Structured-grid multigrid with Taylor-Hood finite elements

by

© **Lukas Spies**

A thesis submitted to the School of Graduate Studies in partial fulfillment of the requirements for the degree of Masters of Science.

Department of Scientific Computing

Memorial University

August 2016

St. John's, Newfoundland and Labrador, Canada

Abstract

Recent years have seen renewed interest in the numerical solution of the Stokes Equations. At the same time, new computational architectures, such as GPUs and many-core processors, naturally perform best with the regular data access and computation patterns associated with structured-grid discretisations and algorithms. While many preconditioning approaches ignore the underlying mesh geometry, our approach is to develop a structured-grid implementation, taking advantage of the highly structured data-access patterns and employing stencil-based calculations. This opens up many opportunities for fine-grained parallelism, allowing us to take advantage of multicore and accelerated architectures. In this thesis, we will consider an implementation of a structured-grid monolithic Multigrid approach for Q2-Q1 finite-element discretisations, comparing its efficiency to an unstructured grid solver implemented in Trilinos. With the aim to eventually target large heterogeneous systems, we will discuss an implementation for moving from a serial code to the GPU by means of OpenCL and compare the efficiency of all three versions. Speedup factors of about 6.3x were observed for the GPU implementation over a serial implementation in Trilinos for a problem on a 768x768 mesh in 2D.

Acknowledgements

This thesis was made possible with the help and support of various people. First and foremost I want to thank my advisor, Scott MacLachlan. He devoted uncountable many hours discussing difficulties I was facing in my research, being an invaluable source of support. I would also like to thank Thomas Benson for spending many hours giving us timings of his Trilinos solver for comparison, and his fantastic help in debugging our code. I also owe a sincere thank you to Luke Olson for helping us figure out the best way forward for parallelising our code, preventing us from potentially starting out into the wrong direction. The many hours spent doing research and writing code haven't always been easy, but it always helped to get support from my wonderful girlfriend, who always believed in me. My parents have also been incredible helpful, not doubting for a second that I can do it. Last but not least I want to thank God for giving me this amazing opportunity to be here in Newfoundland, doing the research I did, making many new friends, and seeing some of this beautiful part of the world. None of that would have been possible without his love and provision.

Statement of contribution

The work represented in this thesis is the result of collaborative research between Lukas Spies and Scott MacLachlan, with its intellectual property equally co-owned by both. The thesis itself was written by Lukas Spies with the input and guidance of Scott MacLachlan. The Trilinos data was provided by Thomas Benson, who also helped us in debugging our code. Luke Olson helped us to find the best way to parallelise our code, and Andy Wathen suggested the use of the mass matrix instead of the triple matrix product in our Braess-Sarazin solver.

Table of contents

List of tables

List of figures

List of code

Chapter 1

Introduction

The focus of this thesis is on the development of algorithms for solving the Stokes equations. We will focus on their use in modern high-performance computing by performing an analysis of their implementation and efficiency targeting large heterogeneous systems. Using a structured-grid finite-element discretisation, we can express all calculations in stencil form that naturally break up into many small and independent calculations that are well suited for employing GPUs.

The Stokes equations are a well-known model of flow in situations where viscosity is the dominant physical force. While the Stokes equations themselves are idealized equations ignoring important physical effects such as temperature, their numerical simulation serves as a first test case for these more complicated flows. A well-known example is the so-called "Pitch drop experiment", which measures the flow of a piece of pitch that can be modelled with these equations. Also in the field of geodynamics, magma dynamics are modelled by similar equations [1, 2], as is the dynamics of the mantle at global scales [3, 4].

In recent years there have been many advances in the development of new architectures (e.g., Intel's Xeon Phi) providing large heterogeneous systems with large numbers of CPUs and GPUs, prompting us to adapt our codes to fully take advantage of the possible parallelisation. This has caused an increasing interest in structured-grid approaches as these approaches can not only be parallelised very naturally, but also come with a very logical structure hiding a lot of memory storage overhead in the way the data is stored. Also, the regular access patterns enables us to both write GPU codes (as we can directly address the memory) and it allows for predictable caching increasing the overall efficiency. This reduces the memory and communication cost

dramatically, making such approaches an ideal candidate for these new architectures.

Structured-grid approaches have been developed and are in use today, with the *Black Box Multigrid (BoxMG) algorithm* [5–11] being a popular choice. BoxMG is known to effectively solve various PDEs that have been discretised on logically structured grids in two or three dimensions. As the name suggests, this algorithm is intended as a "black box", i.e., the user only needs to provide a fine-grid discretisation, a right-hand side and an initial guess for the solution. BoxMG uses a fixed coarse-grid structure and, thus, can be efficiently implemented using structured data representations. This also allows the use of direct addressing, i.e., having the actual data object exposed to the algorithm, typically leading to better efficiency than when using indirect addressing.

In this thesis, we extend the BoxMG approach to the linear systems that arise from a Q2-Q1 discretisation on a structured mesh in 2D. Based on the already observed advantages of the structured-grid algorithms in terms of performance and due to the regular geometric structure of the underlying mesh, we can expect to see good performance of these algorithms when moving our computations to the GPU.

1.1 Outline of Thesis

Poisson's Equation, written in two dimensions as

$$-\Delta u = -u_{xx} - u_{yy} = f(x, y),$$

is an elliptic partial differential equation of degree 2, named after the French mathematician and physicist Simon Denis Poisson. Even though it is of rather simple nature, this type of equation is of importance in various disciplines, e.g., in electrostatics and Newtonian gravity.

Using this equation, in Section 2.1, we derive a finite-difference discretisation. This discretisation allows us to employ computers in our quest to find a solution. One of the most common ways to solve such an equation is by means of *relaxation*. We will consider an analysis of two of the most popular relaxation schemes, *weighted Jacobi* in Section 2.3 and *red-black Gauss-Seidel* in Section 2.4, and demonstrate their shortcomings. In particular, in Section 2.5, we will highlight their issue with reducing smooth error components found in any computed approximation.

Following the analysis of the two relaxation schemes, we will develop the *two-grid* algorithm, in Section 2.6, in a first attempt to complement these approaches. The two-grid algorithm tries to deal with the smooth components of the error by projecting it onto a coarser grid. The step of going from a fine grid to a coarser grid typically enables us to get a better handle on the error. While we could do a simple direct solve on the coarser grid, offering some benefits for dealing with the error in an approximation, this still leads to too large a problem to be solved efficiently. A natural extension of the two-grid algorithm is, then, the *multigrid* algorithm, derived in Section 2.8. As the name already suggests, instead of working with only two grids we would work with multiple coarser grids. This allows us to get a nice handle on both the issue of smooth error components and the size of the problem. By projecting the error of an approximation onto a coarser grid and then in turn projecting the error of the error equation onto the next coarser grid repeatedly, this results in an efficient way to improve an approximation to the real solution on the finest grid. See [12, 13] for details.

Implementing this algorithm is a rather straight-forward task that has been done many times before, with our implementation presented in Section 2.9. We are particularly interested in speeding up the calculations by parallelising them using OpenMP, Section 2.10. OpenMP is a specification that allows us to employ high-level parallelism. It is based on compiler directives that are placed in the source code. There are also abstraction layers that allow the writing of a single source code for multiple parallelisation specifications. This will not be part of this thesis, for a summary on these abstractions see [14]. For Poisson's equation, we will restrict our efforts to OpenMP on multicore CPUs. The improvements achieved by this type of parallelism is shown in Section 2.11 from different angles. One of the main focuses of our analysis will be a comparison of a parallelised weighted Jacobi and a parallelised red-black Gauss-Seidel as part of our multigrid solver.

The remainder of the thesis will be devoted to the *Stokes equations*, written in two dimensions as

$$\begin{aligned} -\nabla \cdot (2\nu\epsilon(\mathbf{u})) + \nabla p &= \mathbf{f} \\ \nabla \cdot \mathbf{u} &= 0 \end{aligned}$$

over some domain $\Omega \in \mathbb{R}^2$. This type of equation often comes up, e.g., in areas of geodynamics. It is named after George Gabriel Stokes, an Irish born mathematician

and physicist. In Section 3.1, we will be deriving a finite-element discretisation of this equation over the unit square, $[0,1]^2$, using Q2-Q1 finite elements, also called Taylor-Hood finite elements, on a regular mesh [15]. Such a discretisation results in specific matrices with certain structural properties that we will pay special attention to in Section 3.2, highlighting their obvious and well-defined structure. This structure allows us to do all of our computations using stencils, eliminating any need of using unstructured sparse matrix storage schemes. Thus, we can compute our system matrix in a structured way, shown in Section 3.3, saving large amounts of time that are typically spent on handling and computing with a general sparse matrix.

Our algorithm of choice to solve Stokes equations is *GMRES*, the Generalised Minimal Residual method, developed in Section 3.5. It is an iterative algorithm, part of the Krylov subspace methods family [16]. If we were to work with perfect precision, this algorithm would give the exact solution in a finite number of steps. It can, however, approximate solutions to systems of equations consisting of millions of unknown in a few iterations with satisfactory accuracy, given an appropriate preconditioner.

Preconditioning an algorithm, discussed in Section 3.6, is a way of making an intelligent guess based on all the information available. We will be using a multigrid preconditioner, shown in Section 3.8, to do exactly that, improving the efficiency significantly. Its algorithm is identical in nature to the multigrid algorithm used to solve Poisson's equation. However, the way we move between the different meshes needs to be adapted to take care of the more complicated element structure. Also, we won't be able to use many of the standard relaxation schemes like Jacobi or Gauss-Seidel, as these involve an inverse of a matrix that is singular for the Stokes equations. Thus, we need to find an alternative way to do relaxation. One of the most common choices is to use a *Braess-Sarazin* relaxation scheme, presented in Section 3.9 [12, 17]. After doing some rather complex matrix-matrix computations, this allows us to compute an update for any approximation in a few rather simple steps.

Following this, in Chapter 4, we will demonstrate our implementation of a GMRES solver taking advantage of the highly structured underlying mesh geometry. The first implementation is a purely sequential or serial approach, i.e., using only a single CPU. The next step is an extension of this serial algorithm to be able to employ a single GPU to do most of the computations. In Chapter 5, we will compare these two approaches giving us a nice insight of the possibilities and shortcomings from employing a GPU

based parallelism.

As a final step, in Section 5.4, we will consider a variation of the Braess-Sarazin relaxation scheme. Instead of computing a rather complicated triple matrix product, it should be theoretically possible to simply use the mass matrix of the system matrix and get comparable results. This is explored in Subsection 5.4.2, comparing the two variants and their timings.

Chapter 2

Poisson's Equation

Before we enter the world of finite elements and the Stokes equations, we will consider the simpler linear equation called *Poisson's equation*. We will attempt to solve it by means of a finite difference discretisation and a simple multigrid algorithm, to get familiar with the techniques used thereafter.

In two dimensions, Poisson's equation can be written as

$$-\Delta u = -u_{xx} - u_{yy} = f(x, y), \tag{2.1}$$

with appropriate Dirichlet boundary conditions. Our domain of choice is the unit square, $[0, 1]^2$, and we know the system has a unique solution. Despite being of rather simple nature, this type of equation is of importance in a variety of physical fields, for example, in electrostatics and Newtonian gravity.

In order to find a solution to Poisson's Equation, we could employ a Green's function or separation of variables approach. However, in real-life problems, this is usually not a practicable path to choose due to complexity in $f(x, y)$. Typically, it is enough to find a numerical approximation to the solution of Poisson's Equation. There are various methods available that do exactly that. We will be using a *finite difference* discretisation and a simple *multigrid* algorithm for that purpose. For more details, see, for example, [13].

2.1 Finite Difference Discretisation

In order to be able to numerically solve any equation, we have to find a suitable discretisation of the solution defined by finitely many points. There are many ways to do such a thing, maybe the simplest of which is a *finite difference discretisation*. We take the domain, $\{(x, y) : 0 \leq x, y \leq 1\}$ in two dimensions, and divide it into n^2 elements or partitions. For simplicity, the lengths of the elements is taken to be constant of length $h_x = h_y = \frac{1}{n}$ and, thus, we will use a simple h in place of either. Now, we can introduce the grid points $(x_i, y_j) = (ih, jh)$. The resulting grid of such a discretisation is shown in Figure 2.1.

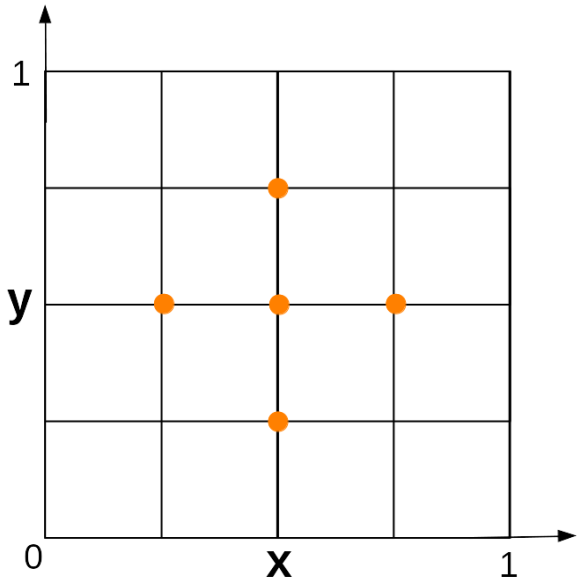


Figure 2.1: Discretised grid with typical relations between grid points, $n = 4$.

In order to actually discretise Equation (2.1), we simply replace its derivatives by second-order finite difference approximations. This leads to the system of linear equations

$$\frac{-u_{i-1,j} + 2u_{ij} - u_{i+1,j}}{h^2} + \frac{-u_{i,j-1} + 2u_{ij} - u_{i,j+1}}{h^2} = f_{ij}, \tag{2.2}$$

defining $u_{i,j}$ for $1 \leq i, j \leq n-1$, with the Dirichlet boundary conditions $u_{i0} = u_{in} = u_{0j} = u_{nj} = 0$, $0 \leq i, j \leq n$. For simplicity, we denote u_{ij} as the approximation to the true solution $u(x_i, y_j)$ and f_{ij} to be the exact value of the right-hand side $f(x_i, y_j)$. We now have $(n-1)^2$ interior grid points and the same number of unknowns in our problem. We order the points in lexicographical order by lines of constant i. If we

collect all the unknowns of the ith row of the grid in the vector $\mathbf{u_i} = (u_{i1}, ..., u_{i,n-1})^T$, $1 \leq i \leq n-1$, and similarly let $\mathbf{f_i} = (f_{i1}, ..., f_{i,n-1})^T$, then we can express the full discretised system of equations that are formed by (2.2) in block matrix form as

$$\begin{bmatrix} B & -aI & 0 & \cdots & 0 \\ -aI & B & -aI & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & -aI \\ 0 & \cdots & 0 & -aI & B \end{bmatrix} \begin{bmatrix} \mathbf{u_1} \\ \mathbf{u_2} \\ \vdots \\ \mathbf{u_{n-2}} \\ \mathbf{u_{n-1}} \end{bmatrix} = \begin{bmatrix} \mathbf{f_1} \\ \mathbf{f_2} \\ \vdots \\ \mathbf{f_{n-2}} \\ \mathbf{f_{n-1}} \end{bmatrix}. \tag{2.3}$$

This new linear system is symmetric, block tridiagonal and sparse. Its block dimension is $(n-1)$, with each diagonal block B being an $(n-1) \times (n-1)$ tridiagonal matrix containing the coefficients of the system,

$$B = \frac{1}{h^2} \begin{bmatrix} 4 & -1 & 0 & 0 & 0 & \dots & 0 \\ -1 & 4 & -1 & 0 & 0 & \dots & 0 \\ 0 & -1 & 4 & -1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 4 & -1 & 0 \\ 0 & \dots & \dots & 0 & -1 & 4 & -1 \\ 0 & \dots & \dots & \dots & 0 & -1 & 4 \end{bmatrix}. \tag{2.4}$$

Each off-diagonal block matrix is a multiple of the $(n-1) \times (n-1)$ identity matrix, I, with $a = \frac{1}{h^2}$. As a short-hand notation, when referring to the discretised system as a whole, we will denote the system as

$$A\mathbf{u} = \mathbf{f}, \tag{2.5}$$

where A is the full system matrix and $\mathbf{f}$ is the full right-hand side. The vector $\mathbf{u}$ is simply the collection of all $\mathbf{u_i}$'s, and the vector $\mathbf{f}$ is similarly the collection of all $\mathbf{f_i}$'s.

For each interior grid point, we can represent the associated equation in stencil form as

$$A_{i,j} = \frac{1}{h^2} \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}, \tag{2.6}$$

which emphasises the typical relation between the unknowns at any interior grid point, as also visualised in Figure 2.1.

2.2 Error and Residual Equation

Given a finite-difference discretisation, now suppose that after k iterations of some method, we have an approximate solution $\mathbf{u}^{(\mathbf{k})}$. Then we can formulate two measures of the error in our approximation. The first and most obvious way is to simply take the difference between the actual solution and our approximation,

$$\mathbf{e}^{(\mathbf{k})} = \mathbf{u} - \mathbf{u}^{(\mathbf{k})}. \tag{2.7}$$

This measure is simply called the *error* or *algebraic error*. Taking the 2-norm (or Euclidean norm) of the error gives us a concrete number expressing the error in our approximation,

$$||\mathbf{e}^{(\mathbf{k})}||_2 = \left(\sum_{j=1}^{(n-1)^2} (e_j^{(k)})^2 \right)^{1/2}. \tag{2.8}$$

As we can see, in order to calculate the error $\mathbf{e}^{(\mathbf{k})}$, we need to know the exact solution itself. Unfortunately, this typically isn't the case. Thus, instead of the error, one often uses the *residual* as a measure of how well we are doing,

$$\mathbf{r}^{(\mathbf{k})} = \mathbf{f} - A\mathbf{u}^{(\mathbf{k})}. \tag{2.9}$$

The residual simply expresses by how much our approximation $\mathbf{u}^{(\mathbf{k})}$ fails to satisfy the original system (2.5). Again using the 2-norm (2.8), the size of the overall residual can be expressed using a single number. Since the system has a unique solution, we have that $\mathbf{r}^{(\mathbf{k})} = \mathbf{0}$ only if $\mathbf{e}^{(\mathbf{k})} = \mathbf{0}$. Looking closely at the properties of the error and the residual, we can see that the error satisfies a set of equations related to the actual solution and right-hand side. This leads us to the *residual equation*,

$$A\mathbf{e}^{(\mathbf{k})} = A(\mathbf{u} - \mathbf{u}^{(\mathbf{k})}) = A\mathbf{u} - A\mathbf{u}^{(\mathbf{k})} = \mathbf{f} - A\mathbf{u}^{(\mathbf{k})} = \mathbf{r}^{(\mathbf{k})}. \tag{2.10}$$

Now assume we have some computed approximation $\mathbf{u}^{(\mathbf{k})}$ to $\mathbf{u}$. It is very easy to compute the residual $\mathbf{r}^{(\mathbf{k})} = \mathbf{f} - A\mathbf{u}^{(\mathbf{k})}$. In order to improve our approximation $\mathbf{u}^{(\mathbf{k})}$, we might solve the residual equation for the error at step k, $\mathbf{e}^{(\mathbf{k})}$, and then compute

a new approximation $\mathbf{u^{(k+1)}}$ by correcting it using the error information,

$$\mathbf{u^{(k+1)}} = \mathbf{u^{(k)}} + \mathbf{e^{(k)}}. \tag{2.11}$$

Since the exact error, $\mathbf{e^{(k)}}$, is difficult to find, we generally solve only for an approximation of it, as in the relaxation schemes that follow.

2.3 (Weighted) Jacobi

There are many relaxation schemes that go along these lines of residual correction. One of the most popular choices is the Jacobi method. The normal Jacobi update is a simple calculation that takes in the current approximation for the four neighbouring unknowns and the component of the right-hand side corresponding to the current point, much in the same way as illustrated in (2.6). In component form, it can be written as

$$u_{ij}^{(k+1)} = \frac{1}{4}(u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)} + u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + h^2 f_{ij}), \quad 1 \leq i, j \leq n-1. \tag{2.12}$$

However, for easier notation and to be able to see better what is going on, we will be using matrix notation. For Jacobi, the matrix A is split into

$$A = D - L - U, \tag{2.13}$$

where D is the diagonal of A, and $-L$ and $-U$ are the strictly lower and upper triangular parts of A. Substituting $D - L - U$ in place of A, we get

$$(D - L - U)\mathbf{u} = \mathbf{f}. \tag{2.14}$$

Isolating the diagonal terms of A and multiplying across by D^{-1} gives

$$\mathbf{u} = D^{-1}(L + U)\mathbf{u} + D^{-1}\mathbf{f}. \tag{2.15}$$

From this, we can define the Jacobi iteration matrix as

$$\begin{aligned} R_J &= D^{-1}(L + U) \\ &= I - D^{-1}A, \end{aligned} \tag{2.16}$$

leading to the matrix form of the Jacobi method

$$\mathbf{u^{(k+1)}} = R_J \mathbf{u^{(k)}} + D^{-1}\mathbf{f}. \tag{2.17}$$

We will now do a simple yet important modification to this Jacobi iteration. First, we do the same calculation as in (2.12), but store the intermediate result in u^*_{ij}. Then we can form a weighted average of the current and new approximation,

$$\begin{aligned} u_{ij}^{(k+1)} &= (1-\omega)u_{ij}^{(k)} + \omega u^*_{ij} \\ &= u_{ij}^{(k)} + \omega(u^*_{ij} - u_{ij}^{(k)}), \quad 1 \le i,j \le n-1, \end{aligned} \tag{2.18}$$

where $\omega \in \mathbb{R}$ is a weighting factor that has to be chosen. Again, denoting the above iteration in matrix form gives us

$$\mathbf{u^{(k+1)}} = R_\omega \mathbf{u^{(k)}} + \omega D^{-1}\mathbf{f}, \tag{2.19}$$

with $R_\omega = (1-\omega)I + \omega R_J = I - \omega D^{-1}A$, the weighted Jacobi iteration matrix. Using such an iteration matrix, R, we can easily derive the following recurrence relation for the error,

$$\mathbf{e^{(k+1)}} = R\mathbf{e^{(k)}}, \tag{2.20}$$

which we will use in the analysis of convergence of these schemes.

2.4 (Red-Black) Gauss-Seidel

Going from the Jacobi method to the Gauss-Seidel method, we only have to do a minor change to the algorithm: components of the new approximation are used as soon as they are computed and available. In fact, doing so removes the need to store the intermediate results of the new approximation in a separate array. Intermediate results are simply stored in $\mathbf{u}$, overwriting the previous component and, thus, reducing the storage cost required.

Denoting the iteration update in a similar fashion as for (2.12),

$$u_{ij}^{(k+1)} = \frac{1}{4}(u_{i,j-1}^{(k+1)} + u_{i-1,j}^{(k+1)} + u_{i,j+1}^{(k)} + u_{i+1,j}^{(k)} + h^2 f_{ij}), \quad 1 \le i,j \le n-1, \tag{2.21}$$

we can express the method in matrix notation as

$$\mathbf{u}^{(\mathbf{k+1})} = R_G \mathbf{u}^{(\mathbf{k})} + (D - L)^{-1}\mathbf{f}, \tag{2.22}$$

where $R_G = (D - L)^{-1}U$.

From (2.3) and (2.4), we know that the diagonal of the system matrix A is positive with all off-diagonal entries being ≤ 0. Since A is diagonally dominant, it is an *M-matrix*. With this condition given, it can be shown that, if both methods are converging, then the Gauss-Seidel method always converges faster than the Jacobi method. However, as we eventually want to do some parallelising of the algorithm, the Jacobi method gives the obvious advantage that it doesn't matter in which order the updates are computed. For Gauss-Seidel, this is not the case anymore, as it now matters in which order new components are computed.

There are various ways to introduce a similar property to the Gauss-Seidel method. One of the most popular ways leads to the so-called *red-black Gauss-Seidel method*, an illustration of which is given in Figure 2.2. For the red-black Gauss-Seidel method,

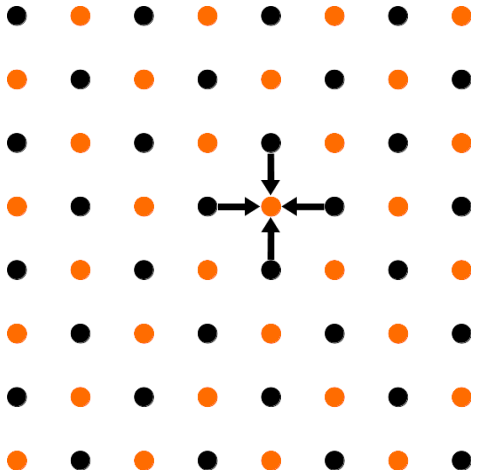


Figure 2.2: Red-Black ordering of nodes

we first update all the "even" (red) components, i.e., whenever $i + j$ is even, following which we update all the "odd" (black) components, i.e., whenever $i + j$ is odd, both times applying the same calculations as for the standard Gauss-Seidel iteration, (2.21). After doing these two distinct sets of calculations once, every entry in $\mathbf{u}$ will have an updated value, i.e., it is a complete partition.

In particular when looking at Figure 2.2, it becomes clear why this is of advantage for parallelising the algorithm: In order to compute an update for a node with even

index sum, $i + j$, all we need are values of nodes with odd index sum and vice versa. Thus, it is possible to compute an update to all the nodes with even index sum simultaneously, followed by an update to all the nodes with odd index sum simultaneously.

There are many more relaxation methods. However, we will only be comparing weighted Jacobi and red-black Gauss-Seidel for the Poisson problem in this chapter.

2.5 Smooth Error Components

We will be using the weighted Jacobi method as an example of why relaxation schemes are not the final answer in finding a good approximation to the solution of essentially any system. Recalling the weighted Jacobi iteration matrix $R_\omega = (1-\omega)I + \omega R_J$, we can rewrite R_ω as

$$R_\omega = I - \frac{\omega h^2}{4} \begin{bmatrix} B & -I & & & \\ -I & B & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & B & -I \\ & & & -I & B \end{bmatrix}. \tag{2.23}$$

Considering R_ω in this form, a relationship between the eigenvalues of R_ω and A emerges,

$$\lambda(R_\omega) = 1 - \frac{\omega h^2}{4}\lambda(A). \tag{2.24}$$

Thus, we first need to compute the eigenvalues of A,

$$\lambda_{kl}(A) = \frac{4}{h^2}\sin^2\left(\frac{k\pi}{2n}\right) + \frac{4}{h^2}\sin^2\left(\frac{l\pi}{2n}\right), \quad 1 \le k, l \le n-1, \tag{2.25}$$

and thus, the eigenvalues of the iteration matrix R_ω are given by

$$\lambda_{kl}(R_\omega) = 1 - \omega\left[\sin^2\left(\frac{k\pi}{2n}\right) + \sin^2\left(\frac{l\pi}{2n}\right)\right]. \tag{2.26}$$

Clearly, for any value $0 < \omega \le 1$, we have $|\lambda_{kl}(R_\omega)| < 1$. This also gives us that for any such ω

$$\lambda_{1,1}(R_\omega) = 1 - 2\omega \sin^2\left(\frac{\pi}{2n}\right) = 1 - 2\omega \sin^2\left(\frac{\pi h}{2}\right) \approx 1 - \frac{\omega \pi^2 h^2}{2}, \tag{2.27}$$

where $\lambda_{1,1}$ is the eigenvalue associated with the smoothest mode. This implies that $\lambda_{1,1}$ will always be close to 1. This is a problem, as the largest (in absolute value) eigenvalue of the system matrix, called the *Spectral Radius*, ρ, is an asymptotic measure of convergence. It predicts the worst-case error reduction over many iterations. It tells us approximately how many iterations are required to reduce the error by a factor of 10^{-d}, i.e., by d decimal digits. This is the case when the condition (2.28) is approximately satisfied,

$$[\rho(R))]^m \leq 10^{-d} \tag{2.28}$$

where R is the iteration matrix of the chosen method, m is the number of iterations necessary, and d is the number of decimal digits we want to reduce the error by. Solving for m we get

$$m \geq -\frac{d}{\log_{10}[\rho(R)]}. \tag{2.29}$$

The quantity $-\log_{10}(\rho(R))$ is called the asymptotic convergence rate. In order to get the approximate number of iterations required to reduce the error by one decimal digit, one simply has to consider its reciprocal. It now becomes clear, why an eigenvalue close to 1 is a problem, as then the convergence rate decreases drastically. This can also be seen when considering the convergence factor per iteration, ρ_k, expressed as

$$\rho_k = \frac{|e^{(k)}|}{|e^{(k-1)}|}. \tag{2.30}$$

We do not want a convergence factor close to 1, as this would signal very slow convergence. A plot of the convergence factor for the first 100 iterations of the weighted Jacobi scheme is shown in Figure 2.3, where we can see that the convergence factor starts out well below 1 but rises very quickly and, after only a few iterations, becomes very close to 1.

Often times, in order to get a better approximation, one would decrease the grid spacing h. However, this will only worsen the convergence of the smooth components of the error as it would push the largest eigenvalue of A closer and closer to 1. Thus, we need something to complement such relaxation schemes, something that can efficiently

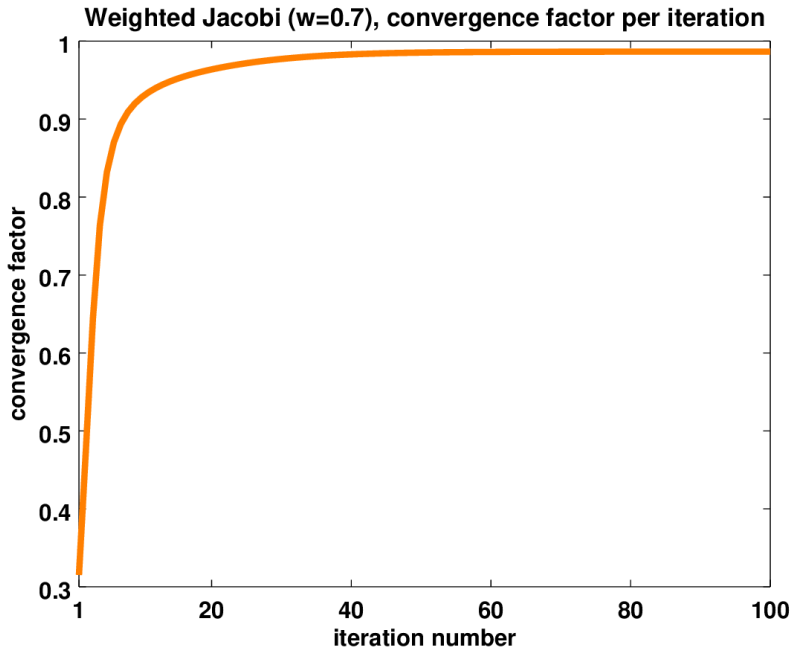


Figure 2.3: Convergence factor (per iteration) of Weighted Jacobi, $\omega = 0.7$

dampen the smooth components of the error.

Even though the above analysis was done for the weighted Jacobi method, it is important to note that the Gauss-Seidel method is qualitatively similar, and also suffers from this problem.

2.6 Two-Grid

One idea for complementing the relaxation schemes is to first use a standard relaxation scheme until the smooth components of the error are dominant. These smooth error components are then projected onto a coarser grid. In doing so, smooth error components typically appear more oscillatory on coarser grids. This then allows again the use of a simple relaxation scheme to dampen the coarse-grid error. Using the information obtained therewith, we can correct our approximation on the finer grid. This is the fundamental idea of the *two-grid method*, illustrated in Figure 2.4. There are many ways to choose the coarse grid in a two-grid method. The most straightforward way is to simply halve the degrees of freedom in each dimension, i.e., the coarse grid has twice the grid spacing of the fine grid.

Thus, we only have to know two things in order to proceed: How to move from

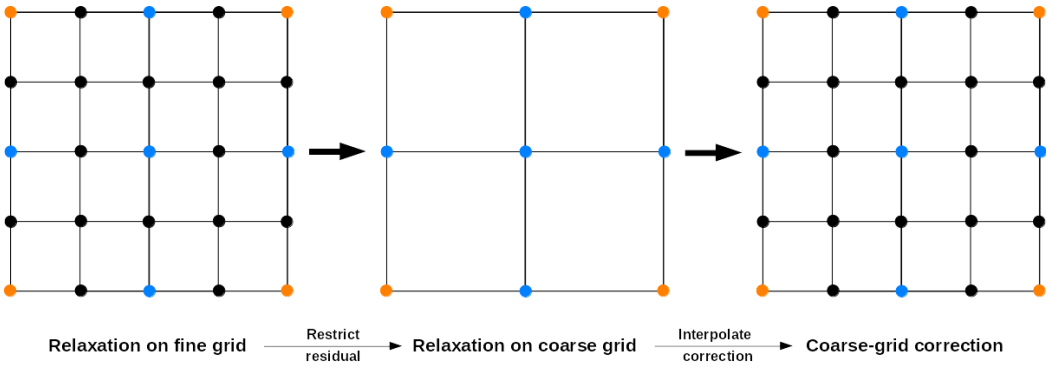


Figure 2.4: Two-grid method

the fine grid to the coarse grid (restriction), and back (interpolation)?

2.7 Restriction and Interpolation

There are many possible ways we can choose to handle the restriction and interpolation of the vectors defined on the grid. For restriction, we will halve the degrees of freedom and then take a weighted average of the fine-grid nearest neighbours. This type of restriction is called *full weighting*, illustrated in Figure 2.5.

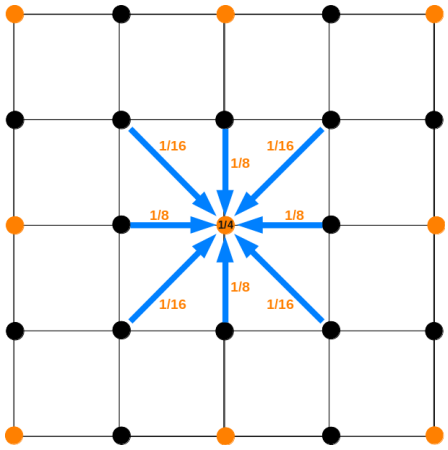


Figure 2.5: Weighted Restriction of fine-grid points (black) to coarse-grid points (orange)

Denoting a component of the finer grid as v_{ij}^h and a component of the coarser grid as

v_{ij}^{2h}, we can calculate the coarse-grid values by

$$\begin{aligned} v_{ij}^{2h} = \frac{1}{16}[&v_{2i-1,2j-1}^{h} + v_{2i-1,2j+1}^{h} + v_{2i+1,2j-1}^{h} + v_{2i+1,2j+1}^{h} \\ &+ 2(v_{2i,2j-1}^{h} + v_{2i,2j+1}^{h} + v_{2i-1,2j}^{h} + v_{2i+1,2j}^{h}) \\ &+ 4v_{2i,2j}^{h}] \end{aligned} \tag{2.31}$$

for $1 \leq i, j \leq \frac{n}{2} - 1$.

Linear interpolation follows a similar idea. Each fine-grid component is either the same as a coarse-grid point, or is a weighted average of neighbouring coarse grid points. It is worth noting that interpolating a coarse-grid error approximation to an oscillatory error on the finer grid does not work very well. In order to achieve good performance, the error on the fine grid should be smooth. An illustration of weighted interpolation can be seen in Figure 2.6.

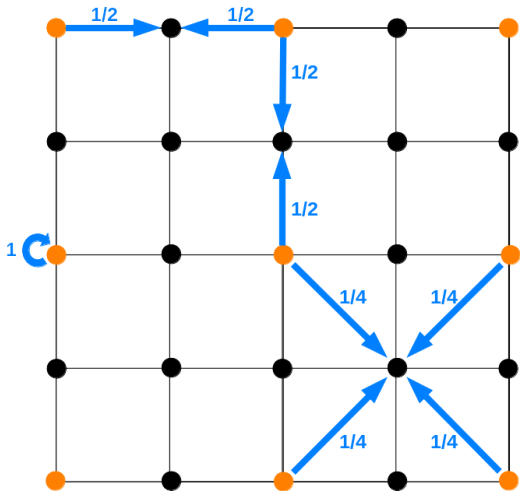


Figure 2.6: Weighted Interpolation of coarse-grid points (orange) to fine-grid points (black)

The new components of the fine grid are then calculated by

$$\begin{aligned}
v^h_{2i,2j} &= v^{2h}_{ij}, && 1 \le i,j \le \frac{n}{2}-1 \\
v^h_{2i+1,2j} &= \frac{1}{2}(v^{2h}_{ij} + v^{2h}_{i+1,j}), && 0 \le i \le \frac{n}{2}-1, 1 \le j \le \frac{n}{2}-1 \\
v^h_{2i,2j+1} &= \frac{1}{2}(v^{2h}_{ij} + v^{2h}_{i,j+1}), && 1 \le i \le \frac{n}{2}-1, 0 \le j \le \frac{n}{2}-1 \\
v^h_{2i+1,2j+1} &= \frac{1}{4}(v^{2h}_{ij} + v^{2h}_{i+1,j} + v^{2h}_{i,j+1} + v^{2h}_{i+1,j+1}), && 0 \le i,j \le \frac{n}{2}-1.
\end{aligned} \tag{2.32}$$

In our discussion, we considered only the case when the coarse grid has exactly half the number of grid intervals (i.e., twice the grid spacing) as the fine grid. This is a common choice in many settings as it is a natural and easy way to do it. For some problems, a ratio of 3 instead of 2 can potentially give an improvement in convergence [18]. However, for simplicity and convenience, we will stick to the choice of a ratio of 2. Note that these choices satisfy the requirement that the combined order of interpolation and restriction is greater than or equal to the order of the equation (2 in the case of Poisson) [12, p. 295].

The weights used for restriction and interpolation can easily be modified for the case when h_x differs from h_y, i.e., when the mesh consists of rectangles instead of squares. This, however, will not be further explored in this thesis.

Even though the two-grid algorithm with the above interpolation and restriction already yields a significant improvement over direct methods, it is not the final answer in the quest to complement the relaxation schemes. This is due to the simple fact that even though the coarser grid has half the size of the finer grid, it is still a large problem to be solved, typically still too big to solve it efficiently. Thus, it is merely a first step in this quest. However, extending two-grid to something that also takes care of this issue is rather straightforward. This leads us to the multigrid algorithm.

2.8 Multigrid

As the name already suggests, in multigrid we do not only have two grids (a fine and a coarse one), but we now deal with multiple grids, as illustrated in Figure 2.7.

Starting out with the actual problem on the finest grid, the error equation for each grid is repeatedly restricted onto a coarser grid where it is relaxed each time, until a coarsest grid with dimension chosen beforehand is reached. Once the coarsest of all

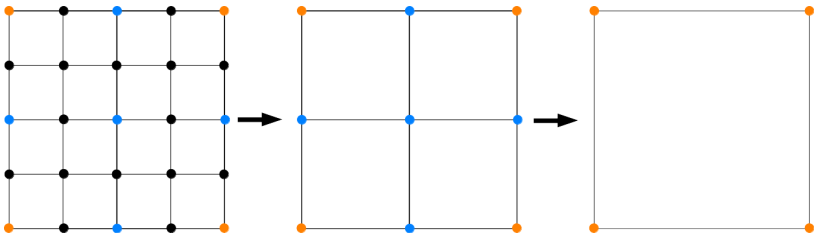


Figure 2.7: Repeated Grid Coarsening

grids is reached, the error correction is repeatedly interpolated to the next finer grid where it is again relaxed. This is done until the actual problem on the finest grid is reached again. This results in the so-called V-cycle, shown in Figure 2.8.

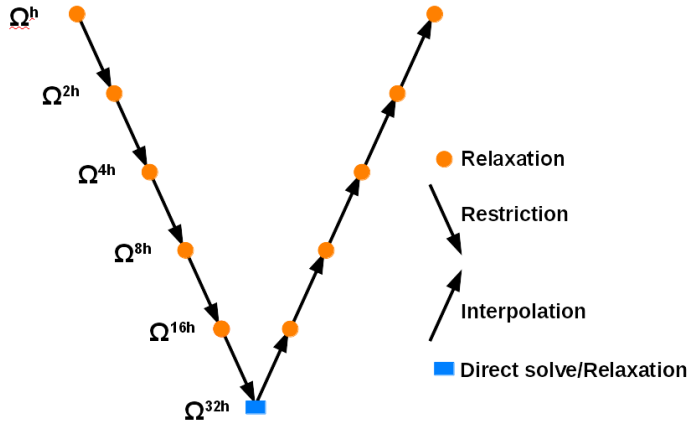


Figure 2.8: Multigrid V-cycle

On the coarsest grid, before interpolating the solution, we could do what is typically done and do a direct solve of that system. Since the size of the problem at that stage is very small due to repeated coarsening, this comes at a very low cost. Alternatively, it is also possible to only solve the system approximately, e.g., by relaxing several times.

2.9 Implementation

Our language of choice to implement a multigrid solver for Poisson's equation was C++. As all the pieces of the puzzle combined compose a rather simple and straightforward algorithm, it was done without the use of any classes. Instead, the main part of the implementation is a simple function that is called recursively for each new level, Code 2.1.

It takes various parameters:

- v: vector containing the current approximation
- f: vector containing the right-hand side
- *level*: the current level in the multigrid V-cycle
- *nu1* & *nu2*: Number of pre- and post-relaxation runs
- *func*: smoother to use (Jacobi or Gauss-Seidel)

Inside of the function body, we first run the pre-relaxation iterations, as they are always performed, no matter on which level we currently are (lines 3-4). If the current function call is on the coarsest level, then we jump directly to post-relaxation (lines 16-17) and pass the correction back up. At this point, on the coarsest level, it would also be possible to perform a direct solve.

Code 2.1: V-cycle function definition, simplified

```
1  double *vcycle(double *v, double *f, int level, int nu1, int nu2, void(*
       func)(double*,double*,int)) {

3      for(int j = 0; j < nu1; ++j)
4          func(v,f,level);

6      if(level > 2) {

8          // ... Calculate residual ...

10         double *new_f = restrict(residual, level);
11         double *vec_asc = vcycle(new_v, new_f, level-1, nu1, nu2, func);
12         interpolate(v,vec_asc, level-1);

14     }

16     for(int i = 0; i < nu2; ++i)
17         func(v,f,level);

19     return v;

21 }
```

If we are not on the coarsest level, we are performing three steps:

1. Move error equation to coarser grid.
   We calculate the residual of the current approximation (lines around 8), which we will restrict to the coarser grid (line 10) and treat as the new right-hand side on the coarser grid.

2. Recursive function call.
   Having restricted the residual to the coarser grid as new right-hand side, we then call the V-cycle function again (line 11). The new initial guess on the coarser grid is a zero initial guess; we are optimistic and start by guessing that we are at the right solution before typically being proven wrong.

3. Correct approximation with solution from coarser grid.
   Whatever solution we get on our coarser grid, we first interpolate it to the (finer) grid we are currently on and then simply add it on to our previous approximation (line 12).

Both the *restrict()* and *interpolate()* functions are of very simple nature, they consist of a few small loops implementing exactly what Equations (2.31) and (2.32) are stating. The only addition to the *interpolate()* function is that it adds the interpolated values to the already existing values - shown in Code 2.2 -, i.e., it takes proper care of the coarse-grid correction.

Code 2.2: V-cycle: adding on of interpolated values

```
1 v_h = v_h + interpolate(v_2h);
```

2.10 Parallelisation with OpenMP

Writing an implementation of a V-cycle solver is not a difficult task and has been done many times by many people. However, we will also be doing some parallelising of our code and compare performance between the serial and the parallel case. We will be restricting our efforts to multicore CPUs and, thus, employ OpenMP to improve the performance of our code.

"OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs" [19, OMPAPI.General.01].

OpenMP parallelises code by using multithreading. In multithreading, a master thread forks a specific number of slave threads and the system divides the task among them. All of the threads then run concurrently, with the option of having variables in shared memory accessible by all threads simultaneously.

The task of dividing the main task amongst all the slave threads can be controlled by a scheduler in OpenMP. There are a four different types of schedulers available [20]:

1. *static*: Iterations are divided into "chunks" of a certain size. These chunks are then assigned to threads in the team in round-robin fashion in order of thread number.

2. *dynamic*: Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be distributed.

3. *guided*: Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be assigned. The chunk sizes start large and as chunks are scheduled shrink to the chunk size indicated at the start.

4. *auto*: The decision regarding scheduling is delegated to the compiler and/or runtime system.

In addition to a scheduler, there are also various directives to control the data access. Memory can be marked as either one of the following:

1. *shared*: A single copy of a variable used by all threads simultaneously - medium performance.

2. *private*: Each thread owns its individual copy of a variable - fast performance.

3. *atomic*: A shared variable, that is updated atomically - slow performance.

While adjusting the scheduler and data-access directives can improve the performance of the parallelised code to some extent, they are still not sufficient to achieve excellent performance for this class of algorithms.

Marking a section of code for parallelisation is done in a very simple way by adding a pre-processor directive that forces the threads to be set up and ready to go before the section is executed. In order to be able to distinguish between the threads, each one gets a unique *id* allocated, with the master thread having id 0. After the parallelised

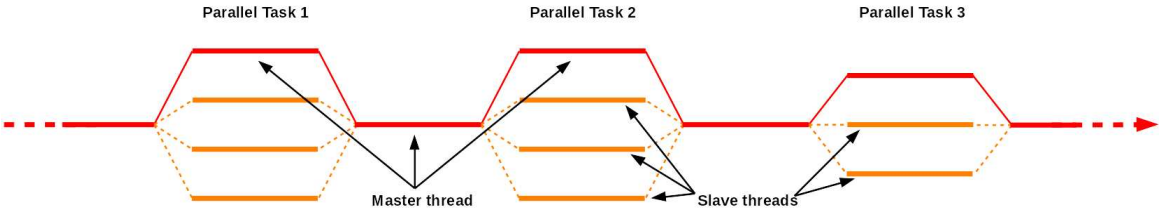


Figure 2.9: OpenMP multithreading

section has been executed, all threads join back together with the master thread which continues to run to the end of the program. Figure 2.9 visualises this behaviour.

Adding OpenMP parallelisation to some existing code is very easy and does not require any changes to the underlying code basis. Adding the already-mentioned pre-processor directives is all that is needed. An example of how to parallelise a simple for loop with OpenMP is given in Code 2.3.

Code 2.3: Parallelising a simple for loop with OpenMP

```
1 #pragma omp for shared(out)
2 for(unsigned int i = 0; i < 1e10; ++i)
3     out[i] = in1[i]+in2[i];
```

Using this type of pre-processor directive, it becomes very easy to create a highly parallelised version of our implementation of a multigrid solver for Poisson's equation. We only have to be careful to not write to the same shared memory address from two concurrent threads at the same time.

2.11 Numerical results

We ran our implementation on a machine with 16 physical and 16 virtual cores provided by two *Intel Xeon E5* CPU's. Three different grid sizes were chosen: 4096x4096, 8192x8192, 16384x16384. For each one, the code was run both *with* GCC optimisation level *-O3* and without for both the Jacobi and Gauss-Seidel relaxation methods. The results were analysed for two different criteria.

The first analysis, shown in Figure 2.10, shows the relative speed of the code. In each case it was run until the 2-norm of the residual at step k compared to the 2-norm of the initial residual was reduced by a factor of 10^{-6}. All timings were set relative to the fastest time, with the fastest time getting a value of 1 assigned to it.

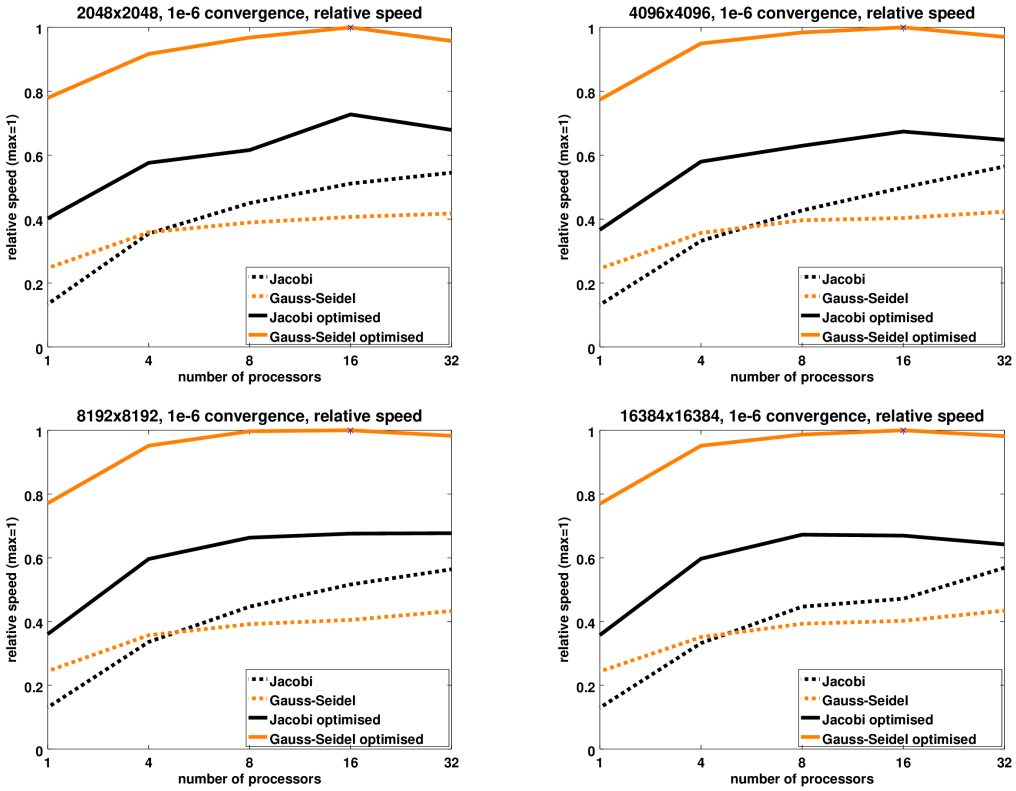


Figure 2.10: Relative speed of Jacobi and Gauss-Seidel with and without optimisation on varying number of cores

As we would expect, enabling the typical optimisations of GCC alone gives already a much better performance. The version with Gauss-Seidel is significantly faster than the version with Jacobi, as fewer iterations are required to achieve the desired convergence. However, it also becomes apparent from the graphs that with or without any GCC optimisations, Gauss-Seidel is sped up at a much lower rate, the non-optimised version eventually being the slowest one of the four choices. Considering that in order to perform a red-black Gauss-Seidel iteration two for-loops are required, this behaviour was to be expected. Given the simple structure, though, GCC is well capable of tweaking its implementation internally to give the very nice performance we can observe when using the optimisation flag.

Another interesting aspect to look at is when going from 16 cores up to 32 cores: With no optimisations, this step again improves the overall performance at least to some degree. Some of the optimisations GCC performs internally appear to be of parallelising nature, causing some threads to compete for cores leading to the slight

dip in performance.

We also analysed the data in regards to simply the possible speedup we can get by introducing OpenMP, illustrated in Figure 2.11.

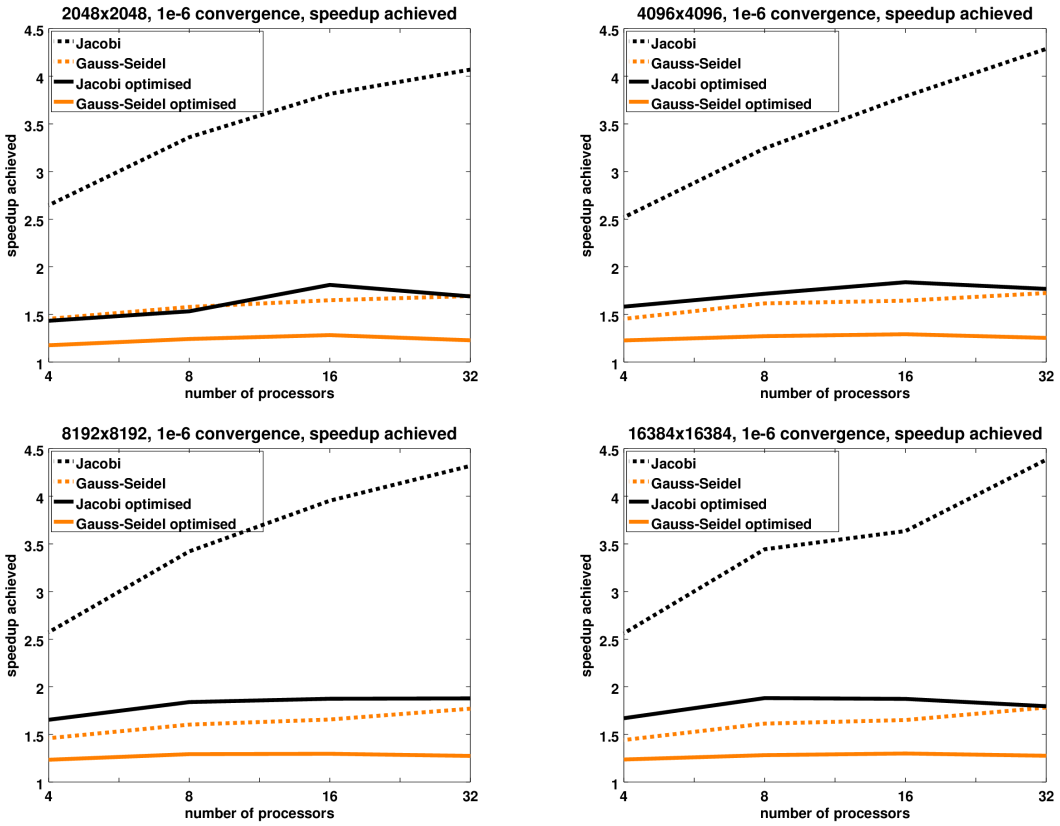


Figure 2.11: Speedup achieved by introducing OpenMP parallelisation on varying number of cores

Depending on the setup and optimisation level, the benefit from adding OpenMP is rather low. For example, when considering the Gauss-Seidel relaxation method, the highest speedup achieved by introducing OpenMP is just below 1.3. For optimised Jacobi, this factor goes up to just below 1.9. This is a little less than we might expect, in particular given the hardware setup of 32 cores. Clearly, GCC does a very good job in optimising the underlying code to give high performance without any third-party library in use.

Interestingly, when disabling all the GCC optimisations, introducing OpenMP can yield a very high speedup of up to 4.4 when using the Jacobi smoother. However, OpenMP does a rather bad job in increasing the performance of red-black Gauss-Seidel. A speedup of 1.8 is the best that seems possible. This also agrees with the

behaviour we have seen in Figure 2.10, where red-black Gauss-Seidel with no other optimisations but OpenMP eventually resulted in the slowest of the four cases.

In conclusion, OpenMP *can* indeed be an easy-to-implement way to increase the performance of the code. Especially if some parallelisation is to be added after the code has been written already. However, despite being so easy to use, the performance boost that can be obtained by adding OpenMP can remain below expectations, even when using additional control directives. A lot of it depends on the underlying hardware that is available, and the compiler optimisations enabled.

Chapter 3

Stokes equations

Stokes flow is a type of fluid flow where viscous forces are much greater than advective inertia. Flow with such properties occurs in many places in nature, as, for example, in geodynamics (for instance, the flow of lava), or the swimming movement of microorganisms. The equations of motion in this regime are called the Stokes equations, which are a simplification of the steady-state Navier-Stokes equations. We will consider the incompressible Stokes equations in the domain $\Omega \in \mathbb{R}^2$, written as

$$-\nabla \cdot (2\nu\epsilon(\mathbf{u})) + \nabla p = \mathbf{f} \tag{3.1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{3.2}$$

in combination with appropriate boundary conditions. We denote as $\mathbf{u}$ the fluid velocity, and as p the pressure, while the right-hand side, $\mathbf{f}$, is an applied external force, ν is the fluid viscosity (considered here to be constant), and $\epsilon(\mathbf{u}) = \frac{1}{2}(\nabla\mathbf{u}+\nabla\mathbf{u}^T)$ is the strain-rate tensor. As a simplification, we assume the viscosity ν to be constant, thus, we can remove the strain-rate tensor and simplify (3.1) as

$$-2\nu\nabla^2\mathbf{u} + \nabla p = \mathbf{f} \tag{3.3}$$

Along the boundary of the domain, we enforce Dirichlet boundary conditions for the normal components of the velocity, while leaving the pressure without any constraint. Considering the case of enclosed flow, along the boundary we enforce the no-flux condition

$$\mathbf{u} \cdot \mathbf{n} = 0 \text{ on } \partial\Omega, \tag{3.4}$$

where $\mathbf{n}$ is an outward-pointing unit normal vector. Define the usual spaces $\mathbf{H}_0^1(\Omega)$ as

$$\mathbf{H}_0^1(\Omega) = \{\mathbf{v} \in \mathbf{H}^1(\Omega) : \mathbf{v} \cdot \mathbf{n} = 0 \text{ on } \partial\Omega\} \tag{3.5}$$

and $L^2(\Omega)/\mathbb{R}$ as the quotient space of equivalence classes of elements of $L^2(\Omega)$ that differ by a constant. Thus, we can formulate the continuous weak form as finding $(\mathbf{u}, p) \in H_0^1 \times L^2(\Omega)/\mathbb{R}$ satisfying

$$a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) = (\mathbf{f}, \mathbf{v}) \quad \forall \mathbf{v} \in \mathbf{H}_0^1(\Omega) \tag{3.6}$$

$$b(\mathbf{u}, q) = 0 \qquad \forall q \in L^2(\Omega)/\mathbb{R}, \tag{3.7}$$

where

$$a(\mathbf{u}, \mathbf{v}) = 2\nu \int_\Omega \nabla\mathbf{u} : \nabla\mathbf{v}, \tag{3.8}$$

$$b(\mathbf{v}, p) = \int_\Omega p\nabla \cdot \mathbf{v}. \tag{3.9}$$

In order to employ any numerical method or algorithm to find a solution to the weak formulation (3.6) and (3.7), we first have to perform a suitable discretisation.

3.1 Discretisation

Before considering a discretisation, we must specify a domain to be used, which depends on the application. Here, we will stick to the simplest case and consider the unit square, $[0, 1]^2$. There is a very natural way to think about a discretisation of a 2-dimensional space over any domain: A simple grid or mesh consisting of regular shapes. Two of the most popular choices of shape are triangles and rectangles. Depending on the domain, one or the other may be a preferable choice: If the domain is rather irregular, e.g., containing curved boundaries, then it is possible to cover the whole domain more accurately using triangles. If, however, the domain is of very regular nature, e.g., a simple quadrilateral (or rectangular) domain, then rectangular elements might offer slightly better accuracy. Since we are indeed working with a regular square domain, the unit square, we will be using rectangles for our elements. In particular, we will employ the so-called Taylor-Hood elements, or Q2-Q1 elements, on a uniform mesh.

Q2-Q1 elements are called Q2-Q1 as they use biquadratic polynomials (Q2) as basis functions for the velocity and bilinear polynomials (Q1) as basis functions for the pressure that are both continuous across element boundaries. In one dimension, a typical set of basis functions, Φ, is illustrated in Figure 3.1. Each is a piecewise defined biquadratic polynomial, with a value of 1 at one degree of freedom, and 0 at every other node. Taking the same one-dimensional basis functions in both the x

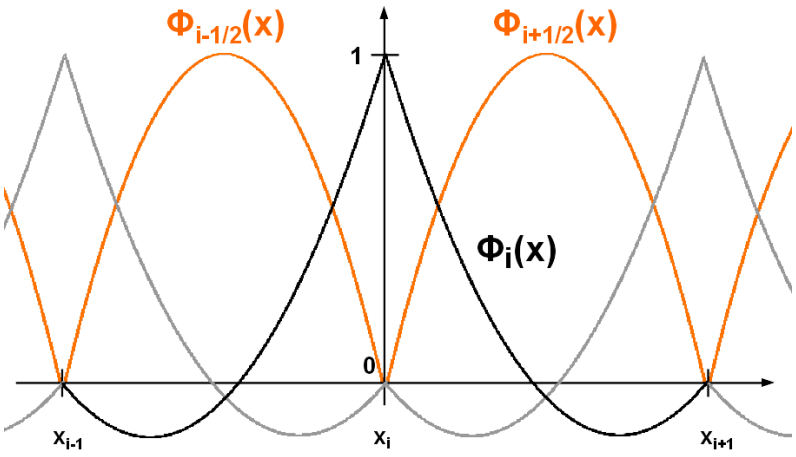


Figure 3.1: Q2 basis functions

and y directions and multiplying them together will yield the two-dimensional basis functions for the Q2 elements.

Similarly for the Q1 elements, a typical set of basis functions, Ψ, in one dimension is illustrated in Figure 3.2, as piecewise defined linear polynomials, each with a value

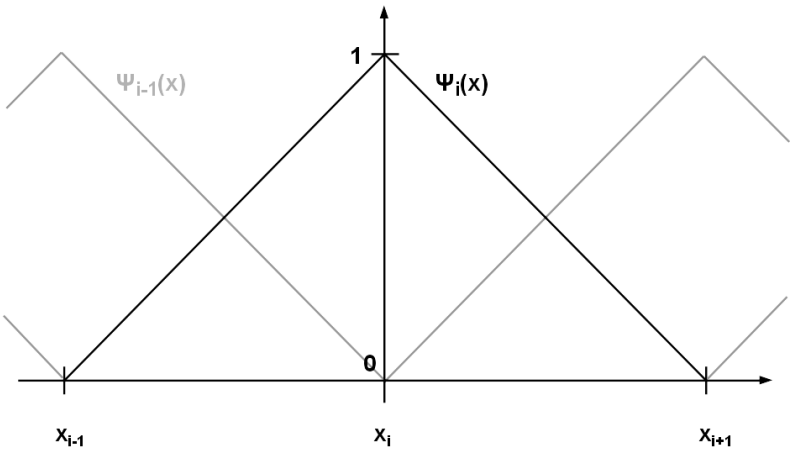


Figure 3.2: Q1 basis functions

of 1 at one node and 0 everywhere else. Again, taking the same basis functions in both the x and y directions and multiplying both will yield the two-dimensional basis functions for the Q1 elements.

Having rectangles as elements and the respective basis functions defined, we simply cover the whole domain with equally spaced rectangles. This works out perfectly for us, as we are working with a square domain. A 2x2 element patch out of the whole grid is illustrated in Figure 3.3. In order to accommodate our biquadratic polynomials,

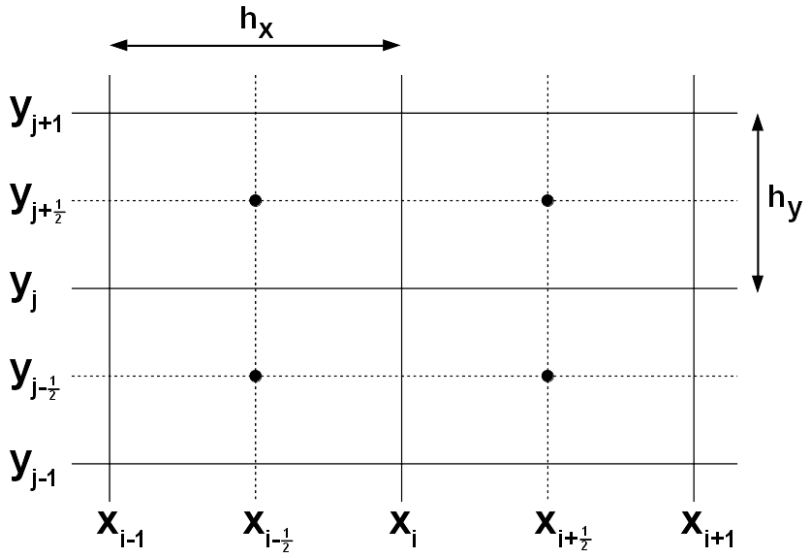


Figure 3.3: 2x2 element patch

we need a total of 9 degrees of freedom per element: 4 nodes, 2 x-edges, 2 y-edges and 1 cell center. This is necessary as we need at least 3 degrees of freedom in each dimension for a biquadratic polynomial to be well-defined. Equivalently, whenever we are looking at the bilinear polynomials, only 2 degrees of freedom in each dimension are necessary for a well-defined representation of the basis functions.

This discretisation of the Stokes equations directly relates back to the weak form defining matrices L and B by

$$\mathbf{v}^{\mathbf{T}} L\mathbf{u} = a(\mathbf{u}, \mathbf{v}), \tag{3.10}$$

$$\mathbf{v}^{\mathbf{T}} Bp = b(\mathbf{v}, p), \tag{3.11}$$

$$q^T B^T \mathbf{u} = b(\mathbf{u}, q), \tag{3.12}$$

where it is important to note that the $\mathbf{u}$, $\mathbf{v}$, p, and q on the right-hand side refer to

the functions $\mathbf{u}(x, y)$, $\mathbf{v}(x, y)$, $p(x, y)$, and $q(x, y)$, whereas the $\mathbf{u}$, $\mathbf{v}$, p, and q on the left-hand side refer to the solution vectors containing the coefficients of the functions with respect to the basis functions.

Given necessary conditions for optimality, the weak form can be expressed as the linear system

$$Ax = \begin{bmatrix} L & B \\ B^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} \mathbf{f_u} \\ f_p \end{bmatrix} = b, , \tag{3.13}$$

where L is the discretisation of the Laplacian, B is the discretisation of the gradient operator, and B^T is the discretisation of the divergence operator. The four components L, B, $\mathbf{u}$, and $\mathbf{f_u}$ can be broken down into their respective x and y components,

$$\begin{aligned} L &= \begin{bmatrix} L_x & 0 \\ 0 & L_y \end{bmatrix}, \quad B = \begin{bmatrix} B_x \\ B_y \end{bmatrix} \\ \mathbf{u} &= \begin{bmatrix} u_x \\ u_y \end{bmatrix}, \quad \mathbf{f_u} = \begin{bmatrix} f_x \\ f_y \end{bmatrix} \end{aligned} .$$

This is important to know as we will be calculating the entries of the matrices in the x and y directions separately.

3.2 Structural Properties

Using such a discretisation of the domain leads to a highly structured format for the linear system (3.13): Any degree of freedom (i.e., node, edge or cell-center) only interacts at most with all other degrees of freedom within no more than the four surrounding elements (2x2 element patch). This stems from the fact that the basis functions in use are zero everywhere else but the elements containing the node associated with the degree of freedom itself. Thus, the total number of non-zeros in each row of the matrix corresponding to each degree of freedom are of fixed size and independent of the grid size. They are summarised in Table 3.1 and an illustration can be found in Figure 3.4.

If all the elements are sorted in a structured way, it becomes apparent that we will always know exactly which values at which entries in the system matrix we have to

	nodes	x-edges	y-edges	centers	**Total**
nodes	9	6	6	4	**25**
x-edges	6	3	4	2	**15**
y-edges	6	4	3	2	**15**
centers	4	2	2	1	**9**

Table 3.1: Number of non-zeros per row (for velocity degrees of freedom)

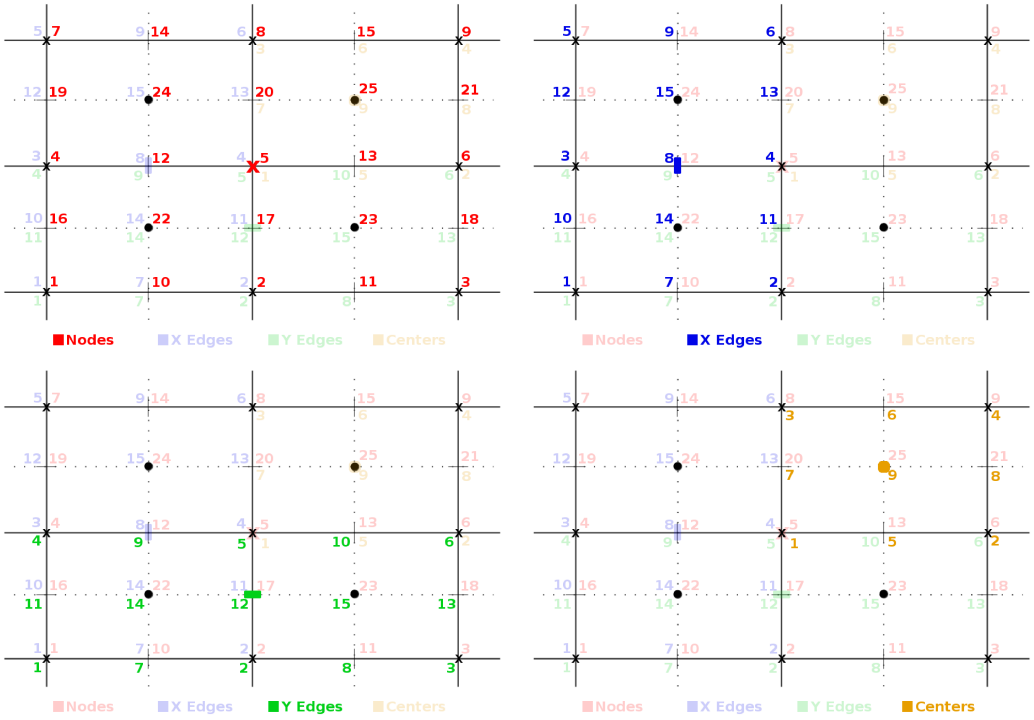


Figure 3.4: The four cases yielding non-zero entries in the system matrix (velocity degrees of freedom)

consider. This is clear, especially when looking at Figure 3.4. Thus, for each degree of freedom, we can safely take any of the other entries in the matrix to be zero and to remain zero. Additionally, realising that these numbers do not depend on the size of the problem but are fixed, we can see clearly that taking advantage of this property simplifies our calculations tremendously. Given these well-defined structural properties, we will eventually be able to parallelise all our matrix-vector calculations very nicely, as the individual calculations for each degree of freedom naturally are separated into independent parts each containing only very few calculations.

3.3 Computing the System Matrix

The entries of the matrices that compose (3.13) can be easily calculated. In particular since we only have to compute a few entries in each row. Let (i, j) be the index (row, column) of one degree of freedom. Then we can define the Q2 basis functions as

$$\begin{aligned}
\phi_{i-1}(x) &= \frac{2}{h^2}(x - x_{i-\frac{1}{2}})(x - x_i), & x_{i-1} \le x \le x_i \\
\phi_{i-\frac{1}{2}}(x) &= \frac{4}{h^2}(x - x_{i-1})(x - x_i), & x_{i-1} \le x \le x_i \\
\phi_i(x) &= \frac{2}{h^2}(x - x_{i-1})(x - x_{i-\frac{1}{2}}), & x_{i-1} \le x \le x_i \\
\phi_i(x) &= \frac{2}{h^2}(x - x_{i+\frac{1}{2}})(x - x_{i+1}), & x_i \le x \le x_{i+1} \\
\phi_{i+\frac{1}{2}}(x) &= \frac{4}{h^2}(x - x_i)(x - x_{i+1}), & x_i \le x \le x_{i+1} \\
\phi_{i+1}(x) &= \frac{2}{h^2}(x - x_i)(x - x_{i+\frac{1}{2}}), & x_i \le x \le x_{i+1}
\end{aligned} \tag{3.14}$$

in the x direction. Similar calculations (replacing x by y) will give the basis functions in the y direction.

Taking (k, l) to be another degree of freedom, we can then compute the entry of L_x in the row corresponding to (i, j) and in the column corresponding to (k, l) by

$$\begin{aligned}
(L_x)_{(i,j),(k,l)} &= \int_\Omega \nabla\phi_{(k,l)}(x, y) \cdot \nabla\phi_{(i,j)}(x, y) d\Omega \\
&= \int_\Omega (\partial_x\phi_k(x))\phi_l(y)(\partial_x\phi_i(x))\phi_j(y) + \phi_k(x)(\partial_y\phi_l(y))\phi_i(x)(\partial_y\phi_j(y)) d\Omega
\end{aligned} \tag{3.15}$$

where we use that

$$\phi_{(k,l)}(x, y) = \phi_k(x)\phi_l(y). \tag{3.16}$$

Taking advantage of the structural properties of L_x, this integral can be simplified

and broken down into

$$\begin{aligned}(L_x)_{(i,j),(k,l)} = &\left[\int_{x_{i-1}}^{x_{i+1}} (\partial_x \phi_k(x))(\partial_x \phi_i(x))dx\right]\left[\int_{y_{j-1}}^{y_{j+1}} \phi_l(y)\phi_j(y)dy\right] \\ &+ \left[\int_{y_{j-1}}^{y_{j+1}} (\partial_y \phi_l(y))(\partial_y \phi_j(y))dy\right]\left[\int_{x_{i-1}}^{x_{i+1}} \phi_k(x)\phi_i(x)dx\right]\end{aligned} \tag{3.17}$$

In order to obtain the entries for the matrix L_y, we can simply do the exact same calculations as for L_x.

A similar approach can be taken to calculate the entries of the matrices B_x and B_y. Again, let (i, j) be the index of a Q2 degree degree of freedom, but now let (k, l) be the index of a Q1 degree of freedom. Define the one-dimensional Q1 basis function $\Psi_k(x)$ as

$$\Psi_k(x) = \begin{cases} \dfrac{x - x_{k-1}}{x_k - x_{k-1}} & \text{if } x_{k-1} \le x \le x_k \\ \dfrac{x_{k+1} - x}{x_{k+1} - x_k} & \text{if } x_k \le x \le x_{k+1} \\ 0 & \text{otherwise.} \end{cases} \tag{3.18}$$

Then the entries in the rows corresponding to (i, j) and the columns corresponding to (k, l) of B_x and B_y are calculated by

$$(B_x)_{(i,j),(k,l)} = \int_\Omega (\partial_x \phi_{(i,j)})\psi_{(k,l)}dA \tag{3.19}$$

$$(B_y)_{(i,j),(k,l)} = \int_\Omega (\partial_y \phi_{(i,j)})\psi_{(k,l)}dA \tag{3.20}$$

Again taking advantage of the well-defined structural properties of the matrices, we can re-write (3.19) and (3.20) as

$$(B_x)_{(i,j),(k,l)} = \int_{x_{i-1}}^{x_{i+1}} (\partial_x \phi_i(x))\psi_k(x)dx \cdot \int_{y_{j-1}}^{y_{j+1}} \phi_j(y)\psi_l(y)dy \tag{3.21}$$

$$(B_y)_{(i,j),(k,l)} = \int_{y_{j-1}}^{y_{j+1}} (\partial_y \phi_j(y))\psi_l(y)dy \cdot \int_{x_{i-1}}^{x_{i+1}} \phi_i(x)\psi_k(x)dx \tag{3.22}$$

3.4 Boundary Values

The boundary of the grid has to be be treated differently than the interior. This becomes very clear, especially when considering the interaction of each Q2 and Q1 degree of freedom with its neighbours, as shown in Figure 3.4.

- For the Q2 elements, we will enforce Dirichlet boundary conditions. This means, that we will set the boundary of the right-hand side vectors to the true solution, with the diagonal entries in the rows of the system matrices L_x and L_y corresponding to the boundary set to 1 and all other entries in these rows set to 0.

- For the Q1 elements, we do not enforce any boundary conditions. However, we do need to do different calculations for these degrees of freedom, as for each degree of freedom along the boundary, some of the neighbours don't exist. Thus, the basis functions for these non-existent neighbours can be thought of as being both constant and zero, leading to boundary values that differ from the interior.

In our experiments, we will do one more step that affects our boundary: Once we have all the matrices and vectors set up, we will symmetrise the system matrix, which essentially removes any interaction of interior degrees of freedom with the boundary. This step is not a necessary step, but it is a simple step that can make the analysis a little easier. It certainly makes it easier to compare our setup with different implementations, as these often work only with symmetric matrices.

3.5 GMRES

For solving the discretised system of the Stokes equations, we will use the method called GMRES (***G****eneralised* ***M****inimal* ***RES****idual method*) [16, Chapter 6.5]. This method, part of the Krylov subspace methods family, is an iterative numerical algorithm for solving large, sparse systems of linear equations. Theoretically, if we were able to work with perfect precision, this algorithm would give the exact solution in a finite number of steps. However, it is of more interest as an approximation method, as it can approximate solutions to systems of equations consisting of millions of unknowns in few iterations with satisfactory accuracy, given an appropriate preconditioner.

3.5.1 The algorithm

Denote the system of linear equations by $A\mathbf{x} = \mathbf{b}$, with the matrix A assumed to be a real $n \times n$ invertible matrix with full rank, so that the system has a unique solution. Denote the L^2 or *Euclidean norm* (or, in short, *2-norm*) of any vector $\mathbf{v}$ by $||\mathbf{v}||$.

Then, given some initial guess $\mathbf{x^{(0)}}$, the GMRES algorithm is minimising in m steps ($m \in \mathbb{N}$) the *2-norm* of the residual

$$\mathbf{r^{(m)}} = ||\mathbf{b} - A\mathbf{x^{(m)}}|| \tag{3.23}$$

over the *Krylov subspace*

$$\mathbf{x^{(m)}} - \mathbf{x^{(0)}} \in \mathcal{K}_m(A, \mathbf{r^{(0)}}) = span\{\mathbf{r^{(0)}}, A\mathbf{r^{(0)}}, ..., A^{m-1}\mathbf{r^{(0)}}\}. \tag{3.24}$$

In order to minimise (3.23) over (3.24), an orthonormal basis $\{\mathbf{v^{(1)}}, ..., \mathbf{v^{(m)}}\}$ to the subspace $\mathcal{K}_m$ is constructed using the *Arnoldi algorithm*. The Arnoldi algorithm applies a (modified) Gram-Schmidt process to the Krylov subspace, which relies on the fact that the first vector $\mathbf{v^{(1)}}$ is normalised, i.e., $\mathbf{v^{(1)}}$ scaled to have a norm of 1. Lines 3-15 of Algorithm 1 shows the steps involved in the algorithm. Denoting the $n \times m$ matrix formed by $\mathbf{v^{(1)}}, ..., \mathbf{v^{(m)}}$ as $V^{(m)}$, we can write the vector $\mathbf{x^{(m)}} - \mathbf{x^{(0)}} \in \mathcal{K}_m(A, \mathbf{r^{(0)}})$ as

$$\mathbf{x^{(m)}} = \mathbf{x^{(0)}} + V^{(m)}\mathbf{y^{(m)}} \tag{3.25}$$

with $\mathbf{y^{(m)}} \in \mathbb{R}^m$. From the Arnoldi algorithm, we can derive a relation between the two matrices $V^{(m)}$ and $V^{(m+1)}$:

$$\begin{aligned} AV^{(m)} &= V^{(m)}H^{(m)} + \omega^{(m)}(\mathbf{e}^{(m)})^T \\ &= V^{(m+1)}\bar{H}^{(m)} \\ &= V^{(m+1)}(Q^{(m)})^T Q^{(m)}\bar{H}^{(m)} \\ &= V^{(m+1)}(Q^{(m)})^T\bar{R}^{(m)} \end{aligned} \tag{3.26}$$

where $\bar{H}^{(m)}$ is the $(m+1) \times m$ upper *Hessenberg matrix* filled with the values of h_{ij}, with $H^{(m)}$ being obtained from $\bar{H}^{(m)}$ by deleting its last row, and where $\bar{R}^{(m)} = (Q^{(m)})^T H^{(m)}$ is derived from the QR-factorisation of $H^{(m)} = Q^{(m)}\bar{R}^{(m)}$. Letting $\mathbf{e_1} = (1, 0, 0, ..., 0)^T$ be the first vector of the standard basis of $\mathbb{R}^{m+1}$ and setting $\beta = ||\mathbf{r^{(0)}}||$, then, when $\mathbf{v^{(1)}} = \mathbf{r^{(0)}}/\beta$ and since the columns of $V^{(m)}$ are orthonormal, we have

that

$$||\mathbf{r^{(m)}}|| = ||\mathbf{b} - A\mathbf{x^{(m)}}|| = ||\beta\mathbf{e_1} - \bar{H}^{(m)}\mathbf{y^{(m)}}||. \tag{3.27}$$

Thus, the value of the approximate solution $\mathbf{x^{(m)}}$ can be found by minimising the 2-norm of the right-hand-side of (3.27). It is important to note that we don't actually compute $\mathbf{y^{(m)}}$ at each step. We know that the vector $\mathbf{y^{(m)}}$ that minimises $||\beta\mathbf{e_1} - \bar{H}^{(m)}\mathbf{y}||$ is given by

$$\mathbf{y^{(m)}} = (R^{(m)})^{-1}\mathbf{g^{(m)}} \tag{3.28}$$

with $R^{(m)}$ the $m \times m$ upper triangular matrix obtained from $\bar{R}^{(m)}$ by deleting its last row, and with $\mathbf{g^{(m)}}$ the m-dimensional vector obtained from $\bar{\mathbf{g}}^{\mathbf{(m)}} = (\gamma^{(1)}, ..., \gamma^{(m+1)})^T$ also by deleting its last component. Thus the residual vector at step m satisfies

$$\begin{aligned} \mathbf{b} - A\mathbf{x^{(m)}} &= V^{(m+1)}(\beta\mathbf{e_1} - \bar{H}^{(m)}\mathbf{y^{(m)}}) \\ &= V^{(m+1)}(Q^{(m)})^T(\gamma^{(m+1)}\mathbf{e^{(m+1)}}) \end{aligned} \tag{3.29}$$

which leads to

$$||\mathbf{b} - A\mathbf{x^{(m)}}|| = |\gamma^{(m+1)}|. \tag{3.30}$$

Thus, instead of computing $\mathbf{y^{(m)}}$ at each step m, we can simply consider $|\gamma^{(m+1)}|$. If this value is small enough, then we know the algorithm can be stopped. We then delete the last row of $\bar{R}^{(m)}$ and $\bar{\mathbf{g}}^{\mathbf{(m)}}$ and solve the resulting upper triangular system to obtain $\mathbf{y^{(m)}}$. Then the approximate solution $\mathbf{x^{(m)}} = \mathbf{x^{(0)}} + V^{(m)}\mathbf{y^{(m)}}$ is computed. For a more detailed discussion of GMRES including proofs, see [16, §6.5.3].

The essence of the GMRES algorithm can be captured in these five main steps:

1. Use the Arnoldi iteration to calculate orthonormal basis vectors $\mathbf{v^{(m)}}$
2. Check whether $||\mathbf{r^{(m)}}||$ is small enough by checking the value of $|\gamma^{(m+1)}|$
3. Calculate $\mathbf{y^{(m)}}$ from $R^{(m)}$ and $\mathbf{g^{(m)}}$
4. Calculate $\mathbf{x^{(m)}} = \mathbf{x^{(0)}} + V^{(m)}\mathbf{y^{(m)}}$
5. Repeat as long as the residual $\mathbf{r^{(m)}}$ is still too big

Another way to look at the GMRES algorithm is by looking at a pseudocode implementation. Algorithm 1, taken from [16, Page 165], gives a nice summary of the algorithm.

Algorithm 1 GMRES algorithm

1: Compute $\mathbf{r^{(0)}} = \mathbf{b} - A\mathbf{x^{(0)}}$, $\beta = ||\mathbf{r^{(0)}}||$ and $\mathbf{v^{(1)}} = \mathbf{r^{(0)}}/\beta$.
2: Define the $(m+1) \times m$ matrix $H^{(m)}$ and set elements h_{ij} to zero
3: **for** j = 1,2,...,m **do**
4: Compute $\mathbf{w^{(j)}} = A\mathbf{v^{(j)}}$
5: **for** $i = 1, 2, ..., j$ **do**
6: $h_{ij} = (\mathbf{w^{(j)}}, \mathbf{v^{(j)}})$
7: $\mathbf{w^{(j)}} = \mathbf{w^{(j)}} - h_{ij}\mathbf{v^{(i)}}$
8: **end for**
9: $h_{j+1,j} = ||\mathbf{w^{(j)}}||$
10: **if** $h_{j+1,j} = 0$ **then**
11: $m = j$
12: **break**
13: **end if**
14: $\mathbf{v^{(j+1)}} = \mathbf{w^{(j)}}/h_{j+1,j}$
15: **end for**
16: Define the $(m+1) \times m$ Hessenberg matrix $H^{(m)} = \{h_{ij}\}_{1 \leq i \leq m+1, 1 \leq j \leq m}$
17: Compute $\mathbf{y^{(m)}}$, the minimiser of $||\beta\mathbf{e_1} - H^{(m)}\mathbf{y}||^2$
18: Compute $\mathbf{x^{(m)}} = \mathbf{x^{(0)}} + V^{(m)}\mathbf{y^{(m)}}$

Examining Algorithm 1 carefully, it can be seen that there is only one possibility for the algorithm to breakdown: In the Arnoldi loop, when $\mathbf{w^{(j)}} = 0$, i.e., when $h_{j+1,j} = 0$ at step j. As the $(j+1)^{st}$ Arnoldi vector can't be generated anymore, the algorithm stops. However, this means that the residual vector is the zero vector, i.e., the algorithm gives the exact solution at this step, as any new vector generated by the algorithm already lies in the generated subspace [16, Proposition 6.10].

3.5.2 Restarted GMRES

The *restarted GMRES* algorithm is a commonly used variant that allows explicit control of memory requirements. It is motivated by the fact that GMRES is an algorithm with complexity of order $O(nm^2)$, i.e., it grows quadratically with the number of iterations. Thus, as m becomes large the growth in memory and computational requirements become impractical. This is what restarted GMRES targets. The idea behind it is to perform a certain number of iterations with GMRES, compute the final approximation to the solution and take this computed approximation as the new initial guess for a new run of GMRES. It is described in Algorithm 2, taken from [16, Page 172].

Algorithm 2 Restarted GMRES

1: Compute $\mathbf{r^{(0)}} = \mathbf{b} - A\mathbf{x^{(0)}}$, $\beta = ||\mathbf{r^{(0)}}||_2$, and $\mathbf{v11(1)} = \mathbf{r^{(0)}}/\beta$.
2: Generate the Arnoldi basis and the matrix $\bar{H}^{(m)}$ using the Arnoldi algorithm starting with $\mathbf{v^{(1)}}$.
3: Compute $\mathbf{y^{(m)}}$, which minimises $||\beta\mathbf{e_1} - \bar{H}^{(m)}\mathbf{y}||_2$.
4: Compute $\mathbf{x^{(m)}} = \mathbf{x^{(0)}} + V^{(m)}\mathbf{y^{(m)}}$.
5: If satisfied then Stop, else set $\mathbf{x^{(0)}} := \mathbf{x^{(m)}}$ and go to 1.

One well-known difficulty with this approach is that this algorithm can stagnate when the matrix is not positive definite. For our experiments we will not be using restarted GMRES as the addition of a preconditioner, Section 3.6, reduces the number of iteration required for convergence to a small number (as will be seen in Chapter 5) rendering a restart of the algorithm unnecessary. However, there is clearly a role for it, and other limited-memory algorithms, in implementations on current GPU architectures.

3.6 Preconditioning

When considering real-life problems, the system matrix of the discretised system is almost always ill-conditioned in one way or another. In order to be able to efficiently use an iterative solver like GMRES for solving such systems, it becomes necessary to transform (or precondition) the given problem into a different one. This is done in such a way that it results in a system that has the same solution as the ill-conditioned problem, but with nicer properties, e.g., a better condition number or faster convergence of GMRES.

3.6.1 Left- and Right-Preconditioning

To precondition a system, the main task is to find a non-singular matrix, M, that transforms the original problem. Considering the standard form of a linear problem, $A\mathbf{x} = \mathbf{b}$, we can apply the preconditioning matrix M in two different ways:

1. Right-Preconditioning: $MA\mathbf{x} = M\mathbf{b}$
2. Left-Preconditioning: $AM\mathbf{u} = \mathbf{b}$, where $\mathbf{u} = M^{-1}\mathbf{x}$

The first thing to note about the two operators, MA and AM, is that their spectra are identical. Just based on this, it is reasonable to expect both to yield similar

convergence, however, it is known that eigenvalues alone do not always govern convergence.

When preconditioning the original problem on the left, GMRES minimises the preconditioned residual norm

$$||M(\mathbf{b} - A\mathbf{x})||. \tag{3.31}$$

This is fine, as long as M is well-conditioned for the problem at hand. However, if M is poorly conditioned, as any "good" M should be for an ill-conditioned A, this can greatly change the approximation generated. In practise, this can mean that the algorithm seems to have converged to some solution, based on the residual norm (3.31), when in fact it is still far off in terms of the norm in (3.23).

On the other hand, when preconditioning the original problem on the right, then the residual norm minimised by GMRES remains the same as originally the case, GMRES is still trying to minimise

$$||\mathbf{b} - A\mathbf{x}||_2 \text{ with } \mathbf{x} = M\mathbf{u} \tag{3.32}$$

In practise this means that GMRES will never claim to have converged when in fact it hasn't (yet). The solution, once obtained, is known to be the correct solution sought after up to the desired precision. Another advantage of right-preconditioning is that it allows flexible GMRES (FGMRES), see Section 3.7, where the preconditioner can change from one iteration to the next. These two advantages typically make right-preconditioning the preferred choice. The right-preconditioned GMRES algorithm becomes Algorithm 3 [16, Page 270].

3.7 FGMRES

There is a variant of GMRES, called FGMRES (*Flexible* GMRES), which is equivalent to the standard GMRES. FGMRES highlights the fact that it is possible to tweak the GMRES algorithm to make it possible to change the preconditioner at every step. This enables us to, e.g., adapt the preconditioning matrix M_i at iteration i based on some criteria or for algorithmic convenience. Considering the standard preconditioned GMRES algorithm, Algorithm 3, in Line 13, the assembled solution $\mathbf{x^{(m)}}$ is computed by means of a linear combination of the preconditioned vectors $\mathbf{z^{(i)}} = M\mathbf{v^{(i)}}$, $i = 1, ..., m$, which are also computed in Line 3 when computing the

Algorithm 3 Preconditioned GMRES algorithm

Compute $\mathbf{r^{(0)}} = \mathbf{b} - A\mathbf{x^{(0)}}$, $\beta = ||\mathbf{r^{(0)}}||$ and $\mathbf{v^{(1)}} = \mathbf{r^{(0)}}/\beta$.
for j = 1,2,...,m **do**
    Compute $\mathbf{w} = AM\mathbf{v^{(j)}}$
    for $i = 1, 2, ..., j$ **do**
        $h_{i,j} = (\mathbf{w}, \mathbf{v^{(i)}})$
        $\mathbf{w} = \mathbf{w} - h_{i,j}\mathbf{v^{(i)}}$
    end for
    Compute $h_{j+1,j} = ||\mathbf{w}||$
    Compute $\mathbf{v^{(j+1)}} = \mathbf{w}/h_{j+1,j}$
    Define $V^{(m)} = [\mathbf{v^{(1)}}, ..., \mathbf{v^{(m)}}]$, $\bar{H}^{(m)} = \{h_{i,j}\}_{1 \leq i \leq j+1, 1 \leq j \leq m}$
end for
Compute $\mathbf{y^{(m)}} = \text{argmin}_{\mathbf{y}} ||\beta\mathbf{e_1} - \bar{H}^{(m)}\mathbf{y}||_2$
Compute $\mathbf{x^{(m)}} = \mathbf{x^{(0)}} + MV^{(m)}\mathbf{y^{(m)}}$

vector w. Typically, the $\mathbf{z^{(i)}}$'s would not be stored, but only the $\mathbf{v^{(i)}}$'s, as each $\mathbf{z^{(i)}}$ can be computed by applying M to the respective $\mathbf{v^{(i)}}$. However, if we instead store the actual $\mathbf{z^{(i)}}$'s throughout the algorithm, then we can have different preconditioning matrices $M^{(i)}$ at iteration i. There is no need to always choose the same M, as we now can compute the approximate solution based on the $\mathbf{z^{(i)}}$'s,

$$\mathbf{x^{(m)}} = \mathbf{x^{(0)}} + Z^{(m)}\mathbf{y^{(m)}} \tag{3.33}$$

where $Z^{(m)}$ is the matrix whose columns consist of all the $\mathbf{z^{(i)}}$'s. The modified algorithm can be denoted as in Algorithm 4 [16, Page 273].

3.8 Multigrid

For our preconditioner, we will be using a multigrid V-cycle, similar to as described in Section 2.8. However, now we will use weighted restriction and interpolation adapted to the Q2-Q1 finite-element discretisation. Figure 3.5 shows the weights chosen for restriction for the four different types of Q2 degrees of freedom. These values come from the finite element interpolation operators and are only accurate for square cells, i.e., $h_x = h_y$. For rectangular cells, the values obtained by similar calculation from the interpolation operators would differ. In order to do interpolation, the same weights are chosen, but just in the reverse direction, i.e., each fine-grid degree of freedom gets a weighted average of the coarse-grid degrees of freedom it contributed to with the

Algorithm 4 Flexible GMRES algorithm

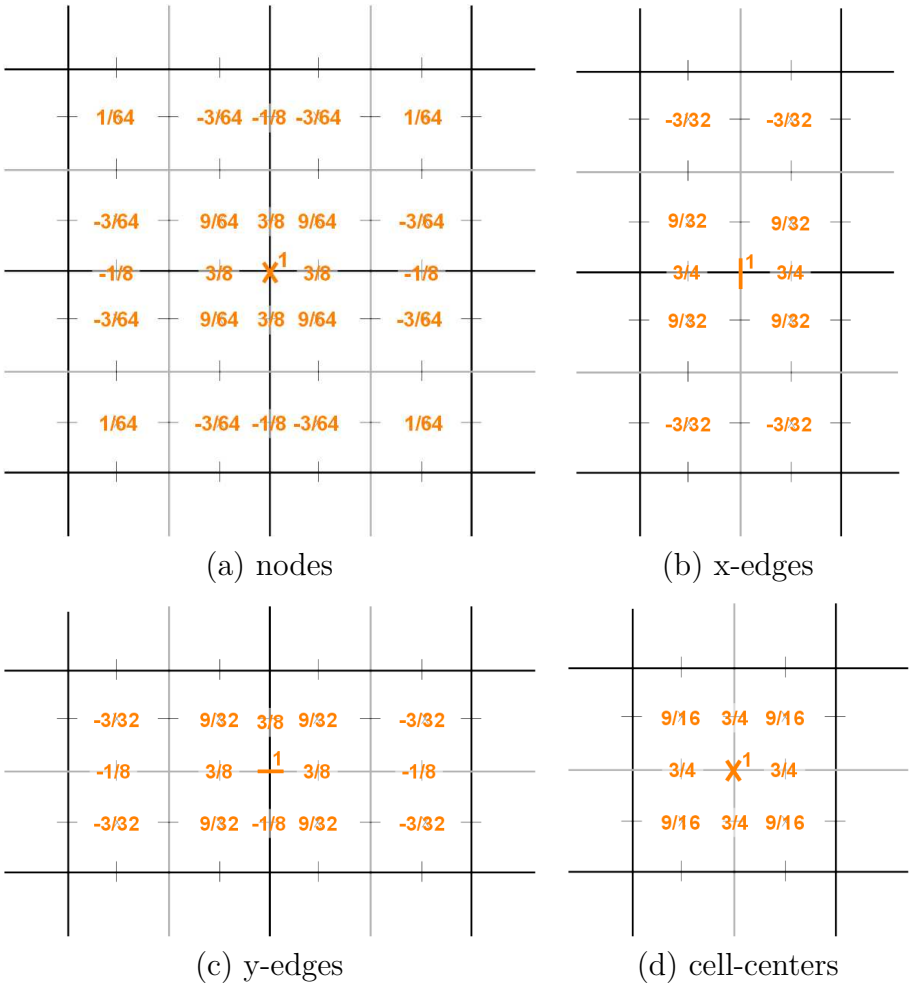
Compute $\mathbf{r^{(0)}} = \mathbf{b} - A\mathbf{x^{(0)}}$, $\beta = ||\mathbf{r^{(0)}}||$ and $\mathbf{v^{(1)}} = \mathbf{r^{(0)}}/\beta$.
for j = 1,2,...,m **do**
    Compute $\mathbf{z^{(j)}} = M^{(j)}\mathbf{v^{(j)}}$
    Compute $\mathbf{w} = A\mathbf{z^{(j)}}$
    for $i = 1, 2, ..., j$ **do**
        $h_{i,j} = (\mathbf{w}, \mathbf{v^{(i)}})$
        $\mathbf{w} = \mathbf{w} - h_{i,j}\mathbf{v^{(i)}}$
    end for
    Compute $h_{j+1,j} = ||\mathbf{w}||$
    Compute $\mathbf{v^{(j+1)}} = \mathbf{w}/h_{j+1,j}$
    Define $Z^{(m)} = [\mathbf{z^{(1)}}, ..., \mathbf{z^{(m)}}]$, $\bar{H}^{(m)} = \{h_{i,j}\}_{1 \le i \le j+1, 1 \le j \le m}$
end for
Compute $\mathbf{y^{(m)}} = \text{argmin}_{\mathbf{y}} ||\beta\mathbf{e_1} - \bar{H}^{(m)}\mathbf{y}||_2$
Compute $\mathbf{x^{(m)}} = \mathbf{x^{(0)}} + Z^{(m)}\mathbf{y^{(m)}}$

weight associated with the fine-grid degree of freedom. For the Q1 degrees of freedom, we will use the same weights as for Poisson's equation, given in Section 2.7.

Another aspect of Multigrid that we need to answer when dealing with the Stokes equations with Taylor-Hood elements is what to do with the coarse-grid matrices L_x, L_y, B_x and B_y. We will be constructing them by means of a Galerkin coarsening. Let P be the interpolation matrix, such that for any given vector, $\mathbf{u}^{2h}$, on the coarse grid, Ω^{2h}, the same vector, $\mathbf{u}^h$, interpolated to the finer grid, Ω^h, can be computed by $P\mathbf{u^{2h}}$. Going in reverse, i.e., restricting $\mathbf{u}^h$ onto the coarser grid is done by multiplying with the transpose, $P^T\mathbf{u}^h$. Denoting the interpolation matrix for the Q1 degrees of freedom as P_1 and, equivalently, the interpolation matrix for the Q2 degrees of freedom as P_2, then the coarse-grid matrices can be calculated by

$$\begin{aligned} L_x^{2h} &= P_2^T L_x^h P_2 \\ L_y^{2h} &= P_2^T L_y^h P_2 \\ B_x^{2h} &= P_1^T B_x^h P_2 \\ B_y^{2h} &= P_1^T B_y^h P_2 \end{aligned} \tag{3.34}$$

For our analysis, we are using the standard restriction and interpolation matrices, that come from the standard finite-element interpolation operators. Due to using



(a) nodes (b) x-edges

(c) y-edges (d) cell-centers

Figure 3.5: Weighted restriction of Q2 degrees of freedom for square cells

these standard operators, the calculations in (3.34) can be replaced by a simple re-discretisation of the fine-grid matrices on the coarser grid, both for square and rectangular cells.

However, we still need to do a slight modification to our system matrix on the coarsest grid in order to be able to do a direct solve. Taking the system matrix as described above will give us one zero eigenvalue (in the case of the coarsest grid being a single element, we get two). As we do not enforce any boundary conditions on the pressure, any computed solution for the pressure is only determined up to a constant. For example, taking the vector with all 0's for the velocity and all 1's for the pressure will give us a zero matrix-vector product, i.e., a constant pressure lives inside the nullspace. The reason why we get two zero eigenvalues for a single element is due

to the fact that the discretisation for a single element is unstable. In either case, the matrix, as is, cannot be inverted for a direct solve. For grids larger than one element, this problem can be easily dealt with by tying down the very last entry in the pressure right-hand side and, thus, removing it from the nullspace. This is done simply by adding a 1 into the very bottom right corner of the system matrix (two 1's for a single element). More details on this can be found in [15].

3.9 Braess-Sarazin

We will be using the so-called Braess-Sarazin smoother as a relaxation method for our multigrid algorithm [17, 21–24]. We won't be using a standard Jacobi- or Gauss-Seidel-type relaxation method, as our system matrix is not positive definite. In fact, for these relaxation schemes, we would need to invert either the diagonal or lower triangular parts parts of the system matrix, both of which are singular and, hence, cannot be inverted. Thus, we have to find an alternative solution, with Braess-Sarazin being one of the more common choices. It allows us to compute a smoothing update for both the velocity and pressure in very simple steps. It does so by computing a simpler approximation to the true solution by solving a global saddle-point problem. Given the Stokes equations (3.1) and (3.2) and given some approximation $\mathbf{u}^{old}$ and p^{old}, the "ideal" Braess-Sarazin update takes the form

$$\begin{bmatrix}\mathbf{u}\\ p\end{bmatrix}^{new} = \begin{bmatrix}\mathbf{u}\\ p\end{bmatrix}^{old} + \omega_{BS}\begin{bmatrix}tD & B^T\\ B & 0\end{bmatrix}^{-1}\left(\begin{bmatrix}f\\ g\end{bmatrix} - A\begin{bmatrix}\mathbf{u}\\ p\end{bmatrix}^{old}\right) \tag{3.35}$$

with D being a diagonal matrix with entries from L, t being a scaling parameter for D, and ω_{BS} being an underrelaxation parameter for the global update. To solve (3.35), we first express this system in the factorised form

$$\begin{bmatrix}tD & 0\\ B & S\end{bmatrix}\begin{bmatrix}I & \frac{1}{t}D^{-1}B^T\\ 0 & I\end{bmatrix}\begin{bmatrix}\delta\mathbf{u}\\ \delta p\end{bmatrix} = \begin{bmatrix}r_{\mathbf{u}}\\ r_p\end{bmatrix} \tag{3.36}$$

where $S = -\frac{1}{t}BD^{-1}B^T$ is the Schur complement, and $\delta\mathbf{u}$ and δp are the update for $\mathbf{u}^{old}$ and p^{old}. One thing to note about the matrix S is that by forming the matrix-matrix-matrix product, the resulting Q1-Matrix interacts with a 4x4 element patch instead of "only" a 2x2. From (3.36), we can extract two equations that need to be

solved,

$$S\delta p = r_p - \frac{1}{t} B D^{-1} r_{\mathbf{u}} \tag{3.37}$$

$$\delta \mathbf{u} = \frac{1}{t} D^{-1}(r_{\mathbf{u}} - B^T \delta p). \tag{3.38}$$

Though it's possible to solve this system exactly, we only solve it approximately, giving us an "inexact" Braess-Sarazin method that is sufficiently accurate for our purposes [25]. We will be using a single sweep of a standard Weighted Jacobi method on (3.37) with weight ω to get a value for δp, which we then use in (3.38) to get a value for $\delta \mathbf{u}$. The new approximations $\mathbf{u}^{new}$ and p^{new} are then computed as

$$\mathbf{u}^{new} = \mathbf{u}^{old} + \omega_{BS} \delta \mathbf{u}, \tag{3.39}$$

$$p^{new} = p^{old} + \omega_{BS} \delta p. \tag{3.40}$$

For our analysis, we will always keep ω_{BS} fixed at 1 to simplify identification of good choices of the other parameters, t and ω, leaving the question of optimal parameter choices for future work.

Chapter 4

Implementation

We have implemented the FGMRES algorithm with a multigrid preconditioner in C++, eventually adding OpenCL code for parallelisation purposes (discussed later). The first task consists of writing classes for all of our data structures. As we are using an object-oriented language, we are able to hide all the stencil calculations in these classes giving us a very clean FGMRES implementation.

4.1 Data Classes

The various classes that we needed for our data structures are

- *Q2Vector*: A vector class holding the data for all the Q2 degrees of freedoms in the mesh, plus a halo of ghost cells all around the mesh to simplify the computations. Inside, the data is stored in 4 different arrays: One array holding the values for all the nodes, one for all the x-edges, one for the y-edges, and one for the cell-centers. All the degrees of freedom are ordered by rows, starting at the origin (i.e., bottom left corner). The data This class is used to store the approximations for the discretised velocity.

- *Q1Vector*: This is the vector class for the data for all the Q1 degrees of freedom, plus a halo of ghost cells all around the mesh to simplify the computations . It functions similarly to the *Q2Vector* class, but limits itself to the nodal values needed for a Q1 approximation. This class is used to store the approximations for the discretised pressure.

- *Q2Matrix*: A matrix class that holds for each Q2 degree of freedom in the mesh (node, edge, center) its interaction with its neighbours. As we saw in Table 3.1 and Figure 3.4, all degrees of freedom in the mesh interact with a fixed number of degrees of freedom in the neighbouring elements. Thus, inside the Q2Matrix class, the data is stored in 4 different two-dimensional arrays, with the same ordering as for the *Q2Vector* class. This class is used for the discretised Laplacian.

- *Q2Q1Matrix*: A matrix class that relates all Q2 degrees of freedoms to Q1 degrees of freedom. The implementation is such that it can be used for computing matrix-vector products with both the matrix and its transpose in (3.13). This class is used for the discretised gradient.

We need to add one more data structure that we need in our Braess-Sarazin relaxation method:

- *Q1MatrixBs*: A matrix class that relates to the *Q1Vector* class as the *Q2Matrix* class relates to the *Q2Vector* class, with the important difference that it includes a larger circle of neighbours for interaction as needed for the Schur complement matrix S.

In all of our data structures, the actual data is stored in simple arrays of double's that are exposed as public objects. This allows us to directly index the arrays without having to go through some type of API, giving us the best performance possible. Also, due to the fact that we are only storing the values for the stencil calculations, we have reduced the memory required to a minimum. This not only allows us to consider larger problems, but it is another factor that improves the performance of the code as it affects the memory access time in a beneficial way.

For all of our data structures, we take advantage of the object-oriented nature of C++ and overload the required operations for the various matrix-vector and vector-vector calculations. With the data structures taken care of, we now have to setup the objects we will need and fill them with the initial data.

- *Data*: This is the main class for setting up our data. It first of all creates all the vectors and matrices that we will need in FGMRES. The next step in the class consists of filling the matrix entries with the right values. This is done with calls to static member functions of the classes described below, *MatrixA*,

MatrixBX, and *MatrixBY*. Following this, the right-hand side and initial guess are filled in. As we are using Dirichlet Boundary Conditions, the boundary of the right-hand-side velocity vectors is set to the true solution. The interior of the right-hand side is calculated by means of a 9-point Gaussian Quadrature. In order to simplify our later calculations, the resulting matrices are symmetrised. As our interior is already symmetric, we only need to take care of the boundary degrees of freedom. We set the initial guess to a user-selected solution (possible values: true solution (for debugging), random initial guess, zero initial guess).

- *MatrixA*, *MatrixBX*, *MatrixBY*: These classes (or rather namespaces) contain a single static inline function that fills in the values of the matrix object whose reference has been passed in. The values to be entered have been calculated as described in Section 3.3.

4.2 Source Code

With our discretised system fully set up, we can now turn our attention to the three main pieces of the actual algorithm: 1) *multigrid* preconditioner, 2) *Braess-Sarazin* smoother, and 3) *FGMRES*.

4.2.1 Multigrid preconditioner

The *multigrid V-cycle* that we will use as a preconditioner is implemented in a very similar fashion as described in Section 2.9. Code 4.1 shows a simplified version of the V-cycle recursive function that is called whenever a V-cycle solve is required.

The layout of the code here is very similar to Code 2.1, the main difference is what happens on the coarsest grid. Instead of simply running our smoother a number of times, we now do an exact solve (Line 4), discussed in detail in Chapter 5. If we are not on the coarsest grid (Lines 5-23), we do exactly the same steps as described before: Pre-smoothing (Lines 7-8, Equations (3.39) and (3.40)), restriction (Lines 10-12, Figure 3.5), multigrid solve on coarser grid (Line 14), interpolation (Lines 16-18), and post-smoothing (Lines 20-21, Equations (3.39) and (3.40)).

Code 4.1: Preconditioner: V-cycle function definition, simplified

```
1  void VCycle::run_vcycle(Q2Vector *new_u1, Q2Vector *new_u2, Q1Vector *
       new_p, Q2Vector *new_rhs_u1, Q2Vector *new_rhs_u2, Q1Vector *
       new_rhs_p, unsigned int cur_level) {

3      if(new_u1->xdimension <= 6 || cur_level == 1)
4          exact_solve(...);
5      else {

7          for(unsigned int pre = 0; pre < prerelexation; ++pre)
8              bs[max_levels-cur_level].solve(...);

10         Q2Vector coarse_rhs_u1 = (/* u1 residual */)restrict();
11         Q2Vector coarse_rhs_u2 = (/* u2 residual */).restrict();
12         Q1Vector coarse_rhs_p = (/* p residual */).restrict();

14         run_vcycle(...);

16         *new_u1 += coarse_u1.interpolate();
17         *new_u2 += coarse_u2.interpolate();
18         *new_p += coarse_p.interpolate();

20         for(unsigned int post = 0; post < postrelexation; ++post)
21             bs[max_levels-cur_level].solve(...);

23     }
24 }
```

4.2.2 Braess-Sarazin smoother

The implementation of the *Braess-Sarazin* smoother, called in Lines 8 and 21 of Code 4.1, comes with a rather straight-forward layout of the code. In the setup phase, we build up the Schur complement matrix, as it is always the same no matter the data:

1. Extract diagonal of L and also invert it at the same time. This is a rather trivial step and can be done with four simple loops, one for each type of degree of freedom.

2. Multiply the inverted diagonal matrix, D^{-1}, times the discretised gradient matrix, B. Doing this calculation is, also, very straightforward, as it follows the layout of our data structure perfectly. Thus, it can also be done with four simple loops.

3. Multiply the above product on the left by B^T. This is the hardest part of the setup phase. Essentially, for each Q1 node we have to figure out with which Q2 degrees of freedom it interacts. These Q2 degrees of freedom corresponds to rows of the product calculated in the previous step, with the Q1 node corresponding to a column. Doing this in an efficient and non-concurrent way, i.e., making sure we don't write to the same entry two different times (this is important for parallelisation), requires careful coding and is not an easy task. Code 4.2 gives a small insight into the amount of work involved in getting it properly set up for parallel computations: We do the computation node by node and

Code 4.2: Braess-Sarazin setup: Multiply $D^{-1}B$ on the left by B^T

```
1  for(unsigned int node = 0; node < num_nodes; ++node) {
2      for(unsigned int sur = 0; sur < 25; ++sur) {
3
4          if(/* ... along (or next to) boundary with 'sur' outside ... */)
5              continue;
6
7          unsigned int surnode = (node/xdimension -2)*xdimension + (node%
               xdimension-2) + (sur/5)*xdimension + sur%5;
8          double sum_x = 0, sum_y = 0;
9
10         for(unsigned int k = 0; k < 9; ++k) {
11
12             unsigned int k_sur = node-((xdimension+1)/2)-1 + (k/3)*((
                   xdimension+1)/2) + k%3;
13             if(/* ... no connection between k_sur and surnode ... */)
14                 continue;
15
16             unsigned int sur_i, sur_j = /* ... calculate values based on
                    node, surnode and k\_sur ... */
17             sum_x += Bx.nodes[k_sur][sur_i]*Dinv_Bx.nodes[k_sur][sur_j];
18             sum_y += By.nodes[k_sur][sur_i]*Dinv_By.nodes[k_sur][sur_j];
19
20         }
21
22         /* ... also compute surrounding edges and cell centers ... */
23
24         unsigned int sur_i_to_j = 12 - (node/((xdimension+1)/2) -
               surnode/((xdimension+1)/2))*5 - (node%((xdimension+1)/2) -
               surnode%((xdimension+1)/2));
25         Bx_Dinv_Bx.nodes[node][sur_i_to_j] = sum_x;
26         By_Dinv_By.nodes[node][sur_i_to_j] = sum_y;
27
28     }
29 }
```

for each node surrounding index by surrounding index. This is the purpose of the two for loops in Lines 1-2 that wrap around everything. Lines 4-5 ensure that we do not go outside of the mesh. This is necessary as we do not enforce any boundary condition on the pressure and thus compute all the way to the boundary. Any surrounding index that would point outside of the mesh needs to be filtered out. Line 7 computes the index of the node of the surrounding index, in turn whose surrounding indices are looped over with the for loop started in Line 10. However, before entering the for-loop, we set two summation variables to 0, they will contain the total up value for the current node in the matrix triple product. Inside of the loop, we once again compute the actual index of the surrounding index of *surnode*, the surrounding index of the actual node we compute for. In Line 13, we have to double-check if the two surrounding nodes, *surnode* and *k_sur*, actually do interact. If they don't, we simply continue to the next iteration, Line 14. If we are okay, then we compute the surrounding indices based on k_sur, Line 16, and build the product of the two matrices at that entry before adding that value onto the summation variables, Lines 17-18. All of this, Lines 10-20 have to be repeated in a similar way for the surrounding x- and y-edges and the surrounding cell-centers. At the end of this, all that is left is to set the calculated value to the correct location of the matrix holding the triple matrix product, Lines 24-26.

4. All the above calculations have (naturally) been done separately (yet simultaneously) for the x and y components. To form the Schur complement matrix, we now need to add up the two calculated matrix products and scale by the inverse of the constant t.

Whenever we want to relax using our Braess-Sarazin implementation (i.e., for pre-/post-smoothing in the V-cycle), we can simply call the solve function repeatedly. A snippet of that function is shown in Code 4.3.

In Lines 5-6, we compute the right hand side as shown in (3.37). We then use a simple weighted Jacobi update with a zero initial guess in Lines 8 and 9 to compute an approximation δp. The calculated δp is then plugged into Equation (3.38) to compute δu, Lines 11-12. These updates are then used to update our approximation, Lines 14-16.

As we will be looking at a lot more code below, we will be using a shorthand notation for future code snippets to make them more concise. With this shorthand,

Code 4.3: Relaxation: Braess-Sarazin solve function, simplified

```
1  void BraessSarazin::solve(Q2Vector *soln_u1, Q2Vector *soln_u2, Q1Vector
       *soln_p, Q2Vector *rhs_u1, Q2Vector *rhs_u2, Q1Vector *rhs_p) {

3      /* ... Calculate residual ... */

5      Q1Vector rhs = (residual_p - (Bx*(Dx_inv*residual_u1)
6                      + By*(Dy_inv*residual_u2))*(1.0/t));

8      for(unsigned int index = 0; index < rhs.num_nodes; ++index)
9          p_update.nodes[index] = omega/S.nodes[index][12]*rhs.nodes[index
               ];

11     Q2Vector u1_update = Dx_inv*(residual_u1-Bx*p_update)*(1.0/t);
12     Q2Vector u2_update = Dy_inv*(residual_u2-By*p_update)*(1.0/t);

14     *soln_p += p_update;
15     *soln_u1 += u1_update;
16     *soln_u2 += u2_update;
17 }
```

Lines 14-16 of Code 4.3 would be denoted in a single line by

```
1      *soln_# += #_update
```

with the # being a placeholder for the respective components.

4.2.3 FGMRES

Now that we have all the various pieces of the algorithm set up, we can pull them all together in our GMRES implementation. As it is a rather long implementation, we will examine it in multiple steps. Code 4.4 shows the setup of the iteration.

Line 3 initialises the iteration counter, whose value will eventually be the return value of the function. Around Line 5, we compute the initial residual vector. Given our operator overloading in the classes for our data structures, we can directly use the very intuitive matrix notation. Using the initial residual, we compute the overall 2-norm in Line 6. If the initial guess to the system is already within our desired tolerance we wrap up, Lines 8-11, and return an iteration count of 0, Line 10. Otherwise, we normalise the residual vectors, Line 13, compute what value for a 2-norm would signal achieved convergence, Line 15, and store the 2-norm as the first value in our Hessenberg system, Line 16.

Code 4.4: GMRES: Setting up, simplified

```
1  int GMRES::gmres(Q2Vector *soln_u1,Q2Vector *soln_u2,Q1Vector *soln_p) {

3      unsigned int its = 0;

5      work_#[0] = data.rhs_# - (data.A#*data.soln_#+data.B#*data.soln_#);
6      beta = sqrt(pow(work_u1[0].twoNorm(),2)+pow(work_u2[0].twoNorm(),2)+
           pow(work_p[0].twoNorm(),2));

8      if(beta < convergence_level) {
9          ...
10         return 0;
11     }

13     work_#[0] = work_#[0]/beta;

15     eps1 = convergence_level*beta;
16     rs[0] = beta;

18     while(its < maxiter && beta > eps1 && beta > 1e-15) {

20         // Iteration loop

22     }

24     // Computing solution

26     return its+1;
27 }
```

This concludes the setup of the GMRES algorithm, and we are now ready to enter the iteration loop. Code 4.5 shows a simplified version of the loop.

Firstly, we use our preconditioner (Lines 3-4) to improve our convergence. We then use the preconditioned array to re-compute the residual of our system, around Line 6. In Lines 8-14, we use the Arnoldi algorithm with a modified Gram-Schmidt process to form an orthogonal basis to our Krylov subspace. The main work of the Arnoldi algorithm, though, happens in Lines 8-11, were we do the actual orthogonalisation. Lines 13-14 simply serve to normalise the new vectors. Following this, in Lines 16-29, we update the factorisation of the Hessenberg matrix stored in *hh* by performing the previous transformations on the i^{th} column of *hh*. In Line 31, we simply update the residual norm stored in *beta* by the newly calculated norm. Once we have convergence within our tolerance, this will ensure that the iteration loop is terminated, Line 1. All that is left is to increment the iteration count by 1, Line 32.

Code 4.5: GMRES: Iteration loop, simplified

```
1  while(its < maxiter && beta > eps1 && beta > 1e-15) {

3      cycle.run(&prec_u1[its], &prec_u2[its], &prec_p[its],
4                &work_u1[its], &work_u2[its], &work_p[its]);

6      work_#[its+1] = data.A#*prec_#[its] + data.B#*prec_#[its];

8      for(unsigned int j = 0; j <= its; ++j) {
9          hh[its][j] = work_#[its+1].getInnerProductWith(work_#[j]) + ...
10         work_#[its+1] = work_#[its+1] - work_#[j]*hh[its][j];
11     }

13     hh[its][its+1] = ... // 2-norm of work_#[its+1]
14     work_#[its+1] = work_#[its+1]/hh[its][its+1];

16     for(unsigned int k = 0; k < its; ++k) {
17         double t = hh[its][k];
18         hh[its][k] = c[k]*t + s[k]*hh[its][k+1];
19         hh[its][k+1] = -s[k]*t + c[k]*hh[its][k+1];
20     }

22     double gamma = sqrt(pow(hh[its][its],2) + pow(hh[its][its+1],2));

24     c[its] = hh[its][its]/gamma;
25     s[its] = hh[its][its+1]/gamma;
26     rs[its+1] = -s[its]*rs[its];
27     rs[its] = c[its]*rs[its];

29     hh[its][its] = c[its]*hh[its][its] + s[its]*hh[its][its+1];

31     beta = fabs(rs[its+1]);
32     ++its;
33 }
```

After we either have reached convergence or have exceeded our maximum number of iterations, we then need to assemble the solution at the end. This is necessary as GMRES doesn't compute the intermediate approximations. Code 4.6 shows a simplified version of the corresponding code.

Assembling the solution is very straightforward and happens in two simple steps: First, we need to solve the upper triangular system, Lines 4-11. Following this, our "solution" that GMRES gives us is stored in the array *rs*, which we then use together with the preconditioned arrays to compute the final solution, Lines 13-17. As a final step, we return the total iteration count, Line 19. The addition of 1 to the iteration

Code 4.6: GMRES: Assembling solution, simplified

```
2       ...

4       rs[its] = rs[its]/hh[its][its];
5       for(unsigned int ii = 1; ii <= its; ++ii) {
6           unsigned int k = its-ii;
7           double val = rs[k];
8           for(unsigned int j = k+1; j <= its; ++j)
9               val = val - hh[j][k]*rs[j];
10          rs[k] = val/hh[k][k];
11      }

13      *soln_# = data.soln_#;
14      for(unsigned int j = 0; j <= its; ++j) {
15          double val = rs[j];
16          *soln_# = *soln_# + (prec_#[j]*val);
17      }

19      return its+1;
20  }
```

count is done to take into account that *C++* starts counting at 0.

4.3 GPU Considerations

Moving to the GPU is not the easiest task, though it is also not a very hard one. However, some consideration is needed before we are able to do this move efficiently and with good resulting performance.

1. In order to get good performance on the GPU, we need to break our calculations down into as small as possible independent pieces, the smaller and simpler the better [26]. These small calculations are then wrapped in so-called *kernels*. Kernels are essentially functions containing code that can be called and executed on the GPU. Code 4.7 shows the kernel function used for multiplying a matrix by a double. Such small and independent calculations fall out of our code naturally, as we are expressing all our calculations in stencil form. This gives us the type of fine-grained parallelism that we are looking for.

2. We need to choose a framework to implement our GPU code (the kernels) in. As our structured implementation is written in C++, the two obvious choices

Code 4.7: OpenCL: Kernel function for matrix multiplied by double

```
1 kernel void matrix_multiply_by_number(global const double * inp1,
2                                       global const double * inp2,
3                                       global double * ret) {
4     unsigned int global_id = get_global_id(0);
5     ret[global_id] = inp1[global_id]*(*inp2);
6 }
```

are CUDA and OpenCL:

- OpenCL (the *Open C*omputing *L*anguage) is an open standard for parallel programming for heterogeneous systems. With OpenCL, it is possible to execute the exact same kernels on various types of devices (GPUs, CPUs, DSPs, FPGAs, and more) from many different vendors.
- CUDA also offers a standard for parallel programming, though not an *open* standard. Contrary to OpenCL, it is limited to running on NVIDIA GPUs.

On NVIDIA GPUs, in terms of relative performance, they both are about the same. This is due to the fact that OpenCL is (when compiled) internally translated to the CUDA API on NVIDIA GPUs. Thus, the only somewhat noticeable difference is in the compile time at start, but not much in the runtime. However, OpenCL offers the flexibility and freedom to be used on more than just NVIDIA GPUs (heterogeneous (OpenCL) vs. homogeneous (CUDA)). Due to this flexibility and also the author's strong preference of open standards and open-source software, we will be using OpenCL to parallelise our code for GPUs.

3. When operating on the GPU, there naturally arises the need to communicate between the CPU and GPU. Unfortunately, this communication is very expensive, so minimising/hiding this need to communicate becomes a high priority. In particular when operating on multiple GPUs simultaneously, the communication path GPU-1 $\rightarrow$ CPU-1 $\rightarrow$ CPU-2 $\rightarrow$ GPU-2 can have detrimental effects on the overall performance.

4. At the moment, we do not have a direct solver on the GPU available. As part of our multigrid preconditioner, we need to do a direct solve on the coarsest level in the V-cycle. In order to do a direct solve, we currently need to move the data back to the CPU, perform a direct solve (using UMFPACK [27–30]),

and move the data back to the GPU. This adds unnecessary communication costs, thus developing a direct solver on the GPU could potentially be highly beneficial. Alternatively, we could consider doing relaxation instead of a direct solve on the coarsest grid. This avoids the communication, but may incur extra computations in the form of more V-cycles to achieve convergence. We will investigate how the performance of a direct vs. an approximate solve on the coarsest grid compare.

With all of this in mind, we wrote the necessary OpenCL kernels to perform a full multigrid-preconditioned GMRES solve on a single GPU. The logic used in both the serial and the parallel version is essentially identical, with no (additional) shortcuts taken on either side. A few examples of the kernels are given below.

4.3.1 OpenCL Kernels and Handler Examples

The first thing we have to take care of when writing an OpenCL version of a code consisting of many different classes, is how to manage the various OpenCL objects necessary throughout the code. We chose to create a meta object class *OCL* that contains all the data and to which a pointer gets passed around. A simplified version is shown in Code 4.8. This meta class performs 5 main tasks:

1. In Lines 5 to 10 it tries to find a platform to run on. Such a platform can be, e.g., *Intel OpenCL* or *NVIDIA CUDA*.

2. After a platform is found, it is checked for possible devices to run on, Lines 12 to 17. This could be, e.g., an *Intel Xeon Phi* device or an *NVIDIA Tesla K20Xm* GPU.

3. Once a platform and device is found, a context is created on the device, Line 19. A context is used by the OpenCL runtime for managing queues, memory, program and kernel objects and for executing kernels on a device specified in the context.

4. Inside of the context, in Line 20, a command queue is created. All kernel calls will be enqueued in this queue and are executed either in-order or out-of-order. We will be running all of them in-order.

Code 4.8: OpenCL: OCL meta class, simplified

```
1  class OCL {
2  public:
3      explicit OCL(...) {

5          std::vector<cl::Platform> all_platforms;
6          cl::Platform::get(&all_platforms);
7          if(all_platforms.size() == 0)
8              throw cl::Error(CL_INVALID_PLATFORM);
9          else
10             platform = all_platforms[0];

12         std::vector<cl::Device> all_devices;
13         platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devices);
14         if(all_devices.size() == 0)
15             throw cl::Error(CL_INVALID_DEVICE);
16         else
17             default_device = all_devices[0];

19         context = cl::Context({default_device});
20         queue = cl::CommandQueue(context,default_device);

22         std::string all_opencl_kernels = Kernels::getString();
23         programs = cl::Program(context, all_opencl_kernels, true);
24     }

26     cl::Platform platform;
27     cl::Device default_device;
28     cl::Context context;
29     cl::CommandQueue queue;
30     cl::Program programs;
31 };
```

5. Finally, in Line 22, we retrieve all the kernels written from another meta class that simply parses a few text files containing the kernels. They are then compiled in Line 23 into an executable format lying on the GPU. These kernels are now ready to be called by our code.

After having established all the needed OpenCL objects, we can turn our attention to converting a function from serial to OpenCL. This can be a relatively straightforward task, provided the serial version does not write to the same memory address at two different stages in the algorithm. If it does not, then the main task of converting to OpenCL lies in computing the correct indices based on where in the order each individual kernel call is. If the serial code does write to the same memory in more than

one location, this task becomes a little more complicated, as parts of the code most likely have to be re-written to avoid concurrent writes to the same memory address.

As an example, let us consider the *restrict()* function for a Q1Vector, Code 4.9.

Code 4.9: OpenCL: Q1Vector::restrict()

```
1  kernel void q1vector_restrict(global const double * const fine_input,
2                                global const int * const _coarse_xdimension,
3                                global const int * const _coarse_ydimension,
4                                global double * const coarse_output) {
5
6      unsigned int coarse_index = get_global_id(0);
7      unsigned int coarse_xdimension = *_coarse_xdimension;
8      unsigned int coarse_ydimension = *_coarse_ydimension;
9      unsigned int fine_xdimension = coarse_xdimension*2-1;
10
11     if(coarse_index >= coarse_xdimension*coarse_ydimension) return;
12
13     unsigned int fine_index = (2*(coarse_index/coarse_xdimension)+1)*(
           fine_xdimension+2) + 2*(coarse_index%coarse_xdimension)+1;
14
15     double val = fine_input[fine_index];
16     val += 0.5 * fine_input[fine_index-1];
17     val += 0.5 * fine_input[fine_index+1];
18     val += 0.5 * fine_input[fine_index-(fine_xdimension+2)];
19     val += 0.5 * fine_input[fine_index+(fine_xdimension+2)];
20     val += 0.25 * fine_input[fine_index-1 + (fine_xdimension+2)];
21     val += 0.25 * fine_input[fine_index-1 - (fine_xdimension+2)];
22     val += 0.25 * fine_input[fine_index+1 + (fine_xdimension+2)];
23     val += 0.25 * fine_input[fine_index+1 - (fine_xdimension+2)];
24
25     coarse_output[(coarse_index/coarse_xdimension+1)*(coarse_xdimension
           +2) + coarse_index%coarse_xdimension+1] = val;
26 }
```

First, in Line 6, we store the current global id. This is the index of the current call of the kernel and corresponds to the index in our coarse mesh. Lines 7-9 are for our convenience, storing some values that are used multiple times in variables. In Line 11 we need to check if the current kernel index is a valid mesh index. OpenCL groups all kernel calls into work groups of a certain size (typical choices are 128, 192, or 256). Thus, it needs to pad the indices coming from the mesh to produce a number divisible by the work group size. Obviously, there is no need to do any work for these indices. In Line 13, we compute the find-grid index (including ghost elements) corresponding to the coarse-grid index, which we then use in Lines 15-23 as a base for the index of the surrounding nodes for computing the weighted average. Finally, in Line 25, we

write the computed value to the global array.

This kernel can be called from our C++ code with just a few lines of code, given in Code 4.10. First, we compute the coarse-grid dimensions, Lines 1-2, and we create a

Code 4.10: C++: Calling OpenCL kernel for Q1Vector::restrict()

```
1  unsigned int coarse_xdimension = xdimension/2+1;
2  unsigned int coarse_ydimension = ydimension/2+1;

4  Q1Vector ret(coarse_xdimension, coarse_ydimension, ocl, onAnDevice);

6  try {
7      cl::Buffer coarse_xdimension_buf(ocl->context,
8                  &coarse_xdimension, (&coarse_xdimension)+1,
9                  true);

11     cl::Buffer coarse_ydimension_buf(ocl->context,
12                 &coarse_ydimension, (&coarse_ydimension)+1,
13                 true);

15     auto kernel = cl::make_kernel
16                     <cl::Buffer&, cl::Buffer&, cl::Buffer&, cl::Buffer&>
17                     (ocl->programs, "q1vector_restrict");

19     kernel(cl::EnqueueArgs(ocl->queue,
20                     cl::NDRange(ret.ghost_num_nodes__global_ocl),
21                     cl::NDRange(ret.ghost_num_nodes__local_ocl)),
22                     elements_buf[0], coarse_xdimension_buf,
23                     coarse_ydimension_buf, ret.elements_buf[0]);

25     cl::copy(ocl->queue, elements_buf[0], &nodes[0], (&nodes[
           ghost_num_nodes-1])+1);

27 } catch(cl::Error error) {
28     ocl->displayException("Q1Vector::restrict()",
29                           error.what(), error.err());
30     exit(1);
31 }
```

new Q1Vector object on the coarser mesh, Line 4. We wrap the actual use of OpenCL objects and functions into a *try-catch* block, Lines 6-31, as this allows us to catch any exception thrown by OpenCL and display a somewhat meaningful error message, Lines 28-29 before terminating our code, Line 30. Such an exception can be thrown, e.g., when the GPU is running out of memory, or when a faulty kernel call was done. Inside of the *try* block, we first create two buffers on the GPU holding the coarse-grid dimensions, Lines 9-13. Our code relies heavily on C++11 features, that allows us to

do many things with OpenCL with a single line of code. Without C++11, this would require multiple lines of code and would lead to slightly more cluttered code. With the two buffers set up, we then proceed to create the OpenCL kernel. Again, this can be done conveniently in a single line of code, here broken over the Lines 15-17. This call consists of three parts: In Line 15 we call the OpenCL function *make_kernel()*, in Line 16 we specify how many parameters the kernel will take and of what type they are (here we have four OpenCL buffers), and in Line 17 we specify the name of the kernel and where to look for it. The return type of the *make_kernel()* function is very complex, but as it is unambiguously defined, we can simply denote it to be of type *auto* and let the compiler fill the specifics in. Calling the kernel, Lines 19 to 23, is once again very straightforward, although it does require a number of parameters to be passed. In Line 19 we call the *kernel()* function, which takes as its first parameter an object of type *cl::EnqueueArgs*. This object is defined over the three lines 19 to 21 and contains (a) the command queue, Line 19, (b) the total number of kernel calls to perform, Line 20, and (c) the local work group size, Line 21. Following all this, the remaining arguments, Lines 22 to 23, for the *kernel()* function are simply the buffers that are passed on as parameters to the kernel itself. The number of parameters here and their type has to be identical to what was specified in Line 16. After all this is done, the Q1Vector object *ret* contains the computed values, the final thing that is left to do is to copy the values from the GPU back to the CPU, Line 25. This call is also very straightforward, it takes 4 parameters: (1) The command queue, (2) the memory on the device, (3) the memory on the host, and (4) how much data to copy.

In a similar fashion all of the computations done in serial have been converted to OpenCL kernels with corresponding calls from the C++ code. Once that is all done, both versions of the code should produce the exact same results when run with the same parameters.

Chapter 5

Numerical Results

The implementation for solving the Stokes equations was run on a server machine equipped with 2 Intel Xeon E5-2650 v2 CPUs with 128GB DDR3 RAM and 2 NVIDIA Tesla K20Xm GPUs with 6GB GDDR5 SRAM. Each Intel CPU has a total of 8 cores with each core having 2 threads, giving a total of 32 threads (16 cores), all of them clocked at 1866MHz. Each NVIDIA GPU has a total of 2688 processor cores, each clocked at 732MHz.

For our first experiments, we let our code run on either a single CPU or a single GPU. Ultimately, the goal of this project is to target heterogeneous systems, making use of as much of the computing power available as possible. This is discussed further in Section 6.2.

Any of our numerical experiments were run for a relative convergence of 10^{-10}, i.e., until the 2-norm of the residual at step m divided by that of the initial residual is less than or equal to 10^{-10}.

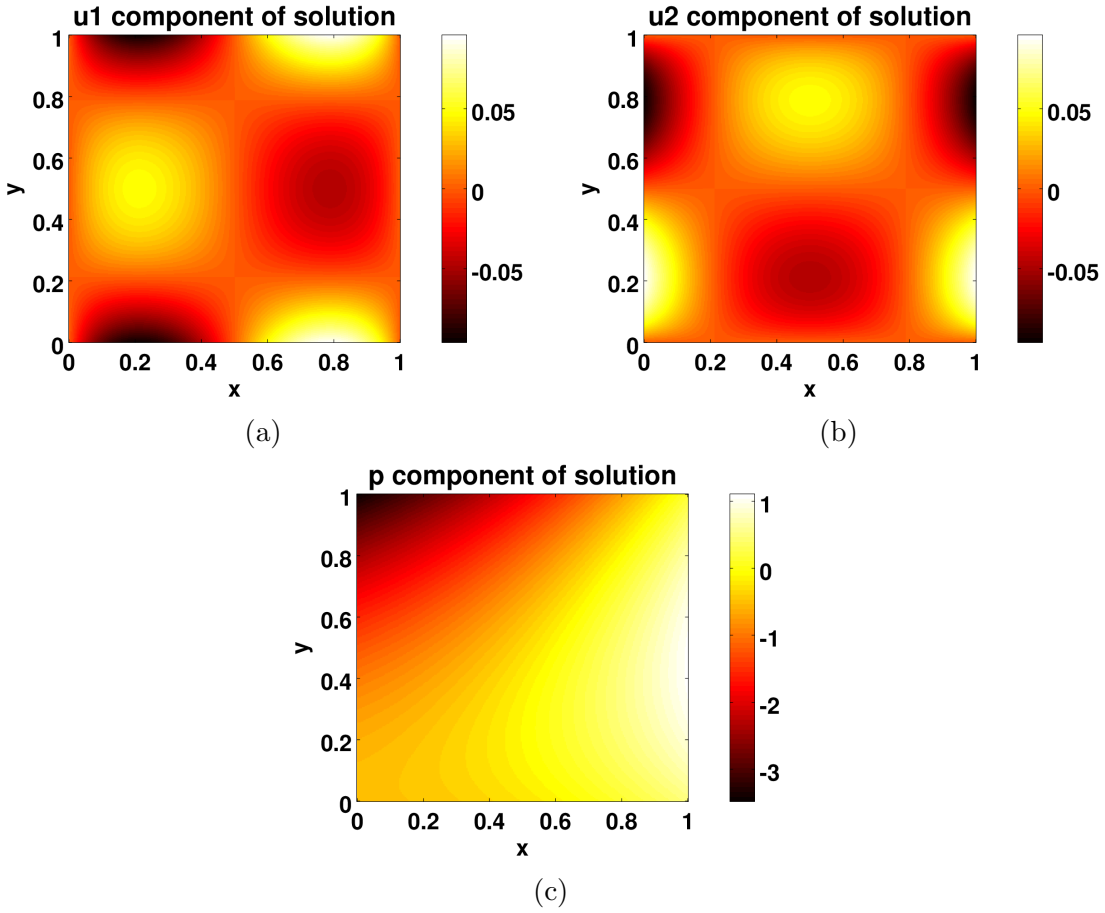
5.1 Sample Problem

For all our experiments, we will be using the sample problem from [31, equations 7.1 and 7.2] with the known analytical solution

$$\mathbf{u}^* = \begin{pmatrix} x(1-x)(2x-1)(6y^2-6y+1) \\ y(y-1)(2y-1)(6x^2-6x+1) \end{pmatrix} \tag{5.1}$$

$$p^* = x^2 - 3y^2 + \frac{8}{3}xy \tag{5.2}$$

We compute the right-hand side vector, $\mathbf{f}$, by substituting (5.1) and (5.2) into (3.1). For simplification, we will fix the viscosity constant $\nu = \frac{1}{2}$. A visualisation of this analytical solution is shown in Figure 5.1. To verify the predicted convergence rates for u_x, u_y, and p, we computed the difference between our discretised solution and the continuous solution, which indeed approaches zero as the number of elements in the mesh increases.



(a) (b) (c)

Figure 5.1: Visualisation of analytical solution:
(a) u_1 component, (b) u_2 component, (c) p component

5.2 Parameter study

There are a few parameters in our code that we need to choose. The first set of parameters that we will take a closer look at are the parameters involved in the

Braess-Sarazin smoother: the Jacobi weight ω, and the scaling factor t. The Braess-Sarazin weight ω_{BS} is an underrelaxation parameter for the global update. Its best value depends on the actual problem. For our experiments, we will keep it fixed at 1, as this simplifies our analysis. Following this study, we will analyse the best choice of number of levels for our multigrid V-cycle preconditioner and what should be done on the coarsest level each time (exact vs. approximate solve).

5.2.1 Braess-Sarazin parameters

Finding appropriate values for both t and ω for the Braess-Sarazin smoother has to be done simultaneously, as the best choice for one might affect the other. Based on some small experiments, it seemed fitting to let ω vary over the range $[0.7, 2]$ and t over the range $[0.1, 1]$. Doing so for various grid sizes (64x64, 128x128, 256x256, 512x512) yields the results as shown in Figure 5.2.

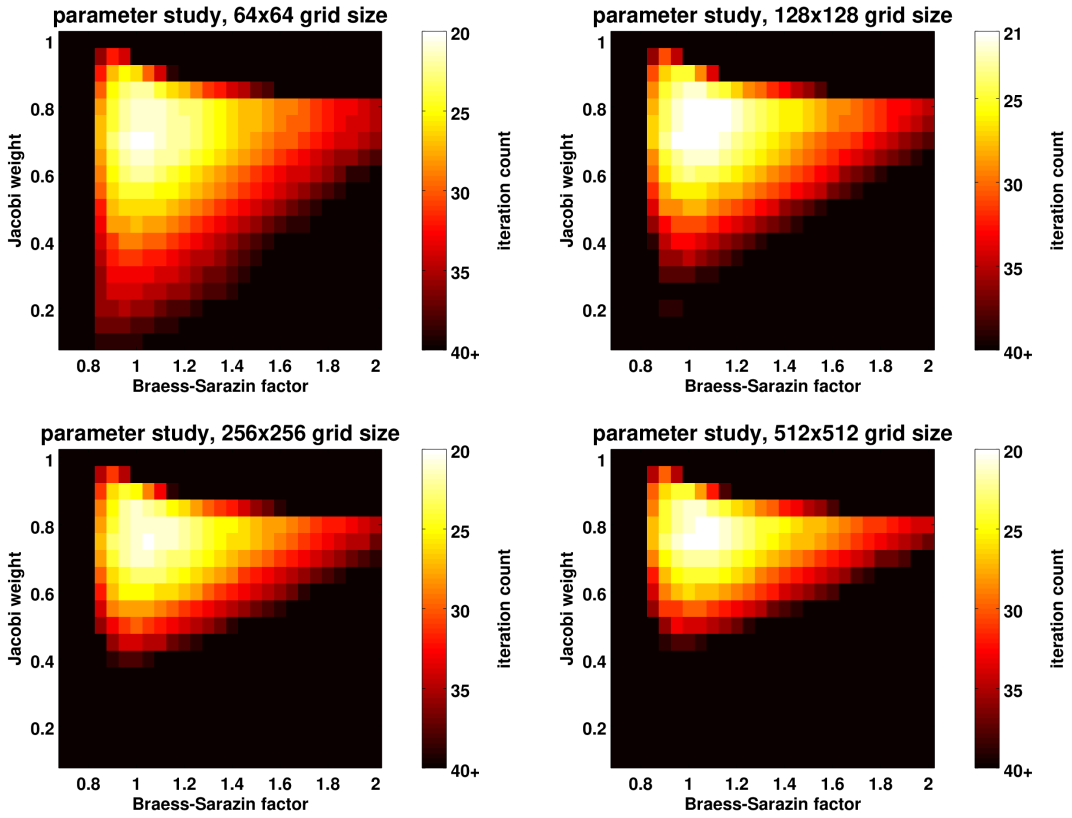


Figure 5.2: Braess-Sarazin parameter study: ω vertically, t horizontally; grid sizes: 64x64, 128x128, 256x256, 512x512

There are a few things we can observe from the graphs:

1. There is a clear but small range for t that gives best convergence across all grid sizes with the best values between 1 and 1.1.

2. The optimal range for ω shifts slightly as the grid size increases. For the 64x64 grid size, the best value seems to be 0.7. But for the 256x256 and 512x512 grid size the optimal value shifts to 0.75 to 0.8.

3. Choosing too large of a value for t is less of a problem than choosing too small of a value. While a value of $t = 0.8$ exhausts the maximum iteration count for all choices of ω, a value of $t = 1.5$ might still work okay (depending on the corresponding choice of ω).

As a result of this parameter study, our choice of parameters will be $t = 1.05$ and $\omega = 0.75$ for the remainder of this thesis, unless stated otherwise.

5.2.2 V-Cycle Level Depth

The depth of the V-cycle can potentially have a significant impact on the overall performance. Even though the iteration count remains about the same when changing this, doing a direct solve on too large of a coarse system can result in a much increased overall solve time, as can going too far down to do an approximate solve introducing unnecessary computational overhead (particularly for OpenCL).

To find out which level depth works best, the GMRES code was run on four different grid sizes with a varying size of the coarsest grid in the V-cycle for both an approximate and an exact solve on the coarsest grid. Figure 5.3 shows the results for the grid size of 512x512. The results for other grid sizes yield the same overall picture. Note that in the the left-hand subplots (a) and (c) the limits of the x-axis vary depending on which line is considered. The individual x-axis limits for each line are shown in the legend of these plots.

Looking at Figure 5.3, parts (a) and (c), we can see that no matter what setting we choose for an approximate solve on the coarsest grid of the V-cycle, doing an exact solve gives slightly better results. The effect is rather minuscule in the serial case, Figure 5.3 (a), compared to an approximate solve on the coarsest grid possible (2x2), however, for the parallel implementation an exact solve offers us a speedup of about

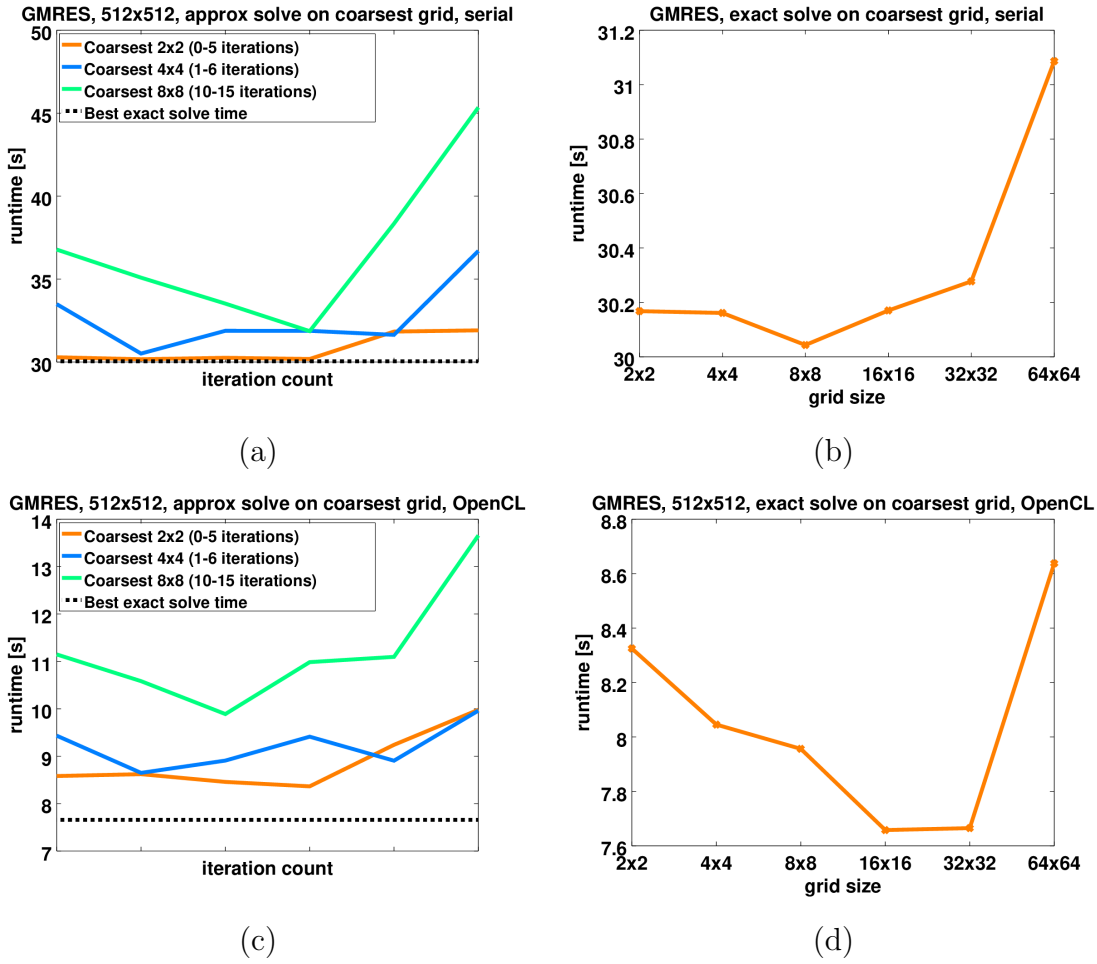


Figure 5.3: Approximate vs. exact solve on various coarsest grid sizes

12%, Figure 5.3 (c). This tells us that we cannot go wrong by opting to do an exact solve all the time and discard the approximate solve for our numerical experiments.

When doing an exact solve on the CPU the depth of the V-cycle does not make much of a difference, as long it is "deep enough", Figure 5.3 (b). Taking any coarsest grid coarser than 32x32 is observed to give roughly equally fast results. Anything larger, though, will cause the direct solve on the coarsest grid to take up too much time, yielding an overall increase in computation time.

For the GPU version, we get consistently the best results when choosing a 16x16 or 32x32 grid as our coarsest mesh size for the exact solve, Figure 5.3 (d). The timing difference between these two grid sizes is only a few milliseconds, they are essentially equal. For smaller meshes, the problem to be solved is simply not large enough for the GPU; thus, the overhead that we introduce by moving data to and from the

GPU becomes more significant than the increased communication cost of doing a direct solve on the 16x16 or 32x32 mesh. Similarly to the serial version, an even larger coarsest mesh results in a too complex direct solve, coupled with an increasing amount of necessary communication between the GPU and CPU.

Based on this analysis, we will perform recursive coarsening to a 2x2 grid for our serial version, and to a 32x32 grid for our parallel version, doing an exact solve on the coarsest grid in both versions.

5.3 Results

With all our parameters set, we will now take a closer look at how our implementations (CPU and GPU) perform, comparing them to an unstructured grid solver in Trilinos with identical setup [21, 22]. The parameters are chosen as discussed in the previous sections. We ran all three versions multiple times to ensure to get their best timing possible. For all experiments, we ran all three versions over a large range of grid sizes: 64x64, 96x96, 128x128, 192x192, 256x256, 384x384, 512x512, and 768x768. Given the elements we have chosen and the setup we are using, the total number of degrees of freedom for each problem is much larger than it might seem at first. Table 5.1 gives an estimate of the number of degrees of freedom associated with each grid size and the corresponding number of non-zero entries in the system matrix. Even though for our CPU implementation we can go to larger grid sizes than 768x768, this is the limit for the GPU version. In fact, on the GPU, we can only run a maximum of 24 iterations of GMRES at this grid size, as this already exhausts the available memory. However, as can be seen in Table 5.2, this is sufficient to achieve convergence, given the right parameter choice. Should more iterations be required, e.g., to achieve even higher accuracy, an implementation of restarted GMRES, Subsection 3.5.2, might be a possible solution. Extending our current implementation to allow for this is a rather straight-forward task. Our aim is to eventually target large heterogeneous systems, Section 6.2, thus for most applications this won't be necessary. On such systems each GPU only operates on a small sub-partition of the full grid pushing each individual grid size most of the time back into these size constraints and thus avoiding the downsides of *restarted* GMRES. For the few applications where this poses a problem, such an extension to our implementation is one possible solution.

There are two different timings that we can compare: The setup time and the solve

grid size	degrees of freedom (millions)	number of non-zeros (millions)
64x64	0.037	0.984
96x96	0.084	2.201
128x128	0.148	3.902
192x192	0.334	8.752
256x256	0.592	15.54
384x384	1.331	34.90
512x512	2.364	62.01
768x768	5.316	139.4
1024x1024	9.447	247.7
1536x1536	21.25	557.2
2048x2048	37.77	990.4

Table 5.1: Number of degrees of freedom and non-zeros for various grid sizes

time. Our main focus will be on the solve time, as we focussed on optimising this part when writing our code. Nevertheless, even though there are still various optimisations of the setup phase possible, our code is already significantly more efficient than the Trilinos solver we are comparing against. Figure 5.4 (a) shows the total setup time of the two serial codes and Figure 5.4 (b) shows the same data on a loglog scale. Clearly, our code is significantly faster in setting up the problem compared to the Trilinos code, up to 2.5 times. Also, our code scales much better: In the loglog plot, the slope of the best-fit line for the Trilinos code is about 1.10, whereas the slope of the best-fit line for our code is only 1.015. This means that our code scales almost perfectly and that its complexity grows almost linearly with the grid size. However, the Trilinos code scales slightly worse than linearly. This difference doesn't seem like much, but as we can see in Figure 5.4 it eventually makes the large difference of over a factor of 2 on a 2048x2048 grid. One of the reasons why the Trilinos code is slower and scales worse in setting up the system lies in the type of matrix assembly required. Assembling a matrix for an unstructured system is known to typically take a substantial amount of time and is one of the primary disadvantages of finite elements over finite differences. Careful algorithm design can reduce the associated cost making it scale close to linearly (as we observed in Figure 5.4), but it is still an area of ongoing

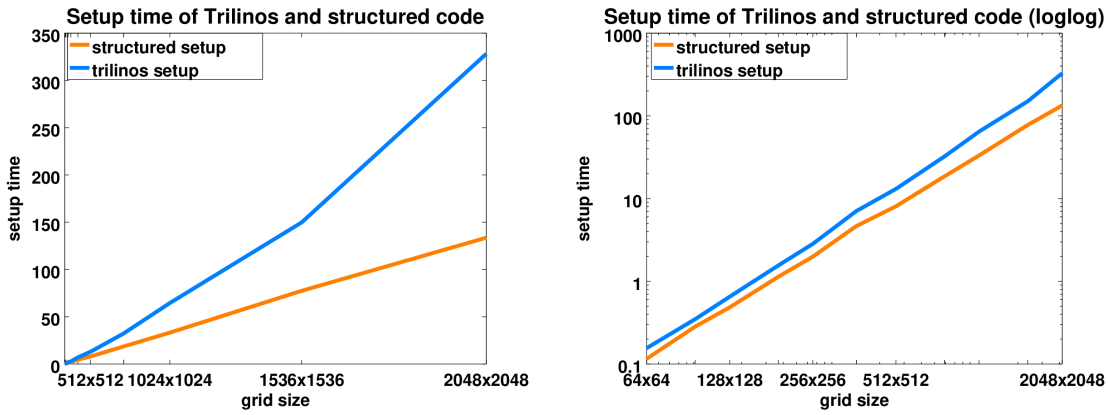
grid size	iteration count
64x64	21
96x96	21
128x128	21
192x192	20
256x256	20
384x384	20
512x512	20
768x768	20
1024x1024	20
1536x1536	20
2048x2048	21

Table 5.2: Iteration count for various grid sizes for a relative convergence tolerance of 10^{-10}

research. For more details, see [32, 33] and [34, Chapter 6].

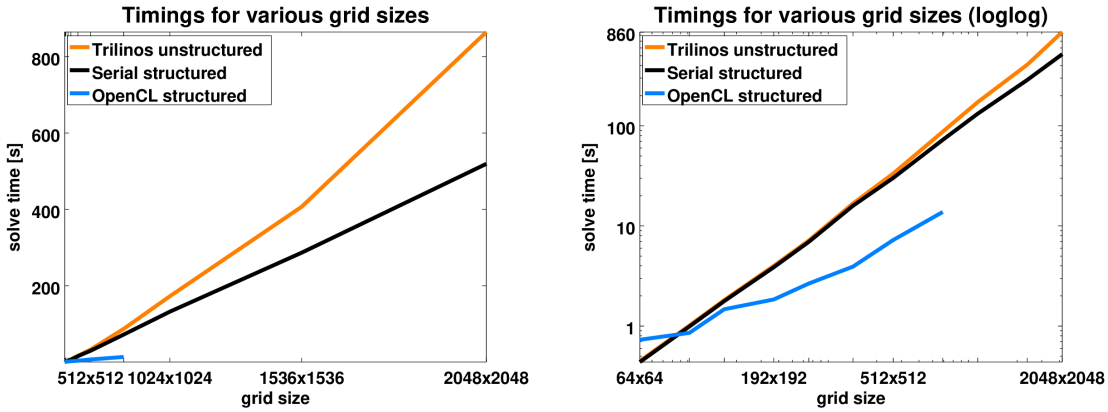
For the remainder of our analysis, we will focus solely on the GMRES total solve time. The first part of Figure 5.5, part (a), compares the actual solve time in seconds of the three versions, while the second part, Figure 5.5 (b), puts the same data onto a loglog scale with the slope of the lines showing the respective solver behaviour.

As we can see, both of our versions, serial and parallel, outperform the Trilinos solve. In particular, considering the second graph, we see that the serial version of our implementation has a small but clearly noticeable better convergence behaviour than the Trilinos solve. On the loglog scale, the best-fit slope of the Trilinos solve time is 1.091 whereas our serial version has a best-fit slope of 1.0275. Our parallel version on the GPU clearly outperforms either serial version, as we hoped it would. It also scales better on the GPU: Doubling the grid size increases the solve time by a factor less than 2, i.e., it grows less than linearly with the grid size. To find its best-fit slope on the loglog scale, we should ignore the first three data points, i.e., the data points for the 64x64, 96x96, and 192x192 grid size, as the GPU does not scale well for grid sizes that are too small. Thus, considering all the data points starting with the 192x192 grid size, the best-fit slope on the loglog scale is about 0.727. Being less than 1 is due to the fact that the GPU can handle large amounts of small calculations



(a) (b)

Figure 5.4: Comparison of setup time in seconds, (a) normal and (b) loglog scale



(a) (b)

Figure 5.5: Comparing (a) actual solve times, (b) solve times on loglog scale

much better than smaller amounts, i.e., larger problems can take better advantage of the computing power of the GPU. This can also be seen by looking at Table 5.3.

The reason why the GPU shows bad performance for small problem sizes lies in the way GPUs are designed to operate. While a CPU consist of a few cores optimised for sequential processing, a GPU consist of typically thousands of smaller cores. Each individual core of the GPU offers lower performance than each individual core of the CPU, however, all the cores on the GPU combined are very efficient in handling multiple tasks simultaneously. Thus, if the problem size is very small, only a small partition of all the cores on the GPU are active at any given time. Since each individual core is significantly slower than the CPU equivalent, this results in the observed lower performance overall.

grid size	factor	Trilinos	factor	Structured	factor	Parallel	factor
64x64	—	446.6	—	437.00	—	729.49	—
96x96	2.25	1010	2.26	987.44	2.26	854.30	1.17
128x128	1.78	1819	1.80	1777.4	1.80	1472.7	1.72
192x192	2.25	3992	2.20	3868.2	2.18	1842.9	1.25
256x256	1.78	7167	1.80	6935.8	1.79	2659.6	1.44
384x384	2.25	16780	2.34	15923	2.30	3925.4	1.48
512x512	1.78	33410	1.99	30202	1.90	7240.0	1.84
768x768	2.25	87230	2.61	72327	2.39	13775	1.90

Table 5.3: Solve times in milliseconds and their growth rate factors (how much each value grows relative to the previous row)

The first two columns of Table 5.3 show the grid sizes and by what factor each grid size has increased relative to the next smaller grid size. Similarly, the following three sets of two columns show how the timings grow relative to the solve time of the next smaller grid size. As can be seen, the Trilinos solve time (with one exception), though close, does not grow perfectly linearly with the grid size, but slightly worse. The serial version of our implementation behaves similarly, though it's closer to being linear than the Trilinos solver. The parallel version behaves very nicely, growing on average by a factor of 1.54 per doubling of the grid size (on average).

We can also analyse the results from another angle, the speed-up of our own implementation compared to the Trilinos solver. These two comparisons are shown in Figure 5.6.

These graphs confirm what we have seen already, that both the serial and parallel version of our implementation offer better performance than the Trilinos solver. On the largest grid size that we can run on the GPU, the speedup that we can achieve with our parallel version compared to the Trilinos code is about a factor of 6.3. There's only a single case where the parallel version has lower performance than Trilinos, for the smallest grid size of 64x64. This is due to the overhead of operating on the GPU, and the comparably little work involved in solving this small problem.

In particular, when looking at the data from different angles, it is clear that the move from an unstructured grid solver (Trilinos) to a structured grid solver (our own implementation) offers an advantage in terms of performance. Also, the structured

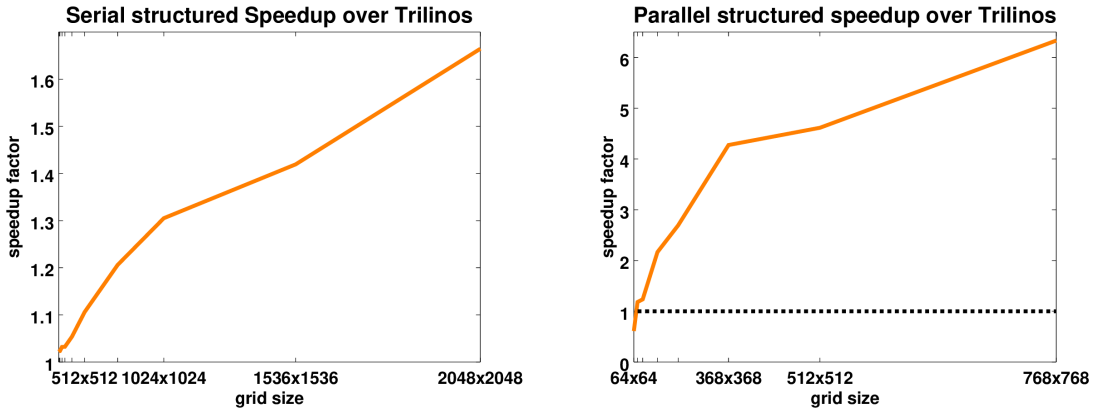


Figure 5.6: Speed-up of our serial and parallel structured versions relative to Trilinos

grid implementation makes it very easy to add fine-grained parallelism, which is of advantage when moving to the GPU.

5.4 Mass matrix

To conclude, we will explore a variant of our Braess-Sarazin solver, demonstrated in Section 3.9. In our implementation of Braess-Sarazin, we compute the triple matrix required for the approximate Schur complement, see (3.36). This is a very expensive calculation and rather difficult to implement, a snippet is shown in Code 4.2. Thus, it is of interest to find a viable alternative that is cheaper to compute and yet offers comparable efficiency.

From [15, 35], we know that the continuous inf-sup condition

$$\sup_{\mathbf{u}\in H^1} \frac{p\nabla\cdot\mathbf{u}}{||\mathbf{u}||_1} \geq \gamma||p||_{L_2}, \tag{5.3}$$

always holds. However, going from the continuous to the discrete case, this inequality might not always be true, since the continuous u that maximises $\frac{p\nabla\cdot u}{||u||_1}$ may not be in the discrete space. Writing the discrete form as

$$\max_{\mathbf{u}} \frac{pB^T\mathbf{u}}{(\mathbf{u^T}A\mathbf{u})^{1/2}} \geq \gamma(p^TMp)^{1/2}, \tag{5.4}$$

and doing a simple change of variable, $\mathbf{z} = A^{1/2}\mathbf{u}$, gives us

$$(pB^T\mathbf{u})^T = \mathbf{u}^\mathbf{T}Bp = \mathbf{z}^\mathbf{T}A^{-1/2}Bp, \tag{5.5}$$

which allows us to re-phrase the above inequality as

$$\max_{\mathbf{z}} \frac{\mathbf{z}^\mathbf{T}A^{-1/2}Bp}{(\mathbf{z}^\mathbf{T}\mathbf{z})^{-1/2}} \geq \gamma(p^TMp)^{1/2}. \tag{5.6}$$

The max of this inequality over the discrete space is achieved by

$$\mathbf{z}^\mathbf{T} = p^TB^TA^{-1/2}, \tag{5.7}$$

giving us

$$\frac{p^TB^TA^{-1}Bp}{(p^TB^TA^{-1}Bp)^{1/2}} \geq \gamma(p^TMp)^{1/2}. \tag{5.8}$$

Simplifying the left-hand side of the inequality, re-arranging some terms and squaring both sides, we get

$$\Gamma^2 \geq \frac{p^TB^TA^{-1}Bp}{p^TMp} \geq \gamma^2. \tag{5.9}$$

The proof of the upper bound, Γ^2, is trivial and can be found in [15]. This tells us that the two matrices $B^TA^{-1}B$ and M are spectrally equivalent. Thus, instead of doing the work to compute the triple matrix product for the approximate Schur complement, we should be able to simply replace that with the easy-to-compute mass matrix, M.

In order to find out how well the mass matrix performs in practise and how it compares to the triple matrix product, we first need to do another parameter study for the Braess-Sarazin parameter t and the Jacobi weight ω, similarly to what was done in Subsection 5.2.1.

5.4.1 Parameter study

Based on some small experiments, a good test range for the two parameters appears to be $[0.8, 1.4]$ for the Braess-Sarazin factor, t, and $[1.4, 2.6]$ for the Jacobi weight, ω. The result of this study is shown in Figure 5.7.

The optimal choice of values appears to be around $t = 1$ and $\omega = 2$. Something interesting to note is that we need a value for ω well greater than 1 in order to get

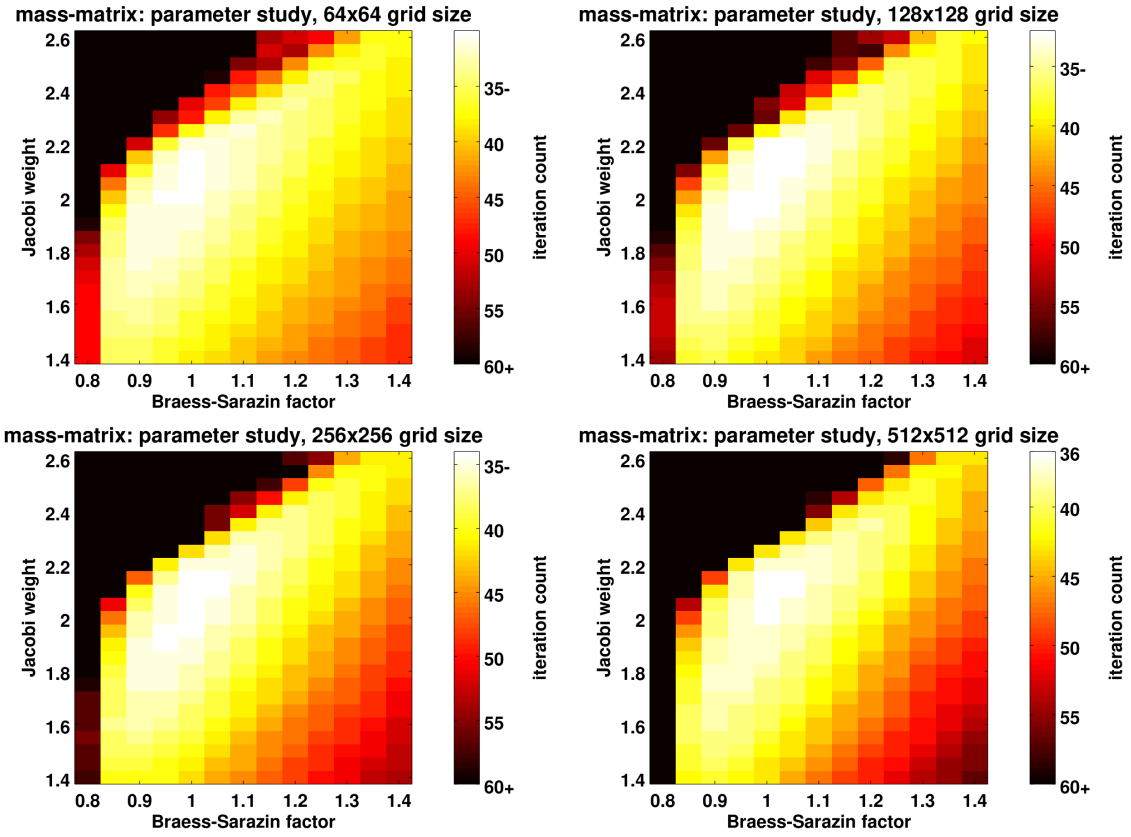


Figure 5.7: Parameter study of Braess-Sarazin parameter t and Jacobi weight ω for various grid sizes

somewhat comparable performance, in contrast to weighted Jacobi where ω would be expected to lie inside $[0, 1]$; thus, we have to employ rather extreme overrelaxation. We will be using the aforementioned values for the two parameters in the following analysis for comparing the efficiency of the two variants.

5.4.2 Comparison

While the mass matrix is easier to compute than the triple matrix product, this does not guarantee an overall speedup. Figure 5.8 shows the solve times of both variants on a loglog scale for various grid sizes. Both variants were run with their respective optimal pairs of parameters until a relative reduction in the 2-norm of the residual of at least 10^{-10} was achieved. We can see that the triple matrix product consistently yields significantly faster solve times than when using the mass matrix in its place. In fact, when using the mass matrix we need more iterations to converge to the same

level of accuracy, leading to solve times of between 50% and 100% longer!

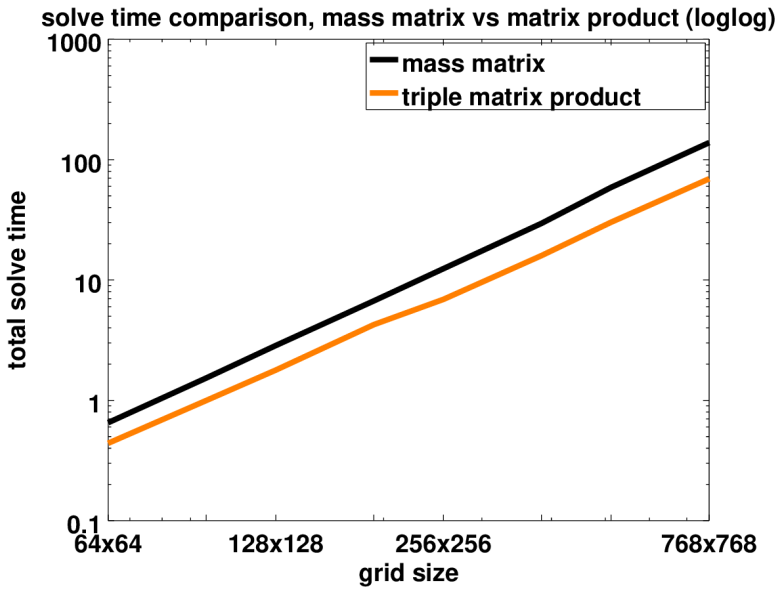


Figure 5.8: Comparison of timings when mass matrix and when triple matrix product is used for Braess-Sarazin

However, we save a certain amount of time when setting up the Braess-Sarazin solver with the mass matrix. If these savings are larger than the additional cost paid during the solve phase, this approach could very well be considered as a viable alternative. Table 5.4 shows the time that is required for computing the mass matrix and the time required to compute the triple matrix product, for comparison to the time lost during the solve phase. It is worth noting that for either variant we do not compute the full matrix, as we only require the diagonal of the matrices for the single Jacobi iteration. The timings paint a very clear picture. Computing the mass matrix is done very quickly, almost instantaneously, whereas the triple matrix product takes between 220 and 720 times longer to compute. Even though the relative saving is immense when choosing the mass matrix, the absolute time saved is significantly less than the absolute time lost due to an increased iteration count, as shown in Table 5.5. When working with a problem of size 768x768, the overall time lost by choosing to work with the mass matrix is about 67.5 seconds.

Even though it does require significantly more work to implement the triple matrix product efficiently compared to the near-trivial computations required in order to obtain the mass matrix, it pays off eventually. Table 5.6 confirms this by showing the total iteration count required for various grid sizes. It is also important to note that

grid size	matrix product	mass matrix
64x64	8.894	0.017
96x96	28.37	0.039
128x128	41.71	0.068
192x192	95.69	0.154
256x256	174.2	0.684
384x384	399.9	1.771
512x512	717.4	3.200
768x768	1604	7.237

Table 5.4: Comparison of timings (in milliseconds) when setting up the mass matrix vs. triple matrix product

the solve time per iteration remains about the same, as we are still doing the exact same amount of calculations during the solve phase.

grid size	setup time saved	solve time lost	balance
64x64	8.88	212.24	203.36
96x96	28.33	535.72	507.39
128x128	41.65	1069.6	1027.95
192x192	95.53	2447.5	2351.97
256x256	173.5	5512.5	5339.0
384x384	398.1	13587	13188.9
512x512	714.2	28578	27863.8
768x768	1597	69100	67503

Table 5.5: Time saved during setup vs. time lost during solve, in milliseconds

grid size	iteration count	
	triple matrix product	mass matrix
64x64	21	30
96x96	21	31
128x128	21	32
192x192	20	33
256x256	20	34
384x384	20	35
512x512	20	36
768x768	20	37
1024x1024	20	38
1536x1536	20	39
2048x2048	21	41

Table 5.6: Comparison of iteration count when using triple matrix product and when using mass matrix

Chapter 6

Conclusion and Future Work

6.1 The Stokes equations

For the Stokes equations, we considered the commonly used Q2-Q1 or Taylor-Hood elements, which are piecewise biquadratic in velocity (Q2) and bilinear in pressure (Q1). Using these elements, we cover the domain in identical rectangles, giving us a very nice geometric structure of the underlying mesh. This approach produces very sparse and regular matrices, with only a few non-zero entries in each row of the system matrix. As this number of non-zeros per row is fixed and not dependent on the size of the problem, this allowed us to exploit this property by formulating all of our calculations in a clean stencil form, removing the need to use general sparse matrix storage.

To solve the Stokes equations, we used preconditioned GMRES, the Generalised Minimal Residual method. We employed a multigrid V-cycle as the preconditioner, using Braess-Sarazin as the relaxation method. We use Braess-Sarazin instead of standard Jacobi or Gauss-Seidel type methods, as these cannot be applied to the saddle-point system at hand. For classical Braess-Sarazin, we had to compute the triple matrix product $BD^{-1}B^T$, required for the Schur complement of the block-matrix used in the Braess-Sarazin update. This allowed us to then employ a standard weighted Jacobi relaxation scheme on an equation yielding an update to the pressure. Based on the pressure update, we were then able to compute an update to the velocity.

The full GMRES algorithm with the multigrid preconditioner was implemented using C++. Due to its object-oriented nature, this simplified our task of writing the code tremendously, though there still remained some challenges in realising an efficient

algorithm. We used OpenCL to parallelise our serial C++ algorithm for GPUs. As all our calculations naturally break up into small and independent calculations, this allows us to use the same logic in both the serial and parallel cases. Also, as we heavily exploited the geometric structure of the underlying mesh, the memory requirements were low enough so that even though we were only using a single GPU, we anticipated being able to run our code with relatively large grid sizes. This was confirmed during our numerical experiments.

Our first experiments were used in order to choose some of the parameters in our algorithm. In particular, we had to pick two parameters for the Braess-Sarazin relaxation scheme, a weight for Jacobi, ω, and a scaling factor, t, and we had to consider parameters within the multigrid V-cycle, determining the depth of the cycle and whether to use an exact or approximate solve on the coarsest level. Our experiments showed that it is most efficient to always do an exact solve instead of an approximate solve. When operating on a single CPU, going as far down as possible, to a 2x2 element patch, yielded the best results, whereas it was beneficial to stop at a grid of size 32x32 when running on a single GPU.

All of our numerical results for the overall algorithm were compared to a Trilinos solver [21], implementing the same algorithm using an unstructured grid setup. We have seen that our structured-grid implementation clearly outperforms the Trilinos solver, with the amount of work growing almost perfectly with the grid size. These results were almost identical for considering both the setup and solve phases. Our parallel version outperformed both serial versions clearly, as expected. Compared to the Trilinos solver, we were able to achieve a speedup of about 6.3 on a grid of size 768x768, and compared to our own serial implementation we achieved a speedup of about 5.25 on the same grid size. Also, the amount of time required grows less than linearly with the grid size, at a rate of about 0.73 for large enough grid sizes. This is due to the fact that the more independent calculations a GPU has to do, the better it can perform.

We also shortly explored the possibility of replacing the complicated triple-matrix product in the Braess-Sarazin smoother by the mass matrix of the system matrix, A, as suggested by Prof. Wathen in personal conversation. We have seen that even though the mass matrix and the triple matrix product are spectrally equivalent, the mass matrix leads to poorer performance of the overall algorithm. The time saved due to an almost trivial setup is minuscule compared to the time lost during the solve

phase.

6.2 Future Work

Using a single GPU already yields very nice results, as we have seen. However, this is not where we want to stop. Instead of only using a single CPU or a single GPU, we want to be able to employ large heterogeneous systems with many CPUs and many GPUs.

Extending our structured-grid implementation to such heterogeneous systems can be done by using MPI. In partitioning our unit square domain into non-overlapping partitions, this allows us to break our computations up into smaller pieces that can each be computed by a different CPU and/or GPU. It does introduce the need of communication between the threads of computation; however, the amount of communication can be kept at a minimum by ensuring that we

1. partition the large square domain into smaller square or rectangular partitions, minimising the length of the boundary between different partitions;

2. using non-overlapping partitions with only ghost elements reaching across the partition boundary.

These two conditions allow us to write an MPI-parallelised version of our structured-grid implementation that is perfectly equivalent to our serial implementation and scales nicely with the grid size, i.e., the communication cost is not a dominating factor.

Once the serial code has been partitioned on a CPU level, we will again add GPU parallelism to the mix by means of writing some OpenCL code. However, the communication requirements with that will be significantly higher than with the pure CPU version, as in this case values have to be repeatedly moved back and forth between the CPUs and GPUs, in addition to the inter-CPU communication.

The problem with moving data between CPUs and GPUs lies in the fact that this data transfer is going through the PCIe bus of the mainboard. Unfortunately, the PCIe bus has a rather low data throughput, low enough for the performance to take a big hit when moving a lot of data back and forth. There have been some attempts to circumvent this problem. E.g., CUDA allows inter-GPU communication without the need of going through the CPU *if* the two GPUs are located on the same mainboard,

i.e., if there is a physical connection between them apart from the PCIe connection. As an alternative to the CUDA-specific workaround, the only real alternative is to develop GPUs that can connect either directly or develop a connection along the lines of PCIe but with faster data throughput. Such "new" architectures have already been developed. For example, Intel developed *Intel MIC (Intel Many Integrated Core Architecture)* that is currently in use in its Intel Xeon Phi processors. One competitor of Intel in the world of GPUs is NVIDIA with its *Tesla* branded product lines. Either solution solves the data throughput for inter-CPU or inter-GPU communication with a custom solution, although NVIDIA restricts that usage solely to the CUDA standard.

Thus, when moving to heterogeneous systems, we can indeed hope to get some good speedup. It certainly allows us to solve much larger problems, as each core only manages a small partition of the full domain. Though it will involve a lot of work, it is well within the possibilities. However, due to simple technical limitations outside of our reach, the speedup achieved might be much lower than anticipated given the computing power involved.

Along with new architectures being developed, the need arises to re-write code for each of these architectures to allow it to run on them. This has sparked interest in the development of abstractions and tools that allow the use of a single source code to support multiple accelerations and parallelisation strategies. The clear advantage of these approaches is that no code would have to be re-written for any new architecture as long as the tools in use support them. A major drawback, however, is that the optimal algorithmic structure can vary significantly for different architectures leading to sub-optimal performance. Hence, for each application, the advantages of using an abstraction layer have to be carefully weighted against their disadvantages before choosing either to use or not to use such tools.

Besides targeting heterogeneous systems, there is currently work underway to do a better analysis of how to choose the parameters in use. For our experiments, we used a simple parameter study to obtain the optimal values, which required us to run our code repeatedly with many combinations of parameters. Finding a heuristic way to determine the optimal parameter choice, e.g., by means of a Fourier analysis, would, thus, be highly beneficial.

Another research topic is the relaxation scheme used in the multigrid V-cycle. Even though Braess-Sarazin works rather well and also can be parallelised quite nicely, it is not always a stable algorithm; there are known cases in which the algorithm simply

breaks down [22]. A related approach is the *Vanka* relaxation scheme [36]. Instead of solving one global saddle-point problem it solves a number of independent local saddle-point problems. Most importantly, Vanka is known to be stable, i.e., it is not seen to break down as easily as Braess-Sarazin. Replacing Braess-Sarazin by Vanka may yield better results in terms of accuracy and, potentially, also in terms of parallelisation. Each local saddle-point problem can be solved simultaneously, making it an ideal candidate for both CPU and GPU parallelisation.

We hope to extend our code in the future to allow it to solve variable coefficient problems. Even though most parts of our code already are flexible enough to handle this, the main part that still requires some work is the task of Galerkin coarsening. When coarsening the grids in our multigrid V-cycle, we currently simply do a re-discretisation of the system matrix on these coarser grids, see Section 3.8. To allow our code to solve variable coefficient problems, we need to find an efficient way to compute the triple matrix product required for the Galerkin coarsening. In particular, these triple matrix products have proven to be very memory intensive, which is a problem when computing on a GPU. Along with an extension of our code to allow variable coefficient problems, we also hope to extend it such that it is possible to solve nonlinear problems, e.g., the *Navier-Stokes* or *Incompressible Elasticity* equations.

Bibliography

[1] Sander Rhebergen, Garth N. Wells, Andrew J. Wathen, and Richard F. Katz. Three-field block preconditioners for models of coupled magma/mantle dynamics. *SIAM J. Sci. Comput.*, 37(5):A2270–A2294, 2015.

[2] Sander Rhebergen, Garth N. Wells, Richard F. Katz, and Andrew J. Wathen. Analysis of block preconditioners for models of coupled magma/mantle dynamics. *SIAM J. Sci. Comput.*, 36(4):A1960–A1977, 2014.

[3] T. Geenen, M. ur Rehman, S.P. MacLachlan, G. Segal, C. Vuik, A.P. van den Berg, and W. Spakman. Scalable robust solvers for unstructured FE modeling applications; solving the Stokes equation for models with large, localized viscosity contrasts. *Geochemistry, Geophysics, Geosystems*, 10(9), 2009.

[4] T. Geenen, M. ur Rehman, S.P. MacLachlan, G. Segal, C. Vuik, A.P. van den Berg, and W. Spakman. Scalable robust solvers for unstructured FE geodynamic modeling applications; solving the Stokes equation for models with large localized viscosity contrasts in 3D spherical domains. In J. C. F. Pereira and A. Sequeira, editors, *V European Conference on Computational Fluid Dynamics ECCOMAS CFD 2010*, 2010. Submitted.

[5] J. E. Dendy, Jr. Black box multigrid. *J. Comput. Phys.*, 48(3):366–386, 1982.

[6] J. E. Dendy, Jr. Black box multigrid for systems. *Appl. Math. Comput.*, 19(1-4):57–74, 1986. Second Copper Mountain conference on multigrid methods (Copper Mountain, Colo., 1985).

[7] S. P. MacLachlan, J. D. Moulton, and T. P. Chartier. Robust and adaptive multigrid methods: comparing structured and algebraic approaches. *Numer. Linear Algebra Appl.*, 19(2):389–413, 2012.

[8] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[9] J. W. Ruge and K. Stüben. Algebraic multigrid. In *Multigrid methods*, volume 3 of *Frontiers Appl. Math.*, pages 73–130. SIAM, Philadelphia, PA, 1987.

[10] A. Brandt, S. McCormick, and J. Ruge. Algebraic multigrid (AMG) for automatic multigrid solution with application to geodetic computations. In *Technical Report*. Institute for Computational Studies, Colorado State University, 1982.

[11] A. Brandt, S. McCormick, and J. Ruge. Algebraic multigrid (AMG) for sparse matrix equations. In *Sparsity and its applications (Loughborough, 1983)*, pages 257–284. Cambridge Univ. Press, Cambridge, 1985.

[12] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, Inc., San Diego, CA, 2001. With contributions by A. Brandt, P. Oswald and K. Stüben.

[13] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A multigrid tutorial*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2000.

[14] E. Cyr, E. Phibbs, M. Heroux, J. Brown, E. Coon, M. Hoemmen, R. Kirby, T. Kolev, J. Sutherland, and C. Trott. Algorithms and Abstractions for Assembly in PDE Codes: Workshop Report. 2015.

[15] Howard C. Elman, David J. Silvester, and Andrew J. Wathen. *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*. Numerical Mathematics and Scientific Computation. Oxford University Press, Oxford, second edition, 2014.

[16] Yousef Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2003.

[17] D. Braess and R. Sarazin. An efficient smoother for the Stokes problem. *Appl. Numer. Math.*, 23(1):3–19, 1997. Multilevel methods (Oberwolfach, 1995).

[18] J. E. Dendy, Jr. and J. D. Moulton. Black box multigrid with coarsening by a factor of three. *Numer. Linear Algebra Appl.*, 17(2-3):577–598, 2010.

[19] OpenMP ARB. http://openmp.org/openmp-faq.html, 2013.

[20] OpenMP ARB. http://openmp.org/mp-documents/openmp3.1-ccard.pdf, 2011.

[21] Thomas R. Benson. *Multigrid-based preconditioning for saddle-point problems*. ProQuest LLC, Ann Arbor, MI, 2015. Thesis (Ph.D.)–Tufts University.

[22] James H. Adler, Thomas R. Benson, Eric C. Cyr, Scott P. MacLachlan, and Raymond S. Tuminaro. Monolithic multigrid methods for two-dimensional resistive magnetohydrodynamics. *SIAM J. Sci. Comput.*, 38(1):B1–B24, 2016.

[23] Maxim Larin and Arnold Reusken. A comparative study of efficient iterative solvers for generalized Stokes equations. *Numer. Linear Algebra Appl.*, 15(1):13–34, 2008.

[24] V. John and L. Tobiska. A coupled multigrid method for nonconforming finite element discretizations of the 2D-Stokes equation. *Computing*, 64(4):307–321, 2000. International GAMM-Workshop on Multigrid Methods (Bonn, 1998).

[25] W. Zulehner. A class of smoothers for saddle point problems. *Computing*, 65(3):227–246, 2000.

[26] Nathan Bell, Steven Dalton, and Luke N. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM J. Sci. Comput.*, 34(4):C123–C152, 2012.

[27] Timothy A. Davis. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Software*, 30(2):196–199, 2004.

[28] Timothy A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Math. Software*, 30(2):167–195, 2004.

[29] Timothy A. Davis and Iain S. Duff. An unsymmetric-pattern multifrontal method for sparse *LU* factorization. *SIAM J. Matrix Anal. Appl.*, 18(1):140–158, 1997.

[30] Timothy A. Davis and Iain S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Trans. Math. Software*, 25(1):1–20, 1999.

[31] Blanca Ayuso de Dios, Franco Brezzi, L. Donatella Marini, Jinchao Xu, and Ludmil Zikatanov. A simple preconditioner for a discontinuous Galerkin method for the Stokes problem. *J. Sci. Comput.*, 58(3):517–547, 2014.

[32] Robert C. Kirby, Matthew Knepley, Anders Logg, and L. Ridgway Scott. Optimizing the evaluation of finite element matrices. *SIAM J. Sci. Comput.*, 27(3):741–758 (electronic), 2005.

[33] Anders Logg and Garth N. Wells. Dolfin: Automated finite element computing. *ACM Trans. Math. Softw.*, 37(2):20:1–20:28, April 2010.

[34] Anders Logg, Kent-Andre Mardal, and Garth N. Wells, editors. *Automated solution of differential equations by the finite element method*, volume 84 of *Lecture Notes in Computational Science and Engineering*. Springer, Heidelberg, 2012. The FEniCS book.

[35] Daniele Boffi, Franco Brezzi, and Michel Fortin. *Mixed finite element methods and applications*, volume 44 of *Springer Series in Computational Mathematics*. Springer, Heidelberg, 2013.

[36] S. P. Vanka. Block-implicit multigrid solution of Navier-Stokes equations in primitive variables. *J. Comput. Phys.*, 65(1):138–158, 1986.