TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik
Scientific Computing

Hybrid Geometric-Algebraic Matrix-Free Multigrid on Spacetrees

Marion Weinzierl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Johann Schlichter
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Hans-Joachim Bungartz
	2. Prof. Irad Yavneh, Ph.D., Technion – Israel Institute of Technology, Haifa, Israel
	3. Univ.-Prof. Dr. Miriam Mehl

Die Dissertation wurde am 11. April 2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 5. Juli 2013 angenommen.

Abstract

Linear solvers are the motor for many computer simulations that are based on partial differential equations (PDEs). For a wide range of problems, multigrid solvers belong to the most efficient ones. Their convergence rate is mostly independent of the mesh size of the underlying problem discretisation, and thus they have optimal complexity.

During the last two decades, there has been a strong focus on algebraic multigrid, which can easily employ accurate unstructured grids and is very robust. But increased accuracy requirements, complex models, and huge amounts of data from engineering, scientific or, e.g., medical applications require the use of supercomputers. Their architecture forces researchers to rethink their algorithms and data handling. For example, algebraic multigrid suffers from a serious performance decrease on parallel architectures, due to high setup costs and communication overhead, unstructured data access, indirect addressing, and a large memory footprint. Therefore, the less robust geometric multigrid is now reconsidered. For the data, spacetrees have turned out to not only provide fast data access but also an efficient structure for performing the computations.

This work combines the advantages of geometric and algebraic multigrid. It defines the solver on a geometrically coarsened structured grid, but instead of geometric multigrid operations the much more robust BoxMG by Dendy using operator-dependent intergrid transfer operators and Petrov-Galerkin coarse-grid operators is used. The solver is implemented on a spacetree as underlying data- and computational structure, ensuring efficient data handling, data locality, and low communication overhead. It is integrated into and parallelised in the PDE solver framework Peano, which has a small memory footprint and is very memory-efficient. This results in a robust solver that is tailored to high performance computers.

Zusammenfassung

Lineare Löser sind der Motor für viele Computersimulationen, die auf Partiellen Differentialgleichungen (Partial Differential Equations, PDEs) basieren. Für eine Vielzahl von Problemen gehören Mehrgitterlöser zu den effizientesten Lösern. Ihre Konvergenzrate ist meist unabhängig von der Gitterweite der zugrundeliegenden Problem-Diskretisierung. In diesem Sinne haben sie optimale Komplexität.

Während der letzten beiden Jahrzehnte bestand eine starke Fokussierung auf algebraisches Mehrgitter, das problemlos mit unstrukturierten Gitter umgehen kann und sehr robust ist. Jedoch erfordern erhöhte Genauigkeitsanforderungen, komplexe Modelle und riesige Datenmengen aus Ingenieurs-, Wissenschafts- oder Medizinanwendungen die Verwendung von Großrechnern. Deren Architektur führt dazu, dass Wissenschaftler ihre Algorithmen und die Datenverwaltung und - verarbeitung überdenken müssen. Algebraisches Mehrgitter, z.B., erfährt auf parallelen Architekturen einen ernsthaften Leistungseinbruch. Dieser liegt in hohen Setup-Kosten, Zusatzkosten für die Kommunikation, unstrukturiertem Datenzugriff, indirekter Adressierung und einem hohen Speicherverbrauch begründet. Infolgedessen wird seit einiger Zeit das weniger robuste geometrische Mehrgitter wieder in Betracht gezogen. Als Datenstruktur haben sich Spacetrees bewährt, da sie nicht nur schnellen Datenzugriff gewährleisten, sondern auch eine effiziente Rechenstruktur darstellen.

Diese Arbeit kombiniert die Vorteile von geometrischem und algebraischem Mehrgitter. Sie definiert den Löser auf einem geometrisch vergröberten Gitter, anstelle von geometrischen Mehrgitteroperationen wird jedoch das viel robustere BoxMG von Dendy, welches operator-abhängige Intergrid-Transfer-Operatoren und Petrov-Galerkin-Grobgitteroperatoren verwendet, eingesetzt. Der Löser wird auf einem Spacetree als zugrundeliegende Daten- und Rechenstruktur implementiert, wodurch eine effiziente Datenverarbeitung, Datenlokalität und geringe Kommunikationskosten gewährleistet sind. Er wird in das PDE-Löser-Framework Peano, das einen geringen Speicherverbrauch hat und sehr speichereffizient ist, integriert und darin parallelisiert. Dadurch erhalten wir einen robusten Löser, der die Anforderungen von Hochleistungsrechnern erfüllt.

Danksagung

An dieser Stelle möchte ich mich bei den Leuten bedanken, die mich während der Zeit meiner Promotion begleitet und unterstützt haben.

Meinen Prüfern Hans-Joachim Bungartz, Irad Yavneh und Miriam Mehl danke ich für die Übernahme dieses Amtes. Hans danke ich inbesondere für seine Funktion als mein Doktorvater und die Anregungen und Anmerkungen zu dieser Arbeit, außerdem für seine Unterstützung während meines Mutterschutzes und der Elternzeit. Vielen Dank an Miriam für die Diskussionen und die ausführlichen Hinweise und Verbesserungsvorschläge zu meiner Arbeit. Irad danke ich für all das, was ich von ihm über Mehrgitteralgorithmen lernen durfte, für sein Engagement und die Zeit, die er sich für unsere Zusammenarbeit genommen hat, für seine Anleitungen und seine Hinführung zum BoxMG-Algorithmus, der nun einen Grundpfeiler dieser Arbeit darstellt. Außerdem danke ich ihm, seiner Familie und seiner Forschungsgruppe (hier insbesondere Eran Treister) für die unglaubliche Gastfreundschaft während meines Forschungsaufenthalts in Haifa.

Meinen Kollegen Christoph Riesinger und Tobias Weinzierl danke ich für das Korrekturlesen meiner Arbeit und ihre Anmerkungen. Christoph verdient zusätzlich besonderen Dank für die Zusammenarbeit im Nicht-Forschungsbereich: Ohne ihn als fleißigen und engagierten Koordinationskollegen für den CSE-Studiengang wäre meine Zeit für diese Arbeit noch knapper gewesen. Auch Christa Halfar war hier eine wichtige Stütze. Tobias danke ich zudem für seine Ausführungen und Erklärungen zum Peano-Framework und dessen Besonderheiten und Tücken (auch wenn er letztere nicht unbedingt als solche sieht), die Diskussionen, insbesondere zur Parallelisierung, und die Unterstützung bei der Portierung meines Codes von der alten auf die neue Peano-Version.

Auch meinen anderen Kollegen möchte ich danken, da sie das Umfeld für meine Promotionszeit bestimmt haben. Während meines Auslandsaufenthalts habe ich gemerkt, dass mir dieser bunte und manchmal anstregende Haufen doch gefehlt hat – erst aus der Entfernung merkt man manchmal, wie einem eine solche Gruppe Halt gibt.

Meinem Mann Tobias Weinzierl danke ich für all seine Unterstützung während der letzen Jahre. Ich danke ihm dafür, dass er mit mir die Elternzeit und die Betreuung unserer wunderbaren und anspruchsvollen Tochter teilt und mir insbesondere in den letzten Monaten der Promotion häufig den Rücken freigehalten hat. Ich danke ihm für seine Geduld und sein Verständnis, für seine Fürsorglichkeit, seine

Ermunterungen und dafür, dass er für mich da ist.

Schließlich danke ich meiner Familie und meiner Schwiegerfamilie für ihren Zuspruch und ihre Auf- und Ermunterungen während meiner Promotionszeit.

Contents

1 Introduction **1**
- 1.1 Motivation and Objective . . . 1
- 1.2 Related Work . . . 4
- 1.3 Thesis Structure . . . 5

2 Multigrid Methods **7**
- 2.1 The Multigrid Principle . . . 7
- 2.2 Appearances of Multigrid . . . 15
- 2.3 BoxMG . . . 20

3 Spacetrees as Data and Computational Structure **25**
- 3.1 Spacetrees and Spacetree Grids . . . 25
- 3.2 Spacetrees as Computational Structure Using Space-Filling Curves . . 26

4 Geometric-Algebraic Multigrid on Spacetrees – Algorithm Prototyping **31**
- 4.1 Requirements . . . 31
- 4.2 Nonsymmetric BoxMG With Coarsening by Three . . . 32
- 4.3 Element-by-Element Multigrid and BoxMG . . . 43

5 Geometric-Algebraic Multigrid on Spacetrees – Target Implementation **65**
- 5.1 Peano Framework . . . 65
- 5.2 Sequential Hybrid Multigrid in the Peano Framework . . . 68
- 5.3 Parallelisation . . . 76

6 Conclusion and Outlook **95**

1 Introduction

1.1 Motivation and Objective

Computer simulations of blood flow in the vessels, tsunamis approaching a coast line, the heating process of a piece of metal, supernovae and galaxies have one thing in common: The underlying processes are mostly modelled by partial differential equations (PDEs). But the model alone is not enough for a simulation. In order to make stars and planets move and blood and water flow, we need solvers for the PDEs. They are in this sense the "motor" of the simulation.

There are two important critical point in today's scientific computing and computational science world: Memory and computational complexity. On the one hand the – for example scientific, engineering, or medical – problems that are aimed to be solved on computers are becoming larger and larger, increasing the amount of data to be handled and thus the need for supercomputers. On the other hand, supercomputers have only a restricted memory capacity per core, and memory access is expensive. Hence, effort has to be taken in order to organise the data in such a way that the inter-core communication is kept low and the memory access is as efficient as possible. In addition, the methods used for solving a problem have to be scalable in terms of numbers of iterations needed to solve a problem with a large number of unknowns.

These issues are directly connected to energy consumption. We cannot simply build bigger and faster computers and thus getting more and more efficient. The heat produced by the processors (and the therefore required cooling facilities, which also need a lot of energy) and the energy needed for the computations and – even more – for the memory operations limit the capacities of nowadays computer architectures. In order to overcome this, the architectures will have to be completely redesigned [64]. Hardware specialists are working on architectures with lower energy consumption and intelligent cooling systems. The "Green500 list" [40, 84] provides a ranking of the 500 most energy-efficient supercomputers – the awareness for the energy issue increases. However, changing the architectures and with that improving their energy efficiency is not the only working point for this issue. We also have to work on the way we use parallel computers.

"It is widely recognized [...] that emerging constraints on energy consumption will have pervasive effects on HPC; power and energy consumption must now be added to the traditional goals of algorithm design, viz. correctness and performance." is

stated in the 2010 report of the Advanced Scientific Computing Advisory Committee (ASCAC) of the U.S. Department of Energy [2]. And: *"[...] memory will become the rate-limiting factor along the path to exascale, and investments should accordingly be made in designing algorithms with reduced memory requirements. Examples where this work is appropriate include [...] algorithmically scalable matrix-free methods (e.g., multigrid) for sparse systems of equations [...]"* [2].

In her keynote speech at the 36th International Symposium on Computer Architecture (ISCA) in 2009, Yelick pointed out "Ten Ways to Waste a Parallel Computer". Two of them were *"Run bad algorithms"* and *"Don't rethink your algorithms"* [110].

In order to reach for new insights and develop new technologies we need supercomputers. But we should not "waste parallel computers". Instead, we should carefully design algorithms that are efficient on these special architectures. Parallelisation does not only mean: Rewrite your code using MPI, OpenMP or something like that. This would not exploit the capabilities of supercomputers. Additionally, one has to work on the algorithms themselves, and make use of existing efficient algorithms that fit to the computer architecture that is available.

Tuning algorithms to specific problems, geometries, and architectures is a task that scientific computing specialists might be willing to take. Application specialists, and therefore the users of the software packages containing the algorithms, however, demand software which is usable as a black box and solves not only a single problem efficiently, but is robust for a large class of problems.

This thesis focuses on the design, assembling and implementation of multigrid methods in order to obtain robust algorithms that run efficiently on high performance computers. We concentrate on solving the convection-diffusion equation and the diffusion equation with variational coefficients as model problems. Of course, the algorithms are also applicable to other problems.

For a wide range of simulation problems, multigrid (MG) solvers belong to the most efficient solvers of the underlying linear or even non-linear systems. Their convergence rate is mostly independent of the mesh size of the problem discretisation, and thus they have optimal complexity in terms of number of operations per unknown.

There are two main groups of multigrid solvers: While geometric MG methods, which are the original MG methods and usually use structured grids, define the coarse grid and the intergrid transfer operators in a purely geometric manner, algebraic multigrid (AMG) does not need to know about the geometry at all, and usually only considers the system matrix of the PDE. AMG can easily use very accurate, unstructured meshes for the discretisation and is robust for a wider range of problems.

Currently, a shift is happening in the research community for multigrid solvers. After having concentrated research mostly on algebraic multigrid and unstructured problems since more than twenty years ago, for some years now structured and semi-

structured approaches have been re-considered, and this trend is increasing (see, e.g., [47, 13, 66]). The reason lies in modern high performance computing architectures: Unstructured data access and high communication overhead, indirect addressing, high setup costs and a large memory footprint decrease the performance of AMG solvers on supercomputers significantly. Therefore, the gain in using an exact unstructured discretisation and algebraic multigrid methods vanishes as compared to solvers working with structured data resolution.

Our objective is to design an algorithm that combines the advantages of both approaches. It should be robust and efficient for a large range of applications, and at the same time have good parallelisation properties. The algorithm shall be designed in a way that it can use spacetrees (a generalisation of quadtrees/octrees) both as data structure and as computational structure, in order be able to handle huge amounts of data in an efficient way, and support adaptive grids. It shall be possible to integrate the algorithm in a state-of-the-art spacetree framework, making use of memory-efficient data storage and traversal properties and parallelisation facilities.

In order to achieve these goals, we chose a way in-between the pure geometric and the pure algebraic multigrid: the BoxMG method by Dendy, which is defined on geometrically coarsened, structured grids, but uses operator-dependent intergrid transfer operators and Petrov-Galerkin coarse-grid operators, making it much more robust than geometric multigrid. We shall test and improve the efficiency and robustness of the BoxMG solver on a multigrid hierarchy that uses coarsening by a factor of three, as this coarsening factor is required by our target framework. For supporting a spacetree implementation, we develop a solver that applies and stores both the system operator and the intergrid transfer operators in a matrix-free element-wise manner, ensuring data locality and low communication overhead on parallel computers. An additional challenge that comes along with this approach is finding a suitable smoother: On spacetrees, damped Jacobi is usually the smoother of choice, as other smoothers cannot easily cope with the locality paradigm. However, for many applications it does not yield satisfying efficiency and degrades the robustness of a multigrid smoother on a structured grid. Therefore, more powerful alternatives for damped Jacobi shall be tested in this thesis. Finally, we shall integrate our solver into the target framework and test its parallel behaviour.

With this approach we take a wholistic view on the solver software. While keeping the main focus on the efficient PDE solver, we take into account the various additional requisites that are posed to the software: efficiency in the data administration, in the data traversal scheme, and sequential as well as parallel performance. All these demands determine the design of our algorithms.

1.2 Related Work

Matrix-free parallel PDE solvers based on spacetree/octree grids are currently an active field of research. The following listing is not complete. We present here some examples of recent work in order to demonstrate the state-of-the-art in this field and to show were the contributions of this thesis lie.

The software framework Peano [100] for parallel adaptive PDE solvers on spacetrees is based on work of Günther [48], Pögl [77] and Krahnke [65]. In its current form it was mainly developed by T. Weinzierl [99, 101]. The underlying idea of using a special data structure (here: stacks) for a memory-efficient parallel PDE solver on spacetrees is similar to the hash map approach introduced by Griebel and Zumbusch some years earlier [47] but goes further in optimising data locality and cache-usage. In [99], Weinzierl presents results for a matrix-free parallel geometric multigrid solver in the Peano framework. That work was the motivation and starting point for developing a multigrid solver that does not suffer from the robustness shortcomings of geometric multigrid solvers and at the same time has the parallelisation advantages of a structured-grid approach. The achieved algorithm is integrated into the Peano framework, and we use Peano's spacetree structure and parallelisation environment in order to test the parallel behaviour.

With the geometric multigrid library Dendro [29], Sampath and Biros as well present a matrix-free parallel geometric multigrid method on octrees [81]. They also use structured grids, but apply a Petrov-Galerkin coarse grid operator instead of re-discretising the operator on the coarse grid. Their method was later integrated into the p4est framework (Burstedde et al. [26]), which uses a "forest of octrees", i.e., a collection of octrees, where each octree represents an adaptive structured grid, but the overall grid is unstructured. In that work [91], a geometric multigrid solver is used for the octrees, with an AMG solver on the coarsest grid. Parallel results are given for a conjugate gradient solver preconditioned by the multigrid solver.

Another parallel matrix-free geometric multigrid preconditioner was presented recently by Flaig and Arbenz [41]. The idea of combining structured and unstructured grids is also implemented in the Hierarchical Hybrid Grid (HHG) solver [14], which uses regular structured grids in each cell of an unstructured overall grid. Here as well, parallel results are presented for geometric multigrid, e.g. in [13, 44]. As software framework providing libraries for matrix-free PDE solving we want to mention deal.ii [11, 10], which offers a geometric multigrid implementation, too.

Although we present results only for the integration of our algorithm into Peano, the integration into frameworks such as p4est or HHG as solver on the structured meshes is straightforward, and we would expect that also here a considerable improvement of the robustness and efficiency can be achieved.

Improved robustness for parallel multigrid methods was also the aim of the work of Bader [5]. He uses recursive substructuring [82, 60] as domain decomposition

method, which likewise results in a tree structure, and combines geometric multigrid with the Petrov-Galerkin coarse grid operator. In order to obtain a coarse grid system that is able to cope with strongly convection-dominated flows, he introduces additional unknowns at the coarse cell boundaries. This extension leads to an optimal coarse grid for this special type of problems.

To summarise, the multigrid method presented in this thesis combines and extends the following features, which are only partly covered by the approaches described above: First, we use **operator-dependent restriction and prolongation** as defined by Dendys BoxMG method. This improves the robustness of the solver compared to pure geometric multigrid solvers, for example for problems with jumping diffusion coefficients. Second, and this is similar to some of the cited methods, we develop a **spacetree-based**, **matrix-free** method that stays in a **strictly local**, **cell-wise** diction and defines not only the system operator, but also restriction and prolongation (for arbitrary prolongation and restriction operators) in such a way. The Petrov-Galerkin coarse-grid operator is computed by a product of these local cell operators. Third, we develop and test variants of **smoothers** that are applicable on spacetrees and at the same time powerful enough to handle challenging problems. And fourth, we integrate and parallelise our method in a framework that implements the spacetree traversal in a **memory-efficient** way and provides an interface for **parallelisation**.

Further references for the topics handled in this thesis will be given in the respective chapters.

1.3 Thesis Structure

This thesis is structured as follows: In Chap. 2, we introduce the basics of multigrid methods and describe the methods that we use in more detail, especially Dendys BoxMG. Chap. 3 gives a brief overview about spacetrees and spacetree grids, and how they can be used as data and computational structure for simulations. We also discuss the restrictions, benefits and special requirements of a spacetree-based multigrid implementation. The design and prototypical testing in Matlab of the components of our multigrid algorithm is described in Chap. 4. Two prototypes are developed: The first implements the BoxMG method for coarsening by a factor of three (as required by the target framework Peano) for nonsymmetric problems. A finite-differences discretisation that can easily switch between first- and second-order is introduced, and we report on the numerical robustness and efficiency of the method for the convection-diffusion equation, also for the challenging recirculating flow problem. The second prototype already implements the multigrid operators in the way required by the spacetree, but not the spacetree itself. The intergrid transfer operators are defined in a way that their evaluation is a matrix-vector product, just

as for the system operator. This yields an elegant formulation of the computation of the Petrov-Galerkin coarse grid operator. Different smoothers are examined in order to find a powerful relaxation that can be implemented efficiently on a spacetree. Here, also a variant of the hybrid smoothers used in parallel computing is introduced. The principles and methods developed with the second prototype can be directly integrated into the Peano framework. We introduce the main principles of this framework and the parallelisation of our algorithms in Chap. 5. Results of the target implementation and parallelisation results for shared memory as well as distributed memory are presented. Chap. 6 summarises the outcome of the thesis and its conclusions, and gives an outlook to possible extensions and questions raised by this work.

2 Multigrid Methods

Multigrid methods were introduced in the 1960s by Fedorenko and Bakhvalov [38, 39, 9] and further developed in the 1970s by Brandt [16, 17] and Hackbusch [49, 50] independently from each other. In the following decades, multigrid methods have been extended and improved by different groups. Some of the first books and monographs on multigrid methods which appeared in the 1980s are [90, 19, 51].

In this chapter we briefly introduce the most important multigrid principles and methods. A comprehensive guide to multigrid methods with a lot of useful references is [95]. As a beginners tutorial we would recommend [24]. [108] is a short paper that introduces the main idea and concepts of multigrid in a very comprehensible way.

2.1 The Multigrid Principle

Multigrid methods provide fast numerical solvers for discretised linear (especially elliptic) and even nonlinear partial differential equations (PDEs). In this work, we consider partial differential equations $Au = f$ on a two-dimensional domain $\Omega \in \mathbb{R}^2$ and use a regular Cartesian grid for discretisation.[1] We call A our system operator, f the right-hand side, and u the unknown. The discretised equation is written as $A^h u^h = f^h$, with mesh width $h = 1/(n-1)$ and n the number of discretisation points. Here, A^h is a matrix, and u^h and f^h are vectors. The discretised domain, i.e. the grid, is denoted as Ω^h.

The basic idea of multigrid methods is the following (see, e.g., [95]): Using a standard relaxation/smoothing method such as Jacobi or Gauss-Seidel (see Sec. 2.1.1) as a solver for the problem given by a discretised elliptic PDE on Ω^h, we observe that the error $e^h = u^h - \tilde{u}^h$ (with u^h being the exact solution to the discretised problem and $\tilde{u}^h$ the current approximation to u^h) becomes smooth after some relaxation steps, but its maximum and absolute value only slowly decrease. In terms of error frequencies this means: High-frequency error components are eliminated, but low-frequent components remain (*smoothing property*). Yavneh referred to this effect in his paper [108] as *"Relaxation [...] irons out the wrinkles but leaves the fat."* The remaining smooth error can be respresented well on a coarser grid, e.g., $\Omega^H = \Omega^{2h}$ (*approximation property*). In addition, it appears "wrinkled" again on

[1] The fundamentals, though, are also applicable to $\Omega \in \mathbb{R}^d$ with higher ($d > 2$) or lower ($d = 1$) dimension, and multigrid is also applicable on, e.g., triangular or unstructured grids.

the coarse grid as the term “wrinkled” (i.e., high-frequency) has to be seen relative to the mesh width. Thus, further parts of the error can be reduced efficiently (and for a lower cost, as we have less grid points) by relaxation on the coarse grid. This way we can descent to coarser and coarser grids, e.g. Ω^{4h}, Ω^{8h},..., Ω^1 (with Ω^1 containing only one grid point, for example), and step-by-step eliminate the error components of all frequencies.

In the remainder, we always talk about discretised equations and omit, for better readability, the superscript h, except where we use it to distinguish between the grids in the multigrid hierarchy.

How to realise the idea described above in practise? For knowing the error of our solution guess we have to know the solution – and then our problem would already be solved. Additionally, the error becomes smooth, not the solution itself, therefore the solution might not be well approximated on the coarse grid. What equation can we solve on the coarse grid that will help us reducing the error without knowing it? The residual, which is defined as $r = f - A\tilde{u}$, is for linear problems directly related to the error by A (*residual equation*):

$$Ae = r, \tag{2.1}$$

as $Ae = A(u - \tilde{u}) = Au - A\tilde{u} = f - A\tilde{u} = r$ if A is linear. Therefore, the approximation property is also fulfilled for the residual. Instead of applying relaxation to $Au = f$ we can equivalently relax the residual equation with initial guess $e = 0$. By doing this on the coarse grid, we receive an approximation $\tilde{e}$ of the error which we can use to correct the fine grid equation (*coarse grid correction*). In order to use the residual from the fine grid for the residual equation on the coarse grid and the error approximation from the coarse grid for correcting the fine grid solution, we have to define appropriate intergrid transfer operators for restriction from the fine grid to the coarse grid (denoted as operator R) and prolongation from the coarse grid to the fine grid (denoted as operator P). Possible choices for restricton and prolongation operators are discussed in Sec. 2.1.2.

This principal idea can be applied recursively, and we obtain the multigrid V-cycle as given in Alg. 2.1 (see also Fig. 2.1a). A V-cycle with ν_{pre} pre-relaxation steps (i.e., smoothing steps before restriction) and ν_{post} post-relaxation steps (i.e., smoothing steps after prolongation) is referred to as V(ν_{pre}, ν_{post}) cycle.

The four multigrid operations *relaxation*, *restriction*, *prolongation* and *solving* can be arranged and recursively nested in various ways in order to obtain multigrid cycles with different convergence behaviour. Fig. 2.1 shows four multigrid schemes that are frequently used in praxis. The FMG-cycle is something special here: We do not only use the correction scheme, but in addition we get ourselves a good initial guess by solving the coarse grid problem and prolong its solution to the next finer level. Then we do a V-cycle here, prolong the solution up again, and so forth, until we do one last V-cycle starting from the finest level. This is a quite effective scheme. In

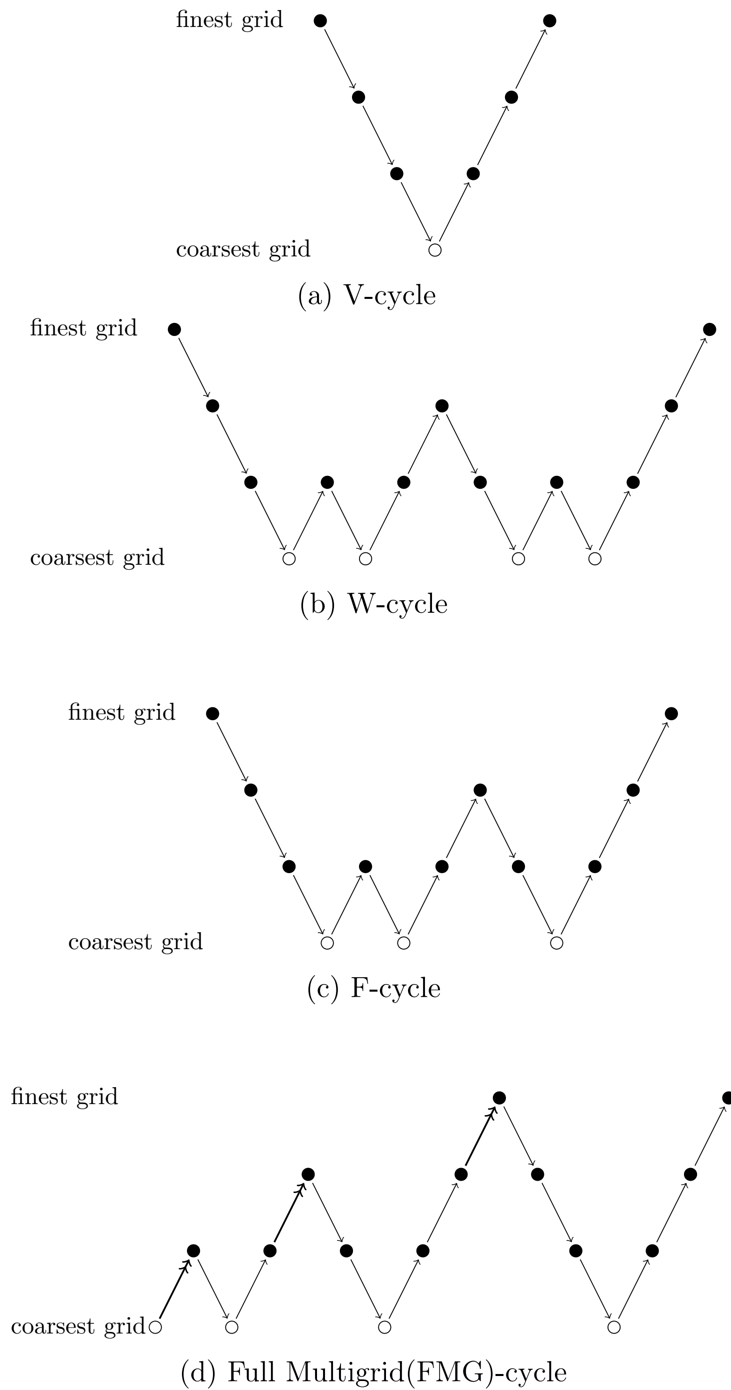


Figure 2.1: Standard multigrid cycles on four grid levels. Filled circles stand for smoothing, empty circles for solving the coarse grid problem; arrows pointing downwards are for restriction, arrows pointing upwards for prolongation. Double-headed arrows in the FMG-cycle stand for prolongation of the approximation to the solution.

Algorithm 2.1 Multigrid V-Cycle for the Correction Scheme

function V-Cycle(A^h, $\tilde{u}^h$, f^h, ν_{pre}, ν_{post})
  if $\Omega_h == \Omega_1$ **then**
    Solve $A^h\tilde{u}^h = f^h$.
  else
    Relax $A^h\tilde{u}^h = f^h$ ν_{pre} times.
    Compute fine grid residual: $r^h \leftarrow f^h - A^h\tilde{u}_h$.
    Restrict the residual to the next coarser level: $r^H \leftarrow Rr^h$.
    Solve residual equation on the coarse grid by recursion:
    $\tilde{e}^H \leftarrow$ V-CYCLE(A^H, 0, r^H, ν_{pre}, ν_{post}).
    Correct the current approximation on the fine grid: $\tilde{u}^h \leftarrow \tilde{u}^h + P\tilde{e}^H$.
    Relax $A^h\tilde{u}_h = f^h$ ν_{post} times.
  end if
  return $\tilde{u}^h$.
end function

order to choose the best scheme for a given problem, one has to look at the trade-off between benefit (in terms of efficiency and robustness) and effort (in terms of number of operations needed until convergence).

In the following sections we will have a brief look at the above mentioned multigrid components (relaxation, intergrid transfer operators, coarse grid problem) and their possible realisations.

2.1.1 Relaxation

The choice of the relaxation (or smoothing) method is a crucial decision for a (geometric) multigrid solver. The two most common methods are Gauss-Seidel and (damped) Jacobi iteration. Both use the idea of eliminating the residual in every grid point k and in iteration step i by applying the correction:

$$u_k^{(i+1)} := u_k^{(i)} + \frac{1}{a_{kk}} \cdot r_k^{(i)},$$

with a_{kk} the diagonal element of A in line k, and $r_k = f_k - A_k u_k$ the respective residual. The difference is that for the Gauss-Seidel method the residual is updated immediately with the new $u_k^{(i+1)}$:

$$r_k^{(i)} = f_k - \sum_{j=1}^{k-1} a_{kj} u_j^{(i+1)} - \sum_{j=k}^{n} a_{kj} u_j^{(i)},$$

whereas in the Jacobi method the residual is computed in the beginning and not updated during the iteration:

$$r_k^{(i)} = f_k - \sum_{j=1}^{n} a_{kj} u_j^{(i)}.$$

Written in matrix form, the iteration reads:

$$u^{(i+1)} = u^{(i)} + M^{-1} r^{(i)}$$

with $M = D_A$ (D_A = diagonal matrix of A) for Jacobi and $M = L_A + D_A$ (L_A = lower triangular matrix of A) for Gauss-Seidel.

Gauss-Seidel iterations will usually converge faster, as they use the updated information immediately. On the other hand, for Jacobi iterations the update at one grid point does not depend on the others, what makes it easy to parallelise. [2]

Depending on the start error, Jacobi tends to overshoot in the correction: A negative error can become positive instead of small. This can often be avoided by introducing a damping factor $\omega < 1.0$ for the correction:

$$u_k^{(i+1)} := u_k^{(i)} + \omega \frac{1}{a_{kk}} \cdot r_k^{(i)}.$$

For Gauss-Seidel the opposite is true: The correction is often too small, such that an overrelaxation by a factor $\omega > 1.0$ improves the convergence. Gauss-Seidel with overrelaxation is called successive overrelaxation (SOR). An important note is that good solvers are not necessarily good smoothers: For multigrid, it is important that the high error frequencies are eliminated quickly, not the overall error (i.e., the error becomes smooth but not necessarily small). High frequencies are, in this context, those which cannot be represented correctly on the next coarser grid. The optimal damping or overrelaxation parameter and the smoothing factor (i.e., the worst factor by which high frequency error components are reduced per relaxation step [95]) can be determined by looking at the different frequencies in relation to the mesh width h (*smoothing analysis*) and the coarsening factor. We do not explain the analysis here but refer to the literature, e.g. [95] and references therein.

There are many variants even in these two basic smoothers. For example, instead of the lexicographic order of traversing the grid, red-black Gauss-Seidel uses a checkerboard pattern: First, all “red” grid points are processed, then all “black” grid points. This results in a different behaviour with improved convergence for certain types of problems. In addition, this scheme is easy to parallelise. The update of all “red” grid points (and then the “black” ones, resp.) can be done in parallel, as they do not depend on each other.

[2] In Sec. 4.3.4 we will see that there are circumstances in which Jacobi is much cheaper to implement than Gauss-Seidel.

Another variant are line smoothers. Here, the residual of a whole grid line is eliminated at once. The convergence is thus improved, but of course this is more expensive than point-wise smoothing, as a whole tridiagonal system of equations has to be solved. The line equivalent to red-black Gauss-Seidel is also called "zebra" line relaxation. Line smoothers can be regarded as a special type of block smoothers, which operate at $m \times m$ blocks of the domain and eliminate the residual there in one step by a direct solver.

Other smoothers that are sometimes used for multigrid are, e.g., Incomplete Lower Upper factorisation (ILU) [53, 106, 85] and Sparse Approximation Inverse (SPAI) smoothers [56, 93]. Again, the trade-off between benefit and effort has to be considered for the choice of the relaxation method for a specific problem.

2.1.2 Intergrid Transfer Operators

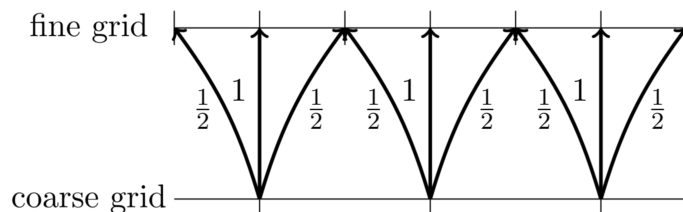
Intergrid transfer operators transfer information between the grid levels of the multigrid hierarchy. The operator for the transfer from a coarse grid to the next finer grid is called prolongation or interpolation, the operator for the transfer from a fine grid to the next coarser grid is called restriction.

The standard prolongation operator used for multigrid is (bi)linear interpolation. In one dimension for seven fine grid points and coarsening by two (i.e., the mesh width between two grid levels differs by a factor of two, $h \rightarrow 2h$), this is how the prolongation matrix looks:

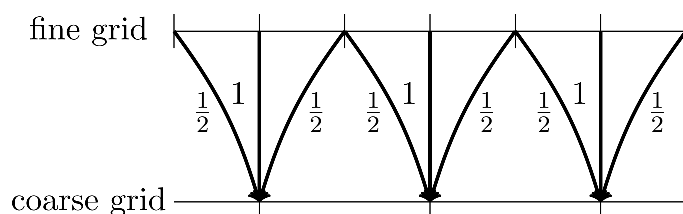
$$P = \frac{1}{2} \begin{pmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.2}$$

Here, three coarse grid points contribute to seven fine grid points (see also Fig. 2.2a). For a fine grid point at the same location as a coarse grid point, the value is simply copied to the fine grid. For a fine grid point at position k in between two coarse grid points, each of these coarse grid point contributes one half of its value: $u_k^h = 0.5 \left(u_{k-1}^H + u_{k+1}^H \right)$. In stencil notation, this corresponds to the stencil $P = \frac{1}{2} \left[\begin{array}{ccc} 1 & 2 & 1 \end{array} \right]$, which reflects the contribution of a coarse grid point to the three nearest fine grid points.

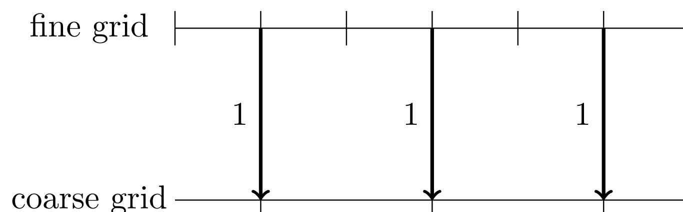
In this work, we stick to the stencil notation for both the intergrid transfer operators and the system matrix, as this is also the way how the operators are applied in our implementations.



(a) Prolongation by linear interpolation.



(b) Restriction by full weighting.



(c) Restriction by injection.

Figure 2.2: Examples for 1D intergrid transfer operators for coarsening by a factor of two.

In two dimensions, the stencil for bilinear interpolation is:

$$P = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix},$$

i.e., a coarse grid point contributes to nine fine grid points. For coarsening by a factor of three, the 2D bilinear interpolation stencil reads:

$$P = \frac{1}{9}\begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 6 & 4 & 2 \\ 3 & 6 & 9 & 6 & 3 \\ 2 & 4 & 6 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix},$$

i.e., a coarse grid point contributes to 25 fine grid points.

We can apply the same stencils as above with inverted direction as restriction operators, resulting in full weighting (see Fig. 2.2b). In matrix notation this would mean that $R = P^T$. Instead of a slim and tall matrix as in 2.2, we receive a wide and short matrix. Application of the one-dimensional matrix then means computing the weighted average of the adjacent fine grid values in order to receive the restricted value for the respective coarse grid vertex.

An alternative to full weighting is injection: There, we simply use the stencil $R = \begin{bmatrix} 1 \end{bmatrix}$. This means that the value of the coarse grid point is copied from the corresponding fine grid point at the same location (shown in Fig. 2.2c).

Bilinear interpolation works quite well if we want to prolong something that is smooth – as the residual should hopefully is after relaxation. However, if the problem we want to solve contains, for example, jumps in the coefficients, or a singularity, bilinear intergrid transfer operators might not be able to transport the information correctly onto the next grid level. For example, bilinear interpolation can lead to a pollution effect, i.e., the coarse grid correction is influenced by the wrong diffusion coefficients. The same is true for the residual if full weighting is used as restriction. Therefore, for this kind of problems it is necessary to take the problem (i.e. the system operator) into consideration when defining prolongation and restriction. This is done in algebraic multigrid methods (see Sec. 2.2.6). Another example for operator-dependent multigrid is described in Sec. 2.3. Additionally, we have to take care that the system operator itself is represented well on the coarser level. This is subject of Sec. 2.1.3.

2.1.3 The Coarse Grid Problem

It is very important to take care that we solve the correct problem on the coarse grid – otherwise the coarse grid correction does not make sense. Therefore, we do not only have to transfer the residual correctly, but also the system matrix.

The geometric approach for bringing the operator to the next coarser grid level is rediscretisation. This means that we simply discretise the continuous operator A with a wider mesh size H, e.g. $H = 2h$. Again there are, however, certain problems, as problems with discontinuous coefficients where the jumps are not located on coarse grid lines, when rediscretisation is not sufficient to receive good convergence rates [3, 95]. The point where the coefficient jump appears might shift on the coarse grid, or the jump can even "get lost". Moreover, one might not be able to simply rediscretise the problem, as the underlying geometry is not known (see Sec. 2.2.6) and/or an unstructured mesh is used. In these cases, the Galerkin coarse grid operator is used. It is defined as follows:

$$A^H = P^T A^h P.$$

We apply first the system operator A^h to the prolongation P, and then the restriction operator, defined as the transposed of the prolongation operator P, to the result. A generalisation is the Petrov-Galerkin coarse grid operator, where the restriction operator R is no longer fixed to be the transposed of P:

$$A^H = RA^h P.$$

However, a Galerkin approach with bilinear interpolation and full weighting as restriction is often not sufficient for a jumping coefficient problem, as the resulting coarse grid operator might not represent the fine grid problem well. In this case, operator-dependent intergrid transfer operators as mentioned in the previous section are necessary.

The solution of the coarse grid problem on Ω_1 can either be done by a direct solver or (if the coarse grid is coarse enough) by applying enough relaxation steps to solve the problem exactly.

2.2 Appearances of Multigrid

From the previous section one might already guess that there is not "the" multigrid method, in fact there is a wide variety of methods. In this section we will point out different appearances of multigrid methods without going in further details, as those are not the topic of this thesis. This is not an exhaustive listing of methods – the intention is just to give an idea of the width of the field of multigrid and to introduce some more important aspects. We do not touch, for example, advanced methods as multigrid for systems of equations (see [95] and references therein).

2.2.1 Coarsening Factors

So far, in our examples we have mostly used a coarsening factor of two, meaning that, when going from the fine grid to the coarse grid, we select every other point

along every coordinate direction: $H = 2h$. This coarsening factor is by far the most popular one. There are, however, some reasons that can lead to using larger coarsening factors. In [35], it is argued that by coarsening by a factor of three for cell-centred discretisations (i.e., the unknowns lie on the cell centres instead of on the vertices of the grid) one receives naturally a nested hierarchy of grids. This means that the unknowns on the coarse grid will lie at the same positions as on the fine grid, i.e. the coarse grid unknowns can be regarded as a subset of the fine grid unknowns (if we do not take the level into account). For coarsening by a factor of two, this is only the case for vertex-centred discretisations. Another reason for coarsening by a larger factor than two could be parallelisation, where a bigger coarsening factor and therefore less levels in the multigrid hierarchy means less communication. On the other hand, larger intergrid transfer operators will be needed, what again increases the communication [109]. A third reason for other coarsening factors may be that the coarsening factor is determined by the application or software framework in use. This is the case for the Peano framework [100] which we use for our target implementation (see Chap. 5). Here, coarsening by a factor of three is an inherent property of the underlying spacetree (see Chap. 3) and therefore of all implemented algorithms. It is important for the choice of the coarsening factor to keep in mind that the error must still be approximated well on the coarse grid – meaning that we do not "lose" frequencies due to a too big mesh width, or we have aliasing effects, because the error frequencies are misinterpreted due to the discretisation on the coarse grid – and that the intergrid transfer operators do not become inefficiently large.

In this thesis, we describe our algorithms in a general manner and give examples for coarsening by a factor of two as well as for a factor of three. Our implementations and experiments, however, employ coarsening by a factor of three.

2.2.2 Alternatives to the Correction Scheme

The correction scheme (CS) as given in Alg. 2.1 is only one possible way to apply the multigrid principle. In particular, it only works for linear problems, as the linear residual equation Eq. 2.1 is in general not true for nonlinear A. Recall from Sec. 2.1 that we used $Ae = A(u - \tilde{u}) = Au - A\tilde{u} = f - A\tilde{u} = r$, with r being the residual, u the exact solution and $\tilde{u}$ the current approximation to the solution, for deriving the linear residual equation. If A is nonlinear, the second equality does generally not hold. Therefore, the nonlinear residual equation

$$Au - A\tilde{u} = r \tag{2.3}$$

has to be used. In addition, we need nonlinear smoothers, such as nonlinear Gauss-Seidel relaxation [76].

Multigrid methods for nonlinear problems are Newton multigrid (i.e. a Newton method for the outer iteration and multigrid for the inner iteration, see, e.g. [24]) and the Full Approximation Scheme – also called Full Approximation Storage – (FAS) (explained, for example, in [17]). FAS can be seen as a generalisation of CS. In addition to the restriction of the residual, we also have to transfer the current approximation of the solution, $\tilde{u}$, to the coarse grid (see Eq. 2.3). The coarse grid problem to be solved is therefore $A^H u^H = r^H + A^H \tilde{u}^H$. This means that we receive a full approximation of the solution on the coarse grid. For the correction of the fine grid approximation, we then have to compute the coarse grid approximation of the error, $\tilde{e}_H = u^H - \tilde{u}^H$, and transfer that back to the fine grid.

An efficient variant to FAS is the Hierarchical Transformation multigrid (HT-MG) method [45]. Another interesting note is that Griebel showed in [46] that applying Gauss-Seidel to a system of hierarchical bases is equivalent to a correction scheme V-cycle.

2.2.3 Adaptive Multigrid

In order to improve the efficiency of multigrid methods, it often makes sense not to use a regularly refined grid (i.e. with the same grid spacing everywhere), but to adaptively refine only those regions more where it is necessary. This might be at the boundary of the geometry in the simulation, or due to the behaviour of the solution (e.g. at singularities).

For Brandt, adaptive grids were the starting point for his Multilevel Adaptive Technique (MLAT) [16, 17], and therefore for his multigrid idea. In the eighties, the Fast Adaptive Composite-Grid method (FAC) [52, 67] was developed as an alternative to the MLAT approach. The main difference between the two is their hierarchy of grids: MLAT is working on the subdomain with a regular grid per level (on the finest level, the subdomain with the finest resolution is processed, then the finest subdomain is coarsened and the next-coarser subdomain is processed etc.). As on the subdomain with the coarser resolution we have no finer grid "above", and therefore no correction equation available, the correction scheme does not work that easily. The Full Approximation Scheme, which was described in Sec. 2.2.2, provides the full approximated solution on every level and is the method of choice to use for MLAT. FAC works on the composite grids, which means that every grid level is adaptive. In contrast to MLAT, we have to take special care of the so-called "hanging nodes", i.e. the vertices at the interface between two resolution levels.

In [78], Rüde presents a mathematical theory for fast and robust adaptive multilevel methods and also shows how to realise those methods in an efficient implementation.

2.2.4 Multigrid as Preconditioner

We have discussed so far the application of multigrid as a stand-alone solver. Multigrid methods are even more popular as preconditioners for Krylov subspace methods, such as the conjugate gradient (CG) [55] or the generalized minimal residual (GMRES) [79] method – multigrid people sometimes call this "acceleration of multigrid" by these methods, although it is more intuitive and common to look at it the other way around: MG methods accelerate the Krylov subspace methods. The accelerated multigrid is often much more robust than using multigrid alone (see e.g. [4, 74, 88, 95] and references therein). Thus, less effort has to be put into optimising the single multigrid components for a special type of problem, and even a weaker version of multigrid such as additive multigrid (see Sec. 2.2.5) can be used. However, Krylov subspace methods are dependent on the mesh width of the problem discretisation, and therefore on n, the number of unknowns of the problem. By a multigrid preconditioner, this dependence can be reduced, and often even the same optimal $O(n)$ complexity can be reached – but with a quite large constant, what makes the preconditioned methods less efficient for really big problems than pure (optimised) multigrid methods.

2.2.5 Additive vs. Multiplicative Multigrid

Multigrid methods as described above are also called multiplicative multigrid, in contrast to additive multigrid (see [12] and references therein). The difference between the two is that in additive multigrid the residual is computed from the unsmoothed u, but the coarse grid correction is added to the smoothed u. Therefore, the smoothing can be performed on an all grid levels in parallel. This approach results in an additive iteration matrix. Obviously, this method is advantageous for parallelisation, but other than that, additive multigrid cannot compete with the multiplicative version (see [12] for a detailed comparison) and is at most applied as a preconditioner. A popular example is the BPX preconditioner [15] (with the name stemming from the initials of the authors).

2.2.6 Geometric vs. Algebraic Multigrid

Another, for this thesis most important, distinction in multigrid methods is that between geometric and algebraic multigrid (AMG). Pure geometric multigrid methods define their coarse grid problem with regard to the underlying geometry: They do a geometric coarsening of the fine grid (taking every other grid point, for example), define geometric intergrid transfer operators (as described in Sec. 2.1.2), and mostly use rediscretisation for the construction of A^H (see Sec. 2.1.3). Pure algebraic multigrid methods, in contrast, usually do not consider the geometry at all. They only

work with the entries of the system matrix. Thus, AMG is able to easily cope with unstructured grids.

As always, there is also a way in between these two extreme directions. One example is BoxMG, as described in Sec. 2.3. It is interesting to note, however, that comparatively few people take the middle-way so far (maybe with the exception of geometric multgrid that uses Galerkin coarse grid operators).

Below, we briefly describe the AMG principles. For a detailed description, we refer to the literature, e.g. [87, 24, 88, 37].

The original AMG method was developed in the eighties [20, 86, 21, 89]. It introduced three basic principles for AMG:

- **The black box principle** stands for the fact that one can use the MG solver as a black box – you only need to give in the system matrix, the right-hand side, and an initial guess, and do not have to bother about the geometry or anything else.

- **Algebraic smoothness** of the error is defined by the convergence properties of the smoother. The error is algebraically smooth per definition if it is reduced slowly by the smoothing operator. An algebraically smooth error can look quite "rough" in a geometric sense.

- **Operator-dependent coarsening and intergrid transfer operators** has the meaning that the choice of coarse grid points as well as the prolongation P and the restriction R depend on the entries of the system matrix.

Additionally, the concept of **strength of connections**, i.e. a measure for determining how strong a matrix entry depends on another, was defined. Based on this and the observation that the algebraically smooth error varies slowly in the direction of strong dependence, classical AMG uses heuristics for selecting the coarse grid points. The smooth error characterisation is then used to receive the prolongation operator.

In contrast to geometric multigrid, where the coarsening strategy, the intergrid transfer operators and often also the coarse grid operators are fixed and the smoother has to be chosen in an optimal way, in AMG only the smoother (which is often a simple point Gauss-Seidel) is fixed and the rest is computed from the system operator. This means that we have an extensive setup phase which mostly dominates the computational costs. On the other hand, the resulting method is usually more robust than geometric MG methods.

A very popular AMG method is Smoothed Aggregation [96, 98, 97, 22, 23]. Here, the fine grid points are clustered to disjoint aggregates, according to their strength of connection. A tentative piece-wise constant prolongator $\tilde{P}$ is then defined by the

aggregates (i.e., $\tilde{P}_{ij} = 1$ if point i belongs to aggregate j) and improved by smoothing with damped Jacobi relaxation (see Sec. 2.1.1). Such, the actual prolongation operator is received as $P = (I - \omega D^{-1} A)\tilde{P}$.

2.3 BoxMG

The so-called "Black Box Multigrid" (BoxMG) was introduced by Dendy in 1982 [30] and is based on work of Alcouffe et al. [3]. It has shown to yield robust and efficient multigrid solvers. During the last decades, the BoxMG method was extended for a large class of problems (see, e.g., [31, 34, 32, 33, 71, 35, 66]).

We call BoxMG a geometric-algebraic hybrid as it has properties of both kinds of multigrid methods: As in geometric multigrid, a structured grid and geometric coarsening are used. For the intergrid transfer operators, on the other hand, operator-dependent restriction and prolongation are defined and the coarse grid operator is computed in a (Petrov-)Galerkin way (see Sec. 2.1.3). The idea is, similar as in AMG (see previous section), that the user only needs to provide the fine grid discretisation of the problem and the right-hand side. The problem is then solved in a "black box", without the need of tuning the algorithm to the problem and the underlying geometry (as it is necessary in geometric multigrid in order to receive satisfying performance). However, the geometric structured grid approach implies that special care has still to be taken for the smoother.

Recently, MacLachlan et al. investigated the relationship between BoxMG and AMG in detail [66] and pointed out that, under a certain point of view, BoxMG can be seen as a special case of classical AMG (with the definition of "strong connections" being a geometric question instead of an algebraic one). In [104], Wienands and Köstler presented a framework for the construction of prolongation operators in which the BoxMG method was recovered as a special case.

2.3.1 The BoxMG Prolongation Operator

As mentioned above BoxMG defines the intergrid transfer operators dependent on the system operator, and uses the Petrov-Galerkin principle to compute the coarse grid operators.

In the following, we describe BoxMG for two-dimensional problems, however, the principle is also applicable for three dimensions, see [32].

BoxMG for Coarsening by Two

Let us first consider the standard approach, i.e., for coarsening by a factor of two. The following description partly follows [35] and [88].

The idea of BoxMG can be theoretically derived from the standard coarse-fine splitting (see, e.g., [88]). For defining prolongation, we want to use information about the relationship between the variables on the fine grid (in terms of the system operator A^h), and especially how variables on coarse grid positions influence the other variables. For that purpose we split the discretised domain Ω^h into two disjoint subsets: $\Omega^h = C^h \cup F^h$. C^h consists of those variables which are at the same position as coarse grid variables, Ω^H, and $F^h = \Omega^h \setminus C^h$. The equation system $A^h u^h$ is reordered such that we have blocks of F-variables influenced by F-variables (A_{ff}), F-variables influence by C-variables (A_{cf}), C-variables influenced by F-variables (A_{fc}), and C-variables influenced by C-variables (A_{cc}). With this we can write the system in the form

$$A^h u^h = \begin{bmatrix} A_{ff} & A_{fc} \\ A_{cf} & A_{cc} \end{bmatrix} \begin{bmatrix} u_f \\ u_c \end{bmatrix} = \begin{bmatrix} f_f \\ f_c \end{bmatrix} = f^h$$

and the prolongation and restriction operator as

$$P = \begin{bmatrix} I_{fc} \\ I_{cc} \end{bmatrix}, R = \begin{bmatrix} I_{fc}^T & I_{cc}^T \end{bmatrix}, \tag{2.4}$$

with I_{fc} being the intergrid transfer operator from coarse grid variables to fine grid variables on non-coarse grid positions and I_{cc} the intergrid transfer operator from coarse grid variables to fine grid variables on coarse grid positions. If we define $I_{cc} = \mathbb{I}_c$, i.e., the values for fine grid variables on coarse grid positions are copied from the coarse grid, and $I_{fc} = -A_{ff}^{-1} A_{fc}$, and if A_{ff}^{-1} exists, the Galerkin coarse grid operator is the Schur complement: $A^H = C^H = -A_{cc} - A_{cf} A_{ff}^{-1} A_{fc}$, and we receive a direct method. This method, however, requires inverting A_{ff} and A^H.

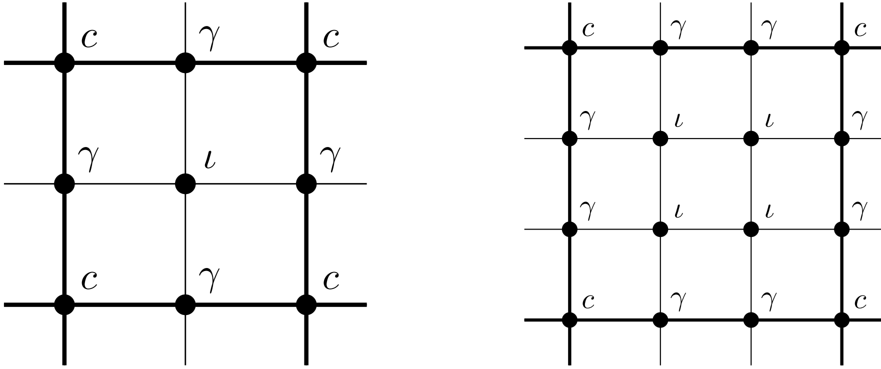


Figure 2.3: Fine grid point types for BoxMG prolongation for coarsening by two (left) and three (right). c points coincide with the coarse grid.

In order to avoid this, A_{ff}^{-1} is approximated in a cheaper way. Therefore, the BoxMG approach splits the fine grid points in F^h again into γ points (:= fine grid

points on coarse grid lines) and ι points (:= fine grid points in the middle of coarse grid cells) – see Fig. 2.3. In block-form, the system $A^h u^h = f^h$ can now be written as

$$A^h u^h = \left[\begin{array}{cc|c} A_{\iota\iota} & A_{\iota\gamma} & A_{\iota c} \\ A_{\gamma\iota} & A_{\gamma\gamma} & A_{\gamma c} \\ \hline A_{c\iota} & A_{c\gamma} & A_{cc} \end{array}\right] \left[\begin{array}{c} u_\iota \\ u_\gamma \\ \hline u_c \end{array}\right] = \left[\begin{array}{c} f_\iota \\ f_\gamma \\ \hline f_c \end{array}\right] = f^h. \tag{2.5}$$

To get an efficient method, the influence of ι points to γ points is neglected and γ points are interpolated from c points only. That way. we receive an operation that stays locally in a coarse grid cell without using information from adjacent cells. [3] Formally we get:

$$\widehat{A}^h u^h = \left[\begin{array}{cc|c} A_{\iota\iota} & A_{\iota\gamma} & A_{\iota c} \\ 0 & \widehat{A}_{\gamma\gamma} & \widehat{A}_{\gamma c} \\ \hline A_{c\iota} & A_{c\gamma} & A_{cc} \end{array}\right] \left[\begin{array}{c} u_\iota \\ u_\gamma \\ \hline u_c \end{array}\right] = \left[\begin{array}{c} f_\iota \\ f_\gamma \\ \hline f_c \end{array}\right] = f^h. \tag{2.6}$$

Eq. 2.4 then becomes

$$P = \left[\begin{array}{c} \widehat{I}_{fc} \\ I_{cc} \end{array}\right], R = \left[\begin{array}{cc} \widehat{I}_{fc}^T & I_{cc}^T \end{array}\right] \tag{2.7}$$

with

$$\widehat{I}_{fc} = -\left[\begin{array}{cc} A_{\iota\iota} & A_{\iota\gamma} \\ 0 & \widehat{A}_{\gamma\gamma} \end{array}\right]^{-1} \left[\begin{array}{c} A_{\iota c} \\ \widehat{A}_{\gamma c} \end{array}\right] = -\widehat{A}_{ff}^{-1}\widehat{A}_{fc} \tag{2.8}$$

by inverting the homogeneous equation associated with the upper block of A_h in Eq. 2.6.

Now the question is: How can we define $\widehat{A}_{\gamma\gamma}$ and $\widehat{A}_{\gamma c}$ such that $\widehat{A}$ is close to A? The BoxMG method collapses the two-dimensional stencils belonging to γ points to one-dimensional stencils by integrating (i.e., summing up) over the dimension perpendicular to the corresponding coarse grid line. This approach assumes that the error varies along the coarse grid line and is constant in direction perpendicular to that line.

We use compass-notation for 9-point stencils on the fine grid:

$$A^h(i,j) = \left[\begin{array}{ccc} -NW & -N & -NE \\ -W & O & -E \\ -SW & -S & -SE \end{array}\right],$$

with (i,j) being the position on the grid.

[3] This gives us a method that perfectly fits an element-wise setting as described in the next chapters.

For a γ point on a horizontal (x-)line, u_γ, for example, the collapsed stencil is

$$\hat{A}_{\gamma c} = \left[\begin{array}{ccc} -\overline{W} & \overline{O} & -\overline{E} \end{array}\right], \tag{2.9}$$

with

$$\overline{W} = W + SW + NW, \quad \overline{O} = O - S - N \text{ and } \quad \overline{E} = E + SE + NE \ . \tag{2.10}$$

The computation of the interpolation weights in practice happens in three steps, with steps 2 and 3 using the result of the previous one(s):

1. c points u_c get the interpolation weight 1 (injection):
$$u^h_{cO} = 1u^H_{cO}. \tag{2.11}$$
2. γ points are interpolated from the corresponding c points using Eq. 2.9 and 2.11. For a γ point on a horizontal line, again, we get:
$$u^h_{\gamma O} = \frac{\overline{W}u^h_{cW} + \overline{E}u^h_{cE}}{\overline{O}} \tag{2.12}$$
3. Interpolation weights for ι points x_ι are derived from the corresponding c points and γ points using the full stencil and Eqs. 2.11 and 2.12:
$$\begin{aligned} u^h_{\iota O} = &\frac{NWu^h_{cNW} + Nu^h_{\gamma N} + NEu^h_{cNE}}{O} + \frac{Wu^h_{\gamma W} + Eu^h_{\gamma E}}{O} \\ &+ \frac{SWu^h_{cSW} + Su^h_{\gamma S} + SEu^h_{cSE}}{O}. \end{aligned} \tag{2.13}$$

The prolongation is defined as $R = P^T$ and the coarse grid operator in a Galerkin way: $A^H = P^T A^h P$. With this scheme, it turns out that for the 9-point Laplace operator, BoxMG constructs a bilinear prolongation operator (see Sec. 2.1.2), and with that the Laplace operator is reobtained on the coarse grid. This provides a perfect test case for a BoxMG code.

Two enhancements to this algorithm are proposed in [31, 35]:

First, the result of a Jacobi step is added along with the prolongation by $u^h \leftarrow u^h + Pu^H + \frac{f^h - A^h u^h}{diag(A^h)}$. This generally improves the convergence at a small cost, because the numerator in the last term is the old residual, which has already been computed before the coarse grid correction is applied.

Second, a heuristic switch, motivated by experience, is proposed for the diagonal weight $\overline{O}$ of $\widehat{A}_{\gamma\gamma}$ in Eq. 2.12:

$$\text{diag}(\widehat{A}_{\gamma\gamma}) = \begin{cases} \overline{O} & O > (1 + \min(\frac{|\overline{W}|}{O}, \frac{|\overline{E}|}{O})(\overline{W} + \overline{E}) \\ \overline{W} + \overline{E} & O \leq (1 + \min(\frac{|\overline{W}|}{O}, \frac{|\overline{E}|}{O})(\overline{W} + \overline{E}) \end{cases}.$$

This switch is said to improve the convergence for mixed or Dirichlet boundary conditions. In the case of a zero sum operator, i.e. $\overline{O} = \overline{W} + \overline{E}$ (e.g. for pure diffusion problems at inner points), this switch is, of course, unnecessary.

BoxMG for Nonsymmetric Problems

For nonsymmetric problems, it is proposed in [31] to use $A^h_{sym} = \frac{1}{2}[A^h + (A^h)^T]$ for the construction of P as described in the previous section, and $(A^h)^T$ for the construction of R. This choice is motivated by two general principles for intergrid transfer operators: First, for a positive-definite system operator A, the resulting operators should reduce to those given in the standard approach as described in the previous section. Second, if A_h has an upstream bias, the restriction R should have a downstream bias. This second principle is cited from [18]. Several alternatives for the choice of P and R are tested in [31], with the one given above resulting in the best convergence rates and being most robust.

For smoothing, the robust Kaczmarz relaxation [62] which is also applicable to equation systems which have non-diagonally-dominant (and therefore not positive-definite) system operators is recommended in [31]. The main idea of this relaxation method is that, for solving $Au = f$ we define $u = A^T y$ and solve $AA^T y = f$ instead, as AA^T is positive-definite. The update rule for relaxing the kth row in A is $u_k^{(i+1)} = u_k^{(i)} + \frac{A_{kj} r_k^{(i)}}{\sum_l A^2_{kj_l}}$.

BoxMG for Coarsening by Three

We will now look at BoxMG on a grid that is coarsened by three, which is a generalisation of the approach described before (see [35]). Reasons for using a coarsening factor three rather than two are discussed in Sec. 2.2.1.

Again, the fine grid points are split into c, γ and ι points (see Fig. 2.3). Eq. 2.5 remains the same for the coarse-fine splitting, and, as before, we eliminate the connections from ι to γ points and receive Eqs. 2.6, 2.7 and 2.8.

However, the dimensionalities of $\widehat{A}_{\gamma\gamma}$ and $A_{\iota\iota}$ have changed: $\widehat{A}_{\gamma\gamma}$ is a block diagonal matrix with each block a 2×2 matrix (instead of a diagonal matrix), and also $A_{\iota\iota}$ is block diagonal (instead of diagonal) with each block a 4×4 matrix.

As before, the stencils for γ points are collapsed by averaging the stencil entries perpendicular to the coarse grid line (see Eqs. 2.9 and 2.10). Instead of one equation with one unknown (which can be inverted directly by hand) we now receive for γ points two equations with two unknowns, and for ι points four equations with four unknowns. By solving the equations in the same order as before, and using the results of the previous steps for the current step, we receive the interpolation weights.

The enhancements described in the coarsening-by-two section can be adopted.

3 Spacetrees as Data and Computational Structure

In science and engineering, the simulation models nowadays become more and more accurate, and due to that very complex, and the amount of data that has to be processed in a simulation increases. In order to be able to efficiently apply PDE solving methods such as multigrid, one has to define a suitable way how to organise and access the data. One way that proved to be efficient for this purpose are spacetrees. The hierarchical structure of spacetrees mirrors the grid hierarchy in multigrid methods. This makes spacetrees a perfect fit as a data (and as we will see in Sec. 3.2 also as a computational) structure for multigrid. In order to benefit from spacetrees, however, it is important to store the tree data structure in a way that has small memory demand, allows an efficient tree traversal and is, in the best case, cache- and memory-efficient in terms of data access (see, e.g. [47, 43, 101]).

3.1 Spacetrees and Spacetree Grids

A spacetree [43] is a hierarchical data structure represented by a tree. The root of the tree represents the complete data domain, which is then subsequently divided into smaller domains on each tree level. A k^d (space)tree is a spacetree in d dimensions for which each domain is divided into k^d domains of equal size on the next level. [1] A 2^2 spacetree, also called quadtree, with the corresponding spacetree grid is shown in Fig. 3.1 in the upper row, the row below shows a 3^2 spacetree. The 3D equivalent to a quadtree, a 2^3 spacetree, is called an octree.

In Alg. 3.1 we formalise the recursive construction of a spacetree. It is called with SPACETREE(root), with the root being one node that contains the complete data domain, for example the geometry of the simulated scenario. This domain is refined (e.g. by bi- or tripartitioning). If the maximum refinement level is not yet reached, the method is called for each subdomain that is not resolved sufficiently, i.e. these subdomains are refined further.

By that algorithm, we receive an adaptively refined spacetree grid (as in Fig. 3.1 and already briefly discussed in Sec. 2.2.3): Its refinement level at a certain point in the domain depends on how "interesting" that point is, i.e., how complicated the

[1] The original definition of k^d trees also allows the subdomains to be of different size.

Algorithm 3.1 Spacetree Construction

```
function SPACETREE(node n)
    refine n
    if maximum refinement level is reached then
        return
    else
        for all children c of n do
            if c is not resolved sufficiently then
                SPACETREE(c)
            end if
        end for
    end if
end function
```

geometry is there, for example. We only resolve that parts of the domain further where it is necessary. Another refinement criterion could be the behaviour of the solution at that point: Does it differ a lot from that at the neighbouring points (then we should refine further), or is it very smooth (then a finer resolution is not necessary)? Several other refinement criteria can be found in literature. In contrast, a regularly refined grid has the same resolution and therefore the same spacetree depth on the whole domain.

Spacetrees enable fast data access and access to the parents, children, and (depending on the implementation) sometimes also the siblings of a vertex. We choose an approach where the data is accessed in an element-wise manner, with each element being a cell of the spacetree grid, and represented by one node of the spacetree. The structure can be efficiently stored as linearised bit-code, with each bit standing for a tree node and stating whether it is a leaf or not [43]. For defining a linear order for the nodes (and thus also for the data located at the nodes), the traversal scheme of the tree has to be fixed. Here, space-filling curves, as described in the next section, have turned out to have some useful properties.

3.2 Spacetrees as Computational Structure Using Space-Filling Curves

By using spacetrees as computational structure we mean that the computations are performed directly on the tree – instead of loading the data from the tree into separate data structures in order to do the computations –, exploiting its structure. That way, an elegant recursive formulation of algorithms is possible and one is easily able to cope with adaptive grids. As pointed out in the previous section, we want

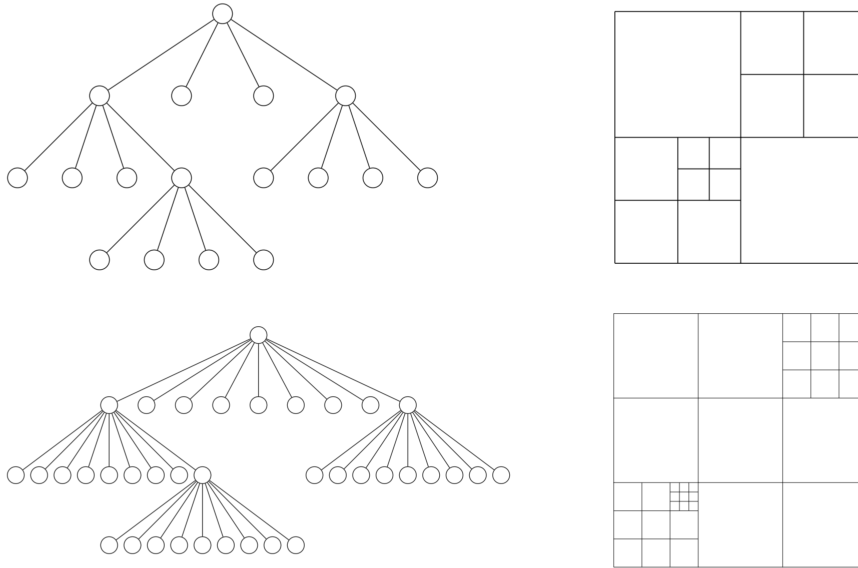


Figure 3.1: Spacetrees (left) for a bisected (top) and trisected (bottom) adaptive grid, and the corresponding spacetree grids (right). We use lexicographical ordering of the grid cells and a Cartesian coordinate system.

to store the spacetree structure and the data efficiently in a linearised form. The data located at the tree nodes has therefore to be processed in a sequential manner by doing a spacetree traversal. If we do a depth-first traversal of the spacetree and define a fixed order of traversing the children of a node, we get a space-filling curve (see, e.g. [80, 6]). A space-filling curve is a surjective and continuous mapping from a 1D interval to an interval in a higher dimension. [2] We will here look only at 2D space-filling curves on squares.

The structure of the grid and the order in which the children of a node are visited correspond to different types of space-filling curves. In Fig. 3.2, we show three examples in 2D:

- the z-curve, which is often used used for quadtree grids and results from the so-called Morton order, which is defined from the bits of the binary presentation of the cell coordinates,
- the Hilbert curve, which also is defined for quadtree grids,
- and the Peano curve, which is defined for 3^2 (tripartitioned) spacetree grids.

There is a number of other space-filling curves, also, for example, for triangulated grids (the Sierpinski curve, see [6]), and many of them can be extended to 3D.

[2] The continuity property might be fulfilled only in a special mathematical sense, for example for the z-curve as given below. See [80, 6] for a detailed mathematical discussion.

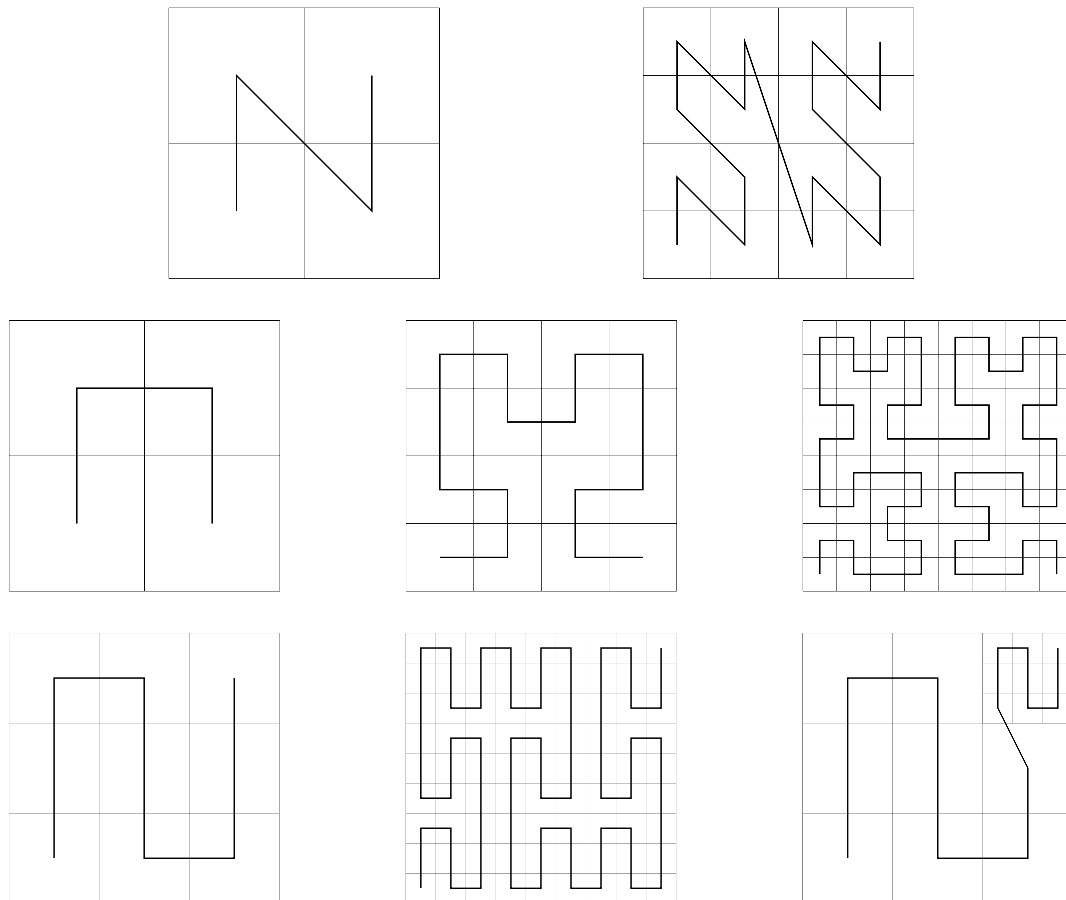


Figure 3.2: The first two iterations of the z-curve resulting from Morton order (top), the first three iterations of a variant of the Hilbert curve (middle) and the first two iterations and an adaptive version of a variant of the Peano curve (bottom).

A detailed introduction to spacetrees and space-filling curves with applications in scientific computing can be found in [6].

In Alg. 3.2 we give a general formulation for the construction of a space-filling curve. The three example curves given above result from different basic patterns for the traversal, and how these are transformed (i.e. scaled, rotated or mirrored) and connected upon refinement. The algorithm defines the traversal of an adaptive grid if for one domain only part of the subdomains are refined further.

One important difference between the z-curve and the Hilbert and Peano curve is the so-called "connectedness", which means that two subsequently visited cells share at least one edge. This implies that there are no jumps over cells in the curve, as between the eighth and the ninth visited cell in the middle of the right-hand z-curve in Fig. 3.2. Connectedness is one of the key properties required to enable a

Algorithm 3.2 Construction of a Space-filling Curve

```
function SpacefillingCurve(domain d, splitting factor f, pattern p, transfor-
mation t)
    Split d into f equal subdomains s.
    Based on t, define a new transformation t̃ of p for each s.
    Use transformed p to order all s.
    for all s do
        if s has to be refined then
            SPACEFILLINGCURVE(s, f, p, t̃).
        end if
    end for
end function
```

cache- and computationally efficient stack-based implementation of the traversal of an adaptive grid, as described in [48, 77, 65, 68, 99, 101] for the Peano curve and discussed in [6] for various other curves. The main idea is the following: When a vertex is touched for the first time during the traversal, its data is put on one of several temporary input data structures. If the vertex is touched for the last time, its data is put on one of several temporary output data structures. It turns out that the order of processing the vertices for the first time is exactly inverse to the order of processing them for the last time. This becomes clear when looking at the Peano or Hilbert curve in Fig. 3.2. Therefore, stacks are the natural choice as a data structure. It turns out that with this approach, good spatial and temporal data locality is achieved. These locality properties are the key to cash efficiency (see [101] and references therein).

When doing computations on a spacetree using a traversal with space-filling curves as described above and staying in a strict local (i.e., we have no information from other cells), cell-wise mode, one has some restrictions on which other nodes are (easily) accessible. Fig. 3.3 shows this for one node in the spacetree (see also [6]). There are direct connections from a node/grid cell to its parent and to its children, meaning that they are visited in a row during a traversal and hence lie in the memory in a sequential order. Therefore, in an implementation it is easily possible to have them accessible at the same time and thus exchange information between them without loss of memory efficiency. Also, if we allow all children of a node to be accessible at the same time, a child can access its siblings. But we see that we cannot directly access the neighbours of a tree node/grid cell. The issue with this property becomes especially remarkable in the middle of the domain: The four adjacent cells sharing the middle vertex are not directly connected by edges in the spacetree. This means that we have no direct access from one cell to the neighbouring cell, and in the depth-first traversal defined by a space-filling curve there may lay several other

cells between them. That has to be taken into account when designing algorithms on spacetree grids.

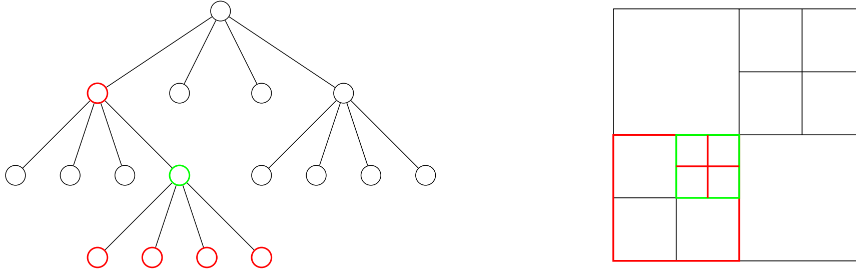


Figure 3.3: The green highlighted spacetree node/spacetree grid cell has direct connections only to its parent and its children (highlighted in red), i.e., during a traversal, they are visited in a row.

One advantage of spacetrees is that they provide an intrinsic domain decomposition for parallelisation: Each processor receives a subtree of the spacetree. In order to allow an efficient information exchange at the processor boundaries, the space-filling curve should fulfil the connectedness property as explained above. In addition, staying in a strict cell-wise, local notation for the algorithm turns out to have big advantages when going to parallel architectures, as the need for information from other processors – and therefore the need for expensive inter-processor communication – is reduced.

In the next chapters we describe how to design a robust and efficient, parallelisable multigrid algorithm on spacetree grids, and discuss in detail what implications the cell-wise traversal has on the algorithm.

4 Geometric-Algebraic Multigrid on Spacetrees – Algorithm Prototyping

By algorithm prototyping we mean that we test our ideas and design the algorithms in an environment that allows us easy and fast implementation and testing of single building blocks before integrating the algorithms into the target framework.

For this thesis, we did the algorithm prototyping in Matlab. It allows us to decouple the algorithms from the framework structure and the single components from each other, in kind of a unit testing idea [42]. Thus, it is also possible to compare our methods to methods without the restrictions we receive from the underlying spacetree and the element-wise principle in the target framework, so that we can analyse the benefits and drawbacks of such a spacetree-based implementation. At the same time, by designing the algorithm such that it meets exactly the requirements of the target framework, the integration into this framework is straightforward.

Later we used the prototyping codes to verify the target implementation and to generate test cases for it. Such, the overhead for keeping the codes consistent paid off.

Two main prototyping codes were developed: Prototype 1, described in Sec. 4.2, completely ignores any spacetree ideas and locality properties and was used to test the robustness and efficiency of the chosen BoxMG method for coarsening by a factor of three and hard, nonsymmetric problems. Prototype 2, described in Sec. 4.3, uses cell-wise operators and local smoothers as required by a spacetree-based implementation, but it is still not implemented on a spacetree. So we can, for comparisons, apply also smoothers that are not feasible in a strict cell-wise implementation. Chap. 5 finally brings the developed methods to the target framework with its strict spacetree principles.

4.1 Requirements

The following demands are posed to the desired algorithm: robustness and efficiency for a wide range of problems, suitability for parallelisation, and suitability for integration into a state-of-the-art spacetree-based software framework.

In the following sections we develop a suitable algorithm and test its properties. All implementations will be for coarsening by a factor of three, as this is required in the target platform. The principles, however, apply also to other coarsening factors and will be illustrated both for a factor two and a factor three.

The first requirement is tested and improved in a general setting in Prototype 1. Here, we combine several methods, mostly developed by Dendy, in order to receive a multigrid method for coarsening by three which can also cope with nonsymmetric problems. In addition, we make some observations about the discretisation using finite differences.

The fulfilment of the second and the third requirement are reached in Prototype 2. The parallelisability and suitability for a spacetree implementation is brought in by the cell-wise operators and the development and testing of suitable smoothers. It turns out that the choice of a good smoother in a spacetree setting is not trivial, and at the same time it is crucial for maintaining the robustness and efficiency of the BoxMG method.

The fulfilment of the parallelisation requirement is finally tested in Chap. 5, when the methods developed in Prototype 1 and 2 are implemented and parallelised in the target framework.

4.2 Nonsymmetric BoxMG With Coarsening by Three

The robustness of the BoxMG method for coarsening by two was already shown, mostly by Dendy et al., in several papers [30, 31, 34, 32, 33, 71, 66], including for nonsymmetric problems. Dendy and Moulton showed that the approach can be extended to coarsening by three multigrid [35] and is, using a red-black line ("zebra") smoother, robust also for problems with jumps in the coefficients.

In [109], we showed that by combining the approaches from [31] (BoxMG for nonsymmetric problems) and [35] (BoxMG for coarsening by a factor of three) (see Sec. 2.3) with a powerful symmetric line Gauss-Seidel smoother, we get a coarsening-by-three multigrid solver which is able to handle also nonsymmetric problems and the very challenging recirculating flow problem. In addition, we introduced a new notation for the finite-difference discretisation of the problem, which contains an artificial diffusion parameter that allows us to switch easily between first and second order discretisation in order to demonstrate the behaviour of our multigrid solver for different discretisation orders.

The following sections follow partly [**?** 109] in a revised version.

4.2.1 Convection-Diffusion Equation

Our model problem for Prototype 1 is the two-dimensional convection-diffusion equation written in flux form as follows:

$$\begin{aligned} Au = -\nabla \cdot (\epsilon \nabla u) + (au)_x + (bu)_y &= f, \qquad && (x,y) \in \Omega, \\ u_n = k, \quad \text{or} \quad u &= g, && (x,y) \in \partial\Omega. \end{aligned} \tag{4.1}$$

Here, $\epsilon > 0$ is the diffusion coefficient and $a(x,y)$ and $b(x,y)$ are the point-wise convection velocities in the x and y direction, respectively. The given functions $f(x,y)$ and $g(x,y)$ (or $k(x,y)$) are the right-hand side forcing and the boundary condition, respectively, and u_n denotes the derivative of u in the direction normal to the boundary. Our variable is $u(x,y)$, which describes, for example, the concentration of a passive tracer in a fluid.

This equation becomes very challenging for convection-dominated problems, i.e., if ϵ is very small, as in this case the system operator is highly nonsymmetric. If $a \neq b$ the problem is anisotropic, what makes it hard to solve for geometric multigrid with point-wise smoothers and standard-coarsening (see, e.g., [95]).

The problematic part of the convection-diffusion equation is the convection part. Therefore, we will have a closer look at this part in the following sections.

4.2.2 Discretisation

We discretise our problem at the cell-centers of a Cartesian grid, using central differences for the Laplacian (diffusion term) and an upstream discretisation for the first derivatives (convection terms). We test both first order and second order upstream discretisations, as explained below. We denote by h the mesh size of the discrete problem, $A^h u^h = f^h$. In our discussion, h is assumed to be uniform, but in our implementation we allow distinct and variable mesh-sizes in the x and y directions.

In order to obtain the possibility to change between first and second order discretisation, we rewrite the convection operator in the following way: In [109], we show that any consistent near-neighbour upstream discretisation of a 2D convection operator $A_{conv} = a\partial_x + b\partial_y$ (assuming, without loss of generality, non-negative a and b) can be written in the form

$$A^h_{conv} = A^h_0 + c(h)D^h \,, \tag{4.2}$$

where $c(h)$ is a free parameter,

$$A^h_0 = \frac{1}{2h}\begin{bmatrix} 0 & 0 & 0 \\ -a+b & a+b & 0 \\ -a-b & a-b & 0 \end{bmatrix}, \quad \text{and} \quad D^h = \frac{1}{h^2}\begin{bmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & -1 & 0 \end{bmatrix}.$$

The operator A_0^h can be interpreted as a second order accurate discretisation of A_{conv} with respect to the location $(x - \frac{h}{2}, y - \frac{h}{2})$, where (x, y) is the centre of the stencil. The operator D^h is a second order accurate approximation to the mixed second derivative at the same point. This means that Eq. 4.2 can be interpreted as a second order accurate discretisation of the operator $a\partial_x + b\partial_y + c(h)\partial_{xy}$, centred at $(x - \frac{h}{2}, y - \frac{h}{2})$. The term $c(h)\partial_{xy}$ can be considered as an anisotropic artificial diffusion which, for consistency, must vanish as $h \to 0$. Note that choosing $c(h) = \frac{1}{2}(a + b)h$ yields the standard first order upstream stencil, which is known to be stable for convection-dominated problems:

$$A_{conv}^h = \frac{1}{h} \begin{bmatrix} 0 & 0 & 0 \\ -a & a+b & 0 \\ 0 & -b & 0 \end{bmatrix}.$$

Choosing $c(h) = 0$ yields a second order accurate compact upstream discretisation. Note that in this case the operator contains large positive off-diagonal terms, and yet we shall see that our solver handles this operator quite well.

Having these two possible choices for $c(h)$ at hand, we can easily switch between a first and a second order discretisation of the convection part in Eq. 4.1. From the analysis given in [109], it turns out that any upstream discretisation of A_{conv} of the form written in Eq. 4.2 can be called *stable* if $c(h)$ is positive, *neutrally stable* if $c(h) = 0$, and *unstable* if $c(h)$ is negative.

4.2.3 Intergrid Transfer Operators and Coarse Grid Problem

The coarse grid operators are constructed using the Petrov-Galerkin method (see Sec. 2.1.3). For the intergrid transfer operators and the coarse grid problem, we have again a look at the convection part of the operator, as the diffusion part is expected to be unproblematic. In our experiments in Sec. 4.2.5 we will see that this is the case also for non-constant coefficients.

By direct computation it is found that the nonsymmetric BoxMG algorithm for an upstream discretisation of the convection operator A_{conv} and coarsening by three yields the following prolongation and restriction operators in the constant coefficient case:

$$P = \frac{1}{9} \begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 6 & 4 & 2 \\ 3 & 6 & 9 & 6 & 3 \\ 2 & 4 & 6 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix}, \quad R = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}.$$

These P and R produce a coarse grid discretisation that is a second order accurate

approximation to the fine grid convection operator (also by direct computation):

$$
\begin{aligned}
[A^H_{conv}] &= 9([A^H_0] + c(h)[D^H]) \qquad (4.3)\\
&= 9\left(\frac{1}{2H} \begin{bmatrix} 0 & 0 & 0 \\ -a+b & a+b & 0 \\ -a-b & a-b & 0 \end{bmatrix} + \frac{c(h)}{H^2} \begin{bmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & -1 & 0 \end{bmatrix} \right),
\end{aligned}
$$

where $H = 3h$ is the coarse-grid mesh-size. The operators are of appropriate accuracy for achieving fast convergence with a V-cycle (see [107]). In addition, we see that, as we appeal to coarser and coarser grids, the relative value of $c(h)$ in Eq. 4.2 tends monotonically to zero, as $c(h)$ stays the same in the coarse grid Eq. 4.3 as in the fine grid Eq. 4.2. Therefore, it can be concluded that a stable upstream discretisation stays stable after coarsening.

4.2.4 Relaxation

We already stressed that, due to the structured grid structure, the robustness and efficiency of BoxMG depends a lot on the smoother that is used.

In [31], the "safe" but slow Kaczmarz relaxation was proposed as appropriate smoother for nonsymmetric problems (see also Sec. 2.3.1). In order to improve the efficiency, we use symmetric x- and y-line Gauss-Seidel (see Sec. 2.1.1) and show that this also yields a robust method for nonsymmetric problems. By symmetric x/y-Gauss-Seidel we mean that we go over lines of constant y/x, first in ascending and then in descending x/y direction, and eliminate the residuals along each line simultaneously. This requires solving tridiagonal linear systems. Line Gauss-Seidel is commonly used for convection-dominated convection-diffusion problems and for high-Reynolds number flows (see, e.g., [102] and references therein).

Stability of Relaxation

By the Fourier analysis in [109], line Jacobi relaxation is found to be non-divergent for stable upstream discretisations, i.e., for $c(h) > 0$, of a convection operator A_{conv} with constant coefficients. An analysis of line Gauss-Seidel relaxation for the same case shows that if the relaxation is carried out in downstream ordering, then it is a stable marching process, which naturally yields an exact solver in the case of pure upstream convection. On the other hand, when it is carried out in upstream ordering (the "wrong" direction) it remains nondivergent. These results hold for any $c(h) \geq 0$. Together with the observations about the coarse grid operator given in Sec. 4.2.3 it can be concluded that symmetric line Gauss-Seidel, with line relaxation carried out in alternating directions, can be expected to yield a robust smoother for

this problem, despite the fact that the matrices are not M-matrices.[1] As line Gauss-Seidel is known to be robust for pure diffusion problems, adding the diffusion part should not hurt the robustness. This is also confirmed for first order discretisation by the two-grid analysis (see [95, 103] for details of this type of local Fourier analysis) in [109] for the constant coefficient convection-diffusion problem.

These results hold for coarsening by a factor of three, but an analysis for the coarsening-by-two case is expected to yield comparable results, as the coarsening factor is no essential feature in the analysis.

In our experiments in the next section, also the non-constant coefficient case is considered and our method is shown to be robust also there.

4.2.5 Numerical Experiments

To show the discretisation accuracy, robustness, and convergence behaviour of our approach, we present numerical results for several scenarios. In Fig. 4.1, the streamlines of the different problem setups are shown. The domain in the following numerical examples is $\Omega = [-\frac{1}{2}, \frac{1}{2}] \times [-\frac{1}{2}, \frac{1}{2}]$. In all these tests we use a V(1, 1) cycle with one symmetric x-line Gauss-Seidel pre-relaxation and one symmetric y-line Gauss-Seidel post-relaxation. Unless stated otherwise, we coarsen down to a 9×9 grid and use a direct solver there. The notation $||.||$ refers to the l_2 norm. We start with a random initial guess and impose a zero right-hand side. (This choice has no effect on the asymptotic convergence behaviour.) The iteration is stopped when the residual norm $||r_i|| = ||f^h - A^h u_i^h||$ is reduced by at least a factor of 10^8. Here, i denotes the iteration number. In addition to the number of iterations needed for reaching this goal, we track the convergence factor ρ, which is defined as $\rho := \frac{||r_i||}{||r_{i-1}||}$, after each iteration i. ρ_{mean} stands for the (geometric) mean convergence factor: $\rho_{mean} := \sqrt[n]{\frac{||r_{end}||}{||r_0||}}$.

Accuracy of Discretisation

First, we test whether the discretisation switch for the convection part as described in Sec. 4.2.2 yields the expected accuracy.

Table 4.1 (left) displays the accuracy of the discrete solution with $c(h) = 0$ and $c(h) = \frac{1}{2}(a + b)h$ for the convection equation ($\epsilon = 0$) with constant coefficients $a = 1$ and $b = 0.5$. The inflow boundary conditions are $g(-0.5, y) = \frac{(y^2-0.25)^4}{0.25^4}$ $\forall y$, and $g(x, -0.5) = 0$ $\forall x$, while at the outflow boundaries homogeneous Neumann conditions are imposed. We denote by e the difference between the discrete solution

[1] An M-matrix is a non-singular square matrix with all off-diagonal entries ≤ 0 and all real eigenvalues positive. Most proofs for the convergence behaviour of a smoother only hold for M-matrices.

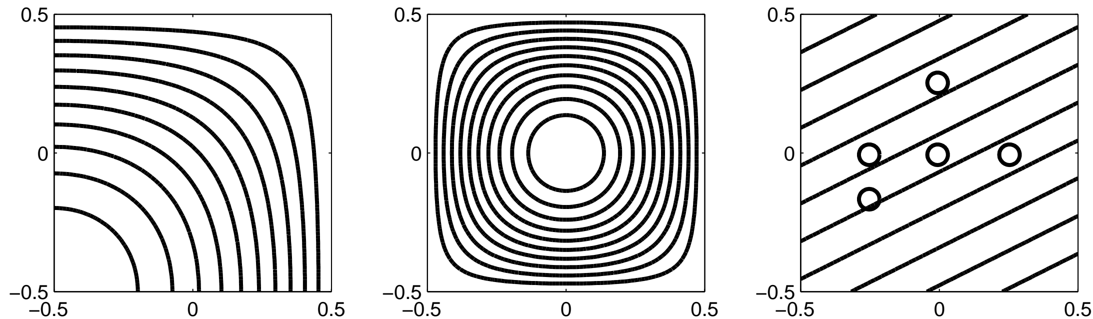


Figure 4.1: Streamlines for the loop-segment, recirculating-flow, and jumping-diffusion coefficient problems.

and the exact analytical solution to the differential equation, sampled on the grid, with e_{max} denoting the maximum norm of this error, $||e||_\infty$.

We also check the accuracy for a problem in which a and b depend on the location (x, y). We use $a = 8y(\frac{1}{4} - x^2)$ and $b = -8x(\frac{1}{4} - y^2)$ (see streamlines for the loop-segment problem in Fig. 4.1), imposing the same inflow boundary condition as above at the left-hand boundary, $g(-0.5, y) = \frac{(y^2-0.25)^4}{0.25^4}$ $\forall y$, homogeneous Neumann conditions at the outflow boundary, $g_n(x, -0.5) = 0$ $\forall x$, and homogeneous Dirichlet conditions at the remaining two boundaries. For $\epsilon \to 0$, the solution u^h at the outflow boundary should exactly equal u^h at the inflow boundary. Therefore we compute $\tilde{e}_{max} = ||u^h_{in} - u^h_{out}||_\infty$ as a measure of accuracy. The results are shown in Table 4.1 (right).

For both problems we see that the error with the second order discretisation, $c(h) = 0$, is almost exactly proportional to h^2, whereas for $c(h) = \frac{1}{2}(a + b)h$ it is proportional to h as expected.

Constant Coefficients			Loop Segment		
$\epsilon = 0$, $a = 1$, $b = 0.5$			$\epsilon = 0$		
$1/h$	e_{max} (1st order)	e_{max} (2nd order)	$1/h$	$\tilde{e}_{max}$ (1st order)	$\tilde{e}_{max}$ (2nd order)
27	0.2788	0.0108459	27	0.1587	0.0025276
81	0.1249	0.0011998	81	0.0623	0.0002803
243	0.0466	0.0001337	243	0.0220	0.0000311
729	0.0161	0.0000149	729	0.0075	0.0000035

Table 4.1: Maximal discretisation error for first order $c(h) = \frac{1}{2}(a + b)h$ and second order $c(h) = 0$ discretisation, both for the constant coefficient and the loop segment problem.

Robustness and Efficiency of the Multigrid Solver

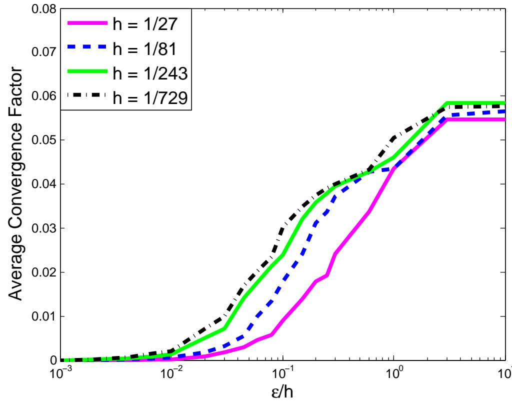
The following numerical experiments demonstrate the robustness and the convergence behaviour of our multigrid solver.

Constant Coefficients and Loop Segment As in the previous subsection we consider the convection-diffusion equation with constant and non-constant coefficients. For assessing the convergence behaviour of our multigrid solver we use the geometric-mean convergence factor ρ_{mean}. Figure 4.2 shows ρ_{mean} as a function of the diffusion coefficient with the first order and second order accurate discretisations for various mesh resolutions. For the first order discretisation the convergence factors are uniformly excellent. For the second order discretisation the convergence is still excellent over a wide range of diffusion coefficients, with some deterioration occurring for certain values of ϵ, which gets worse as the grid is refined. As expected, the convergence rates are excellent in the strongly viscous case, and in the other extreme – the strongly convective regime – the relaxation becomes a stable direct solver.

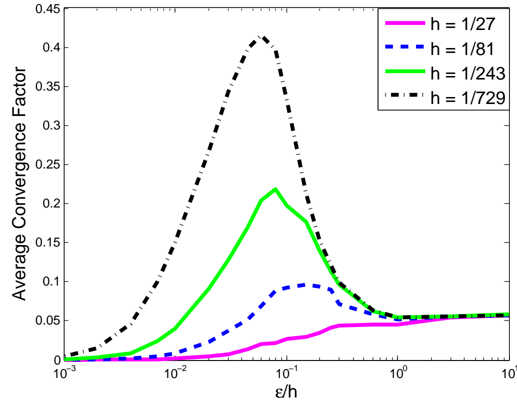
Jumping Coefficients Next, we introduce jumps in the diffusion coefficient for the case of constant convection coefficients by sharply increasing the diffusion in one or more circular regions (see the right-hand panel of Figure 4.1 for the streamlines and positions of the jumps). The rest of the setup remains as before. Tab. 4.2 shows the results for $\epsilon = 10^2$ in one or five circles, respectively, and $\epsilon = 10^{-4}$ in the rest of the domain – a jump by six orders of magnitude. This experiment is motivated by heat flow problems where the property of the conductive material varies in certain regions due to imperfections or embedded materials. See Secs. 2.1.2 and 2.1.3 for an explanation why these kind of problems are so challenging for pure geometric multigrid methods.

As expected for BoxMG methods, the solver that we use appears to be robust. Again, the solver shows better convergence behaviour for the first order discretisation than for the second order discretisation, however, less significantly so. Interestingly, there is no deterioration with the mesh size for the single-jump case. That means that the solver is able to cope with this problem perfectly.

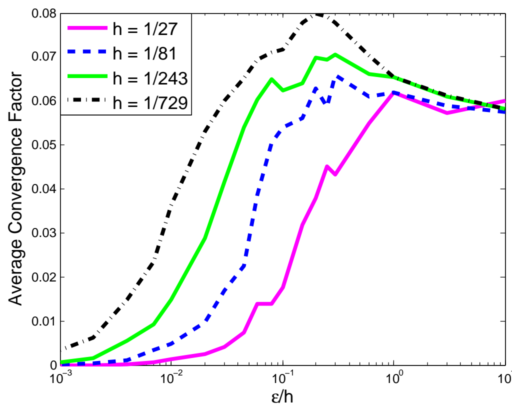
Closed Characteristics The next problem that we consider is a flow with closed characteristics, i.e., recirculating flow. For the convection coefficients we use $a = sin(\pi y)cos(\pi x)$ and $b = cos(\pi y)sin(\pi x)$ and we apply homogeneous Dirichlet boundary conditions on all boundaries. The streamlines are shown in Figure 4.1 (middle). This problem is particularly challenging, as we have no starting point and no end point, and therefore the relaxation cannot "push" the error out of the domain, and in the middle of the domain we have a stagnation point, i.e. the velocity of the flow



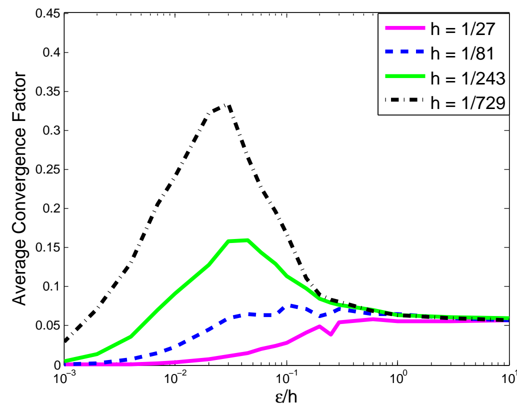
(a) Constant coefficients, first order discretisation.



(b) Constant coefficients, second order discretisation.



(c) Loop segment, first order discretisation.



(d) Loop segment, second order discretisation.

Figure 4.2: The mean convergence factor ρ_{mean} as a function of the diffusion coefficient ϵ for the constant coefficient problem with convection coefficients $a = 1$ and $b = 0.5$, and the loop-segment problem, both with $c(h) = \frac{1}{2}(a + b)h$ (first order discretisation, left) and $c(h) = 0$ (second order discretisation, right). Note the considerable difference of the convergence factor for first and second order discretisation (scale of y-axis).

one jump			five jumps		
$\epsilon = 10^{-4}[10^2]$, $a = 1$, $b = 0.5$			$\epsilon = 10^{-4}[10^2]$, $a = 1$, $b = 0.5$		
$1/h$	# cycles	ρ_{mean}	$1/h$	# cycles	ρ_{mean}
27	10 (11)	0.145 (0.182)	27	8 (12)	0.090 (0.208)
81	11 (13)	0.187 (0.233)	81	13 (17)	0.239 (0.323)
243	9 (10)	0.121 (0.156)	243	16 (17)	0.306 (0.332)
729	8 (9)	0.093 (0.126)	729	21 (22)	0.411 (0.426)

Table 4.2: The constant coefficient problem with one or five jumps by a factor 10^6 in the diffusion coefficient in circular regions of radii 1/27. The numbers in parentheses show results for the second order discretisation, $c(h) = 0$.

is zero. Our results for first and second order discretisation can be seen in Tabs. 4.3 and 4.4. The solver yields excellent convergence behaviour for a diffusion coefficient ϵ down to nearly 10^{-6} for the first order discretisation, and nearly 10^{-4} for the second order discretisation. In this setup we only coarsen down to a 27×27 grid and use a direct solver there. If we coarsen further, then already for $\epsilon = 10^{-4}$ the behaviour in the second order discretisation case is erratic, with occasional divergence. This behaviour can be traced to the formation of very poor operators in the vicinity of the central stagnation point on very coarse grids. In Tab. 4.3 we can observe that for the second order discretisation and a rather small ϵ, i.e. $\epsilon = 10^{-4}$, the two-grid cycle yields rather bad convergence. The three-grid cycle is already significantly better, and with four grids again an improvement is reached. The reason for this behaviour becomes clear when looking at the discretised equation: The convection term contains the factor $\frac{1}{h}$, whereas the diffusion term contains the factor $\frac{1}{h^2}$. The smaller the mesh width h is, the more dominant becomes the diffusion term, and the better is the convergence behaviour of the solver for the second order discretised equation. This effect is not that dominant for the less exact first order discretisation, as we can see in Tab. 4.3. The first order discretisation is more robust than the second order discretisation, and the results are excellent even for small ϵ and very fine grids, also when we coarsen down to 9×9.

$\epsilon = 10^{-4}$			$\epsilon = 10^{-3}$		
$1/h$	# cycles	ρ_{mean}	$1/h$	# cycles	ρ_{mean}
81	4 (74)	0.009 (0.779)	81	7 (8)	0.050 (0.098)
243	7 (20)	0.062 (0.390)	243	7 (8)	0.072 (0.083)
729	9 (14)	0.122 (0.266)	729	7 (7)	0.069 (0.069)

Table 4.3: Convergence rate results for the recirculating flow problem with varying mesh sizes. The numbers in parentheses refer to the second order discretisation.

$1/h = 243$		
ϵ	# cycles	ρ_{mean}
10^0	7 (7)	0.059 (0.059)
10^{-1}	7 (7)	0.060 (0.059)
10^{-2}	7 (7)	0.063 (0.062)
10^{-3}	7 (8)	0.072 (0.083)
10^{-4}	7 (20)	0.062 (0.390)
10^{-5}	9 (>200)	0.125 (0.926)
10^{-6}	20 (DIV)	0.400 (DIV)
10^{-7}	59 (DIV)	0.730 (DIV)

Table 4.4: Convergence rate results for the recirculating flow problem and a variety of diffusion coefficient values ϵ. The numbers in parentheses show results for the second order discretisation. If the method converges slowly we calculate ρ_{mean} after 200 V-cycles.

Comparison to Galerkin Approach To conclude this part, we compare the nonsymmetric BoxMG solver employing the Petrov-Galerkin approach, to the more common approach of Galerkin coarsening (comparisons of BoxMG to a solver using bilinear interpolation and full weighting as restriction are shown in Sec. 4.3.6). To this end we perform numerical tests with two Galerkin variants (i.e., $P = R^T$). In the first case, we employ the restriction operator of the nonsymmetric BoxMG solver, which tends to the upstream restriction in the convection-dominated regime, and its transpose as the prolongation. In the second case, we employ the prolongation operator of the nonsymmetric BoxMG solver, which is based on the symmetric part of the operator, and its transpose as the restriction. As we can see in Tab. 4.5, the first alternative, denoted as Galerkin 1, yields stable coarse grid operators, but slow convergence in intermediate ϵ regimes, where convection is dominant but the diffusion is still not negligible. The second choice, denoted as Galerkin 2, on the other hand, yields unstable coarse grid operators in the convection-dominated (small ϵ) regime, and hence divergence. The convergence behaviour for these two choices is explained in [109] by the sum of the orders of restriction and prolongation: Yavneh found in [107], that this sum has to be at least three for getting a good coarse grid correction for smooth characteristic components (i.e., components which are much smoother in the characteristic direction than in the direction orthogonal to it) in the case of first order discretisation. For our first alternative for the intergrid transfer operators the sum is two, as both operators are first order. Therefore, we receive poor coarse grid correction in the convection-dominated regime, but we have stable upstream schemes on the coarse grid and therefore still get convergence. In the second case, both operators are second order and we get a sum of four, but

$1/h = 243$, $a = 1$, $b = 0.5$			
ϵ	ρ_{mean}^{BoxMG}	$\rho_{mean}^{Galerkin1}$	$\rho_{mean}^{Galerkin2}$
10^{0}	$5.79 \cdot 10^{-2}$	$5.78 \cdot 10^{-2}$	$5.79 \cdot 10^{-2}$
10^{-1}	$5.86 \cdot 10^{-2}$	$5.86 \cdot 10^{-2}$	$5.86 \cdot 10^{-2}$
10^{-2}	$5.70 \cdot 10^{-2}$	$6.55 \cdot 10^{-2}$	$5.79 \cdot 10^{-2}$
10^{-3}	$3.81 \cdot 10^{-2}$	$1.51 \cdot 10^{-1}$	DIV
10^{-4}	$6.10 \cdot 10^{-3}$	$3.80 \cdot 10^{-2}$	DIV
10^{-5}	$7.05 \cdot 10^{-5}$	$7.01 \cdot 10^{-4}$	DIV
10^{-6}	$9.71 \cdot 10^{-7}$	$7.03 \cdot 10^{-6}$	DIV

$1/h = 243$			
ϵ	ρ_{mean}^{BoxMG}	$\rho_{mean}^{Galerkin1}$	$\rho_{mean}^{Galerkin2}$
10^{0}	$6.00 \cdot 10^{-2}$	$5.93 \cdot 10^{-2}$	$5.90 \cdot 10^{-2}$
10^{-1}	$5.98 \cdot 10^{-2}$	$5.87 \cdot 10^{-2}$	$5.94 \cdot 10^{-2}$
10^{-2}	$6.35 \cdot 10^{-2}$	$1.14 \cdot 10^{-1}$	$6.21 \cdot 10^{-2}$
10^{-3}	$1.16 \cdot 10^{-1}$	$5.51 \cdot 10^{-1}$	DIV
10^{-4}	$1.50 \cdot 10^{-1}$	$7.04 \cdot 10^{-1}$	DIV
10^{-5}	$5.01 \cdot 10^{-1}$	$7.36 \cdot 10^{-1}$	DIV
10^{-6}	$5.51 \cdot 10^{-1}$	$6.97 \cdot 10^{-1}$	DIV

Table 4.5: Convergence rate results for the constant coefficient (top) and the recirculating flow problem (bottom) for various diffusion coefficients. The second column (ρ_{mean}^{BoxMG}) shows the mean convergence factor for our algorithm, the third column ($\rho_{mean}^{Galerkin1}$) for $P = R^T$ with upstream R, and the last column ($\rho_{mean}^{Galerkin2}$) for $R = P^T$ and P created with the symmetric part of A. In all three cases we use $c(h) = 0.5h(a + b)$ (standard first order upstream), and we coarsen down to 9×9.

the coarse grid operators get more and more unstable in the convection-dominated regime if we coarsen, resulting in divergence. For our nonsymmetric BoxMG solver, the sum is exactly three in the convection-dominated regime, and we have shown (see Sec. 4.2.3) that we get stable coarse grid operators in the convection-dominated regime. Therefore, our solver is robust and yields excellent convergence factors.

4.3 Element-by-Element Multigrid and BoxMG

We now move some steps towards the target implementation on a spacetree by developing a method that already fulfils the basic requirement: The operations have to be implemented in a strict cell-wise (i.e., element-by-element) manner. The BoxMG method naturally fits very well to this principle, as the intergrid transfer operators are defined in such a way, that no information outside the respective coarse grid cell is used.That means, that we only need a node and its children from the spacetree to be available at the same time (see Sec. 3.2). But two challenges have to be overcome for our solver: First, the Petrov-Galerkin coarse grid operator for arbitrary prolongation and restriction operators has to be computed in this cell-wise setting, and second, a suitable smoother has to be found that is both as local as necessary and as powerful as possible. The resulting code is our second prototype, Prototype 2.

Element-by-element finite element methods were introduced in the eighties [58, 57, 105, 94]. The basic idea is to define the operations in an element-based manner, i.e., an operator is applied on a single element at a time and only uses information locally available from that element, and these local parts are then accumulated. Already in the beginning, part of the motivation behind was to get algorithms that are easy to parallelise: You do not have one big operator that has to be stored and that is applied in one expensive sequential step, but a lot of small, maybe distributedly stored, operators that can be applied in parallel. In comparison to applying the whole operator as one matrix, or applying it row-wisely as stencils which are located at the vertices, the element-wise approach has also the advantage that no ghost-cell layers at the processor boundaries are needed. This concept during the last years was combined with spacetrees as underlying structure in order to achieve efficient matrix-free methods (see, e.g., [99, 81, 41]).[2]

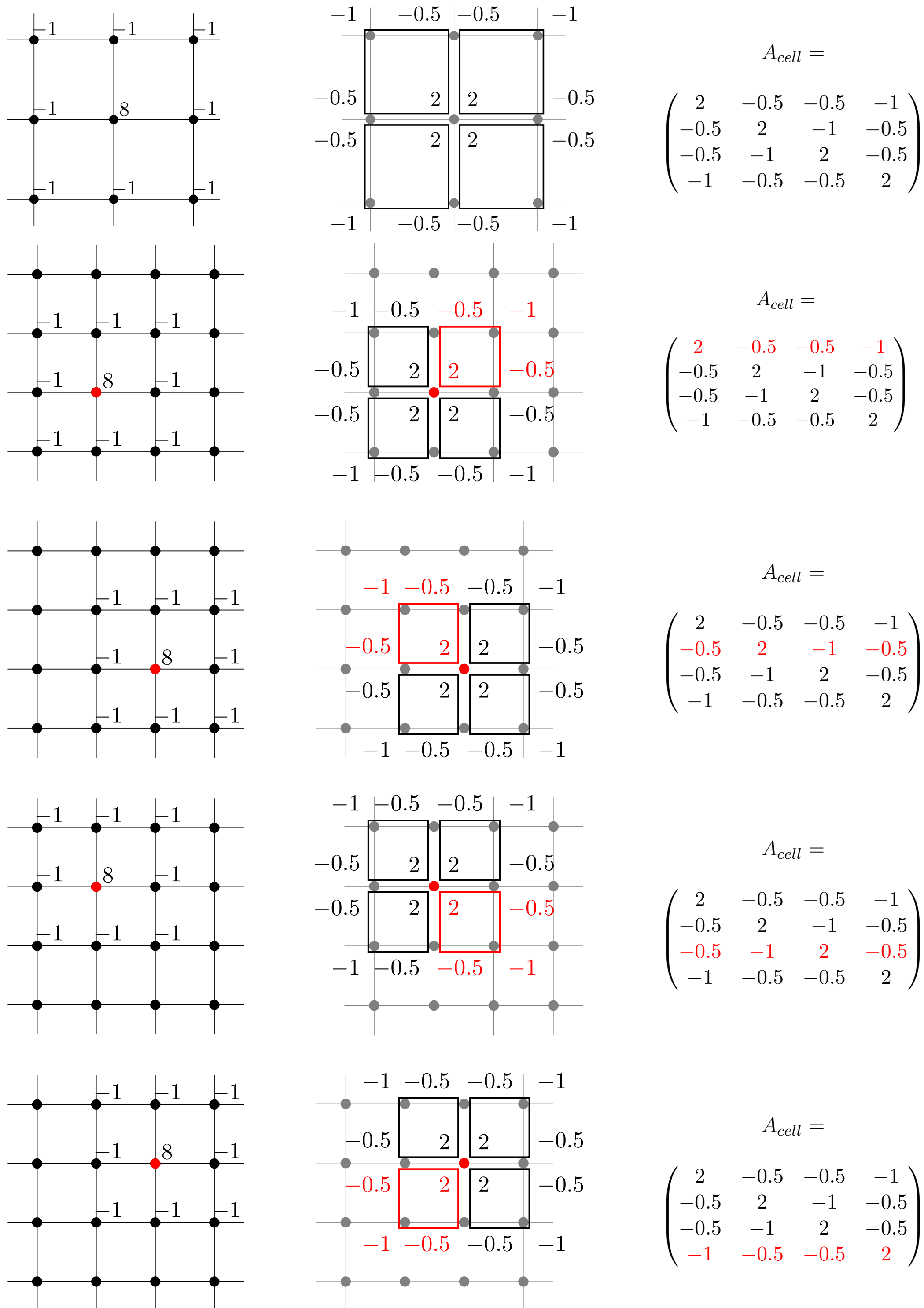


Figure 4.3: The nine-point Laplace stencil (left column) is split into its cell-wise components (middle column). If this is done for all vertices of a cell (in this example, the stencil is assumed to be the same on all vertices), we can assemble its cell-based system operator (right column).

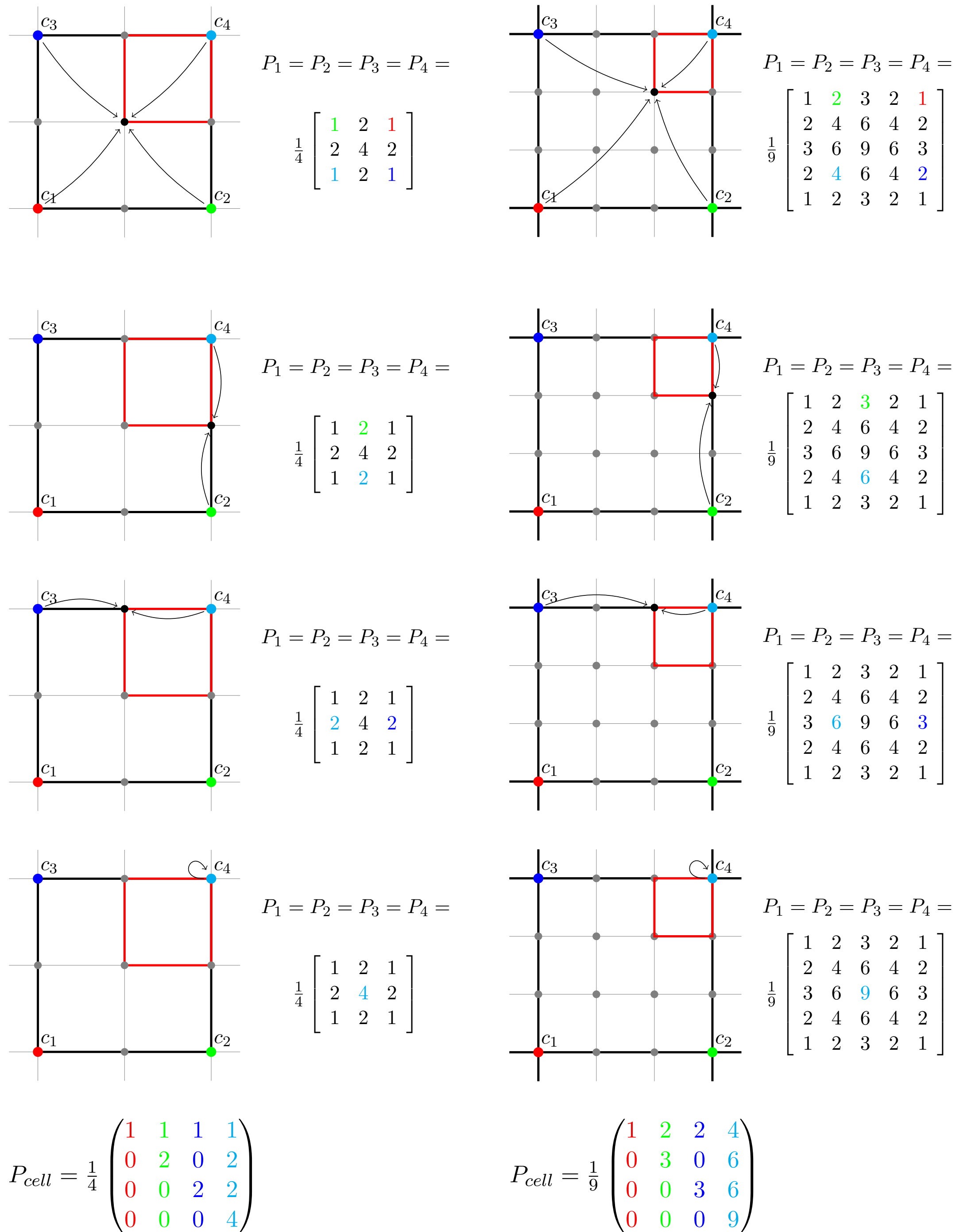


Figure 4.4: Assembling of the cell-based prolongation operator for one fine grid cell in a bisected (left) and trisected (right) coarse grid cell. The prolongation operators located at the four coarse grid vertices are the same in this example. The respective values from the prolongation operator are highlighted in each step. If a coarse grid vertex does not influence a fine grid vertex the corresponding value is 0.

4.3.1 The Cell Operators

The element-by-element principle used in this work is illustrated for 2D in Figs. 4.3 and 4.4: The cell-based system operator A_{cell} and intergrid transfer operators P_{cell} and R_{cell} are derived from the stencil operators of all vertices belonging to the cell. This results in $2^D \times 2^D$ matrices. For A_{cell}, this was already described before, e.g. in [99]. We define now a splitting for the intergrid transfer operators in a manner that makes a cell-wise application and an elegant computation of the Galerkin coarse grid operator possible for arbitrary P and R (see below).

Similar to A_{cell}, the values of the intergrid transfer operators have to be scaled – see Fig. 4.5. One can equivalently transfer the operators back from cell representation to vertex representation. For that, of course all respective cell operators are needed (i.e., one vertex operator is assembled from several – in 2D four – cell operators).

The evaluation of such a cell operator is a matrix-vector product (MatVec) – in 2D, for example, a 4×4 matrix times a 4×1 vector (u_{cell}, containing the values of the four vertices of a cell). We receive a vector containing the contributions to the vertices belonging to the cell. In order to get the result on a vertex, the corresponding vector entries of all adjacent cells have to be accumulated: For A_{cell}, the stencil is split and put on the adjacent cells of the central vertex (see Fig. 4.3). A row of A_{cell} contains the weights for the operator centred at a vertex, a column contains the weights of all four operators touching a vertex in that cell. Therefore, we can simply sum up all four ($= 2^D$) vector entries of the result vectors $A_{cell}u_{cell}$ of the four ($= 2^D$) adjacent cells to get the new value of the central vertex. Instead of nine values, as with the standard vertex-based nine-point stencil, we now have to sum up sixteen ($= 2^D \times 2^D$) values.

For the intergrid transfer operators P_{cell} and R_{cell}, the operator splitting is not as intuitive as for A_{cell}, because we have to think in two levels, the coarse and the fine grid. A vertex does not directly influence its neighbours, but the respective vertices at the next coarser or finer level. That means in practice that for the intergrid transfer the corresponding nodes of two spacetree levels have to be available.

P_{cell} contains in a row the contribution weights of all coarse grid vertices to one fine grid vertex. In a column, for one coarse grid vertex the contribution weights to all fine grid vertices of the fine grid cell are stored. For R_{cell} this holds in a transposed way: In a row, a coarse grid vertex is fixed, a column corresponds to a fixed fine grid vertex. If $R = P^T$ (in matrix notation, equivalent to $R = P$ in stencil notation), then $R_{cell} = P_{cell}^T$. For the prolonged value of a fine grid vertex, the respective entries of $P_{cell}u_{coarse-cell}$, which can be viewed as located on the vertices of

[2]The term “matrix-free” is used for methods that do not store the system matrix as one big global matrix, although in fact also these methods store the whole operator implicitly, distributed over the domain. It is also used for methods where the matrix is in fact not stored. Jacobian-free Newton-Krylov methods, for example, just store an approximation of the matrix-vector product that they need. See [63] and references therein for further information.

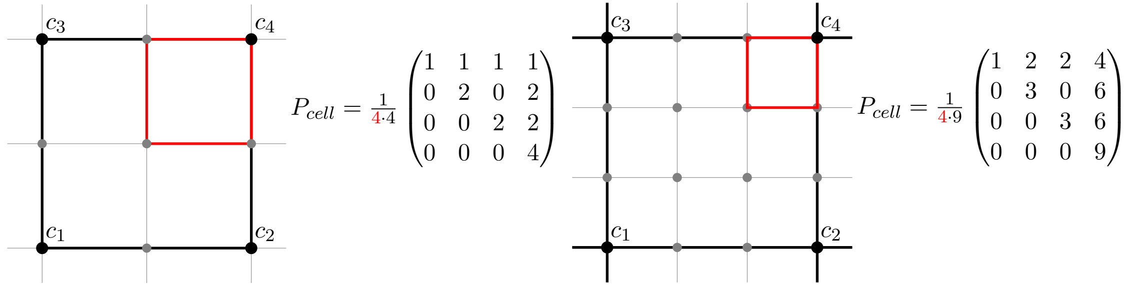


Figure 4.5: The cell-based prolongation operator has to be scaled according to the number of cells adjacent to one vertex. This is not necessary when using it for the computation of the Petrov-Galerkin coarse grid operator, as we compute $A^H = RA^hP$ in a cell-wise manner and A^h_{cell} is already scaled accordingly (see Fig. 4.3).

the fine grid cell, of the four adjacent cells of the vertex are added up. These are four summands: We only sum up those values which lie directly on the respective vertex. The accumulation for the stencil application was already done with the MatVec. To get the restricted value for a coarse vertex, we first accumulate the results of $R_{cell}u_{fine-cell}$ for all fine grid cells that are covered by the restriction stencil on the vertices of the coarse grid cell. Then again, the vertex value is received as the sum of the four values at that vertex.

To summarise, we store the system operator as stencil at the fine grid vertices and the intergrid transfer operators as stencils at the coarse grid vertices. For application, the cell operators are assembled from those stencils. The unknowns u and the right-hand sides f are stored at the vertices. We now discuss the following alternatives to this scheme:

1. storing u and f in 4×1 vectors at the cells,
2. storing P and R cell-wisely as P_{cell} and R_{cell} at the fine grid cells,
3. storing P and R as 4×1 vectors at the fine grid vertices,
4. not storing P and R at all, but re-compute them everytime from A,
5. and implementing prolongation and restriction vertex-wisely.

The considerations 2 and 3 are valid in a similar manner also for A.

Alternative 1: Using the cell-wise operators, we seem to be able to do a whole multigrid cycle in cell-view. So one might ask why we do not only store and process vectors at cells instead of accumulating the vertex values. When talking about restriction and prolongation, this would work. We can also compute the Galerkin

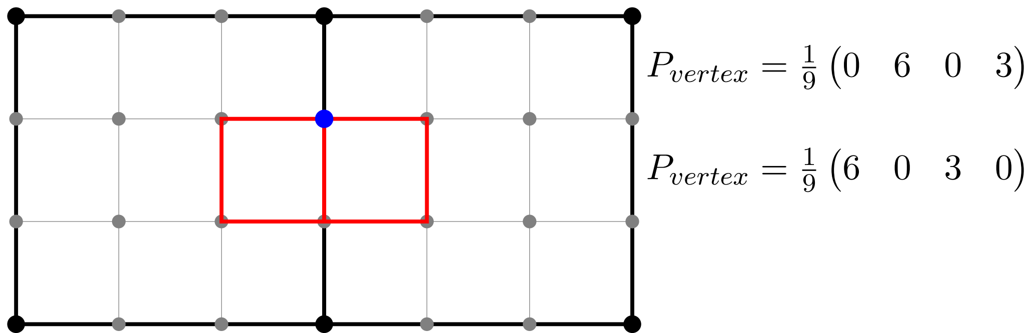


Figure 4.6: For the highlighted fine grid vertex on the coarse cell boundary, the respective lines in the prolongation operator matrices contain the same values, but not in the same order, as the numbering of the coarse grid vertices is different.

operator completely cell-wisely (see Sec. 4.3.2). But when applying the system stencil, for example in smoothing, information has to be transferred through the grid. In cell-view, the value at the bottom left vertex in Fig.4.3, for example, does only affect the other three vertices of that cell (and itself). But in order to let information go to the top right vertex, e.g., information has to be exchanged between the cells in the next step. This is why the values for u (and also for the right-hand side f when doing the restriction of the residual) are accumulated and stored at the vertices.

Alternatives 2 and 3: At first sight, storing P and R at the coarse grid vertices seems to be unnecessary: Why not storing them as 4×4 cell operators on the fine grid cells, or as vectors of length four on the fine grid vertices? Storing them on the fine grid cells is possible and can be done without restrictions. The obvious drawback, however, is, that we store the values redundantly: One vertex contribution from the original operator P or R has to be stored in four different cell operators. Now one could argue that if we stored the values as vectors on the fine grid vertices, we could simply assemble the cell operators from them without having this redundancy. This would work for the vertices inside a coarse grid cell. The problem for vertices at the coarse cell boundaries, however, is shown in Fig. 4.6: As the numbering of the coarse grid vertices is different in the two coarse grid cells, the two respective coarse grid vertex contributions have the the positions 2 and 4 in the left hand cell, and 1 and 3 in the right-hand cell. That means the order of the values in a vector at a fine vertex should be different depending on whether we regard the fine grid vertex as belonging to the left- or right-hand cell. If we would store this additional information somehow, or introduce some kind of rule (we always store the values in the order given by the left-hand cell, for example), the treatment of this boundary cases would cause an overhead that would annihilate the benefit of this data structure.

Alternative 4: Especially for the implementation on supercomputers, where we have restricted memory per core, but a lot of computing power [3], one might decide to not store P and R at all, but, if possible, recompute them on-the-fly every time for the fine grid cells inside a coarse grid cell. For the operator-dependent intergrid transfer operators defined by BoxMG, this is easily possible – see Sec. 4.3.3.

Alternative 5: By using the information transfer between the grid levels, the intergrid transfer operators could also be implemented in a vertex-wise manner, with the parent distributing/collecting the respective fractions to/from the children. As discussed in Chap. 3, the spacetree allows us to directly access all children of a node. This means that we have one coarse grid cell with all its children (i.e., four resp. nine fine grid cells) available. We could then update the (fine for prolongation, coarse for restriction) vertices according to the intergrid transfer operators on the four coarse vertices. For prolongation, the contributions of all four coarse vertices have to be accumulated on the fine vertex. For restriction, the value at a coarse vertex is valid when all four adjacent cells of a coarse vertex have been processed and the respective contributions of the fine vertices have been accumulated. However, that way we have to take care not to accumulate the values from vertices lying on the coarse cell boundary twice (or four times for a coarse vertex), and the elegant implementation of the Petrov-Galerkin principle as described in the next section would not be possible.

4.3.2 Element-by-Element Petrov-Galerkin

The Petrov-Galerkin coarse grid operator computation can completely be done cell-wisely, as shown in Alg. 4.1 for one coarse grid cell. The input parameters for the algorithm are the fine grid system operators and the intergrid transfer operators of all fine grid cells inside the respective coarse grid cell in cell representation, the output is the coarse grid operator in cell representation. As pointed out above, one can easily transfer the operators on the grid from one representation to the other. Therefore, if we want to compute A_{coarse} on a vertex from $A_{coarse-cell}$, we need $A_{coarse-cell}$ from the four adjacent cells.

We can see in Alg. 4.1, that by defining the cell operators as given above, we keep the form $A_{coarse} = R \cdot A_{fine} \cdot P$ for the Petrov-Galerkin coarse grid operator in cell representation.

[3] On modern supercomputers, the overall memory and the number of cores is increasing, but the memory per core stagnates. This is on the one hand due to runtime issues – the bigger the memory is, the slower is the memory access –, on the other hand due to energy issues – holding data in the memory needs more energy than doing computations. In addition, communication and parallelisation facilities like MPI [72] also need a lot of memory.

Algorithm 4.1 Cell-Wise Petrov-Galerkin

```
function PETROV-GALERKIN(A_{fine-cells}, P_{fine-cells}, R_{fine-cells})
    A_{coarse-cell} = Matrix4x4(zeros)
    for all fine grid cells fc do
        A_{coarse-cell} += R_{fc} · A_{fc} · P_{fc}
    end for
    return A_{coarse-cell}
end function
```

4.3.3 Element-by-Element BoxMG

For the computation of the BoxMG intergrid transfer operators as described in Sec. 2.3, we need two spacetree levels available: the coarse grid cell and its children. This is all we need for solving the respective equation system. Due to the stencil collapsing at the coarse-cell boundaries, we do not need any information from neighbouring cells. Therefore, the BoxMG algorithm is perfectly suited for implementation on a spacetree (Chap. 3).

The intergrid transfer operators can be computed in a setup phase and stored either as 5×5 stencil on the coarse grid vertices (then we have to assemble the cell operators every time for restriction and prolongation, and also for the computation of the (Petrov-)Galerkin coarse grid operator) or as 4×4 cell operator on the fine grid cells – see Sec. 4.3.1. However, due to memory considerations (especially on supercomputers) or in dynamically adaptive simulations, it might make sense not to store them permanently, but recompute the operators every time we enter a coarse grid cell (and go down to its children), and discard them when we leave the cell. That way, we never have the full five-times-five intergrid transfer stencils available, but only the four three-times-three parts that overlap the active coarse grid cell.

4.3.4 Smoothers for Element-by-Element Multigrid

Finding a suitable smoother (Sec. 2.1.1) is not trivial for strict cell-wise spacetree multigrid. In Sec. 3.2, we already mentioned the impact of the spacetree structure (when staying in strict cell-wise manner) on the way of doing computations. One issue were the restrictions posed on the information transfer: From a given node in the tree, only direct access to its parent and its children (and maybe its siblings) is possible. If two or more neighbouring cells in the spacetree grid share one vertex, we do not know in which order the cells are traversed, and there might lie several cells in between in the order of traversal. When applying one of the cell-wise operators described in the previous section, without further information (e.g. a counter which keeps track of the number of "visits" of a vertex), only at the end of a complete traversal of the spacetree grid we can be sure that all adjacent cells of a vertex were

visited and their cell operators were applied, and therefore the vertex holds a valid value. This makes it often necessary to do several traversals in order to perform an algorithm, what makes some algorithms almost unfeasible: For a point Gauss-Seidel smoother, which is one of the most popular smoothers for multigrid, we would need to do an extra traversal for the update of every value, as always the most “up-to-date” residual is needed, and the residual at a vertex is “outdated” as soon as one of the neighbouring vertices is updated. Therefore, the residual $r = f - Au$ has to be recomputed after every update, and this requires the application of the system operator stencil. To apply A on u at a vertex in a cell-wise setting, however, we need to visit all adjacent cells of that vertex – also those which lie “behind” it in the current traversal.

A point Jacobi smoother is easy to implement on spacetrees, as the residual is computed once in advance at every vertex and then used unchanged during the traversal. We can therefore either do the Jacobi smoothing in two traversals (one for the residual computations and one for the updates), or in one traversal by tracking whether already all adjacent cells at a vertex where visited and doing the smoothing update as soon as the value at the vertex is valid. Because of the use of the “outdated” residuals, however, Jacobi smoothing is often not efficient enough for challenging problems.

An “affordable”, but not cheap, alternative would be red-black Gauss-Seidel with four colours, as we use a nine-point stencil. This smoother needs one Jacobi smoothing iteration per colour. Red-black Gauss-Seidel is a better smoother than Jacobi, but we would need four times as many traversals.

Line relaxation, as used in [35] and [109], is again not feasible on spacetrees, as a tridiagonal system has to be solved for each line, and we therefore need all involved vertices to be accessible at the same time (or a lot of traversals for doing it somehow else).

When examining this problem, one can observe that we have a situation that is somehow related to the situation when looking for a smoother on a parallel computer: We only have restricted communication possibilities and therefore have to find a smoother that acts more or less locally. In a parallel setting using domain decomposition, the inner part of a subdomain can exchange information easily, but at the domain (i.e., processor) boundaries information exchange is expensive and only possible in a restricted manner (for example once per grid traversal). On a spacetree, the subdomain where information exchange is easily possible is defined by the children of a node, or, more exactly in terms of the spacetree grid, by the inner fine grid vertices of a coarse grid cell. The fine grid vertices on coarse cell boundaries already need information from the neighbouring coarse cell when applying the stencil for smoothing. Having these similarities in mind, it makes sense to have a look at the literature dealing with parallel smoothers, see e.g. [1, 69, 70, 27, 7, 8] and references therein. Obviously, parallel smoothing is still an open field of research.

One approach that is often considered in this context are hybrid (Gauss-Seidel) smoothers.

Hybrid Gauss-Seidel smoothers perform Gauss-Seidel relaxation in the inner part of the domain where communication is possible – i.e., on a processor or, in our case, in the interior of a coarse cell. On the boundaries, Jacobi updates are performed. Note that this will reduce to a block-Jacobi smoother if we do a lot of Gauss-Seidel iterations at the inner points. These smoothers do not always yield convergence, especially if the system operator matrix is not diagonal dominant and the blocks are rather small [7, 1]. Still, hybrid Gauss-Seidel smoothers are considered to be robust and efficient in most practical cases [7].

In the case of spacetree grids, the blocks on which Gauss-Seidel relaxation can be performed are very small. For coarsening by a factor of two, only the single point in the middle of the coarse grid cell is guaranteed to already hold a completely updated value when leaving the coarse grid cell, for coarsening by a factor of three, the four interior points. We will see in our experiments, however, that the hybrid smoother for coarsening by a factor of three yields some improvement as compared to a Jacobi smoother.

Combining the hybrid smoothing idea with the scheme used for the computation of the BoxMG intergrid transfer operators, we propose the following relaxation scheme which can be applied on a spacetree (see also Fig. 2.3 for nomenclature of the points):

- First, perform Jacobi smoothing on the c points (fine grid points lying on coarse grid positions) and γ points (fine grid points lying on coarse grid lines). This step uses the precomputed residual at all respective vertices.
- Relax γ points with the residual computed with the collapsed (1D) stencil (as in Eq. 2.9) in a Gauss-Seidel manner, i.e., using the updated values from the c points and maybe the γ point at that coarse grid line updated before.
- Perform Gauss-Seidel relaxation on the ι points (fine grid points in the interior of coarse grid cells) using the updated values at the c and γ points and maybe the ι points of that cell updated before.

We call this scheme a "box smoother", as it works on a box-shaped domain and it uses the BoxMG idea of collapsed stencils at γ points. It differs from the standard hybrid approach in that we do an additional "kind-of-Gauss-Seidel" step at the γ points. An alternative would be to leave out the Jacobi step at the γ points, but in that case we would disregard the information about the "real" residual (i.e., the residual computed with the 3×3 stencil) that we have anyway. The scheme can be applied in two grid traversals: First, the residual is computed for all vertices, then the smoothing is performed for all coarse grid cells. As all sixteen vertices of the coarse grid cell have to hold valid values for the residual (i.e., all their adjacent cells have to be visited) before a coarse grid cell can be processed it does not make sense

to apply the scheme in one traversal. In order to avoid to do the update at the γ points twice and at the c points four times, i.e., for every adjacent coarse grid cell, we either have to track whether a point was already updated during the traversal (by a bit flag, e.g.), or we simply divide the γ update by two and the c update by four.

In [35], a similar scheme was proposed and called "pattern relaxation". In contrast to our scheme, they propose to solve the ι and γ system directly in a block Gauss-Seidel manner after doing the update at the c points, and they do no additional Jacobi step at the γ points. They report that the convergence rates of a V(1, 1) cycle lies in between those of a V(1, 1) and a V(2, 2) cycle with red-black point Gauss-Seidel as smoothing method.

By traversing the grid along the space-filling curve first in one direction and then in the other, we can get a symmetric smoother.

Another type of smoothers often discussed for parallel applications are polynomial smoothers, as e.g. Chebyshev smoothers (see [7, 1] and references therein). Here, the update rule $u^{k+1} = u^k + p(A)(b - Au^k)$, with $p(A) = \sum_{0\leq j\leq n} \alpha_j A^j$ is used. The choice $n = 0$ reduces this rule to the Jacobi smoother. The computation of the coefficients α_j, however, is based on an estimation of the eigenvalues of A, what makes these type of smoothers again unattractive for our purposes.

4.3.5 Diffusion Equation

Considering the fact that we cannot easily use line smoothers or similar on spacetrees (see discussion above), we ease our model problem from Sec. 4.2.1 a bit. We test our Prototype 2, which implements the cell-wise operators as described above, using the two-dimensional Poisson equation with variable coefficients:

$$\begin{aligned} Au = -\nabla(\epsilon\nabla u) = -\alpha\frac{\partial^2 u}{\partial x^2} - \beta\frac{\partial^2 u}{\partial y^2} &= f, \qquad && (x,y) \in \Omega, \\ u_n = k, \quad \text{or} \quad u &= g, \qquad && (x,y) \in \partial\Omega. \end{aligned} \tag{4.4}$$

Here, $\epsilon = (\alpha, \beta)$ and α and β are the diffusion coefficients in the x and y direction, respectively. The given functions $f(x, y)$ and $g(x, y)$ (or $k(x, y)$) are the right-hand side forcing and the boundary condition, respectively, and u_n denotes the derivative of u in the direction normal to the boundary. Our variable is $u(x, y)$, which describes, for example, for a fixed point in time the heat expansion in a material which may have different diffusion properties in x and y direction, or the charge in a conductive material. In our experiments, we use an isotropic problem, i.e. in terms of the diffusion part of Eq. 4.1, $\epsilon = \alpha = \beta$, and impose different patterns of jumps in the diffusion coefficients (see Fig. 4.7). This is enough to pose some difficulties for pure geometric multigrid using bilinear interpolation and full weighting as restriction –

see explanation in Secs. 2.1.2 and 2.1.3 and the comparison of BoxMG intergrid transfer operators and bilinear interpolation in Fig. 4.10.

We discretise our problem at the vertices of a Cartesian grid, using finite elements. The coarse grid operators are constructed using the Galerkin method (see Sec. 2.1.3), i.e., $A^H = P^T A^h P$. The implementation is for a Petrov-Galerkin method, however, we use $R = P^T$ in our experiments.

4.3.6 Smoother Experiments

We now present results for a BoxMG solver with the cell-wise operators as described above. The results are for multigrid with coarsening by a factor of three. We did not implement the spacetree data structure and traversal by a space-filling curve for this prototype, and therefore can easily do comparisons of different smoothers, including Gauss-Seidel relaxation. However, as all underlying principles for a spacetree implementation and a respective traversal are implemented, the integration into such a framework is straightforward, as we will see in Chap. 5.

The domain in the following experiments is $\Omega = [0,1] \times [0,1]$. We show results for V-cycles with different smoothers and different numbers of pre- and postsmoothing steps. We coarsen down to a 3×3 grid and solve the problem exactly there. As before, we start with a random initial guess and impose a zero right-hand side. Again, the iteration is stopped when the l_2 residual norm $||r_i|| = ||f^h - A^h u_i^h||$, with i denoting the iteration number, is reduced at least by a factor of 10^8. The number of iterations needed for reaching this goal and the geometric mean ρ_{mean} of the convergence factor ρ (see Sec. 4.2.5) is tracked. If more than 200 V-cycles are needed for meeting the stopping criterion, we report the convergence factor after 200 cycles. If not denoted otherwise, results are given for the BoxMG implementation. Bilinear interpolation/full weighting comparative measurements are explicitly declared as such.

The scenarios used in the experiments are shown in Fig. 4.7: A simple Poisson problem with diffusion coefficient equal to one in the whole domain, a problem with a vertical jump of a factor 10^3 in the diffusion coefficient, and a "minimal checkerboard"-like configuration, also with a jump of a factor 10^3. Finally we have the layer problem, where the diffusion jump by a factor 10^{-10} is in a vertical layer of width h. In all these problems, an additional difficulty occurs for multigrid methods using bilinear interpolation/full weighting when the jump is not exactly at 1/3, but shifted, for example, by h, such that the coarsening happens across the discontinuity.

Convergence Behaviour

Finding the optimal damping parameter ω for the relaxation method is a critical task. Instead of doing a complex smoothing analysis (see, e.g., [95]), we simply do

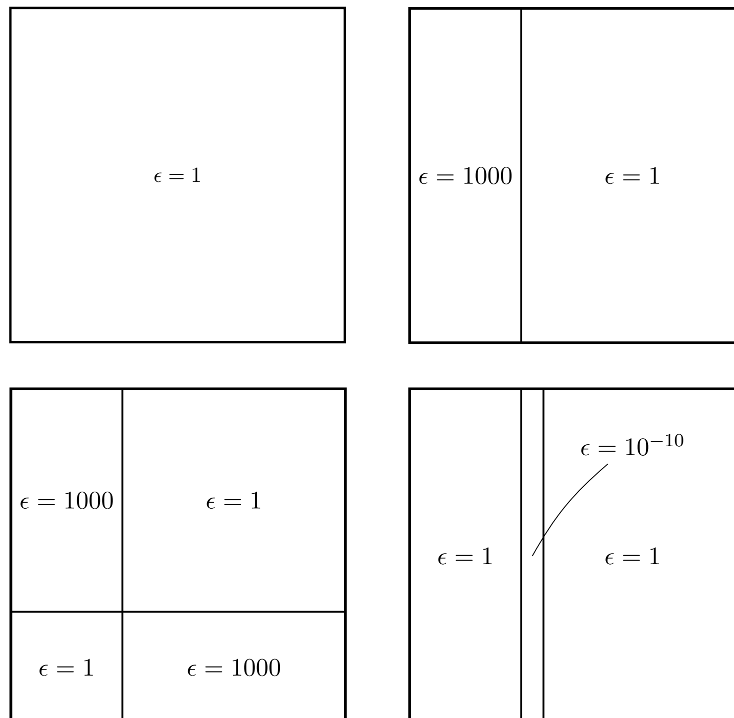


Figure 4.7: Diffusion coefficients for the homogeneous Poisson problem (top left), the coefficient jump problem (top right), the shifted minimal checkerboard problem (bottom left) and the layer problem (bottom right).

an exhaustive search for the optimal damping factor in a reasonable interval (we chose $[0.5, 2.0]$ in the most cases, as this turned out to be the interval for which convergence could be obtained for all experiments and a clear minimum lay within the limits). We use a step width of 0.05 for stepping through the interval and apply the BoxMG solver for each choice of ω.

Fig. 4.8 shows the convergence behaviour for the simple Poisson problem and the coefficient jump problem on a 27×27 grid. We can observe that the solver shows, as expected, an excellent convergence rate. The optimum for the homogeneous Poisson problem lies at $\omega = 1.0$ for V(2,2) cycles with a Jacobi smoother, $\omega = 1.1$ for V(1,1) cycles with Gauss-Seidel, and at $\omega = 1.15$ for V(2,2) Gauss-Seidel cycles. In Tab. 4.6, we see that we get the desired multigrid convergence that is independent of the mesh width h (except of a sometimes slightly better convergence of the two-grid cycle, i.e., at the 9×9 grid). As we know, BoxMG results in bilinear interpolation and full weighting for restriction for the homogeneous Poisson equation, so the same results hold for using the standard geometric multigrid method with bilinear interpolation.

GS V(1, 1), $\omega = 1.1$		
$1/h$	# cycles	ρ_{mean}
9	8	0.0863
27	10	0.1387
81	10	0.1506
243	10	0.1482
729	10	0.1475

GS V(2, 2) $\omega = 1.1$		
$1/h$	# cycles	ρ_{mean}
9	5	0.0193
27	6	0.0381
81	6	0.0401
243	6	0.0398
729	6	0.0395

Jac V(2, 2) $\omega = 1.0$		
$1/h$	# cycles	ρ_{mean}
9	9	0.1072
27	9	0.1219
81	9	0.1190
243	9	0.1147
729	9	0.1114

Table 4.6: The number of V-cycles for meeting the stopping criterion and average convergence factor ρ_{mean} for the homogeneous Poisson problem. We used Gauss-Seidel with the optimal $\omega = 1.1$ for the V(1,1) cycles (left) and for the V(2,2) cycles (middle), and Jacobi V(2,2) cycles (right) with $\omega = 1.0$.

The situation is not that ideal for the coefficient jump and the checkerboard problem, as shown in Tab. 4.7. Here, a clear deterioration with an increasing number of levels is visible. There is basically no improvement from the BoxMG method compared to the bilinear interpolation counterpart. If we examine the intergrid transfer stencils, we see, that actually BoxMG again results in something very similar to bilinear interpolation. But still, we get satisfying convergence for a V(2,2) cycle and Gauss-Seidel smoother.

Fig. 4.9 shows that the choice of the optimal damping parameter does not only depend on the smoother we use, but also on the mesh width (and thus the number of levels in the multigrid setting). Therefore, the measurements in Tab. 4.7 do not reflect the optimal convergence behaviour of the method. Nevertheless, the minima do not lie on a horizontal line in Fig. 4.9 – and this is what would be the ideal case for

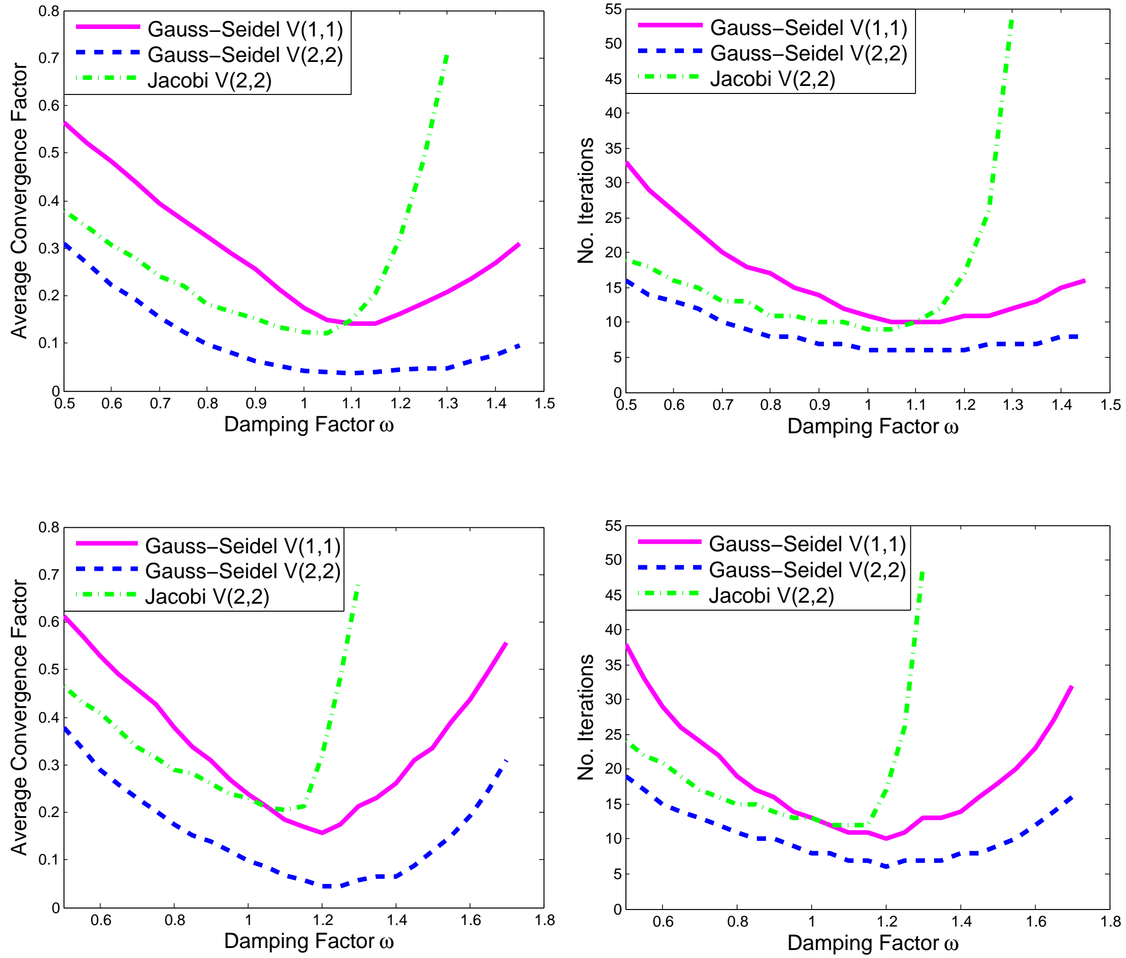


Figure 4.8: The average convergence factor ρ_{mean} (left) and the number of V-cycles for meeting the stopping criterion (right) for the Poisson problem with diffusion coefficient $\epsilon = 1.0$ (top) and for the coefficient jump problem problem, both on a 27×27 grid.

GS V(1, 1) $\omega = 1.2$		
$1/h$	# cycles	ρ_{mean}
9	8	0.0955
27	10	0.1524
81	52	0.7016

GS V(2, 2) $\omega = 1.2$		
$1/h$	# cycles	ρ_{mean}
9	5	0.0138
27	6	0.0453
81	26	0.4915

Jac V(2, 2) $\omega = 1.1$		
$1/h$	# cycles	ρ_{mean}
9	8	0.0892
27	12	0.2039
81	70	0.7682

Table 4.7: The number of V-cycles for meeting the stopping criterion and average convergence factor ρ_{mean} for the coefficient jump problem. We used Gauss-Seidel with the optimal $\omega = 1.2$ for the V(1,1) (left) and the V(2,2) (middle) cycles and $\omega = 1.1$ for the Jacobi V(2,2) cycles (right).

a multigrid solver. We notice that the optimal overrelaxation factor ω increases with a decrease of the mesh width, and therefore with the deterioration of the multigrid convergence.

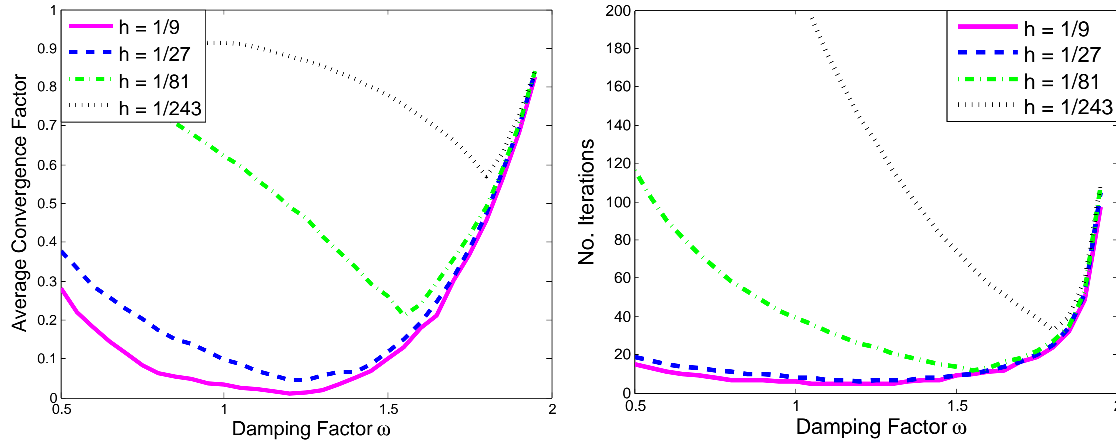


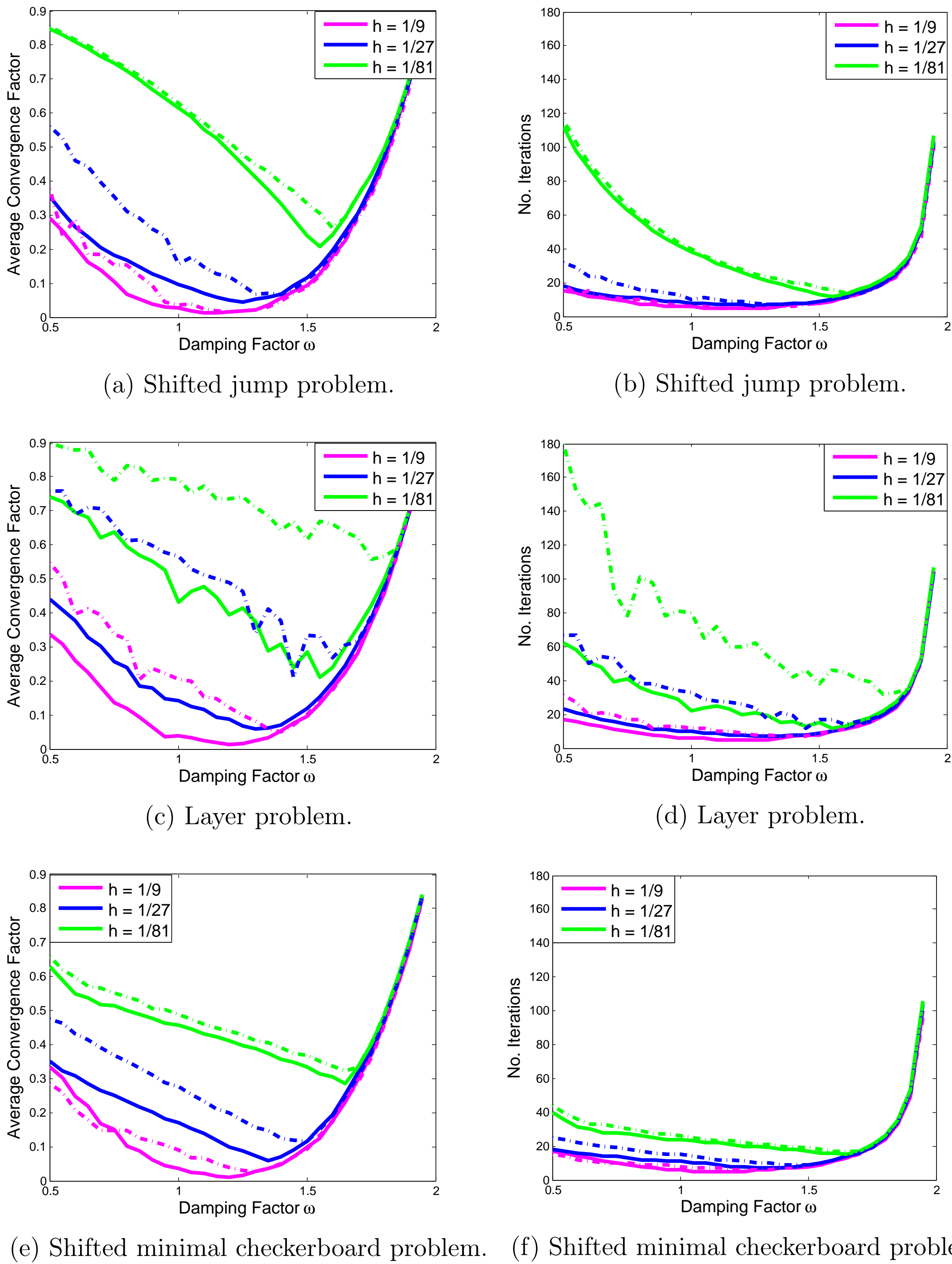
Figure 4.9: The average convergence factor ρ_{mean} (left) and the number of iterations (right) for the coefficient jump problem solved by a V(2,2) Gauss-Seidel multigrid for different mesh widths h.

A comparison between BoxMG intergrid transfer operators and bilinear interpolation/full weighting is shown in Fig. 4.10. The coefficient jump problem and the minimal checkerboard problem are "shifted", i.e. the jump is shifted by h from 1/3 (for the checkerboard in both coordinate directions.). We can see that BoxMG shows better convergence rates and is stable for a higher number of grid levels. Especially for the layer problem, the solver using bilinear interpolation shows a strong deterioration with decreasing mesh width. As explained in Secs. 2.1.2 and 2.1.3, this is because bilinear interpolation and full weighting can lead to a pollution effect – the correction of a fine grid point (or the residual at a coarse grid point, respectively) is influenced by wrong diffusion coefficients – and due to coarse grid operators which do not represent the fine grid problem appropriately.

The results given above underline the fact that the choice of the smoother affects the robustness and efficiency of the solver substantially. The Gauss-Seidel smoother is much more robust than Jacobi, but, as we have discussed in Sec. 4.3.4, is no option for the target implementation on a spacetree. Therefore, we test the hybrid smoother and the alternative relaxation scheme proposed in Sec. 4.3.4.

Smoother Variants for Spacetrees

We examine three smoother variants in this chapter (see Fig. 2.3 for nomenclature of the points):



(a) Shifted jump problem.

(b) Shifted jump problem.

(c) Layer problem.

(d) Layer problem.

(e) Shifted minimal checkerboard problem.

(f) Shifted minimal checkerboard problem.

Figure 4.10: The average convergence factor ρ_{mean} (left) and the number of iterations (right) for three of our model problems solved by a V(2,2) Gauss-Seidel multigrid for different mesh widths h. The solid lines show the result for a BoxMG solver, the dotted line for a solver using bilinear interpolation and full weighting as restriction.

- a 4×4 block Jacobi, where we do a Jacobi update (using the precomputed residual) at c and γ points and doing a block Gauss-Seidel step for the ι points, i.e., solve the inner 4×4 system directly using the updated values at the c and γ points,

- a hybrid Gauss-Seidel smoother as used in parallel multigrid (see Sec. 4.3.4), which does a Jacobi update at the c and γ points (using the precomputed residual) and one Gauss-Seidel sweep, using the updated values at the c and γ and points and maybe the ι points which were already updated before, at the ι points,

- and a "box smoother", which is the same as a hybrid smoother, except that we do an additional Gauss-Seidel step with collapsed stencils at the γ points after the Jacobi step there (see Sec. 4.3.4 for detailed description).

All these smoother can be applied in two traversals: The first for computing the residual at all vertices, and the second for performing the update per coarse grid cell. For all three variants we have to take care not to do the update twice/four times for c and γ points (see Sec. 4.3.4).

For the box smoother, we need three damping parameters ω: $\omega^{(1)}$ for the Jacobi on c and γ points, $\omega^{(2)}$ for the Gauss-Seidel on ι points, and $\omega^{(3)}$ for the Gauss-Seidel on γ points. For the classical hybrid variant we need two ω: $\omega^{(1)}$ for Jacobi and $\omega^{(2)}$ for Gauss-Seidel. For the block Jacobi, only one damping parameter is needed. We compare our smoothers to a point Gauss-Seidel and a point Jacobi. The goal is that one of the smoothers given above, which are all easy to implement on a spacetree, results in convergence rates which are better than the Jacobi V-cycle and in the ideal case comparable to those of the Gauss-Seidel V-cycle.

In Fig. 4.11 we show an example for the convergence behaviour of our multigrid solver with a hybrid smoother when it is applied to the homogeneous Poisson problem. The minimum lies at $\omega^{(1)}_{opt} = 1.0$ and $\omega^{(2)}_{opt} = 1.3$.

Tab. 4.8 shows the results for a V(2,2) cycle with different smoothers, applied to our problem configurations from Fig. 4.7. In some cases the optimal convergence factor was not received by a single ω_{opt}, but for a range of ωs. In these cases, we list this range. We did again an exhaustive search for the optimal ω with a step width of 0.05 for $\omega \in [0.5, 2.0]$. For the box smoother, we took $\omega^{(1)}_{opt}$ and $\omega^{(2)}_{opt}$ from the hybrid smoother and did the exhaustive search only for $\omega^{(3)}_{opt}$. In our experiments it turned out that the best results for the box smoother are received by applying the box smoother only on the finest grid and the hybrid smoother on the coarser grids, therefore the results in Tab. 4.8 are for this scheme. We observe several facts: First, the three block-type smoothers beat the Gauss-Seidel smoother only for the homogeneous Poisson problem. Second, interestingly, the hybrid smoother yields better results than the block Jacobi smoother (recall that the hybrid smoother

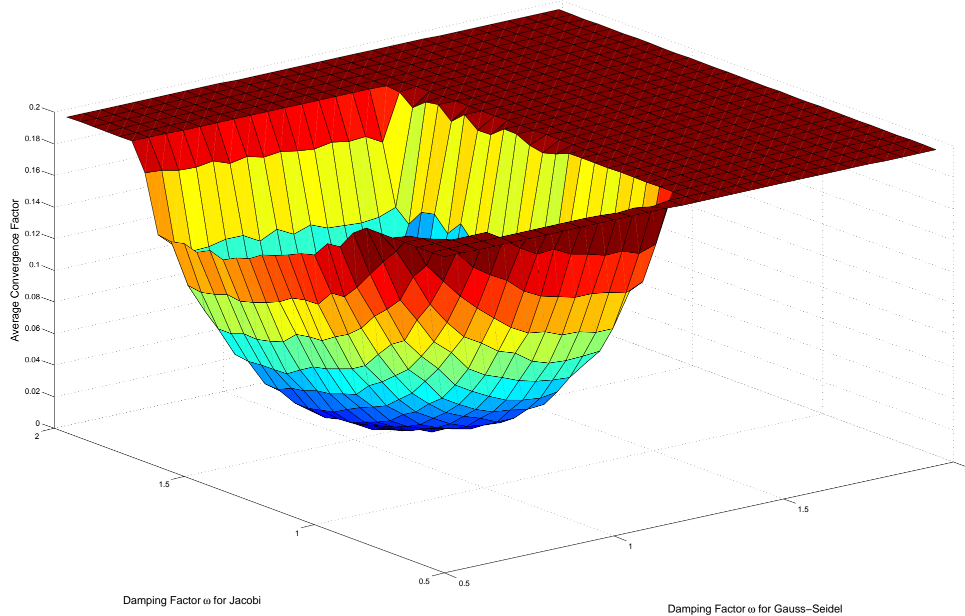


Figure 4.11: The average convergence factor ρ_{mean} for different damping parameters for a hybrid smoother in a V(2,2) cycle applied to the homogeneous Poisson problem. The values for $\rho_{mean} > 0.2$ were cut off.

would result in a block Jacobi if we did enough Gauss-Seidel steps for the ι points to solve the inner 4×4 system exactly). This behaviour was already shown in [7]. Third, by applying the box smoother on the finest grid, we can improve the convergence factor of the hybrid smoother for almost all problems. Only for the shifted jump problem, the damping factors are so small that we assume that the improvement is in the range of the variance due to the random initial conditions. Considering this variance, we also have to keep in mind that the optimal ρ and ω given in the tabular are no exact values, but vary due to the initial conditions. In order to receive reliable results for ρ_{mean} and ω_{opt}, we should repeat our experiments very often with different random initial conditions, and then take the average of all received values. In this work, we do not aim to give exact values here, but results that let us compare the different smoothers.

Although the box smoother on the finest level gives some additional improvement compared to the pure hybrid smoother, we stay with the hybrid smoother for the next experiments. Here, we apply the hybrid smoother to our test problems for different mesh widths h. The results are shown in Tab. 4.9. We see that there is a deterioration of the convergence with the decreasing h, but the convergence factor and the number of iterations until the stopping criterion is met remain satisfying. Comparing these results to those in Fig. 4.10, we conclude that the stability of the method with the hybrid smoother is for these examples comparable to that with the

Homogeneous Poisson					
Smoother	min(# cycles)	min(ρ_{mean})	$\omega_{opt}^{(1)}$	$\omega_{opt}^{(2)}$	$\omega_{opt}^{(3)}$
Jacobi	8	0.0931	1.05	–	–
Gauss-Seidel	6	0.0361	1.1	–	–
Block Jacobi	5	0.0241	1.0	–	–
Hybrid	5	0.0184	1.0	1.3	–
Box	5	0.0113	1.0	1.3	0.65-1.0

Shifted Jump					
Smoother	min(# cycles)	min(ρ_{mean})	$\omega_{opt}^{(1)}$	$\omega_{opt}^{(2)}$	$\omega_{opt}^{(3)}$
Jacobi	8	0.0878	1.05	–	–
Gauss-Seidel	6	0.041	1.2	–	–
Block Jacobi	8	0.0962	1.1	–	–
Hybrid	7	0.0800	1.15	1.5	–
Box	7	0.0770	1.15	1.5	0.0-0.25

Shifted Minimal Checkerboard					
Smoother	min(# cycles)	min(ρ_{mean})	$\omega_{opt}^{(1)}$	$\omega_{opt}^{(2)}$	$\omega_{opt}^{(3)}$
Jacobi	9	0.1070	1.1	–	–
Gauss-Seidel	7	0.0629	1.35	–	–
Block Jacobi	9	0.1224	1.55	–	–
Hybrid	8	0.0940	1.15	1.5	–
Box	8	0.0892	1.15	1.5	0.6

Layer					
Smoother	min(# cycles)	min(ρ_{mean})	$\omega_{opt}^{(1)}$	$\omega_{opt}^{(2)}$	$\omega_{opt}^{(3)}$
Jacobi	9	0.1168	1.05	–	–
Gauss-Seidel	6	0.0453	1.2	–	–
Block Jacobi	8	0.0879	1.15	–	–
Hybrid	7	0.0650	1.1	1.45	–
Box	7	0.0563	1.1	1.45	1.15

Table 4.8: Comparison of a V(2,2,) cycle with different smoothers for the homogeneous Poisson problem, the shifted jump problem, the shifted minimal checkerboard problem, and the layer problem, all on a 27×27 grid.

Gauss-Seidel smoother. By trading additional work in terms of pre- and postsmoothing steps or another multigrid cycle (see Sec. 2.1) with better convergence, we could improve the behaviour of each method further.

Homogeneous Poisson				
$1/h$	min(# cycles)	min(ρ_{mean})	$\omega_{opt}^{(1)}$	$\omega_{opt}^{(2)}$
9	4	0.0085	1.0	1.25
27	5	0.0184	1.0	1.3
81	6	0.0268	1.05	1.35

Shifted Jump				
$1/h$	min(# cycles)	min(ρ_{mean})	$\omega_{opt}^{(1)}$	$\omega_{opt}^{(2)}$
9	5	0.0123	1.05	1.3
27	7	0.0800	1.15	1.5
81	34	0.5777	1.35	1.85

Shifted Minimal Checkerboard				
$1/h$	min(# cycles)	min(ρ_{mean})	$\omega_{opt}^{(1)}$	$\omega_{opt}^{(2)}$
9	5	0.0125	1.0	1.25
27	8	0.0940	1.15	1.5
81	21	0.4080	1.3	1.7

Layer				
$1/h$	min(# cycles)	min(ρ_{mean})	$\omega_{opt}^{(1)}$	$\omega_{opt}^{(2)}$
9	5	0.0123	1.05	1.25
27	7	0.0650	1.1	1.45
81	12	0.2117	1.0	1.65

Table 4.9: Minimal number of V-cycles for reaching the stopping criterion, convergence factors ρ_{mean}, and optimal damping parameters ω for a V(2,2) cycle with a hybrid smoother for different mesh widths h.

4.3.7 Conclusion for Spacetree-Based Implementations

We now come back to the question of benefits and drawbacks of the spacetree-based implementation of our multigrid solver that was mentioned in the beginning of this chapter.

We showed that the BoxMG method enables us to build robust operator-dependent intergrid transfer operators for geometrically coarsened structured grids. Building

these operators can easily be done in a strictly local, cell-wise spacetree setting. Also the construction of Petrov-Galerkin coarse grid operators with arbitrary prolongation and restriction is possible in this setting. The critical component on structured spacetree grids, however, is the smoother. Here, the locality paradigm prevents the application of, for example, line smoothers and Gauss-Seidel smoothers, and thus reduces the robustness and efficiency compared to solvers which can apply this kind of smoothers. In addition, the convergence behaviour of the smoother is strongly influenced by the damping factor. The optimal damping/overrelaxation factor ω, though, depends on the problem and also the number of levels in the multigrid hierarchy. Thus, a robust "black box" smoother can hardly be achieved without a preceding determination of the optimal ω. Here, either methods as described in [69] have to be applied, or an ω that is known to yield good results in most cases has to be chosen.

It is worth noting that the three block-type smoother variants benefit from the coarsening factor three that we use. Coarsening by a factor of two and the resulting quadtree structure (with only four children per node, see Sec. 3.1) would yield "smoothing blocks" with only nine instead of sixteen vertices, and only one vertex in the interior of such a block where Gauss-Seidel smoothing can be performed. We expect that this affects the efficiency and effectivity of the smoothers considerably.

The benefits of the spacetree approach in terms of efficient data administration and traversal, and suitability for adaptivity and parallelisation were already discussed in Chap. 3. In the next chapter, we will see how our algorithm from Prototype 2 can be integrated into and parallelised in the target framework.

5 Geometric-Algebraic Multigrid on Spacetrees – Target Implementation

In the previous chapter, we designed and tested a hybrid algebraic-geometric multigrid algorithm for spacetrees that has improved robustness and efficiency as compared to state-of-the-art spacetree multigrid implementations due to operator-dependent intergrid transfer operators and a hybrid Gauss-Seidel smoother. Now we turn to the target framework Peano in order to implement and parallelise our solver there. We first describe the PDE solver framework Peano, its basic principles and the implications for the implementation. Then we present some details of the implementation of the hybrid MG solver in Peano, the parallelisation both in a shared memory and a distributed memory version, and runtime results on a supercomputer.

5.1 Peano Framework

The software framework Peano [100] for parallel adaptive PDE solvers on spacetrees is based on work of Günther [48], Pögl [77] and Krahnke [65]. In its current form it was mainly developed by T. Weinzierl [99, 101].

Peano uses tripartitioned spacetree grids and, as the name suggests, the Peano curve for the (strict cell-wise) spacetree traversal (see Chap. 3). For details concerning the stack approach and other characteristics of the framework, we refer to [99, 101] and take the point of view of a user of the given spacetree traversal infrastructure.

As such, we first have to define what kind of infrastructure we need. This can be done in definition files, where we specify the data structures for the cells, vertices, and the global state (which can include global variables as, for example, the current grid level of the multigrid solver, the residual norm, etc.), and a specification file, where we define basic read and write functions for the cell, vertex and state classes and specify the required mappings and adapters (which will be explained below). The Peano Prototyping Tool *PDT* (former called *PeProt*), which is based on the Data Structure Generator *DaStGen* [25], then generates the whole basic class infrastructure needed for plugging into the spacetree traversal.

The traversal itself is hidden away in the kernel, which we as Peano users basically do not need to (and should not) touch. The only thing we need to take care of is our solver, and maybe extension of our cell, vertex, and state classes. Therefore, we implement the cell-wise operators as described in Sec. 4.3.1. Now we have to apply the solver to the data on the spacetree vertices during the traversal.

For this purpose, several events are defined at the spacetree traversal, for example when a vertex is created, when we enter or leave a cell, when a vertex is touched for the first or the last time, and so forth. Fig. 5.1 shows as an example the order of events during a traversal of a simple two-level 1D spacetree. The recursive extension to multiple levels is straightforward. The order of the individual algorithmic steps reflected by the events is prescribed by properties of a depth-first spacetree traversal.

We now have to define which operations have to be done at which events. Here, the mappings and adapters mentioned above come into play: An adapter is a class with functions providing plug-in points at the events during the spacetree traversal. In these event functions we have to specify the tasks which shall be executed at that event. An adapter run needs one spacetree traversal. Of course, we have to take care of the order of the execution of the different operations, and of the dependencies between the tasks. Before we can, e.g., apply smoothing at a vertex, the residual has to be computed completely, and therefore the system operator as to be applied to all adjacent cells of a vertex (see also Sec. 4.3.1). We can be sure that this has happened when *touchVertexLastTime* is called for the vertex, therefore the smoothing can take place after completing the residual computation in that event in the same adapter. However, we cannot prolong the correction to the next finer level in the same adapter, as we do not know when all vertices involved in the prolongation have been processed. Hence, we still need several adapters for a (multiplicative) multigrid cycle, and therefore several spacetree traversals.

As a realisation of the *separation of concerns* software pattern, in order to make the code more clear, and to make it possible to execute different stages of a solver separately, an adapter is split into several mappings which are each dedicated to one single task (such as smoothing, computing the residual, restriction, ...). That way it is also possible to use mappings in various adapters, and code duplication is avoided.

When integrating a (multigrid) solver into Peano, there is a number of practical issues to consider concerning the events: First, as a vertex is uniquely defined by its position and its level in the spacetree, there might exist vertices at the same position at different levels. For each, *touchVertexFirstTime* etc. is called. Second, whether at one level first *enterCell* and the associated events are called for all cells at that level and then the *leaveCell* block, or the *enterCell* block and the *leaveCell* block for one cell, and then the same for the next cell etc., is not specified and can both happen. We do not have any information from adjacent cells or about the order they are accessed, we only know the order of events for one local cell. Therefore, the

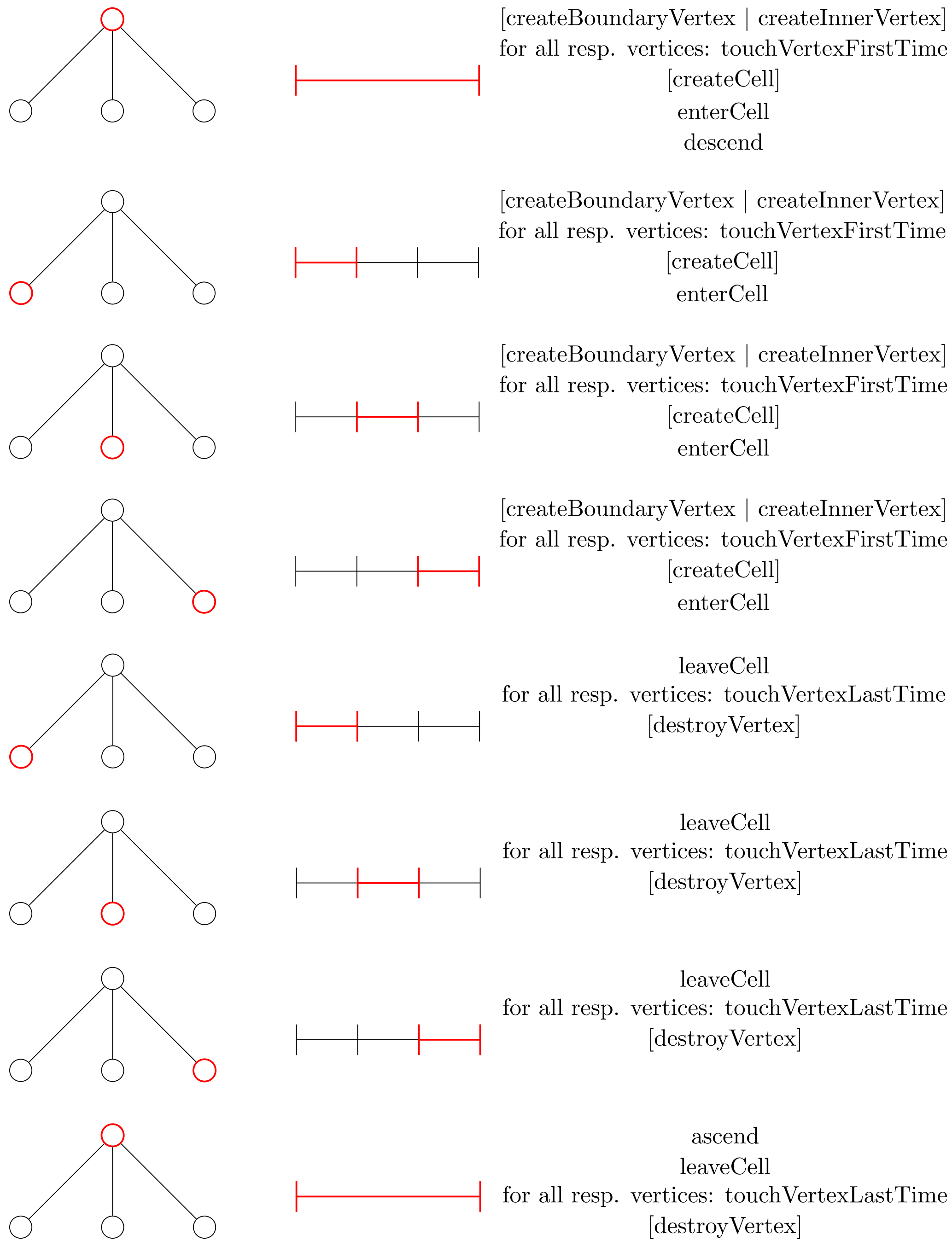


Figure 5.1: A simple 1D example for the order of (sequential) events during the spacetree traversal. At the beginning of the traversal *beginIteration* and at the end *endIteration* is called. Parallel events and events concerning hanging vertices are omitted. Square brackets stand for events which only occur during the setup/cleanup phase or in an adaptive setting.

order in Fig. 5.1 can be internally exchanged in any manner that fulfils the constraint that for one cell the *enterCell* block is executed before the *leaveCell* block. Third, there is a number of parallelisation-specific events which are not listed in Fig. 5.1. We explain the parallelisation-related issues in Sec. 5.3. The events dealing with hanging nodes on an adaptive grid are not considered here. Additionally, we have to take into account that the signature of the events varies. That means that we do not have access to the same data in all events. For example, in *touchVertexFirstTime* and *touchVertexLastTime*, we only have access to one vertex and the coarse vertices of the parent cell, whereas in *enterCell* and *leaveCell* we have access to the four vertices of the cell and the parent cell with its vertices. In *ascend* and *descend* we have access to a coarse cell (including vertices) and all its children. Therefore, the event in which an operation shall be executed has to be chosen carefully. These issues are directly related to the discussion about locality and communication in the spacetree in Sec. 3.2.

There are several reasons for the variable order of cells in the traversal and of the events: The simplest is that, due to the properties of the stack approach (see Sec. 3.2), the grid is traversed along alternating directions along the space-filling curves, i.e., the direction is inverted after each traversal. Additionally, it changes due to the colouring in shared memory parallelisation (see Sec. 5.3.1). And finally, Peano does an internal recursion unrolling on regular patches, which are then traversed level-wisely. All these internal processes lead to the fact that the information that we have is very local, and we cannot rely on the exact order of events that happen outside the current cell.

In the following section we show for the multigrid V-cycle how this concept of events, adapters and mappings can be implemented.

5.2 Sequential Hybrid Multigrid in the Peano Framework

We present the design of a sequential BoxMG solver in Peano which is equivalent to the solver described in Sec. 4.3 (using a Jacobi or hybrid smoother).

5.2.1 Implementation of the Multigrid V-Cycle

For implementing a multigrid V-cycle, we need basically the following operations (see also Sec. 2.1): Computing the residual, smoothing, restricting the residual to a coarser level, prolonging the correction to a finer level, solve the equation on the coarsest grid exactly. In addition, we have to construct the coarse grid operators and the intergrid transfer operators. For inspecting the convergence behaviour, we need to compute the global residual norm, and of course we have to setup the

Task	Mapping	Events
Setup experiment	*SetupExperiment*	*createInnerVertex*, *createBoundaryVertex*
Compute vertex residual	*ComputeResidual*	*leaveCell*
Accumulate global residual	*ComputeResidual*	*touchVertexLastTime*, *endIteration*
Smooth	*Smooth*	*touchVertexLastTime*
Compute Galerkin coarse grid operator	*ComputeGalerkin-CoarseGridOperator*	*leaveCell*
Compute BoxMG restriction operator	*ComputeBoxMG-Restriction*	*ascend*
Compute BoxMG prolongation operator	*ComputeBoxMG-Prolongation*	*descend*
Restrict residual	*Restrict*	*leaveCell*
Prolong correction	*Prolong*	*leaveCell*

Table 5.1: Tasks in the solver with the corresponding event mappings and the events in which the main operations (disregarding initialisation etc.) of these tasks take place.

experiment in the beginning, i.e., discretise our problem, initialise the variables u and the right-hand sides f on the grid, and so forth.

In Tab. 5.1, we list the tasks needed in the element-wise BoxMG multigrid implementation, the respective mappings in Peano, and the events in which their main operations (disregarding initialisation etc.) are implemented. We notice that the cell-wise operators are, obviously, implemented in *leaveCell* (*enterCell* would also be possible). The computation of the BoxMG intergrid transfer operators needs all the children of one spacetree node, therefore it has to be implemented in *ascend* or *descend*. From a shared memory parallelisation point of view it is better to choose *leaveCell* or *enterCell* instead of *ascend* or *descend*, if possible. In the first case, up to four cells in a coarse grid cell can be processed in parallel if 2^D colouring on the fine grid is used (see Sec. 5.3.1), whereas in the latter case, 2^D colouring has to be applied to the coarse grid in order to avoid data races. Within each colour, the nine subcells then are processed sequentially. Thus, the concurrency is smaller compared to 2^D colouring on the fine level. For the accumulation of the global residual and for the smoothing, the contributions of the application of the cell-wise system operator to all adjacent cells have to be made before, therefore these tasks have to be executed in *touchVertexLastTime*. However, the signature of this event does not provide access to the global state, which holds also the global residual.

Therefore, we have to do the accumulation on an additional global residual variable in the mapping, and write this variable to the the global state in *endIteration*, where the state is available. [1]

In this example, we study a Jacobi smoother. For a hybrid smoother as described in Sec. 4.3.4, the operations have to be performed in *ascend* or *descend* in order to be able to do the Gauss-Seidel update on the inner points of a 3×3 cell patch. In our implementation, we do the complete smoothing update in these events. A variant would be doing the Jacobi update at the c and γ points in *touchVertexFirstTime* or *touchVertexLastTime*, and the Gauss-Seidel update of the γ and ι points in *ascend*. In this case, however, we would have to determine for the Jacobi step whether a point is a c point or a γ point at that level. This would theoretically be possible, but it induces a runtime overhead due to case distinctions. A second variant would be changing the smoother such that we do Jacobi smoothing for all points (including ι points) and then the Gauss-Seidel update only at γ and ι points. The advantage of these variants would be that we would not need an additional preceding traversal for the computation of the residual to be sure that the correct residual is available at the c and γ points. However, we will see in Sec. 5.3.2 that we need an additional traversal for the correct computation of the residual, anyway, if we do distributed memory parallelisation. Both variants are not implemented in our solver.

A naive version of the multigrid solver embeds each mapping in one adapter. For the execution of each adapter we need one spacetree traversal. We implemented our multigrid V-cycle (see also Fig. 2.1a) as a state automaton with states *RESTRICT*, *PROLONG*, *PRESMOOTH*, *POSTSMOOTH* and *SOLVE* (the last one meaning solving exactly on the coarsest level). Each state corresponds to one stage in the multigrid cycle and determines which adapters are executed next. In addition, we need the state variable *activeLevel*, which holds the current level in our grid hierarchy. It determines the level where we have to smooth/restrict/prolong, and whether we are already on the coarsest or the finest level (and therefore, in which state we have to switch). A multiplicative multigrid (see Sec. 2.2.5) cycle is inherently sequential regarding the sequence of grid levels, therefore we cannot perform one task at all levels simultaneously. Instead, we have to do at least one spacetree traversal for each level, both when we go up and when we go down in the V-cycle. Naturally, the spacetree traversal starts at the root of the tree, and thus also the event sequence starts at the coarsest grid level. In multigrid, most cycles (including the V-cycle that we use for our solver) start at the finest level. Hence, for the first smoothing step we have to traverse the tree down to the leaves and perform smoothing there, after restriction we have to go to the second-deepest level, etc..

In [99], so-called "tree-cuts" were introduced in order to enhance the performance of a tree traversal in a multilevel setting. The idea is that finer grid levels can be

[1] That the state is only available in *beginIteration* and *endIteration* has internal reasons concerning the parallelisation.

"cut off", i.e. are not included into the tree traversal, when working on a coarse level. They then can be added again level by level when going to finer levels. Thus, for grids with a large number of unknowns (where the administrative overhead becomes negligible), almost linearly an improvement of the runtime by a factor of nine per level could be achieved. Unfortunately, this feature is not yet available in current Peano versions. [2]

The experimental setup as well as the computation of the coarse grid and intergrid transfer operators has to be done only once in a non-adaptive setting where the system operator does not change during execution time. We can do the computation of the coarse grid and intergrid transfer operators during the first V-cycle in the *PRESMOOTH* stage (the computation of the prolongation operator could also be done in the *POSTSMOOTH* stage, however, we will put the two mappings for the computation of the prolongation and the restriction operator in the same adapter later on). The residual computation has to be done before each smoothing step, and again before the residual is restricted. All in all, for this straightforward approach we would need the following number of spacetree traversals for the first V(1,1) cycle (and for all V-cycles in an adaptive setting):

$$\begin{aligned} t = & \overbrace{1}^{\text{setup}} + \overbrace{2 \cdot (l-1)}^{\text{residual + presmooth}} + \overbrace{2 \cdot (l-1)}^{\text{residual + postsmooth}} + \overbrace{2 \cdot (l-1)}^{\text{intergrid transfer operators}} \\ & + \overbrace{(l-1)}^{\text{Galerkin operator}} + \overbrace{2 \cdot (l-1)}^{\text{residual + restrict}} + \overbrace{(l-1)}^{\text{prolong}} + \overbrace{2 \cdot s}^{\text{solve coarsest level}} \\ & = 1 + 10 \cdot (l-1) + 2 \cdot s, \end{aligned} \tag{5.1}$$

with t being the number of traversals, l being the number of levels in the grid hierarchy and s being the number of smoothing iterations used on the coarsest level to solve the problem there exactly. In the experiments, we use $s = 10$ Jacobi iterations as solver on the 3×3 level.

Note that the number of traversals does not directly yield a conclusion for the runtime of the algorithm, as the runtime per traversal is not constant and depends on the number of levels, the operations performed per traversal, and so forth.

As mentioned in the previous section, we can eliminate spacetree traversals by putting mappings together in an adapter whenever the dependencies allow that. For example, we define the adapters *computeResidualAndSmooth*, *computeIntergridTransferOperators* (combining *computeBoxMGRestriction* and *computeBoxMGProlongation*) and *computeResidualAndCoarseGridOperatorAndSmooth*. Alg. 5.1 shows the resulting (simplified) code snipped for the state-machine realised as *switch-case* construct. The adapters are written in italic. Such, we obtain for the first V-cycle

[2] Currently, an improved version of tree-cuts, allowing to cut off subtrees in single subdomains in adaptive settings, is developed.

Algorithm 5.1 Multigrid State-Machine for V(1,1) Cycle

```
activeLevel = finest level
switch state do
    case PRESMOOTH
        computeIntergridTransferOperators
        computeResidualAndCoarseGridOperatorAndSmooth
        state = RESTRICT
    case RESTRICT
        computeResidual
        restrict
        activeLevel = activeLevel - 1
        if activeLevel == coarsestLevel then
            state = SOLVE
        else
            state = PRESMOOTH
        end if
    case SOLVE
        for number of iterations needed to solve exactly do
            computeResidualAndSmooth
        end for
        state = PROLONG
    case PROLONG
        prolong
        activeLevel = activeLevel + 1
        state = POSTSMOOTH
    case POSTSMOOTH
        computeResidualAndSmooth
        if activeLevel == finestLevel then
            break
        else
            state = PROLONG
        end if
end switch
```

a complexity of

$$t = 1 + 6 \cdot (l - 1) + s.$$

If we do not need to recompute the intergrid transfer operators in a static setting, we safe another $(l - 1)$ traversals for the next V-cycles.

If we assume at least two presmoothing steps, we can merge another smoothing step into the adapter which computes the intergrid transfer operators.

Other adapter merging strategies are proposed in [99]. They, however, focus on purely geometric multigrid. We will stay with the approach given above, as its simplicity will make the parallelisation more or less straightforward later on.

5.2.2 Results for Sequential Implementation

We perform test runs on one island of the SuperMUC (see [92]) which is a Sandy Bridge-EP system with eight cores per processor, two processors per node and 512 nodes per island. Thus, we have sixteen physical threads and 32 logical threads when using hyperthreading. The SuperMUC consists of eighteen islands. The subsytem used for our experiments has two GByte memory per core and 256 kByte RAM (L2 cache), a peak performance of 3.185 PFlop/s, and a clock speed of 2.7 GHz.

In Tab. 5.2 we compare the runtimes of an adapter performing Jacobi smoothing to the two adapters (residual computation and smoothing update) performing hybrid smoothing (see Sec. 4.3.4). In addition, we compare these runs to the runs of an adapter doing both the residual computation and the hybrid smoothing update (without caring about whether the residual is already correct everywhere). This last adapter is just for "fairness" reasons when comparing the hybrid smoother to the Jacobi adapter. We could also have made a comparison of the hybrid smoother update with the Jacobi update without residual computation. We see that the *ComputeResidualAndHybridSmooth* adapter is always slightly slower than the Jacobi (*ComputeResidualAndSmooth*) adapter. Recalling that in the Jacobi adapter we only do one update per vertex in *touchVertexLastTime*, whereas for the hybrid smoother we update the c points four times and the γ points two times in *ascend* (or *descend*), this runtime increase is reasonable. When looking at the runtimes of the two adapters which do the hybrid smoothing in practise, the difference becomes more significant. The variants for the implementation of the hybrid smoother as described in Sec. 5.2, which would allow to do the whole update in one traversal, might be worth considering for a sequential or shared memory implementation. However, as already mentioned in that section, we will need an extra adapter run for the computation of the residual, anyway, when doing distributed memory parallelisation (see Sec. 5.3.2). In Tab. 5.2 it also becomes apparent that a run of an adapter doing residual computation and a smoothing update does not take significantly longer than a run of the pure residual computation adapter. This shows again (see also,

Jacobi		
Adapter	# Unknowns	Time[s]/Vertex
ComputeResidualAndSmooth	6,400	$3.72 \cdot 10^{-4}$
ComputeResidualAndSmooth	58,564	$2.55 \cdot 10^{-4}$
ComputeResidualAndSmooth	529,984	$2.74 \cdot 10^{-4}$

Hybrid Smoother		
Adapter	# Unknowns	Time[s]/Vertex
ComputeResidual	6,400	$3.57 \cdot 10^{-4}$
HybridSmooth	6,400	$3.08 \cdot 10^{-4}$
ComputeResidualAndHybridSmooth	6,400	$4.10 \cdot 10^{-4}$
ComputeResidual	58,564	$2.54 \cdot 10^{-4}$
HybridSmooth	58,564	$2.08 \cdot 10^{-4}$
ComputeResidualAndHybridSmooth	58,564	$3.02 \cdot 10^{-4}$
ComputeResidual	529,984	$3.02 \cdot 10^{-4}$
HybridSmooth	529,984	$2.34 \cdot 10^{-4}$
ComputeResidualAndHybridSmooth	529,984	$3.34 \cdot 10^{-4}$

Table 5.2: Runtimes per unknown for a hundred sequential smoother adapter runs for a Jacobi and a hybrid smoother. The hybrid smoothing is done in two adapter runs in order to ensure the use of the correct residual, however, for comparison, the runtime of an adapter doing both residual computation and hybrid smoothing update is also given.

e.g., [99, 101]) that simulations in the Peano framework are memory-bound rather than compute-bound: The time needed for loading the data etc. for a spacetree grid traversal dominates the computational time.

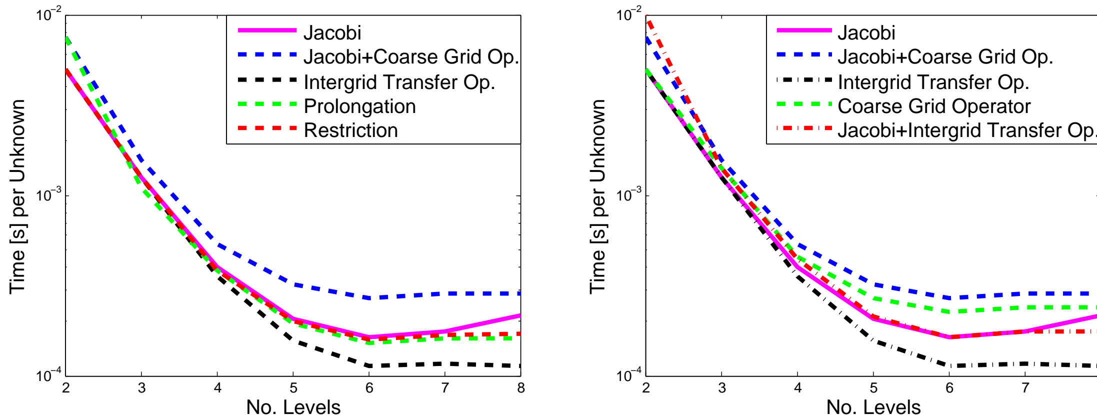


Figure 5.2: Runtimes per unknown on the finest level for a hundred iterations of the adapters of the V-cycle (left) and comparison of combined and pure adapters (right). We omit the adapter for pure residual computation and refer to Tab. 5.2 for its runtimes.

Next, we perform a hundred iterations on the finest level for the adapters of the V-cycle as given in Alg. 5.1. The results are given in Fig. 5.2. Two levels means that the finest level is a 3×3 grid and we have four inner points and therefore four unknowns, for three levels it is a 9×9 grid with 64 unknowns, and so forth. For better readability, we omit the pure residual computation adapter (from Tab. 5.2 we can see that its runtimes are always a bit longer than those of the Jacobi adapter).

The runtime of the “slowest” adapter, i.e., that performing Jacobi smoothing and computation of the Petrov-Galerkin coarse grid operators, differs by a bit less than a factor of 2.5 from the “fastest” adapter, i.e., that doing the computation of the intergrid transfer operators. In the right-hand plot of Fig. 5.2, we compare the runtimes of the pure operator computations adapters with adapters doing an additional Jacobi smoothing. In Sec. 5.2.1 we already pointed out that for a V-cycle with two or more presmoothing steps, we can combine an additional smoothing step with the computation of the intergrid transfer operators. The runtime of the Jacobi smoothing adapter is given as reference. Again it becomes apparent (especially for the computation of the intergrid transfer operators) that the additional computational load is almost negligible in comparison to the overhead due to data loading and so forth.

As expected, in all experiments the administrative overhead becomes much smaller for a sufficient large number of vertices. For less than five levels, this overhead dominates the runtime. Up to between five and six levels (i.e., 6,400 and 58,564

unknowns), the runtime per unknown goes down, and then it approximately stagnates.

5.3 Parallelisation

In this section, we discuss the parallelisation of our multigrid implementation in Peano. For a general overview of parallel multigrid methods we refer to [59, 27, 70]. Parallel smoothers are also discussed in [1, 7, 8].

5.3.1 Shared Memory Parallelisation

Peano offers shared memory parallelisation due to OpenMP [75], Intel Threading Building Blocks (Intel TBB) [61], and Cobra [28, 54]. For the present experiments, we used TBB.[3] The three different approaches, however, can be exchanged via a compile switch.

Peano realises functional and data decomposition. While the functional decomposition makes different tasks such as geometry updates of cells and vertex load phases run in parallel, the important gain in concurrency stems from parallel data processing.

The data decomposition approach realised in Peano so far is a 2^D colouring on regular grids (with D being the dimension). In [36], it is shown that by identification of regular patches in adaptive grids, it can also be applied in an adaptive setting. The grid cells are coloured in such a way that there are no two adjacent cells of the same colour at a vertex (see Fig. 5.3). Following the idea of red-black Gauss-Seidel, Peano runs through the 2^D colours sequentially. For each colour, the active adapter (and its mappings) are copied up to p times, with p being the number of threads available. The cells of this colour then are distributed among these adapter copies following the well-known fork-join multithreading pattern for parallel *for*s: All the adapter instances for one colour are executed in parallel, event by event. When all colours are processed, the properties of the adapter copies are merged. See [36] and [73] for more details.

Using this method, the results in the fine grid vertices are accumulated as usual, just like in the sequential case, and the threads do not interfere with each other. Data races arising from cell events accessing the same vertices are avoided a priori. As a user who wants to implement a multigrid solver, we only have to take care of the TBB setup, of initialising the threads correctly in the copy constructor, and the merging of global variables. In the present work, this is of relevance for the overall residual norm, which has to be merged in an additional merging step. For

[3] We decided to use TBB as Cobra is not freely available, and current OpenMP versions yield bad runtimes for nested parallel sections, as they are inherent in Peano.

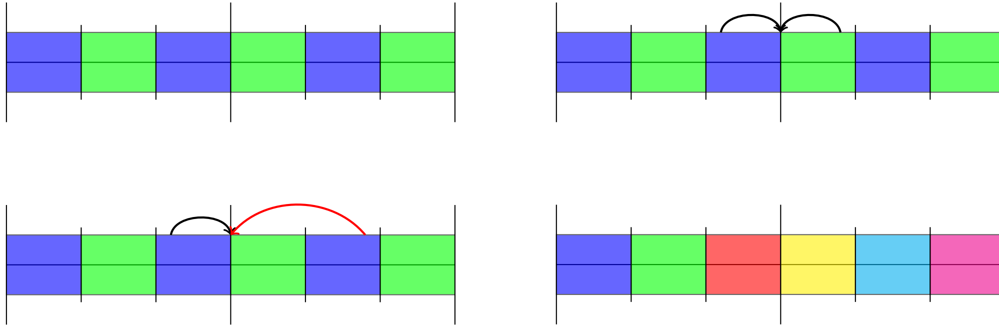


Figure 5.3: 1D examples for six fine grid cells in two coarse grid cells with a 2^D colouring (top left) and a 6^D colouring (bottom right). We observe that 2^D colouring is not sufficient if fine grid cells need writing access to coarse grid vertices (top right and bottom left).

this, the event *mergeWithWorkerThread* is provided, where the residuals computed from the different mapping instances/threads have to be summed up. Here, it is important to keep in mind how the global residual norm is computed from the single residuals at vertex positions (k, j): As we use the l_2 norm, the residual norm in one iteration step i is $||r_i|| = ||f^h - A^h u_i^h|| = \sqrt{\sum_{k,j}(r_i(k,j))^2}$. Practically, this means that in *touchVertexLastTime*, we accumulate the squared local residuals to a global variable. In the end, we take the square-root of the accumulated residuals. If we have two threads in a parallel setting, both have accumulated part of the residual. Therefore, we can only take the square-root after having merged both threads, i.e., after summing up both parts of the residual on the global variable.

During the implementation of the (shared memory) parallel version of our multigrid solver, it turned out that the 2^D colouring approach is not sufficient for our kind of multigrid solver. For the restriction, the computation of the coarse grid operators, and the computation of the intergrid transfer operators, we write the result on the coarse grid vertices in *enterCell* or *leaveCell*. A problem implied directly by the 2^D colouring is illustrated in Fig. 5.3: Although no two fine grid cells of the same colour can access a fine grid vertex at the same time, it can happen that two equally coloured cells access the same coarse grid vertex. In the worst case, the following data race is possible: Two mappings read the value at the coarse grid vertex. Then, both compute their update, accumulate it to the value they received, and one after the other writes the new value back. However, the update of the mapping that writes back first will get lost, as it is not considered for the update of the second mapping. In order to avoid this kind of situation, the fine grid cells in neighbouring coarse grid cells (i.e., coarse grid cells which share a vertex) all must have different colours. This results in a 6^D colouring, as shown for 1D in Fig. 5.3. Of course, this yields a lower concurrency level, as the number of cells that can be processed in

parallel goes down significantly.

Results for Shared Memory Parallelisation

We perform test runs on one node of the SuperMUC. As before, we perform a hundred iterations on the finest level for the Poisson problem for an adapter that does a pure Jacobi smoothing, an adapter that does Jacobi smoothing and computes the coarse grid operator, and an adapter that does Jacobi smoothing and computes the intergrid transfer operators.

For the Jacobi adapter, 2^D colouring is used, as no communication between two levels happens. The Galerkin adapter needs 6^D colouring. For the intergrid grid transfer adapter, a 2^D colouring on the coarse grid is applied. As the computation of the intergrid transfer operators happens in *ascend* and *descend*, where we have a coarse grid cell and all its children available, the maximal concurrency level is equivalent to a 6^D colouring on the fine grid with only 2^D colours in total. In the Galerkin computation, we have more colours and therefore more "phases" to run through, whereas in the BoxMG computation, we have a higher workload per colour/phase.

The results are shown in Fig. 5.4 in two variants: with and without so-called "tree splits". By splitting the tree into subtrees, parallel loading and storing of the corresponding parts of the grid data is enabled (see [83]). As the realisation is subject to other work (as before, we only use this feature that is provided by the framework), we will here not go into further details, but just show the differences in the results. In both versions, a clear speedup can be observed up to sixteen processors. For hyperthreading (more than sixteen processors), the overhead is so big that there is no further performance improvement. It is common knowledge that hyperthreading for PDE solvers is of limited value. We can see that the tree splits improve the scaling behaviour considerably. For the no-tree-splits version, we get a speedup of about 2.4 for the intergrid transfer operator adapter when going from one to 32 threads (with hyperthreading), in the tree split version this factor almost doubles to 4.56 (see also Tab. 5.3).

There are two interesting effects that we can observe in the results for the run without tree splits: First, for the Jacobi smoother, the performance decreases when using more than sixteen threads, i.e., when using hyperthreading (as sixteen is the number of physical threads per node). Second, for all adapters, there is a small runtime degradation when going from eight to ten threads. As we have eight cores per processor and two processors per node, we assume that his can be considered a Non-Uniform Memory Access (NUMA) effect: As soon as a processor accesses "foreign" memory, i.e., memory of another processor, the performance decreases. But when going to twelve processors, the foreign-memory-access penalty is compensated by the greater computing power. It would be interesting to see whether the same

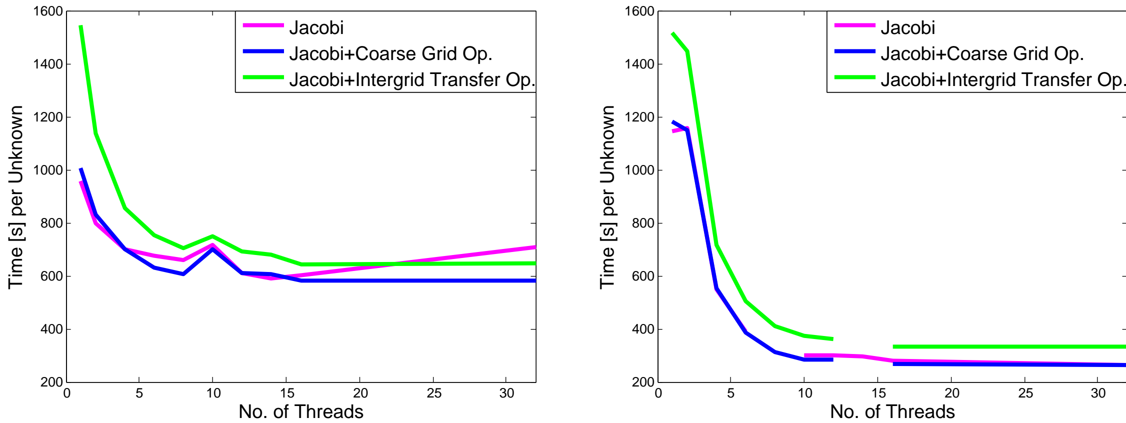


Figure 5.4: Scaling results for TBB parallelisation with 4,778,596 unknowns without (left) and with (right) tree splits.

behaviour can also be observed for other problem and solver types.

In Tab. 5.3, the speedups as quotient between the runtime when using one thread and the runtime when using 32 threads and hyperthreading are listed for different numbers of levels (and therefore numbers of unknowns on the finest level), with and without tree splits. In accordance with the previous results in Fig. 5.4, the Jacobi adapter yields better results for sixteen threads than with hyperthreading switched on. Again, the improvement due to the tree splits becomes apparent. We also see that satisfying scaling results are only achieved starting from six levels, which corresponds to 58,564 unknowns. This is probably due to the concurrency level: For 2^D colouring, $n/2^D = n/4$ cells and for 6^D colouring, at most $n/6^D = n/36$ cells (with n being the number of cells) can be processed in parallel. Therefore, taking into account the parallelisation overhead (including task management, task deployment, task communication, global barriers, work stealing, and so forth), only for a rather large number of cells we get an improved runtime due to the TBB parallelisation. Furthermore, the results highlight the impact of the computational workload on the speedup: The best results are achieved by the adapter that does the computation of the intergrid transfer operators. Here, several equation systems have to be solved per coarse grid cell. In comparison, the computation and the update in the Jacobi smoother yield a rather small workload. We expect better scaling when the computational workload increases (e.g., for a 3D problem).

5.3.2 Distributed Memory Parallelisation

The distributed memory parallelisation in Peano is realised with the Message Passing Interface (MPI) [72]. Naturally, things become a bit more complicated when dealing with distributed memory, as now no longer the whole data domain is available for all

No Tree Splits			
# Levels	Jacobi	Galerkin	BoxMG
2	1.01	0.99	0.99
3	1.00	1.00	1.00
4	1.01	0.99	1.00
5	0.95	0.95	0.95
6	1.17	1.14	1.53
7	1.80	1.63	2.20
8	1.35	1.72	2.40

Tree Splits			
# Levels	Jacobi	Galerkin	BoxMG
2	1.01	1.01	1.00
3	1.01	1.01	1.00
4	0.80	1.07	0.85
5	1.18	0.98	1.05
6	2.03	2.33	2.05
7	3.46	3.69	4.37
8	4.31	4.52	4.56

Table 5.3: Speedups for the pure Jacobi adapter, Jacobi plus computation of Galerkin coarse grid operators, and Jacobi plus computation of BoxMG intergrid transfer operators. We show results for runs without (top) and with (bottom) tree splits. The speedup is computed as quotient between the runtime when using one thread and the runtime when using 32 threads with hyperthreading. For the pure Jacobi adapter, a better speedup can be observed without hyperthreading (see Fig. 5.4).

processors (also called "ranks" in MPI). Similar to the shared memory parallelisation, Peano takes care of the underlying domain decomposition and load balancing tasks in the spacetree (see [99]), but we have to make sure that application-specific data is merged and distributed at the right places and at the right time.

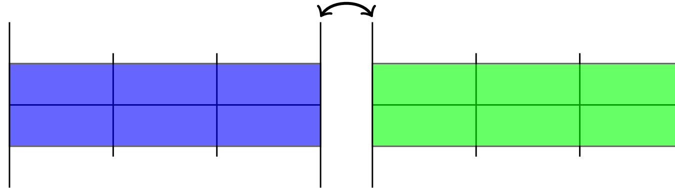


Figure 5.5: 1D example for the domain decomposition approach. The two subdomains (green and blue) are assigned to different processors. The vertex at the subdomain boundary which is adjacent to both subdomains is dublicated. After each iteration, information has to be exchanged between the processors for this vertex.

The main idea of Peano's domain decomposition is that vertices are duplicated at the subdomain/processor boundaries such that each processor holds its own copy. The cells as well as the subpartitions of the spacetree are not overlapping. After completing a traversal, each processor receives copies of all vertices also held by other processors, i.e., those along the parallel boundary, for data exchange. This principle is illustrated for a simple 1D example in Fig. 5.5. For such a domain decomposition, at least three ranks are needed in Peano: Rank 0, the global master rank, is responsible for input and output, the traversal, "administrative" tasks, such as dynamic load balancing (which we do not use in our experiments) and the global state variables. Peano induces a tree topology on the MPI ranks, i.e., each rank besides rank 0 is a worker rank for another rank and responsible for a multiscale subdomain. In turn, each rank can deploy subtrees of its spacetree to other ranks. In this case, the other rank is its worker, and the local master rank still holds a copy of the remote subtree's root locally.

There are two types of communication for distributed memory parallelisation in Peano: Asynchronous and synchronous. Asynchronous communication means that the data is merged only for the next traversal. One rank does not wait for information from other ranks until it continues its computations. This is realised as non-blocking communication in MPI. Synchronous communication means that the data is merged in the same traversal. In MPI, this is realised as blocking communication.

In Peano, the realisation of these concepts is as follows: At the processor boundaries, two ranks send each other the vertex they share, such that each can merge the vertex data from the other rank to its own vertex data. This is a "Point-to-Point" communication between two worker ranks. It is asynchronous: One rank

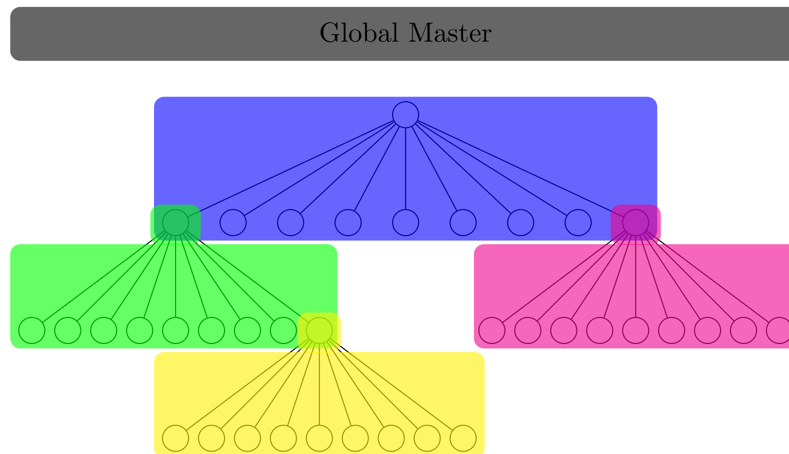


Figure 5.6: The domain decomposition on a spacetree is implied by its subtrees. Here, we show an example with five ranks. The blue rank has only rank 0 (black) as master and is itself local master of the green and the magenta rank. The yellow rank has the green rank as local master. "Broadcast" communication will go from black to blue, then to green and magenta, and then from green to yellow. "Reduce" communication will go the other way around.

does not wait until it receives the vertex data from the vertex counterpart at the other rank, but the traversal is continued, and only at the beginning of the next traversal, the local and neighbour information is merged. The synchronous communication always happens between master and worker rank and is completed in the same traversal. Here, we have three types in Peano: Collective communication from the global master to all ranks, indicating, for example, which adapter has to be used next, "broadcast" processes (a global variable is send to all workers, e.g.), which are realised in a Point-to-Point manner as communication along the tree from each local master to its worker, and "reduce" processes, the counterpart to "broadcast", where the global master rank collects data from the workers and merges it into the global variables. This is also realised as Point-to-Point communication between worker and local master along the tree. In Fig. 5.6, we show an example for an domain decomposition with five ranks on a spacetree, and explain what we mean by "communication along the tree".

For the Peano user, a number of additional events that provide communication plug-in points is available in the mappings. We do not cover the whole sequence of events here, but discuss some events as examples in order to give an idea about the issues one has to deal with when doing distributed memory parallelisation in Peano. Please refer to the documentation and the Wiki at [100] for a detailed explanation of the parallel communication events.

For our cell-wise multigrid implementation, at first the event *mergeWithNeighbour*, which is responsible for merging the vertex values at the domain/processor boundaries, is important. This event is called before *touchVertexFirstTime*. In addition, for the computation of the overall average residual ρ_{mean}, we have to implement *mergeWithWorker* in the state class. When merging the global residual computed by two ranks we have to add up the contributions of the different ranks before we take the square-root, just as before. At the domain boundary, however, there is something missing: Recall that the residual norm in one iteration step i is $||r_i|| = ||f^h - A^h u_i^h|| = \sqrt{\sum_{k,j}(r_i(k,j))^2}$, with (k,j) the positions of the vertices in the grid. When computing the residual at a vertex as described in Sec. 5.2.1, we accumulate the contributions of the adjacent cells in *leaveCell* and write the squared vertex residual to the global residual in *touchVertexLastTime*. Let us call the contribution from the four adjacent cells on a 2D grid c_1, c_2, c_3, and c_4, with $r_i(k,j) = c_1 + c_2 + c_3 + c_4$. If the vertex at grid point (k,j) lies at a boundary, we have written, for example, $a^2 := (c_1 + c_3)^2$ to the global residual norm from one (e.g. the "green") rank and $b^2 := (c_2 + c_4)^2$ from the other (the "blue") rank. However, $a^2 + b^2 \neq (a+b)^2 = a^2 + 2ab + b^2$, i.e., $2ab = 2(c_1 + c_3)(c_2 + c_4)$ is missing. This contribution has to be added in *mergeWithNeighbour* to the global mapping residual and then written to the global state residual in *endIteration* (see Sec. 5.2.1). As this is done by both ranks, each only has to add $ab = (c_1 + c_3)(c_2 + c_4)$ to the global residual. If more than two ranks are adjacent, the mapping on each rank will add the respective contribution automatically that way. Due to this asynchronous communication, the residual at the processor boundaries is not correct until the next iteration starts – for every residual computation it lags behind the residual in the rest of the domain.

At some points in our multigrid algorithm, we must have two levels available, e.g. for the intergrid transfer operations and for computing the coarse grid operators. As discussed before, Peano provides access to the coarse grid cell in several events. However, if we are for example in *leaveCell* and want to access a coarse grid vertex which lies on another rank – in Fig. 5.6, this is the case if the root of the green or of the magenta subtree is our current fine grid cell, and the root of the blue subtree is our current coarse grid cell –, we do not receive a valid value here and also writing does not yield the expected result. Thus, we have to catch this case by asking whether the coarse grid cell is held by the same rank. If not, we have to plug into communication events between master and worker: *mergeWithMaster* has the same signature as *leaveCell*, so we just can do here what we wanted to do there. Multiscale algorithmic steps are computed on the rank holding the responsibility for the coarse data. This works if we want to alter the (coarse grid) data on the master, as for restriction and the computation of the coarse grid operators. For the prolongation, similar issues arise.

ascend and *descend* have to be treated carefully if some of the children belong to another (worker) rank. In Fig. 5.6, we have this situation for the green and the magenta subtree root in the blue rank and for the yellow subtree root in the green rank. In this case, both master and worker hold copies of the cell (including refinement information and so forth) similar to parallel boundary subdomain vertices. But we manually have to ensure that PDE-specific data such as stencils are kept consistent by the respective mappings.

Another challenge lies in the fact that communication is not possible anytime, but only at fixed points during the traversal. These points underlie restrictions posed by the asynchronous communication described above. For domain decomposition, synchronising this communication (i.e., let each node wait until it has the information from all neighbouring ranks) would lead to a sequential execution of the tasks. As said before, *mergeWithNeighbour* is not called at the end of the vertex processing, i.e., after *touchVertexLastTime*, but at the beginning of the vertex processing in the next traversal, i.e., before *touchVertexFirstTime*. This means that we basically need an additional traversal for merging all the data by asynchronous communication at the processor boundaries. Accordingly, we have to implement merge mappings – mappings were only the merging events are implemented such that the relevant data is merged at the boundary vertices. These mappings can now be put together in adapters with other mappings, with the goal that we have to do as little as possible additional spacetree traversals.

Let us look again at the example of residual computation and smoothing. We know that at the processor boundaries only at the beginning of the next traversal the correct residual is available. Therefore, the adapter *computeResidualAndSmooth* does not work any longer and has to be splitted into something like *computeResidual* and *mergeResidualAndSmooth*. The computing of the residual, however, can be combined again with a merge of the values from the previous operation (for example restriction), and so forth.

Alg. 5.2 shows how Alg. 5.1 has to be extended in order to realise the parallel version of the V(1,1) cycle. As we can see, there are a lot more adapters involved. We count again the spacetree traversals needed. By looking at the cases in Alg. 5.2 we obtain:

$$\begin{aligned} t &= \overbrace{1}^{\text{SETUP}} + \overbrace{4}^{\text{STARTCYCLE}} + \overbrace{4\cdot(l-2)}^{\text{PRESMOOTH}} + \overbrace{2\cdot(l-1)}^{\text{POSTSMOOTH}} + \overbrace{2\cdot(l-1)}^{\text{RESTRICT}} + \overbrace{(l-1)}^{\text{PROLONG}} + \overbrace{2\cdot s}^{\text{SOLVE}} \\ &= 1 + 9\cdot(l-1) + 2\cdot s, \end{aligned}$$

as before with t being the number of traversals, l being the number of levels in the grid hierarchy, and s being the number of smoothing iterations used on the coarsest level to solve the problem there exactly. These are still $(l-1)$ traversals less than in the naive sequential implementation considered in Eq. 5.1.

Results for Distributed Memory Parallelisation

We did our experiments for distributed memory parallelisation again on the SuperMUC.

For the first experiment, we use 96 MPI ranks on 12 nodes. We run a hundred Jacobi smoothing iterations (consisting of runs of the adapters *computeResidual* and *mergeResidualAndSmooth* – see Alg. 5.2) on the grid at level eight for different buffer

Algorithm 5.2 Multigrid State-Machine for Parallel V(1,1) Cycle

```
activeLevel = finest level
switch state do
    case STARTCYCLE
        computeIntergridTransferOperators
        mergeIntergridTransferOperatorsAndComputeCoarseGridOperator
        mergeCoarseGridOperatorAndComputeResidual
        mergeResidualAndSmooth
        state = RESTRICT
    case PRESMOOTH
        mergeRestrictAndComputeIntergridTransferOperators
        mergeIntergridTransferOperatorsAndComputeCoarseGridOperator
        mergeCoarseGridOperatorAndComputeResidual
        mergeResidualAndSmooth
        state = RESTRICT
    case RESTRICT
        computeResidual
        mergeResidualAndRestrict
        activeLevel = activeLevel - 1
        if activeLevel == coarsestLevel then
            state = SOLVE
        else
            state = PRESMOOTH
        end if
    case SOLVE
        mergeRestrictAndComputeResidual
        mergeResidualAndSmooth
        for number of iterations needed to solve exactly do
            computeResidual
            mergeResidualAndSmooth
        end for
        state = PROLONG
```

Algorithm 5.2 Multigrid State-Machine for Parallel V(1,1) Cycle *(continued)*

```
    case PROLONG
        prolong
        activeLevel = activeLevel + 1
        state = POSTSMOOTH
    case POSTSMOOTH
        mergeProlongAndComputeResidual
        mergeResidualAndSmooth
        if activeLevel == finestLevel then
            break
        else
            state = PROLONG
        end if
end switch
```

sizes. Due to a very simple load balancing balancing (that is beyond the scope of this work), an overlap of one cell with the boundary is needed. That results in two additional cells per coordinate axis for a square domain. Therefore, level eight has 2,890,000 unknowns, which are duplicated at the parallel subdomain boundaries. The buffer size is the number of vertices at the the domain boundaries which are clustered before they are sent in an asynchronous communication step. If the buffer size is one, a copy of a vertex is sent to neighbouring ranks immediately after *touchVertexLastTime* is called for this vertex. For a larger buffer size, multiple vertices are first collected in an additional buffer of the given size. At the end of the traversal or whenever the buffer is full, the whole buffer is sent to all neighbours. This way, latency impact is reduced at cost of fewer data exchanges with increased data cardinality. The only global communication is the algorithm control and the reduction of the residual. We do these measurements in order to determine a reasonable buffer size to use.

In Tab. 5.4, we can see how the performance of the Jacobi smoother depends on the buffer size. The runtime comprises the whole communication and data merging of both adapters. Then, the average is taken of the runtime of both adapters in all hundred iterations. The best performance is achieved for a buffer size around 768, i.e., 768 vertices are collected and send in a bunch. This value is used for all subsequent experiments. Looking at the number of vertices at the boundaries, a buffer of this size means that all vertices are collected before they are sent, i.e., no vertices are send in the background. Assumably, this would change with a higher computational workload.

Next, we once more run Jacobi smoothing, split into the two adapters *computeResidual* and *mergeResidualAndSmooth*. The runs are performed on different grid

levels, i.e., for different numbers of unknowns and for different rank/node configurations. Again, an overlap is used, and unknowns are duplicated at the parallel subdomain boundaries.

In Tabs. 5.5 and 5.6, the runtimes and speedups on level eight are shown. Experiments for smaller problem sizes yield similar insight but come along with higher noise and overhead. Experiments for bigger problem sizes do not fit into the memory of a single SuperMUC node and thus require additional discussion on well-suited scaling and its impact in order to allow a comparison of runtimes.

An efficiency, i.e., speedup relative to the number of working MPI ranks, of around fifty percent is obtained in this example. We also see that it makes some difference how the tasks are distributed on the nodes. From textbook knowledge, we expect that communication between ranks assigned to the same node (i.e., sharing a common memory) is faster than inter-node communication, as the MPI realisation can handle such data exchange efficiently. Such an expectation suggests that the runtimes are the better the more ranks share a node. This expectation does not fit directly to our experimental results. Neither are the differences significant, nor is there a clear trend to an optimal rank-to-node assignment. A detailed analysis of the reasons is beyond the scope of this work. We just note that a performance degradation often comes into play when we assign more than two ranks per node. There might be correlations to the NUMA effects we already observed for shared memory parallelisation (see Sec. 5.3.1) or the fact that each processor is equipped with input/output devices of its own.

In Tabs. 5.7 and 5.8, we present runtime and speedup results for an adapter doing the intergrid transfer operator computation, and an adapter doing Jacobi smoothing and coarse grid operator computation (including parallel merges). The results are again for a fine grid of level eight. We see that the efficiency is similar to that of the Jacobi adapters: Again, the maximum efficiency reached is around fifty percent.

Analysing the load balancing and communication for the MPI ranks, it turns out that, although the load balancing is theoretically perfect in terms of the balance of the tree structure of the ranks, it does not take into account that the handling of the parallel subdomain boundary vertices is more expensive (due to merges) than the handling of other vertices. This has the effect that the ranks that are responsible for the subdomains at the boundary of the whole domain finish their work very fast, but the rank responsible for the centre of the domain needs much longer. Thus, the "fast" ranks have to wait for the "slow" rank for communication. For a higher computational workload, the administrative overhead at the subdomain boundaries would become less dominant, and therefore, this effect would become less significant.

In order to improve the parallel efficiency of simulations in the Peano framework, it might be worthwhile to study this behaviour in more detail, and to include the communication patterns into the load balancing process. As the present work only intends to use the parallelisation infrastructure of Peano, this is subject to other

work.

When comparing the results of this work with up-to-date parallel multigrid implementations, one should also keep in mind that we include the computation of the coarse grid operators and intergrid transfer operators into the runtime measurements. These tasks are for algebraic multigrid usually hidden away in a setup phase. But if one aims for solving dynamically adaptive problems, the recomputation of the operators becomes necessary, and therefore these computations have to be included into the runtime discussion. We showed that in our implementation, the additional operator computation increases the runtime of the other components, as the smoother, only by a small factor.

Jacobi	
Buffer Size	Time [s]
48	78.06
64	80.80
96	75.82
128	86.15
160	84.41
192	82.79
224	82.23
256	83.18
384	61.08
512	61.54
640	60.98
768	59.55
896	60.79
1024	60.61
1280	60.20
1536	60.59
1792	61.37
2048	61.33
2304	61.06
2560	61.40
2816	60.30
3072	59.90
3328	60.96
3584	60.38
3840	60.21
4096	59.88

Table 5.4: Runtimes of a hundred Jacobi smoothing iterations on a grid at level eight for different buffer sizes. Measurements with a buffer size of 32 and smaller ran into a timeout, i.e. individual processors had to wait more than 120 seconds.

computeResidual					
Worker Ranks	Nodes	Tasks/Node	Time [s]	Time [s]/Unknown	Speedup
1	1	1	518.065	0.0002	1.0
1	1	2	518.418	0.0002	0.9993
1	2	1	519.403	0.0002	0.9974
3	1	4	409.214	0.0001	1.2660
3	2	2	435.386	0.0002	1.1899
3	4	1	431.235	0.0001	1.2014
7	1	8	151.971	$5.25 \cdot 10^{-5}$	3.4090
7	2	4	152.068	$5.25 \cdot 10^{-5}$	3.4068
7	4	2	151.615	$5.24 \cdot 10^{-5}$	3.4170
7	8	1	152.531	$5.27 \cdot 10^{-5}$	3.3965
9	1	10	98.4291	$3.40 \cdot 10^{-5}$	5.2633
9	10	1	94.8550	$3.27 \cdot 10^{-5}$	5.4617
15	1	16	87.0087	$3.00 \cdot 10^{-5}$	5.9542
15	2	8	100.531	$3.47 \cdot 10^{-5}$	5.1533
15	4	4	90.2456	$3.11 \cdot 10^{-5}$	5.7406
15	8	2	89.0646	$3.07 \cdot 10^{-5}$	5.8167
15	16	1	93.9838	$3.24 \cdot 10^{-5}$	5.5123
19	2	10	95.0762	$3.28 \cdot 10^{-5}$	5.4489
19	10	2	95.0293	$3.28 \cdot 10^{-5}$	5.4516
31	2	16	81.498	$2.81 \cdot 10^{-5}$	6.3568
31	4	8	73.4493	$2.53 \cdot 10^{-5}$	7.0534
31	8	4	75.7137	$2.61 \cdot 10^{-5}$	6.8424
31	16	2	79.1297	$2.72 \cdot 10^{-5}$	6.5470
31	32	1	83.8835	$2.89 \cdot 10^{-5}$	6.1760
39	4	10	74.9587	$2.58 \cdot 10^{-5}$	6.9113
39	10	4	64.5914	$2.22 \cdot 10^{-5}$	8.0206
63	4	16	16.8722	$5.80 \cdot 10^{-6}$	30.7052
63	8	8	15.9732	$5.49 \cdot 10^{-6}$	32.4334
63	16	4	15.9005	$5.46 \cdot 10^{-6}$	32.5818
63	32	2	15.6592	$5.38 \cdot 10^{-6}$	33.0837
63	64	1	25.7467	$8.84 \cdot 10^{-6}$	20.1216
79	8	10	16.1229	$5.53 \cdot 10^{-6}$	32.1322
79	10	8	15.3009	$5.25 \cdot 10^{-6}$	33.8585
127	8	16	16.2726	$5.57 \cdot 10^{-6}$	31.8366
127	16	8	14.9439	$5.11 \cdot 10^{-6}$	34.6673
127	32	4	15.1203	$5.17 \cdot 10^{-6}$	34.2629
127	64	2	14.9072	$5.10 \cdot 10^{-6}$	34.7527
127	128	1	14.8106	$5.07 \cdot 10^{-6}$	34.9793

Table 5.5: Runtimes and speedups of a hundred residual computation iterations on a grid at level eight for different rank/node configurations.

mergeResidualAndSmooth					
Worker Ranks	Nodes	Tasks/Node	Time [s]	Time [s]/Unknown	Speedup
1	1	1	374.214	0.0001	1.0
1	1	2	375.753	0.0001	0.9959
1	2	1	375.59	0.0001	0.9963
3	1	4	302.146	0.0001	1.2385
3	2	2	305.992	0.0001	1.2230
3	4	1	301.493	0.0001	1.2412
7	1	8	113.783	$3.93 \cdot 10^{-5}$	3.2888
7	2	4	111.642	$3.86 \cdot 10^{-5}$	3.3519
7	4	2	110.264	$3.81 \cdot 10^{-5}$	3.3938
7	8	1	111.327	$3.84 \cdot 10^{-5}$	3.3614
9	1	10	75.8111	$2.62 \cdot 10^{-5}$	4.9361
9	10	1	73.5338	$2.54 \cdot 10^{-5}$	5.0890
15	1	16	68.5318	$2.36 \cdot 10^{-5}$	5.4604
15	2	8	76.7127	$2.65 \cdot 10^{-5}$	4.8781
15	4	4	71.5165	$2.47 \cdot 10^{-5}$	5.2326
15	8	2	65.1934	$2.25 \cdot 10^{-5}$	5.7401
15	16	1	73.4484	$2.53 \cdot 10^{-5}$	5.0949
19	2	10	73.3653	$2.53 \cdot 10^{-5}$	5.1007
19	10	2	72.7358	$2.51 \cdot 10^{-5}$	5.1449
31	2	16	69.4056	$2.39 \cdot 10^{-5}$	5.3917
31	4	8	61.4844	$2.12 \cdot 10^{-5}$	6.0863
31	8	4	59.9939	$2.07 \cdot 10^{-5}$	6.2375
31	16	2	65.5109	$2.26 \cdot 10^{-5}$	5.7122
31	32	1	61.9214	$2.13 \cdot 10^{-5}$	6.0434
39	4	10	58.6273	$2.02 \cdot 10^{-5}$	6.3829
39	10	4	51.7720	$1.78 \cdot 10^{-5}$	7.2281
63	4	16	15.1646	$5.21 \cdot 10^{-6}$	24.6768
63	8	8	13.9206	$4.78 \cdot 10^{-6}$	26.8820
63	16	4	14.0333	$4.82 \cdot 10^{-6}$	26.6661
63	32	2	13.7028	$4.71 \cdot 10^{-6}$	27.3093
63	64	1	21.7658	$7.48 \cdot 10^{-6}$	17.1928
79	8	10	14.5609	$5.00 \cdot 10^{-6}$	25.6999
79	10	8	13.9961	$4.80 \cdot 10^{-6}$	26.7370
127	8	16	15.2577	$5.22 \cdot 10^{-6}$	24.5262
127	16	8	13.1757	$4.51 \cdot 10^{-6}$	28.4018
127	32	4	13.5878	$4.65 \cdot 10^{-6}$	27.5404
127	64	2	13.5968	$4.65 \cdot 10^{-6}$	27.5222
127	128	1	13.5368	$4.63 \cdot 10^{-6}$	27.6442

Table 5.6: Runtimes and speedups of a hundred Jacobi update iterations on a grid at level eight for different rank/node configurations.

computeIntergridTransferOperators					
Worker Ranks	Nodes	Tasks/Node	Time [s]	Time [s]/Unknown	Speedup
1	1	1	738.7840	$2.56 \cdot 10^{-4}$	1.0000
1	1	2	1404.7100	$4.86 \cdot 10^{-4}$	0.5259
1	2	1	1414.4600	$4.89 \cdot 10^{-4}$	0.5223
3	1	4	1113.0200	$3.85 \cdot 10^{-4}$	0.6643
3	2	2	1175.9300	$4.07 \cdot 10^{-4}$	0.6287
3	4	1	1155.9700	$4.00 \cdot 10^{-4}$	0.6396
7	1	8	408.9730	$1.41 \cdot 10^{-4}$	1.8101
7	2	4	414.2570	$1.43 \cdot 10^{-4}$	1.7870
7	4	2	411.3070	$1.42 \cdot 10^{-4}$	1.7998
7	8	1	412.5780	$1.42 \cdot 10^{-4}$	1.7943
9	1	10	159.8880	$5.52 \cdot 10^{-5}$	4.6315
15	1	16	134.6760	$4.64 \cdot 10^{-5}$	5.5041
15	2	8	146.2480	$5.04 \cdot 10^{-5}$	5.0686
15	4	4	142.2710	$4.91 \cdot 10^{-5}$	5.2103
19	2	10	144.0450	$4.96 \cdot 10^{-5}$	5.1491
31	2	16	131.9980	$4.54 \cdot 10^{-5}$	5.6266
31	4	8	116.8570	$4.02 \cdot 10^{-5}$	6.3556
31	8	4	136.2300	$4.69 \cdot 10^{-5}$	5.4518
39	4	10	188.5980	$6.49 \cdot 10^{-5}$	3.9403
63	4	16	40.6201	$1.40 \cdot 10^{-5}$	18.3224
63	8	8	39.6569	$1.36 \cdot 10^{-5}$	18.7674
79	8	10	45.5196	$1.56 \cdot 10^{-5}$	16.3649
127	8	16	38.6299	$1.32 \cdot 10^{-5}$	19.3353

Table 5.7: Runtimes and speedups of a hundred iterations of the computation of the intergrid transfer operators on a fine grid at level eight for different rank/node configurations.

Jacobi + Galerkin					
Worker Ranks	Nodes	Tasks/Node	Time [s]	Time [s]/Unknown	Speedup
1	1	1	895.0750	$3.10 \cdot 10^{-4}$	1.0000
1	1	2	1630.2200	$5.64 \cdot 10^{-4}$	0.5491
1	2	1	1641.1400	$5.68 \cdot 10^{-4}$	0.5454
3	1	4	1293.0300	$4.47 \cdot 10^{-4}$	0.6928
3	2	2	1348.5100	$4.66 \cdot 10^{-4}$	0.6643
3	4	1	1347.7300	$4.66 \cdot 10^{-4}$	0.6646
7	1	8	472.0180	$1.63 \cdot 10^{-4}$	1.9001
7	2	4	475.8630	$1.64 \cdot 10^{-4}$	1.8847
7	4	2	475.9180	$1.64 \cdot 10^{-4}$	1.8845
7	8	1	477.3180	$1.65 \cdot 10^{-4}$	1.8790
9	1	10	177.2460	$6.12 \cdot 10^{-5}$	5.0618
15	1	16	155.5120	$5.36 \cdot 10^{-5}$	5.7750
15	2	8	170.0270	$5.86 \cdot 10^{-5}$	5.2820
15	4	4	163.1100	$5.63 \cdot 10^{-5}$	5.5060
19	2	10	167.6070	$5.78 \cdot 10^{-5}$	5.3614
31	2	16	155.4730	$5.35 \cdot 10^{-5}$	5.7876
31	4	8	135.5250	$4.66 \cdot 10^{-5}$	6.6395
31	8	4	158.4800	$5.45 \cdot 10^{-5}$	5.6778
39	4	10	219.0400	$7.53 \cdot 10^{-5}$	4.1104
63	4	16	45.3971	$1.56 \cdot 10^{-5}$	19.8626
63	8	8	45.0715	$1.55 \cdot 10^{-5}$	20.0061
79	8	10	51.9492	$1.78 \cdot 10^{-5}$	17.3730
127	8	16	44.3180	$1.52 \cdot 10^{-5}$	20.4191

Table 5.8: Runtimes and speedups of a hundred iterations of Jacobi smoothing and the computation of the coarse grid operators on a fine grid at level eight for different rank/node configurations.

6 Conclusion and Outlook

In this work, we developed a multigrid solver that combines the following properties: robustness and efficiency, suitability for parallelisation, and suitability for integration into a state-of-the-art spacetree-based software framework. The approach chosen is a combination of geometric and algebraic multigrid, with operator-dependent BoxMG intergrid transfer operators, (Petrov-)Galerkin coarse grid operators and a structured grid. The solver was designed and implemented in a cell-wise, strictly local manner in order to keep the communication overhead low on parallel computers. It was then integrated into and parallelised in the PDE solver framework Peano. All implementations were done for coarsening by a factor of three. The principle ideas, however, are also applicable for coarsening by other factors.

The BoxMG method without the restrictions posed by a cell-wise spacetree based implementation was found to be efficient and robust for coarsening by a factor of three also for challenging problems such as the convection-dominated convection-diffusion equation, jumping diffusion coefficients and recirculating flow problems. However, to achieve this level of robustness, line smoothers were required. These are not feasible in a strictly local spacetree setting, as well as point Gauss-Seidel smoothers. As multigrid solvers on structured grids highly depend on the smoother in use, we tested different alternatives to the Jacobi relaxation which is usually applied on spacetrees, but suffers from robustness and efficiency deficiencies. By using block-smoother-like hybrid smoothers, as often used in parallel multigrid, and a variant called “box smoothing”, we could achieve some improvement in efficiency and robustness. The efficiency of a point Gauss-Seidel smoother, though, could not be reached. This leads to the conclusion that, for a spacetree setting, hard problems as those mentioned above might require to ease the locality paradigm, for example by allowing access to nodes of more than two levels of a subtree at once. Thus, the blocks on which Gauss-Seidel or a similar efficient smoother can be performed would be increased. If the locality paradigm has to be preserved, it still remains unclear whether it is possible to realise more robust and efficient smoothers, though the present work already took a step towards that direction. For coarsening by a factor of two, the proposed smoother would probably yield a smaller improvement, as the “blocks” in the interior of a coarse grid cell where Gauss-Seidel smoothing can be applied would consist of only one vertex.

For the multigrid solver, we could show that the BoxMG approach improves the robustness on a structured grid a lot compared to using bilinear interpolation and

full weighting as restriction.

The integration of the designed algorithm into the Peano framework is more or less straightforward if the paradigms of the framework are followed carefully. The parallelisation, both for shared and distributed memory, requires special care for the computation of global information, such as the global residual norm. For distributed memory parallelisation, the information exchange at the domain boundaries has to be performed accurately, and the multilevel communication in intergrid transfer operations and in the computation of coarse grid operators poses some challenges. All in all, the usage of Peano paid off as many technical details (such as spacetree traversal, parallelisation, etc.) are hidden from the solver component. Peano's architecture and software paradigm could be proved to be a step towards maintainable and usable high performance computing software for state-of-the-art numerics. On the other hand, many improvements of the Peano framework were triggered due to the present work, and the underlying software's maturity has been improved due to the implementations of the presented solver.

Future tests for more complicated problems on bigger machines will reveal whether the present work is a step towards massively parallel state-of-the-art multigrid solvers.

Next steps that could be taken include the extension of our solver to adaptive grids. In the spacetree setting, this is straightforward, and also Peano offers the respective infrastructure. For this, it is a natural choice to switch from the correction scheme used in this work to a full approximation storage scheme, for example due to Griebel's Hierarchical Transformation Multigrid.

As noted in [109], the BoxMG scheme for coarsening by a factor of three can also be extended to 3D, as already done for coarsening by a factor of two in [32]. The number of γ and ι systems to be solved would increase in this case, and the points in the inner part of a 3D cell, called κ points in [109], would require to solve an additional 8×8 system.

Another step that could be taken is testing the performance of our solver as a preconditioner, for a example for a conjugate gradient solver. Furthermore, an extension to hybrid parallelisation, i.e., the combination of shared and distributed memory parallelisation, should be straightforward in Peano, given the parallel implementations we developed in this work.

While the extensions to 3D and adaptive settings are straightforward, the full potential of the present approach will in particular reveal for problems where the coarse grid systems change frequently. As such, dynamically adaptive problems and non-linear operators are of special interest. Our experiments have revealed that the permanent recomputation of the coarse grid systems is affordable with relatively low additional costs which pay off for such settings. Also the treatment of systems of PDEs might be interesting, as the algorithm can afford to increase the computational workload significantly without a significant runtime penalty. Such systems arise from

complex PDEs, but also from stochastic problems where multiple runs for equal or similar operators have to be performed in parallel.

Bibliography

[1] M. Adams, M. Brezina, J. Hu, and R. Tuminaro. Parallel Multigrid Smoothing: Polynomial versus Gauss-Seidel. *Journal of Computational Physics*, 188(2):593–610, 2003.

[2] Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee. The Opportunities and Challenges of Exascale Computing, 2010.

[3] R. E. Alcouffe, A. Brandt, J. E. Dendy, and J. W. Painter. The Multi-Grid Method for the Diffusion Equation with Strongly Discontinuous Coefficients. *SIAM Journal on Scientific and Statistical Computing*, 2(4):430–454, 1981.

[4] S. F. Ashby and R. D. Falgout. A Parallel Multigrid Preconditioned Conjugate Gradient Algorithm for Groundwater Flow Simulations. *Nuclear Science and Engineering*, 124:145–159, 1996.

[5] M. Bader. *Robuste, parallele Mehrgitterverfahren für die Konvektions-Diffusions-Gleichung.* Doctoral thesis, Technische Universität München, München, 2001.

[6] M. Bader. *Space-Filling Curves - An Introduction with Applications in Scientific Computing*, volume 9 of *Texts in Computational Science and Engineering.* Springer-Verlag, 2013.

[7] A. B. Baker, R. D. Falgout, T. V. Kolev, and U. Meier Yang. Multigrid Smoothers for Ultraparallel Computing. *SIAM Journal on Scientific Computing*, 33(5):2864–2887, 2011.

[8] A. H. Baker, R. D. Falgout, T. Gamblin, T. Kolev, M. Schulz, and U. Meier Yang. Scaling Algebraic Multigrid Solvers: On the Road to Exascale. In C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, editors, *Competence in High Performance Computing 2010*, pages 215–226. Springer Berlin Heidelberg, 2012.

[9] N. S. Bakhvalov. On the Convergence of a Relaxation Method with Natural Constraints on the Elliptic Operator. *USSR Computational Mathematics and Mathematical Physics*, 6(5):101–135, 1966.

[10] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a General Purpose Object Oriented Finite Element Library. *ACM Transactions on Mathematical Software*, 33(4):24/1–24/27, 2007.

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

[12] P. Bastian, W. Hackbusch, and G. Wittum. Additive and Multiplicative Multi-Grid – a Comparison. *Computing*, pages 345–364, 1998.

[13] B. Bergen, T. Gradl, F. Hülsemann, and U. Rüde. A Massively Parallel Multigrid Method for Finite Elements. *Computing in Science Engineering*, 8(6):56 –62, 2006.

[14] B. K. Bergen. *Hierarchical Hybrid Grids: Data Structures and Core Algorithms for Efficient Finite Element Simulations on Supercomputers*. Doctoral thesis, Technische Fakultät, Friedrich-Alexander Universität Erlangen-Nürnberg, 2005.

[15] J. H. Bramble, J. E. Pasciak, and J. Xu. Parallel Multilevel Preconditioners. *Mathematics of Computation*, 55:1–22, 1990.

[16] A. Brandt. Multi-Level Adaptive Technique (MLAT) for Fast Numerical Solution to Boundary Value Problems. In H. Cabannes and R. Temam, editors, *Proceedings of the Third International Conference on Numerical Methods in Fluid Mechanics*, volume 18 of *Lecture Notes in Physics*, pages 82–89. Springer Berlin Heidelberg, 1973.

[17] A. Brandt. Multi-level Adaptive Solutions to Boundary-Value Problems. *Mathematics of Computation*, 31:333–390, 1977.

[18] A. Brandt. Multigrid Solvers for Non-Elliptic and Singular-Pertubation Steady State Problems, 1981. unpublished.

[19] A. Brandt. Multigrid Techniques: 1984 Guide with Applications to Fluid Dynamics, 1984.

[20] A. Brandt, S. F. McCormick, and J. W. Ruge. Algebraic Multigrid (AMG) for Automatic Multigrid Solution with Application to Geodetic Computations. Technical report, Institute for Computational Studies, Colorado State University, Fort Collins, Colorado, 1982.

[21] A. Brandt, S. F. McCormick, and J. W. Ruge. Algebraic Multigrid (AMG) for sparse matrix equations. *Sparsity and its Applications*, pages 257–284, 1984.

[22] M. Brezina, R. D. Falgout, S. P. MacLachlan, T. A. Manteuffel, S. F. McCormick, and J. W. Ruge. Adaptive Smoothed Aggregation (αSA). *SIAM Journal on Scientific Computing*, 25:1896–1920, 2004.

[23] M. Brezina, T. A. Manteuffel, S. F. McCormick, J. Ruge, and G. Sanders. Towards Adaptive Smoothed Aggregation (αSA) for Nonsymmetric Problems. *SIAM Journal on Scientific Computing*, 32(1):14–39, 2010.

[24] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial.* Society for Industrial and Applied Math, 2nd edition, 2000.

[25] H.-J. Bungartz, M. Mehl, T. Weinzierl, and W. Eckhardt. DaStGen - A Data Structure Generator for Parallel C++ HPC Software. In M. Bubak, G. van Albada, P. Sloot, and J. Dongarra, editors, *ICCS 2008: Advancing Science through Computation, Part III*, volume 5103 of *Lecture Notes in Computer Science*, pages 213–222. Springer-Verlag, Heidelberg, Berlin, 2008.

[26] C. Burstedde, L. C. Wilcox, and O. Ghattas. `p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[27] E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro, and U. M. Yang. A Survey of Parallelization Techniques for Multigrid Solvers. In M. Heroux, P. Raghavan, and H. Simon, editors, *Parallel Processing for Scientific Computing*, 2006.

[28] Cobra. http://www.exascience.com/cobra/, last visited 03/17/2013.

[29] Dendro. http://www.cc.gatech.edu/csela/dendro/html/index.html, last visited 03/17/2013.

[30] J. E. Dendy. Black Box Multigrid. *Journal of Computational Physics*, 48(3):366–386, 1982.

[31] J. E. Dendy. Black Box Multigrid for Nonsymmetric Problems. *Applied Mathematics and Computation*, 13(3-4):261–283, 1983.

[32] J. E. Dendy. Two Multigrid Methods for Three-Dimensional Problems with Discontinuous and Anisotropic Coefficients. *SIAM Journal on Scientific and Statistical Computing*, 8(5):673–685, 1987.

[33] J. E. Dendy. Black Box Multigrid for Periodic and Singular Problems. *Applied Mathematics and Computation*, 25(1):1–10, 1988.

[34] J. E. Dendy. Black Box Multigrid for Systems. *Applied Mathematics and Computation*, 19:57–74, 1988.

[35] J. E. Dendy and J. D. Moulton. Black Box Multigrid with Coarsening by a Factor of Three. *Numerical Linear Algebra with Applications*, 17:577–598, 2010.

[36] W. Eckhardt and T. Weinzierl. A Blocking Strategy on Multicore Architectures for Dynamically Adaptive PDE Solvers. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, PPAM 2009*, volume 6068 of *Lecture Notes in Computer Science*, pages 567–575. Springer-Verlag, 2010.

[37] R. D. Falgout. An Introduction to Algebraic Multigrid. *Computing in Science and Engineering*, 8:24–33, 2006.

[38] R. P. Fedorenko. A Relaxation Method for Solving Elliptic Difference Equations. *USSR Computational Mathematics and Mathematical Physics*, 1(4):1092–1096, 1961.

[39] R. P. Fedorenko. The Speed of Convergence of One Iterative Process. *USSR Computational Mathematics and Mathematical Physics*, 4(3):227–235, 1964.

[40] W.-C. Feng and K. W. Cameron. The Green500 List. http://www.green500.org/, last visited 03/17/2013.

[41] C. Flaig and P. Arbenz. A Highly Scalable Matrix-Free Multigrid Solver for μFE Analysis Based on a Pointer-Less Octree. In I. Lirkov, S. Margenov, and J. Wasniewski, editors, *Large-Scale Scientific Computing*, volume 7116 of *Lecture Notes in Computer Science*, pages 498–506. Springer Berlin Heidelberg, 2012.

[42] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[43] A. Frank. *Organisationsprinzipien zur Integration von geometrischer Modellierung, numerischer Simulation und Visualisierung*. Doctoral thesis, Technische Universität München, 2000.

[44] B. Gmeiner, T. Gradl, H. Köstler, and U. Rüde. Highly Parallel Geometric Multigrid Algorithm for Hierarchical Hybrid Grids. In *NIC Symposium 2012 – Proceedings*, volume 45 of *Series of the John von Neumann Institute for Computing (NIC)*, pages 323–330, 2012.

[45] M. Griebel. *Zur Lösung von Finite-Differenzen- und Finite-Element-Gleichungen mittels der Hiearchischen-Transformations-Mehrgitter-Methode*. Doctoral thesis, Technische Universität München, 1990.

[46] M. Griebel. *Multilevelmethoden als Iterationsverfahren über Erzeugendensystemen*. Habilitation thesis, TU München, Stuttgart, 1994.

[47] M. Griebel and G. W. Zumbusch. Parallel Multigrid in an Adaptive PDE Solver Based on Hashing and Space-Filling Curves. *Parallel Computing*, 25:827–843, 1999.

[48] F. Günther. *Eine cache-optimale Implementierung der Finite-Elemente-Methode*. Doctoral thesis, Technische Universität München, 2004.

[49] W. Hackbusch. Ein iteratives Verfahren zur schnellen Auflösung elliptischer Randwertprobleme. Technical report, Universität Köln, Mathematisches Institut, Angewandte Mathematik, 1976.

[50] W. Hackbusch. On the Multi-Grid Method Applied to Difference Equations. *Computing*, 20(4):291–306, 1978.

[51] W. Hackbusch. *Multi-Grid Methods and Applications*, volume 4 of *Springer Series in Computational Mathematics*. Springer, Berlin, 1985.

[52] L. B. Hart, S. F. McCormick, A. O'Gallagher, and J. W. Thomas. The Fast Adaptive Composite-Grid Method (FAC): Algorithms for Advanced Computers. *Applied Mathematics and Computation*, 19(1-4):103–126, 1986.

[53] P. W. Hemker. The incomplete LU-decomposition as a relaxation method in multigrid algorithms. *Boundary and Interior Layers - Computational and Asymptotic Methods*, pages 306–311, 1980.

[54] C. Herzeel and P. Costanza. Dynamic Parallelization of Recursive Code: Part 1: Managing Control Flow Interactions with the Continuator. *SIGPLAN Notices – OOPSLA '10*, 45(10):377–396, 2010.

[55] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952.

[56] T. Huckle. Sparse Approximate Inverses and Multigrid Methods. In *Sixth SIAM Conference on Applied Linear Algebra*, Snowbird, UT, 1997.

[57] T. J. Hughes, I. Levit, and J. M. Winget. An Element-by-Element Solution Algorithm for Problems of Structural and Solid Mechanics. *Computer Methods in Applied Mechanics and Engineering*, 36:241–254, 1983.

[58] T. J. Hughes, I. Levit, and J. M. Winget. Element-by-Element Implicit Algorithms for Heat Conduction. *Journal of the Engineering Mechanics Division*, 109:576–585, 1983.

[59] F. Hülsemann, M. Kowarschik, M. Mohr, and U. Rüde. Parallel Geometric Multigrid. In A. M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, number 51 in Lecture Notes in Computational Science and Engineering, pages 165–208. Springer, 2005.

[60] R. Hüttl. *Ein iteratives Lösungsverfahren bei der Finite-Element-Methode unter Verwendung von rekursiver Substrukturierung und hierarchischen Basen.* Doctoral thesis, Technische Universität München, 1996.

[61] Intel Threading Building Blocks. http://threadingbuildingblocks.org/, last visited 03/17/2013.

[62] S. Kaczmarz. Angenäherte Auflösung von Systemen linearer Gleichungen. *Bulletin International de l'Académie Polonaise des Sciences et des Lettres*, 35:355–357, 1937.

[63] D. A. Knoll and D. E. Keyes. Jacobian-free Newton-Krylov methods: a Survey of Approaches and Applications. *Journal of Computational Physics*, 193(2):357–397, Jan. 2004.

[64] P. Kogge. The Tops in Flops. *Spectrum, IEEE*, 48(2):48–54, 2011.

[65] A. Krahnke. *Adaptive Verfahren höherer Ordnung auf cache-optimalen Datenstrukturen für dreidimensionale Probleme.* Doctoral thesis, TU München, 2005.

[66] S. P. MacLachlan, J. D. Moulton, and T. P. Chartier. Robust and Adaptive Multigrid Methods: Comparing Structured and Algebraic Approaches. *Numerical Linear Algebra with Applications*, 19(2):389–413, 2012.

[67] S. F. McCormick. *Multilevel Adaptive Methods for Partial Differential Equations.* Frontiers in Applied Mathematics. Society for Industrial and Applied Math, 1989.

[68] M. Mehl, T. Weinzierl, and C. Zenger. A Cache-Oblivious Self-Adaptive Full Multigrid Method. *Numerical Linear Algebra with Applications*, 13(2-3):275–291, 2006.

[69] U. Meier Yang. On the Use of Relaxation Parameters in Hybrid Smoothers. *Numerical Linear Algebra with Applications*, 11(2-3):155–172, 2004.

[70] U. Meier Yang. Parallel Algebraic Multigrid Methods - High Performance Preconditioners. In A. M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture*

Notes in Computational Science and Engineering, pages 209–236. Springer Berlin Heidelberg, 2006.

[71] J. D. Moulton, J. E. Dendy, and J. M. Hyman. The Black Box Multigrid Numerical Homogenization Algorithm. *Journal of Computational Physics*, 142(1):80–108, 1998.

[72] Message Passing Interface. http://www.mcs.anl.gov/research/projects/mpi/, last visited 03/17/2013.

[73] S. Nogina, K. Unterweger, and T. Weinzierl. Autotuning of Adaptive Mesh Refinement PDE Solvers on Shared Memory Architectures. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *PPAM 2011*, volume 7203 of *Lecture Notes in Computer Science*, pages 671–680, Heidelberg, Berlin, 2012. Springer-Verlag.

[74] C. W. Oosterlee and T. Washio. On the Use of Multigrid as a Preconditioner. In P.E.Bjørstad, M. Espedal, and D. Keyes, editors, *Ninth International Conference on Domain Decomposition Methods*, 1998.

[75] OpenMP. http://openmp.org/, last visited 03/17/2013.

[76] J. M. Ortega and W. C. Rheinboldt. Iterative Solution of Nonlinear Equations in Several Variables. *Academic Press*, 1970.

[77] M. Pögl. *Entwicklung eines cache-optimalen 3D Finite-Element-Verfahrens für große Probleme*. Doctoral thesis, Technische Universität München, Düsseldorf, 2004.

[78] U. Rüde. *Mathematical and Computational Techniques for Multilevel Adaptive Methods*. Habilitation thesis, Technische Universität München, Philadelphia, 1993.

[79] Y. Saad and M. H. Schultz. GMRES: a Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[80] H. Sagan. *Space-Filling Curves*. Springer, 1 edition, 1994.

[81] R. S. Sampath and G. Biros. A Parallel Geometric Multigrid Method for Finite Elements on Octree Meshes. *SIAM Journal on Scientific Computing*, 32(3):1361–1392, 2010.

[82] M. Schneider. *Verteilte adaptive numerische Simulation auf der Basis der Finite-Elemente-Methode*. Doctoral thesis, Technische Universität München, 1995.

[83] M. Schreiber and T. Weinzierl. SFC-Based Communication Metadata Encoding for Adaptive Mesh Refinement. In *ParCo 2013 Proceedings*, Advances of Parallel Computing, 2013. submitted.

[84] T. Scogland, B. Subramaniam, and W.-C. Feng. Emerging Trends on the Evolving Green500: Year Three. In *7th Workshop on High-Performance, Power-Aware Computing*, Anchorage, Alaska, USA, May 2011.

[85] R. Stevenson. Modified ILU as a smoother. *Numerische Mathematik*, 68:295–309, 1994.

[86] K. Stüben. Algebraic Multigrid (AMG): Experiences and Comparisons. *Applied Mathematics and Computation*, 13(3-4):419–451, 1983.

[87] K. Stüben. Algebraic Multigrid (AMG): An Introduction with Applications, 1999.

[88] K. Stüben. Appendix a: An Introduction to Algebraic Multigrid. In U. Trottenberg, C. W. Oosterlee, and A. Schüller, editors, *Multigrid*. Elsevier Science Inc., 2001.

[89] K. Stüben and J. W. Ruge. Algebraic Multigrid. In S. F. McCormick, editor, *Multigrid Methods*, volume 3 of *Frontiers in Applied Mathematic*, pages 73–130. Society for Industrial and Applied Mathematics, Philadelphia, 1987.

[90] K. Stüben and U. Trottenberg. Multigrid Methods: Fundamental Algorithms, Model Problem Analysis and Applications. In W. Hackbusch and U. Trottenberg, editors, *Multigrid Methods*, volume 960 of *Lecture Notes in Mathematics*. Springer Berlin Heidelberg, 1982.

[91] H. Sundar, G. Biros, C. Burstedde, J. Rudi, O. Ghattas, and G. Stadler. Parallel Geometric-Algebraic Multigrid on Unstructured Forests of Octrees. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 43:1–43:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[92] SuperMUC. http://www.lrz.de/services/compute/supermuc/systemdescription/, last visited 03/17/2013.

[93] W.-P. Tang and W. L. Wan. Sparse Approximate Inverse Smoother for Multigrid. *SIAM Journal on Matrix Analysis and Applications*, 21:1236–1252, 1999.

[94] T. E. Tezduyar and J. Liou. Element-by-Element and Implicit-Explicit Finite Element Formulations for Computational Fluid Dynamics. In *Proceedings from the 1st International Symposium on Domain Decomposition Methods for*

Partial Differential Equations, pages 281–300, Philadelphia, PA, USA, 1988. Society for Industrial and Applied Mathematics.

[95] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid.* Academic Press, San Diego, U.S.A., 2001.

[96] P. Vaněk. Acceleration of Convergence of a Two-Level Algorithm by Smoothing Transfer Operators. *Applications of Mathematics*, 37:265–274, 1992.

[97] P. Vaněk, M. Brezina, and J. Mandel. Convergence of Algebraic Multigrid Based on Smoothed Aggregation. *Numerische Mathematik*, 88:559–579, 2001.

[98] P. Vaněk, J. Mandel, and M. Brezina. Algebraic Multigrid by Smoothed Aggregation for Second and Fourth Order Elliptic Problems. *Computing*, 56:179–196, 1996.

[99] T. Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids.* Doctoral thesis, Technische Universität München, 2009.

[100] T. Weinzierl et al. Peano – A Framework for PDE Solvers on Spacetree Grids, 2012. http://www.peano-framework.org.

[101] T. Weinzierl and M. Mehl. Peano – A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids. *SIAM Journal on Scientific Computing*, 33(5):2732–2760, 2011.

[102] P. Wesseling and C. W. Oosterlee. Geometric Multigrid with Applications to Computational Fluid Dynamics. *Journal of Computational and Applied Mathematics*, 128:311–334, 2001.

[103] R. Wienands and W. Joppich. *Practical Fourier Analysis for Multigrid Methods.* Chapman & Hall/CRC, Boca Raton, FL, USA, 2005.

[104] R. Wienands and H. Köstler. A Practical Framework for the Construction of Prolongation Operators for Multigrid Based on Canonical Basis Functions. *Computing and Visualization in Science*, 13:207–220, 2010.

[105] J. M. Winget. *Element-by-Element Solution Procedures for Nonlinear Transient Heat Conduction Analysis.* Ph.d. thesis, California Institute of Technology, 1984.

[106] G. Wittum. On the Robustness of ILU Smoothing. *SIAM Journal of Scientific and Statistical Computing*, 10(4):699–717, 1989.

[107] I. Yavneh. Coarse-Grid Correction for Nonelliptic and Singular Perturbation Problems. *SIAM Journal on Scientific Computing*, 19(5):1682–1699, 1998.

[108] I. Yavneh. Why Multigrid Methods Are So Efficient. *Computing in Science and Engineering*, 8:12–22, 2006.

[109] I. Yavneh and M. Weinzierl. Nonsymmetric Black Box multigrid with coarsening by three. *Numerical Linear Algebra with Applications*, 19(2):246–262, 2012.

[110] K. Yelick. Ten Ways to Waste a Parallel Computer. Talk at the 36th International Symposium on Computer Architecture (ISCA), June 2009.