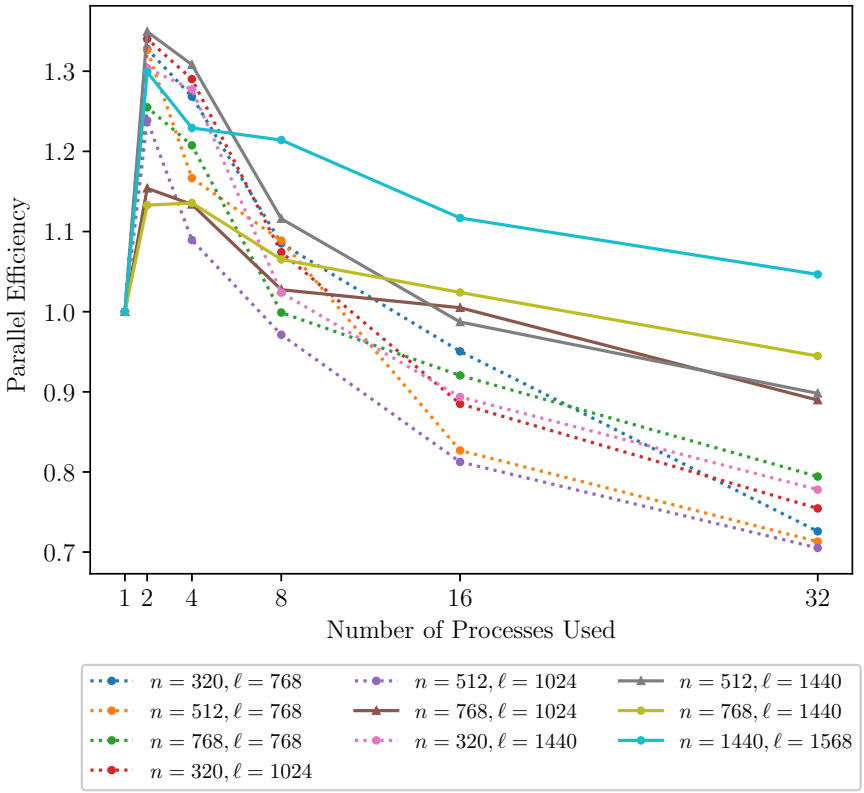
the number of communications taking place between processes. We note that as we increase the number of degrees of freedom, P_{eff}^{32} increases. In particular, for the values $\ell = 1568$ and $n = 1440$, $P_{eff}^{32} > 1$. That is, our metric for measuring parallel efficiency implies that the all-at-once implementation is more efficient on 32 processes than it is on a single process. The parallel efficiency results are very good overall; they indicate that the all-at-once implementation scales quite well for sufficiently large problem sizes.

In Section 2 we introduced an all-at-once formulation of the wave equation. For the wave equation we maintain $g = 0$ throughout. Three formulas were considered as candidates to approximate the time derivative. The CD formulation performed poorly by selecting the smooth initial condition u_0^s, and GMRES failed to converge to a solution by selecting the non-smooth initial condition

$$u_0^{ns}(x) = \begin{cases} \cos^2 4\pi\big(x - \frac{1}{2}\big), & x \in \big(\frac{3}{8}, \frac{5}{8}\big), \\ 0, & \text{otherwise}, \end{cases} \tag{4.2}$$

when applied to the system $\mathcal{R}_{CD}^{-1}\mathcal{C}_{CD}\mathbf{U} = \mathcal{R}_{CD}^{-1}\mathbf{b}_{CD}$. The CD formulation performed poorly on u_0^s in the sense that the solution displays aggressive numerical dissipation, regardless of how large n and ℓ are chosen to be. However, the number of GMRES iterations required to solve the system $\mathcal{R}_{CD}^{-1}\mathcal{C}_{CD}\mathbf{U} = \mathcal{R}_{CD}^{-1}\mathbf{b}_{CD}$ with the smooth initial condition remained at 2 for all values of n and ℓ tested. For these reasons the CD

Figure 4.2: *The parallel efficiency of our implementation of GMRES used to solve the all-at-once formulation of the preconditioned heat equation system* $\mathcal{P}^{-1}\mathcal{A}\mathbf{U} = \mathcal{P}^{-1}\mathbf{b}$.



formulation will not be considered further.

Iteration counts for BD2 and BD4 are shown in Table 4.2 and Table 4.3. Plots of the solution of the wave equation, obtained using the corresponding all-at-once formulations, are shown in Figure 4.3. All of these results were obtained with the non-smooth initial condition u_0^{ns}. Iteration counts for BD2 are low and the wave speeds are largely conserved. The drawback of BD2 is the introduction of numerical dissipation in the solution, as seen in Figure 4.3 (a), although the dissipation is very subtle compared to that observed in the solution of the CD formulation. BD4 rectifies this drawback, but iteration counts are much higher. Similarly to the CD approximation, when the smooth initial condition was used, the number of GMRES iterations required to solve the problem remained fixed at 2 for both BD2 and BD4 formulations. Our observations indicate that the all-at-once formulation of the wave equation is sensitive to the choice of initial conditions.

We note that the iteration counts observed for both the all-at-once formulation of the heat equation and the all-at-once formulation of the wave equation relate to the

theory derived in Chapter 2. When GMRES is applied to the preconditioned all-at-once formulation of the heat equation we observe iteration counts of 2 for a selection of initial conditions and all values of n and ℓ tested. However, when GMRES is applied to the BD2 preconditioned formulation of the wave equation we observe iteration counts of ~ 8 for large values of n and ℓ.

Table 4.2: *GMRES iteration counts k required to solve the wave equation system. v is the wave speed of the approximate solution. The wave speed of the exact solution is 1. (a) $\mathcal{R}_{BD2}^{-1}\mathcal{C}_{BD2}\mathbf{U} = \mathcal{R}_{BD2}^{-1}\mathbf{b}$, (b) $\mathcal{R}_{BD4}^{-1}\mathcal{C}_{BD4}\mathbf{U} = \mathcal{R}_{BD4}^{-1}\mathbf{b}_{BD4}$. Tolerance was set to 10^{-5}.*

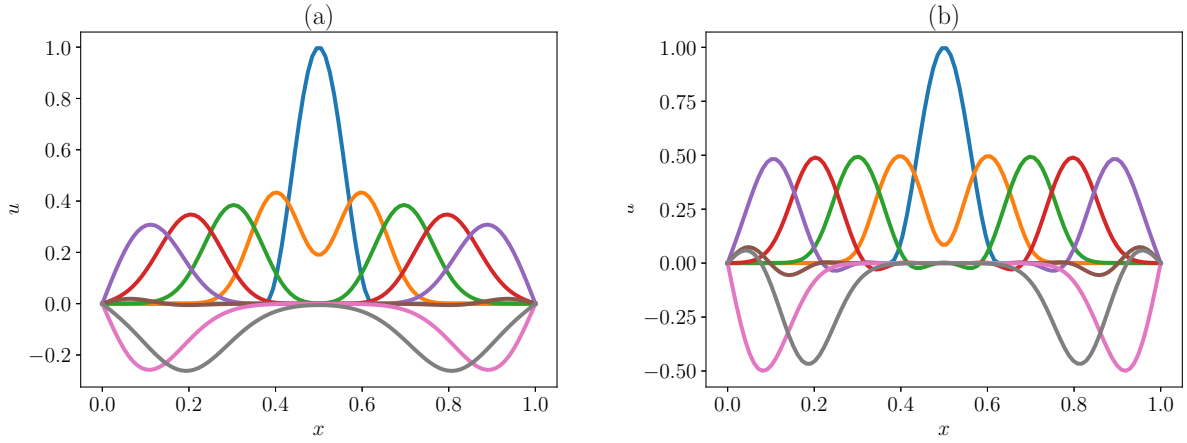
(a)		k	v	(b)		k	v
	$n = 32$	5	1.03		$n = 32$	60	1.03
$\ell = 32$	$n = 64$	5	1.01	$\ell = 32$	$n = 64$	100	1.01
	$n = 96$	5	1.01		$n = 96$	180	0.67
	$n = 32$	6	2.06		$n = 32$	80	1.03
$\ell = 64$	$n = 64$	6	1.01	$\ell = 64$	$n = 64$	120	1.01
	$n = 96$	6	1.34		$n = 96$	200	0.67
	$n = 32$	6	3.09		$n = 32$	140	3.10
$\ell = 96$	$n = 64$	8	1.52	$\ell = 96$	$n = 64$	180	1.52
	$n = 96$	6	1.01		$n = 96$	9216	1.01

Table 4.3: *GMRES iteration counts k required to solve $\mathcal{R}_{BD2}^{-1}\mathcal{C}_{BD2}\mathbf{U} = \mathcal{R}_{BD2}^{-1}\mathbf{b}$ for larger values of n and ℓ. Tolerance was set to 10^{-5}.*

		k
	$n = 320$	8
$\ell = 768$	$n = 512$	8
	$n = 768$	8
	$n = 320$	8
$\ell = 1024$	$n = 512$	8
	$n = 768$	8
	$n = 320$	8
$\ell = 1440$	$n = 512$	8
	$n = 768$	8

Timed results for the solution of the wave equation using the all-at-once formulation are provided in Table 4.4, and parallel efficiency results are displayed in Figure

Figure 4.3: *Solution profiles for the wave equation. The profiles were taken at equally spaced time intervals and the non-smooth initial condition* u_0^{ns} *was used.* $n = \ell = 128$. *(a)* $\mathcal{R}_{BD2}^{-1}\mathcal{C}_{BD2}\mathbf{U} = \mathcal{R}_{BD2}^{-1}\mathbf{b}_{BD2}$, *(b)* $\mathcal{R}_{BD4}^{-1}\mathcal{C}_{BD4}\mathbf{U} = \mathcal{R}_{BD4}^{-1}\mathbf{b}_{BD4}$.



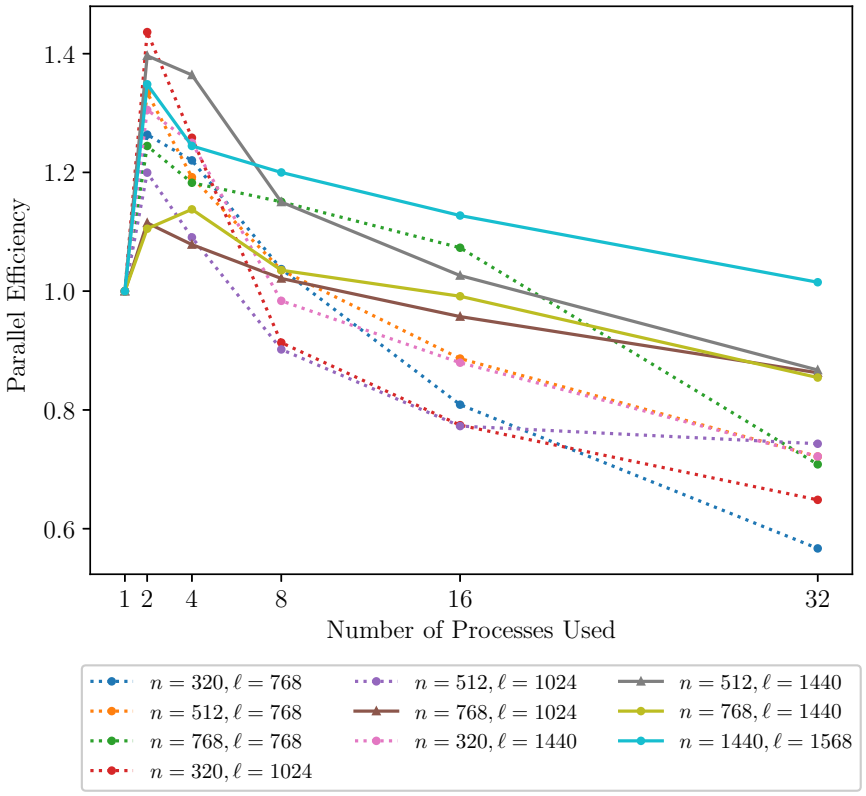
4.4. The system associated with the wave equation resulting from using BD2 is less sparse than the system associated with the heat equation and so we would expect the computations to take longer. Also note that we have a reduction in parallel efficiency for all values of n and ℓ tested. However, the parallel efficiency does seem to increase with the number of degrees of freedom. This trend was observed in the parallel efficiency results for the heat equation.

There are two interesting points to make here. The first is that while the timed results are longer to solve the wave equation than the heat equation, this is simply due to extra calculations being done on each of the processes, as opposed to extra communication. In the era of exascale computation these extra calculations will be essentially free due to the exaggerated cost of communication, which will be one of the main foci in the design of algorithms to be executed on such systems. The second point is that the BD2 implementation of the wave equation has the same cost per GMRES iteration as a higher order all-at-once implementation of the heat equation. Therefore this gives us some idea of how well the all-at-once method performs as we increase the number of steps in the method used to approximate the time derivative in the heat equation.

In Chapter 3 we discussed an alternative implementation of the all-at-once method that made use of the FFT. From a complexity analysis of this alternative implementation it appeared that it would perform better than the original implementation. However, as we noted, there is an increase in communication cost associated with

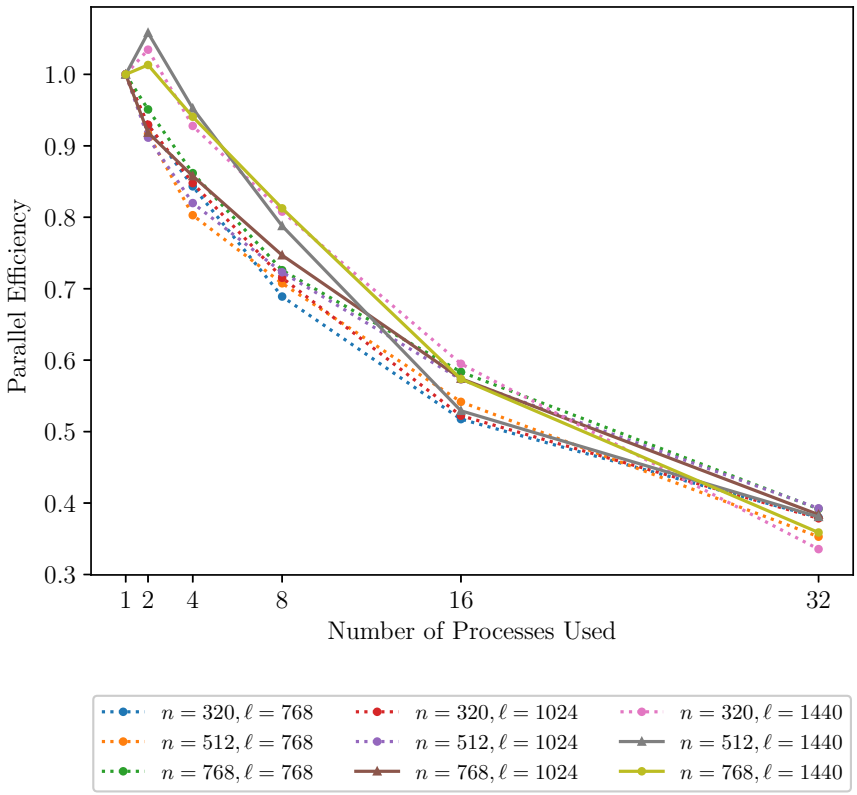
this system.

Figure 4.4: *The parallel efficiency of our implementation of GMRES used to solve the all-at-once formulation of the preconditioned wave equation system* $\mathcal{R}_{BD2}^{-1}\mathcal{C}_{BD2}\mathbf{U} = \mathcal{R}_{BD2}^{-1}\mathbf{b}_{BD2}$. *The smooth initial condition* u_0^s *was used.*



Referring to Figure 4.5 we can see that the parallel efficiency plummets as we increase the number of processes used in the calculation. The main reason behind this poor scalability is the extra communication. The computational cost per process and the number of communications required to carry out the multiplication by $\mathcal{A}$ remains the same. However, due to the extra vector transpose operators we need a further $\sim 4p$ communications; this is because the vector transpose only requires single communication for each of p processes. This does not appear to be a significant increase. We believe that there are various mitigating and aggravating factors present. One aggravating factor, it would seem, is evident from the significant degradation in parallel efficiency when increasing the number of processes from 8 to 16 using the FFT-implementation. In this case it is very likely that the processes reside on different CPU dies and so the processes that are receiving data from the sending node would naturally have to wait longer if they reside on a CPU die different from the

Figure 4.5: *The parallel efficiency of the alternative implementation of GMRES used to solve the all-at-once formulation of the preconditioned heat equation system* $\mathcal{P}^{-1}\mathcal{A}\mathbf{U} = \mathcal{P}^{-1}\mathbf{b}$.



sending node; this would, of course, increase the time it takes to communicate. A mitigating factor in the first implementation is that the problem is very likely going to fit better in local memory over a larger number of processes. The number of communications taking place in this instance is not enough to overcome the increase in parallel efficiency due to the better access to local memory.

4.2.1 Application to Spatial Domains in Two Dimensions

The all-at-once method can be applied to problems that reside in spatial dimensions greater than one. However, some alterations have to be made. For example, the Thomas algorithm has to be replaced by a multi-grid algorithm (see [8]). In Section 2.6 we showed that the eigenvalues of the all-at-once formulations for the heat and the wave equation are mostly equal to 1, and that the remaining eigenvalues are those of matrices that are related to linear combinations of the mass and stiffness matrices.

Table 4.4: *The timed results (in seconds) for solving the system* $\mathcal{R}_{BD2}^{-1}\mathcal{C}_{BD2}\mathbf{U} = \mathcal{R}_{BD2}^{-1}\mathbf{b}_{BD2}$ *using GMRES with tolerance set to* 10^{-5}*. The iteration count remained at a constant value of 2 for all values of* n *and* ℓ *tested.* p *is the number of processes used in the calculations. The smooth initial condition* u_0^s *was used.*

		$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
$\ell=768$	$n=320$	79.07	31.29	16.20	9.53	6.11	4.36
	$n=512$	163.68	61.33	34.33	19.76	11.54	7.09
	$n=768$	251.37	100.99	53.14	27.31	14.64	11.09
$\ell=1024$	$n=320$	153.37	53.39	30.47	20.99	12.38	7.39
	$n=512$	287.23	119.72	65.84	39.81	23.23	12.08
	$n=768$	497.12	222.93	115.24	60.84	32.46	18.01
$\ell=1440$	$n=320$	328.17	125.71	65.64	41.70	23.32	14.21
	$n=512$	680.15	243.53	124.65	73.92	41.42	24.51
	$n=768$	960.33	434.46	211.01	115.95	60.54	35.12
$\ell=1568$	$n=1440$	2211.97	820.21	444.31	230.42	122.63	68.10

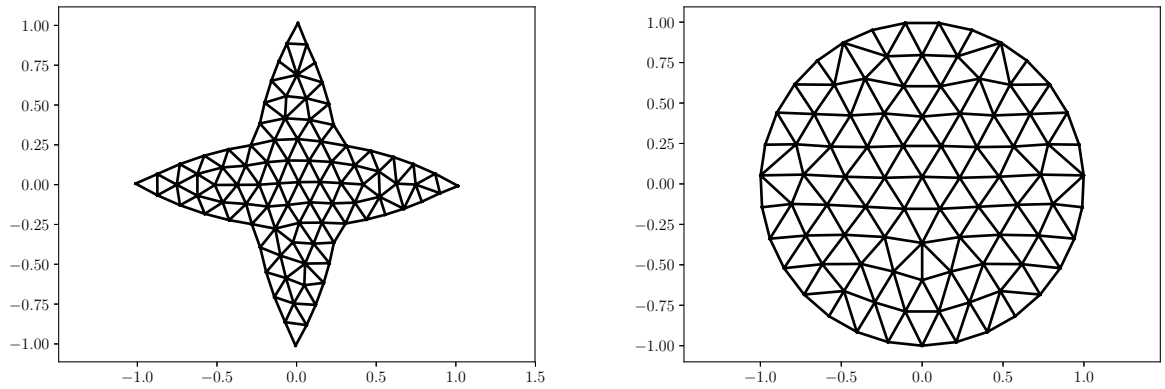
In two spatial dimensions these matrices no longer have a banded structure and so we can expect the eigenvalue profile of the preconditioned monolithic systems to alter slightly.

In [14] there were no restrictions imposed on the spatial structure of the domain. Then again, there was no investigation into how the spatial structure of the domain might alter the results of the all-at-once method. That is, we are interested in how the shape of the domain alters the effectiveness of the all-at-once method. In this section we are going to concentrate on three spatial domains: a disc, a square and a star shape. We are also going to restrict our attention to the wave equation. Our reasoning for this is that when GMRES was applied to the all-at-once formulation of the heat equation, GMRES converged in a fixed number of iterations regardless of which one of these domains was chosen. On triangulating a square domain, we can guarantee uniform triangles. However, as we can see from Figure 4.6, the triangle sizes vary when the domains are not as simple as a square. In the star-shaped domain it seems as though we have less variation in the triangle size than we have in the triangulation of the disc.

To apply the all-at-once method to a two-dimensional domain we have to alter our software slightly. We used a modified version of the `MassStiff` FEM solver.[1] This

[1] `MassStiff` is a basic automatic FEM solver which I have submitted for completion of the "C++ for Scientific Computing" special topic.

Figure 4.6: *Domains used in the application of the all-at-once method to problems that reside in spatial dimensions of order 2.*



software was modified to use an implementation of the `distmesh2d` triangulation routine of [17] to control the mesh size.

In Figure 4.7 we provide GMRES iteration counts for the preconditioned system associated with the wave equation on the three domains that we restricted our attention to. The BD2 formulation was used. As we can see, the iteration count increases as we increase ℓ. It appears that the lack of uniformity in the triangulation of the circular domain results in the iteration count being more sensitive to increases in ℓ than in domains with a more uniform triangulation.

4.3 Non-Uniform in Time Discretisations

In Chapter 2 we introduced an extension to the all-at-once method that enables us to consider problems that are non-uniformly discretised in time. The key behind the extension is the Neumann approximation of the preconditioner $\mathcal{Q}^{-1}$, given by (2.24). The non-uniform temporal discretisation that was used to obtain the numerical results associated with the system $\mathcal{Q}_i^{-1}\mathcal{B}\mathbf{U} = \mathcal{Q}_i^{-1}\mathbf{b}$ is given by

$$t_j = \begin{cases} 0, & j = 0, \\ \frac{1}{\ell}(j + \delta(\texttt{Rand(0,1,j)} - 0.5)), & j \in [1, \ell - 1], \\ 1, & j = \ell, \end{cases} \tag{4.3}$$

where j is an integer, δ is a real number such that $\delta \in (0, 1)$ and `Rand(0,1,j)` is a random real number between 0 and 1.

Figure 4.7: *The relationship between the time steps and the number of GMRES iterations required to solve the preconditioned system associated with the BD2 formulation of the wave equation. Neumann boundary conditions were used. The initial profile* $u_0(x, y) = \sin(xy)$ *was used.*

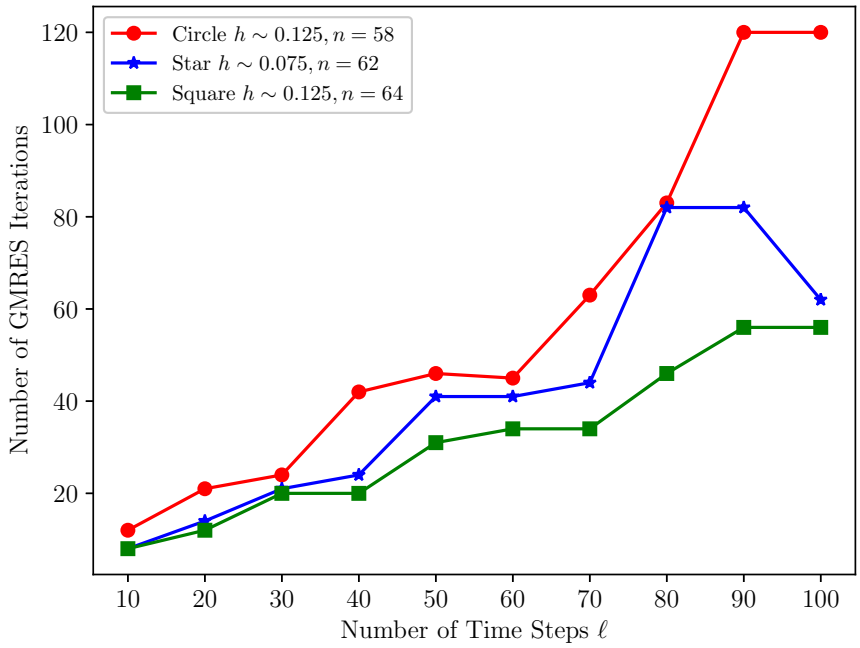


Table 4.5 shows the GMRES iteration counts required to solve $\mathcal{Q}_i^{-1}\mathcal{B}\mathbf{U} = \mathcal{Q}_i^{-1}\mathbf{b}$ for increasing values of i. We also show the difference between the time taken to solve the problem with $i = 2$ and the time taken to solve the problem with $i = 1$, $\Delta_{2,1}$. That is, a positive value of $\Delta_{2,1}$ indicates that it took longer to solve the problem with $i = 2$ than it did with $i = 1$. Firstly, $\Delta_{2,1} > 0$ for all values of n, ℓ and δ tested. This somewhat validates the claim we made in the previous section according to which the iteration count would need to reduce by more than half for any improvements to be observed in the timed results. For $n = 768, \ell = 1024$ and $\delta = 0.7$, the iteration count reduced by half as a result of increasing i from 1 to 2; in this case $\Delta_{2,1} = 21.68$, which is quite significant. On a positive note, Table 4.5 highlights the robustness of the all-at-once formulation in the presence of temporal perturbations. For instance, increasing δ from 0.1 to 0.9 increases the iteration count by 2 for all values of n and ℓ tested. Theorem 2.6.1 suggests that if $\mathcal{Q}^{-1}$ was applied exactly then we would expect GMRES to converge within $n + 1$ iterations. From Table 4.5 we can see that even though we are applying very crude estimates of $\mathcal{Q}^{-1}$ ($\mathcal{Q}^{-1} \approx \mathcal{P}^{-1}$) as a preconditioner,

Table 4.5: *GMRES iteration counts for* $\mathcal{Q}_i^{-1}\mathcal{B}\mathbf{U} = \mathcal{Q}_i^{-1}\mathbf{b}$. *The tolerance was set to* 10^{-5}. *The values of* n *and* l *for each table are given as (a)* $n = 320, \ell = 768$, *(b)* $n = 512, \ell = 768$, *(c)* $n = 768, \ell = 768$, *(d)* $n = 512, \ell = 1024$, *(e)* $n = 768, \ell = 1024$, *(f)* $n = 1024, \ell = 1024$. $\Delta_{1,2}$ *is the difference between the time taken to solve the problem with* $i = 2$ *and the time taken to solve the problem using* $i = 1$. *In this particular experiment, 16 processes were utilised.*

(a)	$i=1$	$i=2$	$i=3$	$\Delta_{2,1}$	(b)	$i=1$	$i=2$	$i=3$	$\Delta_{2,1}$
$\delta = 0.9$	6	4	3	9.32	$\delta = 0.9$	6	5	3	25.28
$\delta = 0.8$	6	4	2	9.36	$\delta = 0.8$	6	5	2	29.73
$\delta = 0.7$	4	4	2	12.83	$\delta = 0.7$	4	4	2	20.83
$\delta = 0.6$	4	4	2	12.38	$\delta = 0.6$	4	3	2	21.38
$\delta = 0.5$	4	4	2	11.75	$\delta = 0.5$	4	3	2	16.75
$\delta = 0.4$	4	4	2	10.33	$\delta = 0.4$	4	3	2	15.33
$\delta = 0.3$	4	3	2	7.90	$\delta = 0.3$	4	3	2	14.90
$\delta = 0.2$	4	3	2	5.61	$\delta = 0.2$	4	3	2	15.61
$\delta = 0.1$	4	3	2	4.32	$\delta = 0.1$	4	3	2	15.32

(c)	$i=1$	$i=2$	$i=3$	$\Delta_{2,1}$	(d)	$i=1$	$i=2$	$i=3$	$\Delta_{2,1}$
$\delta = 0.9$	6	4	3	34.13	$\delta = 0.9$	6	5	3	62.65
$\delta = 0.8$	6	4	2	33.75	$\delta = 0.8$	6	4	3	51.10
$\delta = 0.7$	6	3	2	21.68	$\delta = 0.7$	4	4	3	59.12
$\delta = 0.6$	4	3	2	31.87	$\delta = 0.6$	4	4	3	58.01
$\delta = 0.5$	4	3	2	33.43	$\delta = 0.5$	4	3	2	43.71
$\delta = 0.4$	4	3	2	32.42	$\delta = 0.4$	4	3	2	42.33
$\delta = 0.3$	4	3	2	31.31	$\delta = 0.3$	4	3	2	43.51
$\delta = 0.2$	4	3	2	31.91	$\delta = 0.2$	4	3	2	42.51
$\delta = 0.1$	4	3	2	32.36	$\delta = 0.1$	4	3	2	44.32

(e)	$i=1$	$i=2$	$i=3$	$\Delta_{2,1}$	(f)	$i=1$	$i=2$	$i=3$	$\Delta_{2,1}$
$\delta = 0.9$	6	4	3	59.11	$\delta = 0.9$	6	4	3	82.11
$\delta = 0.8$	6	4	3	58.12	$\delta = 0.8$	4	4	3	103.12
$\delta = 0.7$	4	3	3	63.22	$\delta = 0.7$	4	4	2	102.18
$\delta = 0.6$	4	3	2	65.26	$\delta = 0.6$	4	3	2	80.31
$\delta = 0.5$	4	3	2	63.31	$\delta = 0.5$	4	3	2	75.42
$\delta = 0.4$	4	3	2	64.23	$\delta = 0.4$	4	3	2	74.15
$\delta = 0.3$	4	3	2	67.53	$\delta = 0.3$	4	3	2	78.13
$\delta = 0.2$	4	3	2	64.21	$\delta = 0.2$	4	3	2	74.21
$\delta = 0.1$	4	3	2	65.26	$\delta = 0.1$	4	3	2	75.26

we are observing iteration counts far less than n. Increasing i from 2 to 3 does halve the iteration count in some cases, but since the cost incurred by communication is

going to be increased further, there is no advantage in doing so.

Table 4.6: *MINRES iteration counts for* $|\mathcal{Q}|_i^{-1} Y\mathcal{B}\mathbf{U} = |\mathcal{Q}|_i^{-1} Y\mathbf{b}$. *A hyphen indicates that the iterations were greater than 15000.*

	No Precon.	$i=1$	$i=2$	$i=3$
$n=64$, $\ell=96$	3546	6	3	2
$n=96$, $\ell=96$	7654	8	4	2
$n=96$, $\ell=128$	10386	14	8	3
$n=128$, $\ell=96$	11546	14	8	3
$n=128$, $\ell=128$	-	16	7	3
$n=160$, $\ell=128$	-	15	8	3

In Chapter 2 we discussed how we might symmetrise the monolithic systems resulting from the all-at-one formulations. An extension of that method was to consider non-uniform symmetric temporal discretisations. In [14] the application of symmetrising a monolithic system and then applying an absolute value preconditioner yielded good results. However, in this case we concentrate on the non-uniform temporal discretisation. In Table 4.6 we provide results for these calculations. Preconditioning with $i = 1$ is very effective, but preconditioning with $i > 1$, as we discovered in applying the Neumann approximation to the preconditioner $\mathcal{Q}$, is not advantageous.

4.4 Distance from Self-Adjointness

First, we repeat the observation that the monolithic systems $\mathcal{A}$, $\mathcal{C}_{BD2}$ and $\mathcal{C}_{BD4}$ are not symmetric. $\mathcal{C}_{CD}$ is symmetric but as the CD formulation was not successful, it will not be considered further because of the severe numerical dissipation observed. It would be highly advantageous if preconditioning the monolithic systems would not only improve the eigenvalue profile of the system, as we have shown, but also symmetrise the system.

In our cases preconditioning does not symmetrise the monolithic systems. However, it would be of interest to see how close our systems are to being symmetric. A feature of symmetric matrices is that they are orthogonally diagonalisable, and as a result the eigenvectors of such systems form an orthogonal basis. This gives us an intuitive way of measuring how far a matrix is from being symmetric. In Figure 4.8 we display the angles between the eigenvectors of the preconditioned monolithic

systems. To obtain this data we took each eigenvector and then measured its angle between all other eigenvectors and then appended the angle to a list.

Figure 4.8: *The angles between eigenvectors of the matrices (a)* $\mathcal{P}^{-1}\mathcal{A}$*, (b)* $\mathcal{R}_{BD2}^{-1}\mathcal{C}_{BD2}$ *and (c)* $\mathcal{R}_{BD4}^{-1}\mathcal{C}_{BD4}$*.* $n = 32, \ell = 32$*.*

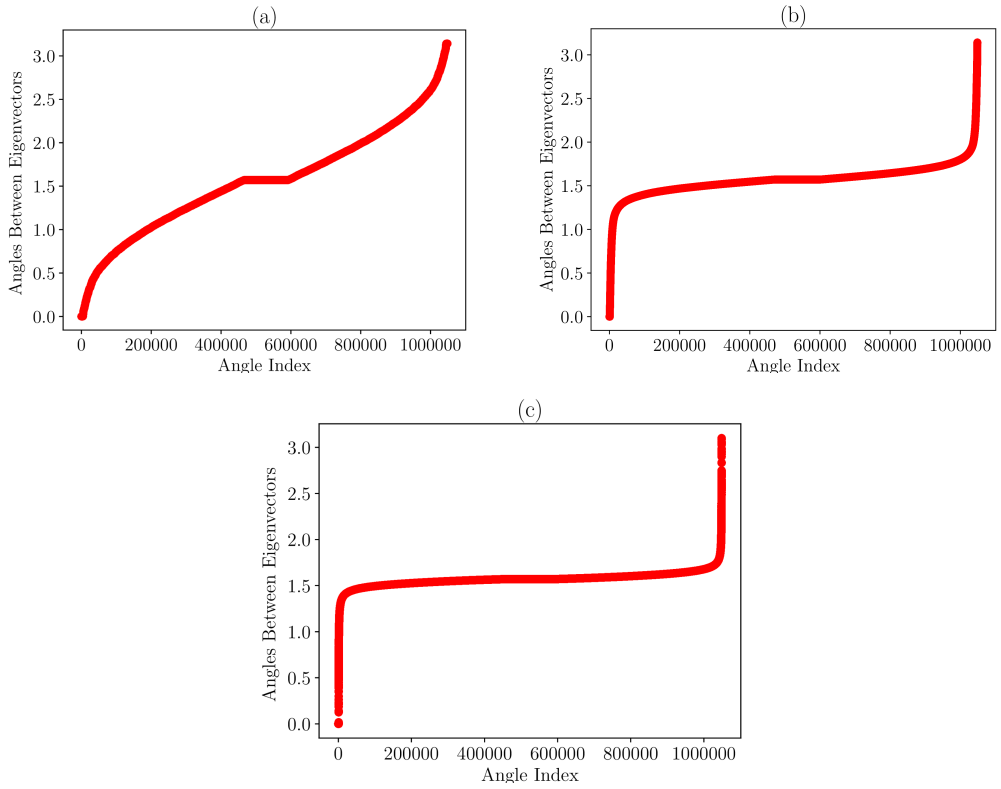


Figure 4.8 shows some intriguing results. If we attempt to correlate the GMRES iteration counts for the all-at-once formulation of the heat equation and the relevant formulations of the wave equation with the angles between the eigenvectors, we can see that the system that requires the smallest GMRES iteration count in order to obtain a solution has the least number of angles that are equal to $\pi/2$. By our metric for measuring how far a matrix is from being symmetric, the matrices that are further away from being symmetric converge faster. We should bear in mind that we have a very small sample set of 3 matrices. There is an alternative way to interpret this data which does fall in line with our observed convergence results. The system associated with the heat equation does have a smaller number of eigenvectors whose angles are close to π, or close to 0, than the system associated with the wave equation formulations. Also, the iteration numbers for the wave equation are greater than those for the heat equation. This could suggest that the initial conditions that

we have selected for the heat equation can be written as a linear combination of eigenvectors that have sufficiently large angles between them and so the solution is "found" by GMRES relatively fast. When the same initial conditions are selected for the wave equation formulations the situation might not be as desirable. The initial condition might now be written as a large number of eigenvectors, the angles between them being small. This would lead to more GMRES iterations to find an approximate solution to the systems associated with the wave equation formulation than to the system associated with the heat equation formulation.

Chapter 5

Final Remarks

5.1 Conclusion

We have described the all-at-once method in the form in which it was originally presented in [14], and provided a parallel implementation that seeks to be a proof-of-concept code to the claims made in [14], namely that the all-at-once method is parallelisable. In this work we have extended the all-at-once method via a Neumann approximation approach and provided an application to the wave equation, which is a hyperbolic problem. Our results surrounding the application of the all-at-once method to the wave equation are promising; the increase from first order to second order backward difference temporal approximations significantly reduces numerical dissipation. The implementation details of the all-at-once method have been discussed and the timed results appear very promising in the interest of scalability.

5.2 Future Work

There is a selection of avenues that we would have liked to pursue but, due to time constraints, could not. In Chapter 2 we alluded to the fact that initial implementations of the preconditioner application using GPU-bound processes were promising, but we failed to execute a working GMRES solver due to a lack of time. We believe that this would have been an effective method for applying the all-at-once method. This is because the financial cost of GPU-bound cores is a lot smaller than CPU-bound cores, which would make the all-at-once method available to lower-budget computing facilities. We would have liked to explore different methods surrounding the approximation of the preconditioner associated with the non-uniform temporal

formulation of the heat equation.

Appendix A

Supplementary Details for Theorem 2.6.4

Note that we can write the preconditioner as

$$\mathcal{R} = \mathcal{C} + E_1 A_2 E_{l-1}^T + E_2 A_2 E_l^T + E_1 A_1 E_l^T. \tag{A.1}$$

If we let $\tilde{\mathcal{C}}_1 = \mathcal{C} + E_1 A_2 E_{l-1}^T + E_2 A_2 E_l^T$, we can apply the Sherman-Morrison-Woodbury (SMW) formula to the equation

$$\mathcal{R} = \tilde{\mathcal{C}}_1 + E_1 A_1 E_l^T \tag{A.2}$$

to obtain the inverse

$$\mathcal{R}^{-1} = \tilde{\mathcal{C}}_1^{-1} - \tilde{\mathcal{C}}_1^{-1} E_1 (A_1^{-1} + E_l^T \tilde{\mathcal{C}}_1^{-1} E_1)^{-1} E_l^T \tilde{\mathcal{C}}_1^{-1}. \tag{A.3}$$

If we let $\tilde{\mathcal{C}}_2^{-1} = \mathcal{C} + E_1 A_2 E_{l-1}^T$, then we can apply the SMW formula to our expression for $\tilde{\mathcal{C}}_1$ to obtain

$$\tilde{\mathcal{C}}_1^{-1} = \tilde{\mathcal{C}}_2^{-1} - \tilde{\mathcal{C}}_2^{-1} E_2 (A_2^{-1} + E_l^T \tilde{\mathcal{C}}_2^{-1} E_2)^{-1} E_l^T \tilde{\mathcal{C}}_2^{-1}. \tag{A.4}$$

Finally we can apply the SMW formula to $\tilde{\mathcal{C}}_2$ to obtain

$$\tilde{\mathcal{C}}_2^{-1} = \mathcal{C}^{-1} - \mathcal{C}^{-1} E_1 (A_2^{-1} + E_{l-1}^T \mathcal{C}^{-1} E_1)^{-1} E_{l-1}^T \mathcal{C}^{-1}. \tag{A.5}$$

By substituting the expression for $\tilde{\mathcal{C}}_2^{-1}$ into the expression for $\tilde{\mathcal{C}}_1^{-1}$ and then substituting the resulting expression into the expression for $\mathcal{R}^{-1}$ we obtain

$$\mathcal{R}^{-1} = \mathcal{C}^{-1} - (Y_6 + Y_7 + Y_8)\mathcal{C}^{-1}, \tag{A.6}$$

where Y_6, Y_7 and Y_8 are given in the proof of Theorem 2.6.4. On multiplying the above expression by $\mathcal{C}$ on the right we obtain the required result.

Appendix B

Application Code

The application-specific file is given in this section. The code forms the monolithic matrices and then obtains the solution of the monolithic matrices using preconditoned GMRES.

```
#include <iostream>
#include <iomanip>
#include <mpi.h>
#include "MatrixHelper.h"
#include "ParallelRoutines.h"

using namespace std;


// Declarations of routines used in GMRES.
void Update(std::complex<double>*x,int lengthofx, int k,int m,
        std::complex<double>*h,std::complex<double>*s,
        std::vector<std::complex<double>*>v);
void GeneratePlaneRotation(std::complex<double> &dx,
        std::complex<double> &dy, std::complex<double> &cs,
        std::complex<double> &sn);
void ApplyPlaneRotation(std::complex<double> &dx,
        std::complex<double> &dy, std::complex<double> &cs,
        std::complex<double> &sn);

int main(int argc, char * argv[])
{
//  DECLARATION OF VARIABLES
    double start,end;           /* Used in timing the GMRES routine. */
    int i,j,k, N = 320,L=512;   /* N is the number of spatial steps,
                                        L is the number of time steps. */
    double h = 1.0/(N-1);       /* The size of the spatial discretisation step. */
    double timestep = 1.0/L;    /* Time step length. */
    double delta = 0.0;         /* Perturbation of temporal domain. */


    std::complex<double> *A,*x,*q,*y,*pointertolargeblocked,*b;
    std::complex<double> *massContig,*stiffContig;
```

```
34      std::complex<double> *F,*D,*Ft;
35
36      std::vector<double> timesteps,times,perts;
37
38      int totalnodes,mynode;
39      std::vector<std::complex<double>*> Wblocks,Ablocks;
40      std::complex<double>** UMonolithic,**UtMonolithic;
41      MPI_Init(&argc,&argv);
42      MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
43      MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
44
45  // RESERVATION OF MEMORY
46
47      // Intermediate calculation vectors.
48      y = new std::complex<double>[N*L];
49      x = new std::complex<double>[N*L];
50      q = new std::complex<double>[N*L];
51
52
53      // Right-hand-side vector.
54      b = new std::complex<double>[N*L];
55
56  //  INITIALISATION OF STRUCTURES
57
58  // All vectors are given to all nodes. While this increases the
59  // memory requirements of the program it significantly reduces
60  // the communication cost.
61
62
63  // FORMATION OF THE MATRICES ON NODE 0
64
65  // We take a different ideology with matrices. Due to their size
66  // only small matrices will be distributed across all nodes.
67
68      if(mynode==0)
69      {
70          // This gives us the times.
71          std::srand(std::time(nullptr));
72          for(i=0;i<L+1;i++) times.push_back(i*timestep +
73                          delta*timestep*(((double) rand() / (RAND_MAX)) - 0.5));
74
75          times[0] = 0; times[L] = 1;
76
77          // Calculate time steps.
78          for(i=1;i<L;i++)timesteps.push_back(times[i]-times[i-1]);
79
80          // Calculate perturbations.
81          for(i=0;i<L+1;i++)perts.push_back(timesteps[i]-timestep);
82
83          // Reservation of memory on the master node.
84          tridiag mass; CreateTridiag(N,mass);
85          tridiag stiff; CreateTridiag(N,stiff);
86          F = CreateMatrixContiguous(L,L);
87          D = CreateMatrixContiguous(L,L);
```

```
88          Ft = CreateMatrixContiguous(L,L);
89
90          // Formation of the mass and stiffness matrix.
91          // The Fourier basis matrix F, its transpose Ft,
92          // and the diagonal matrix of eigenvalues D are
93          // also formed.
94          FormMassStiff(h,N,mass,stiff);
95          FormFourier_Diag_FourierTranspose(L,F,D,Ft);
96
97  //
98  // FORMATION OF W: BEGIN
99  //
100
101         // W is the large tridiagonal matrix that needs to be inverted.
102
103         // Temporary structures used in formation of matrices.
104         tridiag stiffA0;CreateTridiag(N,stiffA0);
105         SetTriDiagEqualTo(N,stiff,stiffA0);
106
107         MultiplyTriDiagByConst(N,timestep,stiffA0);
108         // At this point we have
109         //  stiffA0 = timestep*K
110
111         std::vector<tridiag> tridaigVec;
112
113         // Reserve memory.
114         for(int i=0;i<L;i++)
115         {
116             tridiag temp; CreateTridiag(N,temp);
117             tridaigVec.push_back(temp);
118         }
119         // Multiply A1 by eigenvalue and ADD to A0.
120         for(int i=0;i<L;i++)
121         {
122             tridiag temp;CreateTridiag(N,temp);
123             SetTriDiagEqualTo(N,mass,temp);
124             MultiplyTriDiagByConst(N,(1.0-D[i*L+i]),temp);
125             AddTriDiag(N,temp,stiffA0,tridaigVec[i]);
126         }
127         // Reserve memory for contiguous counterparts.
128         for(int i=0;i<L;i++)
129         {
130             std::complex<double> *pointertolargeblocked =
131                                                 new std::complex<double>[3*N-2];
132             Wblocks.push_back(pointertolargeblocked);
133         }
134         // Fill the contiguous counterparts.
135         for(int i=0;i<L;i++)
136         {
137             for(j=0;j<N-1;j++) Wblocks[i][j] = std::get<0>(tridaigVec[i])[j];
138             for(j=0;j<N  ;j++) Wblocks[i][N-1+j] = std::get<1>(tridaigVec[i])[j];
139             for(j=0;j<N-1;j++) Wblocks[i][2*N-1+j] = std::get<2>(tridaigVec[i])[j];
140             Wblocks[i][N-1] = 1;
141             Wblocks[i][2*N-2] = 1;
```

```
142         }
143
144         // Packaging into monolithic system.
145
146 //
147 // FORMATION OF W: END
148 //
149
150 // We form the blocks of A and pass the -M as a separate argument.
151
152 //
153 // FORMATION OF Ablocks: BEGIN
154 //
155
156         // We form the monolithic A by considering
157         // block diagonal entries to be A0 = M + timestep[i]*K
158         // and the sub-diagonal entries to have
159         // A1 = -M.
160         tridaigVec.clear();
161         // Reserve memory.
162         for(int i=0;i<L;i++)
163         {
164             tridiag temp; CreateTridiag(N,temp);
165             tridaigVec.push_back(temp);
166         }
167         // Multiply A1 by time step and package.
168         for(int i=0;i<L;i++)
169         {
170             tridiag temp;CreateTridiag(N,temp);
171             SetTriDiagEqualTo(N,stiff,temp);
172             MultiplyTriDiagByConst(N,timesteps[i],temp);
173             AddTriDiag(N,temp,mass,tridaigVec[i]);
174         }
175         // Reserve memory for contiguous counterparts.
176         for(int i=0;i<L;i++)
177         {
178             std::complex<double> *pointertolargeblocked
179                                         = new std::complex<double>[3*N-2];
180             Ablocks.push_back(pointertolargeblocked);
181         }
182         // Fill the contiguous counterparts.
183         for(int i=0;i<L;i++)
184         {
185             for(j=0;j<N-1;j++) Ablocks[i][j] = std::get<0>(tridaigVec[i])[j];
186             for(j=0;j<N  ;j++) Ablocks[i][N-1+j] = std::get<1>(tridaigVec[i])[j];
187             for(j=0;j<N-1;j++) Ablocks[i][2*N-1+j] = std::get<2>(tridaigVec[i])[j];
188             Ablocks[i][N-1] = 1;
189             Ablocks[i][2*N-2] = 1;
190         }
191 //
192 // FORMATION OF Ablocks: END
193 //
194         // Formation of the monolithic U.
195         UMonolithic = CreateMatrix(L,L);
```

```
196         UtMonolithic = CreateMatrix(L,L);
197
198         for(i=0;i<L;i++)
199         {
200             for(j=0;j<L;j++)
201             {
202                 UMonolithic[i][j] = F[j*L+i];
203                 UtMonolithic[i][j] = Ft[j*L+i];
204             }
205         }
206
207         // Reservation of memory for massContig.
208         massContig = new std::complex<double>[3*N-2];
209
210         // Formation of contiguous mass matrix.
211         for(j=0;j<N-1;j++) massContig[j] = std::get<0>(mass)[j];
212         for(j=0;j<N  ;j++) massContig[N-1+j] = std::get<1>(mass)[j];
213         for(j=0;j<N-1;j++) massContig[2*N-1+j] = std::get<2>(mass)[j];
214
215
216 //
217 // FORMATION OF b: BEGIN
218 //
219         // b = [Mu_0 0 ... 0].
220
221         // Initial condition.
222         std::complex<double>* u0 = new std::complex<double>[N];
223         std::complex<double>* U0 = new std::complex<double>[N];
224
225         // The smooth initial condition.
226         for(i=0;i<N;i++) u0[i] = sin(2*M_PI*i*h);
227
228 /*
229         // The non-smooth initial condition.
230         for(i=0;i<N;i++)
231         {
232            if(i*h<0.5-1.0/8 || i*h > 0.5+1.0/8)
233                u0[i] = 0.0;
234            else
235            {
236                u0[i] = cos(4*M_PI*(i*h-0.5))*cos(4*M_PI*(i*h-0.5));
237            }
238         }
239 */
240         std::complex<double> prod =0;
241         U0[0] = massContig[N-1]*u0[0] +  massContig[2*N-1]*u0[1];
242         for(j=1;j<N-1;j++)
243         {
244             U0[j]= massContig[j-1]*u0[j-1] +
245                         massContig[N-1+j]*u0[j] + massContig[2*N-1+j]*u0[j+1];
246         }
247         U0[N-1] = massContig[2*N-2]*u0[N-1] + massContig[N-2]*u0[N-2];
248
249         for(i=0;i<N;i++) b[i] = U0[i];
```

```
250 //
251 // FORMATION OF b: END
252 //
253     }
254 
255     // The right-hand side is broadcast. This is consistent
256     // with our ethos for this code.
257     MPI_Bcast(b,N*L,MPI_DOUBLE_COMPLEX,0,MPI_COMM_WORLD);
258 
259 //
260 //  THE CALCULATION PHASE
261 //
262 
263 
264     // Temp vectors used in GMRES routine.
265     std::complex<double>* temp = new std::complex<double>[N*L];
266     std::complex<double>* temp2 = new std::complex<double>[N*L];
267 
268     // Residual vector.
269     std::complex<double>* r0 = new std::complex<double>[N*L];
270 
271     std::complex<double> normb,beta,resid; /* norm of preconditioned RHS.*/
272 
273     //  For tol = 10^{-5} we get a fixed iteration count of 2 for suff. large n,l.
274     std::complex<double> tol = 0.00001;    /* tolerance */
275 
276     int complete = 0;
277     int max_iter = 10;
278     int m = max_iter;
279 
280     std::complex<double>* s = new std::complex<double>[m+1];
281     std::complex<double>* sn = new std::complex<double>[m+1];
282     std::complex<double>* cs = new std::complex<double>[m+1];
283 
284     std::complex<double>* H = CreateMatrixContiguous(m+1,m+1);
285 
286     // Standard procedure is to measure the time taken to complete
287     // the calculation on the master node.
288     if(mynode == 0)
289     {
290         start = MPI_Wtime();
291     }
292 
293 
294 // ---------------------- CALCULATION PHASE BEGIN --
295 
296     // As we are measuring on node 0, we want all processes to
297     // "meet up" before entering the calculation stage.
298 
299     MPI_Barrier(MPI_COMM_WORLD);
300 
301 
302 // ---------------------------- GMRES BEGIN --
303 
```

```
304     // Our GMRES routine follows Trefethens NLA very closely.
305     // The sub routines are C++ implementations of the Wiki MATLAB implementation.
306 
307     j=1;
308 
309     // Calculation of ||P^{-1}b||.
310     ApplyPreconditioner(mynode,totalnodes,N,L,UMonolithic,UtMonolithic,Wblocks,b,temp);
311     CalculateNorm(mynode,totalnodes,N,L,temp,normb);
312 
313     // Calculation of r = P^{-1}(b-A*x).
314     MultiplyByHeatSystem(mynode,totalnodes,N,L,Ablocks,massContig,x,temp);
315     VectorSubtraction(mynode,totalnodes,N,L,b,temp,temp2);
316     ApplyPreconditioner(mynode,totalnodes,N,L,UMonolithic,UtMonolithic,
317         Wblocks,temp2,r0);
318 
319     // Calculate beta = ||r||.
320     CalculateNorm(mynode,totalnodes,N,L,r0,beta);
321 
322     if(normb.real() ==0)
323         normb = 1;
324 
325     resid = beta/normb;
326 
327     if(resid.real() < tol.real())
328     {
329         tol = resid;
330         max_iter = 0;
331         complete = 1;
332     }
333 
334     std::vector<std::complex<double>*> v;
335     for(int i=0;i<m+1;i++)
336     {
337         std::complex<double> *pointer = new std::complex<double>[N*L];
338         v.push_back(pointer);
339     }
340     while(j<=max_iter)
341     {
342         // Calculate v[0] = r0 * (1/beta)
343         SetEqualTo(mynode,totalnodes,N,L,r0,v[0],(1.0/beta));
344         s[0] = beta;
345 
346         for(i=0;i<m && j <= max_iter;i++,j++)
347         {
348 
349             if(complete) continue;
350 
351 //=========================================  ARNODLI BEGIN
352 
353             MultiplyByHeatSystem(mynode,totalnodes,N,L,Ablocks,massContig,v[i],temp);
354             ApplyPreconditioner(mynode,totalnodes,N,L,UMonolithic,
355                 UtMonolithic,Wblocks,temp,temp2);
356 
357             for(k=0;k<=i;k++)
```

```
358             {
359                 // w = temp2
360                 std::complex<double> dotprodoutput;
361                 DotProduct(mynode,totalnodes,N,L,temp2,v[k],dotprodoutput);
362                 H[k+i*m] = dotprodoutput;
363                 PlusEqualTo(mynode,totalnodes,N,L,v[k],temp2,-1.0*dotprodoutput);
364             }
365 
366             CalculateNorm(mynode,totalnodes,N,L,temp2,H[(i+1)+i*m]);
367             SetEqualTo(mynode,totalnodes,N,L,temp2,v[i+1],(1.0/H[(i+1)+i*m]));
368 
369 //======================================  ARNOLDI END
370 
371             for(k=0;k<i;k++)
372                 ApplyPlaneRotation(H[k+i*m],H[(k+1)+i*m],cs[k],sn[k]);
373 
374             GeneratePlaneRotation(H[i+i*m], H[(i+1)+i*m], cs[i], sn[i]);
375             ApplyPlaneRotation(H[i+i*m], H[(i+1)+i*m], cs[i], sn[i]);
376             ApplyPlaneRotation(s[i], s[i+1], cs[i], sn[i]);
377 
378             resid = fabs(s[i+1]);
379             if(resid.real()/normb.real() < tol.real())
380             {
381                 Update(x,N*L,m,m,H,s,v);
382                 tol = resid;
383                 max_iter = j;
384                 complete = 1;
385             }
386             if(complete) continue;
387 
388         }
389 
390         Update(x,N*L,m-1,m,H,s,v);
391 
392         // Calculation of r = P^{-1}(b-A*x).
393         MultiplyByHeatSystem(mynode,totalnodes,N,L,Ablocks,massContig,x,temp);
394         VectorSubtraction(mynode,totalnodes,N,L,b,temp,temp2);
395         ApplyPreconditioner(mynode,totalnodes,N,L,UMonolithic,UtMonolithic,Wblocks,temp2,r0);
396 
397 
398         // Calculate beta = ||r||.
399         CalculateNorm(mynode,totalnodes,N,L,r0,beta);
400         resid = beta/normb;
401         if(resid.real() < tol.real())
402         {
403             tol = resid;
404             max_iter = j;
405             complete = 1;
406         }
407         if (complete)continue;
408 
409     }
410 
411 //
```

```
412 // --------------------------- GMRES END --
413 //
414
415 // ---------------------- CALCULATION PHASE END --
416
417     MPI_Barrier(MPI_COMM_WORLD);
418     if(mynode == 0)
419     {
420         end = MPI_Wtime();
421         // Print the time taken for the calculation to complete.
422         std::cout << end- start << std::endl;
423         std::cout << std::endl;
424         // How many iterations did it take for GMRES to terminate?
425         std::cout << j << std::endl;
426     }
427     MPI_Finalize();
428 }
429
430 // Definitions of the routines used in GMRES.
431
432 void Update(std::complex<double>*x,int lengthofx, int k,int m,
433         std::complex<double>*h,std::complex<double>*s,std::vector<std::complex<double>*>v)
434 {
435     std::complex<double>*y = new std::complex<double>[k+1];
436     for(int i=0;i<k+1;i++) y[i] = s[i];
437
438     for(int i=k;i>=0;i--)
439     {
440         if(h[i+i*m] == 0.0) continue;
441         y[i] = y[i]/h[i+i*m];
442         for(int j=i-1; j>= 0;j--)
443         {
444             y[j] -= h[j+i*m]*y[i];
445         }
446     }
447
448     for(int j=0;j<=k;j++)
449     {
450         for(int mm =0;mm<lengthofx;mm++)
451             x[mm] += v[j][mm]*y[j];
452     }
453 }
454
455 void GeneratePlaneRotation(std::complex<double> &dx, std::complex<double> &dy,
456  std::complex<double> &cs, std::complex<double> &sn)
457 {
458     if (dy == 0.0)
459     {
460         cs = 1.0;
461         sn = 0.0;
462     }
463     else if (fabs(dy.real()) > fabs(dx.real()))
464     {
465         std::complex<double> tmp = dx / dy;
```

```
466         sn = 1.0 / sqrt( 1.0 + tmp*tmp );
467         cs = tmp * sn;
468     }
469     else
470     {
471         std::complex<double> tmp = dy / dx;
472         cs = 1.0 / sqrt( 1.0 + tmp*tmp );
473         sn = tmp * cs;
474     }
475 }
476
477 void ApplyPlaneRotation(std::complex<double> &dx, std::complex<double> &dy,
478  std::complex<double> &cs, std::complex<double> &sn)
479 {
480     std::complex<double> temp  =  cs * dx + sn * dy;
481     dy = -sn * dx + cs * dy;
482     dx = temp;
483 }
```

References

[1] V.A. Barker and O. Axelsson. *Finite Element Solution of Boundary Value Problems*. Elsevier, 1984.

[2] M. Bruna, P. K. Maini, and M. Robinson. Particle-Based Simulations of Reaction-Diffusion Processes with Aboria. *CoRR*, abs/1805.11007, 2018.

[3] T. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006.

[4] G. Antonucci. Iterative Solution of Evolutionary PDEs via All-At-Once Methods. *Thesis submitted for completion of M.Sc Mathematical Modelling and Scientific Computing*, 2017.

[5] A. J. Goddard and A. J. Wathen. A Note on Parallel Preconditioning for All-At-Once Evolutionary PDEs. *Submitted to Electronic Transactions on Numerical Analysis*, 2018.

[6] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 3rd edition, 1996.

[7] A. Greenbaum, V. Pták, and Z. Strakoš. Any Nonincreasing Convergence Curve is Possible for GMRES. *SIAM Journal on Matrix Analysis and Applications*, 17(3):465–469, 1996.

[8] V. E. Henson and U. M. Yang. BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner. *Applied Numerical Mathematics*, 41(1):155–177, 2002.

[9] N. Higham. *Functions of Matrices*. SIAM, 2008.

[10] J. Lions, Y. Maday, and G. Turinici. A "Parareal" in Time Discretization of PDE's. *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics*, 332(7):661–668, 2001.

[11] E. László, M. Giles, and J. Appleyard. Manycore Algorithms for Batch Scalar and Block Tridiagonal Solvers. *ACM Transactions on Mathematical Software*, 42(4):31:1–31:36, 2016.

[12] C. F. Van Loan. *Computational Frameworks for the Fast Fourier Transform.* SIAM, 1992.

[13] C. F. Van Loan. The Ubiquitous Kronecker Product. *Journal of Computational and Applied Mathematics*, 123(1):85–100, 2000.

[14] E. McDonald, J. Pestana, and A. J. Wathen. Preconditioning and Iterative Solution of All-At-Once Systems For Evolutionary Partial Differential Equations. *SIAM*, 40(2):A1012–A1033, 2018.

[15] V. Olshevsky, I. Oseledets, and E. Tyrtyshnikov. Tensor Properties of Multilevel Toeplitz and Related Matrices. *Linear Algebra and its Applications*, 412(1):1–21, 2006.

[16] C. C. Paige and M. A. Saunders. Solution of Sparse Indefinite Systems of Linear Equations. *SIAM Journal on Numerical Analysis*, 12(4):617–629, 1975.

[17] P.-O. Persson and G. Strang. A Simple Mesh Generator in MATLAB. *SIAM Review*, 46(2):329–345, 2004.

[18] R. Falgout, J. Dongarra, J. Hittinger, J. Bell, L. Chacon, M. Heroux, P. Hovland, E. Ng, C. Webster, and S. Wild. Applied Mathematics Research for Exascale Computing. Technical report, U.S. Department of Energy, 2014.

[19] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[20] G. Strang. A Proposal for Toeplitz Matrix Calculations. *Studies in Applied Mathematics*, 74(2):171–176, April 1986.

[21] N. L. Trefethen and D. Bau. *Numerical Linear Algebra.* SIAM, 1997.

[22] A. J. Wathen. Preconditioning. *Acta Numerica*, 24:329–376, 2015.

[23] A. J. Wathen, D. Silvester, and H. Elman. *Finite Elements and Fast Iterative Solvers with Applications in Incompressible Fluid Dynamics.* Oxford University Press, 2005.