Inaugural-Dissertation

zur Erlangung der Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Ruprecht-Karls-Universität
Heidelberg

vorgelegt von
Diplom-Mathematiker Markus Blatt
aus Erlangen

Tag der mündlichen Prüfung: 2. Juli 2010

A Parallel Algebraic Multigrid Method for Elliptic Problems with Highly Discontinuous Coefficients

February 4, 2010

Gutachter: Prof. Dr. Peter Bastian
Dr. Robert Scheichl

Abstract

The aim of this thesis is the development of a parallel algebraic multigrid method suitable for solving linear systems arising from the discretization of scalar and systems of partial differential equations. Among others it is suitable from conforming finite element methods, finite volume methods, and discontinuous Galerkin methods. The method is especially tailored for the solution of diffusion problems with highly oscillating and discontinuous diffusion coefficients.

The presented approach uses a new strength of connection measure for guiding the construction of the coarse level matrices. It uses a heuristic greedy aggregation algorithm that allows for aggressive coarsening. It is able to detect weak connections in the matrix graph even for anisotropic diffusion with bi- and trilinear finite elements and thus leads to semi-coarsening even for these cases. At the same time it keeps the stencil size from the finer levels and thus the total operator complexity low even for three dimensional problems. This leads to a very low memory consumption of our solver compared with other methods.

We develop extensions of the solver to systems of partial differential equation by using special blocking approaches of the unknowns. These blockings are emulated by the underlying matrix and vector data structures. As the blocking is already available to the compiler, it can be exploited to produce automatically more efficient code.

For the solution of the linear systems stemming from Discontinuous Galerkin discretizations, we employ the subspace of continuous linear basis function as the space associated with the first coarse level. The further coarsening is done by using the above algorithm. For the method of Baumann and Oden we need to use overlapping Schwarz methods as smoothers to get a convergent method. Their local subspaces are constructed using our aggregation algorithm on the blocks consisting of all unknowns associated with each element.

Finally we present a parallelisation approach for iterative solvers and use it to parallelise our algebraic multigrid method. In our approach the information about the data decomposition is kept apart from the linear algebra solvers and data structures. It is used to keep the data stored in the local memory of the process consistent. Using our proposed consistency model, the efficient sequential linear algebra solvers and data structures can be reused without the need to rewrite the actual solver algorithms.

Zusammenfassung

Gegenstand dieser Arbeit ist die Entwicklung eines parallelen algebraischen Mehrgitterverfahrens zur Lösung linearer Systeme, die durch die Diskretisierung von skalaren und Systemen von partiellen Differentialgleichungen entstehen. Unter anderem ist es geeignet für lineare System aus der Diskretisierung mit konformen finite Elemente Verfahren, finiten Volumen Verfahren und unstetigen Galerkin Verfahren. Die Methode ist besonders geeignet zur Lösung der Diffusionsgleichung mit stark oszillierenden und unstetigen Diffusionskoeffizienten.

Das entwickelte Verfahren benützt ein neues Maß zur Erkennung von „starken“ Verbindungen im Matrixgraphen. Dieses leitet die Aggregation der Unbekannten der Matrix und damit die Konstruktion gröberer Matrizen. Die Vergröberung basiert auf einem neu entwickelten heuristischen Algorithmus, der auch aggressives Vergröbern erlaubt. Dieser kann schwache Verbindungen auch für anisotrope Diffusion mit bi- und trilinearen finiten Elementen erkennen und vergröbert nur in Richtung starker Verbindungen. Zur gleichen Zeit lässt der Algorithmus die Größe des Besetzheitssterns der Matrizen auf den groben Ebenen vergleichbar klein wie bei der Matrix auf der feinsten Ebene. Somit ist auch die Komplexität des Operators und damit der Speicherverbrauch des Lösers im Vergleich mit anderen algebraischen Mehrgitterlösern gering.

Wir entwickeln Erweiterungen des Lösers für Systeme partieller Differentialgleichungen, indem wir die Unbekannten auf eine spezielle Weise blocken. Diese Blöcke werden in den benutzten Matrix- und Vektordatenstrukturen nachgebildet. Die Struktur dieser Blöcke ist dem Compiler bereits bekannt. Deswegen kann er sie ausnützen, um besonders effizienten Code zu generieren.

Für die Lösung linearer Systeme, die aus unstetigen Galerkin Diskretisierungen entstanden sind, benutzen wir den Unterraum der stetigen linearen Ansatzfunktionen als Raum der ersten gröberen Ebene. Ab hier benutzen wir zur weiteren Vergröberung den obigen Algorithmus. Für die Methode von Baumann und Oden benötigen wir überlappende Schwarz Verfahren als Glätter, um Konvergenz zu erreichen. Die lokalen Probleme dieser Glätter konstruieren wir mit Hilfe unseres Aggregationsalgorithmus. Wir benützen die geblockte Variante ähnlich wie für Systeme. Alle mit den Basisfunktionen eines Elements assoziierten Unbekannten bilden zusammen einen Block.

Schließlich präsentieren wir unseren Parallelisierungansatz für iterative Löser und benutzen ihn, um unser algebraisches Mehrgitterverfahren

zu parallelisieren. Bei unserem Ansatz wird die Information über die Datenverteilung und Kommunikationsmuster nicht in die Löser und Datenstrukturen integriert. Basierend auf diesen extern gespeicherten Informationen wird dafür gesorgt, dass die Daten vorgegebene Konsistenzmodelle erfüllen. So können wir die sequentiellen Löseralgorithmen und die Datenstrukturen der linearen Algebra wiederverwenden, ohne die Löser neu schreiben zu müssen. Gleichzeitig wird die Kommunikation der Daten auf ein Minimum beschränkt und die Effizienz der sequentiellen linearen Algebra kann auch im parallelen Fall genutzt werden.

Acknowledgement

First of all, I would like to express my deep gratitude to the adviser of this thesis, Prof. Dr. Peter Bastian. When I told him that I would like to write a dissertation, he, immediately, provided me with a post. He always trusted in me and provided me with great freedom of research. The support that I experienced during the preparation of this thesis in his group was outstanding.

The diversity of his group in terms of my colleagues' scientific background was very fruitful in forming me as a scientist. This group is truly interdisciplinary in itself and lets one get insights into many different fields and applications. Explaining current problems and insights to people with a different background turned out to be very useful. Sometimes rephrasing problems in another scientific language made contradictions and errors apparent, even before anybody commented on the problem. While being grateful to all members of the group, my special thanks go to my dear colleagues Christian Engwer and Olaf Ippisch, who constantly discussed scientific issues and real world stuff with me from the start of this thesis.

I would like to thank Dr. Rob Scheichl for many discussions about applying algebraic multigrid methods to problems with jumping coefficients as well as to DG discretizations, and Dr. Klaus Johannsen for sharing his experiences with applying multigrid to NIPG discretizations.

I would like to thank my fellow DUNE developers for putting so much effort into the DUNE library. Without them it would not have been possible to test my solver with such challenging problems. Providing my solvers with the library resulted into many comments from them. This made some flaws apparent that I would not have found on my own.

I would like to thank the users of DUNE and ISTL for trying it out and using it in productive code. While this poses a real challenge onto the developers, it forces us to always provide high quality code. Without the users, DUNE would not be as usable and of such high quality.

Especially, I would like to thank the Statoil research centre in Trondheim, Norway, for evaluating my solvers for their oil reservoir simulator and providing me with an upscaling example and code for usage in this thesis.

Furthermore, I would like to thank my father Hans-Peter and my brother Simon for proofreading the final version of my thesis in nearly no time. I am grateful to the rest of family, namely my mother Elvira and my sisters Sarah and Sophie for their implicit support by just being there. Special

thanks go to my cousin Bärbel Gundlach, who was there when I needed someone to take over some of my everyday duties. Her immediate help gave me more time to work on the subject of this thesis.

Last but not least, I would like to thank Michaela for her patience and constant emotional support during the preparation of this thesis. She always believed in my abilities and assured me of them.

Contents

List of Tables

List of Figures

List of Algorithms

1 Introduction

Solving partial differential equations (PDE) is an ubiquitous task in scientific computing. In simulations a large part of the computation time is spent in solving the large sparse linear systems arising from the discretizations of these partial differential equations. With increasing memory and computation power of modern computers higher and higher resolutions of these discretizations are feasible. Therefore, the usage of scalable linear solvers and preconditioners is mandatory today. One of the most efficient ways to achieve scalability is the usage of multi-level methods. Among them are the so-called geometric (GMG) and algebraic multigrid (AMG) methods.

While often being the more efficient method the former has some drawbacks, too. For complicated problems the method has to be adapted to the particular problem solved. Special care has to be taken if the coefficients of the PDE are non-smooth. In this case standard geometric multigrid methods converge very slowly. For anisotropic diffusion problems, where the diffusion in one direction is very low or high compared with other directions, convergence is similarly slow. In both cases good convergence can be recovered when using problem adapted coarsening schemes, smoothers, and grid interpolation operators. This often requires expert knowledge that cannot be expected from users of the method. In addition the method needs an adapted grid hierarchy that often is not available from commercial applications or rather complicated to compute for unstructured grids. Therefore, algebraic multigrid methods are used in these cases more often. They do not need to be adjusted as accurately and normally just need the plain linear system as input.

Today a wide variety of algebraic multigrid methods exist. Those without the need for geometric information can be subdivided into two major classes: the methods based on interpolation and the methods based on aggregation. In interpolation AMG the fine level unknowns are subdivided into unknowns that will also be present on the coarser level and those that will not. Accordingly the interpolation operators between the levels are constructed. The development of these methods started with Ruge and Stüben [1987], Brandt et al. [1984], Brandt [1986]. In the other

class the fine level unknowns are aggregated and each of these aggregates represents one unknown on the coarser level. For the inter-level interpolation they either use piece-wise constant interpolation as in Raw [1985], Braess [1995] or more accurate interpolation that is created by smoothing as in Vaněk [1992], Vaněk et al. [1996a]. The former approach is very memory efficient but the convergence rate is not independent of the problems size. For the latter approach the computation of the hierarchy is more expensive and the matrices on the coarser levels are not as sparse as with piecewise constant interpolation.

While the standard AMG methods mentioned above work very well for a wide variety of problems, their performance on some finite element problems (e.g. thin-body elasticity on unstructured grids) is unsatisfactory. These cases deviate substantially from the M-Matrix case on which the traditional AMG heuristics are based. This lead to the development of algebraic multigrid based on element interpolation (AMGe), see Brezina et al. [2000]. It uses multigrid convergence theory and the local stiffness matrices for the individual finite elements to produce better interpolation parameters than in the classical interpolation AMG. Later the ideas from AMGe were also used to construct coarse grids and finite elements in Jones and Vassilevski [2001]. In addition to the global matrix and the local stiffness matrices topology information of the elements of the finest grid is needed for this solver.

A new challenge is the development of parallel solvers suitable for usage on today's supercomputers. These make thousands of processors available to the simulation software and can achieve more than one petaflops as their peak computation performance. The most common parallel approaches are either domain decomposition methods or multigrid methods. Parallel algebraic multigrid methods based on classical coarsening schemes have the problem that the complexity of the solver increases with the problem size and the number of processors used. That is, the matrices on coarser levels are not as sparse as for the sequential method. This leads to higher memory consumption and execution times. Therefore, this area is still an active topic of research as recent publication like Alber and Olson [2007], Griebel et al. [2008], Sterck et al. [2008] show.

An important application area of algebraic multigrid is the solution of the diffusion equation

$$-\nabla \cdot (\mathsf{K}\nabla u) = f$$

for the case that the diffusion coefficient K (also called permeability) is highly variable and might even have large jumps throughout the domain.

Such equations model for example flow in heterogeneous porous media. For this kind of problem algebraic multigrid methods should be able to detect the jumps and adapt their coarsening scheme such that the solver is still robust. Other iterative methods are more likely to have problems as the condition of the resulting matrix is not only dependent on the resolution of the discretization but also on the ratio of the permeability jumps. A new two-level preconditioner based on domain decomposition was presented and analysed in Scheichl and Vainikko [2007]. The coarse space of this preconditioner was constructed by means of an aggregation method borrowed from algebraic multigrid.

For finite difference and low order continuous finite element discretizations the use of (algebraic) multigrid solvers is well established. In the seventies the development of the discontinuous Galerkin (DG) finite element method for discretizing partial differential equations started. It is based on a totally discontinuous finite element space and has many advantageous properties. Due to the missing continuity constraint of the basis functions the usage with non-conforming unstructured grids is easy and allows for flexible mesh adaptation techniques. The choice of the basis functions used is flexible and allows for variable polynomial order. For the above presented flow problem an important property of the DG discretizations is its element-wise mass conservation. A disadvantage is that the resulting linear systems have a higher number of degrees of freedom when compared with the continuous methods. This leads to even larger and more ill-conditioned systems and makes the use of optimal solvers even more mandatory. The quest for such solvers is still ongoing as recent publication like Gopalakrishnan and Kanschat [2003], Johannsen [2005], Dobrev et al. [2006], Dobrev [2007], Antonietti and Ayuso [2008], Antonietti [2007], Prill et al. [2009], Ayuso and Zikatanov [2009] show. The most promising types of solvers in this area are again multigrid and domain decomposition methods.

1.1 Contributions

For the sequential version the algebraic multigrid method based on aggregation, we invent a new strength of connection measure to guide the coarse level construction. It is symmetric and leads to semi-coarsening for anisotropic diffusion even when using bi- and trilinear finite elements. Using this criterion the aggregation is done with a new heuristic greedy aggregation algorithm. This allows for rather aggressive coarsening while

keeping the stencil size of the coarse level matrices at a minimum. Still the method is suitable as an efficient preconditioner to Krylov methods.

The code of our algebraic multigrid method uses generic block matrices and vectors. The structure and size of their blocks is already known at compile time and can be exploited by the compiler to optimise the code. Due to this implementation the method is used without major modifications for coupled systems of partial differential equations.

We present a generic parallelisation approach of the used linear algebra based on block matrices and block vectors. In contrast to many current approaches the information about the data decomposition and communication pattern is not incorporated into the data structures directly. This information is stored outside of the data structures in so-called parallel index sets. These are used to impose data consistency and at the same time reuse the fast sequential linear algebra components. This leads to the parallel solvers of the "Iterative Solver Template Library" (ISTL), which is available as an open source software.

The algebraic multigrid method is extended to discontinuous Galerkin discretizations. As a first coarse space we use the space of continuous linear basis functions. Especially for higher order trial functions this leads to a tremendous reduction of the number of degrees of freedom used on the finer levels. This approach works for both symmetric and non-symmetric interior penalty discontinuous Galerkin discretization provided that the penalty parameter is chosen to be sufficiently large. For the method of Oden and Baumann, that is lacking the penalty parameter, additional measures have to be taken. We employ overlapping Schwarz methods as smoothers on the fine level. For the construction of the local subdomains of these smoothers we use our greedy aggregation algorithm.

1.2 Outline

In Chapter 2 we introduce our model problem. We describe the different discretization methods used in this thesis, namely finite element methods, finite volume methods and discontinuous Galerkin methods. An important part of the chapter is how the discretizations are mapped to the resulting linear equation systems. We describe blocking strategies of the unknowns for higher order discontinuous Galerkin methods and systems of partial differential equations.

In Chapter 3 we describe the sequential version of our algebraic multigrid method based on aggregation. We use the notation of abstract Schwarz

methods to describe the smoothers of our method. After describing a natural extension to systems of partial differential equations, we conclude the chapter by introducing representatives of the model problem. These models have high contrast jumps in their coefficients. With numerical results we prove the scalability and efficiency of our approach for discretizations with continuous finite element and finite volume methods.

In Chapter 4 we extend our approach to discontinuous Galerkin methods. As the first coarse space we use the space of continuous linear functions. For the method of Baumann and Oden we propose one-level overlapping Schwarz methods as smoothers on the finest level. The local problems of these smoothers will be constructed algebraically. The robustness of the method will be shown on various model problems and for various polynomial orders.

In Chapter 5 we describe the parallelisation of our method. The efficiency of the approach will be demonstrated using the model problems.

In the previous chapters all results were obtained using model problems. This does not do justice to our multigrid solver as it is suitable to be applied to real word examples. Therefore, in Chapter 6 we will show various real world applications we applied our solver to.

To complete the presentation of our method, we dedicate Chapter 7 to implementational details of our method. Although not interesting to non-practitioners, we regard this as an integral part of the thesis. The efficiency of our method is highly related to the implementation; namely, to the way the finite element discretization is mapped to the matrix and vector data structures. In the first part we describe the matrix and vector data structures used together with the preconditioner and solver interface. Then we describe the developed components used to turn the sequential linear algebra into scalable parallel solvers. This approach is the main cause of our good scalability results presented in earlier chapters. Finally we describe the interface and components of our AMG method.

We conclude the thesis in Chapter 8 with a summary of our developments and results.

2 Model Problem and Discretization

In this chapter we will introduce the second order elliptic model problem that we will investigate. We will use this problem to outline finite element (FE), finite volume (FV) and some important discontinuous Galerkin (DG) discretization methods. For a more detailed discussion of finite element approximations we refer to the monographs Braess [1997], Ciarlet [1978]. Detailed description of finite volume methods can be found in Mitchev [1996], Bey [1998]. A very comprehensive unified analysis of discontinuous Galerkin finite element methods is presented in Arnold et al. [2002]. We refer people interested in an introduction to DG to the monograph Riviere [2008].

Let us first define the Sobolev spaces that will be used in this thesis. For a bounded connected open subset $\mathcal{D}$ of $\mathbb{R}^d$, $d = 1, 2, 3$, let $L^2(\mathcal{D})$ be the space of square integrable functions on $\mathcal{D}$, and let $(\cdot,\cdot)_{\mathcal{D}}$ and $\|\cdot\|_{0,\mathcal{D}}$ denote the inner product and norm on $L^2(\mathcal{D})$ (or $(L^2(\mathcal{D}))^d$), respectively. We denote by $H_0^1(\mathcal{D})$ the completion of infinitely differentiable functions with compact support under the norm

$$|u|_{1,\mathcal{D}} = \|\nabla u\|_{0,\mathcal{D}}\,.$$

Let the dual space of $H_0^1(\mathcal{D})$ be denoted by $H^{-1}(\mathcal{D})$. For $0 < s < 1$, let $H^{-s}(\mathcal{D})$ denote the space obtained by real interpolation between $H^{-1}(\mathcal{D})$ and $L^2(\mathcal{D})$. For non-negative integers m, the Sobolev space $H^m(\mathcal{D})$ is the set of functions in $L^2(\mathcal{D})$ with distributional derivatives up to order m also in $L^2(\mathcal{D})$. If s is a positive real number between non-negative integers m and $m+1$, $H^s(\mathcal{D})$ is the space obtained by real interpolation between $H^m(\mathcal{D})$ and $H^{m+1}(\mathcal{D})$. The norm on $H^r(\mathcal{D})$ is denoted by $\|\cdot\|_{r,\mathcal{D}}$.

2.1 Model Problem

Let Ω be a polyhedral domain in $\mathbb{R}^d$, $d = 1, 2, 3$, and let $\mathbf{n}$ denote the outward unit normal to $\partial\Omega$. The boundary is decomposed into two disjoint components Γ_D and Γ_N with $\partial\Omega = \Gamma_D \cup \Gamma_N$, $\Gamma_D \cap \Gamma_N = \emptyset$ and Γ_D has positive measure. Then we consider the following second-order elliptic boundary

value problem:

$$\begin{aligned} -\nabla \cdot (\mathsf{K}\nabla u) &= f \text{ in } \Omega \\ u &= g_D \text{ on } \Gamma_D \\ (K\nabla u) \cdot \mathbf{n} &= g_N \text{ on } \Gamma_N \,. \end{aligned} \tag{2.1}$$

Here u is the unknown function and $f \in L^2(\Omega)$, $g_D \in H^{1/2}(\Gamma_D)$, and $g_N \in H^{-1/2}(\Gamma_N)$ are given functions. Furthermore, we assume that the coefficient matrix $\mathsf{K}(x) \in (L^\infty(\Omega))^{d\times d}$ is symmetric, uniformly bounded, and positive definite, such that

$$c_1|\zeta|^2 \leq K(x)\zeta \cdot \zeta \leq c_2|\zeta|^2 \quad \forall \zeta \in \mathbb{R}^d \,,$$

for given constants $c_1, c_2 \in \mathbb{R}$, $c_1, c_2 > 0$.

Let $\widetilde{g}_D \in H^1(\Omega)$ be an extension of g_D inside the domain, that is $\widetilde{g}_D|_{\Gamma_D} = g_D$, and define the space

$$H^1_0(\Omega;\Gamma_D) = \{v \in H^1(\Omega) : v|_{\Gamma_D} = 0\} \,.$$

Then the corresponding variational formulation of problem (2.1) reads: find $u \in \widetilde{g}_D + H^1_0(\Omega;\Gamma_D)$ such that

$$(K\nabla u, \nabla v)_\Omega = (f, v)_\Omega + (g_N, v)_{\Gamma_N} \,, \quad \forall v \in H^1_0(\Omega, \Gamma_D) \,. \tag{2.2}$$

2.2 Galerkin Approximation

The next step is now to approximate our variational problem in finite dimensional subspaces. Instead of starting from the concrete problem (2.2), we use a more abstract setting that includes our example.

Let V be a Hilbert space, let $\mathcal{A}(\cdot,\cdot) : V\times V \to \mathbb{R}$ be a continuous V elliptic bilinear form, and let $\mathcal{L} : V \to \mathbb{R}$ be a continuous linear form. Then an abstract variational problem is given by:
find $u \in V$ such that

$$\mathcal{A}(u, v) = \mathcal{L}(v) \quad \forall v \in V \,.$$

Our aim is now to approximate the solution using the Galerkin method. Note that the same methodology can be applied for the Petrov-Galerkin approximation. As it is not needed for the discretisations in this thesis we restrict ourselves to the simpler Galerkin projection.

Let $V_h \subset V$ be our trial and test function space of finite dimension n. Then the Galerkin projection leads to the finite dimensional problem:
find $u_h \in V_h$ such that

$$\mathcal{A}_h(u_h, v_h) = \mathcal{L}_h(v_h) \quad \forall v_h \in V_h \,. \tag{2.3}$$

Note that $\mathcal{A}_h(\cdot,\cdot)$ and $\mathcal{L}_h(\cdot)$ might differ from the original bilinear and linear form of the continuous problem.

Fixing an arbitrary basis $\Phi = \{\phi_1, \ldots, \phi_n\}$ of our function space V_h, we are able to transform the finite dimensional variational problem (2.3) into an algebraic equation system. We represent our finite dimensional solution as

$$u_h = \sum_{j=1}^{n} \mathsf{u}_j \phi_j$$

and set this into the variational form (2.3) to get

$$\sum_{j=1}^{n} \mathcal{A}(\phi_j, v)\mathsf{u}_j = \mathcal{L}(v) \quad \forall v \in V_h \,.$$

It suffices to test against all basis functions $\phi_i \in \Phi$. This represents a system of linear equations given by $\mathsf{A}\mathbf{u} = \mathbf{b}$ with

$$\mathsf{A} = \begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \ldots & a_{nn} \end{pmatrix}, \quad a_{ij} = \mathcal{A}(\phi_j, \phi_i)$$

and

$$\mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}, \quad b_i = \mathcal{L}(\phi_i) \,.$$

We will use a special notation for matrices and vectors in this thesis. We denote finite index sets by I with possibly sub- or superscript or both. I does not have to be ordered, necessarily. From our point of view a vector $b \in \mathbb{K}^I$ is a mapping $\mathbf{b} : I \to \mathbb{K}$. A matrix $\mathsf{A} \in \mathbb{K}^{I \times I}$ is a mapping $\mathsf{A} : I \times I \to \mathbb{K}$. This notation will be very useful for describing our algebraic multigrid method and the parallelisation approach.

To construct our finite dimensional function space V_h, we will always use a primal triangulation of our domain Ω. (We will also use the words mesh and grid instead of triangulation.)

Let $\mathcal{T} = \{\tau_1, \dots \tau_M\}$ be a subdivision of our set $\overline{\Omega} = \bigcup_{i=1}^{M} \tau_i$ into a finite number of closed subsets τ_i with non-empty interior and Lipschitz-continuous boundary $\partial\tau_i$. We assume that $\mathcal{T}$ is a regular triangulation. That is, the intersection of two elements is either empty or a common vertex, edge, or face. For any element $\tau \in \mathcal{T}$ we denote its diameter with h_τ and the boundary of τ with $\partial\tau$. Furthermore, let $h := \max_{\tau\in\mathcal{T}} h_\tau$ denote the characteristic mesh size of the whole partition.

2.2.1 Continuous Galerkin Finite Element

The first class of approximations used in this thesis are continuous Galerkin finite element methods. In this class both trial and test function space are the same. As trial and test functions we will use continuous functions that are piecewise polynomial on the elements of our triangulation.

We will use the following multi-index notation. Let $a = (a_1, \dots, a_d)$, $a_i \in \mathbb{N}_0$, be a multi-index for a given dimension d, then we define $|a| = \sum_{i=1}^{d} a_i$, $|a|_\infty = \max_{i=1}^{d} a_i$, and $x^a = \prod_{i=1}^{d} x_i^{a_i}$ for a vector $x \in \mathbb{R}^d$.

Then the space of polynomials on $\mathbb{R}^d$ of maximum total degree k is denoted by

$$P_k = \{u : \mathbb{R}^d \to \mathbb{R} \mid u(x) = \sum_{|a|\le k} c_a x^a,\ c_a \in \mathbb{R}\},$$

and the space of polynomials on $\mathbb{R}^d$ with maximum degree k in each component is denoted by

$$Q_k = \{u : \mathbb{R}^d \to \mathbb{R} \mid u(x) = \sum_{|a|_\infty \le k} c_a x^a,\ c_a \in \mathbb{R}\}.$$

Similar to above, we incorporate the Dirichlet boundary condition into the space $C(\overline{\Omega})$ of continuous functions on $\overline{\Omega}$ by defining

$$C_0(\Omega, \Gamma_D) = \{u \in C(\overline{\Omega}) \mid u|_{\Gamma_D} = 0\}.$$

Then the spaces that are polynomial when restricted to the elements of our grid and respect the Dirichlet boundary condition on Γ_D are

$$P_k(\mathcal{T}) = \{u \in C_0(\Omega, \Gamma_D) \mid u|_\tau \in P_k\ \forall \tau \in \mathcal{T}\}$$

for triangulations consisting only of simplicial elements and

$$Q_k(\mathcal{T}) = \{u \in C_0(\Omega, \Gamma_D) \mid u|_\tau \in Q_k\ \forall \tau \in \mathcal{T}\}$$

for triangulations consisting only of quadrilateral and hexahedral elements in two and three dimensions, respectively.

Using one of these finite dimensional spaces as both trial and test function space for the original bilinear and linear form as defined by the variational formulation (2.2), the resulting linear system follows from the above described Galerkin approximation.

2.2.2 Discontinuous Galerkin Finite Element

Let $\mathcal{T}$ be a triangulation of our domain Ω as defined above. The set of all edges ($d = 2$) or faces ($d = 3$) of the elements $\tau \in \mathcal{T}$ will be denoted by $\mathcal{E}$. We will call its elements edge regardless of the dimension d. Let $\mathcal{E}^I$ and $\mathcal{E}^B$ denote the sets of all interior and all boundary edges, respectively. We will assume that the Dirichlet boundary Γ_D is the union of a non-empty set of boundary edges. It will be denoted by $\mathcal{E}^D$. Consequently, $\mathcal{E}^N := \mathcal{E}^B \setminus \mathcal{E}^D$ will denote the edges where the Neumann boundary condition is imposed. Thus, we have $\mathcal{E} = \mathcal{E}^I \cup \mathcal{E}^D \cup \mathcal{E}^N$.

Over the triangulation $\mathcal{T}$ we define the *broken Sobolev space*:

$$H^s(\mathcal{T}) = \{v \in L^2(\Omega) \mid \forall \tau \in \mathcal{T}, v|_\tau \in H^s(\tau)\}, \quad s \geq 0 .$$

In order to define the discontinuous Galerkin discretization, we need to require that K and g_N are sufficiently smooth. Namely, we assume that $\mathsf{K} \in (H^{1,\infty}(\mathcal{T}))^{d\times d}$ and $g_N \in L^2(\Gamma_N)$.

For our discretization we will use the following space of discontinuous piecewise polynomial functions of total degree $k \geq 0$ as the space of trial and test functions

$$\mathcal{V}_k := \{v \in L^2(\Omega) : v|_\tau \in P_k, \forall \tau \in \mathcal{T}\} .$$

Let $e \in \mathcal{E}^I$ (see Figure 2.1) be an edge shared by two elements τ_1 and τ_2 and let $\mathbf{n}_e$ denote the unit normal vector pointing from τ_1 to τ_2. That is, we fix a direction for each pair of elements sharing an edge. For a boundary edge $e \in \mathcal{E}^B$ we denote with $\mathbf{n}_e$ the unit normal vector pointing outside of Ω. Let ϕ be a function defined on both sides of the edge e as ϕ_1 and ϕ_2 from the sides of the elements τ_1 and τ_2, respectively. For example ϕ could be a trace of a function in $H^s(\mathcal{T})$, $s > 1/2$, or the normal derivative, $\nabla v \cdot \mathbf{n}$ of a function $u \in H^s(\mathcal{T})$, $s > 3/2$. For such ϕ we define the *jump* and *average operators* as :

$$[\![\phi]\!] = \phi_1 - \phi_2 \text{ and } \{\phi\} = \frac{\phi_1 + \phi_2}{2} .$$

Consequently, for a boundary interface $e \in \mathcal{E}^B$ and ϕ only defined on the interior side of it as ϕ_i we set

$$[\![\phi]\!] = \phi_i \text{ and } \{\phi\} = \phi_i$$

Further, we shall need the piecewise constant function

$$h_e = h_e(x) = \begin{cases} |e| & \text{for } x \in e \in \mathcal{E}, d = 2 \\ |e|^{1/2} & \text{for } x \in e \in \mathcal{E}, d = 3 \end{cases} .$$

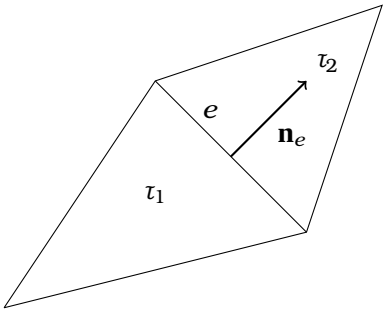


Figure 2.1: Interior edge shared by two elements

We assumed $K \in (H^{1,\infty}(\mathcal{T}))^{d\times d}$, and $g_N \in L^2(\mathcal{T})$. With this assumption we are allowed to define traces of $K\nabla u$ if $u \in H^s(\mathcal{T})$ for some $s > 3/2$. Then we define the DG bilinear form

$$\begin{aligned} \mathcal{A}_{DG}(u,v) = &\sum_{\tau\in\mathcal{T}} (K\nabla u, \nabla v)_\tau - \sum_{e\in\mathcal{E}^I\cup\mathcal{E}^D} (\{K\nabla u\cdot\mathbf{n}_e\}, [\![v]\!])_e \\ &+ \sum_{e\in\mathcal{E}^I\cup\mathcal{E}^D} \sigma\,(\{K\nabla v\cdot\mathbf{n}_e\}, [\![u]\!])_e + \sum_{e\in\mathcal{E}^I\cup\mathcal{E}^D} \frac{\mu}{h_e}\,([\![u]\!], [\![v]\!])_e \end{aligned} \tag{2.4}$$

and the linear form

$$\mathcal{L}_{DG}(v) = \sum_{\tau\in\mathcal{T}} (f,v)_\tau + \sum_{e\in\mathcal{E}^N} (g_N, v)_e + \sum_{e\in\mathcal{E}^D} \sigma(g_D, K\nabla v\cdot\mathbf{n}_e)_e + \sum_{e\in\mathcal{E}^D} \frac{\mu}{h_e}(g_D, v)_e\,.$$

The choices of σ and the penalty parameter μ define the following well known DG methods:

- $\sigma = -1$ and $\mu \geq \mu_0$ sufficiently large define the symmetric interior penalty (IP or SIPG) method, Wheeler [1978], Arnold [1982], Arnold et al. [2002].

- $\sigma = +1$ and $\mu > 0$ define the non-symmetric interior penalty (NIPG) method, Rivière and Wheeler [1999], Arnold et al. [2002].
- $\sigma = 1$ and $\mu = 0$ define the method of Baumann and Oden, Baumann and Oden [1999], Rivière and Wheeler [1999], Oden et al. [1998].

With the above defined bilinear and linear forms the discrete DG problem can be written as:
find $u \in \mathcal{V}_k$ such that

$$\mathcal{A}_{DG}(u, v) = \mathcal{L}_{DG}(v)\,, \quad \forall v \in \mathcal{V}_k\,. \tag{2.5}$$

For the case of high contrasts jumps in K we will use the following modification of the above methods. Assuming that K is constant on each cell of the grid, the value at an inner edge in the average computation is approximated by the harmonic average of K in the neighbouring cells. In this simple case using harmonic averages is the same as the method using weighted averages that has been analysed in Ern et al. [2009]. We will briefly describe the latter approach.

Let K_1 and K_2 be the permeability tensors in the two neighbouring cells τ_1 and τ_2 of the inner edge e. Then we redefine the average of the flux as

$$\{K\nabla v \cdot \mathbf{n}_e\} = \frac{\delta_2 \mathsf{K}_1 \nabla v \cdot \mathbf{n}_e + \delta_1 \mathsf{K}_2 \nabla v \cdot \mathbf{n}_e}{\delta_1 + \delta_2}\,, \quad \text{with } \delta_i = \mathbf{n}_e \cdot \mathsf{K}_i \mathbf{n}_e\,, \quad \text{for } i = 1, 2\,. \tag{2.6}$$

In addition the penalty parameter μ is scaled by the factor

$$\frac{\delta_1^2 \mathsf{K}_2 + \delta_2^2 \mathsf{K}_1}{(\delta_1 + \delta_2)}\,. \tag{2.7}$$

In Ern et al. [2009] the above introduced weighted average were needed for the error analysis of the advection part in the classical advection-diffusion equation. For the diffusive part of the problem they were not needed. Despite having no advection in our problems the approach turned out to lead to better conditioned linear systems. Without this modification often no convergence was achieved with the approach presented in Chapter 4.

2.2.3 Finite Volume Method

Finite volume methods are subdivided into vertex and cell centred methods. The names indicate where in the triangulation the unknowns are

associated to. We will only consider cell centred finite volume methods and restrict us to the case of low order discretizations.

For the cell centred finite volume method we assume that either for each element of our grid there is a circumscribed circle around it or that each element is a Voronoi region. In the first case the unknowns are associated with the centres of the circumscribed circle and in the latter case with the centres of mass of the Voronoi regions. In both cases we will call these points cell centres. The elements of the mesh are also called boxes for this method. As trial and test functions we will use the space $\mathcal{V}_0$ of discontinuous functions that are piecewise constant on the elements of the triangulation.

Analogous to the discontinuous Galerkin method we fix the direction of the unit normal of each inner edge $e \in \mathcal{E}^I$ and define the jump operator $[\![\cdot]\!]$ accordingly. Let $l : \mathcal{E} \to \mathcal{T}$ and $r : \mathcal{E} \to \mathcal{T}$ be the function that returns the element where the unit normal n_e starts and where it points to, respectively. We define the function $d : \mathcal{T} \times \mathcal{T} \to \mathbb{R}^n$ that returns the distance between the centres of two cells. For a boundary edge e let the function $d : \mathcal{E} \to \mathbb{R}$ return the distance to the cell centre of the corresponding element. We use the approximation $\widetilde{u} : \mathcal{T} \to \mathbb{R}$ of a function at the cell centre of an element. One suitable approximation is $\widetilde{u}(\tau) = \frac{1}{m(\tau)} \int_\tau u \, dx$, where $m(\tau)$ denotes the Lebesgue measure of element τ.

Then we can write the cell-centred finite volume scheme in a discrete weak form as:
find $u \in \mathcal{V}_0$ such that

$$\mathcal{A}_{FV}(u, v) = \mathcal{L}_{FV}(v) \quad \forall v \in \mathcal{V}_0 \, .$$

Here the bilinear form is given by

$$\mathcal{A}_{FV}(u, v) = -\sum_{e \in \mathcal{E}^I} \left(\widetilde{\mathsf{K}} \frac{\widetilde{u}(r(e)) - \widetilde{u}(l(e))}{d(r(e)), l(e))}, [\![v]\!] \right)_e + \sum_{e \in \mathcal{E}^D} \left(\mathsf{K} \frac{\widetilde{u}(l(e))}{d(e)}, [\![v]\!] \right)_e \, , \quad (2.8)$$

with $\widetilde{\mathsf{K}}$ being the harmonic average of the permeabilities of the two adjacent cells, and the linear form is defined as

$$\mathcal{L}_{FV}(v) = \sum_{\tau \in \mathcal{T}} (f, v)_\tau + \sum_{e \in \mathcal{E}^N} (g_N, v)_e + \sum_{e \in \mathcal{E}^D} \left(\mathsf{K} \frac{g_D}{d(e)}, [\![v]\!] \right)_e \, . \quad (2.9)$$

For many people familiar with cell centred finite volume methods the above representation of the method in terms of a linear and a bilinear form may seem rather artificial. It is used here to have a common representation of the discrete problems for all discretizations. This allows

us to describe our solver in a uniform representation regardless of the discretization method used.

Clearly, due to the support of our basis functions and the jump formulation the above is equivalent to the traditional form being

$$\sum_{e\in\partial\tau\cap\mathcal{E}^I}\int_e \widetilde{\mathsf{K}}\frac{\widetilde{u}(r(e))-\widetilde{u}(l(e))}{d(r(e)),l(e))}\,ds+\sum_{e\in\partial\tau\cap\mathcal{E}^D}\int_e \mathsf{K}\frac{\widetilde{u}(l(e))-\widetilde{g}_D(e)}{d(l(e))}\,ds = \int_\tau f\,dx+\sum_{e\in\partial\tau\cap\mathcal{E}^N}\int_e g_N\,ds \quad \forall\tau\in\mathcal{T}\,.$$

2.3 Algebraic Notation and Blocking Strategy

In the previous sections we described how to discretize partial differential equations. This procedure results in a linear system of equations that is ready to be represented in matrix and vector data structures. Up to now we did neither look at the ordering of the degrees of freedom nor did we think about how these degrees of freedom are associated to physical entities. That is, we neglected important structural information of the discretization of partial differential equations. But this structure is in most cases naturally there and should be exploited. One should do this whenever there is a gain in efficiency to be expected.

2.3.1 Higher-order Methods

When considering higher order conforming finite element methods with trial and test functions of polynomial degree $k > 1$, the degrees of freedom are not only associated with the vertices of the grid. For example for triangles and polynomial degree $k = 2$ there are additional degrees of freedom associated with the edge midpoints. For rectangular elements and $k = 2$ there are degrees of freedom associated with the vertices, the edge midpoints and the cell centres. Instead of creating a scalar matrix one could block the degrees of freedom according to the different kind of geometric entities they are associated with, namely the vertices, the edges, and the interior of the cells. For two dimensions this kind of blocking is illustrated in Figure 2.2 for triangles and polynomial degree $k = 3$. In this and the following figures all unknowns associated with the same matrix block have the same colour. This kind of blocking would result in a block matrix with non-zero blocks that are themselves sparse matrices.

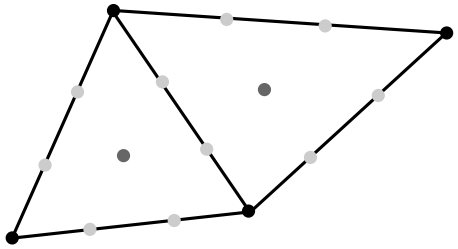


Figure 2.2: P_3 codim blocks

2.3.2 Systems of PDE

We consider a linear system of q partial differential equations for q functions $u_1, \ldots, u_q$ given in the form

$$\sum_{j=1}^{q} L_{ij} u_j = f_i\,, \quad i = 1, \ldots, q\,,$$

with differential operators L_{ij}. A prominent example is the Stokes problem

$$\begin{pmatrix} -\Delta & \nabla \\ \nabla\cdot & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix}.$$

For the variational formulation, let $V^1, \ldots, V^q$ be appropriate Hilbert spaces, and suppose that

$$\mathcal{A}_{ij}(\cdot,\cdot) : V^i \times V^j \to \mathbb{R}\,, \quad i,j = 1, \ldots, q$$

are continuous bilinear forms, and let

$$L_i : V^i \to \mathbb{R}, \quad i = 1, \ldots, q$$

be appropriately defined linear forms. Then we consider the following problem: find $u = (u_1, \ldots, u_n) \in V = V^1 \times \ldots \times V^q$ such that

$$\sum_{j=1}^{n} \mathcal{A}_{ij}(u_j, v) = L_i(v) \quad \forall v \in V^i, i = 1 \ldots, q\,.$$

Using the above described Galerkin approach to approximate the Hilbert spaces $V^1, \ldots, V^q$ with finite spaces $V_h^1, \ldots, V_h^q$ of dimension $n_1, \ldots, n_q$, respectively, and choosing appropriate basis functions results in a linear system of algebraic equations.

Let $N_1 = 0$, $N_k = N_{k-1} + n_k$ for $0 < k \leq q$, and $\Phi_i = \{\varphi_{N_i+1}, \ldots, \varphi_{N_i+n_i}\}$ be a basis of the vector space V_h^i. Depending on the ordering of the unknowns, several blocking approaches arise.

One approach of numbering and blocking the unknowns is called *equation-based blocking*. The equations are ordered like the basis functions and each equation represents a row of the resulting linear system on its own. That is, no blocking is used. Suppose the ordering of the basis functions is as described by the indices used above and let $g(i) = \max\{k \mid N_k \leq i\}$ be the function that selects the index of the space V_h^k that φ_i is a basis function of. Then this approach results in a scalar linear system $\mathsf{A}\mathbf{u} = \mathbf{f}$, where the entries of the matrix and right hand side vector are $\mathsf{A}_{ij} = \mathcal{A}_{g(i),g(j)}(\varphi_j, \varphi_i)$ and $\mathbf{f}_i = \mathcal{L}_{g(i)}(\varphi_i)$, respectively.

In the *unknown-based blocking* approach all basis functions that belong to the same space V_h^k are blocked together. Clearly, this results in a linear system

$$\begin{pmatrix} \mathsf{A}_{11} & \ldots & \mathsf{A}_{1q} \\ \vdots & \ddots & \vdots \\ \mathsf{A}_{q1} & \ldots & \mathsf{A}_{qq} \end{pmatrix} \begin{pmatrix} \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_q \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_q \end{pmatrix},$$

where the matrix entries $\mathsf{A}_{ij} \in \mathbb{R}^{n_i \times n_j}$ are themselves (sparse) matrices with entries given by $(\mathsf{A}_{ij})_{kl} = \mathcal{A}_{ij}(\varphi_{N_j+l}, \varphi_{N_i+k})$. The vector entries $\mathbf{f}_i \in \mathbb{R}^{n_i}$ are themselves vectors with entries $(\mathbf{f}_i)_k = \mathcal{L}_i(\varphi_{N_i+k})$.

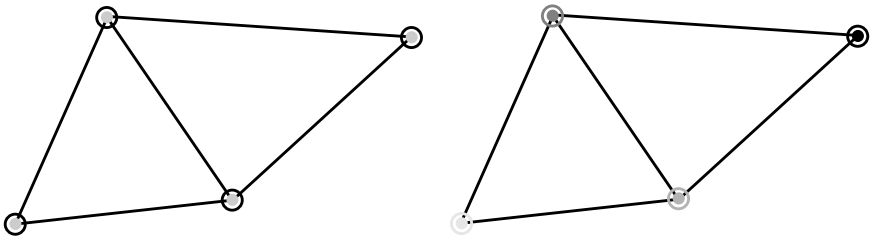
As long as V_i^h, $i = 1, \ldots, q$, all use the same nodal basis, it is possible to block all basis functions associated with one point together. The resulting linear operator is a sparse matrix which has small dense matrices as its entries. Supposing that the dimension of V_h^i is m for all i, the resulting linear system looks like

$$\mathsf{A}\mathbf{x} = \begin{pmatrix} \mathsf{A}_{11} & \ldots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{m1} & \ldots & A_{mm} \end{pmatrix} \begin{pmatrix} \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_m \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_m \end{pmatrix} = \mathbf{f},$$

$$\mathsf{A}_{ij} = \begin{pmatrix} \mathcal{A}_{11}(\varphi_{N_1+j}, \varphi_{N_1+i}) & \ldots & \mathcal{A}_{1q}(\varphi_{N_q+j}, \varphi_{N_1+i}) \\ \vdots & \ddots & \vdots \\ \mathcal{A}_{q1}(\varphi_{N_1+j}, \varphi_{N_q+i}) & \ldots & \mathcal{A}_{qq}(\varphi_{N_q+j}, \varphi_{N_q+i}) \end{pmatrix} \in \mathbb{R}^{q \times q},$$

$x_i, b_i = (\mathcal{L}_k(\varphi_{N_k+i}))_{k=1}^q \in \mathbb{R}^q$, $i = 1, \ldots, m$. Note that most of the matrix blocks A_{ij} are zero. We will call this blocking strategy *point-based blocking*.

This kind of blocking approach will be needed in the next chapter. Whenever we have a matrix A with equally sized matrix blocks in $\mathbb{R}^{k \times k}$, we



(a) Unknown-based blocking (b) Point-based blocking

Figure 2.3: Blocking Strategies for PDE systems

say in abuse of notation that for such a matrix $A \in \mathbb{K}^{n \times n}$ holds. Here we suppose that $\mathbb{K} = \mathbb{R}^k$ for a fixed $k \in \mathbb{N}$.

In Figure 2.3 we illustrate the unknown-based and point-based blocking strategies. One unknown is represent by a circle and the other one by a ring. Same colours identify degrees of freedom in the same matrix block.

2.3.3 Discontinuous Galerkin

For discontinuous Galerkin methods we can use several geometric entities to associate our basis functions with. How this is possible depends on the chosen basis of the DG trial function space $\mathcal{V}_k$. For example Hemker et al, Hemker et al. [2003, 2004], consider cell-based as well as point-based blocking. They prefer the latter as in this case traditional smoothers are more stable and have a better smoothing property. For the case of basis functions with total degree less or equal to one Figure 2.4 illustrates the blocking they use for their point-based blocking approach.

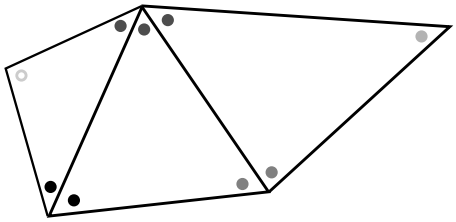


Figure 2.4: Hemker blocks for DG

Depending on the chosen set of basis functions there might not be any natural association with points. Therefore, we will pursue the *cell-based blocking* approach when dealing with discontinuous Galerkin methods.

The discontinuity of the functions of our DG trial function space guides this natural way of choosing the blocks. Let Φ be the chosen basis of the DG trial function space $\mathcal{V}_k$. Then for each of the elements $\tau \in \mathcal{T}$ we can select all basis functions $\Phi_\tau = \{\phi \in \Phi \mid \operatorname{supp} \phi \subset \tau\}$ with support just in that element. The span of these basis functions forms a subspace

$$\mathcal{V}_\tau = \operatorname{span} \Phi_\tau \subset \mathcal{V}_k$$

of our trial function space. And the decomposition of $\mathcal{V}_k$ into these subspaces is additive, that is

$$\bigotimes_{\tau \in \mathcal{T}} \mathcal{V}_\tau = \mathcal{V}_k \,.$$

We fix an arbitrary numbering $I_\mathcal{T} : \mathcal{T} \to \mathbb{N}$ of our grid elements. And use it to induce a numbering $I_\Phi : \Phi \to \mathbb{N}$ of our basis functions with the following property: Let $\tau, \upsilon \in \mathcal{T}$ be two arbitrary grid elements with $I_\mathcal{T}(\tau) < I_\mathcal{T}(\upsilon)$. Then the numbering of the associated basis functions fulfils

$$I_\Phi(\phi) < I_\Phi(\psi)\,, \quad \phi \in \Phi_\tau,\ \psi \in \Phi_\upsilon \,.$$

Let $s : \mathcal{T} \to \mathbb{N}$ and $e : \mathcal{T} \to \mathbb{N}$ be the functions returning the lowest and highest index of the basis functions associated with an element $\tau \in \mathcal{T}$. Then using the described blocking approach our linear system is represented by

$$\begin{pmatrix} \mathsf{A}_{\tau_1\tau_1} & \dots & A_{\tau_1,\tau_M} \\ \vdots & \ddots & \vdots \\ A_{\tau_M\tau_1} & \dots & A_{\tau_M\tau_M} \end{pmatrix} \begin{pmatrix} u_{\tau_1} \\ \vdots \\ u_{\tau_M} \end{pmatrix} = \begin{pmatrix} f_{\tau_1} \\ \vdots \\ f_{\tau_M} \end{pmatrix}$$

with the matrix entries being dense small matrices

$$A_{\tau\upsilon} = \begin{pmatrix} \mathcal{A}(\phi_{s(\upsilon)}, \phi_{s(\tau)}) & \dots & \mathcal{A}(\phi_{e(\upsilon)}, \phi_{s(\tau)}) \\ \vdots & \ddots & \vdots \\ \mathcal{A}(\phi_{s(\upsilon)}, \phi_{e(\tau)}) & \dots & \mathcal{A}(\phi_{e(\upsilon)}, \phi_{e(\tau)}) \end{pmatrix}$$

and the vector entries being vectors

$$\mathbf{f}_\tau = \begin{pmatrix} \mathcal{L}(\phi_{s(\tau)}) \\ \vdots \\ \mathcal{L}(\phi_{e(\tau)}) \end{pmatrix}$$

themselves. For varying polynomial degree per element the described blocking is illustrated in Figure 2.5.

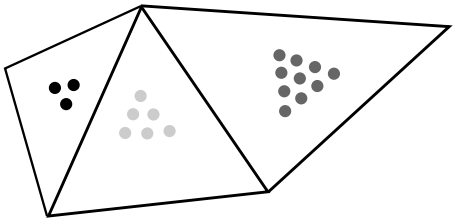


Figure 2.5: DG p element blocks

2.3.4 Matrix and Vector Data Structures

The presented and other recursive block structure can be recreated in special data structures for both matrices and vectors. Using these data structures, it is straight forward to formulate block versions of iterative methods. Furthermore, it is possible to exploit caching effects and make this fixed block structure already known to the compiler. Both can then be used to produce efficient simulation code.

In this thesis the block structure is mapped using the Iterative Solver Template Library (ISTL). The capabilities and advantages as well as details of the implementation can be found in Chapter 7.1.

3 Algebraic Multigrid Methods

In this chapter we describe our sequential algebraic multigrid method. We will use the framework of Schwarz methods for this as many preconditioners and solvers can be described in this framework. Among them are the smoothers used in thesis as well as the algebraic multigrid methods we develop.

3.1 Schwarz Methods

In this section we formally introduce Schwarz methods. The reader is referred to the monographs Smith et al. [1996] and Toselli and Widlund [2005] for more details. The presentation is inspired by the latter publication.

Let $\mathcal{V}$ be a real finite-dimensional Hilbert space equipped with the inner product $(\cdot,\cdot)$ inducing the norm $\|\cdot\|$. Given a coercive bilinear form,

$$\mathcal{A}(\cdot,\cdot) : \mathcal{V} \times \mathcal{V} \to \mathbb{R}\,,$$

and an element $f \in \mathcal{V}'$ of its dual, we consider the general problem of finding $u \in \mathcal{V}$, such that

$$\mathcal{A}(u,v) = f(v), \quad \forall v \in \mathcal{V}\,. \tag{3.1}$$

Note, that once a basis is given a function $u \in \mathcal{V}$ is uniquely determined by a set of degrees of freedom. Here and later on, we use the same notion for describing the function space and the spaces of degrees of freedom, and functions and corresponding vectors of degrees of freedom. Similarly, we will use the same notation for a linear functional $f \in \mathcal{V}$ and corresponding vector with elements obtained by applying f to the basis functions of $\mathcal{V}'$. Recall, that if A is the stiffness matrix relative to the bilinear form $\mathcal{A}(\cdot,\cdot)$ and the given basis, our problem is equivalent to the linear system

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

with A positive definite.

In Schwarz methods (also called subspace correction methods) the space $\mathcal{V}$ is subdivided into a family of subspaces $\{\mathcal{V}_i, i = 0, \ldots, N, \mathcal{V}_i \subset \mathcal{V}\}$ and according extension operators are defined

$$R_i^T : \mathcal{V}_i \to \mathcal{V} .$$

We assume that $\mathcal{V}$ admits the following decomposition

$$\mathcal{V} = \sum_{i=0}^{N} R_i^T \mathcal{V}_i$$

into the above subspaces.

On the chosen subspaces $\{\mathcal{V}_i\}$ we introduce the local bilinear forms

$$\widetilde{\mathcal{A}}_i(\cdot,\cdot) : \mathcal{V}_i \times \mathcal{V}_i \to \mathbb{R}, \quad i = 0, \ldots, N$$

and the local stiffness matrices associated with them,

$$\widetilde{\mathsf{A}}_i : \mathcal{V}_i \to \mathcal{V}_i .$$

A special case is to use the original bilinear form on the subspaces. In this case we choose

$$\widetilde{\mathcal{A}}_i(u_i, v_i) = \mathcal{A}(R_i^T u_i, R_i^T v_i), \quad u_i, v_i \in V_i$$

and find our corresponding operator to be

$$\widetilde{A}_i = R_i A R_i^T .$$

Note, that in this case we use exact solvers for our local problems.

Our subspace correction methods will be described in terms of projection-like operators

$$P_i = R_i^T \widetilde{P}_i : \mathcal{V} \to R_i^T \mathcal{V}_i \subset \mathcal{V}, \quad i = 0, \ldots, N, \tag{3.2}$$

where $\widetilde{P}_i : \mathcal{V} \to \mathcal{V}_i$ is defined by

$$\widetilde{\mathcal{A}}_i(\widetilde{P}_i u, v_i) = \mathcal{A}(u, R_i^T v_i) \quad \forall v_i \in \mathcal{V}_i .$$

The P_i can be written in matrix form as

$$\mathsf{P}_i = \mathsf{R}_i^T \widetilde{\mathsf{A}}_i^{-1} \mathsf{R}_i \mathsf{A}, \quad 0 \le i \le N .$$

Once a set of subspaces and local bilinear forms are given, we can define various subspace correction methods. The most prominent representatives are the purely additive and multiplicative subspace correction methods.

Let $\mathcal{P} \subset \{P_0, \ldots, P_N\}$ be a tuple of projection-like operators defined according to (3.2). Note, that the index i of P_i does not need to correspond to the index of a subspace and that $P_i = P_j$, $i \neq j$ is allowed. Then the *additive Schwarz operator* of this tuple is defined as

$$P^{\mathrm{ad}}(\mathcal{P}) = \sum_{P \in \mathcal{P}} P \,.$$

The *multiplicative Schwarz operator* is defined as

$$P^{\mathrm{mu}}(\mathcal{P}) = I - E^{\mathrm{mu}}(\mathcal{P}) \,,$$

with the error propagator $E^{\mathrm{mu}}(\mathcal{P})$ being defined by

$$E^{\mathrm{mu}}(\mathcal{P}) = (I - P_N)(I - P_{N-1}) \ldots (I - P_0) \,.$$

All the above mentioned operators are actually preconditioned operators for the original matrix A and can be written as the product of a suitable preconditioner and A. The preconditioner can be specified in terms of extension $\{R_i^T\}$, restrictions $\{R_i\}$, and the local operators $\{\widetilde{\mathsf{A}}_i\}$. For the additive method this is straight forward and reads

$$\mathsf{P}^{\mathrm{ad}}(\{P_0, \ldots, P_N\}) = (\mathsf{A}^{\mathrm{ad}})^{-1}\mathsf{A} \,, \quad (A^{\mathrm{ad}})^{-1} = \sum_{i=0}^{N} \mathsf{R}_i^T \widetilde{\mathsf{A}}_i^{-1} \mathsf{R}_i \,.$$

Note, that many of the traditional stationary iterative methods can be categorised as subspace correction methods. Among them are the Jacobi and Gauss-Seidel method. These are often used as smoothers in multigrid methods and the development of algebraic multigrid methods is based on their properties.

Let the decomposition of our Hilbert space $\mathcal{V} = \sum_{i=0}^{N} R_i^T \mathcal{V}_i$ be non-overlapping. That is, $\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$ for $i \neq j$. Then, if we use exact direct solvers on all subspaces $\mathcal{V}_i$, the additive method $(\mathsf{A}^{\mathrm{ad}})^{-1}$ and multiplicative method $(\mathsf{A}^{\mathrm{mu}})^{-1}$ are the Jacobi and Gauss-Seidel method, respectively. If each $\mathcal{V}_i$ is spanned by only one basis function, these are the scalar versions. Otherwise they are block version of the smoothers.

3.1.1 Multi-Level Methods

Unfortunately, single level methods are most effective only for a small number of subdomains. For a big number of subdomains this effectiveness gets lost. Applying single level Schwarz methods only reduces the high frequency error components while only leading to a humble reduction of the error itself. With today's available computing power and therefore tackled problem sizes, one is often confronted with a rather huge number of subdomains. This will be especially true for the method we will introduce in the next chapter which uses rather small local problems.

A remedy to this problem is to use an additional coarse problem or even a (nested) sequence of coarser problems. The idea of multi-level methods is to approximate the smooth error, obtained after some iteration steps using our local problems, in a function space $\mathcal{W} \subset \mathcal{V}$. This finite dimensional function space is assumed to be coarser than the original one. That is, either a coarser grid is used or a reduced basis (spanning the global domain) of the original on the same grid is used. The latter will be the case in our method described in the next chapter.

Methods using just one coarse space are called two-level methods. The idea can be applied recursively. The resulting method is then called a multi-level method. We can describe these multi-level methods in the abstract Schwarz framework defined above. For simplicity we restrict ourselves to the case of nested coarse spaces. With algebraic multi-level methods in mind we will denote the coarsest space with $\mathcal{V}^L$, where L is the largest index used. This contrasts traditional two-level domain decomposition literature but seems more natural as the algorithm, that constructs the coarse spaces, will start with the fine function space and build the coarser ones.

Let $\mathcal{V}^l$, $l = 0, \dots L$, $\mathcal{V}^i \subset \mathcal{V}^j$ for $i > j$, $\mathcal{V}^0 = \mathcal{V}$, be a nested sequence of finite-dimensional Hilbert spaces. We decompose each such global space $\mathcal{V}^l$, $0 < l \leq L$ into n_l local subspaces $\mathcal{V}^l_k \subset \mathcal{V}^l$, $k = 0, \dots, n_l - 1$. As before we consider corresponding extension operators from the coarse spaces

$$(R^l)^T : \mathcal{V}^l \to \mathcal{V}^0 = \mathcal{V}$$

as well as from the local subspaces of the coarse spaces

$$(R^l_k)^T : \mathcal{V}^l_k \to \mathcal{V}^0 = \mathcal{V}\,.$$

We assume that $\mathcal{V}^l = \sum_{k=0}^{n_l-1} (R^l_k)^T V^l_k$ holds.

Together with appropriate local bilinear forms $\widetilde{\mathcal{A}}^l_k(\cdot,\cdot)$, and $\widetilde{\mathcal{A}}^l(\cdot,\cdot)$ we can define corresponding projection-like operators P^l_k and P^l as above.

Usually exact solvers are used for these problems. Using these operators, we can now define additive and multiplicative multi-level methods. One possibility for the projection operator tuple for a multi-level method is

$$\mathcal{P} = \{P_0^0, \ldots, P_{n_0-1}^0, P_0^1, \ldots, P_{n_{L-1}-1}^{L-1}, P^L, P_{n_{L-1}-1}^{L-1}, \ldots, P_0^{L-1}, \ldots, P_0^0\}.$$

For the multiplicative version this a symmetric multi-level method with one step of Gauss-Seidel for pre-smoothing and one step of backwards Gauss-Seidel as post-smoothing.

For the multiplicative and hybrid versions of the multi-level methods with more than one coarse level the usual formulation does not use the Schwarz framework. Instead on each level the so-called smoothing operator S^l is defined. This may be the multiplicative operator according to the tuple $\mathcal{P}^l = (P_0^l, \ldots P_{n_l-1}^l)$. In addition, prolongation $\widetilde{P}^l : \mathcal{V}^l \to \mathcal{V}^{l-1}$ and restriction operators $\widetilde{R}^l : \mathcal{V}^{l-1} \to \mathcal{V}^l$ are needed. These are defined as $\widetilde{R}^l = R^l(R^{l-1})^T$ and $\widetilde{P}^l = (\widetilde{R}^l)^T$. Using these ingredients we are able to define the multi-level algorithm in the traditional formulation.

Let $\mathbf{b}^l, \mathbf{x}^l \in \mathcal{V}^l$ and $\widetilde{\mathcal{A}}^l : \mathcal{V}^l \times \mathcal{V}^l \to \mathbb{R}$ be our appropriately defined bilinear forms on each level $l = 0, 1, \ldots, L$. How these are constructed will be subject to the next section. Furthermore, let S^l be our smoothing operators on level $l = 0, 1, \ldots, L$.

Then the *multi-level algorithm* (or *multigrid algorithm*) with ν_1 steps of pre-smoothing and ν_2 steps of post-smoothing described in Algorithm 3.1 defines an approximate inverse of A^0 in the function space $\mathcal{V}^0$. The parameter γ is used to choose the kind of multigrid cycle used. For $\gamma = 1$ one step of the multigrid cycle is called V-cycle and for $\gamma = 2$ it is called W-cycle.

Algorithm 3.1 Multi-Level Method

procedure MLM(l, ν_1, ν_2, γ, $\{\mathsf{A}^l\}_{l=0}^L$, $\{\mathsf{S}^l\}_{l=0}^L$, $\{\mathbf{x}^l\}_{l=0}^L$, $\{\mathbf{b}^l\}_{l=0}^L$, $\{\widetilde{\mathsf{R}}^l\}_{l=1}^L$, $\{\widetilde{\mathsf{P}}^l\}_{l=1}^L$)
  if $l == L$ **then**
    $\mathbf{x}^L \leftarrow (\mathsf{A}^L)^{-1}\mathbf{b}^L$ ▷ Solve exactly on the coarse level
  else
    for int i=0; i < ν_1; i++ **do** ▷ Presmoothing
      $\mathbf{x}^l \leftarrow \mathbf{x}^l + \mathsf{S}^l(\mathbf{b}^l - \mathsf{A}^l\mathbf{x}^l)$
    end for
    $\mathbf{b}^{l+1} \leftarrow \widetilde{\mathsf{R}}^{l+1}(\mathbf{b}^l - \mathsf{A}^l\mathbf{x}^l)$ ▷ Restrict defect
    $\mathbf{x}^{l+1} \leftarrow 0$
    for int $i = 0$; $i < \gamma$, $i++$ **do**
      MLM($l+1$, ν_1, ν_2, γ, $\{\mathsf{A}^l\}_{l=0}^L$, $\{\mathsf{S}^l\}_{l=0}^L$, $\{\mathbf{x}^l\}_{l=0}^L$, $\{\mathbf{b}^l\}_{l=0}^L$, $\{\widetilde{\mathsf{R}}^l\}_{l=1}^L$, $\{\widetilde{\mathsf{P}}^l\}_{l=1}^L$)
    end for
    $\mathbf{x}^l \leftarrow \mathbf{x}^l + \widetilde{\mathsf{P}}^{l+1}\mathbf{x}^{l+1}$;
    for int i=0; i < ν_2; i++ **do** ▷ Postsmoothing
      $\mathbf{x}^l \leftarrow \mathbf{x}^l + \mathsf{S}^l(\mathbf{b}^l - \mathsf{A}^l\mathbf{x}^l)$
    end for
  end if
end procedure

3.2 Scalar Aggregation AMG

In the last section we introduced the multi-level Algorithm. What we did not describe so far was how the different levels are created. One straight forward way is to start either with the fine or coarse level grid and then create the other grids by coarsening and refining the grids uniformly, respectively. The resulting linear systems may then be created either by rediscretizing on each grid level or by using a Galerkin product using the grid level interpolation operators.

While this fixed scheme works well for some problems there are other scenarios where either the coarsening needs to be adapted to the problem to solve, problem specific interpolation operators or robust smoothers need to be chosen. All these tasks are far from being trivial for complex grids and real world problems in three dimensions.

Algebraic multigrid methods (AMG) use an opposite approach. Traditionally they rely on rather simple smoothers, such as the Jacobi or the Gauss-Seidel method. Using the fine level discretization matrix these methods automatically construct the matrices of the coarser levels. To guarantee an efficient interplay of the smoothing operators and the coarse

grid correction, the interpolation and coarse level matrix are constructed based on properties of the smoother for the current matrix.

There exist two big classes of algebraic multigrid methods. They differ in the way the coarse level matrices are constructed. Ruge and Stüben proposed a variable based approach in Ruge and Stüben [1987]. They divide the degrees of freedom into two sets. The set of unknowns that only appear on the fine level and the set of unknowns that are present on both the coarse and the fine level. The other big class of algebraic multigrid methods are the ones that are aggregation based, see Braess [1995], Vaněk et al. [1996a,b], Raw [1985]. These methods build aggregates of the fine level degrees of freedom until these aggregates form a non-overlapping partitioning of all fine level degrees of freedom. Then each aggregate is represented by one coarse level unknown. We will use the latter approach here.

3.2.1 Algebraic Smoothness

The coarsening of our algebraic multigrid method will be governed by the properties of the smoothers, namely the damped Jacobi method and the Gauss-Seidel method. To examine this behaviour we need to introduce the following discrete Sobolev scalar products and norms.

Definition 3.1. Let $A \in \mathbb{K}^{I\times I}$ be symmetric and positive definite, $D := \mathrm{diag}(A)$ be the diagonal matrix containing the diagonal blocks of A as nonzero blocks, $u \in \mathbb{K}^I$, $v \in \mathbb{K}^I$ and $\langle\cdot,\cdot\rangle$ denote the Euclidean scalar product. Then we define the following scalar products

$$\langle \mathbf{u}, \mathbf{v}\rangle_0 := \langle D\mathbf{u}, \mathbf{v}\rangle \tag{3.3}$$

$$\langle \mathbf{u}, \mathbf{v}\rangle_1 := \langle A\mathbf{u}, \mathbf{v}\rangle \tag{3.4}$$

$$\langle \mathbf{u}, \mathbf{v}\rangle_2 := \langle D^{-1}A\mathbf{u}, A\mathbf{v}\rangle\,, \tag{3.5}$$

along with their associated norms $\|\cdot\|_i$, $i = 0, 1, 2$. Here $\langle\cdot,\cdot\rangle_1$ is called the *energy inner product* and $\|\cdot\|_1$ the *energy norm*.

Let $\mathbf{u}^*$ be the exact solution of $A\mathbf{x} = \mathbf{b}$ and let $\mathbf{u}$ be the current guess of it. Then the current residual $\mathbf{r}$ is defined by $\mathbf{r} = \mathbf{b} - A\mathbf{u}$ and the error by $\mathbf{e} = \mathbf{u} - \mathbf{u}^*$. Therefore, $\|e\|_1^2 = \langle \mathbf{r}, \mathbf{e}\rangle$ is the scalar product of the residual and the error, and $\|e\|_2^2 = \langle D^{-1}\mathbf{r}, \mathbf{r}\rangle$ is the scalar product of the residual and itself scaled with the inverse of the diagonal part of A.

The damped Jacobi and Gauss-Seidel method fulfil the following smoothing property:

Theorem 3.2 (Ruge and Stüben [1987], Clees [2005])**.** *Let* $\mathsf{A} \in \mathbb{K}^{I\times I}$ *be symmetric and positive definite. Then for the smoothing operator* S *of the damped Jacobi and the Gauss-Seidel method the inequalities*

$$\|\mathsf{S}\mathbf{e}\|_1^2 \leq \|\mathbf{e}\|_1^2 - a\|\mathbf{e}\|_2^2, \tag{3.6}$$
$$\|\mathsf{S}\mathbf{e}\|_1^2 \leq \|\mathbf{e}\|_1^2 - a\|\mathsf{S}\mathbf{e}\|_2^2 \tag{3.7}$$

hold for arbitrary $\mathbf{e} \in \mathbb{K}^I$ *with some constant* $a > 0$*. These inequalities are called smoothing properties.*

Proof. Note that for damped Jacobi method with the smoothing operator $\mathsf{S} = \mathsf{I} - \omega\mathsf{D}^{-1}\mathsf{A}$,

$$\|\mathsf{S}\mathbf{e}\| = \langle \mathsf{A}(\mathsf{I} - \mathsf{D}^{-1}\mathsf{A})e, e\rangle = \langle \mathsf{A}\mathbf{e}, \mathbf{e}\rangle - \omega\langle \mathsf{D}^{-1}\mathsf{A}e, e\rangle = \|\mathbf{e}\|_1^2 - \omega\|\mathbf{e}\|_2^2$$

holds. The complete proof is presented in Ruge and Stüben [1987] for $\mathbb{K} = \mathbb{R}$ and Clees [2005] for the block versions of the iterative methods. □

Because of (3.6) the error $\mathbf{e}$ is reduced well by these methods as long as $\|\mathbf{e}\|_2$ is comparable to $\|e\|_1$. The error reduction becomes insufficient for the case that

$$\|e\|_2 \ll \|e\|_1 \tag{3.8}$$

holds. (This implies that, at least on average $|r_i| \ll a_{ii}|e_i|$ holds componentwise for the current error e and residual r.) In this case we speak of the *algebraic smoothness* of the error $\mathbf{e}$.

Definition 3.3. The error $\mathbf{e}$ is called *algebraically smooth* if it is not sufficiently reduced by applying the smoothing operator S, i. e.

$$\|\mathsf{S}\mathbf{e}\|_1 \approx \|\mathbf{e}\|_1 . \tag{3.9}$$

This does not necessarily mean that the error would also be considered smooth in a geometrical sense. Let us have a look at the simple anisotropic diffusion problem

$$\begin{aligned} -\nabla \cdot \left(\begin{smallmatrix} 1 & 0 \\ 0 & 10^{-3} \end{smallmatrix}\right) \nabla u &= 1 \quad \text{on } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned}$$

on $\Omega = (0,1)^2$ discretized via cell-centred finite volumes with twenty cells in each direction. Figure 3.1 shows the initial defect and the one after five and ten Gauss-Seidel steps. While the defect after ten steps is

still highly oscillatory in one direction it is nevertheless considered algebraically smooth,

Because of

$$\|e\|_1^2 = \langle \mathsf{A}\mathbf{e}, \mathbf{e}\rangle = \langle \mathsf{D}^{-\frac{1}{2}}\mathsf{A}\mathbf{e}, \mathsf{D}^{\frac{1}{2}}\mathbf{e}\rangle \leq \|\mathsf{D}^{-\frac{1}{2}}\mathsf{A}\mathbf{e}\|\|\mathsf{D}^{\frac{1}{2}}\mathbf{e}\| = \|e\|_2\|e\|_0$$

an algebraically smooth error implies $\|e\|_1 \ll \|e\|_0$ for symmetric positive definite matrices. Using the ith row-sum $s_i = \sum_j a_{ij}$ we write this more explicitly as

$$\langle \mathsf{A}\mathbf{e}, \mathbf{e}\rangle = \frac{1}{2}\sum_{i,j}(-a_{ij})(e_i - e_j)^2 + \sum_i s_i e_i^2 \ll \sum_i a_{ii}e_i^2 \,. \tag{3.10}$$

For the important case that $s_i \approx 0$ holds, this means that on average for each i the inequality

$$\sum_j \frac{-a_{ij}}{a_{ii}}\frac{(e_i - e_j)^2}{e_i^2} \ll 1 \tag{3.11}$$

holds. Therefore, in the direction of large negative connections algebraically smooth errors will vary slowly. That is, if $\frac{|a_{ij}|}{a_{ii}}$ is relatively large, the error e will not change much between e_i and e_j. This property will guide our coarsening approach.

Unfortunately, not all discretization matrices will have only negative off-diagonal values. There are cases where positive off-diagonal values may occur. Prominent examples are higher order discretizations and discretizations with mixed derivatives. The anisotropic diffusion problem discretized using bilinear or trilinear finite elements is another example. We will refer to this case in Subsection 3.4 with some examples. Right now, we want to see how positive off-diagonal values influence the smoothness of the error.

One class of discretization matrices with positive off-diagonal values are *essentially positive type* matrices. A symmetric positive definite matrix A belongs to this class if there exists a constant $c > 0$, such that for all $\mathbf{e}$

$$\sum_{i,j}(-a_{ij})(e_i - e_j)^2 \geq c\sum_{i,j}(-a_{ij}^-)(e_i - e_j)^2$$

with $a_{ij}^- = \frac{1}{2}(a_{ij} - |a_{ij}|)$. Therefore, we can follow the lines of Stüben [1999], Clees [2005] and extend (3.10) to

$$\frac{c}{2}\sum_{i,j}|-a_{ij}^-|(e_i - e_j)^2 + \sum_i s_i e_i^2 \ll \sum_i a_{ii}e_i^2 \,. \tag{3.12}$$

Thus we treat positive off-diagonal values as weak connections. Even if there is a positive connection between two vertices, an algebraically smooth error will not change rapidly between them as long as there exists a path of strong negative connections between them.

Up to now we always considered the case that $s_i \approx 0$. We now turn to weakly diagonally dominant matrices. If A is of this class then $a_{ii} \geq \sum_{j \neq i} |a_{ij}|$ holds for all rows i. If a symmetric positive matrix A is only approximately of this class then $t_i = a_{ii} - \sum_{j \neq i} |a_{ij}|$ is a measure of how diagonal dominant a row is. We can relate this measure to s_i using $a_{ij}^+ = \frac{1}{2}(|a_{ij}| + a_{ij})$ by

$$s_i = t_i + 2 \sum_{j \neq i} a_{ij}^+ .$$

Substituting this into (3.10) we get

$$\begin{aligned}
\langle \mathsf{A}\mathbf{e}, \mathbf{e} \rangle =& \frac{1}{2} \sum_{i,j} |a_{ij}^-| (e_i - e_j)^2 - \frac{1}{2} a_{ij}^+ (e_i - e_j)^2 + \sum_i s_i e_i^2 = \\
& \frac{1}{2} \sum_{i,j} |a_{ij}^-| (e_i - e_j)^2 + \sum_{i,j} a_{ij}^+ (2e_i^2 - \frac{(e_i - e_j)^2}{2}) + \sum_i t_i e_i^2 = \\
& \frac{1}{2} \sum_{i,j} |a_{ij}^-| (e_i - e_j)^2 + \frac{1}{2} \sum_{i,j} a_{ij}^+ (e_i^2 + e_j^2 - (e_i - e_j)^2) + \sum_i t_i e_i^2 = \\
& \frac{1}{2} \sum_i \left(\sum_{j \neq i} |a_{ij}^-| (e_i - e_j)^2 + \sum_{j \neq i} a_{ij}^+ (e_i + e_j)^2 \right) + \sum_i t_i e_i^2
\end{aligned}$$

in accordance to Stüben [1999], Clees [2005]. Assuming that A is approximately weak diagonally dominant, i.e. $t_i \approx 0$, an algebraically smooth error now satisfies

$$\sum_{j \neq i} \frac{|a_{ij}^-|}{a_{ii}} \frac{(e_i - e_j)^2}{e_i^2} + \sum_{j \neq i} \frac{a_{ij}^+}{a_{ii}} \frac{(e_i + e_j)^2}{e_i^2} \ll 1 \tag{3.13}$$

on average for each i. Therefore, for a_{ij} positive and $\frac{a_{ij}}{a_{ii}}$ relatively large the error at vertex i approximates the negative value at vertex j.

3.2.2 Aggregation Approach

Our aggregation approach uses the graph of the matrix and is guided by the algebraic smoothness of the error.

Definition 3.4. Let $\mathsf{A} = (a_{ij}) \in \mathbb{K}^{I\times I}$ be a (block) matrix mapping the tensor product $I \times I$ of the index sets to $\mathbb{K}$.

Then $G(\mathsf{A}) = (V(\mathsf{A}), E(\mathsf{A}), w_V, w_E)$ is called the *weighted graph of matrix* A. $V(\mathsf{A}) = I$ is the set of ordered vertices representing the unknowns and $E(\mathsf{A}) = \{(i_1,j_1),(i_2,j_2),\ldots,(i_m,j_m)\}$, $i_k \neq j_k$, is the set of directed edges such that (i,j) only exists if and only if $a_{ji} \neq 0$. Edge (i,j) is starting at vertex i and pointing to vertex j.

The functions

$$w_V : V(A) \to \mathbb{R} : \quad i \mapsto w_N(i) \quad \text{and,} \tag{3.14}$$

$$w_E : E(A) \to \mathbb{R} : \quad (i,j) \mapsto w_E((i,j)), \quad i \neq j\,. \tag{3.15}$$

are weight functions used to classify the vertices and nodes, respectively

The weight functions w_V and w_E will be needed for our extension to systems of PDE described later. When dealing with systems we usually solve these systems fully coupled. This results in a matrix with point-based blocking. That is $\mathbb{K} = \mathbb{R}^k$ for small k. In this case the weight functions have to be appropriately defined. For a scalar matrix $\mathsf{A} = (a_{ij})_{i\in I, j\in I} \in \mathbb{R}^{I\times I}$ we will simply define them as

$$w_V(i) := a_{ii} \quad \text{and} \quad w_E((i,j)) = a_{ji}^{-}\,, \tag{3.16}$$

respectively. Note, that this approach treats positive off-diagonal values as zero.

We base our coarsening on the conclusion from equation (3.11) that a smooth error varies slowly for $\frac{|a_{ij}|}{a_{ii}}$ relatively large. Using this we can classify the edges and vertices of our graph as follows:

Definition 3.5. The *neighbours of vertex* i in the graph $G(\mathsf{A})$ of matrix A are

$$N(i) := \{j \in V(\mathsf{A}) \mid \exists (i,j) \in E(\mathsf{A})\}$$

Let

$$\gamma_{\max}(i) := \max_{k\in N(i)} \frac{w_E((k,i))\; w_E((i,k))}{w_V(i)\; w_V(k)} \tag{3.17}$$

then edge (j,i) is called *strong* if and only if

$$\frac{w_E((i,j))\; w_E((j,i))}{w_V(i)\; w_V(j)} > a\; \min(\gamma_{\max}(i), \gamma_{\max}(j)) \tag{3.18}$$

for a given threshold $0 < a < 1$.

A vertex i is called *isolated* if for a given threshold $\beta \approx 0$

$$\gamma_{\max}(i) < \beta$$

holds. By $I(V(\mathsf{A})) \subset V(\mathsf{A})$ all isolated nodes of the graph $G(A)$ are denoted.

Let $i \in V(\mathsf{A})$ and $j \in V(\mathsf{A})$, then we say that vertex i influences vertex j and vertex j depends on vertex i if and only if the edge $(i,j) \in E(\mathsf{A})$ exists and is strong.

Let $N_a(i) \subset N(i)$ be the set of all neighbours of vertex i connected to it via a strong edge when using the threshold a.

Note, that (3.17) and (3.18) are very similar to the traditional AMG strength of connection measure. In Ruge and Stüben [1987], the classical interpolation AMG, the measure is that a connection is strong as long as

$$-a_{ij} \geq a \max_{j \neq i} |a_{ij}| .$$

This was later improved by ignoring positive off-diagonals in the right hand side of the inequality. The classical criterion was developed for symmetric positive definite M-matrices. Despite of this fact, it works for non-symmetric M-matrices like they arise in convection-diffusion problems. The improved version even works for approximately weakly diagonally dominant matrices. As will be pointed out in Subsection 3.2.3, it is not suitable for aggregation methods for problems with discontinuous diffusion coefficients. For this problems the classification of the coupling a_{ij} and a_{ji} at the interface is not consistent. While being robust to row-wise scaling of the linear system, column-wise scaling is able to break the criterion. The traditional strength of connection criterion for aggregation AMG in Vaněk et al. [1996b] is

$$|a_{ij}| \geq a \sqrt{a_{ii} a_{jj}} .$$

It assumes the matrix A to be a symmetric positive definite M-matrix. If positive off-diagonal elements exist then it might falsely classify them as strong. In contrast to Ruge and Stüben's and our measure, for the measure of Vaněk et al. a connection is not strong relative to other connections of the vertex, but because of the absolute size of the off-diagonal entry relative to the diagonal entries. With our criterion positive off-diagonal values are treated as weak connections and the criterion works even for approximately weak diagonally dominant matrices. Our criterion uses both entries a_{ij} and a_{ji} for the decision whether a_{ij} is strong. Due to taking the

minimum in the right hand side of the inequality (3.18), if a_{ij} is classified as strong the same happens to a_{ji}. This is not the case for the other criterions and is important for non-symmetric matrices like they arise for convection dominated convection-diffusion equations. If discretized using upwind finite differences both connections along the convective flux are considered strong.

For an appropriate value of our threshold a we can reinterpret Subsection 3.2.1 by the following remark. The values $a = \frac{1}{3}$ and $\beta = 10^{-5}$ were found adequate for our numerical tests.

Remark 3.6. An algebraically smooth error varies slowly along strong edges of the matrix graph.

Given a matrix $A^l \in \mathbb{K}^{I^l \times I^l}$ at level l we search for a disjoint splitting of all vertices $V(A^l)$ into aggregates $\mathcal{A}^l = \{\mathcal{A}^l_i\}_{i \in I^{l+1}}$, $I^{l+1} \subset I^l$, with $\mathcal{A}^l_i \cap \mathcal{A}^l_j = \emptyset$ for $i \neq j$. Based on this splitting we then define the prolongation operator

$$\widetilde{P}^l : \mathbb{K}^{I^{l+1}} \to \mathbb{K}^{I^l}$$

from the coarser level $l+1$ to level l as

$$(\widetilde{P})_{ij} = \begin{cases} 1 & \text{if } i \in \mathcal{A}^l_j \\ 0 & \text{otherwise} \end{cases} . \tag{3.19}$$

The corresponding restriction operator $\widetilde{R}^l = (\widetilde{P})^T : \mathbb{K}^{I^l} \to \mathbb{K}^{I^{l+1}}$ is simply the transpose of the prolongation operator. Now the corresponding matrix on level $l+1$ is defined by the rescaled Galerkin product

$$A^{l+1} = \frac{1}{\omega}(\widetilde{R}^l) A^l (\widetilde{R}^l)^T , \tag{3.20}$$

where ω is a constant (e.g. $\omega = 1.8$). This constant is actually incorporated into the prolongator where the coarse grid correction is simply scaled by ω. It was shown in Braess [1995] that this approach increases convergence of aggregation AMG. It is easy to see that the matrix entries of A_{l+1} can be computed directly by

$$A^{l+1}_{ij} = \frac{1}{\omega} \sum_{m \in \mathcal{A}^l_i} \sum_{n \in \mathcal{A}^l_j} A^l_{m,n} . \tag{3.21}$$

Starting at the fine level 0 with the matrix $\mathsf{A}^0 \in \mathbb{K}^{I^0 \times I^0}$, we get a hierarchy of matrices $\{\mathsf{A}^l\}_{l=0}^{L-1}$ along with prolongation and restriction operators by recursively applying the aggregation procedure above until the dimension

of our coarse matrix is small enough. This setup phase gives us the formal components needed for applying the multigrid Algorithm 3.1.

The crucial part is now how we build the aggregates. The major goal is to choose the aggregates in a way such that a smooth error does not change much within the aggregates. By Remark 3.6 and the smoothing properties of our iterative schemes this means that all vertices of an aggregate naturally should be connected by strong edges between each other. If we cannot satisfy this, we still insist that all vertices are connected with each other. While a smooth error will not change much between two directly connected vertices, it might still change more between two vertices that are not directly connected but influence each other by a longer path of strong connections.

Definition 3.7. A *path connecting vertex i with j* is an ordered set of edges $\{(a_k, b_k)\}_{k=0}^{p}$ with $a_0 = i$, $a_{l+1} = b_l$ for $l = 0, 1, \ldots p-1$ and $b_p = j$. The length of the path is p.

There will be many different paths between two arbitrary vertices of an aggregate. Not all paths will be equally important. But the shorter the minimal path between two vertices is, the less variations in the smooth error may occur of course.

Definition 3.8. We denote by $\mathrm{dist}(i,j)$ the distance between two vertices i and j of a graph $G(\mathsf{A})$. That is, $\mathrm{dist}(i,j)$ is the length of the shortest path connecting i with j or j with i.

The distance between two sets $B, C \subset V(\mathsf{A})$ is defined by

$$\mathrm{dist}(A,B) = \min_{i \in B, j \in C} \mathrm{dist}(i,j)\,.$$

Of course, we are not really interested in the distance between just two vertices of an aggregate but we want to minimise the distance between all pairs of vertices in our aggregate. Therefore, we introduce a more appropriate and shorter notation.

Definition 3.9. Let $B \subset V(\mathsf{A})$ be a set of vertices of the graph $G(\mathsf{A})$ such that for all vertices $v, w \in B$ there exists a path between them not leaving B. Let $p_{\min}(v,w)$ denote for each pair of vertices the length of the shortest path connecting v and w without leaving B, i. e. all vertices along that path are also in B.

Then $\mathrm{diam}(B) = \max\{p_{\min}(v,w) \ : \ v, w \in B\}$ denotes the diameter of the set of vertices B.

Additionally, we want to preserve the sparsity of the matrix, as otherwise the computational work for solving on the coarser levels will become higher than necessary. Taking a closer look at (3.21) reveals that this can be achieved by maximising the number of connections between the vertices within each aggregate. A side effect of this is that the more strong connections there are between two vertices the less likely a smooth error is to change.

Motivated by geometric multigrid we want to achieve a prescribed coarsening rate $\#\mathcal{A}/\#V(\mathsf{A})$. This is done approximately by prescribing the minimum and maximum number of vertices allowed in an aggregate. Thus one can adjust the algorithm to one's needs, either low memory consumption sacrificing convergence or better convergence needing more memory.

Our greedy aggregation algorithm is described by Algorithm 3.2. In the algorithm iso(V) is the subset of all isolated vertices in V. Until all non-isolated vertices are aggregated, we start a new aggregate with a non-isolated vertex. The first aggregate is seeded with a vertex that has the least connections to other vertices. The other aggregates are seeded with vertices that are non-aggregated neighbours of the last aggregate. If no such vertex is present we use a non-isolated non-aggregated vertex with the least connection to non-aggregated vertices. At the same time we associate the index of the seed vertex with this new aggregate and add it to the index set I of the coarse level. The algorithm returns both the index set I for the coarse level and the set of all aggregates $\mathcal{A}$ it has built.

In a first step outlined in Algorithm 3.3 we add new nodes to our aggregate until we reach the minimal prescribed aggregate size $s_{\min}$. The new vertex has the most strong connections to vertices of the aggregate. Here we prefer if both edge (i,j) and edge (j,i) are strong between two vertices i and j. (This allows us to use even the unsymmetric Ruge and Stüben strength of connection measure). In the algorithm $\text{cons}_1(v,\mathcal{A})$ and $\text{cons}_2(v,\mathcal{A})$ return the number of one-way and two-way connections between the vertex v and all vertices of the aggregate $\mathcal{A}$, respectively. If there are more candidates, we choose the vertex that adds the least new connections in the coarse matrix. As a measure for this we use the function connect$(v,\mathcal{A})$. It counts neighbours of v that are not yet aggregated or belong to an aggregate that is not yet connected to aggregate $\mathcal{A}$ once. Neighbours of v that belong to aggregates that are already connected to aggregate $\mathcal{A}$ are counted twice. The new vertex will have the most connections to other not yet aggregated vertices which are are already neighbours of the aggregate. The function neighbours$(v,\mathcal{A})$ counts the number of neighbours of vertex v that are also not yet aggregated neighbours of the

Algorithm 3.2 Build Aggregates

procedure AGGREGATION(V, E, $s_{\min}$, $s_{\max}$, $d_{\max}$)
  $U \leftarrow V \setminus \text{iso}(V)$ ▷ First Candidates are non-isolated vertices
  $I \leftarrow \emptyset$ ▷ Coarse index set
  Select arbitrary seed $v_i \in \{u \in U \mid \#N_a(u) \leq \#N_a(w) \forall w \in V\}$
  while $U \neq \emptyset$ **do**
    $\mathcal{A}_i \leftarrow \{v_i\}$
    $U \leftarrow U \setminus A_i$
    $I \leftarrow I \cup \{i\}$
    GROWAGGREGATE($\mathcal{A}$, V, E, $s_{\min}$, $d_{\max}$, U)
    ROUNDAGGREGATE($\mathcal{A}$, V, E, $s_{\max}$, U)
    if $\#\mathcal{A}_i = 1$ **then** ▷ Merge one vertex aggregate with neighbour
      $C \leftarrow \{\mathcal{A}_j, j \in I \setminus \{i\} \mid \exists w \in \mathcal{A}_j \text{ with } w \in N_a(v_i)\}$
      if $C \neq \emptyset$ **then**
        Choose $\mathcal{A}_k \in C$
        $I \leftarrow I \setminus \{i\}$
        $\mathcal{A}_k \leftarrow \mathcal{A}_k \cup \mathcal{A}_i$
      end if
    end if
    if $U \neq \emptyset$ **then**
      Select arbitrary seed $v_i \in U \cap \{w \mid N(w) \cap \mathcal{A}_i \neq \emptyset\}$
    end if
  end while
  $U \leftarrow \text{iso}(V)$ ▷ Aggregate isolated vertices
  while $U \neq \emptyset$ **do**
    Select arbitrary seed $v_i \in U$
    $\mathcal{A}_i \leftarrow \{v_i\}$
    $U \leftarrow U \setminus A_i$
    $I \leftarrow I \cup \{i\}$
    GROWISOAGGREGATE($\mathcal{A}$, V, E, $s_{\min}$, $d_{\max}$, U)
  end while
  $\mathcal{A} \leftarrow \{\mathcal{A}_i \mid i \in I\}$
  return $(\mathcal{A}, I)$
end procedure

aggregate $\mathcal{A}$. This criterion tries to maximise the number of candidates for choosing the next candidate. Of course the aggregate is not allowed to have a bigger diameter than the prescribed value $d_{\max}$ when the new vertex is added.

Algorithm 3.3 Grow Aggregate Step

function GROWAGGREGATE($\mathcal{A}$, V, E, $s_{\min}$, $d_{\max}$, U)
  while $\#\mathcal{A} \leq s_{\min}$ **do** ▷ Makes aggregate $\mathcal{A}$ bigger until its size is $s_{\min}$
    $C_0 \leftarrow \{v \in N(\mathcal{A}) \mid \operatorname{diam}(\mathcal{A}, v) \leq d_{\max}\}$ ▷ Limit the diameter of the aggregate
    $C_1 \leftarrow \{v \in C_0 \mid \operatorname{cons}_2(v, \mathcal{A}) \geq \operatorname{cons}_2(w, \mathcal{A}) \forall w \in N(\mathcal{A})$
    if $C_1 = \emptyset$ **then** ▷ No candidate with two-way connections
      $C_1 \leftarrow \{v \in C_0 \mid \operatorname{cons}_1(v, \mathcal{A}) \geq \operatorname{cons}_1(w, \mathcal{A}) \forall w \in N(\mathcal{A})$
    end if
    if $\#C_1 > 1$ **then** ▷ More than one candidate
      $C_1 \leftarrow \{v \in C_1 \mid \frac{\operatorname{connect}(v,\mathcal{A})}{N(v)} \geq \frac{\operatorname{connect}(w,\mathcal{A})}{N(w)} \forall w \in C_1\}$
    end if
    if $\#C_1 > 1$ **then** ▷ More than one candidate
      $C_1 \leftarrow \{v \in C_1 \mid \operatorname{neighbours}(v, \mathcal{A}) \geq \operatorname{neighbours}(w, \mathcal{A}) \forall w \in C_1\}$
    end if
    if $C_1 = \emptyset$ **then break**
    end if
    Select one candidate $c \in C_1$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{c\}$ ▷ Add candidate to aggregate
    $U \leftarrow U \setminus \{c\}$
  end while
end function

A second step tries to make the aggregates "rounder". It is sketched in Algorithm 3.4. Until we reach the maximum allowed size $s_{\max}$ of our aggregate we add all non-aggregated neighbour vertices that have more connections to the aggregate than to other non-aggregated vertices.

If after these two steps the aggregate still consists of only one vertex, we try to find another aggregate that the seed node is strongly connected to. If such an aggregate exists, the aggregate with the seed is removed and it's associated index is also removed from index set I. Then the seed vertex is added to the found aggregate.

After all non-isolated vertices are aggregated, we try to build aggregates for the isolated vertices. We build these aggregates from neighbouring isolated vertices that have at least one common aggregate consist-

Algorithm 3.4 Round Aggregate Step

function RoundAggregate($\mathcal{A}$, V, E, $s_{\max}$, U)▷ Makes aggregate $\mathcal{A}$ until its size is $s_{\max}$
  while $\#\mathcal{A} \leq s_{\max}$ **do**
    $C \leftarrow \{v \in N_a \mid \text{cons}(v, \mathcal{A}) > \text{cons}(v, U)\}$
    Select arbitrary candidate $c \in C$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{c\}$ ▷ Add candidate to aggregate
    $U \leftarrow U \setminus \{c\}$
  end while
end function

ing of non-isolated vertices as their neighbour. This is done in function `growIsoAggregate` whose algorithm is not presented here.

This algorithm is inspired by the aggregation algorithm presented by Raw [1985]. In a first step Raw adds all vertices with direct strong connection to the aggregate as long as the size is below the maximum size. Then he adds all neighbour vertices with two or more connections to the aggregate to make it round. If the size of the aggregate is still below the prescribed minimum rate he repeats the two steps. In contrast our algorithm only adds one neighbouring vertex at a time to the current aggregate, the one with the best properties of all current neighbours. While this is certainly more work and we need to limit the aggregate's diameter, it allows us to better keep track of and optimise the aggregate's properties. Additionally, in contrast to our approach Raw's algorithm should have problems creating appropriate aggregates for 27-point stencils like they occur for Q_1 finite elements in 3D.

3.2.3 Properties of the method

Interface Preservation for Highly Discontinuous Coefficients

Some of the model problems that we will present in Section 3.4 will have a highly discontinuous permeability tensor K of the model problem equation

$$\nabla \cdot (\mathsf{K} \nabla u) = f \,.$$

We will always assume that the manifold where the permeability jump occurs is aligned with the boundary of the grid elements.

It is well-known that multigrid methods can cope with this situation for example by preserving the interface in the coarsening. In contrast, the

convergence rates become unsatisfactory with standard geometric multigrid coarsening. This was demonstrated for two dimensional problems and geometric multigrid methods in Wang [2000].

The question is what our coarsening algorithm does at these interfaces. For the finite element method with linear simplicial elements let us consider the two dimensional case on a uniform Cartesian grid. Assume that the permeability jumps from an arbitrary small value $\mathsf{K} = \left(\begin{smallmatrix} \epsilon & 0 \\ 0 & \epsilon \end{smallmatrix}\right)$, $\epsilon > 0$ at the left hand side to the matrix $\mathbb{1}$ on the right hand side. Then the stencil looks like

$$\begin{bmatrix} & -\epsilon & \\ -\epsilon & 4\epsilon & -\epsilon \\ & -\epsilon & \end{bmatrix}, \begin{bmatrix} & -\frac{\epsilon+1}{2} & \\ -\epsilon & 2\epsilon+2 & -1 \\ & -\frac{\epsilon+1}{2} & \end{bmatrix}, \text{ and } \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}$$

for vertices in the interior of the region with the low permeability ϵ, at the interface vertex, and in the interior of the region with the high permeability, respectively.

Naturally, we have strong connections to every neighbour in the interior of both regions. For the vertices at the interface the value of the edge weight function is $\frac{1}{8+8\epsilon}$ for edges from the high permeability region, $\frac{1}{16}$ for edges along the interface, and $\frac{\epsilon}{8+8\epsilon}$ for edges from the low permeability region. See Figure 3.2 and Figure 3.3 for an illustration of the edge measure functions for the fine and first coarse level. Therefore, for ϵ sufficiently large the connections from the low permeability regions will be regarded as weak as the threshold is $\frac{\alpha}{8+8\epsilon}$. The connection from the interface vertices is weak to the low permeability region and strong to the high permeability region. Therefore, aggregation of interface vertices only occurs along the interface or with vertices of the high permeability region. The same holds for the coarser levels.

If we would have used the classical Ruge and Stüben measure, the edge (j, i) is strong if $-a_{ij} \geq \alpha \max_{k \neq i} |a_{ik}|$ holds. Then the connection from the interface to the low permeability region would have been strong and the other way around it would have been weak. In this case our algorithm has a different result depending on where the aggregation starts. Clearly, this measure is not natural for aggregation.

Let the cell-centred finite volume discretizations be such that the interface of a permeability jump always coincides with the boundaries of a set of finite volumes. Then the scalar permeability at the interface is given as the harmonic average of the permeabilities of the two boxes sharing the interface as an edge. Let d_ϵ and d_1 be the diameter of the box in the low and the box in the high permeability region, respectively. Let K_ϵ and K_1 be

the permeability of the box in the low and the box in the high permeability region, respectively. Then the interface permeability is given by

$$\mathsf{K}_\Gamma = \frac{\mathsf{K}_\epsilon \mathsf{K}_1 (d_\epsilon + d_1)}{d_1 \mathsf{K}_\epsilon + d_\epsilon \mathsf{K}_1}.$$

Assuming a uniform cube grid the stencils associated with the box next to the interface in the low permeability region and the box next to the interface in the high permeability region are

$$\begin{bmatrix} & -\epsilon & \\ -\epsilon & \frac{3\epsilon^2+5\epsilon}{1+\epsilon} & -\frac{2\epsilon}{1+\epsilon} \\ & -\epsilon & \end{bmatrix}, \text{ and } \begin{bmatrix} & -1 & \\ -\frac{2\epsilon}{1+\epsilon} & \frac{3+5\epsilon}{1+\epsilon} & -1 \\ & -1 & \end{bmatrix}, \text{ respectively.}$$

The resulting edge weights in stencil notation are

$$\begin{bmatrix} & \frac{(1+\epsilon)^2}{(3\epsilon+5)^2} & \\ \frac{1+\epsilon}{12+20\epsilon} & & \frac{4\epsilon}{(3+5\epsilon)(5+3\epsilon)} \\ & \frac{(1+\epsilon)^2}{(3\epsilon+5)^2} & \end{bmatrix}, \text{ and } \begin{bmatrix} & \frac{(1+\epsilon)^2}{(3+5\epsilon)^2} & \\ \frac{4\epsilon}{(3+5\epsilon)(5+3\epsilon)} & & \frac{1+\epsilon}{20+12\epsilon} \\ & \frac{(1+\epsilon)^2}{(3+5\epsilon)^2} & \end{bmatrix}$$

for the box at the interface in the low and high permeability regions. The behaviour according to the value of ϵ is plotted in Figure 3.5 and in Figure 3.4, respectively. The measure for the connection across the interface is clearly much smaller than the rest for small ϵ. Therefore, aggregation across an interface with a high contrast coefficient jump will not happen.

Cell-Centred Finite Volume Preservation

Under admittedly rather restrictive and unrealistic assumptions, we want to show that for a cell-centred finite volume discretization our construction of the coarse matrix is again a finite volume discretization.

Theorem 3.10. *Let $\mathcal{T}$ be a two or three dimensional Cartesian grid with uniform cell width h and let $\mathcal{A}_h$ and $\mathcal{L}_h$ be the bilinear and linear form of the cell-centred finite volume discretization of the model problem with homogeneous Dirichlet boundary, $g_D = 0$, as given in* (2.8) *and* (2.9)*. Furthermore, let the aggregates $\mathcal{A}_i$, $i = 1, \ldots, N$, be constructed such that the union of all cells associated with each aggregate form a square in the two or cube in the three dimensional case with width H.*

Then the constructed coarse matrix $\frac{h}{H}\widetilde{\mathsf{R}}\mathsf{A}\widetilde{\mathsf{R}}^T$ represents a matrix from a cell-centred finite volume discretization using a Cartesian grid $\mathcal{T}_H$ with uniform cell width H.

Proof. Due to the specific nature of the aggregates, each aggregate is a cell of a grid $\mathcal{T}_H$.

Let $\widetilde{\mathsf{R}}^T$ be the prolongation operator constructed according to the aggregates. Recall, that for finite volume discretizations we approximate the functions with functions that are piecewise constant on the boxes. The prolongation of such a function is piecewise constant on the aggregates of the fine grid. Therefore, such a prolongated function can only have jumps on the edges that coincide with the aggregate boundaries.

Due to the construction using the Galerkin product, the coarse level bilinear form is given by

$$\begin{aligned}\mathcal{A}_H(u_H, v_H) &= \frac{h}{H}\mathcal{A}_h(\widetilde{\mathsf{R}}^T u_H, \widetilde{\mathsf{R}}^T v_H) \\ &= -\sum_{e\in\mathcal{E}_h^I}\left(\widetilde{\mathsf{K}}\frac{\widetilde{\mathsf{R}}^T u_H(r(e)) - \widetilde{\mathsf{R}}^T u_H(l(e))}{\frac{H}{h}d(r(e)), l(e))}, [\![\widetilde{\mathsf{R}}^T v]\!]\right)_e .\end{aligned} \tag{3.22}$$

Since prolongated functions can only change their value at the edges that coincide with the aggregate boundaries, it suffices to sum over these edges. The nominator of the quotient is just the change of a function that is constant on the aggregates. For this kind of function it does not matter where in an aggregate we take its value. The denominator is just the distance of the centres of the coarse level cells as we deal with a uniform cell width. Additionally, we have shown above that the aggregation will not cross the interfaces where we have a jump in the permeability tensor K. Therefore, the permeability is constant in each aggregate on the fine level and the harmonic measure for the edges of the aggregate is the same whether taken for the neighbouring fine level cells or the neighbouring coarse level cells. Thus, equation (3.22) is just the bilinear form of the finite volume discretization on the coarse grid.

For the linear form it follows with the same reasoning that

$$\mathcal{L}_H(v_H) = \mathcal{L}_h(\mathsf{R}^T v_H) = \sum_{\tau\in\mathcal{T}_h}(f, v)_\tau + \sum_{e\in\mathcal{E}^N}(g_N, v)_e = \sum_{\tau\in\mathcal{T}_H}(f, v)_\tau + \sum_{e\in\mathcal{E}_H^N}(g_N, v)_e$$

is equivalent to the linear form obtained by the cell-centred finite volume scheme on $\mathcal{T}_H$. □

Please note, that the assumptions made in Theorem 3.10 are rather artificial as our heuristical aggregation algorithm will usually not create such regular aggregates. Therefore, how to choose a uniform relaxation parameter ω for the Galerkin product (3.20) to obtain the coarse matrix

is not clear. In general we settled for the value $\omega = 0.8d$ where d is the width of the aggregates to be achieved.

Semi-Coarsening for Anisotropic Problems

It is well known that the traditional strength of connection measures are able to detect anisotropic behaviour of the permeability tensor and force semi-coarsening for linear systems stemming from finite difference and finite volume methods. The same holds true for our new measure.

Unfortunately, the stencils for the anisotropic problem discretized with Q_1 finite elements have relatively large positive off-diagonal values. Therefore the traditional Ruge Stüben measure, Ruge and Stüben [1987], and the traditional measure used for aggregation AMG, Vaněk et al. [1996b], might falsely treat connections along the low permeability direction as strong.

The 2D stencil of the anisotropic problem with low permeability in the x-direction discretized with Q_1 finite elements is

$$\frac{1}{3}\begin{bmatrix} -\frac{\epsilon+1}{2} & \epsilon-2 & -\frac{\epsilon+1}{2} \\ 1-2\epsilon & 4\epsilon+4 & 1-2\epsilon \\ -\frac{\epsilon+1}{2} & \epsilon-2 & -\frac{\epsilon+1}{2} \end{bmatrix}.$$

This is not a problem for the isotropic problem with $\epsilon = 1$, where the diagonal value is $\frac{8}{3}$ and the off-diagonal values are all $-\frac{1}{3}$. But for the limit case of the anisotropic problem with $\epsilon = 0$ the stencil reads

$$\frac{1}{3}\begin{bmatrix} -\frac{1}{2} & -2 & -\frac{1}{2} \\ 1 & 4 & 1 \\ -\frac{1}{2} & -2 & -\frac{1}{2} \end{bmatrix}.$$

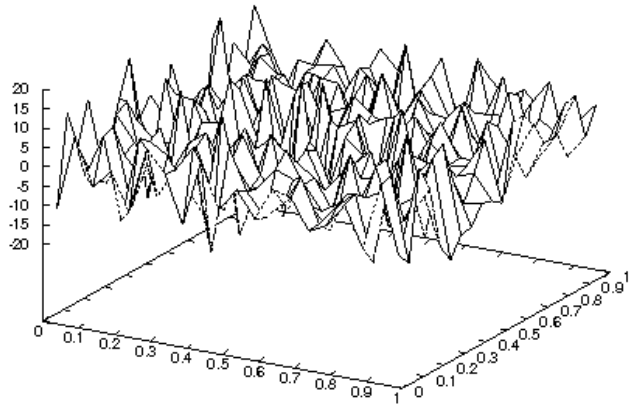
Clearly, depending on how our threshold a for identifying strong connections is chosen, the positive connections in x-direction might falsely be regarded as strong. Actually, this never occurred to us because the default threshold $a = \frac{1}{3}$ still ignores them for two dimensional problems.

The picture is quite different for the problems in three dimensions. In this case the stencil for the anisotropic problem is illustrated in Figure 3.6. Again the low permeability ϵ is along the x-direction and the permeability is 1 along the other coordinate directions. The two limit cases of the stencil, that is $\epsilon = 0$ and $\epsilon = 1$, can be seen in Figure 3.7 and Figure 3.8, respectively.

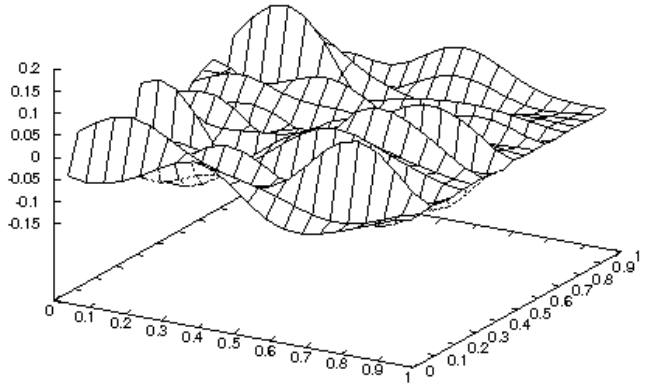
When we did not treat positive off-diagonals as weak connections, only those connection satisfying

$$\frac{a_{ij}^2 9^2}{16^2} \geq a\frac{1}{16}$$

were considered strong for the limit case $\epsilon = 0$. Except for very small values a these are just the positive off-diagonals. Therefore, the method coarsened along weak connections. Our new criterion treats positive off-diagonal values as weak connections. Thus it prevents coarsening in the wrong direction and results in reasonable convergence rates even for the above cases.



(a) Initial Defect



(b) Defect after 10 SSOR steps

Figure 3.1: Algebraic Smoothing via SOR

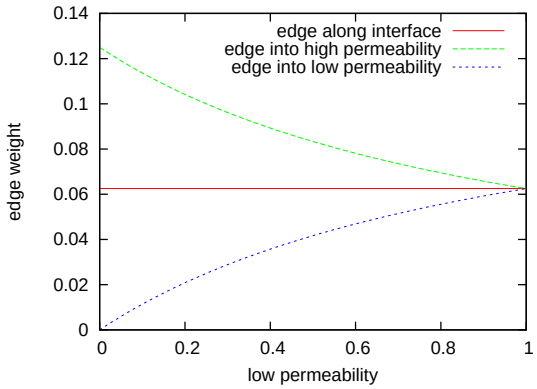


Figure 3.2: Edge Measure at Coefficient Jump Interface (P_1): Fine Level

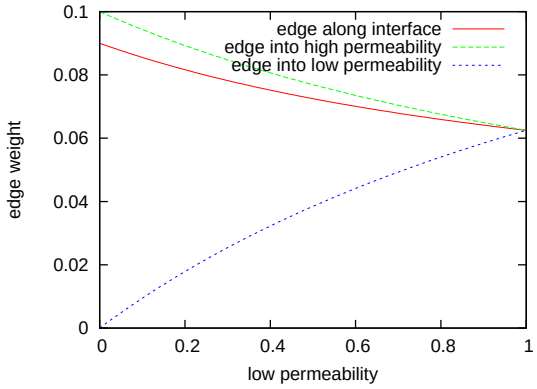


Figure 3.3: Edge Measure at Coefficient Jump Interface (P_1): First Coarse Level

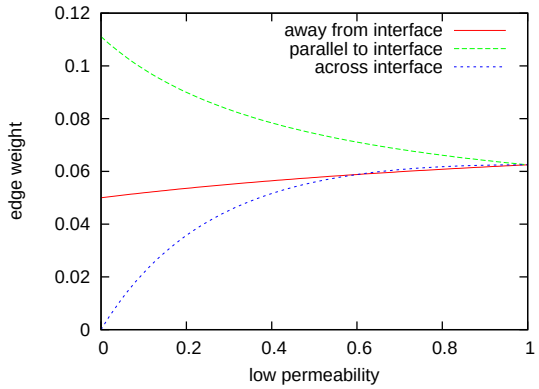


Figure 3.4: Edge Measures in High-Permeability Region at Coefficient Jump Interface (FV)

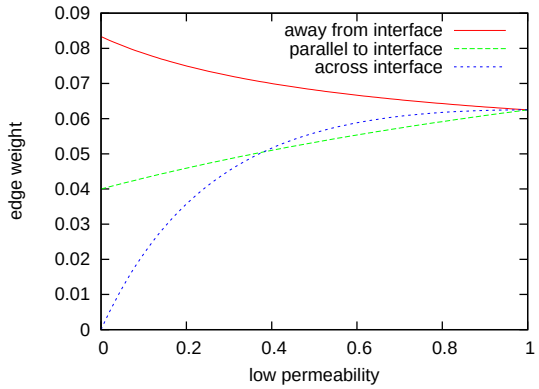


Figure 3.5: Edge Measures in Low-Permeability Region at Coefficient Jump Interface (FV)

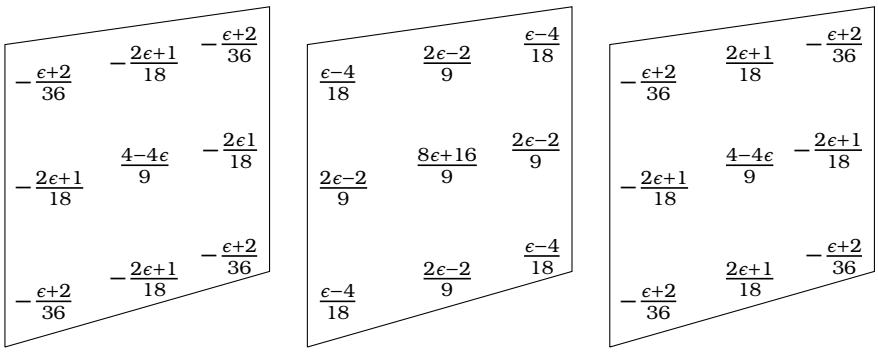


Figure 3.6: $Q1$-Stencil of Anisotropic Model Problem

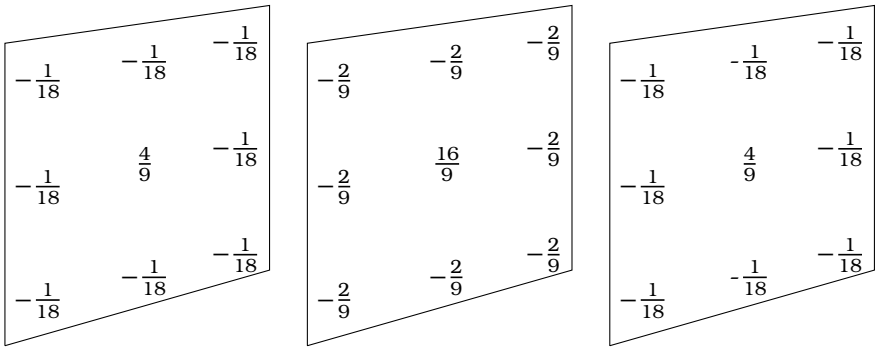


Figure 3.7: $Q1$-Stencil of Anisotropic Model Problem $\epsilon = 0$

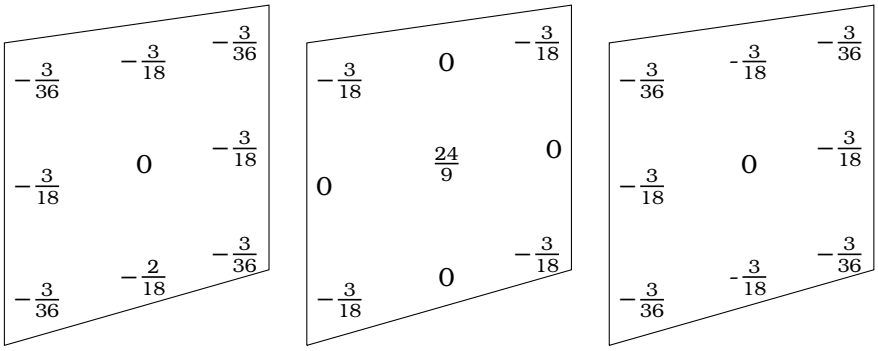


Figure 3.8: $Q1$-Stencil of Isotropic Model Problem

3.3 Extension to Systems of PDE

Applying the scalar version of AMG unmodified to linear systems arising from systems of partial differential equations is rather ineffective. For example the AMG algorithm could find connectivity between different physical properties where there is none. When it comes to saddle point problems like the Oseen or Stokes problem most of the standard discretizations lead to indefinite system matrices. The scalar algorithms cannot handle this situation.

To overcome these problems different methods were proposed already in Ruge and Stüben [1987]. Later the proposed approaches were documented methodically and investigated further in Clees [2005]. We follow the methodology in the latter publication. There three different approaches are discussed. One approach just treats the resulting linear system as a scalar one and applies AMG to it. It is called *variable-based AMG*. This might result in the problems mentioned above. The other two approaches make use of the underlying geometry and/or physical meaning of the variables.

3.3.1 Unknown-based AMG

The *unknown-based AMG* uses a matrix with unknown-based blocking. That is, all degrees of freedom belonging to one unknown, e.g. pressure, are grouped together. Here the unknown, usually a physical quantity, is a function that is to be approximated. As an example let us look at the stationary Stokes problem. The unknowns here are the two velocity components u (in x-direction) and v (in y-direction) and the pressure p. Grouping these together the resulting linear system looks like

$$\begin{pmatrix} \mathsf{A}_{uu} & \mathsf{A}_{uv} & \mathsf{B}_{1,up}^T \\ \mathsf{A}_{vu} & \mathsf{A}_{vv} & \mathsf{B}_{1,vp}^T \\ \mathsf{B}_{2,up} & \mathsf{B}_{2,vp} & \mathsf{C}_{pp} \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \\ \mathbf{p} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_u \\ \mathbf{f}_v \\ \mathbf{f}_p \end{pmatrix}.$$

The idea is now to apply the variable-based (scalar) AMG approach to each diagonal block of the matrix to create the prolongation operators $P_1, \ldots, P_n$ for each unknown. Then the global prolongation operator is constructed from these operators as

$$\mathsf{P} = \begin{pmatrix} \mathsf{P}_1 & 0 & \ldots & 0 \\ 0 & \mathsf{P}_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \mathsf{P}_n \end{pmatrix}.$$

Using the usual Galerkin product $A^{l+1} = P^T A^l P$, we build the linear operator hierarchy.

As smoothers we use non-overlapping additive Schwarz methods with no coarse grid correction. The local problems correspond to the degrees of freedom associated with one unknown. And we use inexact local solvers consisting of only a few steps of traditional stationary iterative methods.

The convergence of this approach for the classical Ruge and Stüben approach was investigated in Clees [2005]. It was observed that this approach works well if the smoother is able to produce an algebraically smooth error for each unknown separately. This is the case for a multiplicative smoother as long as the cross-couplings between different unknowns are weak.

3.3.2 Point-based AMG

Whenever the unknowns are strongly coupled it is advisable to use the *point-based AMG* approach. Here all degrees of freedom associated with a grid entity of a given co-dimension (element, face, or point) but of different physical quantities are grouped together. This approach is naturally limited to the case where all unknowns are discretized using the same grid and the number of unknowns associated with each point is constant. The resulting linear system is a sparse matrix, which has small dense equally sized matrices as it's entries. That is

$$A\mathbf{x} = \begin{pmatrix} A_{11} & \dots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{m1} & \dots & A_{mm} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} = \mathbf{b}$$

with $A_{ij} \in \mathbb{R}^{k\times k}$, $x_i, b_i \in \mathbb{R}^k$, $i,j = 1,\dots,m$.

Aggregation Approach

Setting $\mathbb{K} := \mathbb{R}^k$ we can use our approach from the previous subsections for the coarsening process. What is left to do, is to define and use appropriate weight functions for our weighted matrix graph (Definition 3.4).

As already pointed out the discussion about algebraic smoothness carries over to the point-based approach. Note, that in inequality 3.11 we need to invert a matrix. Using our weight function approach we can avoid this costly operation. Various weight functions are used in practice.

The approach of Raw published in Raw [1985] was one of the first approaches of this type. There the point-based approach was used to solve

the Navier–Stokes equation using a cell-centred finite volume discretization on non-staggered grids. For this approach both weight functions simply return the pressure-pressure coupling of the matrix block. Recently, in Papadopoulos and Tchelepi [2003] the same approach proved superior over the usage of simple matrix norms in terms of both robustness and convergence, even for smoothed aggregation AMG. Note, that this also allows the detection of positive off-diagonal values.

The weight functions might also be simple matrix norms. For example in Griebel et al. [2003] the Frobenius norm was used for solving linear elasticity problems. As an alternative, the row-sum matrix norm is advocated in Clees [2005].

Of course, it is also possible to choose the weight functions independent of the original problem and use weights based on the coordinates of the points associated to the blocks or an auxiliary scalar problem to compute them. Both approaches were also investigated in Clees [2005].

In all cases the aggregation algorithm works on a graph that just stores the strength of connection and can therefore be used without modifications as in the scalar case. As a consequence, the aggregation algorithm is independent of the block size. Only the computation of the Galerkin product has a higher complexity due to the block size.

Smoothers

As smoothers we use multiplicative and additive one-level Schwarz methods with exact subdomain solvers. Each of the local subspaces is spanned by the basis functions associated with one point. The subspaces are non-overlapping and disjoint. In addition several incomplete block LU factorisation are available as smoothers.

3.4 Numerical Results

Let us take a look at the question how our AMG method works for some important model problems. As the method itself is not an optimal solver we will use it as preconditioner to the conjugate gradient method.

Recall that the model problem problem is

$$\begin{aligned} -\nabla \cdot (K \nabla u) &= f \text{ in } \Omega \\ u &= g_D \text{ on } \Gamma_D \\ (K \nabla u) \cdot \mathbf{n} &= g_N \text{ on } \Gamma_N \,. \end{aligned}$$

on the domain $\Omega \subset \mathbb{R}^d$, $d = 2, 3$, with $\Gamma_N \cup \Gamma_D = \partial\Omega$, $\Gamma_N \cap \Gamma_D = \emptyset$. We will restrict us to following representative models and show how the resulting aggregates and the robustness of the methods look like.

Example 3.11. The first model problem is a Poisson equation with pure Dirichlet boundary conditions. The problem is given by

$$\begin{aligned} -\Delta u &= (2d - 4\|x\|^2)e^{-\|x\|^2} && \text{in } \Omega = (0,1)^d \\ u &= e^{-\|x\|^2} && \text{on } \partial\Omega\,. \end{aligned}$$

Our greedy coarsening strategy can be adjusted to achieve arbitrarily sized aggregates. As stated before, the prolongation as well as the restriction are piecewise constant which prevent our method form being optimal. As a first test we take a look at how the aggregate size influences the convergence of our method and the time needed to achieve it. In Table 3.1 we present the results for our method when applied to the Poisson problems discretized with Q_1 finite elements on a structured grid with 1024×1024 and $64 \times 64 \times 64$ elements in two and three dimensions, respectively.

size	d_a	C_A	lev.	It	TB	TS	TT
2–3	1	1.79	11	16	6.87	11.95	18.82
4–6	2	1.33	6	19	5.95	9.98	15.93
9–12	4	1.10	5	35	5.86	15.78	21.64
16–18	6	1.05	4	40	6.43	17.23	23.66
25–30	8	1.03	4	53	7.36	21.37	28.73

(a) 2D: 1024×1024 elements

size	d_a	C_A	lev.	It	TB	TS	TT
2–3	1	1.98	9	14	9.20	6.77	15.97
4–6	2	1.33	5	9	6.56	2.89	9.45
8–10	3	1.14	4	12	6.45	3.36	9.81
27–30	6	1.04	3	16	10.36	4.03	14.39
64–70	9	1.01	3	20	20.62	4.94	25.56

(b) 3D: $64 \times 64 \times 64$ elements

Table 3.1: Convergence Dependency on Aggregate Size

Here, the computations were done on an Intel Core 2 Duo CPU P9500 with 2.53GHz, the software was compiled using the GNU C++ compiler version 4.3 with -O3 optimisation flags. The V-cycle of our AMG with one step of SSOR for pre- and post-smoothing was used as a preconditioner to the conjugate gradient solver. Generally, the iterative solver stops if the

initial defect has been reduced by a factor of 10^{-8} in the l^2-norm. If not noted otherwise this applies to the other computations in this thesis, too.

In the tables the first column (labelled size) represents the prescribed minimum and maximum aggregate size. In the second column, d_a represents the allowed maximal diameter an aggregate might have. Furthermore, we present with C_A the *operator complexity*. That is the sum of the number of nonzeros of the matrices at all levels divided by the number of nonzeros of the matrix at the finest level. It is measure for the memory consumption of the method. Other acronyms used in the tables of this section are:

lev. The number of levels in the operator hierarchy.

h The width of the elements of the grid.

It The number of iterations needed to achieve a relative defect reduction of 10^{-8}.

TB Time needed to build the AMG hierachies. This is often called setup time.

TS Time needed for iteratively solving the linear systems. (TB is excluded here.)

TIt Time needed for one iteration of the solver.

TT The total time needed to solve the linear system including the setup time. (TT=TB+TS)

Using a big prescribed aggregate size, we are able to have a very low operator complexity. At the same time we sacrifice some of the convergence. Clearly, we can achieve operator complexities comparable with geometric multigrid for a prescribed aggregate size of four and eight unknowns for two and three dimensional problems, respectively. At the same time these aggregate sizes lead to the shortest total time to solution.

In Table 3.2 we present the time used for building the hierarchies for varying stencil sizes. In the first column we tell the discretization method that was used, in the second column we give the resulting degrees of freedom, and the third column contains the stencil size. This is the actual size used in the sparsity pattern of the matrix. Due to the discretization algorithm used this might be bigger than expected, as the implementation sometimes chooses to store some of the nonzeros, too. The last column presents the time needed for building the hierarchy and in brackets the

time normalised to the number rows of the finite volume discretization. In Figure 3.9 we plot the normalised time needed to build the AMG hierarchy against the stencil size and fit it using least squares to the function $f(x) = ax + bx^2$. The fitted values for the two dimensional problems are $a = 0.0921636$, and $b = 0.0040941$ and for the three dimensional problem they are $a = 0.0946918$ and $b = 0.00365989$. Although the complexity turns out to be quadratic, the constant b is low compared to a. Therefore, for small stencil sizes, like the ones observed here, the complexity is still approximately linear.

FE	DOFS	nnz/row	C_A	TB
CC-FV 2D	262144	5	1.25	0.5
P_1 2D	263169	7	1.22	0.94 (0.94)
Q_1 2D	263169	9	1.25	1.13 (1.13)
CC-FV 3D	262144	7	1.35	1.19
P_1 3D	230945	19	1.14	2.47 (2.80)
Q_1 3D	274626	27	1.14	5.62 (5.36)

Table 3.2: Hierarchy Build Time vs. Stencil Size

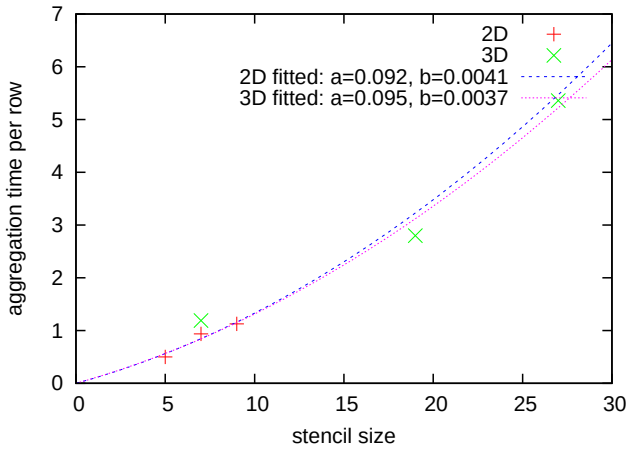


Figure 3.9: Hierarchy Build Time as a Function of Stencil Size

As a reference for the problems with jumping coefficients we present the results of our method applied to the Poisson problem in Tables 3.3 and 3.4. As expected the method is not optimal, but the number of iterations needed for convergence increases linearly with the number of levels in

the hierarchy. Both, the time needed for building the hierarchy and the time needed for one iteration step, increases linearly with the number of unknowns.

1/h	lev.	TB	TS	It	TIt
64	2	0.16	0.028	8	0.0035
128	3	0.080	0.076	11	0.0070
256	4	0.33	0.33	13	0.025
512	5	1.41	1.96	17	0.12
1024	6	5.87	9.35	19	0.49

(a) Q_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
64	2	0.012	0.036	11	0.0033
128	3	0.068	0.096	14	0.0069
256	4	0.29	0.42	17	0.024
512	5	1.18	2.30	20	0.12
1024	6	4.84	16.32	24	0.48

(b) P_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
64	2	0.008	0.024	7	0.0034
128	3	0.040	0.052	9	0.0058
256	4	0.16	0.22	11	0.020
512	5	0.68	1.23	13	0.094
1024	6	2.68	6.23	15	0.42

(c) Cell-Centred Finite Volumes

Table 3.3: Poisson Problem 2D

Example 3.12. Let our diffusion problem be given by

$$\begin{aligned} -\nabla \cdot \{\mathsf{K}(x)\nabla u\} &= 1 && \text{in } \Omega = (0,1)^d, \\ u &= 0 && \text{on } \partial\Omega\,. \end{aligned}$$

The isotropic permeability tensor $\mathsf{K}(x) = k(x)\mathbb{1}$, $\mathbb{1}$ being the identity matrix, has jumps in a chequerboard manner. The chequerboard has 8 cells of width $H = 1/8$ in each dimension. Let the function $\lfloor\cdot\rfloor$ return the maximum integer value that is equal to or smaller than the argument.

1/h	lev.	TB	TS	It	TIt
16	2	0.07601	0.048	7	0.006858
32	3	0.776	0.292	9	0.03245
64	4	6.936	3.256	12	0.2714

(a) $Q1$ Finite Elements

1/h	lev.	TB	TS	It	TIt
16	2	0.036	0.032	9	0.003556
32	3	0.348	0.228	12	0.019
64	4	3.092	2.612	16	0.1633

(b) P_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
16	2	0.016	0.06	8	0.007501
32	3	0.232	0.26	10	0.026
64	4	2.036	1.624	12	0.1353

(c) Cell-Centred Finite Volumes

Table 3.4: Poisson Problem 3D

Then the permeability field is described by

$$k(x) = \begin{cases} 20.0 & \lfloor x_0/H \rfloor \text{ even, } \lfloor x_1/H \rfloor \text{ even, and } \lfloor x_2/H \rfloor \text{ even} \\ 0.002 & \lfloor x_0/H \rfloor \text{ odd, } \lfloor x_1/H \rfloor \text{ even, and } \lfloor x_2/H \rfloor \text{ even} \\ 0.2 & \lfloor x_0/H \rfloor \text{ even, } \lfloor x_1/H \rfloor \text{ odd, and } \lfloor x_2/H \rfloor \text{ even} \\ 2000.0 & \lfloor x_0/H \rfloor \text{ odd, } \lfloor x_1/H \rfloor \text{ odd , and } \lfloor x_2/H \rfloor \text{ even} \\ 1000.0 & \lfloor x_0/H \rfloor \text{ even, } \lfloor x_1/H \rfloor \text{ even, and } \lfloor x_2/H \rfloor \text{ odd} \\ 0.001 & \lfloor x_0/H \rfloor \text{ odd, } \lfloor x_1/H \rfloor \text{ even, and } \lfloor x_2/H \rfloor \text{ odd} \\ 0.1 & \lfloor x_0/H \rfloor \text{ even, } \lfloor x_1/H \rfloor \text{ odd, and } \lfloor x_2/H \rfloor \text{ odd} \\ 10.0 & \lfloor x_0/H \rfloor \text{ odd, } \lfloor x_1/H \rfloor \text{ odd, and } \lfloor x_2/H \rfloor \text{ odd} \end{cases},$$

in three dimensions and by

$$k(x) = \begin{cases} 20.0 & \lfloor x_0/H \rfloor \text{ even, and } \lfloor x_1/H \rfloor \text{ even} \\ 0.002 & \lfloor x_0/H \rfloor \text{ odd, and } \lfloor x_1/H \rfloor \text{ even} \\ 0.2 & \lfloor x_0/H \rfloor \text{ even, and } \lfloor x_1/H \rfloor \text{ odd} \\ 2000.0 & \lfloor x_0/H \rfloor \text{ odd, and } \lfloor x_1/H \rfloor \text{ odd} \end{cases},$$

in two dimensions.

The aggregates and permeabilities are illustrated in Figure 3.10. The colours denote the permeability and the solid black lines are the boarders of the aggregates on the coarsest level. For the discretization we used

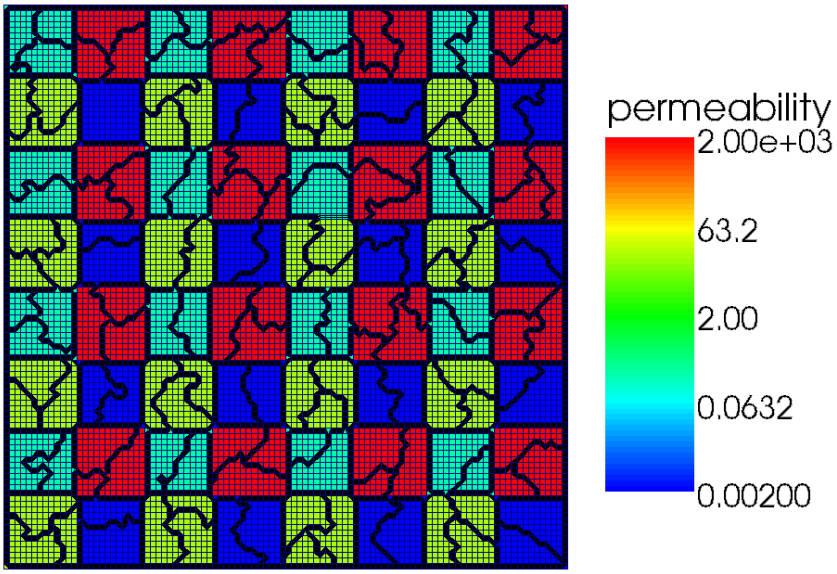


Figure 3.10: Aggregates for chequerboard permeability

a uniform cube grid with 100×100 cells and $Q1$ finite elements. Note, that the aggregation respects the permeability jumps. The permeability remains constant in the interior of each element of our grid. Our degrees of freedom are associated with the vertices of the grid. The vertices on the boundary between high and low permeability regions get added to the aggregates in the region with higher permeability. This is in accordance to our discussion in the previous section.

In Tables 3.5, and 3.6 we present the results for our solver applied to different problem sizes. We see that the the setup time needed is comparable to time needed for the Poisson problem. The number of iterations needed for convergence is again linear in the number of levels of the hierarchy. Only the constant in this dependency is bigger than for the Poisson problem.

Example 3.13. A diffusion problem with smoothly varying permeability field is given by

$$\begin{aligned} -\nabla \cdot \{k(x)\nabla u\} &= f && \text{in } \Omega = (0,1)^d, \\ u &= g && \text{on } \Gamma_D, \\ -\nabla u \cdot \nu &= 0 && \text{on } \Gamma_N, \end{aligned}$$

1/h	lev.	TB	TS	It	TIt
64	2	0.016	0.032	9	0.0036
128	3	0.076	0.088	13	0.0068
256	4	0.33	0.44	17	0.026
512	5	1.40	2.41	21	0.11
1024	6	5.77	13.66	28	0.49

(a) Q_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
64	2	0.016	0.036	11	0.0033
128	3	0.068	0.096	14	0.0068
256	4	0.30	0.52	20	0.026
512	5	1.22	2.92	25	0.12
1024	6	4.93	16.18	33	0.49

(b) P_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
64	2	0.008	0.0024	7	0.0034
128	3	0.036	0.068	10	0.0068
256	4	0.16	0.23	11	0.021
512	5	0.66	1.23	13	0.095
1024	6	2.74	6.12	15	0.41

(c) Cell-Centred Finite Volumes

Table 3.5: Chequerboard Permeability Field 2D

with

$$\Gamma_D = \{x \,|\, x_0 = 0 \vee x_0 = 1\}, \quad \Gamma_N = \partial\Omega \setminus \Gamma_D,$$

and

$$g(x) = \begin{cases} 1 & x_0 = 0 \\ 0 & x_0 = 1 \end{cases}.$$

The values of the scalar permeability field $k(x)$ are chosen as a log-normal random field. That is, $\log k(x)$ is a realisation of a homogeneous, isotropic Gaussian random field with exponential covariance function with mean 0, variance σ^2 and correlation length scale λ.

An example for the aggregates on the coarsest level and the distribution of the permeability values is given in Figure 3.11. We used a uniform cube grid with element diameter $h = 1/256$ and a Q_1 discretization scheme. The permeabilities were distributed with variance $\sigma^2 = 8$, and correlation length $\lambda = 0.16$. As the permeabilities change smoothly, the aggregates do

1/h	lev.	TB	TS	It	TIt
16	2	0.072	0.104	14	0.007
32	3	0.68	0.34	11	0.031
64	4	6.32	3.69	14	0.26

(a) Q_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
16	2	0.028	0.052	12	0.0043
32	3	0.30	0.31	17	0.018
64	5	2.76	3.00	19	0.16

(b) P_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
16	3	0.016	0.024	8	0.0030
32	4	0.18	0.11	9	0.012
64	5	1.75	1.28	11	0.12

(c) Cell-centred Finite Volumes

Table 3.6: Chequerboard Permeability Field 3D

not follow the small jumps exactly. Still, the tendency of the permeability field is fuzzily resembled by them.

The random field is given by a continuous function. For the discretization we pick the permeability value at the cell centre and assume it to be constant on the cell. Therefore, for higher resolution the ratio between the lowest and highest permeability value increases. Besides, by the problem size the condition of the linear system is influenced by this ratio.

In Tables 3.7, and 3.8 we present the results for our solver applied to this problem. We used variance $\sigma^2 = 4$ and correlation length $\lambda = 1/64$. We see the same complexity behaviour as for the previous problems.

Example 3.14. In this example the problem is described with the same equations as in Example 3.13. The only difference lies in the discrete values of the permeability field used. It is again generated using the log-normal distribution described above. Once the values are computed for each cell, we compute the mean permeability over the whole domain. Then we compute the mean value of all permeabilities below the mean and replace these permeabilities with this new mean value. Similarly, we compute the mean value of all permeabilities bigger than the global mean permeability and replace them with the calculated upper mean value. This results in a permeability field that can attain two discrete values. In contrast to the previous example, the permeabilities now make huge

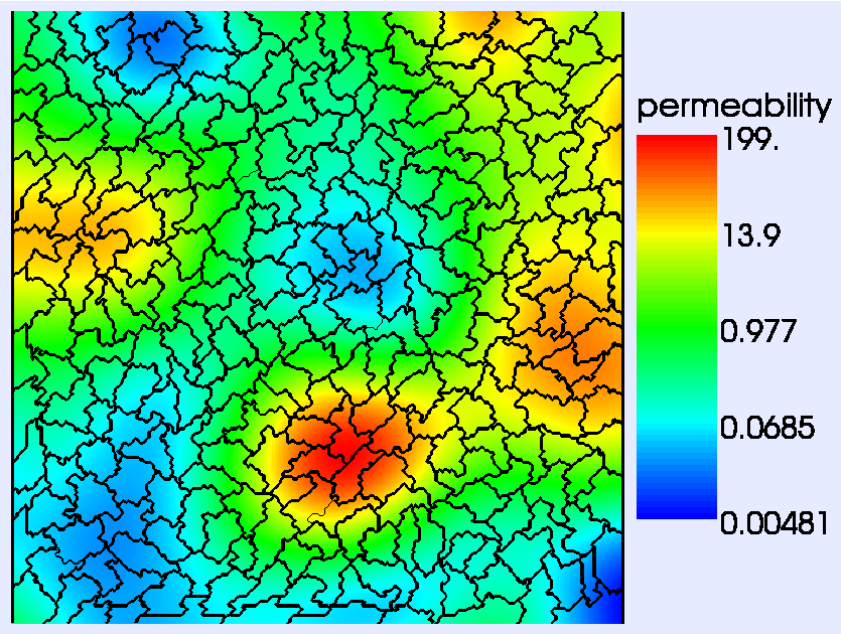


Figure 3.11: Aggregates for log-normally distributed permeability field ($\sigma^2 = 8$, $\lambda = 0.16$)

jumps instead of just changing gradually.

In Figure 3.12 the permeabilities are illustrated and the borders of the coarsest aggregates are represented by solid lines. Here we used a uniform 2D cube grid with cell width $h = 1/256$. Note, that the aggregation process honours the coefficient jumps and does not cross over the jump. This is in accordance to the discussion above.

Due to the averaging of the permeability values, the ratio between the highest and lowest permeability of a realisation is lower than for the non-clipped Example 3.13. Still, the jumps that occur rapidly throughout the domain are larger than previously. This makes this example very challenging for linear solvers.

In Tables 3.9, and 3.10 we present the results for our problem when applied to this example. In the computations we used variance $\sigma^2 = 8$ in all computations For the two dimensional problem the correlation length used is $\lambda = 1/64$ and for the three dimensional case $\lambda = 1/16$.

In Table 3.11 we present the statistics of our solver for a varying correlation length and variance. We see that the build time of our AMG is

1/h	lev.	TB	TS	It	TIt
64	2	0.02	0.028	8	0.0035
128	3	0.09601	0.08801	11	0.008001
256	4	0.4	0.376	14	0.02686
512	5	1.636	1.728	15	0.1152
1024	6	6.964	9.849	20	0.4924

(a) Q_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
64	2	0.016	0.036	12	0.003
128	3	0.08001	0.104	14	0.007429
256	4	0.348	0.472	17	0.02777
512	5	1.432	2.416	21	0.1151
1024	6	6.012	11.9	24	0.496

(b) P_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
64	2	0.008001	0.024	7	0.003429
128	3	0.048	0.056	9	0.006223
256	4	0.22	0.292	12	0.02433
512	5	0.8001	1.3	14	0.09286
1024	6	3.344	6.196	15	0.4131

(c) Cell-Centred Finite Volumes

Table 3.7: Random Permeability Field 2D ($\sigma^2 = 8$, $\lambda = 1/16$)

robust against changes of the variance and correlation length. For decreasing correlation length the average size of the areas with one diffusion coefficient become smaller. The number of iterations needed for convergence increases gradually. The convergence of our method decreases more steeply for correlation lengths smaller than 8 times the cell width. Still convergence is achieved even for the nearly uncorrelated case ($\lambda = 1/512$). In comparison to the algebraic domain decomposition solver in Scheichl and Vainikko [2007] the number of iterations increases more steeply. For fixed correlation length $\lambda = 64$ and increasing σ the number of iterations needed for convergence increase only slightly. For this test the average size of an area with fixed diffusion coefficient stays the same, but the size of the jumps increases for increasing variance. Therefore, the linear systems get harder. Although not being totally robust against variance changes, our preconditioner can handle high variances very well.

Finally, we repeat some of the scalability tests with a fixed variance

1/h	lev.	TB	TS	It	TIt
16	2	0.088	0.064	7	0.009143
32	3	0.8521	0.312	9	0.03467
64	4	7.172	3.28	12	0.2734

(a) Q_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
16	2	0.04	0.044	10	0.0044
32	3	0.38	0.256	13	0.01969
64	4	3.212	2.82	17	0.1659

(b) P_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
16	2	0.02	0.064	9	0.007112
32	3	0.224	0.272	10	0.0272
64	4	2.008	1.708	12	0.1423

(c) Cell-Centred Finite Volumes

Table 3.8: Random Permeability Field 3D ($\sigma^2 = 8$, $\lambda = 1/4$)

of $\sigma^2 = 8$ and the correlation length $\lambda = 4h$ linked to the diameter h of the grid elements. Note that this means that the problem actually gets harder the finer the mesh is. This is not only due to the increased problem size but also due to the more often varying diffusion coefficient. For cell-centred finite volumes we present the results of this test in Table 3.12. In comparison with Table 3.9(c) (with fixed $\lambda = 1/16$) the number of iterations needed for convergence increases more steeply now, especially for low correlation length. For the three dimensional problem the increase in number of iterations is nearly the same as for fixed correlation length. A similar total increase was observed for an algebraic domain decomposition in Scheichl and Vainikko [2007], too. The increase for their solver is less steep for large h (and λ) but then becomes steeper and steeper for smaller h (and λ). The total increase in number of iterations is the same as for our solver. Note that we used a different approach for computing the permeability fields here. Therefore, the problems might still be rather different and the above comparison should only be used as a rough estimate.

Example 3.15. The classical anisotropic diffusion problem is described

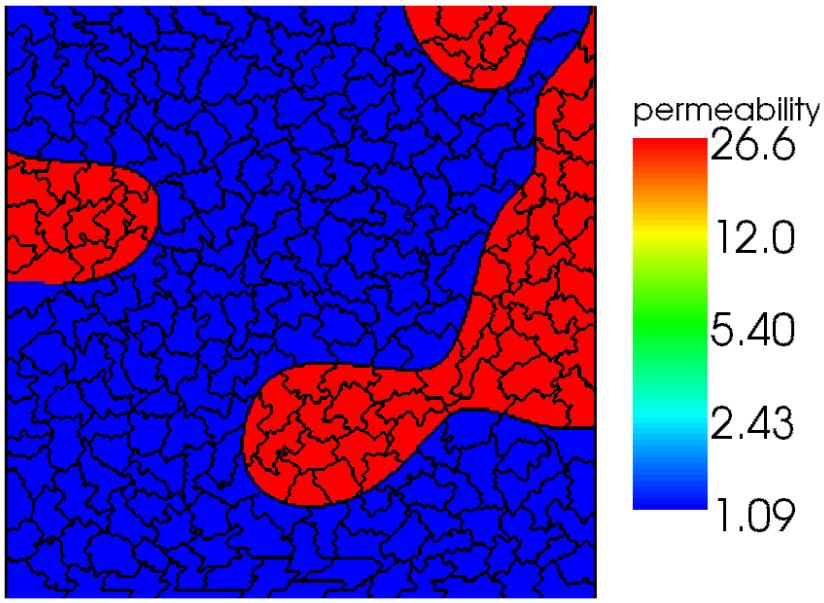


Figure 3.12: Aggregates for clipped log normal distribution of permeability field

by

$$\begin{aligned} -\nabla \cdot (\mathsf{K}(x)\nabla u) &= 1 && \text{in } \Omega = (0,1)^d, \\ u &= 0 && \text{on } \partial\Omega, \end{aligned}$$

with a diagonal tensor $K(x) \in \mathbb{R}^{d\times d}$ given by

$$K_{ij}(x) = \begin{cases} \epsilon & i = j = 0 \\ 1 & i = j > 0 \\ 0 & \text{else} \end{cases},$$

In Figure 3.13 the coarse level aggregates are illustrated for $\epsilon = 10^{-6}$. This time all points belonging to one aggregate have the same colour. Note that for this example the aggregation process follows the strong connections of the matrix graph. These are always pointing in vertical direction for this problem.

In Table 3.13, and 3.14 we present the solver statistics for solving the anisotropic model problem in two and three dimensions with $\epsilon = 10^6$.

1/h	lev.	TB	TS	It	TIt
64	2	0.016	0.032	8	0.004
128	3	0.09601	0.08001	11	0.007273
256	4	0.392	0.368	13	0.02831
512	5	1.608	1.824	16	0.114
1024	6	6.804	9.361	19	0.4927

(a) Q_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
64	2	0.016	0.04	12	0.003334
128	3	0.08401	0.104	14	0.007429
256	4	0.348	0.456	17	0.02683
512	5	1.44	2.328	20	0.1164
1024	6	5.76	11.96	24	0.4985

(b) P_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
64	2	0.012	0.032	9	0.003556
128	3	0.056	0.08001	11	0.007273
256	4	0.248	0.292	13	0.02246
512	5	0.98	1.62	16	0.1013
1024	6	4.128	8.457	20	0.4228

(c) Cell-Centred Finite Volumes

Table 3.9: Clipped Random Permeability Field 2D ($\sigma = 8$, $\hat{\eta} = 1/16$)

For the three dimensional problem, discretized with Q_1 finite elements (Table 3.14(a)) the numbers in brackets represent the values if we do not treat positive off-diagonal values as weak connections. Clearly, the iteration count decreases drastically if they are treated as weak connections. For the biggest problem, the total time to solution dropped by a factor of three. For the two dimensional problem, these changes do not have any affect as the positives off-diagonal values already are treated as weak connections due the setting of the threshold $\alpha = \frac{2}{3}$ as discussed above. The iteration steps needed for convergence increase again with the levels. For the anisotropic problem the constant in the complexity is bigger. Still, the behaviour is satisfactory as geometric multigrid methods without semi-coarsening break down for this kind of problem.

1/h	lev.	TB	TS	It	TIt
16	2	0.08401	0.072	8	0.009
32	3	0.8401	0.332	10	0.0332
64	4	7.088	3.532	13	0.2717

(a) Q_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
16	2	0.04	0.048	11	0.004364
32	3	0.38	0.284	14	0.02029
64	4	3.236	3.028	18	0.1682

(b) P_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
16	2	0.02	0.064	8	0.008001
32	3	0.228	0.264	10	0.0264
64	4	2.04	1.744	13	0.1342

(c) Cell-Centred Finite Volumes

Table 3.10: Clipped Random Permeability Field 3D ($\sigma = 8$, $\lambda = 1/4$)

λ	1/16	1/32	1/64	1/128	1/256	1/512
It	16	16	20	24	35	46
TS	1.59	1.67	2.02	2.42	3.56	4.75
TB	0.98	1.01	1.02	1.04	1.09	1.10

(a) $\sigma = 8$

σ	2	4	6	8	10
It	15	18	19	20	20
TS	1.45	1.81	1.89	2.02	2.10
TB	0.84	1.02	1.02	1.02	1.01

(b) $\lambda = 1/64$

Table 3.11: Clipped Permeability Problem with varying σ and λ, 512×512 grid, Cell-Centred Finite Volumes

1/h	lev.	TB	TS	It	TIt
64	2	0.012	0.032	9	0.003556
128	3	0.06	0.084	12	0.007
256	4	0.256	0.428	19	0.02253
512	5	1.068	2.4	24	0.1
1024	6	4.492	14.61	34	0.4298

(a) two dimensional problem

1/h	lev.	TB	TS	It	TIt
16	2	0.016	0.064	8	0.008001
32	3	0.228	0.288	11	0.02618
64	4	2.068	1.928	14	0.1377

(b) three dimensional problem

Table 3.12: Clipped Random Permeability Field ($\sigma^2 = 8$, $\lambda = 4h$), Cell-Centred Finite Volumes

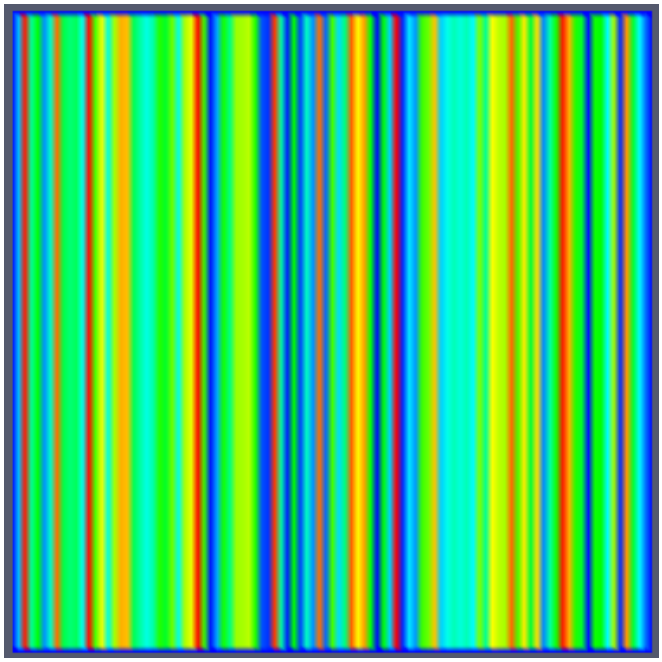


Figure 3.13: Aggregates for Anisotropic Problem

1/h	lev.	TB	TS	It	TIt
64	2	0.012	0.064	14	0.0046
128	3	0.060	0.15	18	0.0082
256	5	0.29	0.84	33	0.026
512	6	1.13	4.16	36	0.12
1024	7	4.82	22.62	44	0.51

(a) Q_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
64	2	0.012	0.052	14	0.0037
128	3	0.076	0.18	23	0.0080
256	5	0.33	1.03	39	0.026
512	6	1.35	6.24	52	0.12
1024	7	5.43	33.97	66	0.51

(b) P_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
64	2	0.008	0.02	7	0.0029
128	3	0.04	0.06	10	0.006
256	5	0.17	0.25	12	0.021
512	6	0.74	1.27	14	0.091
1024	7	3.07	7.04	18	0.39

(c) Cell-Centred Finite Volumes

Table 3.13: Anisotropic problem 2D

1/h	lev.	TB	TS	It	TIt
16	2	0.060	0.044 (0.052)	9 (18)	0.005
32	3	0.60	0.34 (0.59)	11 (36)	0.031
64	4	5.35	3.39 (25.99)	16 (72)	0.25

(a) Q_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
16	2	0.028	0.044	12	0.0037
32	3	0.31	0.34	18	0.019
64	4	2.86	4.16	26	0.16

(b) P_1 Finite Elements

1/h	lev.	TB	TS	It	TIt
16	2	0.02	0.06	8	0.0075
32	3	0.22	0.30	11	0.027
64	4	1.99	2.01	14	0.14

(c) Cell-Centred Finite Volumes

Table 3.14: Anisotropic problem 3D

3.5 Related Work and Conclusions

Algebraic multigrid methods have been around for a long time now and applied to various problems. Surprisingly, problems with randomly jumping permeability coefficients have rarely been investigated. Nevertheless, we try to compare the performance of existing approaches to solving problems that are similar to our model problems.

In the adaptive AMG approach, Brezina et al. [2006], standard interpolation AMG is extended by automatically generating smooth vectors using the linear system itself. Then the vector entries are used as additional weights for the construction of more accurate interpolation operators. The adaptive AMG is tested on a diffusion problem with jumping coefficients on the unit square. Three dimensional problems are not considered. Using a uniform rectangular grid, 20% of its cells are randomly picked. The diffusion constant in these elements is set to 10^{-8} and in the rest of the cells it is set to 1. The problem is discretized using bilinear finite elements. Recall, that for this kind of problems the condition number of the matrix increases not only because of the bigger problem, but also because of the more jumping diffusion. Using a hand tuned optimal number of pre- and post-smoothing steps for adaptive AMG to achieve the optimal solution total time for solution, the time needed for a 1024×1024 grid is a factor 521 bigger than for a 64×64 grid. Not tuning the smoothing steps leads to an even bigger factor of 735. At the same time, the number of unknowns only increases by a factor 256. The main cause for the suboptimal scalability is that the setup time increases drastically for bigger problems. For comparison we solve a similar problem with our approach and present the statistics in Table 3.15. For our approach the setup time scales optimally.

1/h	TB	TS	It	TT
64	0.016	0.032	9	0.048
128	0.09201	0.08001	11	0.172
256	0.392	0.352	13	0.844
512	1.632	1.884	16	3.41
1024	6.744	10.19	20	16.94

Table 3.15: 2D Problem from Brezina et al. [2006], Q_1 Finite Elements

The number of iterations needed for convergence increases more steeply. Still the total solution time for our approach only increases by a factor of 352. We did not apply any parameter tuning for this problem, but

used our AMG as a preconditioner to the conjugate gradient solver. For smoothed aggregation algebraic multigrid an adaptive extension similar to the one above is presented by Brezina et al. [2005].

The same problem is investigated using AMG based on compatible relaxation and energy minimisation in Brannick and Zikatanov [2007]. Again this method strives for constructing optimal interpolation operators for classic interpolation AMG. Unfortunately, only asymptotic convergence factors, and operator complexities are published. Therefore, we estimate the number of iterations needed for convergence and corresponding computational effort based on these values. For the above problem on a 512×512 uniform grid the estimated number of iterations increases by a factor of 1.5 compared with the problem using a 128×128 grid. Taking into account the increasing operator complexity reported, we assume that for this kind of problem the scalability of our AMG as a preconditioner to the conjugate gradient solver is at least comparable. For the anisotropic model problem the convergence rates for AMG based on compatible relaxation appear to be optimal. The operator complexity increases only slightly more than for our AMG. Therefore, the time needed for solving with this solver should scale better than with our AMG approach.

Despite the good convergence properties of the above AMG methods, one should bear in mind that better convergence is achieved by a more complex setup phase. This additional cost has to be remedied by an immanent decrease of the time needed for the solution. Additionally the memory consumption of the methods are bigger than that of our AMG. While these methods are very promising, to our best knowledge efficient implementations are still lacking.

A set of model problems similar to those with the log-normally distributed random fields in two dimensions were considered by Scheichl and Vainikko [2007]. The authors propose an algebraic domain decomposition method with a coarse grid correction as a preconditioner. The construction of the coarse grid is done algebraically using an aggregation scheme based on strong connections. The calculations reported are limited to two-dimensional problems and only cell-centred finite volumes are used for the discretization. The memory consumption of our AMG should be less than for the domain decomposition approach Scheichl and Vainikko. Concerning the convergence rates achieved our solver turns out to be a little less robust for varying correlation length and variance. In contrast to our approach the CPU time needed per iteration of the domain decomposition method of Scheichl and Vainikko does not scale optimally with the problem size. This might partly remedy the increase in the num-

ber of iterations needed for convergence for bigger problems.

4 AMG for Discontinuous Galerkin

Based on the content of the previous chapters, we will now introduce our algebraic multigrid method for discontinuous Galerkin discretizations of our model problem (2.1).

4.1 Auxiliary Coarse Space

Although our goal is a true multi-level method, we will start with a two-level method. We will extend it to a multi-level method later on. We need a coarser non-local subspace of the original fine level space $\mathcal{V}$. As before we will use the notation $\mathcal{V}^1$ to denote this first coarse level space. Instead of coarsening our grid, we choose to reduce the polynomial order of our approximation. Additionally, we insist that the trial and test functions of the coarse space are continuous. To achieve both at the same time, we consider the space of continuous piecewise linear basis functions P_1 on the same triangulation as our coarse space.

Depending on the polynomial order used in our discontinuous Galerkin approach, this can result in a much coarser space than before. Consider a problem in two space dimension discretized on a structured mesh consisting of $N \times N$ square elements and using polynomial order 2 on the fine level. That means that each of the N^2 elements has six discontinuous basis functions associated with it; for example $\{1, x, y, x^2, xy, y^2\}$. On the coarse level there are only $(N+1)^2$ linear basis functions with associated degrees of freedom.

Using the space of continuous basis functions, the DG bilinear form (2.4) simplifies to

$$\begin{aligned}\mathcal{A}(u,v) = &\sum_{\tau\in\mathcal{T}} (K\nabla u, \nabla v)_\tau - \sum_{e\in\mathcal{E}_D} (K\nabla u\cdot\mathbf{n}_e, v)_e \\ &+ \sum_{e\in\mathcal{E}_D} \sigma\,(K\nabla v\cdot\mathbf{n}_e, u)_e + \sum_{e\in\mathcal{E}_D} \frac{\mu}{h_e}(u,v)_e\ .\end{aligned}$$

For elements away from the Dirichlet boundary the matrix is just the traditional one for continuous basis functions. The penalty terms vanish

on the interior faces of the grid. Only on the Dirichlet boundaries we still see contributions of the integrals over the element boundary as they appear in the bilinear form of DG methods.

Note that an extension operator from the continuous coarse space onto the discontinuous fine space cannot be found in a purely algebraic way since the geometry information of the grid has to be taken into account. In our case we choose to define the extension operator by the natural embedding of the continuous trial space into the discontinuous trial space.

Let $u \in \mathcal{V}^1 \subset \mathcal{V}$ be a function of our continuous finite element space. Let $\Psi = \{\psi_0, \dots, \psi_n\}$ be the basis used for representing u in our continuous space and $\Phi = \{\phi_0, \dots, \phi_m\}$ be the basis used to represent it in the discontinuous space. Then we can represent u in the two spaces as

$$u = \sum_{i=0}^{n} \mathbf{c}_i \psi_i \; = \sum_{i=0}^{m} \mathbf{d}_i \phi_i$$

with the corresponding coefficient vectors $\mathbf{c} \in \mathbb{R}^n$ and $\mathbf{d} \in \mathbb{R}^m$.

In particular, this means that the integral equality

$$\sum_{j=0}^{m} \left(\phi_j, \phi_i\right)_\Omega \; \mathbf{d}_j = \left(\sum_{j=0}^{m} \mathbf{d}_j \phi_j, \phi_i\right)_\Omega = \left(\sum_{k=0}^{n} \mathbf{c}_k \psi_k, \phi_i\right)_\Omega = \sum_{k=0}^{n} (\psi_k, \phi_i)_\Omega \; \mathbf{c}_k$$

holds for each basis function $\phi_i \in \Phi$. Recall that the support of each element of our discontinuous basis Φ is limited to the closure of one element of our grid $\mathcal{T}$. Therefore, the entries of the matrices M, and N,

$$M_{ij} = (\phi_j, \phi_i)_\Omega, \quad N_{ij} = (\psi_k, \phi_i)_\Omega \,,$$

can be computed by integrating over one element of the grid $\mathcal{T}$ instead of integrating over the whole grid. Furthermore, the matrix M is block diagonal. Using these matrices, we define the extension operator algebraically as

$$R_0^T = M^{-1} N \,.$$

4.2 Overlapping Smoothers

One problem with the non-symmetric interior penalty methods is that traditional smoothers, like Jacobi or Gauss-Seidel, loose their smoothing properties for low and high penalty parameters $\mu \geq 0$. This behaviour was observed in Johannsen [2005], where geometric multigrid methods were

applied to NIPG discretizations. In the same paper, Johannsen numerically shows that the corresponding subspace correction methods using overlapping subspaces are robust in terms of both low and high penalty parameters.

Therefore, we use overlapping Schwarz methods as smoothers in our algebraic multi-level method when applied to NIPG discretizations with small or large penalty parameter μ. In the other cases we fall back to using non-overlapping versions. We are neglecting geometric properties of the grid and want to use solely the properties of our discretization matrix to construct the (overlapping) subspaces.

Recall that we block all unknowns associated with basis functions $\Phi_\tau = \{\phi \mid \operatorname{supp} \phi \subset \tau\}$ with support in an element of our mesh together. Let $\{\omega_i\}_{i=1}^n$ be a non-overlapping decomposition of the grid. Then this gives us our initial non-overlapping subspaces $\widetilde{\mathcal{V}}_i = \operatorname{span}\{\phi \in \Phi_{\omega_i}\}$.

If we use non-overlapping smoothers for μ of appropriate size, each ω_i will consist of exactly one grid element. That is, we use simple block versions of the traditional smoothers.

In the other cases we augment the initially non-overlapping partitions ω_i to achieve sufficient overlap. For each grid element $\tau \in \omega_i$ we add all neighbouring elements that share an edge with τ to the partition. Then for each such aggregate we define the subspace $\mathcal{V}_i$ as the span of all basis functions with nonzero support in the now overlapping partition ω_i. Note that this augmentation can be achieved purely algebraically using the graph of the block matrix as we use cell-based blocking. In Figure 4.1 one of the achieved overlapping subspaces is shown together with the matrix graph of the discretization matrix. Here the initial non-overlapping subspace consists of only one grid element.

For the construction of the initial non-overlapping subspace decomposition we utilise our aggregation algorithm 3.2 for the point-based AMG. We use the matrix graph of the block matrix of the DG discretization and the usual aggregation criterion using the row sum norm as the weight function for the strength of connection criterion.

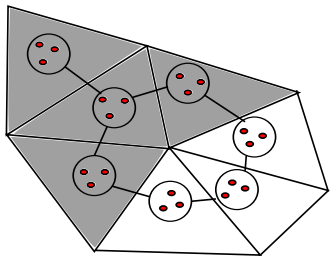


Figure 4.1: One subspace (shaded) of the smoother

4.3 Multi-Level method

As already described in Section 4.1, our coarse space is consisting of continuous functions. Therefore, the matrix on this level is scalar. We extend the two-level method to a multi-level method by simply applying our aggregation multigrid method as described in Chapter 3 on the auxiliary coarse space instead of directly solving the linear system.

Note that no additional information (such as boundary conditions on the coarse level) have to be provided. Let A be the matrix and $\mathbf{b}$ be the right hand side vector resulting from the discontinuous Galerkin discretization and let R_0 be the restriction from the space of DG functions to the space of continuous function. Then the matrix and right hand side vector for the first coarse level is constructed algebraically from this input as

$$A_0 = R_0 A R_0^T$$

and

$$\mathbf{b}_0 = R_0 \mathbf{b}\,,$$

respectively.

4.4 Numerical Results

We start the analysis of our method by solving the Poisson equation. The overlapping smoothers, introduced above, need far more computing time than the non-overlapping versions used for penalty parameters μ of appropriate size. Still, we believe that their usage is already justified by this very simple test case. We compare the non-overlapping multiplicative smoother with the overlapping smoother. The former will be labelled SOR and the latter OSOR from now on. Both methods are used as the smoother

on the DG level for our multi-level method. The multi-level method is used as the preconditioner of the BiCGSTAB solver. We are interested in the behaviour of the method for NIPG with varying penalty parameter μ. Recall that the limit case $\mu = 0$ represents the method of Baumann and Oden. The number of iterations needed to achieve a relative residual reduction of 10^{-8} are given in Tables 4.1 and 4.2 for the two dimensional and three dimensional case. The problem is discretized on a structured cube grid. We will stick to this simple grid for all our tests in this section. Where "–" appears instead of numbers we could not achieve any convergence with the method.

In these and the following tables, we will use additional acronyms to clarify the discretization and method used. *NIPG*(p, μ) means that NIPG was used with penalty parameter μ and order p for the discretization. For OBB the penalty parameter is omitted as it is naturally 0. $V(s_{dg}, s_{cg}, \nu_1, \nu_2)$ means that our AMG method with a V-cycle was used as a preconditioner. On the DG level s_{DG} was used as the smoother and on the coarser levels, s_{CG} was used as the smoother. On all levels (DG as well as CG) we performed ν_1 pre- and ν_2 post-smoothing steps.

μ	10^3	10^2	10	1	10^{-1}	10^{-2}	10^{-3}	10^{-4}	0
It. SOR	55	18	8.5	10	–	–	–	–	–
It. OSOR	12	5	4.5	3.5	6	6	6	6	6

Table 4.1: Robustness of smoothers for Poisson problem 2D, $1/h = 128$ elements, NIPG(2,μ)

μ	10^3	10^2	10	1	10^{-1}	10^{-2}	10^{-3}	10^{-4}	0
It. SOR	–	118.5	46.5	12	7.5	–	–	–	–
It. OSOR	46.5	16	7	4	3	3.5	4	3.5	3.5

Table 4.2: Robustness of smoothers for Poisson problem 2D, $1/h = 16$, NIPG(2,μ)

Clearly, using overlapping smoothers makes the method inherently more robust against changes in the penalty parameter. We regard this feature as very important as we consider the method of Baumann and Oden a good choice for problems with high contrast jumps in the coefficients.

In Table 4.3, we studied the robustness of the method of Baumann and Oden and NIPG against the polynomial order of the basis functions used.

It turns out that our method works for all tried orders. Clearly, the behaviour of the solver for OBB is nearly optimal in the number of iterations needed for orders $p \leq 5$. Due to the increasing operator complexity of the fine level matrix for higher orders, the total solution time still doubles with each increase in the polynomial order. The dimension of the square dense matrix blocks is presented in the third column (DOF/E). For NIPG the number of iterations needed for convergence increases with the number of degrees of freedom.

p	Dof	Dof/E	TB	TS	It.
2	98304	6	0.51	100.33	15
3	163840	10	0.58	337.02	14.5
4	245760	15	0.65	869.12	14.5
5	344064	21	0.97	2104.51	14
6	458752	28	1.34	6856.74	22

(a) OBB(p), V(OSOR,SOR,1,1)

p	Dof	Dof/E	TB	TS	It.
2	98304	6	0.93	1.76	7.5
3	163840	10	1.04	5.13	9
4	245760	15	1.20	12.79	10
5	344064	21	1.76	65.33	17
6	458752	28	2.38	608.80	23

(b) NIPG(p,3.9), V(OSOR,SOR,1,1)

Table 4.3: DG: Laplace 2D for higher orders p, $1/h = 128$

Concerning the efficiency of our method we see a slight increase of the iteration count with increasing problem size for the two and three dimensional Poisson problem in Tables 4.4 and 4.5 using the NIPG discretization, respectively. This is expected and due to the suboptimal complexity of the multi-level method for the first continuous coarse level. For the SIPG method we use a higher penalty parameter and the resulting linear system is more well conditioned. Accordingly, the number of iterations stays nearly constant for increasing problem size. We see the same behaviour for OBB. This time it is a result of the better smoothing properties of our overlapping smoothers. Here and later on, our overlapping local problems constructed by aggregation consist of approximately thirty-five cells.

The rest of our examples are the model problems with jumping coefficients introduced in Section 3.4. Using the unmodified versions of SIPG,

1/h	Dof	levels	TB	TS	It	TIt
32	6144	2	0.032	1.54	4	0.385
64	24576	3	0.2	6.24	4	1.56
128	98304	4	0.8481	28.49	5	5.697
256	393216	5	3.408	116.7	5	23.34
512	1572864	6	13.78	663.7	6	110.6
1024	6291456	7	58.14	3888	8	486.1

(a) OBB(2), V(OSOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
32	6144	2	0.02	0.09601	5	0.0192
64	24576	3	0.152	0.212	6	0.03534
128	98304	4	0.624	0.768	6	0.128
256	393216	5	2.572	3.132	6	0.522
512	1572864	6	10.39	12.91	6	2.151
1024	6291456	7	42.9	47.54	5	9.509

(b) SIPG(2,10), V(SOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
32	6144	2	0.02	0.104	6	0.01733
64	24576	3	0.152	0.216	6	0.036
128	98304	4	0.636	1.068	8	0.1335
256	393216	5	2.584	4.664	8	0.583
512	1572864	6	10.54	22.01	10	2.201
1024	6291456	7	43.69	115.9	13	8.917

(c) NIPG(2,3.9), V(SOR,SOR,1,1)

Table 4.4: DG: Laplace 2D

NIPG, and OBB, we were not able to achieve convergence for reasonably big problem sizes. Therefore, all our results are using the versions with weighted averages (2.6) and adapted penalty parameter (2.7) as introduced in Ern et al. [2009].

For the chequerboard model problem, Example 3.12, we present the results in Tables 4.6 and 4.7. Our solver for OBB needs the same number of iterations until convergence for the big problems of large size. For the smallest problem, the convergence is much better. In this case, the non-overlapping local problems are exactly the sixty-four cells of the chequerboard. Therefore, the augmented overlapping subdomains are the optimal subdomains for this problem. Concerning the other discretizations, we observe a similar behaviour as for the Poisson problem.

1/h	DOF	levels	TB	TS	It	TIt
8	5120	2	0.056	5.296	3	1.765
16	40960	3	0.8921	56.32	4	14.08
32	327680	4	7.752	569.9	4	142.5
64	2621440	5	63.76	6035	5	1207

(a) OBB(2), V(OSOR,SOR1,1)

1/h	DOF	levels	TB	TS	It	TIt
8	5120	2	0.04	0.416	13	0.032
16	40960	3	0.756	2.708	24	0.1128
32	327680	4	6.624	28.69	36	0.797
64	2621440	5	54.65	336.7	53	6.352

(b) SIPG(2,10), V(SOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
8	5120	2	0.044	0.376	10	0.0376
16	40960	3	0.752	2.268	15	0.1512
32	327680	4	6.676	21.89	20	1.094
64	2621440	5	55.65	345.4	39	8.858

(c) NIPG(2,3.9), V(SOR,SOR,1,1)

Table 4.5: DG: Laplace 3D

For the problem with the log-normally distributed permeability field (Example 3.13), the results for varying problem size are given in Tables 4.8 and 4.9. For the clipped version described in Example 3.14, the numbers can be found in Tables 4.10 and 4.11. In both cases, we left the variance fixed at $\sigma^2 = 8$ and the correlation length is scaled with the grid width as $\lambda = 4h$. Therefore, the permeability fields become less smooth and the problems become harder for bigger problems. This is reflected in the increasing number of iterations needed for convergence. This time we observe such an increase even for OBB and SIPG.

We examine the number of iterations needed for the relative defect reduction of 10^{-8} for different variances of the clipped log normally distributed random problem 3.14 in Table 4.12. We use OBB for the discretization and the overlapping smoothers for the fine level. The first row contains the variance used for the problem, the second row the number of iterations needed for convergence, and the last row the ratio between the highest and lowest permeability value. In all runs the correlation length in each dimension is $\lambda = 1/64$. The number of iterations only increases slightly for massively increasing variance.

1/h	DOF	levels	TB	TS	It	TIt
32	6144	2	0.028	1.308	3	0.436
64	24576	3	0.2	11.59	7	1.656
128	98304	4	0.8361	45.67	7	6.524
256	393216	5	3.424	151	6	25.17
512	1572864	6	13.82	676.1	6	112.7
1024	6291456	7	58.47	3739	7	534.1

(a) OBB(2), V(OSOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
32	6144	2	0.02	0.156	9	0.01733
64	24576	3	0.148	0.28	8	0.035
128	98304	4	0.648	1.132	8	0.1415
256	393216	5	2.616	4.632	8	0.579
512	1572864	6	10.58	18.53	8	2.317
1024	6291456	7	43.77	88.74	10	8.874

(b) SIPG(2,10), V(SOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
32	6144	2	0.02	0.152	9	0.01689
64	24576	3	0.148	0.396	11	0.036
128	98304	4	0.648	1.68	12	0.14
256	393216	5	2.608	6.172	11	0.5611
512	1572864	6	10.63	26.69	12	2.224
1024	6291456	7	44.12	107.6	12	8.967

(c) NIPG(2,10), V(SOR,SOR,1,1)

Table 4.6: DG: Chequerboard 2D

In Table 4.13 we keep the variance fixed at 4 and compare the number of iterations needed for varying correlation length λ. The number of iteration steps needed by our OBB solver turns out to be totally robust against changes of the correlation length.

In all the examples we have used SuperLU to solve the local problems of our fine level smoothers when OBB was used. To save memory and thus do reasonably big computations, we do not store and reuse the calculated factorisations. Therefore, in each smoothing step we factorise and backwards resubstitute each of the local problems one by one. As the majority of the SuperLU computation time is used for the factorisation this leads to the long times needed for one iteration. For smaller problem sizes there is the possibility to only compute the factorisation once and reuse them in

1/h	DOF	levels	TB	TS	It	TIt
16	40960	3	0.8041	201.1	14	14.36
32	327680	4	7.424	2166	13	166.6
64	2621440	5	66.2	1.14e+04	8	1425

(a) OBB(2), V(OSOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
16	40960	3	0.752	6.02	51	0.118
32	327680	4	6.472	49.19	57	0.863
64	2621440	6	59.49	627.4	91	6.895

(b) SIPG(2,10), V(SOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
16	40960	3	0.76	5.816	53	0.1097
32	327680	4	6.388	58.53	69	0.8483
64	2621440	5	56.21	290.1	45	6.448

(c) NIPG(2,10), V(SOR,SOR,1,1)

Table 4.7: Chequerboard 3D

each iteration step. If memory consumption is not a limiting parameter, this approach decreases the solution time drastically.

1/h	DOF	levels	TB	TS	It	TIt
32	6144	2	0.036	1.708	4	0.427
64	24576	3	0.196	6.856	4	1.714
128	98304	4	0.8481	30.56	5	6.112
256	393216	5	3.436	234	9	26
512	1572864	6	13.92	1207	11	109.7
1024	6291456	7	57.22	9536	18	529.8

(a) OBB(2), V(OSOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
32	6144	2	0.024	0.09201	5	0.0184
64	24576	3	0.148	0.248	7	0.03543
128	98304	4	0.644	0.744	5	0.1488
256	393216	5	2.628	8.397	14	0.5998
512	1572864	6	10.62	35.84	15	2.389
1024	6291456	7	44.61	153.4	16	9.589

(b) SIPG(2,10), V(SOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
32	6144	2	0.02	0.104	6	0.01733
64	24576	3	0.148	0.252	7	0.036
128	98304	4	0.648	1.132	9	0.1258
256	393216	5	2.644	9.173	16	0.5733
512	1572864	6	10.71	37.07	16	2.317
1024	6291456	7	45.25	173.2	19	9.113

(c) NIPG(2,3.9), V(SOR,SOR,1,1)

Table 4.8: Log Random Problem 2D

1/h	DOF	levels	TB	TS	It	TIt
8	5120	2	0.052	4.504	3	1.501
16	40960	3	0.8921	55.05	3	18.35
32	327680	4	7.752	482.6	3	160.9
64	2621440	5	63.92	6132	5	1226

(a) OBB(2), V(OSOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
8	5120	2	0.044	0.316	12	0.02633
16	40960	3	0.752	4.204	38	0.1106
32	327680	4	6.672	60.76	75	0.8102
64	2621440	5	54.04	841.9	131	6.427

(b) SIPG(2,10), V(SOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
8	5120	2	0.044	0.312	9	0.03467
16	40960	3	0.752	1.432	13	0.1102
32	327680	4	6.512	37.58	47	0.7995
64	2621440	5	53.66	1285	201	6.391

(c) NIPG(2,3.9), V(SOR,SOR,1,1)

Table 4.9: Log Random Problem 3D, NIPG(2,3.9), V(SOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
32	6144	2	0.032	1.612	4	0.403
64	24576	3	0.204	6.284	4	1.571
128	98304	4	0.8481	28.8	5	5.76
256	393216	5	3.424	116.4	5	23.29
512	1572864	6	13.76	659.8	6	110
1024	6291456	7	58.15	3930	8	491.3

(a) OBB(2), V(OSOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
32	6144	2	0.032	0.128	5	0.0256
64	24576	3	0.2	0.208	6	0.03467
128	98304	4	0.648	0.784	6	0.1307
256	393216	5	2.608	3.304	6	0.5507
512	1572864	6	10.62	13.14	6	2.191
1024	6291456	7	45.05	47.59	5	9.517

(b) SIPG(2,10), V(SOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
32	6144	2	0.02	0.132	6	0.022
64	24576	3	0.204	0.208	6	0.03467
128	98304	4	0.648	1.008	8	0.126
256	393216	5	2.624	4.556	8	0.5695
512	1572864	6	10.62	21.34	10	2.134
1024	6291456	7	45.3	114.1	13	8.774

(c) NIPG(2,3.9), V(SOR,SOR,1,1)

Table 4.10: Clipped Log Random Problem 2D

1/h	DOF	levels	TB	TS	It	TIt
8	5120	2	0.072	5.304	3	1.768
16	40960	3	0.8921	56.2	4	14.05
32	327680	4	7.724	568.8	4	142.2
64	2621440	5	63.65	6010	5	1202

(a) OBB(2), V(OSOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
8	5120	2	0.044	0.42	13	0.03231
16	40960	3	0.764	2.8	24	0.1167
32	327680	4	6.668	29.23	36	0.8121
64	2621440	5	54.8	338.6	53	6.388

(b) SIPG(2,10), V(SOR,SOR,1,1)

1/h	DOF	levels	TB	TS	It	TIt
8	5120	2	0.044	0.332	10	0.0332
16	40960	3	0.768	1.724	15	0.1149
32	327680	4	6.688	15.67	20	0.7834
64	2621440	5	54.86	247.3	39	6.341

(c) NIPG(2,3.9), V(SOR,SOR,1,1)

Table 4.11: Clipped Log Random Problem 3D

σ^2	1	2	4	8	16
It.	9	10	13	21.5	55
$\max_{\tau\nu} \frac{k_\tau}{k_\nu}$	5.5	12.3	44	369	10^5

Table 4.12: 2D Clipped Random Problem: Varying Variance

λ	1/2	1/4	1/8	1/16	1/32	1/64
It.	9.5	9	9.5	9	8.5	11.5

Table 4.13: 2D Clipped Random Problem: Varying Correlation Length

4.5 Related Work and Discussion

To our best knowledge the first publications of AMG as a solver to discontinuous Galerkin discretizations is Dobrev [2007]. It contains a case study of applying (smoothed) element agglomeration AMG to linear systems from piecewise linear SIPG discretizations. The coarse level matrices are created using a Galerkin product $\mathsf{P}^T\mathsf{A}\mathsf{P}$. The prolongation operator P from the first coarse grid is defined by the natural embedding of the corresponding space of piecewise constant trial functions into that of piecewise linear trial functions both defined on the finest grid. A recursive graph bisection algorithm is used on the adjacency graph of the grid elements to form the remaining coarser meshes. The elements of the coarse meshes are agglomerations of the elements of the fine grid. The prolongation matrices from these meshes represent the natural embeddings of the piece-wise constant trial spaces on the coarse mesh into those on the next fine mesh. No strength of connection criterion is used and therefore this approach cannot be applied to problems with jumping diffusion coefficients. Furthermore the complexity of the hierarchy building is $O(N\log(N))$, where N is the number of unknowns.

More recently, the application of smoothed aggregation algebraic multigrid was investigated by Prill et al. [2009]. The authors solve linear systems from NIPG and SIPG discretizations with piecewise constant and bi-linear quadrilateral elements. Higher order elements are discarded with the note that p-multigrid could be used to reduce the polynomial order. No numerical tests for jumping diffusion coefficients were performed. Compared to non-smoothed aggregation multigrid the presented approach produces considerable fill-in on the coarser levels. This effect would even be more amplified for higher order discretizations.

In contrast to the above mentioned approaches we have shown that our preconditioner can be used for higher order discretization and the discretizations by the Baumann and Oden method. Additionally, they are robust preconditioner for problems with highly discontinuous diffusion coefficients.

5 Parallelisation

Solving large sparse linear systems is an ubiquitous task in the numerical solution of partial differential equations. Increasing demands of computationally challenging applications both in problem size and algorithm complexity, have lead to the development of parallel scalable solver libraries for these tasks. Commonly used parallel iterative solver libraries are hypre, Falgout and Yang [2006], PETSc, Balay et al. [1997], and Trilinos, Heroux et al. [2005]. Of these only the last one provides a decent C++ interface.

All of the parallel solver libraries mentioned above work with distributed data structures (matrices and vectors) which implicitly know the data distribution and communication patterns. In contrast to this, in our approach the data distribution and communication is not built into the linear algebra data structures. This leads to a clear separation of parallelisation aspects and sequential linear solver components. Moreover the components can be adapted to the data distribution of PDE codes using either overlapping or non-overlapping grids.

In our approach the information about the data decomposition and communication interfaces is provided from outside by distributed index sets. These are kept apart from the linear algebra data structures. They are used to impose an abstract consistency model onto the building blocks (scalar products, preconditioners and parallel operators) of our iterative solvers. This allows us to minimise the communication steps in the solvers.

In the next section, we describe the proposed domain decomposition together with the parallel discretization approach. This kind of discretization is a prerequisite for our parallelisation approach. We devote Section 5.2 to the building blocks of our parallel solvers including our parallel smoothers. In Section 5.3, we finally describe the parallelisation approach of our AMG preconditioner and conclude this chapter in Section 5.4 with numerical results for our model problems. These will prove the good scalability of our approach.

5.1 Domain Decomposition

A crucial part in parallel solvers is the construction of the operators from a given domain decomposition. The usage of these operators is twofold. On the one hand, they are used for matrix vector products, for example to compute the current residual in iterative methods or for the orthogonalisation process in Krylov methods. On the other hand, they are used to construct preconditioners. In this section we will show how we set them up to handle both tasks efficiently.

We decompose the domain Ω into P (the number of processors) non-overlapping subdomains Ω_i with

$$\overline{\Omega} = \bigcup_{i=1}^{P} \overline{\Omega}_i, \quad \Omega_i \cap \Omega_j = \emptyset \text{ for } i \neq j\,.$$

The decomposition is provided from outside, for example by partitioning tools or a parallel grid manager. Therefore, the data decomposition can be either overlapping or non-overlapping. We would like to be able to cover both cases. Therefore, we introduce the additional (possibly overlapping) decomposition $\widetilde{\Omega}_i$, with

$$\Omega_i \subseteq \widetilde{\Omega}_i \subseteq \Omega\,.$$

This decomposition is the one that is provided to us. Note that

$$\widetilde{\Omega}_i = \Omega_i$$

holds for a non-overlapping domain decomposition. For an overlapping domain decomposition we have

$$\Omega_i \subsetneq \widetilde{\Omega}_i.$$

5.1.1 Finite Element Spaces

Let V_h be a finite element space, that is generated by a basis $\Phi_h = \{\phi_1, \phi_2, \ldots, \phi_N\}$. Then it is obvious that any $u \in V_h$ can be represented as $u = \sum_{i=1}^{N} x_i \phi_i$. The coefficients form a vector $x = (x_1, \ldots, x_N)^T \in \mathbb{R}^N$.

For any finite dimensional subset $I \subset \mathbb{N}$ we define $\mathbb{R}^I$ as the vectors $x = (x_{k_1}, x_{k_2}, \ldots, x_{k_M})^T$, where $k_i \in I$ and $M = |I|$. Assuming that Φ_h is a nodal basis, that is every $\phi_i \in \Phi_h$ is associated with some position $z_i \in \overline{\Omega}$, we define for every subspace $\omega \subset \overline{\Omega}$ the index set

$$I_\omega = \{k \in \{1, \ldots, N\} | z_k \in \overline{\omega}\}$$

as the (not necessarily consecutive) index set I_ω corresponding to the domain ω for the nodal basis Φ_h.

Thus, I_{Ω_i} denotes all indices of basis functions associated with the subdomain Ω_i. Note that $I_\Omega = \{1, 2, \ldots, N\}$ holds.

5.1.2 Restriction

We define the following restriction operator for an arbitrary subdomain $\omega \in \Omega$:

$$R_\omega : \mathbb{R}^{I_\Omega} \to \mathbb{R}^{I_\omega} \quad \text{by} \quad (R_\omega \mathbf{x})_k = (\mathbf{x})_k \quad \forall k \in I_\omega \tag{5.1}$$

and the corresponding prolongation operator

$$R_\omega^T : \mathbb{R}^{I_\omega} \to \mathbb{R}^{I_\Omega} \quad \text{by} \quad (R_\omega^T \mathbf{x}_\omega)_k = \begin{cases} (\mathbf{x}_\omega)_k & k \in I_\omega \\ 0 & k \notin I_\omega \end{cases} . \tag{5.2}$$

Note that R_ω just selects the coefficients of $\mathbf{x}$ that are associated with the subdomain ω and we have

$$R_\omega R_\omega^T = \mathbb{1}_\omega$$

where $\mathbb{1}_\omega$ denotes the identity on $\mathbb{R}^{I_\omega}$.

Lemma 5.1. *(Partitioning of $\mathbb{R}^{I_\Omega}$) There exists a (not necessarily unique) disjoint partitioning*

$$I_\Omega = \bigcup_{i=1}^{P} I_i , \quad I_i \cap I_j = \emptyset \quad \forall i \neq j$$

with $I_i \subset I_{\Omega_i}$.

Proof. Since $\cup \overline{\Omega}_i = \overline{\Omega}$ it is obvious that

$$\bigcup_{i=1}^{P} I_{\Omega_i} = I_\Omega$$

is a possibly overlapping decomposition. This immediately gives a constructive set

$$I_i = I_{\Omega_i} \setminus \left(\bigcup_{j=1}^{i-1} I_{\Omega_j} \right).$$

□

From now on, I_i is a disjoint partitioning based on the domain decomposition Ω_i. With $\mathbb{R}^{I_i}$ we associate the canonical restriction $R_i : \mathbb{R}^{I_\Omega} \to \mathbb{R}^{I_i}$ and the prolongation $R_i^T : \mathbb{R}^{I_i} \to \mathbb{R}^{I_\Omega}$ in the same way as in equations (5.1) and (5.2).

5.1.3 Parallel Representations

In a parallel implementation $x \in \mathbb{R}^{I_\Omega}$ cannot be stored in one process but is represented by individual pieces. As there might be the need to store more entries than the domain $\widetilde{\Omega}_i$ contains, we introduce the super-set $\widetilde{I}_i \supseteq I_{\widetilde{\Omega}_i}$ denoting all indices for which process i stores values. Let $\mathcal{P} = \{1, \ldots, P\}$ be the set of processes being used. Then each process $i \in \mathcal{P}$ stores the piece $\mathbf{x}^i \in \mathbb{R}^{\widetilde{I}_i}$ of the global vector $\mathbf{x}$.

The goal is now to use purely sequential matrix and vector data structures and operations and still be able to do parallel computations reusing sequential linear algebra kernels. Therefore, one has to impose certain constraints onto the local representations of global vectors when entering the kernel methods as well as to guarantee certain representations upon exit of the methods. These constraints will be defined here.

Definition 5.2 (Consistent/Valid representation of a vector)**.** A vector $\mathbf{x}$ is stored in a *consistent representation* on the decomposition $J_i \subseteq \widetilde{I}_i$, $i \in \mathcal{P}$ and $\cup_{i\in\mathcal{P}} J_i = I_\Omega$, if and only if for all of its components $\mathbf{x}^i$

$$(R_{\widetilde{I}_i}^T \mathbf{x}^i)_k = (\mathbf{x})_k \quad \forall k \in J_i \,,$$

on all processes $i \in \mathcal{P}$ holds.

For the case $J_i = \widetilde{I}_i$ this means that all entries in the local vector $\mathbf{x}^i$ are the same as the corresponding entries in the global vector $\mathbf{x}$. In this case $\mathbf{x}$ is said to be *consistent*.

For the case $J_i = I_i$ we speak of a *valid representation* of a vector $\mathbf{x}$.

Definition 5.3 (Additive representation of a vector)**.** A vector $\mathbf{x}$ is stored in an *additive representation* if and only if

$$\mathbf{x} = \sum_{i=1}^{P} R_{\widetilde{I}_i}^T \mathbf{x}^i \,.$$

It is obvious, that if a vector x is stored in an additive representation it can easily be transformed into a consistent representation on the decomposition $\widetilde{I}_i$, $i \in \mathcal{P}$, by

$$\mathbf{x}^i = R_{\widetilde{I}_i} \sum_{i=1}^{P} R_{\widetilde{I}_i}^T \mathbf{x}^i \,.$$

For this operation communication is needed.

A special case of the additive representation is the unique representation of a vector:

Definition 5.4 (Unique representation of a vector)**.** A vector $\mathbf{x}$ is stored in a *unique representation* on the processes of $\mathcal{P}$ if and only if

$$(R_{\widetilde{I}_i}^T \mathbf{x}^i)_k = \begin{cases} \mathbf{x}_k & k \in I_i \\ 0 & k \notin I_i \end{cases}$$

holds for all processors $i \in \mathcal{P}$.

Note that each vector x being stored in a valid representation can easily be transformed to a unique representation by a local projection, that sets all entries associated to indices of $\widetilde{I}_i \setminus I_i$ to zero. This is a purely local operation requiring no communication.

5.1.4 Operators

Let $A : \mathbb{R}^{I_\Omega} \to \mathbb{R}^{I_\Omega}$ be the global operator that shall be represented by applying local operators $A_i : \mathbb{R}^{\widetilde{I}_i} \to \mathbb{R}^{\widetilde{I}_i}$, to be defined later. These operators need to be carefully crafted and consistency constraints have to be imposed on the global vector operated on and onto the result of the application. A second purpose of the local operators is for the construction of preconditioners, which must not be neglected. Our goal is to define the local operators in a way such that the sequential preconditioners, e.g. SOR, can still compute updates stored in a valid representation.

We assume that the mesh $\mathcal{T} = \{\tau_1, \ldots, \tau_M\}$ is compatible with the subdomains. That is, with

$$\mathcal{T}(\omega) = \{\tau \in \mathcal{T} \mid \tau \cap \omega \neq \emptyset\}$$

being the elements of the grid that are part of a subdomain $\omega \subset \Omega$, it holds that

$$\bigcup_{\tau \in \mathcal{T}(\Omega_i)} \overline{\tau} = \overline{\Omega}_i \quad \text{and} \quad \bigcup_{\tau \in \mathcal{T}(\widetilde{\Omega}_i)} \overline{\tau} = \overline{\widetilde{\Omega}}_i$$

holds for all $i \in \mathcal{P}$.

With each element $\tau \in \mathcal{T}$ we associate the restriction $R_\tau : \mathbb{R}^{I_\Omega} \to \mathbb{R}^{I_\tau}$ in the way defined above. The global operator A in the finite element method is constructed locally in an additive way

$$A = \sum_{\tau \in \mathcal{T}} R_\tau^T A_\tau R_\tau \tag{5.3}$$

where A_τ, the so-called *local stiffness matrix*, is associated with the element τ.

Although we allow overlapping subdomains ($\Omega_i \subsetneq \widetilde{\Omega}_i$) here, we want to achieve local operators that only compute values for the unique partition I_i.

Since the subdomains Ω_i define a non-overlapping decomposition of Ω and the mesh is compatible, we have

$$\begin{aligned} A &= \sum_{i=1}^{P} \sum_{\tau \in \mathcal{T}(\Omega_i)} R_\tau^T A_\tau R_\tau = \sum_{i=1}^{P} \sum_{\tau \in \mathcal{T}(\Omega_i)} R_{\Omega_i}^T R_{\Omega_i} R_\tau^T A_\tau R_\tau R_{\Omega_i}^T R_{\Omega_i} \\ &= \sum_{i=1}^{P} R_{\Omega_i}^T \underbrace{\left(\sum_{\tau \in \mathcal{T}(\Omega_i)} R_{\Omega_i} R_\tau^T A_\tau R_\tau R_{\Omega_i}^T \right)}_{=: A_{\Omega_i}} R_{\Omega_i} = \sum_{i=1}^{P} R_{\Omega_i}^T A_{\Omega_i} R_{\Omega_i} . \end{aligned} \tag{5.4}$$

The local operator A_{Ω_i} is a mapping $A_{\Omega_i} : \mathbb{R}^{I_{\Omega_i}} \to \mathbb{R}^{I_{\Omega_i}}$. Thus, we have obtained an additive decomposition of the operator A. The application of the local operators A_{Ω_i} can be computed in each processor i in parallel without communication as long as the vector is stored in a consistent representation on I_{Ω_i}, $i \in \mathcal{P}$.

Unfortunately, the entries of the local matrix row might not be equal to the corresponding entries in the global matrix representation. Therefore, the result of the application of the local operators A_{Ω_i} is not stored in a valid representation on $\overline{\Omega}_i$ but in an additive representation. This means that, before continuing any computations, a communication step would be needed to store the vector in a valid representation. In addition, it is still not possible to create preconditioners working directly on the operator representation, like block Gauss-Seidel, that are able to compute updates that are consistent on I_{Ω_i} without additional communication or storing an additional representation of the operator for the preconditioner. Furthermore, overlapping subdomains $\Omega_i \neq \widetilde{\Omega}_i$ are not captured yet by this representation.

A remedy to this situation is to store on process i for each local matrix row corresponding to an index in I_i all off-diagonal nonzero entries of the global matrix with the corresponding global value. This means that our vectors might need to store additional values, due to the additional matrix entries. Therefore, the index set I_{Ω_i} needs to be augmented in the following way.

Let $G(A) = (V, E)$ be the graph of the global sparse matrix A. Then we set $\widehat{I_i} = I_{\Omega_i} \cup \{j \in I \mid \exists i \in I_{\Omega_i} \text{ with } (j, i) \in E\}$ and define the new index set $\widetilde{I_i} = I_{\widetilde{\Omega}_i} \cup \hat{I}_i$.

Now we construct the local operator mapping $A_{\widetilde{I_i}} : \mathbb{R}_{\widetilde{I_i}} \to \mathbb{R}_{\widetilde{I_i}}$ as

$$(A_{\widetilde{I_i}})_{a\beta} = \begin{cases} \left(\sum_{j=1}^{P} (R_{\Omega_i}^T A_{\Omega_i} R_{\Omega_i}) \right)_{a\beta} & \text{if } a \in I_i \wedge \beta \in \hat{I}_i \\ \delta_{a,\beta} & \text{if } a \notin I_i \\ 0 & \text{else} \end{cases} , \tag{5.5}$$

where

$$\delta_{a,\beta} = \begin{cases} 1 & \text{if } a = \beta \\ 0 & \text{else} \end{cases}$$

denotes the Kronecker delta.

Graphically the operator $A_{\widetilde{I_i}}$ has the following structure

$$\widetilde{I_i} \left\{ \begin{array}{|c|c|c|} \hline A_{ii} & * & 0 \\ \hline 0 & I & 0 \\ \hline 0 & 0 & I \\ \hline \end{array} \right. \quad \begin{array}{l} \\ \hat{I}_i \\ I_i \end{array} \left\{ \right. \quad ,$$

where

$$(A_{ii})_{a\beta} = \left(\sum_{j=1}^{P} R_{\Omega_i}^T A_{\Omega_i} R_{\Omega_i} \right)_{a\beta} = (A)_{a\beta} \tag{5.6}$$

are the entries of the sub-matrix (principal sub-matrix) A_{ii} of A with respect to the indices I_i. The entries of the matrix denoted by $*$ are equal to the entries in global matrix representation.

Note that for the case $\widehat{I_i} \not\subseteq I_{\widetilde{\Omega_i}}$, e.g. real non-overlapping grids, computing this local operator requires communication. Using this local operator, we are in position to compute an update stored in a valid representation provided that the vector is stored in a consistent representation on the decomposition $\widehat{I_i}$, $i \in \mathcal{P}$.

Using a modified local operator $S_i A_{\widetilde{I_i}}$, where $S_i = R_{\widetilde{I_i}} R_i^T R_i R_{\widetilde{I_i}}^T$ sets all entries x_i, $i \notin I_i$, to 0, results in the additive decomposition:

$$A = \sum_{i=1}^{P} R_{\widetilde{I_i}}^T S_i A_{\widetilde{I_i}} R_{\widetilde{I_i}} .$$

Assuming that the vector to which the operator A is applied is stored in a consistent representation on the decomposition $\widehat{I_i}$, $i \in \mathcal{P}$, the application of the local operators $S_i A_{\widetilde{I_i}}$ results in a vector being stored in a unique representation. Together with the mentioned constraints this operation itself does not require any communication.

5.2 Parallel Solver Components

While other parallel solver libraries, like PETSc, Balay et al. [2004], use parallel data structures (matrices and vectors) that (implicitly) know the data distribution and communication patterns we decided to clearly separate the parallelisation aspects from the data structures used. This is done by imposing an abstract consistency model onto our sequential linear algebra. It allows us to reuse the sequential linear algebra and just introduce synchronisation points into our algorithms to guarantee that the data is in the right state for the operation.

Based on the description of the domain decomposition and according parallel discretization we now introduce the building blocks of our parallel solvers.

5.2.1 Scalar Products and Norms

One of the building blocks of Krylov methods is computing scalar products and norms on the underlying vector spaces.

Let $\mathbf{x}, \mathbf{y}$ be vectors stored in a valid representation. Then a parallel scalar product can easily be computed by calculating

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{P} (R_{I_i}\mathbf{x}) \cdot (R_{I_i}\mathbf{y}) \,.$$

Note that the projection can be done locally and the sum requires one global communication step. Therefore, we do not impose any additional prerequisites onto $\mathbf{x}, \mathbf{y}$.

5.2.2 Linear Operators

In our algorithms a linear operator is used for two operations. The first is applying it to a vector $\mathbf{x}$ and storing it in another vector $\mathbf{y} = \mathsf{A}\mathbf{x}$. The second operation is scaling the result of the operator application and adding it to another vector, that is $\mathbf{y} = \mathbf{y} + a\mathsf{A}\mathbf{x}$. The prerequisites for both operations are that $\mathbf{x}$ is stored in a consistent representation and $\mathbf{y}$ is valid. On completion of both operations, $\mathbf{y}$ has to be stored in a unique representation. This post-requisite is only necessary to ensure that the result is in a representation to be processed by one of our parallel preconditioners.

As described in Section 5.1.3, the parallel storage of our operators is such that for every matrix row with index $k \in I_i$, $i \in \mathcal{P}$, associated with the unique decomposition, all non-zero column entries of the global matrix are

known to process i. Therefore, it suffices to apply the local operators defined above. After that, we do a local projection to the unique representation using the local projector S_i. That is $\mathbf{y}^i = S_i A_{\widetilde{I}_i} \mathbf{x}^i$ and $\mathbf{y}^i = S_i(\mathbf{y}^i + a A_{\widetilde{I}_i} \mathbf{x}^i)$, respectively.

5.2.3 Preconditioners and Smoothers

As preconditioners and smoothers we only consider non-overlapping additive Schwarz methods with inexact subdomain solvers. The local subdomains correspond to the span of the basis functions associated to the disjoint partitioning $I_i, i = 1, \ldots, P$, introduced above. The inexact subdomain solvers we use are traditional stationary iterative methods like Jacobi, Gauss-Seidel or SSOR. In addition, there is an incomplete LU decomposition method available.

Our local operators $A_{\widetilde{I}_i}$, $i \in \mathcal{P}$, do not correspond to the disjoint partitioning, but also allow matrix entries that represent influences from unknowns outside of I_i on process i. To prevent these influences and end up with the preconditioners above, the vector to which the preconditioner is applied has to be stored in a unique representation. After the application of the preconditioner, we assure that the result is stored consistently. This requires one communication step after the application of the local (sequential) preconditioners.

Let $M_{\widetilde{I}_i}$, $i \in \mathcal{P}$, be the sequential preconditioner computed for matrix $A_{\widetilde{I}_i}$ and $\mathbf{d}^i$ the consistently stored defect. Then the local update $\mathbf{u}^i$ is computed by applying the parallel preconditioner as

$$\mathbf{u}^i = R_{\widetilde{I}_i} \sum_{p \in \mathcal{P}} S_i M_{\widetilde{I}_i} d^i$$

5.2.4 Solvers

With the operators, preconditioners, scalar products, and norms fulfilling the pre- and post-requisites for their application as posed in the previous subsections (see Table 5.1), we can use the sequential formulation of Krylov and other iterative solvers. Provided that both, right and left hand side of our linear system, are consistent, we have fully parallel solvers without any explicit parallelisation of the solver algorithms.

component	operation	pre-condition	post-condition
scalar product norm		x valid	x unchanged
linear operator	$\mathbf{y} = \mathsf{A}\mathbf{x}$ $\mathbf{y} = \mathbf{y} + a\mathsf{A}\mathbf{x}$	$\mathbf{x}$ consistent, $\mathbf{y}$ valid	$\mathbf{y}$ unique
preconditioner smoother	$\mathbf{u} = \mathsf{M}\mathbf{d}$	$\mathbf{d}$ unique	$\mathbf{u}$ consistent

Table 5.1: Consistency Constraints onto Vector Representations for Solver Components

5.3 Parallel AMG

The multigrid Algorithm 3.1 is already inherently parallel once its single components are parallelised. We already described the parallel smoothers used for our multigrid method in Subsection 5.2.3. Therefore, only the coarsening, the prolongation, and the restriction are left to be described.

5.3.1 Coarsening Strategy

The parallelisation of the coarsening algorithm described in Chapter 3 is rather straightforward. It becomes simple and massively parallel, since the aggregation will only occur on vertices that correspond to the matrix A_{ii}. Using this approach, the coarsening process will of course deal better with the algebraic smoothness if the disjoint matrix A_{ii} is split along weak edges.

The parallel approach is described in Algorithm 5.1. It builds the aggregates $\widetilde{\mathcal{A}}^i$ of this level and the parallel index sets $\widetilde{I}_i^{\text{coarse}}$ for the next level in parallel. The parameters are the edges and vertices of the matrix graph $G(\mathsf{A}_{\widetilde{I}_i}) = (V_{\widetilde{I}_i}, E_{\widetilde{I}_i})$ and the disjoint index set I_i. The rest of the parameters are the same as for the sequential Algorithm 3.2. As a first step a subset (V_i, E_i) of the input graph that corresponds to the index set I_i is created. These graphs $\{(V_i, E_i)\}_{i \in \mathcal{P}}$ form a disjoint partitioning of the global matrix graph. Then the sequential aggregation algorithm is executed on this sub-graph. Based on the outcome of this aggregation a map between indices and corresponding aggregate indices is built and the information is published to all other processes that share vertices of the overlapping graph. Now every process knows the aggregate index of each vertex of his part of the overlapping graph and constructs the overlapping coarse index set and the aggregates. Note that this algorithm only needs one

communication step.

Algorithm 5.1 Parallel Aggregation

procedure ParallelAggregation(I_i, $V_{\widetilde{I}_i}$, $E_{\widetilde{I}_i}$, $s_{\min}$, $s_{\max}$, $d_{\max}$)
On process $i \in \mathcal{P}$**:**
$V_i \leftarrow \{v_k \in V_{\widetilde{I}_i} \mid k \in I_i\}$ ▷ Only vertices owned by i
$E_i \leftarrow \{(k,l) \in E_{\widetilde{I}_i} \mid k \in I_i \wedge l \in I_i\}$ ▷ Only edges between vertices owned by i
$(I_i^{\text{coarse}}, \mathcal{A}^i) \leftarrow$Aggregation($V_i$, E_i, $s_{\min}$, $s_{\max}$, $d_{\max}$)
$\mathbf{a}^i \leftarrow 0 \in \mathbb{N}^{\#V_{\widetilde{I}_i}}$
for $\mathcal{A}_k \in \mathcal{A}^i$ **do**
$(\mathsf{R}_{\widetilde{I}_i}^T \mathbf{a}^i)_j \leftarrow k \quad \forall v_j \in \mathcal{A}_k$
end for
$\mathbf{a}^i \leftarrow R_{\widetilde{I}_i} \sum_{q \in \mathcal{P}} \mathsf{R}_{\widetilde{I}_q}^T \mathbf{a}^q$ ▷ Communicate aggregates mapping
$\widetilde{I}_i^{\text{coarse}} \leftarrow \{k \mid \exists v_j \in V_{\widetilde{I}_i} \text{ with } (\widetilde{R}_{\widetilde{I}_i}^T \mathbf{a}^i)_j = k\}$ ▷ Coarse overlapping index set
$\widetilde{\mathcal{A}}_k^i \leftarrow \{v_j \in V_{\widetilde{I}_i} \mid (\widetilde{R}_{\widetilde{I}_i}^T \mathbf{a}^i)_j = k\}$
$\widetilde{\mathcal{A}}^i \leftarrow \{\widetilde{\mathcal{A}}_k^i \mid \exists v_j \in V_{\widetilde{I}_i} \text{ with } (\widetilde{R}_{\widetilde{I}_i}^T \mathbf{a}^i)_j = k\}$
return $(\widetilde{I}_i^{\text{coarse}}, \widetilde{\mathcal{A}}^i)$
end procedure

Remark 5.5. For each aggregate on process i, that consists of indices in $\widetilde{I}_i \setminus I$ on the fine level, the child node, representing that aggregate on the next coarser level, is again associated with an index in $\widetilde{I}_i \setminus I$. This means that for all vertices in I_i on the coarse level all neighbours they depend on or influence are also stored in process i.

The coarse level matrix is then calculated using the Galerkin product (3.20). To satisfy the constraints of our local operators (5.5), we need to set the diagonal values to 1 and the off-diagonal values to 0 for all matrix rows corresponding to the overlap region $\widetilde{I}_i^{l+1} \setminus I_i^{l+1}$. The coarse level matrix has the structure proposed by our parallelisation approach (5.5) just as the fine level matrix. Therefore, all matrix-vector operations can be performed locally on each processor on all levels of the hierarchy provided that the vectors are stored consistently.

5.3.2 Data Agglomeration

Note that our aggregation Algorithm 5.1 does not build any aggregates that cross over the borders of our disjoint partitioning. Therefore, it is

reasonable to choose the fine level partitioning of the unknowns with the aggregation in our mind. For problems with jumping coefficients, like the model problems introduced in Section 3.4, it would be ideal if the borders of our disjoint partitioning coincide with the high contrast jumps in the permeability of our problems. If this is not feasible, there should at least be a region with only mildly varying coefficients around the border of the partitioning, that is big enough for aggregation until the coarsest level.

On the fine level, we rely on the user (or third party software) providing our solver with a reasonable partitioning of the global matrices and vectors onto the available processes. Most of the time this will simply not be the case as the discretization software often cannot be configured in a way to produce a partitioning with the properties defined above.

The currently available supercomputers, like the Jugene in Jülich, are already providing more than one PetaFlops to users. These supercomputers make hundreds of thousands of cores available for usage. Even if the coarsening approach is pushed to its limit, the coarse level system would still have at least as many unknowns as the number of processors participating in the computation. Solving such a coarse level system in parallel would mean doing very few floating point operations between many communication steps. Therefore this computation would be limited by the available bandwidth of the communication network.

In addition, parallel direct sparse solver do not scale very well for large numbers of processes. In a recent report, Gupta et al. [2009], various parallel sparse direct solvers are evaluated. It was shown that even for the most scalable solver, the one of the Watson Sparse Matrix Package, Gupta [2000], this means that for a fixed number of 6400 unknowns the time needed for solving on 16 processes drops only by a factor six compared to the time needed when using only one process.

To overcome these problems, we agglomerate the data onto fewer processes whenever the average number of unknowns on a level drops below the prescribed coarsening target. The new partitioning of the matrix graph is computed using parallel graph partitioning software ParMETIS, Karypis and Kumar [1998]. As the input graph, we use the weighted graph of the global matrix. Its edge weights are set 1 for edges that are considered strong by our strength of connection measure and 0 otherwise. This tells the graph partitioning software that weak connections can be cut at no cost and leads to partitionings that keep small connected regions on one process. We believe that this approach results in sufficient coupling of strongly connected unknowns on coarser grids.

This kind of aggregation is repeated recursively until on the coarsest

level there is only one participating process. We can now use a sequential sparse direct solver as the coarse level solver.

In Figure 5.1, the interplay of the coarsening and the data agglomeration process is sketched. Each node represents a stored matrix. Next to it the level index is written. Note that on each level, where data agglomeration happens, some processes store two matrices, a non-agglomerated and an agglomerated one. The latter is attribute with an inverted comma after the level number.

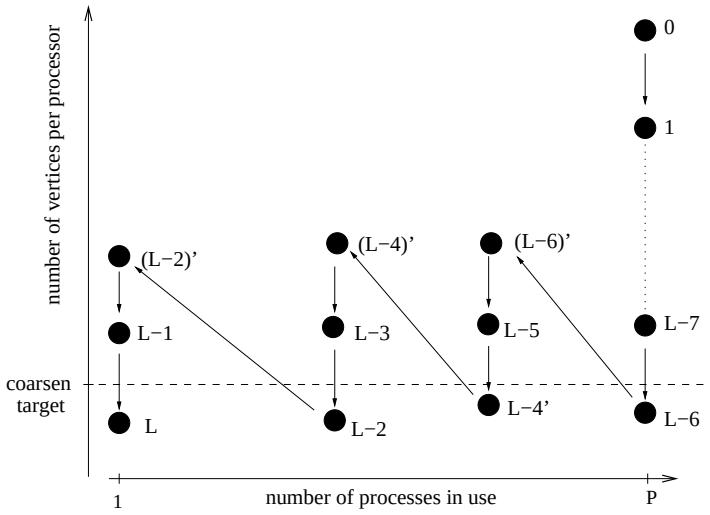


Figure 5.1: Data agglomeration

Whenever data agglomeration happened, the parallel smoothers use the not yet agglomerated matrix as input for efficiency reasons.

5.3.3 Prolongation and Restriction

For both, prolongation and restriction, we require the input vectors to be stored consistently. After the operation completes the output vector is stored in a consistent representation. Because of Remark 5.5, the prolongation to level l can be carried out locally without the need for any communication.

Unfortunately, for having a consistent vector after restricting locally from level l, all vertices of the aggregate representing the coarse level node on process i would have to be stored on the same processor. But this is just the case for aggregates consisting of indices in I_i and not for the

others. Therefore, the locally restricted vector is just stored in a valid representation and has to be made consistent using communication.

Whenever the matrix was agglomerated on the fine level, both prolongation and restriction need an additional communication step. In the restriction, the data is gathered onto fewer processors and in the prolongation, the data is scattered to the additional processes participating on the fine level.

5.4 Scalability Tests

In this section we measure the performance of our parallelisation approach experimentally. We use the model problems presented in Section 3.4. We only consider Q_1 finite elements on a structured square and cube grid in two and three dimensions, respectively. We use the V-cycle of our AMG method with one step of pre- and post-smoothing as a preconditioner to the conjugate gradient method. As a smoother, we use the approach above with one step of SSOR as inexact local solver. We stop the coarsening process at the first level with less than 2000 unknowns. Due to data agglomeration to fewer processes, all unknowns of the coarsest level reside on one process and a sequential coarse solver (SuperLU) is used.

If not labelled otherwise, all calculations contained in this section were performed on the Helics II cluster at the Interdisciplinary Centre for Scientific Computing at the university of Heidelberg. The cluster consists of 156 compute nodes. Each node has two dual core AMD Opteron 2220 CPUs with 2.8 Ghz. The nodes are interconnected by a 10G Myrinet network. In each run all four cores of a node were used.

In the presented tables the same notation was used as for our sequential tests. Additionally, we present the number of processes used labelled with "procs" and the number of grid cells labelled with $1/h$.

We start with a weak scalability test. For this test the problem size per process stays fixed. For an optimal method without any communication cost the total time to solve the linear system should stay the same regardless of how many processes are used. In Table 5.8 we present the time needed by the solver and other information for the model problems in two dimensions. We see an increase in the build time (TB) when going from one to sixteen processes. Here the number of neighbours, that a process communicates with during the aggregation and solution phase, increases. The maximum number of communication partners is reached with sixteen processes and stays the same for larger process numbers.

This is reflected in the build time that reaches a plateau for big process numbers. The remaining increase is due to the gradual data agglomeration. The same arguing holds true for the time needed for one iteration step (TIt) as it uses the same communication patterns. Still the total time to solution (TT) increases gradually. We expect this, as even the sequential method has this slight dependency of the iteration count on the number of levels used.

The differences in the behaviour of the method are due to the data agglomeration strategy using ParMETIS. The new partitionings that ParMETIS creates turn out to be different from run to run of the same problem. For the chequerboard problem the number of iterations needed stays constant from four to sixty-four processes. This is due to the geometry of the chequerboard. It has eight cells in each direction. If we change the number of cells to seven, the iteration count increases gradually as expected. For the anisotropic problem, the time needed for one iteration step stays constant for the parallel runs. Therefore, the resulting repartitionings of ParMETIS seem to be optimal in terms of communication time needed to redistribute the data.

In Table 5.3 we present the efficiency for the weak scalability. We define this weak efficiency as

$$E = \frac{T_S}{T_P},$$

where T_S is the time of our sequential method, and T_P is the time needed by our parallel method using p processes. Note that this measure is easy to compute but not really fair to our method. If we would compute the same problems sequentially, the number of iterations needed for convergence would increase with the problem size. This dependency is neglected here and we assume that our method needs a constant number of iterations regardless of the problem size in the sequential version.

For the three dimensional model problems our test results are presented in Tables 5.4 and 5.5.

Obviously our method has various overheads besides the communication overhead. This additional overhead is mainly due to the data agglomeration onto fewer and fewer processes on the coarse levels. It is the cause for the increase in the time needed per iteration (TIt) and the build time when moving from one to four participating processes. Still the overall efficiency for all the problems is approximately $\frac{1}{3}$ on 256 processes. And it does not decrease very drastically for increasing process numbers.

In Tables 5.4 and 5.5 we present results from the weak scalability tests for the three dimensional model problems. Due to more computational

work per unknown int the aggregation and the matrix vector products, the time needed for the data agglomeration and the time needed for cummunication is a lesser percentage of the total time needed for solution. Therefore, the overall scalability is better and we see a smaller drop in efficiency when going from one process to the next bigger number of processes.

To show that our solver scales well on state of the art supercomputers, we show scalability results of computations performed on Jugene located at Forschungszentrum Jülich in Table 5.6 and 5.7. Jugene is a Blue Gene / System P machine manufactured by IBM that provides more than one petaflops as overall peak performance. The data agglomeration strategy used for the tests on Jugene differs slightly from the one used on Helics. For Helics we always used 1/8 of the processes after the agglomeration. For Jugene we choose the number of processes after the data agglomeration such that the number of unknowns per process is a least eight times the coarsen target. This strategy is more aggressive and a less predictable number of agglomeration steps. Using 4096 processes data is only agglomerated twice. First to 162 and than to on process. For the run with 512 processes we also agglomerate data twice. The additional cost for this reflected in the drastic increase for the build time between the run with 512 and the one with 4096 processes. The computing power per cpu core of Jugene is less than that of Helics, while the network of Jugene has lower latency and is faster than that of Helics. This results in a better scalability of the time needed per iteration for scalability test on Jugene.

For completeness, we perform a strong scalability test on helics, too. That is, we keep the global problem size fixed for each run and just increase the number of processors participating in the computation. For this kind of test the efficiency is defined as

$$E = \frac{T_S}{pT_P}\,.$$

As expected the efficiency of our method decreases very drastically due to the less and less parallel nature of the algorithm on the coarser levels. For 256 processes the data agglomeration starts already on the fine level and the computation is actually done with only sixty-four processes. The time needed by the solver for the two dimensional model problems can be found in Table 5.8.

Last but not least, we repeat the scalability test using the two dimensional problem with the clipped random premeability field in Table 5.9. This time we leave the variance $\sigma^2 = 8$ fixed while scaling the correlation

length $\lambda = 4h$ with the width h of the grid elements. Recall, that the problems not only get harder because of the increasing problem size but also because of the decreasing correlation length. Due to this fact the number of iterations needed to achieve convergence increase more steeply now. Considering the increasing hardness of the problems the scalability is reasonably good.

procs	1/h	lev.	TB	TS	It	TIt	TT
1	1024	6	6.94	16.06	19	0.8453	23
4	2048	7	7.98	26.2	23	1.14	34.2
16	4096	8	8.74	30.7	27	1.14	39.4
64	8192	9	9.3	50.3	32	1.57	59.6
256	16384	10	9.93	56.4	37	1.52	66.3

(a) Poisson Problem

procs	1/h	lev.	TB	TS	It	TIt	TT
1	1024	6	6.85	22.68	27	0.84	29.53
4	2048	7	7.62	46.8	41	1.14	54.4
16	4096	8	8.36	59.5	43	1.38	67.8
64	8192	9	9.55	57	43	1.32	66.5
256	16384	10	9.46	100	66	1.52	110

(b) Chequerboard Permeability Problem

procs	1/h	lev.	TB	TS	It	TIt	TT
1	1024	6	7.72	23.15	24	0.9646	30.87
4	2048	7	8.28	40.9	31	1.32	49.1
16	4096	8	9.01	45.5	34	1.34	54.5
64	8192	9	9.16	52.5	39	1.35	61.7
256	16384	10	9.51	82.3	48	1.71	91.8

(c) Random Permeability Problem ($\lambda = 1/64$, $\sigma = 4$)

procs	1/h	lev.	TB	TS	It	TIt	TT
1	1024	7	7.9	50.97	45	1.133	58.87
4	2048	8	9.31	101	54	1.88	111
16	4096	9	10.1	120	67	1.8	131
64	8192	10	10.2	141	78	1.8	151
256	16384	11	10.7	161	88	1.83	171

(d) Anisotropic Permeability Problem

procs	1/h	lev.	TB	TS	It	TIt	TT
1	1024	6	7.21	21.13	22	0.9605	28.34
4	2048	7	8.33	50.1	31	1.62	58.4
16	4096	8	9.09	65.3	41	1.59	74.4
64	8192	9	9.5	68.6	49	1.4	78.1
256	16384	10	10.1	77.6	52	1.49	87.7

(e) Clipped Random Permeability Problem ($\lambda = 1/64$, $\sigma = 4$)

Table 5.2: Weak Scalability on Helics for 2D problems

procs	1/h	TB	TS	TIt	TT
4	2048	0.87	0.613	0.742	0.673
16	4096	0.794	0.523	0.744	0.583
64	8192	0.746	0.319	0.538	0.386
256	16384	0.699	0.285	0.555	0.347

(a) Poisson Problem

procs	1/h	TB	TS	TIt	TT
4	2048	0.899	0.485	0.737	0.543
16	4096	0.819	0.381	0.607	0.435
64	8192	0.717	0.398	0.634	0.444
256	16384	0.724	0.227	0.554	0.27

(b) Chequerboard Permeability Problem

procs	1/h	TB	TS	TIt	TT
4	2048	0.932	0.566	0.732	0.628
16	4096	0.857	0.509	0.721	0.567
64	8192	0.843	0.441	0.717	0.501
256	16384	0.812	0.281	0.562	0.336

(c) Random Permeability Problem ($\lambda = 1/64$, $\sigma = 4$)

procs	1/h	TB	TS	TIt	TT
4	2048	0.849	0.503	0.603	0.532
16	4096	0.786	0.423	0.63	0.451
64	8192	0.775	0.362	0.628	0.39
256	16384	0.74	0.317	0.62	0.344

(d) Anisotropic Permeability Problem

procs	1/h	TB	TS	TIt	TT
4	2048	0.866	0.422	0.595	0.485
16	4096	0.793	0.323	0.603	0.381
64	8192	0.759	0.308	0.686	0.363
256	16384	0.716	0.272	0.644	0.323

(e) Clipped Random Permeability Problem ($\lambda = 1/64$, $\sigma = 4$)

Table 5.3: Weak Efficiency on Helics for 2D Problems

procs	1/h	lev.	TB	TS	It	TIt	TT
1	64	4	7.93	6.17	12	0.5142	14.1
8	128	5	8.79	11.1	16	0.696	19.9
64	256	6	10.7	18	20	0.899	28.7
512	512	7	12.5	31.3	25	1.25	43.8

(a) Poisson Problem

procs	1/h	lev.	TB	TS	It	TIt	TT
1	64	4	7.91	6.93	13	0.5331	14.84
8	128	5	8.88	16.9	25	0.675	25.8
64	256	6	10.8	32	36	0.889	42.8
512	512	7	16.5	69.6	55	1.27	86.1

(b) Chequerboard Permeability Problem

procs	1/h	lev.	TB	TS	It	TIt	TT
1	64	4	8.03	8.64	16	0.54	16.67
8	128	5	9.49	16.9	23	0.737	26.4
64	256	6	11	29.2	30	0.974	40.2
512	512	7	13.6	53.8	42	1.28	67.4

(c) Random Permeability Problem ($\lambda = 1/64$, $\sigma^2 = 4$)

procs	1/h	lev.	TB	TS	It	TIt	TT
1	64	4	6.96	8.59	16	0.5369	15.55
8	128	6	8.08	17.8	25	0.714	25.9
64	256	7	10	26.3	31	0.847	36.3
512	512	7	13.8	54.1	38	1.42	68

(d) Anisotropic Permeability Problem

procs	1/h	lev.	TB	TS	It	TIt	TT
1	64	4	8.18	7.1	13	0.5462	15.28
8	128	5	9.36	14.5	18	0.804	23.8
64	256	6	11	19.2	22	0.875	30.3
512	512	8	12.7	38.9	30	1.3	51.6

(e) Clipped Random Permeability Problem ($\lambda = 1/64$, $\sigma^2 = 4$)

Table 5.4: Weak Scalability on Helics for 3D Problems

procs	1/h	TB	TS	TIt	TT
8	128	0.902	0.554	0.738	0.707
64	256	0.742	0.343	0.572	0.492
512	512	0.633	0.197	0.411	0.322

(a) Poisson Problem

procs	1/h	TB	TS	TIt	TT
8	128	0.891	0.411	0.79	0.576
64	256	0.731	0.216	0.6	0.346
512	512	0.479	0.0995	0.421	0.172

(b) Chequerboard Permeability Problem

procs	1/h	TB	TS	TIt	TT
8	128	0.846	0.51	0.733	0.631
64	256	0.732	0.296	0.555	0.415
512	512	0.59	0.161	0.421	0.247

(c) Random Permeability Problem ($\lambda = 1/64$, $\sigma^2 = 4$)

procs	1/h	TB	TS	TIt	TT
8	128	0.861	0.482	0.752	0.6
64	256	0.695	0.327	0.634	0.429
512	512	0.504	0.159	0.377	0.229

(d) Anisotropic Permeability Problem

procs	1/h	TB	TS	TIt	TT
8	128	0.874	0.491	0.679	0.641
64	256	0.742	0.369	0.624	0.505
512	512	0.643	0.182	0.421	0.296

(e) Clipped Random Permeability Problem ($\lambda = 1/64$, $\sigma^2 = 4$)

Table 5.5: Weak Efficiency on Helics for 3D problems

procs	1/H	lev.	TB	TS	It	TIt	TT
1	64	4	47.08	31.86	14	2.276	78.94
8	128	5	56.8	50.6	20	2.53	107
64	256	6	89.8	70.9	26	2.73	161
512	512	7	89.57	97.67	35	2.79	187.2
4096	1024	8	120.2	107.2	37	2.897	227.4

(a) Time needed by solver

procs	1/h	TB	TS	TIt	TT
8	128	0.83	0.63	0.90	0.74
64	256	0.53	0.45	0.84	0.49
512	512	0.53	0.33	0.82	0.42
4096	1024	0.39	0.30	0.79	0.35

(b) Efficiency of solver

Table 5.6: Jugene: Weak Scalability Clipped Random Permeability Problem 3D ($\lambda = 1/64$, $\sigma^2 = 8$)

procs	1/h	lev.	TB	TS	It	TIt	TT
1	64	4	46.46	24.44	11	2.222	70.9
8	128	5	56.4	39.3	16	2.46	95.7
64	256	6	88	50.2	19	2.64	138
512	512	7	87.5	60.8	23	2.64	148
4096	1024	8	113	77	28	2.75	190

(a) Time needed by solver

procs	1/h	TB	TS	TIt	TT
8	128	0.824	0.622	0.905	0.741
64	256	0.528	0.487	0.841	0.513
512	512	0.531	0.402	0.84	0.478
4096	1024	0.411	0.317	0.808	0.373

(b) Efficiency of solver

Table 5.7: Jugene: Weak Scalability Poisson Problem 3D

procs	1/h	lev.	TB	TS	It	TIt	TT
1	1024	6	6.91	16.2	19	0.8526	23.11
4	1024	6	1.9	5.93	19	0.312	7.83
16	1024	6	0.59	1.52	20	0.076	2.11
64	1024	6	0.33	0.55	19	0.0289	0.88
256	1024	6	0.49	0.4	19	0.0211	0.89

(a) Poisson Problem

procs	1/h	lev.	TB	TS	It	TIt	TT
1	1024	6	7.15	25.95	27	0.9611	33.1
4	1024	6	1.98	11.4	30	0.381	13.4
16	1024	6	0.59	2.85	31	0.0919	3.44
64	1024	6	0.34	1.06	39	0.0272	1.4
256	1024	6	0.48	1.11	37	0.03	1.59

(b) Chequerboard Permeability Problem

procs	1/h	lev.	TB	TS	It	TIt	TT
1	1024	6	7.67	23.25	24	0.9688	30.92
4	1024	6	2.05	13.5	26	0.52	15.6
16	1024	6	0.65	3.16	25	0.126	3.81
64	1024	6	0.32	0.97	31	0.0313	1.29
256	1024	6	0.49	0.76	32	0.0238	1.25

(c) Random Permeability Problem ($\lambda = 1/64$, $\sigma^2 = 4$)

procs	1/h	lev.	TB	TS	It	TIt	TT
1	1024	7	7.83	50.1	45	1.113	57.93
4	1024	7	2.12	24.7	44	0.562	26.9
16	1024	7	0.6	7	47	0.149	7.6
64	1024	7	0.43	1.69	50	0.0338	2.12
256	1024	7	0.59	1.35	45	0.03	1.94

(d) Anisotropic Permeability Problem

procs	1/h	lev.	TB	TS	It	TIt	TT
1	1024	6	6.83	18.55	22	0.8432	25.38
4	1024	6	1.98	10	27	0.371	12
16	1024	6	0.61	2.75	29	0.0948	3.36
64	1024	6	0.34	0.92	31	0.0297	1.26
256	1024	6	0.55	0.67	33	0.0203	1.22

(e) Clipped Random Permeability Problem ($\lambda = 1/64$, $\sigma^2 = 4$)

Table 5.8: Strong Scalability on Helics for 2D Problems

procs	1/h	lev.	TB	TS	It	TIt	TT
1	1024	6	8.29	32.05	38	0.8434	40.34
4	2048	7	9.2	101.4	73	1.389	110.6
16	4096	8	10.62	110.2	85	1.297	120.8
64	8192	9	12.86	162	112	1.447	174.9
256	16384	10	16.66	210.1	125	1.681	226.8

(a) Q_1 Finite Elements

procs	1/h	lev.	TB	TS	It	TIt	TT
1	1024	6	5.58	21.44	32	0.67	27.02
4	2048	7	6.51	53.03	49	1.082	59.54
16	4096	8	7.45	70.02	64	1.094	77.47
64	8192	9	8.82	89.48	71	1.26	98.3
256	16384	10	13.47	109.5	79	1.386	122.9

(b) Cell-Centred Finite Volumes

Table 5.9: Weak Scalability on Helics, Cipped Random Permeability 2D ($\lambda = 4h$, $\sigma^2 = 8$)

5.5 Related Work and Discussion

The first study of parallel aggregation-based algebraic multigrid methods was conducted in Tuminaro and Tong [2000]. The authors compared three different parallelisation approaches. The first one uses a decoupled aggregation scheme. That is, as in our approach, the agglomeration of its vertices is done by each process independently. In contrast to our approach no successive agglomeration of the data to fewer processors on the coarse levels is performed. The second method investigated is a coupled aggregation approach. The aggregation is performed first for the vertices near the boundary of the subdomain of the process. Unfortunately, the neighbouring processes need this aggregation information for forming their aggregates. Therefore this part of the aggregation algorithm is more sequential than for the decoupled approach. For modern supercomputers with many processes this approach should not be used. The third approach considered uses a parallel maximal independent set (MIS) algorithm to compute the root nodes of the aggregates that have an appropriate distance from each other that allows for forming the aggregates. At the subdomain boundaries of the process the root points of the processes with lower rank have to be known and considered by processes with a higher assigned rank. Using all these root points aggregates are built in a decoupled manner. For the three dimensional Laplacian the solution phase of all three approaches scales similarly. Of course the build time of the decoupled scheme is much faster and therefore the total time for solution is faster and scales better in this case. For the two dimensional anisotropic problem their decoupled approach fails to scale as the semi-coarsening is limited by the subdomain diameter. For more than 500 processes the coarse level was too big to be solved directly.

In Joubert and Cullum [2006] the MIS algorithm is used to parallelise the classical AMG based on interpolation. It is available in the LAMG package, see Joubert [2005]. For the three dimensional Laplace discretized with a 27-point stencil a weak scalability test with a fixed problem size of $80 \times 80 \times 80$ per process is performed in the publication. The number of iterations needed for convergence more than doubles for the weak scalability test from one to 3500 processes. Using our AMG as a preconditioner to the conjugate gradient solver, the increase of the number of iterations is a little bit more steep. Nevertheless due to our simple interpolation operators the memory consumption of our preconditioner should be much lower than that of LAMG.

Recently, Yang [2009] presented parallel algebraic multigrid methods

based on interpolation that combine aggressive coarsening with long range interpolation operators. To achieve aggressive coarsening the coarsening is performed twice to compute the coarse level matrix. The second time only on the C points achieved by the first splitting. Thus lower operator complexities can be achieved. To get a robust solver more sophisticated interpolation operators are used. These interpolate even from neighbours that are not directly connected to the vertex but to direct neighbours of it. While the number of iterations needed for convergence is very robust for a weak scalability test, the interpolation is more expensive than the short range interpolation. Additionally, the interpolation operators have to be stored, which increases the memory requirement of the method. Recall, that for our preconditioner no interpolation operators need to be stored and that operator complexities are very low, naturally. Additionally, aggressive coarsening can be achieved easily in one step.

Recently, a new parallel algebraic multigrid solver was applied to a diffusion problem with jumps in the permeability constant. In the $0.1 \times 0.1 \times 0.1$ cubes located in each corner of the unit cube the permeability is 10^{-2}. For $0.1 < x, y, z < 0.9$ the permeability is 10^3 and in the rest of the domain it is 1. The solver used is a parallel algebraic multigrid method based on interpolation. Each process computes multiple coarse grid candidates for its sub-grid. Then one coarse grid per processor is chosen, such that their union constitutes a permissible global coarse grid. Using a $31 \times 31 \times 31$ fixed size sub-grid per processes and a seven-point finite difference stencil for discretization, the number of iterations needed for convergence increases by approximately a factor five when using 4096 instead of one process on a IBM Blue Gene/L machine. The total time needed for solution increases by a factor 30.

For our method we display the number of iterations and time needed by the solver in Table 5.10. We used Q_1 Finite Elements for the discretization on Jugene. Due to this differences in the discretization scheme and machine, the times are not directly comparable. Still we only see an increase by a factor 2 in the number of iterations and by less than a factor 7 in the total time needed for solution.

Concerning problems with randomly jumping permeability fields, no other parallel algebraic multigrid solvers have been studied so far. Our approach is the first to be applied to such problems. As can be seen by comparing the Tables 5.6 and 5.10 these problems are even harder than the one above. Our AMG proves to be a very scalable preconditioner for both kinds of problems.

procs	1/H	lev.	TB	TS	It	TIt	TT
1	32	3	5.353	2.59	9	0.2878	7.944
8	64	4	7.79	4.05	12	0.338	11.8
64	128	5	20.5	6.11	15	0.407	26.6
512	256	6	18.7	7.75	19	0.408	26.4
4096	512	7	40	10.7	23	0.467	50.7

Table 5.10: Jugene: Weak Scalability Test for 3D Jumping Permeability Field as in Griebel et al. [2008]

6 Applications

In the previous chapters we have examined the properties of our solver using a set of model problems. To show that our solvers can and are applied to real world problems, we will present some results from projects where our solvers were used for the arising linear system.

6.1 Water Infiltration into Heterogeneous Soil

The presented parallel solver is used by the virtual institute “Inverse Modelling of Terrestrial Systems”. The aim of the institute is to develop strategies for deriving the flow and transport parameters needed for modelling terrestrial systems. The main focus is on processes that occur at the scale of an agricultural field. Such a field is at the scale of a cell in the management of terrestrial systems. Both the dynamics and heterogeneity within such a field strongly influence the average behaviour of the system. As the modelling takes place on a large scale (landscape), these variabilities within the field cannot be represented explicitly. Therefore, the effective model parameters for the large scale model need to be found from the with-in field structure and dynamics. These are needed to predict the average behaviour at the field scale.

Using a real world system to do this, it is hard to distinguish between effects based on measurement errors, insufficient representation of the heterogeneity, and wrong effective model parameters. Therefore, the estimation of the model parameters is based on a virtual soil plant system. This system consists of highly detailed field models with accurate representations of the with-in field conditions. The details of the fields are state variables and fluxes that are obtained by forward simulations of highly realistic 3D coupled problems.

To estimate the computation time needed for these forward simulations, scalability tests with up to 4096 processes were performed on Jugene. Jugene is an IBM BlueGene/P system with one petaflops overall peak performance located at the Forschungszentrum Jülich.

We simulated water infiltration into a heterogeneous soil unit cell of 1 m

$\times$ 1 m $\times$ 1 m. The infiltration is modelled by

$$\begin{aligned} \frac{\partial \Phi \rho S(p)}{\partial t} &= -\nabla \cdot \mathbf{u} + \rho q \quad \text{in } \Omega\,, \\ \mathbf{u} &= -\frac{\mathsf{K}(p)}{\mu}(\nabla p - \rho g) \quad \text{in } \Omega \subset \mathbb{R}^3\,. \end{aligned} \tag{6.1}$$

In (6.1), $\Phi(x)$ is the porosity of the medium, $\rho(\mathbf{x}, t)$ is the density of the fluid, $\mathsf{K}(\mathbf{x})$ denotes the absolute permeability tensor, $\mu(\mathbf{x}, t)$] is the dynamic viscosity of the fluid, $p(\mathbf{x}, t)$ is the fluid pressure, g denotes the gravity vector, and $\mathbf{u}(\mathbf{x}, t)$ is the macroscopic apparent velocity. That is, the velocity measured by an observer on the macroscopic level.

Initial conditions are hydraulic equilibrium with a ground water table at the lower boundary and a constant infiltration rate of 1mm per day. On the other sides of the domain no-flux conditions are imposed.

The time discretization was done using an implicite Euler scheme with a fixed prescribed time step. In order to keep the ratio between the time and space discretization error constant, the time step is reduced together with the spatial resolution. For the space discretization a cell centred finite volume method was used. We solved the linear system resulting from the nonlinear Newton solver with the BiCGStab solver preconditioned with our algebraic multigrid method using one step of the previously described parallel SSOR method for pre- and post-smoothing. No data agglomeration was performed and the coarse level was solved using an iterative solver.

We measured the weak scalability of the approach with 64^3 elements per process. The results of the linear solver are presented in Table 6.1. In addition to the already introduced notation, we label the number of time steps computed with "t. steps", the sum of the time spent for the iterative solution of all arising linear systems with "$\sum$TS", the total number of newton steps needed with "Newt. steps", and the sum of the number of iteration steps for solving all linear systems with "$\sum$ It". The linear solver is applied in each of the steps of the Newton method. The efficiencies of the hierarchy building, the time steps, and the time per iteration of our linear solver can be found in Table 6.2.

We see that both the build time and the time needed per iteration of the linear solver have nearly optimal efficiency. The efficiency drops slightly until we use sixty-four processes. At this stage we reach the maximum number of neighbours a process needs to communicate data with. As the communication load does not increase for larger process numbers

procs	10^6 DOF	t. steps	$\sum$TS	Newt. steps	$\sum$It	TIt	TB
1	0.26	1	393	7	21	1.76	7.66
8	2.10	2	692	11	46	1.88	8.84
64	16.8	4	1143	16	88	1.92	12.24
512	134	8	1957	26	187	1.95	12.06
4096	1074	16	3033	38	345	1.95	12.01

Table 6.1: Weak Scalability for water infiltration on JUGENE

procs	t. steps	TB	TIt
8	1.13	0.87	0.94
64	1.37	0.63	0.92
512	1.60	0.64	0.90
4096	2.07	0.64	0.90

Table 6.2: Weak Efficiency for Water Infiltration on JUGENE

the efficiency does not drop any more. The number of steps needed by the iterative solver for convergence increases with the larger problem sizes. This increase is as expected if we consider the average number of iteration steps of the linear solver per step of the Newton method. The total increase of the average number of iteration steps per Newton step from the smallest to the largest problem is approximately a factor of three.

We use only half the time step size when we use double the number of unknowns per dimension. Due to the smaller time steps of the bigger problems the resulting linear systems are easier to solve. This is reflected in an increase of efficiency for the total solution time with more processes performing the calculation.

6.2 Two-Phase Flow

With the notation of the last section we will now turn to two-phase flow in a porous media. Let $\Omega \subset \mathbb{R}^3$ be our domain. We extend the notation, introduced in the last section, by denoting the phase with a subscript $a = g, l$. These denote the gas and the liquid phase, respectively. Let the relative permeability of phase a be denoted by K_a, the saturation by S_a, and the capillary pressure by p_c. Then the two-phase problem on the

macroscopic level in $\Omega \times T$, $\Omega \subset \mathbb{R}^d$, $d = 2, 3$, $T = (0, T)$ reads

$$\begin{aligned} \frac{\partial \Phi \rho_a S_a}{\partial t} &= -\nabla \cdot (\rho_a \mathbf{u}_a) + \rho_a q_a \,, \\ \mathbf{u}_a &= -\frac{K_a}{\mu_a} K(\nabla p_a - \rho_a \mathbf{g}) \,, \\ S_l + S_g &= 1 \,, \\ p_g - p_l &= p_c(S_w) \end{aligned} \tag{6.2}$$

with appropriate initial and boundary conditions.

The two-dimensional problem models water infiltration into an artificial heterogeneous structure. The boundary conditions and material setup are chosen to represent the experiment of Rossi et al. [2008]. The heterogeneous structure is a 75 cm × 40 cm × 5 cm tank packed with randomly arranged layers of 0.5 cm × 5 cm × 5 cm. Each layer contains one kind of three quartz sands of different grain size. The layers are inclined with 45°. The space remaining between the layers and the tank boundaries is filled with a mixture of the three sands. The structure is illustrated in Figure 6.1. On the sides of the tank no-flux boundary conditions are

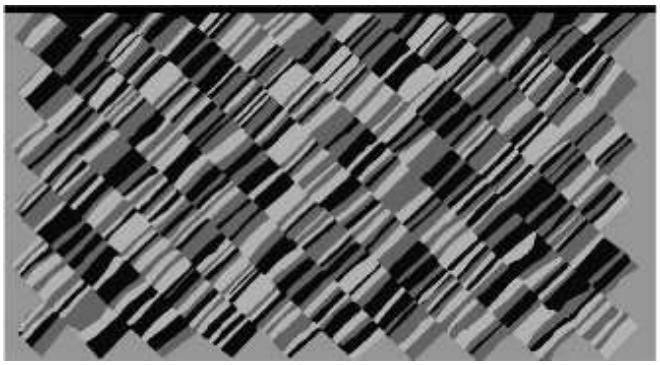


Figure 6.1: Heterogeneous Structure of 2D Sand Tank (Rossi et al. [2008])

imposed. At the upper boundary Neumann conditions with a constant flow rate are set. A constant suction at the lower boundary is imposed by using Dirichlet boundary conditions for the pressure. Initially, the sand is dry and is then wetted by the infiltration described above.

The settings of the three-dimensional problem represent those used by Vogel et al. [2006]. The domain represents a 1 m × 1 m × 0.7 m cuboid of soil consisting of three different horizons under an agricultural field. See Figure 6.2 for an illustration of the soil. The top most horizon from 0 to

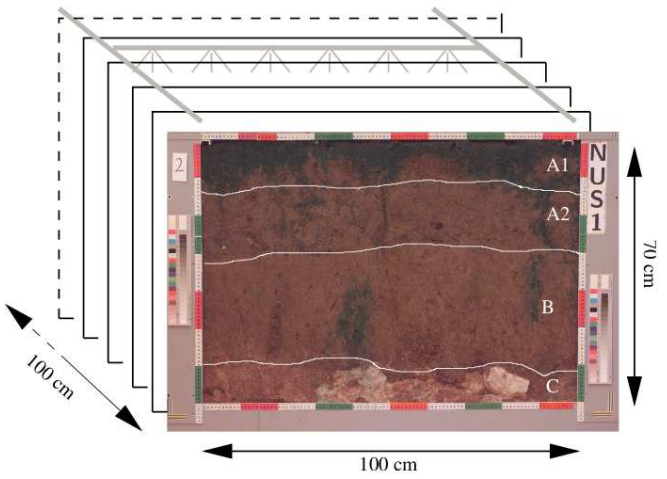


Figure 6.2: Cuboid of Soil beneath an Agricultural Field (Vogel et al. [2006])

25 cm was formerly ploughed and has a plough-pan at 35 cm. Due to a change in ploughing techniques, the first 10 cm form a loose and crumbly surface layer, while the lower regions are markedly more compact with a polyhedric structure and vertical earth-worm burrows. From 35 to 60 cm there is a structured horizon with less organic matter. The bottom most horizon is a cryoturbated, calcareous one including calcareous stones. That is, this horizon contains a mixture of materials from formerly different horizons. Again a constant flux is imposed at the upper boundary and no-flux boundary conditions are imposed at the sides. At the lower boundary gravity flow is assumed.

In contrast to the last section, we will just demonstrate the performance of the linear solver here. For the space discretization a cell centred finite volume method is used. Again a second order backward difference scheme is used for the time discretization and Newtons method is employed as the nonlinear solver. We present the time needed for solving the linear system arising in the third Newton step of the first time step. We compare the variable-based AMG with the point-based AMG using the couplings of the first phase for the strength of connection measure as described in Section 3.3. The results are presented in Table 6.3.

For both the two and three dimensional problem we compute with a coarse and fine resolution of the problem. The second column of the table indicates whether the scalar or the block version of the AMG precondi-

dim	type	DOF	TT	TB	TS	It	TIt
2	scalar	$1.4 \cdot 10^6$	83.11	19.21	63.90	86	0.75
2	scalar	$5.6 \cdot 10^6$	419.48	77.57	341.91	106	3.23
3	scalar	$5.5 \cdot 10^5$	29.62	10.96	18.66	50	0.37
3	scalar	$3.4 \cdot 10^6$	242.27	73.61	168.66	67	2.52
2	block	$1.4 \cdot 10^6$	78.87	5.15	73.72	87	0.84
2	block	$5.6 \cdot 10^6$	303.38	18.37	285.01	96.5	2.95
3	block	$5.5 \cdot 10^5$	22.07	2.95	19.12	49	0.39
3	block	$3.4 \cdot 10^6$	203.55	20.21	183.34	63.5	2.88

Table 6.3: Two-Phase Flow

tioner was used. For both methods the number of steps, that the linear solver needs for convergence, is comparable. For the block versions the complexity for one step of the iterative solver is slightly larger than for the scalar case. The time needed for building the matrix hierarchy for the block version is nearly a factor of four faster than for the scalar version. This increase of efficiency redeems the loss of efficiency of the solution phase and the block version turns out to be faster than the scalar version of our preconditioner for this problem.

6.3 Upscaling for Reservoir Models

In this section we summarise results from Rekdal [2009] where our sequential solver was used for upscaling in oil reservoir models.

The permeability K of a material is a measure of its ability to transmit fluids. It is an important parameter of the reservoir models used in the oil industry.

Often the permeability field $\mathsf{K}(x)$ has heterogeneous variations at many length scales. The finest length scale would be the pore scale ($\approx 10^{-3}m$) where even grains of sand are resolved. Even if the simulation is performed at a much larger scale these fine scale variations influence the coarse scale behaviour. Therefore, these influences must be incorporated into the parameters of the coarse scale model. The determination of the parameters for the coarse scale models is called upscaling.

To calculate the upscaled permeability tensor $\mathsf{K}(x)$, the partial differen-

tial equation

$$\begin{aligned} v &= -\mathsf{K}(x)p \quad \text{in } \Omega \subset \mathbb{R}^3 \\ \nabla \cdot v &= 0 \quad \text{in } \Omega \\ \frac{\partial p}{\partial n} &= f_N \quad \Gamma_N \\ p &= g_D \quad \Gamma_D\,, \end{aligned} \tag{6.3}$$

Darcy's law combined with incompressible fluid conditions, has to be solved for the pressure p for three sets of boundary conditions. Each of the boundary condition sets imposes a net pressure drop of one in a different coordinate direction v. Then the numerical velocity $v^v = -\mathsf{K}(x)p$ has to be found. Now the entries of the permeability tensor can be computed from the net-flow velocity Q_ζ^v in ζ direction for the pressure drop in v direction and the average distance Δ_v between faces in v direction by

$$\mathsf{K}_{\zeta v} = Q_\zeta^v \Delta_v\,. \tag{6.4}$$

We use the function spaces defined by

$$\begin{aligned} H^{\text{div}}(\Omega) &= \{u \in (L^2(\Omega))^3 \mid \nabla \cdot u \in L^2(\Omega)\}\,, \\ H_N^{\text{div}}(\Omega) &= \{u \in H^{\text{div}}(\Omega) \mid u \cdot n = g_N \text{ on } \Gamma_N\}\,, \text{ and} \\ H_0^{\text{div}}(\Omega) &= \{u \in H^{\text{div}}(\Omega) \mid u \cdot n = 0 \text{ on } \Gamma_N\}\,. \end{aligned}$$

Given the bilinear forms

$$\begin{aligned} \mathcal{B}(u, v) &= \int_\Omega u\mathsf{K}^{-1} v \, d\Omega\,, \\ C(v, p) &= \int_\Omega p \nabla \cdot v \, d\Omega\,, \text{and} \\ \mathcal{D}(v, \pi) &= -\int_{\partial\Omega} \pi v \cdot n \, dS\,, \end{aligned}$$

we can formulate the weak form of problem (6.3) as:
find $p \in L^2(\Omega)$ and $v \in H_N^{\text{div}}(\Omega)$ satisfying

$$\begin{aligned} \mathcal{B}(u, v) - C(u, p) &= \mathcal{D}(u, g_D) \; \forall u \in H_0^{\text{div}}(\Omega) \\ C(v, q) &= 0\,, \quad \forall q \in L^2(\Omega)\,. \end{aligned} \tag{6.5}$$

The problem is discretized using a mixed hybrid finite element method. The space of the pressure p is approximated by the space of piecewise

constant functions $P = \mathcal{V}_0$. The velocity v is approximated in the space of lowest order Raviart Thomas elements V defined by

$$\begin{aligned} RT_0(\tau) =& \{u \in (P_1(\tau)^3) \mid u(x) = a + bx, \text{ for } x \in \tau\,, \text{ and } a \in \mathbb{R}^3, b \in \mathbb{R}^3\}\,, \text{ and} \\ V =& \{u \in L^2(\Omega))^3 \mid u|_\tau \in RT_0(\tau) \forall \tau \in \mathcal{T}\}\,. \end{aligned}$$

Note that V allows discontinuities in the normal components of the velocities across element faces. Continuity is enforced using Lagrange multipliers. To use these we define with

$$\begin{aligned} \Pi =& \{\mu \in L^2(\mathcal{E}) \mid \mu \in P_0(e)\,, \forall e \in \mathcal{E}\} \\ \Pi_D =& \{\mu \in \Pi | \mu = g_D \text{ on } \Gamma_D\}\,, \text{ and} \\ \Pi_0 =& \{\mu \in \Pi | \mu = 0 \text{ on } \Gamma_D\} \end{aligned}$$

the spaces of piecewise constant functions on the faces $\mathcal{E}$ of our triangulation $\mathcal{T}$. In abuse of notation, we use the following discrete version of the bilinear form $\mathcal{D}$

$$\mathcal{D}(u, \mu) = \sum_{\tau \in T} \int_{\partial\tau} \mu u \cdot n dS\,.$$

Now the discrete problem is posed as: find $(v, p, \pi) \in V \times P \times \Pi_0$ such that

$$\begin{aligned} \mathcal{B}(u, v) - C(u, p) + \mathcal{D}(u, \pi) =& -\mathcal{D}(u, g_D) \quad \forall u \in V\,, \\ C(v, q) =& 0\,, \quad \forall q \in P\,, \text{ and} \\ \mathcal{D}(v, \mu) =& 0\,, \quad \forall \mu \in \Pi_0\,. \end{aligned} \tag{6.6}$$

Note that the last equation enforces the continuity of the normal velocity components across interior faces.

Using Schur-complement reduction twice on the saddle-point problem we obtain the symmetric and positive linear system

$$S\pi = r\,.$$

This system is solved with the conjugate gradient method preconditioned with our AMG method using ILU0 with a relaxation factor of 0.8 as the smoother. We achieve the fastest solution times for the largest problem by limiting the longest path in an aggregate by one. By this way we force semi-coarsening in most cases and most of the aggregates contain only two vertices. In Table 6.4, we present results from the upscaling calculation with the pressure drop in the x direction and no-flux boundary conditions on all other sides. The permeability data used is coming from

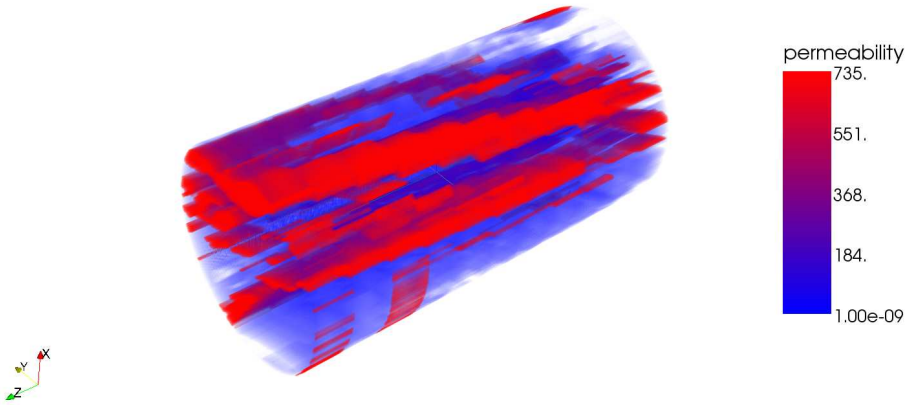


Figure 6.3: Permeability field of the Core Sample (Data provided by Statoil, Rekdal [2009])

DOF	TB	TS	TT	It
13600	0.22	0.20	0.42	17
340000	6.61	5.90	12.51	36
3042424	64.80	132.18	196.98	71

Table 6.4: Upscaling in Reservoir Simulation

a laboratory study of a core sample. The permeability field of the core sample using the highest resolution can be seen in Figure 6.3.

We see that there is an increase of the iteration steps needed to achieve the relative residual reduction of 10^{-8} for larger problem sizes.

In Rekdal [2009] our solver is compared with the commercial solver SAMG. According to their study, our solver needed twice as much time as SAMG but consumed only half the memory for the largest problem. In the mean time, we were able to tune or solver such that the solution time has dropped nearly by a factor of two. This is mainly due to limiting the aggregate diameter and forcing semi-coarsening. Admittedly, there has been no effort to tune SAMG as it is not available to us. We assume that it is still faster than our approach but at the cost of twice the memory consumption of our solver.

7 Implementational Details

7.1 Linear Algebra Interface and Data Structures

The numerical solution of partial differential equations (PDEs) frequently requires solving large and sparse linear systems. Naturally, there are many libraries available for doing sparse matrix/vector computations, see Dongarra [2006] for a comprehensive list.

The widely available Basic Linear Algebra Subprograms (BLAS) standard has been extended to cover also sparse matrices, BLAST Forum [2001]. The standard uses procedural programming style and offers only a FORTRAN and C interface. "Fine grained" interfaces, i.e. with functions consisting only of a few lines of code, such as access to individual matrix elements, impose an efficiency penalty here, as the relative cost for indirect function calls becomes huge.

Generic programming techniques in C++ or Ada offer the possibility to combine flexibility and reuse ("efficiency of the programmer") with fast execution ("efficiency of the program"). They allow the compiler to apply optimizations even for "fine grained" interfaces via static function typing. These techniques were pioneered by the Standard Template Library (STL), Stroustrup [1997]. Their efficiency advantage for scientific C++ was later demonstrated by the Blitz++ library . For an introduction to generic programming for scientific computing see Barton and Nackman [1994], Veldhuizen [1999]. Application of these ideas to matrix/vector operations is available with the Matrix Template Library (MTL, Siek and Lumsdaine [2000]) and to iterative solvers for linear systems with the Iterative Template Library (ITL).

In contrast to these libraries, the "Iterative Solver Template Library" (ISTL), which is part of the "Distributed and Unified Numerics Environment" (DUNE), see Bastian et al. [2005, 2008a,b], is designed specifically for linear systems stemming from finite element discretizations. The sparse matrices representing these linear systems exhibit a lot of structure, e.g.:

- Certain discretizations for systems of PDEs or higher order methods

result in matrices where individual entries are replaced by small blocks, say of size 2×2 or 4×4, see Fig. 7.1(a). Dense blocks of different sizes e.g. arise in *hp* discontinuous Galerkin discretization methods, see Fig. 7.1(b). It is straightforward and efficient to treat these small dense blocks as fully coupled and solve them with direct methods within the iterative method, see e.g. Bastian and Helmig [1999].

- Equation-wise ordering for systems results in matrices having an $n \times n$ block structure where n corresponds to the number of variables in the PDE and the blocks themselves are large and sparse. As an example we mention the Stokes system, see Fig. 7.1(d). Iterative solvers such as the SIMPLE or Uzawa algorithm use this structure.

- Other discretizations, e.g. those of reaction/diffusion systems, produce sparse matrices whose blocks are sparse matrices of small dense blocks, see Fig. 7.1(c).

- Other structures that can be exploited are the level structure arising from hierarchical meshes, a p-hierarchical structure (e.g. decomposition in linear and quadratic part), geometric structure from decomposition in subdomains, or topological structure where unknowns are associated with nodes, edges, faces or elements of a mesh.

Our library takes advantage of this natural block structure at compile time and supports the recursive block structuredness in a natural way.

Other libraries, like MTL, provide the blockings as views to the programmer. As this is done dynamically, the block structure cannot be used efficiently in custom generic algorithms. In the Optimized Sparse Kernel Interface (OSKI), see Vuduc et al. [2005], the sparse matrices are stored as scalar matrices, too. Here the user can provide hints about the dense block sizes which are used at runtime to tune the solvers.

In the next subsection we describe the matrix and vector interface that represents this recursive block structure via templates. In Sect. 7.1.2 we show how to exploit the block structure using template metaprogramming at compile time. Finally we sketch the high level iterative solver interface in Sect. 7.1.3.

7.1.1 Matrix and Vector Interface

The interface of our matrices is designed according to what they represent from a mathematical point of view. The vector classes are representations

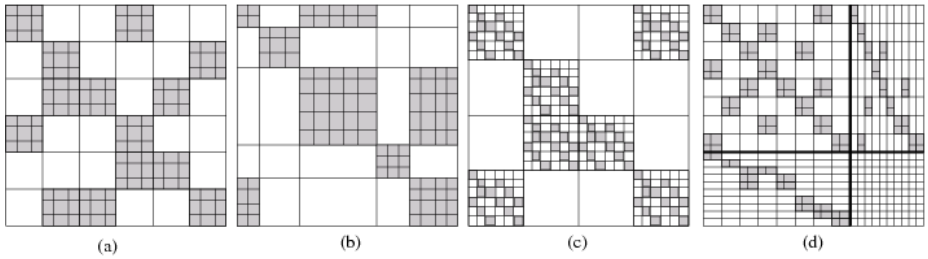


Figure 7.1: Block structure of matrices arising in the finite element method

of vector spaces while the matrix classes are representations of linear maps between two vector spaces.

Vector Spaces

Essentially, a vector space over a field $\mathbb{K}$ is a set V of elements (called vectors) along with vector addition $+ : V \to V$ and scalar multiplication $\cdot : \mathbb{K} \times V \to V$ with the well known properties. See your favourite textbook for details, e.g. Hefferson [2006]. For our application the following way of construction plays an important role: Let V_i, $i = 1, 2, \ldots, n$, be a normed vector space of dimension n_i with a scalar product, then the n-nary Cartesian product

$$V := V_1 \times V_2 \times \ldots \times V_n = \{(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n) | \mathbf{v}_1 \in V_1, \mathbf{v}_2 \in V_2, \ldots, \mathbf{v}_n \in V_n\} \quad (7.1)$$

is again a normed vector space of dimension $\sum_{i=1}^{n} n_i$ with the canonical norm and scalar product.

Treating $\mathbb{K}$ as a vector space itself we can apply this construction recursively starting from the field $\mathbb{K}$.

While for a mathematician every finite dimensional vector space is isomorphic to $\mathbb{R}^k$ for an appropriate k, for our application it is important to know how the vector space was constructed recursively by the procedure described in (7.1).

Vector Classes.

To express the construction of the vector space by n-nary products of other vector spaces, ISTL provides the following classes:

The

`template<class K, int n> class FieldVector`

class template is used to represent a vector space $V = \mathbb{K}^n$ where the field is given by the type `K`. `K` may be **`double`**, **`float`**, `complex<double>` or any other numeric type. The dimension given by the template parameter `n` is assumed to be small.

Example: Use `FieldVector<double,2>` for vectors with a fixed dimension 2.

The

`template<class B> class BlockVector`

class template builds a vector space $V = B^n$ where the "block type" B is given by the template parameter `B`. `B` may be any other class implementing the vector interface. The number of blocks n is given at run-time.

Example: `BlockVector<FieldVector<double,2> >` can be used to define vectors of variable size where each block in turn consists of two **`double`** values.

The

`template<class B> class VariableBlockVector`

class template can be used to construct a vector space having a two-level block structure of the form $V = B^{n_1} \times B^{n_2} \times \ldots \times B^{n_m}$, i.e. it consists of m blocks $i = 1, \ldots, m$ and each block in turn consists of n_i blocks given by the type `B`. In principle this structure could be built also with the previous classes, but the implementation here is more efficient. It allocates memory in one big array for all components. For certain operations it is more efficient to interpret the vector space as $V = B^N$, where $N = \sum_{i=1}^{m} n_i$.

Vectors as Containers

Vectors are containers over the base type `K` or `B` in the sense of the Standard Template Library. Random access is provided via **`operator`**`[](int i)` where the indices are in the range $0, \ldots, n-1$ with the number of blocks n given by the `N` method. Here is a code fragment for illustration:

```
typedef Dune::FieldVector<std::complex<double>,2> BType;
Dune::BlockVector<BType> v(20);
v[3][0] = 2.56;
v[3][1] = std::complex<double>(1,-1);
```

Note how one **`operator`**`[]()` is used for each level of block recursion.

Sequential access to container elements is provided via iterators. The `Iterator` class provides read/write access while the `ConstIterator`

Type	Explanation
field_type	The type of the field of the represented vector space, e.g. double.
block_type	The type of the vector blocks.
size_type	The type used for the index access and size operations.
block_level	The block level of the vector. For `FieldVector` this is 1, and 2 for `BlockVector<FieldVector<K>,n>`.
`Iterator`	The type of the iterator.
`ConstIterator`	The type of the immutable iterator.

Table 7.1: Associated Types of Vector Classes

class provides read-only access. The type names are accessed via the `::`-operator from the scope of the vector class.

A uniform naming scheme enables writing of generic algorithms. See Table 7.1 for the types provided in the scope of any vector class.

Linear Maps

For a matrix representing a linear map (or homomorphism) $A : V \to W$ from vector space V to vector space W the recursive block structure of the matrix rows and columns immediately follows from the recursive block structure of the vectors representing the domain and range of the mapping, respectively. As a natural consequence, we designed the following matrix classes:

Using the construction in (7.1), the structure of our vector spaces carries over to linear maps in a natural way.

The

```
template<class K, int n, int m> class FieldMatrix
```

class template is used to represent a linear map $M : V_1 \to V_2$ where $V_1 = \mathbb{K}^n$ and $V_2 = \mathbb{K}^m$ are vector spaces over the field given by template parameter `K`. `K` may be **`double`**, **`float`**, `complex<`**`double`**`>` or any other numeric type. The dimensions of the two vector spaces given by the template parameters `n` and `m` are assumed to be small. The matrix is stored as a dense matrix. Example: Use `FieldMatrix<`**`double`**`,2,3>` to define a linear map from a vector space over doubles with dimension 2 to one with dimension 3.

The

```
template<class B> class BCRSMatrix
```

class template represents a sparse matrix where the "block type" B is given by the template parameter `B`. `B` may be any other class implementing the matrix interface. The matrix class uses a compressed row storage scheme.

Matrices are containers over the matrix rows. The matrix rows are containers over the type `K` or `B` in the sense of the Standard Template Library. Random access is provided via `operator[](int i)` on the matrix to the matrix rows and on the matrix rows to the matrix columns (if present). Note that except for `FieldMatrix`, which is a dense matrix, `operator[]` on the matrix row triggers a binary search for the column.

For sequential access use `RowIterator` and `ColIterator` for read-/write access or `ConstRowIterator` and `ConstColIterator` for read-only access to rows and columns, respectively.

The following is a small example that prints the sparsity pattern of a matrix of type `M`:

```
typedef typename M::ConstRowIterator RowI;
typedef typename M::ConstColIterator ColI;
for(RowI row = matrix.begin(); row != matrix.end(); ++row)
{
   std::cout << "row␣"<<row.index()<<":␣"
   for(ColI col = row->begin(); col != row->end(); ++col)
     std::cout<<col.index()<<"␣";
   std::cout<<std::endl;
}
```

As with the vector interface, a uniform naming convention enables generic algorithms. See Table 7.2 for the most important names.

7.1.2 Block Recursive Algorithms

A basic feature of the concept described by the matrix and vector classes, is their recursive block structure. Let `A` be a matrix with block level $l > 1$. Then each block A_{ij} can be treated as (or actually is) a matrix itself. This recursiveness can be exploited in a generic algorithm using the defined `block_level` of the matrix and vector classes.

Note that we do not use recursive blocked algorithms on the dense matrix blocks, as described in Elmroth et al. [2004], as the dense blocks resulting from finite element discretizations will generally be small.

Most preconditioners can be modified to honor this recursive structure for a specific number of block levels k. They then work as normal on

Type	**Explanation**
field_type	The type of the field of the vector spaces we map from and to
block_type	The type representing the matrix components
row_type	The container type of the rows.
size_type	The type used for index access and size operations
block_level	The block recursion level, e.g. 1 for `FieldMatrix` and 2 for `BCRSMatrix< FieldMatrix<K,m,n> >`.
RowIterator	The type of the mutable iterator over the rows.
ConstRowIterator	The type of the immutable iterator over the rows.
ColIterator	The type of the mutable iterator over the columns of a row.
ConstColIterator	The type of the immutable iterator over the columns

Table 7.2: Type names in the matrix classes

the offdiagonal blocks, treating them as traditional matrix entries. For the diagonal values a special procedure applies: If $k > 1$ the diagonal is treated as a matrix itself and the preconditioner is applied recursively on the matrix representing the diagonal value $\mathsf{D} = \mathsf{A}_{ii}$ with block level $k - 1$. For the case that $k = 1$, the diagonal is treated as an entry of a block matrix and depending on the algorithm the corresponding linear system is either solved directly or the right hand side is returned.

In the formulation of most iterative methods upper and lower triangular and diagonal solvers play an important role. ISTL provides block recursive versions of these generic building blocks using template metaprogramming, see Table 7.3 for a listing of these methods. In the table matrix A is decomposed into $\mathsf{A} = \mathsf{L} + \mathsf{D} + \mathsf{U}$, where L is a strictly lower block triangular, D is a block diagonal, and U is a strictly upper block triangular matrix. The current residual is denoted by $\mathbf{d} = \mathbf{b} - \mathsf{A}\mathbf{x}$. It is used to calculate the update $\mathbf{v}$ to the current guess $\mathbf{x}$. An arbitrary block recursion level can be given by an additional parameter. If this parameter is omitted it defaults to 1.

Using the same block recursive template metaprogramming technique, kernels for the residual formulations of simple iterative solvers are avail-

function	computation
block triangular and block diagonal solvers	
`bltsolve(A,v,d)`	$\mathbf{v} = (L + D)^{-1}\mathbf{d}$
`bltsolve(A,v,d,`ω`)`	$\mathbf{v} = \omega(L + D)^{-1}\mathbf{d}$
`ubltsolve(A,v,d)`	$\mathbf{v} = L^{-1}\mathbf{d}$
`ubltsolve(A,v,d,`ω`)`	$\mathbf{v} = \omega L^{-1}\mathbf{d}$
`butsolve(A,v,d)`	$\mathbf{v} = (D + U)^{-1}\mathbf{d}$
`butsolve(A,v,d,`ω`)`	$\mathbf{v} = \omega(D + U)^{-1}\mathbf{d}$
`ubutsolve(A,v,d)`	$\mathbf{v} = U^{-1}\mathbf{d}$
`ubutsolve(A,v,d,`ω`)`	$\mathbf{v} = \omega U^{-1}\mathbf{d}$
`bdsolve(A,v,d)`	$\mathbf{v} = D^{-1}\mathbf{d}$
`bdsolve(A,v,d,`ω`)`	$\mathbf{v} = \omega D^{-1}\mathbf{d}$
iterative solvers	
`dbjac(A,x,b,`ω`)`	$\mathbf{x} = \mathbf{x} + \omega D^{-1}(\mathbf{b} - A\mathbf{x})$
`dbgs(A,x,b,`ω`)`	$\mathbf{x} = \mathbf{x} + \omega(L + D)^{-1}(\mathbf{b} - A\mathbf{x})$
`bsorf(A,x,b,`ω`)`	$\mathbf{x}_i^{k+1} = \mathbf{x}_i^k + \omega A_{ii}^{-1}\left[\mathbf{b}_i - \sum_{j<i} A_{ij}x_j^{k+1} - \sum_{j\geq i} A_{ij}\mathbf{x}_j^k\right]$
`bsorb(A,x,b,`ω`)`	$\mathbf{x}_i^{k+1} = \mathbf{x}_i^k + \omega A_{ii}^{-1}\left[\mathbf{b}_i - \sum_{j\leq i} A_{ij}\mathbf{x}_j^k - \sum_{j>i} A_{ij}\mathbf{x}_j^{k+1}\right]$

Table 7.3: Iterative Solver Kernels

able in ISTL. The number of block recursion levels can again be given as an additional argument. See the second part of Table 7.3 for a list of these kernels.

7.1.3 Solver Interface

The solvers of ISTL do not work on matrices directly. Instead we use an abstract operator concept. This allows for using matrix-free operators, i.e. operators that are not stored as matrices in any form. Thus our solver algorithms can easily be turned into matrix-free solvers just by plugging in matrix-free representations of linear operators and preconditioners.

Operators

The

```
template<class X, class Y> class LinearOperator
```

class template is the base class of all linear maps. The template parameter `X` is the type of the domain and `Y` is the type of the range of the operator. A linear operator provides the methods `apply(const X& x, Y & y)` and `applyscaledadd(field_type alpha, const X& x, Y& y)` performing the operations $y = A\mathbf{x}$ and $y = y + aA\mathbf{x}$, respectively. The subclass

```
template<class M, class X, class Y> class
   AssembledLinearOperator
```

represents linear operators that have a matrix representation. Conversion from any matrix into a linear operator is done by the

```
template<class M, class X, class Y> class MatrixAdapter
```

class template.

Scalar Products

For convergence tests and the orthogonalisation process Krylov methods need to compute scalar products and norms on the underlying vector spaces. The base class of all scalar products is the

```
template<class X> class Scalarproduct
```

class templates. The methods `field_type dot(const X& x, const X&y)`, which calculates the scalar product of two vectors, and `double norm(const X& x)`, which calculates the norm of a vector, must be implemented in all subclasses. For sequential programs use

```
template<class X> class SeqScalarProduct
```

which simply maps this to functions of the vector implementations.

Preconditioners

The

```
template<class X, class Y> class Preconditioner
```

class template provides the abstract base class for all preconditioners in ISTL. The method `void pre(X& x, Y& b)` has to be called once before applying the preconditioner multiple times. Here, `x` is the left hand side and `b` is the right hand side of the operator equation. The method may, e.g. scale the system, allocate memory or compute an (I)LU decomposition. The method `void apply(X& v, const Y& d)` applies the preconditioner. Here, `d` should contain the current residual and `v` should use 0

class	implements	s/p	rec.
SeqJac	Jacobi method	s	x
SeqSOR	Successive overrelaxation	s	x
SeqSSOR	Symmetric successive overrelaxation	s	x
SeqILU	Incomplete LU decomposition (ILU)	s	
SeqILUN	ILU decomposition of order N	s	
SeqOverlappingSchwarz	Overlapping Schwarz method with direct local solver	s	
Pamg::AMG	Algebraic multigrid method	s/p	
BlockPreconditioner	Wrapper to turn a sequential preconditioner into a parallel one.	p	

Table 7.4: Preconditioners

as the initial guess. Upon exit of the method, v contains the computed update to the current guess, that is $\mathbf{v} = M^{-1}\mathbf{d}$ where M is the approximate of the operator A characterising the preconditioner. The method `void post (X& x)` should be called after all computations to give the preconditioner the chance to clean up allocated resources.

See Table 7.4 for a list of available preconditioner. They have the template parameters M representing the type of the matrix they work on, X representing the type of the domain, Y representing the type of the range of the linear system. The block recursive preconditioners are marked with "x" in the last column. For them the recursion depth is specified via an additional template parameter `int l`. The column labeled "s/p" specifies whether they support **s**equential and/or **p**arallel mode.

Solvers

All solvers are subclasses of the abstract base class

```
template<class X, class Y> class InverseOperator
```

representing the inverse of an operator from the domain of type X to the range of type Y. The actual solution of the system $A(\mathbf{x}) = \mathbf{b}$ is done in the method `void apply(X& x, Y& b, InverseOperatorResult& r)`. In the `InverseOperatorResult` object some statistics about the solu-

class	**implements**
`LoopSolver`	Just applies the preconditioner multiple times
`GradientSolver`	Preconditioned gradient method
`CGSolver`	Preconditioned conjugate gradient method
`BiCGSTABSolver`	Preconditioned biconjugate gradient stabilized method
`MinRESSolver`	Minimal residual method
`RestartedGMResSolver`	Generalized minimal residual method

Table 7.5: ISTL Solvers

N	500	5000	50000	500000	5000000
MFLOPS	896	775	167	160	164

(a) scalar product

N	500	5000	50000	500000	5000000
MFLOPS	936	910	108	103	107

(b) daxpy operation $y = y + ax$

N, b	100,1	10000,1	1000000,1	1000000,2	1000000,3
MFLOPS	388	140	136	230	260

(c) Matrix-vector product, 5-point stencil, b: block size

	C	ISTL
time / it. [s]	0.17	0.18

(d) Damped Gauß-Seidel ($N = 10^6$, $b = 1$)

Table 7.6: Performance Tests

tion process, e.g. iteration count, achieved residual reduction, etc., are stored. All solvers only use methods of instances of `LinearOperator`, `ScalarProduct` and `Preconditioner`. These are provided in the constructor.

See Table 7.5 for a list of available solvers. All solvers are template classes with a template parameter `X` providing them with the vector implementation used.

7.1.4 Performance Evaluation

We evaluated the performance of our implementation on a Pentium 4 Mobile 2.4 GHz processor with a measured memory bandwidth of 1084 MB/s for the daypy operation ($x = y + az$) in Tables 7.6. The code was compiled with the GNU C++ compiler version 4.0 with -O3 optimization.

In the tables N is the number of unknown blocks (equals the number of unknowns for the scalar cases). In Tables 7.6(a), 7.6(b), 7.6(d), b is the size of the dense blocks. All matrices are sparse matrices of dense blocks. The performance for the scalar product, see Table 7.6(a), and the daxpy operation, see Table 7.6(b), is nearly optimal. For large N the limiting factor is clearly the memory bandwidth. Table 7.6(c) shows that we take advantage of cache reusage for vectors of dense blocks with block size $b > 1$. In Table 7.6(d) we compared the generic implementation of the Gauss Seidel solver in ISTL with a specialized C implementation. The measured times per iteration show that there is no significant lack of computational efficiency due to the generic implementation.

7.2 Parallel Domain Decomposition and Communication Components

When using the data parallel programming model, a set of processes works collectively on the same set of finite data objects. These might be elements of a finite element grid or vector entries in a linear algebra computation. Each process works on different partitions of the global data. Only for this partition it computes updated values.

In large scale parallel codes it is advisable to store the data partition in a local data structure directly in the local memory of the process. Due to data dependencies, the process needs to access data in the partition of other processes, too. This can either be done by communicating these values on demand between the processes whenever they are accessed. This results in data structures that are aware of the data distribution. Or by augmenting the partition of the process such that it additionally includes the data values that the other values depend on. Note that now the partitioning is not disjoint any more but overlapping. Of course, the values, other processes compute, need to be updated using communication at so called synchronisation points of the algorithm

In the latter case the data structures do not need to know anything about the data distribution. This demands more effort from the parallel algorithm designer to make sure that the data used for computations is valid, i.e. contains an updated value if another process computes the data for it. Still it allows for fewer synchronisation points in the algorithms as even in collective operations all input data may already be updated from other processes due to a previous operation. Between the necessary synchronisation points one can take advantage of the fast local memory

access.

Consider representing a random access container x on a set of processes $\mathcal{P} = \{0, \ldots, P-1\}$. It is represented by individual pieces x^p, where x^p is the piece stored on process p of the P processes participating in the calculation. Although the global representation of the container is not available on any process, a process p needs to know how the entries of its local piece x^p correspond to the entries of the global container x which would be used in a sequential program.

In Subsections 7.2.1 to 7.2.2 we present software components that are able to describe this relation between the local data structures and global data distribution. These allow us to precompute the communication interfaces for synchronising arbitrary data. Thus they can be used to easily trigger synchronisation in parallel algorithms. Finally, we will compare the performance of our approach to directly using MPI in Subsection 7.2.3.

7.2.1 Communication Software Components

From an abstract point of view, a random access container $x : I \to K$ provides a mapping from an index set $I \subset \mathbb{N}_0$ onto a set of objects K. Note that we do not require I to be consecutive. The piece x_p of the container x stored on process p is a mapping $x_p : I_p \to K$, where $I_p \subset I$. Due to efficiency, the entries of x_p should be stored consecutively in memory. This means that for the local computation the data must be addressable by a consecutive index starting from 0.

When using adaptive discretisation methods, there might be the need to reorder the indices after adding and/or deleting some of the discretisation points. Therefore, this index does not need to be persistent and can easily be changed. We will call this index *local index*.

For the communication phases of our algorithms these locally stored entries must also be addressable by a global identifier. It is used to store the received values at and to retrieve the values to be sent from the correct local position in the consecutive memory chunk. To ease the addition and removal of discretisation points this global identifier has to be persistent but does not need to be consecutive. We will call this global identifier *global index*.

ParallelIndexSet

Let $I \subset \mathbb{N}_0$ be an arbitrary, not necessarily consecutive, index set identifying all discretisation points of the computation. Furthermore, let

$$(I_p)_{p\in\mathcal{P}}, \quad \bigcup_{p\in\mathcal{P}} I_p = I$$

be an overlapping decomposition of the global index set I into the sets of indices I_p corresponding to the global indices of the values stored locally in the chunk of process p.

Then the

```
template<typename TG, typename TL>  class
   ParallelIndexSet;
```

class template realises the one to one mapping

$$\gamma_p \ : \ I_p \longrightarrow I_p^{\text{loc}} := [0, n_p)$$

of the globally unique index onto the local index.

The template parameter `TG` is the type of the global index and `TL` is the type of the local index. The only prerequisite of `TG` is that objects of this type are comparable using the less-than-operator `<`. Not that this prerequisite still allows attaching further information to the global index or even using this information as the global index. The type `TL` has to be convertible to `std::size_t` as it is used to address array elements.

The pairs of global and local indices are ordered by ascending global index. It is possible to access the pairs via **`operator`**`[](TG& global)` in $log(n)$ time, where n is the number of pairs in the set. In an efficient code it is advisable to access the index pairs using the provided iterators over the index pairs.

Due to the ordering, the index set can only be changed, i.e. index pairs added or deleted, in a special resize phase. By calling the functions `beginResize()` and `endResize()` the programmer indicates that the resize phase starts and ends, respectively. During the call of `endResize()` the deleted indices will be removed and the added index pairs will be sorted and merged with the existing ones.

ParallelLocalIndex

When dealing with overlapping index sets in distributed computing, there often is the need to distinguish different partitions of an index set.

This is accomplished by using the

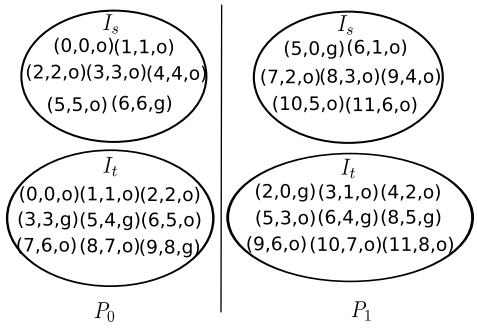


Figure 7.2: Index sets for array redistribution

```
template<typename TA> class ParallelLocalIndex;
```

class template as the type for the local index of class `ParallelIndexSet`. Here the template parameter `TA` is the type of the attributes used, e.g. an enumeration `Flags` defined by

```
enum Flags {owner, ghost};
```

where `owner` marks the indices $k \in I_p$ owned by process p and `ghost` the indices $k \notin I_p$ owned by other processes.

As an example, let us look at an array distributed between two processes. In Figure 7.3, one can see the array a as it appears in a sequential program. Below there are two different distributions of that array. The local views s_0 and s_1 are the parts process 0 and 1 store in the case that a is divided into two blocks. The local views t_0 and t_1 are the parts of a that process 0 and 1 store in the case that a is divided into 4 blocks and process 0 stores the first and third block and process 1 the second and fourth block. The decompositions have an overlap of one and the indices have the attributes `owner` and `ghost` visualised by white and shaded cells, respectively. The index sets I_s and I_t corresponding to the decompositions s_p and t_p, $p \in \{0, 1\}$, are shown in Figure 7.2 as sets of triples (g, l, a). Here, g is the global index, l is the local index and a is the attribute (either o for `owner` or g for `ghost`).

The following code snippet demonstrates how to set up the index set I_s on process 0:

```
// shortcut for index set type
typedef ParallelLocalIndex<Flags> LocalIndex;
typedef ParallelIndexSet<int, LocalIndex > PIndexSet;
PIndexSet sis;
```

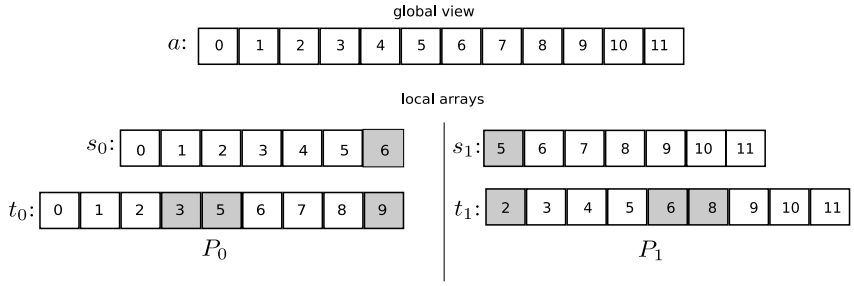


Figure 7.3: Redistributed array

```
sis.beginResize();
for(int i=0; i<6; i++)
  sis.add(i, LocalIndex(i, owner));
sis.add(6, LocalIndex(6, ghost));
sis.endResize();
```

Remote Indices

To set up communication between the processes, every process needs to know which indices are also known to other processes and which attributes are attached to them on the remote side. There are scenarios where data is exchanged between different index sets, e.g. if the data is agglomerated on lesser processes or redistributed. Therefore, communication is allowed to occur between different decompositions of the same index set.

Let $I \subset \mathbb{N}$ be the global index set and

$$(I_p^s)_{p\in\mathcal{P}}, \quad \bigcup_{p\in\mathcal{P}} I_p^s = I, \quad \text{and} \quad (I_p^t)_{p\in\mathcal{P}}, \quad \bigcup_{p\in\mathcal{P}} I_p^t = I$$

be two overlapping decompositions of the same index set I. Then an instance of class `RemoteIndices` on process $p \in \mathcal{P}$ stores the sets of triples

$$r_{p\to q}^s = \{(g,(l,a),b) \,|\, g \in I_q^s \wedge g \in I_p^t, l = \gamma_p^s(g), a = a_p^s(l), b = a_q^t(\gamma_q^t(g))\}$$

and

$$r_{p\to q}^t = \{(g,(l,a),b) \,|\, g \in I_q^s \wedge g \in I_p^t, l = \gamma_p^t(g), a = a_p^t(l), b = a_p^s(\gamma_p^s(g))\}\,,$$

for all $q \in \mathcal{P}$. Here a_p^s and a_p^t denote the mapping of local indices on process p onto attributes for the index set I_p^s and I_p^t as realised by `ParallelLocalIndex`. Note that the sets $r_{p\to q}^s$ and $r_{p\to q}^t$ will only be nonempty if the processes p and q manage overlapping index sets.

For our example in Figure 7.3 and Figure 7.2 the interface between I_s and I_t on process 0 is:

$$\begin{aligned}
r_{0\to 0}^s &= \{(0,(0,o),o),(1,(1,o),o),(2,(2,o),o),(3,(3,o),g),(5,(5,o),g),\\
&\qquad (6,(6,g),o)\}\\
r_{0\to 0}^t &= \{(0,(0,o),o),(1,(1,o),o),(2,(2,o),o),(3,(3,g),o),(5,(4,g),o),\\
&\qquad (6,(5,o),g)\}\\
r_{0\to 1}^s &= \{(2(2,o),g),(3,(3,o),o),(4,(4,o),o),(5,(5,o),o),(6,(6,g),g)\}\\
r_{0\to 1}^t &= \{(5,(4,g),g),(6,(5,o),o),(7,(6,o),o),(8,(7,o),o),(9,(8,g),o)\}
\end{aligned}$$

This information can either be calculated automatically by communicating all indices in a ring or set up by hand if the user has this information available. Assuming that `sis` is the index set I_s and `tis` the index set I_t set up as described in the previous subsection and `comm` is an MPI communicator then the simple call

```
RemoteIndices<PIndexSet> riRedist(sis, tis, comm);
riRedist.rebuild<true>();
```

on all processes automatically calculates this information and stores it in `riRedist`. For a parallel calculation on the local views s_0 and s_1, calling

```
RemoteIndices<PIndexSet> riS(sis,sis, comm);
riS.rebuild<true>();
```

on all processes builds the necessary information in `riS`.

Communication Interface

With the information provided by class `RemoteIndices` the user can set up arbitrary communication interfaces. These interfaces are realised in **`class`** `Interface`. Using the attributes attached to the indices by `ParallelLocalIndex`, the user can select subsets of the indices for exchanging data, e.g. send data from indices marked as `owner` to indices marked as `ghost`.

Basically, the interface on process p manages two sets for each process q it shares common indices with:

$$i_{p\to q}^s = \{l | (g,(l,a),b) \in r_{p\to q}^s | a \in A_s \wedge b \in A_t\}$$

and

$$i^t_{p\to q} = \{l | (g,(l,a),b) \in r^t_{p\to q} | a \in A_t \wedge b \in A_s\}\,,$$

where A_s and A_t are the attributes marking the indices where the source and target of the communication will be, respectively.

In our example these sets on process 0 will be stored for communication if $A_s = \{o\}$ and $A_t = \{o, g\}$:

$$i^s_{0\to 0} = \{0,1,3,5\} \qquad i^t_{0\to 0} = \{0,1,3,4\}$$
$$i^s_{0\to 1} = \{2,3,4,5\} \qquad i^t_{0\to 1} = \{5,6,7,8\}\,.$$

The following code snippet would build the interface above in `infRedist` as well as the interface `infS` to communicate between indices marked as `owner` and `ghost` on the local array views s_0 and s_1:

```
EnumItem<Flags,ghost> ghostFlags;
EnumItem<Flags,owner> ownerFlags;
Combine<EnumItem<Flags,ghost>, EnumItem<Flags,owner> >
   allFlags;

Interface infRedist;
Interface infS;

infRedist.build(riRedist, ownerFlags, allFlags);
infS.build(riS, ownerFlags, ghostFlags);
```

Communicator

Using the classes from the previous sections, all information about the communication is available and we are set to communicate data values of arbitrary container types. The only prerequisite for the container type is that its values are addressable via **`operator`**`[](size_t index)`. This should be safe to assume.

An important feature of our communicators is that we are not only able to send one data item per index, but also different numbers of data elements (of the same type) for each index. This is supported in a generic way by the traits class `emplae`<class V> struct CommPolicy describing the container type `V`. The **`typedef`** `IndexedType` is the atomic type to be communicated and **`typedef`** `IndexedTypeFlag` is either `SizeOne`, if there is only one data item per index, or `VariableSize`, if the number of data items per index is variable.

The default implementation works for all array-like containers which provide only one data item per index. For all other containers the user has to provide its own custom specialisation.

The **class** `BufferedCommunicator` class performs the actual communication. The template parameter `T` describes the type of the parallel index set. It uses the information about the communication interface provided by an object of class `Interface` to set up communication buffers for a container containing a specific data type. It is also responsible for gathering the data before and scattering the data after the communication step. The strict separation of the interface description from the actual buffering and communication allows for reusing the interface information with various different container and data types.

Before the communication can start, one has to call the `build` method with the data source and target containers as well as the communication interface as arguments. Assuming `s` and `t` as arrays s_i and t_i, respectively, then

```
BufferedCommunicator bComm;
BufferedCommunicator bCommRedist;
bComm.build(s, s, infS);
bCommRedist.build(s, t, infRedist);
```

demonstrates how to set up the communicator `bCommRedist` for the array redistribution and `bComm` for a parallel calculation on the local views s_i. The `build` function calculates the size of the messages to send to other processes and allocates buffers for the send and receive actions. The representatives `s` and `t` are needed to get the number of data values at each index in the case of variable numbers of data items per index. Note that, due to the generic programming techniques used, the compiler knows if the number of data points is constant for each index and will apply a specialised algorithm for calculating the message size without querying neither `s` nor `t`. Clean up of allocated resources is done either by calling the method `free()` or automatically in the destructor.

The actual communication takes place if one of the methods `forward` and `backward` is called. In our case in `bCommRedist` the `forward` method sends data from the local views s_i to the local views t_i according to the interface information and the `backward` method in the opposite direction.

The following code snippet first redistributes the local views s_i of the global array to the local views t_i and performs some calculation on this representation. Afterwards the result is communicated backwards.

```
bCommRedist.forward<CopyData<Container> >(s,t);
```

```
  // calculate on the redistributed array
  doCalculations(t);
  bCommRedist.backward<AddData<Container> >(s,t);
```

Note that both methods have a different template parameter, either `CopyData` or `AddData`. These are policies for gathering and scattering the data items. The former just copies the data from and to the location. The latter copies from the source location but adds the received data items to the target entries. Assuming our data is stored in simple C-arrays, `AddData` could be implemented like this:

```
template<typename T>
struct AddData{
  typedef typename T::value_type IndexedType;

  static double gather(const T& v, int i){
    return v[i];
  }

  static void scatter(T& v, double item, int i){
    v[i]+=item;
  }
};
```

Note that arbitrary manipulations can be applied to the communicated data in both methods.

For containers with multiple data items associated with one index, the methods `gather` and `scatter` must have an additional integer argument specifying the sub-index.

7.2.2 Collective Communication

While communicating entries of array-like structures is a prominent task in scientific computing codes, one must not neglect collective communication operations, like gathering and scattering data from and to all processes, respectively, or waiting for other processes. An abstraction for these operations is crucial for decoupling the communication from the parallel programming paradigm used.

Therefore, we designed the

`template<class T> class CollectiveCommunication`

class template which provides information of the underlying parallel programming paradigm as well as the collective communication operations as known from MPI. See Table 7.7 for a list of all functions.

Function	Description
`int rank()`	Get the rank of the process
`int size()`	Get the number of processes
`template<typename T> T sum (T & in)`	Compute global sum
`template<typename T> T prod ( T& in)`	Compute global product
`template<typename T> T min (T & in)`	Compute global minimum
`template<typename T> T max (T & in)`	Compute global maximum
`void barrier()`	Wait for all processes.
`template<typename T> int broadcast (T* inout, int len, int root)`	Broadcast an array from root to all other processes
`template<typename T> int gather (T* in, T* out, int len, int root)`	Gather arrays at a root process
`template<typename BinaryFunction, typename Type> int allreduce(Type* in, Type* out, int len)`	Combine values from all processes on all processes. Combine function is given with `BinaryFunction`

Table 7.7: Collective Communication Functions

Currently there is a default implementation for sequential programs as well as a specialisation working with MPI. This approach allows for running parallel programs sequentially without any parallel overhead simply by choosing the sequential specialisation at compile time. Note that our interface is far more convenient to use than the C++ interface of MPI. The latter is a simple wrapper around the C implementation without taking advantage of the power of generic programming. Moreover our approach can be used without MPI being installed. This of course results in using sequential versions of the solvers.

The collective communication classes were developed before the release of Boost.MPI, see Gregor and Troyer [2006]. In contrast to Boost.MPI, our approach was never meant as a full generic implementation of all MPI functions. Instead, our approach is restricted to the most basic subset of collective operations needed to implement finite element methods and iterative solvers using the previously described components. This lean interface should make it possible to easily port this approach to thread based parallelisation as well as other parallelisation paradigms. This would allow code to easily switch between different paradigms.

7.2.3 Performance Analysis

The performance of the library was compared to direct usage of MPI on the cluster "Helics II" consisting of 156 nodes with two dual core AMD Opteron 2220 2.8 GHz processors interconnected by a 10G Myrinet high speed interconnect.

The test case simulates a parallel finite element computation on a structured parallel tensor product grid in two and three dimension, respectively, with one cell overlap. In each communication step all processes exchange data with their 4 and 6 neighours, respectively, in a forward communication. Now all cells in the ghost cells have consistent data. After this communication step, the data at the ghost indices is consistent with the corresponding data owned by other processes. Now each process adds a random value to all data items and initiates a backward communication. In Figures 7.4 and 7.5, the average time for this operation is depicted for growing message sizes (implied by growing grids) in two and three dimensions.

The version labelled "MPI" uses a custom `MPI_Datatype` based on `MPI_Type_hindexed` modelling our interface information on the sending and receiving side. The version labelled "index set" in the graphs uses the software components as described above. At each communication step

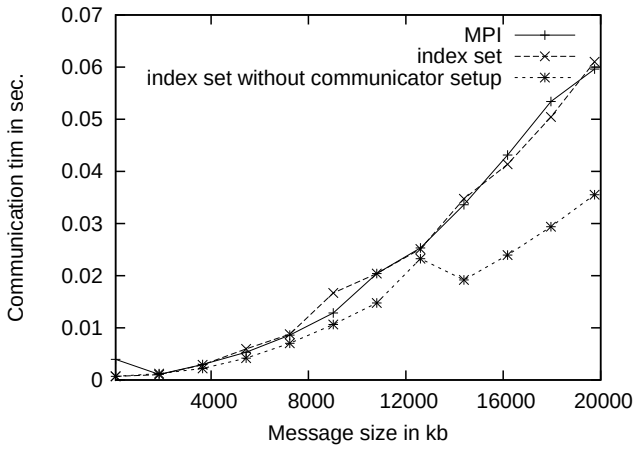


Figure 7.4: Parallel Index Set Performance 2D

the size of the buffers is calculated and the buffers are allocated. After the communication they are freed again. In the third version the buffers are allocated only once for each message size and reused at all communication steps. We see that the presented approach poses no performance penalty on parallel code. In contrast due to the added flexibility of using persistent communication buffers it can even outperform raw MPI code.

7.2.4 Related Work and Conclusion

In contrast to the presented template based approach, the PROMOTER programming model, see Giloi et al. [1995], is realised as a language extension to C++ together with a library which abstracts the communication schemes. The data partitioning and distribution is done at the language level. Unfortunately, this does not allow adaptively changing or redistributing the data as needed for finite element computations on adaptively refined meshes.

The TACO (topologies and collections) framework, see Nolte et al. [2000], overcomes this problem. It uses global object pointers underneath and lets the user specify the data distribution at runtime using distributed linked objects. This allows dynamically adding new objects at runtime. Still this means that all parts of a simulation software, e.g. linear algebra and grids, need to either all use TACO directly or at least use the same data distribution. Especially when using third party software components, resembling the data distribution with TACO might be cumbersome.

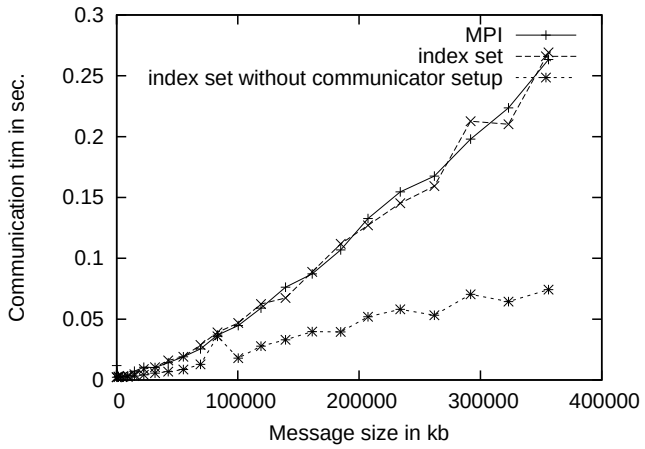


Figure 7.5: Parallel Index Set Performance 3D

The Janus framework, see Gerlach [2002], follows a similar approach to the presented one. The basic abstraction is a mapping of a finite set of distributed objects onto consecutive global indices starting at 0. This abstraction is called a domain. In the parallel case the domain is distributed onto p, the number of processes, mutually disjoint subdomains. In addition each object is mapped onto a local consecutive index starting at 0 on each process. This results in a strided distribution of the collection of objects and according global indices. Adaptively adding new objects calls for renumbering both local and global indices. Furthermore, due to the mutually disjoint distribution, all operations dependent on objects of other subdomains, e.g. sparse matrix vector products, require communication.

In the presented approach, the subdomains need to be augmented to overlapping subdomains according to the data dependencies. This allows for minimising the communication phases. For example, in Krylov solvers not every matrix vector product requires communication using this data distribution. Such overlapping subdomains are either provided by parallel grid managers or directly by the user. As in Janus, each object is identified by a global index. This is mapped to a local index and an attribute used for identifying different partitions of the local domain and to set up communication. Once this mapping is set up, the user can define communication interfaces based on the attributes and perform communications for arbitrary data types. Even if MPI is used underneath, the user is relieved from directly using MPI routines and setting up communication

by hand. Furthermore, one can take advantage of the better performance offered by the presented approach if he has to deal with often recurring communication schemes.

7.3 Algebraic Multigrid Components

In this section we will describe some of the major components developed when we implemented the AMG solvers described in the previous chapters.

7.3.1 Graph components

Our aggregation algorithms use the weighted graph of a matrix to construct the aggregates. As we have shown, this aggregation can be performed on a scalar matrix as well as on a block matrix. In addition, only a subset of the graph is actually used for building the aggregates in the parallel case. The rest of the aggregation information is then added by communicating with other processes. All these requirements can be fulfilled using a flexible design of a graph based on templates and static polymorphism.

Operation	**Effect**
`begin()`	Returns iterator over vertices pointing to first vertex
`end()`	Returns iterator over vertices pointing to the position after the last vertex
`beginEdges(vertex)`	Returns iterator over edges pointing to `vertex` for the first edge
`endEdges(vertex)`	Returns iterator over edges pointing to `vertex` for the position after the last edge
`noVertices()`	Returns the number of vertices
`noEdges()`	Returns the number of edges
`maxVertex()`	Returns vertex with biggest index

Table 7.8: Generic Graph Interface

The interface of all these graphs is described in Table 7.8. It is very slim and basically provides access to iterators over the vertices and edges of the graph.

There are three class templates that implement the graph interface. The `template <class M> class MatrixGraph` class template wraps a matrix of type `M` and provides the graph of it. The

```
template <class G, class VP, class EP, class VM, class EM>
    class PropertiesGraph
```

class template is used to attach weights to a graph of type `G`. The template parameters `VP` and `EP` are the types of the properties attached to the vertices and edges, respectively. The maps of type `VM` and `EM` provide mappings of vertices and Edges to indices, respectively. For the parallel aggregation the `template <class G, class T> class Subgraph` class template provides a subset of a graph of type `G` using the mapping of vertex indices to `bool` for the decision whether a vertex is included or not.

7.3.2 Aggregation and Coarsening

For the sequential aggregation we use an object of type `PropertiesGraph` that wraps a matrix graph of type `MatrixGraph`. In the parallel case, we first create an object of type `SubGraph` of a graph of type `MatrixGraph` according to the disjoint domain partitioning. Then an object of type `PropertiesGraph` is used as wrapper around it to attach the weights to the vertices and edges.

In either case these weights need to be computed based on the strength of connection measure. In all our computations we used a criterion of type `template<class M, class N> class SymmetricDependency` to do this. The template parameter `M` is the type of the matrix used and `N` is the type of the norm that is responsible for computing the scalars from the matrix blocks for the strength of connection computation. Possible values for the latter template parameter are for example `template <int N> Diagonal` that simply uses the N-th diagonal value of the block (e.g. the pressure component) or `RowSum` that computes the row-sum norm of the matrix block.

The information about the aggregates is encapsulated in `template < class V> AggregatesMap`. Its template parameter is the type used to index the vertices. As the name suggests it maps vertices to aggregate numbers and provides iterators over as well as random access to the aggregate numbers. The actual mapping is built by calling its member function

```
template<class M, class G, class C>
```

```
tuple<int,int,int,int> buildAggregates(const M& matrix, G&
    graph, const C& criterion).
```

The template parameter `M` is the matrix type used for the linear system. `G` is type of the matrix graph used. As described at the beginning of this subsection, this differs in the parallel version from the sequential version. The last template parameter is the type that provides the arguments for guiding the aggregation and coarsening process. It provides additional members to the strength of connection criterion. This type is **`template`** `<class T> AggregationCriterion`. The template parameter `T` is the type of the strength of connection criterion used. It is used as the base class of `AggregationCriterion`. All member functions are described in Table 7.9.

7.3.3 Prolongation and Restriction

Both prolongation and restriction are realised in **`template`** `<class V1, class V2, class T>` **`class`** `Transfer`. The first template parameter is the type of the vertex index. The parameter `V2` is type of the vector that is prolongated and restrict. It might be a scalar for variable-based or a block vector for point-based AMG. The last template parameter used is the type of parallel information used. These classes are used to encapsulated the index set based communication as described in Section 7.2. There are specialisations of class `Transfer` for the purely sequential and parallel version. Thus, unnecessary overhead by the communication is avoided if the sequential version is used.

7.3.4 Smoothers

After building the matrix hierarchy, smoothers have to be constructed for each of the matrices on the coarser levels. As we provide many different smoothers ranging from stationary iterative methods to overlapping Schwarz methods, see for example Table 7.4, there is no uniform constructor for them. Still we need a uniform way to construct them.

The only arguments that might be different from level to level are the matrix that the smoother is for and in the parallel version the information about the data decomposition and communication. Therefore we have created the **`template`** `<class S>` **`class`** `ConstructionTraits` class template that constructs the smoothers for us. This class has an associated type called `ConstructionArgs` that provides all the arguments

Operation	Effect
`maxDistance()`	Returns the maximum distance between two aggregate nodes
`setMaxDistance(i)`	Set the maximum distance between two aggregate nodes
`minAggregateSize()`	Returns the minimum number of vertices an aggregate should have
`setMinAggregateSize(i)`	Set the minimum number of vertices an aggregate should have
`maxAggregateSize()`	Returns the maximum number of vertices an aggregate should have
`setMaxAggregateSize(i)`	Set the maximum number of vertices an aggregate should have
`maxConnectivity()`	Returns the maximum number of connection between vertices of an aggregate
`setMaxConnectivity(i)`	Sets the maximum number of connection between vertices of an aggregate
`beta()`	Returns the threshold for deciding whether a vertex is isolated
`setBeta(b)`	Sets the threshold for deciding whether a vertex is isolated
`alpha()`	Returns the threshold for deciding whether a connection is strong
`setAlpha(b)`	Sets the threshold for deciding whether a connection is strong

Table 7.9: `AggregationCriterion` Interface

needed for the construction to the member function `construct(args)`. For each level we just set the matrix of that level in the instance of `ConstructionArgs` and then construct the smoother. This way we can reuse all the other arguments from the previous level.

We use a similar approach for the application of the smoother to the linear system. Some of the non-symmetric smoothers like Gauss-Seidel and the multiplicative version of the overlapping Schwarz method can be applied forward or backward. To use this property, we designed `template < class S> class SmootherApplier`. The smoother can be applied using the member functions `presmooth` and `postsmooth`. In the default case they just call the method `apply` of the preconditioner interface. We pro-

vide template specialisations for the smoothers that can applied forward and backward. These ensure that the forward direction is used for pre- and the backward direction for post-smoothing. Thus, we get a symmetric multigrid method provided that the matrix is symmetric and the same number of pre- and post-smoothing steps are used.

7.3.5 Using AMG

We can use all the classes described in this chapter as building blocks of our AMG method. Depending on what versions we choose, we end up with either a purely sequential algebraic multigrid preconditioner or a parallel one.

We provide sample code for a scalar AMG as a preconditioner in Listing 7.1. The code is well documented and should be self explanatory. We simply construct the type for the scalar sparse matrix and the vector first. Then we choose the type of the coarsening criterion, the type of the smoother, and select the type of the corresponding arguments for the smoother construction. After the types are set up, we create the objects and initial them.

To create a parallel point-based AMG method, we simply use a parallel smoother based on the wrapper `BockPreconditioner` of a sequential one, a parallel Operator of type `OverlappingSchwarzOperator`, a parallel information object of type `OwnerOverlapCopyCommunication`, and a parallel scalar product of type `OverlappingSchwarzScalarProduct`. Everything except the setup of the parallel information and the matrix is the same as for the sequential version. This example can be found in Listing 7.2

```
// Sparse matrix type
typedef Dune::FieldMatrix<double, 1, 1> MatrixBlock;
typedef Dune::BCRSMatrix<MatrixBlock> BCRSMat;
// Vector type
typedef Dune::FieldVector<double, 1> VectorBlock;
typedef Dune::BlockVector<VectorBlock> Vector;
// Sequential operator type
typedef Dune::MatrixAdapter<BCRSMat, Vector, Vector>
   Operator;
// Coarsen criterium type
typedef Dune::Amg::CoarsenCriterion<
  Dune::Amg::SymmetricCriterion<BCRSMat, Dune::Amg::
     FirstDiagonal> >
  Criterion;
//Smoother type and argument type for construction
typedef Dune::SeqSSOR<BCRSMat, Vector, Vector> Smoother;
typedef Dune::Amg::SmootherTraits<Smoother>::Arguments
   SmootherArgs;

BCRSMat mat;
setupMatrix(mat)

Vector b(mat.N()), x(mat.M());
b=0;  x=100;

Operator fop(mat);

SmootherArgs smootherArgs;
smootherArgs.relaxationFactor = 1; // no relaxtion
// max 15 levels and 2000 unknowns on coarsest level
Criterion criterion(15, 2000);

// create scalar product, AMG and solver
Dune::SeqScalarProduct<Vector> sp;
typedef Dune::Amg::AMG<Operator, Vector, Smoother> AMG;
AMG amg(fop, criterion, smootherArgs, 1, 1, 1, false);
Dune::CGSolver<Vector> amgCG(fop, amg, 1e-8, 80, 2);

// solve
Dune::InverseOperatorResult r;
amgCG.apply(x, b, r);
```

Listing 7.1: A sequential scalar AMG example

```
// Type of Block matrix
typedef Dune::FieldMatrix<double, 2, 2> MatrixBlock;
typedef Dune::BCRSMatrix<MatrixBlock> BCRSMat;
// Type of block vector
typedef Dune::FieldVector<double, 2> VectorBlock;
typedef Dune::BlockVector<VectorBlock> Vector;
// Parallel Information and Communication
typedef Dune::OwnerOverlapCopyCommunication<int>
   Communication;
// Parallel operator
typedef Dune::OverlappingSchwarzOperator<BCRSMat, Vector,
   Vector, Communication> Operator;
// Criterion using rowsum norm
typedef Dune::Amg::CoarsenCriterion<
 Dune::Amg::SymmetricCriterion<BCRSMat, Dune::Amg::RowSum>
     > Criterion;
// Sequential and parallel smoother and smoother arguments
typedef Dune::SeqSSOR<BCRSMat, Vector, Vector> Smoother;
typedef Dune::BlockPreconditioner<Vector, Vector,
   Communication, Smoother>
 ParSmoother;
typedef Dune::Amg::SmootherTraits<ParSmoother>::Arguments
   SmootherArgs;

Communication comm(MPI_COMM_WORLD);
BCRSMat mat;
setUpIndexSetsAndMatrix(comm, mat);
Operator fop(mat);
SmootherArgs smootherArgs;
smootherArgs.relaxationFactor = 1;
Criterion criterion(15, 2000);

// Create parallel point-based AMG and solver and apply
Dune::OverlappingSchwarzScalarProduct<Vector,
   Communication> sp(comm);
typedef Dune::Amg::AMG<Operator, Vector, ParSmoother,
   Communication> AMG;
AMG amg(fop, criterion, smootherArgs, 1, 1, 1, false, comm
   );
Dune::CGSolver<Vector> amgCG(fop, sp, amg, 10e-8, 80);
Dune::InverseOperatorResult r;
amgCG.apply(x, b, r);
```

Listing 7.2: A parallel point-based AMG example

8 Summary

In this thesis we have developed a robust and efficient massively parallel algebraic multigrid (AMG) method for linear system arising from the discretization of scalar partial differential equations and systems of them. Among others it is suitable for conforming finite element methods, finite volume methods, and discontinuous Galerkin methods. The AMG preconditioner is especially tailored for diffusion problems with highly oscillating and discontinuous diffusion coefficients. The AMG is massively parallel and scales very well for up to thousands of processes on today's supercomputers.

In particular we have presented a new strength of connection measure for algebraic multigrid methods based on aggregation together with a greedy aggregation algorithm. This approach allows semi-coarsening for anisotropic diffusion problems even when the linear systems stem from bilinear or trilinear finite element discretizations. We have shown that the developed method is a scalable and robust preconditioner for diffusion problems with oscillatory coefficients as well as problems with high-contrast coefficient jumps. In both cases the change in the coefficients is detected and the coarsening is adapted automatically. At the same time the method keeps the sparsity of the coarse level matrices low and therefore the memory consumption of the preconditioner is minimal. The convergence properties for log-normally distributed random fields of diffusion problems are very good. Algebraic domain decomposition two-level methods have been applied to this kind of problems before and turned out to be better than previous AMG approaches. This still holds for the convergence properties when compared to the presented AMG. Despite of this, the scalability of the total CPU time needed to solution of the presented method scales better than the algebraic domain decomposition method. An extension to fully coupled systems of partial differential equation has been developed. The method works well for linear systems from both finite element and finite volume methods. We have shown that the method is applicable to real world problems using sample applications from flow in heterogeneous porous media.

For discontinuous Galerkin methods we have extended the method by

using the space of continuous linear functions as the first coarse level space. Further coarsening is done with the previously developed coarsening by aggregation. We showed that this approach is a robust preconditioner for the symmetric and the non-symmetric interior penalty method with a sufficiently large penalty parameter. For the method of Baumann and Oden we have used overlapping Schwarz methods as smoothers on the fine level. The local subdomains of the smoothers are constructed algebraically using the aggregation algorithm for block matrices. The method proves to be robust for jumping coefficients and high order discretizations. To our best knowledge this is the first AMG preconditioner suitable for application to higher order (p>1) discontinuous Galerkin finite element discretizations. Even lower order DG elliptic problems with such highly discontinuous coefficients have not been tackled with AMG up to now.

A general methodology for parallelising iterative solvers for partial differential equations was developed. Using this abstract approach the "Iterative Solver Template Library" (ISTL) was parallelised. The parallel version of the AMG preconditioner uses the same approach. Using model problems and real world simulations the good scalability of the approach has been shown. The small memory footprint of the sequential method is also an attribute of the parallel method. This allows for solving very large problems on supercomputers with thousands of processors.

The parallelisation approach as well as the AMG method was implemented using advanced C++ programming techniques. Using the block structures of the matrix and vector data structures of ISTL, the implementation is extremely efficient. This holds especially for systems of partial differential equations. Overall, the implementation makes the performance of the solver nearly on a par with commercial AMG solvers based on interpolation for scalar systems. At the same time its memory consumption is far lower making the simulation of larger problems feasible.

The code is publicly available in the Bastian et al. [2005] modules dune-istl (parallel AMG) and dune-udg (AMG for DG) from `http://www.dune-project.org/`.

Index

Glossary

$\langle \cdot, \cdot \rangle$	Euclidian scalar product, 27
$\langle \mathbf{u}, \mathbf{v} \rangle_0$	$:= \langle \mathsf{D}\mathbf{u}, \mathbf{v} \rangle$, 27
$\langle \mathbf{u}, \mathbf{v} \rangle_1$	$:= \langle \mathsf{A}\mathbf{u}, \mathbf{v} \rangle$, 27
$\langle \mathbf{u}, \mathbf{v} \rangle_2$	$:= \langle \mathsf{D}^{-1}\mathsf{A}\mathbf{u}, \mathsf{A}\mathbf{v} \rangle$, 27
$\| \cdot \|_{r,\mathcal{D}}$	The norm on $H^r(\mathcal{D})$., 7
$(\cdot, \cdot)_{\mathcal{D}}$	The inner product on $L^2(\mathcal{D})$ (or $(L^2(\mathcal{D}))^2$), 7
$[\![\phi]\!]$	$\phi_1 - \phi_2$, the jump of a function at an edge, 11
$\{\phi\}$	$\frac{\phi_1+\phi_2}{2}$, the jump of a function at an edge, 11
$\| \cdot \|_{0,\mathcal{D}}$	, 7
$\mathbb{1}_\omega$	Identity on $\mathbb{R}^{I_\omega}$, 89
A	A matrix $\mathsf{A} \in \mathbb{K}^I$, 9
AMG	algebraic multigrid method, 1
$\mathcal{A}(u, v)$	A bilinear form., 8
$\mathbf{b}$	A vector $\mathbf{b} \in \mathbb{K}^I$, 9
$C(\overline{\Omega})$	The space of continuous functions on $\overline{\Omega}$., 10
$C(\overline{\Omega}, \Gamma_D)$	The space of continuous functions on $\overline{\Omega}$ with incorporated Dirichlet boundary conditions on Γ_D., 10
C_A	Operator complexity, that is the sum of the number of nonzeros of the matrices at all levels divided by the number of nonzeros of the matrix at the finest level, 52
DG	discontinuous Galerkin method, 7
$\mathcal{E}$	The set of all edges ($d = 2$) or faces ($d = 3$) of the elements of the mesh $\mathcal{T}$., 11

$\mathcal{E}^B$	Set of all edges at the boundary of the mesh, 11
$\mathcal{E}^I$	Set of all interior edges of the grid, 11
FE	finite element method, 7
FV	finite volume method, 7
GMG	geometric multigrid method, 1
$H_0^1(\mathcal{D})$	The completion of infinitely differentiable functions with compact support under the norm $\|u\|_{1,\mathcal{D}} = \|\nabla u\|_{0,\mathcal{D}}$., 7
$H^m(\mathcal{D})$	The set of functions in $L^2(\mathcal{D})$ with distributional derivatives up to order m also in $L^2(\mathcal{D})$., 7
$H^{-1}(\mathcal{D})$	The dual space of $H_0^1(\mathcal{D})$, 7
$H^{-s}(\mathcal{D})$	, $0 < s < 1$. The space obtained by real interpolation between $H^{-1}(\mathcal{D})$ and $L^2(\mathcal{D})$., 7
$H^s(\mathcal{D})$	, 7
Helics	the Helic II cluster at the Interdisciplinary Centre for Scientific Computing at the university of Heidelberg. The cluster consists of 156 compute nodes. Each node has two dual core AMD Opteron 2220 CPUs with 2.8 Ghz. The nodes are interconnected by a 10G Myrinet network., 100
I	A finite index set, 9
IP	Symmetric interior penalty method, 12
ISTL	Iterative Solver Template Library, 4
Jugene	IBM Blue Gene / System P supercomputer located at Forschungszentrum Jülich, 102
$\mathbb{K}$	A field (e.g. $\mathbb{R}$) or in abuse of notation a vector space of low dimension (e.g. $\mathbb{R}^k$) for treating block matrix with the same notation as scalar matrices., 17

$L^2(\mathcal{D})$	the space of square integrable functions on $\mathcal{D}$, 7
$\mathcal{L}(v)$	A linear form, 8
μ	The penalty parameter for DG methods, 12
NIPG	Non-symmetric interior penalty method, 12
NIPG(μ, p)	Non-symmetric interior penalty method with penalty parameter μ and basis functions of total order p, 75
OBB	DG method of Baumann and Oden, 12
OBB(p)	Baumann's and Oden's method with basis functions of total order p, 75
P_k	$:= \{u : \mathbb{R}^d \to \mathbb{R} \mid u(x) = \sum_{\lvert a\rvert \leq k} c_a x^a,\ c_a \in \mathbb{R}\}$. The space of polynomials on $\mathbb{R}^d$ of maximum total degree k, 10
Q_k	$:= \{u : \mathbb{R}^d \to \mathbb{R} \mid u(x) = \sum_{\lvert a\rvert_\infty \leq k} c_a x^a,\ c_a \in \mathbb{R}\}$. The space of polynomials on $\mathbb{R}^d$ with maximum degree k in each component., 10
residual	$\mathbf{r} = \mathbf{b} - A\mathbf{u}$, 27
SIPG	Symmetric interior penalty method, 12
$\mathcal{T}$	A triangulation of the domain (also called mesh and grid)., 9
$\mathcal{V}_k$	$:= \{v \in L^2(\Omega) : v\lvert_\tau \in P_k, \forall \tau \in \mathcal{T}\}$. the space of discontinuous piecewise polynomial functions of total degree $k \geq 0$, 11

Bibliography

D. M. Alber and L. N. Olson. Parallel coarse-grid selection. *Numer. Linear Algebra Appl.*, 14(8):611–643, 2007. doi: http://dx.doi.org/10.1002/nla.541.

P. F. Antonietti. *Domain Decomposition, Spectral Correctness and Numerical Testing of Discontinuous Galerkin Methods*. PhD thesis, University of Pavia, Italy, 2007.

P. F. Antonietti and B. Ayuso. Multiplicative schwarz methods for discontinous galerkin approximations of elliptic problems. *ESAIM Math. Model. Numer. Anal.*, 42:443–469, 2008.

D. N. Arnold. An interior penalty finite element method with discontinuous elements. *SIAM J. Numer. Anal.*, 19(4):742–760, 1982.

D. N. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini. Unified analysis of discontinuous galerkin methods for elliptic problems. *SIAM J. Numer. Anal.*, 39(5), 2002.

B. Ayuso and L. Zikatanov. Uniformly convergent iterative methods for discontinous galerkin discretizations. *J. Sci. Comput.*, 40(1–3):4–36, 2009.

S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

J. Barton and L. Nackman. *Scientific and Engineering C++*. Addison-Wesley, 1994.

P. Bastian and R. Helmig. Efficient fully-coupled solution techniques for two-phase flow in porous media. Parallel multigrid solution and large scale computations. *awr*, 23:199–216, 1999.

P. Bastian, M. Droske, C. Engwer, R. Klöfkorn, T. Neubauer, M. Ohlberger, and M. Rumpf. Towards a unified framework for scientific computing. In R. Kornhuber, R. Hoppe, J. Périaux, O. W. O. Pironneau, and J. Xu, editors, *Domain Decomposition Methods in Science and Engineering*, volume 40 of *LNCSE*, pages 167–174. Springer-Verlag, 2005.

P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. part ii: implementation and test in dune. *Computing*, 82(2–3):121–138, 2008a.

P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. part i: abstract framework. *Computing*, 82(2–3):103–119, 2008b.

C. E. Baumann and J. T. Oden. A discontinuous hp finite element method for convection-diffusion problems. *Comput. Methods Appl. Mech. Engrg.*, 175(3–4):311–341, 1999.

J. Bey. *Finite-Volumen- und Mehrgitter-Verfahren für elliptische Randwertprobleme*. Advances in Numerical Mathematics. B.G.Teubner, Stuttgart, Leipzig, 1998.

BLAST Forum. Basic linear algebra subprograms technical (BLAST) forum standard, 2001. `http://www.netlib.org/blas/blast-forum/`.

Blitz++. `http://www.oonumerics.org/blitz/`.

D. Braess. Towards algebraic multigrid for elliptic problems of second order. *Computing*, 55:379–393, 1995.

D. Braess. *Finite Elements. Theory, Fast Solvers and Applications in Solid Mechanics*. Cambridge University Press, Cambridge, 1997.

A. Brandt. Algebraic multigrid theory: The symmetric case. *Appl. Math. Comput.*, 19:23–56, 1986.

A. Brandt, S. F. McCormick, and J. W. Ruge. Algebraic multigrid (AMG) for sparse matrix equations. In D. J. Evans, editor, *Sparsity and Its Applications*. Cambridge University Press, Cambridge, 1984.

J. Brannick and L. Zikatanov. Algebraic multigrid methods based on compatible relaxation and energy minimization. In *Domain Decomposition Methods in Science and Engineering XVI*, volume 55 of *Lecture Notes in Computational Science and Engineering*, pages 15–26. Springer, 2007. doi: http://dx.doi.org/10.1007/978-3-540-34469-8.

M. Brezina, A. J. Cleary, R. D. Falgout, V. E. Henson, J. E. Jones, T. A. Manteuffel, S. F. McCormick, and J. W. Ruge. Algebraic multigrid based on element interpolation (amge). *sjsc*, 5(1570–1592), 2000.

M. Brezina, R. Falgout, S. MacLachlan, T. Manteuffel, S. McCormick, and J. Ruge. Adaptive smoothed aggregation (*a*sa) multigrid. *SIAM Review*, 47(2):317–346, 2005.

M. Brezina, R. Falgout, T. Manteuffel, S. MacLachlan, C. McCormick, and J. Ruge. Adaptive algebraic multigrid. *SIAM J. Sci. Comput.*, 27(4): 1261–1286, 2006. ISSN 1064-8275. doi: http://dx.doi.org/10.1137/040614402.

W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial*. SIAM Books, Philadelphia, 2000. Second edition.

P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. North–Holland, Amsterdam, New York, 1978.

T. Clees. *AMG Strategies for PDE Systems with Applications in Industrial Semiconductor Simulation*. PhD thesis, TU Aachen, 2005.

V. A. Dobrev. *Preconditioning of Discontinuous Galerkin Methods for Second order elliptic problems*. PhD thesis, Texas A & M University, 2007.

V. A. Dobrev, R. D. Lazarov, P. S. Vassilevski, and L. T. Zikatanow. Two-level preconditioning of discontinuous galerkin approximations of second-order elliptic equations. *Numer. Linear Algebra Appl.*, 13:753–770, 2006.

J. Dongarra. List of freely available software for linear algebra on the web, 2006. `http://netlib.org/utk/people/JackDongarra/la-sw.html`.

DUNE. `http://www.dune-project.org/`.

E. Elmroth, F, Gustavson, I. Jonsson, and B. Kagström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Rev.*, 46(1):3–45, 2004.

A. Ern, A. F, Stephansen, and P. Zunino. A discontinuous galerkin method with weighted averages for advection-diffussion equations with locally small and anisotropic coefficients. *IMA J. Numer. Anal.*, 29:235–256, 2009.

R. D. Falgout and U. M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In A. M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51, pages 267–294. Springer-Verlag, 2006.

J. Gerlach. *Domain Engineering and Generic Programming for Parallel Scientific Computing*. PhD thesis, TU Berlin, 2002.

W. Giloi, M. Kessler, and A. Schramm. Promoter: A high level, object-parallel programming language. In *Proceedings of the International Conference on High Performance Computing*, New Dehli, India, December 1995.

J. Gopalakrishnan and G. Kanschat. A multilevel discontinuous galerkin method. *Numer. Math.*, 95:527–550, 2003.

D. Gregor and M. Troyer. Boost.MPI. `http://www.boost.org/`, 2006.

M. Griebel, D. Oeltz, and M. A. Schweitzer. An algebraic multigrid method for linear elasticity. *SIAM J. Sci. Comput.*, 25(2):385–407, 2003.

M. Griebel, B. Metsch, and M. A. Schweitzer. Coarse grid classification: AMG on parallel computers. In G. Münster, D. Wolf, and M. Kremer, editors, *NIC Symposium 2008*, volume 39 of *NIC Series*, pages 299–306, February 2008. Also available as SFB 611 preprint No. 368, Universität Bonn, 2008.

L. Grigori, J. W. Demmel, and X. S. Li. Parallel symbolic factorization for sparse LU with static pivoting. *SIAM J. Sci. Comput.*, 29(3):1289–1314, 2007.

A. Gupta. Wsmp: Watson sparse matrix package (part-i: Direct solution of symmetric sparse systems. Technical report, IBM T. J. Watson Research Center, Yorktown Heights, NY, November 2000.

A. Gupta, A. Koric, and T. George. Sparse matrix factorization on massively parallel computers. Technical report, IBM Research Division, 2009.

W. Hackbusch. *Multigrid Methods and Applications*, volume 4 of *Computational Mathematics*. Springer–Verlag, Berlin, 1985.

W. Hackbusch. *Iterative Lösung großer schwach besetzter Gleichungssysteme*. B. G. Teubner Stuttgart, 1993.

J. Hefferson. Linear algebra. in the web, May 2006. `http://joshua.amcvt.edu/`.

P. W. Hemker, W. Hoffmann, and M. H. van Raalte. Two-level fourier analysis of a multigrid approach for discontinuous galerkin discretizations. *SIAM J. Sci. Comput.*, 25(3):1018–1041, 2003.

P. W. Hemker, W. Hoffmann, and M. H. van Raalte. Fourier two-level analysis for higher dimensional discontinuous galerkin discretisation. *Comput. Vis. Sci.*, 7:159–172, 2004.

M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, and K. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Software*, 31, September 2005.

ITL. Iterative template library. `http://www.osl.iu.edu/research/itl/`.

K. Johannsen. Multigrid methods for nonsymmetric interior penalty discontinuous galerkin methods. ICES Report 05-23, University of Texas at Austin, 2005.

J. E. Jones and P. Vassilevski. Amge based on element agglomeration. *sjsc*, 23(1):109–133, 2001.

W. Joubert. Lamg: Los alamos algebraic multigrid code reference manual. Technical Report LA-UR 05-5000, Los Alamos National Laboratory, May 2005.

W. Joubert and J. Cullum. Scalable algebraic multigrid on 3500 processors. *etna*, pages 105–128, 2006.

G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 48(1): 71–95, 1998. ISSN 0743-7315. doi: http://dx.doi.org/10.1006/jpdc.1997.1403.

I. D. Mitchev. *Finite Volume and Finite Volume Element Method for Non-symmetric Problems*. PhD thesis, Texas A&M University, 1996.

MTL. Matrix template library. `http://www.osl.iu.edu/research/mtl/`.

J. Nolte, M. Sato, and Y. Ishikawa. Taco – dynamic distributed collections with templates and topologies. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 1071–1080, London, UK, 2000. Springer-Verlag. ISBN 3-540-67956-1.

J. T. Oden, I. Babuška, and C. E. Baumann. A discontinuous hp finite element method for diffusion problems. *J. Comput. Phys.*, 148(2):491–519, 1998.

A. T. Papadopoulos and H. Tchelepi. Block smoothed AMG preconditioning for oil reservoir simulation systems. Technical report, Oxford University, 2003.

ParMETIS. Available online at `http://www-users.cs.umn.edu/karypis/metis/`.

F. Prill, M. Lukáčová-Medvid'ová, and R. Hartmann. Smoothed aggregation multigrid for the discontinuous Galerkin method. *SIAM J. Sci. Comput.*, 31(5):3503–3528, 2009.

M. Raw. A coupled algebraic multigrid method for the 3d navier-stokes equations. In *Fast Solvers for Flow Problems, Proceedings of the 10th GAMM-Seminar*, volume 49 of *Notes on Numerical Fluid Mechanics*. Vieweg-Verlag, Braunschweig, Wiesbaden, 1985.

A. Rekdal. Permeability upscaling using the DUNE-framework. Master's thesis, Norwegian University of Science and Technology, 2009.

B. Riviere. *Discontinuous Galerkin Methods for Solving Elliptic and Parabolic Equations: Theory and Implementation*, volume 35 of *Frontiers in Mathematics*. SIAM, 2008.

B. M. Rivière and M. F. Wheeler. Improved energy estimates for interior penalty, constrained and discontinuous Galerkin methods for elliptic problems i. *cgs*, 3(3–4):337–360, 1999.

M. Rossi, O. Ippisch, and H. Fühler. Solute dilution under imbition and drainage conditions in a heterogeneous structure: Modelling of a sand tank experiment. *Adv. Water Res.*, 31:1242–1252, 2008.

J. Ruge and K. Stüben. Algebraic multigrid. In S. F. McCormick, editor, *Multigrid Methods*, chapter 4, pages 73–130. SIAM Philadelphia, 1987.

R. Scheichl and E. Vainikko. Additive schwarz with aggregation-based coarsening for elliptic problems with highly variable coefficients. *Computing*, 80(4):319–343, 2007.

J. Siek and A. Lumsdaine. A modern framework for portable high-performance numerical linear algebra. In H. Langtangen, A. Bruaset, and E. Quak, editors, *Advances in Software Tools for Scientific Computing*, volume 10 of *LNCSE*, pages 1–56. Springer-Verlag, 2000.

B. Smith, P. Bjorstad, and W. Gropp. *Domain Decompostion: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

H. D. Sterck, R. D. Falgout, J. W. Nolting, and U. M. Yang. Distance-two interpolation for parallel algebraic multigrid. *Numer. Linear Algebra Appl.*, 15:115–139, 2008. doi: http://dx.doi.org/10.1002/nla.559.

B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

K. Stüben. Algebraic multigrid (AMG): An introduction with applications. Technical Report 70, GMD - Forschungszentrum Informationstechnik GmbH, September 1999.

A. Toselli and O. B. Widlund. *Domain Decomposition Methods - Algorithms and Theory*. Springer-Verlag Berlin Heidelberg, 2005.

R. S. Tuminaro and C. Tong. Parallel smoothed aggregation multigrid: Aggregation strategies on massively parallel machines. In J. Donnelley, editor, *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7803-9802-5. doi: http://dx.doi.org/10.1109/SC.2000.10008.

P. Vaněk. Acceleration of convergence of a two level algorithm by smoothing transfer operators. *Appl. Math.*, 37:265–274, 1992.

P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid based on smoothed aggregation for second and fourth order problems. *Computing*, 56:179–196, 1996a.

P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. In N. D. Melson, T. A. Manteuffel, S. F. McCormick, and C. C. Douglas, editors, *Seventh Copper Mountain Conference on Multigrid Methods*, volume CP 3339, pages 721–735, Hampton, VA, 1996b. NASA.

T. Veldhuizen. Techniques for scientific C++. Technical report, Indiana University, 1999. Computer Science Department.

H.-J. Vogel, I. Cousin, O. Ippisch, and P.Bastian. The dominant role of structure for solute transport in soil: experimental evidence and modelling of structure and transport in a field experiment. *Hydrol. Earth. Syst. Sci.*, 10:495–506, 2006.

R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics Conference Series*, 16: 521–530, Jan. 2005.

W. L. Wang. Interface preserving coarsening multigrid for elliptic problems with highly discontinuous coefficients. *Numer. Linear Algebra Appl.*, 7: 727–741, 2000.

M. F. Wheeler. An elliptic collocation-finite element method with interior penalties. *SIAM J. Numer. Anal.*, 15(1):152–161, 1978.

U. M. Yang. On long range interpolation operators for aggressive coarsening. *Numer. Linear Algebra Appl.*, 2009. doi: http://dx.doi.org/10.1002/nla.689.