Scalable parallel simulation of variably saturated flow

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von
Jose A. Fonseca
aus
San José, Costa Rica

Bonn, 2019

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Gutachter:	Prof. Dr. Carsten Burstedde
2. Gutachter:	Prof. Dr. Stefan Kollet
Tag der Promotion:	12.02.2019
Erscheinungsjahr:	2019

Dedicado a mis padres

Summary

In this thesis we develop highly accurate simulation tools for variably saturated flow through porous media able to take advantage of the latest supercomputing resources. Hence, we aim for parallel scalability to very large compute resources of over 10^5 CPU cores.

Our starting point is the parallel subsurface flow simulator ParFlow. This library is of widespread use in the hydrology community and known to have excellent parallel scalability up to 16k processes. We first investigate the numerical tools this library implements in order to perform the simulations it was designed for. ParFlow solves the governing equation for subsurface flow with a cell centered finite difference (FD) method. The code targets high performance computing (HPC) systems by means of distributed memory parallelism. We propose to reorganize ParFlow's mesh subsystem by using fast partitioning algorithms provided by the parallel adaptive mesh refinement (AMR) library `p4est`. We realize this in a minimally invasive manner by modifying selected parts of the code to reinterpret the existing mesh data structures. Furthermore, we evaluate the scaling performance of the modified version of ParFlow, demonstrating excellent weak and strong scaling up to 458k cores of the Juqueen supercomputer at the Jülich Supercomputing Centre.

The above mentioned results were obtained for uniform meshes and hence without explicitly exploiting the AMR capabilities of the `p4est` library. A natural extension of our work is to activate such functionality and make ParFlow a true AMR application. Enabling ParFlow to use AMR is challenging for several reasons: It may be based on assumptions on the parallel partition that cannot be maintained with AMR, it may use mesh-related metadata that is replicated on all CPUs, and it may assume uniform meshes in the construction of mathematical operators. Additionally, the use of locally refined meshes will certainly change the spectral properties of these operators.

In this work, we develop an algorithmic approach to activate the usage of locally refined grids in ParFlow. AMR allows meshes where elements of different size neighbor each other. In this case, ParFlow may incur erroneous results when it attempts to communicate data between inter-element boundaries. We propose and discuss two solutions to this issue operating at two different levels: The first manipulates the indices of the degrees of freedom, While the second operates directly on the degrees of freedom. Both approaches aim to introduce minimal changes to the original ParFlow code.

In an AMR framework, the FD method taken by ParFlow will require modifications to correctly deal with different size elements. Mixed finite elements (MFE) are on the other hand better suited for the usage of AMR. It is known that the cell centered FD method used in ParFlow might be reinterpreted as a MFE discretization using Raviart-Thomas elements of lower order. We conclude this thesis presenting a block preconditioner for saddle point problems arising from a MFE on locally refined meshes. We evaluate its robustness with respect to various classes of coefficients for uniform and locally refined meshes.

Contents

1 Introduction **1**

2 Simulation of subsurface flow **5**
2.1 Mathematical modeling . . . 5
2.1.1 Mass conservation . . . 7
2.1.2 Conservation of momentum . . . 7
2.1.3 Darcy's Law . . . 8
2.2 Introduction to partial differential equations . . . 9
2.3 Partial differential equations in subsurface flow . . . 12
2.3.1 Saturated case . . . 12
2.3.2 Variably saturated case . . . 13
2.4 Numerical methods . . . 14
2.4.1 Time stepping . . . 14
2.4.2 Finite differences . . . 14
2.4.3 Finite volume method . . . 17
2.4.4 Finite element method . . . 18
2.4.5 Mixed finite elements . . . 20
2.5 Discretization equivalences . . . 21
2.6 Parallel computing . . . 21
2.6.1 The message passage interface . . . 23

3 The subsurface flow simulator ParFlow **25**
3.1 Generalities . . . 25
3.2 Discretization . . . 26
3.3 Solvers . . . 26
3.4 Preconditioners . . . 28
3.5 Mesh management . . . 29
3.6 Compute and communication packages . . . 33
3.7 Discussion . . . 35

4 A modified version of ParFlow **37**
4.1 High performance computing essentials . . . 37
4.2 Mesh parallelization . . . 38
4.3 The software library p4est . . . 39
4.4 Enabling p4est as Parflow' s mesh manager . . . 41
4.4.1 Rearranging the mesh layout . . . 42
4.4.2 Attaching subgrids of correct size . . . 44
4.4.3 Querying the ghost layer . . . 44
4.4.4 Further enhancements . . . 45

4.5 Performance results . . . 50
4.5.1 Weak scaling studies . . . 50
4.5.2 Strong scaling studies . . . 52
4.6 Illustrative numerical experiment . . . 54
4.7 Discussion . . . 57

5 Towards AMR in ParFlow **59**
5.1 AMR with `p4est` . . . 59
5.2 Enabling locally refined meshes in ParFlow . . . 59
5.2.1 Correcting the mesh layout . . . 63
5.2.2 Two proposals to correct the communication layout . . . 63
5.3 Discussion . . . 67

6 A block preconditioner for locally refined meshes **69**
6.1 Problem formulation . . . 70
6.1.1 Weak formulation . . . 71
6.1.2 Discretization . . . 72
6.1.3 Adaptivity . . . 73
6.1.4 Preconditioning . . . 74
6.2 Multigrid for Saddle Point Problems . . . 75
6.2.1 Algebraic multigrid (AMG) . . . 75
6.2.2 Smoothers for saddle point problems . . . 77
6.2.3 AMG setup for saddle point systems . . . 78
6.3 Numerical Results . . . 79
6.3.1 Homogeneous Dirichlet conditions . . . 80
6.3.2 An example with Neumann conditions . . . 83
6.3.3 Non trivial conductivity tensor . . . 83
6.3.4 High conductivity contrast . . . 86
6.4 Discussion . . . 87

7 Conclusion and Outlook **93**

List of Figures

2.1 Example of a connected void space . . . 6
2.2 Darcy's experiment . . . 8
2.3 Finite difference mesh example . . . 15
2.4 Crank-Nicholson stencil . . . 17
2.5 Distributed memory architecture . . . 22

3.1 Background example . . . 30
3.2 Parflow's mesh example . . . 32
3.3 Get patches representation . . . 33
3.4 Compute packages representation . . . 35

4.1 Space filling curve . . . 40
4.2 Lexicographic and SFC ordering comparison . . . 41
4.3 Enabling multiple subgrids per process . . . 43
4.4 A `p4est` brick . . . 44
4.5 `p4est ghost` example . . . 45
4.6 Subgrid storate: default and with `p4est` . . . 46
4.7 Deadlock illustration for Algorithm 4.1 . . . 48
4.8 Weak scaling upstream . . . 51
4.9 Weak scaling modified . . . 52
4.10 Weak scaling modified - split . . . 53
4.11 Weak scaling memory heap . . . 53
4.12 `p4est` timing . . . 54
4.13 Strong scaling single subgrid . . . 55
4.14 Strong scaling Amdahh's law . . . 56
4.15 Strong scaling multiple subgrids . . . 56
4.16 Ilustrative numerical experiment . . . 58

5.1 Local refinement illustration . . . 60
5.2 Work flow to enable AMR . . . 60
5.3 Locally refined grid in ParFlow . . . 62
5.4 Corrected mesh associated to the brick from Figure 5.3c. . . . 63
5.5 Enabling locally refined meshes, approach 2 . . . 65
5.6 Enabling locally refined meshes, approach 1 . . . 65

6.1 Degrees of freedom for the rectangular $\mathcal{RT}_0$ element . . . 72
6.2 Locally refined mesh with hanging nodes . . . 73
6.3 $\mathcal{RT}_0$ degrees of freedom identification for VTK visualization . . . 80
6.4 Error plot for the example defined in Section 6.3.1 and uniform meshes . . . 81
6.5 Error plot for the example defined in Section 6.3.1 and adaptive meshes . . . 81
6.6 Error plot for the example defined in Section 6.3.2 and uniform meshes . . . 83

6.7 Error plot for the example defined in Section 6.3.2 and adaptive meshes 84
6.8 Error plot for the example defined in Section 6.3.3: uniform case 85
6.9 Error plot for the example defined in Section 6.3.3: adaptive case 85
6.10 Sample plot of the one dimensional version of $m(\boldsymbol{x}; \boldsymbol{x_0}, a, b, c)$. 87
6.11 Error plot for the 2d example defined in Section 6.3.4 87
6.12 Velocity magnitude for the numerical solution of the example from Section 6.3.4 88
6.13 y-velocity magnitude extrusion illustrating a two dimensional $\mathcal{RT}_0$ vector field . 88
6.14 Error plot for the 3d example defined in Section 6.3.4 89
6.15 Threshold plot from the computed velocity of the 3d example from Section 6.3.4 89
6.16 Degrees of freedom comparison against error 90

List of Tables

4.1 FLOP/s comparison 52
4.2 Strong scaling setup 55

6.1 Iteration count table for a 2d mixed Poisson system (Section 6.3.1) 82
6.2 Iteration count table for a 3d mixed Poisson system (Section 6.3.1) 82
6.3 Iteration count table, non trivial 2d conductivity tensor (Section 6.3.3) 84
6.4 Iteration count table, non trivial 3d conductivity tensor (Section 6.3.3) 86
6.5 Iteration count, high contrast 2d conductivity tensor (Section 6.3.4) 90
6.6 Iteration count, high contrast 3d conductivity tensor (Section 6.3.4) 90

1 Introduction

Regional hydrology studies are often supported by high spatial resolution simulations of subsurface flow of the order $O(1\,\text{km})$ or even less [121, 171, 57, 168, 108, 92]. Such simulations demand enormous computational time and memory resources, and considerations of efficiency become prominent [104, 116].

This is further emphasized if we consider the so-called hyperresolution trend in global hydrology modeling: The high resolution employed in the regional models is also required when addressing water cycle questions at the global scale [169, 23, 22]. Global models typically employ a much coarser space resolution that ranges between $O(10\,\text{km})$ and $O(100\,\text{km})$. An increase in a factor of ten in the space resolution implies a growth up to of a factor of a hundred in computation and storage requirements [23]. Additionally, using finer resolutions may require to explicitly simulate physical processes at the fine scale instead of averaging or parameterizing them, increasing the computational workload even more [22].

Computer simulations of subsurface flow generally proceed by computing a numerical solution of the three dimensional Richards' equation [136]. Assuming a two-phase water-gas system, the Richards' equation can be derived from the generalized Darcy laws [125] under the assumption that the pressure gradient in the gas phase is small. One of the variants of Richards' equation reads

$$\frac{\partial(\phi s(p))}{\partial t} + \nabla \cdot \boldsymbol{u} = f, \tag{1.1a}$$

$$\boldsymbol{u} = -\boldsymbol{\mathcal{K}}\nabla(p - z), \tag{1.1b}$$

where p denotes the pressure head, $s(p)$ is the pressure-dependent saturation, ϕ the porosity of the medium, $\boldsymbol{u}$ the flux, z is the depth below the surface, $\boldsymbol{\mathcal{K}} = \boldsymbol{\mathcal{K}}(\boldsymbol{x}; p)$ the symmetric conductivity tensor and f the source term. Classical discretization methods for partial differential equations (PDE) are then applied to (1.1). These include finite differences (FD) [152], finite elements (FE) [25], finite volume (FV) [112], discontinuous Galerkin (DG) [86] and mixed finite elements (MFE) [31] methods. All these methods cover the simulation domain with a mesh. The latter is used to define the spacial location of the discrete values that should approximate the continuous unknown (the pressure head in the case of (1.1)). Hyperresolution modeling demands that the size of the mesh elements should be small enough such that the relevant physics are resolved accordingly [104].

A clear computational trend is that the computational power to perform extensive calculations is provided by parallel computing [122]. Hence, effective employment of high performance (parallel) computing strategies (HPC) is fundamental. Computer codes should be able to efficiently use the resources from massively parallel computers with hundreds of thousands of processing units. Currently, the largest machines offer a theoretical performance of the order of dozens of petaflops, that is 10^{16} floating point operations per second (FLOPS) [133]. The next generation of parallel machines offering exaflop (10^{18} FLOPS) performance is expected in the near future.

There has been a significant focus across the hydrology modeling community to incorporate modern HPC paradigms into their simulation codes. Examples are PARSWMS [83], TOUGH2-MP [172], PFLOTRAN [82], Hydrogeosphere HGS [91], RichardsFOAM [127], suGWFOAM [113] and ParFlow [10, 96, 103].

In exploiting HPC resources with parallel computing, the mesh employed by the discretization method plays a key role. Many PDE-based simulators are parallelized by partitioning the mesh into segments. Each one of these is assigned to a unique central processing unit (CPU), which is responsible for updating the degrees of freedom corresponding to that subset only. Dedicated libraries that target the problem of partitioning and distributing a given mesh on a parallel computer are for example ParMETIS [101], Zoltan [55], Chombo [49], peano [166], `p4est` [36], SAMRAI [167] and others. Some of the previously mentioned modeling platforms make use of these libraries in their implementations, for example PARSWMS and PFLOTRAN delegate their mesh management to the library ParMETIS. Others like ParFlow manage the mesh parallelization internally.

Unfortunately, even with the power of current supercomputing facilities, problems requiring extremely high spacial resolution may be infeasible to solve if a uniformly refined mesh is imposed. A valuable tool to reduce the workload of numerical simulations without compromising the accuracy is adaptive mesh refinement (AMR) [11, 20, 56]. AMR builds upon the concept of using a fine mesh locally only where high accuracy is required and keeping it coarse elsewhere. When combined with suitable error indicators that guide when to refine or coarsen the mesh [26, 162, 46], AMR can reduce the workload and run time of a simulation by using less mesh elements in comparison with a uniform mesh [43, 124].

In terms of efficiency, one the most successful approaches to implement AMR is based on space filling curves (SFC) [12, 141]. Here, the information about the size and position of mesh elements is maintained within a tree data structure that can be associated with a recursive refinement scheme where, for example in two space dimensions, a square is subdivided into four half size child squares. The leaves of the tree represent the elements of the mesh and they can be linearly ordered with the help of a SFC. The latter is a mapping between a higher dimensional space and a one dimensional space that induces an ordering of the leaves and hence on the mesh elements. Then, a partition on the mesh is defined by decomposing the SFC into parts containing a balanced amount of elements. Libraries implementing this approach are Dendro [142] for hexahedral meshes and sam(oa)^2 [117] for 2d triangular meshes. These libraries have shown to scale to up to tens of thousands and hundred thousand of processes, respectively.

The canonical domain associated with a tree is a cube (a square in 2D). In order to represent more complex domain shapes, the SFC approach has been extended to consider unions of trees [44]. The software library `p4est` [44, 94] provides efficient algorithms that implement a self-consistent set of parallel AMR operations for hexahedral meshes. It employs a Morton SFC [123] and has shown excellent parallel scalability to up to 3.1 million processes [40, 137, 41].

The goal of this thesis is to develop parallel adaptive solvers for subsurface flow able to take advantage of the latest supercomputing resources. Our starting point is the software library ParFlow, which provides a solver for (1.1) based on a cell centered finite differences discretization on uniform hexahedral meshes. ParFlow exploits HPC resources by using distributed memory parallelism (MPI). We propose to reorganize the mesh subsystem of this library using partitioning algorithms provided by the parallel adaptive mesh refinement library `p4est` with a view to introduce AMR to the subsurface simulation ParFlow. As a related topic, introducing locally refined meshes into ParFlow will certainly change the spectral properties of the discrete mathematical operators appearing after discretization. Hence, we investigate alternative

preconditioners for saddle point problems.

We begin this thesis with a review of the mathematical and computational tools frequently employed in the simulation of subsurface flow. Hence, we prepare the necessary concepts in order to introduce the reader to the simulation platform ParFlow. In particular, we focus on the algorithmic approach ParFlow implements to target HPC systems. This leads to a detailed study of how ParFlow's computational mesh is distributed in parallel. The latter research is of fundamental importance for the overall goal of this thesis: We are able to identify parallel bottlenecks that may limit the code's scalability as well as algorithms based on traditional assumptions that cannot be maintained with an AMR implementation. In particular, we note that ParFlow does not implement a strictly distributed storage of its mesh.

With this knowledge, we proceed to implement an interface that sets `p4est` as new mesh manager without introducing disrupting changes in the original code. Our effort results in a modified version of ParFlow with two valuable features: First, it is backwards compatible with the upstream one. And second, it takes the code a step further in order to take advantage of the computational resources offered by the latest HPC systems [38]. At this point, we are still keeping the ParFlow mesh uniform. Hence, a natural extension is to make use of the AMR functionality provided by `p4est` to introduce locally refined meshes in ParFlow. We discuss how this can be done, issues one may encounter due to the (previously identified) assumptions not compatible with AMR and solutions to overcome such issues.

The work done integrating ParFlow with `p4est` exclusively modifies the mesh logic of the former. We keep the discretization method and numerical solvers. Switching to an AMR framework will have effects on both of these. Regarding the discretization, we require it to be compatible with nonconforming hexahedral meshes, i.e., it should achieve continuity of the flux $\boldsymbol{u}$ across connecting faces of different size. While traditional finite difference schemes are no longer directly applicable, there are MFE spaces suitable to enforce this property, one example being the Raviart-Thomas spaces [135]. Using particular quadratures, a MFE discretization using Raviart-Thomas spaces of lower order is equivalent to the FD approach taken in ParFlow [6]. This suggest that reinterpreting ParFlow's discretization as a MFE method might be beneficial when switching to locally refined meshes. Because a MFE discretization (1.1) leads to a linear system with saddle point structure, in the last part of this thesis we study preconditioners for this kind of system.

This thesis is organized as follows:

In Chapter 2 we summarize the main mathematical and computational tools employed to simulate subsurface flow. We then present the main features of the subsurface flow simulator ParFlow in Chapter 3, with a detailed description of the parallelization of its computational mesh and how it enables but also limits parallel scalability.

In Chapter 4 we present a modified version of ParFlow in which the mesh management has been delegated to the parallel AMR library `p4est`. We realize this in a minimally invasive manner following a principle that we call *reinterpret instead of rewrite* akin to [13]: We carefully select parts of the ParFlow code to reinterpret and reuse most of the existing mesh data structures. Without exploiting the AMR capabilities of `p4est`, this modified version of ParFlow already offers a wider range of configurations the code may be executed with and an improved performance with respect to the upstream version. We conclude this chapter evalu-

ating the scaling performance of the modified version of ParFlow, demonstrating good weak and strong scaling up to 458K processes. This chapter was first published in 2018 [38].

Chapter 5 is devoted to expose our ideas to prepare the ParFlow code for local mesh refinement by activating the AMR functionalities of `p4est`. In particular, we focus on the correctness of the mesh connectivity and the correct propagation of information when the code is executed on a parallel machine. We conclude this chapter proposing two solutions to ensure that ParFlow correctly determines the MPI peers and senders of a message when a locally refined mesh is employed.

We conclude this thesis presenting a block preconditioner for saddle point problems (SPAMG) in Chapter 6. The design of this preconditioner corresponds to work realized by co-author B. Metsch [120]. The SPAMG preconditioner is compared against standard diagonal and Schur preconditioners for a low-order Raviart-Thomas discretization of a mixed diffusion problem on adaptive quadrilateral meshes. We finish the chapter with numerical experiments in 2D and 3D showing that the SPAMG preconditioner displays nearly mesh-independent iteration counts for adaptive meshes and heterogeneous coefficients. We are currently preparing Chapter 6 for publication.

2 Simulation of subsurface flow

In this chapter we provide a summary of the mathematical and computational tools that are frequently employed in the simulation of subsurface flow. We begin by covering some background information concerning the physical concepts of subsurface flow phenomena in order to derive the corresponding model equations.

2.1 Mathematical modeling

In general, a model is a simplified version of a real system or a set of processes that take place within it, which approximately describes relevant features of the system being studied [17, pp. 29]. The fundamental aim of a model is to predict the behavior of a system in order to better understand it. Roughly speaking, a model refers to any construction that approximates a real situation. Developing such construction usually starts by defining a conceptual framework, which is a set of assumptions that express the current level of understanding of the system's behavior. A mathematical model is a translation of this conceptual framework into mathematical language, which in the case of fluid flow simulation usually takes the from of partial differential equations (PDEs). We will give a brief introduction to this topic in Section 2.2 and devote the rest of this section to introduce key concepts and the physical principles for describing the governing equations of subsurface flow.

The subsurface is a porous medium, consequently, we require to properly define what such a medium is. We start recalling

Definition 2.1.1 (Material phase)**.** A material phase is a homogeneous portion of space that is separated from other such portions by a definite physical boundary [17, pp. 42].

Intuitively, a porous medium refers to a material that allows a fluid to infiltrate. Examples of porous media materials are sand, soil, fractured rocks, ceramics, foam, etc. There are two features that all these examples, and in general all porous media, have in common. First, it is possible to identify a portion that permits fluid transport and a portion that acts as an obstacle. The former one is known as the void space, and the latter as the solid matrix or solid phase. Second, the void space and the solid matrix are distributed throughout the material in a very special way: sufficiently large samples of the material at different locations always contain void space and solid phase. In order to facilitate the exposition, we will assume that samples are spheres centered at a given point in space. A more formal definition is the following.

Definition 2.1.2 (Porous medium)**.** A domain $\mathcal{U} \subset \mathbb{R}^d$, $d = 2, 3$ is said to be a porous medium if it satisfies the following properties.

i) $\mathcal{U} = \mathcal{U}_s \cup \mathcal{U}_v$ with $\mathcal{U}_s \cap \mathcal{U}_v = \emptyset$, where $\mathcal{U}_s$ denotes the solid phase and $\mathcal{U}_v$ the void space.

ii) There exists a $r > 0$ such that for every point $x \in \mathcal{U}$, $B_r(x) \cap \mathcal{U}_s \neq \emptyset$ and $B_r(x) \cap \mathcal{U}_v \neq \emptyset$. Here $B_r(x) := \{y \in \mathcal{U} : \|x - y\| < r\}$.

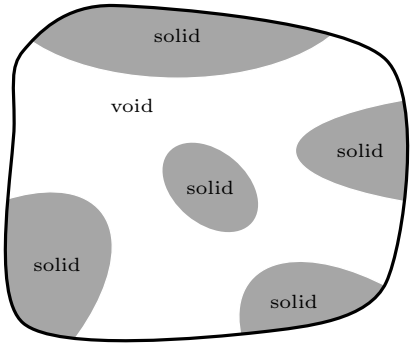


Figure 2.1: Example of a connected void space. The void space permits fluid transport while the solid matrix serves as an obstacle.

Definition 2.1.3 (Representative elementary volume (REV))**.** The set $B_r(x)$ from Definition 2.1.2 is called Representative Elementary Volume, abbreviated REV.

Thus, a porous medium is a domain in which a solid phase and a void space can be identified and for which a REV can be found. We are interested in modeling the situation where multiple fluid phases that occupy the void space move through it. Hence, it is natural to assume that the void space is connected. See Figure 2.1.

Remark 2.1.4. Because gases in general do not maintain a distinct physical boundary between them, it is assumed that there can only be one gaseous phase in a given system.

As introduced above, the movement of fluid phases through the void space is mathematically expressed as a set of PDEs. The construction of such equations is based on the fundamental assumption that each of the phases in the porous medium, including the solid matrix, behave as a continuum that fills up the entire space. This supposition allows to describe state variables and material parameters of a phase as continuous differentiable functions of time and the spacial coordinates.

Remark 2.1.5. An intermediate step would be to regard each phase as a continuum only in the region of space it fills and not over the whole porous medium. Such approach will then require detailed knowledge of the boundary between phases. This information is in general not known and even impossible to determine for practical applications.

The continuum approach overlooks the details at the molecular level. In order to upscale to a coarser level of description, a practical method developed in the engineering sciences is *volume averaging* [15, pp. 7]. The phase behavior at the microscopic scale, e.g., the molecular level, is averaged over a REV in order to obtain the phase behavior as a continuum. Consequently, the size of the REV has to be carefully chosen such that the averaged values of all geometric characteristics of a phase are statistically meaningful. The averaged values are called macroscopic quantities and for a particular porous medium, they can be determined experimentally [16, Sec. 1.1.3]. Another approach is provided by *homogenization* theory, which makes use of asymptotic expansions to let the microscopic scale tend to zero. See for example [88, Ch. 1] for a comprehensive introduction to this technique.

The building blocks of mathematical models based on the continuum assumption are conservation laws. In general, a conservation law states that a measurable property of a system remains constant as the system changes in time. For the case of subsurface flow, the driving equations are based on the conservation of mass and momentum.

2.1.1 Mass conservation

Let $\Omega \subset \mathbb{R}^d$ be a region in the space ($d = 2, 3$) filled with a fluid. We consider the portion of the fluid enclosed in a subdomain $V \subset \Omega$. In addition, we assume that there exists a time dependent vector field, $\boldsymbol{\varphi} : [0, T] \times \Omega$ that describes the motion of a particle at position $\boldsymbol{x}$ with time. Further, we assume that $\boldsymbol{\varphi}(t, \boldsymbol{x})$ is differentiable and invertible for each $t \in [0, T]$. Then, for $t > 0$, the fluid occupies the domain

$$V_t := \{\boldsymbol{\varphi}(t, \hat{\boldsymbol{x}}) \mid \hat{\boldsymbol{x}} \in V\}. \tag{2.1}$$

We said that V_t is the volume moving with the fluid. Our assumptions imply that there is a well defined velocity vector field for each fixed $\boldsymbol{x} \in V_t$, characterized as

$$\boldsymbol{u}(t, \boldsymbol{x}) := \frac{\partial}{\partial t}\boldsymbol{\varphi}(t, \hat{\boldsymbol{x}}), \quad \boldsymbol{x} = \boldsymbol{\varphi}(t, \hat{\boldsymbol{x}}). \tag{2.2}$$

The principle of mass conservation states that the mass of a fluid enclosed in V_t is constant with respect to t. If $\rho(t, \boldsymbol{x})$ denotes the mass density of the fluid, this is mathematically expressed as

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{V_t} \rho(t, \boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0. \tag{2.3}$$

In order to evaluate the time derivative in (2.3) we need the Reynolds transport theorem (see eg. [115]).

Theorem 2.1.6 (Transport Theorem)**.** *Let V_t and $\boldsymbol{u}$ be defined as in (2.1) and (2.2) respectively. Further, let $f : [0, T) \times \Omega \to \mathbb{R}$ a differentiable scalar function. Then, it holds that*

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{V_t} f(t, \boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_{V_t} \left(\frac{\partial}{\partial t} f + \nabla \cdot (f\boldsymbol{u})\right)(t, \boldsymbol{x})\,\mathrm{d}\boldsymbol{x}. \tag{2.4}$$

Using (2.4) in (2.3) yields

$$\int_{V_t} \left(\frac{\partial}{\partial t}\rho + \nabla \cdot (\rho\boldsymbol{u})\right)(t, \boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0, \tag{2.5}$$

which should be valid for every V_t and hence we obtain the *continuity equation*

$$\frac{\partial}{\partial t}\rho + \nabla \cdot (\rho\boldsymbol{u}) = 0. \tag{2.6}$$

2.1.2 Conservation of momentum

We consider a time dependent volume V_t and $\rho(t, \boldsymbol{x})$ as in the last section. The principle of conservation of (linear) momentum states that the rate of change of momentum in the region enclosed by V_t is equal to the sum of forces applied to that region. There are two different types:

- Body forces, which act throughout the volume V_t and can be expressed as

$$\int_{V_t} \rho(t, \boldsymbol{x})\boldsymbol{f}(t, \boldsymbol{x})\,\mathrm{d}\boldsymbol{x}, \tag{2.7}$$

  where $\boldsymbol{f}$ is a given force density per unit volume.

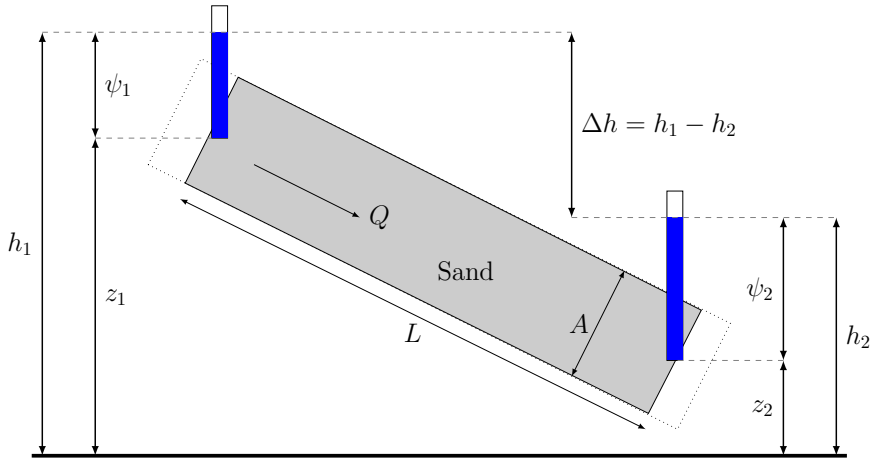


Figure 2.2: Experimental setup to investigate water flow Q through an inclined sand column of length L and cross sectional area A. The quantities $h_i = \psi_i + z_i$, for $i = 1, 2$, represent the hydraulic head (2.12) measured at the two endpoints the of the column. Figure adapted from [17].

- Surface forces, which according to Cauchy's theorem [80, Ch. 5] can be expressed as

$$\int_{\partial V_t} \boldsymbol{\sigma}(t, \boldsymbol{x}) \boldsymbol{n} \, \mathrm{d}s, \tag{2.8}$$

  where $\boldsymbol{\sigma}$ is the symmetric stress tensor and $\boldsymbol{n}$ is the outward unit normal to the boundary of V_t.

The principle of conservation of (linear) momentum is then mathematically expressed as

$$\frac{\mathrm{d}}{\mathrm{d}t} \int_{V_t} \rho(t, \boldsymbol{x}) \boldsymbol{u}(t, \boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} = \int_{V_t} \rho(t, \boldsymbol{x}) \boldsymbol{f}(t, \boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} + \int_{\partial V_t} \boldsymbol{\sigma}(t, \boldsymbol{x}) \boldsymbol{n} \, \mathrm{d}s. \tag{2.9}$$

Applying (2.4) componentwise in the left term of (2.9) and the divergence theorem to the surface integral we obtain

$$\int_{V_t} \left\{ \frac{\partial}{\partial t}(\rho \boldsymbol{u}) + \boldsymbol{u} \cdot \nabla(\rho \boldsymbol{u}) + (\rho \boldsymbol{u}) \nabla \cdot \boldsymbol{u} - \rho \boldsymbol{f} - \nabla \cdot \boldsymbol{\sigma} \right\} (t, \boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} = 0, \tag{2.10}$$

which should be valid for any V_t. Hence, we find the differential form of the momentum equation,

$$\frac{\partial}{\partial t}(\rho \boldsymbol{u}) + \boldsymbol{u} \cdot \nabla(\rho \boldsymbol{u}) + (\rho \boldsymbol{u}) \nabla \cdot \boldsymbol{u} - \rho \boldsymbol{f} - \nabla \cdot \boldsymbol{\sigma} = 0. \tag{2.11}$$

2.1.3 Darcy's Law

Fluid flow in the subsurface is driven by the *hydraulic head*

$$h = \frac{p}{\rho g} + z, \tag{2.12}$$

where z is the elevation with respect to some datum, p and ρ denote the fluid's pressure and density and g the gravity acceleration. The quantity

$$\psi = \frac{p}{\rho g} \tag{2.13}$$

is called *pressure head* and measured in length units. With an experimental setup similar to the one depicted in Figure 2.2, Henri Darcy [51] concluded that the water flux Q in a homogeneous, saturated porous medium, represented in his experiment as a sand column of cross sectional area A filled with water, is proportional to the change in the hydraulic head over a specified length L, multiplied with the cross sectional area of the column, that is,

$$Q = -KA\frac{\Delta h}{L}. \tag{2.14}$$

The proportionally constant K is called *hydraulic conductivity* and has units of length over time. It is a material depend quantity expressing the ease with which a fluid flows through the void space. The hydraulic conductivity is commonly written as

$$K = k\frac{\rho g}{\mu}, \tag{2.15}$$

where μ denotes the dynamic viscosity of the fluid and k is the permeability of the porous medium. The generalization of (2.14) to a three dimensional anisotropic medium, reads (see eg. [17, Sec. 4.1.2])

$$\boldsymbol{q} = -\boldsymbol{\mathcal{K}}\nabla h = -\boldsymbol{\mathcal{K}}\nabla(\psi + z), \tag{2.16}$$

where $\boldsymbol{q}$ denotes *specific discharge*: the volume of fluid passing per unit area of porous medium per unit time. The hydraulic conductivity takes now the form of a second order symmetric tensor $\boldsymbol{\mathcal{K}}$. The relation (2.15) also holds for an anisotropic porous medium, implying that the permeability is also a second order tensor. It is frequently assumed that the principal directions of anisotropy are aligned with a selected coordinate system, implying that $\boldsymbol{\mathcal{K}}$ becomes diagonal.

A key parameter in the description of flow through a porous media is the porosity, which is defined as the volume fraction occupied by the void space and it is usually denoted as ϕ. Instead of the specific discharge, one is usually interested in the average fluid velocity through the porous medium $\boldsymbol{v}$. The relation between these is given by $\boldsymbol{q} = \phi\boldsymbol{v}$, which is intuitively clear from the fact that flow can only occur in the fraction occupied by the void space.

It is important to emphasize that (2.16) is derived under saturated flow conditions, and it may be adjusted to account for unsaturated and two-phase flow. We will present the corresponding equations in Section 2.3. To finish this section, we would like to comment on the range of validity of Darcy's law. In a general fluid flow simulation, the Reynolds number (Re) constitutes a fundamental parameter that helps to differentiate laminar and turbulent flow, occurring at low and higher velocities, respectively. The Reynolds number is as measure of the relative size of the non-viscous (convection) and viscous (diffusion) forces acting on a moving fluid [14]. Experimental evidence indicates that Darcy's law is valid when $\mathrm{Re} < 10$, and that most of the saturated subsurface flow occurs within this range [17, p. 147].

2.2 Introduction to partial differential equations

Partial differential equations appear naturally as modeling tools for many fundamental physical processes. An extensive amount of work has been dedicated in order to understand them as a mathematical object. A fundamental step in order to attempt to solve, either analytically or numerically, a given partial differential equation is to recognize its type. This has implications on the kind of boundary conditions required to define a well posed problem and also affects the choice of a suitable numerical scheme. The objective of this section is to present a brief

introduction to partial differential equations and their classification in the second order case. The material exposed here is covered in classical text books; see e.g., [95, 151, 76, 64] for further details. For the rest of this section, Ω will denote a domain in $\mathbb{R}^d$, $d \geq 1$ with Lipschitz boundary $\Gamma = \partial\Omega$ and $u : \Omega \to \mathbb{R}$ a scalar function smooth enough so that all derivatives appearing make sense.

Definition 2.2.1. (Notation)

i) A multiindex $\boldsymbol{\alpha}$ is an integer-valued vector $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_d)$. Its order is defined as $|\boldsymbol{\alpha}| := \alpha_1 + \alpha_2 + \cdots + \alpha_d$.

ii) Let $\boldsymbol{x} = (x_1, \ldots, x_d)$ be a point in $\mathbb{R}^d$. For a multiindex $\boldsymbol{\alpha}$, we define $D^{\boldsymbol{\alpha}} u(\boldsymbol{x}) := \dfrac{\partial^{|\boldsymbol{\alpha}|} u(\boldsymbol{x})}{\partial^{\alpha_1} x_1 \cdots \partial^{\alpha_d} x_d}$.

iii) For a given $k \in \mathbb{N}$ the set of all partial derivatives of order k is written as $D^k u(\boldsymbol{x}) := \{D^{\boldsymbol{\alpha}} u(\boldsymbol{x}) \mid |\boldsymbol{\alpha}| = k\}$.

Note that $D^k u(\boldsymbol{x})$ can be identified with a point in $\mathbb{R}^{d^k}$, for example for $k = 1$ we identify $Du(\boldsymbol{x})$ with the gradient of u, $\nabla u = \left(\frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_d}\right)$ and $D^2 u(\boldsymbol{x})$ with the Hessian matrix $\boldsymbol{H}u = \left(\frac{\partial^2 u}{\partial x_i \partial x_j}\right)_{i,j=1,\ldots,d}$. With this notation in mind, a general definition of a partial differential equations reads as follows.

Definition 2.2.2. (PDE) Let $k, d \in \mathbb{N}$ and $\Omega \subset \mathbb{R}^d$, $d \geq 2$ denote a domain.

i) A partial differential equation (PDE) of order $k \geq 1$ is a functional relation of the form

$$F(D^k u(\boldsymbol{x}), D^{k-1} u(\boldsymbol{x}), \ldots, Du(\boldsymbol{x}), u(\boldsymbol{x}), \boldsymbol{x}) = 0, \tag{2.17}$$

where $F : \mathbb{R}^{d^k} \times \mathbb{R}^{d^{k-1}} \times \cdots \times \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}$ is a given function and $u : \Omega \to \mathbb{R}$ is the unknown solution.

ii) A *strong solution* of (2.17) is a k-times continuously differentiable function $v(\boldsymbol{x})$ satisfying (2.17).

We are interested in the second order case ($k = 2$) and in particular when (2.17) takes the form

$$\sum_{i,j=1}^{d} a_{ij}\left(Du(\boldsymbol{x}), u(\boldsymbol{x}), \boldsymbol{x}\right) \frac{\partial u(\boldsymbol{x})}{\partial x_i \partial x_j} + b\left(Du(\boldsymbol{x}), u(\boldsymbol{x}), \boldsymbol{x}\right) = 0 \tag{2.18}$$

for some coefficients $a_{ij} : \mathbb{R}^{2d+1} \to \mathbb{R}$ and $b : \mathbb{R}^{2d+1} \to \mathbb{R}$. This kind of equation is known as *quasilinear* second order PDE. This contains the *semilinear* case, when $a_{ij} = a_{ij}\left(u(\boldsymbol{x}), \boldsymbol{x}\right)$ and also the *linear* case when the coefficients a_{ij} depend only on the spacial location and additionally b has the form

$$b = \sum_{i=1}^{d} c_i(\boldsymbol{x}) \frac{\partial u(\boldsymbol{x})}{\partial x_i} + f(\boldsymbol{x}), \tag{2.19}$$

where $f, c_i : \mathbb{R}^d \to \mathbb{R}$ for $i = 1, \ldots, d$ are given scalar functions. Equations like (2.18) may be further classified as elliptic, hyperbolic, and parabolic. Such classification is useful because each

of the three types requires a substantially different kind of analysis, from both the theoretical and numerical point of view. Concerning this classification, the main role is played by the second order differential operator

$$\sum_{i,j=1}^{d} a_{ij}\left(Du(\boldsymbol{x}), u(\boldsymbol{x}), \boldsymbol{x}\right) \frac{\partial u(\boldsymbol{x})}{\partial x_i \partial x_j}, \tag{2.20}$$

and more specifically by the matrix $\boldsymbol{A}(\boldsymbol{x})$ with coefficients $\boldsymbol{A}_{ij}(\boldsymbol{x}) := a_{ij}\left(Du(\boldsymbol{x}), u(\boldsymbol{x}), \boldsymbol{x}\right)$, for $i, j = 1, \ldots, d$.

Definition 2.2.3. a) The equation (2.18) is elliptic at the point $\boldsymbol{x}$ if all the eigenvalues of the matrix $\boldsymbol{A}(\boldsymbol{x})$ are different from zero and have the same sign.

b) The equation (2.18) is parabolic at the point $\boldsymbol{x}$ if the matrix $\boldsymbol{A}(\boldsymbol{x})$ has one zero eigenvalue and the remaining $d-1$ nonzero eigenvalues have the same sign.

c) The equation (2.18) is hyperbolic at the point $\boldsymbol{x}$ if all the eigenvalues of the matrix $\boldsymbol{A}(\boldsymbol{x})$ are different from zero and one eigenvalue has the opposite sign of the remaining $d-1$.

In the parabolic and hyperbolic cases, the eigenvalue displaying a different sign as the remaining ones corresponds to a special variable that is normally associated with time. Examples of the above mentioned classes are:

- The Poisson equation
$$-\Delta u(\boldsymbol{x}) = f(\boldsymbol{x}), \tag{2.21}$$
where $f : \mathbb{R}^d \to \mathbb{R}$ is a given function and $\Delta := \nabla \cdot \nabla$ denotes the Laplace operator. It is the prototype of an elliptic equation.

- If $u : (0, T) \times \Omega \to \mathbb{R}$, with $T > 0$, then the heat equation reads
$$\frac{\partial u(t, \boldsymbol{x})}{\partial t} - \Delta_{\boldsymbol{x}} u(t, \boldsymbol{x}) = 0, \tag{2.22}$$
where $\Delta_{\boldsymbol{x}}$ denotes the Laplace operator acting in the $\boldsymbol{x}$ variable. It is the canonical example of a parabolic PDE.

- For a function u as above, the wave equation,
$$\frac{\partial^2 u(t, \boldsymbol{x})}{\partial^2 t} - \Delta_{\boldsymbol{x}} u(t, \boldsymbol{x}) = 0, \tag{2.23}$$
is the prime example of a hyperbolic PDE.

It is important to emphasize that the classification introduced in Definition 2.2.3 does not cover the whole spectrum of existing PDEs. There are equations not belonging to any of this classes. A second important point is that the classification depends on the spacial location and the solution itself. For example, the Euler-Tricomi equation [109],

$$\frac{\partial^2 u(x, y)}{\partial^2 x} + x \frac{\partial^2 u(x, y)}{\partial^2 y} = 0, \tag{2.24}$$

is elliptic if $x > 0$, parabolic if $x = 0$ and hyperbolic if $x < 0$. Another example is the following variant of the Richards equation [136]

$$\frac{1}{4}\frac{\partial(s(p))}{\partial t} - \nabla \cdot (\nabla(p - z)) = Q, \tag{2.25}$$

where $s(p) : \mathbb{R} \to [0, 1]$ denotes the pressure-head dependent water saturation in a given porous medium, Q is a source term, and z denotes the elevation below the surface. This PDE is elliptic in the fully saturated case ($s(p) \equiv 1$) and parabolic otherwise.

In analogy with ordinary differential equations, none of the equations (2.21)–(2.23) or the more general case (2.18) provide enough information to define a unique solution u. Depending on the type, one may need to prescribe values of the unknown function and/or its derivatives at the boundary of the domain in order to define a well posed problem. For time dependent problems like the parabolic and hyperbolic case, an initial condition, i.e., values of the unknown function at some starting time point t_0, is also required. We speak about Dirichlet boundary conditions when the sought function u is prescribed on parts of the domain's boundary. When the prescribed values correspond to derivatives of u we speak of Neumann boundary conditions. A Robin condition corresponds to a linear combination of the previous two [151].

The notion of strong solution given in Definition 2.2.2 is in general too restrictive for the equations encountered in practice. The regularity conditions implicitly assumed when inserting a function u into equation (2.18) may be too strong. In many cases a strong solution fails to exist because we are looking into the wrong class of functions. Hence, it is convenient to enlarge the definition of solution and consider classes of spaces that allow functions with jumps and even discontinuities. This is in line with the notion that natural phenomena are frequently driven by non-smooth or even discontinuous processes. Such an approach has lead to consider *weak solutions* and variational methods. See e.g., [64, Ch. 6]. We will not go into detail here and postpone the basic definitions until we require them in Chapter 6.

Unfortunately, even when searching for a solution in a suitable function space, the class of PDEs for which an explicit solution may be found is reduced to a few examples. PDEs modeling real physical systems with an acceptable accuracy do not belong to this category and the development of numerical techniques to compute an approximate solution has become a central tool to gain understanding of such systems. We will present a summary of the main numerical methods for solving PDEs in Section 2.4.

2.3 Partial differential equations in subsurface flow

Mathematical modeling of subsurface flow problems usually lead to second order PDEs. In this section we summarize the basic equations for saturated and variably saturated flow. The former occurs when the fluid occupies the whole void space and the latter corresponds to the case where the void space is filled up with a fluid and a gaseous phase (water and air).

2.3.1 Saturated case

For a fluid occupying the whole void space of a porous medium with porosity ϕ, the governing equation may be derived by combining a variant of the continuity equation (2.6) written for the fluid phase and Darcy's law. The former takes the form

$$\frac{\partial}{\partial t}(\phi\rho) + \nabla \cdot (\phi\rho\boldsymbol{u}) = Q, \tag{2.26}$$

where $\boldsymbol{u}$ denotes the mean velocity of the fluid in the void space, we now consider sources (or sinks) Q, measured as volume per unit time. Combining (2.26) with Darcy's law (2.16) and recalling that $\boldsymbol{u}$ relates to the Darcy flux $\boldsymbol{q}$ by $\boldsymbol{u} = \phi^{-1}\boldsymbol{q}$, we obtain

$$\frac{\partial}{\partial t}(\phi\rho) - \nabla \cdot (\boldsymbol{\mathcal{K}}\rho\nabla(\psi + z)) = Q. \tag{2.27}$$

Employing additional assumptions on the compressibility of the fluid and the porous medium deformability, see [17, Sec. 5.1.3], the above equation may be written as

$$S_s\frac{\partial\psi}{\partial t} - \nabla \cdot (\boldsymbol{\mathcal{K}}\nabla(\psi + z)) = Q. \tag{2.28}$$

where S_s denotes the coefficient of specific storage, see [17, pp. 175] for a definition.

2.3.2 Variably saturated case

The governing equation for fluid flow in a porous medium whose void space is occupied by both a gas and a fluid phase can be obtained combining a special form of Darcy's law, the continuity equation for the fluid phase and assuming that the movement of gas is negligible. Darcy's law takes the form

$$\boldsymbol{q}_w = -\boldsymbol{\mathcal{K}}k_r\nabla(\psi_w + z), \tag{2.29}$$

where q_w is the Darcy flux for the water phase, $\boldsymbol{\mathcal{K}}$ the saturated hydraulic conductivity tensor, k_r is a scalar function denoting the relative permeability with respect to the water phase and ψ_w the water pressure head. In addition, if $\boldsymbol{u}_w$ denotes the mean velocity of the water, and $S_w \in [0, 1]$ describes the water saturation, the continuity equation for the water phase is written as

$$\frac{\partial}{\partial t}(\phi\rho S_w) + \nabla \cdot (\phi\rho\boldsymbol{u}_w) = Q. \tag{2.30}$$

Again, $\boldsymbol{u}_w$ is related to the Darcy flux of the water phase by $\boldsymbol{u}_w = \phi^{-1}\boldsymbol{q}_w$. Combining the latter with (2.29) and (2.30) we obtain

$$\frac{\partial}{\partial t}(\phi\rho S_w) - \nabla \cdot (\boldsymbol{\mathcal{K}}k_r\rho\nabla(\psi + z)) = Q. \tag{2.31}$$

Once more, simplifying assumptions on the compressibility of the fluid and the porous medium deformability (see eg. [17, Sec. 5.1.3] or [90, Sec. 4.5.1]), allow (2.31) to be written as

$$S_sS_w\frac{\partial\psi}{\partial t} + \phi\frac{\partial S_w}{\partial t} - \nabla \cdot (\boldsymbol{\mathcal{K}}k_r\nabla(\psi + z)) = Q. \tag{2.32}$$

The water saturation S_w and the permeability k_r depend on the materials properties and may be written in the functional form

$$k_r = k_r(S_w) \quad \text{and} \quad S_w = S_w(\psi). \tag{2.33}$$

These formulas involve fitting coefficients that can be determined by solving an inverse problem. Examples include the Brooks and Corey [33] and Van Genuchten [160] specifications.

2.4 Numerical methods

As pointed out in Section 2.2 most of the "interesting" PDEs cannot be solved analytically and we can at most approximate its solution numerically. In order to perform such a task, we must translate the continuous problem to a discrete one, hence instead of solving a PDE we solve a system of algebraic equations with the hope that the discrete values correctly reproduce the behavior of the continuous unknowns of the PDE. Different types of problems in physics generally correspond to different types of PDEs. In analogy, the numerical methods aiming to solve these equations have substantial differences depending on the type. In this section we present a summary of the main numerical schemes for PDEs. Detailed information on each method can be found in the references presented at the beginning of each subsection. Because equations of parabolic type appear naturally in the study of subsurface flow, we first recall the most basic time discretization scheme.

2.4.1 Time stepping

Our starting point is an initial value problem for a system of ordinary differential equations

$$\frac{d\boldsymbol{w}}{dt} = F(\boldsymbol{w}, t), \tag{2.34a}$$

$$\boldsymbol{w}(0) = \boldsymbol{w}^0, \tag{2.34b}$$

where $t \in (0, T)$, $T > 0$ and $\boldsymbol{w}(t) \in \mathbb{R}^d$. We introduce a partition of $[0, T]$ of N_T intervals of size $\Delta t = T/N_T$. Let $t_n = n \cdot \Delta t$. We use the notation $\boldsymbol{w}^n = \boldsymbol{w}(t_n)$, and for a fixed $\theta \in [0, 1]$ we define by $F^{n+\theta}$ the time average

$$F^{n+\theta} := \theta F(\boldsymbol{w}^{n+1}, t_{n+1}) + (1 - \theta)F(\boldsymbol{w}^n, t_n). \tag{2.35}$$

The θ-method for (2.34) is defined by: Given $\boldsymbol{w}^0$, find $\boldsymbol{w}^1, \ldots, \boldsymbol{w}^{N_T}$ such that

$$\boldsymbol{w}^{n+1} = \boldsymbol{w}^n + \Delta t F^{n+\theta} \quad \text{for } n = 0, \ldots, N_T - 1. \tag{2.36}$$

For each $\theta \in [0, 1]$ and sufficiently small Δt the method is convergent. The order of convergence and additional stability properties, like the step size that guarantees that numerical errors are damped as we iterate, depend on the value of θ. Popular cases are the *forward Euler* scheme for $\theta = 0$, the *Crank-Nicholson* method for $\theta = 1/2$ and the *backward Euler* scheme for $\theta = 1$, see eg. [154, Ch. 12].

2.4.2 Finite differences

In the finite difference method (FD), a PDE is approximately solved by replacing the derivatives appearing in the differential equation by difference quotients. It is one of the most popular methods for solving PDEs arising from subsurface flow problems. The FD method constitutes one of the oldest methods for solving ordinary and partial differential equations. As a consequence, it is supported by an extensive body of literature and benchmarks. See eg. [110, 152, 156].

A first step in many discretization techniques is to cover the domain of interest with a mesh. This mesh is then used to define the spacial location of the discrete values that should approximate the continuous unknown. These values are referred to as *degrees of freedom* in the literature. Finite difference schemes employ orthogonal grids, which for simplicity reasons are

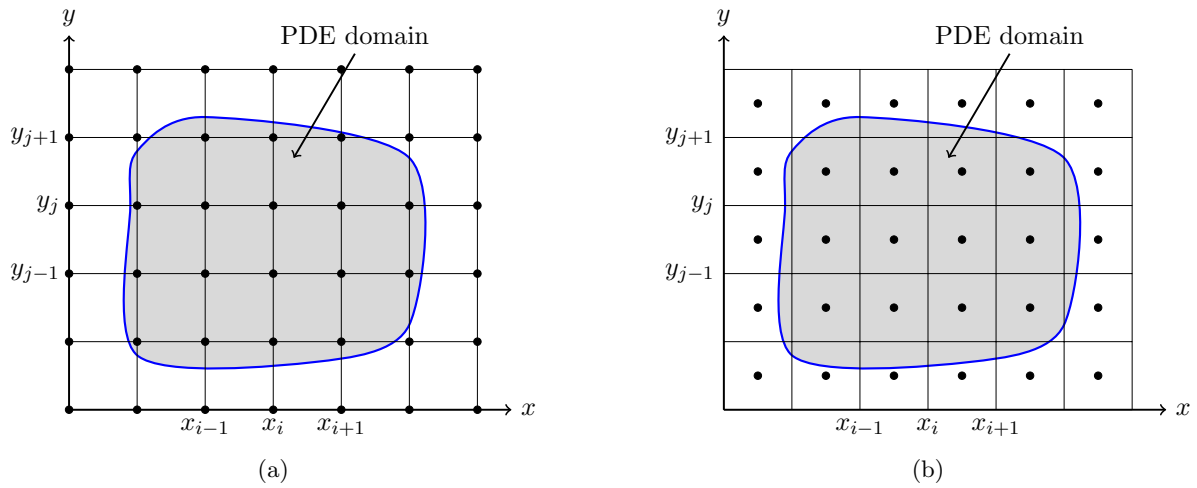


Figure 2.3: Example of a grid-centered (a) and a cell-centered (b) finite difference discretization. The black dots represent the degrees of freedom.

assumed as equally spaced in each coordinate direction. See Figure 2.3 for a two dimensional example.

The actual location of the degrees of freedom is in general problem dependent. Grid-centered and cell-centered FD methods place the degrees of freedom in the mesh nodes and the face of a mesh cell respectively; see Figure 2.3. The cell-centered approach may be constructed by considering mass conservation in every cell of the mesh, a reason for which it is widely used in the modeling community [17, Ch. 8]. It can also be regarded as a special case of the finite volume method. See the end of Section 2.4.3. For problems involving multiple variables (e.g., velocity and pressure) placing all degrees of freedom in the same spatial locations could lead to nonphysical oscillatory patterns in the solution. The staggered grid variant places different unknown values at different locations in order to avoid this situation [78, Ch. 3].

We would like to introduce some concepts that will be useful in Chapter 3 with the following example. Let $\Omega = (a, b)$ be a domain in $\mathbb{R}$ and $T > 0$. For given functions $u_0, k : \Omega \to \mathbb{R}$ consider the problem to find $u : [0, T] \times \Omega \to \mathbb{R}$ satisfying

$$\frac{\partial u(t,x)}{\partial t} - \frac{\partial}{\partial x}\left(k(x)\frac{\partial u(t,x)}{\partial x}\right) = 0 \quad \text{in } [0,T]\times\Omega, \tag{2.37a}$$

$$u(0,x) = u_0(x) \quad \text{in } \Omega, \tag{2.37b}$$

$$k(x)\frac{\partial u(t,x)}{\partial x} = 0 \quad \text{on } [0,T]\times\partial\Omega. \tag{2.37c}$$

Let $\Delta t := T/N_T$ and $\Delta x := (b-a)/(N+1)$ where $N_T, N \in \mathbb{N}$ are some positive numbers. We introduce uniformly spaced meshes in time and space,

$$\{t_n \mid t_n = n\Delta t, n = 0, 1, \ldots, N_T\},$$
$$\{x_{i+1/2} \mid x_{i+1/2} := a + (i+1)\Delta x, i = -1, 0, 1, \ldots, N\}.$$

and employ a cell-centered FD scheme to discretize (2.37), which means that the degree of

freedom corresponding to the mesh cell $K_i := [x_{i-1/2}, x_{i+1/2}]$ is located at

$$x_i := \frac{1}{2}\left(x_{i-1/2} + x_{i+1/2}\right). \tag{2.38}$$

Further, we use the notation $u_i^n = u(t_n, x_i)$. Then, for a given time step t_n, the second order term in (2.37a) is approximated as

$$\frac{1}{\Delta x}\left(k_{i+1/2}\frac{u_{i+1}^n - u_i^n}{\Delta x} - k_{i-1/2}\frac{u_i^n - u_{i-1}^n}{\Delta x}\right). \tag{2.39}$$

The values of the coefficient $k(x)$ in (2.39) can be approximated as $k_{i\pm1/2} = k(x_{i\pm1/2})$, the arithmetic mean

$$k_{i\pm1/2} = \frac{1}{2}\left(k(x_i) + k(x_{i\pm1/2})\right), \tag{2.40}$$

or with the harmonic mean

$$k_{i\pm1/2} = \frac{2k(x_i)k(x_{i\pm1/2})}{k(x_i) + k(x_{i\pm1/2})}. \tag{2.41}$$

For higher dimension domains, and the case of highly heterogeneous $k(x)$, the choose of the arithmetic or harmonic mean can be physically motivated [173]. Additionally, the harmonic mean offers the advantage that it automatically accommodates to zero flux boundary conditions. This property is extended in two and three dimensions because of the tensor product nature of finite difference quotients on orthogonal grids. Combining (2.39) with the θ-method (2.36), a complete discretization scheme for (2.37) reads

$$\begin{aligned}\frac{u_i^{n+1} - u_i^n}{\Delta t} = {} & (1-\theta)\frac{1}{\Delta x}\left(k_{i+1/2}\frac{u_{i+1}^n - u_i^n}{\Delta x} - k_{i-1/2}\frac{u_i^n - u_{i-1}^n}{\Delta x}\right) \\ & + \theta\frac{1}{\Delta x}\left(k_{i+1/2}\frac{u_{i+1}^{n+1} - u_i^{n+1}}{\Delta x} - k_{i-1/2}\frac{u_i^{n+1} - u_{i-1}^{n+1}}{\Delta x}\right).\end{aligned} \tag{2.42}$$

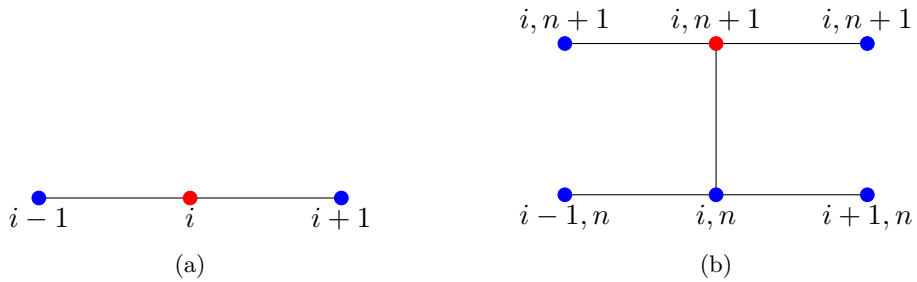
For the case of the forward Euler method, (2.42) defines an explicit iterative process to find an approximate value of the solution. The major drawback of this approach is that the following condition on the time and space steps must hold,

$$\|k(x)\|\frac{\Delta t}{(\Delta x)^2} \leq \frac{1}{2}, \tag{2.43}$$

such that the numerical error introduced at each time step does not grow exponentially. The inequality (2.43) is the CFL condition [50] for this scheme and implies that taking a small time step restricts the spacial step to be even smaller. This translates into additional computational workload, which for three dimensional problems may be prohibitively expensive. The variants with $\theta = 1$ or $\theta = 1/2$ do not have this restriction at the expense of solving a (non-linear) system of equations at each time step.

A concept frequently found in the FD theory is that of a stencil. It refers to a geometric representation of the degrees of freedom relevant for computing a numerical approximation of a derivative at a given point. For example, taking $k \equiv 1$ and $u(t, x) = u(x)$, equation (2.39) gives an approximation of the second order derivative

$$\left.\frac{\partial^2 u(x)}{\partial^2 x}\right|_{x=x_i} \approx \frac{u_{i-1} - 2u_i + u_{i+1}}{(\Delta x)^2}. \tag{2.44}$$



(a) (b)

Figure 2.4: Left, 3 point stencil for approximating the second order derivative. Right, Crank-Nicholson stencil corresponding to (2.42) with $\theta = 1/2$.

The corresponding stencil consists of the point itself and its two neighbors. Time stepping information is frequently included in the stencil as well. See Figure 2.4.

The difference quotients used to approximate a given derivative are normally taken from truncated Taylor expansions. In particular, the FD method requires local smoothness of the target solution. Hence, dealing with discontinuous coefficients becomes an issue. Another drawback from the FD method lies in the use of orthogonal meshes, since these make it difficult to geometrically conform to domains with a complex boundary.

2.4.3 Finite volume method

The finite volume method (FV) is a discretization technique especially well suited for PDEs arising from conservation laws. It its based on the idea of splitting the spatial domain into control volumes (cells) and enforcing a local version of the integral form of the PDE over each cell. Compared to the FD method, where the PDE is solved point-wise, the FV method represents the solution as cell averages. These values are modified at each time step due to fluxes through the cell's boundary. Then, a special reconstruction approximates the fluxes based on the available cell averages.

A major advantage of the FV over FD method is that it does not require a structured mesh. Consequently, flexibility to deal with complex geometries is gained. It also demands less regularity of the target solution and by construction it is locally conservative. The latter property has made the FV method a particularly attractive scheme for modeling problems in which the governing equations are derived from mass balance considerations. The method also builds upon an extensive theoretical framework [111, 67, 112].

We will use (2.37) to explain the main ideas behind the FV method. To this end, we define the flux $F(t,x) := -k(x)\frac{\partial}{\partial x}u(t,x)$ and rewrite (2.37a) as

$$\frac{\partial}{\partial t}u(t,x) + \frac{\partial}{\partial x}F(t,x) = 0. \tag{2.45}$$

Then, integrating (2.45) over a cell $K_i := [x_{i-1/2}, x_{i+1/2}]$ gives

$$\int_{K_i} \frac{\partial}{\partial t}u(t,x)\,\mathrm{d}x + F(t,x_{i+1/2}) - F(t,x_{i-1/2}) = 0. \tag{2.46}$$

Integration with respect to t over a time interval $[t_n, t_{n+1}]$ and Fubini's theorem yields

$$\int_{K_i} (u(t_{n+1},x) - u(t_n,x))\,\mathrm{d}x + \int_{t_n}^{t_{n+1}} F(t,x_{i+1/2})\,\mathrm{d}t - \int_{t_n}^{t_{n+1}} F(t,x_{i-1/2})\,\mathrm{d}t = 0. \tag{2.47}$$

Dividing by the length of K_i, which we will denote by $m(K_i)$ to emphasize the fact that cells may have different sizes, the last equation can be rearranged as

$$\begin{aligned}\frac{1}{m(K_i)}\int_{K_i} u(t_{n+1},x)\,\mathrm{d}x = \frac{1}{m(K_i)}\int_{K_i} u(t_n,x)\,\mathrm{d}x \quad - \\ \frac{1}{m(K_i)}\left\{\int_{t_n}^{t_{n+1}} F(t,x_{i+1/2})\,\mathrm{d}t - \int_{t_n}^{t_{n+1}} F(t,x_{i-1/2})\,\mathrm{d}t\right\}.\end{aligned} \tag{2.48}$$

The integral

$$\frac{1}{m(K_i)}\int_{K_i} u(t_n,x)\,\mathrm{d}x \tag{2.49}$$

represents the cell average of the conserved quantity at time step t_n. Hence, equation (2.48) gives the correct formula to update this average at a given cell when advancing a time step of size Δt, taking into account the flux over the cell's boundary. Because in general, none of the integrals in (2.48) can be computed exactly. The idea of the FV method is to approximate them in a way that in the resulting discrete scheme mimics (2.48). Hence, if Q_i^n approximates (2.49), for example by quadrature, then a FV scheme for solving (2.37a) has the form

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{m(K_i)}\left\{\mathcal{F}_{i+1/2}^n - \mathcal{F}_{i-1/2}^n\right\}, \tag{2.50}$$

where $\mathcal{F}_{i+1/2}^n$ is an approximation of the flux along the cell's boundary at $x = x_{i+1/2}$. Such approximation is in general problem dependent and ideally should take into account the direction and speed at which the information is propagated. See [112, Ch. 4].

Note that the cell's size did not play an active role in the derivation of the method. Consequently, the cells may have different sizes. This also applies for a higher dimensional construction where there is also more flexibility concerning the shape of the cells. The integral formulation relaxes the regularity requirements and even discontinuous coefficients may be treated with the method. The choice of the flux approximation can be generalized to higher dimensions in a systematical way [158]. To finish this section we would like to note that choosing $Q_i^n = u_i^n$ and

$$\mathcal{F}_{i+1/2}^n = (1-\theta)\frac{1}{\Delta x}\left(k_{i+1/2}\frac{u_{i+1}^n - u_i^n}{\Delta x}\right) + \theta\frac{1}{\Delta x}\left(k_{i+1/2}\frac{u_{i+1}^{n+1} - u_i^{n+1}}{\Delta x}\right) \tag{2.51}$$

reduces (2.50) to the cell-centered finite difference scheme (2.42). In particular, if (2.50) defines an explicit scheme the cell's length is also restricted by the CFL condition.

2.4.4 Finite element method

In the finite element (FE) method the PDE is solved in weak form, which is an extended formulation that allows solutions whose derivatives may fail to exist in the classical sense. The solution is sought in a Hilbert space of functions V, defined on the domain Ω where the PDE is posed. For the case of elliptic PDEs, the weak formulation takes the following abstract form: Find $u \in V$ such that

$$a(u,v) = L(v) \qquad \text{for all } v \in V, \tag{2.52}$$

where $a : V \times V \to \mathbb{R}$ is a continuous and symmetric bilinear form on $V \times V$ and $L : V \to \mathbb{R}$ a continuous linear functional on V. The problem (2.52) arises when trying to find the minimum

of the linear functional $J(v) := \frac{1}{2}a(v,v) - L(v)$ over V. The FE method can be regarded as a technique to build finite dimensional subspaces V_m of V, called finite element spaces, such that instead of (2.52) the following problem is solved: Find $u_m \in V_m$ such that

$$a(u_m, v_m) = L(v_m) \qquad \text{for all } v_m \in V_m, \tag{2.53}$$

with the hope that u_m correctly approximates the real solution u of (2.52). To this end, the domain Ω is partitioned into simple subdomains called elements and the functions on V are approximated by polynomials defined locally on each element. By imposing matching conditions on the interfaces between the elements, these local approximations are pasted together in such a manner that a function in V is globally represented using piece-wise continuous polynomials. Like the previously reviewed schemes, the FE method is also supported by a rich amount of literature [150, 25, 28, 63].

We will apply the ideas of the FE method to (2.37) to give an example of how this scheme is used for parabolic PDEs. First, we will suppose that (2.37) was already discretized in time with an explicit Euler method. Hence, our starting point is

$$u^{n+1}(x) = u^n(x) + \Delta t \frac{\partial}{\partial x}\left(k(x)\frac{\partial u^n(x)}{\partial x}\right), \tag{2.54}$$

where we have employed the notation $u^n(\,\cdot\,) = u(t_n, \cdot\,)$. Assuming that the solution at time step $t = t_n$ has been already computed, a weak formulation is obtained by multiplying (2.54) with a smooth function $v(x)$, referred to in the FE community as test function, and integrating over Ω,

$$\int_\Omega u^{n+1}(x)v(x)\,\mathrm{d}x = \int_\Omega u^n(x)v(x)\,\mathrm{d}x - \Delta t \int_\Omega \frac{\partial}{\partial x}\left(k(x)\frac{\partial u^n(x)}{\partial x}\right) v(x)\,\mathrm{d}x. \tag{2.55}$$

Using the product rule (Gauß divergence theorem) and the boundary condition yields the following problem: Find $u^{n+1}(x) \in V$ such that

$$\int_\Omega u^{n+1}(x)v(x)\,\mathrm{d}x = \int_\Omega u^n(x)v(x)\,\mathrm{d}x + \Delta t \int_\Omega k(x)\frac{\partial}{\partial x}u^n(x)\frac{\partial}{\partial x}v(x)\,\mathrm{d}x \qquad \text{for all } v \in V. \tag{2.56}$$

For this example a good choice is to search for solutions in $V = H^1(\Omega)$, which is the standard space of functions whose first order derivatives are square integrable in Ω. Introducing the bilinear form

$$a(u,v) = (u,v)_{0,\Omega} := \int_\Omega u(x)v(x)\,\mathrm{d}x \tag{2.57}$$

and the linear functional

$$L_n(v) = (u^n, v)_{0,\Omega} + \Delta t \left(k\frac{\partial}{\partial x}u^n, \frac{\partial}{\partial x}v\right)_{0,\Omega}, \tag{2.58}$$

we can write (2.56) in the form (2.52).

The FE method will then provide a finite dimensional subspace V_m of V to instead solve the problem: Find $u_m^{n+1}(x) \in V_m$ such that

$$\int_\Omega u_m^{n+1}(x)v(x)\,\mathrm{d}x = \int_\Omega u_m^n(x)v(x)\,\mathrm{d}x + \Delta t \int_\Omega k(x)\frac{\partial}{\partial x}u_m^n(x)\frac{\partial}{\partial x}v_m(x)\,\mathrm{d}x \qquad \text{for all } v_m \in V_m. \tag{2.59}$$

If $\{\phi_1(x), \dots, \phi_m(x)\}$ denotes a basis of V_m, we can express $u^n(x) = \sum_i u_i^n \phi_i(x)$ for some coefficients u_i^n to be determined. Consequently, $\frac{\partial}{\partial x} u^n(x) = \sum_i u_i^n \phi_i'(x)$. Inserting these expansions in (2.56) with $v = \phi_j$ leads to solve the following system of equations,

$$\sum_{i=1}^{m} u_i^{n+1} \int_\Omega \phi_i(x)\phi_j(x)\,\mathrm{d}x = \sum_{i=1}^{m} u_i^n \int_\Omega \left[\phi_i(x)\phi_j(x) + \Delta t k(x)\phi_i'(x)\phi_j'(x)\right]\,\mathrm{d}x, \quad j = 1, \dots, m, \tag{2.60}$$

which is the classical Galerkin scheme [25]. Other choices for the functions v are possible; see the above mentioned literature. With help of the matrices $\boldsymbol{M} := (\boldsymbol{M})_{ij} = (\phi_i, \phi_j)_{0,\Omega}$ and $\boldsymbol{A} := (\boldsymbol{A})_{ij} = -(k\phi_i', \phi_j')_{0,\Omega}$, (2.60) can be written in the more compact form

$$\boldsymbol{M}\boldsymbol{u}^{n+1} = \boldsymbol{M}\boldsymbol{u}^n + \Delta t \boldsymbol{A}\boldsymbol{u}^n, \tag{2.61}$$

where $\boldsymbol{u}^n = (u_1^n, \dots, u_m^n)^\top$.

The previous discussion highlights the main ideas of a FE method. It allows to employ elements variable in shape and size. This feature can be combined with high order local polynomials to improve the quality of the approximation (*hp*-adaptivity) [53]. For our particular example, we note that despite using an explicit time integration scheme, the discretization in general requires to invert a matrix at each time step. While the classical FE method is well suited for elliptic PDEs, on the other hand it is not a natural choice for problems in which information is propagated directionally, even though it can be improved with the assistance of stabilization techniques [89].

2.4.5 Mixed finite elements

For many physical problems the corresponding mathematical models translate into systems of PDEs. Due to the disparity of the unknowns appearing in such systems of equations, there is an interest in obtaining them simultaneously. Mixed finite elements (MFE) refers in general to FE approximations in which each quantity of interest is approximated as a primal variable and sought in a different finite element space. In the area of fluid flow simulation, where the corresponding PDEs are derived from mass balance laws, these finite element spaces may be constructed in such a way that the normal traces of the unknown functions are continuous between element interfaces. That is, MFE schemes are locally conservative [7]. Additionally, MFE methods are supported by a rich amount of theoretical work. See eg. [31, 24, 74].

The canonical example to motivate the use of MFE is the given by the Stokes equations, which model the motion of an incompressible viscous fluid, under the assumption that the convective and acceleration inertial forces are negligible compared to the viscous forces [14]. The Stokes system of equations is posed in terms of two variables: the pressure p and velocity u of the fluid. Thus, a MFE formulation is a natural choice. With the notation from Section 2.4.4 and introducing an additional Hilbert space W, a MFE formulation leads to the following abstract problem: Find $(u, p) \in V \times W$ such that

$$a(u, v) + b(v, p) = L(v) \quad \text{for all } v \in V, \tag{2.62a}$$

$$b(u, q) = \ell(q) \quad \text{for all } q \in W. \tag{2.62b}$$

$b : V \times W \to \mathbb{R}$ is a continuous bilinear form on $V \times W$ and $\ell : W \to \mathbb{R}$ a continuous linear functional on W. The problem (2.62) arises for example when trying to resolve the following

constrained optimization problem:

$$\text{Minimize } \left\{\frac{1}{2}a(u,u) - L(u)\right\} \quad \text{over } V \tag{2.63}$$

$$\text{subject to } b(u,q) = \ell(q) \quad \text{for all } q \in W. \tag{2.64}$$

The MFE methods deal with constructing finite element spaces of the form $V_m \subset V$, $W_s \subset W$ to solve a discrete version of (2.62): Find $(u_m, p_s) \in V_m \times W_s$ such that

$$a(u_m, v_m) + b(v_m, p_s) = L(v_m) \quad \text{for all } v_m \in V_m, \tag{2.65a}$$

$$b(u_m, q_s) = \ell(q_s) \quad \text{for all } q_s \in W_s, \tag{2.65b}$$

In general, the subspaces V_m and W_s cannot be chosen independently. A compatibility condition (LBB condition) involving the bilinear forms $a(\,\cdot\,,\,\cdot\,)$ and $b(\,\cdot\,,\,\cdot\,)$ must be fulfilled in order to guarantee the solvability of the resulting system of algebraic equations [29].

If $\{\phi_i\}_{i=1,\dots,m}$ and $\{\psi_l\}_{l=1,\dots,s}$ denote basis for V_m and W_s respectively, MFE discretization of (2.62) leads to linear systems of equations whose matrix has the following block structure,

$$\begin{bmatrix} \mathbf{A} & \mathbf{B}^\top \\ \mathbf{B} & \mathbf{0} \end{bmatrix}, \tag{2.66}$$

where $\mathbf{A} := (\boldsymbol{A})_{ij} = a(\phi_i, \phi_j)$ and $\mathbf{B} = (\boldsymbol{B})_{li} = b(\phi_i, \psi_l)$. The matrix (2.66) is in general indefinite, meaning that additional care and considerably more computational work must be taken into account in comparison with the systems of equations obtained with other methods. We will revisit MFE in Chapter 6 in the context of a diffusion equation.

2.5 Discretization equivalences

When applied to particular cases, some discretization methods produce the same systems of equations that we could obtain for another method. We showed an example of this situation at the end of Section 2.4.3, where a FV discretization on orthogonal grids that approximates the cell averages as constants, is equivalent to a FD scheme. Such equivalence results are of interest because of two reasons. First, they allow to transfer and study properties from a given method to a second that a priory does not satisfy them. And second, one may profit from implementations of a given method to build new ones at relative low cost in terms of coding.

For example, it has been shown that when applying certain quadrature rules, MFE of lower order are equivalent to cell centered FD schemes [6]. If there is an interest in extending a given implementation that builds on a FD scheme, this result suggests that the code structures may be compatible with an equivalent MFE method. Another example arises when considering the multipoint flux approximation (MPFA) method, which is an extension to FD for non-orthogonal meshes that subdivides elements into subregions that are connected with appropriate continuity criteria [1]. This method has been shown to be equivalent to a particular MFE method [93].

2.6 Parallel computing

The numerical solution of (2.28) or (2.32) is challenging because of two main reasons. The first, is the non-linearity and large variation in the equation's coefficients introduced by the

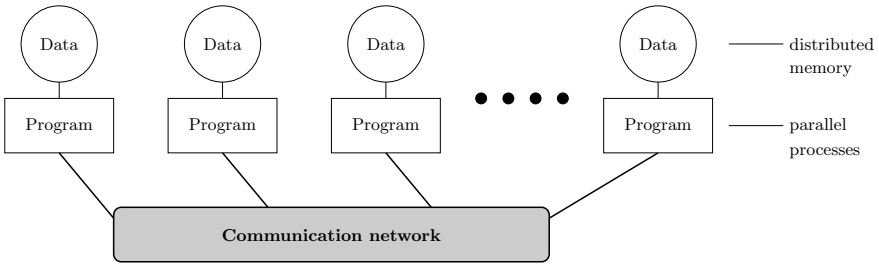


Figure 2.5: Distributed memory architecture

conductivity tensor [96]. The second is the requirement of discretizing very large temporal and spatial domains with a resolution sufficient for detailed, physics based hydrological models [104]. As a consequence, demands for computational time and memory resources for computer simulations of subsurface flow are enormous, and considerations of efficiency become prominent. One way to decrease the solution time while (almost) preserving the available algorithms is to employ parallel computing.

Parallel computing refers to the simultaneous use of multiple compute resources in order to solve a given problem. Hence, a parallel computer is a machine whose architecture is built to allow for such capability. A generally used approach to classify computer architectures was introduced by M.J. Flynn [71], and it categorizes all computers according to the number of instruction and data streams they support. The classes introduced by Flynn are

(a) Single instruction, single data (SISD): refers to a system with a single processing unit and a single data stream. Hence, there is no parallelism at all

(b) Single instruction, multiple data (SIMD): are systems which can handle only one instruction but apply it to multiple data streams simultaneously.

(c) Multiple instruction, single data (MISD): are architectures designed to handle different tasks over the same data stream.

(d) Multiple instruction, multiple data (MIMD): in such system each processing element has its own stream of instructions operating on its own data.

Modern central processing units (CPUs) belong to the latter class. Out of the previous four, MIMD is the most general class and as a consequence it might be further refined. A distinction relevant for this work and often employed in practice is to classify the MIMD machines according to how each processing unit has access to memory. Two classes of systems can be identified. The first one comprises *shared memory* systems, where every process can directly access all the memory available in the whole parallel machine. A second one gathers those MIMD systems where each process has its own local memory and communicates with other processes by sending and receiving messages that travel through a communication network. For the remainder of this document we will consider only distributed memory systems (see also Figure 2.5).

Designing code for MIMD machines usually follows the single program, multiple data model (SPMD) [52]. In this paradigm, a single program is executed on all processing units in an asynchronous manner. On distributed memory machines, SPMD programs make use of message passing to exchange data and coordinate the computations on each of the processes. We will discuss this in more detail in the upcoming section.

2.6.1 The message passage interface

In order to facilitate the implementation of parallel programs, a number of libraries was developed, eg. Zipcode[147], PVM [18], PALMACS [45] and Express [70] to cite a few. Although some of these libraries are portable, not every new system provide an implementation of all common packages. This motivated the establishment of a standard, a message passage interface (MPI) that should facilitate portability and usability of programs in a distributed memory communication environment. The standardization effort involved people from academia, industry and government laboratories and began with the Workshop on Standards for Message Passing in a Distributed Memory Environment in 1992, eventually leading to the creation of the first version of the MPI standard in 1994 [118]. The current version of the standard is MPI-3.1 released in 2015 [119].

MPI is the most widely used of the standards. In the rest of the section, we present a short summary of the basic concepts specified in MPI that are relevant for the topics developed later in this thesis.

Communicators and messages

Two core concepts in MPI are those of communicator and rank. The first one is a set of processes that can communicate with each other. The second denotes a unique number in the range $\{0, 1, \ldots n-1\}$, where n denotes the total number of processes in the communicator. A key feature in MPI is that communicators are static objects. Once created, we are not allowed to add, remove or manipulate the ordering of the processes it contains. Therefore, a rank uniquely determines a process in a given communicator. Such a feature simplifies coding in the SPMD model because it allows to use the processes' rank in a program to selectively execute the portions of the code it should be responsible for [119].

The primary goal of MPI is to provide a framework in which processes can communicate to each other so they can coordinate their activities by explicitly sending and receiving messages. Conceptually, a message consists of an envelope and data. The first should contain enough information such that the sender and the receiver of the message are uniquely identified. In MPI, the envelope comprises the source or destination rank, a tag and the communicator where the ranks are defined. The data is uniquely identified by providing its type, size and starting address in memory.

The MPI functions for sending and receiving a message are called `MPI_Send` and `MPI_Recv`, respectively. We will not discuss their prototypes in detail here, for further information we refer to [119]. Essentially, the send function takes the message envelope and data as arguments. The receive function needs the message envelope and a buffer with enough space to hold the data from a matching send.

`MPI_Send` and `MPI_Recv` are blocking operations in the sense that they will not return until the communication is complete. In the case of the send function this means that the data has actually been sent or that it has been copied to an internal (system defined) buffer. The receive function can only return when the data is fully received and copied into the buffer that was passed as argument. If the system provides no buffering or the message data is too big to fit into the system's buffer, then the send function may block until a matching receive is found.

Communication modes

MPI defines three send modes additional to the default `MPI_Send`, in which the system decides whether or not to buffer the data. These are call synchronous, buffered and ready modes. In the synchronous send mode, the send function will block until a matching receive which is ready to get the message is found. A buffered send mode employs an application defined buffer (that the user should allocate) to ensure that the send operation does not block, even if no matching receive has been posted. Lastly, a ready send mode is used when an application can guarantee that a matching receive has been posted. MPI defines only one `MPI_Recv` function that will match any of the send modes. A more detailed presentation of these communication modes can be consulted in eg. [129].

Non-blocking and collective communication

For each of the communication modes there is a corresponding non-blocking variant. In the case of the send and receive functions defined above, these are named `MPI_Isend` and `MPI_Irecv` respectively. A non-blocking send or receive only initiates the communication operation and returns immediately. At a later point, the code needs to call one of the special functions `MPI_Test`, `MPI_Probe` or `MPI_Wait`. The first one only checks if the operation has been completed and returns even if that is not the case. The second one blocks if the operation is not ready to be completed, for example, if we call a non-blocking send this function blocks if the corresponding matching receive has not been posted. Finally, the latter blocks until the operation completes. An application is not allowed to read or write to the buffer passed to the operation until one of these functions indicates that the communication has been completed. Non-blocking operations can be used in order to overlap computation and communication: one can perform communication in a background process and organize the program such that useful computations are carried out while the messages are in transit.

A collective function in MPI is defined as a routine that has to be executed in all processes of a given communicator. Their objective is to manipulate information that is shared or has to be shared among all members of the communicator. Typically, they implement operations that a single process will not be able to complete alone. An example is the `MPI_Reduce` operation, which may be used to compute the sum of all elements of a vector partitioned in chunks among the processes of the communicator. Additional examples are the functions `MPI_Bcast` and `MPI_Gather`, where the first parcels out data owned by a special process called root to every process in the communicator. The latter performs the inverse operation, in collecting data owned from every process of the communicator into a root process. Up to MPI-2, the collective operations are available in only one mode, which corresponds to the default mode of the `MPI_Send` described above [119].

3 The subsurface flow simulator ParFlow

The software library ParFlow [10, 96, 103, 104] is a complex parallel code that is used extensively for high-performance computing (HPC), specifically for the simulation of surface and subsurface flow. In this chapter we present the upstream version of ParFlow, which is in widespread use and taken as the starting point of the modifications presented later in this thesis. We discuss the numerical tools this library implements in order to perform the simulations it was designed for. Our main focus is on the mesh management.

3.1 Generalities

ParFlow is an integrated hydrology model that targets large scale groundwater flow problems. The code is mainly written in `C` and employs a modular architecture that emulates object-oriented programming. It targets HPC architectures by making use of distributed memory parallelism (MPI).

The main input file in ParFlow is a `tcl` configuration script that is loaded at runtime and contains all parameters necessary to define a problem. The code simulates variably saturated subsurface flow in heterogeneous porous media in three spacial dimensions by solving the Richards equation. Let $\Omega \subset \mathbb{R}^3$ denote the flow domain and its boundary be partitioned as $\partial\Omega = \Gamma_D \cup \Gamma_N$ with $\Gamma_D \neq \emptyset$. The form of the Richards equation implemented in ParFlow is

$$S_s S_w(\psi)\frac{\partial \psi}{\partial t} + \phi\frac{\partial\left(S_w(\psi)\right)}{\partial t} - \nabla \cdot \left(\boldsymbol{\mathcal{K}}(\boldsymbol{x}) k_r(\psi)\nabla(\psi - z)\right) = Q \quad \text{in } \Omega, \tag{3.1a}$$

$$\psi = \psi_D \quad \text{on } \Gamma_D, \tag{3.1b}$$

$$-\left(\boldsymbol{\mathcal{K}}(\boldsymbol{x}) k_r(\psi)\nabla\psi\right) \cdot \boldsymbol{n} = g_N \quad \text{on } \Gamma_N, \tag{3.1c}$$

where $\boldsymbol{n}$ is the outward pointing unit normal vector to $\partial\Omega$, ψ is the pressure-head, S_w denotes the water saturation, S_s the specific storage coefficient, ϕ the porosity of the medium, $\boldsymbol{\mathcal{K}}(\boldsymbol{x})$ is the saturated hydraulic conductivity tensor (also called absolute permeability), $k_r(\psi)$ the relative permeability (water to air), z represents the elevation or depth with respect to some datum or reference point and Q any water source or sink terms. Equation (3.1) is completed with the initial condition

$$\psi = \psi^0(\boldsymbol{x}), \quad t = 0. \tag{3.2}$$

The permeability tensor $\boldsymbol{\mathcal{K}}(\boldsymbol{x})$ is assumed to be of the form

$$\boldsymbol{\mathcal{K}}(\boldsymbol{x}) = K(\boldsymbol{x}) \begin{pmatrix} k_x(\boldsymbol{x}) & 0 & 0 \\ 0 & k_y(\boldsymbol{x}) & 0 \\ 0 & 0 & k_z(\boldsymbol{x}) \end{pmatrix}, \tag{3.3}$$

where K is a scalar field which can be set to a constant, read from a user-provided file or generated by a geostatistical model that allows to set correlation lengths per coordinate direction

[157]. The relative permeability $k_r(\psi)$ and the water saturation $S_w(\psi)$ can be either set to constant, polynomial, or be defined via the Van Genuchten [160] or Haverkamp-Vauclin [84] formulations.

3.2 Discretization

ParFlow employs a cell-centered finite difference method and the implicit Euler scheme for discretization in space and time, respectively. The computational mesh is given by a tensor product with N_t points in each coordinate direction, $t \in \{x, y, z\}$. Following [128], we define

$$\boldsymbol{U}^x_{i+\frac{1}{2},j,k} := (\boldsymbol{\mathcal{K}}(\boldsymbol{x})k_r(\psi))_{i+\frac{1}{2},j,k} \frac{\psi^{n+1}_{i+1,j,k} - \psi^{n+1}_{i,j,k}}{\Delta x}, \tag{3.4a}$$

$$\boldsymbol{U}^y_{i,j+\frac{1}{2},k} := (\boldsymbol{\mathcal{K}}(\boldsymbol{x})k_r(\psi))_{i,j+\frac{1}{2},k} \frac{\psi^{n+1}_{i,j+1,k} - \psi^{n+1}_{i,j,k}}{\Delta y}, \tag{3.4b}$$

$$\boldsymbol{U}^z_{i,j,k+\frac{1}{2}} := ((\boldsymbol{\mathcal{K}}(\boldsymbol{x})k_r(\psi))_{i,j,k+\frac{1}{2}} \frac{\left(\psi^{n+1}_{i,j,k+1} - z_{k+1}\right) - \left(\psi^{n+1}_{i,j,k} - z_k\right)}{\Delta z} \tag{3.4c}$$

and

$$\nabla \cdot \boldsymbol{U}^{n+1} := \frac{\boldsymbol{U}^x_{i+\frac{1}{2},j,k} - \boldsymbol{U}^x_{i-\frac{1}{2},j,k}}{\Delta x} + \frac{\boldsymbol{U}^y_{i,j+\frac{1}{2},k} - \boldsymbol{U}^y_{i,j-\frac{1}{2},k}}{\Delta y} + \frac{\boldsymbol{U}^z_{i,j,k+\frac{1}{2}} - \boldsymbol{U}^z_{i,j,k-\frac{1}{2}}}{\Delta z}. \tag{3.5}$$

Then, for a given time step Δt, discretization results in the nonlinear system of equations

$$(S_s S_w(\psi))_{i,j,k} \frac{\psi^{n+1}_{i,j,k} - \psi^n_{i,j,k}}{\Delta t} + \phi_{i,j,k} \frac{S^{n+1}_w(\psi_{i,j,k}) - S^n_w(\psi_{i,j,k})}{\Delta t} - \nabla \cdot \boldsymbol{U}^{n+1} = Q^{n+1}_{i,j,k}, \tag{3.6}$$

Hence, a zero of the following nonlinear operator should be found in each cell

$$\begin{aligned} F_{i,j,k}(\psi^{n+1}) := {} & \Delta x \Delta y \Delta z \left[(S_s S_w)_{i,j,k} \left(\psi^{n+1}_{i,j,k} - \psi^n_{i,j,k}\right) + \phi_{i,j,k} \left(S^{n+1}_w - S^n_w\right)\right] - \\ & \Delta t \Delta y \Delta z \left[\boldsymbol{U}^x_{i+\frac{1}{2},j,k} - \boldsymbol{U}^x_{i-\frac{1}{2},j,k}\right] - \Delta t \Delta x \Delta z \left[\boldsymbol{U}^y_{i,j+\frac{1}{2},k} - \boldsymbol{U}^y_{i,j-\frac{1}{2},k}\right] - \\ & \Delta t \Delta x \Delta y \left[\boldsymbol{U}^z_{i,j,k+\frac{1}{2}} - \boldsymbol{U}^z_{i,j,k-\frac{1}{2}}\right] - \Delta t \Delta x \Delta y \Delta z \, Q^{n+1}_{i,j,k}, \end{aligned} \tag{3.7}$$

where we have omitted the explicit dependency $S_w = S_w(\psi)$ to keep the notation readable. The values of $\boldsymbol{\mathcal{K}}(\boldsymbol{x})$ and $k_r(\psi)$ (ψ at the current step) at cell interfaces are obtained by harmonic average and upwinding respectively. The latter is a technique commonly used in the FV method and can be understood as an averaging process that takes into account the direction in which information flows; see [112, Sec. 4.8].

3.3 Solvers

The nonlinear problem (3.6) is addressed with a multigrid preconditioned inexact Newton-Krylov method [96]. Given a continuously differentiable function $\boldsymbol{G} : \mathbb{R}^n \to \mathbb{R}^n$, the standard Newton method to find a solution of

$$\boldsymbol{G}(\boldsymbol{x}) = 0 \tag{3.8}$$

Algorithm 3.1: Newton's method

Input : Continuous differentiable function $\boldsymbol{G}$, initial approximation $\boldsymbol{x}^{(0)}$

1 **for** $k = 0, 1, 2, \ldots$ *until convergence* **do**
2 | Solve $\boldsymbol{G}'(\boldsymbol{x}^{(k)})\boldsymbol{\delta x}^{(k)} = -\boldsymbol{G}(\boldsymbol{x}^{(k)})$
3 | Update $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \boldsymbol{\delta x}^{(k)}$
4 **end**
5 **return** $\boldsymbol{x}^{k+1}$

Algorithm 3.1: Classical Newton's method to approximate a zero of a given function.

is given by Algorithm 3.1.

The basic assumption is that (3.8) has a solution $\boldsymbol{x}^*$ where the Jacobian $\boldsymbol{G}'(\boldsymbol{x}^*)$ is invertible and Lipschitz continuous. This ensures that for $\boldsymbol{x}^{(0)}$ close enough to $\boldsymbol{x}^*$, we have second order convergence of the sequence $(\boldsymbol{x}^{(k)})_{k\in\mathbb{N}}$ to the solution $\boldsymbol{x}^*$ (see, e.g., [102, Ch. 3]). When the linear system

$$\boldsymbol{G}'(\boldsymbol{x}^{(k)})\boldsymbol{\delta x}^{(k)} = -\boldsymbol{G}(\boldsymbol{x}^{(k)}) \tag{3.9}$$

is solved by a Krylov iterative process we are using an inexact Newton-Krylov method [102]. In such a case, we have a two level algorithm, in the outer layer is the Newton method itself and in the inner layer the Krylov iterative procedure. An important feature of Krylov methods is that they only require the action of the matrix G' when applied to a vector and not the matrix itself. In the special case of (3.9), the matrix-vector product can be approximated as

$$\boldsymbol{G}'(\boldsymbol{x}^{(k)})\boldsymbol{v} \approx \frac{\boldsymbol{G}(\boldsymbol{x}^{(k)} + \epsilon\boldsymbol{v}) - \boldsymbol{G}(\boldsymbol{x}^{k})}{\epsilon}. \tag{3.10}$$

Provided that the value of ϵ is small enough, the convergence rate of the Newton's method is preserved [34]. A value for ϵ frequently employed in the literature, and introduced in [35] is given by the formula

$$\epsilon = \operatorname{sign}(\boldsymbol{x}^{(k)} \cdot \boldsymbol{v})\beta \max\left\{|\boldsymbol{x}^{(k)} \cdot \boldsymbol{v}|, \|\boldsymbol{v}\|_1\right\} / \boldsymbol{v}^\top \boldsymbol{v}, \tag{3.11}$$

where β is a relative measure of the precision of the non-linear function evaluation. A typical value found in the literature is $\sqrt{\epsilon_0}$, where ϵ_0 is the machine unit precision.

Because (3.9) is solved just approximately, we need a stopping criterion for the Krylov method that should be related to the overall progress achieved in the outer iteration. Line 3 from Algorithm 3.1 is replaced by: find $\boldsymbol{\delta x}^{(k)}$ such that

$$\|\boldsymbol{G}'(\boldsymbol{x}^{(k)})\boldsymbol{\delta x}^{(k)} + \boldsymbol{G}(\boldsymbol{x}^{(k)})\| \leq \eta\|\boldsymbol{G}(\boldsymbol{x}^{(k)})\|. \tag{3.12}$$

The constant η is known as the forcing term [58]. As any fixed value $\eta > 0$ will destroy the quadratic convergence of the Newton method, we require $\eta = \eta_k$ with $\lim_{k\to\infty} \eta_k = 0$ to get super linear and $\eta_k = O\left(\|G(\boldsymbol{x}^{(k)})\|\right)$ to recover second order convergence [59]. Finally, to ensure affine invariance (see e.g., [54, pp. 13] for a definition) we can take

$$\eta_k = O\left(\frac{\|\boldsymbol{G}(\boldsymbol{x}^{(k)})\|}{\|\boldsymbol{G}(\boldsymbol{x}^{(0)})\|}\right). \tag{3.13}$$

Algorithm 3.2: Multigrid MG($\boldsymbol{A}^h, \boldsymbol{x}^h, \boldsymbol{b}^h$)

Input : $\boldsymbol{A}^h, \boldsymbol{x}^h, \boldsymbol{b}^h$
Build the operators $\boldsymbol{I}_h^H$, $\boldsymbol{I}_H^h$ and $\boldsymbol{A}^H$
$\boldsymbol{x}^h \leftarrow$ pre-relaxation on $\boldsymbol{A}^h\boldsymbol{x}^h = \boldsymbol{b}^h$
$\boldsymbol{b}^H \leftarrow \boldsymbol{I}_h^H(\boldsymbol{b}^h - \boldsymbol{A}^h\boldsymbol{x}^h)$
if $\boldsymbol{A}^H$ *is small* **then**
  $\boldsymbol{x}^H \leftarrow (\boldsymbol{A}^H)^{-1}\boldsymbol{b}^H$ `Direct solve`
else
  for $m = 1, \ldots, \mu$ **do**
    $\boldsymbol{x}^H \leftarrow \mathrm{MG}(\boldsymbol{A}^H, \boldsymbol{x}^H, \boldsymbol{b}^H)$ `Recursive solve`
  end
end
Correct $\boldsymbol{x}^h \leftarrow \boldsymbol{x}^h + \boldsymbol{I}_H^h\boldsymbol{x}^H$
$\boldsymbol{x}^h \leftarrow$ post-relaxation on $\boldsymbol{A}^h\boldsymbol{x}^h = \boldsymbol{b}^h$
return $\boldsymbol{x}^h$

Algorithm 3.2: Recursive multigrid method with μ-cycle.

The Jacobian from the operator (3.7) is not symmetric, hence GMRES [140] is chosen as the Krylov subspace method. The robustness of Newton's method is enhanced by including a line-search backtracking procedure that allows to modify the Newton step in order to guarantee progress towards the solution at each iteration [35].

3.4 Preconditioners

Due to the size and spectral properties of the Jacobian system (3.9) corresponding to the nonlinear operator (3.7), ParFlow employs a multigrid preconditioned Krylov solver to reduce the number of iterations. The main idea of multigrid is to accelerate the convergence of a simpler iterative solver called relaxation method (e.g., Gauss-Seidel) by means of a global correction that is computed by solving a coarser problem, i.e., with less variables. Usually, the coarse problem has a similar structure as the original one and it is solved by applying the same idea with an even coarser problem. Eventually, a coarse enough problem is reached so it is feasible to compute the global correction by using a direct method. If we consider a linear system of equations

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{3.14}$$

where $\boldsymbol{A} = \boldsymbol{A}^h \in R^{n\times n}$, the unknown vector $\boldsymbol{x} = \boldsymbol{x}^h$ and the right hand side $\boldsymbol{b} = \boldsymbol{b}^h$ are defined on a given grid with mesh size h, then a multigrid method to approximate the solution of (3.14) is determined by several components: the relaxation method, an interpolation operator $\boldsymbol{I}_H^h$ that transfers vectors from coarse to fine grids, a restriction operator $\boldsymbol{I}_h^H$ that transfers vectors from fine to coarse grids and a coarse operator $\boldsymbol{A}^H$. The procedure is summarized in Algorithm 3.2. Typical values for the parameter μ are $\mu = 1$ (V-cycle) and $\mu = 2$ (W-cycle). When properly designed, multigrid methods have the property that the solver's convergence rate is independent of the size of the problem [32]. The two main choices in ParFlow are the

parallel semi-coarsening multigrid solver (SMG) [144] and the ParFlow multigrid PFMG [10]. We briefly describe them in the following.

SMG

SMG is a multigrid solver that targets linear systems coming from a finite difference, finite volume or finite element discretization of a three dimensional diffusion equation on locally rectangular grids. It introduces a red/black plane coloring and updates the approximate solution at all red planes to satisfy their equations. This is known as plane smoothing [144]. Its key component is a specially constructed interpolation operator I_H^h that accounts for the relationship between red and black plane errors after a relaxation sweep.

PFMG

PFMG is a parallel semi-coarsening multigrid solver similar to SMG with the difference that it employs point-wise smoothing.

The implementation of both preconditioners is provided by the external dependency `hypre` [155]. The PFMG preconditioner was developed inside ParFlow and then evolved into the current `hypre` implementation. The ParFlow version is still available in the code under the name MGSemi. Finally, ParFlow offers a variant of PFMG named PFMGOctree, which essentially is a memory optimized version of the first one.

3.5 Mesh management

ParFlow's computational mesh is uniform in all three dimensions. The count and the spacing of mesh points in each dimension is user defined.

Two basic concepts are used in ParFlow for the mesh management:

(a) The background, which is a global object representing a regular mesh whose size (N_x, N_y. N_z), spacing ($\Delta x, \Delta y, \Delta z$) and position (determined by an anchor point (x_0, y_0, z_0)) are fixed by the user via the `tcl` configuration script. See Figure 3.1.

(b) A subgrid, is defined in terms of the numbering of the mesh nodes. It is a refinement of the Background. See Figure 3.1. Thus, a subgrid s can be represented as a local object whose members are: the owner process (s.rank), the refinement level over the background (s.rx, s.ry, s.rz), the anchor point in the index space (s.ix, s.iy, s.iz) and number of points (s.nx, s.ny, s.nz) per coordinate direction. Specifically, a subgrid s identifies the following set of indices

$$\begin{aligned} &\{s.\mathrm{ix} + i : i = 0, 1, \ldots, s.\mathrm{nx} - 1\} \otimes \\ &\{s.\mathrm{iy} + j : j = 0, 1, \ldots, s.\mathrm{ny} - 1\} \otimes \\ &\{s.\mathrm{iz} + k : k = 0, 1, \ldots, s.\mathrm{nz} - 1\}. \end{aligned} \tag{3.15}$$

The physical location (x_i, y_j, z_k) for a degree of freedom lying in the portion of the domain

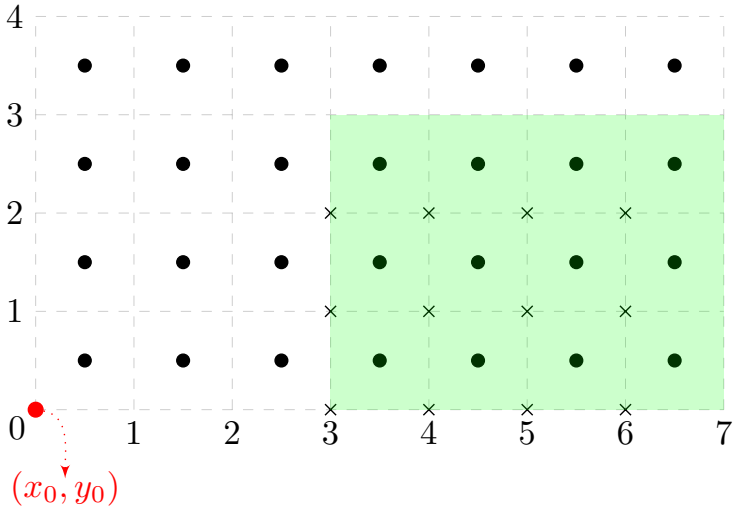


Figure 3.1: An example of a two dimensional background with anchor point (x_0, y_0). The shaded green area represents a subgrid with index anchor point $(s.\mathrm{ix}, s.\mathrm{iy}) = (3, 0)$ and $s.\mathrm{nx} = 4$ and $s.\mathrm{ny} = 3$ points in the x and y coordinate direction, respectively. The crosses represent the elements of the subgrid and the black dots degrees of freedom of the discretization. The dotted boxes emphasize that ParFlow employs a cell-centered FD discretization.

defined by a subgrid s is related to an index vector (i, j, k) by

$$x_i = x_0 + (i + 0.5)\left(\frac{\Delta x}{2^{s.\mathrm{rx}}}\right), \tag{3.16a}$$

$$y_j = y_0 + (j + 0.5)\left(\frac{\Delta y}{2^{s.\mathrm{ry}}}\right), \tag{3.16b}$$

$$z_k = z_0 + (k + 0.5)\left(\frac{\Delta z}{2^{s.\mathrm{rz}}}\right), \tag{3.16c}$$

where (x_0, y_0, z_0) and Δt, $t \in \{x, y, z\}$ denote the background's anchor point and mesh spacing, respectively. The refinement level $(\mathrm{rx}, \mathrm{ry}, \mathrm{rz})$ targets the usage of locally refined grids. The upstream version of ParFlow does not implement this feature, and in practice the refinement level is set to zero for all coordinate directions.

ParFlow's mesh is logically partitioned into non-overlapping Cartesian blocks that correspond to the previously defined subgrids. It is a local object that may contain copies of subgrids owned by different processes. The routine that allocates a new grid essentially performs a loop over all processes in the parallel machine, creates a single subgrid per iteration and appends it to a "subgrid array" that we will denote by S_{all}. The parameters defining a freshly allocated subgrid are determined by the following arithmetic.

Let P_t denote the number of process divisions in the t coordinate direction, for $t \in \{x, y, z\}$. These three values are read from the `tcl` configuration script. The total number of processes must match their product

$$P = P_x P_y P_z. \tag{3.17}$$

The number of mesh points in each direction is configured in the script as N_t and split among

P_t subgrid extents as

$$N_t = m_t \cdot P_t + l_t, \quad m_t \in \mathbb{N}, \quad l_t \in \{0, \ldots, P_t - 1\}, \tag{3.18}$$

where m_t and l_t are uniquely determined by N_t and P_t according to

$$m_t := N_t / P_t, \qquad l_t := N_t \% P_t. \tag{3.19}$$

Here, a/b denotes integer division and $a\%b$ the integer residual from dividing a by b. Both N_t and P_t are defined by the user in the `tcl` configuration script and subject to the constraint (3.17). Now, if p_t is a process number in the range $\{0, ..., P_t - 1\}$ and the triple $p = (p_x, p_y, p_z)$ determines an index into the three dimensional process grid, define

$$c(p_t) := p_t \cdot m_t + \min(p_t, l_t), \tag{3.20}$$

$$q(p_t) := \begin{cases} m_t + 1 & \text{if } p_t < l_t, \\ m_t & \text{otherwise.} \end{cases} \tag{3.21}$$

With these definitions, the subgrid corresponding to p

(a) has the index triple $(c(p_x), c(p_y), c(p_z))$ as anchor point,

(b) has $q(p_t)$ index points in the t coordinate direction,

(c) is owned by process $P_{\text{own}}(p) = P_{\text{own}}(p_x, p_y, p_z)$, where

$$P_{\text{own}}(p_x, p_y, p_z) := (p_z \cdot P_y + p_y) \cdot P_x + p_x \in \{0, \ldots, P - 1\}. \tag{3.22}$$

An example of such a distribution of subgrids is shown in Figure 3.2. From this logic it becomes clear that ParFlow's computational mesh is distributed in a parallel machine by assigning each of its subgrids to exactly one process by means of the rule (3.22). This order of subgrids is called lexicographic.

Internally, ParFlow maintains two arrays of subgrids: the first one corresponds to the above introduced S_{all}, that contains all the subgrids composing the mesh; the second one holds subgrids exclusively owned by the current process. We will denote the later as S_{loc}. From our previous discussion we conclude in the upstream version of ParFlow S_{loc} contains a single subgrid.

ParFlow's subgrids are meant to store the mesh metadata only. Such metadata allows the code to distribute the numerical data (pressure fields, saturation, etc.) in a parallel machine. The most important units of numerical information are the vectors and matrices. In analogy to the computational mesh, vectors and matrices in ParFlow are distributed into subvectors and submatrices. There is a one-to-one correspondence between each of them and the subgrids composing the mesh. Consequently, they are inherently distributed in parallel via the same rule (3.22). In order to consistently operate on subvectors and submatrices, the code requires to establish additional information to exchange data between processes. How exactly this is done is the objective of the next section.

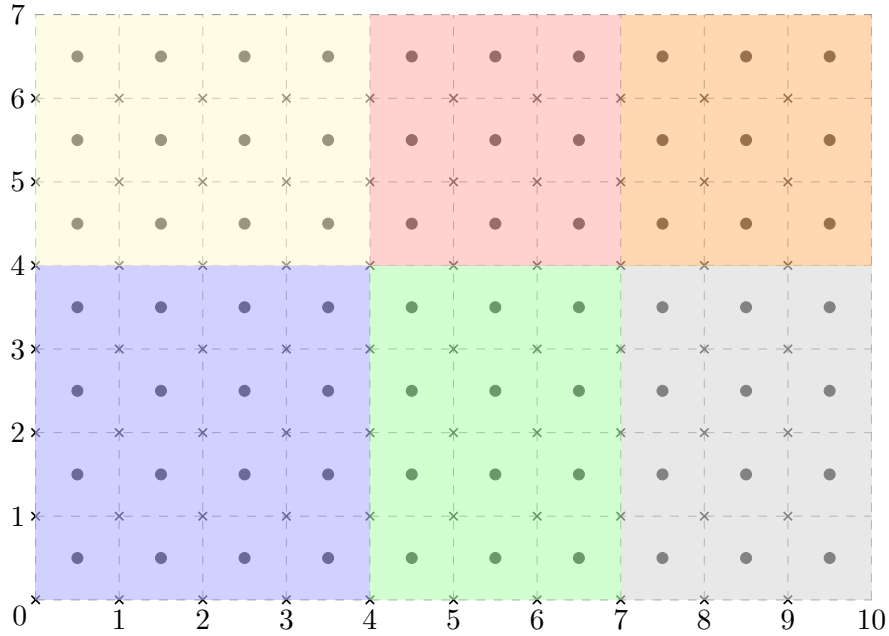


Figure 3.2: Example of a ParFlow mesh with $N_x = 10$ and $N_y = 7$. We take $N_z = 1$ to create a two dimensional mesh, thus the value $P_z = 1$ is implicit. The number of processes is $P_x = 3$, $P_y = 2$, $P = 6$. Each shaded box is a subgrid whose color symbolizes its assignment to a specific process. We use the same convention for the crosses and black dots as in Figure 3.1.

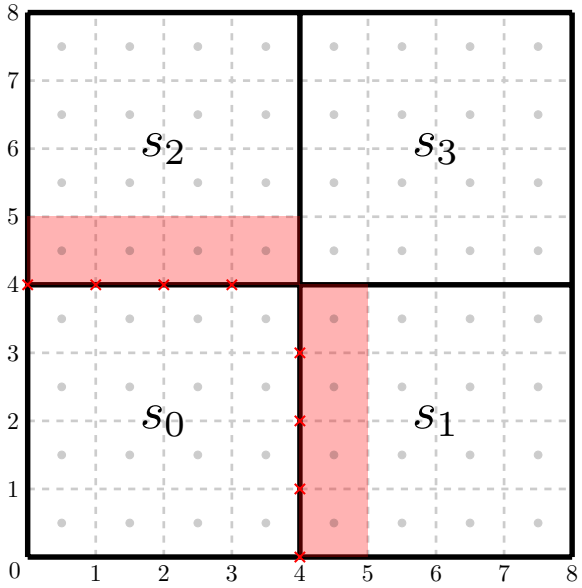


Figure 3.3: A two dimensional grid composed of four subgrids. The areas shaded red give a representation of the patch array computed by Algorithm 3.3 when applied to the lower left subgrid s_0 and the case of a stencil in which each degree of freedom requires information of the adjacent face neighbor. Because the red areas are contained in subgrids s_1 and s_2, the latter two will be included in s_0's neighbors array by Algorithm 3.4.

3.6 Compute and communication packages

At each time step of a simulation, a subset of degrees of freedom close to the boundary of a process subgrid couples to degrees of freedom lying on a foreign process (subgrid) to compute and update to its values. ParFlow denotes this subset as the "dependent region." The dependent region is characterized by the stencil chosen for discretization.

Given a stencil, ParFlow automatically determines the dependent region. For each local subgrid s, special routines loop over all subgrids in the mesh and check which of those are direct neighbors of s with respect to the processes' partition. Specifically, if we denote by $\mathrm{sten}(s)$ the action of a given stencil on the subgrid s, the code constructs a "patch array" whose elements are the non-empty set differences $\mathrm{sten}(s) - s := \{v \in \mathrm{sten}(s) \mid v \notin s\}$; see Figure 3.3. Then, ParFlow inspects which subgrids in the mesh have a non-empty intersection with the cells of this patch array and adds them to a "neighbors array." The whole process is summarized in Algorithm 3.3 and Algorithm 3.4.

Once the neighbors array S_{neigh} is determined, the dependent region (also internally identified as "send region") of a given subgrid s is the set $\{s \cap \mathrm{sten}(t) \mid t \in S_{\mathrm{neigh}}\}$; see Algorithm 3.5, the "receive region" is defined by $\{\mathrm{sten}(s) \cap t \mid t \in S_{\mathrm{neigh}}\}$, and finally the "independent region" is described as the complement of the dependent region. For a given stencil, ParFlow refers to the previously defined sets as a compute package. With the data provided by a compute package, ParFlow is able to determine the source and destination (i.e., sender and receiver) processes and the degrees of freedom relevant to the MPI messages required to perform vector and matrix updates. We will refer to this information as the MPI envelope. ParFlow implements data

Algorithm 3.3: `GetPatches`

```
Input : SubgridArray S, stencil sten
foreach s ∈ S do
    t ← sten(s) − s                          difference as sets of indices
    if t ≠ ∅ then
        Append t to S_patch
    end
end
return S_patch
```

Algorithm 3.3: Determine the patch array with respect to a given stencil.

Algorithm 3.4: `GetNeighbors`

```
Input : SubgridArray S_loc, SubgridArray S_all, Stencil sten
S_patch ← GetPatches(S_loc, sten)
foreach s ∈ S_all do
    foreach ℓ ∈ S_patch do
        if s ∩ ℓ ≠ ∅ then
            Append s to S_neigh
            break                            break loop over S_patch
        end
    end
end
return S_neigh
```

Algorithm 3.4: Determine the neighboring subgrids of the local partition with respect to a given stencil.

Algorithm 3.5: `GetSendRegion`

```
Input : SubgridArray S_loc, SubgridArray S_all, Stencil sten
S_neigh ← GetNeighbors(S_loc, S_all, sten)
foreach s ∈ S_loc do
    foreach t ∈ S_neigh do
        Append s ∩ sten(t) to S_send
    end
end
return S_send
```

Algorithm 3.5: Determine the send region with respect to a given stencil.

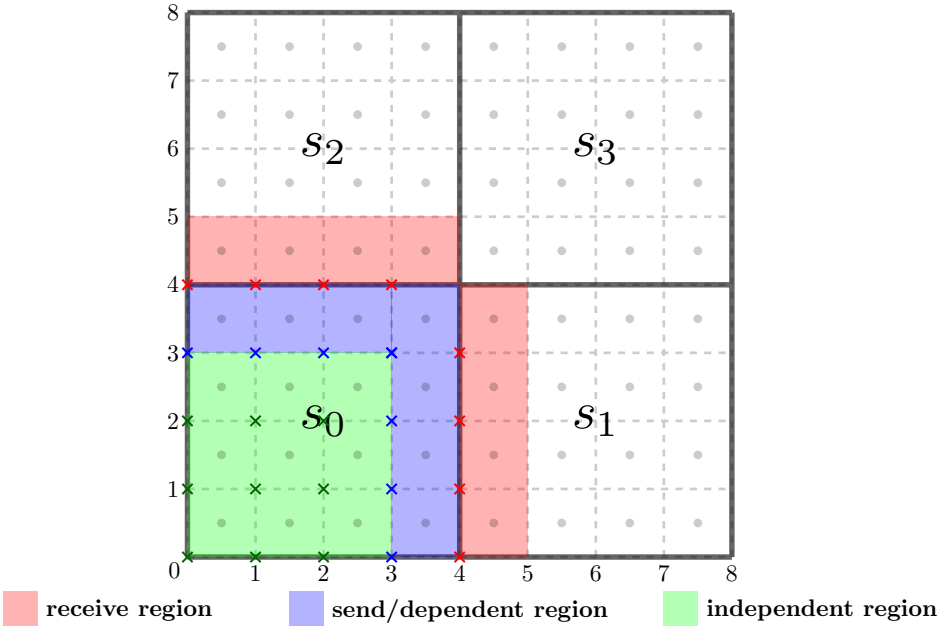


Figure 3.4: A two dimensional grid composed of four subgrids. Graphical representation of the members of a compute package from the perspective of the lower left subgrid s_0 and the case of a stencil in which each degree of freedom requires information of the adjacent face neighbor.

exchange between two neighboring subgrids via the use of a ghost layer, i.e., an additional strip of artificial degrees of freedom along the edges of each subgrid. Hence, data transferred via MPI messages is read from a subgrid of the sender and written into the ghost layer of the receiver. The extent of the ghost layer, i.e., the size of the strip in units of degrees of freedom, is also defined by the stencil.

3.7 Discussion

The subgrid approach used to exploit distributed parallelism has made it possible to efficiently execute the code for machines up to 16k processes [104]. Part of this success is due to the lexicographic ordering principle of the subgrids, which has a significant impact on how the information in the MPI envelope is computed. Each subgrid s, as a data structure, stores its coordinates in the process grid (p_x, p_y, p_z). As a consequence, the processes owning the neighbor subgrids to S are located with simple arithmetic. For example, the top neighbor of s is owned by the process $P_{\text{own}}(p_x, p_y, p_z + 1)$, where P_{own} was defined on (3.22). The fact that there is exactly one subgrid per process implies that the process number provides sufficient information to uniquely identify the sender and receiver of MPI messages.

Nevertheless, a downside of this principle is that the number of processes determines the size of the subgrids. Using few processes requires to use a few large subgrids, while using many processes makes the subgrids fairly small. Furthermore, imagining to allow for multiple subgrids per process, the lexicographic ordering will place them in a row along the x-axis, leading to an elongated and thin shape of a process' domain that has a large surface-to-volume ratio, and

thus prompts a larger than optimal message size.

As discussed at the end of Section 3.5, in order to organize the storage of subgrids, each process holds two arrays subgrids: S_{all} and S_{loc}. The first one stores pointers to the metadata (such as coordinates in the process grid, position and resolution) of all subgrids composing the grid, and the second one stores this information only for the subgrids owned by the current process (which is always one in the upstream version). This implies two main problems: First, the mesh metadata is replicated in every process inside of S_{all}, which leads to a memory usage proportional to the total number of processes P, on every process. In practice, this disallows runs with more than 32k processes. Secondly, we have identified loops which run over the elements of S_{all} (e.g., Algorithm 3.4), which means that the amount of work done by a process is also proportional to P. In the upcoming chapter we will introduce our approach to solve these two issues by delegating ParFlow's mesh management to a specialized library implementing state-of-the-art mesh partition algorithms.

4 A modified version of ParFlow

This chapter is based on the publication [38]. We have carried out minor editions in order to fit into the general notation of this document without changing its core content.

In the previous chapter we summarized the state of the upstream version of the ParFlow code and how the parallelization of its computational mesh enables but also limits parallel scalability. In the present chapter we propose to reorganize this subsystem using fast mesh partitioning algorithms provided by the parallel adaptive mesh refinement library `p4est` [44, 94]. We realize this in a minimally invasive manner by modifying selected parts of the code to reinterpret the existing mesh data structures. In this way, we obtain a modified version of ParFlow that is backwards compatible with the upstream one. We test our work by computing an approximate solution of the Richards equation with both versions and directly comparing the results. Finally, we evaluate the scaling performance of the modified version of ParFlow, demonstrating good weak and strong scaling up to 458K processes, and exercise an example application at large scale.

4.1 High performance computing essentials

High performance computing (HPC) refers to the use of aggregate computing power to target data/compute intensive assignments that are impractical or even impossible to address with standard machines. A clear computational trend is that such computational power is provided by parallel computing [122]. Hence, we consider a parallel computer that implements the message passage interface (MPI) standard. It consists of multiple physical compute nodes connected by a network. Each node has access to the memory physically located in that node, thus we speak of distributed memory and distributed parallelization. A node has multiple central processing units (CPUs), consisting of one or more CPU cores each, with each core running one or more processes or threads. For the purpose of this thesis, we will use the terms process, CPU core, and CPU interchangeably, really referring to one MPI process as the atomic unit of parallelization.

Parallel computing is employed because of two main reasons. First, the increase in the computational power provided by adding more nodes in principle allows a code to be executed faster because more floating points operations per time can be performed. Second, the inclusion of more nodes enlarges the amount of available memory. This opens the possibility of executing larger simulations as the number of nodes increases. A measure of how much a particular application benefits from adding more processes is the *speedup*, which is defined as the ratio between the execution time on a parallel and a single process system. Theoretical upper limits to the speedup are provided by Amdahl's [4] and Gustafson's [81] laws. If a fraction $f \in (0, 1)$ of a given code is perfectly parallelizable while the remaining fraction $(1 - f)$ is completely

sequential, then the speedup according to Amdahl's law is given by

$$S_{\text{Amdahl}} := \frac{1}{(1-f) + \frac{f}{P}}, \tag{4.1}$$

where P denotes the number of processes. Equation (4.1) is derived under the assumption of a fixed work load. On the other hand, Gustafson argued that instead of a fixed workload, usually more processes are added to solve larger problems. Hence, he assumed that execution time remains constant by running the code using p processes. The speedup according to Gustafson's law is given by

$$S_{\text{Gustafson}} := (1-f) + fP. \tag{4.2}$$

Equations (4.1) and (4.2) are the most basic models to estimate the speedup, and may be extended to account for more general assumptions; see eg. [99, 87].

The ideal hypothesis of parallel computing is that subdividing a task fairly among several processes will result in a proportional reduction of the overall runtime. In order to effectively produce such behavior, parallel codes should meet two basic requirements: maintain a balanced work-load per process and minimize process-intercommunication, both in terms of the number of messages and the message sizes. The number of messages to be sent and received depends on the exact assignment of the mesh elements to processes: Two different assignments for the same global mesh topology can lead to significantly different communication volumes. One guideline that helps bounding the communication is to make sure that each process has only a constant number of other processes to communicate with, independent of the size and shape of the mesh elements and the total number of processes. Ensuring this property usually requires to carefully define both the topology of the mesh and the assignment of elements to processes itself.

Once the communication pattern is established, i.e., it has been determined which process sends a message to which, the impact of sending and receiving the messages can be reduced by performing the communication in a background process and organizing the program such that useful computations are carried out while the messages are in transit. The MPI standard supports this design by providing routines for non-blocking communication, and most modern codes use them in one way or another to good effect.

4.2 Mesh parallelization

Like ParFlow, many PDE-based simulators are parallelized by partitioning the domain of interest into subdomains. Each subdomain is then assigned to a process, which is responsible for updating the degrees of freedom corresponding to that subdomain only. The chosen partition strategy has a profound effect on the simulator's performance. Most numerical simulations require synchronization during the lifetime of the application. Hence, ensuring a balanced work load is mandatory. A suboptimal partitioning may lead to performance losses caused by faster processes having to wait for the slower ones. This is further complicated when a simulator models physical phenomena with multiple phases that may have their own optimal load distribution. As an example, in a subsurface simulation, solving the Richards equation on the non-saturated part of the domain is computationally more demanding than in the saturated part. Consequently, the work load may not be adequately balanced by just trying to equalize the number of degrees of freedom on each process.

Another key aspect is communication overhead: Any time spent transmitting or receiving data contributes to a reduction in speedup. This suggests that the partition scheme should to some extent minimize the communication volume. Nevertheless, the communication volume depends on the communication network, which is far from trivial to judge for modern HPC systems. Additionally, for practical effects the details on this network are hidden from the user. Even when the HPC system offers some control on how the communication pattern is mapped on the communication network, defining such a mapping is a task that demands expertise with the particular architecture. A more generic approach is to minimize communication time instead. As pointed out earlier, a good strategy to mask communication time is to perform data transfers while useful computations are performed.

Mesh parallelization libraries can be classified according to the partition algorithm they employ. In a first class, the mesh is associated with an adjacency graph for which a graph partitioning algorithm is then applied. The latter aims to balance the number of vertices in each partition and simultaneously minimize the number of edges that need to be split in order to create the partition [21]. Examples of graph based partition libraries are METIS[100], ParMETIS [47], Scotch [48], and Zoltan [55].

Graph–based partitioners do not explicitly employ the geometric information provided by the coordinates of the mesh vertices. A second class of partitioners is the geometry-based partition methods, which do make use of such information. Two examples are the recursive coordinate bisection method [146] and the recursive circle bisection [77]. The first one recursively splits the graph in two parts choosing a different coordinate direction per iteration. The second one essentially maps the vertices of the mesh onto a sphere of a higher dimension, performs a partition there and then translates the result back to define a partition of the mesh. The library Zoltan also provides routines implementing these methods.

A subfamily of geometric partitioning algorithms that is relevant for this thesis comprises the methods based on space filling curves [141, 12] (SFC). A SFC is a mapping between a higher dimensional space and a one dimensional space. This map induces an ordering of the mesh elements, defined by their position on the curve. Then, the curve is partitioned into parts containing a balanced amount of elements. When the mesh connectivity information is stored in a spacial data structure like an octree, a SFC can be used to define a linear ordering of the tree's leaves and hence provides a tool to decompose it [12, Sec. 1.1].

In the case of hexahedral meshes (such as used in ParFlow) the logic of common SFCs can be formulated with higher simplicity due to the local tensor product structure. It is also well-suited to compute topological entities like face-neighbors, children, and parents of given mesh elements. Moreover, SFCs are memory efficient because for a mesh element only the coordinates of one anchor node plus its refinement level have to be stored. Using SFCs on hexahedral meshes has been demonstrated to be fast and scalable [3, 134, 94, 41].

4.3 The software library p4est

Tree based parallel adaptive mesh refinement (AMR) refers to methods in which the information about the size and position of mesh elements is maintained within an octree data structure whose storage is distributed across a parallel computer. An octree is basically a 1:8 (3D; 1:4 in 2D) tree structure that can be associated with a recursive refinement scheme where a cube (square) is subdivided into eight (four) half-size child cubes (squares). The leaves of the tree either represent the elements of the computational mesh directly, or can be used to hold other atomic

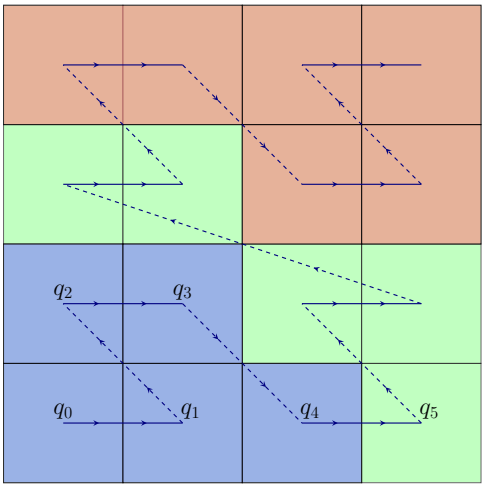


Figure 4.1: The space filling curve (SFC; zig-zag line) determines an ordering of the 16 quadrants q_i obtained by two recursive subdivisions of a square. In this example we use the SFC to partition the quadrants between three processes (color coded). We use dashed lines when two elements that are adjacent in the SFC ordering are not direct face neighbors in the domain. Since most diagonal lines connect quadrants that are still indirectly face-adjacent, the process domains are localized, which makes their surface-to-volume ratio less than that of the elongated strip domains produced by a lexicographic ordering. In fact, it is known that with this SFC each process' subdomain has at most two disconnected pieces [42].

data structures (for example one subgrid each). We will refer to the leaves of an octree/quadtree as quadrants.

The canonical domain associated with an octree is a cube (a square in 2D). When the shape of the domain is more complex, or when it is a rectangle or brick with an aspect ratio far from unity (as is the case for most regional subsurface simulations), it may be advantageous to consider a union of octrees, conveniently called a forest.

The software library `p4est` [44, 94] provides efficient algorithms that implement a self-consistent set of parallel AMR operations. This library creates and modifies a forest-of-octrees refinement structure whose storage is distributed using MPI parallelism. In `p4est` a space filling curve (SFC) determines an ordering of the quadrants that permits fast dynamic re-adaptation and repartitioning; see Figure 4.1. A `p4est` brick structure corresponds to the case in which a forest consists of multiple tree roots that are arranged to represent a rectangular Cartesian mesh.

Even when using a uniform refinement and leaving the potential for adaptivity unused, as we do in the present chapter, the space filling curve paradigm is beneficial since it allows to drop the restriction (3.17): The total number of processes does not need to match the number of patches used to split the computational mesh. Furthermore, using an SFC as opposed to a lexicographic ordering makes each process' domain more local and approximately sphere-shaped, which reduces the communication volume on average; see Figure 4.2.

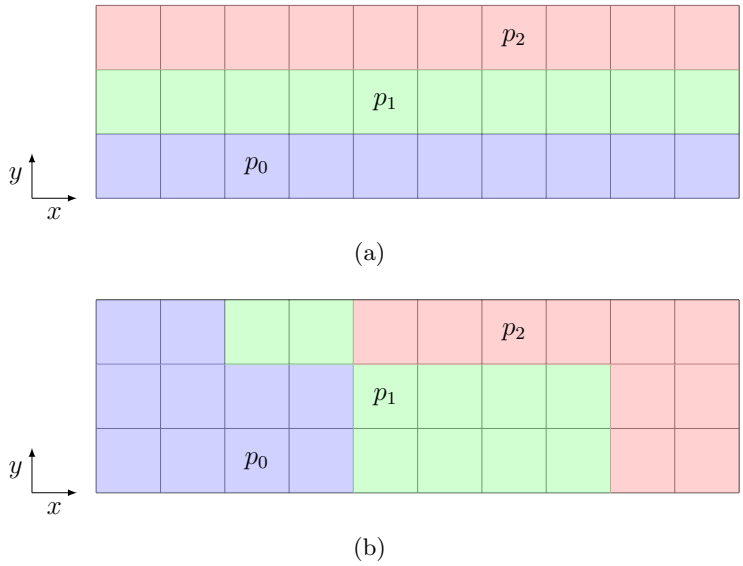


Figure 4.2: Two dimensional mesh whose cells have been distributed among three processes p_0, p_1 and p_2 (color coded). We display the distribution corresponding to a lexicographic and the SFC ordering in (a) and (b), respectively. The lexicographic ordering principle leads to an elongated and thin shape of a process' domain along the x-axis that implies larger messages in comparison to the SFC ordering. For example, if every mesh cell requires values from its adjacent neighbor, process p_0 has to send a message considerably larger to p_1 in (a) than in situation (b).

4.4 Enabling p4est as Parflow' s mesh manager

For large scale computations it is imperative that the mesh storage is strictly distributed. With the exception of a minimally thin ghost layer on every process' partition boundary, any data related to the structure of the process-local mesh should be stored on this process alone. As pointed at the end of Section 3.7, such mesh storage is not implemented in ParFlow's upstream version. This affects the runtime as well as the total memory usage: We identified loops proportional to the total number of processes P in the grid allocation phase and during the determination of the dependent region that consume roughly 40 minutes on 32k processes (and would need 1h and 20 minutes on 64k processes, and so forth). The presence of such loops limit the performance of the code when executed in machines with several hundreds of thousands of processes. Our proposed solution to enable scalability to $\mathcal{O}(10^5)$ processes and more reads as follows.

(a) Implement a strictly distributed storage of ParFlow's computational mesh.

(b) Replace loops proportional to the total number of processes with constant-size loops.

(c) Allow ParFlow to use multiple subgrids per process.

The first two items are essential to enhance the scalability of the code. We aim to proceed in a minimally invasive way, reusing most of ParFlow's mesh data structures. This principle may be called *reinterpret instead of rewrite*. Of course, establishing an optimized non-lexicographic

and distributed mesh layout is a fairly heavy task, which is why we delegate it to a special-purpose software library as described above. This removes much of the burden from the first item and lets us concentrate on the second, which requires to audit and modify the code's accesses to mesh data. The third item serves to decouple the number of subgrids allocated from P, which adds to the flexibility in the setup of simulations.

4.4.1 Rearranging the mesh layout

The space filling curve mentioned above provides a suitable encoding to implement a strictly distributed mesh storage. Computation of parallel neighborhood relations between processes is part of the `p4est` algorithm bundle and encoded in the `p4est ghost` structure. Mesh generation, distribution and computation of parallel neighborhood relations are scalable operations in `p4est`, in the sense that they have demonstrated to execute efficiently on parallel machines with up to 458k processes [94].

Delegating the mesh management from ParFlow to `p4est` by identifying each `p4est` quadrant with a ParFlow subgrid allows us to inherit the scalability of `p4est` with relatively few changes to the ParFlow code. In particular, the following features are achievable.

(i) ParFlow's mesh storage can be trimmed down to reference only the process-local subgrid(s) and their direct parallel neighbors. As a consequence, the memory occupied by mesh storage no longer grows with P, and loops over subgrids take far less time.

(ii) The rule of fixing one subgrid per process can be relaxed. This is due to the fact that `p4est` has no constraints on the number of quadrants assigned to a process, including the case of empty processes.

(iii) The computation of ParFlow's dependent region is simplified by querying neighbor data available through the `p4est ghost` structure.

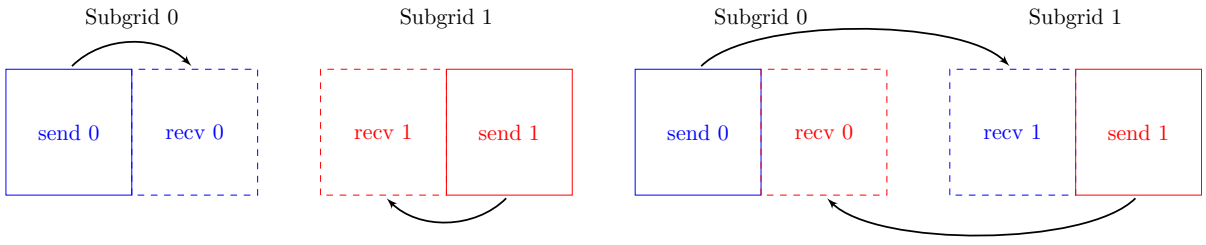
The main challenges arising while implementing the identification of a subgrid with a quadrant are the following.

(i) Parallel neighborhood relations between processes are only known to `p4est`. Such information must be correctly passed on to the numerical code in ParFlow.

(ii) The lexicographic ordering of the subgrids will be replaced by the ordering established by `p4est` via the space filling curve. This means we must modify the message passing code to compute correct neighbor process indices.

(iii) We should add support for configurations in which a process owns multiple subgrids. This will enlarge the range of parallel configurations available and enable the option to execute the code on small size machines without necessarily reducing the number of subgrids employed. Additionally, we prepare the code for a subsequent implementation of dynamic mesh adaptation, which operates by changing the number of subgrids owned by a process at runtime.

Our proposed solution to the third challenge without introducing disruptive changes in the code is to extend the existing ParFlow data structures such that subgrids owned by a common process p exchange information via `MPI` messages that p sends to itself. With this approach we can reuse most of the existing code supporting `MPI` communication and the only problem to



(a) Wrong communication

(b) Desired communication

Figure 4.3: Communication pattern enforced by a stencil requiring information from the adjacent neighbor. Both subgrids are owned by the same processor. MPI messages with the same rank as sender and receiver may result in undesired transfer of data as in (a). In our extensions we provide additional information to the message envelopes such that the send data is written into the correct subgrid (b).

address is to ensure that the communicated data is written into the proper memory location. See Figure 4.3.

The remainder of this section is dedicated to describe how to generate the ParFlow mesh using `p4est`. Essentially, we create a forest of octrees with a specifically computed number of process-local quadrants and then attach a suitably sized subgrid to each of them. Subsequently, we discuss how to obtain information on the parallel mesh layout from the `p4est ghost` interface.

The concept of a fixed number of processes per coordinate direction is not present in `p4est`. Only the total number of processes is required to compute the process partition by exploiting the properties of the space filling curve. Hence, we do not make use of the values of P_t, $t \in \{x, y, z\}$, specified by the user. Instead, we add three variables to the `tcl` reference script that provide values for m_t, the desired numbers of points in a subgrid along the t coordinate directions. Then, we rearrange the arithmetic of (3.18) to compute P_t and l_t as derived variables satisfying

$$N_t = m_t \cdot P_t + l_t, \quad P_t \in \mathbb{N}, \quad l_t \in \{0, \ldots, m_t - 1\}. \tag{4.3}$$

This construction only interprets P_t as the number of subgrids in the t direction (while the upstream version of ParFlow configures it so).

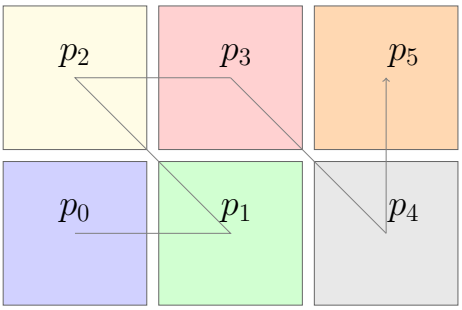
We must create a `p4est` object with $K := P_x \times P_y \times P_z$ total quadrants. To do so, we find the smallest box containing P_t quadrants in the t direction and then refine it accordingly. Let k_0 and g be defined as

$$k_0 := \max_{k \in \mathbb{N}} \left\{ 2^k \mid \gcd(P_x, P_y, P_z) \right\}, \qquad g := 2^{k_0}. \tag{4.4}$$

Thus, g is the biggest power of two dividing the greatest common divisor of P_x, P_y and P_z. Then, a `p4est` brick with dimensions

$$P_x/g, \quad P_y/g, \quad P_z/g \tag{4.5}$$

and refined k_0 times will have exactly K quadrants. A brick mesh resulting from applying these rules is shown in Figure 4.4.



Process	Subgrid Size
p_0	4×4
p_1	3×4
p_2	4×3
p_3	3×3
p_4	3×4
p_5	3×3

Figure 4.4: Left, example of a `p4est` brick distributed among 6 processes. Its dimensions are obtained by (4.5) after using $N_x = 10$, $N_y = 7$, $N_z = 1$ and $m_x = m_y = 3$, $m_z = 1$ as input values in formula (4.3). Right, the size of the subgrids attached to each quadrant according to formula (3.21).

4.4.2 Attaching subgrids of correct size

The subgrids must be a partition of the domain in the sense that their interiors are pairwise disjoint and the union of all of them covers the grid defined by the user. These conditions impose restrictions on the choice of the parameters N_t and m_t. Specifically, for each $t \in \{x, y, z\}$ we must require

$$\sum_{p_t=0}^{P_t-1} q(p_t) = N_t \tag{4.6}$$

in order to satisfy (4.3). Recall that $q(p_t)$ is defined in (3.21) as the length in grid points of the p_t'th subgrid along direction t. Condition (4.6) is checked prior to the grid allocation phase, and in case of failure quits the program with a suitable error message specifying the pair of parameters that violates it.

We inspect the bottom left corner of each of the quadrants in the `p4est` brick, which by construction are only those that are local to the process, to choose the proper size of the subgrid that should be attached to it. In the native `p4est` format, each of these corners is encoded by three 32 bit integers that we scale with the quadrant multiplier g from (4.4). This translates it into integer coordinates that match the enumeration $p_t \in \{0, \dots P_t - 1\}$ and are consistent with the rule (3.21). Thus, we can use these numbers to determine the position and dimensions of the ParFlow subgrid metadata structure that we allocate and attach to each quadrant.

4.4.3 Querying the ghost layer

We utilize the `ghost` interface of `p4est` to obtain the neighborhood information required for parallel updates. Specifically, the ghost data structure provides an array of off-process quadrants that are direct face neighbors to the local partition; we call these ghost quadrants (see Figure 4.5). We should then populate these quadrants with suitable subgrid metadata that we use to track their identification on their respective owner processes. The `p4est ghost` object provides the necessary information to do this, including the lower left corner of each ghost quadrant. In fact, we can use the enumeration $p_t \in \{0, \dots P_t - 1\}$ that serves as input to equation (3.21) to compute the dimensions of both the local and ghost subgrids. This is most

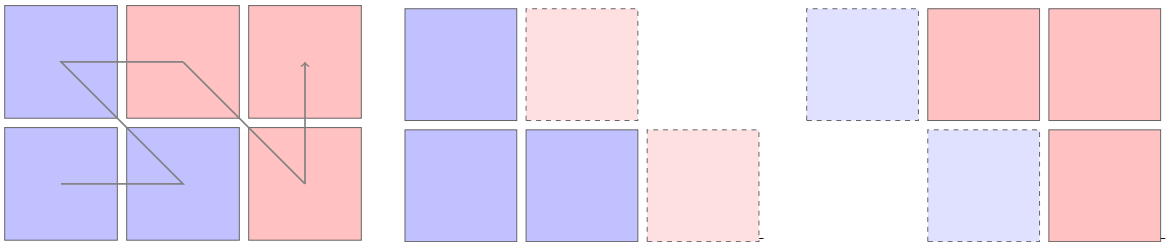


Figure 4.5: In (a) we display a `p4est` brick with six quadrants (identified with one subgrid each) distributed among two processes (color coded). In (b) and (c) we show the view of the brick from processes zero and one, respectively. The solid boxes represent process-local quadrants, for which we allocate mesh metadata and degrees of freedom storage. The dashed boxes represent quadrants in the ghost layer. For the ghost quadrants we allocate mesh metadata but no storage for the degrees of freedom. Just as the local quadrants, the ghost quadrants are traversed in the order of the space filling curve (gray arrow).

easily done by extending the loop over the local `p4est` brick quadrants described above such that it also visits the ghost quadrants.

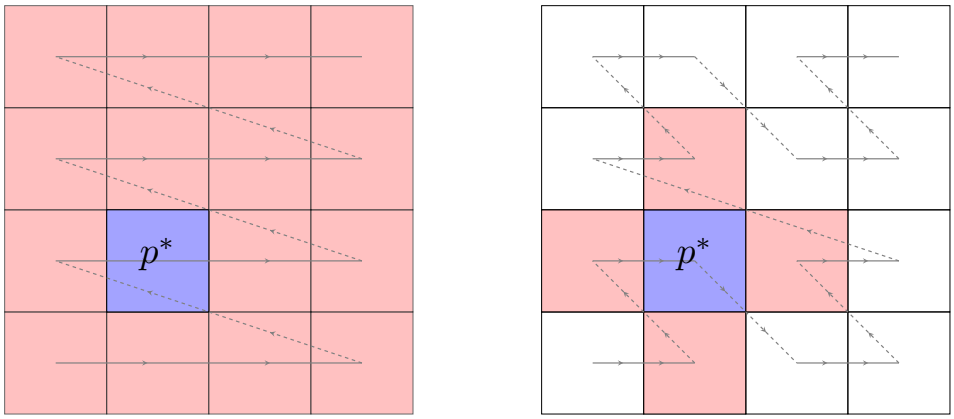
While we retain the interpretation of the subgrids array of the upstream version, storing pointers to the process-local subgrids, we remove almost all storage in the array S_{all}: The modified version stores pointers to local and ghost subgrids in it, which are roughly a $1/P$ fraction of the total number of subgrids. Thus, we avoid allocation of the metadata of all other, locally irrelevant subgrids that would be of order P; see Figure 4.6. The advantage of this method is that most of the ParFlow code does not need to be changed: When it loops over the S_{all} array, the loops will be radically shortened, but the relevant logic stays the same. This is the most significant change to enable scalability to the full size of the Juqueen supercomputer [98] (we describe these demonstrations in Section 4.5).

4.4.4 Further enhancements

We have edited ParFlow's code for reading configuration files. If `p4est` is compiled in, we activate the corresponding code at runtime depending on the value of a new configuration variable. Hence, even if compiled with `p4est`, a user has the flexibility to still use the upstream version of ParFlow on a run-by-run basis.

We have set up an automatic test suite to check consistency of our code extensions with the upstream version of ParFlow. Specifically, we use ParFlow's Richards solver to approximate the solution of a two dimensional Laplace equation for different mesh configurations. We run this solver two times, once with `p4est` activated and once without. For three dimensional problems we repeat the exercise but for the solution of the Richards equation $p = xyzt + 1$. We compare the pressure output file for both runs directly with a tolerance of 1e–8.

Additionally, we have written an alternative routine to access and distribute the information from the user-written configuration file. As before, the file is read from disk by one process and sent to all other processes. We have updated the details of this procedure, since we noticed that for high process counts (greater equal 65,536) the routine distributed incorrect data due to an



(a) (b)

Figure 4.6: Allocated subgrids in processor p^*. An owned subgrid is colored blue and red stands for foreign subgrids. Left, in the default ParFlow version each processor allocates space for all the subgrids in the grid. Right, using `p4est` we allocate space only for the off-processor subgrids that are direct neighbors of p^*. The solid / dashed line represents the ordering of the subgrids.

integer overflow, causing the program to crash during the setup phase. The modified version delegates this task to the `MPI_Bcast` routine, which works reliably and fast on the usual data size of a few kilobytes.

Many algorithms in ParFlow rely on the lexicographic ordering of the subgrids in order to retrieve neighboring information. Because our aim is to introduce minimal code modifications, we identified that only two of them require changes when switching to the SFC ordering principle. A first one is presented in Algorithm 4.1. It requires information to be propagated from bottom to top z-level subgrids. The upstream version of Algorithm 4.1 heavily relies on the assumption of one subgrid per process. The `Receive` in line 7 and the `Send` in line 13 correspond to blocking `MPI` receive and send operations, respectively. Additionally, the conditions in lines 6 and 12 are easily checked when the subgrids follow a lexicographic ordering. For example, given a subgrid s identified with the process coordinates (p_x, p_y, p_z) with $p_z > 0$, its bottom neighbor is the subgrid owned by the process $P_{\text{own}}(p_x, q_y, p_z - 1)$; see (3.22). Our modified version allows configurations in which a process can hold multiple subgrids from different levels; leaving the algorithm unchanged may lead to a deadlock situation: The code hangs indefinitely because of processes waiting simultaneously for each other; see Figure 4.7. To avoid this situation, we introduced an additional loop over the z-levels in the grid and made the `Send` in line 13 non-blocking, because with the SFC ordering, we can at most guarantee that a matching receive has been posted one level before the current one. These changes correspond with the blue shaded lines in Algorithm 4.1.

The second one is presented in Algorithm 4.2. It operates column-wise on the grid. For each subgrid in the z–dimension an array of floating point numbers is created and sent to the bottom most subgrid s_0. Then, the process owning the latter performs a reduction operation on the data and broadcasts the result back to all subgrids in s_0's column. The assumption of one subgrid per process plays a key role in the upstream implementation; the `Send` in lines 5 and 13 and the `Receive` of lines 6 and 9 correspond to blocking `MPI` operations. The lexicographic ordering

Algorithm 4.1: Vertical propagation

```
Input : Local subgrids S_loc, z-level array z_levels
for ℓ ∈ z_levels do
    for s ∈ S_loc do
        if s.iz ≠ ℓ then
            skip this subgrid
        end
        if s has a bottom neighbor s_b then
            Receive z from s_b                          Blocking receive
        else
            z ← s.iz        anchor point from s in the z direction
        end
        z ← do work on z
        if s has a top neighbor s_t then
            Send z to s_t   made non-blocking Isend in the modified version
        end
    end
    Complete Isend operations from level ℓ − 1 to ℓ with Wait
end
```

Algorithm 4.1: Propagates information from bottom to top subgrids. The blue shaded lines correspond to the changes introduced in our modified version of ParFlow. When a process holds multiple subgrids, as our modified version allows, these subgrids may be located at different heights. In order to propagate the information in the correct order, our modification first inspects subgrids at given z–level ℓ, sends data to the next level and checks whether information from a level below has to be received.

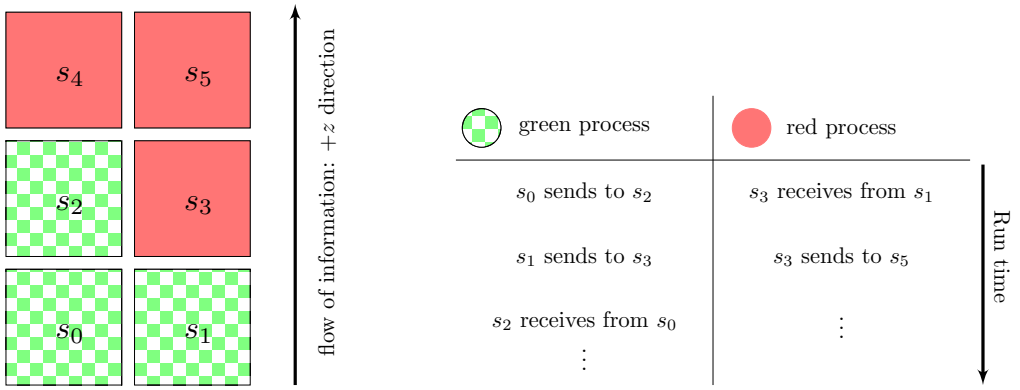


Figure 4.7: Left, we consider a grid consisting of two subgrids in the x-coordinate and three in the z-coordinate directions. The grid is distributed among two processes (color coded). Right, the sequence of messages according to Algorithm 4.1 that may lead to a deadlock situation if no further changes are introduced: The first send in the green process blocks until the corresponding receive is posted, note that the green process is also responsible for the latter receive. Simultaneously, the first receive in the red process blocks until the second send in the green process is posted, but this is not going to happen because the green process is waiting for its first send to complete.

implies that checking which process holds the lowest subgrid in a column is a straightforward operation. For example, given a subgrid s identified with the process coordinates (p_x, p_y, p_z), then the subgrid s_0 in line 2 is the subgrid owned by process $P_{\text{own}}(p_x, q_y, 0)$; see (3.22). The same logic applies to localize the subgrids in a given column. Hence, using blocking `MPI` operations is safe in the upstream version of ParFlow because it implements the loops over a subgrid's column from bottom to top and posts the receive and send operations in the appropriate order.

Implementing Algorithm 4.2 with the subgrid ordering established by `p4est` requires special care for two reasons: First, we cannot use a simple arithmetic to find s_0 given s. Second, considering that our modified version of ParFlow allows processes to hold multiple subgrids, we have to audit the blocking operations present in the algorithm to avoid deadlock situations. Let $\hat{s}$ denote the projection of s onto a given plane $z = z_0$. We recall that each subgrid s is attached to a `p4est` quadrant q. Thus, we require to identify the quadrant $\hat{q}$ holding $\hat{s}$. Given the quadrant q, it is simple to build $\hat{q}$. If additionally, we can figure out which octree $\hat{q}$ belongs to, we can use the function `p4est_quadrant_find_owner()` to retrieve the owner process of $\hat{q}$ in the parallel partition. Assuming we proceed in this way, our modifications are reflected as the red and blue shaded lines in Algorithm 4.2, they stand for removed and newly added lines, respectively. Our idea is to loop twice over the local subgrids; during the first loop we only post all the necessary send operations to the rank owning the lowest subgrid. The second loop takes care of receiving the data, performing the reduction and broadcast procedures.

While running numerical tests, we also encountered some memory issues. Essentially, memory allocation in ParFlow was increasing exponentially with the number of processes. With the help of the profiling tool Scalasca [75], we located the source of the problem in the preconditioner. Preconditioners in ParFlow are managed by the external dependency `hypre` [155], which is generally known for its scalability. Since the bug had already been resolved by the

Algorithm 4.2: Calculate elevations

```
   Input : Local subgrids S_loc
 1 foreach s ∈ S_loc do
 2 |   s_0 ←lowest subgrid in s's column
 3 |   e_a ← Compute elevation array
 4 |   if s ≠ s_0 then
 5 |   |   Send e_a to s_0                    made non-blocking in the modified version
 6 |   |   Receive reduced e_a from s_0
 7 |   else
 8 |   |   foreach s_t ∈ s_0 's column do
 9 |   |   |   Receive e_a from s_t
10 |   |   end
11 |   |   Reduce operation on received e_a's
12 |   |   foreach s_t ∈ s_0 's column do
13 |   |   |   Send reduced e_a to s_t
14 |   |   end
15 |   end
16 end
17 foreach s ∈ S_loc do
18 |   s_0 ←lowest subgrid in s's column
19 |   if s = s_0 then
20 |   |   foreach s_t ∈ s_0 's column do
21 |   |   |   Receive e_a from s_t
22 |   |   end
23 |   |   Reduce operation on received e_a's
24 |   |   foreach s_t ∈ s_0 's column do
25 |   |   |   Isend reduced e_a to s_t
26 |   |   end
27 |   else
28 |   |   Ireceive reduced e_a from s_0
29 |   end
30 end
31 Complete Ireceive operations with Wait
32 Complete Isend operations with Wait
```

Remove these lines in the modified version

Algorithm 4.2: Retrieves information from the grid, performs a column-wise reduction on it followed by a column-wise broadcast operation. It is used internally in ParFlow to enforce initial pressure conditions. Our convention is to use red for lines that have been deleted and blue for newly added ones in our modification. In order to ensure that the communication operations are executed cleanly, we implement an additional loop over the local subgrids. This takes care of posting all the send operations to the rank owning the lowest subgrid in a column. The second loop performs the column-wise reduction and broadcast operations, respectively. Note that except the `Receive`'s in line 21, all the operations are non-blocking in the modified version. In line 21 it is safe to use blocking receives because they match the sends from line 5 posted in an earlier loop.

`hypre` community, an update to the latest version was sufficient to cure the issue.

4.5 Performance results

In order to evaluate parallel performance, we follow the concepts of strong and weak scaling studies. In a strong scaling analysis, a fixed problem is solved on an increasing number of processes and the speedup is reported. For a perfectly parallelized code, the speedup should be proportional to the increase in the number of processes. In a weak scaling study, we increase the problem size and the number of cores simultaneously such that the work and problem size per core remain the same. In the optimal case, the runtime should remain constant for such a study.

Weak and strong scaling studies in this work are assessed on the massively parallel supercomputer JUQUEEN [98]. JUQUEEN is an IBM Blue Gene/Q system with 28,672 compute nodes, each with 16 GB of memory and 16 compute cores, for a total of 458,752 cores. The machine supports four way simultaneous multi-threading, though we do not make use of this capability in our studies and always run one process per core.

For each of the experiments, we collect timing information for the entire simulation. Additionally, we report timings for different components of the simulation like the solver setup, the solver itself and `p4est` wrap code executed. Concerning the wrap code, we added additional code to execute `p4est` interface functions, retrieve information from their results and propagate it to PARFLOW variables. Particularly important for our purposes are the measurements related to the solver setup: The parameters for grid allocation and the management data required for the parallel exchange of information are computed during this phase.

In the past, parallel scalability of PARFLOW has been evaluated mainly using weak scaling studies; see e.g., [103, 104, 128, 73]. In order to produce unbiased comparisons with these results, which are based on past upstream versions of PARFLOW, we keep one subgrid per process in most of our weak scaling studies, even though we have extended the modified version to use one or optionally more. We make use of this new feature in our strong scaling studies; see Figure 4.15.

4.5.1 Weak scaling studies

In this section we present results on the weak scalability of the modified version of PARFLOW. We set up a test case in which a global nonlinear problem with integrated overland flow must be solved. The test case was published previously in [116] and consists of a 3D regular topography problem in which lateral flow is driven by slopes based on sine and cosine functions. The problem has a uniform subsurface with space discretization $\Delta x = \Delta y = 1.0$ m, $\Delta z = 0.5$ m. It is initialized with a hydrostatic pressure distribution such that the top 10.0 m of the aquifer are initially unsaturated. By doubling the number of grid points N_x and N_y, the horizontal extent of the computational grid is increased by a factor of four per scaling step. The number of grid points in the vertical direction N_z remains constant per scaling step. The unit problem has dimensions $N_x = N_y = 50$ and $N_z = 40$, meaning that the problem size per process is fixed to 100,000 grid points. The problem was simulated until time $t = 10.0$ s using a uniform time step $\Delta t = 1.0$ s.

In order to offer a self-contained comparison with possible improvements in the PARFLOW model platform, we conduct the weak scaling study twice, once with the upstream version of PARFLOW and then with the modified version with `p4est` enabled. In the first case, we see that

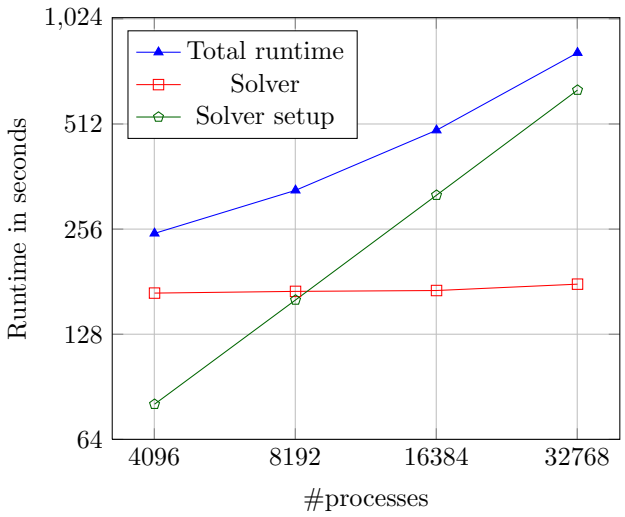


Figure 4.8: Weak scaling timing results for the upstream version of ParFlow. The total runtime grows with the number processes. Breaking it down into solver setup and solver execution shows that the former is responsible for the increase of the total runtime. The solver setup time grows from 81 to 639 seconds, which is perfectly proportional to the 8x increase in problem size.

the total runtime grows with the number of processes. As can be seen in Figure 4.8, the solver setup routine is responsible for this behavior. Its suboptimal scaling was already reported in [104], which is in line with us noticing several loops over the all-subgrids array in this part of the code, which make the runtime effectively proportional to the total number of processes. In the experiments we were not able to run the upstream code for 65,536 processes or more, which we attribute to a separate issue in the routine that reads user input from the configuration reference script, as we detail in Section 4.4.4.

Repeating the exercise with the modified version of ParFlow dramatically improves the weak scaling behavior of the solver setup; see Figures 4.9 and Figure 4.10. With `p4est` enabled, the setup time drops from over ten minutes at 32k processes to under two seconds (by a factor of over 300). Additionally, our patch to the routine that reads the user's configuration allows us to use as many as 262,144 processes for this study with nearly optimal, flat weak scaling. Our implementation of a strictly distributed storage of ParFlow's mesh also leads to a reduction in memory usage at large scale; see Figure 4.11. This is significant for computers like Juqueen, which may only offer about two hundred megabytes if the executable is large, especially in combination with multi-threading.

We also made use of this scaling study to evaluate the relative cost of using `p4est` as new mesh backend by measuring the overall timing of `p4est` related functions introduced in in ParFlow. Our results are displayed in Figure 4.12 and show that the wrap code amounts to at most 7% of the total execution time.

Additionally, we employed this test case to estimate the performance of the code in terms of the floating point operations per second (FLOP/s) and compare them to the theoretical peak of the Juqueen machine. Measurements were obtained by instrumenting the code with the Scalasca profiler, which gives access to the hardware counters from the Performance Application Interface (PAPI) [130]. In particular, the number of floating point operations from a whole run

	FLOP/s	
# Processes	Upstream	Modified
256	1.223×10^8	1.226×10^8
1024	1.192×10^8	1.194×10^8
4096	1.167×10^8	1.169×10^8

Table 4.1: Floating point operations per second (FLOP/s) for the solver component of the upstream and modified version of ParFlow, respectively. The numbers are identical up to two significant digits. Both versions of ParFlow use roughly 3.6% of the theoretical peak performance that we expect to get from a single Juqueen process, which amounts to 3.2×10^9 FLOP/s.

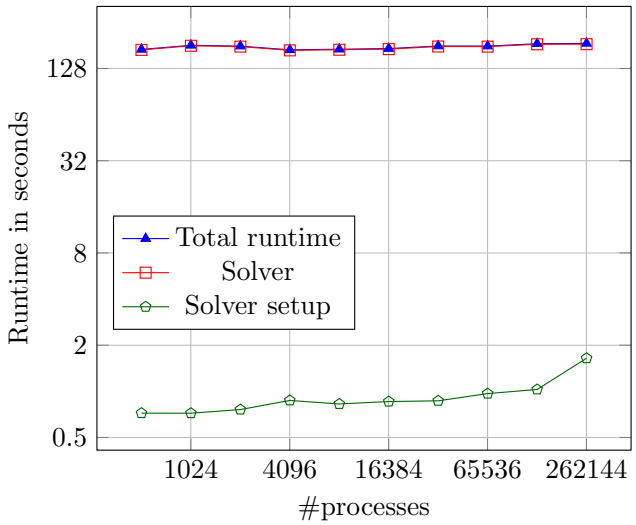


Figure 4.9: Weak scaling timing results of the modified version of ParFlow. The total and solver runtimes are nearly identical. In comparison to Figure 4.8, the solver setup executes in negligible time, ranging between 0.72 and 1.64 seconds.

have been collected and the FLOP/s estimated as a derived metric using the CUBE browser [143]. We display our results in Table 4.1.

4.5.2 Strong scaling studies

In this section, we report our results on the strong scalability of the modified version of ParFlow. The test case is the same as in the previous section, with the exception of fixing N_x and N_y while trying a range of process counts. We divide the results of this study into two categories, depending on whether we allow for multiple subgrids per process or not.

We start with one subgrid per process. In order to keep the problem size fixed when adding more processes, we adjust the subgrid dimensions properly, i.e., by decreasing the subgrid sizes in the same proportion as the number of processes increases. We run three scaling studies, the smallest of which uses a configuration with roughly 671 million grid points. In each subsequent study we increase the problem size by a factor of four. Hence, the largest problem has around

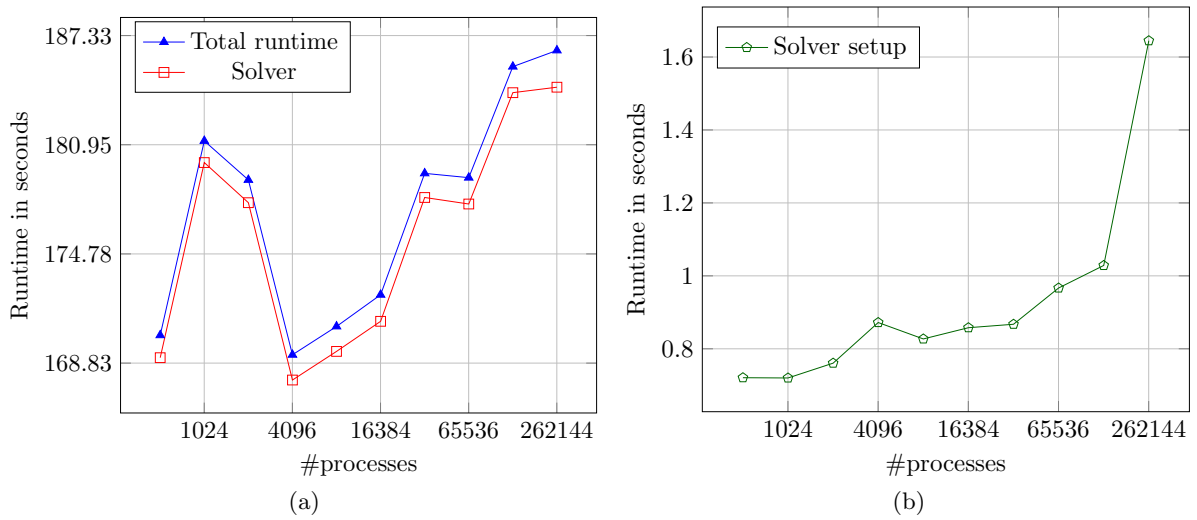


Figure 4.10: Split of weak scaling timing results from Figure 4.9. In (a) we display the total and solver runtimes that vary little in relative terms. In (b) we see that the solver setup time of the modified version of ParFlow stays under two seconds wallclock time. Please note the zoom and offset on the vertical axis.

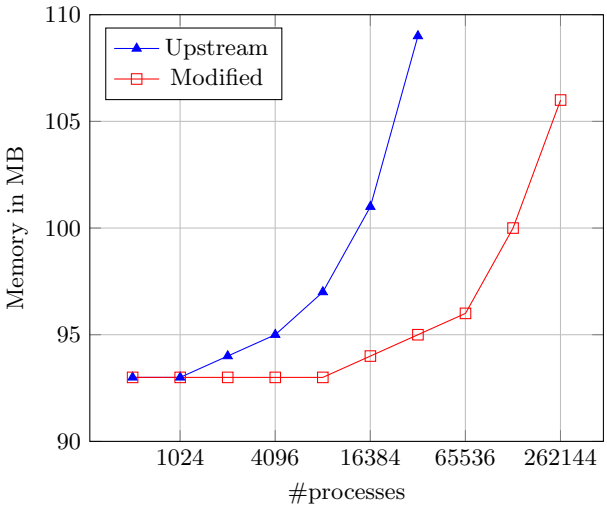


Figure 4.11: Weak scaling memory usage for the upstream and modified versions of ParFlow, respectively. Using the memory allocation information provided by the `mallinfo()` function [72], we record the maximum heap allocation per process and plot the maximum of this quantity over all processes.

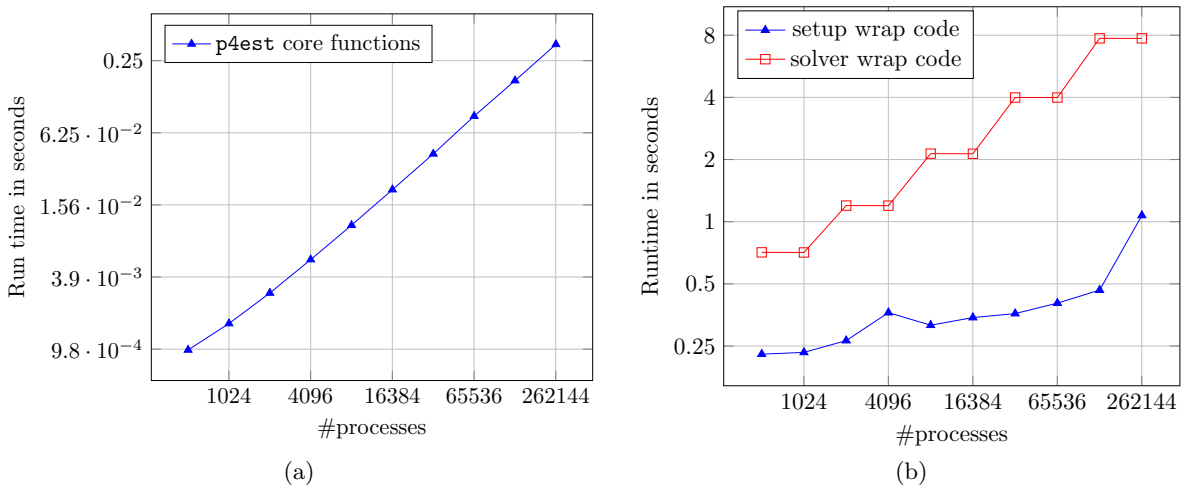


Figure 4.12: Left: overall timing for `p4est` toplevel functions used in the code. The absolute runtimes are well below one second. Right: We show timing results for the wrap code executed when `p4est` is enabled. We measure solver and setup related routines independently. Compared with the total solver time (see previous figures), the wrap code amounts to a fraction of about 7% at most.

10.7 billion grid points. In order to use the full JUQUEEN system under our self-imposed restriction of keeping one subgrid per process, while still obtaining runtimes comparable to the previous scaling studies, we tweak the subgrid dimensions and number of processes requested in such a way that the resulting problem size is as close as possible to 10.7 billion. Table 4.2 presents a summary of the main parameters defining these scaling studies; Figure 4.13 and Figure 4.14 contain our runtime results.

We have designed the modified version of PARFLOW such that it allows for configurations in which one process may hold multiple subgrids. In practice, this option provides additional flexibility when a problem with a certain size must be run, but the number of processes available depends on external factors. Using this feature, we are able to conduct a strong scaling study without changing the subgrid size with each scaling step. To illustrate this and additionally to test that the new code supporting such configurations performs well, we take the medium size problem defined in Table 4.2 and execute a classical strong scaling analysis by changing only the number of processes. We do this for three different but fixed subgrid sizes. We present our results in Figure 4.15. We observe that increasing the number of subgrids per process incurs slightly higher simulation times but still offers nearly optimal strong scaling behavior and is thus a viable option compared to the single-subgrid configuration.

4.6 Illustrative numerical experiment

In soil hydrology the challenge of scale is ubiquitous. Heterogeneity in soil hydraulic properties exists from the sub-centimeter to the kilometer scale related to, e.g., micro- and macro-porosity and spatially continuous soil horizons, respectively. This heterogeneity impacts the flow and transport processes in the shallow soil zone and interactions with land surface processes. Examples are groundwater recharge and leaching of nitrate and pesticides and the resulting impact

	Subgrid size $m_x \times m_y \times m_z$	number of processes $P_x \times P_y \times P_z$	Problem size $P_x m_x \cdot P_y m_y \cdot P_z m_z$
Study 1	$128 \times 128 \times 40$	$32 \times 32 \times 1$	671 088 640
	$64 \times 64 \times 40$	$64 \times 64 \times 1$	671 088 640
	$32 \times 32 \times 40$	$128 \times 128 \times 1$	671 088 640
	$16 \times 16 \times 40$	$256 \times 256 \times 1$	671 088 640
	$8 \times 8 \times 40$	$512 \times 512 \times 1$	671 088 640
Study 2	$128 \times 128 \times 40$	$64 \times 64 \times 1$	2 684 354 560
	$64 \times 64 \times 40$	$128 \times 128 \times 1$	2 684 354 560
	$32 \times 32 \times 40$	$256 \times 256 \times 1$	2 684 354 560
	$16 \times 16 \times 40$	$512 \times 512 \times 1$	2 684 354 560
Study 3	$128 \times 128 \times 40$	$128 \times 128 \times 1$	10 737 418 240
	$64 \times 64 \times 40$	$256 \times 256 \times 1$	10 737 418 240
	$32 \times 32 \times 40$	$512 \times 512 \times 1$	10 737 418 240
Full system run	$18 \times 32 \times 40$	$896 \times 512 \times 1$	10 569 646 080

Table 4.2: Relevant parameters for the strong scaling study under the restriction of one subgrid per process. The problem size remains constant per scaling study by setting up the subgrid dimensions inversely proportional to the number of processes.

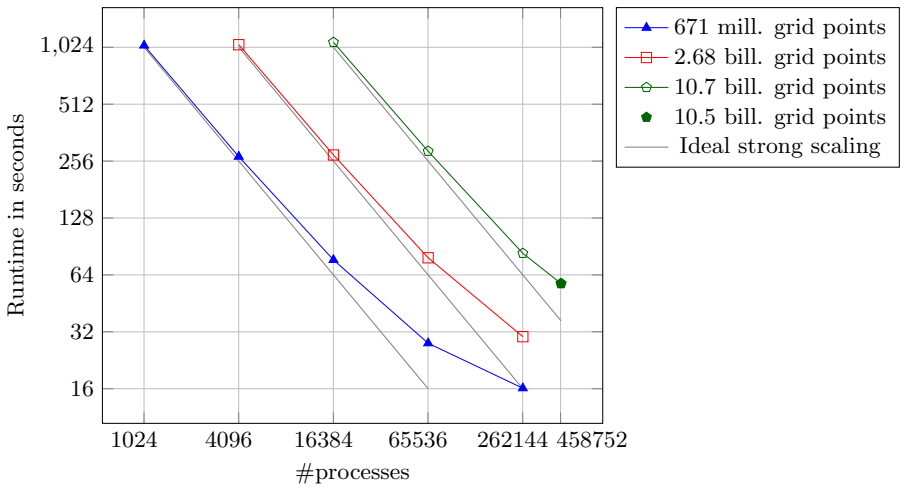


Figure 4.13: Strong scaling timing results of the modified version of ParFlow for different problem sizes. We plot the total runtime for each case. The solid green circle corresponds to the full size of the Juqueen system at 458,752 processes.

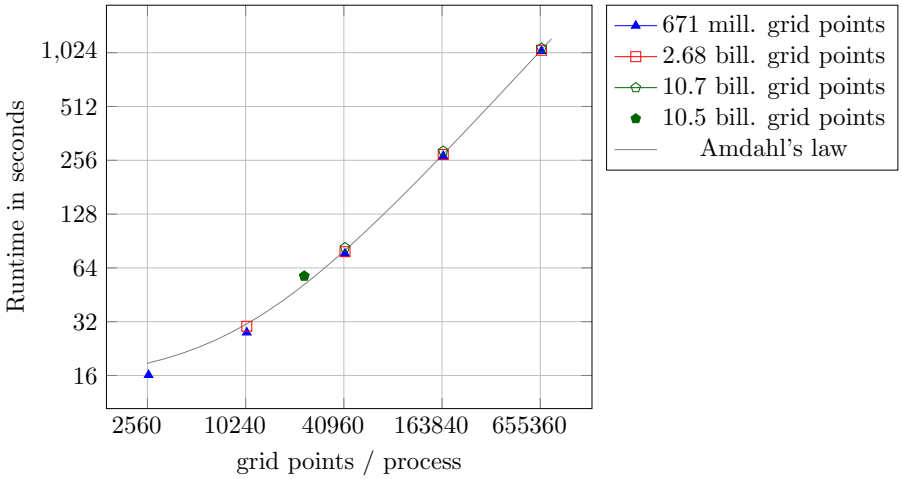


Figure 4.14: Strong scaling timing results of the modified version of ParFlow for different problem sizes. We plot the total runtime for each case against the number of grid points per process. The solid curve shows a fit of Amdahl's law $t = c_1 \cdot x + c_2$, where x denotes the number of grid points per MPI rank. The fitted parameters are $c_1 = 0.0016$ and $c_2 = 14.7$. This diagram offers another perspective on the optimality of weak scaling: We observe that measurements for simulations with the same number of grid points per process lie on top of each other in the vertical.

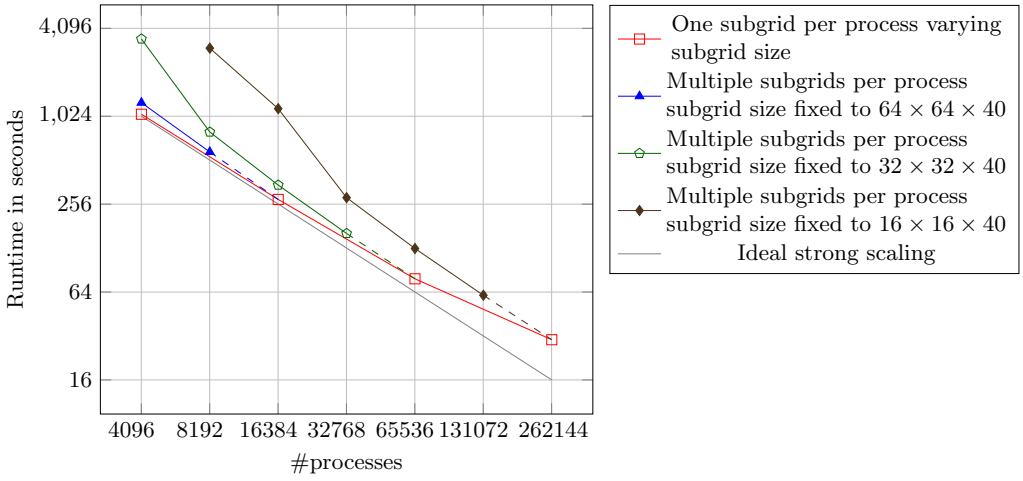


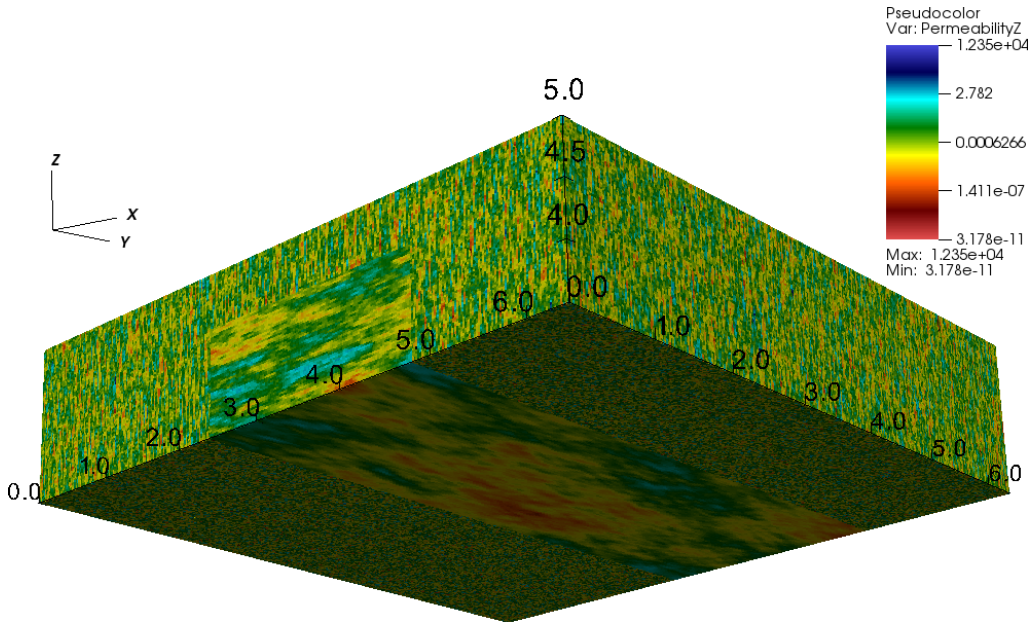
Figure 4.15: Strong scaling timing results of the modified version of ParFlow. We compare runs with single and multiple subgrids per process. The red line corresponds to the 2.68 billion degrees of freedom single-subgrid problem (also colored red in Figure 4.13). Here we choose three fixed subgrid sizes (blue, green, brown). Increasing the number of processes eventually leads to a single subgrid per process, which is the case already covered in Figure 4.13. This limit is indicated by the final dotted segment in each graph. The largest number of subgrids per process (left end point of each graph) is four for the run represented by the blue curve, 16 for the green and 32 for the brown.

on shallow aquifers. One major structural soil feature is defined by small scale preferential flow paths, developed from cracking and biota, that serve as high velocity conduits in the vertical direction. In large scale simulations, accounting simultaneously for layered soil horizons and macro porosity at the plot scale on the order to 10^2 to 10^3 m has been essentially impossible because of the high spatial resolution required and the enormous size of the system of equations resulting from the boundary and initial value problem defined by the 3D Richards equation.

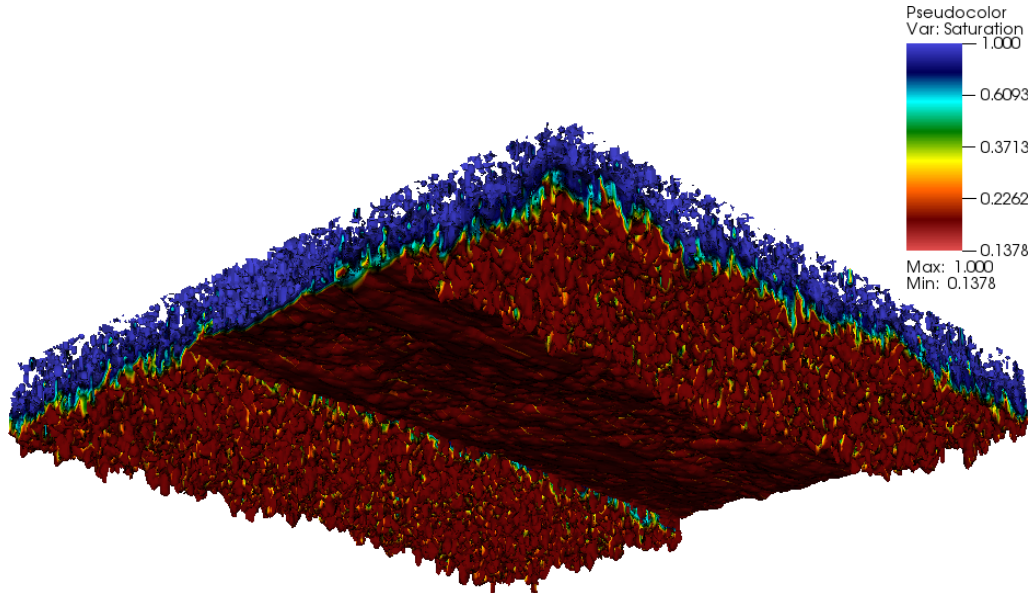
The improved parallel performance offered by the modified version of ParFlow motivates us to eventually target such complex simulations. In order to illustrate this, we simulate a hypothetical example configuration focused on the presence of macroporosity and layered soil horizons. The numerical experiment chosen solves an infiltration problem on a Cartesian domain. The initial water table is implemented as a constant head boundary at the bottom of the domain with a five meter unsaturated zone on top of it. The heterogeneous permeability parameter is simulated with a spatially correlated log-transformed Gaussian random field. We employ two realizations of such a field to model vertical and lateral preferential flow paths, respectively. Both random field realizations are obtained with a parallel random field generator implemented in ParFlow (the turning-bands algorithm [157]). We display an example of the outcome of such realizations and the saturation field obtained from our experiment in Figure 4.16. The total compute time required was roughly 280,000 core hours. While the modified version of ParFlow can be scaled easily to use large multiples of this number, using such amounts of time must be carefully justified. At this point, it makes sense to reserve extended scientific studies for a later research that will focus exclusively on the design and usefulness of such simulations.

4.7 Discussion

In this chapter we have presented our approach in order to improve the parallel performance of the subsurface simulator ParFlow. In particular, our work takes the code a step further in order to take advantage of the computational resources offered by the latest HPC systems. Our approach was to couple the ParFlow and `p4est` libraries such that the latter acts as the parallel mesh manager of the former. We achieved this with relatively small and local changes to ParFlow that constitute a reinterpretation rather than a redesign. This modified version of ParFlow offers a wider range of runnable configurations and improved performance with respect to the upstream version. We report good weak and strong scaling up to 458,752 MPI ranks on the Juqueen supercomputer. As a result, the modified version of ParFlow has been accepted into the High-Q-Club at the Jülich Supercomputing Centre [97]. The improved performance of the modified version of ParFlow opens the possibility of bigger and more realistic simulations. Considering our enhancements, most parallel bottlenecks are removed and implementation scalability is in principle unlimited. We note that we did not address the algorithmic efficiency of the time stepper or the preconditioner, since the mathematics of the solver remain unchanged. A natural extension of the work presented in this chapter is to make use of the AMR functionality provided by `p4est` to introduce locally refined meshes in ParFlow. In the next chapter, we will further investigate the mesh structures in ParFlow and propose minimal changes to advance in this direction.



(a)



(b)

Figure 4.16: Picture a) shows a slice of the permeability field, generated by combining two log-transformed Gaussian random fields with standard deviations ranging over 3 orders of magnitude. In b) we display the saturation field obtained after 32,928 time steps, which correspond to 9.83 seconds simulated time. The wall clock compute time required for this simulation was around 17 hours using 16,384 MPI ranks of JUQUEEN.

5 Towards AMR in ParFlow

In Chapter 4 we exposed the integration of ParFlow with the parallel AMR library `p4est`. We demonstrated with numerical examples how this enlarged the range of process counts that ParFlow may be executed with and improved parallel scalability. These results were obtained for uniform meshes and hence without explicitly exploiting the AMR capabilities of the `p4est` library. The objective of this chapter is to expose some of the ideas we developed in order to prepare the code for local mesh refinement. We focus on the correctness of the mesh connectivity when the code is executed on a parallel machine.

5.1 AMR with `p4est`

In Section 4.3 we introduced the concept of an octree and how it is naturally associated to a mesh covering a cubic (square in 2D) domain. Furthermore, the class of domains that may be represented can be enlarged by considering unions of octrees, conveniently named forest of octrees. A core functionality of tree based AMR libraries like `p4est` is to dynamically change the mesh elements by traversing the quadrants of the corresponding forest and either

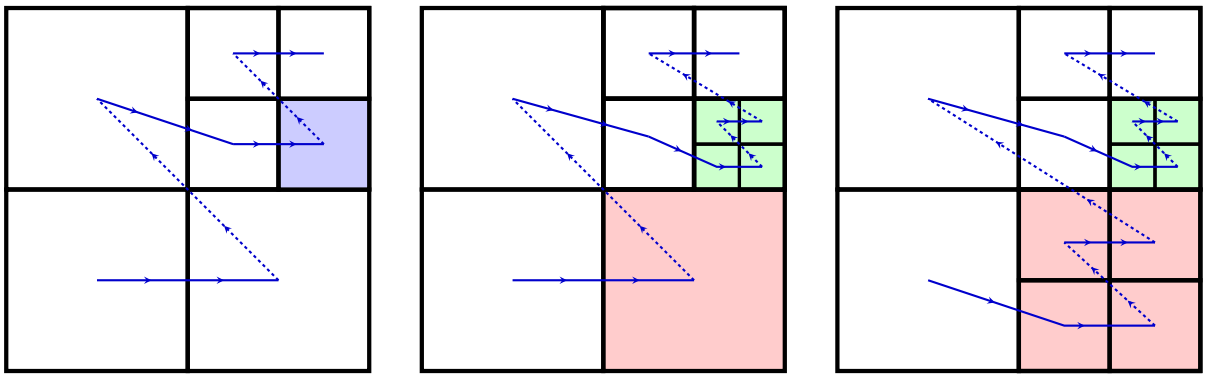
- refine the mesh by replacing a quadrant by its eight (four in 2D) children,
- coarsen the mesh, replacing a family of eight quadrants (four in 2D) by its common parent.

The `p4est` implementation of the previous routines takes as argument a user defined callback function that marks the quadrants to be considered for refinement or coarsening. A key feature of the SFC approach used in `p4est` is that the ordering of the quadrants is maintained after refine or coarsen are executed. Hence, manipulation of the mesh resolution can be achieved with small movements of data. Furthermore, quadrants of different refinement level are allowed to be neighbors of each other. This translates into meshes where elements of different sizes share parts of a mesh face or edge. Optionally, the `p4est` library guarantees that the size of the difference is at most a factor of two. This is known as 2:1 balance condition which we consider in our work; see Figure 5.1.

5.2 Enabling locally refined meshes in ParFlow

In Section 4.4, we exposed our approach to delegate ParFlow's mesh management to the software library `p4est`. The aim of this section is to extend it to include the case when the `p4est` object represents a locally refined mesh, and to carry out the minimal modifications such that ParFlow executes cleanly in parallel when a locally refined mesh is employed.

Our starting point is the `p4est` brick that we also use to create a uniform mesh with the input data provided by the user. We first refine this brick and then proceed to attach a suitable subgrid to each of its quadrants. We maintain the (user defined) number of points m_t in a subgrid along the t coordinate direction for $t \in \{x, y, z\}$. Local refinement is then enforced by halving the subgrid mesh spacing in accordance with the quadrant level. See Figure 5.2.



(a) (b) (c)

Figure 5.1: Local refinement. Starting from the mesh in (a), the blue shaded quadrant is marked for refinement. The refine routine will replace it by its four children to obtain the mesh displayed in (b). After refinement, the size difference between the red and green shaded quadrants in (b) amounts to a factor of four. We may enforce the 2:1 balance condition by refining the red shaded quadrant to produce the mesh depicted in (c). In all three figures the blue zig-zag line denotes the corresponding space filling curve, which determines the ordering of the elements.

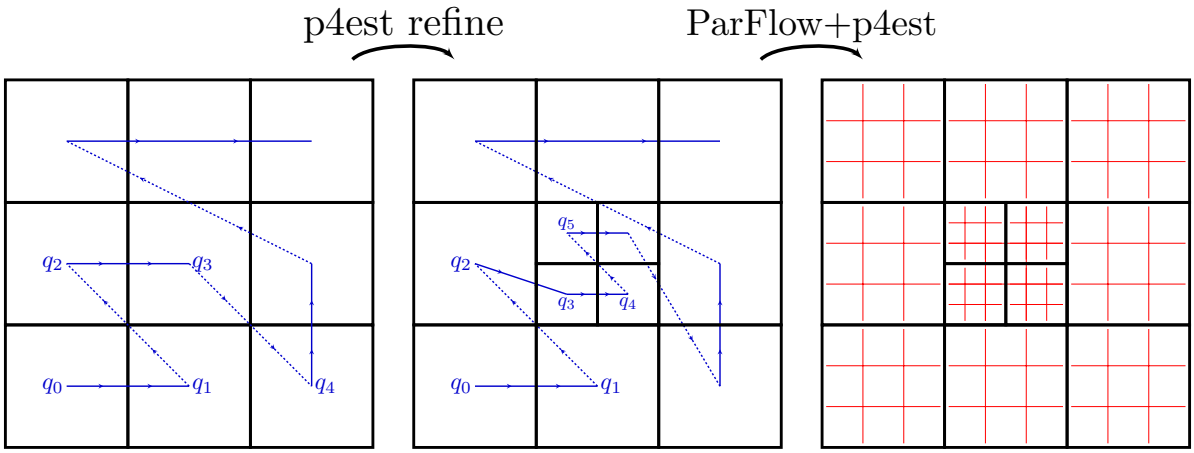


Figure 5.2: Work flow to enable AMR in ParFlow. We create a `p4est` object representing the mesh displayed in the left picture. Its dimensions are obtained by (4.5) after using $N_x = 9$, $N_y = 9$, $N_z = 1$ and $m_x = m_y = 3$, $m_z = 1$ as input values in formula (4.3). The corresponding space filling curve is shown in blue. With the `p4est` refine routine, this `p4est` object is modified to represent the mesh displayed in the middle. We keep our idea of attaching a subgrid to each of the quadrants in the later `p4est` object. The number of points per subgrid remains fixed but we adjust the mesh spacing when attaching a subgrid to a refined quadrant.

A key observation in the approach presented in Section 4.4 is that the scaled bottom left corner of the brick's quadrants matches the enumeration employed by the upstream version of ParFlow. This allows us to keep most of the original algorithms that operate on the mesh. In particular, we preserve those employed to determine the necessary information to perform parallel updates; see Algorithm 3.3 in Section 3.6. With the introduction of a locally refined mesh we may loosen the this property of the quadrant's corners. In order to illustrate this we consider the following example.

Suppose that we start with a computational mesh of dimensions $N_x \times N_y = 20 \times 20$, partitioned into four subgrids of dimensions $m_x \times m_y = 10 \times 10$, as displayed in Figure 5.3a. We then activate the AMR capabilities of `p4est` to refine the upper right corner of the associated `p4est` brick object; see Figure 5.3c. If left unchanged, the methodology from Section 4.4 to place a subgrid inside of the quadrants of the modified brick will lead to an erroneous mesh as shown in Figure 5.3d. Specifically, subgrids placed inside the newly created quadrants will have incorrect size and physical location. The reason is that the scaled lower left corner (c_x, c_y) from a child quadrant does no longer translate into integer coordinates that are consistent with the rule (3.21). In order to apply this rule, remember that we required $c_t \in \{0, 1, \ldots, P_t - 1\}$, where P_t is configured as the number of subgrids in the t direction and fixed at the beginning of the simulation. In the particular case of Figure 5.3d, we have $P_x = P_y = 2$ which will enforce $(c_x, c_y) \in \{0, 1\} \times \{0, 1\}$, but as shown in Figure 5.3c, the four new quadrants arising from the refinement process satisfy $(c_x, c_y) \in \{2, 3\} \times \{2, 3\}$. The issue with the proper scaling of the subgrids corresponding to these quadrants is straightforward to fix, and we will discuss the details in Section 5.2.1. A more cumbersome problem relates to the algorithms that determine the send and receive regions for a given stencil.

According to formula (3.20), the subgrids s_1 and s_3, placed inside the quadrants $\hat{q}_1$ and $\hat{q}_3$ with scaled corners $(1, 0)$ and $(2, 3)$, respectively, identify the following two sets of indices (3.15):

$$I_1 := \{i : i = 0, 1, \ldots, 9\} \otimes \{10 + j : j = 0, 1, \ldots, 9\}, \tag{5.1a}$$
$$I_3 := \{20 + i : i = 0, 1, \ldots, 9\} \otimes \{30 + j : j = 0, 1, \ldots, 9\}, \tag{5.1b}$$

which are completely separated. In the configuration from Figure 5.3c, the quadrants $\hat{q}_1$ and $\hat{q}_3$ share a mesh face. Hence, a stencil requiring information from the adjacent neighbor will involve communication between the subgrids placed inside these quadrants. For example, subgrid s_1 needs access to the degrees of freedom corresponding to the indices on the stripe $\{(20 + i, 10) : i = 1, \ldots, 9\}$ from s_3, that is, those located at the bottom of its top neighbor subgrid. However, ParFlow will try to determine this stripe by shifting the set I_1 and then intersecting the outcome with I_3, which will result in the empty set. We propose two ideas to overcome this issue in Section 5.2.2.

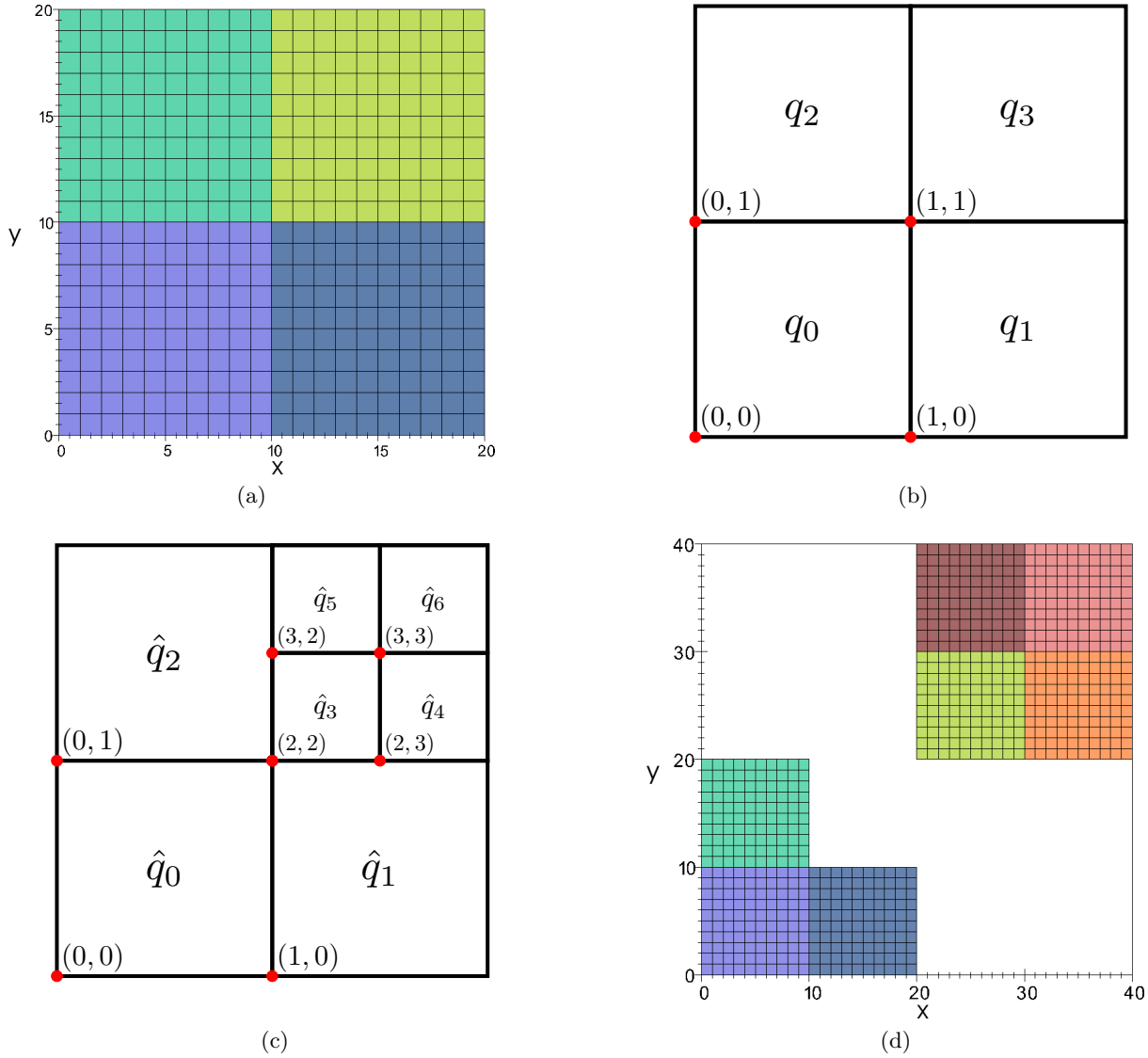


Figure 5.3: In (a) we show a ParFlow mesh obtained using $N_x = 20$, $N_y = 20$, $N_z = 1$ and $m_x = m_y = 10$, $m_z = 1$ as input values in formula (4.3). The corresponding `p4est` brick object is shown in (b). We refine the latter to produce the brick displayed in (c). For each of the quadrants q_i in (b) and $\hat{q}_j$ in (c) we display its lower left corner scaled by the multiplier g from (4.4). Without further modifications, the approach exposed in Section 4.4 to place a subgrid inside each of the quadrants of the brick (c) leads to the erroneous mesh depicted in (d).

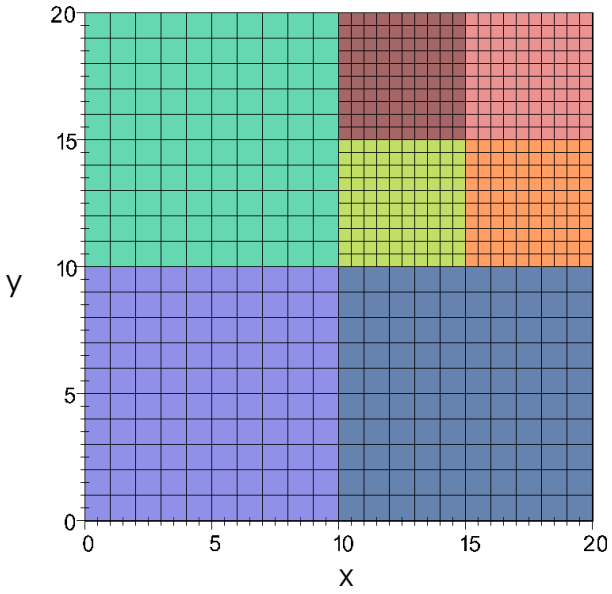


Figure 5.4: Corrected mesh associated to the brick from Figure 5.3c.

5.2.1 Correcting the mesh layout

In Section 3.5, we saw that the physical location of a point lying in a subgrid s is given by

$$x_i = x_0 + (i + 0.5)\left(\frac{\Delta x}{2^{s.\mathrm{rx}}}\right), \tag{5.2a}$$

$$y_j = y_0 + (j + 0.5)\left(\frac{\Delta y}{2^{s.\mathrm{ry}}}\right), \tag{5.2b}$$

$$z_k = z_0 + (k + 0.5)\left(\frac{\Delta z}{2^{s.\mathrm{rz}}}\right), \tag{5.2c}$$

where (x_0, y_0, z_0) is the background's anchor point, $\Delta x, \Delta y, \Delta z$ and $\mathrm{rx}, \mathrm{ry}, \mathrm{rz}$ denote the mesh spacing and refinement per coordinate direction, respectively. We will assume that the refinement level $\ell \geq 0$ is the same in all coordinate directions. Hence, our modified version of ParFlow takes

$$\mathrm{rx} = \mathrm{ry} = \mathrm{rz} = \ell. \tag{5.3}$$

This is compatible with the upstream version, where ℓ is fixed to zero.

If the subgrid s is attached to a given quadrant q, we need to relate ℓ with the refinement level q_L of q in the `p4est` object. By the construction in Section 4.4.1, the quadrants of the `p4est` object representing the initial uniform mesh have refinement level k_0, where k_0 was defined in (4.4). Therefore, the correct value is

$$\ell = q_L - k_0. \tag{5.4}$$

Taking a look back into the introductory example, this is the required modification to produce the mesh depicted in Figure 5.4, which is spaced correctly.

5.2.2 Two proposals to correct the communication layout

The main issue appearing with the introduction of a locally refined mesh is that ParFlow may fail to correctly determine the send and receive regions; see Section 3.6. In the case of the

send region, Algorithm 3.5 assumes that if two subgrids s and $\tilde{s}$ are neighbors, then the indices corresponding to the coordinate direction of the mesh face they share are consecutive numbers. For example, in a two dimensional situation where s and $\tilde{s}$ share a face in the x direction, it follows that

$$|s.\mathrm{ix} - \tilde{s}.\mathrm{ix}| = \mathrm{nx}. \tag{5.5}$$

If s and $\tilde{s}$ correspond to subgrids attached to quadrants of different levels of refinement q and $\tilde{q}$, respectively, this property does not hold anymore. The immediate consequence is that ParFlow will miss the fact that s and $\tilde{s}$ are neighbors. We have come up with two variants that aim to solve this issue. In both cases the parent of the finer quadrants plays a fundamental role. Without loss of generality, let $\tilde{q}$ denote the finer quadrant, by the 2:1 balance condition the parent of $\tilde{q}$ has the same refinement level as q. In analogy, we may define the parent subgrid of $\tilde{s}$ as the subgrid with the same dimensions as $\tilde{s}$ but with the lower left corner computed from the parent quadrant of $\tilde{q}$. With these definitions in mind, our two proposed solutions are the following.

(A) If $\tilde{s}$ denotes the finer subgrid, then the parent subgrid of $\tilde{s}$ and s will be neighbors and the indices of the mesh face they share will be compatible in the sense explained above. From this observation, we may construct the parent subgrid on the fly when subgrids of different levels are intersected to compute the send (or receive) region. See Figure 5.5. In particular,

i) If we try to intersect a coarse subgrid s with a fine shifted one $\tilde{s}$, we get the right values by replacing $\tilde{s}$ by its shifted parent and performing the intersection.

ii) If we try to intersect a fine subgrid $\tilde{s}$ with a coarse shifted one s, we get the right values by replacing $\tilde{s}$ by its parent and then map the result of the intersection back to $\tilde{s}$.

Hence, this variant operates in the indices of the degrees of freedom to reuse the existing ParFlow infrastructure that handles parallel communication. A procedure summarizing this approach is show in Algorithm 5.1.

(B) In the second variant, we operate with the degrees of freedom directly: When a given subgrid requires data from a different size one to update the values it is responsible for, our idea is to construct the parent subgrid s_p of the finer and use it to carry out the required calculations; see Figure 5.6. Specifically,

i) A coarse subgrid s requires data from a fine subgrid $\tilde{s}$ to complete a calculation. Then, we construct the parent of $\tilde{s}$, project the values $\tilde{s}$ is responsible for to s_p and perform the calculation using s and s_p.

ii) A fine subgrid $\tilde{s}$ requires data from a coarse subgrid s to complete a calculation. Here, we construct the parent of $\tilde{s}$, project the values $\tilde{s}$ is responsible for to s_p, perform the calculation using s and s_p and finally interpolate the result back to $\tilde{s}$.

Note that in both cases of the variant (B), the calculation involving s and s_p will require extra communication not included in the ParFlow functionality and hence must be explicitly coded.

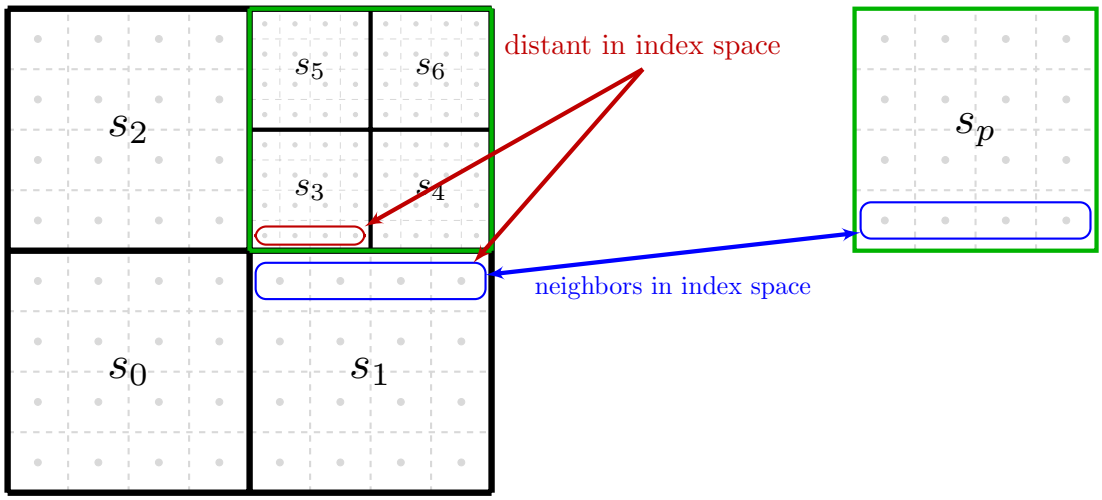


Figure 5.5: Assuming a stencil requiring information from the adjacent neighbors, the subgrids s_1 and s_3 need to communicate. Specifically, they should share with each other the degrees of freedom enclosed in red and blue. Because the corresponding indices are not neighbors in the index space, we may build the parent subgrid of s_p of s_3. In the index space s_p and s_1 are neighbors, so we can use this fact to compute the send and receive regions with respect to s_p and map them accordingly to s_3. This covers both cases from the variant (A) described in Section 5.2.2.

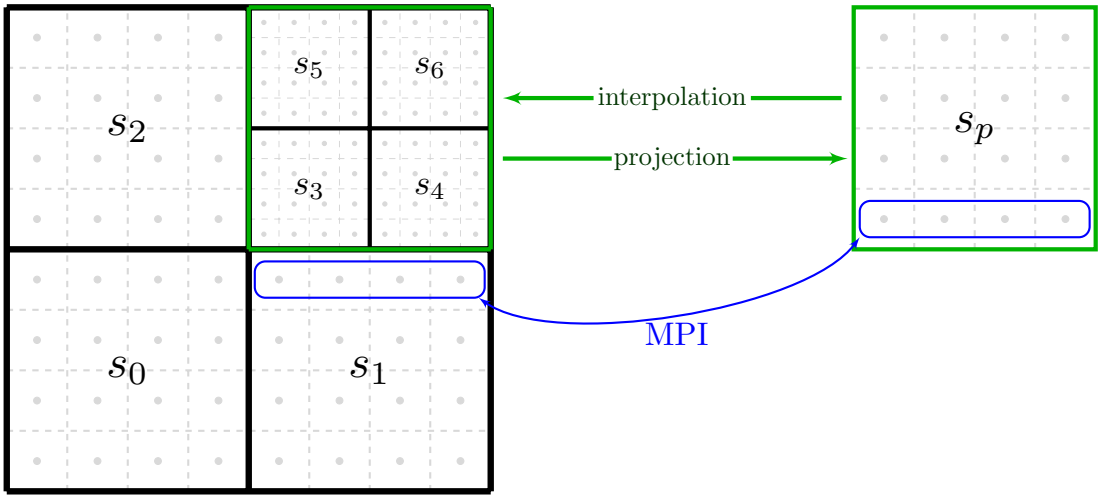


Figure 5.6: Assuming a stencil requiring information from the adjacent neighbors, the subgrid s_4 needs data from s_1 to update its own degrees of freedom. We may build the parent subgrid s_p of s_4, project s_4's values to s_p and use the latter to complete the calculation. This step will require communication between s_p and s_1. After the computation is complete, we interpolate the result back to s_4. This corresponds to case ii) from the variant (B) described in Section 5.2.2. Case i) follows a similar path omitting the interpolation step and interchanging the roles of s_1 and s_4.

Algorithm 5.1: Send Region with locally refined meshes

Input : SubgridArray S_{loc}, SubgridArray S_{all}, Stencil sten

```
1  S_neigh ← GetNeighbors(S_loc, S_all, sten)        Neighbors array from Algorithm 3.4
2  S_patch ← GetPatches(S_neigh, sten)               Patch array from Algorithm 3.3
3  foreach s ∈ S_all do
4      foreach s̃ ∈ S_patch do
5          if s.ℓ < s̃.ℓ                                          s is finer than s̃
6          then
7              s̃_p ← parent subgrid of s̃
8              t ← s ∩ sten(s̃_p)        s̃ is a shifted subgrid, intersect with shifted parent
9          else
10             if s.ℓ > s̃.ℓ                                      s is coarser than s̃
11             then
12                 s_p ← parent subgrid of s
13                 t′ ← s_p ∩ s̃
14                 t ← t′ projected onto s        transfer result back to the finer subgrid
15             else
16                 t ← s ∩ s̃                                     same level subgrids
17             end
18         end
19         if t ≠ ∅ then
20             Append s to S_send
21         end
22     end
23 end
24 return S_send
```

Algorithm 5.1: Modification of Algorithm 3.5 to compute the send region with respect to a given stencil and taking into account the case of different size subgrids. We recall that Algorithm 3.4 computes the neighbors of all local subgrids with respect to a given stencil. Meanwhile Algorithm 3.3 computes an array consisting of the non-empty differences $\text{sten}(s) - s := \{v \in \text{sten}(s) \mid v \notin s\}$, for each subgrid s it takes as argument; see Section 3.6 for further details. This corresponds to the variant (A) explained in Section 5.2.2.

5.3 Discussion

In this chapter we have discussed the algorithmic approach to activate the usage of locally refined grids in ParFlow. We have focused exclusively on the mesh subsystem and its correct functioning when the code is executed in parallel. That is, ParFlow must correctly determine the message envelopes when the AMR capabilities of `p4est` are activated. This step still leaves a long way to go in order to establish the appropriate data representation and numerical mathematics context. For example, we did not address how we want to interpolate fluxes at the boundary of two subgrids with different mesh spacing. We will investigate this in future work.

6 A block preconditioner for locally refined meshes

This chapter is based on the manuscript [39]. We have carried out minor editions in order to match the notation of this thesis, without changing its core content. The content from Section 6.2 corresponds to work realized by co-author B. Metsch. We decided to keep it in this thesis in order to offer a self-contained exposition of the results of this chapter.

In many applications one is more interested in the gradient of the solution of the Poisson equation than in the solution itself. One approach is to use standard methods such as finite differences (FD), finite elements (FE) or finite volumes (FV), to obtain an approximate solution and then to compute its gradient via numerical differentiation, which may lead to a loss in accuracy. A mixed finite element (MFE) discretization appears to be a more natural choice since the gradient of the solution is part of the unknown variables of the formulation. Generally, in MFE methods the vector valued quantity is approximated at least with the order of accuracy of the scalar unknown [31]. In addition, the computed solution satisfies mass conservation at the element level, a property highly relevant in fields like fluid flow simulation where the governing partial differential equations (PDE) are derived from mass balance laws.

Another PDE in which MFE is a natural choice is the Stokes system, here a FE or FD discretization could lead to spurious oscillations on the scalar (pressure) variable since this is in general not unique. Mixed Stokes and Poisson systems have a similar mathematical structure in the sense that both lead to a block saddle point problem. Nevertheless, they require in general different MFE spaces. For the Poisson case, continuity is only required in the normal component of the vector (velocity) variable. This has the consequence that geometry transformations are different if one is interested in working with isoparametrically mapped elements [24, Sec. 2.1.3].

MFE discretizations lead to symmetric indefinite systems of algebraic equations that may be solved by iterative methods. In order to obtain a solution with a reasonable investment of computational resources, the use of optimal preconditioners becomes mandatory. Although the existence of an optimal preconditioner for general (anisotropic) coefficients remains an open question, there has been a significant amount of work in this direction. In [8], the authors propose two preconditioners for the operator $(I - \text{grad div})$ in two dimensions, one based on domain decomposition and another on multigrid. The latter was generalized to operators of the form $(\rho I - \mu \text{grad div})$ and three-dimensional problems [9]. From the observation that a MFE discretization of a generalized diffusion equation is well posed in a product of two different discrete function spaces, [132] and [131] propose two block-diagonal preconditioners. One of them is what we will refer to as Schur complement preconditioner and consists of a lumped diagonal approximation of the $(1,1)$ block and an algebraic multigrid V-cycle to approximate a Schur complement on the $(2,2)$ block. The preconditioners introduced in these chapter are shown to be optimal with respect to the mesh size and various classes of coefficients (such as a conductivity tensor). Recent work [105] introduces a new preconditioner whose key ingredient is the approximation of the $(1,1)$ block with an auxiliary space multigrid method [106] that offers optimality with respect to a larger class of coefficients. The authors provide numerical

evidence for two dimensional problems posed on uniform rectangular meshes.

For some problems we may require a very fine mesh in order to correctly resolve the phenomena we are trying to model. A uniform mesh might be undesirable or even impractical given the computational resources it requires. One solution to this problem is to use locally refined meshes, which use the correct resolution only in the portion of the domain that really requires it [11, 20, 56]. In a uniform mesh, the vertices of the mesh elements are shared between neighboring elements. Introducing local refinement may destroy this property and introduce the so called hanging nodes. A hanging node is a vertex of a mesh element that lies in an edge or face of a neighboring element. When using locally refined meshes, there are essentially two possibilities, either we allow the presence of hanging nodes or not [126, 2]. In the context of MFE, implementations making use of adaptively refined meshes with hanging nodes are not the standard, nevertheless the theoretical framework has been established in the early 90s [65, 66]. None of the above mentioned methods for preconditioning address the case of adaptively refined meshes. Hence, the goal of this chapter is to present a multigrid preconditioner that retains its robustness in such case by taking into account the different scales in the saddle point structure.

6.1 Problem formulation

In this section we briefly review the mixed formulation of a Laplacian-like partial differential equation. The material covered here is standard and found in many textbooks; see e.g., [24, 28], and augmented with a few recent results.

Let $\Omega \in \mathbb{R}^d$ for $d \in \{2, 3\}$ be a bounded Lipschitz domain with boundary $\Gamma = \partial\Omega$. $H^m(\Omega)$ will denote the standard Hilbert space of functions in $L^2(\Omega)$ whose weak derivatives up to order $m \geq 0$ are also square integrable. $H^m(\Omega)$ is endowed with the usual norm and seminorm,

$$\|v\|_m := \sum_{|\alpha| \leq m} \int_\Omega |D^\alpha v| \, \mathrm{d}x, \quad |v|_m := \sum_{|\alpha| = m} \int_\Omega |D^\alpha v| \, \mathrm{d}x. \tag{6.1}$$

We define the velocity space

$$\boldsymbol{H}(\operatorname{div}; \Omega) := \{\boldsymbol{v} \in (L^2(\Omega))^d : \operatorname{div} \boldsymbol{v} \in L^2(\Omega)\}, \tag{6.2}$$

which is a Hilbert space with the norm

$$\|\boldsymbol{v}\|^2_{\operatorname{div}} = \|\boldsymbol{v}\|^2_0 + \|\operatorname{div} \boldsymbol{v}\|^2_0. \tag{6.3}$$

We assume that $\Gamma = \Gamma_D \cup \Gamma_N$, with $\Gamma_D \cap \Gamma_N = \emptyset$, and that Γ_D has nonzero $(d-1)$-dimensional Lebesgue measure. Lastly, define

$$H^1_{0,D}(\Omega) := \{\phi \in H^1(\Omega) : \phi|_{\Gamma_D} = 0\}. \tag{6.4}$$

In the following, we use bold mathematical symbols to denote vectors and matrices over $\mathbb{R}^d$ and the default font for scalar quantities.

We consider the equation

$$-\operatorname{div} \boldsymbol{\mathcal{K}}(\boldsymbol{x}) \nabla p = f \quad \text{in } \Omega, \tag{6.5a}$$

$$p = p_0 \quad \text{on } \Gamma_D, \tag{6.5b}$$

$$\boldsymbol{\mathcal{K}}(\boldsymbol{x}) \nabla p \cdot \boldsymbol{n} = g \quad \text{on } \Gamma_N, \tag{6.5c}$$

where $\boldsymbol{n}$ is the outer normal vector to the boundary Γ. The conductivity tensor $\boldsymbol{\mathcal{K}}(\boldsymbol{x})$ is a $d \times d$ symmetric positive definite matrix whose smallest eigenvalue is bounded uniformly away from zero. Furthermore, the data is required to satisfy

$$f \in L^2(\Omega), \quad p_0 \in H^{1/2}(\Gamma_D) \quad \text{and} \quad g \in L^2(\Gamma_N), \tag{6.6a}$$

where $H^{1/2}(\Gamma_D)$ stands for the space spanned by functions of the form $q\,|_{\Gamma_D}$ with $q \in uH^1(\Omega)$. Introducing the variable $\boldsymbol{u} = \boldsymbol{\mathcal{K}}\nabla p$ leads to the mixed first-order system

$$\begin{aligned} \boldsymbol{u} &= \boldsymbol{\mathcal{K}}\nabla p \quad && \text{in } \Omega, && (6.7a)\\ -\operatorname{div} \boldsymbol{u} &= f \quad && \text{in } \Omega, && (6.7b)\\ p &= p_0 \quad && \text{on } \Gamma_D, && (6.7c)\\ \boldsymbol{u} \cdot \boldsymbol{n} &= g \quad && \text{on } \Gamma_N. && (6.7d) \end{aligned}$$

6.1.1 Weak formulation

To derive the mixed weak formulation of (6.7) we introduce the following space,

$$\boldsymbol{H}_{0,N}(\Omega) := \{\boldsymbol{\tau} \in \boldsymbol{H}(\operatorname{div};\Omega) : \langle \boldsymbol{\tau} \cdot \boldsymbol{n}, \phi \rangle = 0 \text{ for all } \phi \in H^1_{0,D}(\Omega)\}. \tag{6.8}$$

Due to particular properties of The dual paring $\langle \cdot, \cdot \rangle$ is defined via Green's formula; see [24, pp. 50]. Multiplying the first equation in (6.7) by $\boldsymbol{\mathcal{K}}^{-1}$ and then by a test function $\boldsymbol{v} \in \boldsymbol{H}_{0,N}(\Omega)$, the second by some $q \in L^2(\Omega)$, integrating over Ω and using Green's formula on the gradient term yields the following weak formulation: Find $(\boldsymbol{u}, p) \in \boldsymbol{H}_{0,N}(\Omega) \times L^2(\Omega)$ such that

$$\begin{aligned} \int_\Omega \boldsymbol{\mathcal{K}}^{-1}\boldsymbol{u} \cdot \boldsymbol{v}\,\mathrm{d}x + \int_\Omega p \operatorname{div} \boldsymbol{v}\,\mathrm{d}x &= \int_{\Gamma_D} p_0(\boldsymbol{v} \cdot \boldsymbol{n})\,\mathrm{d}s \quad \text{for all } \boldsymbol{v} \in \boldsymbol{H}_{0,N}(\Omega), && (6.9a)\\ \int_\Omega q \operatorname{div} \boldsymbol{u}\,\mathrm{d}x &= -\int_\Omega fq\,\mathrm{d}x \quad \text{for all } q \in L^2(\Omega). && (6.9b) \end{aligned}$$

Let us define the bilinear forms $a : \boldsymbol{H}(\operatorname{div};\Omega) \times \boldsymbol{H}(\operatorname{div};\Omega) \to \mathbb{R}$ and $b : \boldsymbol{H}(\operatorname{div};\Omega) \times L^2(\Omega) \to \mathbb{R}$,

$$a(\boldsymbol{u}, \boldsymbol{v}) := \int_\Omega \boldsymbol{\mathcal{K}}^{-1}\boldsymbol{u} \cdot \boldsymbol{v}\,\mathrm{d}x, \quad b(\boldsymbol{v}, p) := \int_\Omega p \operatorname{div} \boldsymbol{v}, \tag{6.10}$$

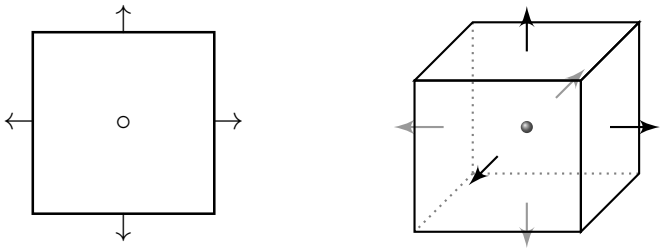
and linear functionals $\mathbb{A} : \boldsymbol{H}(\operatorname{div};\Omega) \to \mathbb{R}$, $\mathbb{B} : L^2(\Omega) \to \mathbb{R}$,

$$\mathbb{A}(\boldsymbol{v}) := \int_{\Gamma_D} p_0(\boldsymbol{v} \cdot \boldsymbol{n})\,\mathrm{d}s, \quad \mathbb{B}(q) := -\int_\Omega fq\,\mathrm{d}x. \tag{6.11}$$

To enforce the Neumann boundary condition (6.5c), (6.7d), let $\tilde{p}$ a classical solution of (6.5) with $f = 0$ and $p_0 = 0$. Then, taking $\tilde{\boldsymbol{u}} = \nabla\tilde{p}$ the mixed weak formulation of (6.7) is: Find $\boldsymbol{u} = \tilde{\boldsymbol{u}} + \boldsymbol{u}_0$ with $\boldsymbol{u}_0 \in \boldsymbol{H}_{0,N}(\Omega)$ and $p \in L^2(\Omega)$ such that

$$\begin{aligned} a(\boldsymbol{u}_0, \boldsymbol{v}) + b(\boldsymbol{v}, p) &= \mathbb{A}(\boldsymbol{v}) - a(\tilde{\boldsymbol{u}}, \boldsymbol{v}) \quad \text{for all } \boldsymbol{v} \in \boldsymbol{H}_{0,N}(\Omega), && (6.12a)\\ b(\boldsymbol{u}, q) &= \mathbb{B}(q) - b(\tilde{\boldsymbol{u}}, q) \quad \text{for all } q \in L^2(\Omega). && (6.12b) \end{aligned}$$

Existence and uniqueness of a solution for the problem (6.12) follows from standard arguments, namely establishing coercivity of the bilinear form $a(\cdot, \cdot)$, and the Ladyženskaja-Babuška-Brezzi (LBB) condition; see eg. [31].



(a) (b)

Figure 6.1: Degrees of freedom for the rectangular $\mathcal{RT}_0$ element in two (a) and three (b) dimensions. For the velocity, the degrees of freedom are normal components at the edge (face) mid sides of an element. The pressure is located at the center of the element.

6.1.2 Discretization

We pick finite dimensional subspaces $\boldsymbol{X}_0^h \subset \boldsymbol{H}_{0,N}(\Omega)$ and $V^h \subset L^2(\Omega)$ and define the following problem: Find $(\boldsymbol{u}_h, p_h) \in \boldsymbol{X}_0^h \times V^h$ such that

$$a(\boldsymbol{u}_h, \boldsymbol{v}_h) + b(\boldsymbol{v}_h, p_h) = \tilde{\mathbb{A}}(\boldsymbol{v}_h) \quad \text{for all } \boldsymbol{v}_h \in \boldsymbol{X_0^h}, \tag{6.13a}$$

$$b(\boldsymbol{u}_u, q_h) = \tilde{\mathbb{B}}(q_h) \quad \text{for all } q_h \in V^h, \tag{6.13b}$$

where the linear forms $\tilde{\mathbb{A}}$ and $\tilde{\mathbb{B}}$ take into account the Neumman boundary condition like in (6.12), that is

$$\tilde{\mathbb{A}}(\boldsymbol{v}) := \begin{cases} \mathbb{A}(\boldsymbol{v}) - a(\tilde{\boldsymbol{u}}, \boldsymbol{v}) & \text{if } \Gamma_N \neq \emptyset, \\ \mathbb{A}(\boldsymbol{v}) & \text{else,} \end{cases} \tag{6.14a}$$

$$\tilde{\mathbb{B}}(q) := \begin{cases} \mathbb{B}(q) - b(\tilde{\boldsymbol{u}}, q) & \text{if } \Gamma_N \neq \emptyset, \\ \mathbb{B}(q) & \text{else.} \end{cases} \tag{6.14b}$$

To ensure that (6.13) is well posed, the pair of spaces $(\boldsymbol{X}_0^h, V^h)$ must be chosen such that the LBB condition is fulfilled for the discrete problem. Many spaces with this property have been developed since the early seventies, such as the Raviart-Thomas [135] and Brezzi-Douglas-Marini [30] spaces. In this chapter we stick to the lowest order Raviart-Thomas space discretization ($\mathcal{RT}_0$) defined on rectangles/hexahedra. Hence, in an element Ω_e the velocity and pressure test functions take the form

$$\boldsymbol{v}|_{\Omega_e} = \begin{Bmatrix} (a_0 + b_0 x, a_1 + b_1 y)^T & \text{if } d = 2 \\ (a_0 + b_0 x, a_1 + b_1 y, a_2 + b_2 z)^T & \text{if } d = 3 \end{Bmatrix}, \quad p|_{\Omega_e} = c_0, \tag{6.15}$$

respectively, where a_i, b_i for $i = 0, \ldots, d-1$ and c_0 are constants. The degrees of freedom are shown in Figure 6.1. The following approximation properties are well known for $\mathcal{RT}_0$ in the context of affine elements defined on uniform meshes [24],

$$\|\boldsymbol{u} - \boldsymbol{u}_h\|_0 \leq Ch\|\boldsymbol{u}\|_1, \tag{6.16a}$$

$$\|p - p_h\|_0 \leq ch\left(\|p\|_1 + \|\boldsymbol{u}\|_1\right), \tag{6.16b}$$

where we assume that the pair $(\boldsymbol{u}, p)$ fulfills the regularity requirements required by the right hand side of (6.16).

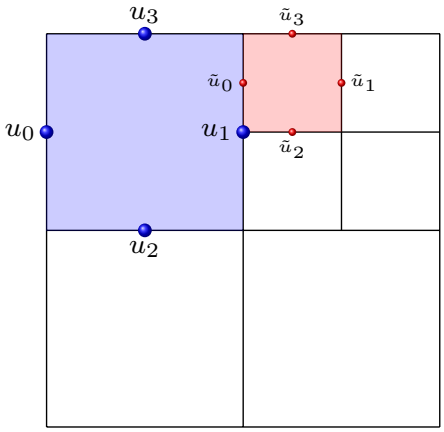


Figure 6.2: Locally refined mesh with hanging nodes. To enforce continuity of the flux normals for a $\mathcal{RT}_0$ discretization, the velocity value on a hanging node is defined by the corresponding non-hanging node lying in the same edge. For the case in the picture we define $\tilde{u}_0 := -u_1$.

6.1.3 Adaptivity

For well behaved (smooth) problems posed on convex domains, the use of a uniform mesh usually offers satisfactory results when computing a numerical solution, that is, there is an optimal trade-off between numerical effort (computational resources) invested and effective reduction of the error. Nevertheless, there are situations in which the mesh resolution required to accurately reproduce the physical behavior of the underlying PDE becomes computationally impractical if imposed on the whole domain.

Adaptive mesh refinement (AMR) provides a valuable tool in order to reduce the computational complexity in such situations by increasing the mesh resolution only locally where is required. As stated in section Section 6.1.2, we will work with meshes composed of rectangles/hexahedra. The refinement schemes used in this chapter include the case of a mesh with hanging nodes, that is, we allow (a nonempty) intersection of two elements to be a complete side of a neighboring element. Additionally, we specify to use 2:1 balanced meshes: The length ratio between a coarse and a fine element is at most of factor two; see Figure 6.2. Non 2:1 balanced meshes would also be possible, although this will require more technical work regarding the definition of MFE spaces and the parallelization. Continuity of fluxes across interfaces for this kind of meshes can be enforced in several ways. One is to eliminate te degrees of freedom at hanging nodes [65]. Another approach is to use Mortar finite elements [5]. We will follow the former option. Due to our assumption of 2:1 balance, for a $\mathcal{RT}_0$ discretization given a hanging node with flux value $\tilde{u}$, there is only one non-hanging node u lying in the same edge/face; see Figure 6.2.

Estimates for locally refined meshes using hanging nodes have been studied in [65, 66]. Essentially, it is shown that the $\mathcal{RT}_0$ spaces still respect the LBB condition after introducing locally refined grids, and hence the estimates (6.16) are still valid.

6.1.4 Preconditioning

The discretization of (6.7) via stable MFE leads to a saddle point problem defined by the following block matrix,

$$\mathcal{A}_h := \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B}^\top \\ \boldsymbol{B} & 0 \end{bmatrix} = \begin{bmatrix} \mathrm{Id}_h & -\mathrm{grad}_h \\ \mathrm{div}_h & 0 \end{bmatrix}, \tag{6.17}$$

where $\boldsymbol{A}$ is the vector mass matrix and $\boldsymbol{B}$ is the discrete divergence operator. This is a well studied problem and there are several solution methods available; see e.g., [19] for a comprehensive review. They range from Uzawa algorithms and its variants [27], projection methods [79], to block factorization methods [60, 114]. In contrast to most methods that treat flux and pressure individually, we introduce a monolithic block multigrid method [120].

To prepare the following exposition, let us briefly discuss notation and some background. We consider the case that matrix (6.17) is symmetric and indefinite. The factorization

$$\mathcal{A}_h = \begin{bmatrix} \boldsymbol{I} & 0 \\ \boldsymbol{B}\boldsymbol{A}^{-1} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{A} & 0 \\ 0 & -\boldsymbol{S} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} & \boldsymbol{A}^{-1}\boldsymbol{B}^\top \\ 0 & \boldsymbol{I} \end{bmatrix}, \qquad \boldsymbol{S} = \boldsymbol{B}\boldsymbol{A}^{-1}\boldsymbol{B}^\top, \tag{6.18}$$

implies that $\mathcal{A}_h$ is congruent to a block diagonal matrix [68]. This fact motivates the use of a preconditioner of the form

$$\mathcal{B} = \begin{bmatrix} \boldsymbol{M} & 0 \\ 0 & -\boldsymbol{N} \end{bmatrix}, \tag{6.19}$$

where $\boldsymbol{M}$ and $\boldsymbol{N}$ satisfy

$$\boldsymbol{M}\boldsymbol{A} \approx \boldsymbol{I}, \tag{6.20a}$$

$$\boldsymbol{N}\boldsymbol{S} \approx \boldsymbol{I}. \tag{6.20b}$$

A simple choice is to take $\boldsymbol{M}$ as the inverse of the lumped mass matrix $\boldsymbol{A}$. The Schur complement $\boldsymbol{S}$ represents the operator $-\Delta_h$. Hence, (6.20b) suggests $\boldsymbol{N} \approx \Delta_h^{-1}$, and the second block of the preconditioner $\mathcal{P}$ should approximate the inverse of a discrete Laplacian in pressure space. Then, we can use a solver for elliptic operators such as multigrid to apply $\boldsymbol{N}$. Nevertheless, it is a concern that our pressure belongs to $L^2(\Omega)$, and strictly speaking we do not have the required regularity to apply Δ_h^{-1}. An option to deal with this problem is to use the auxiliary space technique [170], in which the idea is to use a multigrid preconditioner for continuous pressure elements and then project it in to the desired space of discontinuous pressure elements in combination with a suitable smoothing operator. We note that some approaches only use a one-sided factorization of (6.18), which leads to a block-triangle form of $\mathcal{B}$; eg. [62, 107]. For problems in which the (1,1) block of $\mathcal{A}_h$ has a non-symmetric term, this variant offers a faster converge of the iterative solver at the price of evaluating an additional sparce matrix vector product with respect to the block diagonal preconditioner [69].

In the context of the (Navier-)Stokes equations, dedicated approximations to the inverse of the Schur complement have been proposed [62, 61]. These include the pressure Schur complement methods [159] and the least squares commutator preconditioner a.k.a. BFBt and its extensions [60]. Following the presentation from [138], the BFBt approximation of the inverse of the Schur complement can be written

$$\boldsymbol{S}_{BFBt}^{-1} := (\boldsymbol{B}\boldsymbol{C}^{-1}\boldsymbol{B}^\top)^{-1}(\boldsymbol{B}\boldsymbol{C}^{-1}\boldsymbol{A}\boldsymbol{D}^{-1}\boldsymbol{B}^\top)(\boldsymbol{B}\boldsymbol{D}^{-1}\boldsymbol{B}^\top)^{-1}, \tag{6.21}$$

where $\boldsymbol{C}$ and $\boldsymbol{D}$ are diagonal and symmetric positive definite matrices. The original choice sets $\boldsymbol{C}$ and $\boldsymbol{D}$ to the lumped velocity mass matrix of the system. New modifications have been

introduced in an effort to improve the effectiveness of the preconditioners in cases where the equations present high variability in the (scalar) coefficients [137, 138]. For the Poisson instead of the Stokes equations, $\boldsymbol{A}$ is the velocity mass matrix instead of a discrete Laplacian. Hence, with $\boldsymbol{C} = \boldsymbol{D} \approx \boldsymbol{A}$ the method reduces to $\boldsymbol{S}_{BFBt} \approx \boldsymbol{S}$, the usual Schur complement.

6.2 Multigrid for Saddle Point Problems

Multigrid (MG) methods provide efficient preconditioners for important classes sparse of linear systems, in particular if the matrix system arises from the discretization of an elliptic PDE. Their main advantage is that they scale linearly in the number of unknowns N, i.e. that they require only $O(N)$ computational work and memory. Multigrid has been primarily developed for symmetric positive M-matrices as they typically arise from FD/FV/FE discretizations of (scalar) second order elliptic PDEs.

As already indicated in the previous section, multigrid algorithms can be used inside Schur complement preconditioners to invert one (or both, depending on the application) of the diagonal blocks. While this approach allows an easy re-use of existing and well-established techniques, it does not longer guarantee linear convergence for the overall on the block system. The multigrid cycles are applied only to the sub-problem(s), while couplings between the unknowns are handled by the outer iteration (for example GMRES, BiCGStab or (inexact) Uzawa). In consequence, the outer iteration essentially determines the overall convergence speed, even if the inner sub-blocks can be solved quickly.

The question is whether it is possible to build a multigrid hierarchy for the coupled system to take into account the cross-couplings on all levels. Several developments have been made in this direction. Geometric multigrid methods for (Navier-)Stokes have been proposed in e.g. [161, 145]. In [163, 164, 165], this has been extended to a "semi-algebraic" AMGe-approach, where the coarse levels still have to determined geometrically. In this section, we will use the completely algebraic method introduced in [120]. As in classical AMG for M-matrices, this method only requires the matrix to build a robust multigrid hierarchy and hence can adapt itself to difficulties such as anisotropic or jumping coefficients as well as meshes that cannot be coarsened geometrically.

6.2.1 Algebraic multigrid (AMG)

The whole hierarchy of grids $\{\Omega_l\}_{l=1}^{L}$, operators $\{\boldsymbol{\Lambda}_l\}_{l=1}^{L}$, and transfer operators $\{\boldsymbol{P}_l\}_{l=1}^{L-1}$, $\{\boldsymbol{R}_l\}_{l=1}^{L-1}$ needed in the multigrid cycle (*solution phase*) is computed from the system matrix $\boldsymbol{\Lambda}$ during a *setup phase*. In this chapter we will understand the term 'grid' as a set of indices, AMG methods require no geometric mesh information. The strength of AMG is its ability to deal with difficulties in the operator, such as heterogeneous strongly varying coefficients as well as unstructured meshes, for which a hierarchy cannot be (easily) identified. To set the stage for our proposed method, let us briefly recapitulate the classical Ruge-Stüben AMG setup [139, 153]. We start on the finest level $l = 1$ using the fine system matrix $\boldsymbol{\Lambda}_1 := \boldsymbol{\Lambda}$ and the finest set of degrees of freedom indices $\Omega_1 = \{1, \ldots, n_1\}$.

1. We decompose the grid Ω_l into the set of *fine grid points* $\mathfrak{F}_l$ and *coarse grid points* $\mathfrak{C}_l$. The latter form the next coarser grid $\Omega_{l+1} := \mathfrak{C}_l$ of size n_{l+1}.

2. We compute the interpolation matrix

$$\boldsymbol{P}_l \in \mathbb{R}^{n_l \times n_{l+1}}, \quad \boldsymbol{P}_l := \begin{bmatrix} \boldsymbol{P}_{l,\mathfrak{F}} \\ \boldsymbol{I}_{\mathfrak{C}} \end{bmatrix}. \tag{6.22}$$

   The submatrix $\boldsymbol{P}_{l,\mathfrak{F}}$ contains the interpolation weights for all fine grid points $i \in \mathfrak{F}_l$. For the coarse grid points $i \in \mathfrak{C}_l$, interpolation is just the identity.

3. We compute the next coarser matrix by the Galerkin product

$$\boldsymbol{\Lambda}_{l+1} := \boldsymbol{P}_l^\top \boldsymbol{\Lambda}_l \boldsymbol{P}_l. \tag{6.23}$$

The algorithm is then recursively applied to the input matrix $\boldsymbol{\Lambda}_{l+1}$. If n_l is small enough such that $\boldsymbol{\Lambda}_l$ can be efficiently factored by a direct solver, the recursion is terminated. In our experiments we use a redundant serial solver if $n_l = 1000$.

We recall that for a fast multigrid algorithm, the error components not efficiently reduced by smoothing (the *smooth vectors*) must be well represented within the range of the interpolation operator $\boldsymbol{P}_l$. In AMG, we take a simple smoothing scheme like Jacobi or Gauss-Seidel relaxation. For symmetric positive definite M-matrices, an investigation of the smooth error components $\boldsymbol{e} = (e_1, \ldots, e_n)^\top$ reveals that these satisfy $\boldsymbol{D}^{-1}\boldsymbol{\Lambda}\boldsymbol{e} \approx 0$, where $\boldsymbol{D} := \operatorname{diag}(\boldsymbol{\Lambda})$. If we denote the coefficients of $\boldsymbol{\Lambda}$ by λ_{ij}, this means

$$e_i \approx -\frac{1}{\lambda_{ii}} \sum_{j \neq i} \lambda_{ij} e_j. \tag{6.24}$$

Equation (6.24) already delivers us a template for the interpolation formula: The value at the fine grid point i should be approximated by the values at those points j for which $-\lambda_{ij}$ is relatively large, while those (scaled) entries λ_{ij} also provide the interpolation weights. In consequence, a substantial amount of those large negative connections should lead (directly or indirectly) to coarse grid points $j \in \mathfrak{C}_l$. This imposes certain connections on the selection of the coarse grid points $\mathfrak{C}_l \subset \Omega_l$. Many algorithms to the coarse grid selection and construction of the interpolation have been proposed. We do not describe them in detail here, but refer to [139, 153, 85, 149, 148] The restriction matrix is taken as the transpose of the interpolation.

The Galerkin ansatz for the coarse grid matrix (6.23) has two benefits: First, for symmetric positive definite $\boldsymbol{\Lambda}_l$, $\boldsymbol{\Lambda}_{l+1}$ is also symmetric positive definite for any full (column) rank interpolation operator $\boldsymbol{P}_l$. Second, the resulting two-grid correction operator $\boldsymbol{I} - \boldsymbol{P}_l \boldsymbol{\Lambda}_{l+1}^{-1} \boldsymbol{P}_l \boldsymbol{\Lambda}_l$ is an orthogonal projector, which (extended recursively over all levels and combined with smoothing) ensures that the multigrid cycle converges [153].

We now present an algebraic multigrid approach for a monolithic solution of saddle point problems of the form

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{B}^\top \\ \boldsymbol{B} & -\boldsymbol{C} \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ p \end{bmatrix} = \begin{bmatrix} \boldsymbol{v} \\ q \end{bmatrix}. \tag{6.25}$$

We will construct a multigrid hierarchy of saddle point matrices indexed by l,

$$\mathcal{A}_l = \begin{bmatrix} \boldsymbol{A}_l & \boldsymbol{B}_l^\top \\ \boldsymbol{B}_l & -\boldsymbol{C}_l \end{bmatrix}, \tag{6.26}$$

as well as block interpolation matrices $\mathcal{P}_l$. Note that we include a lower right block $\boldsymbol{C}_l$, which will be zero on the first level, $\boldsymbol{C}_1 = 0$, and non-zero for $l \geq 2$.

6.2.2 Smoothers for saddle point problems

Classical relaxation schemes like the Jacobi or Gauss-Seidel iteration are not suitable for saddle point systems, since the smoothing properties for these schemes rely on the positive definiteness of the matrix. We present two dedicated smoothing schemes for saddle point systems [145]: First, a predictor-corrector algorithm, which combines segregated sweeps over the physical components, and second, a box relaxation scheme, where small saddle point subsystems are solved within a global Schwarz method.

Uzawa relaxation. Our first option is a symmetric inexact Uzawa relaxation scheme. Within each iteration, a predictor $\boldsymbol{u}^*$ for the flux is computed first, which is used to update $\boldsymbol{u}$. Finally, employing the updated values of p_l^{it+1}, the next iterate $\boldsymbol{u}_l^{it+1}$ is computed,

$$\boldsymbol{u}_l^* = \boldsymbol{u}^{it} + \hat{\boldsymbol{A}}_l^{-1}\left(\boldsymbol{v} - \boldsymbol{A}_l\boldsymbol{u}_l^{it} - \boldsymbol{B}_l^\top p_l^{it}\right), \tag{6.27a}$$

$$p_l^{it+1} = p_l^{it} + \hat{\boldsymbol{S}}_l^{-1}\left(\boldsymbol{B}_l\boldsymbol{u}_l^* - \boldsymbol{C}_l p_l^{it} - q\right), \tag{6.27b}$$

$$\boldsymbol{u}_l^{it+1} = \boldsymbol{u}_l^{it} + \hat{\boldsymbol{A}}_l^{-1}\left(\boldsymbol{v} - \boldsymbol{A}_l\boldsymbol{u}_l^{it} - \boldsymbol{B}_l^\top p_l^{it+1}\right). \tag{6.27c}$$

The matrices $\hat{\boldsymbol{A}}_l$ and $\hat{\boldsymbol{S}}_l$ are chosen such that $\hat{\boldsymbol{A}}_l - \boldsymbol{A}_l$ and $\hat{\boldsymbol{S}}_l - \boldsymbol{B}_l\hat{\boldsymbol{A}}_l^{-1}\boldsymbol{B}_l^\top - \boldsymbol{C}_l$ are symmetric positive definite. Furthermore, they should be easily invertible. For example, we can choose to use the scaled diagonals of $\boldsymbol{A}_l$ and $\boldsymbol{B}_l\hat{\boldsymbol{A}}_l^{-1}\boldsymbol{B}_l^\top + \boldsymbol{C}_l$, respectively. The magnitude of the scaling can be obtained with the help of the power iteration on the latter matrices. For convergence and further properties of this smoother we refer to [145].

Vanka smoothing. The second alternative requires us to first decompose the computational domain into small overlapping patches. To this end, we employ the non-zero structure of $\boldsymbol{B}_l$ and construct a patch $\Omega_{l,j}$ for each row of $\boldsymbol{B}_l$: For the j-th row, $\Omega_{l,j}$ consists of the index j as well as all indices i such that there exists an entry $b_{ji} \neq 0$

$$\Omega_{l,j} := \{i : \ b_{ji} \neq 0\} \times \{j\}. \tag{6.28}$$

The transfer between the global domain Ω_l and the subdomains $\Omega_{l,j}$ is accomplished by (optionally scaled) injection operators $\boldsymbol{V}_{l,j}$ (flux) and $\boldsymbol{W}_{l,j}$ (displacement),

$$\boldsymbol{V}_{l,j} = \operatorname{diag}(\boldsymbol{v}_{l,i})_{i=1,\dots,n_l}\boldsymbol{J}_j \tag{6.29}$$

$$\boldsymbol{W}_{l,j} = \boldsymbol{J}_j, \tag{6.30}$$

where $\boldsymbol{J}_j$ is a binary matrix with ones at the fine points and zero else. First, the residuals are restricted to the subdomain,

$$\boldsymbol{v}_{l,j} = \boldsymbol{V}_{l,j}^\top\left(\boldsymbol{v} - \boldsymbol{A}_l\boldsymbol{u}_l - \boldsymbol{B}_l^\top p_l\right), \tag{6.31a}$$

$$q_{l,j} = \boldsymbol{W}_{l,j}^\top\left(q - \boldsymbol{B}_l\boldsymbol{u}_l + \boldsymbol{C}_l^\top p_l\right). \tag{6.31b}$$

Then, on each subdomain, a small saddle point problem of the form

$$\begin{bmatrix} \hat{\boldsymbol{A}}_{l,j} & \boldsymbol{B}_{l,j}^\top \\ \boldsymbol{B}_{l,j} & \boldsymbol{B}_{l,j}\hat{\boldsymbol{A}}_{l,j}^{-1}\boldsymbol{B}_{l,j}^\top - \hat{\boldsymbol{S}}_{l,j} \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_{l,j} \\ p_{l,j} \end{bmatrix} = \begin{bmatrix} \boldsymbol{v}_{l,j} \\ q_{l,j} \end{bmatrix}, \tag{6.32}$$

is solved and finally the updates are prolongated to the global domain,

$$\boldsymbol{u}_l = \boldsymbol{u}_l + \boldsymbol{V}_{l,j}\boldsymbol{u}_{l,j}, \tag{6.33a}$$
$$p_l = p_l + \boldsymbol{W}_{l,j}p_{l,j}. \tag{6.33b}$$

The small matrices $\hat{\boldsymbol{A}}_{l,j}$, $\boldsymbol{B}_{l,j}$ and $\hat{\boldsymbol{S}}_{l,j}$ (the latter is just a scalar) are defined by

$$\hat{\boldsymbol{A}}_{l,j} = \boldsymbol{V}_{l,j}^\top \hat{\boldsymbol{A}}_l \hat{\boldsymbol{V}}_{l,j}, \tag{6.34a}$$
$$\boldsymbol{B}_{l,j} = \boldsymbol{W}_{l,j}^\top \boldsymbol{B}_l \hat{\boldsymbol{V}}_{l,j}, \tag{6.34b}$$
$$\hat{\boldsymbol{S}}_{l,j} = \beta^{-1}(\boldsymbol{C}_{l,j} + \boldsymbol{B}_{l,j}\hat{\boldsymbol{A}}_{l,j}^{-1}\boldsymbol{B}_{l,j}^\top). \tag{6.34c}$$

where $\boldsymbol{C}_{l,j}$ denotes the j-th diagonal entry of $\boldsymbol{C}_l$ and $\hat{\boldsymbol{V}}_{l,j} = \operatorname{diag}(\boldsymbol{v}_{l,i}^{-1})_{i=1,\dots,n_l}\boldsymbol{J}_j$, i.e. its scaling factors are inverse to those of (6.29). Again, $\hat{\boldsymbol{A}}_l$ is a scaled version of the diagonal of $\boldsymbol{A}_l$ such that $\hat{\boldsymbol{A}}_l - \boldsymbol{A}_l$ is positive definite. Likewise, $\beta > 0$ is chosen such that

$$\operatorname{diag}(\hat{\boldsymbol{S}}_{l,j}) - \left(\boldsymbol{C}_l + \boldsymbol{B}_l\hat{\boldsymbol{A}}_l^{-1}\boldsymbol{B}_l^\top\right) \tag{6.35}$$

is symmetric positive definite.

The iteration can be performed either additively (i.e., the residuals are computed once, then all subdomain solves are performed independently) or multiplicatively (after each subdomain solve the residuals are updated). In the latter case, in most cases it is beneficial to perform a symmetric sweep, i.e., after a complete sweep the subdomain solves are performed in reverse order. If we choose

$$\boldsymbol{v}_{l,i} = \frac{1}{\sqrt{|\{j : b_{ji} \neq 0\}|}}, \tag{6.36}$$

the additive smoother coincides with the Uzawa method described in the previous section [145]. On the other hand, in the multiplicative case the simple choice of

$$\boldsymbol{v}_{l,i} = 1 \tag{6.37}$$

may result in a faster convergence.

6.2.3 AMG setup for saddle point systems

The starting point for our saddle point AMG method (SPAMG) is the block diagonal matrix

$$\mathcal{B} = \begin{bmatrix} \boldsymbol{A}_l & 0 \\ 0 & \boldsymbol{B}_l\hat{\boldsymbol{A}}_l^{-1}\boldsymbol{B}_l^\top + \boldsymbol{C}_l \end{bmatrix}. \tag{6.38}$$

This operator is symmetric positive definite Let us assume that we can apply the classical AMG setup algorithm as described in Section 6.2.1 to each of the blocks $\boldsymbol{A}_l$ and $\boldsymbol{B}_l\hat{\boldsymbol{A}}_l^{-1}\boldsymbol{B}_l^\top + \boldsymbol{C}_l$. We hence construct coarse grids and interpolation operators $\boldsymbol{P}_{l,u}$ and $\boldsymbol{P}_{l,p}$ for each of these blocks and obtain a block interpolation operator

$$\mathcal{P}_l := \begin{bmatrix} \boldsymbol{P}_{l,u} & 0 \\ 0 & \boldsymbol{P}_{l,p} \end{bmatrix}. \tag{6.39}$$

Now, one idea would be to construct the coarse grid operator in the usual Galerkin way, $\mathcal{A}_{l+1} := \mathcal{P}_l^\top \mathcal{A}_l \mathcal{P}_l$. Unlike in the symmetric positive definite case, however, we cannot be sure that $\mathcal{A}_{l+1}$ is invertible, thus we might attain an unusable multigrid hierarchy. To prevent this, we modify the interpolation operator $\mathcal{P}_l$ by the multiplication of a stabilization term. To this end, we re-write the velocity prolongation block such that the rows corresponding to the fine grid points come first,

$$\boldsymbol{P}_{l,u} = \begin{bmatrix} \boldsymbol{P}_{l,u,CF} \\ \boldsymbol{I}_{C_l} \end{bmatrix}, \tag{6.40}$$

where $\boldsymbol{I}_{C_l}$ is the identity injecting the values from level $l+1$ to level l and $\boldsymbol{P}_{l,u,CF}$ contains the interpolation weights from coarse to fine. Analogously, we write

$$\boldsymbol{B}_l^\top = \begin{bmatrix} \boldsymbol{B}_{l,F}^\top \\ \boldsymbol{B}_{l,C}^\top \end{bmatrix}, \quad \hat{\boldsymbol{A}}_l^\top = \begin{bmatrix} \hat{\boldsymbol{A}}_{l,F}^\top & 0 \\ 0 & \hat{\boldsymbol{A}}_{l,C}^\top \end{bmatrix}. \tag{6.41}$$

The stabilized interpolation operator is computed by

$$\tilde{\mathcal{P}}_l := \begin{bmatrix} \boldsymbol{I}_{F_l} & 0 & -\hat{\boldsymbol{A}}_{l,F}^{-1} \boldsymbol{B}_{l,F}^\top \\ 0 & \boldsymbol{I}_{C_l} & 0 \\ 0 & 0 & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{P}_{l,u,CF} & 0 \\ \boldsymbol{I}_{C_l} & 0 \\ 0 & \boldsymbol{P}_{l,p} \end{bmatrix}. \tag{6.42}$$

Now, we use the modified Galerkin product

$$\mathcal{A}_{l+1} := \tilde{\mathcal{P}}_l^\top \mathcal{A}_l \tilde{\mathcal{P}}_l \tag{6.43}$$

to compute the coarse grid operator. For this matrix, we can show an inf-sup-condition if (a) the fine grid matrix $\mathcal{A}_l$ fulfills such a condition and (b) the interpolation operator $\boldsymbol{P}_{l,u}$ for the velocity satisfies certain approximation properties, which most usual AMG interpolation schemes do [120, Lemma 4.6]. The invertibility of the coarse grid matrix (6.43) is ensured by its lower right block

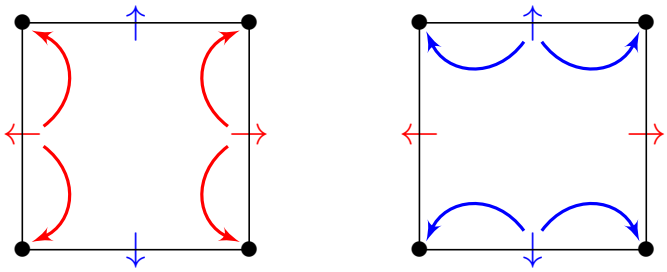
$$\boldsymbol{C}_{l+1} = \boldsymbol{P}_{l,p}^\top \left[\boldsymbol{C}_l + 2\boldsymbol{B}_{l,F} \hat{\boldsymbol{A}}_{l,F}^{-1} \boldsymbol{B}_{l,F}^\top + \boldsymbol{B}_{l,F} \hat{\boldsymbol{A}}_{l,F}^{-1} \boldsymbol{A}_{l,F} \hat{\boldsymbol{A}}_{l,F}^{-1} \boldsymbol{B}_{l,F}^\top \right] \boldsymbol{P}_{l,p}, \tag{6.44}$$

This can be understood as a partial Schur complement that ensures the stability of the coarse grid matrix.

6.3 Numerical Results

In this section, we evaluate the effectiveness of SPAMG compared to the diagonal and Schur preconditioners for uniform and adaptive meshes. We choose the examples based on manufactured solutions, that is, we prescribe a target pressure field and compute the velocity field, boundary conditions, and right hand sides in order to satisfy (6.7). Examples 1 to 3 are typical benchmarks with smooth coefficients. In example 4 we challenge the numerical solver and the preconditioner by using a conductivity tensor with strong coefficient variation. We employ three different flavors of smoothers inside SPAMG: The inexact Uzawa scheme (6.27), and two variants of the multiplicative Vanka smoother, either using the scaling (6.36) ("Vanka Scale"), or non-scaled injection (6.37) ("Vanka One").

We delegate the parallel mesh management to the octree-based AMR software library `p4est` [44, 94]. This library provides a collection of algorithms that implement scalable parallel AMR



(a) x components (b) y components

Figure 6.3: $\mathcal{RT}_0$ degrees of freedom identification for VTK visualization of a two dimensional vector field. Bold curved arrows indicate value copy.

operations. In particular, we employ `p4est` to create and modify a hexahedral triangulation of the unit square/cube and to introduce a numbering of the degrees of freedom suitable for a $\mathcal{RT}_0$ discretization. Regarding solvers and preconditioners, we use the GMRES [140] implementation provided by the software library `hypre` [155]. In order to approximate the product $\boldsymbol{Nr}$ for a given residual $\boldsymbol{r}$, we employ one SPAMG V-cycle. We use the parallel multigrid implementation BoomerAMG from `hypre` as the frame to which our saddle-point AMG is added.

We visualize the pressure and velocity fields arising from the $\mathcal{RT}_0$ discretization using the VTK file format. The pressure scalar field, being approximated by discontinuous piecewise constants is trivially written as cell data. For the velocity vector field, we require to save it as point data. The VTK routines require information for each element corner, while there are 2^d corners per element, we have 2 velocity degrees of freedom per coordinate direction for $d = 2, 3$. We have addressed this mismatch by replicating the computed velocity value to the corners of the face containing it. In Figure 6.3 we show the result for a two dimensional vector field.

6.3.1 Homogeneous Dirichlet conditions

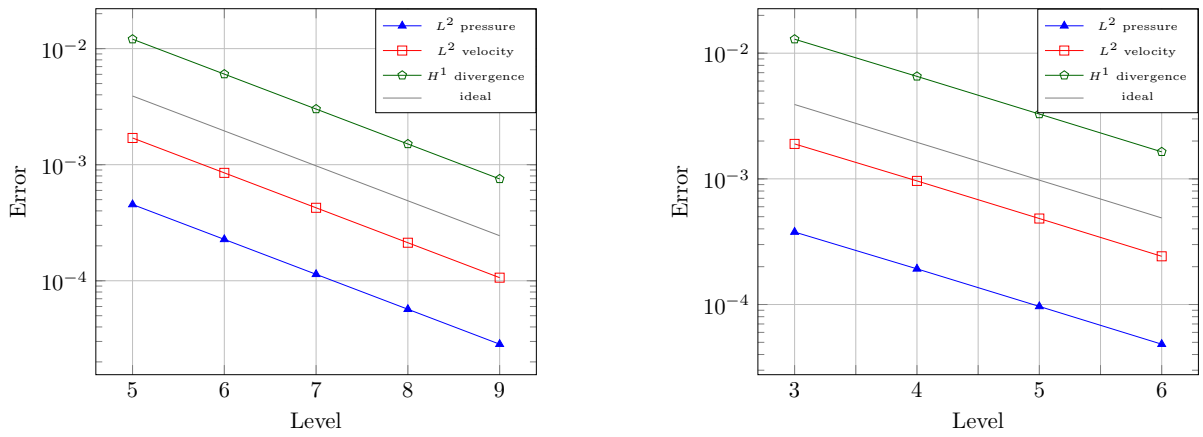
We solve (6.7) with an identity conductivity tensor and homogeneous Dirichlet boundary conditions. We compute the right hand side based on the manufactured solution

$$p(x, y) = (x^2 - x^3)(y^2 - y^3) \tag{6.45}$$

in $2d$ and

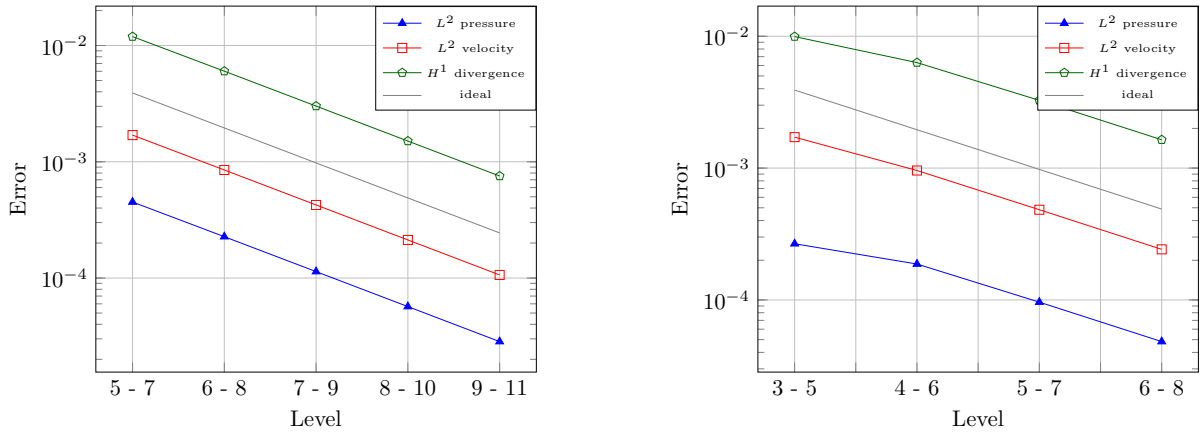
$$p(x, y, z) = (x^2 - x^3)(y^2 - y^3)(z - z^2) \tag{6.46}$$

in $3d$, respectively. With this example we verify the correctness of our implementation of the $\mathcal{RT}_0$ discretization. The discretization error converges the predicted rates as confirmed in Figure 6.4 and Figure 6.5. The iteration counts displayed in Table 6.1 and Table 6.2 confirm the (well known) robustness of the Schur preconditioner for uniform meshes. For adaptive meshes, the Schur complement preconditioner incurs high iteration counts, particularly for the 3d case. The three variants of the SPAMG preconditioner retain mesh independent iteration counts for both uniform and adaptive meshes in this example.



(a) (b)

Figure 6.4: Error plot for the numerical solution of a mixed Poisson system corresponding to the example specified in Section 6.3.1 (homogeneous Dirichlet boundary conditions) in two (a) and three dimensions (b) for uniform meshes and identity conductivity tensor. The level ℓ is related to the mesh size h via $h = 2^{-\ell}$. We confirm the expected convergence rates predicted by (6.16).



(a) (b)

Figure 6.5: Error plot for the numerical solution of a mixed Poisson system corresponding to the example specified in Section 6.3.1 in two (a) and three dimensions (b) for adaptive meshes and identity conductivity tensor. We choose to refine an element of side length h by two additional levels whenever its centroid lies within the circle/sphere of radius $2^d h^2$ around the point $(\frac{1}{2}, \frac{1}{2})$ for $d = 2$ and $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ for $d = 3$. Because of the smoothness of the solution and the equations coefficients we do not expect that local refinement translates into an improvement of the approximation with respect to a uniform case mesh. (Figure 6.4).

Level	# Iterations					
	NoPC	Diag	Schur	SPAMG		
				Uzawa	Vanka one	Vanka scale
4	459	299	22	9	8	8
5	>1000	773	22	10	8	8
6	-	>1000	24	12	9	10
7	-	-	24	14	10	10
8	-	-	24	14	10	11
9	-	-	24	14	10	11

(a)

Level	# Iterations					
	NoPC	Diag	Schur	SPAMG		
				Uzawa	Vanka one	Vanka scale
$4-7$	>1000	>1000	87	10	7	7
$5-8$	-	-	87	10	7	8
$6-9$	-	-	99	13	10	10
$7-10$	-	-	98	13	10	10
$8-11$	-	-	112	15	11	11
$9-12$	-	-	149	15	11	11

(b)

Table 6.1: Number of iterations required by the GMRES solver for a two dimensional mixed Poisson system (Section 6.3.1) on a uniform (a) and adaptive mesh (b). The relative tolerance of the linear solve is fixed to 10^{-6}. We display iteration counts employing no preconditioning in column two, a diagonal lumped mass matrix in column three, a Schur complement preconditioner in column four and three different smoothers in the SPAMG preconditioner in columns five to seven from each table. The level ℓ is related to the mesh size h via $h = 2^{-\ell}$.

Level	# Iterations			
	Schur	SPAMG		
		Uzawa	Vanka one	Vanka scale
3	20	9	7	7
4	22	11	8	8
5	22	13	9	9
6	24	14	9	10
7	24	15	11	12

(a)

Level	# Iterations			
	Schur	SPAMG		
		Uzawa	Vanka one	Vanka scale
$3-6$	238	11	8	8
$4-7$	322	11	8	8
$5-8$	338	13	9	9
$6-9$	383	14	10	10
$7-10$	390	17	12	12

(b)

Table 6.2: Number of iterations required by the GMRES solver for a three dimensional mixed Poisson (Section 6.3.1) discretized on a uniform (a) and adaptive mesh (b). We use the same setup as in Table 6.1.

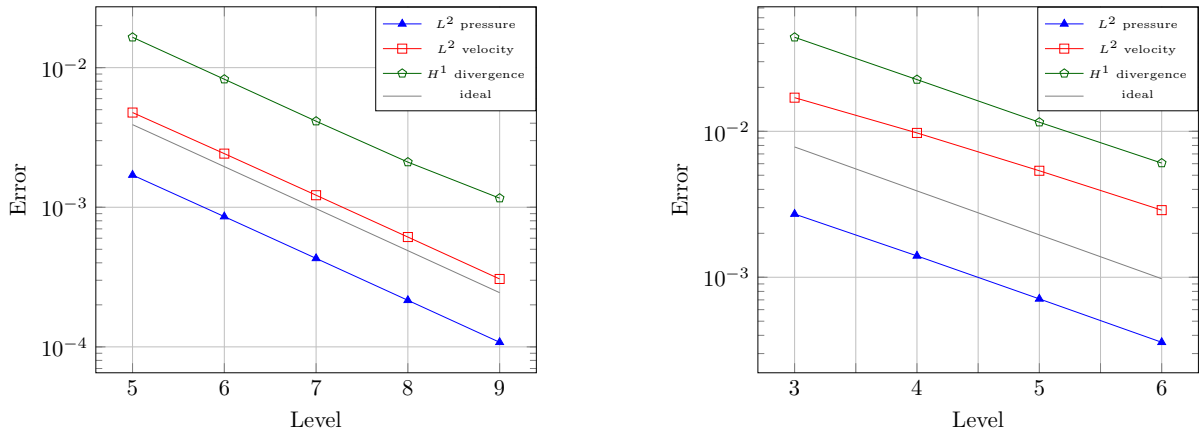


Figure 6.6: Error plot for the numerical solution of a mixed Poisson system specified in Section 6.3.2 (inhomogeneous Dirichlet/Neumann boundary conditions) in two (a) and three (b) dimensions for uniform meshes.

6.3.2 An example with Neumann conditions

We solve (6.7) with an identity tensor. We impose homogeneous mixed homogeneous Dirichlet / Neumann boundary conditions and compute the right hand side based on the exact solution

$$p(x, y) = xy(1 - y)(1 - x)^2 \tag{6.47}$$

in $2d$ and

$$p(x, y, z) = xy(1 - y)(1 - x)^2(1 - z) \tag{6.48}$$

in $3d$, respectively. The Neumann boundary is set at $y = 0$ and $y = 1$ in both cases. As in the previous section, the theoretical converge rates agree with the bounds (6.16). Due to the smoothness solution and the equation constant coefficient, no additional benefit is expected from local adaptation of the mesh; see Figure 6.6 and Figure 6.7.

6.3.3 Non trivial conductivity tensor

In this example we approximate the solution of (6.7) for the case of a non-diagonal conductivity tensor $\mathcal{K}$. We impose non-zero Dirichlet boundary conditions. The Manufactured solution is

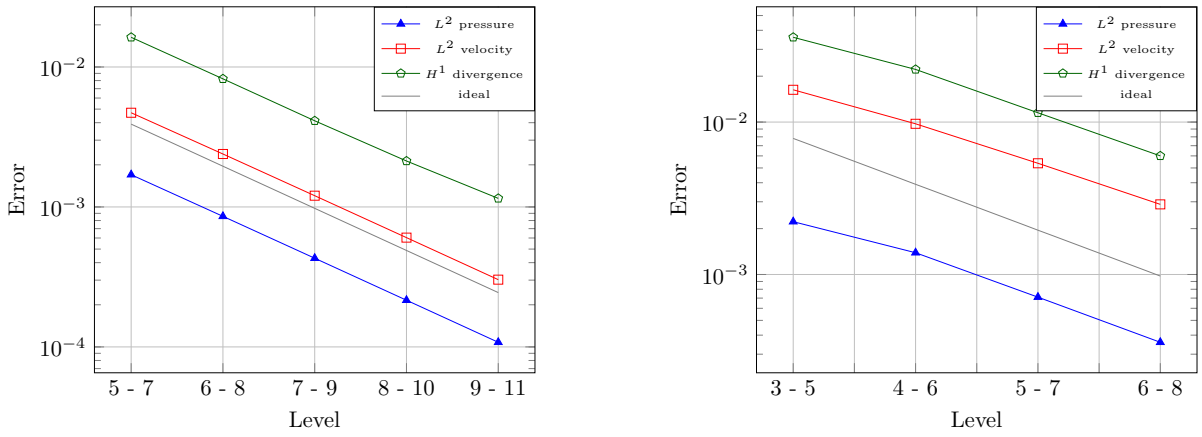
$$p(x, y) = e^x \sin(y) \tag{6.49}$$

in $2d$ and

$$p(x, y, z) = e^x \sin(y)(1 + z^2) \tag{6.50}$$

in $3d$, respectively. The conductivity tensor is given by

$$\mathcal{K}(x, y) = \begin{pmatrix} e^{x/2+y/4} & \sin(2\pi x) \\ \sin(2\pi x) & e^{x/4+y/2} \end{pmatrix} \tag{6.51}$$



(a) (b)

Figure 6.7: Error plot for the numerical solution of a mixed Poisson system specified in Section 6.3.2 in two (a) and three (b) dimensions for adaptive meshes. The refinement criteria is chosen as in Figure 6.5.

Level	# Iterations					
	NoPC	Diag	Schur	SPAMG		
				Uzawa	Vanka one	Vanka scale
4	>1000	>1000	85	18	12	11
5	-	-	102	20	13	13
6	-	-	113	23	14	14
7	-	-	130	24	16	15
8	-	-	142	25	16	17
9	-	-	169	25	17	18

(a)

Level	# Iterations					
	NoPC	Diag	Schur	SPAMG		
				Uzawa	Vanka one	Vanka scale
$4-7$	>1000	>1000	250	18	11	10
$5-8$	-	-	255	19	13	12
$6-9$	-	-	415	22	14	13
$7-10$	-	-	338	24	15	15
$8-11$	-	-	394	24	16	16
$9-12$	-	-	539	24	16	17

(b)

Table 6.3: Number of iterations required by the GMRES solver for a two dimensional mixed Poisson system (Section 6.3.3) discretized in a uniform (a) and adaptive mesh (b). We use the same setup as in Table 6.1.

in $2d$ and

$$\boldsymbol{\mathcal{K}}(x,y,z) = \begin{pmatrix} e^{x/2+y/4} & \sin(2\pi x) & 0 \\ \sin(2\pi x) & e^{x/4+y/2} & 0 \\ 0 & 0 & e^{z} \end{pmatrix} \tag{6.52}$$

in $3d$.

The example presented in this section aims to test the effectiveness of the SPAMG preconditioner for a full conductivity tensor example. The three variants of the SPAMG preconditioner offer mesh independent iteration counts for uniform and adaptive meshes. The Schur complement again produces growing iteration counts, in particular for the three dimensional case and even for uniform meshes. See Table 6.3 and Table 6.4.

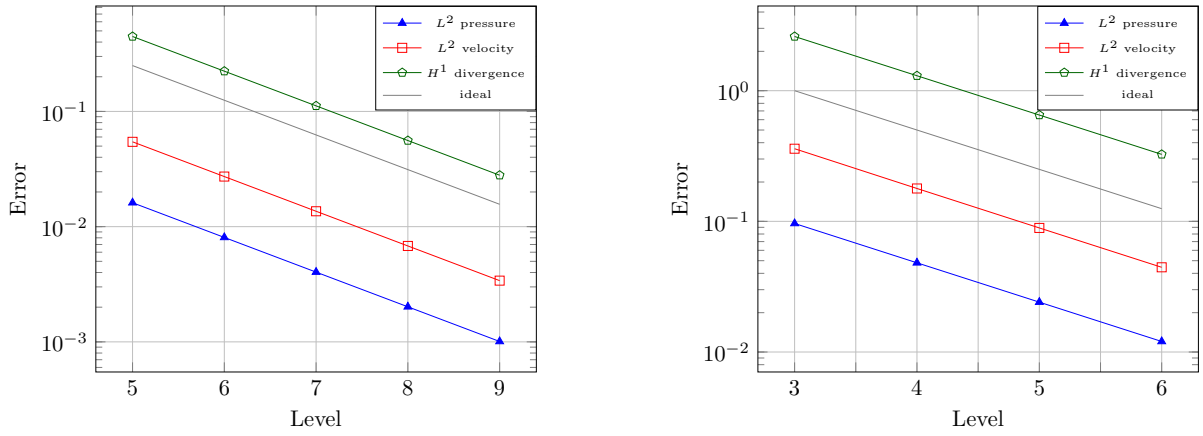


Figure 6.8: Error plot for the numerical solution of a mixed Poisson system specified in Section 6.3.3 in two (a) and three (b) dimensions for uniform meshes.

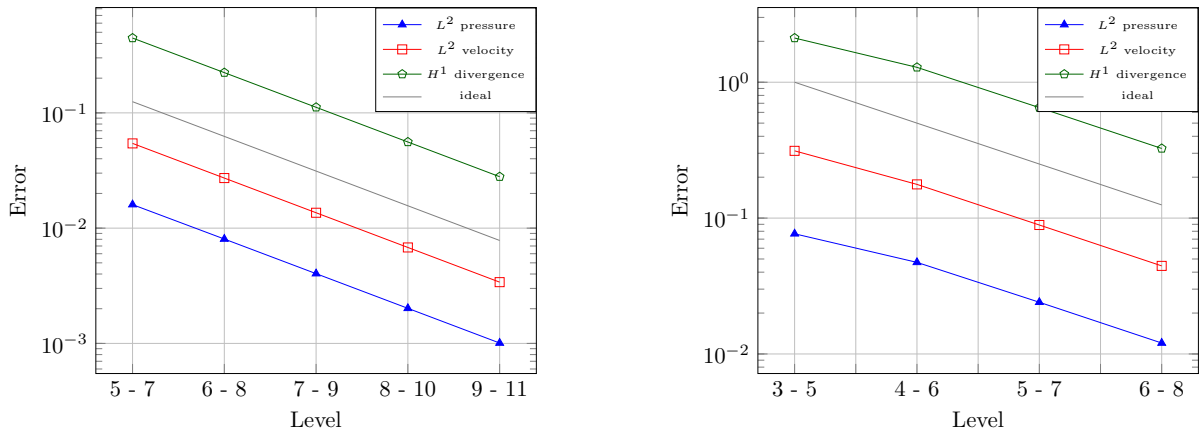


Figure 6.9: Error plot for the numerical solution of a mixed Poisson system specified in Section 6.3.3 in two (a) and three (b) dimensions for adaptive meshes. The refinement criteria is chosen as in Figure 6.5.

Level	# Iterations			
	Schur	SPAMG		
		Uzawa	Vanka one	Vanka scale
3	65	19	13	13
4	84	21	14	14
5	103	23	15	16
6	113	24	17	17
7	128	26	18	19

(a)

Level	# Iterations			
	Schur	SPAMG		
		Uzawa	Vanka one	Vanka scale
$3-6$	330	22	14	14
$4-7$	456	23	15	14
$5-8$	486	24	16	15
$6-9$	471	25	17	18
$7-10$	493	27	18	20

(b)

Table 6.4: Number of iterations required by the GMRES solver for a three dimensional mixed Poisson system (Section 6.3.3) discretized on a uniform (a) and adaptive mesh (b). We use the same setup as in Table 6.1.

6.3.4 High conductivity contrast

We solve (6.7) with a conductivity tensor that exhibits strong coefficient variation. We enforce inhomogeneous mixed Dirichlet boundary conditions and compute the right hand side terms based on the manufactured solution

$$p(x, y) = \sin(x)e^y \tag{6.53}$$

in $2d$ and

$$p(x, y, z) = \sin(x)e^y(1 + z^2) \tag{6.54}$$

in $3d$. The conductivity Tensor $\boldsymbol{\mathcal{K}}$ is given by a identity matrix scaled pointwise by a continuously differentiable function $m(\boldsymbol{x}; \boldsymbol{x}_0, a, b, c)$ constructed to fulfil the following properties:

- $m(\boldsymbol{x}; \boldsymbol{x}_0, a, b, c) = 1$ if $\|\boldsymbol{x} - \boldsymbol{x}_0\| \geq b$,
- $m(\boldsymbol{x}; \boldsymbol{x}_0, a, b, c) = 1 - c$ if $\|\boldsymbol{x} - \boldsymbol{x}_0\| \leq a$ and
- $m(\boldsymbol{x}; \boldsymbol{x}_0, a, b, c) \in (1 - c, 1)$ if $a < \|\boldsymbol{x} - \boldsymbol{x}_0\| < b$.

Such a function can be constructed by defining

$$h(t) := \begin{cases} e^{-1/t} & \text{if } t > 0, \\ 0 & \text{else} \end{cases} \tag{6.55}$$

and

$$m(\boldsymbol{x}; \boldsymbol{x_0}, a, b, c) := 1 - c\frac{h(b - \|\boldsymbol{x} - \boldsymbol{x}_0\|)}{h(b - \|\boldsymbol{x} - \boldsymbol{x}_0\|) + h(\|\boldsymbol{x} - \boldsymbol{x}_0\| - a)}. \tag{6.56}$$

Hence, the first three arguments define the location and radius of the support of m. The parameter c allows us to tune the coefficients to vary across the domain. From an application point of view, if c is close to one the function m models a medium in which almost no flow is allowed within a circle (sphere) centered at $\boldsymbol{x}_0$ with radius a. Given this information, it is clear that the velocity field is likely to have a strong gradient within the ring-shaped region $a < \|\boldsymbol{x} - \boldsymbol{x_0}\| < b$. Therefore, we choose to refine a given element on the mesh whenever it overlaps this region.

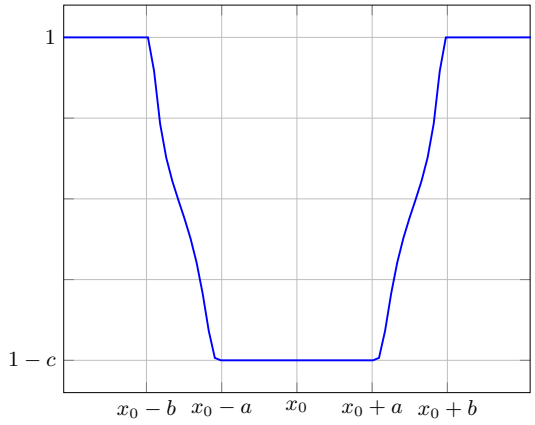
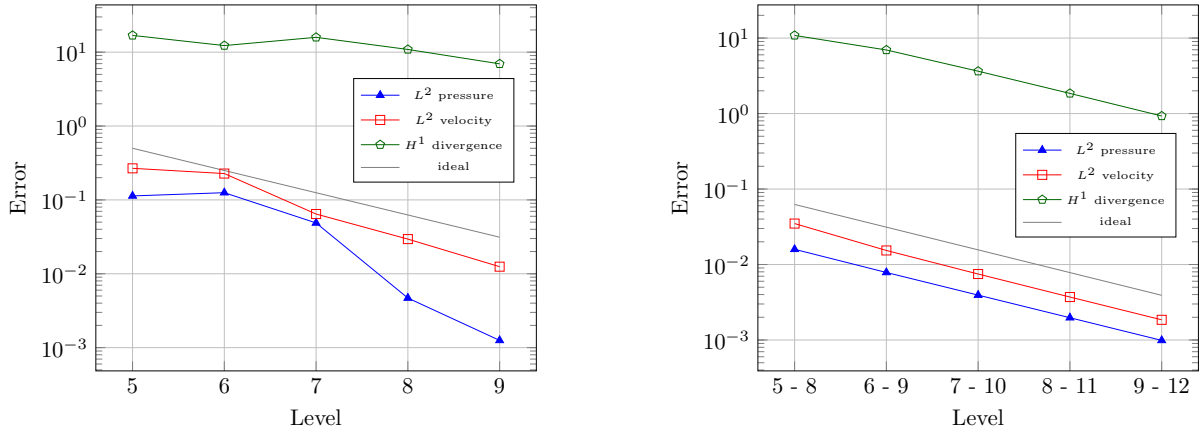


Figure 6.10: Sample plot of the one dimensional version of $m(\boldsymbol{x}; \boldsymbol{x_0}, a, b, c)$.



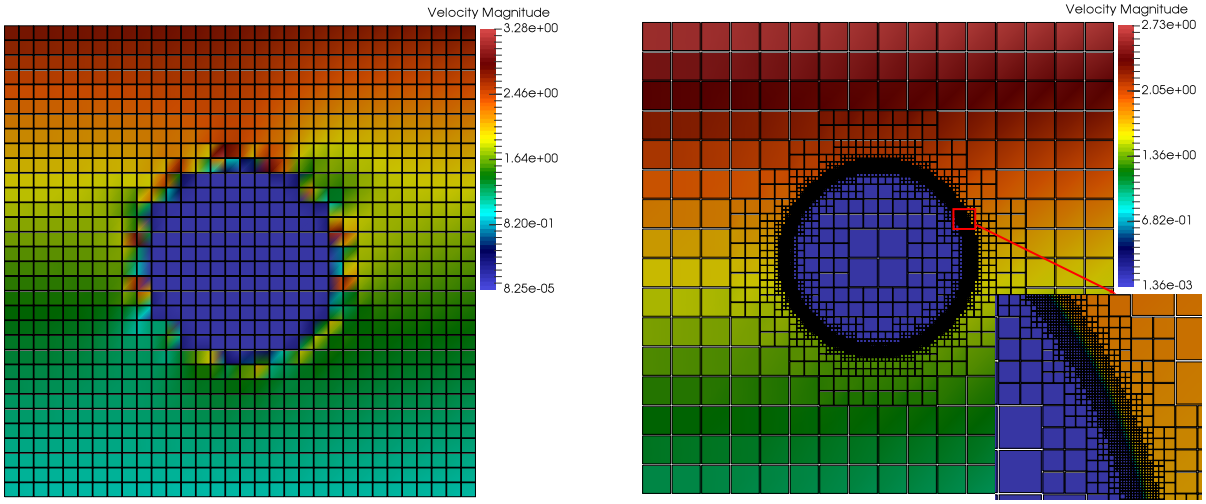
(a) (b)

Figure 6.11: Error plot for the numerical solution of a mixed Poisson system corresponding to the example defined in Section 6.3.4 (high conductivity contrast) in two dimensions for uniform (a) and adaptive (b) meshes.

This example targets effectiveness of the SPAMG preconditioner in the presence of high conductivity contrast and thus closer to what is encounter in practice. The three variants of the SPAMG preconditioner offer mesh independent iteration counts for uniform and adaptive meshes. The Schur complement produces growing iteration counts, in particular when we activate the local refinement to resolve the strong gradient in the region $a < \|\boldsymbol{x} - \boldsymbol{x_0}\| < b$. This behavior causes that the GMRES solver fails to reduce the error within the prescribed maximum number of iterations. See Table 6.5 and Table 6.6.

6.4 Discussion

The numerical examples exposed in this document show that the SPAMG preconditioner yields iteration counts which are nearly independent of the mesh size. This behavior holds for uniform and locally refined meshes. This allows us to profit from the effectiveness of AMR: Less variables



(a) (b)

Figure 6.12: Velocity magnitude for the numerical solution corresponding to the example of Section 6.3.4 in two dimensions for a uniform level 5 mesh (a) and an adaptive mesh from level 4 to 10 (b).

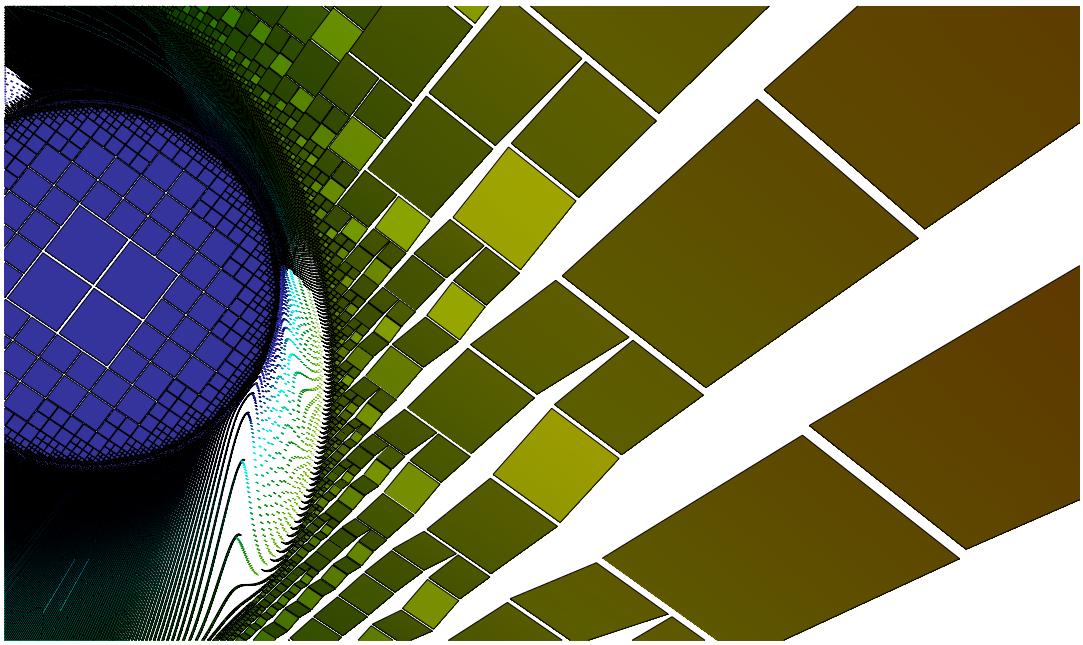
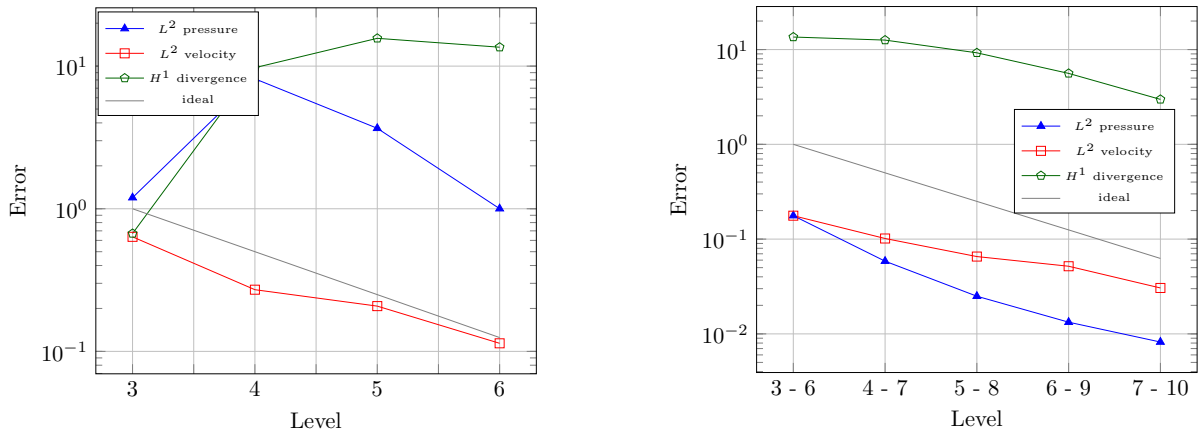


Figure 6.13: y-velocity magnitude extrusion illustrating a two dimensional $\mathcal{RT}_0$ vector field for a level 4 to 10 adaptively refined mesh. By construction, the y-velocity component is continuous in the y-direction and discontinuous in the x-direction.



(a) (b)

Figure 6.14: Error plot for the numerical solution corresponding to the example defined in Section 6.3.4 (high contrast, $c = 0.999$) in three dimensions for uniform (a) and adaptive (b) meshes.

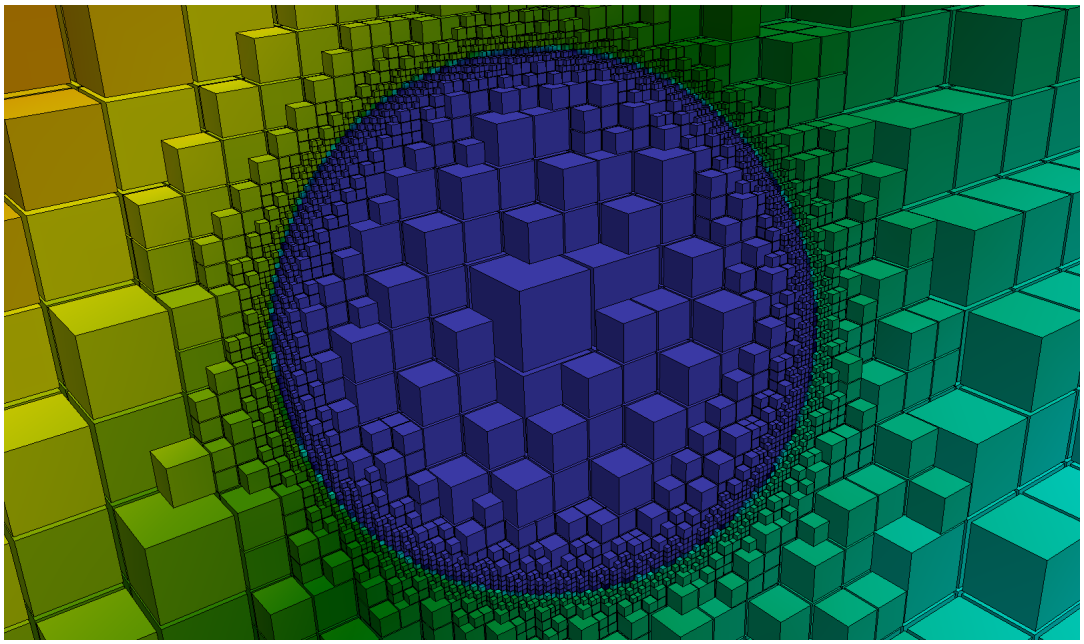
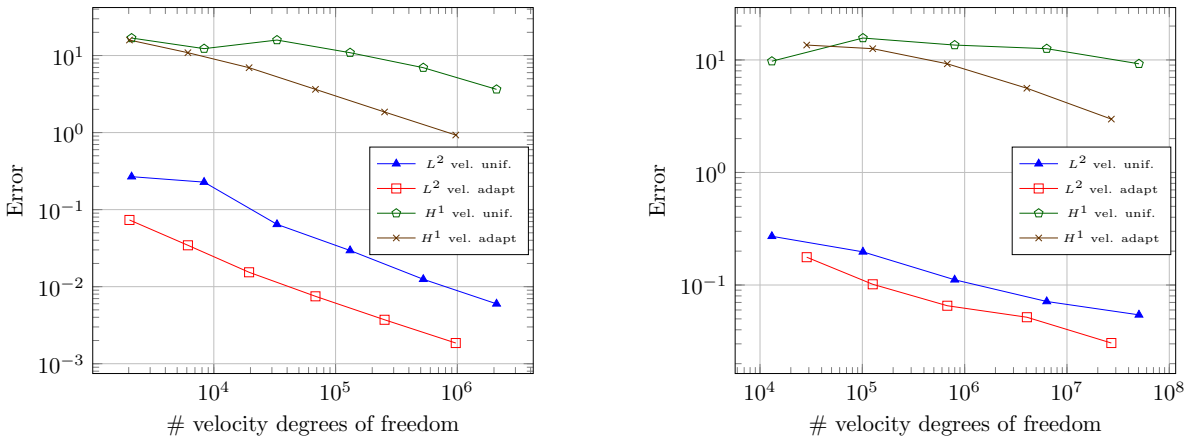


Figure 6.15: Threshold plot from the computed velocity of a three dimensional version of the example defined in Section 6.3.4. An adaptive mesh from level 4 to 8 was used in the calculation.



(a) (b)

Figure 6.16: For example defined in Section 6.3.4, we compare the number of degrees of freedom with uniform and adaptive meshes against the L^2 and H^1 errors of the velocity for a two (a) and tree dimensional problem (b).

Level	# Iterations					
	NoPC	Diag	Schur	SPAMG		
				Uzawa	Vanka one	Vanka scale
4	>1000	>1000	36	12	8	9
5	-	-	56	14	9	8
6	-	-	55	14	9	10
7	-	-	47	15	10	11
8	-	-	43	15	10	11
9	-	-	41	15	11	11

(a)

Level	# Iterations					
	NoPC	Diag	Schur	SPAMG		
				Uzawa	Vanka one	Vanka scale
4 – 7	>1000	>1000	379	12	8	8
5 – 8	-	-	547	13	8	8
6 – 9	-	-	930	13	9	9
7 – 10	-	-	>1000	15	10	10
8 – 11	-	-	-	15	10	10
9 – 12	-	-	-	16	10	11

(b)

Table 6.5: Number of iterations required by the GMRES solver for a two dimensional mixed Poisson system defined by the example in Section 6.3.4 discretized on a uniform (a) and adaptive mesh (b).

Level	# Iterations			
	Schur	SPAMG		
		Uzawa	Vanka one	Vanka scale
3	40	13	9	9
4	45	13	8	9
5	56	15	9	9
6	53	15	10	10
7	47	15	11	11

(a)

Level	# Iterations			
	Schur	SPAMG		
		Uzawa	Vanka one	Vanka scale
3 – 6	637	15	11	11
4 – 7	855	14	10	11
5 – 8	>1000	15	11	11
6 – 9	-	16	12	12
7 – 10	-	20	14	15

(b)

Table 6.6: Number of iterations required by the GMRES solver for a three dimensional mixed Poisson system defined by the example in Section 6.3.4 discretized in a uniform (a) and adaptive mesh (b). We use the same setup as in Table 6.1.

compared to the uniform mesh case lead to a better approximation of the expected solution (e.g., Figure 6.16). Regarding the Schur complement strategy, our results show that in the presence of adaptive meshes denigrates the effectiveness of the preconditioner. Hence, SPAMG can be a valuable alternative as preconditioner for saddle point systems arising from a MFE discretization of a second order elliptic problems. In future work we would like to investigate the robustness of SPAMG with respect to more strongly heterogeneous and anisotropic coefficients and compare with newly developed strategies such as the auxiliary space multigrid presented in [105]. Additionally, we would like to extend our implementation to test the behaviour of the SPAMG with higher order Raviart-Thomas elements and investigate additional choices of the matrices $\boldsymbol{C}$ and $\boldsymbol{D}$ in the BFBt preconditioner (6.21). For example, motivated by the work in [137], we are interested in the choice $\boldsymbol{C} = \boldsymbol{D} = \sqrt{\boldsymbol{A}}$.

7 Conclusion and Outlook

In this thesis we are concerned with the development of highly accurate simulation tools for variably saturated flow through porous media. Our starting point is the parallel subsurface flow simulator ParFlow.

We begin by studying ParFlow's management of the computational mesh in detail, focusing on how its parallelization enables but also limits parallel scalability. We extend the parallel code ParFlow such that it can scale to the size of today's supercomputer installations. We achieve this by reorganizing the parallel mesh subsystem to rely on the fast partitioning algorithms provided by the software `p4est`. This requires tweaks to the logic of ParFlow's index arithmetic and a reinterpretation of some data structures. As a main result, there is no longer any need to store the entire mesh connectivity information on each processor. This has significantly reduced ParFlow's memory footprint, effectively removing the memory bottleneck that had previously prevented ParFlow from running at extreme scales. It has also reduced the setup time from as much as 40 minutes at 32k cores previously to several seconds at 458k with our modified version.

The above developments improve the scalability of ParFlow such that it has become a true petascale application, with successful demonstrations running on the entire 28-rack IBM BlueGene/Q system at the Jülich Supercomputing Centre (JSC). As a result, the modified version of ParFlow has been accepted into the High-Q-Club at the JSC [97]. Due to our changes to the code being local and transparent, the modified version of ParFlow is backwards compatible with the upstream version. This work will be merged into the public version of ParFlow in the near future, making the extended speed and scalability available to the greater public.

We note that we did not address the algorithmic efficiency of the time stepper or the preconditioner and the I/O subsystem. The mathematics of the solver remain unchanged. Future developments in this regard will automatically inherit and benefit from the improvements in scalability presented in this thesis. Regarding the I/O implemented in ParFlow, we investigate its scalability and conclude that it produces excessive overhead and limits the size of simulations that can be analyzed visually [37]. Future work in this subsystem will be crucial to fully benefit from our developments and the upcoming supercomputers.

Another contribution of this thesis is the algorithmic approach to prepare the usage of locally refined meshes in ParFlow. AMR allows meshes where elements of different size neighbor each other. In this case, we notice that ParFlow incurs erroneous results when it attempts to communicate data between inter-element boundaries. We propose and discuss two solutions to this issue operating at two different levels: The first manipulates the indices of the degrees of freedom. While the second operates directly on the degrees of freedom. In our opinion, both options can be implemented without introducing disruptive changes to the code. Here, it is important to point out that this effort constitutes a first step in establishing the data representation of the code and numerical mathematics context. For example, we make no statement of which approach is better suited for implementation and did not address how do we want to interpolate fluxes at the boundary of two subgrids with different mesh spacing. The latter question has to be postponed until the decision on the discretization method has been

made. Results on the equivalence between specific MFE and certain FD schemes [6] motivate us to consider a MFE discretization based on low order Raviart-Thomas elements [135] for ParFlow. We will investigate this in future work.

Lastly, we present the SPAMG preconditioner designed by our co-author B. Metsch in [120]. With numerical examples we demonstrate that it displays iteration counts that are nearly independent of the mesh size for a mixed formulation of a diffusion equation. The value of this result is that it holds for uniform and locally refined meshes and various classes of coefficients (such as the conductivity tensor). In future work we would like to test the preconditioner with more general coefficients than the ones presented in this thesis and to compare it with the most recent strategies like the auxiliary space multigrid from [105]. Another important item for future research is to evaluate the parallel scalabilty of the SPAMG preconditioner. Finally, relating the work presented in the last chapter of this thesis with our research with ParFlow, a full AMR implementation will require preconditioners with the robustness that SPAMG displays with respect to the mesh refinement and equation coefficients. The actual preconditioners in ParFlow are not well suited for the SFC approach to mesh adaptation, hence an implementation of a preconditioner like SPAMG will be a key ingredient in a fully operational version of ParFlow with AMR.

Acknowledgments

I would like to express my deep gratitude to my advisor Prof. Dr. Carsten Burtedde, who introduced me to the topics of scientific and high performance computing. I would like to thank him for his support and excellent guidance throughout the years I spent working in his group. Furthermore, I am thankful to Prof. Dr. Stefan Kollet for co-reviewing this thesis.

I gratefully acknowledge the support of the University of Costa Rica, the Bonn International Graduate School of Mathematics (BIGS), the collaborative research initiative TR32 at the University of Bonn, and the Jülich supercomputing centre.

I would like to express my very great appreciation to Prof. William Alvarado at the University of Costa Rica for motivating me to pursue my graduate studies abroad. I also want to thank my friends in Costa Rica for their constant support, in particular those from my home town Aserrí (grito Poaseño goes here!). Our social media groups were an important source of motivation for me.

I am indelibly thankful to Isa for her support, comprehension and patience during all these years. Those weekends where each one worked on its own dissertation belong to the most productive time in the preparation of this document. Thanks for all the traveling adventures during our free time, with a special mention: Θεσσαλονίκη. For helping me to improve my German and for all the bread that we baked together. Ενχαριστώ πάρα πολύ!

Finally, thanks to my beloved parents for supporting me during this time, this work is dedicated to them.

Bibliography

[1] I. Aavatsmark, *An introduction to multipoint flux approximations for quadrilateral grids*, Computational Geosciences, 6 (2002), pp. 405–432.

[2] M. Ainsworth and B. Senior, *Aspects of an adaptive hp-finite element method: Adaptive strategy, conforming approximation and efficient solvers*, Computer Methods in Applied Mechanics and Engineering, 150 (1997), pp. 65–87.

[3] G. M. 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, AFIPS '67 (Spring), New York, NY, USA, 1967, ACM, pp. 483–485.

[4] T. Arbogast, L. C. Cowsar, M. F. Wheeler, and I. Yotov, *Mixed finite element methods on non-matching multiblock grids*, SIAM Journal on Numerical Analysis, 37 (2000), pp. 1295–1315.

[5] T. Arbogast, M. F. Wheeler, and I. Yotov, *Mixed finite elements for elliptic problems with tensor coefficients as cell-centered finite differences*, SIAM Journal on Numerical Analysis, 34 (1997), pp. 828–852.

[6] T. Arbogast, M. F. Wheeler, and N.-Y. Zhang, *A nonlinear mixed finite element method for a degenerate parabolic equation arising in flow in porous media*, SIAM Journal on Numerical Analysis, 33 (1996), pp. 1669–1687.

[7] D. Arnold, R. Falk, and R. Winther, *Preconditioning in $H(\mathrm{div})$ and applications*, Mathematics of Computation of the American Mathematical Society, 66 (1997), pp. 957–984.

[8] D. N. Arnold, R. S. Falk, and R. Winther, *Multigrid in $H(\mathrm{div})$ and $H(\mathrm{curl})$*, Numerische Mathematik, 85 (2000), pp. 197–217.

[9] S. F. Ashby and R. D. Falgout, *A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations*, Nuclear Science and Engineering, 124 (1996), pp. 145–159.

[10] I. Babuska and W. C. Rheinboldt, *Error estimates for adaptive finite element computations*, SIAM Journal on Numerical Analysis, 15 (1978), pp. 736–754.

[11] M. Bader, *Space-Filling Curves: An Introduction with Applications in Scientific Computing*, Texts in Computational Science and Engineering, Springer, 2012.

[12] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, *Algorithms and data structures for massively parallel generic adaptive finite element codes*, ACM Transactions on Mathematical Software, 38 (2011), pp. 14:1–14:28.

[13] G. K. Batchelor, *An introduction to fluid dynamics*, Cambridge University Press, New York, NY, 2000.

[14] J. Bear, *Dynamics of fluids in porous media*, Dover Publications, Inc., 1972.

[15] J. Bear and Y. Bachmat, *Introduction to Modeling of Transport Phenomena in Porous Media*, Kluwer, Dordrecht-Boston-London, 1991.

[16] J. Bear and A. H.-D. Cheng, *Modeling groundwater flow and contaminant transport*, vol. 23, Springer Science & Business Media, 2010.

[17] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, *A User's Guide to PVM Parallel Virtual Machine*, tech. rep., University of Tennessee, Knoxville, TN, USA, 1991.

[18] M. Benzi, G. H. Golub, and J. Liesen, *Numerical solution of saddle point problems*, Acta Numerica, 14 (2005), pp. 1–137.

[19] M. J. Berger and P. Colella, *Local adaptive mesh refinement for shock hydrodynamics*, Journal of Computational Physics, 82 (1989), pp. 64–84.

[20] C.-E. Bichot and P. Siarry, *Graph partitioning*, John Wiley & Sons, 2013.

[21] M. F. P. Bierkens, *Global hydrology 2015: State, trends, and directions*, Water Resources Research, 51 (2015), pp. 4923–4947.

[22] M. F. P. Bierkens, V. A. Bell, P. Burek, N. Chaney, L. E. Condon, C. H. David, A. Roo, P. Dll, N. Drost, J. S. Famiglietti, M. Flrke, D. J. Gochis, P. Houser, R. Hut, J. Keune, S. Kollet, R. M. Maxwell, J. T. Reager, L. Samaniego, E. Sudicky, E. H. Sutanudjaja, N. Giesen, H. Winsemius, and E. F. Wood, *Hyper-resolution global hydrological modelling: what is next?*, Hydrological Processes, 29 (2014), pp. 310–320.

[23] D. Boffi, F. Brezzi, and M. Fortin, *Mixed finite element methods and applications*, Springer, 2013.

[24] D. Braess, *Finite Elements. Theory, Fast Solvers, and Applications in Solid Mechanics*, Cambridge University Press, Cambridge, New York, 1997.

[25] D. Braess and R. Verfürth, *A posteriori error estimators for the Raviart-Thomas element*, SIAM Journal on Numerical Analysis, 33 (1996), pp. 2431–2444.

[26] J. H. Bramble, J. E. Pasciak, and A. T. Vassilev, *Analysis of the ineact Uzawa algorithm for saddle point problems*, SIAM Journal on Numerical Analysis, 34 (1997), pp. 1072–1092.

[27] S. C. Brenner and L. R. Scott, *The Mathematical Theory of Finite Element Methods*, Springer Verlag, second ed., 2002.

[28] F. Brezzi, *On the existence, uniqueness and approximation of saddle-point problems arising from Lagrangian multipliers*, ESAIM: Mathematical Modelling and Numerical Analysis – Modélisation Mathématique et Analyse Numérique, 8 (1974), pp. 129–151.

[29] F. Brezzi, J. J. Douglas, and L. D. Marini, *Two families of mixed finite elements for second order elliptic problems*, Numerische Mathematik, 47 (1985), pp. 217–235.

[30] F. Brezzi and M. Fortin, *Mixed and Hybrid Finite Element Methods*, Computational Mathematics, Vol. 15, Springer–Verlag, Berlin, 1991.

[31] W. L. Briggs, V. E. Henson, and S. McCormick, *A Multigrid Tutorial*, SIAM, 2nd ed., 2000.

[32] R. H. Brooks and A. T. Corey, *Properties of porous media affecting fluid flow*, Journal of the Irrigation and Drainage Division, 92 (1966), pp. 61–90.

[33] P. N. Brown, *A local convergence theory for combined inexact-Newton/finite-difference projection methods*, SIAM Journal on Numerical Analysis, 24 (1987), pp. 407–434.

[34] P. N. Brown and Y. Saad, *Hybrid Krylov methods for nonlinear systems of equations*, SIAM Journal of Scientific and Statistical Computing, 11 (1990), pp. 450–481.

[35] C. Burstedde, *p4est: Parallel AMR on forests of octrees*, 2010. http://www.p4est.org/ last accessed March 20th, 2017.

[36] C. Burstedde, J. A. Fonseca, and S. Kollet, *The simulation platform Parflow*, in JUQUEEN Extreme Scaling Workshop 2017, D. Brömmel, W. Frings, and B. J. N. Wylie, eds., no. FZJ-JSC-IB-2017-01 in JSC Internal Report, Jülich Supercomputing Centre, 2017, pp. 37–42.

[37] C. Burstedde, J. A. Fonseca, and S. Kollet, *Enhancing speed and scalability of the ParFlow simulation code*, Computational Geosciences, 22 (2018), pp. 347–361.

[38] C. Burstedde, J. A. Fonseca, and B. Metsch, *A block-AMG saddle point preconditioner with application to mixed Poisson problems on adaptive quad/cube meshes*, 2018. Unpublished.

[39] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. C. Wilcox, *Extreme-scale AMR*, in SC10: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM/IEEE, 2010.

[40] C. Burstedde and J. Holke, *p4est: Scalable algorithms for parallel adaptive mesh refinement*, in JUQUEEN Extreme Scaling Workshop 2016, D. Brömmel, W. Frings, and B. J. N. Wylie, eds., no. FZJ-JSC-IB-2016-01 in JSC Internal Report, Jülich Supercomputing Centre, 2016, pp. 49–54.

[41] C. Burstedde, J. Holke, and T. Isaac, *On the number of face-connected components of Morton-type space-filling curves*, Foundations of Computational Mathematics, (2018), pp. 1–26.

[42] C. Burstedde, G. Stadler, L. Alisic, L. C. Wilcox, E. Tan, M. Gurnis, and O. Ghattas, *Large-scale adaptive mantle convection simulation*, Geophysical Journal International, 192 (2013), pp. 889–906.

[43] C. Burstedde, L. C. Wilcox, and O. Ghattas, *p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees*, SIAM Journal on Scientific Computing, 33 (2011), pp. 1103–1133.

[44] R. Calkin, R. Hempel, H.-C. Hoppe, and P. Wypior, *Portable programming with the PARMACS message-passing library*, Parallel Computing, 20 (1994), pp. 615–632.

[45] C. Carstensen, *Some remarks on the history and future of averaging techniques in a posteriori finite element error analysis*, ZAMM. Zeitschrift für Angewandte Mathematik und Mechanik. Journal of Applied Mathematics and Mechanics, 84 (2004), pp. 3–21.

[46] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, *Hypergraph-based dynamic load balancing for adaptive scientific computations*, in Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07), IEEE, 2007.

[47] C. Chevalier and F. Pellegrini, *PT-Scotch: A tool for efficient parallel graph ordering*, Parallel Computing, 34 (2008), pp. 318–331.

[48] P. Colella, D. T. Graves, N. Keen, T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. Van Straalen, *Chombo Software Package for AMR Applications. Design Document.*, Applied Numerical Algoirthms Group, NERSC Division, Lawrence Berkeley National Laboratory, Berkeley, CA, May 2007.

[49] R. Courant, K. Friedrichs, and H. Lewy, *Über die partiellen Differenzengleichungen der mathematischen Physik*, Mathematische Annalen, 100 (1928), pp. 32–74.

[50] H. Darcy, *Les fontaines publiques de la ville de Dijon*, Victor Dalmont, 1856.

[51] F. Darema, *The SPMD model: Past, present and future*, in Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, London, UK, UK, 2001, Springer-Verlag, pp. 1–.

[52] L. Demkowicz, *Computing with hp-Adaptive Finite Elements I. One- and Two-Dimensional Elliptic and Maxwell Problems*, Chapman & Hall CRC, 2003.

[53] P. Deuflhard and G. Heindl, *Affine invariant convergence theorems for Newton's method and extensions to related methods*, SIAM Journal on Numerical Analysis, 16 (1979), pp. 1–10.

[54] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, *Zoltan data management services for parallel dynamic applications*, Computing in Science and Engineering, 4 (2002), pp. 90–97.

[55] W. Dörfler, *A convergent adaptive algorithm for Poisson's equation*, SIAM Journal on Numerical Analysis, 33 (1996), pp. 1106–1124.

[56] T. L. A. Driessen, R. T. W. L. Hurkmans, W. Terink, P. Hazenberg, P. J. J. F. Torfs, and R. Uijlenhoet, *The hydrological response of the Ourthe catchment to climate change as modelled by the HBV model*, Hydrology and Earth System Sciences, 14 (2010), pp. 651–665.

[57] S. C. Eisenstat and H. F. Walker, *Globally convergent inexact Newton methods*, SIAM Journal on Optimization, 4 (1994), pp. 393–422.

[58] S. C. Eisenstat and H. F. Walker, *Choosing the forcing terms in an inexact Newton method*, SIAM Journal on Scientific Computing, 17 (1996), pp. 16–32.

[59] H. Elman, D. Silvester, and A. Wathen, *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*, Oxford University Press, Oxford, 2014.

[60] H. C. Elman, *Preconditioning for the steady-state Navier–Stokes equations with low viscosity*, SIAM Journal on Scientific Computing, 20 (1999), pp. 1299–1316.

[61] H. C. Elman and D. J. Silvester, *Fast nonsymmetric iterations and preconditioning for Navier-Stokes equations*, SIAM Journal on Scientific Computing, 17 (1996), pp. 33–46.

[62] A. Ern and J.-L. Guermond, *Theory and Practice of Finite Elements*, vol. 159 of Applied Mathematical Sciences, Spinger-Verlag, 2004.

[63] L. C. Evans, *Partial Differential Equations*, Graduate studies in mathematics, American Mathematical Society, second ed., 2010.

[64] R. Ewing, R. Lazarov, T. Russell, and P. Vassilevski, *Local refinement via domain decomposition techniques for mixed finite element methods with rectangular Raviart-Thomas elements*, Domain Decomposition Methods for PDE's, TF Chan, R. Glowinski, J. Periaux, and O.B. Widlund, eds., SIAM, Philadelphia, (1990), pp. 98–114.

[65] R. Ewing and J. Wang, *Analysis of mixed finite element methods on locally refined grids*, Numerische Mathematik, 63 (1992), pp. 183–194.

[66] R. Eymard, T. Gallouët, and R. Herbin, *Finite volume methods*, Handbook of numerical analysis, 7 (2000), pp. 713–1018.

[67] M. Fiedler, *Special matrices and their applications in numerical mathematics*, Dover publications, Mineola, N.Y, 2008.

[68] B. Fischer, A. Ramage, D. J. Silvester, and A. J. Wathen, *Minimum residual methods for augmented systems*, BIT Numerical Mathematics, 38 (1998), pp. 527–543.

[69] J. Flower and A. Kolawa, *The Express programming environment*, tech. rep., Parasoft Corporation Report, 1990.

[70] M. J. Flynn, *Some Computer Organizations and Their Effectiveness*, IEEE Transactions on Computers, C-21 (1972), pp. 948–960.

[71] Free Software Foundation, *Mallinfo(3) Linux Programmer's Manual*. http://man7.org/linux/man-pages/man3/mallinfo.3.html (last accessed: November 2018).

[72] F. Gasper, K. Goergen, P. Shrestha, M. Sulis, J. Rihani, M. Geimer, and S. Kollet, *Implementation and scaling of the fully coupled terrestrial systems modeling platform (TerrSysMP v1.0) in a massively parallel supercomputing environment – a case study on JUQUEEN (IBM Blue Gene/Q)*, Geoscientific Model Development, 7 (2014), pp. 2531–2543.

[73] G. N. Gatica, *A Simple Introduction to the Mixed Finite Element Method: Theory and Applications*, Springer Science & Business Media, 2014.

[74] M. Geimer, F. Wolf, B. Wylie, E. Ábrahám, D. Becker, and B. Mohr, *The Scalasca performance toolset architecture*, Concurrency and Computation: Practice and Experience, 22 (2010), pp. 702–719.

[75] D. Gilbarg and N. Trudinger, *Elliptic Partial Differential Equations of Second Order*, Springer-Verlag, Berlin, second ed., 2001.

[76] J. Gilbert, G. Miller, and S. Teng, *Geometric Mesh Partitioning: Implementation and Experiments*, SIAM Journal on Scientific Computing, 19 (1998), pp. 2091–2110.

[77] M. Griebel, T. Dornseifer, and T. Neunhoeffer, *Numerical simulation in fluid dynamics: a practical introduction*, vol. 3, Siam, 1997.

[78] J. L. Guermond, P. Minev, and J. Shen, *An overview of projection methods for incompressible flows*, Computer Methods in Applied Mechanics and Engineering, 195 (2006), pp. 6011–6045.

[79] M. E. Gurtin, *An introduction to continuum mechanics*, vol. 158, Academic press, 1982.

[80] J. L. Gustafson, *Reevaluating Amdahl's law*, Commun. ACM, 31 (1988), pp. 532–533.

[81] G. E. Hammond, P. C. Lichtner, and R. T. Mills, *Evaluating the performance of parallel subsurface simulators: An illustrative example with PFLOTRAN*, Water Resources Research, 50 (2014), pp. 208–228.

[82] H. Hardelauf, M. Javaux, M. Herbst, S. Gottschalk, R. Kasteel, J. Vanderborght, and H. Vereecken, *PARSWMS: A parallelized model for simulating three-dimensional water flow and solute transport in variably saturated soils*, Vadose Zone Journal, 6 (2007), pp. 255–259.

[83] R. Haverkamp and M. Vauclin, *A Comparative Study of Three Forms of the Richard Equation used for Predicting One-Dimensional Infiltration in Unsaturated Soil*, Soil Science Society of America Journal, 45 (1981), pp. 13–20.

[84] V. E. Henson and U. M. Yang, *BoomerAMG: a Parallel Algebraic Multigrid Solver and Preconditioner*, Applied Numerical Mathematics, 41 (2002), pp. 155–177. Also available as technical report UCRL-JC-141495, Lawrence Livermore National Laboratory, March 2001.

[85] J. S. Hesthaven and T. Warburton, *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*, Springer Science & Business Media, 2007.

[86] M. D. Hill and M. R. Marty, *Amdahl's law in the multicore era*, Computer, 41 (2008).

[87] U. Hornung, *Homogenization and porous media*, vol. 6 of Interdisciplinary Applied Mathematics, Springer Science & Business Media, 1997.

[88] T. J. R. Hughes, *The Finite Element Method*, Dover, New York, 2000.

[89] P. S. Huyakorn and G. F. Pinder, *Computational methods in subsurface flow*, Academic Press, 2012.

[90] H.-T. Hwang, Y.-J. Park, E. Sudicky, and P. Forsyth, *A parallel computational framework to solve flow and transport in integrated surface-subsurface hydrologic systems*, Environmental Modelling & Software, 61 (2014), pp. 39–58.

[91] W. Immerzeel, P. Kraaijenbrink, J. Shea, A. Shrestha, F. Pellicciotti, M. Bierkens, and S. de Jong, *High-resolution monitoring of Himalayan glacier dynamics using unmanned aerial vehicles*, Remote Sensing of Environment, 150 (2014), pp. 93 – 103.

[92] R. Ingram, M. F. Wheeler, and I. Yotov, *A multipoint flux mixed finite element method on hexahedra*, SIAM Journal on Numerical Analysis, 48 (2010), pp. 1281–1312.

[93] T. Isaac, C. Burstedde, L. C. Wilcox, and O. Ghattas, *Recursive algorithms for distributed forests of octrees*, SIAM Journal on Scientific Computing, 37 (2015), pp. C497–C531.

[94] F. John, *Partial Differential Equations*, vol. 1 of Applied Mathematical Sciences, Springer-Verlag New York, 1982.

[95] J. E. Jones and C. S. Woodward, *Newton-Krylov-multigrid solvers for large-scale, highly heterogeneous, variably saturated flow problems*, Advances in Water Resources, 24 (2001), pp. 763–774.

[96] Jülich Supercomputer Centre, *High-Q Club*. https://www.fz-juelich.de/ias/jsc/EN/Expertise/High-Q-Club/_node.html (Last accessed: December 2018).

[97] Jülich Supercomputing Centre, *JUQUEEN: IBM Blue Gene/Q supercomputer system at the Jülich Supercomputing Centre*, Journal of large-scale research facilities, A1 (2015).

[98] B. H. Juurlink and C. Meenderinck, *Amdahl's law for predicting the future of multicores considered harmful*, ACM SIGARCH Computer Architecture News, 40 (2012), pp. 1–9.

[99] G. Karypis and V. Kumar, *A parallel algorithm for multilevel graph partitioning and sparse matrix ordering*, Journal of Parallel and Distributed Computing, 48 (1998), pp. 71–95.

[100] G. Karypis, K. Schoegel, and V. Kumar, *ParMETIS – Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 3.1*, 2013.

[101] C. T. Kelley, *Solving nonlinear equations with Newton's method*, vol. 1, Siam, 2003.

[102] S. J. Kollet and R. M. Maxwell, *Integrated surface-groundwater flow modeling: A free-surface overland flow boundary condition in a parallel groundwater flow model*, Advances in Water Resources, 29 (2006), pp. 945–958.

[103] S. J. Kollet, R. M. Maxwell, C. S. Woodward, S. Smith, J. Vanderborght, H. Vereecken, and C. Simmer, *Proof of concept of regional scale hydrologic simulations at hydrologic resolution utilizing massively parallel computer resources*, Water Resources Research, 46 (2010), p. W04201.

[104] J. Kraus, R. Lazarov, M. Lymbery, S. Margenov, and L. Zikatanov, *Preconditioning heterogeneous $H(\mathrm{div})$ problems by additive Schur complement approximation and applications*, SIAM Journal on Scientific Computing, 38 (2016), pp. A875–A898.

[105] J. Kraus, M. Lymbery, and S. Margenov, *Auxiliary space multigrid method based on additive Schur complement approximation*, Numerical Linear Algebra with Applications, 22 (2015), pp. 965–986.

[106] M. Kronbichler, T. Heister, and W. Bangerth, *High accuracy mantle convection simulation through modern numerical methods*, Geophysical Journal International, 191 (2012), pp. 12–29.

[107] M. Kuznetsov, A. Yakirevich, Y. Pachepsky, S. Sorek, and N. Weisbrod, *Quasi 3d modeling of water flow in vadose zone and groundwater*, Journal of Hydrology, 450451 (2012), pp. 140–149.

[108] L. D. Landau and E. M. Lifshitz, *Fluid Mechanics*, vol. 6 of Landau and Lifshitz Course of Theoretical Physics, Butterworth Heinemann, Oxford, second ed., 1987.

[109] R. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations*, Society for Industrial and Applied Mathematics, 2007.

[110] R. J. LeVeque, *Numerical Methods for Conservation Laws*, Birkhäuser Verlag, Basel, Boston, Berlin, second ed., 1992.

[111] R. J. LeVeque, *Finite volume methods for hyperbolic problems*, Cambridge University Press, 2002.

[112] X. Liu, *Parallel modeling of three-dimensional variably saturated ground water flows with unstructured mesh using open source finite volume platform OpenFoam*, Engineering Applications of Computational Fluid Mechanics, 7 (2013), pp. 223–238.

[113] K.-A. Mardal, J. Sundnes, H. P. Langtangen, and A. Tveito, *Systems of PDEs and block preconditioning*, in Advanced Topics in Computational Partial Differential Equations, H. Langtangen and A. Tveito, eds., vol. 33 of Lecture Notes in Computational Science and Engineering, Springer, 2003, pp. 199–236.

[114] J. E. Marsden and A. J. Tromba, *Vector Calculus*, W. H. Freemann and Company, New-York, second ed., 1981.

[115] R. M. Maxwell, *A terrain-following grid transform and preconditioner for parallel, large-scale, integrated hydrologic modeling*, Advances in Water Resources, 53 (2013), pp. 109–117.

[116] O. Meister, K. Rahnema, and M. Bader, *Parallel memory-efficient adaptive mesh refinement on structured triangular meshes with billions of grid cells*, ACM Trans. Math. Softw., 43 (2016), pp. 19:1–19:27.

[117] Message Passing Interface Forum, *MPI: A message-passing interface standard*, tech. rep., University of Tennessee, Knoxville, TN, USA, May 1994.

[118] Message Passing Interface Forum, *MPI: A message-passing interface standard, version 3.1*, 2015.

[119] B. Metsch, *Algebraic Multigrid (AMG) for Saddle Point Systems*, PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2013.

[120] H. Middelkoop, K. Daamen, D. Gellens, W. Grabs, J. C. J. Kwadijk, H. Lang, B. W. A. H. Parmet, B. Schädler, J. Schulla, and K. Wilke, *Impact of climate change on hydrological regimes and water resources management in the Rhine basin*, Climatic Change, 49 (2001), pp. 105–128.

[121] C. T. Miller, C. N. Dawson, M. W. Farthing, T. Y. Hou, J. Huang, C. E. Kees, C. Kelley, and H. P. Langtangen, *Numerical simulation of water resources problems: Models, methods, and trends*, Advances in Water Resources, 51 (2013), pp. 405–437. 35th Year Anniversary Issue.

[122] G. M. Morton, *A computer oriented geodetic data base; and a new technique in file sequencing*, tech. rep., IBM Ltd., 1966.

[123] A. Müller, J. Behrens, F. X. Giraldo, and V. Wirth, *Comparison between adaptive and uniform discontinuous Galerkin simulations in dry 2d bubble experiments*, Journal of Computational Physics, 235 (2013), pp. 371 – 393.

[124] M. Muskat, *Physical principles of oil production*, IHRDC, Boston, MA, Jan 1981.

[125] R. Niekamp and E. Stein, *An object-oriented approach for parallel two- and three-dimensional adaptive finite element computations*, Computers & Structures, 80 (2002), pp. 317–328.

[126] L. Orgogozo, N. Renon, C. Soulaine, F. Hnon, S. Tomer, D. Labat, O. Pokrovsky, M. Sekhar, R. Ababou, and M. Quintard, *An open source massively parallel solver for Richards equation: Mechanistic modelling of water fluxes at the watershed scale*, Computer Physics Communications, 185 (2014), pp. 3358–3371.

[127] D. Osei-Kuffuor, R. Maxwell, and C. Woodward, *Improved numerical solvers for implicit coupling of subsurface and overland flow*, Advances in Water Resources, 74 (2014), pp. 185–195.

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

[129] *Performance applications programming interface (PAPI)*. http://icl.cs.utk.edu/papi/(Last accessed September 7, 2017).

[130] C. E. Powell, *Parameter-free $H(\mathrm{div})$ preconditioning for a mixed finite element formulation of diffusion problems*, IMA Journal of Numerical Analysis, 25 (2005), pp. 783–796.

[131] C. E. Powell and D. Silvester, *Optimal preconditioning for Raviart-Thomas mixed formulation of second-order elliptic problems*, SIAM Journal on Matrix Analysis and Applications, 25 (2003), pp. 718–738.

[132] Prometeus GmbH, *Top500 the list.* https://www.top500.org/lists/ (last accessed: December 2018).

[133] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, et al., *Petascale direct numerical simulation of blood flow on 200K cores and heterogeneous architectures*, in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society, 2010, pp. 1–11.

[134] R. A. Raviart and J. M. Thomas, *A mixed finite element method for 2nd order elliptic problems*, in Mathematical Aspects of the Finite Element Method, vol. 606, Springer, 1977, pp. 292–315.

[135] L. A. Richards, *Capillary conduction of liquids through porous media*, Physics, 1 (1931), pp. 318–33.

[136] J. Rudi, A. C. I. Malossi, T. Isaac, G. Stadler, M. Gurnis, P. W. J. Staar, Y. Ineichen, C. Bekas, A. Curioni, and O. Ghattas, *An extreme-scale implicit solver for complex PDEs: highly heterogeneous flow in earth's mantle*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2015, p. 5.

[137] J. Rudi, G. Stadler, and O. Ghattas, *Weighted BFBT preconditioner for Stokes flow problems with highly heterogeneous viscosity*, SIAM Journal on Scientific Computing, 39 (2017), pp. S272–S297.

[138] J. Ruge and K. Stüben, *Algebraic Multigrid*, in Multigrid Methods, S. F. McCormick, ed., Frontiers in Applied Mathematics, SIAM, Philadelphia, 1987, ch. 4, pp. 73–130.

[139] Y. Saad and M. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, 7 (1986), pp. 856–869.

[140] H. Sagan, *Space-Filling Curves*, Springer, 1994.

[141] R. S. Sampath, S. S. Adavani, H. Sundar, I. Lashuk, and G. Biros, *Dendro: Parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees*, in SC'08: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM/IEEE, 2008.

[142] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr, *Cube v4: From performance report explorer to performance analysis tool*, Procedia Computer Science, 51 (2015), pp. 1343–1352.

[143] S. Schaffer, *A Semicoarsening Multigrid Method for Elliptic Partial Differential Equations with Highly Discontinuous and Anisotropic Coefficients*, SIAM Journal on Scientific Computing, 20 (1998), pp. 228–242.

[144] J. Schöberl and W. Zulehner, *On Schwarz-type Smoothers for Saddle Point Problems*, Numerische Mathematik, 95 (2003), pp. 377–399.

[145] H. Simon, *Partitioning of unstructured problems for parallel processing*, Computing Systems in Engineering, 2 (1995), p. 135148.

[146] A. Skjellum and A. Leung, *A portable multicomputer communication library atop the reactive kernel*, in Distributed Memory Computing Conference, 1990., Proceedings of the Fifth, vol. 2, IEEE, 1990, pp. 767–776.

[147] H. D. Sterck, R. D. Falgout, J. W. Nolting, and U. M. Yang, *Distance-two interpolation for parallel algebraic multigrid*, Numerical Linear Algebra With Applications, 15 (2008), pp. 115–139.

[148] H. D. Sterck, U. M. Yang, and J. J. Heys, *Reducing complexity in parallel algebraic multigrid preconditioners*, SIAM Journal on Matrix Analysis and Applications, 27 (2006), pp. 1019–1039.

[149] G. Strang and G. J. Fix, *An Analysis of the Finite Element Method*, Wellesley-Cambridge Press, 1988.

[150] W. A. Strauss, *Partial differential equations: an introduction*, John Wiley and Sons, 2008.

[151] J. Strikwerda, *Finite Difference Schemes and Partial Differential Equations, Second Edition*, Society for Industrial and Applied Mathematics, 2004.

[152] K. Stüben, *Algebraic Multigrid (AMG): An Introduction with Applications*, in Multigrid, U. Trottenberg, C. W. Oosterlee, and A. Schüller, eds., Academic Press, London, 2001.

[153] E. Süli and M. David, *An introduction to numerical analysis*, Cambridge University Press, Cambridge New York, 2003.

[154] The Hypre Team, ***hypre*** *– High Performance Preconditioners Users Manual*, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2012. Software version 2.0.9b.

[155] J. W. Thomas, *Numerical partial differential equations: finite difference methods*, vol. 22, Springer Science & Business Media, 2013.

[156] A. F. B. Tompson, R. Ababou, and L. W. Gelhar, *Implementation of the three-dimensional turning bands random field generator*, Water Resources Research, 25 (1989), pp. 2227–2243.

[157] E. F. Toro, *Riemann solvers and numerical methods for fluid dynamics: a practical introduction.*, Springer Verlag, Heidelberg, New-York, 1997.

[158] S. Turek, *Efficient Solvers for Incompressible Flow Problems: An Algorithmic and Computational Approache*, vol. 6, Springer, 1999.

[159] M. Van Genuchten, *A Closed-form Equation for Predicting the Hydraulic Conductivity of Unsaturated Soils*, Soil Science Society of America Journal, 44 (1980), pp. 892–898.

[160] S. P. Vanka, *Block-Implicit Multigrid Solution of Navier-Stokes Equations in Primitive Variables*, Journal of Computational Physics, 65 (1986), pp. 138–158.

[161] D. A. Venditti and D. L. Darmofal, *Adjoint error estimation and grid adaptation for functional outputs: Application to quasi-one-dimensional flow*, Journal of Computational Physics, 164 (2000), pp. 204–227.

[162] M. Wabro, *Algebraic Multigrid Methods for the Numerical Simulation of the Incompressible Navier-Stokes Equations*, dissertation, Institut für Numerische Mathematik, Johannes Kepler Universität, Linz, 2003.

[163] M. Wabro, *Coupled algebraic multigrid methods for the Oseen problem*, Computing and Visualization in Science, 7 (2004), pp. 141–151.

[164] M. Wabro, *AMGe-Coarsening Strategies and Application to the Oseen Equations*, SIAM Journal on Scientific Computing, 27 (2006), pp. 2077–2097.

[165] T. Weinzierl and M. Mehl, *Peano—a traversal and storage scheme for octree-like adaptive Cartesian multiscale grids*, SIAM Journal on Scientific Computing, 33 (2011), pp. 2732–2760.

[166] A. M. Wissink, R. D. Hornung, S. R. Kohn, N. Elliott, and S. S. Smith, *Large Scale Parallel Structured AMR Calculations Using the SAMRAI Framework*, in SC Conference(SC), vol. 00, 11 2001, p. 22.

[167] W. K. Wong, S. Beldring, T. Engen-Skaugen, I. Haddeland, and H. Hisdal, *Climate change effects on spatiotemporal patterns of hydroclimatological summer droughts in norway*, Journal of Hydrometeorology, 12 (2011), pp. 1205–1220.

[168] E. F. Wood, J. K. Roundy, T. J. Troy, L. Van Beek, M. F. Bierkens, E. Blyth, A. de Roo, P. Döll, M. Ek, J. Famiglietti, et al., *Hyperresolution global land surface modeling: Meeting a grand challenge for monitoring earth's terrestrial water*, Water Resources Research, 47 (2011).

[169] J. Xu, *Two-grid discretization techniques for linear and nonlinear PDEs*, SIAM Journal on Numerical Analysis, 33 (1996), pp. 1759–1777.

[170] H. Yamamoto, K. Zhang, K. Karasaki, A. Marui, H. Uehara, and N. Nishikawa, *Numerical investigation concerning the impact of CO_2 geologic storage on regional groundwater flow*, International Journal of Greenhouse Gas Control, 3 (2009), pp. 586–599.

[171] K. Zhang, Y.-S. Wu, and K. Pruess, *Users guide for TOUGH2-MP a massively parallel version of the TOUGH2 code*, Lawrence Berkeley National Laboratory, 2008. Report LBNL-315E.

[172] J. Zhu, *Equivalent parallel and perpendicular unsaturated hydraulic conductivities: Arithmetic mean or harmonic mean?*, Soil Science Society of America Journal, 72 (2008), pp. 1226–1233.