A Parallel Geometric Multigrid Method for Adaptive Finite Elements

Thomas Conrad Clevenger
Clemson University, clevenger.conrad@gmail.com

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations

Recommended Citation
https://tigerprints.clemson.edu/all_dissertations/2523

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.
A Parallel Geometric Multigrid Method for Adaptive Finite Elements

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Mathematical Sciences

by
Thomas Conrad Clevenger
December 2019

Accepted by:
Dr. Timo Heister, Committee Chair
Dr. Qingshan Chen
Dr. Leo Rebholz
Dr. Fei Xue
Abstract

Applications in a variety of scientific disciplines use systems of Partial Differential Equations (PDEs) to model physical phenomena. Numerical solutions to these models are often found using the Finite Element Method (FEM), where the problem is discretized and the solution of a large linear system is required, containing millions or even billions of unknowns. Often times, the domain of these solves will contain localized features that require very high resolution of the underlying finite element mesh to accurately solve, while a mesh with uniform resolution would require far too much computational time and memory overhead to be feasible on a modern machine. Therefore, techniques like adaptive mesh refinement, where one increases the resolution of the mesh only where it is necessary, must be used. Even with adaptive mesh refinement, these systems can still be on the order of much more than a million unknowns (large mantle convection applications like the ones in [90] show simulations on over 600 billion unknowns), and attempting to solve on a single processing unit is infeasible due to limited computational time and memory required. For this reason, any application code aimed at solving large problems must be built using a parallel framework, allowing the concurrent use of multiple processing units to solve a single problem, and the code must exhibit efficient scaling to large amounts of processing units.

Multigrid methods are currently the only known optimal solvers for linear systems arising from discretizations of elliptic boundary valued problems. These methods can be represented as an iterative scheme with contraction number less than one, independent of the resolution of the discretization [24, 54, 25, 103], with optimal complexity in the number of unknowns in the system [29]. Geometric multigrid (GMG) methods, where the hierarchy of spaces are defined by linear systems of finite element discretizations on meshes of decreasing resolution, have been shown to be robust for many different problem formulations, giving mesh independent convergence for highly adaptive meshes [26, 61, 83, 18], but these methods require specific implementations for each type of
equation, boundary condition, mesh, etc., required by the specific application. The implementation in a massively parallel environment is not obvious, and research into this topic is far from exhaustive.

We present an implementation of a massively parallel, adaptive geometric multigrid (GMG) method used in the open-source finite element library deal.II [5], and perform extensive tests showing scaling of the v-cycle application on systems with up to 137 billion unknowns run on up to 65,536 processors, and demonstrating low communication overhead of the algorithms proposed. We then show the flexibility of the GMG by applying the method to four different PDE systems: the Poisson equation, linear elasticity, advection-diffusion, and the Stokes equations. For the Stokes equations, we implement a fully matrix-free, adaptive, GMG-based solver in the mantle convection code ASPECT [13], and give a comparison to the current matrix-based method used. We show improvements in robustness, parallel scaling, and memory consumption for simulations with up to 27 billion unknowns and 114,688 processors. Finally, we test the performance of IDR(s) methods compared to the FGMRES method currently used in ASPECT, showing the effects of the flexible preconditioning used for the Stokes solves in ASPECT, and the demonstrating the possible reduction in memory consumption for IDR(s) and the potential for solving large scale problems.

Parts of the work in this thesis has been submitted to peer reviewed journals in the form of two publications ([36] and [34]), and the implementations discussed have been integrated into two open-source codes, deal.II and ASPECT. From the contributions to deal.II, including a full length tutorial program, Step-63 [35], the author is listed as a contributing author to the newest deal.II release (see [5]). The implementation into ASPECT is based on work from the author and Timo Heister. The goal for the work here is to enable the community of geoscientists using ASPECT to solve larger problems than currently possible. Over the course of this thesis, the author was partially funded by the NSF Award OAC-1835452 and by the Computational Infrastructure in Geodynamics initiative (CIG), through the NSF under Award EAR-0949446 and EAR-1550901 and The University of California – Davis.
Table of Contents

Title Page ................................................................. i
Abstract ...................................................................... ii
List of Tables ............................................................. vi
List of Figures ............................................................. viii

1 Introduction ............................................................. 1

2 Foundations ............................................................. 7
  2.1 PDE Systems ........................................................... 7
  2.2 The Finite Element Method ......................................... 10
  2.3 Finite Element Theory for the Stokes Equations ............... 22
  2.4 Krylov Subspace Methods .......................................... 26
  2.5 Multigrid Methods ................................................... 33
  2.6 Matrix-free Methods ................................................. 42
  2.7 Parallel Computing .................................................. 43
  2.8 Software ............................................................... 46

3 Parallel, Adaptive Geometric Multigrid ......................... 53
  3.1 Review of Current Literature ....................................... 54
  3.2 Parallelization of Geometric Multigrid ........................... 55
  3.3 Experimental Study of Mesh Partitioning ....................... 60
  3.4 Numerical Results .................................................. 65

4 Applications to the Stokes Equations .............................. 81
  4.1 Review of Current Literature ....................................... 82
  4.2 Stokes System Used in ASPECT ................................. 85
  4.3 Linear Solver ........................................................ 87
  4.4 Results ............................................................... 92

5 Performance Analysis of Krylov Methods for Stokes Computations ........................................... 105
  5.1 Current Solvers used for Stokes in the Literature .............. 106
  5.2 Cheap vs. Expensive Preconditioning in ASPECT ............. 106
  5.3 Timing of Individual Components ................................. 108
  5.4 Solver Performance ................................................ 111
  5.5 Flexibility ............................................................ 112
  5.6 Recommendation for ASPECT ..................................... 114

6 Conclusions ............................................................. 117
List of Tables

2.1 Number of CG iterations for solving Poisson with $Q_1$ element. ......................... 31
3.1 CG iterations needed to reduce the residual by $1e8$. ........................................... 65
3.2 Number of outer conjugate gradient iterations and runtimes for solving the elasticity example on a 28-core setup. ................................................................. 72
3.3 GMRES iterations to reduce residual by $10^8$. Various renumbering strategies for additive smoothers with a $Q_1$ element. Value listed next the the method name is the number of pre- and post-smoothing steps per level per iteration. ............................................................ 77
3.4 GMRES iterations to reduce residual by $10^8$. Various renumbering strategies for additive smoothers with a $Q_3$ element. Value listed next the the method name is the number of pre- or post-smoothing steps per level per iteration. ............................................................... 78
3.5 GMRES iterations to reduce residual by $10^8$. $Q_1$ element. Value listed next the the method name is the number of pre- or post-smoothing steps per level per iteration. ............................................................ 78
3.6 GMRES iterations to reduce residual by $10^8$. $Q_3$ element. Value listed next the the method name is the number of pre- or post-smoothing steps per level per iteration. ............................................................ 79
3.7 GMRES solve time (in seconds) to reduce residual by $10^8$. $Q_3$ element. ............... 79
4.1 Number of outer FGMRES iterations to reduce residual by $1e6$ in the inclusion benchmark. The FGMRES restart length is set to 100. ................................................................. 93
4.2 FGMRES iterations required to reduce the residual by $1e6$ for the $n$-sinker benchmark with increasing $n$ and DR($\mu$). Run on a 3D mesh with 860K degrees of freedom ($[Q_2]^{\text{dim}} \times Q_1$ element), distributed over 32 processors. ................................................................. 96
4.3 FGMRES iterations required to reduce the residual by $1e6$ for the GMG preconditioner on problems depicted in Figure 4.5a (Globally refined mesh). ................................................. 96
4.4 Timing comparison between AMG and GMG for the $n$-sinker benchmark for an adaptively refined, 3D mesh, with 18.5M DoFs ($[Q_3]^{\text{dim}} \times Q_1$ element) on one node (48 cores). Timings are from the Stampede2 machine. ................................................................. 98
4.5 FGMRES iterations required to reduce the residual by $1e6$ on problems depicted in Figure 4.6a. ................................................................. 99
4.6 Memory consumption required for major components of AMG and GMG for globally refined, 3D mesh, with 113K DoFs ($[Q_2]^{\text{dim}} \times Q_1$ element) on 1 cores. ................................................. 101
4.7 FGMRES iterations required to reduce the residual by $1e6$ for Sinker benchmark with zero Neumann boundary conditions on top and no-normal flux boundary conditions on sides and bottom. The first mesh is a globally refined cube with 4 levels of refinement, then each subsequent mesh is obtain by refinement with a Kelly estimator, roughly doubling the number of cells/DoFs. The FGMRES restart length was set to 150 to ensure neither method needed a restart to give a fair comparison of the methods. ................................................................. 102
4.8 FGMRES iterations required to reduce the residual by $1e8$ for the annulus benchmark with both no-normal flux and prescribed boundaries. Run using $[Q_2]^2 \times Q_1$ elements on a single core, and a FGMRES restart length of 100. Each problem size obtained from a sequence of global refinements. ................................................................. 104
4.9 L2-norm of the error for velocity and pressure, and the rate of decrease between each global refinement step, given by $e_h/e_{2h}$, for the annulus benchmark with no-normal flux boundaries. Discretizations with a $[Q_2]^{\text{dim}} \times Q_1$ element should yield error rates of $2^3$ and $2^2$ for velocity and pressure, respectively, which we observe here.

5.1 Memory consumption required for major components of AMG- and GMG-based methods for globally refined, 3D mesh, with 113K DoFs ($[Q_2]^{\text{dim}} \times Q_1$ element) on 1 core. Estimates for FGMRES(50) (105 total vectors) and GMRES(50) (55 total vectors) are given.

5.2 Number of scalar products $(v, w)$, matrix-vector products $Av$, preconditioner application $B^{-1}v$, and required temporary vectors in $i$ iterations for each Krylov method considered. We are assuming no restart for GMRES.

5.3 Iteration counts to reduce the residual by 1e6 for $n$-sinker benchmark with 8 sinkers and changing $\text{DR}(\mu)$. Time (in seconds) to solution for solves in Table 5.3. The fastest runtime in each row is highlighted red. Timings from runs on Palmetto using 40 cores.

5.5 Outer iteration counts to reduce the residual by 1e6 for $n$-sinker benchmark with 8 sinkers and $\text{DR}(\mu)=1e4$. Here we vary the relative tolerance used for the CG solve on the $A$-block in the expensive preconditioner.

5.6 Time (in seconds) to solution for solves in Table 5.5. The fastest runtime in each row is highlighted red. Timings from runs on Palmetto using 40 cores.

5.7 Memory consumption required for major components of AMG- and GMG-based methods for globally refined, 3D mesh, with 113K DoFs ($[Q_2]^{\text{dim}} \times Q_1$ element) on 1 core. FGMRES(50) (105 total vectors) estimates are given for both AMG and GMG, and both GMRES(50) (55 total vectors) and IDR(2) (16 total vectors) estimates are given for GMG.
List of Figures

1.1 Multigrid v-cyle. Each level $\ell$ defines a space with different resolution, where $\ell = 2$ is the finest space and $\ell = 0$ is the coarsest space. Blue circles represent smoothing operations, black lines represent transfers between the spaces, and the red square represents coarse grid correction. .......................................................... 2

1.2 Adaptive refinement of a particular time step for a mantle convection solve over a quarter shell domain using the ASPECT code [1]. .......................................................... 3

1.3 Strong and weak scaling for a matrix-free Stokes solve using the GMG-based preconditioner (4.7) for the $n$-sinker benchmark defined in Section 4.4.2, with global refinement on a 3D box mesh with $[Q_2]^3 \times Q_1$ finite elements. Timings are from the new Frontera machine in Austin, T.X. (number 5 in the Top500). In the legend, “DoFs” (degrees of freedom) are the number of unknowns in the finite element system. The plot shows strong scaling between 15-30K unknowns per core. ...................... 4

2.1 Example of conforming and non-conforming meshes with identical mesh size. Red circles indicate hanging nodes. .......................................................... 16

2.2 Global refinement. .......................................................... 16

2.3 Adaptive refinement. .......................................................... 17

2.4 Adaptive refinement of a particular time step for a mantle convection solve over a quarter shell domain using the ASPECT code [1]. .......................................................... 17

2.5 Finite element spaces $Q_k$, $k = 1, 2, 3$ with associated degrees of freedom. .......................................................... 18

2.6 Mapping $F_K$ .......................................................... 19

2.7 Multigrid v-cyle. Blue circles represent smoothing, black lines represent transfers, red square represents coarse solve. .......................................................... 34

2.8 Simple GMG mesh hierarchy given by two global refinements of the unit square. .......................................................... 35

2.9 Splitting of the mesh $T_\ell$ and the space $V_\ell$ into subspaces for local smoothing. Superscript $S$ refers to the cells and functions strictly on level $\ell$, used for smoothing. Superscript $L$ is the support of functions actually defined on lower levels and $I$ are basis functions for node functionals on the interface which have support in both subdomains .......................................................... 39

2.10 Theoretical scaling according to Amdahl’s Law where $p$ is the amount of a code written in parallel. .......................................................... 45

2.11 Representation of a deal.II mesh. The cell label $\ell.T$ is used for cell $T$ on level $\ell$. .......................................................... 47

2.12 Example of a space-filling curve as used by p4est. .......................................................... 47

2.13 Partition among 3 processors of an adaptive mesh. Processor 0 (left) is red, processor 1 (center) is blue, and processor 2 (right) is green. One layer of ghost cells included. Cells without coloring are called artificial cells and not representative of the actual mesh discretization, only the minimal information needed by each processor to reconstruct its active cells. .......................................................... 48
3.1 Example partitioning for an adaptively refined mesh from a single coarse cell with 7 active cells distributed between three processors indicated by color and label (red #0, green #1, blue #2). Ownership of the level cells is determined by the “first-child rule” (round shapes in the middle). Left: The partitioning of the active mesh with the space-filling Z-curve. Middle: Tree representation of the mesh and its partitioning. Right: Partitioning of the three multigrid levels. .................................................. 59

3.2 Visualization of the different mesh refinement sequences .................................................. 62

3.3 Partitioning efficiency for various meshes. Dashed lines indicate less than 1,000 active cells per processor. Top: global, circle, quadrant refinement. Bottom: annulus refinement in two (left) and three (right) dimensions. .......................................................... 63

3.4 Communication ratio (communicated number of children over total number) in level transfer ............................................................................................................................ 66

3.5 Viscosity (left) and solution (right) of the inclusion test .......................................................... 66

3.6 Strong scaling for timing of a matrix-free v-cycle in 2D and 3D for small (left) and large (right) problem size of the annulus refinement. Timings done on the SuperMUC. 69

3.7 Weak scaling for timing of a matrix-free v-cycle in 2D and 3D for the annulus refinement. Timings done on the SuperMUC. .................................................. 70

3.8 Displacement magnitude (left) and distribution of domain over 28 processor cores (right) of three-dimensional elasticity example. .......................................................... 71

3.9 Solution with (left) and without (right) SUPG. .............................................................. 76

3.10 Downstream (left), random (middle), and upstream (right) cell reordering. ....................... 77

4.1 Sparsity pattern for the matrix (4.6) for full coupling of velocity components (left) and partial coupling (right). 3D Stokes with 402 degrees of freedom (375 velocity, 27 pressure), ordered component-wise. A red pixel represents a non-zero matrix entry. We see that for partial coupling we have approximately 3x fewer entries as with full coupling. .................................................. 89

4.2 Reference solution for the inclusion benchmark. Velocity along with viscosity is shown on the left, and pressure on the right. .................................................. 93

4.3 Strong scaling for inclusion benchmark, global refinement, 2D mesh with \([Q_2]^{dim} \times Q_1\) element. Timing are from the Palmetto cluster. .................................................. 94

4.4 Sinker benchmark solution for \(n = 8, DR(\mu) = 1e4\) .................................................. 96

4.5 Strong and weak scaling for GMG for the \(n\)-sinker benchmark, global refinement, 3D mesh with \([Q_2]^{dim} \times Q_1\) element. Timings are from the Frontera machine. .................................................. 97

4.6 Weak scaling comparison for the \(n\)-sinker benchmark, adaptive refinement, 3D mesh, \([Q_2]^{dim} \times Q_1\) element. The red dashed line represents the ideal weak scaling we can expect based on the imbalance of the parallel partition, explain in Section 3.2.2. Timings are from the Stampede2 machine. .................................................. 100

4.7 Annulus benchmark reference solution for \(k = 4\). .................................................. 103

5.1 Scaling for scalar product, matrix-vector product, and preconditioner application for 3D Stokes discretization with 53M DoFs. Timings from runs on Frontera. ........................................ 109

5.2 Model for solve times for GMRES(50), IDR(1), and IDR(2) based on component timings in Figure 5.1 from Frontera for different iterations counts. ........................................ 110

5.3 Model for solve times for GMRES(50), IDR(1), and IDR(2) based on component timings in Figure 5.1 from Frontera for 100 iterations. .................................................. 111
Chapter 1

Introduction

Applications in a variety of scientific disciplines use systems of Partial Differential Equations (PDEs) to model various physical phenomenon. Examples could include the creeping flow of the earth’s mantle, material deformations in structural mechanics, or heat transfer through an object, to name a few. Numerical solutions to these models are often found using the Finite Element Method (FEM), a process of creating a system of equations described on a discretized domain whose solutions approximate the solution to the system of PDEs. These linear systems can often be very large, and creating efficient solvers which have a time to solution that is proportional to the problem size can be very difficult. Direct solvers, using Gaussian Elimination, clearly do not fit these requirements as the time to solution grows cubically with the number of unknowns, and therefore we look for iterative solvers for these systems.

Multigrid methods are currently the only such iterative methods that satisfy this criterion. These methods are optimal solvers for linear systems arising from discretizations of elliptic boundary valued problems as they can be represented as an iterative scheme with contraction number less than one, independent of the resolution of the discretization [24, 54, 25, 103], with optimal complexity in the number of unknowns in the system [29]. They are centered around the idea of a \textit{v-cycle}: smoothing out the high frequency error of the residual on a fine mesh using some cheap approximate solver, then adding a correction of the remaining error from a coarser space with a more powerful method, smoothing out the accumulated error that arises from transferring this correction back to the finer space. Figure 1.1 gives a representation of a v-cycle with 3 levels, where the coarse grid correction from the finest space is itself a multigrid v-cycle. There are different types of multigrid methods
which are defined by the definition of the finer and coarser spaces. In this thesis we will discuss two main types: Algebraic multigrid (AMG), where the spaces are defined purely algebraically as a hierarchy of linear systems derived from the finite element system being solved, and Geometric multigrid (GMG), where the spaces are defined by linear systems of finite element discretizations on fine and coarse meshes. AMG is powerful as it can be seen as a black-box method where the user only needs to supply the system that they would like to solve and the particular AMG implementation takes care of the rest, however, as will be seen in Chapter 4, AMG often deteriorate with more complicated problem setups. On the other hand, GMG, using geometric information in the setup of the level space, can be robust for many different problem formulations, but requires specific implementations for each type of equation, boundary condition, meshes, etc., required by the specific application.

Considering again the problems which arise from the different scientific applications, often times the domain will contain localized features that require very low resolution of the underlying finite element mesh to accurately solve. If we take the convection in the earth’s mantle as an example, very high resolution could be needed to resolve areas where large changes in the viscosity of the mantle is observed, or at the interface between fast and slow moving materials [79]. Given a large domain, as is common for many of these applications, a mesh with uniform resolution would require far too much computational time and memory overhead to be feasible on a modern machine [31]. To combat this, many applications rely on adaptive mesh refinement, i.e., increasing the resolution of the mesh only where it is necessary, thereby creating a mesh with different resolutions for different
Figure 1.2: Adaptive refinement of a particular time step for a mantle convection solve over a quarter shell domain using the ASPECT code [1].

parts of the domain. For the mantle convection example, one could choose to increase the mesh resolution around interesting features defined by the application (e.g., large changes in viscosities or densities of the fluid), or in areas where a previous solution experienced large jumps in its gradient. Figure 1.2 depicts a mesh where the refinement is base on a gradient-based error estimator from a previous solve. Creating code that can work for adaptive mesh refinement, and solvers that are robust at higher levels of adaptivity, is often very difficult. For multigrid methods, AMG can still be used as a black box solver (easy implementation), however, as will be seen in Chapter 4, there is a significant loss in approximation quality when running on highly adaptive meshes. On the other hand, GMG can be quite difficult to implement, but should give mesh independent convergence, even with higher levels of adaptivity [26, 61, 83, 18].

Even with adaptive mesh refinement, these systems can still be on the order of much more than a million unknowns (large mantle convection applications like the ones in [90] show simulations on over 600 billion unknowns), and attempting to solve on a single processing unit is infeasible due to limited computational time and memory required. For this reason, any application code aimed at solving large problems must be built using a parallel framework, allowing the concurrent use of multiple processing units to solve a single problem. Further, with computer architectures becoming large and larger, it is imperative that these parallel codes exhibit efficient scaling to large amounts of processing units, that is, the runtime of a code should be proportional to the size of the problem and inversely proportional to the number of processing units used for the computation.
Figure 1.3: Strong and weak scaling for a matrix-free Stokes solve using the GMG-based preconditioner (4.7) for the $n$-sinker benchmark defined in Section 4.4.2, with global refinement on a 3D box mesh with $[Q_2]^3 \times Q_1$ finite elements. Timings are from the new Frontera machine in Austin, T.X. (number 5 in the Top500). In the legend, “DoFs” (degrees of freedom) are the number of unknowns in the finite element system. The plot shows strong scaling between 15-30K unknowns per core.

Figure 1.3 gives an example, shown in Chapter 4, of both strong scaling (runtime of a problem with constant size is inversely proportional to the number of processing units) and weak scaling (runtime of problems whose size increases proportional to the number of processing units should be constant) of a multigrid method used for solving a Stokes systems with up to 27 billion unknowns, and serves as a preview for the results achieved in this thesis.

For these large scale computations, then, we want a combination of all of these methods, namely, we seek efficient parallel adaptive geometric multigrid methods for a variety of finite element applications. The implementation of such a method can be difficult and are far from exhaustively explored in the literature. We present such a method and demonstrate its flexibility by showing applications to four different PDE systems: the Poisson equation, linear elasticity, advection-diffusion, and the Stokes equations. We show scaling results on some of the worlds largest supercomputers, solving systems with up to 137 billion unknowns, distributed on up to 65,536 processors. Parts of the work in this thesis has been submitted to peer reviewed journals in the form of two publications ([36] and [34]), and the implementations discussed have been integrated into two open-source codes, deal.II [5] (parallel adaptive finite element library) and ASPECT [13] (mantle convection application code built on top of deal.II). The implementations in deal.II are based on work from many others as well, and careful consideration will be given to highlighting the specific contributions by the
author throughout this thesis. From these contributions, including a full length tutorial program, Step-63 [35], the author is listed as a contributing author to the newest deal.II release (see [5]). The implementation into ASPECT is based on work from the author and Timo Heister. The goal for the work here is to enable the community of geoscientists using ASPECT to solve larger problems than currently possible. Over the course of this thesis, the author was partially funded by the NSF Award OAC-1835452 and by the Computational Infrastructure in Geodynamics initiative (CIG), through the NSF under Award EAR-0949446 and EAR-1550901 and The University of California – Davis.

We will use the following structure for the remainder of the thesis:

In Chapter 2, we will give a description of all necessary mathematical foundations for the methods discussed in this thesis, as well as a description of the open-source software/application codes used to obtain the results in subsequent chapters. The discussion will also contain a detailed list of contributions by the author to each open-source code.

In Chapter 3, we give a detailed description of the parallel implementation of the adaptive multigrid method discussed in Section 2.5. We introduce the “first-child rule” for distributing the level hierarchy among processors in a parallel computation and run extensive experiments to test the scalability and communication overhead of a v-cycle for various refinement schemes run with increasing processor counts. Then we demonstrate the robustness and scalability of the developed multigrid v-cycle as a preconditioner within a Krylov subspace solve for the Poisson equations and linear elasticity. Finally, we close the chapter with an application of the multigrid method using additive and multiplicative Schwarz-type, cell-based smoothers for the advection-diffusion equation, demonstrating convergence independent of mesh size and polynomial degree, giving a comparison with point smoothers Jacobi and SOR. The results from this section are split between in a publication [36] by the author with with co-authors Timo Heister, Guido Kanschat, and Martin Kronbichler and a deal.II tutorial program [35] written by the author and Timo Heister and extensively reviewed by Wolfgang Bangerth.

In Chapter 4, we give a detailed description of the current matrix-based solver used for the Stokes equations inside ASPECT, and present a new solver based on a matrix-free multigrid method, including a description of all necessary new features required for such an implementation. We demonstrate the robustness and scalability of the new solver for adaptive and massively parallel computations, and give detailed comparisons with the current matrix-based solver for several bench-
marking codes used in ASPECT. The results from this Chapter come from a full scale implementation by the author and Timo Heister of the matrix-free method inside the ASPECT code.

In Chapter 5, we give a test of the convergence behavior, parallel scaling, and memory overhead of the IDR(1) and IDR(2) Krylov methods for the Stokes equations, and compare with the restarted FGMRES Krylov method used inside ASPECT. All results are using the matrix-free multigrid solver. We the make a recommendation of which solvers should used be inside ASPECT, specifically discussing the benefits of the IDR(2) for large scale computations.

In Chapter 6, we summarize the results of each chapter and discuss the overall impact of the work from this thesis.
Chapter 2

Foundations

This chapter is focused on giving both the mathematical and programming foundations necessary for the work in subsequent chapters, as well as listing the specific contributions of the author to both the deal.II and ASPECT.

2.1 PDE Systems

We will consider results for the following systems of Partial Differential Equations (PDEs): the Poisson equation (sometimes referred to as diffusion), linear elasticity, advection-diffusion, and the Stokes equations. Here, they are listed in the order in which the results will appear, however, we will spend much more time in this chapter discussing the Poisson equation and the Stokes equations as they are the focus of the majority of the later chapters.

Each PDE will be defined on a spacial domain $\Omega \subset \mathbb{R}^d$ which is assumed to be bounded and connected. We denote $\partial \Omega$ to be the boundary of this domain and assume it is Lipschitz continuous. The solutions ($u$ typically signifying velocities or displacements and $p$ signifying pressure) can be either be scalar-valued or vector-valued (the latter denoted with bold font, i.e., $\mathbf{u}$).

2.1.1 The Poisson Equation

Consider the Poisson problem given by,
Find $u : \Omega \to \mathbb{R}$ s.t.

\[
-\Delta u = f \text{ in } \Omega \\
u = 0 \text{ on } \partial \Omega.
\] (2.1)

This is an elliptic PDE [40] and it is the basis of many well-known mathematical models including Newtonian gravitation, electromagnetism, and heat transfer among many others [89]. For each PDE discussed below, Poisson-like operators are present, and therefore the solution of systems derived from the Poisson equations will be very important to the work of this thesis.

While (2.1) gives a simple form of the equation, we will also consider vector-valued solutions $u$, as well as various different boundary conditions including the non-zero Dirichlet condition $u = g_D$, the Neumann boundary conditions $\nabla u \cdot n = g_N$, and normal-flux (or free-slip) boundary conditions $u \cdot n = g_F$. Here, $n$ denotes the outer-facing normal vector on $\partial \Omega$.

### 2.1.2 Linear Elasticity

Next, we will consider linear elasticity given by,

Find $u : \Omega \to \mathbb{R}^d$ s.t.

\[
-\nabla \cdot \sigma(u) = f \text{ in } \Omega \\
\sigma(u) = \lambda \text{tr}(\varepsilon(u))I + 2\mu\varepsilon(u) \\
\varepsilon(u) = \frac{1}{2} (\nabla u + \nabla u^T) \\
u = 0 \text{ on } \partial \Omega.
\] (2.2)

where $\sigma$ is referred to as the stress tensor, $\varepsilon(u)$ is referred to as the strain-rate tensor or the symmetric gradient of $u$, and $\lambda, \mu$ are referred to as the Lamé coefficients. The solution $u$ represents displacement of a deformable medium $\Omega$ after an external force $f$ is applied, and the value $\lambda + 2/3 \cdot \mu$ describes the compressibility of the material represented by the domain (large values imply nearly incompressible) [39]. In Section 3.4.2, we consider a single example of linear elasticity where $\lambda = \mu = 1$. 

8
2.1.3 Advection-Diffusion

Next, consider the advection-diffusion equation given by,

Find \( u : \Omega \rightarrow \mathbb{R} \) s.t.

\[
-\varepsilon \Delta u + \beta \cdot \nabla u = f \quad \text{in } \Omega \\
\varepsilon > 0 \text{ is the diffusion constant, } \beta : \Omega \rightarrow \mathbb{R}^d \text{ is the given advection direction, and } f : \Omega \rightarrow \mathbb{R} \]

is a source term. The solution \( u \) could be used to model the temperature of a fluid moving through a heated pipe \([39]\), or a pollutant being diffused inside a river \([89]\), to give two examples.

Notice that,

1. if \( \beta = 0 \), this is the Poisson equation \((2.1)\), and

2. if \( \varepsilon = 0 \), this is a stationary advection equation. In this case, one can use the method of characteristics to solve for the solution (see \([40, 38]\) for a description of the method of characteristics). It should be noted that in this case, one must only supply inflow boundary conditions, that is, boundary conditions on

\[
\partial \Omega_{-} = \{ x \in \partial \Omega : \beta \cdot n < 0 \},
\]

due to the fact that the solution \( u \) at any part of the domain can be calculated by tracing the characteristic directions back to a point on the inflow boundary. It is therefore not a boundary valued problem, but an initial valued problem \([38]\).

We will limit our discussion to the case when \( \varepsilon > 0 \) and \( \beta \neq 0 \). For this case, consider the following dimensionless value called the Peclet number, given by

\[
\mathcal{P} := \frac{||\beta||L}{\varepsilon},
\]

where \( L \) is the length scale of the domain. This value characterizes the kind of equation we are considering: If \( \mathcal{P} > 1 \), we say the problem is advection-dominated, else if \( \mathcal{P} < 1 \) we will say the problem is diffusion-dominated. For most practical problems we have advection-dominated transport \([38]\), and for the purposes of this thesis, this is the more interesting case, as diffusion dominated
problems become numerically similar to the Poisson equation.

2.1.4 The Stokes Equations

Lastly, consider the Stokes equation as formulated in [39],

Find $u : \Omega \rightarrow \mathbb{R}^d$ and $p : \Omega \rightarrow \mathbb{R}$ s.t.

$$
-\mu(x)\Delta u + \nabla p = f \quad \text{in } \Omega \\
\nabla \cdot u = g \quad \text{in } \Omega \\
u = 0 \quad \text{on } \partial\Omega
$$

(2.4)

where $\mu(x) > 0$ represents viscosity of a fluid, $f : \Omega \rightarrow \mathbb{R}^d$ is a forcing term, and $g : \Omega \rightarrow \mathbb{R}$ describes the compressibility of the fluid. Solution variables $u : \Omega \rightarrow \mathbb{R}^d$ and $p : \Omega \rightarrow \mathbb{R}$ represent the velocity and pressure of a fluid respectively. When $g = 0$, we say that the fluid is incompressible. These equation are useful for modeling creeping flows which occur in at least one of the following: small geometries, small velocities, or high viscosities [77]. Examples include blood moving through the body (small geometries) [38], or mantel convection inside the earth (small velocities and large viscosities), the latter being the focus of the ASPECT code discussed in Section 2.8.2. Solvers for the Stokes equations will be the focus of Chapter 4.

2.2 The Finite Element Method

The PDE systems listed in Section 2.1 will be solved using the Finite Element Method (FEM). The idea underlying the Finite Element Method is to create a variational formulation of a PDE system called a weak formulation, discretize the domain of the problem by describing the solution as a linear combination of a basis of piecewise polynomials, and then solving the resulting linear system. A more detailed introduction can be found in [39, 97, 23, 64] among others.

Here, we will define all the necessary mathematical notation and theory for an application of the Finite Element Method. For simplicity, consider the Poisson equation with zero Dirichlet boundary conditions on a bounded domain $\Omega \subset \mathbb{R}^d$, $d = 2$ or $3$ with Lipschitz continuous boundary $\partial\Omega$ and sufficiently smooth data $f$. The strong form of the Poisson problem is given by (2.1).
2.2.1 Weak formulation

Choosing test functions \( v \in V \), multiplying from the left, and integrating over \( \Omega \) gives

\[
- (v, \Delta u) = (v, f)
\]

where

\[
(f, g) := (f, g)_{L^2(\Omega)} = \int_{\Omega} f \cdot g
\]

is the \( L^2(\Omega) \) inner product. Then, using Green’s Theorem as derived in [39], we have

\[
(\nabla v, \nabla u) - (v, \nabla u \cdot n)_{\partial \Omega} = (v, f)
\]

(2.5)

where \( n \) is the outward unit normal on \( \partial \Omega \) and \( \nabla u \cdot n \) is the derivative in the outward normal direction. Here, we will choose the function space \( V \) to have zero boundary conditions so that this integral vanishes from equation (2.5), and we are left with

\[
(\nabla v, \nabla u) = (v, f).
\]

(2.6)

Since \( u \) and \( v \) require the same regularity for the integrals to be well-defined, we will choose them from the same function space \( V \), whose functions are zero on the boundary. This gives the following problem referred to as the weak formulation:

Find \( u \in V \) such that

\[
a(u, v) = f(v) \quad \forall v \in V
\]

(2.7)

where

\[
a(u, v) = (\nabla v, \nabla u), \quad f(v) = (v, f).
\]
2.2.1.1 Function Spaces

Next, we will define all function spaces, inner products, and norms used for the rest of the thesis. It is natural to start with the space of all square integrable functions

\[ L^2(\Omega) := \left\{ u : \Omega \to \mathbb{R} \mid \int_{\Omega} |u|^2 \, dx < \infty \right\} \]

as is typical in most of the literature [64, 39, 29, 77]. An example of the importance of this space in applications for fluid flows (as discussed in [77]) is, given a solution with density \( \rho_0 \) (constant) and velocity \( u \), then

\[
\text{kinetic energy} = \frac{1}{2} \rho_0 \int_{\Omega} |u|^2 \, dx
\]

and thus, \( L^2(\Omega) \) represents all functions with finite kinetic energy. We define the norm for this space by

\[
\|u\|_0^2 := \int_{\Omega} |u|^2 \, dx
\]

and, given that this is a Hilbert space equipped with an inner product [64], we will define for \( u, v \in L^2(\Omega) \),

\[
(u, v)_\Omega := \int_{\Omega} u \cdot v \, dx
\]

Since the weak formulation (2.7) contains the term \((\nabla u, \nabla v)_{L^2(\Omega)}\), we will need to define spaces with more regularity for our solution variables, and so we will consider the space

\[ H^1(\Omega) := \left\{ u : \Omega \to \mathbb{R} \mid u \in L^2(\Omega) \text{ and } \nabla u \in \left[ L^2(\Omega) \right]^d \right\} \]

and the corresponding space \( H^1_0(\Omega) \subset H^1(\Omega) \),

\[ H^1_0(\Omega) := \left\{ u : \Omega \to \mathbb{R} \mid u \in H^1(\Omega) \text{ and } u|_{\partial\Omega} = 0 \right\} \]

where \( |_{\partial\Omega} \) is the operator defining the restriction to the boundary. These spaces are also Hilbert spaces equipped with norm

\[
\|u\|_1^2 := \|u\|_0^2 + \|\nabla u\|_0^2,
\]
semi-norm

\[ |u|_1 := \|\nabla u\|_0, \]

and inner product

\[ (u, v)_1 := (u, v)_\Omega + (\nabla u, \nabla v)_\Omega. \]

The Poincaré-Friedrich’s inequality [23] relates the \( H^1 \) and \( L^2 \) norm

\[ \|u\|_0 \leq C |u|_1 \leq C \|u\|_1. \] (2.8)

The dual space to \( H^1_0(\Omega) \), denoted \( H^{-1}(\Omega) \), can be of interest when dealing with irregular data, as it is less restrictive than \( L^2(\Omega) \subset H^{-1}(\Omega) \) [77]. The space \( H^{-1}(\Omega) \) is defined as the closure of \( L^2(\Omega) \) with respect to the norm

\[ \|f\|_{-1} := \sup_{v \in H^1_0(\Omega)} \frac{(v, f)}{|v|_1}. \]

This dual space is used and discussed in greater detail in [39, 77].

Finally, the concept of the space \( H^1 \) can be extended to

\[ H^k(\Omega) := \left\{ u : \Omega \to \mathbb{R} \mid u \in L^2(\Omega) \text{ and } D^\alpha u \in L^2(\Omega) \forall |\alpha| \leq k \right\} \]

with norm

\[ \|u\|_k^2 := \sum_{|\alpha| \leq k} \|D^\alpha u\|_0^2, \]

where \( \alpha \) is a multi-index and \( D^\alpha \) represents the partial derivatives with multi-index \( \alpha \). These spaces are examples of Sobolev spaces, and are foundational in the study of PDEs. A detailed discussion can be found in [40].

2.2.1.2 Well-posedness of the Weak Formulation

Consider again the generic weak formulation in (2.7). We say that it is well-posed if there exits a solution and that solution is unique and depends continuously on the right-hand side data. The Lax-Milgram theorem is a famous result which gives the conditions for which such a \( u \) exists.

**Theorem 2.9** (Lax-Milgram Theorem). Let \( V \) be a Hilbert space with norm \( \|\cdot\|_V \) dual space \( V' \).
The problem defined: Find $u \in V$ such that

$$a(u, v) = f(v) \quad \forall v \in V$$

is well-posed if the following hold:

1. $a(u, v)$ is a bilinear form,

2. $a(u, v)$ is continuous, i.e., for some $\alpha > 0$,

   $$a(u, v) \leq \alpha \|u\|_V \|v\|_V$$

3. $a(\cdot, \cdot)$ is coercive, i.e., for some $\beta > 0$,

   $$a(u, u) \geq \beta \|u\|_V^2$$

4. $f(v)$ is a linear functional and continuous:

   $$f(v) \leq \gamma \|v\|_V.$$

Additionally, $\forall f \in V'$, the solution will be bounded:

$$\|u\|_V \leq \frac{1}{\beta} \|f\|_{V'}.$$

**Proof.** See [9] for proof of result.

For the Poisson problem, $V = H^1_0(\Omega)$ is a Hilbert space, $f \in L^2(\Omega) \subset H^{-1}(\Omega) = V'$,

$$a(u, v) = (\nabla u, \nabla v)_{L^2(\Omega)} \leq \|u\|_1 \|v\|_1,$$

$$a(u, u) = (\nabla u, \nabla u)_{L^2(\Omega)} = \|u\|_1^2 \leq \|u\|_1^2,$$

and, by (2.8),

$$f(v) = (v, f)_{L^2(\Omega)} \leq \|v\|_0 \|f\|_0 \leq C \|v\|_1 \|f\|_0.$$
Therefore the Poisson problem defined by (2.7) is well-posed with

\[ \|u\|_V \leq \|f\|_V. \]

### 2.2.2 Galerkin Method

We will seek discrete solutions to the weak formulation (2.7) by way of the *Galerkin method*. This method consists of defining a finite dimensional space \( V_h \subset V \) and constructing an approximation \( u_h \) as a linear combination of the basis functions of \( V_h \). The problem can be reformulated as

Find \( u_h \in V_h \) such that

\[ a(u_h, v_h) = f(v_h) \quad \forall v_h \in V_h. \tag{2.10} \]

We must also note that this problem is solved on the discrete mesh \( \Omega_h \).

#### 2.2.2.1 Mesh Discretization

We start by decomposing \( \Omega_h \) into a set of \( M \) quadrilateral cells \( K_i \) forming what is called a *triangulation*\(^1\) which we will denote by \( T_h = \{K_i\}_{i=1}^M \). From [23, 39], a triangulation is said to be *conforming* if

(i) \( \bar{\Omega}_h = \bigcup_{i=1}^M K_i \)

(ii) \( K_i \cap K_j, i \neq j \), is either

(a) a single point (vertex)

(b) a entire edge (in 2D or 3D) or face (in 3D)

Define \( h = \max_{1 \leq i \leq M} \text{diam}(K_i) \) to be the mesh size. Figure 2.1 gives an example of both a conforming and a non-conforming mesh with the same mesh size \( h \). In Section 2.8.1, we will relax condition (ii) to allow for “hanging nodes”, i.e., vertex of \( K_i \) lies on edge of \( K_j \) (red nodes in Figure 2.1).

---

\(^1\)The name “triangulation” comes from the fact that \( \Omega_h \) is often decomposed into triangles (in 2D). For examples which use triangle meshes, see [39, 23, 64]. We will restrict our focus to quadrilaterals, as used in deal.ii.
Figure 2.1: Example of conforming and non-conforming meshes with identical mesh size. Red circles indicate hanging nodes.

Figure 2.2: Global refinement.

Mesh Refinement  We will see later, the accuracy of the discrete solution \( u_h \) is directly related to the mesh size \( h \), that is, as \( h \to 0 \), typically we have \( u_h \to u \) where \( u \) is the true solution to our problem. Therefore, refining each cell (as depicted in Figure 2.2) results in a decrease of \( h \) and more accurate \( u_h \). This is what is referred to as \( h \)-refinement. It is also worth noting that this is not the only type of mesh refinement. For instance one could think of increasing the order of the finite element space, say from linear elements to quadratic elements, which is referred to as \( p \)-refinement (\( p \) representing the degree of the finite element). For this, the reader is referred to [108]. We will continue to refer to \( h \)-refinement as just refinement.

While Figure 2.2 give a refinement scheme where each cell is refined, it is possible to consider a refinement scheme where only a subset of the cells are refined (Figure 2.3). We will refer to this as adaptive refinement. From [15], we can think of adaptive refinement in one of two ways:

1. given an amount of work \( N \) (think of work in terms of the number of cells), we would like to solve with minimum error, or

2. given an error tolerance, we would like to solve with the minimum work \( N \).
For either, we can choose to only refine the cells which exist around “interesting features”, e.g., singularities in the underlying solution, discontinuities in coefficients, or large gradient jumps in the computed solution. Using these techniques of refinement, it has been shown that one can often recover convergence properties for solutions that lack the required regularity conditions [15].

While the amount of computing resources that can be saved by using adaptive mesh refinement can be great (see Figure 2.4), it does not come without it’s difficulties in implementation. For this reason, special care must be taken when generalizing algorithms to work on adaptively refinement meshes.

### 2.2.2.2 Finite Element Spaces

Once we have defined our discrete mesh $\Omega_h$, we must now define our finite element function space $V_h$ from the discretized weak formulation (2.10). Consider the following definition from [39]
of the space of polynomials of degree $k$ on the unit square (or unit cube in 3D)

\[ Q_k = \left\{ q : \hat{K} \to \mathbb{R} \text{ with } q(x) = \sum_{0 \leq i_1, \ldots, i_d \leq k} \alpha_{i_1, \ldots, i_d} x_1^{i_1} \cdots x_d^{i_d} : \alpha_{i_1, \ldots, i_d} \in \mathbb{R} \right\} \]  

(2.11)

The following gives the formal definition of the Lagrangian Finite Element:

**Definition 2.12.** Let $\hat{K}$ be the unit cuboid in $\mathbb{R}^d$, $d = 2, 3$. Let $Q_k$ be the space defined in (2.11) with $n = \dim(Q_k)$. Let $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ be linear forms acting on the elements of $P$ such that, for nodes $\{a_i\}$ with coordinates

\[ \left( \frac{i_1}{k}, \ldots, \frac{i_d}{k} \right), \quad 0 \leq i_1, \ldots, i_d \leq k \]

we have for $p \in P$

\[ \sigma_i(p) = p(a_i), \quad 1 \leq i \leq n. \]

The elements of $\Sigma$ are referred to as the degrees of freedom. The triplet $\{\hat{K}, Q_k, \Sigma\}$ is referred to as a *Finite Element*. Figure 2.5 gives a representation of $Q_k$ and $\Sigma$.

Since the finite element in definition 2.12 is defined on a reference element $\hat{K} = [0, 1]^d$, we must introduce a mapping

\[ F_K : \hat{K} \to K, \]

depicted in Figure 2.6, which maps values defined on the reference element to values defined on the real element in Cartesian space.

Now, we can define the space $V_h \subset V$ as

\[ V_h := \{ v \in V : \forall K \in T_h, v|_K \circ F_K \in Q_k \}. \]  

(2.13)
Using this definition of $V_h$, let $\{\varphi_0, \ldots, \varphi_{n-1}\}$ be a basis of $V_h$ as defined in [39], where $n$ is the total number of degrees of freedom on $T_h$ and $n_K$ are the local degrees of freedom on cell $K$. We can then define functions on the mesh as linear combinations of the $\{\varphi_i\}$ basis, i.e., $f(x) = \sum_{i=0}^{n-1} c_i \varphi_i(x)$ for coefficients $c_i$. Choosing a quadrature rules with $n_q$ points $x_q$ and weights $\omega_q$, we have that the integration over $\Omega_h$ of the function $f$ is given by

$$
\int_{\Omega} f(x) \, dx = \sum_{K \in T_h} \int_K \sum_{i=0}^{n-1} c_i \varphi_i(x) \, dx
$$

$$
= \sum_{K \in T_h} \int_K \sum_{i=0}^{n_K-1} c_i J_K^{-1} \cdot \hat{\varphi}_i(\hat{x}) \cdot |\det J_K(\hat{x})| \, dx
$$

$$
\approx \sum_{K \in T_h} \sum_{q=0}^{n_q-1} w_q |\det J_K(\hat{x}_q)| \sum_{i=0}^{n_K-1} c_i J_K^{-1}(\hat{x}_q) \cdot \hat{\varphi}_i(\hat{x}_q)
$$

where $J_K$ is the Jacobian of the transformation $F_K$ and $\hat{\varphi}_i(\hat{x})$ is the basis function on the reference element $\hat{K}$. We refer to these $\{\hat{\varphi}_i\}$ functions as \textit{nodal shape functions}: polynomials of degree $k$ centered around degree of freedom node $a_i$ (from definition 2.12), and which satisfy

$$
\hat{\varphi}_i(\alpha_j) = \begin{cases} 
0 & \text{if } i \neq j \\
1 & \text{if } i = j
\end{cases}
$$

(2.15)

Thus, the nodal shape functions have only local support on $\Omega_h$.

\textbf{Continuous vs. Discontinuous Finite Elements}  So far the discussion has been centered around continuous finite elements, where bordering cells share degrees of freedom along their shared

\footnote{There are other finite element shape functions that are not nodal, i.e., do not follow (2.15). However, as we are only considering nodal shape functions, we will from here on refer to “nodal shape functions” and just “shape functions”.}
features. This does not have to be the case, however. Discontinuous Galerkin (DG) finite element methods are an alternative to continuous methods. We mention these methods here only because the geometric multigrid method described in Section 2.5 will be generalized in such a way to include them, and as an example in Chapter 3 we will include a computation with a DG element to demonstrate the flexibility of the developed method.

Starting with the same mesh, we introduce finite element spaces which are not conforming to the space \( V_h \), in particular spaces, with no continuity requirements. Therefore, the straight-forward discretization using (2.10) is inconsistent and typically not converging to the continuous solution. This is remedied by introducing so-called flux terms on the interfaces, which guarantee consistency and stability of the method. Accordingly, the bilinear form on \( T_h \) depends on the mesh itself and we write: find \( u_h \in V_h \) such that

\[
a_h(u_h, v_h) = f_h(v_h) \quad \forall v_h \in V_h.
\]

As an example, we mention the interior penalty method for the Poisson equation [6], with the bilinear form

\[
a_h(u, v) := \sum_{K \in T_h} \int_K \nabla u \cdot \nabla v \, dx + \sum_{F \in F_h} \int_F \left[ \sigma uv - \partial_n uv - u \partial_n v \right] ds
\]

\[
+ \sum_{F \in F_h} \int_F \left[ \sigma \left[ [u] [v] - 2 [\nabla u] \{ \nu n \} - 2 \{ u n \} \{ \nabla v \} \right] ds. \tag{2.17}
\]

Here, \( F_L \) are the \((d-1)\)-dimensional interfaces between mesh cells of \( T_L \) and \( F_L \) are the facets of cells on the boundary of \( \Omega_h \). Every face \( F \in F_L \) has two adjacent cells, say \( K^+ \) and \( K^- \). We call the restriction of the finite element functions \( u \) and \( v \) to these cells \( u^+, \), \( u^- \), \( v^+ \), and \( v^- \), respectively. With these definitions, we have the jump and mean value operators

\[
[u] = u^+ - u^-,
\]

\[
\{ u \} = \frac{u^+ + u^-}{2}. \tag{2.18}
\]

The analysis from here on will be generalized to include both continuous and discontinuous finite elements, and the inner products \( a(\cdot, \cdot) \) and \( a_h(\cdot, \cdot) \) can be used interchangeably.
2.2.2.3 Linear System

Using the above definitions, we will write \( u_h \in V_h \) as \( u_h = \sum_{j=0}^{n-1} u_j \varphi_j \), where \( \{\varphi_1, \ldots, \varphi_{n-1}\} \) is the finite basis of \( V_h \). We can write the discrete problem (2.10) as

Find \( u_h = \sum_{j=0}^{n-1} u_j \varphi_j \) such that

\[
a(u_h, \varphi_i) = f(\varphi_i) \quad 0 \leq i \leq n - 1
\]

\[
\Leftrightarrow \sum_{j=0}^{n-1} u_j \cdot a(\varphi_j, \varphi_i) = f(\varphi_i) \quad 0 \leq i \leq n - 1
\]  

\[
\Leftrightarrow \sum_{j=0}^{n-1} u_j \cdot (\nabla \varphi_i, \nabla \varphi_j) = (\varphi_i, f) \quad 0 \leq i \leq n - 1
\]

\[
\Leftrightarrow \text{Solving } AU = F \text{ where } A_{i,j} = (\nabla \varphi_i, \nabla \varphi_j) \text{ and } F_i = (\varphi_i, f)
\]

We compute the integrals \( (\nabla \varphi_i, \nabla \varphi_j) \) and \( (\varphi_i, f) \) using the same process described in (2.14), i.e., a loop over cells and quadrature with transformations to and from the reference elements. This will result in a system that is sparse due to the fact that each \( \varphi_i \) has local support in \( \Omega_h \), and, for the specific case of the Poisson equations, \( A \) is positive definite [64].

2.2.2.4 Error Estimates

We now have a well defined, discretized weak formulation given by equation (2.10). Given that \( V_h \subset V \), an obvious result of the Lax-Milgram Theorem is the following:

\[
\|u_h\|_V \leq \frac{1}{\beta} \|f\|_V,
\]

where \( a(u, u) \geq \beta \|u\|_V \). Further, we have the following results:

Lemma 2.20 (Cea’s Lemma). For \( a(\cdot, \cdot) \) and \( f(\cdot) \) satisfying the conditions in Theorem 2.9 and \( V = H^k_0(\Omega) \),

\[
\|u - u_h\|_1 \leq \frac{\alpha}{\beta} \inf_{v_h \in V_h} \|u - v_h\|_1.
\]

Proof. Proof is straightforward and given in [23]; more generic version given in [39].

Theorem 2.21. Let \( u \) be a solution to problem (2.7) with sufficient regularity, i.e., \( u \in H^{k+1}(\Omega) \cap H^k_0(\Omega) \), \( k \geq 1 \), and \( V_h \) be the discrete space given by (2.13). Then the discrete solution \( u_h \) of problem...
(2.10) has the following estimate

\[ \|u - u_h\|_1 \leq Ch^k \|u\|_{k+1} \]  \hspace{1cm} (2.22)

where \( C \) is independent of \( h \). Further,

\[ \|u - u_h\|_0 \leq Ch^{k+1} \|u\|_{k+1} \]  \hspace{1cm} (2.23)

where, again, \( C \) is independent of \( h \).

Proof. See [44]. \( \Box \)

Therefore, as we increase the polynomial degree of the finite element, we should see an increase in the order of convergence.

\section*{2.3 Finite Element Theory for the Stokes Equations}

Using principles laid out in Section 2.2, we seek a finite element approximation to the Stokes equations given in (2.4). These approximations are referred to as mixed (finite element) approximation due to the fact that we are solving for variables (velocity and pressure) from more than one independent approximation spaces [29, 64]. Choosing these spaces is non-trivial as it takes special combinations of elements to produce convergent solutions.

\subsection*{2.3.1 Weak Formulation}

Multiplying (2.4) by smooth enough test functions \((\mathbf{v}, q)\) and integrating over the domain yields

\[ \int_\Omega \mathbf{v} \cdot (-\Delta \mathbf{u}) \, dx + \mathbf{v} \cdot \nabla p \, dx = \int_\Omega \mathbf{v} \cdot \mathbf{f} \, dx \]

and

\[ \int_\Omega q \cdot (\nabla \cdot \mathbf{u}) \, dx = \int_\Omega q \cdot g \, dx. \]

Using integration by parts, we have

\[ -\int_\Omega \mathbf{v} \cdot (\Delta \mathbf{u}) \, dx = \int_\Omega \nabla \mathbf{v} : \nabla \mathbf{u} \, dx \]
and
\[
\int_{\Omega} v \cdot \nabla p \, dx = - \int_{\Omega} (\nabla \cdot v) \cdot p \, dx
\]
which gives the following weak formulation of (2.4):

Find \( u \in V \) and \( p \in Q \) such that
\[
a(u, v) + b(v, p) = f(v) \quad \forall v \in V \\
b(u, q) = g(q) \quad \forall q \in Q
\] (2.24)

where
\[
a(u, v) = \int_{\Omega} \nabla v : \nabla u \, dx, \quad b(v, p) = \int_{\Omega} (\nabla \cdot v) p \, dx, \\
f(v) = \int_{\Omega} vf \, dx \quad \quad \quad \quad g(q) = - \int_{\Omega} qg \, dx.
\]

For these integrals to be well defined, consider \( V = [H^1_0(\Omega)]^d \), \( Q = L^2(\Omega) \) and \( f \in [L^2(\Omega)]^d \) (or \( f \in [H^{-1}(\Omega)]^d \)) with \( f(v) = \langle v, f, \rangle_{-1} \), however, note that if \((u, p)\) is a solution to (2.24), then \((u, p + C)\) is also a solution for any constant \(C\). Because of this, it is common to require

\[ p \in Q = L^2_0(\Omega), \]

where
\[
L^2_0(\Omega) := \left\{ p \in L^2(\Omega) : \int_{\Omega} p \, dx = 0 \right\},
\]
i.e, the space of all \( L^2(\Omega) \) functions with mean zero. This function space is also a Hilbert space under the \( L^2(\Omega) \) norm and inner product [77]. Consider the following result

**Theorem 2.25.** For any \( f \in [H^{-1}(\Omega)]^d \) and \( g \in L^2_0(\Omega) \), there exists a unique solution \((u, p) \in V \times Q\) to (2.4) and (2.24) with

\[
\|u\|_1 + \|p\|_0 \leq c_1 \|f\|_{-1} + c_2 \|g\|_0.
\]

**Proof.** See Theorem 4.3 in [39] for a proof of this result.

**Remark 2.26.** This proof relies two main steps:

(i) Showing the coercivity of the bilinear form \( a(\cdot, \cdot) \) and
(ii) Showing function spaces $V$ and $Q$ satisfy the well-known inf-sup condition

$$\exists \beta > 0 \text{ s.t. } \inf_{q \in Q} \sup_{v \in V} \frac{b(v, q)}{\|v\|_1 \|q\|_0} \geq \beta \quad (2.27)$$

2.3.2 Discrete Problem

On a regular triangulation $T_h \subset \Omega_h$, and for finite dimensional spaces $V_h \subset V$ and $Q_h \subset Q$, consider the following discrete problem:

Find $u_h \in V_h$ and $p_h \in Q_h$ such that

$$a(u_h, v_h) + b(v_h, p_h) = f(v_h) \quad \forall v \in V_h$$
$$b(u_h, q_h) = g(q_h) \quad \forall q \in Q_h. \quad (2.28)$$

We have the following theorem for existence and uniqueness:

**Theorem 2.29.** Problem (2.28) is well-posed if and only if $V_h$ and $Q_h$ satisfy the discrete inf-sup condition

$$\exists \beta_h > 0 \text{ s.t. } \inf_{q_h \in Q_h} \sup_{v_h \in V_h} \frac{b(v_h, q_h)}{\|v_h\|_1 \|q_h\|_0} \geq \beta_h. \quad (2.30)$$

Further,

$$\|u - u_h\|_1 + \|p - p_h\|_0 \leq C_1 \inf_{v_h \in V_h} \|u - v_h\|_1 + C_2 \inf_{q_h \in Q_h} \|p - q_h\|_0. \quad (2.31)$$

**Proof.** See Proposition 4.14 in [39] for a proof of this result. \qed

A possible choice for the pair $(V_h \times Q_h)$ is given by the so-called Taylor-Hood elements, first introduced in [100], given by

$$V_h = \left\{ v \in V : \forall K \in T_h, \ v|_K \circ F_K \in [Q_{k+1}]^d \right\}$$
$$Q_h = \left\{ q \in Q : q \in C^0(\Omega), \ \forall K \in T_h, \ q|_K \circ F_K \in Q_k \right\} \quad (2.32)$$

for $k \geq 1$, where $Q_k$ is the standard Lagrangian finite element space of degree $k$ polynomials defined on cell $K$, and $F_K$ is the mapping from reference cell to mesh cell, all discussed in Section 2.2.2.2.\(^3\)

This element pair is widely known to satisfy the inf-sup condition (2.30) [39, 29], and the following error estimate holds:

\(^3\)Note that for methods where the pressure space of Stokes is discontinuous (as is common in the literature [86, 82, 91]), one only needs to drop the requirement that $q \in C^0(\Omega).
**Theorem 2.33.** Let \((u, p)\) be a solution to problem (2.4) with sufficient regularity, i.e., \(u \in [H^{k+1}(\Omega) \cap H^1_0(\Omega)]^d\) and \(p \in H^k(\Omega) \cap L^2_0(\Omega)\), and let \(k \geq 1\) and \((V_h \times Q_h)\) be the Taylor-Hood elements given by (2.32). Then the discrete solution \((u_h, p_h)\) of problem (2.28) has the following estimate

\[
\|u - u_h\|_1 + \|p - p_h\|_0 \leq Ch^{k+1} (\|u\|_{k+1} + \|p\|_k)
\]  

(2.34)

where \(C\) is independent of \(h\). Further,

\[
\|u - u_h\|_0 \leq Ch^{k+2} (\|u\|_{k+1} + \|p\|_k)
\]  

(2.35)

where, again, \(C\) is independent of \(h\).

**Proof.** See [39].

It should be noted that these are not the only element pairs to produce convergent discrete solution (see [39] for discussion on other possible choices). For example, one could use a \(Q_{k+1} \times P_k^{\text{disc}}\) element, where \(P_k^{\text{disc}}\) is the space of discontinuous polynomials of order \(k\) (see, e.g., [86, 82, 91], for codes written with this element for \(k = 2\)). For the remainder of this thesis, however, we will use the Taylor-Hood elements \(Q_k \times Q_{k-1}\).

### 2.3.2.1 Linear System

As in Section 2.2.2.3, we seek coefficients \(u_j\) and \(p_j\) where, for finite element shape functions \(\varphi^u_j\) and \(\varphi^p_j\),

\[
\begin{align*}
  u_h &= \sum_{j=0}^{N_u-1} u_j \varphi^u_j \\
  p_h &= \sum_{j=0}^{N_p-1} p_j \varphi^p_j
\end{align*}
\]  

(2.36)

Here, \(N_u\) and \(N_p\) are the total number of degrees of freedom for velocity and pressure respectively. Then, for each \(0 \leq i \leq N_u - 1\) and \(0 \leq l \leq N_p - 1\),

\[
\begin{align*}
  a(\varphi^u_i, u_h) + b(\varphi^u_i, p_h) &= f(\varphi^u_i) \\
  b(u_h, \varphi^p_l) &= g(\varphi^p_l)
\end{align*}
\]  

(2.37)
Solving this system for coefficients $U = \{u_i\}$ and $P = \{p_j\}$ is equivalent to solving the block linear system
\[
\begin{pmatrix}
A & B^T \\
B & 0
\end{pmatrix}
\begin{pmatrix}
U \\
P
\end{pmatrix}
= \begin{pmatrix}
F \\
G
\end{pmatrix}
\] (2.38)

where
\[
A_{ij} = 2\mu \int_{\Omega} \nabla \varphi_i^u : \nabla \varphi_j^u \, dx, \quad B_{ij} = \int_{\Omega} \varphi_i^p (\nabla \cdot \varphi_j^u) \, dx,
\]
\[
F_j = \int_{\Omega} \varphi_j^u f \, dx, \quad G_j = -\int_{\Omega} \varphi_j^p g \, dx.
\]

This is often referred to as a **saddle-point problem** (in its most basic form) [19], and this system can often be non-trivial to solve as it is not diagonally dominant and there exists a one-dimensional null space for the pressure since we do not constrain the mean pressure of $p$. Chapter 4 will be focused on finding robust solvers for such a system.

### 2.4 Krylov Subspace Methods

The solution to large, linear systems by means of direct solvers will be infeasible for large-scale finite element systems. At a complexity of $O(n^3)$, where $n$ is the number of unknowns in the system, one typically cannot solve a system of more than $10^6$ unknowns directly in a reasonable amount of time, leading to the need for the development of iterative solvers. One such class of iterative solvers commonly used in finite elements (and many other disciplines) are Krylov Subspace methods. For this section we will be focused on solving the generic problem: Find $x \in \mathbb{R}^n$ s.t. $Ax = b$ for $b \in \mathbb{R}^n$ and non-singular $A \in \mathbb{R}^n \times \mathbb{R}^n$.

Following notation in [93], given an initial guess $x_0$, Krylov methods find iterates $x_m$ in the affine subspace $x_0 + \mathcal{K}_m$ by requiring $b - Ax_m \perp \mathcal{L}_m$, where $\mathcal{K}_m$ is the Krylov subspace defined as

\[
\mathcal{K}_m(A, v_1) = \text{span} \{ v_1, A v_1, A^2 v_1, \ldots, A^{m-1} v_1 \}
\] (2.39)

for some vector $v_1$, and $\mathcal{L}_m$ is some other $m$-dimensional subspace which defines the particular method.
2.4.1 GMRES

The first Krylov subspace method we will consider is the Generalized Minimum Residual Method or GMRES developed in [94]. Here we take $L_m = AK_m$ and $v_1 = r_0 / \|r_0\|$ where $r_0 = b - Ax_0$, and we seek approximates of the form $x = x_0 + z$ such that

$$x = \arg\min_{z \in K_m} \|b - A(x_0 + z)\|.$$  \hspace{1cm} (2.40)

2.4.1.1 Arnoldi’s Method

Before defining the GMRES algorithm, we first must introduce Arnoldi’s method. Introduced in [8], Arnoldi’s method provides an orthonormal basis $v_1, \ldots, v_m := V_m$ for the Krylov subspace

$$K_m = \text{span} \{v_1, Av_1, A^2v_1, \ldots, A^{m-1}v_1\}.$$  \hspace{1cm} (2.41)

Algorithm 1 gives the implementation in exact arithmetic using Gram-Schmidt. In practice, however, using modified Gram-Schmidt or Householder orthogonalization techniques is common as these methods are more robust with respect to round-off error [93].

Given: $v_1$ s.t. $\|v_1\| = 1$

for $j := 1, \ldots, m$

- Compute $h_{i,j} = (Av_j, v_i)$, $i = 1, \ldots, j$
- Compute $w_j = Av_j - \sum_{i=1}^{j} h_{i,j}v_i$
- Set $h_{j+1,j} = \|w_j\|$  
- if $h_{j+1,j} = 0$ then  
  Stop
- Set $v_{j+1} = w_j/h_{j+1,j}$

Algorithm 1: Arnoldi’s method for finding orthogonal basis of $K_m$.

2.4.1.2 GMRES Algorithm

Once we have obtained $V_m = \{v_1, \ldots, v_m\} \in \mathbb{R}^n \times \mathbb{R}^m$ and $\bar{H}_m = \{h_{i,j}\} \in \mathbb{R}^{m+1} \times \mathbb{R}^{m}$, then we have the well known result from [93] among others

$$AV_m = V_{m+1}\bar{H}_m.$$  \hspace{1cm} (2.42)
Setting $z = V_m y$, GMRES can be seen as a minimization of

$$J(y) = \left\| r_0 \right\| v_1 - A V_m y \right\|$$

$$= \left\| V_{m+1} \left( \left\| r_0 \right\| e_1 - \bar{H}_m y \right) \right\|$$

$$= \left\| \left\| r_0 \right\| e_1 - \bar{H}_m y \right\|$$ (2.43)

with solution $x_m = x_0 + V_m y_m$ which minimizes the norm of the residual over all possible solutions of the form $x_0 + \mathcal{K}_m$ [94].

Algorithm 2 gives the GMRES implementation. For the stopping criteria, from [94], if one uses a QR-factorization when solving the minimization problem, the norm of the residual associated with $x_m$ is recovered by looking at the $(m+1)$st component of $Q_m \left\| r_0 \right\| e_1$, where $Q$ is from the QR factorization of $A$, and therefore no extra residual computation is required.

for $m := 1, 2, \ldots$, until convergence, \ldots do

Compute $V_m$ and $\bar{H}_m$ by Arnoldi’s method
Compute $y_m$ by minimizing (2.43)
Set $x_m = x_0 + V_m y_m$, Stop if criteria satisfied.

Algorithm 2: GMRES method for solving $Ax = b$.

The power of GMRES is that in every step the method is guaranteed to not breakdown for any invertible, or even non-invertible, matrix, unless the residual vector is zero [93], and that in each step the norm of the residual is guaranteed to be reduced [94]. Thus we have guaranteed convergence.

However, as a major drawback to the method, in each step we must store an extra vector in the space $\mathbb{R}^n$ and compute an extra matrix-vector product, which can be a major disadvantage if we want to solve large problems which require near maximal storage before the GMRES solve. One way of reducing the storage requirement is the restarted GMRES method or GMRES($m$). Discussed in [94, 93], this method restarts the GMRES algorithm after each $m$ iterations and therefore the storage requirement remains bounded, however, we are no longer guarantee convergence unless $m = n$. While this method is widely used, GMRES($m$) without a suitable preconditioner (discussed later) has been known to stagnate when $A$ is not positive definite [93]. GMRES($m$) will be used interchangeably with GMRES in the remainder of the thesis, as we will always use some restarted version.
2.4.2 CG

The Conjugate Gradient method, or CG, (developed in [59]) is a Krylov subspace method for solving symmetric, positive-definite (SPD) systems of equations. Here we take \( L_m = K_m \), that is, we search for new solutions \( x_m \) whose residuals are orthogonal to the Krylov subspace at step \( m \). Derived from Arnoldi’s method, the CG algorithm searches for iterates \( x_m = x_0 + V_m y_m \) where \( y_m = H_m^{-1} \|r_0\| e_1 \), and by using the fact that \( A \) is SPD (and therefore \( H_m \) is tri-diagonal), can be expressed without the necessity of inverting any matrices. Algorithm 3 gives the statement of the method. Looking at the implementation, CG requires the storage of only 4 vectors as opposed to \( m + 1 \) for GMRES.

\[
\begin{align*}
\text{Set } & p_0 = r_0 \\
\text{for } & m := 1, 2, \ldots, \text{until convergence, } \ldots \text{ do} \\
\text{Compute } & \alpha_m = (r_m, r_m)/(Ap_m, p_m) \\
\text{Set } & x_{m+1} = x_m + \alpha_m p_m \\
\text{Set } & r_{m+1} = r_m - \alpha_m Ap_m, \text{ Stop if criteria satisfied.} \\
\text{Compute } & \beta_m = (r_{m+1}, r_{m+1})/(r_m, r_m) \\
\text{Compute } & p_{m+1} = r_{m+1} + \beta_m p_m
\end{align*}
\]

Algorithm 3: CG method for solving \( Ax = b \).

It has been show that the search directions \( p_j \) are A-orthogonal, i.e. \( (Ap_i, p_j) = 0 \) for \( i \neq j \) and the residual vectors are orthogonal to one another, and similar to the steepest decent algorithm, for CG there is also the result that for exact solution \( x^* \),

\[
\|x_m - x^*\|_A \leq 2 \left[ \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right]^m \|x_0 - x^*\| \tag{2.44}
\]

where \( \kappa \) is the condition number of the matrix \( A \) [93]. For these reasons CG is among the cheapest Krylov subspace methods, and it is the method of choice for SPD systems.

2.4.3 IDR(s)

Lastly we will consider the IDR(s) method proposed in [96]. This method is based on the induced dimension reduction (IDR) method, an iterative method for solving linear systems based on the IDR Theorem from [106], which states that one can generate a nested set of spaces \( G_j \) where
\( G_0 = \mathcal{K}_n(A, v_0) \) for \( v_0 \) non-zero, and

\[
G_j = (I - \omega_j A)(G_{j-1} \cap S)
\]

for any \( S \subset \mathbb{C}_n \) and chosen scalars \( \omega_j \). The IDR method chooses updates such that the residuals lie in \( G_j \), and since \( G_j = 0 \) for some \( j < n \), the algorithm converges, with convergence properties closely related to BiCGStab depending on the choice of \( S \) and \( \omega_j \). In 2008, IDR(s) was proposed. This method used the IDR theorem and, by defining \( S \) to be the null space of a set of \( s \) randomly chosen, orthonormal vectors of size \( n \), showed that now the dimension reduction of \( \{G_j\} \) was bounded by \( s \), that is, for \( d_j = \dim(G_j) \),

\[
0 \leq d_j - d_{j+1} \leq d_{j-1} - d_j \leq s,
\]

instead of unknown (as in the previous IDR theorem). This implies that the speed of convergence of the algorithm can be improved by increasing \( s \). For the case of \( s = 1 \), this is similar to the original IDR method, and, with cleverly chosen \( \omega_j \), equivalent to BiCGStab [96]. In the same paper, IDR(s) was shown to approach the convergence rate of GMRES as \( s \) was increased, which is important since, unlike GMRES, IDR(s) has no minimum residual guarantee. Also it is of note that, unlike GMRES, IDR(s) has a short-term recurrence, requiring the storage of \( 5 + 3s \) vectors, but, as \( s \) increases, the method becomes more and more expensive, requiring additional matrix-vector products. The reader is referred to [48] for an efficient implementation of IDR(s). For implementation purposes, following [48], we define \( P \) as \( s \) randomly chosen vectors whose entries come from a normal distributions with mean 0 and standard deviation 1 for the test in Chapter 5.

### 2.4.4 Preconditioning

Depicted in equation (2.44), the convergence of Krylov solvers are known to depend on the condition number \( \kappa \) of the linear system \( A \). In the Finite Element context, as one continues to refine the mesh, \( \kappa \) increases without bound (from [29], \( \kappa \in \mathcal{O}(h^{-2}) \) for uniform refinement), and therefore the convergence of the Krylov subspace iteration slows down. To demonstrate this, Table 2.1 gives
<table>
<thead>
<tr>
<th>DoFs</th>
<th>CG iters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,089</td>
<td>41</td>
</tr>
<tr>
<td>4,225</td>
<td>84</td>
</tr>
<tr>
<td>16,641</td>
<td>168</td>
</tr>
<tr>
<td>66,049</td>
<td>329</td>
</tr>
<tr>
<td>263,169</td>
<td>661</td>
</tr>
</tbody>
</table>

Table 2.1: Number of CG iterations for solving Poisson with $Q_1$ element.

the iteration counts for a CG solve of

$$-\Delta u = 1 \text{ in } \Omega$$

$$u = 0 \text{ on } \partial \Omega.$$ 

where $\Omega = [0,1]^2$ for different refinements of the mesh. Given that we would like to solve systems with much more than a million unknowns, these solvers by themselves will not be sufficient.

To combat this, it is common to use a technique called preconditioning. Consider the system

$$AM^{-1}y = b, \quad x = M^{-1}y$$

which is commonly referred to as the right-preconditioned system.\(^4\) If $M$ is defined such that the condition number $\kappa(AM^{-1})$ is small, then we should see faster convergence inside the Krylov method [93]. Further, for an effective preconditioner on large scale computations, $M^{-1}$ should be relatively quick to compute, and, for Finite Elements, the preconditioner is considered “optimal” if $\kappa(AM^{-1})$ is independent of $h$, giving constant Krylov iterations for different refinements of the mesh [38].

For an example of how preconditioning is used, consider Algorithm 4 which gives the Preconditioned CG method. Here we find the preconditioned search directions $q_m = M^{-1}p_m$ and store an extra preconditioned residual vector $z_m = M^{-1}r_m$. It should be noted that we must store an additional two vectors, and that in every iteration we need one application of the preconditioner.

For preconditioned variants of GMRES and IDR($s$), the reader is referred to [93] and [104], respectively. It should be noted that while GMRES, like CG, only needs a single preconditioner application per iteration, IDR($s$) will need $s+1$ preconditioner applications. For complex preconditioners, this can make these methods quite slow, a topic that will be explored in Chapter 5.

\(^4\)Other variants are left-preconditioning and split-preconditioning, discussions of which can be found in [93]. We only consider right-preconditioning as the residuals during the Krylov iteration match the residuals of the un-preconditioned system, which is not true of the other two.
Set $z_0 = M^{-1}r_0$ and $q_0 = z_0$

**Algorithm 4: Preconditioned CG method for solving $Ax = b$ with preconditioner $M$.**

**2.4.4.1 Flexibility of a Krylov Method**

Generally, when discussing preconditioned Krylov methods, it is assumed that the preconditioner $M$ can be written as a linear operator, and that in each iteration, the preconditioner is unchanged, however this will not always be the case. We discuss an example of such a preconditioner in Section 4.3, where the application contains a CG solve which cannot be written as a linear operator, and for which $M_i \neq M_j$ for $i \neq j$. For these types of preconditioners it is important that we have a *flexible* Krylov subspace method, that is, a Krylov method that can handle varying, non-linear preconditioners. For the computations throughout the thesis, we require that both GMRES and IDR($s$) are flexible.

For GMRES, each iterate $x_m$ is expressed as a linear combination of preconditioned vectors $z_j = M^{-1}v_j$ where $j = 1, \ldots, m$. The implementation should not store each $z_j$, however. Instead, at the end of the solve, one can apply the preconditioner to $V_my_m$, recovering all the preconditioned vectors $z_j$. For a preconditioner which changes, however, at any iteration $i$, $z_j = M_i^{-1}v_j \neq M_j^{-1}v_j$, and therefore, for GMRES to be flexible, we must store the $z_j$ vectors in addition to the $v_j$ vector since we cannot recompute them. This version is called FGMRES and was developed in [92]. The implementation is straightforward, requiring very little extra lines of code compared with GMRES [92], however it roughly doubles the storage requirements of GMRES, which exacerbates the main issue of the method.

A flexible version of IDR($s$) was proposed in [104]. Since there exists no long-term recur-
rence in the algorithm, there is no additional storage requirement, however it was explained that when including variable preconditioning, the properties of the IDR theorem are lost. Nevertheless, they show that with mild variation in the preconditioning, preconditioned IDR(s) appeared flexible [104]. However, the question about strongly varying preconditioning is open. Also, since \( s + 1 \) preconditioning steps are required for each iteration, they claim in [104] that keeping the preconditioner constant over each iteration allowed for the recovery of IDR theorem properties, but this is not possible for the implementations in this thesis since the preconditioners must be defined before the iteration, and the same preconditioner is used in each step. When using IDR(s) in Chapter 5, we will explore the flexibility of the method with higher levels of variation in the preconditioning.

2.5 Multigrid Methods

We will now turn our attention to another type of iterative methods: multigrid methods. Multigrid methods are known to be “optimal” iterative methods for linear systems arising from discretizations of an elliptic boundary valued problems as they can be represented as an iterative scheme with contraction number less than one, independent of \( h \) [24, 54, 25, 103], with optimal complexity. The main idea behind these methods is that of the v-cycle, given by the following steps:

1. **Pre-smoothing**: Apply a series of cheap, iterative, approximate solvers aimed at reducing only the high-frequency errors of the current residual. This allows for an accurate representation on a coarser solution space.

2. **Restriction**: Represent the residual from the fine space onto a coarser space.

3. **Coarse-grid correction**: Calculate a correction of the remaining low-frequency errors of the residual via a direct solve on the coarse space.

4. **Prolongation**: Represent the correction of the solution on the coarse space to the finer space.

5. **Post-smoothing**: Apply a series of cheap, iterative, approximate solvers aimed at reducing the accumulated error from prolongation.

This two-grid algorithm can be extended to a Multigrid algorithm by recursively calling the two-grid algorithm as the coarse grid correction. Figure 2.7 gives a graphical representation of the v-cycle.
Here, we refer to the hierarchy of spaces has having level $\ell$, where $\ell = 0$ is the coarsest space and, for now, we will say $\ell = L$ is the finest space. For all applications, we will consider the pre-smoothing and post-smoothing to be identical operations, and they will be referred to as just smoothing.

### 2.5.1 Geometric Multigrid for Globally Refined Meshes

Dating back to the 1960’s [41], so called geometric multigrid methods (GMG) have been used which employ a hierarchy of meshes

\[ \mathcal{T}_0 \sqsupset \mathcal{T}_1 \sqsupset \cdots \sqsupset \mathcal{T}_L, \]  \hspace{1cm} (2.46)

where the symbol “$\sqsupset$” denotes that each mesh is nested, that is, every cell of a mesh on the left of this symbol is the union of one or more cells of the mesh on the right. In this section, we describe a generation of meshes $\mathcal{T}_{\ell+1}$ from $\mathcal{T}_\ell$ by selecting a subset of its cells and refining these isotropically, bisecting each edge, generating $2^d$ children. A simple, three level example is given in Figure 2.8.

As in Section 2.2, we define finite element spaces $V_\ell$ on these meshes by defining local shape function spaces on each cell $K \in \mathcal{T}_\ell$ and concatenating these spaces, identifying shape functions on adjacent cells which are associated to joint degrees of freedom. For most finite elements (including the ones we consider here) the shape functions on a cell $K$ can be represented as linear combinations of the shape functions on its children in the mesh hierarchy. Therefore, the mesh hierarchy above
induces a chain of finite element spaces

\[ V_0 \subset V_1 \subset \cdots \subset V_L. \] (2.47)

Again, referring to the Poisson equation (2.1) and its weak formulation (2.7), we discretize on each level in the hierarchy and have the following formulation:

Find \( u_\ell \in V_\ell \), such that

\[ a_\ell(u_\ell, v_\ell) = f_\ell(v_\ell) \quad \forall v_\ell \in V_\ell. \] (2.48)

For conforming finite element methods, the bilinear forms and the right hand side are simply the restrictions of \( a(\cdot, \cdot) \) and \( f(\cdot) \) to the space \( V_\ell \). For DG and other stabilized schemes, they contain additional terms for consistency and stability. Associated with the bilinear form \( a_\ell(\cdot, \cdot) \) is a linear operator \( A_\ell : V_\ell \to V_\ell \) defined by

\[ \langle A_\ell u_\ell, v_\ell \rangle_{V_\ell} = a_\ell(u_\ell, v_\ell) \quad \forall u_\ell, v_\ell \in V_\ell. \] (2.49)

Here, the inner product on \( V_\ell \) is the one used in the conjugate gradient method, typically the Euclidean norm of the coefficient vector of a function \( u_\ell \in V_\ell \) with respect to the nodal basis of \( V_\ell \) (see, for instance, the discussion of mesh dependent norms in [24, 29] and their relation to the inner
product of $L^2(\Omega)$). Based on the embeddings in (2.47), we define the grid transfer operators

$$R_T^\ell : V_\ell \rightarrow V_{\ell+1} \quad v \mapsto v \quad (2.50)$$

$$R_\ell : V_{\ell+1} \rightarrow V_\ell \quad \langle R_\ell u, v_\ell \rangle_{V_\ell} = \langle u, v_\ell \rangle_{V_{\ell+1}} \quad \forall v_\ell \in V_\ell. \quad (2.51)$$

On each mesh level $\ell$, we employ a smoother $S_\ell(u_\ell, g_\ell)$, which acts on some right hand side $g_\ell$ and the current iterate $u_\ell$, defined in Algorithm 5. Here, $m_\ell$ is referred to as the number of smoothing steps, and can vary level by level. $B_\ell$ is the type of relaxation method that defines the smoother, for instance, the diagonal of the operator $A_\ell$ (called the Jacobi method), or the lower triangle part (called Gauss-Seidel). Similarly, additive and multiplicative Schwarz methods (introduced in Section 3.4.3) fit into this concept, as well as nonlinear methods like CG or GMRES. In this case, $m_\ell$ is called the number of smoothing steps and $B_\ell$ is the type of relaxation method,

```
function S_\ell(u_\ell, g_\ell):
    for k := 1, \ldots, m_\ell do
        u_\ell ← u_\ell - B_\ell^{-1}(A_\ell u_\ell - g_\ell)
    return u_\ell
```

Algorithm 5: Smoothing operations for level $\ell$ of the Multigrid v-cycle. $m_\ell$ is the number of smoothing steps, $B_\ell$ is the relaxation method.

for instance the diagonal for the Jacobi method or the lower triangle for Gauss–Seidel. Similarly, additive and multiplicative Schwarz methods fit into this concept, but it also extends to nonlinear methods like CG or GMRES.

Now we can define the v-cycle in abstract form, given by Algorithm 6. We see that on level $\ell$ we only need $A_\ell$ and not $f_\ell$ since we are transferring residuals down for the right-hand side of the coarse grid correction.

```
function VCYCLE(\ell, g_\ell):
    if \ell > 0 then
        u^1 ← S_\ell(0, g_\ell)
        u^2 ← u^1 + R_T^{\ell-1}VCYCLE(\ell-1, R_{\ell-1}(g_\ell - A_{\ell-1}u^1))
        u^3 ← S_\ell(u^2, g_\ell)
        return u^3
    else
        return A_0^{-1}g_\ell
```

Algorithm 6: Multigrid v-cycle defined on level $\ell$. 

36
2.5.1.1 Algorithm Complexity

For the globally refined case, we have the following result.

**Proposition 2.52.** The total work of a multigrid v-cycle from a globally refined mesh is $O(n_L)$.

**Proof.** Let $W_\ell$ be the work on level $\ell$ and let $n_\ell = \dim(V_\ell)$. Since the mesh is globally refined, we have that

$$n_{\ell-1} = \alpha n_\ell, \quad \alpha < 1.$$  

In the discrete since, $n_\ell$ is the number of degrees of freedom on level $\ell$, and thus $\alpha_{\text{max}} \approx 1/4$ for 2D and $\alpha_{\text{max}} \approx 1/8$.

Let $m_\ell$ be the number of pre/post-smoothing steps and the smoothers and transfer operators on level $\ell$ be $O(n_\ell)$, and assume that the work on the coarsest grid is negligible, i.e., $W_0 \in O(1)$. Then for $\ell \leq L$, the work up to level $\ell$ is given by

$$W_\ell \leq C(2m_\ell)n_\ell + W_{\ell-1}$$

$$\leq C(n_\ell + n_{\ell-1} + \cdots + n_1 + n_1)$$

$$= C \left( 1 + \sum_{i=1}^{\ell-1} \prod_{j=i+1}^{\ell} \alpha_j \right) n_\ell$$

$$\leq C \left( 1 + \sum_{i=1}^{\ell-1} [\alpha_{\text{max}}]^{\ell-i} \right) n_\ell$$

$$= C \left( 1 + \sum_{k=1}^{\ell-1} [\alpha_{\text{max}}]^k \right) n_\ell$$

$$\leq C \cdot \frac{1}{1 - \alpha_{\text{max}}} \cdot n_\ell$$

Therefore $W_\ell \in O(n_\ell)$, which gives that the total v-cycle work, is $W_L \in O(n_L)$ \qed

2.5.2 Geometric Multigrid for Adaptively Refined Meshes

Several different kinds of adaptive multigrid methods can be distinguished from the types of meshes and level spaces. We can consider a conforming method, generated by bisection or by refinement into $2^d$ children in $d$ dimensions dividing all edges and subsequent closure (red-green refinement). These methods have been implemented for triangular and tetrahedral meshes, and seem to be restricted to these meshes with the exception of a closure mechanism for quadrilaterals based
on division of each edge into three. The alternative are non-conforming methods, most prominently the one-irregular meshes where the difference in refinement between two cells sharing a common edge may not exceed one level. While this constraint is mathematically not necessary, it simplifies the code significantly. This method has been implemented for triangular meshes as well as meshes based on (deformed) hypercubes. As discussed in Section 2.2.2.1, since the meshes are non-conforming, additional care has to be taken to ensure conformity of associated finite element spaces. This is achieved by “elimination of hanging nodes” resulting in algebraic constraints on the possible finite element functions on the finer cell as discussed in Section 2.8.1.

After a locally refined mesh has been constructed, typically in an adaptive algorithm, and its finite element space has been properly defined with or without “hanging nodes”, the resulting mesh has cells on different levels. Thus, using an algorithm employing smoothing operations on all cells on “level \( \ell \) or less” is not of optimal complexity on arbitrary meshes. Two remedies have been proposed: local smoothing [26, 65, 61] and global coarsening [83, 18, 98]. We apply the former for several reasons. First, computational complexity is slightly lower and optimal on all meshes, while there are (extreme) examples for suboptimal complexity of global coarsening. Second, the smoothing operation is always run on meshes without hanging nodes; while this is not an issue for point smoothers like the Jacobi method, it facilitates block smoothers, in particular patch smoothers as in [7, 66]. Finally, implementation on vectorizing and multicore architectures is fairly straightforward.

2.5.2.1 Local smoothing

It is important to first note that we can think of the refinement procedure as producing a tree (or a forest, if \( T_0 \) consists of several cells), where each node is a cell in the mesh hierarchy. The mesh on which we discretize the PDE consists of the leaves of this tree (or forest) and will be denoted as the active mesh \( T_L \). Since it is obtained by local refinement, it consists typically of cells on different levels up to level \( L \). For \( \ell < L \), we will define the level mesh \( T_\ell \) consisting of all cells at level \( \ell \) and of all leaves of the refinement tree with level less than \( \ell \).

With such a definition of \( T_\ell \), a fairly coarse cell can be part of many different level meshes. In order to obtain an algorithm with optimal complexity, smoothing for the degrees of freedom of a given cell should only happen on a single level as these cells will again be present after the transfer and identical smoothing would again be applied, repeating work unnecessarily. This is where local
smoothing derives its name: while we are running a multigrid method for the whole finite element space \( V_\ell \), we restrict smoothing only to the mesh cells which are actually on level \( \ell \). This splitting is explained in Figure 2.9. The mesh \( T_\ell \) is split into the submesh \( T_\ell^S \) of cells strictly on level \( \ell \) and \( T_\ell^L \) of cells on lower levels than \( \ell \). For DG methods, this immediately results in a splitting \( V_\ell = V_\ell^S \oplus V_\ell^L \), where the support of each subspace is its corresponding submesh. The splitting for continuous methods is more complicated, since there are finite element basis functions with support straddling the interface and thus in both \( T_\ell^S \) and \( T_\ell^L \). We now give a short review of the structure of the operators in the multigrid method outlined in [65, 61]. Here, the goal is to implement the algebraic equivalent of the full multigrid method for the space hierarchy \( \{ V_\ell \} \) with operators obeying the subspace splitting.

We start with the observation: conforming methods require that the function on the refined side of a refinement edge coincides with the function on the coarse side. This translates into elimination of degrees of freedom on the refined side and results in \( V_{\ell}^I \subset V_{\ell-1} \). Thus, we can restrict smoothing on level \( \ell \) to \( V_\ell^S \) and can ignore \( V_\ell^I \). Furthermore, in the case of DG methods, \( V_\ell^I = \{0\} \), such that in both cases we can write \( V_\ell = V_\ell^S \oplus V_\ell^I \oplus V_\ell^L \). Our assumptions on local smoothing translate to

\[
S_\ell \begin{pmatrix} x^S \\ x^I \\ x^L \end{pmatrix}, \begin{pmatrix} g^S \\ g^I \\ g^L \end{pmatrix} = \begin{pmatrix} S_\ell^S(x^S, g^S) \\ x^I \\ x^L \end{pmatrix},
\]

(2.54)
where \( S^S(x^S,g^S) \) is now the local smoother on \( V^S_\ell \) only. We observe that, for functions in \( V^L_\ell \), the embedding operator \( R^T_{\ell-1} \) maps a function from \( V_{\ell-1} \) to itself. Therefore, \( R_{\ell-1} \) is the identity on \( V^L_\ell \). Thus, \( R_{\ell-1} \) has the structure

\[
R_{\ell-1} \begin{pmatrix} x^S \\ x^I \\ x^L \end{pmatrix} = R^S_{\ell-1} x^S + R^I_{\ell-1} x^I + x^L. \tag{2.55}
\]

Residuals on the other hand must be computed correctly on the whole space \( V_\ell \) according to

\[
\begin{pmatrix} r^S_\ell \\ r^I_\ell \\ r^L_\ell \end{pmatrix} = \begin{pmatrix} g^S \\ g^I \\ g^L \end{pmatrix} - \begin{pmatrix} A^S_\ell & A^{SI}_\ell & A^{SL}_\ell \\ A^{IS}_\ell & A^I_\ell & A^{IL}_\ell \\ A^{LS}_\ell & A^{LI}_\ell & A^L_\ell \end{pmatrix} \begin{pmatrix} x^S \\ x^I \\ x^L \end{pmatrix}. \tag{2.56}
\]

Note that the matrices \( A^{SL}_\ell \) and \( A^{LS}_\ell \) are the flux matrices of a DG method on the refinement edge and thus vanish for conforming methods. Furthermore, we see in the v-cycle algorithm, that this residual is immediately restricted to the coarse space \( V_{\ell-1} \). Since the restriction acts as identity on \( V^L_\ell \), we can avoid computing \( r^L_\ell \) and defer it to the lower level. Thus, the matrix \( A^L_\ell \) is not needed in computations at all. The matrix \( A^S_\ell \) is used for smoothing on level \( \ell \). The off-diagonal matrices correspond to coupling between degrees of freedom on the cells at the interface, and are needed in addition to \( A^S_\ell \) for a consistent multigrid method.

A major advantage of local smoothing is its fairly simple data structure. The level meshes \( T^S_\ell \) do not have hanging nodes, such that the results of cellwise operations can be entered into global vectors very efficiently without any elimination process.

For local smoothing, we then have a similar result to Proposition 2.52.

**Proposition 2.57.** The total work of a multigrid v-cycle from an adaptively refined mesh is \( O(\sum_{i=1}^{L} n^S_i) \).

**Proof.** The proof is trivial, namely, letting \( n^S_\ell = \text{dim}(V^S_\ell) \), we have

\[
W_\ell \leq C(n^S_\ell + n^S_{\ell-1} + \cdots + n^S_1 + n^S_1) \in O\left(\sum_{i=0}^{\ell} n^S_i\right). \tag{2.58}
\]
In all but the most extreme cases (i.e., the case where each level has the same number of cells),

\[ \sum_{i=0}^{L} n_i^S \in O(n_L). \]

Therefore we claim that local smoothing, defined above and in [65, 61], yields optimal complexity. Chapter 3 will be focused on generating a parallel implementation of this algorithm.

### 2.5.3 Algebraic Multigrid

Developed in the early 1980s [27, 28], Algebraic multigrid (AMG) is a method which constructs a purely algebraic hierarchy of level spaces, that is, given a coefficient matrix \( A \), an AMG algorithm cleverly constructs a hierarchy of \( A_\ell \) matrices only by using the matrix entries of at the next finer level. This can then be seen as a “black box method” as it only relies on the entries of \( A \) and requires no information about the underlying PDE or geometry. For smaller problem sizes, and with low processor counts, this method is a state-of-the-art preconditioner for Poisson-like problems, however, as we increase the problem size and processors used in the computations this method can begin to deteriorate in terms of timings, as the matrices become more and more dense each level down (see Section 4.4 for a demonstration of this behavior). Also, if the underlying geometry of the problem is complicated (say, using adaptive refinement), then AMG may also deteriorate in its approximation since no information about the underlying geometry is used to approximate the coarser matrices. For this reason geometric methods are attractive as they should not suffer this deterioration for complex geometries. For more discussion on AMG methods, we refer the reader to [103, 38]. Chapter 4 will feature an in-depth comparison of AMG (from the Trilinos ML [46] distribution) and the GMG method developed in Chapter 3 for the Stokes equations.

### 2.5.4 Multigrid as a Preconditioner

Multigrid methods, with their convergence properties independent of \( h \) and \( O(n) \) complexity, can be seen as effective iterative solvers. In [29], a theoretical full-multigrid cycle is proposed and proofs the convergence is given. In applied research, [49] offers an example for a Stokes solve using a full-multigrid cycle with a Uzawa-type smoother. While these are certainly not the only such examples in the literature, the majority of the work we will cite in subsequent chapters utilize the multigrid v-cycle as a preconditioner within a Krylov solve instead of a solver itself.
The reasons for using multigrid as a preconditioner are that, while multigrid theory suggests multigrid is guaranteed to be a solver for elliptic PDEs, nothing is generally said about the speed of convergence. On the other hand, Krylov methods have been shown (some through theory, mostly through experiment) to be very efficient solvers when paired with adequate preconditioners. We then expect that, given multigrid’s approximation properties independent of $h$, as a preconditioner we should see fast convergence of Krylov methods, and convergence which does not slow as we refine our space. This is supported by experiments in the literature [65, 61] and from results in the following chapters of this thesis.

2.6 Matrix-free Methods

From the previous sections, it is clear that solving discrete Finite Element systems rely heavily on the computation of matrix-vector products with sparse matrices. These matrix-vector products tend to dominate the total amount of work required for the average finite element program [72], and, as these matrices can be quite large and do not fit into the cache of a modern machine, accessing the data from RAM has becomes a major bottleneck in finite element computations as we start to spend more time accessing entries from memory than actually computing with the entries [75]. This is the major motivation for so-called matrix-free methods which define a matrix-vector product as a linear operator that recomputes the entries of a matrix each time it is applied, as opposed to storing the entire matrix and accessing its entries.

Given the way we construct the matrix entries in (2.19) as a sum over cells, we can write a matrix-vector product with $A$ as

$$v = Au = \sum_{T \in T_h} P_{\text{local-global}}^T A_K P_{\text{local-global}}^u u_k$$

(2.59)

where $P_{\text{local-global}}$ is a projection of a global vector defined over the entire $T_h$ to a single cell $K$, and $A_K$ and $u_K$ are the matrix and vector restricted to the cell $K$. When computing a matrix-vector product, matrix-free methods recompute each local matrix $A_K$ ad-hoc, thereby increasing the amount of computation required but eliminating the need to store/access matrix entries from the main memory. Since we are using hexahedral meshes, and thereby have a natural tensor structure.
of both quadrature and shape functions, any matrix-free matrix-vector product can utilize summation-factorization techniques (described in [72]) to reduce the complexity of cell matrix application. While this is also possible on a tetrahedral mesh, there is typically a much higher cost associated with these computations [67]. Lastly, there is a natural utilization of the level of vectorization of the processing unit on which the program is being run, namely, vectorization over cells. This is due to the fact that each cell matrix application is essentially floating point operation and small matrix-vector operations, independent of the other cells, which can be computed simultaneously. Codes can make use of libraries like BLAS [21] which provides very fast, vectorized matrix-vector products for small and dense matrices. All of this leads to methods that are considerably faster for the evaluation of matrix-vector products than with assembled matrices (which often use no vectorization), especially for higher order finite elements ([72] shows matrix-free is consistently faster than matrix-based for $Q_2$ and higher).

The other benefit is, obviously, that far less memory is being used. The storage of the system matrix dominates the total storage of a matrix-based finite element computation (see Table 4.6), and therefore removing this storage requirement enables computations on significantly larger meshes. Thus, with the save in both time and memory, matrix-free computations are almost a necessity for extreme scale computations.

These methods, however, tend to be more difficult to implement than standard matrix-based finite elements, as new matrix-free operators must be built for each different operator used in a problem. We are also restricted to using routines that only require matrix-vector products, and in the case of smoothers within a multigrid v-cycle, we can no longer use smoothers which require the access of individual matrix entries (like Gauss-Seidel). We can, however, easily recover the diagonal of any matrix-free operator by applying the matrix to the unit vectors, and therefore smoothers like Jacobi and Chebyshev are still available.

2.7 Parallel Computing

As we want to solve large scale problems, we require that the algorithms that we develop must be written inside a parallel framework. In this thesis we will consider two types of parallelization:

1. distributed work among multiple processing units with message passing between processors,
2. and vectorization of data manipulation contained on a single processor.

Our goal is to distribute the work of an adaptive, matrix-free geometric multigrid v-cycle among processors such that each processor has roughly the same amount of work, can execute their portion of the program simultaneously and with minimal communication (most importantly, minimal all-to-all communication), avoids the duplication of large amounts of data among every processor, and utilizes the vectorization as much as possible. For testing a parallel algorithms, we will consider the following:

1. **strong scaling**: solving a problem with constant size and varying processor count, and

2. **weak scaling**: solving a problem with varying size and processor count, where the ratio between the two is held constant.

Strong scaling can be seen as a test for the limit of scaling in terms of unknowns per processor count, namely, how many processors are still beneficial to use for a certain problem size. Weak scaling is a test for efficiency of an algorithm assuming adequate resources are available, namely, how large of a problem can be solved efficiently given enough processors. If an algorithm exhibits poor strong scaling, it will not necessarily exhibit poor weak scaling, whereas poor weak scaling tends to predict poor strong scaling. This is because strong scaling test the limits of parallel performance, namely, how many how small can the unknowns per processor be before the implementation loses efficiency. As we will see, strong scaling tend to follow a pattern where the efficiency is based on unknowns per processor, and therefore the loss in efficiency is hidden as we keep this value constant for weak scaling.

For the purpose of scaling, it is important to parallelize as much of the code as possible, as if even a small fraction of the code is serial (executed on only one processor, or duplicating work on each processor), the strong scaling can be greatly inhibited. Amdahl’s Law [4] gives that the theoretical scaling speed-up of any program is limited by the percentage of that program which is serial. It is expressed in the formula

$$\text{speed-up} = \frac{1}{1 - P + P/n},$$

where $P$ is the percentage of the code which can be written in parallel. From this, $1/$speed-up gives the theoretical strong scaling of the program, and, from Figure 2.10, we see that a code where just
1% of the program must be serial sees virtually no scaling past 512 processors.

To write parallel programs, we will use various libraries which set up a communication framework between each of the processors using MPI (Message Passing Interface) [43], a software which interfaces between the processing units on a machine and offers functionality to send and receive data from one processor to another. Three of the most common implementations are OpenMPI [45], IntelMPI, and MPICH [51], and it varies from machine to machine which libraries, and which versions, are available. With any of these libraries, however, the user can write C++ code which is run by all processors, adding in synchronization points and processor specific code using the commands provided by MPI.

2.7.1 Parallel Computing for Finite Elements

In the Finite Element context, a partition of work is usually found in the creation of a subdomain of cells within the larger domain, which is then assigned to a single processor. This processor is then responsible for updating all necessary information for the unknowns contained within its subdomain, with special care taken for the unknowns that lie on the interface between processors [58]. Libraries like P4EST [32] and METIS [68] offer partitioning algorithms for the active mesh aimed at a fair distribution of work among processors with a minimal interface, where the mesh is never duplicated on any one processor. For the multigrid hierarchy, a partitioning scheme is given and extensively tested in Chapter 3.

Once the mesh is distributed, there exist libraries like PETSc [11], Trilinos [57], and
deal.II [12] which offer parallel linear algebra objects such as matrices and vectors, as well as operations on these objects. Since the linear system \( A \) is constructed over cells, each processor can fill in their own contributions to these objects with special care taken for unknowns which lie on an interface. For matrix-free computations, each matrix-free operator can be written such that processors add in the contribution to a matrix-vector product from their owned cells. Then, using solvers and preconditions which rely matrix-vector products, one can solve a fully parallel finite element system.

2.8 Software

A large part of the contributing work of this thesis lies in the implementation of the methods discussed throughout into the open source, C++ libraries deal.II[5], and the open-source mantle convection application code ASPECT [13]. Each code is widely used in their respective communities and the implementations provided are all aimed (i) helping the current users solve problems more efficiently, and (ii) attracting new users to the codes, facilitating an expanding community. Here will be a brief description of each library and a summary of the contributions related to this thesis.

2.8.1 deal.II Library

deal.II[5], which stands for Differential Equations for Analysis in Library, is an open source, C++ finite element library which provides tools for finding numerical solutions to PDE systems by way of the finite element method. The library offers basic serial implementations as well as scalable, parallel codes. The following is a basic overview of various features of the library; for greater detail see [14] for an overview of the basics of the library and [12] for a discussion on adaptivity and massively parallel computations.

- Mesh generation: The deal.II Triangulation class is central to the finite element code. It provides the functionality to represent and iterate through cells of a discretized mesh. These meshes can come from one of many provided functions in deal.II or read in from an external source. The cells described by this class are line segments in 1D, quadrilaterals in 2D, and hexahedra in 3D. The representation of the mesh uses a tree structure where each cell comes from the refinement of a coarser cell, for example, Figure 2.11 describes the unit square globally refined once and adaptively refined in the top left quadrant, along with its tree representation.
Figure 2.11: Representation of a deal.II mesh. The cell label $\ell.T$ is used for cell $T$ on level $\ell$.

Figure 2.12: Example of a space-filling curve as used by p4est.

This tree structure will become important when discussing the parallel distribution of both the active mesh (the mesh we would like to compute on) and the level meshes (the collection of meshes that describe how we have reached the active mesh).

There are three types of Triangulation we will discuss in deal.II for parallel computations:

- Distributed: Triangulation which uses the mesh partitioning code p4est [32] to create a balanced partition of the active mesh. This is accomplished through the use of a space-filling curve which iterates through the cells in the tree, assigning each to a processor in order. Figure 2.12 gives an example of a space-filling curve and the resulting distribution for 3 total processors. The result is a scalable partition of the active mesh where the entire mesh is never duplicated on a single processor. Each processor only has knowledge of their own cells and one layer of cells around their cells known as the ghost layer. Figure 2.13 gives a a partition for a much larger mesh.
Figure 2.13: Partition among 3 processors of an adaptive mesh. Processor 0 (left) is red, processor 1 (center) is blue, and processor 2 (right) is green. One layer of ghost cells included. Cells without coloring are called artificial cells and not representative of the actual mesh discretization, only the minimal information needed by each processor to reconstruct its active cells.

- Shared: Triangulation which stores the entire mesh on each processor. This triangulation allows the user to have more control over the distribution, using, for example, a program like METIS [68] to distribute the mesh as opposed to p4est.

- Fully Distributed: Triangulation which stores a fully distributed version of the coarse mesh. This is useful for meshes created by an external mesh generator, where the coarse mesh is distributed in parallel and too large to duplicate on each processor.

- Finite element \( \{ \hat{T}, Q_k, \Sigma \} \): deal.II provides classes which describe the finite element space \( Q_k \) and DG\(Q_k \) (most common), as well as other types of finite elements on quadrilateral meshes, and a class called DoFHandler for the enumerating, partitioning, and storing of information related to degrees of freedom. There is functionality for computing shape function, gradients, Hessians, etc. and necessary mapping to and from the reference element.

It is of note that, as mention in Section 2.2.2.1, deal.II allows for meshes which contain hanging nodes (see Figure 2.1). This requires additional functionality, namely we must constrain the values at these nodes to the conforming nodes surrounding them. The hanging node constraints are based on a polynomial interpolation of the degree of freedom to the surrounding degrees of freedom on the mesh.

There are deal.II defined sparse matrix and vector types for assembling the associated linear system for serial computations, or, for parallel computations, deal.II provides access to the scalable libraries Trilinos and PETSc for matrix and vector types. There is also a deal.II
distributed vector type for parallel, matrix-free computations.

- **Solvers:** *deal.II* provides implementations of various Krylov subspace solvers (CG, GMRES, FGMRES, BiCGStab, MINRES, to name a few) as well as direct solvers for sparse matrices. The Krylov solvers are templated on both vector and matrix type and can therefore be executed with both matrix-free and matrix-based computations, as well as in parallel.

- **Preconditioners:** Each Krylov method inside *deal.II* takes in an argument for the preconditioner. A few preconditioners offered are the identity preconditioner (i.e., no preconditioning), SOR and SSOR, Jacobi, Chebyshev, GMG, and AMG (Trilinos ML [46]).

The implementation of the serial GMG method is based on the discussion in Section 2.5 and the papers cited, and the parallel GMG method at the base of the discussion in Chapter 3 and [36] (see Section 2.8.1.1 for a list of my contributions to parallel GMG inside *deal.II*). There, we offer extensive testing for the distribution and parallel scalability of the method. From each subsequent mesh refinement, *deal.II* stores the new mesh and any level in the hierarchy can be accessed. There is also a `MGConstrainedDoFs` class that, much like the `DoFHandler` class, stores information about the ways in which the degrees of freedom are constrained on the various levels. Also provided are transfer classes between the levels inside the `v_cycle()` function: `MGTransferPrebuilt` constructs transfer matrices based on the *deal.II* `SparseMatrix` class, whereas `MGTransferMatrixFree` contains functions that either prolongate to the next finer space or restrict to the next coarser space using information about the mesh, and not by building a physical matrix.

- **Matrix-free:** *deal.II* offers support for matrix-free computations based on the work of Kronbichler and Kromann [72]. The interface allows the user to define the operation of the cell matrix $A_k$ (see Section 2.6) for the specific application and a `cell_loop()` function which loops through the cells of the mesh and applies the local contribution. The implementation takes into account generic constraints on the degrees of freedom defined by the user, takes advantage of the amount of vectorization allowed by the users machine (assuming *deal.II* has been compiled with this information), and offers support for both continuous and discontinuous Galerkin finite elements. The matrix-free implementation also provides support for multilevel computations where the user can define the level cell matrix and level constraint
objects. This functionality is used extensively in all remaining chapters of this thesis, as the majority of GMG results are using a matrix-free framework.

- Post Processing: **deal.II** provides error estimators useful for adaptive refinement, helper functions for creating .vtu and .vtk output which can be used in software such as Paraview, as well as other tools useful for post processing.

### 2.8.1.1 List of My Contributions to **deal.II**

Throughout my PhD I have contributed roughly 21,500 lines of code to the **deal.II** library. Here are a list of my contributions:

- For the shared triangulation discussed above, I have implemented the **p4est** algorithm for distributing the mesh, as well as an interface for a user-defined partition. I also added functionality related to level mesh partitioning to make multigrid computations possible on this type of triangulation.

- For visualization, I have implemented a function `write_mesh_per_processor_as_vtu()` for visualizing the level mesh hierarchy of a given triangulation and its partition in a .vtu format, as well as a function which allows for the visualization of a sparsity pattern in .svg format.

- In an adaptive, GMG, finite element code, the user needs to create a refinement edge interface matrix on each level which contains all the entries of the system matrix that also lie on a refinement edge. Before we would use the sparsity pattern from the system matrix, which contain many more non-zero entries than required. I implemented functionality for creating a sparsity pattern which only contained entries where the interface matrix would actual require a non-zero value, saving memory in matrix-based computations.

- I added a function which computes the workload imbalance associated with a GMG hierarchy as discuses in Section 3.3 and [36].

- For the computations in Chapter 5, I added an IDR(s) solver to the library.

- Before, the only boundary conditions supported for GMG computations were Dirichlet or zero-Neumann boundary conditions. I implemented no-normal-flux (or free-slip) boundary conditions of the form

\[ u \cdot n = 0 \]
for square or box domains. I also implemented an interface for users to define custom constraints to be applied on each level, allowing for GMG computations with any boundary conditions. The implementation of these user defined constraints amounts to distributing the constraints to the solution vector directly before prolongation to a finer level. This functionality is directly used in ASPECT where many of the applications use free-slip boundary conditions on both box and shell domains.

- I added the Step-63 [35] tutorial which solves the advection-diffusion PDE (2.3) with a GMG method based on Schwarz smoothing. The results are discussed in Section 3.4.3. For this tutorial, I implemented a degree of freedom renumbering algorithm which gives a random ordering of indices for the degrees of freedom on each level.

- I have contributed to the Step-50 tutorial which aims at solving the Poisson equation, demonstrating convergence and parallel scaling of the GMG method in deal.II. This tutorial is still a work in progress, and an incomplete version is available on the development version of deal.II.

- In an effort to limit new contributions from breaking other parts of the code, deal.II has a large test suite that is run for each new addition to the library. Therefore, any addition to the code must be accompanied by its own tests, ensuring that no new addition in the future will break the functionality being implemented. In all, I have added 24 individual tests to the test suite.

2.8.2 ASPECT Code

Written on top of the deal.II library, ASPECT[13], which stands for Advanced Solver for Problems in Earth’s ConvecTion aims to provide state-of-the-art algorithms for solving problems related to convection in the earth’s mantle [71, 56]. From [71], the code is aimed at solving the Boussinesq model for temperature $T$, velocity $u$, and pressure $p$ of a fluid, given by

\begin{align}
-\nabla \cdot (2\mu \varepsilon(u)) + \nabla p &= \rho(T)g \\
\nabla \cdot u &= h \\
\frac{\partial T}{\partial t} + u \cdot \nabla T - \nabla \cdot (\kappa \nabla T) &= \gamma
\end{align}

(2.60)
where $\varepsilon(u) = \frac{1}{2} (\nabla u + (\nabla u)^T)$ is the strain-rate tensor, $\mu$ is the viscosity of the fluid, and $\kappa, \gamma,$ and $g$ are the thermal conductivity, heat sources, and gravity vector, respectively. The method for solving (2.60) is given in great detail in [71]. In short, the equations are discretized in time, and a semi-implicit BDF-2 scheme is applied. For each time step, an advection-diffusion, temperature solve and a steady-state Stokes solve are required. The size of the Stokes system dominates the overall size of the problem, and therefore often a majority of the time is spent on these solves. Chapter 4 will be focused on applying the adaptive, parallel, matrix-free GMG preconditioner described in Chapter 3 to the computation of the Stokes equations inside ASPECT, and comparing against the AMG-based preconditioner currently used in the code.

2.8.2.1 List of My Contributions to ASPECT

My primary contribution to ASPECT is providing the implementation for solving the Stokes system with a matrix-free GMG method. This required roughly 4,000 lines of code and is currently available for incompressible or compressible flow, averaged coefficients, no-normal-flux or Dirichlet boundary conditions, and $Q_{k+1}/Q_k$, $k \geq 1$ finite elements. In Chapter 4 we demonstrate results using the the latest version of ASPECT containing these features. Much like deal.II, ASPECT has a test suite for ensuring consistency. For the additions listed above, I’ve added 5 tests to the ASPECT test suite.
Chapter 3

Parallel, Adaptive Geometric Multigrid

Multigrid methods are solvers for elliptic partial differential equations with optimal complexity in the number of total variables, but optimal performance in a massively parallel environment depends on more than complexity alone. Sufficiently many concurrent operations must allow utilization of a sufficiently large part of the system, and it is not clear a priori if multigrid methods with their hierarchy of coarse meshes and synchronization due to grid transfer will be efficient on such systems. In this chapter we present an overview of the Geometric Multigrid method and give data structures, algorithms, and show results for an efficient, massively parallel implementation.

This chapter is a combination of [36] and [35]. [36] is a publication titled “A Flexible, Parallel, Adaptive Geometric Multigrid method for FEM” with co-authors Timo Heister, Guido Kanschat, and Martin Kronbichler, submitted to ACM Transactions on Mathematical Software (TOMS) journal. This is one of the top journals in mathematical software, and we have received very favorable reviews. The expectation is for publishing before the end of the calendar year. Much of the work in my Master’s project [33] was included in the publication. [35] is a deal.II tutorial program written by Thomas C. Clevenger and Timo Heister (with feedback and corrections given by Wolfgang Bangerth). The program is aimed at solving the advection-diffusion equations using a geometric multigrid preconditioners with additive and multiplicative Schwarz smoothing.
3.1 Review of Current Literature

Publications describing adaptive, parallel geometric multigrid methods for unstructured meshes are not very common. One of the first optimal complexity adaptive, parallel geometric multigrid methods for unstructured grids was given in [16] (1996) for 2D meshes, and the first such method for 3D mesh was given in [76] (2001). In 2006, the authors of [17] gave a summary of such a method for 2D and 3D hexahedral meshes, but no parallel scaling was demonstrated. Also in 2006, the authors of [20] gave a parallel multigrid method for unstructured, globally refined, tetrahedral meshes in 2D and 3D. In 2016, this multigrid solver was used in [49] to solve large Stokes systems with up to 2.2 trillion unknowns on 786,432 threads (this solver is discussed in Section 4.1).

To our knowledge, the first and only massively parallel geometric multigrid method for unstructured, adaptively refined meshes was given in [98] (2012). There, the authors described a geometric multigrid method on adaptively refined, hexahedral meshes using global coarsening to define the level mesh hierarchy. For global coarsening, each multigrid level mesh covers the entire computational domain, and the mesh on each level contains hanging nodes. Each level mesh is created by coarsening each cell, and the coarsest level is the level in which not every cell can be coarsened. As an optimization, each level is then repartitioned with a smaller number of processors if there are not at least 1,000 cells per processor. This saves in communication overhead for smaller meshes where computation time is much lower. Using a matrix-free setup, they demonstrate strong scaling of a Poisson solve with up to 124 million cells and 131,072 processors, showing a favorable comparison to AMG (Trilinos ML [46]) in terms of setup and solve time as well as scalability. They then demonstrate weak scaling up to 56 billion degrees of freedom on up to 262,144 processors, again showing a favorable comparison to AMG. Lastly, they demonstrate weak scaling of a hybrid GMG method with an AMG coarse solve for up to 105 billion cells on 262,144 processors. These results are impressive, and offer an interesting alternative to the local smoothing described in [65, 61] and summarized in Section 2.5.

The submitted publication [36] is the only such instance for adaptive, parallel geometric multigrid methods on unstructured meshes using local smoothing to define the multigrid hierarchy. We find this important since highly parallel, adaptive multigrid methods for elliptic PDEs on unstructured meshes are often components to state-of-the-art solvers for many different PDEs. As we will see in Section 4.1, the largest scale runs in mantle convection applications are done in [90] which
use the global coarsening method in [98] for preconditioning within Stokes solves of up to 600 billion unknowns.

3.2 Parallelization of Geometric Multigrid

In this section, we discuss the design of a parallel multigrid algorithm which scales well with the number of processors, but is still algebraically equivalent to the weathered sequential algorithm. This way, we emphasize data and communication structures alone and do not have to worry about mathematical properties. Our algorithm is also synchronized between applications of residual, smoothing, grid transfer operators, and coarse grid solvers. Hence, our focus lies in the parallel implementation of these operators.

The abstraction to design parallel data structures to provide an approach that is equivalent to a serial algorithm for linear algebra and Finite Element computations is well-known. Libraries like PETSc [11, 10] and Trilinos [57] have provided linear algebra data structures (vectors, sparse matrices) and algorithms (iterative solvers) with this abstraction for a long time. Up to a point, this isolates the user (for example finite element library implementors) from having to directly interface with the underlying parallel computing framework. The abstraction is of course not perfect, because operations like finite element assembly need to be partitioned between the processors. Nevertheless, this allows the design of parallel algorithms on a higher level, like it is done in deal.II for example, see [12].

The workload is typically distributed by first partitioning the cells of the computation using graph based partitioner or using space-filling curves (like METIS [68], Zoltan [22], or p4est [32]). This partitioning can be used to distribute cell-based work, like matrix or residual assembly, and can be used to generate a partitioning of degrees of freedom that is needed for the row-wise partitioning of linear algebra objects (vectors, matrices), which requires a rule to decide on the ownership of degrees of freedom on the interface between processor boundaries of the cells. The only difficulty is the correct assignment and communication of ghost cells and ghost indices, while the communication for matrix-vector products and finite element assembly of foreign entities only involves neighboring processors and is typically provided by the linear algebra libraries.

Here, we will follow the same approach for the partitioning of cells and degrees of freedom on each level of the multigrid hierarchy: after partitioning of all cells strictly on level \( l \) in some
way, we use this to partition the degrees of freedom accordingly. Like above, it is advantageous for large computations if only the parts of the mesh that is relevant for the current processor are stored locally.

There are different options for partitioning cells on each multigrid level. For the algorithms discussed here, it can be considered arbitrary and independent on the partitioning of the active cells. We will discuss different strategies and the approach we take in Section 3.2.1.

While knowledge about the whole mesh is not required, we need ghost neighbors on each level\(^1\) and information about parents/siblings for transfer operations. This allows us to compute and exchange the necessary information about degrees of freedom for smoothing and grid transfer. In our scheme, this ownership information is readily available without global communication.

To summarize, prolongation and restriction are conceptually multiplication of distributed vectors of unknowns with a distributed, rectangular transfer matrix and, as such, equivalent to the serial transfer. Known algorithms for sparse matrix-vector scale well in parallel. In practice, we implement the transfer without building transfer matrices by defining these operations in a matrix-free framework (see Section 2.6).

Smoothers are conceptually often local operations on individual degrees of freedom or cells. Additive smoothers (Jacobi, additive Schwarz, etc.) can be run in parallel on all processors and are still equivalent to the serial method. Other sequential smoothers (Gauss-Seidel, multiplicative Schwarz, etc.) can not be used immediately.

There are several options for coarse solvers. First, if the problem is reduced to a very small number of cells and processors, the runtime is negligible and (parallel) direct solvers can be applied. In other cases, when the coarse mesh has still a large number of degrees of freedom, switching to algebraic multigrid can be advantageous and is common in the literature [99]. Coarsening geometrically to very few cells might not be ideal from the performance standpoint, in which case stopping the coarsening at a specific level and switching to algebraic multigrid can be a useful approach.

### 3.2.1 Partitioning Strategy for Mesh Hierarchy

When partitioning the cells on each level of the multigrid hierarchy, there are several conflicting goals to keep in mind:

1. Minimize communication for transfer operations between multigrid levels.

\[^1\]This can refer to cells on different levels in adaptive computations.
2. Fair work balance on each level (same number of cells per processor).

3. Minimize interface between processor boundaries on each level (minimizes communication in smoother applications).

4. Minimize required additional storage for the mesh hierarchy if local cells have little overlap between levels.

One choice is to ignore (1) and do an independent partition on each level, as it is proposed for in [98] for instance. Note that the multigrid method there is based on global coarsening instead of local smoothing, so each level is an adaptively refined mesh that needs to be partitioned. This satisfies (2) but requires duplicate storage, violating (4). Note that (1) is satisfied for mostly globally refined meshes, but duplicate storage (4) is still required.

Another option is to ignore (2) and partition based on the partitioning of the active cells to minimize communication cost and storage requirements (goal (1) and (4)). This is the approach we decided to use here. We will see that we satisfy goal (2) for mostly globally refined meshes and that we can quantify the partitioning efficiency (see Section 3.2.2).

Note that both approaches behave similarly for globally refined meshes, while goals (1) and (2) are conflicting for an adaptive scheme. Finally, note that (2) is only desired when assuming sequential operations on each level, as the multigrid algorithm suggests, but one could design a parallel method that does not require synchronization on each level.

In the following, we will partition the multigrid cells by the “first-child rule” as follows: First, distribute the active cells using a space filling curve (we use p4est [32] as described in [12]). Second, for each cell in the hierarchy, recursively assign the parent of a cell to the owner of the first child cell.

For an example with seven cells see Figure 3.1. This approach has the following consequences:

1. The cells and their parents are already present on each processor and the ownership of parents is known without any communication. This means the partitioning of the multigrid hierarchy can be done without communication.

2. No duplicated storage for the mesh is needed as all parent cells are already stored locally (goal (4)).
3. Transfer operations are local and require only little transfer at processor boundaries (goal (1)).

4. Processors drop out automatically on coarser levels, which is desired.

### 3.2.2 Partitioning efficiency model

Our model for the complexity of the partitioned workload, in short parallel complexity, is based on the assumption that parallelization is completely achieved by MPI ranks and that within each rank all operations are sequential. Below, we develop a complexity model based on this assumption, estimating the parallel complexity of our algorithm in terms of mesh cells per level.

Let $N_\ell$ be the number of cells on level $\ell$ and $N_{\ell,p}$ of the subset owned by proc $p$. We assume that the workload for each cell is equal, such that $N_{\ell,p}$ is proportional to the total amount of work a processor has to invest on level $\ell$. Obviously, the optimal parallel complexity is

$$W_{opt} = \frac{1}{n_p} \left[ W_0 + \sum_{\ell=1}^{L} \sum_p N_{\ell,p} \right] = \frac{1}{n_p} \left[ W_0 + \sum_{\ell} N_{\ell} \right].$$

Here, $W_0$ is the complexity of the coarse grid solver, which may be expected to be different from the smoother. In particular, we may not be able to distribute the load over all processors and thus the factor $1/n_p$ may not apply.

This calculation is based on perfect of work and neglects communication overhead. In particular, it is not achievable if grid transfers are synchronized, as in our implementation. In this case, we can only distribute the work on each level, such that we are bound from below on each level by

$$W_{\ell, opt} = \left\lfloor \frac{1}{n_p} N_{\ell} \right\rfloor,$$

where $\lceil n \rceil$ is the smallest integer greater or equal to $n$. Therefore, the best achievable work time with syncing between levels is

$$W_{sync} = W_0 + \sum_{\ell=1}^{L} W_{\ell, opt}.$$
On the other hand, with imperfect distribution of work, the limiting effort on each level is

\[ W_\ell = \max_p N_{\ell,p} , \]

and the total parallel complexity is \( W \) and partitioning efficiency due to imbalance against a hypothetical optimal partitioning are given by

\[ W = W_0 + \sum_{\ell=1}^{L} W_\ell, \quad E = \frac{W_{\text{opt}}}{W}. \]   \hspace{1cm} (3.1) \]

We give an example for these estimates for the mesh hierarchy displayed in Figure 3.1. It consists of 7 active cells obtained by successive refinement of a single coarse cell. The partitioning is done for three processors. The ownership of the active cells is determined by p4est using a space-filling curve (Z-curve, dashed line on the left picture) or depth-first traversal (from left to right) in the tree representation depicted in the middle. The ownership of cells in the multigrid hierarchy (round circles in the tree) is determined by copying the active ownership and then applying the “first-child rule” recursively. For example, the parent of the four smallest cells on level 2 is red (#0) because the first (bottom-left) child also belongs to processor #0 (red). One result of this partitioning is that processors drop out on coarser levels automatically. Here, processor #1 (green) recuses itself on level 1 and only processor #0 (red) remains on the coarsest level (here a single cell). The coarsest mesh is not necessarily completely owned by processor #0 if it consists of more than
The optimal parallel complexity is simply the number of all cells divided by the number of processors, hence $W_{opt} = \frac{9}{3} = 3$. On the other hand, assuming the coarse grid solver has the same complexity as the work load per cell on higher levels,

$$W_{sync} = W_{0, opt} + W_{1, opt} + W_{2, opt} = \left\lceil \frac{1}{3} \right\rceil + \left\lceil \frac{4}{3} \right\rceil + \left\lceil \frac{4}{3} \right\rceil = 5.$$ 

Comparing to

$$W = \sum_{\ell} W_{\ell} = 1 + 3 + 3 = 7,$$

we obtain $\mathbb{E} = 3/7$. In other words, our model predicts a 133\% slower multigrid cycle compared to optimal partitioning. Compared to the optimal partitioning with synchronization between levels, is 40\%. The slowdown with respect to $W_{sync}$ is due to the non-optimal partitioning on level 1, where processor # 0 (red) works on three cells while the other processors have to wait. An optimal partitioning would only require operating on two cells sequentially on that level. Compared to $W_{opt}$, we simply do not have enough cells to keep three processors busy.

This example suggests that the efficiency of the algorithm depends significantly on the base of comparison, $W_{sync}$ or $W_{opt}$. In fact, a closer inspection of the definitions reveals that they only differ by rounding up the load on each level to the next multiple of $n_p$, a difference which drops below 1\% as soon as we have 100 cells on each processor. Below, we only use $W_{opt}$ when we assess the efficiency of our mesh hierarchy distribution in order to avoid the accusation of cheating.

3.3 Experimental Study of Mesh Partitioning

Making general conclusions about the partitioning efficiency is difficult as it depends on the number of processors, coarse mesh, and refinement done. Instead, we experimentally determine the efficiency for several different test cases.

Our tests are based on sequences of meshes based on the following construction and refinement schemes, which are all based on a coarse mesh consisting of a single cell. By $L$, we denote the level of the finest cells as defined in Section 2.5.1. Their generation algorithms are as follows:
“global”: Uniform refinement of the coarse mesh $L$ times, obtaining a active mesh of $4^L$ cells in two dimensions.

“circle”: $L$ times refinement of all mesh cells with at least one point inside the circle of radius $1/4\pi$ around the origin.

“quadrant”: After one global refinement of the coarse mesh, refine the cell in the negative quadrant $L-1$ times.

“annulus”: After $L-3$ global refinements, add the steps:

1. Refine all cells whose center lies in the circle (sphere for 3D) of radius 0.55.
2. Refine all cells in the shell between radius 0.3 and 0.43.
3. Refine all cells in the shell between radius 0.335 and 0.39.

All these procedures are completed by a closure after each refinement step, which ensures mesh conformity in the sense that

1. when crossing a face (boundary of co-dimension one), the level of a cell changes at most by one,
2. if a degree of freedom is located on the boundary of several cells, the levels of these cells differ at most by one.

These conditions are imposed in the deal.II library for practical reasons, because they simplify several aspects of the implementation. They can be relaxed at a high price of software complexity. Figure 3.2 shows the resulting meshes for different number of refinements.

Figure 3.3 gives the partitioning efficiency $E$ for a varying number of processor counts and problem sizes on each of the mesh refinement types listed in this section. If a certain problem is unreasonably large for a given processor count a dashed line is used in the plot. The cutoff for what is a reasonable problem size would of course be dependent on many factors (e.g. finite element degree, components per cell, etc.), but here it is chosen to be 1,000 active cells per processor since no actual computation is being performed.

Unsurprisingly, for a globally refined mesh we are at 100% efficiency (this also holds in 3D, not shown). This is due to the fact that processor counts are multiples of $2^d$, which means partitioning on each level stays the same.
(a) Mesh sequence “circle”: successive refinement of all cells intersecting the ball of radius $1/4\pi$ around the center. Leaf mesh at 5, 6 and, 9 levels, resp.

(b) Mesh sequence “quadrant”: refinement of all cells in the lower left quadrant. Leaf mesh at 3, 4 and, 8 levels, resp.

(c) Mesh sequence “annulus”: see text for algorithm. Leaf meshes with finest cells on levels 7, 8, and 9, resp.

Figure 3.2: Visualization of the different mesh refinement sequences.
Figure 3.3: Partitioning efficiency for various meshes. Dashed lines indicate less than 1,000 active cells per processor. Top: global, circle, quadrant refinement. Bottom: annulus refinement in two (left) and three (right) dimensions.
Second, with exception of the “circle” refinement, each refinement scheme gives roughly the same efficiency no matter the problem size, with “circle” refinement following roughly the same shape and actually having slightly better efficiency for larger problems. This suggests that, given a reasonable problem size per processor, the imbalance is not negatively dependent on problem size. Finally it is worth noting that the number of processors in the distribution appears to have no effect on efficiency past a certain point as each mesh refinement plot “levels off”, the exception again being the “circle” refinement (however, it gives a higher efficiency than any other adaptively refined mesh). All of this information together suggests that, given a large enough cell per processor count, the imbalance of this distribution is primarily dependent on the type of mesh refinement refinement scheme used and dependent on the number of processors only to a certain “leveling off” point. In all cases, the efficiency stays above 30% compared to the optimal workload.

3.3.1 Performance model with communication

The second factor determining the performance of parallel algorithms next to load balancing discussed above is communication overhead. Communication is not only much slower than computation, it also consumes more energy, because electrical charges must be transported over fairly long distances. We introduced the first-child rule with the express purpose to reduce communication overhead. In this section, we set out to demonstrate that this goal was achieved.

Since communication is much slower than computation, we design the data layout such that the amount of communicated data is reduced. On a given level, this is achieved by the locality of the \( z \)-curve. Cells with their indices close to each other are on average located close to each other geometrically. Thus, the number of cells on a processor grows faster than the number of cells on the boundary of its subdomain in weak scaling, and it is typically much smaller in absolute figures. This approach has been a standard for many years now.

We chose the first child rule for distributing lower levels in order to achieve a similar goal for grid transfer operations. If most children are on the same processor as their parents, the amount of communicated data is also much lower than the total amount of data processed. In Figure 3.4 we show that the number of “ghost children” is indeed very small compared to the total number of children. And while these numbers are rising with the number of processors, in the worst case observed less than 1% of the cells require communication. Additionally, the total communication volume seems to grow slower than the number of processors involved in the communication. Thus,
<table>
<thead>
<tr>
<th>$1/h$</th>
<th>DoFs</th>
<th>CG iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>16641</td>
<td>7</td>
</tr>
<tr>
<td>128</td>
<td>66049</td>
<td>8</td>
</tr>
<tr>
<td>256</td>
<td>263169</td>
<td>11</td>
</tr>
<tr>
<td>512</td>
<td>1050625</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 3.1: CG iterations needed to reduce the residual by $1e8$. 

while grid transfer and matrix-vector products are global operations, the communication overhead with our proposed strategy is low.

### 3.4 Numerical Results

The algorithm described here has been implemented in deal.II. The partitioning of the adaptively refined meshes uses p4est[32]. The implementation with sparse matrices uses Trilinos [57], while the matrix-free implementation is based on data distribution algorithms built into deal.II.

#### 3.4.1 Poisson Problem

Consider the Poisson equation with homogeneous Dirichlet boundary condition on the domain $\Omega = [-1, 1]^2$

$$-\nabla \cdot (\mu \nabla u) = 1 \quad \text{on } \Omega$$

$$u = 0 \quad \text{on } \partial \Omega$$

with a non-constant viscosity defined by

$$\mu(x) = \begin{cases} 
1e3 & \text{if } \|x\| > 0.35 \\
1 & \text{else}
\end{cases},$$

and discretized with a $Q_1$ element. The smoothing is 4 Jacobi steps with $\omega = 0.9$ damping. Figure 3.5 gives the numerical solution for this test and Table 3.1 gives the iterations needed to reduce the residual by $1e8$. We see that the iteration counts do not increase drastically as the mesh size changes.
Figure 3.4: Communication ratio (communicated number of children over total number) in level transfer

Figure 3.5: Viscosity (left) and solution (right) of the inclusion test
3.4.1.1 Scaling on SuperMUC

Next, consider the constant-coefficient Laplacian on a cube, discretized with $Q_2$ elements, and compare the runtime on a uniform mesh against an adaptively refined case with the annulus refinement. The adaptive mesh is set up such that the number of cells matches with the number of cells in the uniform case within 2%. The computations are run on phase 1 of SuperMUC, providing nodes with $2 \times 8$ cores of Intel Xeon E5-2680 (Sandy Bridge), connected via an Infiniband FDR10 fabric. For pre- and post-smoothing, a Chebyshev iteration of the Jacobi method with Chebyshev degree five, i.e., five matrix-vector products, is selected (see [2]). The parameters of the Chebyshev polynomial are set to damp contributions in the eigenvalue range $[0.08\lambda_{\text{max},\ell}, 1.2\lambda_{\text{max},\ell}]$ on each level $\ell > 0$. The estimate $\bar{\lambda}_{\text{max},\ell}$ of the largest eigenvalue of the matrix $A_\ell$ is computed by a conjugate gradient iteration with 10 iterations from an initial vector of zero mean constructed as $(-5.5, -4.5, \ldots, 4.5, 5.5, -5.5, -4.5, \ldots)^T$. As a coarse solver, the Chebyshev iteration is selected with a degree chosen such that a priori error estimate of the Chebyshev iteration ensures a residual reduction by $10^3$, now for the full eigenvalue range of the coarse level matrix determined by a conjugate gradient solution to a relative tolerance in the un-preconditioned residual of $10^{-3}$.

In order to reveal possible communication bottlenecks, we choose a fast node-level implementation by matrix-free evaluation of the matrix-vector products both for level matrices $A_\ell$ and level transfer [75]. The implementation exploits SIMD vectorization over several cells [72] using four-wide registers on the given Intel Xeon processors. To further enhance performance, we run the multigrid v-cycle in single-precision as suggested in [52]. When combined with a correction in double precision after each v-cycle, e.g. within an outer conjugate gradient solver, the reduced precision (which is of high-frequency character and thus easily damped in subsequent cycles) typically does not alter the multigrid convergence [74].

Figures 3.6 and 3.7 list the strong and weak scaling for the runtime of one multigrid v-cycle run as a preconditioner, including all aforementioned communication steps as well as the conversion from double to single precision and vice versa. The presented numbers are consistent over several runs (with standard deviations of at most 2% of the runtime). Each plot contains runtimes for the uniform and adaptive refinement and the optimal $O(N/p)$ scaling (black dashed line) coinciding with the first data point of the uniform refinement graph. The red dashed line shows the model prediction of the new “optimal scaling” of the adaptively refined mesh computed as $1/E$ multiplied...
by the ideal scaling of the uniform computation for the same processor counts. Given the results in Figure 3.3, the 2D annulus refinement suggests an efficiency gap of a factor close to 3 in two dimensions. Figure 3.7(a) confirms that the model assumption is realistic: the uniform refinement is predicted to be 100% efficient and the adaptive refinement is 31% efficient, so we predict a gap of 3.1x in runtime.

The strong scaling limit of the adaptive implementation follows the one of the uniform case, highlighting the efficiency in the communication setup. In three dimensions with 16.9 million cells, scaling of the uniform mesh case starts to flatten for 8,192 MPI ranks, corresponding to 2048 cells or approximately 54,000 unknowns (DoFs) per MPI rank. For this data point, the absolute runtime for the v-cycle is 0.01 seconds. Given the fact that 11 matrix-vector products are performed per level (8 in the smoother, one for the residual, two for the transfer) for a total of 8 levels, this data point corresponds to approximately $1.1 \cdot 10^{-4}$ seconds per matrix-vector product, which is an expected scaling limit of nearest neighbor communication for up to 26 neighbors combined with some local computation on the given architecture. The adaptive case scales at least as well as the uniform one even beyond 8K cores, and also for the other experiments. Partly, this is due to an overlap of different levels, e.g., when some processors do not own any part of a fine level, they can start working on coarser levels as long as the local communication data arrives. Furthermore, the imbalance also leads to more cells on the processors for a given level in relative terms approximately proportional to the inverse efficiency factor $1/E$.

3.4.2 Linear Elasticity with Discontinuous Galerkin Discretization

The second example will be the Lamé–Navier equations of linear elasticity in space dimension $d$, where $V = [H^1_0(\Omega)]^d$, defined in (2.2). Using methods described in Section 2.2, we obtain the weak formulation

$$a(u, v) := \int_{\Omega} \left[ 2\mu \epsilon(u) : \epsilon(v) + \lambda \nabla \cdot u \nabla \cdot v \right] dx = \int_{\Omega} f \cdot v dx =: f(v). \quad (3.2)$$

We solve these equations on a mesh constructed from three cylinders with the Lamé parameters $\lambda = \mu = 1$ according to the setup in Fig. 3.8. The solid is loaded by surface forces on the upper bases of the top two cylinders. It is fixed at the base of the lower cylinder and traction-free on the sides the cylinders. In order to represent the geometry with a high-quality mesh, we use 2808
Figure 3.6: Strong scaling for timing of a matrix-free v-cycle in 2D and 3D for small (left) and large (right) problem size of the annulus refinement. Timings done on the SuperMUC.
Figure 3.7: Weak scaling for timing of a matrix-free v-cycle in 2D and 3D for the annulus refinement. Timings done on the SuperMUC.
hexahedral cells with one global and a series of up to three adaptive refinements based on a residual-based error estimator. Fig. 3.8 shows how the error estimator chooses to refine around the sharp corners with lower solution regularity. The outer layer of cells is represented by a curved cylindrical manifold aligned with the respective cylinder sides. To smoothly relax the curved surface description into a straight-sided one towards the center of the cylinder, we apply a transfinite interpolation [50] over approximately half the cylinder radius. For approximation, we use vector-valued discontinuous $Q_2$ elements of tensor degree 2 and the symmetric interior penalty method with penalty factor 2.0 weighted by the minimum vertex difference in face-normal direction and the factor $2 \cdot 3 = 6$ to account for the inverse estimate on quadratic shape functions.

We solve the elasticity example with a point-Jacobi smoother with 4 pre- and post-smoothing sweeps and relaxation parameter 0.5 on all levels, using a matrix-based implementation based on Trilinos Epetra linear algebra. On the coarse level, there are $227K = 2808 \times 81$ unknowns and 123M nonzero entries in the matrix. We use a direct solver based on the SuperLUDist package.

The systems are then solved by a conjugate gradient solver on the active mesh preconditioned by the proposed geometric multigrid scheme to a relative tolerance of $10^{-6}$, measured in the un-
<table>
<thead>
<tr>
<th>levels</th>
<th>DoFs</th>
<th>CG its</th>
<th>time (s)</th>
<th>time/DoF</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1,819,584</td>
<td>39</td>
<td>71.64</td>
<td>39.4</td>
</tr>
<tr>
<td>3</td>
<td>2,456,325</td>
<td>40</td>
<td>99.98</td>
<td>40.7</td>
</tr>
<tr>
<td>4</td>
<td>4,916,538</td>
<td>40</td>
<td>189.8</td>
<td>38.6</td>
</tr>
<tr>
<td>5</td>
<td>11,684,817</td>
<td>40</td>
<td>407.0</td>
<td>34.8</td>
</tr>
</tbody>
</table>

Table 3.2: Number of outer conjugate gradient iterations and runtimes for solving the elasticity example on a 28-core setup.

preconditioned residual norm. Table 3.2 displays the number of iterations and runtimes on 28 cores for the two options. The results demonstrate that the multigrid preconditioner yields mesh-independent iteration counts also for the elasticity problem and a more complex geometry. In particular, the run time per unknown is constant or even slightly decreases as the grid is refined, showing that all components in the multigrid algorithm show optimal weak scaling as the problem size is increased.

### 3.4.3 Advection Dominated Problem

To close out this section, we consider a problem where, unlike the previous diffusion dominated Poisson and elasticity problems, we have an advection dominated problem where no multigrid convergence theory exists. From the advection-diffusion problem defined in (2.3), using methods described in Section 2.2, we obtain the weak formulation

\[
a(u, v) := \int_{\Omega} (\varepsilon \nabla v, \nabla u) + (v, \beta \cdot \nabla u) = \int_{\Omega} v \cdot f \, dx =: f(v). \tag{3.3}
\]

#### 3.4.3.1 Streamline diffusion

For diffusion-dominated problems, the standard Galerkin method along with a basic point smoother would be very efficient, however this is not the case for advection-dominated problems. One issue is that a the standard Galerkin approach leads to oscillatory and unstable discretizations. Recalling the Peclet number, given by

\[
P := \frac{||\beta|| L}{\varepsilon},
\]
where $L$ is the length scale of the domain. From [38] we have the following error estimate for this formulation:

$$\|\nabla (u - u_h)\| \leq (1 + P) \inf_{v_h} \|\nabla (u - v_h)\|,$$  

(3.4)

where, for sufficiently smooth $u$ and finite element polynomial degree $k$,

$$\inf_{v_h} \|\nabla (u - v_h)\| \leq \|\nabla (u - I_h u)\| \leq h^k C \|\nabla^k u\|$$  

(3.5)

As a consequence, we obtain the estimate

$$\|\nabla (u - u_h)\| \leq (1 + P) C h^k \|\nabla^k u\|. $$  

(3.6)

So while the numerical solution will converge, the error estimate will be quite large when $\varepsilon \ll \|\beta\| L$, i.e., if the problem is advection-dominated.

To combat this, we will consider the new weak formulation from [38, 63] (among others)

$$a(u_h, v_h) + \sum_K (-\varepsilon \Delta u_h + \beta \cdot \nabla u_h - f, \delta_K \beta \cdot \nabla v_h)_K = F(v_h)$$  

(3.7)

where the sum is done over all cells $K$ with the inner product taken for each cell, and $\delta_K$ is a cell-wise constant stabilization parameter defined in [63]. Essentially, we are adding in the discrete strong form residual to enhance the coercivity of the bilinear form $a(\cdot, \cdot)$ which increases the stability of the discrete solution. This method is commonly referred to as streamline diffusion or SUPG (streamline upwind/Petrov-Galerkin), and was introduced in [30]. Figure 3.9 shows the difference in the standard Galerkin approximation and the SUPG enhanced approximation for the test problem defined in Section 3.4.3.3.

3.4.3.2 Smoothers

So far we have limited the discussion of smoothers to simple, point-based methods like Jacobi and Chebyshev. Here, the term “point-based” is traditionally used to indicate that one solves at one “grid point” at a time; for scalar problems, this means to use a solver that updates one unknown of the linear system at a time, keeping all of the others fixed; one would then iterate over all unknowns in the problem and, once done, start over again from the first unknown until these
“sweeps” converge. Jacobi, Gauss-Seidel, and SOR iterations can all be interpreted in this way. As smoothers in a multigrid method for elliptic PDEs, these point smoothers can work quite well (as seen above), however this is not always the case for other PDEs. One can then consider the more powerful “cell-based” smoothers in these cases.

“Cell-based” methods solve for all unknowns on a cell at once, keeping all other unknowns fixed, repeating for each cell in the mesh. One can think of them as "block" versions of Jacobi, Gauss-Seidel, or SOR, but because degrees of freedom are shared among multiple cells for continuous elements, these blocks overlap and the methods are in fact best be explained within the framework of additive and multiplicative Schwarz methods.

A Schwarz smoother requires a decomposition

$$V = \sum_{j=1}^{J} V_j$$

(3.8)

of our finite element space $V$. Each subproblem $V_j$ also has a Ritz projection $P_j : V \rightarrow V_j$ based on the bilinear form $a(\cdot, \cdot)$. This projection induces a local operator $A_j$ for each subproblem $V_j$. If $\Pi_j : V \rightarrow V_j$ is the orthogonal projector onto $V_j$, one can show $A_j P_j = \Pi_j^T A$ (see Lemma 2.2 in [102]).

With this we can define an additive Schwarz smoother for the operator $A$ as

$$B^{-1} = \sum_{j=1}^{J} P_j A^{-1} = \sum_{j=1}^{J} A_j^{-1} \Pi_j^T.$$  

(3.9)

In other words, we project our solution into each subproblem, apply the inverse of the subproblem $A_j$, and sum the contributions up over all $j$. Note that one can interpret the point-wise (one unknown at a time) Jacobi method as an additive Schwarz method by defining a subproblem $V_j$ for each degree of freedom. Then, $A_j^{-1}$ becomes a multiplication with the inverse of a diagonal entry of $A$.

Since a continuous finite element is used, it is not hard to see that for the degrees of freedom which lie on the boundary of a cell will have contributions from multiple cells, i.e., the blocks are overlapping. For this reason we must add in a damping parameter as to remove the extra information that will accumulate. For the results below this parameter is set at 0.25 since a degree of freedom can be in at most 4 blocks. Note that this lowers the effectiveness of the method as information will be
damped from degree of freedoms that share fewer than 4 block. One can think of a more sophisticated approach in which variable damping based on the number of block the degree of freedom lies is used.

So far, we discussed additive smoothers where the updates can be applied independently and there is no information flowing within a single smoother application. A multiplicative Schwarz smoother addresses this and is defined by

\[ B^{-1} = \left( I - \prod_{j=1}^{J} (I - P_j) \right) A^{-1}. \]  

(3.10)

In contrast to above, the updates on the subproblems \( V_j \) are applied sequentially. This means that the update obtained when inverting the subproblem \( A_j \) is immediately used in \( A_{j+1} \). The standard Gauss-Seidel (or SOR) method can be seen as a multiplicative Schwarz method with a subproblem for each degree of freedom. Here, information travels from one subproblem to the next and therefore there is no need for damping. This is a major advantage to multiplicative methods. The other major advantage to the traveling of information from subproblem to subproblem is that, since here we have an advection term, information is transported along streamlines in the given advection direction. With a small diffusion constant \( \varepsilon \) (i.e. advection-dominated), a smoother that transports information can likely be made to be more effective if the subproblems are executed in the downstream direction. If we want to solve one unknown (or block of unknowns) at a time in the order in which these unknowns (or blocks) are enumerated, then this information propagation property requires reordering degrees of freedom or cells (for the cell-based smoothers) accordingly so that the ones further upstream are treated earlier and those further downstream are treated later. The influence of the ordering will be visible in the results shown below.

For the remainder of this chapter, the additive Schwarz smoother based on cell patches will be called “block Jacobi” and the multiplicative Schwarz smoother based on cell patches will be called “block SOR.”

### 3.4.3.3 Test Problem

We will be considering the following test problem: \( \Omega = [-1, 1] \times [-1, 1] \setminus B_{0.3}(0) \), i.e., a square with a circle of radius 0.3 centered at the origin removed. In addition, we use \( \varepsilon = 0.005 \),
\[ \mathbf{\beta} = [-\sin(\pi/6), \cos(\pi/6)], \quad f = 0, \text{ and Dirichlet boundary values} \]

\[
g = \begin{cases} 
1 & \text{if } x = -1 \text{ or } y = -1, \quad x \geq 0.5 \\
0 & \text{otherwise} 
\end{cases} 
\]

Figure 3.9 depicts the solutions with (left) and without (right) streamline diffusion. Without streamline diffusion we see large oscillations around the boundary layer, demonstrating the instability of the standard Galerkin finite element method for this problem.

As mentioned above, the ordering of the subproblems has an effect on the convergence speed of multiplicative Schwarz smoother. For this problem, information is transported down the advection direction \( \mathbf{\beta} \), and so we will test 3 different types of renumbering algorithms: downstream, random, and upstream, depicted in Figure 3.10. The reordering for the point-based smoothers is done by reordering the cells (as with the cell-based smoothers) and then renumbering the degrees of freedom on the cells in order.

**GMRES Iterations** As the problem is non-symmetric, GMRES is used as the Krylov subspace method. We will test four different smoothers: Jacobi with 6 steps and a damping parameter of 0.6667 \((Q_1)\) or 0.47 \((Q_3)\), SOR with 3 steps and a damping parameter of 1.0 \((Q_1)\) or 0.62 \((Q_3)\), block Jacobi with 3 steps and a damping parameter of 0.25, and block SOR with 1 step and no damping. Unless previously discussed, these values for found using trial and error. All iteration
Table 3.3 gives the GMRES iterations for the additive smoothers. We see that renumbering has no effect on convergence speed. This is because these smoothers compute operations on subproblem independently and add up the results. Since we can define these smoothers as an application of a sum of matrices, and matrix addition is commutative, the order at which we sum the different components will not affect the end result.

On the other hand, Table 3.4 shows that for multiplicative smoothers, we can speed up convergence by renumbering in the advection direction, and similarly, we can slow down convergence if we do the renumbering in the opposite direction. This is because advection-dominated problems have a directional flow of information (in the advection direction) which, given the right renumbering, multiplicative methods are able to capture. This is, however, dependent on the value of $\varepsilon$. As we increase $\varepsilon$ and the problem becomes more diffusion-dominated, we have a more uniform propagation.
Renumbering Strategy

<table>
<thead>
<tr>
<th>DoFs</th>
<th>Downstream</th>
<th>Random</th>
<th>Upstream</th>
<th>Downstream</th>
<th>Random</th>
<th>Upstream</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>160</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>576</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>7</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>2,176</td>
<td>10</td>
<td>12</td>
<td>15</td>
<td>8</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>8,448</td>
<td>11</td>
<td>15</td>
<td>19</td>
<td>10</td>
<td>11</td>
<td>20</td>
</tr>
<tr>
<td>33,280</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>10</td>
<td>12</td>
<td>21</td>
</tr>
<tr>
<td>132,096</td>
<td>12</td>
<td>16</td>
<td>19</td>
<td>11</td>
<td>12</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 3.4: GMRES iterations to reduce residual by $10^8$. Various renumbering strategies for additive smoothers with a $Q_3$ element. Value listed next the method name is the number of pre- or post-smoothing steps per level per iteration.

<table>
<thead>
<tr>
<th>DoFs</th>
<th>Jacobi (6)</th>
<th>Block Jacobi (3)</th>
<th>SOR (3)</th>
<th>Block SOR (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>160</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>576</td>
<td>11</td>
<td>9</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>2,176</td>
<td>15</td>
<td>13</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>8,448</td>
<td>18</td>
<td>15</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>33,280</td>
<td>20</td>
<td>16</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>132,096</td>
<td>20</td>
<td>16</td>
<td>12</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 3.5: GMRES iterations to reduce residual by $10^8$. $Q_1$ element. Value listed next the method name is the number of pre- or post-smoothing steps per level per iteration.

Of information over the mesh and there is a diminished advantage for renumbering in the advection direction.

From here on, the results are limited to using downstream renumbering. Table 3.5 gives a cross comparison of all four smoothers for $Q_1$ elements and Table 3.6 for $Q_3$ elements. For $Q_1$, both multiplicative smoothers require a smaller combination of smoothing steps and iteration counts than either additive smoother. However, when we increase the degree to a $Q_3$ element, there is a clear advantage for the block smoothers in terms of the number of smoothing steps and iterations required to solve. Specifically, the block SOR smoother gives constant iteration counts over the degree, and the block Jacobi smoother only sees about a 38% increase in iterations compared to 75% and 183% increase for Jacobi and SOR, respectively.

**Cost** Iteration counts do not tell the full story in the optimality of a one smoother over another. Obviously we must examine the cost of an iteration. Block smoothers here are at a disadvantage as
Table 3.6: GMRES iterations to reduce residual by $10^8$. $Q_3$ element. Value listed next the the method name is the number of pre- or post-smoothing steps per level per iteration.

<table>
<thead>
<tr>
<th>DoFs</th>
<th>Jacobi (6)</th>
<th>Block Jacobi (3)</th>
<th>SOR (3)</th>
<th>Block SOR (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>336</td>
<td>15</td>
<td>14</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>1,248</td>
<td>23</td>
<td>18</td>
<td>21</td>
<td>9</td>
</tr>
<tr>
<td>4,800</td>
<td>29</td>
<td>21</td>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>18,816</td>
<td>33</td>
<td>22</td>
<td>32</td>
<td>9</td>
</tr>
<tr>
<td>74,496</td>
<td>35</td>
<td>22</td>
<td>34</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3.7: GMRES solve time (in seconds) to reduce residual by $10^8$. $Q_3$ element.

<table>
<thead>
<tr>
<th>DoFs</th>
<th>Jacobi</th>
<th>Block Jacobi</th>
<th>SOR</th>
<th>Block SOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>74,496</td>
<td>0.68</td>
<td>5.82</td>
<td>1.18</td>
<td>1.02</td>
</tr>
</tbody>
</table>

they are having to construct and invert a cell matrix for each cell. Table 3.7 gives a comparison of solve times for a $Q_3$ element with 74,496 DoFs:

The smoother that requires the most iterations (Jacobi) actually takes the shortest time (roughly 2/3 the time of the next fastest method). This is because all that is required to apply a Jacobi smoothing step is multiplication by a diagonal matrix which is very cheap. On the other hand, while SOR requires over 3x more iterations (each with 3x more smoothing steps) than block SOR, the times are roughly equivalent, implying that a smoothing step of block SOR is roughly 9x slower than a smoothing step of SOR. Lastly, block Jacobi is almost 6x more expensive than block SOR, which intuitively makes sense from the fact that 1 step of each method has the same cost (inverting the cell matrices and either adding or multiply them together), and block Jacobi has 3 times the number of smoothing steps per iteration with 2 times the iterations.

**Additional information** There are a few more important points to mention:

1. For a mesh distributed in parallel, multiplicative methods cannot be executed over the entire domain. This is because they operate one cell at a time, and downstream cells can only be handled once upstream cells have already been done. This is fine on a single processor since the processor will just iterate through the list of cells one after the other. However, in parallel, it would imply that some processors are idle because upstream processors have not finished doing the work on cells upstream from the ones owned by the current processor. Once the upstream processors are done, the downstream ones can start, but by that time the upstream
processors have no work left. In other words, most of the time during these smoother steps, most processors are in fact idle, which, from Amdahl’s Law means we will not have parallel scaling to any significant number of processors (see Figure 2.10). One can use a hybrid method where a multiplicative smoother is applied on each subdomain, but as you increase the number of subdomains, the method approaches the behavior of an additive method. This is a major disadvantage to these methods.

2. Current research into block smoothers suggest that soon we will be able to compute the inverse of the cell matrices much cheaper than what is currently being done inside deal.II. This research is based on the fast diagonalization method (dating back to the 1960s) and has been used in the spectral community for around 20 years (see, e.g., [80]). There are currently efforts to generalize these methods to DG and make them more robust. Also, it seems that one should be able to take advantage of matrix-free implementations and the fact that, in the interior of the domain, cell matrices tend to look very similar, allowing fewer matrix inverse computations. However these methods are still a work-in-progress and no generalized implementations exist for parallel, adaptive computations (see [107, 73] for current state of progress).
Chapter 4

Applications to the Stokes Equations

In this chapter we apply a geometric multigrid v-cycle for the purpose of preconditioning for the Stokes systems used in the mantle convection code ASPECT. The work here can be seen as an extension of Chapter 2 in Ryan Grove’s thesis [53] where he applies a matrix-based block preconditioner for the Stokes equations with constant viscosity, using a geometric multigrid v-cycle for the velocity block. Here we will apply the matrix-free multigrid method discussed in Chapter 3 as part of a fully matrix-free Stokes solve, and test the solver against the current matrix-based solver used inside ASPECT for three different benchmarking problems: 2D single inclusion benchmark defined in [71], 3D n-sinker benchmark defined in [91] (zero boundary conditions) and [82] (free-slip/no-normal flux boundary conditions), and 2D annulus benchmark defined in [101]. We will demonstrate the improved scalability and speedup of the matrix-free method over the matrix-based method for both uniform and adaptive mesh refinement, and for massively parallel applications.

The weak and strong scaling results were submitted as a publication with co-author Timo Heister [34] and are currently under review. All of the results from this chapter are from the development version of ASPECT, where the author (with the help of Timo Heister) has implemented all functionality related to each problem, and, if given the specific parameter files used, the results can be replicated by any ASPECT user.
4.1 Review of Current Literature

Here we will discuss the current state-of-the-art finite element solvers for the Stokes equations with applications to mantle convection. We make a point to distinguish between state-of-the-art in terms of methods for solving Stokes with high variable viscosity, and state-of-the-art in terms of available code for users in the mantle convection community. The reason this distinction is important is that the ultimate goal of this chapter is the incorporation into an open-source code, requiring flexibility for various methods required by users, and compatibility with the code that already exists. This is non-trivial and could be one of the reasons that, as we will see, available, flexible applications often can lag far behind in features to state-of-the-art solvers in the literature.\footnote{Another possible reason could be one of motivation, namely, not all researchers are focused on open-source implementations.}

The open-source codes listed below are aimed at solving similar problems as ASPECT. The Stokes solver used in ASPECT is described in Section 4.3.

pTatin3D \cite{pTatin} is a parallel C++ finite element code for processes relevant to geodynamics, similar to ASPECT. The Stokes system is discretized with $[Q_2]^d \times P_1^{\text{disc}}$ elements and solved using the block preconditioner (4.7). The Stokes solver is described in \cite{pTatin} and discussed in detail below. The library has capabilities for massively parallel, matrix-free Stokes computations on globally refined meshes, using PETSc \cite{PETSc} for parallel data structures.

Citcom (with variants CitcomCU \cite{CitcomCU} and CitcomS \cite{CitcomS}) is a parallel C based finite element code for solving applications in mantle convection. Based on the original 2D code described in \cite{Citcom}, the Stokes equation is discretized using a stabilized $[Q_1]^d \times P_0$ element and the solver used is a matrix-based Schur complement CG/pressure correction scheme with full multigrid method for solves in the velocity space, and a CG solve in the pressure space \cite{CG, CG2}. The code does not allow for locally refined meshes.

TerraFERMA (Transparent Finite Element Rapid Model Assembler) is a code aimed at solving coupled, multi-physics problems that arise in Earth sciences. It is a code based on the open-source finite element library FEniCS \cite{FEniCS} and uses PETSc for parallel data structures. The Stokes system is discretized on a fixed tetrahedral meshes using Taylor-Hood elements $[P_2]^d \times P_1$. The solver used is a right-preconditioned FGMRES method with the same block upper diagonal structure (4.7), where the $A$-block inverse is approximated with one AMG v-cycle and the Schur complement is approximated with a weighted mass matrix.
Other codes include ConMan [69, 70] (a 2D, first order stabilized finite element code) and LaMEM [42, 78] (2D and 3D parallel, $[Q_2]^d \times P_1^\text{disc}$ finite element code using PETSc).

Next, we will consider recent papers focused on solving the Stokes equations with high variable viscosity using finite elements. These publications are focused on extreme parallel scaling, as is the focus of this Chapter.

The authors of [82] present a matrix-free multigrid preconditioner for a $[Q_2]^d \times P_1^\text{disc}$ Stokes discretization on hexahedral meshes, aimed at solving systems with high viscosity contrasts. For solving the linear system in (4.6), they use a right-preconditioned FGMRES method with the same block upper diagonal structure (4.7) used in this chapter. For the approximation of $A^{-1}$, they use a CG solve preconditioned with a hybrid GMG-AMG multigrid method where there exists a hierarchy of geometric levels, matrix-free level matrices, Chebyshev smoothers, and transfer operators, and for the coarse level solve, a discretized, matrix-based system with an AMG v-cycle. For the viscosity $\mu$, they create a $Q_1$ representation of viscosity on the finest level and use the GMG restriction operators to represent on this viscosity on the coarser levels. For the approximation of the Schur complement, as will be done in this chapter, they approximate $S$ with a weighted mass matrix (see Section 4.3.2). Then, unlike in this chapter where $S^{-1}$ is approximated with a CG solve, they further approximate $S^{-1}$ as the inverse of the diagonal of the weighted mass matrix. They test the Stokes solver on the $n$-sinker benchmark (same used in Section 4.4.3.1), and consider runs with up to $10^{-6}$ viscosity contrasts. The test using the $n$-sinker benchmark are run on globally refined meshes with up to 56.6M cells (13,824 coarse cells), and up to 4,096 cores. Finally, they demonstrate the capabilities of their Stokes solver within a 3D time dependent simulation of the propagation of continental break-up, where a Stokes solve on 1M cells was performed at each time step.

Similar to [82], the authors of [90] present a matrix-free multigrid preconditioner for a $[Q_2]^d \times P_1^\text{disc}$ Stokes discretization on hexahedral meshes. They use a right-preconditioned GMRES method with the same block upper diagonal structure (4.7) used in this chapter. For the approximation of $A^{-1}$, they use the hybrid hp-GMG-AMG multigrid method from [98] and discussed in Section 3.1, where there exists a hierarchy of geometric levels, matrix-free level matrices, Chebyshev smoothers, and transfer operators, and for the coarse level solve, a discretized, matrix-based system with an AMG v-cycle. Additionally, the geometric multigrid not only consists of $h$ refinement (as discussed in Section 2.5), but initially consists of $p$-refinement, that is, the first levels of the v-cycle are defined by transferring the residual from the $Q_2$ space to the coarser $Q_1$ space on the highest level mesh.
before then performing $h$-multigrid v-cycle. For the approximation of the Schur complement, they consider the least squares communicator

$$
\hat{S}^{-1} = (BD^{-1}B^T)^{-1}(BD^{-1}AD^{-1}B^T)(BD^{-1}B^T)^{-1}
$$

(4.1)

where $D \equiv \text{diag}(A)$, and they define $K \approx BD^{-1}B^T$ as a nodal Poisson operator where $K^{-1}$ is approximated by a GMG v-cycle. Their preconditioner contains no non-linear or varying parts and therefore the non-flexible GMRES Krylov method can be used. They test their Stokes solver on a realistic mantle convection setup with viscosity contrast of $10^6$ and show impressive scaling results on up to 1.5 million cores and with 602 billion degrees of freedom, using an adaptive refinement scheme based on the viscosity jumps.

Three of the authors from [90] present a new weighted version of the least squares communicator (4.1) in [91], where they show results for the $n$-sinker benchmarks defined in Section 4.4.2 with viscosity contrasts of up to $10^{10}$, and compare the performance to the weighted mass matrix approximation.

Lastly, the authors of [49] present a matrix-free multigrid preconditioner for a stabilized $[\mathbb{Q}_1]^d \times \mathbb{Q}_1$ Stokes discretization on tetrahedral meshes. They compare three solvers: a Schur complement CG/pressure correction scheme, preconditioned MinRES Krylov solve with a block diagonal preconditioner (same as (4.7), only no $B^T$ in the upper right block), and an all-at-once geometric multigrid solver with Uzawa-type smoothers and variable v-cycle. In each method an approximation for $A^{-1}$ and $S^{-1}$ are required. For $A^{-1}$, a multigrid v-cycle (based on [20] and discussed in Section 3.1) is used, and for $S^{-1}$ a lumped mass matrix is used. They run various comparisons on globally refined meshes where the all-at-once multigrid sees the lowest and most consistent iteration counts, as well as lowest time to solution. Weak scaling tests are run on up to 780,000 threads and up to 2.2 trillion unknowns with global refinement.

The remainder of this chapter will be focused on the work necessary to approach these types of large scale runs inside the open-source ASPECT code.
4.2 Stokes System Used in ASPECT

The Stokes system as used in most ASPECT applications is a slight variation to that of problem (2.4), and is given by

\[-\nabla \cdot (2\mu \varepsilon(u)) + \nabla p = f \text{ in } \Omega\]
\[\nabla \cdot u = g \text{ in } \Omega\]
\[u = 0 \text{ on } \partial \Omega.\]

(4.2)

Here we include a function \(\mu : \Omega \rightarrow \mathbb{R}\) representing the viscosity of the fluid and replace \(\nabla u\) by the strain-rate tensor \(\varepsilon(u) = \frac{1}{2} (\nabla u + (\nabla u)^T)\). The function \(f : \Omega \rightarrow \mathbb{R}^d\) usually comes from the density terms on the right-hand side of (2.60), and here we only consider incompressible flow where \(g = 0^2\). In mathematical literature which discuss theoretical results (like [64, 29, 39]), one almost always sees the formulation (2.4), due to the fact that, for constant viscosity and for \(\nabla \cdot u = 0\),

\[-2\nabla \cdot (\mu \varepsilon(u)) = -\mu \left( \nabla \cdot u - \nabla \cdot [\nabla u]^T \right) = -\mu \Delta u\]

(4.3)

since,

\[
\left( \nabla \cdot [\nabla u]^T \right)_i = \sum_{j=1}^d \frac{\partial}{\partial x_j} [\nabla u]_{ij}^T
\]
\[= \sum_{j=1}^d \frac{\partial}{\partial x_j} [\nabla u]_{ji}
\]
\[= \sum_{j=1}^d \frac{\partial}{\partial x_j} \frac{\partial}{\partial x_i} u_j
\]
\[= \frac{\partial}{\partial x_i} \sum_{j=1}^d \frac{\partial}{\partial x_j} u_j
\]
\[= \nabla \cdot [\nabla u]
\]
\[= 0.
\]

\(^2\)ASPECT also allows for both implicitly and explicitly defined compressibility based on velocity (see [56]). The GMG-based solver developed in this chapter also includes explicitly defined compressibility, as this is only a change to the right-hand side function, however, implicit compressibility has not yet been implemented.
The computations used in ASPECT, however, have non-constant viscosity, and so we will continue with the more physically accurate strain-rate tensor. A consequence to using $\varepsilon(u)$ is that we introduce coupling among the different velocity components, something that was not present in $\nabla u$ and provides extra challenges which will be discussed later.

Using the strain rate tensor and adding in a viscosity function requires a change in the bilinear form $a(u, v)$ in (2.24), namely,

$$a(u, v) = \int_{\Omega} \nabla v : (2\mu \varepsilon(u)) = \int_{\Omega} \varepsilon(v) : (2\mu \varepsilon(u)).$$

The last equality holds since, for a linear, symmetric operator $S$ with entries $S_{ij}$, for example, $S = 2\mu \varepsilon(u)$ where

$$1/(2\mu) \cdot S_{ij} = \frac{1}{2} \left( u_{x_i}^j + u_{x_i}^j \right),$$

we have

$$\nabla v : S = \sum_{i=1}^{d} \sum_{j=1}^{d} \frac{\partial v_i}{\partial x_j} \cdot S_{ij}$$

$$= \sum_{i=1}^{d} \frac{\partial v_i}{\partial x_i} \cdot S_{ii} + \sum_{i=1}^{d} \sum_{j=i+1}^{d} \left( \frac{\partial v_i}{\partial x_j} \cdot S_{ij} + \frac{\partial v_j}{\partial x_i} \cdot S_{ji} \right)$$

$$= \sum_{i=1}^{d} \frac{1}{2} \left( \frac{\partial v_i}{\partial x_i} + \frac{\partial v_i}{\partial x_i} \right) \cdot S_{ii} + \sum_{i=1}^{d} \sum_{j=i+1}^{d} \left[ \frac{1}{2} \left( \frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} \right) \cdot S_{ij} + \frac{1}{2} \left( \frac{\partial v_j}{\partial x_j} + \frac{\partial v_i}{\partial x_i} \right) \cdot S_{ji} \right]$$

$$= \sum_{i=1}^{d} \sum_{j=1}^{d} \frac{1}{2} \left( \frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} \right) \cdot S_{ij}$$

$$= \varepsilon(v) : S$$

Thus, $a(u, v)$ remains a symmetric, positive-definite bilinear form as long as $\mu$ is bounded. This problem, however, does becomes more difficult to solve numerically if $\mu$ is discontinuous and/or the dynamic ratio of $\mu$, $\text{DR}(\mu) := \frac{u_{\text{max}}}{u_{\text{min}}}$, is large (as well will show later).
4.3 Linear Solver

Following the notation in [71], we seek coefficients $u_j$ and $p_j$ where, for finite element shape functions $\varphi^u_j$ and $\varphi^p_j$,

$$
\begin{align*}
u_h &= \sum_{j=0}^{N_u} u_j \varphi^u_j \\
p_h &= \sum_{j=0}^{N_p} p_j \varphi^p_j
\end{align*}
(4.4)
$$

Here, $N_u$ and $N_p$ are the total number of degrees of freedom for velocity and pressure respectively. Then, for each $0 \leq i \leq N_u$ and $0 \leq l \leq N_p$,

$$
\begin{align*}
o(\varphi^u_i, u_h) + b(\varphi^u_i, p_h) &= f(\varphi^u_i) \\
b(u_h, \varphi^p_j) &= g(\varphi^p_j)
\end{align*}
(4.5)
$$

Solving this system for coefficients $U = \{u_i\}$ and $P = \{p_j\}$ is equivalent to solving the block linear system

$$
\begin{pmatrix}
A & B^T \\
B & 0
\end{pmatrix}
\begin{pmatrix}
U \\
P
\end{pmatrix} = 
\begin{pmatrix}
F \\
0
\end{pmatrix}
(4.6)
$$

where

$$
A_{ij} = \int_\Omega \varepsilon(\varphi^u_i) : (2\mu \varepsilon(\varphi^u_j)) \\
B_{ij} = -\int_\Omega \varphi^p_i (\nabla \cdot \varphi^u_j) \\
F_j = \int_\Omega \varphi^u_j f.
$$

Consider the following preconditioner

$$
P = 
\begin{pmatrix}
A & B^T \\
0 & -S
\end{pmatrix}
(4.7)
$$

where $S = BA^{-1}B^T$ is called the Schur complement. When used as a right preconditioner for the system matrix in (4.6), we have the following preconditioned system

$$
AP^{-1} = 
\begin{pmatrix}
A & B^T \\
B & 0
\end{pmatrix}
\begin{pmatrix}
A^{-1} & A^{-1}B^T S^{-1} \\
0 & -S^{-1}
\end{pmatrix} = 
\begin{pmatrix}
I & 0 \\
BA^{-1} & I
\end{pmatrix}
(4.8)
$$

The preconditioned system $AP^{-1}$ has only 1 distinct eigenvalue $\lambda = 1$ and will converge in at most 2 iteration for an appropriate Krylov subspace method.\(^3\) This is a common block preconditioner

\(^3\)While it seems that the preconditioned system should converge in only 1 iteration, when constructing the Krylov subspace we actually find that it has dimension 2. See [55] for derivation.
used for Stokes solves, especially in the field of mantle convection (see, e.g., [71, 82, 91]) although it is certainly not the only possible choice (cf. [95]).

While on face value this preconditioner may seem ideal, computing $S^{-1}$ is highly impractical given the fact that it is a fully dense matrix. We would also like to avoid computing $A^{-1}$ exactly as this system can become quite large. For these reasons we seek approximate $\hat{A}^{-1}$ and $\hat{S}^{-1}$ instead of computing $A^{-1}$ and $S^{-1}$ exactly.

4.3.1 Choosing $\hat{A}^{-1}$

Since $A$ comes from a vector Poisson equation, multigrid would appear to be a logical choice given that these methods are widely known to have convergence independent of mesh size $h$ for elliptic boundary value problems [24, 103]. Currently in ASPECT, $\hat{A}^{-1}$ is approximated by 1 AMG v-cycle for each Krylov subspace iteration, where the AMG method is not based on the bilinear form $a(\cdot,\cdot)$ (like for matrix $A$), but the bilinear form

$$\hat{a}(u, v) = \sum_{i=1}^{d} (2\mu \varepsilon([u]_{d e d}), \varepsilon([v]_{d e d}))) = (2\mu \nabla u, \nabla v).$$

We will refer to this as partial coupling as, now, only shape functions of the same velocity component will couple. The reasons for this are three fold:

(i) AMG methods, which depend on the sparsity structure of the underlying matrix, tend to deteriorate when coupling vector components in higher order computations [47],

(ii) the resulting matrix $\hat{A}$ will have far fewer entries (1/3 the entries in 3D, see, e.g., Figure 4.1), and therefore less storage requirements, faster AMG setup and faster application, and

(iii) for smooth $\mu$ with little variation, the partial coupling approximation is spectrally equivalent to the fully coupled system, given by the following inequality from [62],

$$\frac{\mu_{\text{min}}}{C_K^2 \mu_{\text{max}}} \leq \frac{(2\mu \varepsilon(v), \varepsilon(v))}{(2\mu \nabla v, \nabla v)} \leq \frac{\mu_{\text{max}}}{\mu_{\text{min}}},$$

where $C_K \geq 0$ is the constant from Korn’s inequality, and $C_K^2 = 2$ when $v|_{\partial\Omega} = 0$ and $\nabla\cdot u = 0$ [60]. A potential consequence of this formulation is, if $\text{DR}(\mu)$ is large (implying $1/\text{DR}(\mu)$ is small), then $(2\mu \nabla u, \nabla v)$ as an approximation for $(2\mu \varepsilon(u), \varepsilon(v))$ could deteriorate (though
this is not necessarily the case).

In Section 4.4 we will test this AMG implementation of $\hat{A}^{-1}$ as used in ASPECT with both matrix-based and matrix-free versions of GMG. For matrix-based GMG, as with AMG, we will consider a method based on the partially coupled system, however for matrix-free GMG, we can consider the fully coupled system, since, as we are computing each matrix entry on-the-fly at the quadrature level, using the strain rate tensor only consists of adding the off diagonal term in the correct place and dividing by 2; essentially, it is for free. This should be a major advantage to the matrix-free methods, especially for larger values of DR($\mu$).

For smoothers, we will use both Jacobi and Chebyshev methods. Derived from the Chebyshev semi-iterative method (explained in [105]), the Chebyshev smoother is an augmentation of the Jacobi relaxation method. The basic idea of the Chebyshev semi-iteration is that, for each Jacobi step, construct a degree $k$ Chebyshev polynomial expression based on two extreme eigenvalues of our matrix, which, when multiplied as a part of the iteration, will further dampen the error. This would be an issue since it is very difficult to get an accurate approximation of the smallest eigenvalue, however, when used as a smoother we can define an interval $(\lambda^*, \lambda_{\text{max}})$ where $\lambda^*$ is a value such that the interval contains only the large eigenvalues. Then the resulting smoother will dampen the error associated with the largest eigenvalues, which has been shown to be very effective in a
multigrid v-cycle. This method then only needs an approximation to the largest eigenvalue and (a very crude) approximation of the smallest in the defined integral, as well as $k$ additional matrix-vector products. The Chebyshev smoother is useful for matrix-free methods since we only need the diagonal (for Jacobi) and the ability to perform CG iteration for the eigenvalue estimate, both of which are available in the matrix-free framework (see Section 2.6). Overall, this method has been shown to be competitive with parallel block Gauss-Seidel methods in [2] with the added advantage that the parallel implementation is exactly equivalent to serial (unlike multiplicative smoothers like Gauss-Seidel whose performance deteriorates with added processors, see Section 3.2).

4.3.2 Choosing $\hat{S}^{-1}$

A common choice for $S = BA^{-1}B^T$ is a weighted pressure mass matrix $M_p$, where $M_p = (\mu^{-1}\varphi^i, \varphi^j)$ [95, 38, 71]. The reasons for this is that $S$ and $M_p$ are spectrally equivalent for constant viscosities [38], making it a good approximation to the inverse. Applying $\hat{S}^{-1}$ is a simple CG solve (since $M_p$ is symmetric positive definite) with an ILU or Chebyshev preconditioner which usually converges in between 1-5 iterations and is not computationally significant compared to $\hat{A}^{-1}$ [71]. This approximation, however, begins to break down when DR($\mu$) is large (see [91]) and a more sophisticated approach is required. Lastly, since this choice of preconditioner requires a CG solve whose iteration count may change from iteration to iteration, a flexible Krylov subspace method for the outer problem must be used. We use a flexible FGMRES for all computations in this chapter. In Chapter 5 we include a comparison of FGMRES to a more memory conscious IDR(s) methods.

4.3.3 Viscosity Averaging

As discussed in [56], we will be using the harmonic averaging of viscosity over all quadrature points on a cell. The user is responsible for supplying the viscosity in each quadrature point of the active mesh (either directly defined in a data file or by supplying a functional representation of viscosity in the Cartesian space). We then take an average of each value at the quadrature points of a single cell, and store this averaged value in a $Q_{0}^{\text{disc}}$ vector, where each cell is associated with a single entry.

For the viscosity values used for the level operators, we transfer the active mesh $Q_{0}^{\text{disc}}$ vector containing the viscosity to each level in the multigrid hierarchy using the same grid restriction
operator as discussed in [36, 29, 61]. This is accomplished by iterating through the cells of the mesh hierarchy from highest level to lowest level, taking the arithmetic average of the viscosity of each child of a single parent cell and assigning that average as the parent’s viscosity. This is an efficient and scalable way of defining viscosity, as minimal communication is required when transferring level vectors throughout the hierarchy (see Section 3.3.1). It is also flexible as it allows the user to supply any viscosity to the quadrature points without limitation. However, while in [56] it was shown that for discontinuous viscosity, averaging the values on the active mesh leads to optimal error rates in the solution, we may become less accurate for problems with continuous viscosity.

4.3.4 No-normal Flux Boundary Conditions

In many ASPECT applications, part or all of the boundary are given no-normal flux (also called free-slip) boundary conditions. These conditions are of the form

\[ u \cdot n = 0, \]

and they enforce that the fluid’s flow on the boundary is tangential to the boundary itself. In the case of the unit square, for a degree of freedom lying on the boundary described by \( y = 1 \), for example, the normal direction is given by \( n = (0,1)^T \). Therefore if a single support point has \( x \) and \( y \) components \( u_1 \) and \( u_2 \) respectively, we must have the single Dirichlet constraint \( u_2 = 0 \), functionality for which already existed in deal.II. However if we take the case of spherical shell centered at the origin, where support points of any degree of freedom on the boundary is exactly equal to the normal direction, we can solve the equation

\[ u_1 n_1 + u_2 n_2 = 0 \]

for either \( u_1 \) or \( u_2 \), where \( n_1 \) is the \( x \) component and \( n_2 \) is the \( y \) component of \( n \). We make this choice based on the comparison of \( n_1 \) and \( n_2 \), for example, if \( |n_1| > |n_2| \), we have the constraint

\[ u_1 = -\frac{n_2}{n_1}u_2. \]

We compute these constraints for each level, and before each prolongation operation, we set the correct constraints and then transfer the vector to the next finest level. Since smoothing operations
only affect unconstrained entries, setting constraints during the restriction operator would not affect the convergence of the method, therefore we only touch these degrees of freedom during the prolongation and we sill preserve the symmetry of the multigrid method.

4.4 Results

The following section we will give results comparing the current AMG-based solver in ASPECT with the developed GMG-based method, both matrix-based and matrix-free implementations. All timings (unless otherwise stated) represent the median of 5 consecutive runs in order to avoid seeing anomalies due to less or more activity on the network at a given moment, leading to drastically different times. We will test FGMRES iteration counts, strong scaling (constant problem size, varying number of cores) and weak scaling (constant problem size per core), and for timings on adaptively refined meshes we will include the model described by the partition efficiency $E$ discussed in Section 3.2.2.

From here on, GMG and AMG are to be used to signify the the method whose block preconditioner (4.7) contains a GMG or AMG v-cycle for the approximation of $A^{-1}$, respectively.

4.4.1 Inclusion Benchmark

The inclusion problem is a 2D benchmark described in [71]. The Stokes system is solved on the domain $\Omega = [0, 2]^2$ and is characterized by the viscosity function

$$\mu(x) = \begin{cases} 
1e3 & \text{if } \|x - 1\|^2 < 0.4 \\
1 & \text{else}
\end{cases}$$

the right hand side function $f = 0$, and $(u, p) = (u_{ref}, p_{ref})$ on the boundary where $(u_{ref}, p_{ref})$ is given in Figure 4.2. In [56] it was shown that cellwise, harmonic averaging of viscosity produced optimal convergence rates for adaptive refinement. Here, as we are only changing the preconditioner, we will not repeat convergence results of this problem.

For these computations, a $[Q_2]^2 \times Q_1$ element is used. For each successive problem size, we refine the mesh globally and therefore the mesh distribution has $E \approx 1$ for any reasonable parallel distribution (i.e., more than 1000 cells per processor). These computations were run on the Palmetto
Figure 4.2: Reference solution for the inclusion benchmark. Velocity along with viscosity is shown on the left, and pressure on the right.

<table>
<thead>
<tr>
<th>Processors</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>2M DoFs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMG</td>
<td>93</td>
<td>96</td>
<td>95</td>
<td>91</td>
<td>92</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMG (matrix-based)</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMG (matrix-free)</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9M DoFs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMG</td>
<td>92</td>
<td>92</td>
<td>98</td>
<td>94</td>
<td>90</td>
<td>87</td>
<td>86</td>
</tr>
<tr>
<td>GMG (matrix-based)</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>70</td>
</tr>
<tr>
<td>GMG (matrix-free)</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
</tr>
<tr>
<td>37M DoFs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMG</td>
<td>135</td>
<td>130</td>
<td>132</td>
<td>129</td>
<td>106</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>GMG (matrix-based)</td>
<td>65</td>
<td>65</td>
<td>65</td>
<td>65</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>GMG (matrix-free)</td>
<td>58</td>
<td>58</td>
<td>58</td>
<td>58</td>
<td>58</td>
<td>58</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Number of outer FGMRES iterations to reduce residual by 1e6 in the inclusion benchmark. The FGMRES restart length is set to 100.

cluster in Clemson, S.C., using the Intel Xeon E5-2665 16 core nodes which support AVX2 allowing for vectorization over 4 doubles. Table 4.1 gives the outer FGMRES iteration counts for reducing the residual by 1e6. Both matrix-based and matrix-free GMG use a point-Jacobi smoother with 4 pre and post-smoothing steps and relaxation parameter $\omega = 0.4$. The iteration counts for the GMG-based methods are significantly lower than those for the AMG-based method (up to around 50% less), and the iteration counts for GMG are far more consistent than AMG in both number of processors and refinement level. The reason for the consistency in the GMG-based method is that, with an additive smoother (like Jacobi), parallel and serial implementations are equivalent, something that is not true for AMG. Finally we note that, for the same point-Jacobi smoother, the matrix-free iteration counts are slightly lower than for matrix-based due to the fact that we are
Next we will consider timing results for each method for the mesh with 37 million degrees of freedom. Figure 4.3a gives the timing of a single a single application of the Stokes preconditioner. Here we see that AMG is faster than both GMG methods for small node counts, but that the matrix-free GMG method scales further. We also see that the matrix-free GMG is significantly faster than matrix-based GMG. Combining these timings with the iterations counts listed in Table 4.1, we see that the matrix-based variant of GMG is roughly equivalent in solve time to the AMG method, and the matrix-free GMG is around twice as fast as the matrix-based.

4.4.2 \( n \)-sinker Benchmark

Next we will consider the \( n \)-sinker benchmark described [91]. The Stokes system is solved on the unit cube, where there exist \( n \) randomly positioned “sinkers” of higher viscosity throughout the domain. For the computations found here and in Chapter 5, we use the sinker positions defined in the ASPECT code, found in the file aspect/benchmarks/nsinker/nsinker.cc. By specifying DR(\( \mu \)),...
we define a smooth viscosity by \( \mu(x) \in [\mu_{\text{min}}, \mu_{\text{max}}] \) where for \( X(x) \in [0,1] \)

\[
X(x) = \prod_{i=1}^{n} \left[ 1 - \exp \left( -\delta \max \left[ 0, |c_i - x| - \frac{\omega^2}{2} \right] \right) \right]
\]

\[
\mu(x) = X(x) \mu_{\text{min}} + (1 - X(x)) \mu_{\text{max}}.
\]

Here \( \mu_{\text{min}} = DR(\mu)^{-1/2} \), \( \mu_{\text{max}} = DR(\mu)^{1/2} \), \( c_i \) are the center of each sinker, \( \delta = 200 \) controls the exponential decay of the viscosity, and \( \omega = 0.1 \) is the diameter of the sinkers. The right hand side is given by \( f(x) = (0,0,\beta(X(x) - 1)) \) with \( \beta = 10 \) and we use homogeneous Dirichlet boundary conditions for the velocity. Physically, this represents gravity pulling down the high viscosity sinkers. Figure 4.4 gives a representation of both the velocity and the pressure solution of this benchmark. For all runs in this section we are using the matrix-free GMG variant with a degree 4 Chebyshev smoother.

The problem difficulty can be increased by increasing \( n \) or \( DR(\mu) \). Table 4.2 gives the iterations required to reduce the residual of the outer FGMRES solve by \( 10^6 \) for different values of these parameters for both AMG and GMG. We see that both AMG and GMG deteriorate as both \( n \) and \( DR(\mu) \) increase, with GMG being slightly more robust. This problem can likely be addressed using methods derived in [91], where it was shown that this deterioration is due to the approximation loss in the Schur complement solve, and a more sophisticated Schur complement approximation was proposed based on least squares communicators. For the remaining results, we will only consider \( n = 4 \) and \( DR(\mu) = 10^4 \), as this is within the range of problems where our Schur complement approach is sufficient.

All timings in this section were from computations run on the new Frontera machine at The University of Texas at Austin’s Texas Advanced Computing Center. We will be using the Intel Xeon Platinum 8280 (“Cascade Lake”) nodes which have 56 cores and 192GB per node. They support AVX-512 instructions allowing for vectorization over 8 doubles. The deal.II version used is 9.1.0-pre, and we compile using gcc 7.1.0, intel-mpi 17.0.3. The p4est version is 2.0.0, the Trilinos version is 12.10.1 and the ASPECT version is 2.1.0-pre.

Figure 4.5a gives the strong scaling for an application of the Stokes block preconditioner from a globally refined mesh with between 5-8 refinement levels, and Table 4.3 gives the corresponding FGMRES iteration counts during the solve. The dashed scaling lines here are all computed based on the data point at 56 cores/6.7M DoFs, therefore the plot represents both strong and weak scaling.
Figure 4.4: Sinker benchmark solution for \( n = 8 \), \( \text{DR}(\mu) = 1e4 \)

<table>
<thead>
<tr>
<th>DR((\mu))</th>
<th>1e2</th>
<th>1e4</th>
<th>1e6</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 sinkers</td>
<td>46</td>
<td>48</td>
<td>52</td>
</tr>
<tr>
<td>8 sinkers</td>
<td>81</td>
<td>196</td>
<td>229</td>
</tr>
<tr>
<td>12 sinkers</td>
<td>76</td>
<td>171</td>
<td>237</td>
</tr>
<tr>
<td>GMG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 sinkers</td>
<td>20</td>
<td>24</td>
<td>26</td>
</tr>
<tr>
<td>8 sinkers</td>
<td>37</td>
<td>72</td>
<td>128</td>
</tr>
<tr>
<td>12 sinkers</td>
<td>39</td>
<td>80</td>
<td>141</td>
</tr>
</tbody>
</table>

Table 4.2: FGMRES iterations required to reduce the residual by 1e6 for the \( n \)-sinker benchmark with increasing \( n \) and DR(\(\mu\)). Run on a 3D mesh with 860K degrees of freedom (\([Q_2]^{\text{dim}} \times Q_1\) element), distributed over 32 processors.

Table 4.3: FGMRES iterations required to reduce the residual by 1e6 for the GMG preconditioner on problems depicted in Figure 4.5a (Globally refined mesh).

We see scaling to around 15-30K DoFs/core for the preconditioner and roughly constant iteration counts. Figure 4.5b gives the timings for the FGMRES solve\(^4\), and unsurprisingly (since the iteration counts are almost constant) we see the same scaling. Figure 4.5c gives the speedup and efficiency for the FGMRES solve.\(^5\)

For a comparison between the AMG and GMG preconditioners, we will consider an adaptively refined mesh, where for each refinement, the number of cells are roughly doubled. We start with a mesh of 4 global refinements and create each new mesh using a Kelly estimator to refine roughly 1/7 of the cells from the previous refinement cycle, doubling the number of cells in our mesh. Table 4.4 gives the runtimes for such a mesh with 5 levels of adaptive refinement on 48 cores (18.5M degrees of freedom). The “Setup” time includes the distribution of the degrees of freedom, setting up of any sparsity patterns necessary, as well as the setup of the data structures required for

\(^4\)Note that now the dashed lines are no longer all based on the 56 cores/6.7M DoFs data point. Since there is a 1 iterations increase in the FGMRES solve from the 6.7M DoF run to the 53M DoF run, we no longer expect the times to be comparable in a weak scaling sense. Now the dashed lines for the three largest problem size are all based on the data point at 48 cores/52M DoFs.

\(^5\)For these runs, at high core counts and with 15K DoFs/core, the timings scale very poorly. We believe this is an issue with the machine and not with our implementation as the Frontera machine is still in its development phase. Therefore, to minimize this effect we are taking the minimum of 5 runs (instead of the median).
(a) One application of the Stokes block preconditioner.

(b) FGMRES solve.

(c) FGMRES solver speedup. Only considering the largest three runs.

Figure 4.5: Strong and weak scaling for GMG for the \( n \)-sinker benchmark, global refinement, 3D mesh with \([Q_2]^{\text{dim}} \times Q_1\) element. Timings are from the Frontera machine.
<table>
<thead>
<tr>
<th></th>
<th>AMG</th>
<th>GMG</th>
<th>factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>12.6s</td>
<td>10.3s</td>
<td>1.2x</td>
</tr>
<tr>
<td>Assemble</td>
<td>32.5s</td>
<td>2.9s</td>
<td>11.2x</td>
</tr>
<tr>
<td>Solve</td>
<td>38.6s</td>
<td>14.8s</td>
<td>2.6x</td>
</tr>
<tr>
<td>Total</td>
<td>83.7s</td>
<td>28.0s</td>
<td>3.0x</td>
</tr>
</tbody>
</table>

Table 4.4: Timing comparison between AMG and GMG for the $n$-sinker benchmark for an adaptively refined, 3D mesh, with 18.5M DoFs ($[Q_2]^{\text{dim}} \times Q_1$ element) on one node (48 cores). Timings are from the Stampede2 machine.

the matrix-free GMG transfer. Here, our GMG method requires roughly 2x the work for distributing the degrees of freedom of the active level (more objects related to DoFs that are implementation specific), as well as distribution of level objects not needed for AMG, but does not need to build any sparsity patterns. This results in roughly equivalent setup times between AMG and GMG, with GMG being slightly faster. The “Assemble” timing includes all matrix assembly (system matrix, preconditioner matrix, AMG setup) as well as assembling the right hand side of the linear system and vectors/tables related to the matrix-free operators. Here is where we see the largest advantage for the GMG method as it has no matrices to assemble, resulting in more than a 10x faster assembly. Combining setup and assembly with the linear solve, we have that the GMG method is around 3x faster for this problem. For time dependent applications, many time steps will typically be solved without further refining the mesh, in which case, we no longer need to call the “Setup” functionality. Then the program time will be dominated by assembly and solve, in which case GMG will be about 4x faster here.

Expanding on Table 4.4, we look at the weak scaling of each component up to 6,144 cores and mesh size of 2.2 billion degrees of freedom. The red dashed line called “imbalance” in the scaling plots represents the ideal weak scaling (black dashed line) multiplied by $1/E$, discussed in Section 3.2.2. Figure 4.6a gives the timing for the preconditioner application, and Table 4.5 gives the number of FGMRES iteration required in the solve. Here we see that, while the AMG preconditioner is cheaper to apply for all but the last data point, the iteration counts for GMG are much lower and stay constant while the AMG iteration counts increase by over 50%. Figure 4.6b shows the solve time and Figure 4.6c shows the speedup. Here we see that, even with the imbalance of the partition, the scaling for GMG is more efficient than AMG, and there is near perfect efficiency when taking into account the imbalance of the mesh partition, which, according to Section 3.3 should remain bounded.
The weak scaling of the setup is shown in Figure 4.6d, and again we see that AMG and GMG are roughly equivalent, with GMG being slightly faster. The setup of the linear system should be optimized in the future since we are roughly on the same order of magnitude as the solve time, and efforts should be made to improve the scaling. The weak scaling of the assembly is shown in Figure 4.6e. Unsurprisingly, the GMG assembly is much cheaper than AMG as there are no matrices to assemble. Scaling for GMG outperforms the prediction based on the imbalance, but this is expected since there is work also on the active mesh, where the \texttt{p4est} distribution is almost perfectly balanced, and not just for the level meshes.

Lastly, Table 4.6 gives a comparison for the memory consumption of each method. These values represent an estimation of the memory consumptions (in MB) of the largest objects for each method on a globally refined mesh with 113K degrees of freedom on a single core. The number of vectors is based on the fact that the typical \texttt{ASPECT} computation has an FGMRES restart length of 50 and therefore, in the worst case, this involves 100 temporary vectors. Then we add in an additional estimate of 5 temporary vectors (solution, system right-hand side, old time steps, etc.). From the table we see that the AMG method requires roughly 2.7x more memory compared to GMG. It is also interesting to note that the largest block of memory for each method is taken up by vectors, and the number of vectors is dominated by the temporary vectors in the FGMRES solver. This is the motivation for finding a new Krylov subspace solver whose temporary vectors are not determined by the number of iterations, allowing us to solve even larger problems. This is the topic of Chapter 5.
Figure 4.6: Weak scaling comparison for the \( n \)-sinker benchmark, adaptive refinement, 3D mesh, \([Q_2]^\text{dim} \times Q_1\) element. The red dashed line represents the ideal weak scaling we can expect based on the imbalance of the parallel partition, explain in Section 3.2.2. Timings are from the Stampede2 machine.
<table>
<thead>
<tr>
<th>Memory (MB)</th>
<th>AMG</th>
<th>GMG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangulation</td>
<td>1.9</td>
<td>1.9</td>
</tr>
<tr>
<td>DoFHandlers</td>
<td>2.8</td>
<td>5.7</td>
</tr>
<tr>
<td>Constraints</td>
<td>1.0</td>
<td>2.7</td>
</tr>
<tr>
<td>$A$</td>
<td>174.2</td>
<td>-</td>
</tr>
<tr>
<td>$B$ and $B^T$</td>
<td>31.4</td>
<td>-</td>
</tr>
<tr>
<td>$A$</td>
<td>58.5</td>
<td>-</td>
</tr>
<tr>
<td>$S$</td>
<td>1.4</td>
<td>-</td>
</tr>
<tr>
<td>Vectors(105)</td>
<td>179.6</td>
<td>179.6</td>
</tr>
<tr>
<td>AMG matrices</td>
<td>59.8</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>510.6</td>
<td>189.9</td>
</tr>
</tbody>
</table>

Table 4.6: Memory consumption required for major components of AMG and GMG for globally refined, 3D mesh, with 113K DoFs ($[Q_2]^{dim} \times Q_1$ element) on 1 cores.

### 4.4.3 No-normal Flux Examples

Here we consider problems with no-normal flux boundary conditions to demonstrate the correctness of the methods implemented in this work and discussed in Section 4.3.4 as well as the robustness of the GMG method with other boundary conditions.

#### 4.4.3.1 3D Sinker Benchmark

We will slightly change the setup of the $n$-sinker benchmark in Section 4.4.2 to match [82] where, instead of zero Dirichlet boundary conditions, we will impose no-normal flux conditions on the sides and bottom of the domain, and the Neumann boundary condition

$$\varepsilon(u) \cdot n = 0$$

on the top of the domain. We will solve again with $n = 4$ sinkers and $\text{DR} (\mu) = 1e4$. Table 4.7 gives the iteration counts. Here, the AMG-based method increases in iterations by about 25% and the GMG-based method by about 15%. However, from previous examples, and from the fact that over 100% of the increase in iterations for the GMG-based method is captured in the first refinement, we expect the AMG-based method to continue to see an increase in iterations over more refinement, where the GMG-based method should maintain constant iteration counts.
<table>
<thead>
<tr>
<th>DoFs</th>
<th>AMG</th>
<th>GMG</th>
</tr>
</thead>
<tbody>
<tr>
<td>113K</td>
<td>96</td>
<td>65</td>
</tr>
<tr>
<td>248K</td>
<td>102</td>
<td>77</td>
</tr>
<tr>
<td>542K</td>
<td>103</td>
<td>78</td>
</tr>
<tr>
<td>1.1M</td>
<td>107</td>
<td>74</td>
</tr>
<tr>
<td>2.4M</td>
<td>110</td>
<td>74</td>
</tr>
<tr>
<td>4.8M</td>
<td>120</td>
<td>75</td>
</tr>
</tbody>
</table>

Table 4.7: FGMRES iterations required to reduce the residual by 1e6 for Sinker benchmark with zero Neumann boundary conditions on top and no-normal flux boundary conditions on sides and bottom. The first mesh is a globally refined cube with 4 levels of refinement, then each subsequent mesh is obtained by refinement with a Kelly estimator, roughly doubling the number of cells/DoFs. The FGMRES restart length was set to 150 to ensure neither method needed a restart to give a fair comparison of the methods.

4.4.3.2 2D Annulus Benchmark

The previous example was using a box domain where no-normal flux constraints were equivalent to zero Dirichlet constraints in one coordinate direction and therefore requiring no extra functionality. Here we consider the 2D annulus benchmark proposed in [101], where the reference solution is depicted in Figure 4.7. The reference solution is defined in polar coordinates, with a tuning parameter \( k \) that describes the number of convection cells in the domain. The velocity of the reference solution \( u_{\text{ref}} \) has the following boundary property: \( u_{\text{ref}} \cdot n = 0 \). Since this benchmark has a reference solution, one could solve by prescribing the correct velocity on the boundary, i.e.,

\[
 u = u_{\text{ref}} \quad \text{on } \partial\Omega
\]

and test that the solver converges to the correct error rates. However, we would like to use no-normal flux constraints

\[
 u \cdot n = 0 \quad \text{on } \partial\Omega.
\]

From analysis in [39], to solve for the underlying reference solution while using no-normal flux constraints, we must add the traction boundary integral

\[
 \int_{\partial\Omega} \left[ (\nabla u_{\text{ref}} \cdot n) \times n \right] \cdot (v \times n) \quad \text{on } \partial\Omega
\]
to the right-hand side. Since this problem is in 2D, where the cross-product is not defined, we take the cross product $\mathbf{v} \times \mathbf{n}$ to be the projection of $\mathbf{v}$ in the tangential direction, i.e.,

$$\mathbf{v} \times \mathbf{n} = (\mathbf{v} \cdot \mathbf{t}) \mathbf{t}$$

where $\mathbf{t} \cdot \mathbf{n} = 0$.

Table 4.8 gives the Stokes iterations required to reduce the residual by $1e8$ for AMG and GMG-based solvers with no-normal flux boundaries and with prescribed boundaries. We see that while the iteration counts for GMG are slightly lower for prescribed boundaries, AMG sees roughly the same iteration counts, and overall, GMG gives more consistent iterations counts than AMG for subsequent mesh refinements. Table 4.9 gives the corresponding error rates for the systems with no-normal flux boundaries. Here we see that both the AMG and GMG-based solvers converge identically and with the theoretically optimal rates.
Table 4.8: FGMRES iterations required to reduce the residual by 1e8 for the annulus benchmark with both no-normal flux and prescribed boundaries. Run using $[Q_2]^2 \times Q_1$ elements on a single core, and a FGMRES restart length of 100. Each problem size obtained from a sequence of global refinements.

<table>
<thead>
<tr>
<th>DoFs</th>
<th>AMG</th>
<th>GMG</th>
<th>AMG</th>
<th>GMG</th>
</tr>
</thead>
<tbody>
<tr>
<td>552</td>
<td>17</td>
<td>15</td>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>1968</td>
<td>18</td>
<td>18</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>7392</td>
<td>60</td>
<td>20</td>
<td>63</td>
<td>14</td>
</tr>
<tr>
<td>28608</td>
<td>61</td>
<td>22</td>
<td>67</td>
<td>13</td>
</tr>
<tr>
<td>112512</td>
<td>77</td>
<td>16</td>
<td>76</td>
<td>13</td>
</tr>
<tr>
<td>446208</td>
<td>83</td>
<td>15</td>
<td>82</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 4.9: L2-norm of the error for velocity and pressure, and the rate of decrease between each global refinement step, given by $e_h/e_{2h}$, for the annulus benchmark with no-normal flux boundaries. Discretizations with a $[Q_2]^\text{dim} \times Q_1$ element should yield error rates of $2^3$ and $2^2$ for velocity and pressure, respectively, which we observe here.

<table>
<thead>
<tr>
<th>DoFs</th>
<th>$|u - u_{\text{ref}}|_2$</th>
<th>rate</th>
<th>$|p - p_{\text{ref}}|_2$</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>552</td>
<td>7.292236e-02</td>
<td>-</td>
<td>1.238553e+00</td>
<td>-</td>
</tr>
<tr>
<td>1968</td>
<td>1.002658e-02</td>
<td>7.3</td>
<td>3.395538e-01</td>
<td>3.6</td>
</tr>
<tr>
<td>7392</td>
<td>1.303234e-03</td>
<td>7.9</td>
<td>8.565015e-02</td>
<td>4.0</td>
</tr>
<tr>
<td>28608</td>
<td>1.650274e-04</td>
<td>7.7</td>
<td>2.133998e-02</td>
<td>4.0</td>
</tr>
<tr>
<td>112512</td>
<td>2.071066e-05</td>
<td>8.0</td>
<td>5.324877e-03</td>
<td>4.0</td>
</tr>
<tr>
<td>446208</td>
<td>2.591438e-06</td>
<td>8.0</td>
<td>1.330392e-03</td>
<td>4.0</td>
</tr>
</tbody>
</table>

| GMG   |  |  |  |  |
| 552   | 7.292236e-02     | -    | 1.238553e+00     | -    |
| 1968  | 1.002658e-02     | 7.3  | 3.395538e-01     | 3.6  |
| 7392  | 1.303234e-03     | 7.9  | 8.565015e-02     | 4.0  |
| 28608 | 1.650274e-04     | 7.7  | 2.133998e-02     | 4.0  |
| 112512| 2.071066e-05     | 8.0  | 5.324877e-03     | 4.0  |
| 446208| 2.592336e-06     | 8.0  | 1.330392e-03     | 4.0  |
Chapter 5

Performance Analysis of Krylov Methods for Stokes Computations

From the comparison of memory consumption in Table 4.6 we see that roughly 95% of the memory for the GMG-based method is from the storage of temporary vectors in the FGMRES method. The reason for this is that, unlike matrix-based methods whose system and preconditioner matrix take up the majority of the memory consumed (see Table 4.6), matrix-free methods have very little additional storage outside of temporary vectors. At the worst case, the number of temporary vectors needed for FGMRES is twice the restart length, where, in ASPECT the restart length is set to 50 by default. From the results in Chapter 4, for even moderately difficult problems in 3D we expect iteration counts of at least 30, meaning that the number of temporary vectors needed is expected to be very close to the maximum number dictated by the restart length. In addition to the increased storage requirement, the FGMRES method also has a quadratic increase in scalar products per iteration. These scalar products are not expected to scale $O(1/p)$, but instead $O(\log p)$, which could have a negative impact on the overall scaling of the method. Therefore we will seek improvements to the current FGMRES method used in ASPECT by considering both GMRES and IDR(s) methods.

It should be noted that, although the Stokes system we wish to solve is symmetric, we are using a non-symmetric preconditioner and therefore MINRES, which offers a minimization of the residual norm and does not require an increase in number of temporary vectors with increased iterations, will not be useful. One can consider the block diagonal preconditioner from [95], however
we will not consider that preconditioner here.

5.1 Current Solvers used for Stokes in the Literature

We refer the reader to Section 4.1 where we discuss the current solvers used for Stokes computations in the literature. Of interest to this chapter are methods which reduce the required number of temporary vectors. These include preconditioned MINRES method in [95], the Uzawa method using CG solves in [85, 49] and the Multigrid solver with an Uzawa-type smoother in [49]. [49] gives a comparison between each of these three methods, and it is fairly conclusive that for large problem sizes, the GMG solver with Uzawa smoother is the most competitive. However, no comparison exists for this method compared to the upper triangular block preconditioner used here and in [71, 82, 90, 91].

In the literature we did not find a case of anyone using the IDR($s$) method for Stokes systems. We will demonstrate the convergence of such methods for $s = 1$ (BiCGStab) and $s = 2$ with various experiments based on the $n$-sinker benchmark from Section 4.4.2.

5.2 Cheap vs. Expensive Preconditioning in ASPECT

The computations in Chapter 4 used the flexible Krylov method FGMRES as the outer solver for the Stokes system since the Schur complement approximation in the ASPECT code required a CG solve whose application could be different at each outer iteration. In theory, since this CG solve contains typically low iteration counts, a suitable non-flexible version of the preconditioner could be used where the CG solve is replaced by one application of the inverse diagonal of the weighted mass matrix (common in the literature, see e.g. [82]) or a single GMG v-cycle (similar to the A-block approximation described in Section 4.3.1). Aside from the typical deterioration of the mass matrix approximation seen for problems with high viscosity contrasts, replacing the mass matrix solve with a one of the non-flexible variants often does not significantly increase the outer iterations required by the Krylov solver and would allow for the use of GMRES instead of FGMRES, effectively halving the memory requirements for temporary vectors. Considering again the memory comparison from Table 4.6, Table 5.1 gives a comparison of the memory consumption required between FGMRES(50) and GMRES(50). Using FGMRES for matrix-based computations
Table 5.1: Memory consumption required for major components of AMG- and GMG-based methods for globally refined, 3D mesh, with 113K DoFs ([Q₂]dim x Q₁ element) on 1 core. Estimates for FGMRES(50) (105 total vectors) and GMRES(50) (55 total vectors) are given.

<table>
<thead>
<tr>
<th></th>
<th>FGMRES(50)</th>
<th>GMRES(50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory (MB)</td>
<td>AMG</td>
<td>GMG</td>
</tr>
<tr>
<td>Triangulation</td>
<td>1.9</td>
<td>1.9</td>
</tr>
<tr>
<td>DoFHandlers</td>
<td>2.8</td>
<td>5.7</td>
</tr>
<tr>
<td>Constraints</td>
<td>1.0</td>
<td>2.7</td>
</tr>
<tr>
<td>A</td>
<td>174.2</td>
<td>-</td>
</tr>
<tr>
<td>B and Bᵀ</td>
<td>31.4</td>
<td>31.4</td>
</tr>
<tr>
<td>Ā</td>
<td>58.5</td>
<td>-</td>
</tr>
<tr>
<td>Ĥ</td>
<td>1.4</td>
<td>-</td>
</tr>
<tr>
<td>Vectors</td>
<td>179.6</td>
<td>179.6</td>
</tr>
<tr>
<td>AMG matrices</td>
<td>59.8</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>510.6</td>
<td>189.9</td>
</tr>
</tbody>
</table>

requires about 1.2x more memory than GMRES, and for matrix-free methods FGMRES requires around 1.8x more memory than GMRES.

The reason that the ASPECT code uses a preconditioner which is flexible in the Schur complement approximation over a non-flexible version is more to do with the need for what is termed inside the code as the expensive preconditioner: the block preconditioner in (4.7) where a full CG solve with a multigrid preconditioner is used as the approximation of \( A^{-1} \) (as opposed to the cheap preconditioner described in Section 4.3.1 which contains a single multigrid v-cycle based on a partially coupled \( A \)-block). The necessity of this expensive preconditioner is due to the fact that, for matrix-based computations, the AMG v-cycle in the cheap preconditioner is based on a partially coupled \( A \)-block (discussed in Section 4.3.1) which is often an insufficient approximation to the fully coupled system. However, as shown in Chapter 4, the new GMG-based method does not have the same deterioration in the \( A \)-block approximation since a full coupling of velocity components is used. Therefore, combined with the fact that the memory saved is more drastic for matrix-free methods, it makes sense for us to consider a cheap preconditioner using GMRES instead of FGMRES.

It should be noted that, while less necessary for the matrix-free solver implemented into ASPECT as part of this thesis, in some very difficult problems an expensive preconditioner containing a full \( A \)-block solve may still be required. Therefore flexibility is still of interest for any Krylov method used.
5.3 Timing of Individual Components

Using the cheap version of the block preconditioner (4.7) where both the velocity and pressure spaces contain a single GMG v-cycle, we analyze the cost of GMRES, IDR(1), and IDR(2) by considering three operations: scalar products, matrix-vector products, and preconditioner applications. We consider the direct solves of size $s \times s$ for the IDR($s$) methods and the least squares computations of up to size $m \times m + 1$ in GMRES($m$) to be negligible as they will be sufficiently small in the context of millions of unknowns, and vector addition and vector scaling will similarly not be included as they should scale $\mathcal{O}(1/p)$ and the run times should be very small.

Table 5.2 gives a cross comparison of the costs and storage requirements of each method and Figure 5.1 gives a scaling plot of each component for a Stokes discretization on a mesh with 53 million degrees of freedom. The problem is run with between 56 and 14,336 processors on the Frontera machine in Austin, TX. The runtimes are an average of 1,000 scalar products, 100 matrix-vector products, and 10 preconditioner applications. The preconditioner application contains one GMG v-cycle on the velocity space, one GMG v-cycle on the pressure space, and a matrix-vector product with $B^T$.

Here we can see the trade offs between the methods, namely, while GMRES requires an increase of vector storage linear in the number of iterations and an increased number of scalar products quadratic in iterations, IDR($s$) methods are inherently more expensive as they require $s + 1$ preconditioner applications, with preconditioning being the most expensive operation. We also see that, while the preconditioner applications scale $\mathcal{O}(1/p)$ to around 15,000 unknowns per core and matrix-vector products scale $\mathcal{O}(1/p)$ all the way to 3,000 unknowns per core, scalar products eventually scale $\mathcal{O}(\log p)$ (the cost of an all-reduce among processors). However, for a reasonable problem size ($\sim 3,000$ unknowns per processor at the last data point), both the cost for a scalar product and the cost of a matrix-vector product at least an order of magnitude less than the cost of a preconditioner application.

Using the values in Table 5.2 and the timings in Figure 5.1, Figure 5.2 gives a model for the theoretical solve time of each method for a “low iteration count solve” of 25 iterations and a “high iteration count” solve of 100 iterations. For GMRES (Figure 5.2a), we include both GMRES with no restart and GMRES(50) (given by the dashed, colored lines). We see that the scaling of GMRES is affected by the restart length for the 100 iteration solve, however the difference is negligible for
Table 5.2: Number of scalar products \((v, w)\), matrix-vector products \(Av\), preconditioner application \(B^{-1}v\), and required temporary vectors in \(i\) iterations for each Krylov method considered. We are assuming no restart for GMRES.

<table>
<thead>
<tr>
<th></th>
<th>GMRES</th>
<th>IDR(1)</th>
<th>IDR(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((v, w))</td>
<td>(\frac{1}{2}i(i-1))</td>
<td>4i</td>
<td>2 + 2i</td>
</tr>
<tr>
<td>(Av)</td>
<td>(1 + i)</td>
<td>(1 + 2i)</td>
<td>(1 + 3i)</td>
</tr>
<tr>
<td>(B^{-1}v)</td>
<td>(1 + i)</td>
<td>(2i)</td>
<td>(3i)</td>
</tr>
<tr>
<td># of vectors</td>
<td>(i)</td>
<td>8</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 5.1: Scaling for scalar product, matrix-vector product, and preconditioner application for 3D Stokes discretization with 53M DoFs. Timings from runs on Frontera.
Figure 5.2: Model for solve times for GMRES(50), IDR(1), and IDR(2) based on component timings in Figure 5.1 from Frontera for different iterations counts.

a restart length of 50. And since a restart length of 50 is the default inside ASPECT for reasons of vector storage (from Table 4.6, this is the restart length where vectors become the largest objects), we don’t consider the increase in or lack of scaling of scalar products to be an issue. For both IDR(1) (Figure 5.2b) and IDR(2) (Figure 5.2c), the scaling of the model solve is dominated by the scaling of the preconditioner, as we expect since neither scalar products nor matrix-vector products increase with the number of iterations and the preconditioner application is the most expensive component. Figure 5.3 gives each method’s model scaling for the 100 iteration solve. IDR(1) is roughly 1.5x as expensive as GMRES(50) for the first data point (102.9s vs. 67.4s) and IDR(2) is roughly 2.3x as expensive as GMRES(50) (155.1s vs. 67.4s).
5.4 Solver Performance

While the cost per iteration for GMRES(50) is superior to both IDR(1) and IDR(2), we must examine the performance of each method based on number of iterations and solve time. Here we will consider the same 3D \( n \)-sinker benchmark as in Section 4.4.2, now with 8 total sinkers and \( \text{DR}(\mu) = 1e2, 1e4, \text{and } 1e6 \), that is, we are considering a problem of increasing difficulty. We start with a globally refined mesh with roughly 860,000 degrees of freedom, and then do 3 steps of adaptive refinement with a Kelly estimator, roughly doubling the problem size in each refinement. Each run is on a single 40 core node on the Palmetto cluster in Clemson, SC.

Table 5.3 gives a comparison of the iteration counts of each method. For all problems, IDR(2) iterations are lower than both GMRES(50) and IDR(1), and the iteration counts for each method remain consistent with each refinement. IDR(1) and IDR(2) also give a slightly more robust solver for the more difficult problems as there is a 250% and 260% increase in iterations, respectively, between \( \text{DR}(\mu) = 1e2 \) and \( \text{DR}(\mu) = 1e6 \), while GMRES(50) sees about a 340% increase in iterations.

Table 5.4 gives the solve times associated with each run in Table 5.3, with the bold, red values representing the fastest runtime in each row. For \( \text{DR}(\mu) = 1e2 \) and \( \text{DR}(\mu) = 1e4 \), the times for all three methods are roughly equivalent, with GMRES(50) being the fastest in all but one solve.
Table 5.3: Iteration counts to reduce the residual by $10^6$ for $n$-sinker benchmark with 8 sinkers and changing $\text{DR}(\mu)$.

<table>
<thead>
<tr>
<th>Solver</th>
<th>GMRES(50)</th>
<th>IDR(1)</th>
<th>IDR(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{DR}(\mu)=10^2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>860K DoFs</td>
<td>38</td>
<td>24</td>
<td>15</td>
</tr>
<tr>
<td>1.8M DoFs</td>
<td>39</td>
<td>25</td>
<td>17</td>
</tr>
<tr>
<td>3.8M DoFs</td>
<td>39</td>
<td>27</td>
<td>17</td>
</tr>
<tr>
<td>7.7M DoFs</td>
<td>39</td>
<td>23</td>
<td>15</td>
</tr>
<tr>
<td>$\text{DR}(\mu)=10^4$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>860K DoFs</td>
<td>82</td>
<td>64</td>
<td>34</td>
</tr>
<tr>
<td>1.8M DoFs</td>
<td>86</td>
<td>57</td>
<td>33</td>
</tr>
<tr>
<td>3.8M DoFs</td>
<td>87</td>
<td>53</td>
<td>36</td>
</tr>
<tr>
<td>7.7M DoFs</td>
<td>88</td>
<td>57</td>
<td>33</td>
</tr>
<tr>
<td>$\text{DR}(\mu)=10^6$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>860K DoFs</td>
<td>162</td>
<td>84</td>
<td>47</td>
</tr>
<tr>
<td>1.8M DoFs</td>
<td>163</td>
<td>94</td>
<td>56</td>
</tr>
<tr>
<td>3.8M DoFs</td>
<td>166</td>
<td>89</td>
<td>49</td>
</tr>
<tr>
<td>7.7M DoFs</td>
<td>172</td>
<td>85</td>
<td>54</td>
</tr>
</tbody>
</table>

and IDR(2) begin faster than IDR(1). For $\text{DR}(\mu)=10^6$, IDR(2) becomes consistently faster than GMRES(50) due to the lower iteration counts.

5.5 Flexibility

Lastly, we will test the IDR($s$) methods for flexibility. As mentioned in Section 2.4.4.1, the flexibility of the IDR($s$) method is relatively unknown and has not been shown (to our knowledge) for problems with high non-linearity of the preconditioner. As discussed in Section 5.2, often times for \textsc{aspect} computations an expensive preconditioner is used where the approximation of $A^{-1}$ is given by a CG solve on the velocity space instead of just a single multigrid v-cycle. We will use such a preconditioner here where the tolerance on the CG solve is varied to show the effects on flexible preconditioning for the IDR($s$) method. For the Schur complement approximation we will continue to use a single multigrid v-cycle. FGMRES(50) will be used for all runs with flexible preconditioning.

Table 5.5 gives the outer Krylov iterations for the problem with $\text{DR}(\mu)=10^4$ for both the expensive preconditioner with a varying tolerance for the $A$-block solve and the cheap preconditioner with only one v-cycle on the velocity space. For flexible preconditioning, FGMRES(50) sees a decrease in iteration counts which is expected since FGMRES is known to be flexible [92] and the flexible preconditioner contains a better approximation of $A^{-1}$. Both IDR(1) and IDR(2) are clearly
Solver | GMRES(50) | IDR(1) | IDR(2)
--- | --- | --- | ---
$\text{DR}(\mu)=1e2$ | | | |
860K DoFs | 0.63 | 0.63 | **0.55**
1.8M DoFs | **1.88** | 2.04 | 2.05
3.8M DoFs | 4.41 | 5.39 | 5.01
7.7M DoFs | **10.45** | 13.44 | 10.84

$\text{DR}(\mu)=1e4$ | | | |
860K DoFs | **1.25** | 1.59 | 1.31
1.8M DoFs | 3.95 | 4.61 | 4.04
3.8M DoFs | **9.70** | 10.75 | 10.89
7.7M DoFs | **22.32** | 25.74 | 22.63

$\text{DR}(\mu)=1e6$ | | | |
860K DoFs | 2.42 | 2.13 | **1.78**
1.8M DoFs | 7.09 | 7.67 | **6.84**
3.8M DoFs | 18.76 | 18.14 | **15.20**
7.7M DoFs | 42.20 | 39.15 | **37.69**

Table 5.4: Time (in seconds) to solution for solves in Table 5.3. The fastest runtime in each row is highlighted red. Timings from runs on Palmetto using 40 cores.

not flexible as, given a better approximation of $A^{-1}$ in the preconditioner, the iteration counts actually increase, and the convergence becomes much less consistent with refinement. As we increase the tolerance on the CG solve inside the preconditioner, the convergence for both FGMRES(50) and IDR(s) becomes slightly faster, and the IDR(s) methods become more consistent. However, the IDR(s) methods never quite see the same convergence as they did with the non-flexible preconditioner, although IDR(2) does still see lower iteration counts than FGMRES(50) with the higher CG tolerances.

Lastly, since the GMG v-cycles dominate the runtime of the CG iteration, and the CG solve for the $A$-block requires an average of 7-8 iterations for a tolerance of 1e-2, 14-15 iterations for a tolerance of 1e-4, and 21-23 iterations for a tolerance of 1e-6, the preconditioner application become much more expensive for an increased CG solve tolerance. Then, since the application of the approximation of $A^{-1}$ dominates the overall runtime of the preconditioner application, we can consider the cost of both the cheap and expensive preconditioner application to be dominated by the number of GMG v-cycles performed in the velocity space. For example, an outer FGMRES solve with an inner CG tolerance of 1e-6, since it requires 21-23 CG iterations, is roughly 21-23x more expensive per iteration than the cheap preconditioner with only a single v-cycle. The effect on the runtime of the increase in v-cycles is evident in Table 5.6 where, for each Krylov method, the cheap preconditioner is significantly faster than the expensive preconditioner, and, even with the slightly
<table>
<thead>
<tr>
<th>Solver</th>
<th>FGMRES(50)</th>
<th>IDR(1)</th>
<th>IDR(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cheap preconditioner</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>860K DoFs</td>
<td>82</td>
<td>64</td>
<td>34</td>
</tr>
<tr>
<td>1.8M DoFs</td>
<td>86</td>
<td>57</td>
<td>33</td>
</tr>
<tr>
<td>3.8M DoFs</td>
<td>87</td>
<td>53</td>
<td>36</td>
</tr>
<tr>
<td>7.7M DoFs</td>
<td>88</td>
<td>57</td>
<td>33</td>
</tr>
<tr>
<td><strong>A-block tol: 1e-2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>860K DoFs</td>
<td>69</td>
<td>79</td>
<td>53</td>
</tr>
<tr>
<td>1.8M DoFs</td>
<td>68</td>
<td>166</td>
<td>78</td>
</tr>
<tr>
<td>3.8M DoFs</td>
<td>69</td>
<td>78</td>
<td>69</td>
</tr>
<tr>
<td>7.7M DoFs</td>
<td>72</td>
<td>123</td>
<td>83</td>
</tr>
<tr>
<td><strong>A-block tol: 1e-4</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>860K DoFs</td>
<td>64</td>
<td>77</td>
<td>43</td>
</tr>
<tr>
<td>1.8M DoFs</td>
<td>65</td>
<td>88</td>
<td>41</td>
</tr>
<tr>
<td>3.8M DoFs</td>
<td>66</td>
<td>70</td>
<td>36</td>
</tr>
<tr>
<td>7.7M DoFs</td>
<td>68</td>
<td>74</td>
<td>47</td>
</tr>
<tr>
<td><strong>A-block tol: 1e-6</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>860K DoFs</td>
<td>64</td>
<td>95</td>
<td>39</td>
</tr>
<tr>
<td>1.8M DoFs</td>
<td>65</td>
<td>70</td>
<td>46</td>
</tr>
<tr>
<td>3.8M DoFs</td>
<td>66</td>
<td>85</td>
<td>38</td>
</tr>
<tr>
<td>7.7M DoFs</td>
<td>68</td>
<td>93</td>
<td>48</td>
</tr>
</tbody>
</table>

Table 5.5: Outer iteration counts to reduce the residual by $1e6$ for $n$-sinker benchmark with 8 sinkers and $\text{DR} (\mu) = 1e4$. Here we vary the relative tolerance used for the CG solve on the $A$-block in the expensive preconditioner.

lower iteration counts for IDR(2), the solve time is no longer competitive to the FGMRES(50) method.

### 5.6 Recommendation for ASPECT

Based on the results in this chapter, the recommendation is to continue to use FGMRES(50) inside ASPECT for matrix-based computations where memory consumption is not limited by temporary vector storage and where the $A$-block approximation will often require a full CG solve. For all matrix-free computations, the Schur complement approximation should be changed from a full CG solve with a weighted mass matrix to a single GMG v-cycle based on the weighted mass matrix. If a full CG solve for the $A$-block of the preconditioner is required, FGMRES(50) should be used, else, GMRES(50) should be used given that the user is not constrained by memory requirements.

For going to larger scale problems where storage of at least 55 temporary vectors is not available, IDR(2) should be used.

Table 5.7 contains a comparison of the memory consumption of each recommended method.
<table>
<thead>
<tr>
<th>Solver</th>
<th>FGMRES(50)</th>
<th>IDR(1)</th>
<th>IDR(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cheap preconditioner</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v-cycle per iteration</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>860K DoFs</td>
<td>0.63</td>
<td>0.63</td>
<td><strong>0.55</strong></td>
</tr>
<tr>
<td>1.8M DoFs</td>
<td><strong>1.88</strong></td>
<td>2.04</td>
<td>2.05</td>
</tr>
<tr>
<td>3.8M DoFs</td>
<td><strong>4.41</strong></td>
<td>5.39</td>
<td>5.01</td>
</tr>
<tr>
<td>7.7M DoFs</td>
<td><strong>10.45</strong></td>
<td>13.44</td>
<td>10.84</td>
</tr>
<tr>
<td>A-block tol: 1e-2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v-cycle per iteration</td>
<td>7-8</td>
<td>14-16</td>
<td>21-23</td>
</tr>
<tr>
<td>860K DoFs</td>
<td><strong>5.67</strong></td>
<td>12.69</td>
<td>12.37</td>
</tr>
<tr>
<td>1.8M DoFs</td>
<td>17.59</td>
<td>85.75</td>
<td>64.11</td>
</tr>
<tr>
<td>3.8M DoFs</td>
<td>43.11</td>
<td>106.07</td>
<td>142.35</td>
</tr>
<tr>
<td>7.7M DoFs</td>
<td>105.06</td>
<td>373.81</td>
<td>385.41</td>
</tr>
<tr>
<td>A-block tol: 1e-4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v-cycle per iteration</td>
<td>14-15</td>
<td>28-30</td>
<td>42-45</td>
</tr>
<tr>
<td>860K DoFs</td>
<td><strong>11.09</strong></td>
<td>22.32</td>
<td>19.66</td>
</tr>
<tr>
<td>1.8M DoFs</td>
<td>37.45</td>
<td>92.09</td>
<td>62.17</td>
</tr>
<tr>
<td>3.8M DoFs</td>
<td>91.87</td>
<td>183.79</td>
<td>141.77</td>
</tr>
<tr>
<td>7.7M DoFs</td>
<td><strong>216.78</strong></td>
<td>427.21</td>
<td>415.68</td>
</tr>
<tr>
<td>A-block tol: 1e-6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v-cycle per iteration</td>
<td>21-23</td>
<td>42-46</td>
<td>63-69</td>
</tr>
<tr>
<td>860K DoFs</td>
<td><strong>15.99</strong></td>
<td>45.97</td>
<td>26.27</td>
</tr>
<tr>
<td>1.8M DoFs</td>
<td><strong>53.37</strong></td>
<td>108.46</td>
<td>109.21</td>
</tr>
<tr>
<td>3.8M DoFs</td>
<td><strong>136.06</strong></td>
<td>343.44</td>
<td>221.67</td>
</tr>
<tr>
<td>7.7M DoFs</td>
<td><strong>322.23</strong></td>
<td>824.97</td>
<td>644.21</td>
</tr>
</tbody>
</table>

Table 5.6: Time (in seconds) to solution for solves in Table 5.5. The fastest runtime in each row is highlighted red. Timings from runs on Palmetto using 40 cores.
Table 5.7: Memory consumption required for major components of AMG- and GMG-based methods for globally refined, 3D mesh, with 113K DoFs ([Q_2]_d x Q_1 element) on 1 core. FGMRES(50) (105 total vectors) estimates are given for both AMG and GMG, and both GMRES(50) (55 total vectors) and IDR(2) (16 total vectors) estimates are given for GMG.

Following Table 4.6 and 5.1, we consider a Stokes solve using the current FGMRES(50) method (matrix-based or matrix-free with flexible preconditioning), a matrix-free solve using GMRES(50) (non-flexible preconditioning), and a solve using IDR(2) (large scale, non-flexible preconditioning). For the GMG-based method, FGMRES(50) requires roughly 1.8x more memory than GMRES(50) and 5x more memory than IDR(2). Comparison the AMG-based solver currently in ASPECT (using FGMRES(50)) and the GMG-based solver using IDR(2), the current AMG-based solver requires 13.5x more memory. This implies that the capabilities of solving large scale problems inside ASPECT could be drastically increased by using an matrix-free Stokes solve with IDR(2), allowing for runs with an order of magnitude more unknowns.
Chapter 6

Conclusions

In this thesis we have continued the development of the deal.II implementation of the geometric multigrid v-cycle for adaptive and parallel computations, and demonstrated its flexibility with applications for the Poisson equation, advection-diffusion, linear elasticity, and Stokes equations. We have shown results on linear systems with as many as 137 billion unknowns, and on as many as 65,536 processors. Results were obtained from runs on four different machines: Palmetto in Clemson, S.C., SuperMuc in Munich Germany, Stampede2 in Austin, T.X., and Frontera in Austin T.X.\footnote{The following Top500 rankings as of June 2019 for each machine: Palmetto (495), SuperMUC (76), Stampede2 (19), and Frontera (5).} All functionality mentioned has been implemented inside one of two open-source codes, deal.II or ASPECT, and designed to be flexible as to maximize the usefulness for the respective communities.

Chapter 3 focused on new research pertaining to the geometric multigrid method inside deal.II. First, we gave a detailed description of the parallel implementation of GMG, complete with extensive experimentation on the efficiency of the partition of the level hierarchy. There we found that, by using the “first-child rule”, we introduces an imbalance of work among processors, but that the imbalance was nonexistent for globally refined meshes, and for adaptively refined meshes, while the imbalance was dependent on the refinement scheme, it was independent of the mesh size and only dependent on the total number of processors to a leveling off point. We also showed that the amount of data that needed to be communicated during transfer operations in the multigrid v-cycle was around 1/1000 of the total data that needed to be transfered. Using this information, we created
a model that predicted the slowdown expected from the imbalance in the partition, and showed
with scaling tests that this model correctly accounted for almost all of the slowdown experienced
in an application of a v-cycle. Scaling tests were run by Martin Kronbichler on the SuperMUC
supercomputer in Munich. Results were given for mesh sizes up to 137 billions unknowns and up
to 65,536 processors and demonstrated $O(n/p)$ scaling for a v-cycle. We then demonstrated the v-
cycle as an effective preconditioner for a linear elasticity problem on an unstructured mesh with 28
processors with adaptive refinement from a residual based estimator, showing CG iteration counts
independent of refinement and better than constant time per degree of freedom. This work was
submitted for publication, with co-authors Timo Heister, Guido Kanschat, and Martin Kronbichler,
and is currently under review.

Next in Chapter 3, we ran numerical tests on the effectiveness of cell-based Schwarz smoothers
for the advection-diffusion equation in 2D. We tested both additive and multiplicative methods again
two common point smoothers, Jacobi and SOR. We examined the effect of renumbering degrees of
freedom based on the advection-direction for multiplicative methods, and demonstrated the robust-
ness of the cell-based smoothers with higher degree finite elements. This experiment was added as
the Step-63 tutorial [35] in deal.II with co-author Timo Heister.

Chapter 4 focused on improving the current Stokes solver in ASPECT. First, we gave a detailed
description of the current AMG-based solver in the ASPECT code and discussed the details of the
new matrix-free GMG-based solver, based on the work in Chapter 3. Using several benchmarking
problems inside the ASPECT code, we demonstrated the robustness and scalability of the GMG-
based solver with different problem formulations, and compared against the AMG-based solver.
The new GMG-based solver was shown to be more robust in terms of FGMRES iterations on
the Stokes system, demonstrated a reduction in overall runtime, and improved scalability over the
AMG-based solver, as well as requiring around 2.2x less memory. Scaling tests were performed on the
new Frontera machine at The University of Texas at Austin’s Texas Advanced Computing Center.
Results were shown for both globally and adaptively refined meshes up to 27 billion unknowns and
up to 114,688 processors. The results in this chapter come from a full scale implementation of the
solver into the ASPECT code. The discussion, along with the strong and weak scaling results, were
submitted for publication with co-author Timo Heister and is currently under review.

Chapter 5 focused on testing the convergence of IDR(s) compared with GMRES($m$) for the
Stokes problem discussed in Chapter 4. We started by showing how the parallel scaling of the precon-
ditioner, matrix-vector products, and scalar products affect both GMRES and IDR(s), specifically noting the change in scaling based on the restart lengths for GMRES(m). It was concluded that a restart length of $m = 50$ (used as a default in ASPECT) was reasonable for the problem discussed, as the scaling of GMRES(50) was still dominated by the preconditioner (as opposed to the scalar products, which are expected to only scale $\log(p)$). It was then shown that, for the cheap version of the block preconditioner (4.7) (using a single GMG v-cycle for the pressure space), IDR(1) and IDR(2) had reliable convergence with similar solve times to GMRES(50), however, when using the expensive version of the preconditioner involving a CG solve on the velocity space, the convergence of both IDR(1) and IDR(2) became more unpredictable, and required stricter tolerances for the CG solve, leading to a more expensive preconditioner evaluation. At the end, it was suggested that GMRES(50) should be used inside ASPECT for situations where the memory is not limited by the storage of temporary vectors, and that IDR(2) would be useful otherwise, assuming there is no variability in the preconditioner.

The results of this thesis open further questions that would be interesting to address in the future. From the work in Chapter 3, it is unclear whether or not there is a better multigrid level cell distribution scheme that minimizes the imbalance of the partition for computations using local smoothing, or if global coarsening techniques, which were not consider in this thesis, could potentially offer a more natural and fair way of distributing the mesh (at the cost of communication and algorithmic complexity). Next, Schwarz smoothers were shown to be very effective inside the multigrid v-cycle for a non-elliptic PDE (advection-diffusion), but only for globally refined, matrix-based, serial computations. Implementations in a parallel framework with matrix-free operators are being investigated (see [107, 73]) but are far from complete, and adaptivity is still an open question. Combining Schwarz smoothing with the work in Chapter 4, an all-at-once multigrid scheme on the entire Stokes system could offer a solver for such systems that is robust for large viscosity contrasts, no longer requiring a Schur complement approximation.

In an effort to extend the work in Chapter 4 to more realistic problems, a new way of evaluating coefficient on the multigrid mesh hierarchy may need to be considered where a projection of the coefficients to a higher order finite element space could offer a more accurate solution over the cellwise constant averaging used in this thesis. There are also still computations done by some in the ASPECT community which require functionality not yet explored in the matrix-free GMG framework, namely, free-surface computations discussed in [56] and melt-transport discussed in [37], and better
Schur complement approximations which are robust with large viscosity contrasts (like those in [91]) should be explored.

Finally, when going from computations on 10,000 cores to computations on 100,000 cores (as was done in ASPECT as a part of this thesis), algorithms which had previously demonstrated efficient parallel scaling had to be reworked so that they continued to scale at the higher core counts. For example, some communication had to be restructured to avoid all-to-all communication that was previously hidden. For extending future computations to run on 1,000,000 cores and beyond, it is expected that more of this restructuring will have to occur, and new ways of handling data, specifically in the matrix-free operators, will need to be implemented.
Bibliography


