A Massively Parallel Finite Element Framework with Application to Incompressible Flows

Dissertation
zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultäten
der Georg-August-Universität Göttingen

vorgelegt von
Timo Heister
aus Hannover

Göttingen 2011

D7
Referent: Prof. Dr. Gert Lube
Korreferent: Prof. Dr. Robert Schaback
Tag der mündlichen Prüfung: 29.04.2011

Contents

Introduction **5**

1 Problem Description and Discretization **9**
- 1.1 Poisson's equation as a prototype 10
- 1.2 The structure of a finite element framework 12
- 1.3 The finite element library deal.II 14
- 1.4 Krylov methods and preconditioning 15
- 1.5 The Navier-Stokes equations 16

2 Parallel Algorithms **27**
- 2.1 Parallel paradigm 27
- 2.2 Parallel linear algebra 33
- 2.3 Parallel finite element software architecture 39

3 Solver Framework for the Oseen Problem **71**
- 3.1 The linear saddle-point system 71
- 3.2 Block-triangular Preconditioning 73
- 3.3 Schur complement preconditioning 73
- 3.4 A preconditioner using Grad-Div stabilization 75

4 Applications and Numerical Results **93**
- 4.1 Parallelization of solvers for flow problems 94
- 4.2 Turbulence modeling 95
- 4.3 Convection in the earth's mantle 101

5 Conclusions and Directions for Future Research **107**

Thanks **111**

Bibliography **113**

Introduction

The incompressible Navier-Stokes equations describe the flow of Newtonian fluids or gases. They are used in many numerical simulations of flow problems. These simulations have become an important task in research and industrial applications. There are numerous examples where the simulation of flow problems is required: indoor air ventilation to simulate cooling and air freshness in buildings, flow of a cooling medium inside devices like radiators, simulation of the interiors of cars or planes, or simulation of the natural convection in the earth's mantle. All these examples have in common that the governing equations are the incompressible Navier-Stokes equations. Otherwise these examples differ tremendously in the requirements and properties.

The Navier-Stokes equations are a system of partial differential equations of second order. The unknowns are the velocity $\boldsymbol{u}$ and pressure p that have to fulfill the equations

$$\begin{aligned} \frac{\partial \boldsymbol{u}}{\partial t} - \nu \triangle \boldsymbol{u} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} + \nabla p &= \boldsymbol{f}, \\ \nabla \cdot \boldsymbol{u} &= 0 \end{aligned}$$

in a given domain. Here, ν is the viscosity of the fluid and the external forces are given by f. A common way to conduct numerical simulations is to apply a finite element discretization to this system.

In the recent years, one can observe a trend regarding flow simulations: the demand for higher accuracy in the numerical resolution with the need to get the results faster. The underlying geometry is getting more complex and the accuracy requirements ask for discretizing with smaller and smaller cells. All this results in a dramatic increase in the problem size – and therefore computational complexity and memory requirements – that must be solved. Additionally, to achieve higher accuracy more elaborate physical models will be required in order to capture more phenomena from the real problem.

To cope with this trend, one has basically two options: First, increase the computational power, i.e., using faster and bigger computers. The requirements can typically only be met by parallel machines. In return, the finite element software must be designed to explicitly to run efficiently on a parallel machine. Second, one can try to improve the efficiency of the algorithms and solvers. This thesis combines these two approaches by deriving, implementing, and verifying a massively parallelized generic finite element framework and developing a fast and robust solver for incompressible flow problems.

The necessity to go to parallel computers can be explained as follows: Processor speeds did not increase much over the last several years and the size of the main memory in a single machine can not be increased easily over certain limits. The increase in computational power is therefore achieved by having several processors cooperate in solving a numerical problem. This has already resulted in increased number of cores in personal computers and servers sold today and this trend is likely to continue. It can also be seen in the development of larger computing clusters, where several independent machines are connected to form a more powerful super computer – the biggest machines have more than 200000 cores today.

Programs can not automatically take full advantage of a parallel machine. Instead, the routines and data structures have to be carefully designed to scale, especially in massively parallel simulations with more than thousand cores. Designing programs for numerical simulations that scale is much more difficult than writing serial programs. It is crucial to split the computation and the data structures between the machines, as every machine has its own memory. When designed in the right way, each machine only stores the information it needs for the computation. This also solves the problem that larger simulations no longer fit into the memory of a single machine. In a finite element program one has to split the computational domain into approximately equal sized chunks and distribute those between the machines.

We discretize the Navier-Stokes equations via the finite element method. In the end after linearization this requires the repeated solution of linear problems of the following saddle point form:

$$\begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{u} \\ p \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix}.$$

The solution of these linear systems of equations typically consumes the majority of the computational time. Therefore, it is of utmost importance to increase the efficiency of that solution process. Due to the size and properties of the system it is not feasible to apply generic solvers for linear systems. Instead, one has to design preconditioners to take advantage of the structure and the underlying properties of the system.

This thesis develops a new efficient preconditioner for the linear systems arising in the solution process. The massively parallel finite element framework is developed in a generic way, so the results can be applied to more than flow problems. It allows to test the scalability separately with simpler problems. Additionally, the library can be useful for a larger number of researchers. For this thesis we implemented a parallel framework in the open source finite element library `deal.II`. We show very promising results with good scalability to more than ten thousand processors. This is already included in `deal.II` [8,9], version 7.0 and allows researchers to take advantage of available computing resources and solve problems with more than a billion unknowns on adaptively refined meshes. Until now, there are only a few specialized programs that can run on that many processors, but no generic and flexible finite element program despite `deal.II`.

With the separate discussion of parallelization of finite element software and linear solvers for flow problems, naturally the question arises on how those two features can be combined. This thesis answers this theoretically and by conducting numerical simulations of realistic applications.

Combining the two topics – parallelization of finite element software and linear solvers for flow problems – leads to an interesting thesis. On the one hand, there is a lot of synergy and dependency between them. On the other hand, the research requires expertise in different research fields. Parallelization is clearly based in computer science. The second part about solvers is dominated by numerical analysis. The applications on the contrary require knowledge in engineering and physics to understand the underlying problems and equations. All this makes the research multi-layered and diverse. Combining different schools of research makes the work as a researcher in applied math very interesting, but it also is a big challenge. Looking beyond the own research field is more and more required in today's society.

The research for this thesis resulted in several published articles and conference proceedings (see [60–62,97]), some only submitted [6,63], and some are still in preparation: [80]. They were created with various co-authors and many of the results presented in this thesis can be found in one or more of these publications. The developments in the finite element library `deal.II` are available for free and so are the example programs step-32, [79], and step-40, [59], in form of tutorial programs as part of `deal.II`.

The outline of this thesis is as follows: Chapter 1 serves as an introduction to the remainder of the thesis. The structure of a finite element software package is introduced with a rather simple Poisson's equation as the model problem. The second part gives a summary about the numerical solution of the incompressible Navier-Stokes equations. Finally, the matter of Grad-Div stabilization is discussed. This part is based on our paper [97] and serves as an important aspect in Chapter 3.

Chapter 2 gives an introduction to parallelization followed by the construction of a massively parallel finite element framework. This research is in large parts published in our paper [6].

In Chapter 3 the focus is on efficient linear solvers for the Oseen problem, which is the underlying linear problem arising from the discretization of the Navier-Stokes equations. We present a new and competitive preconditioner that takes advantage of Grad-Div stabilization in the equations (see Section 1.5.5). This is in large parts based on our paper [63].

Then, the results of Chapter 2 and 3 are combined and studied with realistic applications in Chapter 4. Finally, we conclude with the accomplishments in this thesis in Chapter 5.

Acknowledgements

Timo Heister was partially supported by the German Research Foundation (DFG) through Research Training Group GK 1023.

Some of the computational results were obtained on the Ranger supercomputer at the Texas Advanced Computing Center (TACC) at The University of Texas at Austin. Some computations and the majority of code testing were done on the Brazos and Hurr clusters at the Institute for Applied Mathematics and Computational Science at Texas A&M University. Further, some computations were done using the computing resources at GWDG, Göttingen.

V2.0.2870 R (SUB)

1 Problem Description and Discretization

This chapter serves as an introduction to the main parts of this thesis. We start in Section 1.1 with the discussion of the finite element method for a prototype problem – Poisson's equation. This is standard material in finite element text books. See for example [33, 44] for more details. Therefore, we restrict the discussion to the basics.

To understand the complications involved in parallelization of a finite element program in Chapter 2 we discuss the structure of a finite element program in Section 1.2 based on the mathematical foundation in Section 1.1. We highlight the inner workings of the finite element library `deal.II` in Section 1.3 and detail all the implementations that were done for this thesis.

The linear systems that result from the finite element discretization of partial differential equations must be solved numerically. As this is non-trivial, Section 1.4 introduces the most common way to iteratively solve those linear systems: with Krylov methods. The section also serves as an introduction of the specific solvers and preconditioners for flow problems as detailed later in Section 3. Parallelization of the Krylov methods is then discussed in Section 2.2.3.

The governing equations for incompressible flow problems are the Navier-Stokes equations and we are introducing them in Section 1.5. There, we will follow with the necessary steps from the continuous formulation to the discretized form in time and space. In the end one has to solve – again – linear systems of equations. Of special importance is Grad-Div stabilization as discussed in Section 1.5.5 as it is an ingredient for efficient preconditioning described in Chapter 3. The Navier-Stokes equations will later be extended in Section 4.2 to turbulent flows. The extension to temperature coupling is done in Section 4.3.

The equations, numerical methods, and the mathematical background in the first half of this chapter are fairly standard. Section 1.5.5 highlights some results from our recent paper about Grad-Div stabilization, see [97].

1.1 Poisson's equation as a prototype

To introduce the mode of operation of a finite element problem we start with a rather simple elliptic partial differential equation (short: PDE) known as Poisson's equation. The simple structure allows the introduction of the inner workings of a finite element software while giving enough background to understand the parallelization discussed in the Chapter 2.

Poisson's equation reads

$$\begin{aligned} -\Delta u &= f \quad \text{in} \quad \Omega, \\ u &= 0 \quad \text{in} \quad \partial\Omega. \end{aligned} \tag{1.1}$$

with the unknown scalar field $u : \Omega \longrightarrow \mathbb{R}$ in the domain $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$ and homogeneous boundary conditions on the boundary $\partial\Omega$, which is done for simplicity of presentation. Δ denotes the Laplacian operator here. See Figure 1.1 for an example of a solution.

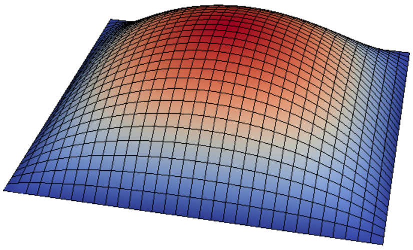


Figure 1.1: Solution of Poisson's equation on the unit square on a regular mesh with right-hand side $f = 1$. The solution u is displayed as the height of the surface.

Even though the PDE consists of one scalar field u as the unknown, it contains much of the complications found in more complex PDEs like the Navier-Stokes equations, see Section 1.5. On the one hand, it can be easily used to benchmark parallel scalability. It is suitable for that because it requires a global flow of information across the domain that is split up between several machines. For that reason we selected Poisson's equation as the first numerical example for benchmarking massive parallel scalability in Section 2.3.5. On the other hand, it also serves as a building block for solvers for the Navier-Stokes equations. The velocity-block of the Stokes problem is a vector valued Laplace operator, projection-type methods contain a pressure-Schur complement solve with Poisson's equation, and block solvers also need to solve Poisson problems for the Schur complement approximation. See Section 3.3.

Solvability and weak solutions

The variational formulation of the strong formulation (1.1) is to find a $u \in V = H_0^1(\Omega)$with

$$a(u, v) = (f, v) \;\; \forall v \in V. \tag{1.2}$$

with the bilinear form $a(u,v) := (\nabla u, \nabla v)_{L_2}$. One can show that $a(\cdot,\cdot)$ is symmetric, bounded, and V-elliptic. With the Lax-Milgram theorem (see for example [33]) we then get unique solvability of (1.2) as long as $f \in V^*$.

Finite element discretization

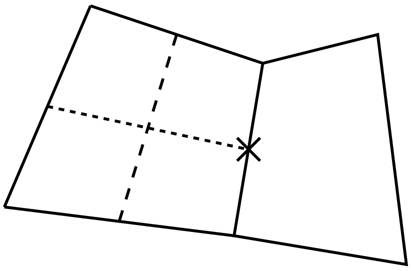


Figure 1.2: Example mesh consisting of two quadrilateral cells. If the left cell is being refined (see dashed lines), a linear ansatz space would have a hanging node (denoted with a cross).

For the discretization we restrict our discussion to quadrilateral cells in dimension $d = 2$ or hexahedral cells with six quadrilateral faces for dimension $d = 3$, respectively. The `mesh` is defined as a set of cells $\mathcal{T}_h = \{K\} \subset \mathbb{R}^d$. The reason to not discuss simplicial meshes here is that `deal.II` only supports those mentioned before, see also Section 1.3 and [8]. The standard Lagrange finite element spaces look different on simplicial meshes (see [44]), but everything else works accordingly. A difference is that one does not have to deal with hanging nodes in adaptive refinement with simplicial cells. Figure 1.2 shows an example mesh. If we were to refine the left cell into four children one would obtain a hanging node for a linear finite element space. The parallel handling of large adaptive meshes in Section 2.3.1 assumes quadrilateral or hexahedral meshes, too.

For each element $K \in \mathcal{T}_h$ we define a bi-/trilinear mapping

$$F_K : \hat{K} \longrightarrow K$$

like in Figure 1.3.

Let $\mathbb{Q}_K$ be the standard tensor product Lagrange finite element space of order

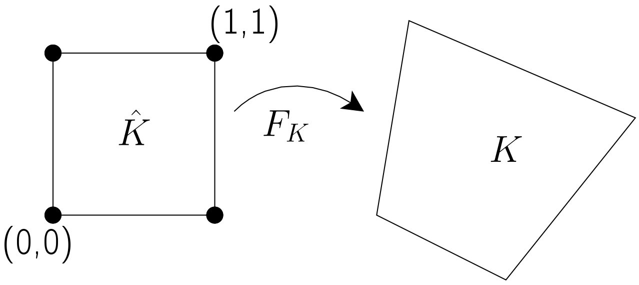


Figure 1.3: Bilinear mapping F_K from the reference cell to the element K.

$k \in \mathbb{N}$ on the reference cuboid $\hat{K} = [0,1]^d$:

$$\mathbb{Q}_k := \operatorname{span}\left\{ f : \hat{K} \longrightarrow \mathbb{R}, \boldsymbol{x} \mapsto x_1^{\alpha_1} \cdots x_d^{\alpha_d} \mid \alpha_i \in \{0,1,\dots,k\} \right\}$$

We then define the finite element space Q_k, $k \in \mathbb{N}$, which is given by

$$Q_k := \{ v \in C(\Omega) \mid v|_K \circ F_K \in \mathbb{Q}_k,\ K \in \mathcal{T}_h \}.$$

A finite element function is the image of a polynomial function on the reference cell on each cell and made continuous across cell boundaries. We denote the discrete space related to the continuous space V with

$$V_h = Q_k \cap V.$$

The discrete space is conforming, i.e., $V_h \subset V$ by construction. See [44] for more details.

Let $a_1, \dots, a_n$ be the Lagrange nodal points, then we have a nodal basis $\{\phi_1, \dots, \phi_n\}$ of V_h with $\phi_i(a_j) = \delta_{ij}$. The discrete solution $u_h \in V_h$ can be expressed as

$$u_h = \sum_i U_i \phi_i$$

with the coefficient vector $U \in \mathbb{R}^n$. With this the discrete weak problem

$$\text{find } u_h \in V_h \text{ with } a_h(u_h, v_h) = f_h(v_h),\ \forall v_h \in V_h$$

with approximations a_h to a and f_h to f can be written equivalently with the standard Galerkin method as the linear system

$$AU = F.$$

The stiffness matrix A is defined through the bilinear form as

$$A_{ij} = a_h(\phi_j, \phi_i) = (\nabla\phi_j, \nabla\phi_i),$$

and F as $F_i = f_h(\phi_i)$. Note that A here is symmetric, positive definite, and sparse. This linear system can be assembled and solved in a finite element program. We are going to discuss this in the next section.

1.2 The structure of a finite element framework

Translating the introduction in the last section into objects and tasks for the implementation of a finite element library results in the following program flow:

1. **Mesh generation.** The mesh is typically represented as a collection of cells with connectivity information. One gains flexibility by encapsulating the mesh completely from the finite element spaces. The mesh can be generated, loaded in, or manipulated by coarsening and refinement from the last mesh in the computation. It is important to have connectivity of cells, faces, edges, and corners. A large amount of memory is required per cell, which is a problem for parallel computation on big meshes. See Section 2.3.1.

2. **Distributing degrees of freedom.** From the mesh and the definition of the finite element space one can take the degrees of freedom on every cell and with this create a global numbering of all unknowns. When not the whole mesh is stored on each machine in a parallel computation, this procedure is a challenge, see Section 2.3.2.

3. **Constraints.** One way to deal with hanging nodes and special boundary conditions (like periodic or non-normal flux conditions) is to introduce an object to deal with constraints between unknowns. A constraint typically has the form

   $$x_i = \sum_j c_{ij} x_j + b_i.$$

   When assembling the linear system one can then identify restricted degrees of freedom and eliminate them. Handling constraints in a parallel program is discussed in Section 2.3.2

4. **The linear system.** Next the linear system can be set up. Storing the matrix requires a complex data structure to allow efficient handling. Storing every matrix entry is not viable, one only stores non-zero entries – typically in compressed row storage format. One way is to create the potential couplings first (e.g., by coupling all degrees of freedom in a cell with each other), and then preallocate the efficient data structure. Then the system can be assembled by looping over all active cells, calculating the local contribution from the basis functions, and transferring the entries into the global matrix and right-hand side vector. The parallel handling is discussed in Section 2.2.2.

5. **Solving.** Next the linear system can be solved with a solver and possibly a preconditioner, see Section 1.4 and Section 2.2.3 for parallelization aspects. Section 3 discusses specific solvers for flow problems and Section 4.1 is about parallelization of those.

6. **Postprocessing.** This step consists of data output or analysis, estimating the error of the solution for adaptive mesh refinement, or similar steps.

With the last step one has to either do the next iteration in a linearization scheme (see Section 1.5.4), recalculate the solution on a better adaptively refined mesh, continue to the next time step (see Section 1.5.2), or be done, depending on the problem. Note that a few complications are omitted in the description here.

1.3 The finite element library deal.II

The development of parallelization routines (Chapter 2), the research about solvers (Chapter 3), and all the numerical simulations were done using the open source finite element library `deal.II`, see [8, 9]. It is well known in the finite element community and is used in numerous projects all over the world.

The library `deal.II` is written in modern, object oriented C++ and allows flexible, and efficient programs using template programming. The types of cells are restricted to quadrilaterals and hexahedrons in two and three dimensions, respectively (also see Section 1.1). One can write mostly dimension-independent programs without paying a price in performance for that. There is support for various libraries for linear algebra (BLAS, PETSc, Trilinos), solvers, input and output, etc.. Many parts of the library are written to take advantage of modern task-based parallelization for multi-core machines using the Intel Threading Building Blocks library, cf. [101]. Alternatively there is support for MPI-based parallelization, that we worked on and are going to use in this thesis, see Section 2.1.1. There is ample documentation and many tutorial programs that highlight various aspects of the library and explain the mathematical side along with the implementation in detail.

To test the implementation and to demonstrate the success of our design we created two documented and freely available tutorial programs. In particular, we developed program step-40, [59], to show the massively parallel solution of Poisson's equation, see Section 1.1. The results in Section 2.3.5 were created with this example. We also created the more sophisticated step-32, [79], which simulates the flow in the earth's mantle, see Section 4.3, and makes use of the new massive parallel implementation and part of the solver design from Chapter 3.

While working on this thesis we did – beside bug fixes and smaller things – numerous contributions to the library:

1. Complete development of a massively parallel implementation touching many parts of the library, see Chapter 2, especially the distributed meshing from Section 2.3.1.
2. Many improvements to the coupling to the parallel linear algebra libraries

PETSc and Trilinos, also see Section 2.2.

3. New objects required for parallel computations, e.g., IndexSet (see Section 2.3.2).

4. More efficient data structures, which also improve performance in serial computations, e.g., ConstraintMatrix (for handling constraints) and SparsityPattern (for describing the stencil of a sparse matrix). We discuss part of the design also in Section 2.3.2.

1.4 Krylov methods and preconditioning

Krylov methods are a family of algorithms to solve a linear system of equations

$$Ax = b \tag{1.3}$$

for the unknowns $x \in \mathbb{R}^n$ to a given square and regular matrix $A \in \mathbb{R}^{n\times n}$ and right-hand side $b \in \mathbb{R}^n$. Krylov methods probably form the best known and fastest iterative methods to solve general linear systems. We make use of the Krylov methods for all the numerical examples and discuss the parallelization of them in Section 2.2.3.

There are many different kinds of algorithms in the family of Krylov methods: GMRES, CG, BiCGStab, BiCG, QMR, and many variants, to name a few. They were introduced in [81] and a good overview is given in [109]. All Krylov methods calculate an approximate solution x_m to the solution x in an iterative process from a starting solution x_0. The solution is usually found in the affine subspace $x_0 + \mathcal{K}_m$ of the solution space $\mathbb{R}^n$, where

$$\mathcal{K}_m(A, v) = \operatorname{span}\{v, Av, A^2v, \dots, A^{m-1}v\} \subseteq \mathbb{R}^n$$

is the Krylov space of order m for the matrix A and to the vector v. The methods do not require access to the entries of the matrix A. They only need to be able to do matrix-vector products. This is a big advantage for sparse matrices, as performance of matrix-vector products is proportional to the number of non-zero entries per row. As matrix-vector products can be parallelized easily, so can the Krylov methods, see Section 2.2.3.

The number of iterations (or the solution speed) depends on the eigenvalue spectrum of the matrix involved, (for example for GMRES, see [109]). One can attain fast performance for clustered eigenvalues away from zero. A natural way of improving convergence speed for matrices where this is typically not the case (e.g., in finite element matrices of interest) is by preconditioning the linear system (1.3). One applies a linear, regular operator P^{-1} to the system, so that the product of A and P^{-1} has an improved eigenvalue spectrum. This is the case when P^{-1} is an approximation for A^{-1}. Typically P^{-1} is not present as a matrix but just as an operator, which is the only thing that is required for the Krylov methods. The most common options for preconditioning are *left-preconditioning*, where one solves

$$P^{-1}Ax = P^{-1}b$$

instead of (1.3), and *right-preconditioning*, where one solves

$$AP^{-1}y = b$$

first, to obtain $x = P^{-1}y$ at the end. In both cases an equivalent system is being solved. Right-preconditioning with GMRES has the advantage of using the real residual $\|Ax_m - b\|$ for the stopping criterion instead of the preconditioned residual.

In a parallel application with a parallelized Krylov solver, the preconditioner has to be parallelized too. The design of parallel black box preconditioners is not the topic of this thesis, we will cover it shortly in Section 2.2.3. The preconditioner presented in Chapter 3 can be parallelized. This is discussed in Section 4.1.

As the preconditioner might involve using another iterative solver, it may in fact no longer be linear. The performance of preconditioned Krylov methods will deteriorate because they assume linearity of P^{-1}. A remedy is to apply so-called flexible Krylov methods, which can cope with that fact for a slightly more expensive iteration. We are going to use flexible GMRES or FGMRES, later. See [108] and [58] for more details about FGMRES.

1.5 The Navier-Stokes equations

This section covers the Navier-Stokes equations from the governing equations, to discretization and stabilization. It lays the foundation for the solvers in Chapter 3 and the numerical examples in Chapter 4.

This is meant as a short overview. We refer to the literature about Navier-Stokes: [52,71,107,120], and references therein.

1.5.1 The governing equations

The flow of Newtonian incompressible fluids is described by the system of the Navier-Stokes equations in a bounded, polyhedral domain $\Omega \subset \mathbb{R}^d$, $d = 2, 3$.

There, one has to find a velocity field $\boldsymbol{u} : [0,T] \times \Omega \to \mathbb{R}^d$ and a pressure field $p : [0,T] \times \Omega \to \mathbb{R}$ in the time interval $[0,T]$ such that

$$\begin{aligned} \frac{\partial \boldsymbol{u}}{\partial t} - \nu \Delta \boldsymbol{u} + (\boldsymbol{u} \cdot \nabla)\, \boldsymbol{u} + \nabla p = \boldsymbol{f} \quad &\text{in} \quad (0,T] \times \Omega, \\ \nabla \cdot \boldsymbol{u} = 0 \quad &\text{in} \quad [0,T] \times \Omega. \end{aligned} \tag{1.4}$$

Here, $\nu = \nu(x) > 0$ is the scalar, kinematic viscosity that may depend on $x \in \Omega$, and $\boldsymbol{f} \in \left[L^2(\Omega)\right]^d$ is a given source term. We assume homogeneous Dirichlet boundary conditions for presentation:

$$\boldsymbol{u} = 0 \quad \text{on} \quad [0,T] \times \partial\Omega$$

and apply the initial condition

$$\boldsymbol{u}(0,\cdot) = \boldsymbol{u}_0 \quad \text{in} \quad \Omega.$$

The equation (1.4) can be derived from the compressible Navier-Stokes equations as follows (also see [120]). A Newtonian fluid is defined by the density field ρ, the pressure field p, and the velocity u in a domain Ω. They are described by the Newtonian laws of *conservation of momentum*:

$$\rho \left(\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \boldsymbol{u} \right) - \mu \Delta \boldsymbol{u} - (3\lambda + \mu) \nabla (\nabla \cdot u) + \nabla p = \boldsymbol{f} \tag{1.5}$$

and *conservation of mass*:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \boldsymbol{u} = 0 \tag{1.6}$$

To close the system, one applies an additional law to connect ρ and p. Assuming the fluid to be homogeneous and incompressible, the density ρ becomes a constant and can be eliminated from the system (1.5),(1.6), which then reduces to the non-dimensional, incompressible Navier-Stokes equations (1.4). The assumption of incompressibility (the second equation in (1.4), stemming from the conservation of mass) is acceptable when effects compressing the fluid can be neglected, i.e., no high velocities, no shock waves.

The kinematic viscosity ν in (1.4) is connected to the *Reynolds number* Re that characterizes the behavior of the fluid through the relation

$$Re = \frac{\rho_{ref} L_{ref} U_{ref}}{\nu},$$

where ρ_{ref}, L_{ref}, and U_{ref} are reference density, length, and velocity from the nondimensionalization, respectively.

The weak formulation of the incompressible Navier-Stokes equations reads:

$$
\begin{aligned}
\frac{\partial}{\partial t}(\boldsymbol{u},\boldsymbol{v}) + (\nu\nabla\boldsymbol{u},\nabla\boldsymbol{v}) + ((\boldsymbol{u}\cdot\nabla)\,\boldsymbol{u},\boldsymbol{v}) - (\nabla\cdot\boldsymbol{v},p) = (\boldsymbol{f},\boldsymbol{v}) \quad &\text{in} \quad (0,T]\times\Omega,\\
(\nabla\cdot\boldsymbol{u},q) = 0 \quad &\text{in} \quad [0,T]\times\Omega.
\end{aligned}
\tag{1.7}
$$

One can show, that for $\boldsymbol{f} \in L^2(0,T;V')$ and $u_0 \in H^1(\Omega)$ with $\nabla\cdot u_0 = 0$ there exists a weak solution to (1.7) and it is unique for dimension $d = 2$, see [120]. Uniqueness in three dimensions is an open question.

Note we only covered the standard Navier-Stokes formulation so far. In practice modifications are used. For turbulent flows it is common to use the symmetric deformation tensor for the diffusion, see [71] for example. We discuss Grad-Div stabilization in Section 1.5.5 and turbulence models in Section 4.2 and the numerical results.

An important special case of (1.4) is the stationary case. Depending on the parameters a flow might tend to a steady state and stop changing in time, so the time derivative vanishes. For problems like the lid-driven cavity it is known that it is steady up to a specific Reynolds number. If this is known and only the steady state is of interest it is possible to directly solve for the stationary solution without running an instationary computation until it arrives at the steady state. For this one simply removes the time derivative in the momentum equations. Typically they are harder to solve than a single step of an instationary calculation. The lid-driven cavity results in Section 3.4.3 are an example. Another interesting case without a time derivative of the velocity in the momentum equation is the case of convection in the earth's mantle, see Section 4.3. There, the system is coupled with a temperature equation, but only the temperature has a time derivative, because the temporal effects of the velocity are negligible.

We continue with the discretization of the instationary Navier-Stokes equations, which involves – as in the case of Poisson's equation in Section 1.1 – the derivation of a weak formulation.

1.5.2 Time discretization

We start by semi-discretizing the continuous Navier-Stokes equations (1.4) in time. For that, the solution $(\boldsymbol{u}, p)$ and the data $\boldsymbol{f}$ are expressed only at discrete time steps $0 = t_0 < t_1 < \ldots < t_{max} = T$ of the time interval $[0, T]$, denoted by the superscript n, e.g., $\boldsymbol{u}^n$.

Primarily we consider two different discretization schemes, the typical *implicit time* *discretization* and an implicit-explicit (short *IMEX*) scheme, c.f. [3]. The fully *implicit time discretization* leads to a sequence of non-linear stationary problems of the form

$$\begin{aligned} -\nu\triangle\boldsymbol{u}^n + c\boldsymbol{u}^n + (\boldsymbol{u}^n\cdot\nabla)\boldsymbol{u}^n + \nabla p^n &= \hat{\boldsymbol{f}}, \\ \nabla\cdot\boldsymbol{u}^n &= 0, \end{aligned} \tag{1.8}$$

where $c > 0$ is a reaction coefficient related to the inverse of the time step size $\tau_n := t_{n+1} - t_n$ and $\hat{\boldsymbol{f}}$ is a modified right-hand side that contains additional terms with data from old time steps. Note that many time discretizations with different properties fit into this implicit scheme, for instance implicit Euler, BDF-2 or diagonal-implicit Runge-Kutta schemes, cf. [57]. We are going to use BDF-2 for the mantle convection in Section 4.3 and 2.3.5.

The stationary system (1.8) is still non-linear. To arrive at a linear system after spatial discretization, it must be linearized first. The typical approach is to apply a fixed-point or Newton-type iteration, see Section 1.5.4. Hence, we have to solve a sequence of linear systems for each time step with a given divergence-free field $\boldsymbol{u}^{n-1}$ in the convective term:

$$\begin{aligned} -\nu\triangle\boldsymbol{u}^n + c\boldsymbol{u}^n + (\boldsymbol{u}^{n-1}\cdot\nabla)\boldsymbol{u}^n + \nabla p^n &= \tilde{\boldsymbol{f}}, \\ \nabla\cdot\boldsymbol{u}^n &= 0. \end{aligned} \tag{1.9}$$

The iteration for the non-linearity in (1.8) implies high computational cost. An explicit time stepping is not desirable because of the strong restrictions on the time step size. A remedy is to treat the non-linear term $(\boldsymbol{u}^n\cdot\nabla)\boldsymbol{u}^n$ in an explicit way, while the remainder of the equation is kept implicit. These methods are called *IMEX-schemes*. An elegant option is to combine an explicit Runge-Kutta scheme for the convection and a diagonal-implicit scheme, as above, for the rest. With this method the system is linear after discretization and has the same structure as (1.9). We use a second order IMEX scheme for the decaying homogeneous turbulence and channel flow benchmarks, see Section 4.2.3 and 4.2.4.

Note that we treat the convection term semi-explicitly instead of fully explicitly, i.e., we use $(\widetilde{\boldsymbol{u}}^n\cdot\nabla)\boldsymbol{u}^n$, where $\widetilde{\boldsymbol{u}}^n$ is the extrapolated convection field. It gives higher accuracy than using $(\widetilde{\boldsymbol{u}}^n\cdot\nabla)\widetilde{\boldsymbol{u}}^n$.

1.5.3 Spatial discretization

If we assume a time discretization as described before – and a linearization if necessary – we are left with solving a sequence of Oseen-type problems (see (1.9)):

$$\begin{aligned} -\nu\triangle\boldsymbol{u} + c\boldsymbol{u} + (\boldsymbol{b}\cdot\nabla)\boldsymbol{u} + \nabla p &= \tilde{\boldsymbol{f}}, \\ \nabla\cdot\boldsymbol{u} &= 0. \end{aligned} \tag{1.10}$$

In a similar way as it is done for Poisson's equation we transform the system into a weak formulation:

$$\begin{aligned} \text{find } (\boldsymbol{u}, p) \in \boldsymbol{V} \times Q := [H_0^1(\Omega)]^d \times L_*^2(\Omega) \text{ with} \\ (\nu \nabla \boldsymbol{u}, \nabla \boldsymbol{v}) + ((\boldsymbol{b} \cdot \nabla)\boldsymbol{u} + c\boldsymbol{u}, \boldsymbol{v}) - (\nabla \cdot \boldsymbol{v}, p) = (\boldsymbol{f}, \boldsymbol{v}) \\ (\nabla \cdot \boldsymbol{u}, q) = 0 \\ \text{for all } (\boldsymbol{v}, q) \in \boldsymbol{V} \times Q. \end{aligned} \tag{1.11}$$

Here, the natural pressure space $L^2(\Omega)$ is normalized as $L_*^2(\Omega) := \{v \in L^2(\Omega) \mid \int_\Omega v \; dx = 0\}$, because the pressure is otherwise only determined up to a constant.

We can rewrite the weak system as:

$$\begin{aligned} a(\boldsymbol{u}, \boldsymbol{v}) + b(\boldsymbol{v}, p) &= (\boldsymbol{f}, \boldsymbol{v}) \qquad \forall \boldsymbol{v} \in \boldsymbol{V} \\ b(\boldsymbol{u}, q) \qquad\quad &= 0 \qquad\qquad\; \forall q \in Q \end{aligned} \tag{1.12}$$

with the bilinear forms

$$\begin{aligned} a(\boldsymbol{u}, \boldsymbol{v}) &:= (\nu \nabla \boldsymbol{u}, \nabla \boldsymbol{v}) + ((\boldsymbol{b} \cdot \nabla)\boldsymbol{u} + c\boldsymbol{u}, \boldsymbol{v}), \\ b(\boldsymbol{v}, p) &:= -(\nabla \cdot \boldsymbol{v}, p). \end{aligned}$$

Using the inf-sup or Babuska-Brezzi condition

$$\inf_{q \in Q} \sup_{\boldsymbol{v} \in \boldsymbol{V}} \frac{(q, \nabla \cdot \boldsymbol{v})}{\|q\|_0 \|\boldsymbol{v}\|_1} \geq \beta > 0 \tag{1.13}$$

between the spaces for velocity and pressure one can show unique solvability of (1.11) or (1.12), see [107] for example.

When moving from the continuous spaces $\boldsymbol{V}$, Q to discrete finite element counterparts $\boldsymbol{V}_h \subset \boldsymbol{V}$ and $Q_h \subset Q$, the inf-sup condition (1.13) has an analog discrete counterpart with the spaces $\boldsymbol{V}_h$ and Q_h. This compatibility of the discrete spaces is also relevant in numerical calculations, as spurious oscillations will destroy the solution, see [107]. Taylor-Hood elements – elements with continuous velocity and pressure, where the velocity degree is one order higher than the pressure – fulfill the discrete inf-sup condition

$$\inf_{q \in Q_h \setminus \{0\}} \sup_{v_h \in \boldsymbol{V}_h \setminus \{0\}} \frac{b(\boldsymbol{v}_h, q_h)}{\|\boldsymbol{v}_h\|_{\boldsymbol{V}} \|q_h\|_Q} \geq \beta_h > 0,$$

see [52] for the proof.

As in Section 1.1 we consider quadrilateral or hexahedral meshes $\mathcal{T}_h$. We denote the discrete spaces with $\boldsymbol{V}_h = [Q_{k+1}]^d \cap \boldsymbol{V}$ and $Q_h = Q_k \cap Q$, where Q_k are the tensor-product polynomials as defined in Section 1.1.

This stable discretization leads to a finite-dimensional, linear saddle point system

$$\begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix} \tag{1.14}$$

with finite element matrices A containing diffusion, reaction and convection and the pressure-velocity coupling B. Note that these are the discretized counterparts to the bilinear forms a and b from above.

A different option would be to take an element pair that does not fulfill the inf-sup condition – for example choosing the same element order for velocity and pressure – and stabilize the system with Pressure Stabilized Petrov Galerkin (PSPG) methods [69] or Local Projection Stabilization (LPS) [19]. We will not discuss that option here, though.

We discuss efficient solvers for the linear system (1.14) in Chapter 3. This is done by solving the linear saddle point system with a Krylov solver. The efficient solution requires special preconditioners tailored to the problem. Parallelization is discussed in Section 4.1.

The solver and preconditioner concept in Chapter 3 works identically with additional, symmetric stabilization terms for the velocity, like local projection stabilization, see [19, 90, 92] and references therein. We omit the definition and the terms for clarity of presentation. The addition of non-symmetric stabilization like Streamline Upwind/Petrov Galerkin (SUPG) methods (see the overview in [107] and [69, 72, 91]) is non-trivial, as it spoils the saddle point structure of the problem.

1.5.4 Linearization

As mentioned in Section 1.5.2, approaching the time discretization of the Navier-Stokes equations with a fully implicit method results in a sequence of stationary, non-linear problems (1.8) to solve. In practice, one applies the finite element discretization from the previous section to that stationary, non-linear problem, and then linearizes the discretized problem.

To formulate the linearization we write it using an abstract non-linear operator $F : \mathbb{R}^n \longrightarrow \mathbb{R}^n$, where n is the dimension of the finite element spaces of velocity and pressure. Evaluating the operator with an argument (b, r) returns the solution of the linear system with convection b. Thus, we are looking for a fixed point

$$F(x) = x.$$

This system can be solved using a Picard iteration or Newton's method, see [71].

The Picard iteration can be written in defect-correction form as

$$x^{k+1} = x^k + \left(F(x^k) - x^k\right).$$

Newton's method is attractive if the derivatives of all terms in the discretization are available, which is not always the case. The iteration for Newton's method is then

$$x^{k+1} = x^k - \left[DF(x^k)\right]^{-1} F(x^k).$$

Either way, one is left with an iteration

$$x^{k+1} = x^k + G(x^k).$$

As convergence of this iteration is only guaranteed for contracting operators, some globalization method might be required. This is necessary for stationary problems in particular. In the stationary lid-driven cavity problem, Section 3.4.3, we are using an adaptive back-tracking procedure, where an $\alpha_k \in (0,1]$ is selected so that the update

$$x^{k+1} = x^k + \alpha_k G(x^k)$$

has a smaller residual, i.e., $\|G(x^{k+1})\| < \|G(x^k)\|$, otherwise the step is rejected and repeated with a smaller α_k. Note that evaluating $\|G(x^{k+1})\|$ is expensive, but it is required in the next iteration anyhow.

1.5.5 Grad-Div stabilization

Additional stabilization for the Oseen problems is useful and often required even for inf-sup stable elements, especially with dominating convection. Recently, the so-called grad-div stabilization has gained attention as a pressure sub-grid modeling and as a least-square type stabilization. We developed solvers than can cope and take advantage of Grad-Div stabilization present in the Oseen problem, see Chapter 3. We refer to our paper [97] for the discussion of parameter design, further analysis, and numerical examples.

Grad-Div stabilization results in a modified bilinear form

$$\tilde{a}(\boldsymbol{u},\boldsymbol{v}) := a(\boldsymbol{u},\boldsymbol{v}) + (\gamma\nabla\cdot\boldsymbol{u}, \nabla\cdot\boldsymbol{v}),$$

which is used instead of $a(\cdot,\cdot)$ with a parameter γ that is to be determined. Thus, we end up with the following discrete system: Find $(\boldsymbol{u}_h, p_h) \in \boldsymbol{V}_h \times Q_h$ such that

$$\begin{aligned} \tilde{a}(\boldsymbol{u}_h,\boldsymbol{v}_h) + b(\boldsymbol{v}_h, p_h) &= (\boldsymbol{f},\boldsymbol{v}_h) \qquad &&\forall \boldsymbol{v}_h \in \boldsymbol{V}_h \\ b(\boldsymbol{u}_h, q_h) &= 0 &&\forall q_h \in Q_h. \end{aligned} \tag{1.15}$$

The parameter $\gamma \geq 0$ is assumed to be constant per element or a global constant. The stabilization improves the numerical accuracy of the solution and helps to reduce spurious oscillations for convection dominated flow.

The parameter choice is non-trivial and we discuss different designs in [97]. In general the optimal value for γ depends on the solution on the element K. For sufficiently smooth solutions the following formula can be derived:

$$\gamma_K \sim \max\left\{ \frac{|p|_{H^k(K)}}{|\boldsymbol{u}|_{H^{k+1}(\tilde{K})}} - \nu, 0 \right\}. \tag{1.16}$$

Note that $\tilde{K}$ is a neighborhood of the element K. In practice, the unknown continuous solution $(\boldsymbol{u}, p)$ in (1.16) has to be replaced by their discrete counterpart $(\boldsymbol{u}_h, p_h)$. Even then, it represents an expensive non-linear model. Additionally, evaluating the norms of $\boldsymbol{u}_h$ and p_h is costly, since higher order derivatives are involved. Therefore a common assumption is $|p|_{H^k(K)} \approx |\boldsymbol{u}|_{H^{k+1}(K)}$ and $\nu \ll 1$, which results in a constant design $\gamma \sim 1$. Unfortunately, this assumption is not valid in general. For example, in the case of a laminar channel flow there holds $|p|_{H^k(K)} \approx \nu|\boldsymbol{u}|_{H^{k+1}(K)}$ and therefore $\gamma \sim \nu$.

In the simpler design of a constant γ for the whole domain, one assumes that the flow is similar everywhere. The optimal constant value is a trade-off between mass and energy balance of the system. The results in [97] show, that the static design is often good enough, but a problem dependent constant is necessary to attain optimal results. Fortunately, the observation that $\gamma \in [0.1, 1]$ is not overstabilizing and is improving accuracy over no Grad-Div stabilization seem to hold true.

Numerical dissipation

Grad-Div stabilization introduces numerical dissipation, so it is of interest to quantify that dissipation. As explained in [97], the rate of numerical dissipation can be measured as $\operatorname{diss}(\boldsymbol{u}_h(t))$, with the ratio

$$\operatorname{diss}(\boldsymbol{v}) = \frac{\|\gamma^{1/2}\nabla \cdot \boldsymbol{v}\|^2}{\|\nabla \boldsymbol{v}\|^2},$$

where $u_h(t)$ is the spatially discretized solution to the Navier-Stokes equations before time discretization. This dissipation rate can be bounded by

$$\mu_h^2 \leq \operatorname{diss}((\boldsymbol{u}_h(t)) \leq M_h^2,$$

where

$$\mu_h^2 = \inf_{\boldsymbol{v} \in \boldsymbol{V}_h^0} \operatorname{diss}(\boldsymbol{v}), \qquad \text{and } M_h^2 = \sup_{\boldsymbol{v} \in \boldsymbol{V}_h^0} \operatorname{diss}(\boldsymbol{v})$$

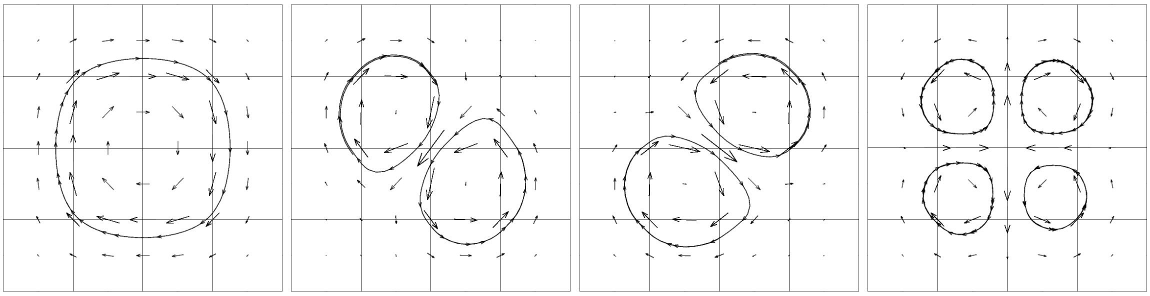


Figure 1.4: Examples for true divergence free functions for Q_2 with $h = 1/4$

and V_h^0 is the space of discretely divergence free functions, i.e.,

$$\boldsymbol{V}_h^0 = \{\boldsymbol{v}_h \in \boldsymbol{V}_h \mid (\nabla \cdot \boldsymbol{v}_h, q_h) = 0, \forall q_h \in Q_h\}.$$

Because the finite element space with order ≥ 2 contains true divergence free functions in $\boldsymbol{V}_h^0$, (see Figure 1.4), μ_h^2 then becomes zero and is of no use for bounding $\mathrm{diss}(\boldsymbol{u}_h(t))$. Let G be the finite element matrix of the assembled divergence term. We will then restrict the infimum to functions that are not in the kernel of the discrete divergence matrix G:

$$\bar{\mu}_h^2 = \inf_{\boldsymbol{v} \in \boldsymbol{V}_h^0 \setminus Ker(G)} \mathrm{diss}(\boldsymbol{v}).$$

Let P be the L^2-orthogonal projection from $\boldsymbol{V}_h$ onto $\boldsymbol{V}_h^0$ and assume $\gamma = 1$. Therefore, the bounds $\bar{\mu}_h$ and M_h are the minimal and maximal non-zero eigenvalue of the system

$$\begin{pmatrix} G & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \lambda \begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix}.$$

For a more detailed explanation, see again [97].

See Figure 1.5 for eigenvalue spectra for the eigenvalue problem above. Table 1.1 finally shows calculated values for the dimension of the divergence free subspaces and bounds for the dissipation rate. From these experiments one can conclude that the dissipation does only slightly depend on the mesh size, while the element order has a much bigger influence. The number of truly divergence free function increases (starting with none for Q_1). Dissipation reaches an order of one on very few scales, while roughly 20 percent of the scales are significantly influenced.

The Grad-Div stabilization with a constant parameter design will be the main building block of the preconditioner in Section 3.4. The advantage of applying Grad-Div stabilization to various problems is clearly visible in the numerical examples in that section.

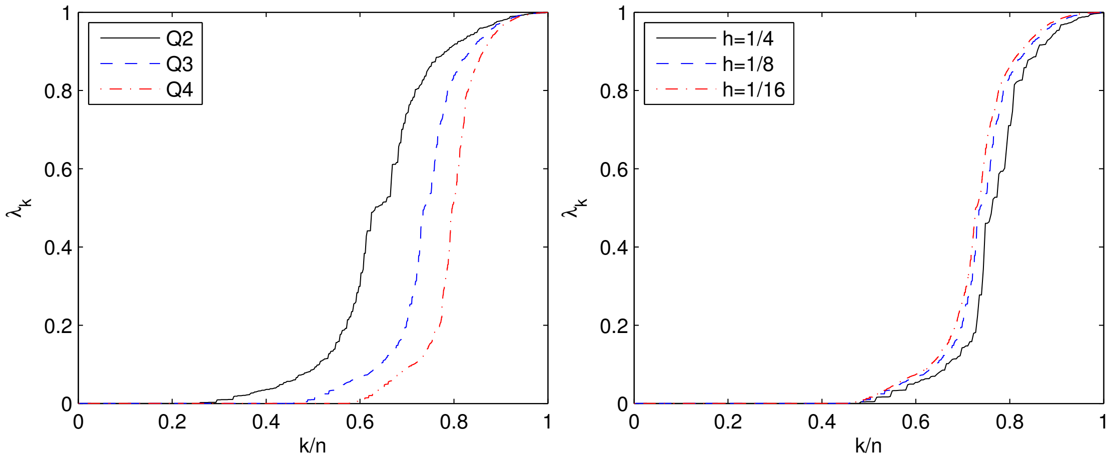


Figure 1.5: Eigenvalue spectra for varying element order and fixed mesh with $h = 1/8$ *(left) and for* Q_3 *on different meshes (right).*

Table 1.1: Dimension of truly div-free functions subspace of $\boldsymbol{V}_h^0$ ($\boldsymbol{G}$=ker G) *and bounds for numerical dissipation on its orthogonal complement subspace in* $\boldsymbol{V}_h^0$.

h	1/4	1/8	1/16	1/32
	Q_1			
dim($\boldsymbol{G}$) / dim($\boldsymbol{V}_h$)	0 / 18	0 / 98	0 / 450	0 / 1922
μ_h^2	4.8812e-2	1.1864e-2	2.9564e-3	7.3869e-4
	Q_2			
dim($\boldsymbol{G}$) / dim($\boldsymbol{V}_h^0/\boldsymbol{G}$)	4 / 94	36 / 414	196 / 1726	900 / 7038
$\tilde{\mu}_h^2$	8.9596e-3	2.4653e-3	6.3427e-4	1.6002e-4
M_h^2	9.9302e-1	9.9961e-1	9.9998e-1	1.0000e-0
	Q_3			
dim($\boldsymbol{G}$) / dim($\boldsymbol{V}_h^0/\boldsymbol{G}$)	36 / 206	196 / 862	900 / 3518	* / *
$\tilde{\mu}_h^2$	4.3242e-3	1.0953e-3	2.7497e-4	6.9336e-5
M_h^2	9.9876e-1	9.9992e-1	9.9999e-1	1.0000e-0
	Q_4			
dim($\boldsymbol{G}$) / dim($\boldsymbol{V}_h^0/\boldsymbol{G}$)	100 / 350	484 / 1438	2116 / 5822	* / *
$\tilde{\mu}_h^2$	2.4296e-3	6.1053e-4	1.5289e-4	3.8241e-5
M_h^2	9.9955e-1	9.9997e-1	1.0000e-0	1.0000e-0

2 Parallel Algorithms

After motivating the need for parallelization, we set up the terminology of parallel computations used later. The basics about parallelization and the different computer architectures and their implications to parallel programming are explained in 2.1. The content there might seem basic, but many of it is non-standard and is learned from experience. Nevertheless there are some good introductory resources about parallelization and MPI, see e.g. [98, 100, 109]. We conclude with the important tenets for parallel software design that is relevant for this thesis in Section 2.1.4. Probably the most important aspect in a parallel finite element program is the handling of the linear algebra, which is explained in 2.2. There we also extend the Krylov methods introduced in 1.4 for usage in parallel computations. The sections 2.1 and 2.2 do not present major new developments, but are required to understand the design of a parallel finite element program.

One of the two main research topics of this thesis is then given in 2.3. Here we derive data structures and algorithms for generic finite element software to run on thousands of processors. It details the research done while developing the parallel architecture of the library `deal.II`, but strives for general applicability. We build on the introduction of finite element software in general (see Section 1.2) and of `deal.II` in particular (see Section 1.3). The Section 2.3 is in large parts published in [6].

2.1 Parallel paradigm

The reason for the existence, development, and spreading of the usage of parallel computers or clusters can be explained with two reasons. First, the demand to calculate bigger and bigger problems for example in astrophysics, earth sciences, etc. combined with the urge to get answers faster at the same time, together increase the demand for faster computers. Second, the development of a single processor is more or less stalled. The processor frequencies did not increase further over the last years and it looks like the physical barriers that are the cause will not be circumvented any time soon, see [118] for details. The remedy was to let several (today even more than a hundred thousand) processors work on one problem at the same time.

This increase in parallelization is happening in two directions. First, the number of computing cores in a single computer is increasing. Second, the computing clusters that combine many machines with a fast network consist of growing numbers of machines since. The number of cores on a single machine is still increasing, but as memory is expensive and can not be increased easily above a certain amount in a single computer, the connection of independent computers in a cluster is unlikely to fade away. Therefore, limiting the development only to multi-core (in a single machine) parallel development is not advisable today. The development of parallel programs that run on computing clusters must deal with the fact, that each computing node has its own memory. This distributed memory architecture implies that the only way to communicate between two nodes is by sending messages from one node to another over the high speed network in the cluster, see Figure 2.1.

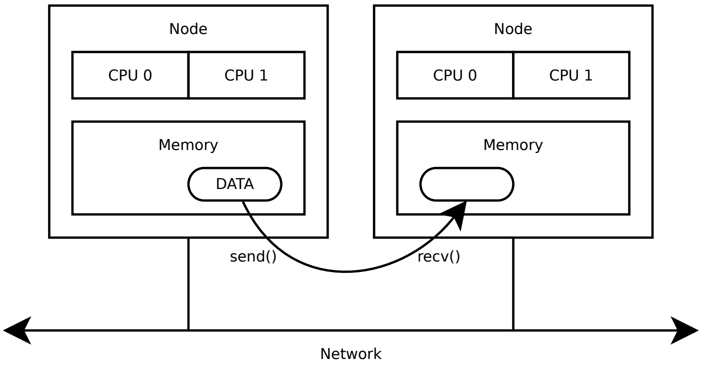


Figure 2.1: Distributed memory cluster. The only way to communicate is by sending messages between machines.

The most common programming model for sending messages between nodes is the message passing interface, MPI [94]. We will discuss it in further detail in Section 2.1.1.

While clusters are becoming more and more common, the number of codes that efficiently utilize these machines is relatively limited. This is mainly due to two reasons: First, existing legacy codes are difficult to parallelize to these massive numbers of processors since data structures have to be conceived entirely differently. Second, algorithm and data structure design is not trivial when the goal is to exploit machines of this size in a scalable way. On the other hand, codes that scale to the largest available machines are almost exclusively purpose-built for individual applications. For example, the codes SPECFEM3D [31], CitcomS [119], and Rhea [26] have been written for particular geophysical applications and are not based on general-purpose finite element libraries. The reason, of course, is that none of the libraries widely used in academic and applied finite element simulations – such as PLTMG [12], DiffPack [23,82], libMesh [76], Getfem++ [102], OOFEM [99], FEniCS/DOLFIN [86,88], or `deal.II` up to version 6.x [8,9] – support massively parallel computations that will run on thousands of processors and routinely solve problems with hundreds of millions of cells and several billion unknowns. This notwithstanding, there clearly is a demand for general-purpose finite element libraries supporting such computations through a relatively simple, generic interface.

We will discuss the requirements and solutions for a generic finite element library that does scale to large clusters in Section 2.3.

2.1.1 The MPI programming model

The usage of MPI is not limited to clusters, but also works on a single multi-core machine. In fact, one node in a computing clusters has more than one core too. This makes the development described in this thesis relevant from multi-core computers that typically stand under your desk today already, up to computing clusters with thousands of cores. An MPI library typically launches the same program several times on different computers or cores at the same time. Each individual so called process (typically a Linux process each) can then communicate with the other processes. They are identified with a unique number $0, \ldots P-1$ (if the number of processes is P). In MPI terminology this is called “rank”, see [94].

In the “pure” MPI model each core in a node of a cluster is treated as its own MPI node. Therefore, a process is launched for each (virtual) core of each machine. This means processes on the same machine also communicate only by exchanging messages. At first sight this looks like a huge disadvantage as they do not take advantage of the shared memory. Note that passing messages between processes on a single node is very fast as they only involve a simple memory copy. You can take advantage of the shared memory on a computing node with the so called hybrid parallelization model, which is not used in this thesis (but can be applied to most results). In that model a single machine or node, which shares the memory between all its cores, is parallelized with a shared memory library like OpenMP or TBB (see [36,101]) and only the communication between such nodes is done using again MPI. The advantages of using such a model are not clear, as every part (that includes external libraries too) of the program must run efficiently on a multi-core machine to take advantage of the hybrid model. This makes it inherently difficult to take advantage of that hybrid parallelization model. For example at the point where this thesis was written, neither PETSc (c.f. [4,5]) nor Trilinos (c.f. [66,67]), which we use for linear algebra supports multi-threaded operations, could do that. As the linear algebra takes a huge part of the runtime we will not pursue the hybrid model further. A pure MPI model has the advantage of being a lot simpler to implement as dealing with concurrency is a hard problem in computer science. The NUMA architecture also works in favor of a pure MPI model. A single computer can have several sockets with a CPU with several cores and memory for each of those sockets. The whole memory is accessible for all cores, but access is much faster on memory "owned" by each core. Exactly one MPI process reads and writes from a buffer (remember processes do not share memory), so that buffer can be allocated in a corresponding "fast" memory bank. Doing this in a multi-threaded application is not straight forward. Here typically the thread who first touches a memory page triggers the allocation in its local and fast memory. Which method is preferred depends on the type and amount of computation versus communication. For a study about this topic, see [29].

That non-withstanding the approach in this thesis does not become obsolete if it becomes feasible to use a hybrid model for parallelization. On the contrary many parts in `deal.II` can optionally run multi-threaded already. Additionally, the amount of computation and storage for the mesh partitioning would immediately decrease in an hybrid model allowing bigger problems and better efficiency. For the presentation we will assume a "pure" MPI model and from now on not differentiate between cores, nodes, processes and threads and use "process" to describe a single instance of the program. The communication routines supplied by MPI range from simple messages (with MPI_send and MPI_recv) where only two processes are involved to collective operations with more or all processes. This is used for example to send data from one to every other process or to do collective operations like calculating the maximum or sum of some values. These collective operations are typically a lot more efficient than sending messages by hand, as they often use a tree pattern to communicate (think snow ball system), which results in logarithmic complexity. How each message is transfered is transparent to the user (be it via shared memory, Ethernet, infiniband or something else). The MPI implementations automatically pick the fastest protocol available.

An important class of MPI operations for the scalability and performance later on are the so called "non-blocking operations". There are non-blocking variants of many operations in MPI, they all return the control to the caller immediately and do the data transfer in the background, so the CPU can return to work on local operations. This is in contrast to the standard – also called blocking – routines, that wait until the transfer is done. Later one can poll if data from others is already available. One can interleave local computations then. The non-blocking operations are very important, as they can completely hide the latency of the communication if the program is written in a way that local computations can be interleaved. Examples for non-blocking communication are matrix-vector products (see 2.2.2) and enumerating degrees of freedom in Section 2.3.2.

2.1.2 Parallel Complexity

It is worth to consider a mathematical model to analyze parallel scalability. This has already been done in [2] many years ago. If you split up your program runtime into a perfectly parallelized fraction $E_p \in [0, 1]$ and in the serial part $E_s = 1 - E_p$ the `speedup` of the program with n processors under ideal conditions is given by

$$\text{speedup}(n) = \frac{1}{1 - E_P + \frac{E_P}{n}}.$$

This highlights the importance of reducing the amount of serial code, because even with arbitrarily many machines, the maximum speedup is given by

$$\text{maxspeedup} = \frac{1}{1 - E_P},$$

which is finite if $E_p < 1$.

Note that time spent in communication does count to the serial part E_s, as a transfer from processor A to B does not get faster the more machines are involved in the whole program.

The `speedup` gives a theoretical limit that can only be beaten in rare cases due to caching effects. Therefore the speedup is usually used as a goal to compare benchmarks with.

To analyze the parallel efficiency in practice there exist two common scalability tests: weak scalability and strong scalability. In weak scalability, the number of processors is kept constant, while the problem size is increased step-by-step. Strong scalability works the other way around: the problem size is kept constant, while the number of processors is increased step-by-step. In both cases one compares to the optimal linear scaling as explained above.

2.1.3 GPUs, vectorization, and other trends

There are a few trends that came up in the recent years like GPUs and support for vectorization in CPUs. The thesis is concerned with the algorithmic aspects of parallelization and do not cover these trends. We want to justify this here.

Graphics processing units (GPUs) supply a machine with a secondary processor that is on the one hand more limited than the CPU but has many more threads than a typical CPU on the other hand. GPUs are very powerful for floating point operations that can be vectorized and can gain from single instruction multiple data parallelization. The weaknesses are the slow transfer from the main memory to the GPU and back, and the limited advantage when it comes to code that is heavy on branching (e.g., logic with a lot of if-then-else). When looking at the structure of a finite element program it is fairly obvious that only solving and assembling lend itself for implementing it on the GPU. We won't discuss this interaction in more detail since we use external packages as black box solvers to solve our linear systems. Once these packages learn to use GPUs, our programs will benefit as well.

Vector processing or vectorization are powerful instructions on CPUs and also GPUs that act on a larger amount of data at the same time. They work on a finer scale in CPUs. Their use is, again, primarily confined to the external solver packages. At the same time, vectorization is only possible if data can be streamed into processors, i.e., if data is arranged in memory in a linear fashion. `deal.II` goes to great lengths to arrange "like" data in linear arrays rather than scattered data structures, and in the same order in which cells are traversed in most operations; consequently, it has been shown to have a relatively low cache miss rate compared to other scientific computing applications (see, for example, the comparison of SPEC CPU 2006 programs – including `447.dealII` – in [64]). Similarly, we enumerate degrees of freedom in such a way that vector entries corresponding to neighboring degrees of freedom are adjacent in memory. This ensures low cache miss rates; not by coincidence, p4est uses a space filling curve to enumerate cells. This ensures this property very well, as does our algorithm to assign degrees of freedom to individual processors (see Section 2.3.2, especially Remark 2).

In summary, the current lack of support for hybrid and GPU-accelerated programming models in widely used external solver packages prevents us from using such approaches currently in `deal.II`. At the same time, several of the algorithms discussed can efficiently be parallelized using multiple threads once this becomes necessary. Finally, our numerical results in Section 2.3.5 show that our methods scale well on a contemporary supercomputer and that the limits of the "flat" MPI model have not yet been reached.

2.1.4 Tenets for parallel code

Here we will sum up the principles for designing a software to scale to a large number of processors:

1. **Distributed storage.** Only store the relevant data necessary for the computation. It is acceptable to trade that for some increases in communication. Do not store data proportional to the problem size.

2. **Fair memory distribution.** An imbalance in the amount of memory stored not only poses problems with the amount of main memory available but also indirectly causes an imbalance in computational cost as memory needs to be traversed.

3. **Avoid serial computations.** Serial computation that is either done redundantly on all machines or with others waiting will lead to a serious bottleneck. Work must be split up between processors.

4. **Fair distribution of computations.** An imbalance in computational complexity will result in sub-par performance, because the slowest machine will determine the scalability.

5. **Hide communication.** Communication carries latency that does not scale with the number of processors, as it remains constant. Prefer nonblocking communication to hide that cost.

6. **Avoid global communication.** All to all communication starts to be a serious problem with a few hundred processors. Global reduces are often unavoidable but have to be used as scarce as possible, see also Section 2.2.2.

7. **Efficient data structures.** As the problem sizes can and will increase dramatically with additional computational power, one has to pay special attention to incorporate efficient data structures.

2.2 Parallel linear algebra

A finite element software package requires linear algebra objects and routines. This is because functions discretized with finite elements are usually represented as vectors of coefficients and the PDEs to solve are normally expressed as linear systems of equations – as long as they are not treated with an explicit time discretization scheme for example. Even with an explicit time discretization scheme one often wants to solve accompanying equations like for example the pressure update in flow problems.

Supplying routines to set up vectors and matrices, and solve the linear systems, is done with linear algebra routines. Parallelization is crucial for the two obvious reasons: Linear systems can become quite large, as the dimension of the matrix is the number of unknowns in the finite element discretization. The dimension of a linear system can be as large as a billion or even more. One machine can thus no longer store the whole system in memory. Distributing the storage between different machines solves that problem and allows a lot bigger problems to be solved than on a single machine. The second reason is that most time in a finite element program is usually spent solving the linear system. The next most time consuming part is assembling the necessary matrices. Therefore the easiest way – and a necessity – is to speed up a calculation with parallel assembly and a parallel solution process of the resulting linear system.

How the storage of vectors and matrices is distributed is explained in the next section. After that we will describe on how important operations on vectors and matrices can be realized and how one can solve the linear systems using those operations.

The following descriptions are fairly standard and implementations are readily available in parallel linear algebra libraries like PETSc or Trilinos. The library `deal.II` for example just interfaces to PETSc or Trilinos with thin wrappers. This makes all operations, solvers, and preconditioners from the libraries available to the user of `deal.II`. Nevertheless the discussion here gives the necessary background to fully understand the parallelization of the whole finite element software.

2.2.1 Memory distribution

The size of the matrices and vectors correspond to the number of unknowns or degrees of freedom of the finite element space. For the explanation here we assume we are dealing with a single finite element space. Coupled problems with different finite elements – e.g., velocity and pressure for an incompressible flow problem – are typically treated as block matrices and block vectors. The blocks are composed of the non-block objects, we will touch on how that works in the next section and restrict our discussion to matrices and vectors corresponding to a single finite element space. Note that matrices do not need to be square when they are corresponding to two finite element spaces of different order, e.g., the coupling matrix B between velocity and pressure for an Oseen type problem.

Vectors and matrices are typically split up row-wise and stored on the different processes. This way each row belongs to exactly one processor. The splitting is done so that each machine has roughly the same number of unknowns to balance the work and memory requirements. The distribution is usually done in a way that each machine gets a contiguous range, i.e., the first processor owns the indices 0 to k_1, the second $k_1 + 1$ to k_2, etc.. This makes local storage and access a lot easier and more efficient, see Figure 2.2. One can – for example – determine if the indices are owned by this processor or not with simple comparison of the start and end index. When one makes the values k_i available on each machine, on can cheaply determine who the owner of a specific row is on each machine.

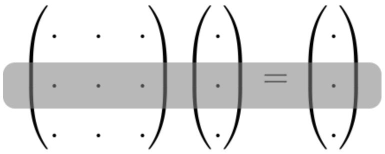


Figure 2.2: Sketch of the distribution of rows of vectors and matrices between processors.

Because a matrix stemming from a finite element discretization is sparse one typically employs a compressed row storage format so that the matrix elements containing zeros are not stored at all. The treatment of the so called sparsity pattern is explained in some detail in 2.3.3.

The distribution of the rows and thus the degrees of freedom also give rise to an interpretation as a domain decomposition method. Each degree of freedom belongs to a cell in the computational domain. A distribution of the degrees of freedom over the processors thus gives a geometric distribution of the computational domain. As degrees of freedom that are close to each other, i.e., both belonging to the same or neighboring cells, often couple and influence each other, it makes sense to try to bring those together on the same processor. This reduces communication between the machine and therefore leads to a much better performance than just randomly assigning ownership. Most importantly all degrees of freedom belonging to a particular cell should live on the same processor. The reasoning is very simple: each processor only needs to visit the cells that contain degrees of freedom that are stored locally when assembling the linear system. Therefore one usually also distributes the cells of the mesh accordingly to match the distribution of the degrees of freedom. Note that there is no one-to-one relationship between degrees of freedom and cells, as they can lay on the boundary between two cells and are thus shared.

The way it is done in `deal.II` is as follows. In the first step the cells of the mesh are evenly distributed between the processors. Here we pay attention to try to minimize the number of faces or edges to other processors. This reduces the amount of communication required later on. The degrees of freedom in the inner part of each cell get assigned to the owner of the cell and degrees of freedom on the interface are assigned to one or the other processor. A renumbering of the indices then ensures contiguous ranges of rows. This scheme can be seen as an algebraic variant of a domain decomposition method as explained above.

Partitioning the mesh is non-trivial, especially if it consists of a large number of cells. One can formulate the partitioning as an optimization problem on a graph, where determining the optimal solution is NP hard. Each cell is represented as a vertex in the graph, and two vertices get connected with an edge if the corresponding cells share a face. One can now partition the vertices of the graph in equally sized groups while reducing the number of edges cut. Of course approximate solutions to that optimization problem are a good alternative, too. There are software packages like METIS (see [74]), that do a very good job at doing exactly that. Note that the graph is smaller than the number of degrees of freedom, especially with higher order elements one can have hundreds and more degrees of freedom per cell. Nevertheless it is still an expensive operation especially in a parallel computation. We explain a different, very efficient approach in 2.3.

We will see in 2.2.2 that the number of faces that are shared between two processes determines the number of communication that is required in matrix-vector products. One can also think of shared degrees of freedom as the number of unknowns where information about the solution of the PDE “flows” between the machines.

The equal distribution of the number of cells is more important than the perfect equilibrium of the number of degrees of freedom. This is the case even though the number of degrees of freedom stored locally determines the amount of memory required. The number of cells on the other hand determines the computational effort being done. For example assembling the linear system is done cell-wise, so the time necessary for assembling is proportional to the maximum number of cells owned by a process. We explain in 2.3.3 how the assembly process is done.

2.2.2 Important operations

Many operations on vectors like building linear combinations are simple to implement, because they do not require communication between the nodes.

For Krylov methods (see 2.2.3) we additionally require scalar products and matrix-vector multiplications. Surprisingly, scalar products can become slower than matrix-vector products depending on the number of processes and the type of the interconnect involved. Some of the following is also explained in [109].

Scalar products consist of a fast local computation and then a global reduction operation. The global reduction requires $\log(P)$ operations and can not be interleaved with local computations.

A matrix-vector product $A \cdot x$ on the other hand consists of a much bigger local computation. Let L, D, R be the local matrix rows (D refers to the diagonal block) and x_{loc} the local entries of the vector of the product in the following form:

$$\begin{pmatrix} \ddots & & \\ L & D & R \\ & & \ddots \end{pmatrix} \begin{pmatrix} \vdots \\ x_{\text{loc}} \\ \vdots \end{pmatrix} =: \begin{pmatrix} \vdots \\ y_{\text{loc}} \\ \vdots \end{pmatrix}$$

The calculation of y_{loc} can be split up into:

$$y_{\text{loc}} = \begin{pmatrix} L & D & R \end{pmatrix} x = D x_{\text{loc}} + \begin{pmatrix} L & 0 & R \end{pmatrix} x$$

The first term only contains local entries and can thus be done without information from other processes. For completing the calculation the process must import the values of x that are not local. Because the matrix is sparse, only a fraction of the vector entries needs to be imported. This can be precalculated at matrix construction time and the transfer to and from the other processes can be done non-blocking while computing the local product with the matrix D. The transfer is called scatter operation, because parts of the local vector are sent to different processes. The pseudo code would look like this:

```
1. create local Vector xtemp
2. begin scatter xtemp<->x (in the background)
3. multiply y.loc=D*x.loc
4. end scatter xtemp<->x
5. multiply y.loc+=(L 0 R)*xtemp
```

The communication latency can thus be hidden behind the local computation and one can expect better parallel scalability than doing scalar products.

The communication volume for the matrix-vector product depends on how many elements must be imported. The distribution of the degrees of freedom on the interface is crucial for that. As soon as a processor does not own at least one degree of freedom from the interface, typically all degrees of freedom of all cells connecting to that one degree have to be imported additionally. A naive implementation may thus double the amount of communication needed. We discuss this in Remark 2 (page 49) later.

Assembling linear systems

Filling matrices and vectors in parallel requires some considerations because some degrees of freedoms – and thus entries – are shared between several machines. The assembly process is done cell-wise, see Section 1.2. Therefore, machines generate contributions to the global matrix that are stored on a different machine. The way it is implemented in PETSc and Trilinos is that they allow adding non-local entries via the interface to the matrix. Internally they are buffering the non-local entries and are sending them out to the corresponding machines after the assembly.

Block vectors and block matrices

When handling coupled PDEs with more than one variable, one typically likes to use block matrices and block vectors, which combine those into one object. One option is to put this data into a normal distributed vector, but then one has to interleave the degrees of freedom of the components to keep the local data be a consecutive range. For preconditioners that operate on parts of the coupled system it is necessary to have the unknowns sorted by component. Otherwise the matrices do not build a block structure.

There is an easy way out that is also implemented in `deal.II`: One can implement the distributed block vectors and matrices by representing them with a one or two dimensional array of distributed vectors or matrices. All operations work as expected by combining operations on the blocks.

2.2.3 Parallel Krylov methods

With the linear system in place distributed between the machines row-wise, one must then solve it. A typical way to do it is to parallelize Krylov methods like CG, GMRES, or BiCGStab as described in Section 1.4. Additionally preconditioners must be adapted too.

The standard Krylov methods (see [109] for a detailed discussion) only contain matrix-vector products, scalar products, and linear combinations of vectors. We already discussed how to implement these operations in 2.2.2. Surprisingly this all that is required to run a Krylov method like GMRES in parallel. The implementation doesn't even have to be implemented with parallelization in mind: the default templated Krylov methods in `deal.II` for example are templated and thus take arbitrary matrix and vector types as long as they supply the necessary operations.

The methods work fairly well with the right preconditioners, but of course they are not optimized for parallel usage. The high number of scalar products create many synchronization points which slow down the algorithm. There exist some optimized parallel variants of the common Krylov methods, see [43] for example. The scalar products in creating the orthonormal basis are the bottleneck, so one tries other orthonormalization procedures than Gram-Schmidt orthogonalization. At the moment this is not implemented in PETSc, Trilinos, or `deal.II`, though and it is not clear how much better the performance would be.

Preconditioners on the other hand can not be easily adapted to distributed computations. One compromise is applying preconditioners like ILU decompositions only to the local diagonal blocks (denoted as D in 2.2.2). Global ILU decompositions are not feasible due to the amount of communication required to set up the preconditioner. With block methods on the other hand the quality of the preconditioner decreases with the number of processes involved, because in the limit the ILU decomposition for example reduces to a diagonal preconditioner. Nevertheless the block methods are very efficient due to zero communication costs. Several preconditioners are readily available in linear algebra packages like PETSc or Trilinos. For more details see [109].

The class of parallel algebraic multi-grid methods are a very good alternative. Because the coarser levels span more than one processor, there is some communication required for the setup and evaluating the preconditioner. On the other hand they scale well with the number of processes. Packages like ML in Trilinos (see [49]) and BoomerAMG from the Hypre package (see [65]) supply very sophisticated implementations. We are going to use those later on.

A third option are geometric multi-grid methods, that promise very good preconditioners. A parallel implementation is less than trivial and requires a lot of information about the mesh structure. There is no finished parallel implementation for arbitrary finite elements with adaptive refinement in `deal.II` or any other finite element package as far as it is known at the moment this thesis was written.

2.3 Parallel finite element software architecture

In this section we will now outline the algorithms that we have implemented in version 7.0 of the open source library `deal.II`, offering the ability to solve finite element problems on fully adaptive meshes of the sizes mentioned in the beginning to a wider community. While `deal.II` provides a reference implementation of both the mentioned algorithms as well as of tutorial programs that showcase their use in actual applications, our goal is to be generic and our methods would certainly apply to other finite element libraries as well. In particular, we will not require specific aspects of the type of finite element, nor will the algorithms be restricted to quadrilaterals (2d) and hexahedra (3d) that are used exclusively in `deal.II`, see Section 1.2 and 1.3.

We will not be concerned with the question of how to efficiently generate and partition hierarchically refined meshes on large numbers of processors, which presents a major challenge on its own. Rather, we will assign this task to an “oracle” that allows `deal.II` to obtain information on the distributed nature of the mesh through a well-defined set of queries. These include for example whether a certain cell exists on the current processor, or whether it is a “ghost” cell that is owned by another processor. Using queries to the oracle, each processor can then rebuild the rich data structures necessary for finite element computations for the “locally owned” part of the global mesh and perform the necessary computations on them. In our implementation, we use the `p4est` algorithms for 2d and 3d parallel mesh topology [28] as the oracle; however, it is entirely conceivable to connect to different oracle implementations – for example packages that support the ITAPS iMesh interface [96] –, provided they adhere to the query structure detailed in this article, and can respond to certain mesh modification directives discussed below.

We will detail the requirements `deal.II` has of the mesh oracle, a description of the way `p4est` works, and the algorithm that builds the processor-local meshes in Section 2.3.1. In Section 2.3.2, we will discuss dealing with the degrees of freedom defined on a distributed mesh. Section 2.3.3 will then be concerned with setting up, assembling and solving the linear systems that result from the application of the finite element method; Section 2.3.4 discusses the parallel computation of thresholds for a-posteriori error indicators and postprocessing of the solution. We provide numerical results that support the scalability of all proposed algorithms in Section 2.3.5.

2.3.1 Parallel construction of distributed meshes

For solving large problems in a finite element program the computational mesh as described in Section 1.2 consumes too much memory to be stored on every machine. Additionally operations covering all cells on each machine obviously can not scale properly with increasing the number of processors. We resolve this limitation based on the tenets in Section 2.1.4 by *distributed* mesh storage with *coarsened overlap*: Each processor still stores a local mesh that covers the whole domain, but this mesh is now different on each processor. It is identical to the global mesh only in the part that is identified by the oracle as “locally owned” by the current processor, whereas the remaining and much larger non-owned part of the local mesh is coarsened as much as possible, rendering its memory footprint insignificant. With this approach the global mesh is not replicated anymore but understood implicitly as the disjoint union of the locally owned parts on each processor. To achieve parallel scalability of the complete finite element pipeline, the storage of degrees of freedom and matrices arising from a finite element discretization must be fully distributed as well, which can be achieved by querying the oracle about ghost cells and creating efficient communication patterns and data structures for index sets as we will explain below.

We encode the distributed mesh in a two-layered approach. The inner layer, which we refer to as the “oracle”, provides rudimentary information on the owned part of the mesh and the parallel neighborhood, and executes directives to coarsen, refine, and re-partition the mesh. The outer layer interacts with the oracle through a well-defined set of queries and builds a representation of the mesh that includes the refinement hierarchy and some overlap with neighboring mesh parts, and is rich enough to provide all information requires for finite element operations. This two-layered approach effectively separates a large part of the parallel management of mesh topology in the oracle from the locally stored representation retaining the existing infrastructure in `deal.II`.

There is a significant amount of literature on how to generate and modify distributed adaptive meshes in parallel. For example, [24, 32, 78, 117, 122] discuss related data structures and algorithms. In the current contribution, we base our work on the open source software library `p4est`, which realizes the oracle functionality in the sense outlined above, and has been shown to scale to hundreds of thousands of processors [28]. However, any other software that allows the well-defined and small list of queries detailed below may equally well be used in place of `p4est`. For example, this could include the packages that support the ITAPS iMesh interface [96].

In this part, we define the general characteristics of the mesh, propose an algorithm to construct the local mesh representation based on querying the oracle, and document mesh modification capabilities required from the oracle.

Assumptions on parallel distributed meshes

We will not be concerned with the technical details of the parallel storage of meshes or the algorithms hidden within the oracle. In particular, for our purposes we only need to be able to infer what cells exist, how they relate to each other via neighborship, and how they have been derived by hierarchic refinement from a small to moderate set of initial coarse mesh cells. We will make the following general assumptions that are respected by both the inner layer or oracle (`p4est`) and the outer layer (implemented within `deal.II`):

- *Common coarse mesh:* All cells are derived by refinement from a common coarse mesh that can be held completely on each of the processors and should therefore not exceed a few 100,000 to a million cells. In general, the common coarse mesh only has to provide a sufficient number of cells to capture the topology of the computational domain, which is often below 100 or even as small as 1, while mesh refinement takes care of geometric details. Only in rare cases does geometric complexity require 100,000s or more coarse mesh cells; consequently, we reserve the dynamic partitioning of coarse cells, which is certainly feasible, for a future extension. Because `deal.II` exclusively supports quadrilaterals and hexahedra, we will henceforth assume that the common coarse mesh only consists of such cells, though this is immaterial for almost all that follows, and our algorithms are equally valid when applied to meshes consisting of triangles or tetrahedra.

- *Hierarchic refinement:* Each of the common coarse mesh cells may be hierarchically refined into four (2d) or eight (3d) children which may in turn be further refined themselves. This naturally gives rise to a quad- or oc*tree* rooted in each common coarse cell, and an appropriate data structure for the entire mesh then is a quad- or oct*forest*. Therefore each cell can be uniquely identified by an index into the common coarse mesh (i.e., its tree number) and an identifier that describes how to walk through the corresponding tree to the (refined) cell.

- *2:1 mesh balance:* We demand that geometrically neighboring cells may differ by only a single refinement level, thereby enforcing that only a single *hanging node* can exist per face or edge. This condition is mostly for convenience, since it simplifies the creation of interpolation operators on interfaces between cells.
- *Distributed storage:* Each processor in a parallel program may only store a part of the entire forest that is not much larger than the total number of cells divided by the number of processors. This may include a fixed number of ghost cell layers, but it cannot be a fraction of the entire mesh that is independent of the number of processors. We explicitly permit that each processor may store parts of more than one tree, and that parts of a given tree may be stored on multiple processors. See also Section 2.1.4.

Note that a mesh is independent of its use; in particular, it has no knowledge of finite element spaces defined on it, or values of nodal vectors associated with such a space. It is, thus, a rather minimal data structure to simplify parallel distributed storage. Furthermore, the separation of mesh and finite element data structures establishes a clean modularization of the respective algorithms and implementations.

A mesh oracle and interface to `deal.II`

`deal.II` must keep rich data structures for the mesh and derived objects. For example, it must know the actual geometric location of vertices, boundary indicators, material properties, etc. It also stores the complete mesh hierarchy and data structures of surfaces, lines and points and their neighborship information for traversal, all of which are required for the rest of the library and to support algorithms built on it.

On the other hand, `p4est` only stores the terminal nodes (i.e., the leaves) of the parallel forest explicitly. By itself, this is not enough for all but the most basic finite element algorithms. However, we can resolve this apparent conflict if `deal.II` builds its own local mesh on the current processor, using the locally stored portion of the parallel distributed mesh stored by `p4est` as the template, and augmenting it with the information required for more complex algorithms. In a sense, this approach forms a synthesis of the completely distributed and lean data structures of `p4est` and the rich structures of `deal.II`. Designing this synthesis in a practical and scalable way is one of the innovations of this research; we will demonstrate its efficiency in Section 2.3.5.

In order to explain the algorithm that reconstructs the local part of a mesh on one processor, let us assume that both `deal.II` and `p4est` already share knowledge about the set of common coarse cells. Then, `deal.II` uses `p4est` as an oracle for the following rather minimal set of queries:

- Does a given terminal `deal.II` cell exist in the portion of the `p4est` mesh stored on the current processor?
- Does a given `deal.II` cell (terminal or not) overlap with any of the terminal `p4est` cells stored on the current processor?
- Does a given `deal.II` cell (terminal or not) overlap with any of the terminal `p4est` ghost cells (defined as a foreign cell sharing at least one corner with a cell owned by the current processor)?
- Is a given `p4est` cell a ghost cell and if yes, which processor owns it?

The algorithm for mesh reconstruction based on only these queries is shown in Fig. 2.3. It is essential that all queries are executed fast, i.e., in constant time or at most $\mathcal{O}(\log N)$, where N is the number of local cells, to ensure overall optimal complexity. Furthermore, no query may entail communication between processors. Note that the reconstruction algorithm makes no assumptions on the prior state of the `deal.II` mesh, allowing for its coarsening and refinement as the oracle may have moved cells to a different processor during adaptive mesh refinement and re-partitioning. In a `deal.II` mesh so constructed, different kinds of cells exist on any particular processor:

- *Active cells* are cells without children. Active cells cover the entire domain. If an active cell belongs to a part of the global mesh that is owned by the current processor, then it corresponds to a leaf of the global distributed forest that forms the mesh. In that case, we call it a *locally owned* active cell.
- *Ghost cells* are active cells that correspond to leaves of the distributed forest that are not locally owned but are adjacent to locally owned active cells.
- *Artificial cells* are active cells that are neither locally owned nor ghost cells. They are stored to satisfy `deal.II`'s invariants of never having more than one hanging node per face or edge, and of storing all common coarse mesh cells. Artificial cells can, but need not correspond to leaves of the distributed forest,

```
copy_to_deal:
do
  for all coarse mesh cells $K$:
    match_tree_recursively($K$)
  refine and coarsen all cells previously marked
while (the mesh changed in the last iteration)

match_tree_recursively($K$):
if (oracle: does $K$ overlap with locally owned or ghost parts of the mesh?)
  if ($K$ has children)
    for each child $K_c$ of $K$
      match_tree_recursively($K_c$)
  else
    if not (oracle: does $K$ exist in the locally owned or ghost part?)
      mark $K$ for refinement
else
  mark the most refined descendents of $K$, or $K$ itself, for coarsening
```

Figure 2.3: Pseudo-code for reconstructing the local part of a mesh in `deal.II`*, based on querying the mesh oracle provided by* `p4est`*. The algorithm starts with an arbitrary mesh and terminates once the mesh contains all cells that the oracle indicates as either locally owned or ghost cells.*

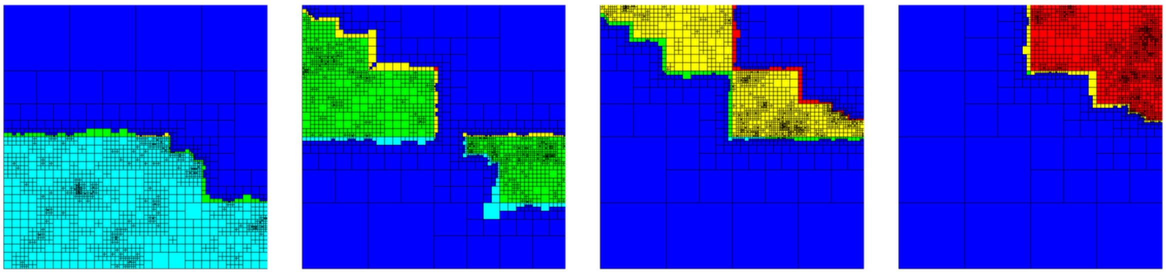


Figure 2.4: Example of an adaptively refined mesh distributed across four processors. The cyan, green, yellow, and red colors indicate which processor owns any given cell. The four panels depict the views each of the four processors has of the mesh. Note that each processor knows only (i) the global cells it owns, and (ii) one layer of ghost cells in the global mesh and their owner processor identifiers. The artificial cells (indicated in dark blue) carry no information. The effective mesh used for computation is the union of the four locally owned parts.

and are skipped in every algorithm inside `deal.II`.

- *Non-active cells* are cells that have children. `deal.II` stores all intermediate cells that form the hierarchy between coarse mesh cells (the roots of the trees) and active cells.

Fig. 2.4 shows the result of executing `copy_to_deal` (Fig. 2.3) on an example mesh distributed among four processors. Note that no processor has knowledge of the entire global mesh – each processor only matches its own cells as well as one layer of ghost cells. Because the parallel partitioning and identification of ghost cells is computed by `p4est` according to a space-filling z-curve [95], the part of the global mesh owned by a processor may not be contiguous. This can be seen in the second panel of the figure.

Remark 1: *Storing artificial cells that do not belong to the coarse mesh appears wasteful since these cells are indeed unnecessary for almost all computations. As pointed out above we only store them to maintain the invariants for which the base library, `deal.II`, has been extensively validated. Clearly, the fraction of artificial cells decreases as the number of cells stored locally increases. For the 2d example discussed in Section 2.3.5, which has 1 coarse cell, our numerical experiments suggest that the ratio* $N_{artificial}/(N_{active}+N_{ghost})$ *is only very weakly dependent on the number of processors, and decreases as* $\mathcal{O}((N_{active}+N_{ghost})^{-0.55})$*. On a fine mesh with 4,096 processors and a total of almost 600 million cells, on average only 3% of the cells stored locally are artificial and on no processor does this number exceed 5%.*

We reflect the different types of cells using the following notation: Let $\mathbb{T}$ denote the set of all terminal cells that exist in the distributed mesh. Furthermore, let $\mathbb{T}^p_{\mathrm{loc}} \subset \mathbb{T}$ be the subset of cells that processor p owns; obviously, $\bigcup_p \mathbb{T}^p_{\mathrm{loc}} = \mathbb{T}$, and we will require that $\mathbb{T}^p_{\mathrm{loc}} \cap \mathbb{T}^q_{\mathrm{loc}} = \emptyset$ for all $p \neq q$. Finally, let $\mathbb{T}^p_{\mathrm{ghost}} \subset \mathbb{T}$ be the set of ghost cells that processor p knows about; we have that $\mathbb{T}^p_{\mathrm{ghost}} \cap \mathbb{T}^p_{\mathrm{loc}} = \emptyset$ and we will assume that each ghost cell $K \subset \mathbb{T}^p_{\mathrm{ghost}}$ has at least one neighbor in $\mathbb{T}^p_{\mathrm{loc}}$ where neighborship is via faces, lines, or vertices. In addition to $\mathbb{T}^p_{\mathrm{loc}}$ and $\mathbb{T}^p_{\mathrm{ghost}}$, each processor stores additional terminal cells that may or may not be terminal cells in $\mathbb{T}$, for example some coarse mesh cells. We will call these *artificial* cells and denote them by $\mathbb{T}^p_{\mathrm{artificial}}$; they are shown in dark blue in Fig. 2.4.

Directives for mesh modification

We will require that the oracle not only responds to the queries listed above, but also performs several operations that modify the distributed global mesh. Such mesh modifications are often used during the startup phase of a simulation, or repeatedly to adapt according to error indicators or to track dynamical features of a simulation that evolves over time.

- Refine and/or coarsen the mesh based on flags set by `deal.II`. Refinement and coarsening shall be executed locally without communication between processors.
- Enforce 2:1 mesh balance by additional refinement where necessary, limiting the level difference between neighboring cells to one. This is done as a postprocessing step to local refinement and coarsening which involves communication with processors that own nearby parts of the mesh.
- Re-partition the cells of the global mesh in parallel to ensure load balance (the most commonly used criterion being to equalize the number of cells among processors). This operation involves mostly point-to-point communication. During the re-partitioning step, additional information shall be attached to the cells. When a cell is migrated from one processor to another, this data is automatically migrated with the cell.

While this functionality can entail considerable complexity, it is likely to be available from implementations of parallel mesh data bases. Thus, we do not consider the above specifications unnecessarily restrictive. In the case of `p4est` we refer the reader to the algorithms presented in [28]. `deal.II` makes use of these capabilities to efficiently implement a number of operations typical of finite element codes; see Section 2.3.4.

2.3.2 Dealing with global indices of degrees of freedom

Once we have a local representation of a distributed mesh, the next step in any finite element program is to connect the finite element space to be used with the triangulation. In `deal.II`, this tasks falls to the `DoFHandler` class [8] that inspects a `FiniteElement` object for the number of degrees of freedom that are required per vertex, line, face, and cell. For example, for a Taylor-Hood ($Q_2^d \times Q_1$) element used for the Stokes equations in d space dimensions, we need $d+1$ degrees of freedom per vertex, and d for each line, quad and hex (if in 3d). The `DoFHandler` will then allocate global numbers for each of the degrees of freedom located on the vertices, lines, quads and hexes that exist in the triangulation. A variant of this class, `hp::DoFHandler`, is able to do the same task if different finite elements are to be used on different cells such as in *hp*-adaptive computations [10].

In the current context, we will have to assign global indices for degrees of freedom defined on a mesh of which we only know a certain part on each processor. In the following subsections, we will discuss the algorithms that achieve this task, followed by strategies to deal with the constraints that result from hanging nodes. Together, indices of degrees of freedom and constraints will completely describe a basis of the finite element space we want to use.

Enumerating degrees of freedom

The simplest way to distribute global indices of degrees of freedom on the distributed mesh would be to first let processor 0 enumerate the degrees of freedom on the cells it owns, then communicate the next unused index to processor 1 that will then enumerate those degrees of freedom on its own cells that have not been enumerated yet, pass the next unused index to processor 2, and so on. Obviously, this strategy does not scale beyond a small number of processors.

Rather, we use the following algorithm to achieve the same end result in a parallel fashion where all processors $p = 0, \ldots, P-1$ work independently unless noted otherwise. This algorithm also determines the ownership of degrees of freedom on the interface between cells belonging to different processors. The rule for decision of ownership is arbitrary but must be consistent and must not require communication. The number of processors involved is typically up to eight for a degree of freedom on a vertex in 3d, but can be even higher for a coarse mesh with complicated topology. We resolve to assign each degree of freedom on an interface between processors to the processor with the smallest processor identifier (the "rank" in MPI terminology).

0. On all active cells (locally owned or not), initialize all indices of degrees of freedom with an invalid value, for example -1.
1. Flag the indices of all degrees of freedom defined on all cells $K \in \mathbb{T}^p_{\text{loc}}$ by assigning to them a valid value, for example 0. At the end of this step, all degrees of freedom on the locally owned cells have been flagged, including those that are located on interfaces between cells in $\mathbb{T}^p_{\text{loc}}$ and $\mathbb{T}^p_{\text{ghost}}$.
2. Loop over all ghost cells $K \in \mathbb{T}^p_{\text{ghost}}$; if the owner of K is processor q and $q < p$ then reset indices of the degrees of freedom located on this cell to the invalid value. After this step, all flagged degrees of freedom are the ones we own locally.
3. Loop over all cells $K \in \mathbb{T}^p_{\text{loc}}$ and assign indices in ascending order to all degrees of freedom marked as valid. Start at zero, and let n_p be the number of indices assigned. Note that this step cannot be incorporated into step (2)

because degrees of freedom may be located on interfaces between more than two processors and a cell in $\mathbb{T}^p_{\text{loc}}$ may not be able to easily determine whether cells that are not locally owned share such an interface.

4. Let all processors communicate the number n_p of locally owned degrees of freedom to all others. In MPI terminology, this amounts to calling `MPI_Allgather`. Shift the indices of all enumerated degrees of freedom by $\sum_{q=0}^{p-1} n_q$. At the end of this step, all degrees of freedom on the entire distributed mesh have been assigned globally unique indices between 0 and $N = \sum_{q=0}^{P-1} n_q$, and every processor knows the correct indices of all degrees of freedom it owns. However, processor p may not know the correct indices of degrees of freedom on the interface between its cells and those owned by other processors, as well as the indices on ghost cells that we require for some algorithms. These remaining indices will be obtained in the next two steps.

5. Communicate indices of degrees of freedom on cells in $\mathbb{T}^p_{\text{loc}}$ to other processors according to the following algorithm:

    a) Flag all vertices of cells in $\mathbb{T}^p_{\text{loc}}$.

    b) Loop over vertices of cells in $\mathbb{T}^p_{\text{ghost}}$ and populate a map that stores for each of the vertices flagged in step (a) the owning processor identifier(s) of adjacent ghost cells.

    c) Loop over all cells in $\mathbb{T}^p_{\text{loc}}$. If according to the previous step one of its vertices is adjacent to a ghost cell owned by processor q, then add the pair *[cell_id, indices of degrees of freedom on this cell]* to a list of such pairs to be sent to processor q. Note that the same pair can be added to multiple such lists if the current cell is adjacent to several other processors' cells. Note also that every cell we add on processor p to the list for processor q is in $\mathbb{T}^q_{\text{ghost}}$. This communication pattern is symmetric, i.e., processor p receives a message from q if and only if it sends to q; this symmetry avoids the need to negotiate communications.

    d) Send the contents of each of these lists to their respective destination processor q using non-blocking point-to-point communication.

    e) From all processors that the current one borders to (i.e., the owners of any of the cells in $\mathbb{T}^p_{\text{ghost}}$), receive a list as created above. Each of the cells in this list refer to a ghost cell; for each of these cells, set the indices of the degrees of freedom on this cell to the ones given by the list unless the index in the list is invalid.

    Note that while the lists created in step (c) contain only cells owned by the current processor, not all indices in them are known as they may lie on an interface to another processor. These will then be the invalid index, prompting the need for the conditional set in step (e). On the other hand, it is easy to see that if an index located on the interface between two ghost cells is set more than once, then the value so set is always either the same (if the ghost cells belong to the same processor) or the invalid marker (in which case we ignore it).

6. At the end of the previous step, all cells in $\mathbb{T}^p_{\text{loc}}$ have their final, correct indices set. However, some ghost cells may still have invalid markers since their indices were sent by processors that at the time did not know all correct indices yet. They do now, however. Consequently, the last step is to repeat the actions of step (5). We can optimize this step by only adding cells to the send lists that prior to step (5e) had invalid index markers.

At the end of this algorithm, each processor knows the correct global indices of degrees of freedom on all of the cells it locally owns as well as on all the ghost cells. We note that this algorithm is not restricted to h-refined meshes but is equally applicable to hp-adaptivity.

A similar algorithm that makes the same decision for degrees of freedom on the interface is detailed in [87], but there are a few crucial differences to our approach: First, their algorithm contains a sequential part to compute the indices of shared degrees of freedom (Stage 2), while ours does that computation in parallel. Second, our approach lends itself to non-blocking communication (see step 5d above). Third, we decided to realize the communication over shared vertices instead of facets, which simplifies the calculation and enables us to send data directly to the destination (instead of sending it indirectly via other processors when only a vertex is shared). Fourth, instead of implementing a more complicated logic for transferring individual degrees of freedom we opted to always send all degrees of freedom belonging to a cell and even to accept sending a cell twice, which can only happen for some cells that touch more than one other processor (see step 6). Because we transfer the data of the whole cell, we ensure knowledge of *all* degrees of freedom on ghost cells, not only those on the interface to locally owned cells as described in [87] or [28]; this is necessary for a number of algorithms that must, for example, evaluate the gradient of the solution on both sides of an interface. While our approach requires sending slightly larger messages, it is overall more efficient because that data does not need to be sent later in an additional communication step. The rationale here is that since the amount of data exchanged is modest in either case, communication cost is dominated by latency rather than message size.

Remark 2: *Our algorithm always assigns degrees of freedom on the interface between processors to the one with the smallest processor identifier. This is in contrast to earlier* *work, see [62]. It results in a slight imbalance: processors with identifiers close to zero tend to own more degrees of freedom than the average, and processors with ranks close to the parallel job size own less, while most processors in the bulk own roughly the average number.*

One may therefore think about constructing a better tie breaker for ownership of degrees of freedom on processor interfaces. `deal.II` *implements such a fairer scheme in a mode where each processor stores the entire mesh, as does the current code of FEniCS/DOLFIN. See also Section 2.2.1. However, our experiments indicate that at least relatively simple schemes do not pay off, for several reasons. First, when different degrees of freedom on the same edge or face are assigned to two different processors A and B, matrix-vector multiplications require roughly twice the amount of data transfer because the connectivity graph between degrees of freedom is partitioned by cutting more edges than when assigning all degrees of freedom on a complete face to one side alone. Second, determining ownership is easily done without communication in our algorithm above. Third, the workload in downstream parts of the finite element code is typically quite well balanced, as the cost for many operations is proportional to the number of local cells – which* `p4est` *balances perfectly – and not to the number of degrees of freedom. Finally, by enumerating the degrees of freedom on at least one of the cells adjacent to an interface in a natural ordering, we improve cache locality and thus the performance when accessing corresponding data. To evaluate these arguments, we carefully analyzed the distribution of degrees of freedom and observed only a small imbalance in memory consumption in our numerical tests, while we found excellent scalability of our matrix-vector product implementation.*

Subsets of degrees of freedom

In the following sections, we will frequently need to identify certain subsets of degrees of freedom (by convention by identifying their respective global indices). To this end, let us define the following subsets of the complete set of indices $\mathcal{I} = [0, N)$:

- $\mathcal{I}^p_{l.o.}$ denotes the set of degrees of freedom *locally owned* by processor p. These are all defined on cells in $\mathbb{T}^p_{\text{loc}}$, though some of the degrees of freedom located on the interfaces of these cells with other processors may be owned by the neighboring processor. We have $n_p = \#\mathcal{I}^p_{l.o.}$, $\bigcup_q \mathcal{I}^q_{l.o.} = \mathcal{I}$, and $\mathcal{I}^p_{l.o.} \cap \mathcal{I}^q_{l.o.} = \emptyset$ for $p \neq q$. Note that following the algorithm described in the previous section, the set of indices in $\mathcal{I}^p_{l.o.}$ is contiguous. However, this is no longer true when degrees of freedom are renumbered later.
- $\mathcal{I}^p_{l.a.}$ denotes the set of degrees of freedom that are *locally active* for processor p. This set contains all degrees of freedom defined on $\mathbb{T}^p_{\text{loc}}$, and $\mathcal{I}^p_{l.a.} \cap \mathcal{I}^q_{l.a.}$

identifies all those degrees of freedom that live on the interface between the subdomains owned by processors p and q if these are neighbors connected by at least one vertex of the mesh.

- $\mathcal{I}^p_{l.r.}$ denotes the set of degrees of freedom that are *locally relevant* for processor p. We define these to be the degrees of freedom that are located on all cells in $\mathbb{T}^p_{\text{loc}} \cup \mathbb{T}^p_{\text{ghost}}$.

These index sets $\mathcal{I}^p_{l.o.} \subset \mathcal{I}^p_{l.a.} \subset \mathcal{I}^p_{l.r.}$ must be represented in a computer program for the algorithms discussed below. Maybe surprisingly, we have found that the data structures chosen for this have an enormous impact on the efficiency of our programs as the number of queries into these index sets is very large. In particular, we will frequently have to test whether a given index is in an index set, and if it is we will have to determine the position of an index within this set. The latter is important to achieve our goal that no processor should ever hold arrays on all elements of $\mathcal{I}$: rather, we would like to compress these arrays by only storing data for all elements of a set $\tilde{\mathcal{I}} \subset \mathcal{I}$, but for this we need to map global indices into positions in index sets and vice versa. The efficient implementation of such operations is therefore an important aspect, in particular if the index set is not simply a single contiguous range of indices.

In `deal.II`, the `IndexSet` class implements all such queries. It stores an index set as the union $\tilde{\mathcal{I}} = \bigcup_{k=0}^{K} [b_k, e_k)$ of K half open, disjoint, contiguous intervals that we store sorted by their first indices b_k. Here, we denote by $\tilde{\mathcal{I}}$ a generic index set that could, for example, be any of the sets defined above. For isolated indices, we have $e_k = b_k + 1$. This data structure allows to test whether an index is in the set in $\mathcal{O}(\log_2 K)$ operations. However, the determination of the position of a given index i in the set would require $\mathcal{O}(K)$ operations: if k' is the interval in which i is located, i.e., $b_{k'} \le i < e_{k'}$, then

$$\operatorname{pos}(i, \tilde{\mathcal{I}}) = \sum_{k=0}^{k'-1} (e_k - b_k) + (i - b_{k'}),$$

where the determination of $k' = \min\{k : i < e_k\}$ can be done in parallel to summing over the sizes of intervals. Similarly, computing the value of the mth index in a set $\tilde{\mathcal{I}}$ would require $\mathcal{O}(K)$ operations on average.

We can remove both these bottlenecks by storing with each interval $[b_k, e_k)$ the number $p_k = \sum_{\kappa=0}^{k-1} (e_\kappa - b_\kappa) = p_{k-1} + (e_{k-1} - b_{k-1})$ of indices in previous intervals. We update these numbers at the end of generating an index set, or whenever they have become outdated but a query requires them. Finding the position of index i then only requires finding which interval k' it lies in – an $\mathcal{O}(\log_2 K)$ operation – and then computing $p_{k'} + i - b_{k'}$. Likewise finding the value of the mth index requires finding the largest $p_k < m$, which can also be implemented in $\mathcal{O}(\log_2 K)$ operations. In summary, storing an index set as a sorted collection $\tilde{\mathcal{I}} \sim \{(b_k, e_k, p_k)_{k=0}^{K}\}$ of triplets allows for efficient implementation of all operations that we will need below.

Constraints on degrees of freedom

The algorithms described above provide for a complete characterization of the basis of the finite element space on each cell. However, since we allow hanging nodes in our mesh, not every local degree of freedom is actually a global degree of freedom: some are in fact constrained by adjacent degrees of freedom. In general, the construction of such constraints for hanging nodes is not overly complicated and can be found in [30,103,114,115]; the algorithms used in `deal.II` are described in [8,10]. We will here focus on those aspects particular to distributed computations.

Constraints on degrees of freedom typically have the form

$$x_i = \sum_{j=0}^{N-1} c_{ij} x_j + b_i, \qquad i \in \mathcal{I}_c \subset \mathcal{I},$$

where $\mathcal{I}_c$ is the set of constrained degrees of freedom, and the *constraint matrix* c_{ij} is typically very sparse. For hanging nodes, the inhomogeneities b_i are zero; as an example, for lowest order elements the constraints on edge mid-nodes have the form $x_2 = \frac{1}{2}x_0 + \frac{1}{2}x_1$. Constraints may also originate from strongly imposed Dirichlet-type boundary values in the form $x_0 = 42$, for example.

Which constraints need to be stored? It is clear that not every processor will be able to store the data that describes *all* the constraints that may exist on the distributed finite element space. In fact, each processor can only construct constraints for a subset of $\mathcal{I}^p_{l.r.} \cap \mathcal{I}_c$ since it has no knowledge of any of the other degrees of freedom. Consequently, the question here is rather which subset $\mathcal{I}^p_c$ of constraints each processor could in principle construct, and which it requires to construct and store locally for the algorithms described below to work.

For sequential computations, one can first assemble the linear system from all cell contributions irrespective of constraints and in a second step "eliminate" constrained degrees of freedom in an in-place procedure (see, for example, [10, Section 5.2]). On the other hand, in distributed parallel computations, no processor has access to a sufficient number of matrix rows to eliminate constrained degrees of freedom after the linear system has already been assembled from its cell-wise contributions. Consequently, we have to eliminate constrained degrees of freedom already when copying local contributions into the global linear system. While this may not be quite as elegant, it has the benefit that we know exactly what degrees of freedom we may have to resolve constraints for. Namely, exactly those that may appear in local contributions to the global linear system: if processor p has a contribution to global entry (i, j) of the matrix, then it must know about constraints on degrees of freedom i and j. Which these are depends on both the finite element as well as the bilinear form in use.

Local contributions to the global linear system are computed by each processor for all cells $\mathbb{T}^p_{\text{loc}}$ (i.e., for all degrees of freedom in $\mathcal{I}^p_{l.a.}$), see Section 1.2 and 2.2.2. For most finite elements and bilinear forms, the local contribution consists of integrals only over each cell $K \in \mathbb{T}^p_{\text{loc}}$ and consequently every processor will only need to know constraints on all degrees of freedom in $\mathcal{I}^p_c = \mathcal{I}_c \cap \mathcal{I}^p_{l.a.}$. Discontinuous Galerkin methods also have jump terms between cells, and consequently need to also know about constraints on degrees of freedom on cells neighboring those that are locally owned; in that case, we need to know about all constraints in $\mathcal{I}^p_c = \mathcal{I}_c \cap \mathcal{I}^p_{l.r.}$.

Dealing with chains of constraints. The considerations above are of only theoretical interest if constraints can be against degrees of freedom that are themselves constrained, i.e., if constraints form chains. This frequently happens in at least two situations. First, it is common in hp-adaptive methods, see for example [10]. In that case, it is even conceivable that chains of constraints extend to the boundary between ghost and artificial cells. Then, the depth of the ghost layer would need to be extended to more than one layer of cells, thereby also expanding the set $\mathcal{I}^p_{l.r.}$. We will not consider such cases here.

The second, more common situation is if we have Dirichlet boundary conditions on degrees of freedom, e.g., constraints of the form $x_0 = 42$. If another constraint, e.g., $x_2 = \frac{1}{2}x_0 + \frac{1}{2}x_1$, references such a degree of freedom x_0 and if the latter is located in the ghost layer, then we must know about the constraint on x_0. For this reason, in `deal.II` each processor always stores all those constraints in $\mathcal{I}^p_c = \mathcal{I}_c \cap \mathcal{I}^p_{l.r.}$ that can be computed on locally owned and ghost cells.

Computing constraints for hanging nodes. Of equal importance to the question of which constraints we need to store is the question how we can compute the necessary constraints that result from hanging nodes. Let us first consider the case of continuous elements with only cell integration, i.e., $\mathcal{I}^p_c = \mathcal{I}_c \cap \mathcal{I}^p_{l.a.}$. Since all of these degrees of freedom are adjacent to locally owned cells, it may appear that it is sufficient to compute constraints by only considering hanging nodes at faces between two locally owned cells, or between a locally owned cell and a ghost neighbor. While we believe that this true in two space dimensions, this is not so in 3d. For example, consider the situation depicted in Fig. 2.5, assuming trilinear finite elements. The degree of freedom indicated by the blue dot is locally active both on processor 0 (white cells) and processor 1 (yellow cells in front). However, since hanging node constraints are computed based on the face between coarse and fine cells, not solely on edges, processor 1 can only know about the constraint on this degree of freedom by computing the constraint on the interface between the white cells, all of which are ghost cells for processor 1.

Since the structure and size of the set $\mathcal{I}_c^p$ depends also on the bilinear form, one can imagine situations in which computing it is even more involved than described in the previous paragraph. For example, if the bilinear form calls for face integrals involving all shape functions from both sides of the face, we would need to have constraints also on all degrees in $\mathcal{I}_{l.r.}^p$ which we may not be able to compute only from a single layer of ghost cells. Fortunately, most discretizations that require such terms have discontinuous shape functions that do not carry constraints on hanging nodes; for a counter example see [73].

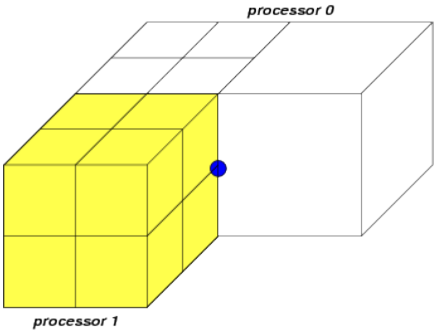


Figure 2.5: Illustration of a situation where constraints must be computed between two ghost cells.

Evaluating constraints. When copying local contributions into the global matrix and right-hand side vector objects during finite element assembly of linear systems, we have to determine for each involved degree of freedom i whether it is constrained or not, and if it is what the coefficients c_{ij}, b_i of its constraint are. For the sequential case, the `deal.II` class `ConstraintMatrix` stores an array of integers for all degrees of freedom. These integers contain the position of the constraint in the list of all constraints, or -1 for $i \notin \mathcal{I}_c$. This guarantees that the query whether i is constrained can be performed in $\mathcal{O}(1)$, as is actually accessing the constraints.

On the other hand, in the parallel distributed case under consideration here, this strategy is not compatible with our desire to never store arrays on all degrees of freedom on a single processor. Rather, we are presented with two options:

- On each processor, the `ConstraintMatrix` stores a sorted container of $\#\mathcal{I}_c^p$ elements each of which contains the index of the constrained degree of freedom and its constraints. Finding whether index i is constrained and if so accessing its constraints can then be done using $\mathcal{O}\left(\log_2(\#\mathcal{I}_c^p)\right)$ operations.

- On each processor, this class stores an array of $\#\mathcal{I}^p_{l.r.}$ integers. Finding whether an index $i \in \mathcal{I}^p_{l.r.}$ is constrained then requires finding the position r_i of i within $\mathcal{I}^p_{l.r.}$ and testing position r_i in the array whether the integer stored there is -1 (indicating that there are no constraints on i) or otherwise is an index into an array describing the constraints on i. As explained in Section 2.3.2, finding r_i can be done in $\mathcal{O}(\log_2 K^p)$ operations where K^p is the number of half-open intervals that are required to describe $\mathcal{I}^p_{l.r.}$.

Here, the second strategy requires a factor of $\#\mathcal{I}^p_{l.r.}/\#\mathcal{I}^p_c$ more memory, but it is cheaper in terms of run time if $K^p \ll \#\mathcal{I}^p_c$. The former is not a significant problem, since storing a single integer for every locally active degree of freedom is not a noticeable expense overall. Whether the latter condition is true depends on a number of application dependent factors: (i) how much local refinement is required to resolve the solution, as this influences $\#\mathcal{I}^p_c$; (ii) the ratio of the number of ghost cells (which roughly determines K^p) to the number of cells (which roughly determines $\#\mathcal{I}^p_c$ up to a factor). The ratio in the second point also depends on the number of refinement steps as well as the number of processors available.

In a number of numerical experiments, we have not been able to conclusively determine which of the two strategies above would be more efficient since the ratio of K^p to $\#\mathcal{I}^p_c$ is highly variable. In particular, neither of these numbers are uniformly much smaller than the other. `deal.II` currently implements the second strategy.

As a final note in this section, let us remark that the strategies described above turn out to be as conservative as one can be with only one layer of ghost cells: we compute even constraints for degrees of freedom located between ghost cells, and we also store the maximal set of constraints available. Coming to the conclusion that both is necessary is the result of many long debugging sessions since forgetting to compute or store constraints does not typically result in failing assertions or other easy to find errors. Rather, it simply leads to the wrong linear system with generally unpredictable, though always wrong, solutions.

2.3.3 Algorithms for setting up and solving linear systems

After creating the mesh and the index sets for degrees of freedom as discussed above, we can turn to the core objective of finite element codes, namely assembling and solving linear systems. We note that for parallel linear algebra, `deal.II` makes use of `PETSc` [4,5] and `Trilinos` [66,67], rather than implementing this functionality directly. For details see Section 2.2.

Setting up sparsity patterns

Finite element discretizations lead to sparse matrices that are most efficiently stored in compressed row format. Both `PETSc` and `Trilinos` allow application programs to pre-set the sparsity pattern of matrices to avoid re-allocating memory over and over during matrix assembly. `deal.II` makes use of this by first building objects with specialized data structures that allow the efficient build-up of column indices for each row in a matrix, and then bulk-copying all the indices in one row into the respective `PETSc` or `Trilinos` matrix classes. Both of these libraries can then store the actual matrix entries in a contiguous array in memory. We note here that each processor will only store matrix and vector rows indexed by $\mathcal{I}^p_{l.o.}$ when using either `PETSc` or `Trilinos` objects. Since $\mathcal{I} = \bigcup_p \mathcal{I}^p_{l.o.}$ and the sets $\mathcal{I}^p_{l.o.}$ are mutually disjoint, we achieve a non-overlapping distribution of rows between the available processors.

In the current context, we are interested in how pre-computing sparsity patterns can be achieved in a parallel distributed program. We can build the sparsity pattern if every processor loops over its own cells in $\mathbb{T}^p_{\text{loc}}$ and simulates which elements of the matrix would be written to if we were assembling the global matrix from local contributions. It is immediately clear that we will not only write into rows r that belong to the current processor (i.e., $r \in \mathcal{I}^p_{l.o.}$), but also into rows r that correspond to degrees of freedom owned by a neighboring processor q but located at the boundary (i.e., $r \in \mathcal{I}^p_{l.a.} \cap \mathcal{I}^q_{l.o.}$), and last but not least into rows which the degree of freedom r may be constrained to (these rows may lie in $\mathcal{I}^p_{l.r.} \cap \mathcal{I}^q_{l.o.}$).

It would therefore seem that processor p must communicate to processor q the elements it will write to in these rows in order for processor q to complete the sparsity pattern of those rows that it locally stores. One may now ask whether it is possible for processor q to determine which entries in rows corresponding to $\mathcal{I}^p_{l.a.} \cap \mathcal{I}^q_{l.o.}$ will be written to by processor p, thereby avoiding communication. This is, in fact, possible as long as there are no constrained degrees of freedom: each processor will simply have to loop over all cells $\mathbb{T}^p_{\text{loc}} \cup \mathbb{T}^p_{\text{ghost}}$, simulate assembly of the matrix, and only record which elements in rows $\mathcal{I}^p_{l.o.}$ will be written to, ignoring all writes to other rows.

Unfortunately, this process does no longer work once constraints are involved, since processors cannot always know all involved constraints. This is illustrated in Fig. 2.6. Consider the situation that the bottom three cells are owned by processor 0, and the rest by processor 1. Then $\mathcal{I}^0_{l.o.} = [0, 8], \mathcal{I}^1_{l.o.} = [9, 20]$, and these two processors will store constraints for $\mathcal{I}^0_c = \{6, 17, 19\}, \mathcal{I}^1_c = \{6, 17, 19, 20\}$ as explained in Section 2.3.2.

Consider now the matrix entries that processor 1 will have to write to when assembling on cell B (shaded yellow). Since degrees of freedom 17 and 20 are constrained to 5,10 and 10,11, respectively, after resolution of constraints we will have matrix entries $(5, 10)$ and $(5, 11)$, among others. But because $5 \in \mathcal{I}^0_{l.o.}$, these entries must be stored on processor 0. The question now is whether processor 0 could know about this without communicating with processor 1. The answer is no: we could have known about entry $(5, 10)$ by simulating assembly on cell A, which is a ghost cell on processor 0. However, processor 0 cannot possibly know about the matrix entry $(5, 11)$: the cells B and C are not in $\mathbb{T}^0_{\text{ghost}}$, and so processor 0 does not know anything about degree of freedom 20 in the first place, and certainly not that it is constrained to degrees of freedom 10 and 11.

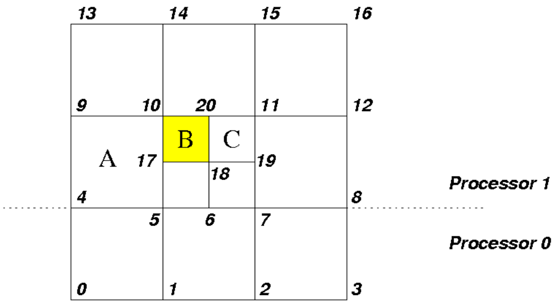


Figure 2.6: Illustrating why sparsity patterns cannot be built up without communication: Degrees of freedom associated with a Q_1 finite element on a mesh split between two processors. Processor 0 owns degrees of freedom $0 \ldots 8$, *processor 1 owns* $9 \ldots 20$.

In summary, we cannot avoid communicating entries into the sparsity pattern between processors, though at least this communication can be implemented point-to-point. We note that in the case of `Trilinos`, the `Epetra_FECrsGraph` class (implementing sparsity patterns) can take care of this kind of communication: if we add elements to rows of the sparsity pattern that are not stored on the current processor, then these will automatically be transferred to the owning processor upon calling `Epetra_FECrsGraph`'s `GlobalAssemble` function. A similar statement holds for `PETSc` objects though there does not seem to be a way to communicate entries of sparsity patterns between processors. Consequently, when interfacing with `PETSc`, we send the entries generated in rows that are not locally owned to the corresponding processor after concluding creation of the sparsity pattern. This way each processor sends one data packet with indices to each of its neighboring processors. The process is fast because each processor only has to look at the rows with indices $\mathcal{I}_{l.r.} \setminus \mathcal{I}_{l.o.}$ and all communication can be done point-to-point in a non-blocking fashion. Received indices are then inserted into the local rows of the sparsity pattern.

Assembling and solving the linear system

After pre-setting the sparsity pattern of the matrix, assembling the linear system happens in the usual way as described in Section 1.2 but only by computing contributions from all cells in $\mathbb{T}^p_{\text{loc}}$. The implications for a parallel implementation are given in Section 2.2.2. Additionally, we also must resolve constraints. Finally the entries must be transferred into the global matrix and vector objects as described in Section 2.2.2.

Once assembled, we must solve the resulting linear system that can contain billions of unknowns. We realize this using parallel Krylov methods as described in 2.2.3. `PETSc` and `Trilinos` offer a large variety of preconditioners, including highly effective algebraic multi-grid preconditioners available through the packages `hypre` [46, 47] and `ML` [49].

2.3.4 Postprocessing

Once a solution to the linear system has been computed, finite element applications typically perform a number of postprocessing steps such as generating graphical output, estimating errors, adaptively refining the mesh, and interpolating the solution from the old to the new mesh. In the following, we will briefly comment on the latter three of these points. We will not discuss generating graphical output – storing and visualizing tens or hundreds of gigabytes of data resulting from massively parallel computations is nontrivial and the realm of specialized tools not under consideration here.

Adaptive refinement of meshes

Once a solution has been computed, we frequently want to adjust the mesh to better resolve the solution. In order to drive this adaptation, we need to (i) compute error indicators for each of the cells in the global mesh, and (ii) determine which cells to refine, for example by setting a threshold on the error indicators above which a cell should be refined (and similarly for coarsening).

The literature contains a large number of methods to estimate the error in finite element solutions, see for example [1, 11, 126] and the references cited in these publications. Without going into detail, it is natural to let every processor p compute error indicators for the cells $\mathbb{T}^p_{\text{loc}}$ it owns. The primary complication from the perspective of parallelization is that in order to compute these indicators, we not only have to have access to all degrees of freedom located on cells in $\mathbb{T}^p_{\text{loc}}$, i.e., to the elements of the solution vector indexed by $\mathcal{I}^p_{l.a.}$, but frequently also solution values on all neighboring cells in order to compute jump residuals at the interfaces between cells. In other words, we need access to solution vector elements indexed by $\mathcal{I}^p_{l.r.}$ while by default every processor only stores solution vector elements it owns, i.e., $\mathcal{I}^p_{l.o.}$. Before computing error indicators, we therefore have to import the missing elements (and preferably only those since, in particular, we cannot expect to store the entire solution vector on each processor). Both `PETSc` and `Trilinos` support this kind of operation.

Once error indicators $e_i \geq 0, i \in [0, N_{\text{cells}})$, where $N_{\text{cells}} = \#\bigcup_p \mathbb{T}^p_{\text{loc}}$, have been computed, we have to decide which cells to refine and coarsen. A typical strategy is to refine and coarsen certain fractions $\alpha_r, \alpha_c \in [0,1]$ of all cells. To do that, we must compute thresholds θ_r, θ_c so that, for example, $\#\{i : e_i \geq \theta_r\} \approx \alpha_r N_{\text{cells}}$. On a single processor, this is easily achieved by sorting the e_i according to their size and choosing that error indicator as the threshold θ_r corresponding to position $\alpha_r N_{\text{cells}}$, though it is also possible to find this threshold without completely sorting the set of indicators e_i. This task can be performed using the algorithm commonly referred to as `nth_element`, which can be implemented with average linear complexity, and is, for example, part of the C++ standard library [116]. On the other hand, `nth_element` does more than we need since it also shuffles the elements of the input sequence so that they are ordered relative to the n-th element we are seeking.

In distributed parallel computations, no single processor has access to all error indicators. Consequently, we could use a parallel `nth_element` algorithm, see for example [121]. We can, however, avoid the partial sorting step by using the distributed algorithm outlined in Figure 2.7. The algorithm computes the threshold θ to an accuracy ϵ. For practical reasons, we are not usually interested in very high accuracy for these thresholds and typically set ϵ so that the while-loop terminates after, for example, at most 25 iterations. Since the interval in which θ must lie is halved in each iteration, this corresponds to a relative accuracy of $\frac{1}{2^{25}} \approx 3 \times 10^{-8}$. The compute time for the algorithm with a fixed maximal number of iterations is then $\mathcal{O}(\frac{N_{\text{cells}}}{P} \log_2 P)$, where the logarithmic factor results from the global reduce and broadcast operations. Furthermore, the constant in this complexity can be improved by letting each processor not only compute the number of cells $n_t^{1/2} = \#\{i : e_i^0 \geq m = \frac{1}{2}(b+e)\}$, but also $n_t^{1/4} = \#\{i : e_i^0 \geq \frac{1}{4}(b+e)\}$ and $n_t^{3/4} = \#\{i : e_i^0 \geq \frac{3}{4}(b+e)\}$, thereby obviating the need for any communication in the next iteration (because the data needed in the next iteration is already available) and cutting the number of communication steps in half. This procedure can of course be repeated to reduce the number of communication steps even further, at the expense of larger numbers of variables n_t sent to processor 0 in the reduction step.

In actual finite element computations, the algorithm as stated turns out to not

```
$b^p = \min e^p, e^p = \max e^p$                                      compute local min and max
$\{b, -e\}$ = MPI_AllReduce ($\{b^p, -e^p\}$, MIN)        compute global min and max
while ($|e - b| \geq \epsilon$)
        MPI_Bcast ($\{b, e\}$, 0 $\to$ all)                       broadcast current interval
        $m = \frac{1}{2}(b + e)$                                        compute interval split point
        $n_t = \#\{i : e_i^p \geq m\}$                                count local elements greater
                                                                       than $m$
        $n_t$ =MPI_AllReduce ($n_t$, SUM)                          accumulate total number of el-
                                                                       ements
        if ($n_t > \alpha N$) then $\{b, e\} = \{b, m\}$               adjust interval
                          else $\{b, e\} = \{m, e\}$
end
return $\theta = m$                                                   return threshold
```

Figure 2.7: Pseudo-code for determining a threshold θ so that approximately αN elements of a vector $(e_i)_{i=0}^{N-1}$ satisfy $e_i \geq \theta$. Each processor only stores a part e^p of n^p elements of the input vector. The algorithm runs on each processor $p, 0 \leq p < P$. This algorithm is a variant of the parallel binary search described in [27].

be very efficient. The reason for this is that for practical problems, error indicators e_i are often scattered across many orders of magnitude, with only large e_i. Consequently, reducing the interval to $\frac{1}{2^{25}}$ of its original size does not accurately determine a useful threshold value θ. This problem can be avoided by using a larger number of iterations. A better alternative is to exploit the fact that the numbers $\log e_i$ are much more uniformly distributed than e_i; one can then choose $m = \exp\left[\frac{1}{2}(\log b + \log e)\right] = \sqrt{be}$. We use this modification in our code if $b > 0$, with at most 25 iterations.

The algorithm outlined above computes a threshold so that a certain fraction of the cells are refined. A different strategy often used in finite element codes is to refine those cells with the largest indicators that together make up a certain fraction αe of the total error $e = \sum_i e_i$. This is easily achieved with minor modifications when determining n_t. In either of these two cases, once thresholds θ_c, θ_r have been computed in this way, each processor can flag those among its cells $\mathbb{T}_{\text{loc}}^p$ whose error indicators are larger than θ_r or smaller than θ_c for refinement or coarsening, respectively.

Transferring solutions between meshes

In time dependent or nonlinear problems, it is important that we can carry the solution of one time step or nonlinear iteration from one mesh over to the next mesh that we obtain by refining or coarsening the previous one. This functionality is implemented in the `deal.II` class `SolutionTransfer`. It relies on the fact that after setting refinement and coarsening flags, we can determine exactly which cells will be refined and which will be coarsened (even though these sets of cells may not coincide with the ones actually flagged, for example because the triangulation has to respect the 2:1 mesh balance invariant).

Since the solution transfer is relatively trivial if all necessary information is available locally, we describe the sequential algorithm first before discussing the modifications necessary for a scalable parallel implementation. To this end, let $x^i, i = 1 \dots I$ be the vectors that we want to transfer to the new mesh. Then the sequential algorithm begins as follows:

- On every terminal (active) cell K that will not be coarsened, collect the values $x^i|_K$ of all degrees of freedom located on K. Add the tuple $(K, \{x^i|_K\}_{i=1}^I)$ to a list of such tuples.
- On every non-terminal cell K that has 2^d terminal children $K_c, c = 1 \dots 2^d$ that will be coarsened, interpolate or project the values from the children onto K and call the result $x^i|_K$. Add the tuple $(K, \{x^i|_K\}_{i=1}^I)$ to a list of such tuples.

Next refine and coarsen the triangulation, enumerate all degrees of freedom on the new mesh, resize the vectors x^i to their correct new sizes and perform the following actions:

- On every terminal cell K, see if an entry for this cell exists in the list of tuples. If so, which will be the case for all cells that have not been changed at all and those whose children have been deleted in the previous coarsening step, extract the values of the solution $x^i|_K$ on the current cell and copy them into the global solution vectors x^i.
- On all non-terminal cells K for which an entry exists in the list of tuples, i.e., those that have been refined exactly once, extract the local values $x^i|_K$, interpolate them to the children $x^i|_{K_c}, c = 1 \dots 2^d$, and copy the results into the global solution vectors x^i.

By ordering the list of tuples in the same way as we traverse cells in the second half of the algorithm, we can make both adding an element to the list and finding tuples in the list an $\mathcal{O}(1)$ operation.

This algorithm does not immediately work for parallel distributed meshes, first because we will not be able to tell exactly which cells will be refined and coarsened without communication (a precondition for the first part of the algorithm), and second because, due to repartitioning, the cells we have after refinement on processor p may not be those for which we stored tuples in the list before refinement on the same processor.

In our parallel distributed re-implementation of the `SolutionTransfer` class, we make use of the fact that the master version of the mesh is maintained by `p4est` and stored independently of the `deal.II` object that represents the mesh including all auxiliary information. Consequently, after `deal.II` notifies `p4est` of which cells to refine and coarsen, and `p4est` performs the necessary mesh modification including the 2:1 mesh balance (see Section 2.3.1), we have the opportunity to determine which cells have been refined and coarsened by comparing the modified, `p4est`-maintained master version of the mesh and the still unchanged mesh data in `deal.II`. This allows us to create the list of tuples in the first part of the algorithm outlined above. In a second step, `deal.II` calls `p4est` to repartition the mesh to ensure a load-balanced distribution of terminal cells; in this step, `p4est` allows attaching additional data to cells that are transferred point-to-point from one processor to another. In our case, we attach the values $x^i|_K$. After partitioning the mesh and re-building the `deal.II` triangulation, we query `p4est` for the stored values on the machine the cell now belongs to. This allows us to perform the second part of the algorithm like in the serial case, without adding communication to `deal.II` itself.

2.3.5 Numerical Results

In the following, we will present two test cases that are intended to demonstrate the scalability of the algorithms and data structures discussed above. The first test case solves a 2d Laplace's equation as introduced in Section 1.1 on an sequence of adaptively refined meshes. The relative simplicity of this example implies that the solver and preconditioner for the linear system – while still the most expensive part of the program – are not completely dominating. Consequently, we will be able to better demonstrate the scalability of the remaining parts of the program. As explained in Section 1.1, the Laplace's equation plays an important prototype as it must be solved in the pressure Schur complement or for projection type methods in instationary flow problems (see Section 3.3). The solution process requires a global flow of information, which makes it a perfect candidate for analyzing parallel scalability. The second test case investigates the solution of a viscous thermal convection problem under the Boussinesq approximation. This serves as a prototype for realistic, coupled, instationary problem with non-trivial preconditioning.

The programs that implement these test cases are available as the step-40 and step-32 tutorial programs of `deal.II`, respectively, [59,79]. Tutorial programs are extensively documented to demonstrate both the computational techniques used to solve a problem as well as their implementation using `deal.II`'s classes and functions. They are licensed in the same way as the library and serve well as starting points for new programs.

The computational results shown in the following subsections were obtained on the Ranger supercomputer at the Texas Advanced Computing Center (TACC) at The University of Texas at Austin. Some computations and the majority of code testing were done on the Brazos and Hurr clusters at the Institute for Applied Mathematics and Computational Science at Texas A&M University.

A simple 2d Laplace test case

The first test case solves the scalar Laplace's equation, $-\Delta u = f$ on the unit square $\Omega = [0,1]^2$. We choose homogeneous boundary values and

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } x_2 > \frac{1}{2} + \frac{1}{4}\sin(4\pi x_1), \\ -1 & \text{otherwise.} \end{cases}$$

The discontinuity in the right-hand side leads to a sinusoidal line through the domain along which the solution $u(\mathbf{x})$ is non-smooth, resulting in very localized adaptive mesh refinement. The equations are discretized using biquadratic finite elements and solved using the conjugate gradient method preconditioned by the `BoomerAMG` implementation of the algebraic multi-grid method in the `hypre` package [46,47]. We call `hypre` through its interface to `PETSc`. Fig. 2.8 shows the solution along with an adaptive mesh at an early stage of the refinement process containing 7,069 cells and a partition onto 16 processors.

To demonstrate the scalability of the algorithms and data structures discussed, we solve the Laplace equation on a sequence of meshes each of which is derived from the previous one using adaptive mesh refinement and coarsening (the mesh in Fig. 2.8 results from three cycles of adaptation). For a given number of processors, we can then show the wall clock time required by the various operations in our program as a function of the number of degrees of freedom on each mesh in this sequence. Fig. 2.9 shows this for 256 and 4,096 processors and up to around 1.2×10^9 degrees of freedom.[1] While we have measured wall clock times for a large

---

[1] Note that the next refinement would yield a number of degrees of freedom that exceeds the range of the 32-bit signed integers used by `hypre` for indexing (`PETSc` can use 64-bit integers for this

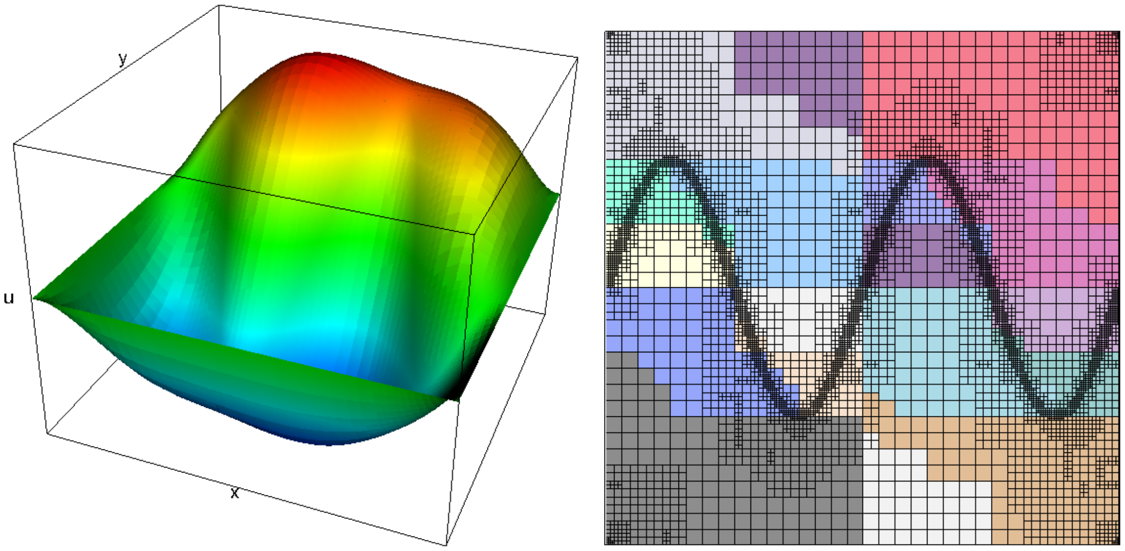


Figure 2.8: Two-dimensional scalar Laplace example. Left: Solution u on the unit square. Right: Adapted mesh at an early stage with 7,069 cells. The partition between 16 processors is indicated by colors.

number of parts of the program, the graph only labels those seven most expensive ones that together account for more than 90% of the overall time. However, as can be seen, even the remaining parts of the program scale linearly. The dominant parts of the program in terms of their wall clock time are:

- *Linear solver:* Setting up the algebraic multi-grid preconditioner from the distributed finite element system matrix, and solving the linear system with the conjugate gradient method including the application of the AMG preconditioner.

- *Copy to `deal.II`:* This is the operation that recreates the mesh in `deal.II`'s own data structures from the more compressed representation in `p4est`. The algorithm is shown in Fig. 2.3.

- *Error estimation:* Given the solution of the linear system, compute and communicate error indicators for each locally owned cell, compute global thresholds for refinement and coarsening, and flag cells accordingly (see the algorithm in Fig. 2.7).

- *Assembly:* Assembling the contributions of locally owned cells to the global system matrix and right-hand side vector. This includes the transfer of matrix and vector elements locally computed but stored on other processors.

---

purpose). Unfortunately, `Trilinos`' `Epetra` package that we use in our second numerical test case suffers from the same limitation.

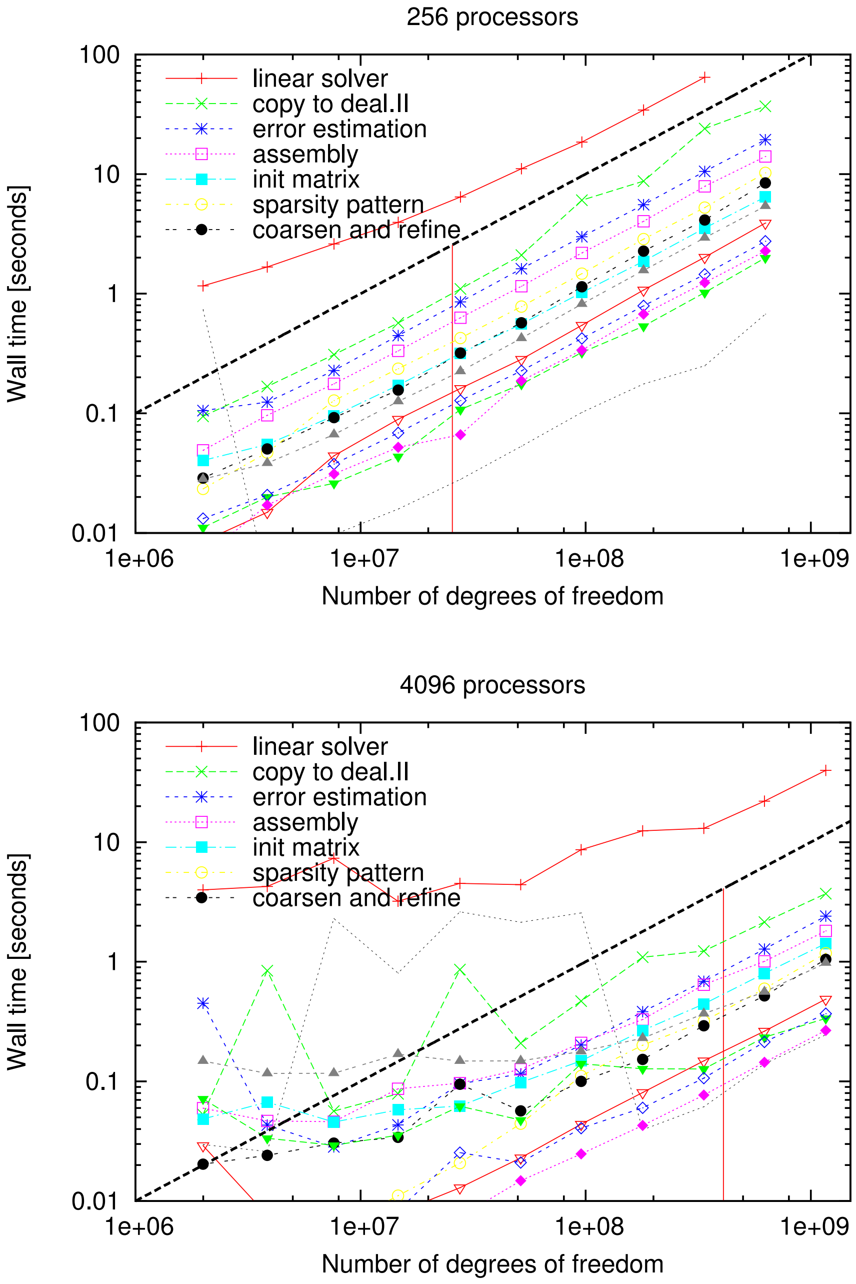


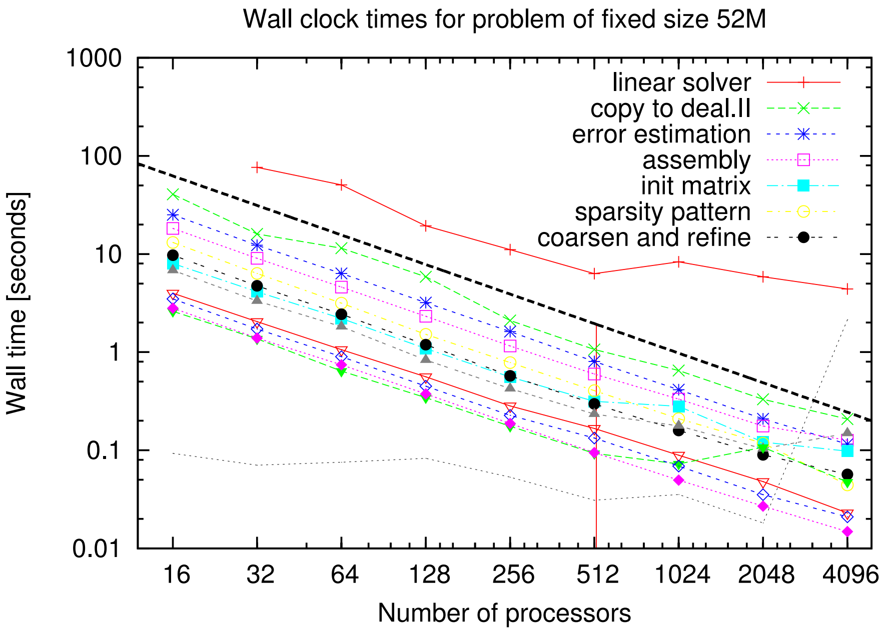
Figure 2.9: Two-dimensional scalar Laplace example. Scaling results on 256 (top) and 4,096 processors (bottom) for a sequence of successively refined grids. The various categories of wall clock times are explained in the text. The labeled categories together account for more then 90% of the total wall clock time of each cycle. In both graphs, the thick, dashed line indicates linear scaling with the number of degrees of freedom. Each processor has more than 10^5 *degrees of freedom only to the right of the vertical red line. Both the small number of elements per processor left of the vertical line and small absolute run times of a few seconds make the timings prone to jitter.*

- *Sparsity pattern:* Determine the locations of non-zero matrix entries as described in 2.3.3.
- *Init matrix:* Exchange between processors which non-locally owned matrix entries they will write to in order to populate the necessary sparsity pattern for the global matrix. Copy intermediate data structures used to collect these entries into a more compact one and allocate memory for the system matrix.
- *Coarsen and refine:* Coarsen and refine marked cells, and enforce the 2:1 cell balance across all cell interfaces (this includes the largest volume of communication within `p4est`; see Section 2.3.1).

The results presented in Fig. 2.9 show that all operations appear to scale linearly (or better) with the number of degrees of freedom whenever the number of elements per processor exceeds 10^5. For smaller element counts per processor, and run times of under a few seconds, most operations behave somewhat irregularly– in particular in the graph with 4,096 processors –, which can be attributed to the fact that in this situation there is simply not enough numerical work to hide the overhead and inherent randomness caused by communication. This behavior is most marked in the Copy-to-`deal.II` and `p4est` re-partitioning operations. (The scalability of the latter has been independently demonstrated in [25].)

Note 2.1: *The second exception is the operation that re-partitions the `p4est` representation of the mesh into equally sized chunks after each processor has refined and coarsened the locally owned cells. This operation is shown using the dashed, rather random curve without markers in Fig. 2.9 for one run of the program, but we note that the timings of this operation differ in subsequent runs with the same executable by up to an order of magnitude; the minimum time from several invocations scales linearly and would follow the lower envelope of the shown curve. Despite considerable efforts, we have been unable to find the root cause of this behavior. The unaccountable time is spent in an `MPI_Waitall` call after each processor sends information to some others and waits for its incoming packets. We have ruled out that some processors simply take longer sending their data by placing an `MPI_Barrier` call right before the wait-all operation, with no effect on the non-deterministic run time of the latter operation. We cannot support any of our other hypotheses without a significant amount of speculation.*

While the results discussed above show that a fixed number of processors can solve larger and larger problems in a time proportional to the problem's size, Fig 2.10 shows the results of a "strong" scaling experiment. Here we select two refinement levels that result in roughly 52 and 335 million unknowns, respectively, and compare run times for different numbers of processors. Again, above roughly 10^5 elements per processor we observe nearly ideal scalability of all algorithms.



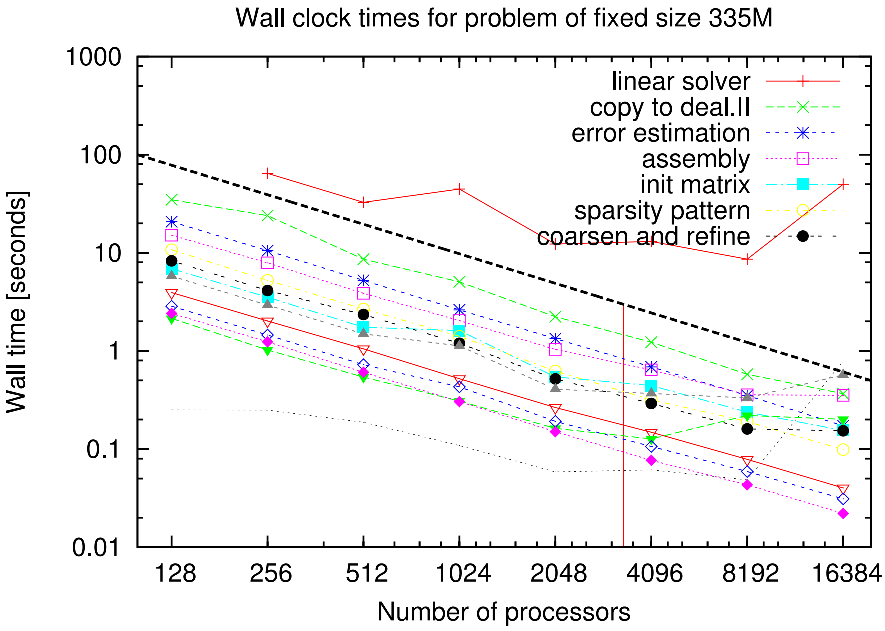


Figure 2.10: Two-dimensional scalar Laplace example. Strong scaling results for a refinement level at which meshes have approximately 52 million (top) and 335 million unknowns (bottom), for up to 16,384 processors. The thick, dashed line indicates linear scaling with the number of processors. Each processor has more than 10^5 *degrees of freedom only to the left of the vertical red line.*

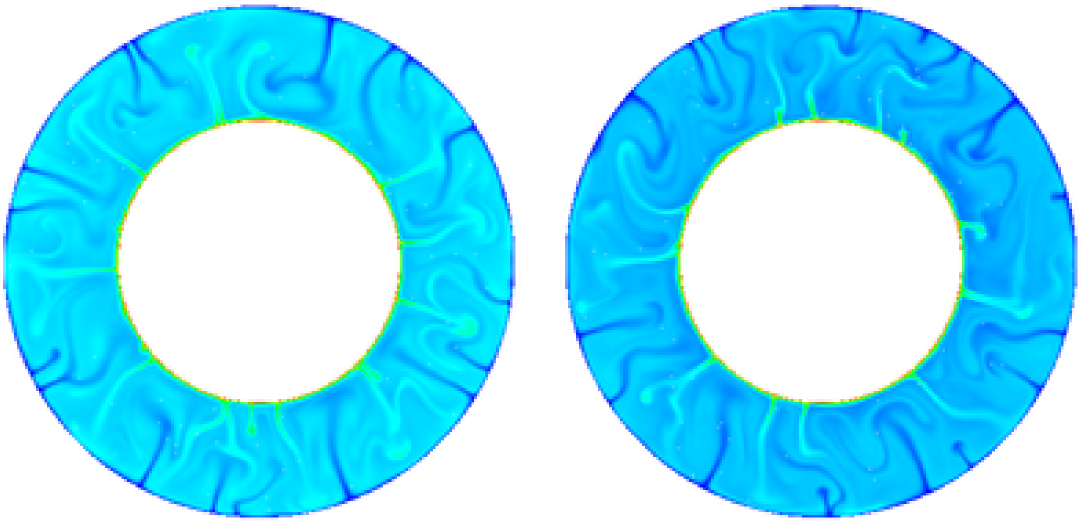


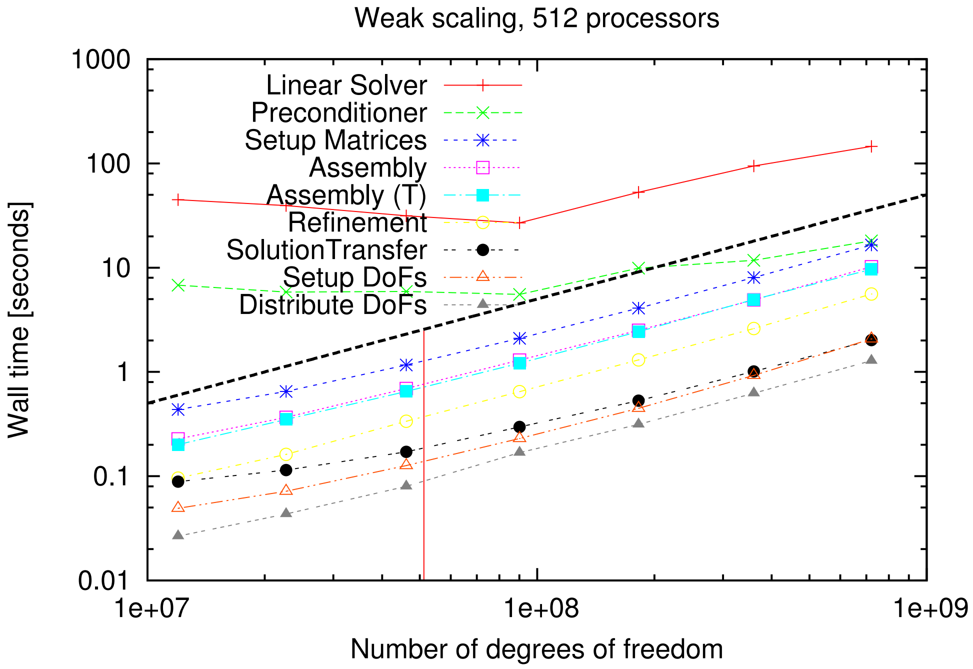
Figure 2.11: Solution of the mantle convection test case at two instants in time during the simulation. Mesh adaptation ensures that the plumes are adequately resolved.

A thermal convection test case

The second test case considers solving the equations that describe convection driven by buoyancy due to temperature variations. For details and more numerical experiments for the test case – which is also published as step-32 in `deal.II` – we refer to Section 4.3.

Let us note that this example serves as a realistic and complex test case. It is a coupled system for velocity, pressure, and temperature. The adaptivity is necessary for the correct temporal development of the solution. Typical solutions at two time steps during the simulation are shown in Fig. 2.11. It also features saddle point solvers for the Stokes part, which is discussed in Section 3.

Figure 2.12 shows scaling results for this test case. There, we time the first time step with $t_n \geq t^* = 10^5$ years for a number of different computations with a variable number of cells (and consequently a variable number of time steps before we reach t^*). The "weak" scaling shown in the left panel indicates that all operations scale linearly with the overall size of the problem, at least if the problem is sufficiently large. The right panel demonstrates strong scalability. Here, scalability is lost once the number of degrees of freedom per processor becomes too small; this happens relatively soon due to the small size of the problem shown here (22 million unknowns overall).



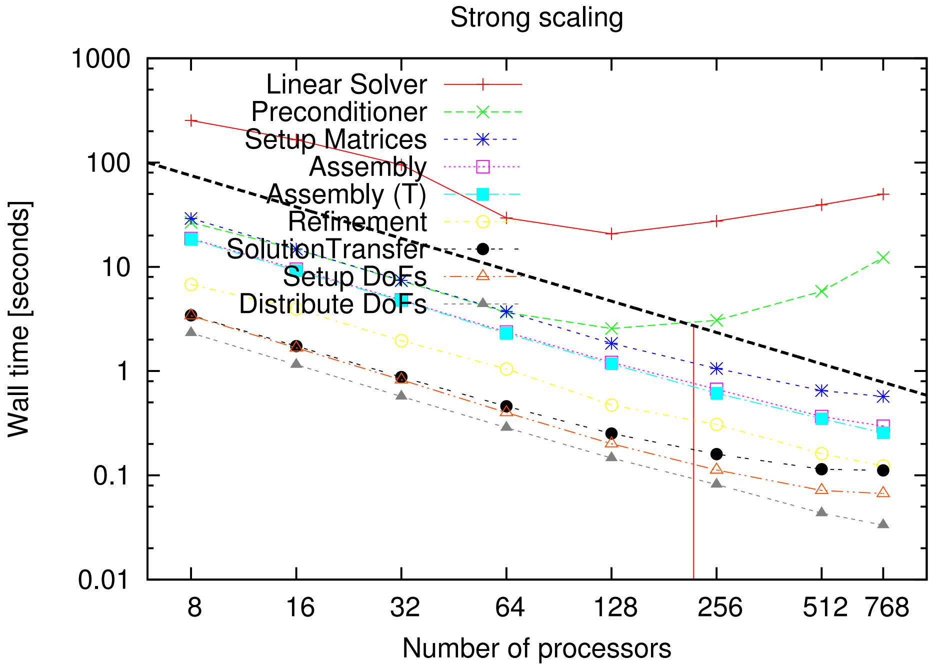


Figure 2.12: Thermal convection example. Weak scaling with 512 processors (top) and strong scaling with roughly 22 million unknowns (bottom). In both graphs, the thick, dashed line indicates optimal scaling. In the left graph, processors have more than 10^5 *degrees of freedom to the right of the vertical red line; in the right graph to the left of the vertical red line.*

3 Solver Framework for the Oseen Problem

This chapter discusses the efficient solution of Oseen-type problems. Preconditioning for those problems is crucial and an active research topic. A fast and robust solution technique is crucial, because solving Oseen-type problems form the main ingredient for solving the Navier-Stokes equations (also see Section 1.5.2). It is common to have stabilization terms as discussed in Section 1.5.5, but preconditioners typically can not cope or behave worse with stabilization, see Section 3.3. A common ansatz is to treat the discrete saddle-point problem with a block-triangular preconditioner as described in Section 3.2. There, the handling of the so called Schur complement is the main difficulty, so Section 3.3 tries to give an overview over the common methods. We present a new preconditioner that is also built around the block-triangular structure in Section 3.4. It is based on an augmented Lagrangian approach, that uses the algebraic properties of Grad-Div stabilization.

Large parts of this chapter also appear in our publication [63], and my diploma thesis [58] was the starting point of the research in Grad-Div based preconditioning.

3.1 The linear saddle-point system

As described in Section 1.5 solving the Navier-Stokes equations boils down to repeatedly solving Oseen problems. The stationary and linear system can be written as

$$\begin{aligned} -\nabla\cdot(\nu\nabla \boldsymbol{u}) + (\boldsymbol{b}\cdot\nabla)\boldsymbol{u} + c\boldsymbol{u} + \nabla p &= \boldsymbol{f} && \text{in } \Omega \\ \nabla\cdot\boldsymbol{u} &= 0 && \text{in } \Omega \\ \boldsymbol{u} &= 0 && \text{on } \partial\Omega. \end{aligned} \tag{3.1}$$

in the continuous form. For the ease of presentation we assume homogeneous Dirichlet boundary conditions for the velocity. Note that we skip stabilization terms for now. Grad-Div stabilization will play an important role in Section 3.4, though.

Moreover, $\boldsymbol{b} \in [L^2(\Omega) \cap W^{1,\infty}(\Omega)]^d$ is a vector field describing the convection acting on the velocity and is typically given by the velocity stemming from a linearization scheme, see Section 1.5.4. The constant reaction coefficient $c \geq 0$ enters the system due to the time discretization and is proportional to the inverse of the time step size. The case $c = 0$ appears after linearizing stationary Navier-Stokes problems. The relation between the parameters ν, $\|b\|$, and c obviously influence the character of the underlying problem. In the limit with only one non-zero parameter the PDE reduces to an equation of second, first, or zeroth oder, respectively. For $\boldsymbol{b} = 0$ and $c = 0$ the system becomes a Stokes problem. The goal is to have preconditioners to handle all cases of parameter ranges and optimally have them automatically adapt to the current case.

For an inf-sup stable discretization (see Section 1.5.3) the resulting linear system reads

$$Mx = G \tag{3.2}$$

with unknowns $x = (U, P)^T$, right-hand side $G = (F, 0)^T$ and block matrix

$$M = \begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix}.$$

The blocks A and B correspond to the bilinear forms a and b as in Section 1.5.3, respectively. The matrix A is positive but non-symmetric for $\boldsymbol{b} \neq 0$. The linear system is sparse as it is typical for linear systems stemming from finite element discretizations. The matrix M forms a saddle-point structure. For an extensive discussion on saddle point problems and their difficulties we refer to [14].

Solving Oseen-type systems has a long history and there are dozens of different solution approaches ranging from Uzawa type methods [21] to projection methods [55], special multi-grid methods [70] or block saddle point preconditioners [38,39,41]. A good overview for solvers for saddle point problems is given in [14].

Here, we consider an iterative Krylov method to solve the arising linear system with a block saddle point preconditioner. A good option is GMRES or flexible GMRES when inner solvers are involved as described in Section 1.4.

Since the linear system is badly conditioned preconditioning is mandatory. The biggest challenge is finding a preconditioner that performs equally well for different mesh sizes h and different coefficient ranges of c, ν, and $\|\boldsymbol{b}\|$. In the case of the Stokes problem (for $\mathbf{b} = 0$) the preconditioning is much simpler, cf. [123]. The most challenging configuration is $\nu \ll 1$, $c = 0$, and $\boldsymbol{b} \neq 0$, cf. [38,39,41].

3.2 Block-triangular Preconditioning

We apply right preconditioning as described in Section 1.4 and [109] with an operator $\mathcal{P}^{-1}$ for solving the system $Mx = G$ (see (3.2)) and calculate the solution $x = \mathcal{P}^{-1}y$ from the auxiliary variable y, which is given as the solution of

$$M\mathcal{P}^{-1}y = G.$$

In general, $\mathcal{P}^{-1}$ is not given by a matrix, since building this inverse is typically not appropriate. Here, we define $\mathcal{P}^{-1}$ as an implicitly defined operator given in a block-triangular way (see [14]):

$$\mathcal{P}^{-1} := \begin{pmatrix} \widetilde{A} & B^T \\ 0 & \widetilde{S} \end{pmatrix}^{-1} = \begin{pmatrix} \widetilde{A}^{-1} & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} I & B^T \\ 0 & -I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & \widetilde{S}^{-1} \end{pmatrix}, \tag{3.3}$$

where $\widetilde{S}$ is an approximation of the Schur complement

$$S = -BA^{-1}B^T \tag{3.4}$$

and $\widetilde{A}$ of the velocity block A, respectively. Thus, applying the preconditioner involves one solve for $\widetilde{A}$, one solve for $\widetilde{S}$, and one matrix-vector multiplication with B^T.

Now the goal is to define good and computationally cheap approximations for $\widetilde{A}$ and $\widetilde{S}$. Exact solves would result in at most two outer GMRES iterations, see [14].

The approximation of A can be handled with a generic preconditioner. One can decide to split the system into velocity components and one is left with discretized convection diffusion operators. With the coupling between the components it might be advised to keep them coupled and solve for them at the same time. Either way, standard methods like algebraic or geometric multi-grid work well, see also Section 1.4 and 2.2.3. The approximation of the Schur complement is more involved and is discussed next.

3.3 Schur complement preconditioning

The Schur complement

$$S = -BA^{-1}B^T$$

can not be treated with standard preconditioners as explicitly forming the inverse of A is way to expensive to be of practical use. Note that we do not need an exact solver with the matrix S because the error of the solution process is controlled via the outer (F-)GMRES iteration. Even quite rough approximations are enough here. We will give an overview over some common preconditioners for the Schur complement.

3.3.1 Diffusion-reaction preconditioner

The idea of this rather common approximation is to ignore the contribution of the convection term in the matrix A in the Schur complement and treat the diffusion and reaction term separately. The standard design for the Schur complement approximation for the Stokes problem using a pressure mass matrix is included in this model. We will base the Grad-Div preconditioner (see Section 3.4) on this design here. The following construction of the approximations are based on [123].

The idea is to separately look at the main building blocks of A. The matrix A can be written as

$$A = \nu L_u + cM_u + N_u + R_u,$$

where L_u represents the diffusion term and M_u is the mass matrix of the velocity space. N_u represents the convective term and R_u the Grad-Div term. Assuming that the diffusion part is dominant the Schur complement can be approximated by

$$S^{-1} \approx -\left[B(\nu L_u)^{-1}B^T\right]^{-1} \approx -\nu M_p^{-1}, \tag{3.5}$$

where M_p is the mass matrix in the pressure space. The approximation can be motivated by assuming that the continuous operators commute, see [123] for details. Note, that we assume ν to be constant. Using the mass matrix for approximating the Schur complement for the Stokes problem is well known – there, one typically normalizes the viscosity to $\nu = 1$.

Similarly, for dominating reaction the Schur complement can be approximated by

$$S^{-1} \approx -\left[B(cM_u)^{-1}B^T\right]^{-1} \approx -cL_p^{-1}, \tag{3.6}$$

where L_p is the stiffness matrix of the pressure Poisson problem with Neumann boundary conditions. It is not known, how to treat the convective term in a similar way. With (3.5) and (3.6) one has good preconditioners for the Schur complement in case of dominating diffusion and reaction, respectively.

Note that the question of the correct boundary conditions is a difficult topic. As a rule of thumb one replaces Dirichlet boundary conditions with Neumann conditions in the Schur complement and vice versa. Neither is optimal, see [123].

To automatically switch between the diffusion and reaction based preconditioners,

$$\widetilde{S}^{-1} = -\nu M_p^{-1} - c L_p^{-1}, \tag{3.7}$$

is suggested in [123], which works remarkably well as long the problem is not convection dominated.

3.3.2 Alternative preconditioners

Before we present a new preconditioner for the Schur complement we take a look at two common approaches from the literature. For more details see [58].

The pressure convection diffusion preconditioner (or PCD for short) introduced in [75] approximates the Schur complement as

$$\widetilde{S}^{-1} = -M_p^{-1} F_p A_p^{-1},$$

where the matrices are operators in the pressure space: M_p is the mass matrix, F_p a scalar convection-diffusion operator corresponding to the convection-diffusion part in A, and A_p a Laplacian.

The BFBT preconditioner (see [39]) does not require any new matrices and is also derived by commutating operators. It approximates the Schur complement by

$$\widetilde{S}^{-1} = \left(BB^T\right)^{-1} \left(BAB^T\right) \left(BB^T\right)^{-1}.$$

Both preconditioners give an h-independent preconditioner, but both introduce a slight ν-dependency. For very small viscosities as the cavity in Section 3.4.3 they fail to converge. In the PCD method one has to deal with appropriate boundary conditions for the new matrices, which is a disadvantage. Solving problems with BB^T like in the BFBT preconditioner is difficult, as the product is a lot denser and not building the product makes preconditioning difficult.

3.4 A preconditioner using Grad-Div stabilization

Recently, it has been shown that *augmented Lagrangian* are very useful for the construction of a preconditioner for the Oseen problem. Augmented Lagrangian approaches are well known and are used in various applications, cf. [48, 53] and references therein. In [15] Benzi and Olshanskii present an augmented Lagrangian-based preconditioner for the Oseen problem that shows impressive results for various h and ν. Due to difficulties in solving the augmented velocity block, a *modified augmented Lagrangian* formulation is presented in [16] and analyzed further in [17]. In the original approach [15] the velocity block is augmented with an algebraic term possessing a large kernel. This gives rise to an efficient preconditioner for the Schur complement but complicates the solve of the velocity block. Moreover, assembling the augmentation term is quite expensive, since a product of sparse matrices has to be computed. The modified version in [16] simplifies the solution of the velocity block by only applying the augmentation to the upper right blocks. Unfortunately at the same time this spoils the quality of the Schur complement approximation and leads to larger number of iterations.

Here, we explain that one can consider Grad-Div stabilization as a different discretization of the augmented Lagrangian term. This motivates the construction of a preconditioner where we replace the augmented Lagrangian term with Grad-Div stabilization. This approach is preferable, because this removes the difficulty of applying the augmentation. Moreover, the numerical experiments in Section 3.4.3 show clearly that the number of iterations for the saddle point problem stays independent of the problem.

3.4.1 The augmented Lagrangian method

In the augmented Lagrangian method one chooses a suitable parameter $\gamma > 0$ and a matrix W to replace the linear system (3.2):

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

with the augmented system

$$\begin{pmatrix} A + \gamma B^T W^{-1} B & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} U \\ P \end{pmatrix} = \begin{pmatrix} F \\ 0 \end{pmatrix}.$$

The system is equivalent because $BU = 0$ is valid for the solution of the linear system due to the second equation.

This linear system is now preconditioned in the same way as before. The augmentation term obviously enters the Schur complement and it is shown in [15] and [58], that

$$\left[B \left(X + \gamma B^T W^{-1} B \right)^{-1} B^T \right]^{-1} = \gamma \left(B X^{-1} B^T \right)^{-1} + \gamma W^{-1}.$$

In [15] the authors suggests to use $W^{-1} = M_p^{-1}$ (or an approximation). This way the diffusion-reaction preconditioner (3.7) from Section 3.3.1 now reads

$$\widetilde{S}^{-1} = -(\nu + \gamma) M_p^{-1} - c L_p^{-1},$$

This augmented Lagrangian preconditioner allows to shift the preconditioner into the diffusion dominated case by increasing the constant γ without modifying the solution.

The augmentation term possess a large kernel. This makes solving for the velocity block rather difficult. The approach has several other disadvantages that are discussed in Section 3.4.4. The Grad-Div preconditioner is going to be similar to this method, but does not use exactly the same augmentation term, but instead Grad-Div stabilization for augmentation.

3.4.2 The preconditioner

Similar to approximations for diffusion and reaction in Section 3.3.1, we will look at the term in the A block which stems from the Grad-Div stabilization. We enhance the system matrix (3.7) to account for Grad-Div stabilization in A by adding the term $\gamma(\nabla \cdot \boldsymbol{u}_h, \nabla \cdot \boldsymbol{v}_h)$, see Section 1.5.5 for details. The resulting preconditioner we are going to derive is again given by

$$\widetilde{S}^{-1} = -(\nu + \gamma)M_p^{-1} - cL_p^{-1}. \tag{3.8}$$

Let $\pi : Q \to Q_h$ be the orthogonal L^2-projector, i.e.,

$$(p - \pi p, q) = 0 \quad \forall q \in Q_h.$$

For $\boldsymbol{u}_h, \boldsymbol{v}_h \in \boldsymbol{V}_h$ the Grad-Div term can be split into the following sum:

$$(\nabla \cdot \boldsymbol{u}_h, \nabla \cdot \boldsymbol{v}_h) = (\pi(\nabla \cdot \boldsymbol{u}_h), \nabla \cdot \boldsymbol{v}_h) + ((I - \pi)(\nabla \cdot \boldsymbol{u}_h), \nabla \cdot \boldsymbol{v}_h). \tag{3.9}$$

Using the fluctuation operator $\kappa := I - \pi$ and the projection property of π we obtain

$$(\nabla \cdot \boldsymbol{u}_h, \nabla \cdot \boldsymbol{v}_h) = (\pi(\nabla \cdot \boldsymbol{u}_h), \pi(\nabla \cdot \boldsymbol{v}_h)) + (\kappa(\nabla \cdot \boldsymbol{u}_h), \kappa(\nabla \cdot \boldsymbol{v}_h)). \tag{3.10}$$

We call the first part *algebraic term* and the second part *stabilizing term*. The following lemma shows that the algebraic term can be written as a product of known matrices.

Lemma 3.1: *The discretized algebraic term of the Grad-Div stabilization is given by*

$$(\pi(\nabla \cdot \phi_j), \pi(\nabla \cdot \phi_i)) = (B^T M_p^{-1} B)_{ij} \qquad \forall i,j \in \{1, \dots, n\}$$

using basis $\{\phi_i\}_{i=1}^n$ *of* $\boldsymbol{V}_h$ *and* $\{\psi_j\}_{j=1}^m$ *of* Q_h. *The matrices* B *and* M_p *are defined by* $(B)_{ij} := (\psi_i, \nabla \cdot \phi_j)$ *and* $(M_p)_{ij} := (\psi_i, \psi_j)$.

Proof: We first define a matrix representation P of the projection π with

$$\pi(\nabla \cdot \phi_j) = \sum_{i=1}^{m} P_{ij}\psi_i \quad \forall j = 1, \ldots, n.$$

Plugging in the definition of π and P into B gives after some rearrangement:

$$\begin{aligned}(B)_{ij} &= (\psi_i, \nabla \cdot \phi_j) = (\pi(\nabla \cdot \phi_j), \psi_i) = (\sum_{k=1}^{m} P_{kj}\psi_k, \psi_i) \\ &= \sum_{k=1}^{m} P_{kj}(\psi_k, \psi_i) = \sum_{k=1}^{m}(\psi_i, \psi_k)P_{kj} = (M_p P)_{ij}.\end{aligned}$$

The augmentation term $B^T M_p^{-1} B$ can now be written as

$$\begin{aligned}(B^T M_p^{-1} B)_{ij} &= (B^T M_p^{-1} M_p P)_{ij} = (B^T P)_{ij} = \sum_{k=1}^{m} (B^T)_{ik} P_{kj} \\ &= \sum_{k=1}^{m} \left(P_{kj}(\psi_k, \nabla \cdot \phi_i)\right) = \left(\sum_{k=1}^{m} P_{kj}\psi_k, \nabla \cdot \phi_i\right) \\ &= (\pi(\nabla \cdot \phi_j), \nabla \cdot \phi_i) = (\pi(\nabla \cdot \phi_j), \pi(\nabla \cdot \phi_i)),\end{aligned}$$

which shows the proposition. □

Adding the term $B^T M_p^{-1} B$ to the system block A is known as the *augmented Lagrangian* approach, see [15], and does not change the solution due to $B^T M_p^{-1} B U = 0$ for a solution (U, P) of the linear system (3.2). Nevertheless, it modifies the algebraic properties of the velocity-velocity block.

The second term $(\kappa(\nabla \cdot \boldsymbol{u}_h), \kappa(\nabla \cdot \boldsymbol{v}_h))$ contains the stabilization for which Grad-Div is used. The difference between discretized Grad-Div stabilization (R_u) and augmentation $B^T M_p^{-1} B$ can be written as

$$(R_u - B^T M_p^{-1} B)_{ij} = (\kappa(\nabla \cdot \phi_j), \kappa(\nabla \cdot \phi_i)).$$

The stabilizing term in the Grad-Div stabilization vanishes for $h \to 0$ and thus in the limit only the algebraic augmentation remains:

Lemma 3.2: *Let $(\boldsymbol{u}_h, p_h) \in \boldsymbol{V}_h \times Q_h$ be a solution of the stabilized linear system (1.15) with corresponding degrees of freedom (U, P). Then, we obtain for Taylor-Hood elements Q_{k+1}/Q_k, $k \geq 1$, and a sufficiently smooth solution $(\boldsymbol{u}, p)$ of the continuous, weak Oseen problem (1.12), i.e., $\boldsymbol{u} \in [H^{k+1}(\Omega)]^d$ and $p \in H^k(\Omega)$,*

$$\left\| \left(R_u - B^T M_p^{-1} B\right) U \right\|_{\mathbb{R}^n} \leq C h^{k+(d-2)/2} \left(\|\boldsymbol{u}\|_{k+1}^2 + \|p\|_k^2 \right)^{1/2}.$$

Proof:

Assume $\boldsymbol{u}_h, \boldsymbol{v}_h \in \boldsymbol{V}_h$. Let $\|\cdot\|_{\mathbb{R}^n}$ denote the Euclidean and $\|\cdot\|_0$ the L^2 norm. Using the basis representations

$$\boldsymbol{u}_h = \sum_{i=1}^{n} U_i \phi_i, \qquad \boldsymbol{v}_h = \sum_{i=1}^{n} V_i \phi_i$$

for $\boldsymbol{u}_h, \boldsymbol{v}_h \in \boldsymbol{V}_h$, we obtain the assertion

$$\begin{aligned}
&\| \left(R_u - B^T M_p^{-1} B\right) U \|_{\mathbb{R}^n} \\
&\quad = \sup_{\|V\|_{\mathbb{R}^n}=1} V^T \left(R_u - B^T M_p^{-1} B\right) U \\
&\quad = \sup_{\|V\|_{\mathbb{R}^n}=1} (\kappa(\nabla \cdot \boldsymbol{v}_h), \kappa(\nabla \cdot \boldsymbol{u}_h)) \\
&\quad \leq \sup_{\|V\|_{\mathbb{R}^n}=1} \|\kappa(\nabla \cdot \boldsymbol{v}_h)\|_0 \|\kappa(\nabla \cdot \boldsymbol{u}_h)\|_0 \\
&\quad \leq \|\kappa\|^2 \sup_{\|V\|_{\mathbb{R}^n}=1} \|\nabla \cdot \boldsymbol{v}_h\|_0 \|\nabla \cdot (\boldsymbol{u}_h - \boldsymbol{u})\|_0 \\
&\quad \leq \underbrace{\|\kappa\|^2}_{=:T_1} \underbrace{\sup_{\|V\|_{\mathbb{R}^n}=1} \|\nabla \cdot \boldsymbol{v}_h\|_0}_{=:T_2} \underbrace{\|\nabla \cdot (\boldsymbol{u}_h - \boldsymbol{u})\|_0}_{=:T_3}.
\end{aligned}$$

We have $T_1 \leq C_1$, because the fluctuation operator κ is continuous. The inverse inequality gives

$$T_2 = \sup_{\boldsymbol{v}_h} \frac{\|\nabla \cdot \boldsymbol{v}_h\|_0}{\|V\|_{\mathbb{R}^n}} \leq C' \sup_{\boldsymbol{v}_h} \frac{h^{-1}\|\boldsymbol{v}_h\|_0}{\|V\|_{\mathbb{R}^n}}$$

and [77], Theorem 3.43 gives

$$\|V\|_{\mathbb{R}^n} \geq C h^{-d/2} \|\boldsymbol{v}_h\|_0$$

with $C > 0$ independent of h. From this it follows (because of $d \geq 2$):

$$T_2 \leq \frac{C'}{C} \sup_{\boldsymbol{v}_h} h^{-1+d/2} \frac{\|\boldsymbol{v}_h\|_0}{\|\boldsymbol{v}_h\|_0} = C_2 h^{-1+d/2}.$$

The a priori error estimation for the Grad-Div stabilized Oseen problem in [89] (Corollary 3.3) gives for sufficiently smooth solutions $\boldsymbol{u}, p$:

$$T_3^2 = \|\nabla \cdot (\boldsymbol{u}_h - \boldsymbol{u})\|_0^2 \leq C_3 h^{2k} \|\boldsymbol{u}\|_{k+1}^2 + C_4 h^{2k} \|p\|_k^2.$$

Putting the terms together finally gives

$$\| \left(R - B^T M_p^{-1} B\right) U\|_{\mathbb{R}^n} \leq C h^{k+(d-2)/2} \left(\|\boldsymbol{u}\|_{k+1}^2 + \|p\|_k^2\right)^{1/2} \to 0 \quad \text{for } h \to 0.$$

$\square$.

The lemma explains why the preconditioner works and behaves very similar to the augmented Lagrangian approach. The results in [16], which show h and ν independent iteration numbers, can therefore expected to be achieved here, too. This is confirmed in Section 3.4.3.

One can interpret the stabilizing effect of Grad-Div as adding a penalty term for the fluctuations of the divergence given by the projection π. The term closely resembles projection-based stabilization, though it is not a local projection and thus can not be assembled easily.

In summary we can view Grad-Div stabilization as the sum of an algebraic term known as augmented Lagrangian and a projection-based stabilization. With this knowledge we can take advantage of the augmented Lagrangian preconditioner as described in Section 3.4.1, because using Grad-Div stabilization effectively also augments the linear system in the exact same way.

The Schur complement for the augmented matrix A with the algebraic term $B^T M_p^{-1} B$ can be simplified to

$$\left[B(A + \gamma B^T M_p^{-1} B)^{-1} B^T\right]^{-1} = \left(B A^{-1} B^T\right)^{-1} + \gamma M_p^{-1}. \tag{3.11}$$

Therefore, in [15] the authors propose to approximate the Schur complement by

$$\widetilde{S}^{-1} = -(\nu + \gamma) M_p^{-1}.$$

Note, that in contrast to the approximation for the diffusion term, (3.11) is exact if B is assumed to have full rank. This only accounts for the algebraic component in the Grad-Div stabilization.

We propose an extension for instationary problems, which is motivated as already explained in Section 3.2. It is not strictly necessary for $c \neq 0$ but accelerates the solution process especially for large c. With this we arrive at (3.8). One can decide on a case by case basis to not implement the last part. Obviously, the coefficient c in there automatically reduces the influence of that term for large time steps and stationary problems.

In short we can use the same approximation for the Schur complement as in the augmented Lagrangian approach, but we do not need to add the augmentation to the A block. This gives a huge advantage over the AL approach, since the solution of the augmented matrix

$$A_\gamma = A + \gamma B^T M_p^{-1} B$$

is very costly. On the one hand assembling A_γ is extremely expensive. In [16] the authors present various tricks like lumping the mass matrix and moving to a cheaper approximated AL formulation where only part of the augmentation is applied. The problem is that the product $B^T B$ possesses much more non-zero entries than A itself. Building a product of sparse matrices is also computationally expensive as one can not easily generate a correct sparsity pattern beforehand. Of course, one can avoid the assembling of A_γ and only supply it as an operator. But then one can not apply preconditioners like algebraic multi-grid or ILU decompositions. On the other hand the iterative solution of A_γ becomes difficult due to the large kernel of $B^T M_p^{-1} B$.

All these problems are not present in our case. The matrix A which already contains the augmentation through Grad-Div stabilization is easy to assemble. One can use various kind of solvers directly.

Replacing the augmentation in $\tilde{A}_\gamma$ by Grad-Div stabilization was proposed and tested in [58] and is also suggested in a comment in [16].

3.4.3 Numerical Results

For the numerical tests we have used the finite element library *deal.II*, see [7,9]. The computations have been performed on unstructured quadrilateral meshes, see Figure 3.1 for an example. We construct a series of those meshes for the parameter studies. Unstructured meshes are more realistic and naturally arise when dealing with complex geometries. Moreover, super-convergence effects are avoided and more realistic error bounds are achieved.

For the outer iteration a flexible GMRES method (see [108,109]) is used. Standard Krylov methods can not be applied, since we use iterative solvers within the preconditioner. Therefore, the preconditioner can not be considered as constant during the outer iterations. The outer iteration loop is stopped when the residual is dropped by a factor of $1e-10$ relative to the starting residual. This stopping criterion is more strict compared to other papers analyzing preconditioners and leads to higher iteration numbers. We have chosen this convergence criterion, since it allows us to see trends easier due to the higher number of iterations. Moreover, a softer criterion can be misleading, since the errors of the solution are very often dominated by the iterative process and not by the approximation properties of the mesh and the finite element space. This is especially true for higher order elements.

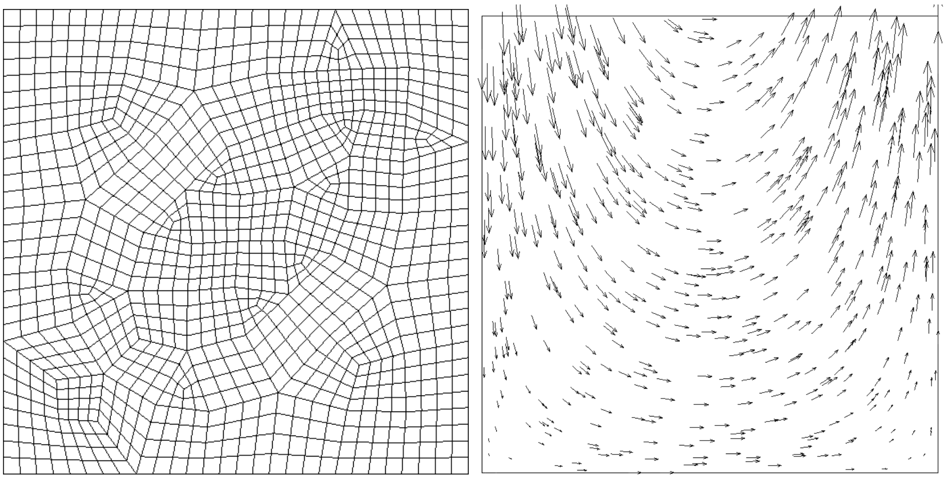


Figure 3.1: Example for an unstructured mesh (left). A series of those meshes is used for the computations to avoid super-convergence effects. The solution of Problem 1 is shown on the right-hand side.

The inner blocks are solved using the direct solver UMFPACK, see [37]. If we use a Krylov method for the inner solve, it is stated.

The Oseen problem

Problem 1 is defined in the domain $\Omega = (0,1)^2$, see [51], example 1. The right-hand side $\boldsymbol{f}$ is calculated from the smooth reference solution

$$\begin{aligned}\boldsymbol{u} &= (\sin(\pi x), -\pi y \cos(\pi x))^T,\\ p &= \sin(\pi x)\cos(\pi y)\end{aligned}$$

with convection vector $\boldsymbol{b} = \boldsymbol{u}$ or $\boldsymbol{b} = 0$, see Figure 3.1 for an illustration of the solution. Note that ν and c can be chosen arbitrarily. Thus, we can test coefficient choices including the Stokes problem. The numerical error can be calculated as the difference between the discrete solution and the reference solution. The case $\boldsymbol{b} = \boldsymbol{u}$ is more complicated than examples often chosen for Oseen problems and resembles a linearization step in a Navier-Stokes problem. The smooth solution enables higher order elements to achieve better convergence rates.

Problem 2 is a modified Green-Taylor vortex for a fixed time step and without an exponential decaying term:

$$\begin{aligned}\boldsymbol{u} &= (-\cos(\omega\pi x)\sin(\omega\pi y), \sin(\omega\pi x)\cos(\omega\pi y))^T,\\ p &= -\frac{1}{4}\cos(2\omega\pi x) - \frac{1}{4}\cos(2\omega\pi y).\end{aligned}$$

Considering $\Omega = (0,1)^2$ the constant ω determines the number of vortices in x- and y-direction and is set to $\omega = 4$. $\boldsymbol{b}$ and the right-hand side $\boldsymbol{f}$ are defined in the same way as in Problem 1. This results in a more complex structure compared to *Problem 1*. Care needs to be taken to not get over-stabilization as can be seen from plots in Figure 3.2.

Figure 3.2 shows the influence of the stabilization on the quality of the solution and the number of iteration steps of the solver. We consider *Problem 1* and *Problem 2* with $\boldsymbol{b} = \boldsymbol{u}$. A similar value for the optimization of both would be desirable. The four plots in Figure 3.2 show different configurations. For large viscosities (upper left) stabilization does not improve the solution. Only for $\gamma > 1$ we see a slight influence. For smaller viscosity $\nu = 1e-3$ there is a clear minimum around $\gamma = 0.3$ for problem 1. We can observe this behavior with and without reaction term (upper right and lower right). The fine structures in *Problem 2* on the other hand are already damped too much with a γ of that size as can be seen in the lower left. The optimum moves to $\gamma = 0.03$ there.

The number of outer iterations drops for larger γ. The number of outer iterations required for the optimal γ from the stabilization point of view is around 10 to 30. When using an iterative solver the difficulty of solving for A increases with larger γ. The solution time does not depend on γ for a direct solver of the A block. For an iterative solver it depends heavily on the iterative algorithm used for the A block. For an estimate of the total cost we also plot the number of total inner iterations required to solve the whole system. In our applications the A block is solved using the GMRES method with ILU(0) preconditioning and diagonal strengthening. While the number of inner iteration increases for larger γ, the number of outer iterations decreases. Fortunately, the optimal choices of γ lie close to the optimal choices with respect to the stabilization. Choosing a worse preconditioner for A, like SOR, the optimum would shift to the right. Thus, the results for the sum of the inner iterations have to be taken with cautiousness as it depends on the preconditioner used. We still include the total number of inner iterations, because it shows, that it is possible to choose γ in such a way that the number of iterations and the error of the solution are small at the same time. All in all the number of outer iterations does not crucially depend on the choice of γ. It is reasonable to calculate solutions without the optimal parameter at hand.

Table 3.1 shows that the number of outer iterations does not depend on the element order or the mesh size. For sufficiently large γ the number of outer iterations is also independent of ν. The optimal value of $\gamma = 0.31$ results in a slightly larger number of iterations. For smaller γ there is a slight dependency on ν. This is (not surprisingly) comparable to the augmented Lagrangian approach.

In Table 3.2 one can see statistics about the number of non-zero elements comparing

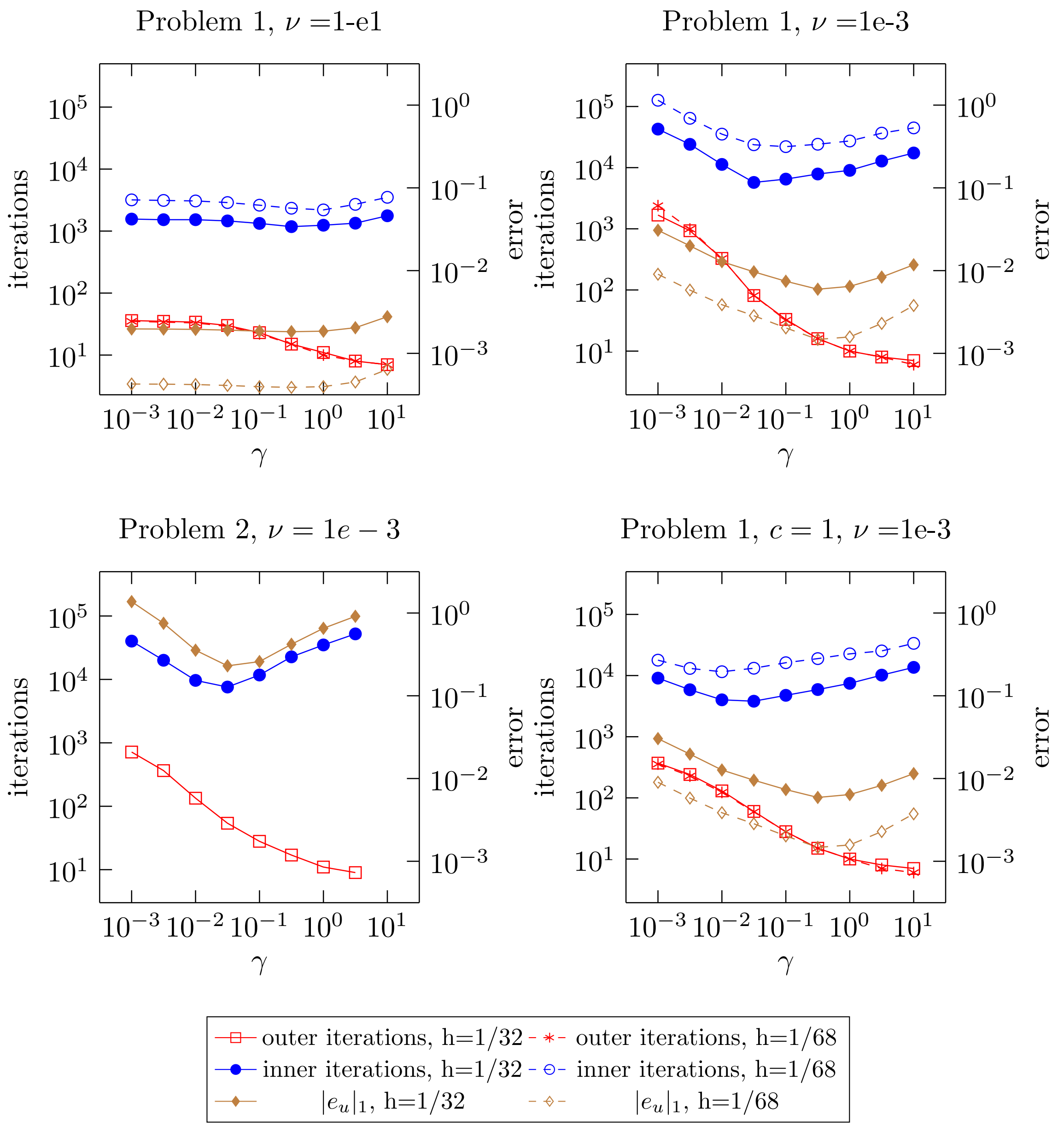


Figure 3.2: Analysis of solver performance and influence of the stabilization on the error with respect to the parameter choice γ. The outer iterations *are given by the number of required iterations for the preconditioned block system. The* inner iterations *show the total number of iterations for the A block summed up over all outer iterations. Note, that the number of inner iterations is heavily dependent on the preconditioner chosen for A (here, ILU(0)) and the result should only be considered quantitatively (Problem 1 and Problem 2 with $\boldsymbol{b} = \boldsymbol{u}$, unstructured mesh). The error is given in the H^1 semi-norm of the difference between the reference solution and calculated finite element solution.*

			γ=1.0			γ=0.31			γ=0.1		
	h:	ν=	1e-1	1e-3	1e-5	1e-1	1e-3	1e-5	1e-1	1e-3	1e-5
Q2Q1	1/16		13	13	13	19	19	20	28	38	38
	1/32		13	12	12	19	19	19	28	38	38
	1/64		13	12	12	18	19	19	27	37	37
Q3Q2	1/16		13	13	13	19	20	20	29	38	38
	1/32		13	12	13	19	19	19	27	37	38
	1/64		13	12	12	18	19	19	27	36	37
Q4Q3	1/16		13	13	13	19	20	20	28	37	38
	1/32		13	12	13	19	19	19	27	37	37
	1/64		13	12	13	18	19	19	27	36	36

Table 3.1: Number of outer FGMRES iterations for different problem sizes with different order of finite element spaces (Problem 1, regular mesh, stopping criterion: relative residual of 1e-10). The number of iterations is clearly independent of mesh size h and element order. Independence of the viscosity is achieved for $\gamma = 1$ and the optimal value $\gamma = 0.31$.

	$\#\{M_{ij} \neq 0\}$	$\#\{\lvert M_{ij}\rvert > 1e-15\}$
Galerkin	50884	15642
with Grad-Div	72250	27966
with augmentation	231704	47755

Table 3.2: Number of non-zero elements in the system matrix M (Problem 1, structured mesh, h=1/16, Q_2-Q_1, 2178+289=2467 unknowns). The second column represents the number of elements in the matrix with an absolute value bigger than 1e-15. The rows represent the base matrix without stabilization, the matrix with Grad-Div stabilization, and the matrix with the augmented Lagrangian term.

	$\gamma = 1$			$\gamma = 0.1$		
ν	Iter	Factor/s	Solve/s	Iter	Factor/s	Solve/s
1e-1	8	4.4	4.4	16	4.4	8.6
1e-2	7	4.5	3.8	23	4.4	11.9
1e-3	7	4.4	3.3	25	4.3	11.4
1e-5	7	4.5	3.5	26	4.4	12.9

Table 3.3: Timings for Problem 1 on an irregular mesh with 19280 cells (stopping criterion: relative residual of 1e-6, subproblems are solved with a direct solver). The number of iterations and the seconds to setup and solve the system are given for different γ and different viscosities.

the Galerkin system matrix, the system with added Grad-Div stabilization, and the augmented Lagrangian formulation. The augmentation decreases the sparsity of the system by a huge margin. Note that the Grad-Div stabilization only adds new entries, where the components of the velocity couple. If we consider for example a diffusion term given by the symmetric deformation tensor $\mathbb{D}$, Grad-Div stabilization would add no additional entries.

In Table 3.3 we measure the runtime[1] for Problem 1 on a irregular mesh with 19280 cells using $Q2/Q1$ elements. The table is set up to give comparable values to Table III in [16]. We use the direct solver UMFPACK for the inner problems. Although the mesh is 10% finer and not regular, our timings are very comparative compared to the augmented Lagrangian preconditioner. Of course, the setup time to assemble the matrices is independent of ν and γ.

So far we have only looked at the Oseen problem. In Table 3.4 we consider different prototypes of equations. This is done by modifying the coefficients in *Problem 1*. We compare the number of iterations with the same block preconditioner for $\gamma = 1$ and $\gamma = 0$. The last choice coincides with the traditional way of preconditioning for example the Stokes problem (see [123]). The choice $\gamma = 1$ improves the quality of the solution in all cases. Surprisingly the Grad-Div preconditioner also helps for the pure Stokes problem. This is most likely due to the fact that the approximation of the diffusion in the Schur complement is worse than the exact approximation for the Grad-Div term. Since the splitting of the Schur complement in diffusion and algebraic term is exact (see (3.11)), no additional error is introduced there. The new preconditioner shows its main advantage in the convection dominated case. The original preconditioner does barely work there. The reaction term helps especially in the reaction dominated case (which represents a time dependent problem with

---

[1] All problems were run on an Intel core 2 duo Laptop with 2.5ghz on one core. The code is neither optimized, nor implemented to take advantage of multiple cores.

	ν	c	$\lVert \boldsymbol{b} \rVert$	h:	1/8	1/16	1/32	1/64	1/128
Stokes	1e-3	0	0	$\gamma = 0$	17	17	18	18	17
				$\gamma = 1$	7	6	6	5	5
Stokes+reaction	1e-3	1	0	$\gamma = 0$	11	12	12	13	13
				$\gamma = 1$	7	7	6	6	6
Oseen	1e-3	0	> 0	$\gamma = 0$	90	638	1772	3645	-
				$\gamma = 1$	13	13	12	12	12
Oseen+reaction	1e-3	1	> 0	$\gamma = 0$	79	171	384	450	441
				$\gamma = 1$	13	12	12	12	11
Reaction dom.	1e-3	100	> 0	$\gamma = 0$	12	20	38	80	117
				$\gamma = 1$	10	10	9	9	9

Table 3.4: Number of iterations for different parameter choices and different mesh sizes (Problem 1, regular mesh). The Grad-Div preconditioner with $\gamma = 1$ is compared to the basic preconditioner ($\gamma = 0$) explained in Section 3.2.

small time step sizes). For smaller h the effect of the reaction term decreases.

Application to Driven Cavity Flow

Let us now consider a more involved application: the two dimensional lid-driven cavity flow. We consider Reynolds numbers up to $Re = 5000$, which result in stationary solutions. The non-linear system is solved on the unit-square with right-hand side $\boldsymbol{f} = 0$ and typical boundary conditions on $\partial\Omega$:

$$\boldsymbol{u}(x,y) = (u(x,y), v(x,y)) = \begin{cases} (1,0) & \text{for } y = 0, \text{ and } 0 < x < 1 \\ (0,0) & \text{else,} \end{cases}$$

i.e., we describe a force to the right at the top border. See [45] for a discussion for the problem with extensive numerical reference data. A plot of a few selected streamlines is given in Figure 3.3 (left).

We solve the non-linear Navier-Stokes problem with a damped fixed point iteration with a simple backtracking algorithm, see Section 1.5.4. All calculations are done on Cartesian meshes, which means the boundary layers are not resolved. In each iteration step one has to solve a Oseen type problem where the convection vector is given by the last iterate of the velocity.

An important quantity of interest is the minimum of the stream function, see [45] for reference values. For meshes with $h = 1/128$ we could reproduce the results for Re=5000 without stabilization but the quality of the minimum of the stream function and the cuts of the velocity increases significantly with Grad-Div stabilization for

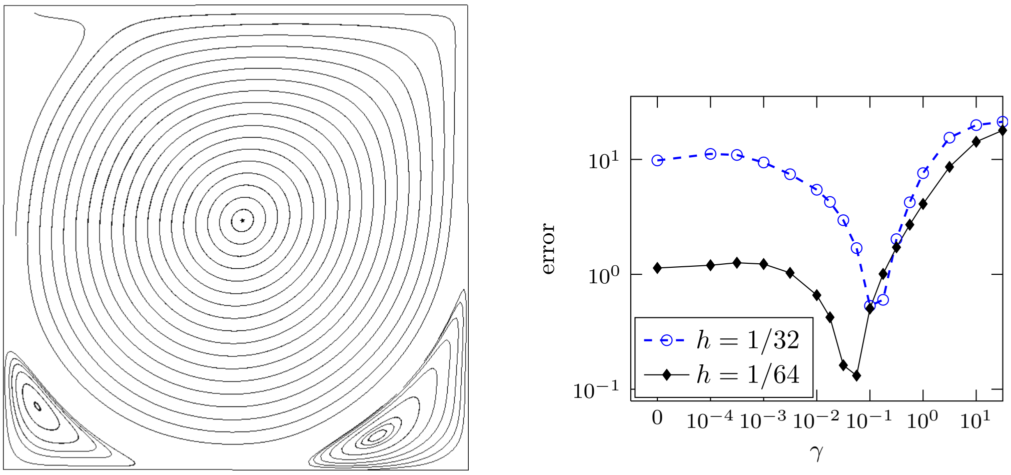


Figure 3.3: Lid driven cavity flow with Re=5000. Left: selected streamlines. Right: Error in minimum of the stream function (see [45]) dependent on the Grad-Div stabilization γ for two different meshes. Grad-Div stabilization decreases the error by more than one order of magnitude, which is better than a regular refinement.

coarser meshes. Figure 3.3, right shows the error of the minimum of the stream function in percent depending on the mesh size and Grad-Div parameter. The optimal γ is at 10^{-1} and is slightly mesh size dependent and tends to zero for finer meshes. The advantage of using Grad-Div stabilization for $h = 1/32$ is fairly obvious and one can gain one order of magnitude in the quality, which is more than a regular refinement. For smaller Reynolds numbers the importance of Grad-Div decreases. While it is still an half an order of magnitude for Re=1000 (optimal value at $\gamma = 0.1$), the effect vanishes for For Re=100. The velocity profiles also improve, which can be seen in Figure 3.4. We plot the second component of the velocity on a horizontal cut in the middle through the domain.

The non-linear iteration is done until the residual is smaller than 10^{-9} and each linear problem is solved with a relative residual of 10^{-2} with respect to the starting residual (which is the same as the non-linear residual). We compare the average number iterations for different choices of Grad-Div stabilization in Table 3.5. For comparison we also used the well-known pressure-convection-diffusion preconditioner (short: PCD) for the Schur complement, see [41]. The two methods are comparable for small Reynolds numbers, where the Grad-Div based preconditioner also works without Grad-Div stabilization. We again see the excellent scaling with the viscosity as in the earlier tests. This is in contrast to the PCD preconditioner, which fails for small viscosities. One can find similar behavior in [15] for example.

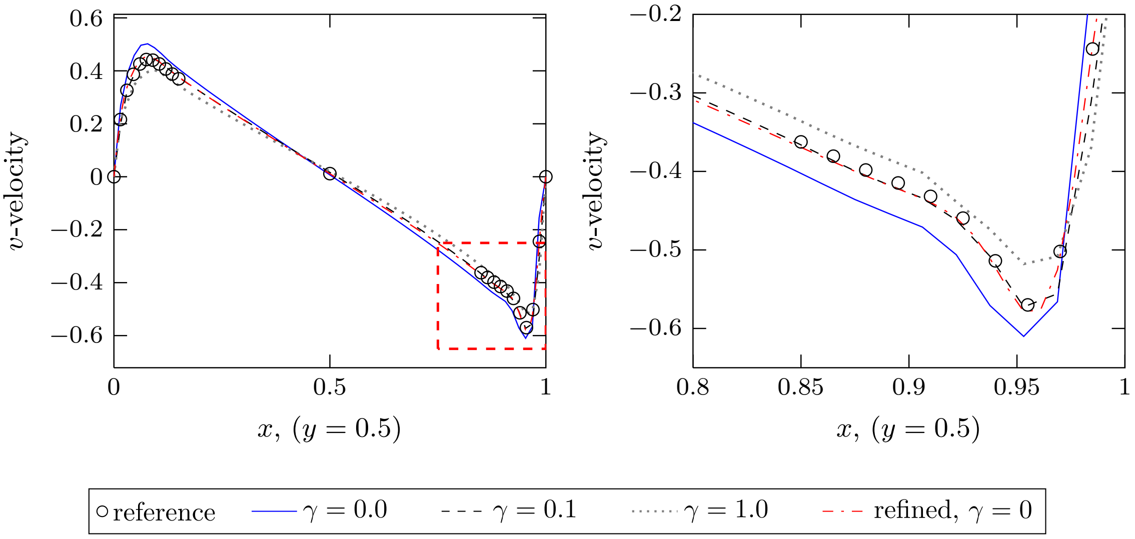


Figure 3.4: v-component of the velocity on a horizontal cut in the middle of the cavity with Re=5000 on a regular mesh with $h = 1/32$ and different values for Grad-Div stabilization in comparison with reference values from [45] and a comparison on a finer mesh with $h = 1/64$ and no Grad-Div stabilization. Right: zoom of left picture.

		$h = 1/32$			$h = 1/64$		
configuration		PCD	GD	#nonlinear	PCD	GD	#nonlinear
$\nu = 1e\text{-}2$	$\gamma = 0$	13	18	15	13	18	15
	$\gamma = \text{opt}$	17	**4**	15	16	**5**	15
$\nu = 1e\text{-}3$	$\gamma = 0$	44	342	34	42	511	29
	$\gamma = \text{opt}$	91	**6**	31	109	**8**	29
$\nu = 2e\text{-}4$	$\gamma = 0$	4822	-	104	1031	-	49
	$\gamma = \text{opt}$	1064	**7**	40	1249	**8**	43

Table 3.5: Number of non-linear iterations and average number of linear iterations per non-linear step for the PCD and the Grad-Div preconditioner in comparison for the cavity with Re $= 100$, Re $= 1000$ and Re $= 5000$ on a regular mesh. The optimal γ from the error point of view is selected.

3.4.4 Review of the Grad-Div Preconditioner

We now discuss the advantages and disadvantages of the Grad-Div preconditioner.

Let us start with the advantages: The number of iteration steps in the solver is independent of h and ν and gives small iteration counts comparable to the augmented Lagrangian approach, see the results in Section 3.4.3. This is an excellent behavior compared to other preconditioners.

The usage of Grad-Div stabilization improves the accuracy of the discretization scheme, cf. [97]. The preconditioner has a wide range of applicability. It can be used for Stokes, Oseen or Navier-Stokes problems, in transient and stationary cases and in different ranges of viscosity. See Table 3.4 for details.

Additionally, the Grad-Div preconditioner is easy to apply. Assembling the linear system is straight-forward. There are no complicated, additional matrices to be assembled for the preconditioner. In contrast, the matrix A_γ in the augmented Lagrangian is much harder to handle, since it is not immediately available as a matrix. One can either implement it as an operator, which restricts the choice of the preconditioner or one has to simplify the mass matrix M_p in the term $B^T M_p^{-1} B$ by lumping for example. Multiplying two sparse matrices is an expensive process and the resulting matrix is more dense than the Grad-Div preconditioned matrix, which only contains additional couplings between the velocity components. In many finite element packages the integration of augmented Lagrangian-type preconditioners is not simple. Often the degrees of freedom on the boundary are treated in the same way as inner degrees of freedom and they get eliminated before or after writing them to the global matrix. This can result in either loosing the symmetry in the B, B^T blocks or in non-zero entries in the second block of the right-hand side. Implementing the Grad-Div preconditioner does not pose any of these difficulties.

Our proposed approach also possesses some disadvantages. The Grad-Div stabilization is required for the Grad-Div preconditioner even when stabilization is not necessary. This adds additional coupling to the A block and slows down the assembly process. Fortunately, most of the real life problems need additional stabilization, see [97]. If the traditional Laplacian in the Navier-Stokes equations is replaced by symmetrized deformation tensors $\mathbb{D}$, as it is regularly done in turbulent flow simulations, the Grad-Div stabilization does not produce additional non-zero entries in the A-block.

Additional non-symmetric stabilization like SUPG spoils the performance of the preconditioner due to the fact that SUPG modifies B^T, which results in $B^T M_p^{-1} B$ not being symmetric any longer. This also applies to the augmented Lagrangian preconditioner and many other preconditioners. Here, we recommend symmetry preserving stabilization techniques like local projection stabilization, which work without problems with the Grad-Div preconditioner. An equal order discretization necessarily introduces a pressure stabilization term that changes the structure of the Schur complement. The Grad-Div preconditioner can then no longer be applied.

Choosing the stabilization coefficient can not be solely done from the stabilization point of view. The preconditioner performance has to be taken into account, too. Different choices of γ influence the quality of the solution. As for most of the preconditioning techniques preconditioning and stabilization can not be treated independently.

A general problem of the proposed class of preconditioners is the assembling of new matrices, which are not part of the primal problem. For the approximate Schur complement $\tilde{S}$, defined in (3.8), one has to assemble and store the matrices M_u and L_p.

Summarized, we think that the advantages clearly outbalance the disadvantages.

3.4.5 Possible extensions

We presented a new preconditioner that shows to be competitive to state of the art solution strategies. It is especially useful in the case where Grad-Div stabilization is already employed. It is helpful to use Grad-Div stabilization in all kind of problems with different parameters and the preconditioner helps in any of those cases.

The preconditioner gives h, ν, and element order independent iteration numbers, as long as the Grad-Div parameter is in a sensible range. It is possible to satisfy good accuracy and fast performance of the solver, because the parameter is not crucially sensitive and is in the same order of magnitude. Especially the ν independence can not be found in the typical preconditioners used today.

Some things will be very interesting to look at but do not fit into the scope of this thesis. Adapting the preconditioner to variable viscosity and varying Grad-Div parameter γ should be possible. It simply results in modifications in the Schur complement approximation. The diffusive part no longer reads

$$-(\nu+\gamma)M_p^{-1},$$

because $\nu+\gamma$ is no longer a constant. Small variations in ν (for example with turbulence models) do not make a difference for the preconditioner, because ν is still quite small compared to the reaction term for example. Choosing an average for $\nu+\gamma$ is a practical solution. When dealing with large jumps from one cell to another one has to include it this in the Schur complement. In [125] the different ways of how to do that are explained. The best was is to move the coefficients into the assembly of a modified mass matrix:

$$(M_p^{\nu,\gamma})_{ij} = \int_\Omega (\nu(x) + \gamma(x))^{-1} \phi_i \phi_j \, \mathrm{d}\Omega.$$

Alternatively one can scale the mass matrix with an approximation for each degree of freedom.

Another interesting aspect is to combine the augmented Lagrangian approach with the Grad-Div stabilization. This is straight-forward and could help in the case where the Grad-Div parameter has to be chosen very small for accuracy reasons. Let γ be the Grad-Div parameter. One can then additionally augment the system with $\gamma' B^T M_p^{-1} B$. The diffusive part of the Schur complement is then chosen as

$$-(\nu + \gamma + \gamma') M_p^{-1},$$

and one can change γ', so that $\gamma + \gamma'$ is optimal for preconditioning.

4 Applications and Numerical Results

This chapter focuses on combining the separate features developed in the earlier parts and validating the developments with realistic applications. This gives insight on how the results of this thesis can be used in practice. The application to the research in the work group in Göttingen serves as an example on how others can benefit from this work.

The aspect of the massive parallel finite element framework from Chapter 2 and the discussion of solvers for flow problems (Chapter 3) were discussed separately so far. They are unified and discussed in Section 4.1.

We continue with a discussion about turbulence benchmarks that are done for research in our work group about variational multi-scale (VMS) methods, see Section 4.2. Parallel solvers were required for the success of the experiments, even though we did not make use of the adaptive mesh refinement, that is one of the central pieces in Chapter 2.

Finally, in Section 4.3 we go into some details of the mantle convection example we started to discuss in Section 2.3.5 before. Here we take advantage of the fully adaptive parallel meshing from Chapter 2. The experiments started as a more complex test case for the parallel adaptive code, but now have a right to exist on its own. The research is continued as part of the Geodynamics AMR Suite of the Computational Infrastructure for Geodynamics (CIG) initiative supported by the NSF, see [34].

The numerical results in this Chapter are done on up to moderate sized clusters only. This is partly due to the nature of the applications that make going to larger problem sizes difficult or unnecessary (turbulence modelling with resolved scales does not make much sense). Additionally, the applications here are still work in progress because the work had to be done after the parallelization effort. The fully three dimensional structure makes linear solvers much more expensive and harder to tune for large problem sizes. Finally, the access to larger computing resources was limited. The massively parallel scalability can be seen in the benchmarks at the end of Chapter 2.

The section about turbulence (Section 4.2) contains recent research from our work group, also see [104,105]. Section 4.3 covers material, which is still work in progress. Parts of it are based on our publication [6] and more will be published in the near future, see [80].

4.1 Parallelization of solvers for flow problems

When solving for flow problems with the solvers presented in Chapter 3 one quickly realizes, that flow problems require a large number of cells and unknowns to be represented accurately. Naturally, the question arises on how to combine parallelization (Chapter 2) with the linear solvers.

Obviously, the parallel framework is flexible enough to cope with all kind of problems, so it can be used for flow problems in particular. The only thing that is left is parallelization of the linear solvers from Chapter 3:

Krylov methods for the outer and inner iterations can be parallelized as explained in Section 2.2.3. The application of a block triangular preconditioner works the same in a distributed parallel model, if you use collective operations as described in Section 2.2.2 for multiplying with B, etc.. Individual blocks like the velocity block or the Schur complement require preconditioners that run in parallel. We can access parallel block ILU or AMG preconditioners as described in Section 2.2.3.

One important thing one needs to pay attention to, is the construction of special matrices for the preconditioner. Matrix products like $B^T B$ can not be built in a practical way, because they are split row-wise between machines. Fortunately, the Grad-Div preconditioner does not require matrices beside the Schur complement, which is a simple Poisson-type problem that can be assembled in parallel. Adding Grad-Div stabilization is no problem, because it is just an additional term in the variational formulation.

Further aspects like postprocessing and the required additional models like wall laws for wall bounded flows, pose further difficulties in the parallel setting. The channel flow example (see Section 4.2.4) requires looking up velocities at the wall nearest degree of freedom, even when this part of the mesh is not locally available.

The outer Krylov method for the saddle point problem (with FGMRES for example, see Chapter 3) in parallel is equivalent to the serial method. The expectation is therefore to get outer iteration numbers that are independent of the number of processors involved, assuming the inner solves are accurate enough. The numerical tests confirm this argument and the iteration numbers are comparable to the test problems in Section 3.4.3 and also independent of the problem size. This can be seen in the remainder of this chapter.

4.2 Turbulence modeling

Turbulent incompressible flows occur in many applications in the industry. A turbulent flow can be characterized by a locally chaotic behavior due to low diffusion and high momentum convection. Flow can become turbulent with high Reynolds numbers. Typically for turbulent regions in a flow are the formation of eddies at different length scales. The small chaotic eddies carry a non-negligible amount of the total energy in the system. Therefore, they can not be omitted in simulations. On the other hand, resolving all ranges of scales in a turbulent simulation is typically not feasible due to computational cost – even with massively parallel supercomputers. A remedy is to resolve the large scales, but to model the influence of the small eddies on the global properties of the flow field. This is a reasonable approach because having access to the behavior of the smallest scales is not important in a simulation. Usually only the global flow pattern is of interest.

As explained before, we make heavy use of parallelization and the solver framework from Chapter 2 and 3.

4.2.1 Large Eddy Simulation and the variational multi-scale method

The scale separation ansatz – splitting up the flow into a resolved and an unresolved part, which is also known as classical Large Eddy Simulation or short LES – makes it possible to simulate turbulent flows. Nevertheless, the computational cost can be quite high, and parallelization is often used in research and commercial turbulent flow simulations. Thus, it is a good application to test the parallelization and the solver framework presented in this thesis.

There are different ways to model the influence of the unresolved scales. A prominent way is the classical Smagorinsky model, see [113], or dynamical Smagorinsky models, see [84]. A good overview can be found in [71]. More recently, an alternative based on the variational multi-scale (VMS) approach has been proposed. Here, the scale separation is done into large scales, small resolved scales, and small unresolved scales, and the influence of the unresolved scales is restricted to affect the small resolved scales only. One of the first discussions of VMS methods is [68], a good overview is given in [18]. Some results from our work group about VMS methods can be found in [104–106].

The main equations for the turbulent flow simulations are the instationary, incompressible Navier-Stokes equations as introduced in Section 1.5. There are some crucial differences, as it is typical in applications. First, it is common to apply a different formulation of the diffusive term. The Laplacian is replaced by the divergence of the deformation tensor (or symmetric gradient) $\mathbb{D}(\boldsymbol{u}) := \frac{1}{2}(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T)$, i.e.,

$$\nabla \cdot (2\nu \mathbb{D}(\boldsymbol{u})) \qquad \text{or} \qquad (2\nu \mathbb{D}(\boldsymbol{u}), \mathbb{D}(\boldsymbol{v}))$$

in the strong and weak formulation, respectively. Note that we discussed this formulation in Section 3.4.4 already. It introduces Grad-Div stabilization in the order of ν to the system. Second, the turbulence model is implemented as an artificial, non-linear diffusion term, that depends on the discrete solution u_h itself. The amount of diffusion is typically written as $\nu_T(u_h) \geq 0$ and is chosen as constant per cell.

For the classical Smagorinsky the diffusion term is modeled as

$$\nu_T = C_s \delta_K^2 \|\mathbb{D}(u)\|_F, \tag{4.1}$$

where C_s is the Smagorinsky constant, δ_K is the filter width, and $\|\cdot\|_F$ the Frobenius norm. The diffusion term looks the same as the original diffusion term:

$$(\nu_T \mathbb{D}(u), \mathbb{D}(u)). \tag{4.2}$$

Therefore, one has a Navier-Stokes problem with varying viscosity $\nu + \nu_T$. If we assume ν_T to be constant per cell (which is not the case in our simulations), we would get Grad-Div stabilization in the order of $\nu + \nu_T$. Note that we don't go into further details regarding the design of the parameter ν_T. It can includes some damping function for wall-bounded flows called van Driest damping. This is of no importance for the solver design, though.

The VMS model is slightly different. See [105] for more details. The coarse scales can be described using a coarse and a fine mesh, a lower and a higher order finite element space, or a combination of both. Let $\mathcal{T}_H$ be the coarse grid with $H \geq h$. The fine space $\mathcal{T}_h$ is typically either a conforming refinement or the same space. For the coarse finite element space L_H of the deformation tensor holds

$$0 \subseteq L_H \subseteq \mathbb{D} V_h.$$

Using the orthogonal projector $\Pi_H : L \to L_H \subset L$, the fluctuation operator $\kappa = Id - \Pi_H$ can be used to separate the scales into coarse and fine scales. With this the diffusion is only acting on the coarse scales:

$$(\nu_T\ \kappa(\mathbb{D}u), \mathbb{D}u).$$

In [105] the model for ν_T is kept close to the classical Smagorinsky model (4.1), only that $\|\kappa\mathbb{D}(u)\|_F$ is used instead of $\|\mathbb{D}(u)\|_F$, i.e., it only sees the small scales. Additionally, Grad-Div stabilization is added as a subgrid model for the pressure, see Section 1.5.5.

4.2.2 Solvers

With the Navier-Stokes equations modified for turbulent calculations as discussed before, the question arises on how to adapt the solver framework to cope with these changes.

The full Smagorinsky model is adding an additional diffusion term, that varies in the domain. Even though the artificial viscosity is much larger than the original viscosity from the physical problem, it is still rather small (for example in the order up to 1e-3 for the channel flow). Therefore, it is not crucial to exactly represent the viscosity. The approach we took is to take an average constant viscosity that matches the problem. This can be done by evaluating the turbulent viscosity during assembly. For the VMS method the same applies. Even though the turbulent viscosity is only applied to the small scales, it is reasonable to approximate it as if it is a full diffusion.

The time step sizes are very small to accurately capture the turbulent behavior in time and to track the high velocities. This makes the problem easy to solve efficiently. As a matter of fact, we can not make use of Grad-Div stabilization, because the optimal parameter range is too small to be used in the preconditioner. Bigger Grad-Div parameters tend to degrade the quality of the solutions considerably. With this setting for the solver it is also reasonable to look at Chorin-type projection methods instead. But even with these big time steps the number of iterations for the iterative solvers are decreased by roughly 10 to 20 percent, when incorporating the averaged viscosity term in the Schur complement (instead of only using the reaction term, as projection methods would use).

4.2.3 Decaying Homogeneous Isotropic Turbulence

We present the simulation of “Decaying Homogeneous Isotropic Turbulence”, see [35], which is a widespread turbulence benchmark, see [105] for details and more numerical results. The results here are based on the work of Lars Röhe, [83,104,105] and the solver related parts are published in our publication [62].

The domain is given by a cube $[0,2\pi]^3$ with periodic boundary conditions. A

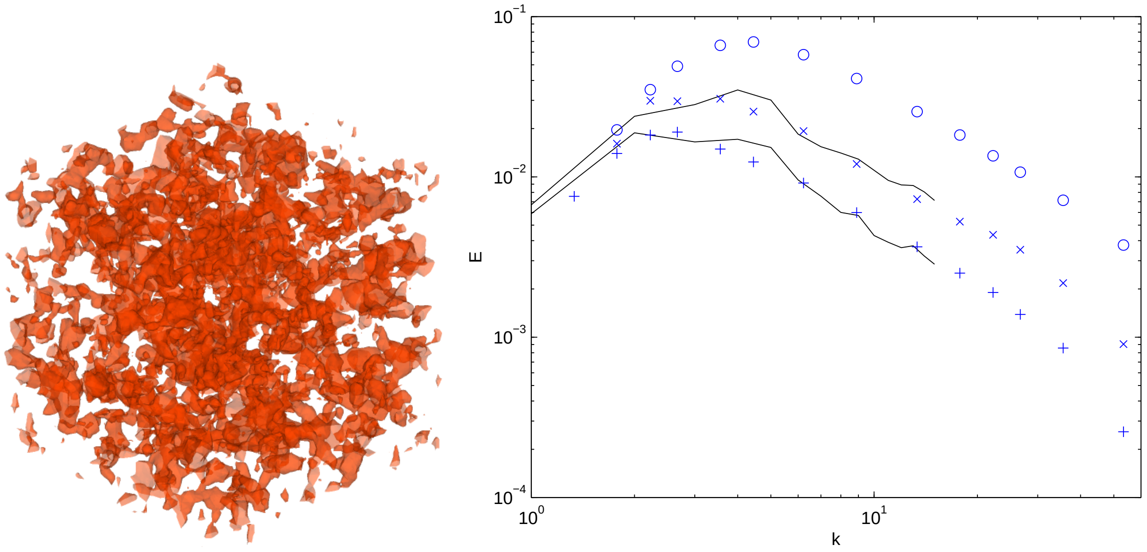
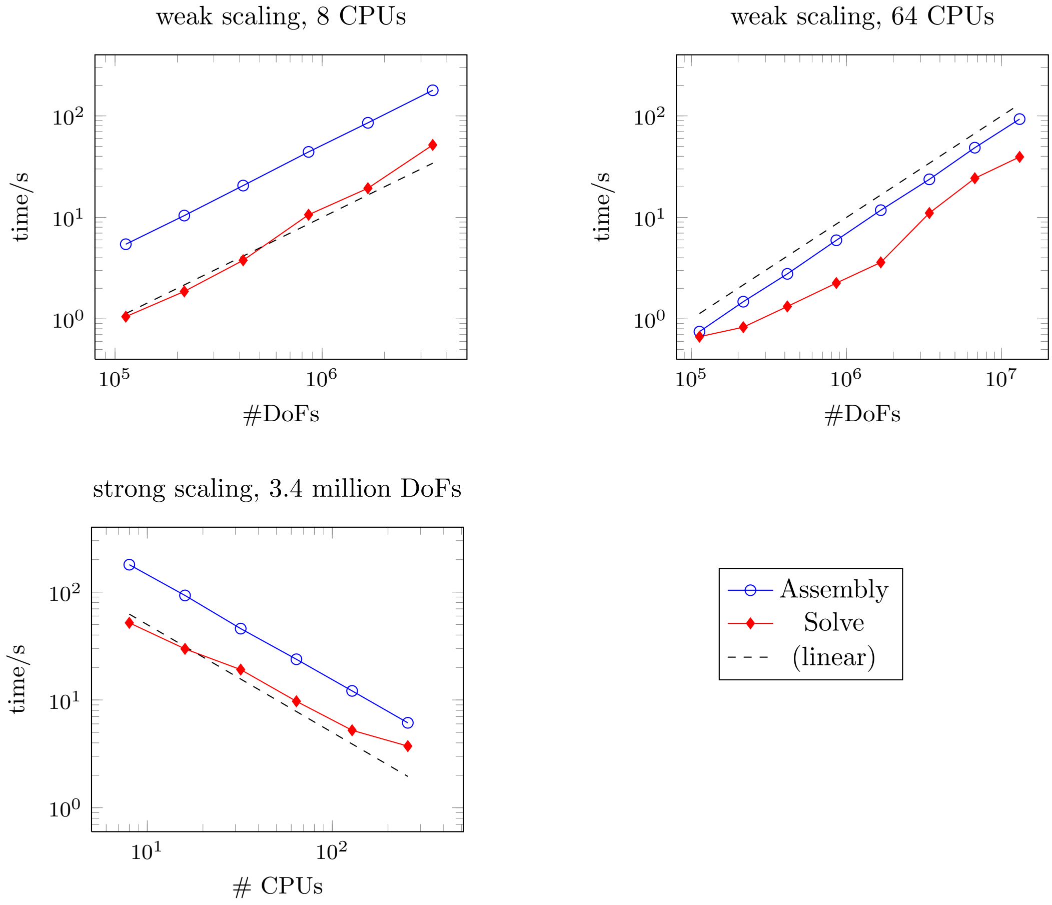


Figure 4.1: Left: iso-surface of initial velocity spectrum; right: energy spectra at $t = 0.87$ *and* $t = 2.00$ *(upper and lower line) and corresponding experimental data (symbols) with starting value.*

starting value (isotropic random velocity, see Figure 4.1) from a given energy spectrum (calculated via Fourier transform) is prescribed. The problem has a Taylor-scale Reynolds number of Re_λ =150 and the viscosity is $\nu \approx$ 1.5e-5 (air). As a turbulence model we choose a standard LES Smagorinsky model or the VMS approach (see (4.1),(4.2) above). The energy dissipation in time is compared to experimental data from [35], see Figure 4.1, right. The calculations were done with Q_2/Q_1 elements on meshes starting from 16^3 cellsHere the filter-width δ_h is given by h. This constant was not optimized but the results show good agreement to experimental data. For time discretization we apply a second order IMEX-scheme with a time step size of 0.0087. See [104] for more details about the setup. The outer FGMRES residual is chosen as 1e-7 to the starting residual, whereas the inner residuals are set to 1e-2 for the velocity and 1e-5 for the pressure (also relative). The algebraic multi-grid behaved sub-optimal for this problem, so block ILU is used to precondition the individual solvers.

There are several important numerical results. The number of outer FGMRES iterations is independent of the number of CPUs, because there is no difference to the serial algorithm. The number of iterations is independent of the mesh size and lies between 5 to 6 iterations, see Table 4.1. This proves that the preconditioner design works well and the accuracy of $\widetilde{A}^{-1}$ and $\widetilde{S}^{-1}$ is sufficient.

Figure 4.2 shows weak and strong scaling on a moderate number of processors. The results were done with the VMS turbulence model and the times shown are the


weak scaling, 8 CPUs

weak scaling, 64 CPUs

strong scaling, 3.4 million DoFs

Figure 4.2: Weak and strong scaling for assembly and solver for the decaying homogeneous isotropic turbulence with VMS.

1/h	# DoFs	# It.
8	2312	5
16	112724	5
32	859812	5
48	2855668	6
64	6714692	5

Table 4.1: Number of FGMRES iterations stays constant with respect to mesh size.

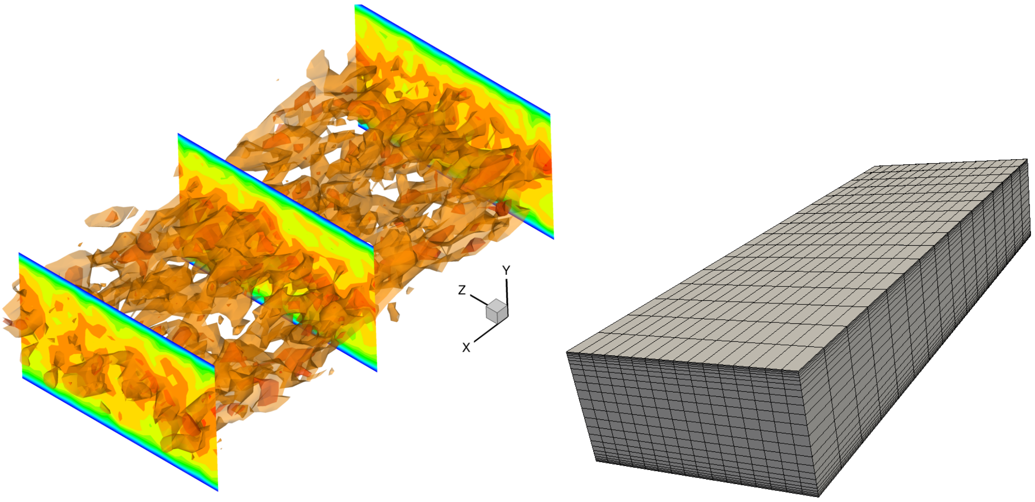


Figure 4.3: Turbulent channel flow with $Re = 180$. *Snapshot of a solution (left) and an example mesh highlighting the wall adapted cells (right).*

numbers of a fixed (early) time step. The time-step size has not been adjusted with changing mesh sizes. This is so the comparison is more fair, because the reaction based preconditioner performance depends on the time-step size. For numerical studies one has to further validate if the step size is still reasonable for the finer meshes. The scaling of the code is excellent in all cases, as can be seen by the slopes in the Figure.

4.2.4 Turbulent Channel Flow

A second common benchmark problem to calibrate turbulence models is the channel flow. The exact parameters for the setup are given in [104]. The flow is simulated in a box with no flow boundaries in one and periodic boundary conditions in the other two directions. To capture the boundary layer at the two walls, the mesh is refined anisotropically towards those walls. See Figure 4.3. Comparison of the velocity profile and the root mean square velocity to DNS data is given in Figure 4.4. More results can be foun in [20, 104].

To look at the scalability of the finite element framework and solver design we look at the time for assembling the system matrix and the solver times for a specific time step. Weak and strong scalability can be seen in Figure 4.5. Again this is with Reynolds number $Re = 180$ and the setup of the problem and the turbulence modeling is done by Lars Röhe, see [104] for more details.

From the results you can see very good scalability up to a very high number of degrees of freedom. For this benchmark, this will come close to a direct numerical

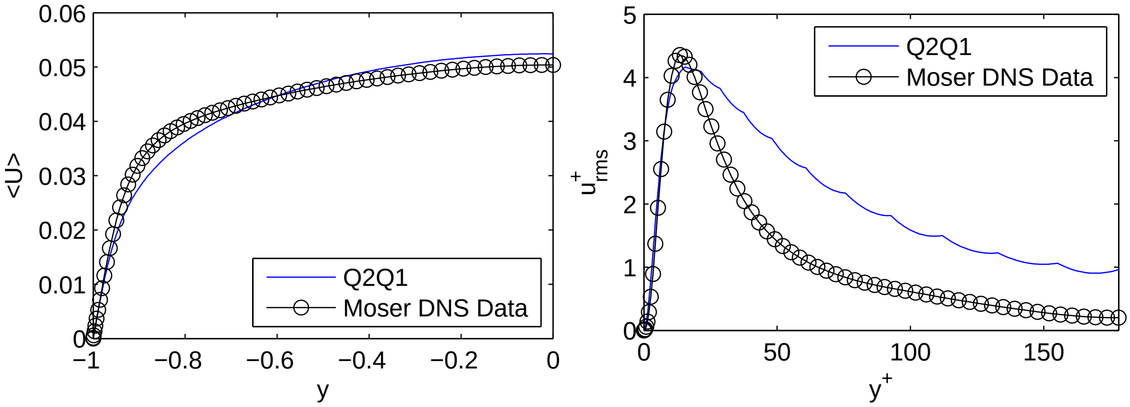


Figure 4.4: Velocity profile and root mean square velocity compared to reference values for channel flow with $Re = 180$.

simulation. Obviously, an implementation that takes advantage of the regular, structured mesh without adaptive refinement, might do better. The same properties are expected to be seen in a more general setting.

4.3 Convection in the earth's mantle

We now consider the equations that describe the convection in the earth's mantle. The movement is driven by buoyancy due to temperature variations. This is the step-32 example program in `deal.II`, see [79]. The numerical results in Section 2.3.5 were done with this program. We discuss this application in some further detail here. As said before, this is still work in progress (see [80]) and a very simplified set of equations. The following description is part of the publication [6].

The problem is modeled with the unknowns velocity, pressure, and temperature $\mathbf{u}, p, T$ using the Boussinesq approximation [93,110]. The equations read

$$\begin{aligned} -\nabla \cdot (2\eta \mathbb{D}(\mathbf{u})) + \nabla p &= -\rho\beta T\mathbf{g}, \\ \nabla \cdot \mathbf{u} &= 0, \\ \frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T - \nabla \cdot \kappa \nabla T &= \gamma. \end{aligned}$$

Here, $\mathbb{D}(\mathbf{u})$ is again the symmetric gradient of the velocity, η and κ denote the viscosity and diffusivity coefficients, respectively, which we assume to be constant in space, ρ is the density of the fluid, β is the thermal expansion coefficient, γ represents internal heat sources, and $\mathbf{g}$ is the gravity vector, which may be spatially variable. These equations are posed on a spherical shell mimicking the earth's mantle, i.e., the region above the liquid iron outer core and below the solid crust.

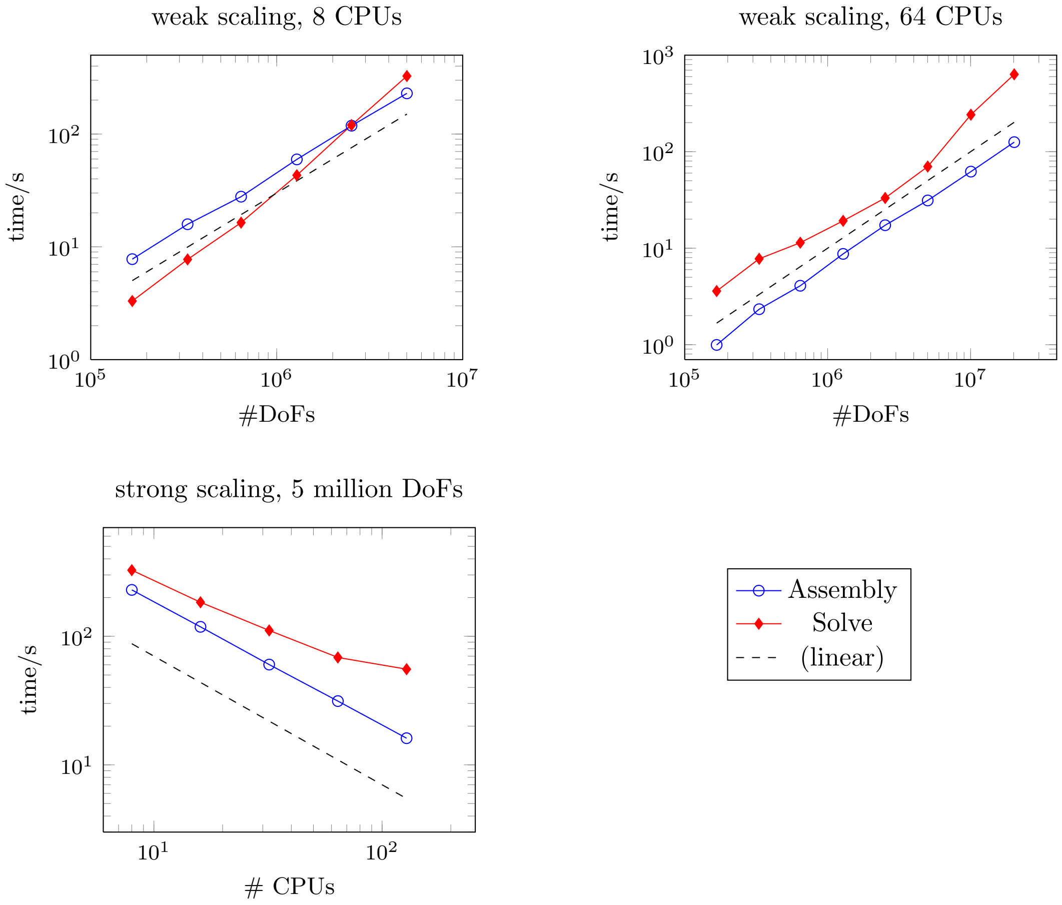


Figure 4.5: Weak and strong scaling for assembly and solver for the turbulent channel flow.

Dimensions of the domain, boundary and initial conditions, and values for the physical constants mentioned above can be found in the description of the step-32 tutorial program that implements this test case, see [79], or in [80]. A typical solution at a specific time step during the simulation is shown in Fig. 4.6. Due to the application we can simplify the Navier-Stokes equations to the Stokes equations. The velocity changes are assumed to be instantaneous, therefore, only the new temperature equation contains a time derivative.

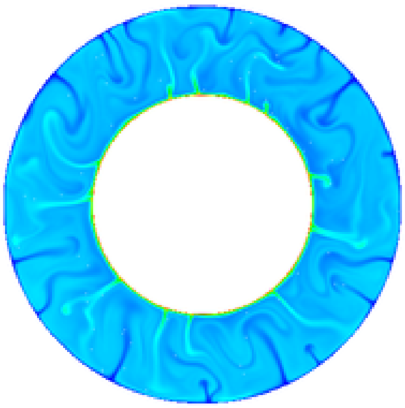


Figure 4.6: Solution of the mantle convection test case at a specific time step during the simulation in 2d. Mesh adaptation ensures that the plumes are adequately resolved.

We spatially discretize this system using $Q_2^d \times Q_1 \times Q_2$ elements for velocity, pressure and temperature elements, respectively. To stabilize the advection equation for the temperature a nonlinear artificial viscosity scheme is added, see [56] for details.

As in the other applications, the discretization for the Stokes system is done using an inf-sup stable Taylor-Hood element (also see Section 1.5.3). We solve the resulting system in time step n by first solving the Stokes part,

$$\begin{pmatrix} A_U & B \\ B^T & 0 \end{pmatrix} \begin{pmatrix} U^n \\ P^n \end{pmatrix} = \begin{pmatrix} F_U^n \\ F_P^n \end{pmatrix}, \tag{4.3}$$

and then using an explicit BDF-2 time stepping scheme to obtain the discretized temperature equation at time step n:

$$(M + \alpha^n A_T)T^n = F_T^n. \tag{4.4}$$

Here, F_U^n, F_P^n, F_T^n are right-hand side vectors that depend on previously computed solutions. α^n is a coefficient that depends on the time step length. A_T is a matrix that results from natural and artificial diffusion of the temperature and M is the mass matrix on the temperature space.

The Stokes system (4.3) is solved using a simplified preconditioner as it is explained in Chapter 3. The Schur complement only consists of the scaled pressure mass matrix, here. The velocity block is solved using the algebraic multi-grid ML from Trilinos and a BiCGStab iteration. The Schur complement is solved using an ILU decomposition of this matrix as a preconditioner. This scheme resembles the one also chosen in [50].

The temperature system (4.4) is solved using the CG method, preconditioned by an incomplete Cholesky (IC) decomposition of the temperature system matrix. Note that the ILU and IC preconditioners are implemented in block Jacobi fashion across the range of different processors, i.e., all coupling between different processors is neglected.

As expected for simulations of reasonably realistic physics, the resulting scheme is heavily dominated by the linear solver, which has to be invoked in every time step whereas the mesh and DoF handling algorithms are only called every tenth time step for example when the mesh is changed. On the other hand, the highly unstructured mesh and the much larger number of couplings between degrees of freedom for this vector-valued problem impose additional stress on many parts of the implementation.

The results for the two dimensional problem are already given in Section 2.3.5. In Figure 4.7 (right) the adaptive mesh of an early time step of the 2d problem is shown. The refinement is done using a standard Kelly error estimator that evaluates the gradient jumps in the temperature field. The symmetry of the plumes in the figure is because the initial temperature field is symmetrically distorted.

Finally, Figure 4.8 presents some early results of the 3d simulations. See Figure 4.7 for a solution snapshot. The calculations were done on a medium sized cluster and not up to 16000 CPUs as it was done in Section 2.3.5. Because of the increased coupling and complexity of the geometry and the flow, a lot more memory is required per degree of freedom. This limits the problem size that can be placed on a single machine, which in return makes it hard to achieve perfect parallel

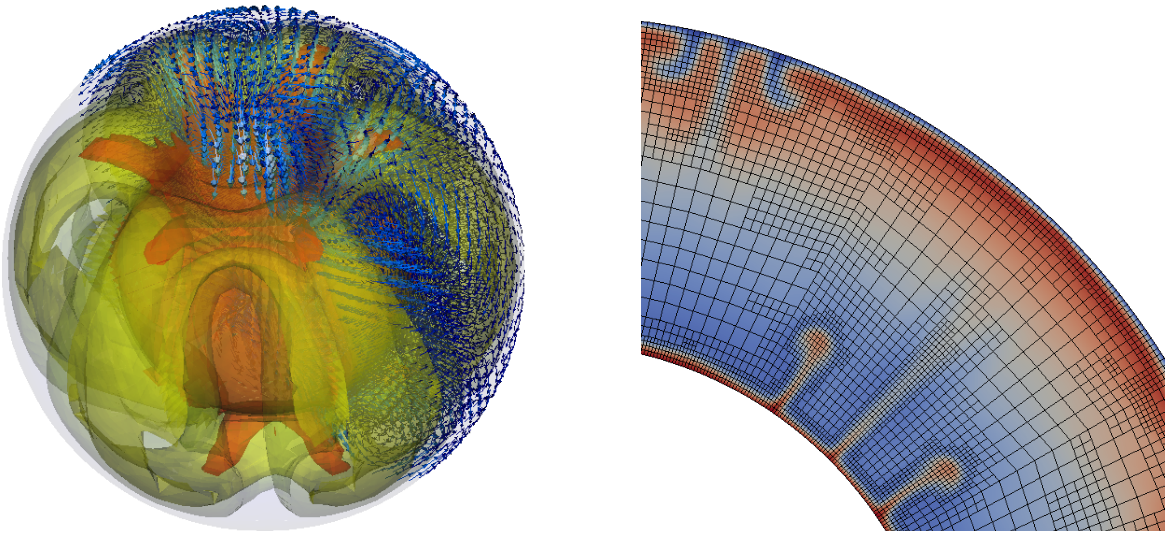


Figure 4.7: Left: Snapshot of an early time step for the 3d simulation. Right: Mesh and temperature field of an early time step of the 2d mantle convection. The adaptivity clearly follows the features of the solution to resolve the detailed features.

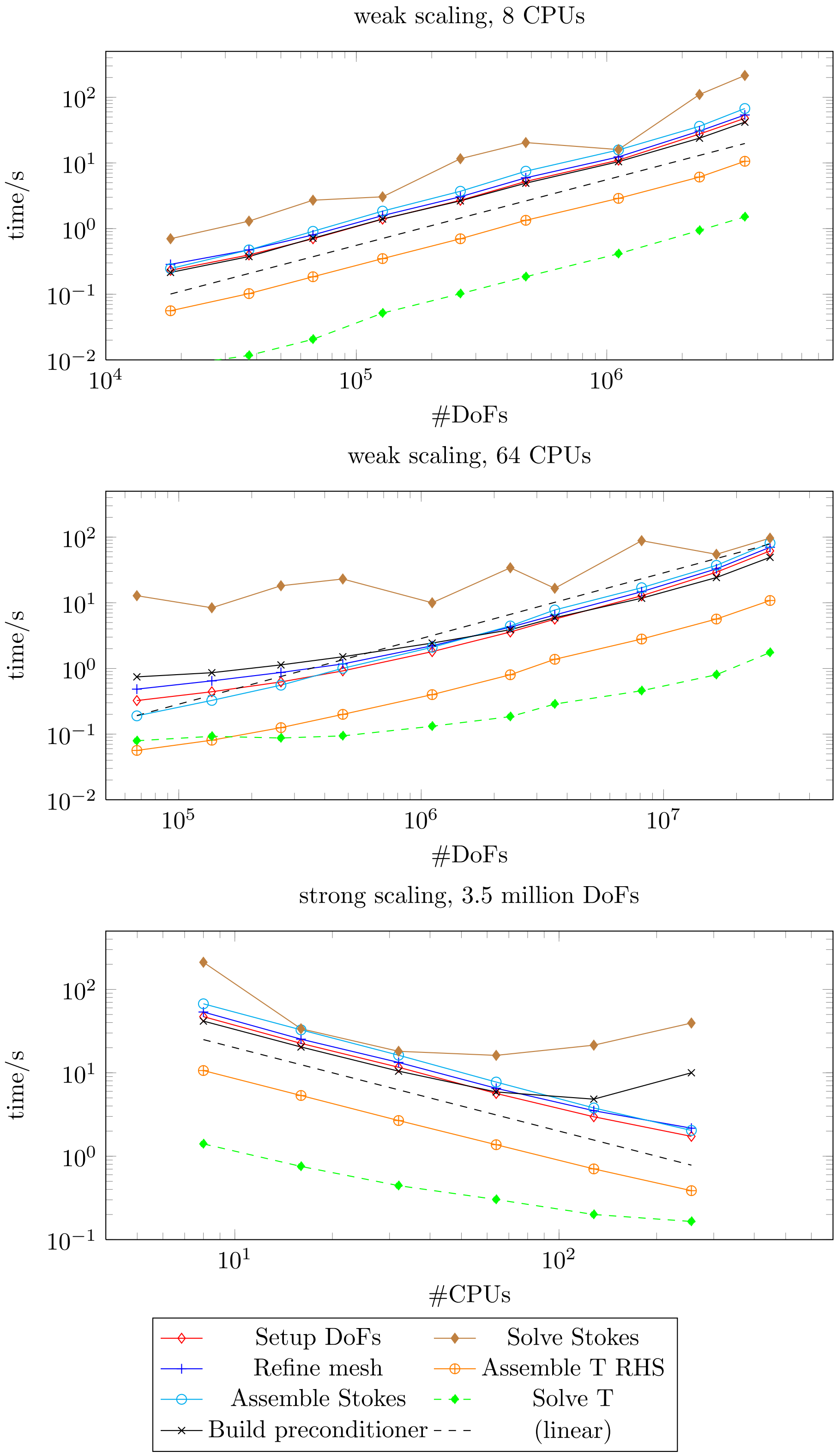


Figure 4.8: Weak and strong scaling of important parts of the 3d mantle convection simulation. These measurements are taken for a specific time step.

performance. Nevertheless, the weak scalability is comparable to the 2d case. The strong scaling of the solver on the other hand is a bit worse than in 2d. This shows that some further optimizations are necessary here. For this test the mesh smoothing of the adaptive refinement was disabled to achieve predictable number of unknowns. This might be the reason for the timing differences for the solution process. All in all, the results show that even in an early state, all algorithms in this complex simulation are scaling close to optimally.

5 Conclusions and Directions for Future Research

Conclusions

We introduced the necessary steps to design and implement a massively parallel fully adaptive finite element framework. In addition to the theory in this thesis, a large amount of time has been spent implementing and optimizing an implementation in the open source library `deal.II`. Iterating over the implementation enabled us to repeatedly find bottlenecks in the data structures and algorithms that stopped the applications to scale to large problem sizes. The library supplies researchers with an easy to use generic library that can be used for all kind of problems discretized with the finite element method. We showed good parallel scalability up to 16000 processor cores and the results promise good performance on even bigger machines. The ability to tackle very large systems with more than a billion of unknowns proof the good scalability of the algorithms and the distributed data structures. These features are already in use by other research groups throughout the world and enable them to use the available computational power without it being necessary to become an expert in parallelization themselves. Many ideas and even source code can be used and easily adapted to other finite element libraries. This would allow even more people to profit from this technology.

Further evidence for the usefulness and importance of this thesis can be seen in the following two projects: First, the research in my research group definitely profited from this advancement in `deal.II`. For example the numerical simulations for the research on turbulence models in [104] and the related publications [105,106] would not have been possible. The turbulent channel flow examples were first made possible by implementing the solvers from this thesis and parallelizing it to run on a workstation. With computing times up to a month this was still far from feasible, so with further optimizations the same test could be done in two days on a local cluster available to the group. Second, the mantle convection test code [79] with this amount of detail on a parallel machine is only possible because of the advancements in `deal.II` due to this thesis. The development of the mantle convection code is continued as part of the Geodynamics AMR Suite of the Computational Infrastructure for Geodynamics (CIG) initiative supported by the NSF, see [34].

The studies of Grad-Div stabilization as a stabilization method and turbulence model on the one hand and as a tool for efficient solvers from the other hand, supply interesting possibilities. Our research in [97] gives new insight to the role of Grad-Div stabilization and quantifies its dissipation. The novel solver approach from Chapter 3 gives a very competitive solution strategy for a wide range of problems, especially for stationary Navier-Stokes problems. Nevertheless, it appears to be useful even for the Stokes problem. Even though the solvers in the mantle convection example and the turbulent flow problems do not take advantage of Grad-Div stabilization explicitly, they fall into this solver framework and profit from this design.

The usefulness and importance of this thesis can already be seen in the applications shown in Chapter 4, what kind of research will be possible based on the material in this thesis is to be seen.

Future research

There are some ideas that did not fit into the constraints of this thesis but may be directions for further research.

While the parallelization part is more or less complete, there are some interesting enhancements possible. Right now, the coarse mesh is stored on every machine. This is no problem in the typical adaptive refined case, but can be a limitation for complex geometries. The geometry of an airplane typically already requires a large number of cells to accurately describe the features of the airplane. Instead of storing them on each node, one could split the mesh into several chunks and only load and maintain part of the geometry locally. It is not complicated to develop, but it requires distributing the storage beforehand in some way.

Another limitation in the massively parallel framework is the current limitation on the size of the linear system. Right now, one can only use 2^{31} (roughly 2 billion) degrees of freedom in total. This is the largest number that can be expressed in the standard 32 bit signed integer data type. This is not mainly a limitation from `deal.II`, because we can go to 64 bit integers easily, but it is a limitation of the other libraries involved: even though PETSc can be compiled with 64 bit indexing, many solvers like BoomerAMG can not be used anymore. Trilinos also only supports 32 bit sizes. As soon as there is support in these libraries, it makes sense to pursue that. With good scalability as long as one has in the order of a hundred thousand degrees of freedom per core, this currently limits how useful it is to go to more than twenty thousand cores.

The topic of hybrid parallelization (as discussed in Section 2.1.3) is going to become more important within the next years as the number of cores per machine increases and the amount of memory per core does not increase with the same rate. This means that parts must be adapted to take advantage of this new system. The major part involves multi-threading the algorithms for the mesh and degree of freedom management. It is likely, that a distributed memory architecture is going to stay, because applications require a lot of memory. Therefore, this thesis will not become obsolete.

The research about Grad-Div gives insight about the optimal Grad-Div parameter in theory. Unfortunately it is not feasible to use the full non-linear model. The constant parameter case is now understood fairly well and can be used in the solver framework. It would be interesting to compare the constant model to a locally adapted design. This requires some work on the preconditioner side, too.

The one disadvantage of the Grad-Div based preconditioner in comparison to the augmented Lagrangian preconditioner is, that you are not free to choose an arbitrary Grad-Div parameter. This is because a too big parameter can over-stabilize and thus spoil the solution. One way would be to combine Grad-Div and augmentation, i.e., add augmentation in the linear system (but only implicitly without forming the system matrix) and using Grad-Div stabilization explicitly in the preconditioner. With this technique one is free to choose the augmentation parameter and has an explicit matrix for efficient preconditioners.

The applications in this thesis are mostly at the beginning right now. With the now available computational power, it will be very interesting what one can do with the convection in the earth's mantle for example.

Thanks

I would like to thank:

- my advisor Gert Lube for his excellent support over the last years
- my co-reviewer Robert Schaback
- Wolfgang Bangerth and Guido Kanschat for the opportunity to visit Texas A&M
- my collaborators Wolfgang Bangerth, Carsten Burstedde, Martin Kronbichler, Gerd Rapin
- my work group (thanks, Lars!)
- Jean Marie, Jochen, my family and friends
- the administrators of the various computing facilities I worked with
- you, as a reader of my thesis
- and all the others I forgot to mention

Thank you.

– "Das tolle an Latex ist, dass man sich nicht um das Layout kümmern muss!"

I would also like to mention: [124].

Bibliography

[1] M. Ainsworth and J. T. Oden. *A Posteriori Error Estimation in Finite Element Analysis*. John Wiley and Sons, 2000.

[2] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[3] U. M. Ascher, S. J. Ruuth, and R. J. Spiteri. Implicit–explicit Runge–Kutta methods for time-dependent partial differential equations. *Appl. Numer. Math.: Transactions of IMACS*, 25(2–3):151–167, 1997.

[4] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.0.0, Argonne National Laboratory, 2008.

[5] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2009. `http://www.mcs.anl.gov/petsc`.

[6] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler. Algorithms and Data Structures for Massively Parallel Generic Finite Element Codes. *submitted*.

[7] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II — a General Purpose Object Oriented Finite Element Library. *ACM Trans. Math. Softw.*, 33(4):27, Aug. 2007.

[8] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

[9] W. Bangerth and G. Kanschat. `deal.II` *Differential Equations Analysis Library, Technical Reference*, 2011. `http://www.dealii.org`.

[10] W. Bangerth and O. Kayser-Herold. Data structures and requirements for *hp* finite element software. *ACM Trans. Math. Softw.*, 36(1):4/1–4/31, 2009.

[11] W. Bangerth and R. Rannacher. *Adaptive Finite Element Methods for Differential Equations*. Birkhäuser Verlag, 2003.

[12] R. E. Bank. *PLTMG: a software package for solving elliptic partial differential equations*. SIAM, Philadelphia, 1998. Users' guide 8.0.

[13] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[14] M. Benzi, G. H. Golub, and J. Liesen. Numerical Solution of Saddle Point Problems. *Acta Numerica*, 14:1–137, 2005.

[15] M. Benzi and M. A. Olshanskii. An Augmented Lagrangian-Based Approach to the Oseen Problem. *SIAM J. Sci. Comput*, 28:2095–2113, 2006.

[16] M. Benzi, M. A. Olshanskii, and Z. Wang. Modified augmented Lagrangian preconditioners for the incompressible Navier-Stokes equations. *Int. J. Numer. Methods Fluids*, 2010.

[17] M. Benzi and Z. Wang. Analysis of augmented lagrangian-based preconditioners for the steady incompressible navier-stokes equations. *submitted*, 2010.

[18] L. Berselli, T. Iliescu, and W. Layton. *Mathematics of large eddy simulation of turbulent flows*. Springer Verlag, 2006.

[19] M. Braack and E. Burman. Local Projection Stabilization for the Oseen Problem and its Interpretation as a Variational Multiscale Method. *SIAM J. Numer. Anal.*, 43(6):2544–2566, 2006.

[20] M. Braack, G. Lube, and L. Röhe. Divergence preserving interpolation on anisotropic quadrilateral meshes. *submitted to CALCOLO*, 2011.

[21] J. Bramble, J. Pasciak, and A. Vasiliev. Analysis of the inexact Uzawa algorithm for saddle point problems. *SIAM J. Numer. Anal.*, 34:1072–1092, 1997.

[22] F. Brezzi. On the Existence, Uniqueness and Approximation of Saddle-Point Problems arising from Lagrangian Multipliers. *R.A.I.R.O. Anal. Numer.*, 1974.

[23] A. M. Bruaset and H. P. Langtangen. A comprehensive set of tools for solving partial differential equations; DiffPack. In M. Dæhlen and A. Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics*, pages 61–90. Birkhäuser, Boston, 1997.

[24] A. Burri, A. Dedner, R. Klöfkorn, and M. Ohlberger. An efficient implementation of an adaptive and parallel grid in DUNE. In *Computational Science and High Performance Computing II. Proceedings of the 2nd Russian-German Advanced Research Workshop, Stuttgart, Germany, March 14 to 16, 2005*, pages 67–82. Springer, 2005.

[25] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. C. Wilcox. Extreme-scale AMR. In *SC '10: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE, 2010.

[26] C. Burstedde, O. Ghattas, M. Gurnis, E. Tan, T. Tu, G. Stadler, L. C. Wilcox, and S. Zhong. Scalable adaptive mantle convection simulation on petascale supercomputers. In *SC '08: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE, 2008.

[27] C. Burstedde, O. Ghattas, G. Stadler, T. Tu, and L. C. Wilcox. Towards adaptive mesh PDE simulations on petascale computers. In *Proceedings of Teragrid '08*, 2008.

[28] C. Burstedde, L. C. Wilcox, and O. Ghattas. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *Accepted for publication in SIAM J. Sci. Comput.*, 2010.

[29] F. Cappello and D. Etiemble. Mpi versus mpi+openmp on the ibm sp for the nas benchmarks. In *Supercomputing, ACM/IEEE 2000 Conference*, page 12, Nov. 2000.

[30] G. F. Carey. *Computational Grids: Generation, Adaptation and Solution Strategies*. Taylor & Francis, 1997.

[31] L. Carrington, D. Komatitsch, M. Laurenzano, M. M. Tikir, D. Michéa, N. L. Goff, A. Snavely, and J. Tromp. High-frequency simulations of global seismic wave propagation using SPECFEM3D GLOBE on 62K processors. In *SC '08: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE, 2008.

[32] K. K. Chand, L. F. Diachin, X. Li, C. Ollivier-Gooch, E. S. Seol, M. S. Shephard, T. Tautges, and H. Trease. Toward interoperable mesh, geometry and field components for PDE simulation development. *Engineering with Computers*, 24:165–182, 2008.

[33] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[34] Computational Infrastructure for Geodynamics. `http://www.geodynamics.org/`.

[35] G. Comte-Bellot and S. Corrsin. Simple eulerian time correlation of full- and narrow-band velocity signals in grid generated isotropic turbulence. *J. Fluid Mech.*, 48:273–337, 1971.

[36] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, Jan.-Mar. 1998.

[37] T. A. Davis. Algorithm 832: UMFPACK V4.3 - An Unsymmetric-Pattern Multifrontal Method. *ACM Trans. Math. Softw.*, 30(2):196–199, 2004.

[38] A. C. de Niet and F. W. Wubs. Two Preconditioners for Saddle Point Problems in Fluid Flows. *Int. J. Num. Meth. Fluids*, 2006.

[39] H. C. Elman. Preconditioning for the Steady-State Navier–Stokes Equations with Low Viscosity. *SIAM J. Sci. Comput.*, 20(4):1299–1316, 1999.

[40] H. C. Elman and D. Silvester. Fast Nonsymmetric Iterations and Preconditioning for Navier-Stokes Equations. Numerical Analysis Report No. 263, Manchester Centre for Computational Mathematics, Manchester, England, 1995. To appear in SIAM Journal of Scientific Computing.

[41] H. C. Elman, D. J. Silvester, and A. J. Wathen. Performance and Analysis of Saddle Point Preconditioners for the Discrete Steady-State Navier–Stokes Equations. *Numerische Mathematik*, 90(4):665–688, 2002.

[42] H. C. Elman, D. J. Silvester, and A. J. Wathen. *Finite Elements and Fast Iterative Solvers with Applications in Incompressible Fluid Dynamics*. Numerical Mathematics and Scientific Computation. Oxford University Press, Oxford, New York, 2005.

[43] J. Erhel. A parallel GMRES version for general sparse matrices. *Electronic Transactions on Numerical Analysis*, 3(12):160–176, 1995.

[44] A. Ern and J. Guermond. *Theory and practice of finite elements*. Springer Verlag, 2004.

[45] E. Erturk, T. Corke, and C. Gökçöl. Numerical solutions of 2-D steady incompressible driven cavity flow at high Reynolds numbers. *Int. J. Numer. Meth. Fl.*, 48(7):747–774, 2005.

[46] R. D. Falgout, J. E. Jones, and U. M. Yang. Pursuing scalability for hypre's conceptual interfaces. *ACM Trans. Math. Softw.*, 31:326–350, 2005.

[47] R. D. Falgout, J. E. Jones, and U. M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In T. J. Barth, M. Griebel, D. E. Keyes, R. M. Nieminen, D. Roose, T. Schlick, A. M. Bruaset, and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 267–294. Springer, 2006.

[48] M. Fortin and R. Glowinski. Augmented lagrangian methods: Applications to the numerical solution of boundary value problems. *Stud. Math. Appl*, 15, 1983.

[49] M. W. Gee, C. M. Siefert, J. J. Hu, R. S. Tuminaro, and M. G. Sala. ML 5.0 smoothed aggregation user's guide. Technical Report SAND2006-2649, Sandia National Laboratories, 2006.

[50] T. Geenen, M. ur Rehman, S. P. MacLachlan, G. Segal, C. Vuik, A. P. van den Berg, and W. Spakman. Scalable robust solvers for unstructured fe geodynamic modeling applications: Solving the Stokes equation for models with large localized viscosity contrasts. *Geoch. Geoph. Geosyst.*, 10:Q09002/1–12, 2009.

[51] T. Gelhard, G. Lube, M. A. Olshanskii, and J. Starcke. Stabilized Finite Element Schemes with LBB-Stable Elements for Incompressible Flows. *J. Comput. Appl. Math.*, 177(2):243–267, 2005.

[52] V. Girault and P.-A. Raviart. *Finite Element Methods for the Navier–Stokes Equations*. Springer–Verlag, New York, 1986.

[53] R. Glowinski and P. Le Tallec. *Augmented Lagrangian and operator-splitting methods in nonlinear mechanics*. Society for Industrial Mathematics, 1989.

[54] G. H. Golub and C. Greif. On Solving Block-Structured Indefinite Linear Systems. *SIAM J. Sci. Comput.*, 24(6):2076–2092, Nov. 2003.

[55] J.-L. Guermond, P. Minev, and J. Shen. An Overview of Projection methods for incompressible flows. *Comp. Meth. Appl. Mech. Engrg.*, 195:6011–6045, 2006.

[56] J. L. Guermond, R. Pasquetti, and B. Popov. Entropy viscosity for conservation equations. In J. C. F. Pereira and A. Sequeira, editors, *V European Conference on Computational Fluid Dynamics (Eccomas CFD 2010)*, 2010.

[57] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems*. Springer-Verlag, Berlin, 1991.

[58] T. Heister. Vorkonditionierungsstrategien für das stabilisierte Oseen-Problem. Master's thesis, Institut für Numerische und Angewandte Mathematik, Georg-August-Universität zu Göttingen, 2008.

[59] T. Heister, M. Kronbichler, and W. Bangerth. deal.II – step-40 tutorial program. `http://www.dealii.org/developer/doxygen/deal.II/step_40.html`.

[60] T. Heister, M. Kronbichler, and W. Bangerth. Generic Finite Element Programming for Massively Parallel Flow Simulations. *Eccomas 2010 Proceedings*, 2010.

[61] T. Heister, M. Kronbichler, and W. Bangerth. Massively Parallel Finite Element Programming. In R. Keller, E. Gabriel, M. Resch, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6305 of *Lecture Notes in Computer Science*, pages 122–131. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15646-5_13.

[62] T. Heister, G. Lube, and G. Rapin. On robust parallel preconditioning for incompressible flow problems. In *Numerical Mathematics and Advanced Applications, ENUMATH 2009*. Springer, Berlin, 2010.

[63] T. Heister and G. Rapin. Efficient augmented Lagrangian-type preconditioning for the Oseen problem using Grad-Div stabilization. *submitted*.

[64] J. L. Henning. Performance counters and development of spec cpu2006. *ACM SIGARCH Computer Architecture News*, 35:118–121, 2007.

[65] V. E. Henson and U. M. Yang. Boomeramg: a parallel algebraic multigrid solver and preconditioner. *Appl. Numer. Math.*, 41(1):155–177, April 2002.

[66] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31:397–423, 2005.

[67] M. A. Heroux et al. Trilinos Web page, 2009. `http://trilinos.sandia.gov`.

[68] T. Hughes, L. Mazzei, and K. Jansen. Large eddy simulation and the variational multiscale method. *Comp. Vis. Sci.*, 3(1):47–59, 2000.

[69] T. J. Hughes, L. P. Franca, and M. Balestra. A new finite element formulation for computational fluid dynamics: V. circumventing the babuska-brezzi condition: a stable petrov-galerkin formulation of the stokes problem accommodating equal-order interpolations. *Comp. Meth. Appl. Mech. Engng.*, 59(1):85

– 99, 1986.

[70] V. John. Higher order finite element methods and multigrid solvers in a benchmark problem for the 3D Navier–Stokes equations. *Int. J. Numer. Meth. Fl.*, 40(6):775–798, 2002.

[71] V. John. *Large Eddy Simulation of Turbulent Incompressible Flows: Analytical and Numerical Results for a Class of LES Models*, volume 34 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2004.

[72] C. Johnson and J. Saranen. Streamline diffusion methods for the incompressible Euler and Navier-Stokes equations. *Math. Comput.*, 47(175):1–18, 1986.

[73] G. Kanschat and B. Rivière. A strongly conservative finite element method for the coupling of Stokes and Darcy flow. *J. Comp. Phys.*, 229:5933–5943, 2010.

[74] G. Karypis and V. Kumar. METIS-Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0. 1995.

[75] D. Kay, D. Loghin, and A. Wathen. A Preconditioner for the Steady-State Navier–Stokes Equations. *SIAM J. Sci. Comput.*, 24(1):237–256, 2002.

[76] B. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers*, 22(3–4):237–254, 2006.

[77] P. Knabner and L. Angermann. *Numerik partieller Differentialgleichungen*. Springer–Verlag, Berlin, Heidelberg, New York, 2000.

[78] M. G. Knepley and D. A. Karpeev. Mesh algorithms for PDE with Sieve I: Mesh distribution. *Scientific Programming*, 17:215–230, 2009.

[79] M. Kronbichler, W. Bangerth, and T. Heister. deal.II – step-32 tutorial program. `http://www.dealii.org/developer/doxygen/deal.II/step_32.html`.

[80] M. Kronbichler, T. Heister, and W. Bangerth. High accuracy mantle convection simulation based on generic finite element programming. *In preparation.*

[81] A. N. Krylov. On the numerical solution of the equation by which in technical questions frequencies of small oscillations of material systems are determined. *(original text in russian), Izvestija AN SSSR (News of Academy of Sciences of the USSR), Otdel. Mat. i. estest. Nauk., 2*, 4:491–539, 1931.

[82] H. P. Langtangen. *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*. Texts in Computational Science and

Engineering. Springer Verlag, 2003.

[83] W. Layton, L. Röhe, and H. Tran. Explicitly uncoupled vms stabilization of fluid flow. *submitted to CMAME*, 2010.

[84] D. Lilly. A proposed modification of the Germano subgrid-scale closure method. *Physics of Fluids A: Fluid Dynamics*, 4:633, 1992.

[85] L. Little and Y. Saad. Block LU Preconditioners for Symmetric and Nonsymmetric Saddle Point Problems, 1999.

[86] A. Logg. Automating the finite element method. *Arch. Comput. Meth. Eng.*, 14(2):93–138, 2007.

[87] A. Logg. Efficient representation of computational meshes. *Int. J. Comput. Sci. Eng.*, 4(4):283–295, 2009.

[88] A. Logg and G. Wells. DOLFIN: Automated finite element computing. *ACM Trans. Math. Softw.*, 37(2):1–28, 2010.

[89] G. Lube and G. Rapin. Residual-Based Stabilized Higher-Order FEM for a Generalized Oseen Problem. *Math. Mod. Meths. Appl. Sci.*, 16:1–18, 2006.

[90] G. Lube, G. Rapin, and J. Löwe. Local Projection Stabilizations for Incompressible Flows: Equal-Order vs. Inf-Sup Stable Interpolation. *submitted*, 2007.

[91] G. Matthies, G. Lube, and L. Röhe. Some remarks on residual-based stabilisation of inf-sup stable discretisations of the generalised Oseen problem. *Computational Methods in Applied Mathematics*, 9(4):368–390, 2009.

[92] G. Matthies, P. Skrzypacz, and L. Tobiska. A unified convergence analysis for local projection stabilisations applied to the Oseen problem. *Math. Model. Numer. Anal.*, 41(4):713–742, 2007.

[93] D. P. McKenzie, J. M. Roberts, and N. O. Weiss. Convection in the Earth's mantle: Towards a numerical solution. *J. Fluid Mech.*, 62:465–538, 1974.

[94] Message Passing Interface Forum. MPI: A message-passing interface standard (version 2.2). Technical report, `http://www.mpi-forum.org/`, 2009.

[95] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM Ltd., 1966.

[96] C. Ollivier-Gooch, L. Diachin, M. S. Shephard, T. Tautges, J. Kraftcheck, V. Leung, X. Luo, and M. Miller. An interoperable, data-structure-neutral

component for mesh query and manipulation. *ACM Trans. Math. Softw.*, 37:29/1–28, 2010.

[97] M. Olshanskii, G. Lube, T. Heister, and J. Löwe. Grad-div stabilization and subgrid pressure models for the incompressible Navier-Stokes equations. *Comp. Meth. Appl. Mech. Engng.*, 198(49-52):3975 – 3988, 2009.

[98] P. Pacheco. *Parallel programming with MPI.* Morgan Kaufmann, 1997.

[99] B. Patzák and Z. Bittnar. Design of object oriented finite element code. *Adv. Eng. Software*, 32(10–11):759–767, 2001.

[100] W. Petersen and P. Arbenz. *Introduction to parallel computing*. Oxford University Press, USA, 2004.

[101] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[102] Y. Renard and J. Pommier. Getfem++. Technical report, INSA Toulouse, available from `http://www-gmm.insa-toulouse.fr/getfem/`, 2006.

[103] W. C. Rheinboldt and C. K. Mesztenyi. On a data structure for adaptive finite element mesh refinements. *ACM Trans. Math. Softw.*, 6:166–187, 1980.

[104] L. Röhe. *Turbulence Modeling by a Projection-Based Variational Multiscale Method for Incompressible Flow Problems*. PhD thesis, University of Göttingen, 2011.

[105] L. Röhe and G. Lube. Analysis of a variational multiscale method for Large-Eddy simulation and its application to homogeneous isotropic turbulence. *Comp. Meth. Appl. Mech. Engng.*, 199(37-40):2331 – 2342, 2010.

[106] L. Röhe and G. Lube. Large-eddy-simulation of wall-bounded turbulent flows – Layer-adapted meshes vs. weak Dirichlet boundary conditions. In *LNCS BAIL (to appear)*, 2010.

[107] H. Roos, M. Stynes, and L. Tobiska. *Robust numerical methods for singularly perturbed differential equations: convection-diffusion-reaction and flow problems*. Springer Verlag, 2008.

[108] Y. Saad. A Flexible Inner-Outer Preconditioned GMRES Algorithm. Technical Report 91-279, Minnesota Supercomputer Institute, University of Minnesota, 1991.

[109] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2003.

[110] G. Schubert, D. L. Turcotte, and P. Olson. *Mantle Convection in the Earth and*

Planets, Part 1. Cambridge, 2001.

[111] D. Silvester and A. Wathen. Fast Iterative Solution of Stabilised Stokes Systems Part I: Using Simple Diagonal Preconditioners. *SIAM J. Numer. Anal.*, 30(3):630–649, 1993.

[112] D. Silvester and A. Wathen. Fast Iterative Solution of Stabilised Stokes Systems Part II: Using General Block Preconditioners. *SIAM J. Numer. Anal.*, 31(5):1352–1367, 1994.

[113] J. Smagorinsky. General circulation experiments with the primitive equations. *Monthly weather review*, 91(3):99–164, 1963.

[114] P. Šolín, J. Červený, and I. Doležel. Arbitrary-level hanging nodes and automatic adaptivity in the hp-FEM. *Math. Comput. Sim.*, 77:117–132, 2008.

[115] P. Šolín, K. Segeth, and I. Doležel. *Higher-Order Finite Element Methods*. Chapman & Hall/CRC, 2003.

[116] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 1997.

[117] H. Sundar, R. Sampath, and G. Biros. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM J. Sci. Comput.*, 30(5):2675–2708, 2008.

[118] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[119] E. Tan, M. Gurnis, L. Armendariz, L. Strand, and S. Kientz. Citcoms user manual version 3.0.1, 2008.

[120] R. Temam. *Navier-Stokes equations and nonlinear functional analysis*. Society for Industrial Mathematics, 1995.

[121] A. Tikhonova, G. Tanase, O. Tkachyshyn, N. M. Amato, and L. Rauchwerger. Parallel algorithms in STAPL: Sorting and the selection problem. Technical Report TR05-005, Parasol Lab, Department of Computer Science, Texas A&M University, 2005.

[122] T. Tu, D. R. O'Hallaron, and O. Ghattas. Scalable parallel octree meshing for terascale applications. In *SC '05: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE, 2005.

[123] S. Turek. *Efficient Solvers for Incompressible Flow Problems: An Algorithmic and*

Computational Approach. Springer, Berlin, 1999.

[124] D. Upper. The unsuccessful self-treatment of a case of "writer's block". *J. Appl. Behav. Anal.*, 7(3):497, 1974.

[125] M. ur Rehman, T. Geenen, C. Vuik, G. Segal, and S. MacLachlan. On iterative methods for the incompressible Stokes problem. *Int. J. Numer. Meth. Fl.*, 2010.

[126] R. Verfürth. A posteriori error estimation and adaptive mesh-refinement techniques. *J. Comput. Appl. Math.*, 50:67–83, 1994.

[127] W. Zulehner. Analysis of Iterative Methods for Saddle Point Problems: a Unified Approach. *Math. Comput.*, 71(238):479–505, 2002.

Curriculum Vitae

Timo Heister

University of Göttingen
Institute for Numerical and Applied Mathematics
Lotzestr. 16-18, D-37083 Göttingen, Germany

Phone:	+49 551-39 5022
Email:	`heister@math.uni-goettingen.de`
Homepage:	`http://num.math.uni-goettingen.de/~heister/`
Born:	December 30th, 1983 in Hannover, Germany
Nationality:	German

1 Education

2008 - now	Ph.D. student at the University of Göttingen
2003 - 2008	Study of mathematics at the University of Göttingen, Diploma thesis about preconditioning for the stabilized Oseen problem - “very good”
1996 - 2002	Gymnasium Burgdorf, Abitur - “very good”
1990 - 1996	Primary School and Middle School, Burgdorf near Hannover

2 Professional Experience

2008 - now	research and teaching assistant, University of Göttingen
2009 - 2011	associated fellowship of the research training group 1023: Identification in Mathematical Models: Synergy of Stochastic and Numerical Methods (DFG)
2007	student assistant, University of Göttingen
2003 - 2008	Software development (dot.net), Meier Consult, Braunschweig
2001 - 2003	Software development (C++), Exortus Software GmbH, Letter

3 Organization of Workshops

- Member of the organizing team of the deal.II Workshop 2010 in Heidelberg, August 23-27

4 Research Visits

September to December 2009	Texas A&M University, Department of Mathematics
February to March 2010	Texas A&M University, Department of Mathematics

5 Presentations at Conferences

(* = invited)

2010-11-26	*	*Augmented Lagrangian based preconditioning using Grad-Div stabilization* Nonstandard Discretizations for Fluid Flows, invitation workshop, Banff (Canada)
2010-10-08		*Less painful turbulence benchmarks - solvers, parallelization, and more* Workshop on Calibration of Viscosity Models for Turbulent Flows, Göttingen (Germany)
2010-10-04		*Parallel Solvers for Incompressible Flow Problems* Research Group Meeting 2010, Goslar (Germany)
2010-09-13		*Massively Parallel Finite Element Programming* EuroMPI 2010, Stuttgart (Germany)
2010-09-01		*Massiv-parallele Finite Elemente Simulation mit deal.II* SourceTalk 2010, Göttingen (Germany)
2010-08-24		*Massive Parallel Computations with deal.II* deal.II Workshop 2010, Heidelberg (Germany)
2010-06-14		*Generic Finite Element Programming for Massively Parallel Flow Simulations* Eccomas 2010, Lisbon (Portugal)
2010-03-09		*Algorithms and Data Structures for Massively Parallel Finite Element Codes* Research Seminar, Texas A&M, College Station (USA)
2010-03-06		*Algorithms and Data Structures for Massively Parallel Finite Element Codes* Finite Element Rodeo 2010, Dallas (USA)
2009-12-21		*On Robust Parallel Preconditioning for Incompressible Flow Problems* Texas A&M, College Station (USA)
2009-06-30		*On Robust Parallel Preconditioning for Incompressible Flow Problems* ENUMATH 2009, Uppsala (Sweden)
2008-04-10		*Preconditioning for the stabilized Oseen Problem* Mini-Workshop on Local Projection Stabilization: Theory and Applications, Göttingen (Germany)

6 Teaching

Winter 10/11	assisting numerical analysis
Fall 10	short deal.II introduction
Fall 10	assisting mathematics in computer science II
Winter 09/10	assisting mathematics in computer science I
Fall 09	assisting mathematics in computer science II
Winter 08/09	assisting mathematics in computer science I

7 Publications

Papers in Refereed Journals

[1] M. Olshanskii, G. Lube, T. Heister, and J. Löwe. Grad-div stabilization and subgrid pressure models for the incompressible Navier-Stokes equations. *Computer Methods in Applied Mechanics and Engineering*, 198(49-52):3975 – 3988, 2009.

Papers Submitted to Refereed Journals

[1] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler. Algorithms and Data Structures for Massively Parallel Generic Finite Element Codes. *submitted.*

[2] T. Heister and G. Rapin. Efficient augmented Lagrangian-type preconditioning for the Oseen problem using Grad-Div stabilization. *submitted.*

Papers in Conference Proceedings

[1] T. Heister, M. Kronbichler, and W. Bangerth. Generic Finite Element Programming for Massively Parallel Flow Simulations. *Eccomas 2010 Proceedings*, 2010.

[2] T. Heister, M. Kronbichler, and W. Bangerth. Massively Parallel Finite Element Programming. In Rainer Keller, Edgar Gabriel, Michael Resch, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6305 of *Lecture Notes in Computer Science*, pages 122–131. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15646-5_13.

[3] T. Heister, G. Lube, and G. Rapin. On robust parallel preconditioning for incompressible flow problems. In *Numerical Mathematics and Advanced Applications, ENUMATH 2009.* Springer, Berlin, 2010.

Miscellaneous

[1] T. Heister. Vorkonditionierungsstrategien für das stabilisierte Oseen-Problem. Master's thesis, Institut für Numerische und Angewandte Mathematik, Georg-August-Universität zu Göttingen, 2008.

8 Skills

- languages: German (native) and English (fluent)
- profound knowledge in object oriented programming with C++
- experience with parallel programming (multi-threading, MPI)
- contributer to the deal.II finite element library, `http://www.dealii.org/`
- knowledge of many other programming languages, libraries, and tools (PETSc, Trilinos, MATLAB, dot.net, Java, asp.net, SQL, XML, OpenGL, DirectX)