Data Structures and Algorithms for the Optimization of Hierarchical Hybrid Multigrid Methods

Datenstrukturen und Algorithmen zur Optimierung hierarchisch-hybrider Mehrgittermethoden

Der Technischen Fakultät der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

zur Erlangung des Doktorgrades

Doktor-Ingenieur

vorgelegt von

Tobias Gradl
aus Kösching

Als Dissertation genehmigt
von der Technischen Fakultät
der Friedrich-Alexander-Universität Erlangen-Nürnberg

Tag der mündlichen Prüfung: 8. Dezember 2015

Vorsitzender des Promotionsorgans: Prof. Dr. Peter Greil

Gutachter: Prof. Dr. Ulrich Rüde
Prof. Dr. Günter Leugering

Abstract

Multigrid methods are among the theoretically most efficient algorithms in numerical simulation. They solve certain classes of equations—e. g., those arising from finite element (FE) discretizations—with optimal complexity. Practically relevant for large-scale simulations, however, are only algorithms that exploit the massive parallelism that characterizes today's high-performance computing landscape. Implementing multigrid methods efficiently on massively parallel computers is challenging, because for some of the core algorithms the distribution of the numerical operations to many processors is not straightforward.

Bergen et al. proved with the *Hierarchical Hybrid Grids* (HHG) software framework that it is possible to solve FE simulations efficiently with multigrid methods on supercomputers [10]. The central concept of HHG is to discretize the simulated domains into patch-wise structured meshes. It facilitates the distribution of the computational work to many processors, but it also restricts HHG's flexibility regarding the types of numerical problems it can be applied to.

This thesis presents performance studies for FE simulations with up to 3×10^{11} degrees of freedom that demonstrate short time to solution and good scalability of HHG on up to 16 384 processor cores. We describe the modifications to the initial version of HHG [10]—e. g., in the build system and the performance measurement methods—that were necessary in order to execute and study HHG on systems of this size.

The central chapter of the thesis is dedicated to adaptive mesh refinement (AMR). The technique makes HHG applicable to a new class of problems, which is characterized by a strong variance in the required mesh resolution across the domain, e. g., the simulation of room acoustics or turbulent flows. AMR allows for the FE mesh to be tailored flexibly to the simulation's characteristics. The multigrid solver can thus spend the computer's resources—memory and processor cycles—on areas where the simulation requires a high resolution. In consequence, the time to solution decreases and the problem size that can be handled increases. When implementing AMR for HHG, it was important to maintain the numerical and software engineering concepts that are crucial for HHG's performance and scalability. We describe how the algorithms and data structures were extended in order to achieve this goal.

There are many other techniques for optimizing the distribution of computational resources in multigrid algorithms. As a contrast to AMR, we present a technique that was developed in joint work with Thekale et al. [47]. A branch and bound search is used to find the optimal number of V-cycles on each level of a full multigrid algorithm. By performing V-cycles on the levels where they yield the best ratio between error reduction and cost, the time to solution of a full multigrid run in a realistic scenario was reduced by 35%.

Zusammenfassung

Mehrgittermethoden gehören zumindest theoretisch zu den effizientesten Algorithmen in der Numerischen Simulation. Einige Klassen von Gleichungen, z.B. die bei der Disktretisierung mit Finiten Elementen (FE) entstehenden Gleichungssysteme, sind damit unter bestimmten Voraussetzungen in optimaler Komplexität lösbar. Für die Anwendung im High-Performance-Computing sind jedoch nur Methoden relevant, die den extremen Parallelismus aktueller Supercomputer ausnutzen können. Mehrgittermethoden effizient für Parallelrechner zu implementieren ist eine Herausforderung, weil für einige der zentralen Algorithmen das Verteilen der numerischen Operationen auf viele Prozessoren nicht trivial ist.

Mit dem Software-Framework *Hierarchical Hybrid Grids* (HHG) zeigten Bergen et al., daß effiziente FE-Simulationen mit Mehrgittermethoden auf Supercomputern möglich sind [10]. Das zentrale Konzept von HHG ist die Diskretisierung des simulierten Gebiets in abschnittsweise strukturierte Gitter. Das erleichtert das Verteilen der Rechenoperationen auf viele Prozessoren, schränkt allerdings auch die Anwendbarkeit von HHG auf bestimmte numerische Probleme ein.

Diese Arbeit demonstriert mit Performance-Studien auf bis zu 16 384 Prozessorkernen und FE-Simulationen mit bis zu 3×10^{11} Freiheitsgraden die hohe Effizienz und Skalierbarkeit von HHG. Um HHG auf Systemen dieser Größe ausführen und analysieren zu können, waren Änderungen an der ursprünglichen HHG-Version [10] nötig, z.B. am Build-System und an den Methoden zur Performance-Messung.

Das zentrale Kapitel der Arbeit widmet sich der adaptiven Gitterverfeinerung (AMR, von *adatpive mesh refinement*). Diese Technik erweitert den Anwendungsbereich von HHG auf Probleme mit starker räumlicher Varianz in der benötigten Gitterweite, z.B. die Simulation von Raumakustik oder von turbulenten Strömungen. AMR ermöglicht eine flexible Anpassung des FE-Gitters an die Simulationscharakteristika. So können die Ressourcen des Computers – Speicher und Prozessorzyklen – gezielt dort eingesetzt werden, wo eine hohe Gitterauflösung nötig ist, und damit die Rechenzeit verringert und die lösbare Problemgröße erhöht werden. Bei der Implementierung von AMR in HHG war es wichtig, die Konzepte aus der Numerik und aus dem Software-Engineering, die für die Performance und Skalierbarkeit von HHG entscheidend sind, zu erhalten. Die Arbeit beschreibt, wie die Algorithmen und Datenstrukturen erweitert wurden, um dieses Ziel zu erreichen.

Eine weitere Methode zur Optimierung von Mehrgittermethoden wurde in Zusammenarbeit mit Thekale et al. entwickelt [47]. Mit einer Branch-And-Bound-Suche wird die Verteilung von V-Zyklen im Full-Multigrid-Algorithmus optimiert. V-Zyklen werden gezielt auf den Leveln ausgeführt, wo sie das beste Verhältnis aus Fehlerreduktion und Kosten erzielen. Mit dieser Methode wurde in einem realistischen Szenario eine Verringerung der Laufzeit von Full Multigrid um 35% erreicht.

Acknowledgments

I would like to thank my colleagues, friends, and family, who accompanied me during my studies and the work for this thesis. You supported me in all kinds of ways, scientifically and personally, critically and encouragingly, and all these aspects were important. I am particularly grateful to my advisor Prof. Dr. Ulrich Rüde for giving me the chance to work in his research group and for continuously supporting me throughout the rather long phase of my research. I also wish to thank Erich Strohmaier and his research group, who hosted me during my stay at Lawrence Berkeley National Laborartory.

My research was funded partly by the Elite Network of Bavaria, the Friedrich-Alexander University Erlangen-Nürnberg, and the Distributed European Infrastructure for Supercomputing Applications (DEISA) project.

Tobias Gradl

Contents

Chapter 1

Introduction

Centuries ago, scientists already used numerical algorithms to predict physical phenomena quite precisely. In 1801, for example, Carl Friedrich Gauss employed the method of least squares to predict the position of the dwarf planet Ceres. Centuries later, there are still phenomena, for example, earthquakes, that even the most advanced computer simulations are not able to predict with sufficient accuracy. The difficulty of simulating a physical phenomenon depends on many factors. Among the most important ones are the scale of the simulated domain, the type and the complexity of the physical interactions, and the time scale that has to be resolved. This variety in the problems' characteristics has naturally lead to a variety of methods for solving them.

A certain group of problems, for which no feasible solution methods are available, so far, is characterized by a combination of two extremes: a large domain and a high resolution. For example, the physical interactions that need to be simulated may take place on small scales, but, on the other hand, extend over large dimensions. Certain types of acoustics simulations belong to this group. The direct simulation of sound propagation, which is based on computing the air pressure differences induced by the sound waves, requires a spatial resolution in the sub-millimeter range in order to resolve the full range of audible frequencies. On the other hand, sound propagates over large distances. In order to simulate the acoustics of, for example, a concert hall, a volume of some thousand cubic meters has to be covered. This means that the sound pressure has to be computed at around 10^{12} points in space.

At the same time, the domains under consideration—like the concert hall—are often not regular, but have objects in the interior and boundaries with small details that the simulation needs to resolve. This is another pair of extremes: large homogeneous areas with no relevant geometric features are contrasted by areas with small details that nevertheless influence the result of the simulation. A similar situation is created by irregularities of the simulated phenomena, for example, by vortices in a flow or by discontinuities that are induced by the combination of different materials. In order to solve such problems with feasible effort, the solver needs to adapt to the problem by concentrating the computational effort where the irregularities occur.

This thesis discusses numerical methods, and their implementations, that are tailored to this group of problems. They combine adaptivity with the capability to perform extremely large-scale simulations. The basis is the *finite element method*, a discretization technique that is especially suited for irregular physical domains. As part of this process, *adaptive mesh refinement* is employed in order to adapt the

mesh to the problem's irregularities. For solving the resulting *sparse, linear system of equations*, the usage of *multigrid methods* ensures high performance on current computer architectures and scalability to extremely large simulations. A parallel implementation opens up the access to state of the art, massively parallel supercomputers.

Benjamin Karl Bergen presented an implementation of this approach and scalability studies for up to 1024 processors in his dissertation [10]. The implementation was named *Hierarchical Hybrid Grids* (HHG). Björn Gmeiner presented applications of HHG and scalability studies up to the Petaflops range in his dissertation [18]. So far, one of the main restrictions of HHG was the missing adaptivity. This dissertation presents the theory and the implementation of adaptive mesh refinement with HHG.

The following chapter provides introductory information about finite element methods, multigrid methods, and HHG, that are required as a basis for the remainder of the thesis. Chapter 3 presents the results of scalability experiments conducted on various supercomputers. Chapter 4 discusses the concept of adaptive mesh refinement and how it is implemented in HHG. A completely different way of adapting the multigrid algorithm to the problem's characteristics is to adaptively distribute the computational effort between the different phases of the solver. This approach is presented in Chapter 5.

Chapter 2

Basics

Contents

2.1 The model problem

Most chapters of this thesis refer to a common model problem. The intent of this thesis is to present novel methods that are *generally* useful for improving the performance of multigrid finite element solvers. The intent is not to present a solver that is tailored to a specific problem. Therefore, we select a model problem that can be handled without complications both by the finite element method and the multigrid

method. The advantage is that lengthy elaborations of special cases can be avoided and the discussion can focus on the details of the new algorithms and their implementation.

Our model problem

$$Au = f$$

is *Poisson's equation*, a linear, elliptic *partial differential equation* (PDE) with the scalar variable u. The differential operator A is the scaled Laplace operator

$$-\mu\Delta \equiv -\mu\frac{\partial^2}{\partial \mathbf{x}} : \mathbb{R}^d \mapsto \mathbb{R}.$$

The Laplace operator is a second-order differential operator. the scaling factor μ is a positive real number.

The equation is defined on the d-dimensional, simply-connected, open domain

$$\Omega \in \mathbb{R}^d$$

with the boundary Γ. The boundary is not considered part of the domain; the domain including its boundary is denoted with

$$\overline{\Omega} = \Omega \cup \Gamma.$$

Dirichlet and Neumann boundary conditions are considered:

$$\Gamma = \Gamma^D \cup \Gamma^N,\ \Gamma^D \cap \Gamma^N = \emptyset.$$

On the Dirichlet boundary Γ^D the solution is defined by the function

$$g : \mathbb{R}^d \mapsto \mathbb{R}.$$

On the Neumann boundary Γ^N the derivative along the normal vector of the boundary is defined by the function

$$h : \mathbb{R}^{d-1} \mapsto \mathbb{R},\ h(x) = \frac{\partial f}{\partial n}(x),$$

where $\partial/\partial n$ is the *normal derivative*, which is defined as

$$\frac{\partial f}{\partial n}(x) = \nabla f(x) \cdot n(x)$$

with $n(x)$ being the outward-pointing normal vector of Γ at x.

This completes our model problem:

$$\begin{aligned} -\mu\Delta u &= f \quad \text{in} \quad \Omega \\ u &= g \quad \text{on} \quad \Gamma^D \\ \mu\frac{\partial u}{\partial n} &= h \quad \text{on} \quad \Gamma^N. \end{aligned} \tag{2.1}$$

HHG operates on three-dimensional domains ($d = 3$). In some explanations and examples $d = 1$ or 2 are used for simplicity.

Poisson's equation is quite an easy problem for numerical methods—disregarding possible complications caused by special characteristics of Ω or f. Finite element and multigrid methods have also been applied to harder problems, though, e. g., with nonlinear [48, Chapter 5.3] or non-elliptic differential operators [48, Chapter 5.1], other boundary conditions [44], and on unbounded domains [38]. Applying the methods presented in this thesis to such problems is an interesting topic for future research.

2.2 Finite element methods

2.2.1 Introduction

For many important differential equations—like the *Navier-Stokes equations* for incompressible flows—an analytic solution is not known. Therefore, numerical methods have been developed in order to get at least a good approximation to the equation's solution. One of them is the finite element method. Nowadays, most readers will know these methods as computer programs, but numerical mathematics does not depend on the availability of machines doing the computations. With enough computing power in the form of people with pencil and paper at hand, they can be, and in fact have been, powerful tools for solving differential equations.

In contrast to other numerical methods for solving PDEs, the finite element method is suited especially well for irregularly-shaped domains. While for many other methods even an L-shaped or circular domain already poses a difficult problem to the modeler and implementer of the method, the finite element method can naturally deal with complex shapes like airplane wings, cars, or the earth's continents.

Because of its flexibility in adapting to complex geometries the finite element method is often the first choice in mechanical engineering applications. The method also has many options for tuning it towards an application's specific needs. Making trade-offs between precision and time to solution—even spatially adapting these parameters—is easily possible without complicating the method. Due to this feature, both large *and* accurate numerical simulations can be implemented without expert knowledge in numerical mathematics. A result of these properties is a variety of computer programs offering user-friendly, stable, and efficient solvers for mechanical engineering and other simulations.

Somewhat more disputed is the question whether the finite element method is suited well for parallel computing. Undoubtedly, there are methods, like the finite difference method, for which it is easier to create parallel implementations. However, the advantages listed above justify the additional effort. And, as Chapter 3 will show, with proper software engineering the implementation of a parallel finite element solver with good performance is possible.

2.2.2 Numerical quadrature

In order to integrate a function analytically, it is necessary to calculate the function's primitive. Since that is not possible in general, various numerical methods for computing integrals approximately have been developed. These techniques are subsumed under the term *numerical quadrature*. The quadrature methods used in HHG approximate the integral of a function $f(x)$ by replacing it with a weighted sum of function values at certain quadrature nodes x_i.

$$\int_\Omega f(x)\,d\Omega \approx \sum_{i=1}^{N} w_i f(x_i) \tag{2.2}$$

There is a variety of rules on choosing the w_i. For a numerical quadrature rule to be useful in practice, it is necessary that its approximation error is bounded and that it can be estimated. Simpson's rule [45, p. 203], for example, is known to integrate polynomials of third degree exactly. Shaidurov has collected a wide variety of

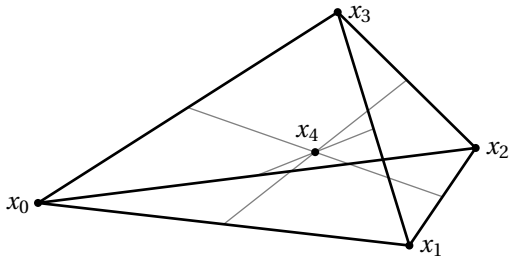

Figure 2.1: A tetrahedron with quadrature nodes.

Table 2.1: Shaidurov's quadrature rule for tetrahedra. The quadrature nodes are given in barycentric coordinates.

node	λ_0	λ_1	λ_2	λ_3	weight
x_0	1	0	0	0	$\frac{1}{120}$
x_1	0	1	0	0	$\frac{1}{120}$
x_2	0	0	1	0	$\frac{1}{120}$
x_3	0	0	0	1	$\frac{1}{120}$
x_4	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{2}{15}$

quadrature rules and their error estimations for one-, two-, and three-dimensional functions in [41].

None of these rules can be applied to arbitrary domains Ω, though. A rule's approximation error can only be estimated with reasonable bounds, if Ω has a well-defined shape. Thus, the three defining criteria for every rule are

- the shape of Ω,
- the locations of the quadrature nodes x_i, and
- the weights w_i at these points.

HHG currently includes quadrature rules for tetrahedra. A tetrahedron is a volume enclosed by four triangular faces (see Fig. 2.1). Every tetrahedron has six edges and four vertices. Shaidurov's quadrature rule for tetrahedra, which is used in HHG, defines five quadrature nodes. In Table 2.1 the coordinates of these nodes $(x_0,\dots,x_4)$ are given in the barycentric coordinate system defined by the tetrahedron's vertices. One node is located at each vertex and one inside the volume. The table also assigns a weight w_i to each of the nodes. The weight is $1/120$ for the nodes at the tetrahedron's vertices and $2/15$ for the node in the center. This quadrature rule integrates polynomials of degree two exactly

2.2.3 Integrals over non-trivial domains

Quadrature rules for which error bounds can be given are only available for a limited set of rather simple geometric shapes. For more complicated shapes, usable

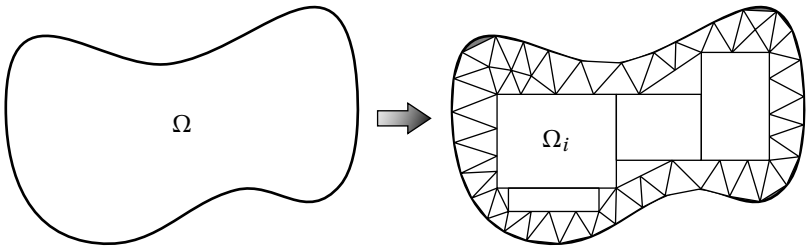

Figure 2.2: Partitioning of a two-dimensional domain Ω into subdomains Ω_i.

quadrature rules could be developed, too, but at a high effort. For arbitrarily shaped domains, however, it is not possible at all to state a usable quadrature rule.

One of the central ideas of the finite element method comes into play at this point. In order to integrate a function over arbitrarily shaped domains, the method makes use of the linearity of the integral operator:

$$\int_\Omega f\,d\Omega = \sum_i \int_{\Omega_i} f\,d\Omega_i \quad \text{where} \quad \Omega = \bigcup_i \Omega_i \quad \text{and} \quad \Omega_i \cap \Omega_j = \emptyset \ \text{ for all } i \neq j\,. \tag{2.3}$$

An integral over an irregularly-shaped domain Ω can thus be calculated as the sum of integrals over subdomains Ω_i that have simpler shapes, for which the integral can be calculated—analytically or numerically. An example for the partitioning of a two-dimensional domain Ω is shown in Fig. 2.2. Such a partitioning will be called *finite element mesh* with the subdomains Ω_i representing the *finite elements*.

The above example already shows some interesting aspects of this technique. First, finite elements mostly—in this thesis: always—have non-curved boundaries, although elements with curved boundaries have successfully been used for special applications [3, p. 207]. The reason for preferring non-curved elements is that these are the easiest ones to integrate over.

The major drawback of these elements is also visible in the figure. Using only elements with non-curved boundaries, it is harder to approximate irregularly-shaped domains with curved boundaries. The areas of Ω that are not padded with subdomains are highlighted in gray in Fig. 2.2. However, the same problem exists for elements with curved boundaries, although possibly to a smaller degree, depending on the shape of Ω. In general, one has to accept that the finite element method's domain partitioning introduces *modeling errors* of this kind.

Among the rectilinear elements, the different types of elements have their advantages in different situations. In Fig. 2.2 rectangular elements are used to fill large parts of the domain with only a few elements. The triangular elements, in contrast, are better for approximating the domain's curved boundary.

2.2.4 The weak form of a PDE

How does being able to integrate over arbitrary domains help us solving PDEs on these domains? The second central idea of the finite element method is to transform the PDE into a form with integrals. In the following paragraphs we show the steps involved in the transformation. After that, we need to show that the solution of the transformed equation is also a solution of the original PDE.

Recall our model problem (2.1). For the moment, we assume that the problem has only homogeneous Dirichlet boundary conditions:

$$\begin{aligned} -\mu\Delta u &= f \quad \text{in} \quad \Omega \\ u &= 0 \quad \text{on} \quad \Gamma. \end{aligned} \tag{2.4}$$

Section 2.2.5 will show how to deal with Neumann and inhomogeneous boundary conditions.

Let us assume that we are satisfied with a solution u that fulfills this PDE "on average". Then we may compute such a *weighted average* by multiplying the PDE with a weighting function v defined in Ω, and integrating:

$$-\int_\Omega \mu\Delta u v = \int_\Omega f v. \tag{2.5}$$

The weighting function v is known as *test function*, because its purpose is to test whether u fulfills the PDE on average.

If arbitrary functions v were admitted, (2.5) would hold for any function u, and the equation would not be very useful for our purposes. On the other hand, it is possible to select an appropriate function space for v such that every u that fulfills (2.5) also fulfills (2.4). Provided that f is continuous, any function u solving (2.4), as well as its partial derivatives up to order two, must be continuous on Ω, and u must be 0 on Γ. These properties are collected in the definition of the subspace

$$C_D^2(\overline{\Omega}) = \left\{ u \in C^2(\overline{\Omega}) : u = 0 \text{ on } \Gamma \right\},$$

where $C^2(\overline{\Omega})$ is the space of twice continuously differentiable functions defined in $\overline{\Omega}$. It is sufficient to also consider only this space for choosing the test functions. In fact, any $u \in C_D^2$ is a solution of (2.5) for all $v \in C_D^2$, if and only if u is a solution of (2.4). For a detailed proof of this theorem, see [22], page 20. Its basic idea is that C_D^2 contains functions v that are greater than 0 only in a ball with radius δ around a point x_0 and have a total weight of 1 (i. e., their integral over the ball evaluates to 1). Using these functions for v, the integrals in (2.5) are weighted averages of $\mu\Delta u$ and f over the ball. Letting $\delta \to 0$ at all points $x_0 \in \Omega$ drives the weighted averages towards the original values of (2.4).

Compared to the original PDE, the integral form has weaker restrictions on the right-hand side. While (2.4) requires f to be continuous over $\overline{\Omega}$, in (2.5) it is only necessary that fv is integrable over Ω. The PDE can be relaxed even further using *Green's identity*, also known as *Gauss's theorem*,

$$-\int_\Omega \Delta u v = \int_\Omega \nabla v \cdot \nabla u - \int_\Gamma v \frac{\partial u}{\partial n}.$$

For a proof of Green's identity see [22], page 17. When taking v from $C_D^2(\overline{\Omega})$, the boundary integral in Green's identity vanishes, because $v = 0$ on Γ. Thus, applying Green's identity to (2.5) yields

$$\int_\Omega \mu\nabla u \cdot \nabla v = \int_\Omega f v. \tag{2.6}$$

With this transformation the requirements on the solution have been relaxed considerably. While u has to be twice differentiable in (2.4), only first derivatives of

u and v are required in (2.6). This change motivates a re-analysis of the function spaces used for solving the PDE.

First of all, note that the weighted average of a partial derivative $\partial u/\partial x$ can be transformed with Green's identity:

$$\int_\Omega \frac{\partial u}{\partial x} v \;=\; -\int_\Omega u \frac{\partial v}{\partial x} \text{ for all } v \in C_D^\infty(\overline{\Omega}).$$

In this context $\partial u/\partial x$ is called *weak partial derivative* of u (the notation is the same as for strong derivatives). If such a weak derivative exists, u is said to be *weakly differentiable*.

Equation (2.6) contains two weak derivatives, ∇u and ∇v, in an integral. For the possibility that $u = v$ the squared weak derivatives are required to be integrable, i. e.,

$$\int_\Omega \left(\frac{\partial u}{\partial x_i}\right)^2 < \infty \text{ for all } i = 1, \ldots, N.$$

The same argument applies for the right-hand side of (2.6). If $f = v$, then f and v must be square-integrable.

These requirements are gathered into the definition of the *Hilbert spaces*

$$H^1(\Omega) \;=\; \left\{ u \in L^2(\Omega) \,:\, \frac{\partial u}{\partial x_i} \in L^2(\Omega) \right\} \quad \text{and}$$
$$H_D^1(\Omega) \;=\; \left\{ u \in H^1(\Omega) \text{ and } u = 0 \text{ on } \Gamma \right\},$$

where

$$L^2(\Omega) \;=\; \left\{ u : \Omega \to \mathbb{R} \,:\, \int_\Omega u^2 < \infty \right\}.$$

Now we can write the *weak form* of (2.4) as

Find $u \in H_D^1(\Omega)$ such that

$$\int_\Omega \mu \nabla u \cdot \nabla v \;=\; \int_\Omega f v \text{ for all } v \in H_D^1(\Omega), \tag{2.7}$$

where $f \in L^2(\Omega)$.

The formulation carries that name because it imposes weaker requirements on solution and right-hand side than the original PDE. Summarizing, the PDE has been relaxed in two ways:

- u need not be twice differentiable in Ω, but only once weakly differentiable in Ω.
- f need not be continuous over $\overline{\Omega}$. It is sufficient that f is square-integrable over Ω and fv is integrable over Ω.

For convenience, we will also use a more concise notation of the weak form,

$$a(u, v) \;=\; \ell(v), \tag{2.8}$$

with the symmetric bilinear form

$$a(u, v) \;=\; \int_\Omega \mu \nabla u \cdot \nabla v$$

and the linear functional

$$\ell(v) \;=\; \int_\Omega f v.$$

2.2.5 Other boundary conditions

The derivation of the weak form up to here assumed that the model problem has only homogeneous Dirichlet boundary conditions. We now analyze which changes have to be made in order to also support other boundary conditions.

Inhomogeneous Dirichlet boundary conditions

The model problem with inhomogeneous Dirichlet boundary conditions is

$$\begin{aligned} -\mu\Delta u &= f \text{ in } \Omega \\ u &= g \text{ on } \Gamma. \end{aligned} \tag{2.9}$$

In the previous section, homogeneous Dirichlet boundary conditions were imposed by choosing the function space $H^1_D(\Omega)$. The approach to define a similar function space that includes the inhomogeneous Dirichlet boundary conditions would not be promising. Therefore, g has to be integrated into the weak form in a different way. In order to reuse the space $H^1_D(\Omega)$, u has to be replaced with a function that is 0 on Γ. This is achieved by using the functions

$$\begin{aligned} &G \in H^1(\Omega) \text{ with } G = g \text{ on } \Gamma \text{ and} \\ &w = u - G \end{aligned}$$

to derive the weak form in the following steps:

$$\begin{aligned} &-\int_\Omega u\Delta(w+G)v = \int_\Omega fv \\ \Rightarrow &\int_\Omega \mu\nabla v\cdot\nabla(w+G) - \int_\Gamma \mu v\frac{\partial(w+G)}{\partial n} = \int_\Omega fv \\ \Rightarrow &\int_\Omega \mu\nabla v\cdot\nabla w + \int_\Omega \mu\nabla v\cdot\nabla G = \int_\Omega fv. \end{aligned}$$

Thus, the weak form of (2.9) is:

$$\begin{aligned} &\text{Find } u = w + G, w \in H^1_D(\Omega) \text{ such that} \\ &\int_\Omega \mu\nabla w\cdot\nabla v = \int_\Omega fv - \int_\Omega \mu\nabla v\cdot\nabla G \text{ for all } v \in H^1_D(\Omega), \\ &\text{where } f \in L^2(\Omega), G \in H^1(\Omega), \text{ and } G = g \text{ on } \Gamma. \end{aligned} \tag{2.10}$$

Neumann boundary conditions

Homogeneous Neumann boundary conditions impose that there be no flux for u across the boundary. The Neumann problem is thus formulated as

$$\begin{aligned} -\mu\Delta u &= f \text{ in } \Omega \\ \frac{\partial u}{\partial n} &= 0 \text{ on } \Gamma. \end{aligned} \tag{2.11}$$

For deriving a weak form that includes these boundary conditions, we apply the same transformations as in the previous section. Since the solution is not necessarily

zero on the boundary, any more, we use $H^1(\Omega)$ instead of $H^1_D(\Omega)$ for u and v.

$$\begin{aligned} -\int_\Omega \mu \Delta u v &= \int_\Omega f v \text{ for all } v \in H^1(\Omega) \\ \Rightarrow \int_\Omega \mu \nabla v \cdot \nabla u &= \int_\Omega f v + \int_\Gamma v \frac{\partial u}{\partial n} \text{ for all } v \in H^1(\Omega) \\ \Rightarrow \int_\Omega \mu \nabla v \cdot \nabla u &= \int_\Omega f v \text{ for all } v \in H^1(\Omega) \end{aligned} \tag{2.12}$$

While in the previous section the boundary integral vanished because $v = 0$ on Γ, it now vanishes due to the homogeneous Neumann boundary conditions.

While it was straightforward to derive the weak from the strong form, proofing the converse argument ((2.12) $\Rightarrow$ (2.11)) requires additional thought, because the function space changed (see [22], page 30). In fact, for the argument to hold, additional smoothness beyond $H^1(\Omega)$ is required for u. The solution must be smooth enough that

- Green's identity applies, and
- ∇u can be restricted to Γ, so $\partial u / \partial n$ can be computed.

While the Dirichlet boundary conditions must be imposed explicitly (by selecting the function space $H^1_D(\Omega)$), the homogeneous Neumann boundary conditions are automatically fulfilled in the finite element method. Due to this property they are also known as *natural* boundary conditions.

Inhomogeneous Neumann boundary conditions are described by

$$\frac{\partial u}{\partial n} = h \text{ on } \Gamma^N.$$

For these boundary conditions, deriving the weak form is straightforward. The only change is that the boundary integral does not vanish, any more:

$$\begin{aligned} -\int_\Omega \mu \Delta u v &= \int_\Omega f v \\ \Rightarrow \int_\Omega \mu \nabla v \cdot \nabla u &= \int_\Omega f v + \int_\Gamma v \frac{\partial u}{\partial n} \\ \Rightarrow \int_\Omega \mu \nabla v \cdot \nabla u &= \int_\Omega f v + \int_{\Gamma^N} h v. \end{aligned} \tag{2.13}$$

2.2.6 The Galerkin method with polynomial basis functions

Recall that our main goal of deriving the weak form (2.7) was not to relax the requirements on the involved functions, but to open the gate towards a new solution method that employs our ability to efficiently compute integrals over non-trivial domains. The *Galerkin method* closes the gap from the weak form to the domain partitioning technique described in Section 2.2.3 by using a clever choice of local basis functions.

The boundary value problem (2.7) cannot be tackled easily and efficiently with computer programs, because the space $H^1(\Omega)$ is infinite-dimensional. Therefore, the Galerkin method computes u^h, an approximation to u in a finite-dimensional sub-space $V^h \subset H^1(\Omega)$. The *projection theorem* claims that the computed approximation is the *best* approximation to u in V^h. For a proof, see [22], page 57.

Since V^h is finite-dimensional, it is spanned by a finite number of basis vectors $w_1, \ldots, w_N$. As any other vector in V^h, the approximation to u can be written as a linear combination of the basis vectors:

$$u^h = \sum_{i=1}^{N} u_i^h w_i \text{ with } u_i^h \in \mathbb{R}. \tag{2.14}$$

Let u^h be an approximation to u in V^h. Then we want to compute the solution of the discrete weak form

$$a(u^h, v) = \ell(v) \text{ for all } v \in V^h.$$

If the discrete form holds for all basis vectors w_i of V^h, then it also holds for all other vectors $v \in V^h$, because $a(\cdot,\cdot)$ and $\ell(\cdot)$ are bilinear and linear, respectively. Therefore, we only need to solve the system of N equations

$$a(u^h, w_i) = \ell(w_i), \quad i = 1, \ldots, N. \tag{2.15}$$

Inserting (2.14) into (2.15) and reordering the terms yields

$$\begin{aligned} &\int_\Omega \mu \nabla \sum_{j=1}^{N} (u_j^h w_j) \cdot \nabla w_i = \int_\Omega f w_i, \quad i = 1, \ldots, N \\ \Leftrightarrow &\sum_{j=1}^{N} \left(u_j^h \int_\Omega \mu \nabla w_j \cdot \nabla w_i \right) = \int_\Omega f w_i, \quad i = 1, \ldots, N. \end{aligned}$$

This is an N-dimensional linear system of equations

$$Au^h = F \tag{2.16}$$

with the *stiffness matrix*

$$A = \{a_{ij}\}_{i,j=1}^{N}, \text{ where } a_{ij} = \int_\Omega \mu \nabla w_j \cdot \nabla w_i,$$

the vector of unknowns

$$u^h = \{u_i^h\}_{i=1}^{N},$$

and the *load vector*

$$F = \{f_i\}_{i=1}^{N}, \text{ where } f_i = \int_\Omega f w_i.$$

The names "stiffness matrix" and "load vector" originate from the domain of mechanical engineering. In the PDE that models the deformation of an object under a certain load the parameter μ is characteristic for the material's stiffness and the right-hand side f describes the external forces that impose the load on the object.

Polynomial basis functions

The Galerkin method itself does not define which subspace V^h and which basis functions w_i are to be used. The final ingredient for the *finite element method* is

a subspace of basis functions with *compact support*. A function is compactly supported, if it is zero everywhere but in a closed, bounded subdomain of Ω. The advantage of such basis functions in the linear system of equations (2.16) becomes clear when looking at the entries of the stiffness matrix,

$$a_{ij} = \int_\Omega \mu \nabla w_j \cdot \nabla w_i \,,$$

Since $\nabla w_i = 0$ in most parts where $w_i = 0$, i. e., outside the compact subdomain, the inner product $\nabla w_j \cdot \nabla w_i$ vanishes for most combinations of i and j, and the corresponding matrix entries a_{ij} are zero. Thus, the matrix A is sparse, and the linear system can be solved efficiently using iterative methods, e. g., multigrid.

Even when restricting the selection of basis functions to the compactly supported ones, there are still a lot of choices. Polynomials, trigonometric functions, exponential functions, etc.; there is not much that has not been tried. Within the scope of this work we restrict ourselves to the class of *polynomial* functions, because they are well-suited for approximating the smooth functions of our model problems.

A polynomial in the two-dimensional vector space $\mathbb{R}^2$ is a function

$$f : (x, y) \in \mathbb{R}^2 \rightarrow a_{00} + a_{10}x + a_{01}y + a_{11}xy + a_{20}x^2 + \ldots + a_{nn}x^n y^n$$

with constant parameters $a_{ij} \in \mathbb{R}$. For a polynomial in $\mathbb{R}^3$ the scheme is expanded analogously by a third coordinate z.

For deciding which basis functions shall construct the space V^h it is usually advantageous to consider the characteristics of the problem that has to be solved, which means going from the abstract notion of function spaces that the Galerkin method is based on back to the real domain of the problem. For a physics simulation in 3D it makes sense to construct V^h such that it suits the characteristics of the domain of interest, or, in other words, such that the error caused by the approximation is as small as possible. A natural choice for the approximation is a *finite element mesh* as described in Section 2.2.3.

A finite element mesh is a partitioning of Ω into a set of open, bounded, polygonal (in 2D) or polyhedral (in 3D) sub-domains E_κ, $\kappa = 1, \ldots, N_E$, where N_E is the total number of finite elements in Ω. The term *finite element* refers to $\overline{E_\kappa}$, i. e., the subdomain *including* its closure. An element's closure is defined by its corners, called *nodes*, which will be denoted by $n_i^{(\kappa)}$, $i = 1, \ldots, N^{(\kappa)}$. The number of nodes $N^{(\kappa)}$ of element E_κ is specific to the its type. For example, for triangles $N^{(\kappa)} = 3$, for quadrilaterals and tetrahedrons $N^{(\kappa)} = 4$.

To simplify the following computations, we only consider *conforming* meshes for now. In Chapter 4 we will also encounter non-conforming meshes as a result of adaptive mesh refinement. Fig. 2.3 illustrates the following definition of a conforming mesh.

Definition 1. *Let the term* edge *denote the line e between two nodes $n_i^{(\kappa)}$ and $n_j^{(\kappa)}$, $i \neq j$, of element E_κ. A finite element mesh is* conforming, *if and only if for all elements*

$$E_\lambda \text{ with } e \cap \overline{E_\lambda} \neq \emptyset \,, \quad \lambda = 1, \ldots, N_E$$

the line e is an edge between two nodes $n_i^{(\lambda)}$ and $n_j^{(\lambda)}$, $i \neq j$.

The overall number of nodes in the mesh will be denoted by N, and the nodes, from a global point of view, will be denoted by n_i, $i = 1, \ldots, N$. Besides the number of

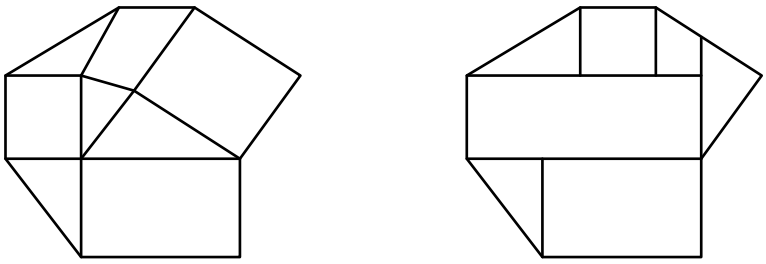

Figure 2.3: A conforming (left) and a non-conforming (right) finite element mesh.

elements in the mesh also the layout of the elements influences the overall number of mesh nodes. Thus, N is generally not uniquely determined by the $N^{(\kappa)}$. However, one can construct a unique mapping from local to global nodes

$$m : (\kappa, i) \to j : n_j = n_i^{(\kappa)} \quad \begin{array}{l} \text{for all } \kappa = 1, \ldots, N_E,\ i = 1, \ldots, N^{(\kappa)}, \\ \text{such that } j \in \{1, \ldots, N\}. \end{array} \tag{2.17}$$

The mapping is generally not reversible, because a node is usually within the closure of several elements.

It is not a coincidence that the variable name N used for the number of nodes here was already used earlier in this section for the number of basis vectors in V^h. We now associate a basis vector with each of the nodes. Each basis vector w_i is given by a corresponding basis function ϕ_i that shall have the following properties.

- ϕ_i is *piecewise polynomial*. The restriction of ϕ_i to any element E_κ is polynomial.
- ϕ_i is 1 on its corresponding node and 0 on all other nodes:

$$\phi_i(n_j) = \delta_{ij} = \begin{cases} 1, \text{ if } i = j \\ 0, \text{ if } i \neq j \end{cases}, \quad i, j = 1, \ldots, N.$$

- ϕ_i is linear along every edge.

The restriction of a basis function to an element is called *local basis function* and is defined as

$$\phi_i^{(\kappa)}(x) = \phi_{m(\kappa,i)}(x) \quad \text{for } x \in E_\kappa \text{ and } i = 1, \ldots, N^{(\kappa)},$$

where m is the mapping defined in (2.17).

2.2.7 Assembling the linear system

For assembling the linear system (2.16) the entries of the stiffness matrix and the load vector,

$$a_{ij} = \int_\Omega \mu \nabla\phi_i \cdot \nabla\phi_j \quad \text{and} \quad f_i = \int_\Omega f\phi_i \quad \text{for } i, j = 1, \ldots, N,$$

have to be computed. Since we have a finite element mesh now, we can apply (2.3) to compute the integrals as the sum of integrals over individual elements:

$$a_{ij} = \sum_{\kappa=1}^{N_E} \int_{E_\kappa} \mu \nabla\phi_i \cdot \nabla\phi_j \quad \text{and} \quad f_i = \sum_{\kappa=1}^{N_E} \int_{E_\kappa} f\phi_i \quad \text{for } i, j = 1, \ldots, N.$$

At this point some of the properties required above for the basis functions come in handy. Since $\phi_i(n_j) = \delta_{ij}$, both ϕ_i and $\nabla\phi_i$ are non-zero only in elements which have the node n_i in their closure. Consequently, the integrals for a_{ij} and f_i evaluate to zero in all other elements. Thus, the global basis functions can be substituted with the elements' local basis functions in the integrals:

$$a_{ij} = \sum_{\kappa=1}^{N_E} \int_{E_\kappa} \mu \nabla\phi_i \cdot \nabla\phi_j = \sum_{\kappa=1}^{N_E} \int_{E_\kappa} \mu \nabla\phi_k^{(\kappa)} \cdot \nabla\phi_l^{(\kappa)} \tag{2.18}$$

$$\text{and} \quad f_i = \sum_{\kappa=1}^{N_E} \int_{E_\kappa} f\phi_i = \sum_{\kappa=1}^{N_E} \int_{E_\kappa} f\phi_k^{(\kappa)} \tag{2.19}$$

$$\text{for} \quad i,j = 1,\ldots,N,\ k,l = 1,\ldots,N^{(\kappa)}.$$

Reference elements

Computing these integrals for every element would still be too expensive in practice. The number of times some of the functions have to be evaluated can be diminished by employing the concept of *reference elements*. For every element type we can define a reference element that can be mapped to all other elements of this type.

Fig. 2.4 shows the mapping between a reference triangle T_R and a triangle E_κ in the finite element mesh. T_R and E_κ are defined by the nodes t_i and $z_i^{(\kappa)}$, respectively, where

$$\begin{aligned} t_1 &= (0,0), \quad z_1^{(\kappa)} = (x_1^{(\kappa)}, y_1^{(\kappa)}), \\ t_2 &= (1,0), \quad z_2^{(\kappa)} = (x_2^{(\kappa)}, y_2^{(\kappa)}), \\ t_3 &= (0,1), \quad z_3^{(\kappa)} = (x_3^{(\kappa)}, y_3^{(\kappa)}). \end{aligned}$$

The term $\tilde{E}_\kappa$ will generally denote the reference element of E_κ, although many of the $\tilde{E}_\kappa$ will refer to the same reference element. For all triangular elements E_κ, for example, $\tilde{E}_\kappa = T_R$. Points in $\overline{\tilde{E}_\kappa}$ are mapped uniquely to points in $\overline{E_\kappa}$ and vice versa by the affine functions $\tau^{(\kappa)}$ and $\chi^{(\kappa)}$:

$$\begin{aligned} z \in E_\kappa = \tau^{(\kappa)}(t) &= z_1^{(\kappa)} + J_\kappa t, && t \in \tilde{E}_\kappa. \\ t \in \tilde{E}_\kappa = \chi^{(\kappa)}(z) &= J_\kappa^{-1}\left(z - z_1^{(\kappa)}\right), && z \in E_\kappa. \end{aligned}$$

The matrix j_κ is the *Jacobian matrix* of $\tau^{(\kappa)}$. For triangles it evaluates to

$$J_\kappa = \begin{pmatrix} x_2^{(\kappa)} - x_1^{(\kappa)} & x_3^{(\kappa)} - x_1^{(\kappa)} \\ y_2^{(\kappa)} - y_1^{(\kappa)} & y_3^{(\kappa)} - y_1^{(\kappa)} \end{pmatrix}.$$

The concept of reference elements has been shown for the simple case of the triangle. However, it is applicable to all element types for which transformations τ and χ can be constructed. The general form of the Jacobian matrix of a transformation in d dimensions is

$$J_\kappa = \begin{pmatrix} \frac{\partial\tau_1^{(\kappa)}(t)}{\partial r_1} & \cdots & \frac{\partial\tau_1^{(\kappa)}(t)}{\partial r_d} \\ \vdots & \ddots & \\ \frac{\partial\tau_d^{(\kappa)}(t)}{\partial r_1} & & \frac{\partial\tau_d^{(\kappa)}(t)}{\partial r_d} \end{pmatrix}, \tag{2.20}$$

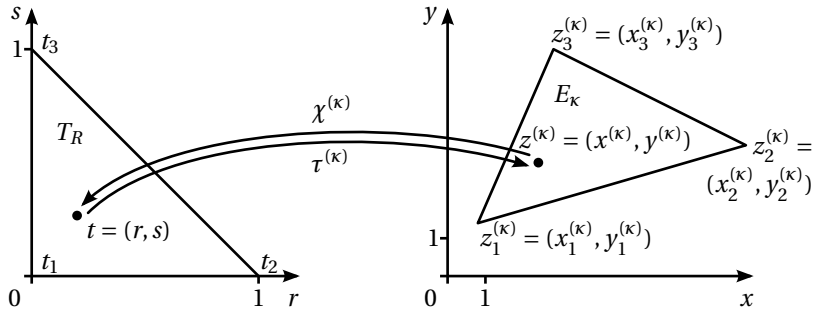

Figure 2.4: Reference triangle and coordinate mapping.

where the r_i are the components of the d-dimensional vector $t = (r_1, \ldots, r_d)^T$.

We are especially interested in element types for which *affine* transformations exist. These have a Jacobian matrix that is constant in t, which allows for the efficient computation of the integrals (2.18) and (2.19) on the reference elements, as we will see below. Besides the triangle, also the other elements used in HHG—the quadrilateral, the tetrahedron and the cuboid—have this property. General quadrilaterals and general hexahedra can not be mapped to the reference square or the reference cube, respectively, by affine transformations. This is one reason why these element types are not supported in HHG.

For every basis function $\phi_i^{(\kappa)}$ the corresponding basis function on the reference element is $\tilde{\phi}_i^{(\kappa)}$ with

$$\begin{aligned} \tilde{\phi}_i^{(\kappa)}(t) &= \phi_i^{(\kappa)}(z) \\ \Leftrightarrow \tilde{\phi}_i^{(\kappa)}(t) &= \phi_i^{(\kappa)}(\tau^{(\kappa)}(t)) \\ \Leftrightarrow \phi_i^{(\kappa)}(z) &= \tilde{\phi}_i^{(\kappa)}(\chi^{(\kappa)}(z)) \end{aligned} \tag{2.21}$$

Note that for two different elements of the same type, E_κ and E_λ, which naturally have different basis functions $\phi_i^{(\kappa)} \neq \phi_i^{(\lambda)}$, the basis functions on the reference element are the same: $\tilde{\phi}_i^{(\kappa)} = \tilde{\phi}_i^{(\lambda)}$. Thus, if we replace $\phi_i^{(\kappa)}$ in (2.18) and (2.19) with $\tilde{\phi}_i^{(\kappa)}$, we will need to perform some expensive computations only once for each element type.

The load vector

The rule for the change of variables in an integral is given by the transformation theorem,

$$\int_{\tilde{\Omega}} a(z)\,dz = \int_{\Omega} |\det(J(b))|\, a(b(t))\,dt, \tag{2.22}$$

where $J(b)$ is the Jacobian matrix of the transformation b, and $|\det(J(b))|$ is the absolute value of b's determinant. For a proof of the transformation theorem, see [2, p. 416].

Equation (2.19) can easily be transferred into a sum of integrals over the refer-

ence element with (2.22), where $a = \phi_\kappa$ and $b = \tau^{(\kappa)}$, and using (2.21):

$$\begin{aligned} f_i &= \sum_{\kappa=1}^{N_E} |\det(J_\kappa)| \int_{\tilde{E}_\kappa} \tilde{f}\tilde{\phi}_k^{(\kappa)}, \quad \text{where} \quad \tilde{f} = f(\tau^{(\kappa)}), \\ &\text{for} \quad i = 1,\ldots,N,\ k = 1,\ldots,N^{(\kappa)}. \end{aligned} \tag{2.23}$$

Note that $|\det(J_\kappa)|$ can be taken out of the integral, because it is constant over $\tilde{E}_\kappa$.

If the problem contains inhomogeneous Neumann boundary conditions, the term

$$\int_{\Gamma^N} hv$$

in (2.13) is not zero and needs to be added to the load vector. It is an integral over a $d-1$-dimensional sub-space of Ω. The steps for computing it are, in principle, the same as for (2.23). By applying (2.3) the integral over Γ^N can be split into a sum of integrals over the boundaries of the individual elements E_κ. Like the elements, also their boundaries can be grouped topologically into primitive types. In 3D, for example, the boundaries of tetrahedra are triangular faces, the boundaries of hexahedra are quadrilateral faces. Like for the elements, reference faces can be defined. Note that the transformations $\tau^{(\kappa)}$ and $\chi^{(\kappa)}$ now map between spaces of different dimension: the d-dimensional space Ω and the $d-1$-dimensional space of the reference faces.

The stiffness matrix

Changing the variables in (2.18) is not quite as simple, because the functions to be transformed are inside the gradient function. The individual components of the gradient vector are

$$\nabla\phi_k^{(\kappa)}(z) = \left(\frac{\partial}{\partial x_1}\phi_k^{(\kappa)}(z),\ldots,\frac{\partial}{\partial x_d}\phi_k^{(\kappa)}(z)\right)^T,$$

where $z = (x_1,\ldots,x_d)^T$. In order to use $\phi_k^{(\kappa)}(z) = \tilde{\phi}_k^{(\kappa)}(\chi^{(\kappa)}(z))$ from (2.21), the chain rule for more-dimensional functions [2, p. 353] has to be applied to each component of the gradient:

$$\frac{\partial}{\partial x_i}\phi_k^{(\kappa)}(z) = \frac{\partial}{\partial x_i}\left[\tilde{\phi}_k^{(\kappa)}\left(\chi^{(\kappa)}(z)\right)\right] = \sum_{j=1}^{d} \frac{\partial\tilde{\phi}_k^{(\kappa)}(t)}{\partial r_j}\cdot\frac{\partial\chi_j^{(\kappa)}(z)}{\partial x_i}.$$

Analogous to the Jacobian matrix J_κ for $\tau^{(\kappa)}$ in (2.20), the Jacobian matrix for the inverse mapping $\chi^{(\kappa)}$ is defined as

$$K_\kappa = \begin{pmatrix} \frac{\partial\chi_1^{(\kappa)}(t)}{\partial x_1} & \cdots & \frac{\partial\chi_1^{(\kappa)}(t)}{\partial x_d} \\ \vdots & \ddots & \\ \frac{\partial\chi_d^{(\kappa)}(t)}{\partial x_1} & & \frac{\partial\chi_d^{(\kappa)}(t)}{\partial x_d} \end{pmatrix}.$$

Note that the terms $\partial\chi_j^{(\kappa)}(z)/\partial x_i$ that appeared after applying the chain rule form the i-th column of K_κ. Thus, the change of variables is achieved with

$$\nabla\phi_k^{(\kappa)}(z) = K_\kappa^T\nabla\tilde{\phi}_k^{(\kappa)}(t),$$

and (2.18) can be transformed into a form using only integrals over the reference elements:

$$a_{ij} = \sum_{\kappa=1}^{N_E} \int_{E_\kappa} \mu \left(K_\kappa^T \nabla \tilde{\phi}_k^{(\kappa)} \right) \cdot \left(K_\kappa^T \nabla \tilde{\phi}_l^{(\kappa)} \right) \quad \text{for} \quad i,j = 1,\ldots,N,\ k,l = 1,\ldots,N^{(\kappa)}. \tag{2.24}$$

A slightly different approach for changing the variables in (2.18) is to start with $\tilde{\phi}_k^{(\kappa)}(t) = \phi_k^{(\kappa)}(\tau^{(\kappa)}(t))$ from (2.21). Transformations analogous to the ones above lead to

$$\nabla \phi_k^{(\kappa)}(z) = J_\kappa^{-T} \nabla \tilde{\phi}_k^{(\kappa)}(t),$$

which shows that $J_\kappa^{-T} = K_\kappa^T$. The practical relevance of this finding is that for implementing the finite element method one can choose between inverting the Jacobian matrix of τ or computing the reverse mapping χ, depending on what is more convenient.

Efficiency considerations

The integrals for stiffness matrix and right-hand side can now be computed efficiently by numerical integration. Still, the challenge of implementing an efficient finite element solver has only just begun.

Usually, most of the entries a_{ij} of the stiffness matrix will be zero, because the corresponding nodes n_i and n_j are not are not in the closure of the same element and, thus, the corresponding basis functions will be zero at these nodes. Therefore, simply iterating over all combinations of i and j is not an efficient approach to compute the stiffness matrix. Instead, one rather iterates over the nodes or elements of the mesh and computes their contributions to the stiffness matrix.

Iterating over the nodes usually causes more work than iterating over the elements. The basis function of a node also contributes to the stiffness matrix entries of neighboring nodes—*neighboring* meaning here, that the nodes are in the closure of the same element. This means, when iterating over the nodes, the gradients of the basis functions have to be computed repeatedly—or stored, which is quite inconvenient. Therefore it is usually better to iterate over the elements and add each element's contributions to the corresponding stiffness matrix entries of all nodes in the element's closure.

Stencil-based solvers, the group of solvers HHG belongs to, do not even build up the stiffness matrix, at all. Instead, they store the element stiffness matrices separately and apply them individually to a vector, with the same result as if they would have been first assembled to a global matrix. Especially if the mesh contains many elements with the same shape and, thus, with the same stiffness matrix, this technique is very efficient, as we will see in Chapter 3.

2.3 Multigrid methods

2.3.1 Introduction

The finite element method introduced in the previous section generates a linear system of equations

$$Au = f \tag{2.25}$$

with n unknowns, i. e., with an $n \times n$ matrix A and n-dimensional vectors u and f. Depending on the geometry that has to be approximated with the finite element mesh and the requirements on the quality of the solution, n can get very large; on the other hand, A is usually very sparse. Various methods have been developed to solve such systems of equations.

Two distinct groups of solvers exist: direct and iterative solvers. Direct methods produce a solution that is exact up to the discretization accuracy after one pass of the algorithm. A well-known representative of this group is Gaussian elimination [45, p. 44]. Because of their robustness direct methods are popular black-box solvers in numerical software packages. Iterative methods produce an approximation that converges towards the exact solution by repeatedly applying an algorithm to the approximation of the previous iteration. Popular representatives of this group are the conjugate gradient method and the Gauss-Seidel method. The multigrid methods presented in this thesis belong to this group, too. Iterative methods have the advantage of requiring, in general, less computational effort and less storage space than the direct methods. Therefore, they are especially suited for large systems of equations. For a comparison of direct and iterative solvers, see [3], Section 7.5.

Local iterative solvers

The iterative methods can be categorized by the way information propagates in the mesh. The *Krylov subspace* schemes, like the conjugate gradient method, are global processes. That means, in each iteration the solution at one node of the mesh can be influenced by information from all other nodes. The *splitting* schemes, among them the Gauss-Seidel method, are local processes. In these schemes the solution at a certain node is only influenced by information from neighboring nodes in each iteration.

The splitting schemes have important advantages, which make them interesting for large sparse systems of equations. For example, they are well suited for parallelization—for the very reason that only local exchange of information and, thus, no global communication is necessary. However, that property also gives the splitting schemes a decisive disadvantage, which is illustrated in Fig. 2.5. The figure shows how the Jacobi algorithm (which will be presented in the next section) typically reduces the error of the approximation after some iterations. For generating the plots, the model problem (2.1) in 1D on the domain $\Omega = [0, 1]$ was discretized with finite differences at a resolution of $1/8$, i. e., with 9 grid points. Homogeneous Dirichlet boundary conditions ($u(0) = 0$ and $u(1) = 0$) and a right-hand side $f = 0$ were used. Since the solution of this problem is $u(x) = 0$, the error after each iteration is equal to the current approximation of the solution. The Jacobi iteration was initialized with random values for u—see the left plot in Fig. 2.5. The middle plot shows the error after 5 iterations, the right plot after 10 iterations. Obviously, the error is *smoothed* very well within a few iterations, but after that it takes many more iterations to bring it close to zero. This is not surprising given that the Jacobi algorithm just computes weighted averages of nearest neighbors.

Multigrid

At this point the multigrid idea comes into play. The notion of what is a “nearest neighbor” is entirely mesh-dependent, so, one can as well discretize the problem at a coarser resolution (i. e., with less mesh points) and continue eliminating the error

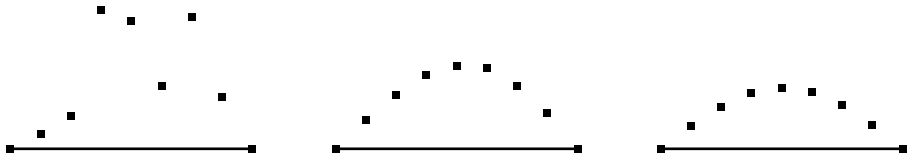

Figure 2.5: Jacobi error reduction. Left: initial error. Middle and right: error after 5 and 10 iterations, respectively.

on that mesh—since the error is smooth already, not much information is lost when observing it at a coarser resolution.

2.3.2 Multigrid building blocks

Two multigrid ingredients, the smoother and the coarse mesh, were already mentioned in the previous section. This section describes them in more detail and introduces the other parts that are necessary for building a complete multigrid algorithm.

Smoothers

The algorithms presented in the following were originally developed for *solving* linear systems of equations, but they often require hundreds of iterations to reduce the error to an acceptable level. In the multigrid context, their main purpose is to *smooth the error*—a task for which they are much better suited, as Fig. 2.5 already indicated.

Algorithm 1 describes the smoother in general. The smoother operates on a linear system determined by the matrix $A \in \mathbb{R}^{N\times N}$ and the right-hand side $f \in \mathbb{R}^N$. The current approximation to the solution u of the system

$$Au = f$$

is denoted with v. The number of iterations the smoother shall perform is given with the configuration parameter ν.

Algorithm 1: The general form of the smoother.

1.1 **Algorithm** `smooth`(A, v, f, ν)
1.2 Perform ν iterations of an iterative solver with good error smoothing properties on the linear system $Av = f$.
1.3 **return** v

The *Jacobi* algorithm is a simple splitting scheme that can be used as a smoother. Given an initial approximation v to the solution, it computes an updated approximation $\overline{v}$ by solving the system's equations independently:

$$\begin{aligned} & Av \approx f \\ \Leftrightarrow & \sum_{j=1}^{N} A_{ij} v_j \approx f_i \text{ for } i = 1 \ldots N \\ \Rightarrow & \overline{v}_i = A_{ii}^{-1} \left(f_i - \sum_{j=1}^{i-1} A_{ij} v_j - \sum_{j=i+1}^{N} A_{ij} v_j \right) \text{ for } i = 1 \ldots N. \end{aligned}$$

The scheme can be written in matrix form, which also explains the term *splitting scheme*. Assume A is split into three matrices

- $D \in \mathbb{R}^{N\times N}$ containing the diagonal entries of A,
- $L \in \mathbb{R}^{N\times N}$ containing the entries below the diagonal of A, and
- $U \in \mathbb{R}^{N\times N}$ containing the entries above the diagonal of A.

Then $A = L + D + U$ and

$$\overline{v} = D^{-1}\left(f - (L+U)v\right). \tag{2.26}$$

Using Banach's fix-point theorem it can be proven that this iteration converges towards the solution of $Au = f$ [40]. The proof is based on the fact that u is a fix point of (2.26), i. e., inserting u for v in the equation yields u as a result. The convergence is determined by the condition of the iteration matrix: if $||D^{-1}(L+u)|| < 1$, then the fix-point iteration converges. This property is fulfilled for the model problem (2.1) as long as $\Gamma^D \neq \emptyset$. The proof can be read in detail for example in .

It turns out that the convergence of the Jacobi algorithm, but also its smoothing efficiency, can be greatly improved by applying damping to the iteration:

$$\overline{v} = (1-\omega)v \;+\; \omega D^{-1}\left(f - (L+U)v\right).$$

The damped Jacobi iteration (Algorithm 2) converges for damping values $0 < \omega \leq 1$. For $\omega = 1$ it is equivalent to the undamped Jacobi iteration.

Algorithm 2: The damped Jacobi smoother.

```
2.1 Algorithm jacobi(A, v, f, ν, ω)
2.2 for n = 1...ν do
2.3     for i = 1...N do
2.4         v̄_i = (1−ω) v_i + ω A_ii^{-1} ( f_i − Σ_{j=1}^{i−1} A_ij v_j − Σ_{j=i+1}^{N} A_ij v_j )
2.5     end
2.6 end
2.7 return v̄
```

An obvious change to the Jacobi algorithm leads to the *Gauss-Seidel* algorithm: instead of using only values of the old approximation for computing the new approximation, one can use new values as soon as they become available. The formula for the individual updates is

$$\overline{v}_i \;=\; A_{ii}^{-1}\left(f_i - \sum_{j=1}^{i-1} A_{ij}\overline{v}_j - \sum_{j=i+1}^{N} A_{ij}v_j\right) \text{ for } i = 1\ldots N,$$

the matrix form is

$$\begin{aligned} \overline{v} \;&=\; D^{-1}\left(f - L\overline{v} - Uv\right) \\ \Leftrightarrow\; \overline{v} \;&=\; (D+L)^{-1}f - (D+L)^{-1}Uv. \end{aligned} \tag{2.27}$$

For the Gauss-Seidel algorithm a "damping" parameter can be used, too. In this case, a value of $\omega > 1$ usually yields the best convergence. That is why the technique

is called *successive over-relaxation* (SOR). The matrix form of the SOR algorithm is

$$\begin{aligned} \overline{v} &= (1-\omega)v + \omega D^{-1}\left(f - L\overline{v} - Uv\right) \\ \Leftrightarrow \overline{v} &= \omega(D+\omega L)^{-1} f + (D+\omega L)^{-1}\left[(1-\omega)D - \omega U\right] v\,. \end{aligned}$$

The iteration is convergent for $0 < \omega < 2$. For $\omega = 1$ it is equal to the Gauss-Seidel scheme, so Algorithm 3, although named `sor`, represents both schemes.

Algorithm 3: The successive over-relaxation smoother.

3.1 **Algorithm** `sor`(A, v, f, ν, ω)
3.2 **for** $n = 1 \ldots \nu$ **do**
3.3 **for** $i = 1 \ldots N$ **do**
3.4 $\overline{v}_i = (1-\omega)v_i + \omega A_{ii}^{-1}\left(f_i - \sum_{j=1}^{i-1} A_{ij}\overline{v}_j - \sum_{j=i+1}^{N} A_{ij}v_j\right)$
3.5 **end**
3.6 **end**
3.7 **return** $\overline{v}$

Although the Jacobi smoother and the Gauss-Seidel smoother (or their damped versions, respectively) are very similar at first glance, they are very different in practical application. Gauss-Seidel shows a significantly better convergence than Jacobi. It also needs less storage space on the computer: while Jacobi needs to store v and $\overline{v}$, Gauss-Seidel can update v in-place, because it uses the updated values as soon as they are available, anyway. Hence, so far, Gauss-Seidel would be preferable. Unfortunately, though, Gauss-Seidel can—in general—not be parallelized, because computing $\overline{v}_i$ requires $\overline{v}_j$ for all $j < i$. In today's computing environment this is such a severe disadvantage that it outweighs all the advantages.

In practice, the issue is usually not that severe, though, because A is sparse, and $\overline{v}_i$ does, therefore, not depend on all $\overline{v}_j$ with $j < i$. The sparsity pattern of A depends on the PDE and the discretization, so, giving a general statement about how well Gauss-Seidel can be parallelized is not possible. Fig. 2.6 shows a 2D grid with 5×5 nodes. Drawn at node 13 is a 5-point operator stencil, a typical result, e. g., from discretizing the Laplace equation with finite elements. The drawing reveals that for updating node 13 updated values are only required from nodes 8 and 12. Node 9, for example, could thus be updated at the same time. The drawing in the middle shows how to exploit this. Sweeping from the lower left to the upper right corner, the nodes on the diagonals can be updated in parallel: after node 1 is updated, nodes 2 and 6 can be updated in parallel, then nodes 3, 7, and 11, and so on. With that technique, instead of N sequential steps, only $\sqrt{N}$ sequential steps are necessary. Note that this technique does not adversely affect the convergence of the Gauss-Seidel algorithm, because for computing $\overline{v}_i$ the values of $\overline{v}_j$ are still available for all $j < i$.

The diagonal sweep is easily applicable only to structured grids, and its parallelism is still mediocre. A technique that is more generally applicable and provides better parallelism—however, at the expense of loosing optimal convergence—is the *red-black* Gauss-Seidel algorithm. It is illustrated in the right drawing of Fig. 2.6. The basic idea is that nodes are assigned "colors" such that nodes of the same color are not connected by the operator stencil. For the 5-point stencil in 2D two colors—thus the name "red-black"— are necessary. Fig. 2.6 shows the resulting node partitioning. Now, first all "black" nodes can be updated in parallel, and then all "red"

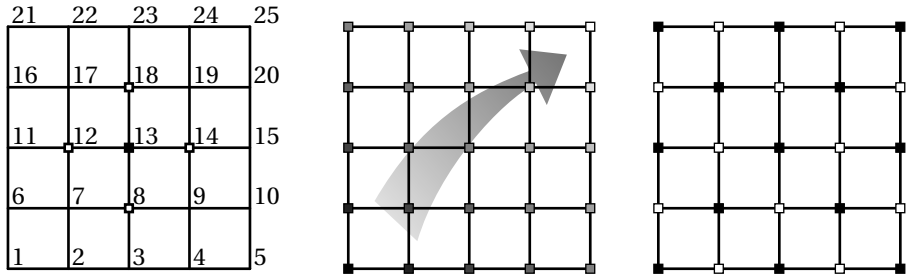

Figure 2.6: 5×5 grid with 5-point operator stencil (left). Two possible update orders: diagonal (middle) and red-black (right).

nodes. Thus, only two sequential steps are necessary, now, and $N/2$ nodes can be updated in parallel. However, not all values of $\overline{v}_j$ for $j < i$ are available, any more, when updating $\overline{v}_i$, which leads to a decreased convergence rate. Anyway, the trade-off between parallelism and convergence is good. Therefore, the algorithm is often used in practice, also in HHG. For the semi-structured tetrahedral meshes used in HHG four colors are necessary; see [19].

Mesh resolutions and levels

One characteristic of a mesh is its *resolution*, i. e., the number of nodes per unit distance, area, or volume. While for a structured, regular grid this measure can be quantified precisely, it may be determined only roughly for an unstructured mesh. In any case, for a given mesh, a mesh with a higher resolution can be constructed either by re-discretizing the domain with different parameters or by adding new nodes between the existing nodes. For constructing a mesh with a lower resolution, the analogous is true: either re-discretize the domain or remove some nodes from the existing mesh. The first approach, re-discretizing the domain, can be combined with the multigrid idea. However, the latter approach is much more suitable for multigrid, because it ensures that all the nodes of the coarse mesh are also present in the fine mesh—a property that is extremely useful when it comes to transferring information between the meshes.

Independent of how the meshes with different resolutions are created, the term *level* is used to address individual meshes and organize them in a hierarchy. A fine mesh with a high resolution is associated with a high level, a coarse mesh with a low level.

Fig. 2.7 shows a hierarchy of meshes in 1D. The mesh at level 0 has two boundary nodes (at $x = a$ and $x = b$) and one interior node. The distance between the nodes is h_0. The meshes at level 1 and 2 were created by adding additional interior nodes between the nodes of the mesh on the previous level. With increasing level, the resolution increases as the distance between the nodes decreases.

Inter-grid transfer operators

As described briefly in the introduction to this section, multigrid aims to correct the error of the fine-grid approximation on a coarser mesh. In order to transfer the error to the coarser mesh and the result of the correction back to the finer mesh, inter-grid transfer operators will be defined. Transfer of information from a fine to a

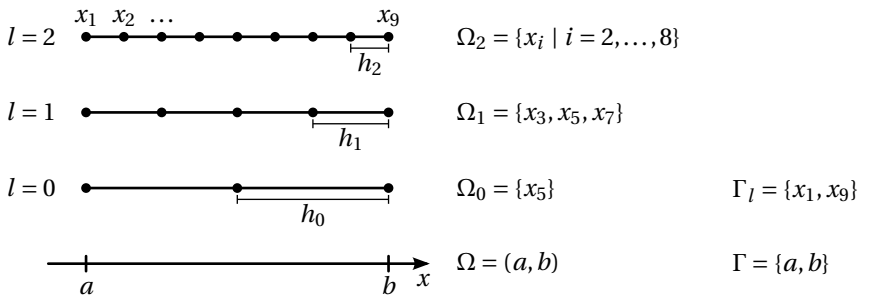

Figure 2.7: Hierarchy of regular grids in 1D.

coarse mesh will be called *restriction*, transfer in the opposite direction will be called *prolongation*.

Both operations can be expressed as matrix-vector multiplications. Assume v_l and v_{l-1} are vectors on the fine mesh and on the coarse mesh, respectively. The restriction operator shall be denoted with R_l^{l-1}; with this operator the vector on the coarse mesh is calculated as

$$v_{l-1} = R_l^{l-1} v_l .$$

If N_l and N_{l-1} are the numbers of nodes on the fine and the coarse mesh,, respectively, then $v_l \in \mathbb{R}^{N_l}$, $v_{l-1} \in \mathbb{R}^{N_{l-1}}$, and $R_l^{l-1} \in \mathbb{R}^{N_{l-1} \times N_l}$. The prolongation operator $P_{l-1}^l \in \mathbb{R}^{N_l \times N_{l-1}}$ fulfills

$$v_l = P_{l-1}^l v_{l-1} .$$

As described above, different ways of generating the mesh hierarchy require different transfer operators. For the sake of simplicity, and since it is also the case in HHG, we assume that all nodes of the mesh at level $l-1$ are also contained in the mesh at level l.

The prolongation operator that is most common in multigrid applications is *linear interpolation* (or bi-linear interpolation, etc., for higher dimensions). While the exact form of the operator depends on the mesh, the basic idea is to compute the values at the new nodes on the fine mesh as an average of the values at the surrounding nodes on the coarse mesh. For the 1D mesh shown in Fig. 2.7 the prolongation operator is

$$P_{l-1}^l = (p_{ij})$$
$$\text{where } p_{ij} = \begin{cases} 1, \text{ if } i-1 = 2(j-1) \\ 1/2, \text{ if } i = 2j \text{ or } i = 2(j-1) \\ 0, \text{ otherwise} \end{cases}$$
$$\text{for } i \in \{1, \ldots, N_l\} \text{ and } j \in \{1, \ldots, N_{l-1}\} .$$

For example, the prolongation by linear interpolation from level 0 to level 1 evaluates to

$$P_0^1 = \begin{pmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 0 \\ 0 & 1 & 0 \\ 0 & 1/2 & 1/2 \\ 0 & 0 & 1 \end{pmatrix} .$$

Two different restriction operators shall be introduced here. One is closely related to the linear interpolation operator. It is called *full-weighting*, because it calculates the value of a node on the coarse mesh as a weighted average of the values of the corresponding node and its neighboring nodes on the fine mesh:

$$R_l^{l-1} = (r_{ij})$$

$$\text{where } r_{ij} = \begin{cases} 1/2, \text{ if } i-1 = 1/2(j-1) \\ 1/4, \text{ if } i = 1/2\,j \text{ or } i-1 = 1/2\,j \\ 0, \text{ otherwise} \end{cases}$$

for the interior nodes, i. e., for $i \in \{2,\dots,N_{l-1}-1\}$ and $j \in \{2,\dots,N_l-1\}$,

$$\text{and where } r_{ij} = \begin{cases} 1, \text{ if } i-1 = 1/2(j-1) \\ 0, \text{ otherwise} \end{cases}$$

for the boundary nodes, i. e., for $i \in \{1, N_{l-1}\}$ or $j \in \{1, N_l\}$.

For example, the full-weighting restriction from level 1 to level 0 evaluates to

$$R_1^0 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1/4 & 1/2 & 1/4 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

A simpler restriction operator is *injection*. This operator maps the value of every entry of the vector on the fine mesh to the corresponding entry of the vector on the coarse mesh, i. e., to the vector entry that is associated with the same node. For our 1D mesh, the operator is defined as

$$\hat{R}_l^{l-1} = \left(r_{ij}\right)$$

$$\text{where } r_{ij} = \begin{cases} 1, \text{ if } j-1 = 2(i-1) \\ 0, \text{ otherwise} \end{cases}$$

for $i = 1,\dots,N_{l-1}$ and $j = 1,\dots,N_l$.

The injection operator from level 1 to level 0 evaluates to

$$\hat{R}_1^0 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

As the name suggests, this operator is an injective mapping, i. e., every value on the coarse mesh is determined by exactly one value on the fine mesh. The advantage of injection over full weighting is the lower computational complexity. Since the operator matrix contains fewer non-zero entries, fewer numerical operations have to be performed. In most cases, however, the full-weighting operator needs to be used, because its smoothing property is beneficial for the convergence of the multigrid algorithm.

The linear system on different levels

Given meshes on different levels, the partial differential operator A of (2.1) can be discretized on each level, which yields a series of discrete operators

$$A_l \in \mathbb{R}^{N_l \times N_l}.$$

Another way of constructing the discrete operators is *Galerkin coarsening*. After discretizing A on level l (yielding A_l), the operators on the coarser levels can be constructed using the prolongation and restriction operators:

$$A_{l-1} = R_l^{l-1} A_l P_{l-1}^l .$$

Galerkin coarsening is necessary in *algebraic* multigrid methods, which work on different levels without explicitly constructing a mesh on each level (see [48], Appendix A).

Within the scope of this thesis, direct construction of the operators on each level will be assumed.

The Two-grid cycle

The building blocks that have been introduced—smoothers, transfer operators, discrete operators on each level—will now be used to construct a solver working with meshes on two levels, l and $l-1$.

The goal is to compute the solution u_l of the linear system

$$A_l u_l = f_l .$$

on level l up to a negligible error.

The two-grid cycle is started with an initial approximation v_l to the solution u_l. The error of this approximation is defined as

$$e_l = u_l - v_l .$$

The first step of the two-grid cycle is to smooth the error by applying ν_1 iterations of a *smoother* to the linear system

$$A_l v_l = f_l ,$$

where ν_1 is a small number independent of N_l.

The result of the smoothing process is an improved approximation $\overline{v}_l$ that has a smooth error $\overline{e}_l$. Concerning smoothness, it is important at this point to distinguish between the solution and the error. The solution and its approximation are generally not smooth, which means that they can *not* be represented on a coarser mesh without loss of detail. Smoothness can only be assumed for the error. In that sense, Fig. 2.5 was deceiving, because there the solution was zero—i. e., very smooth— and its approximation was equal to the error and thus also being smoothed.

Since the error is smooth on level l, it could be restricted to level $l-1$ and eliminated there at a lower cost than on level l. Unfortunately, restricting $\overline{e}_l$ is not possible, because it is not known—if it were known, u_l could be computed immediately.

Fortunately, though, there is a computable quantity that has similar smoothness properties: the *residual*. It is defined as

$$\overline{r}_l = f_l - A_l \overline{v}_l$$

The residual of u_l is zero:

$$0 = f_l - A_l u_l .$$

Subtracting the above equations relates the residual to the error:

$$\begin{aligned} & \overline{r}_l \;=\; A_l\left(u_l - \overline{v}_l\right) \\ \Leftrightarrow\; & \overline{r}_l \;=\; A_l \overline{e}_l\,. \end{aligned}$$

Since A_l is an *elliptic* partial differential operator, the result of multiplying a smooth function with A_l is again a smooth function. Thus, $\overline{r}_l$ can be represented well on the coarse mesh. Thus, a good approximation to the error can be computed on level $l-1$ by solving

$$A_{l-1} v_{l-1} \;=\; f_{l-1}\,,$$

where

$$f_{l-1} \;=\; R_l^{l-1} r_l.$$

This step is called *coarse-grid correction*. In practice, the smoother used on level l is usually employed also for solving the linear system on level $l-1$. The initial guess for the solution can in this case be chosen as $v_{l-1}^* = 0$, because it is an approximation of the residual on level l, which is ideally zero. Since it will be necessary in Chapter 4 to clearly distinguish between the initial guess and the solution on level $l-1$, we give them different names already at this point. The initial guess is denoted with v_{l-1}^*, the solution with v_{l-1}.

The solution v_{l-1} can then be prolongated to level l and used there to correct $\overline{v}_l$:

$$\begin{aligned} & \tilde{e}_l \;=\; P_{l-1}^l v_{l-1} \\ \Rightarrow\; & \tilde{v}_l \;=\; \overline{v}_l + \tilde{e}_l \;=\; \overline{v}_l + P_{l-1}^l v_{l-1}\,. \end{aligned}$$

Therefore, $\tilde{e}_l$ is also called *correction*.

Since on level $l-1$ the linear system has fewer degrees of freedom than on level l, some functions on level l are identical to each other when restricted to level $l-1$—they *alias*. This poses a problem to the two-grid algorithm: the coarse-grid correction can not be prevented from introducing such components into the correction $\tilde{e}_l$. However, since only non-smooth functions can alias on the coarse mesh, those components can be eliminated from $\tilde{v}_l$ by applying ν_2 smoothing iterations after the coarse-grid correction.

With the final smoothing steps, the two-grid cycle is complete. It is summarized in Algorithm 4.

2.3.3 Types of multigrid cycles

The two-grid cycle (Algorithm 4) shows improved computational efficiency over using a simple iterative solver, because the correction is computed on the coarse mesh, which has a significantly smaller number of unknowns. This section addresses two improvements to the two-grid cycle:

- how to solve the linear system on level $l-1$, and
- how to generate a good initial guess on level l.

Algorithm 4: The two-grid cycle.

4.1 **Algorithm** `twogridcycle`$(A, v_l, f_l, R, P, l, \nu_1, \nu_2)$
4.2 $\overline{v}_l$ = `smooth`(A_l, v_l, f_l, ν_1)
4.3 $r_l = f_l - A_l \overline{v}_l$
4.4 $f_{l-1} = R_l^{l-1} r_l$
4.5 $v_{l-1}^* = 0$
4.6 v_{l-1} = `solve`$(A_{l-1} v_{l-1}^* = f_{l-1})$
4.7 $\tilde{v}_l = \overline{v}_l + P_{l-1}^l v_{l-1}$
4.8 v_l = `smooth`$(A_l, \tilde{v}_l, f_l, \nu_2)$
4.9 **return** v_l

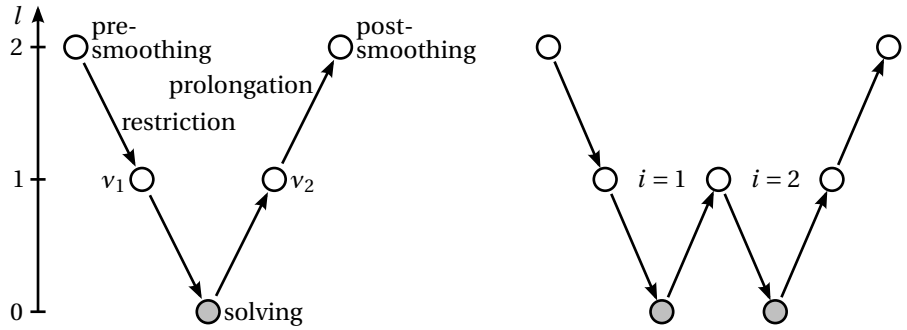

Figure 2.8: V-cycle (left) and γ-cycle with $\gamma = 2$ (right).

The V-cycle

In Fig. 2.7 the number of unknowns differs by a factor of two between two consecutive meshes. For d-dimensional structured grids that are coarsened in all spatial directions the factor is 2^d. This is a great reduction in the number of unknowns. For very large N_l, however, $N_l/2^d$ is still very large, and an iterative solver would sill have slow convergence. Therefore, it makes sense to apply the coarse-grid correction recursively, in the extreme case up to a trivial grid with only one unknown.

The resulting algorithm is called V-cycle, because it can be visualized as shown in Fig. 2.8. Algorithm 5 describes it formally. Compared to the two-grid cycle, it has one additional argument: l_s. This parameter defines the level on which the recursion is stopped and the linear system is solved. The choice of l_s depends on various factors, for example the properties of the mesh and the trade-off between computation and communication in parallel computing.

Note that, depending on how good the initial approximation v_l is, usually several V-cycles are necessary to reduce the error of the approximation to an acceptable level.

The γ-cycle

Since the V-cycle does not necessary compute an accurate enough approximation after one iteration, it can make sense to also apply the recursive V-cycle more than once. Algorithm 6 shows this generalization of the V-cycle, which is called γ-cycle. The parameter γ defines the number of times the cycle is applied recursively. Fig. 2.8

Algorithm 5: The V-cycle.

5.1 **Algorithm** `vcycle`$(A, v_l, f_l, R, P, l, l_s, \nu_1, \nu_2)$
5.2 **if** $l = l_s$ **then**
5.3 $\quad$ `solve`$(A_l v_l = f_l)$
5.4 **else**
5.5 $\quad \overline{v}_l$ = `smooth`(A_l, v_l, f_l, ν_1)
5.6 $\quad r_l = f_l - A_l \overline{v}_l$
5.7 $\quad f_{l-1} = R_l^{l-1} r_l$
5.8 $\quad v_{l-1}^* = 0$
5.9 $\quad v_{l-1}$ = `vcycle`$(A, v_{l-1}^*, f_{l-1}, R, P, l-1, \nu_1, \nu_2)$
5.10 $\quad \tilde{v}_l = \overline{v}_l + P_{l-1}^l v_{l-1}$
5.11 $\quad v_l$ = `smooth`$(A_l, \tilde{v}_l, f_l, \nu_2)$
5.12 **end**
5.13 **return** v_l

illustrates the γ-cycle for $\gamma = 2$.

Algorithm 6: The γ-cycle (W-cycle for $\gamma = 2$).

6.1 **Algorithm** `gammacycle`$(A, v_l, f_l, R, P, l, l_s, \gamma, \nu_1, \nu_2)$
6.2 **if** $l = l_s$ **then**
6.3 $\quad$ `solve`$(A_l v_l = f_l)$
6.4 **else**
6.5 $\quad \overline{v}_l$ = `smooth`(A_l, v_l, f_l, ν_1)
6.6 $\quad r_l = f_l - A_l \overline{v}_l$
6.7 $\quad f_{l-1} = R_l^{l-1} r_l$
6.8 $\quad v_{l-1} = v_{l-1}^* = 0$
6.9 $\quad$ **for** $i = 1 \ldots \gamma$ **do**
6.10 $\quad\quad v_{l-1}$ = `gammacycle`$(A, v_{l-1}, f_{l-1}, R, P, l-1, \nu_1, \nu_2)$
6.11 $\quad$ **end**
6.12 $\quad \tilde{v}_l = \overline{v}_l + P_{l-1}^l v_{l-1}$
6.13 $\quad v_l$ = `smooth`$(A_l, \tilde{v}_l, f_l, \nu_2)$
6.14 **end**
6.15 **return** v_l

While Algorithm 6 assumes that γ is the same on all levels, it is also possible to choose different values of γ on each level. Chapter 5 will discuss an algorithm for optimizing the choice of the parameters γ and l_s.

Full multigrid

The problem of creating a good initial approximation for the V-cycle is solved by the *full multigrid* algorithm. It is illustrated in Fig. 2.9. The algorithm starts with computing the solution on the lowest level l_s, where it is cheap to solve the linear system. The solution is then propagated to the next finer level, where it is used as an initial approximation for one or more γ-cycles.

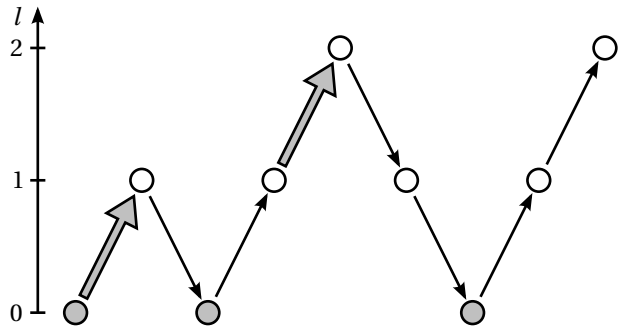

Figure 2.9: The full multigrid scheme with $\mu = 1$ and $\gamma = 1$ for $l_s = 0$ and $l_f = 2$.

Algorithm 7 formally defines this algorithm. The coarsest and finest levels, l_s and l_f, respectively, have to be given as parameters. The parameter μ defines how many complete γ-cycles are executed on each level.

For the prolongation of the solution a new prolongation operator may be necessary. For the prolongation of the correction in the γ-cycle (line 6.12) an operator with a relatively low order of accuracy is necessary, because the correction is smooth on the fine mesh. The solution, however, is generally not smooth on the fine mesh; its smoothness is determined by the differential operator A. Therefore, in order to prolongate the solution from level $l-1$ to level l with sufficient accuracy, the operator has to have an order that is higher than the discretization order of A_l [48]. The operator for prolongating the solution is denoted by $\hat{P}^l_{l-1}$.

Algorithm 7: The full multigrid algorithm.

```
7.1 Algorithm fullmg(A, ν, f, R, P, P̂, l_s, l_f, μ, γ, ν_1, ν_2)
7.2 u_{l_s} = solve(A_{l_s} u_{l_s} = f_{l_s})
7.3 for l = l_s + 1 ... l_f do
7.4 |   v_l = P̂^l_{l-1} u_{l-1}
7.5 |   for i = 1 ... μ do
7.6 |   |   v_l = gammacycle(A, v_l, f_l, R, P, l, l_s, γ, ν_1, ν_2)
7.7 |   end
7.8 end
7.9 return v_l
```

Since the initial approximation to the solution is now very good on each level, small numbers for ν, μ, and γ that are independent of the size of the linear system are sufficient for solving the system on level l_f up to discretization accuracy. As Section 2.3.5 will show in more detail, the overall computational effort of the full multigrid algorithm is proportional to the number of unknowns on level l_f. Thus, the solver has *optimal complexity*.

2.3.4 The full approximation scheme

All multigrid schemes described up to now solve the residual equation $Ae = r$ on the coarse mesh. This technique is not applicable to non-linear PDEs, because for these the residual equation does not necessarily have a solution. For solving non-linear

PDEs, the *full approximation scheme* (FAS) has been developed [14]. This thesis does not consider non-linear PDEs in detail, but FAS is important also for linear PDEs, namely in the context of adaptive mesh refinement (see Chapter 4). Therefore, it is described in the following paragraphs.

Instead of approximating only the residual equation, FAS fully approximates the solution on the coarse mesh. The relation between the residual and the solution can be seen by expanding the residual equation:

$$\begin{aligned} & A_l \overline{e}_l = \overline{r}_l \\ \Leftrightarrow\ & A_l \overline{e}_l + A_l \overline{v}_l - A_l \overline{v}_l = \overline{r}_l \\ \Leftrightarrow\ & A_l \left(\overline{e}_l + \overline{v}_l\right) - A_l \overline{v}_l = \overline{r}_l && (2.28) \\ \Leftrightarrow\ & A_l u_l - A_l \overline{v}_l = \overline{r}_l \,. && (2.29) \end{aligned}$$

Both the current approximation and the residual are known and can be restricted to the coarse mesh.

$$\begin{aligned} v^*_{l-1} &= \hat{R}^{l-1}_l \overline{v}_l \\ d_{l-1} &= R^{l-1}_l \overline{r}_l \,. \end{aligned}$$

Since $\overline{v}_l$ is generally not smooth, injection is used for restricting it. Using full weighting would not pay off; it could even add additional errors, e. g., if $\overline{v}_l$ already contained the exact solution. The current error is not known, but it also has a representation on the coarse mesh, which will be denoted by

$$c_{l-1} = R^{l-1}_l \overline{e}_l \,. \tag{2.30}$$

Using these relations, the expanded residual equation (2.28) can be approximated on the coarse mesh:

$$\begin{aligned} & A_{l-1} \left(R^{l-1}_l \overline{e}_l + \hat{R}^{l-1}_l \overline{v}_l\right) - A_{l-1} \hat{R}^{l-1}_l \overline{v}_l = R^{l-1}_l \overline{r}_l \\ \Leftrightarrow\ & A_{l-1} \left(c_{l-1} + v^*_{l-1}\right) - A_{l-1} v^*_{l-1} = d_{l-1} \end{aligned}$$

Defining

$$v_{l-1} = c_{l-1} + v^*_{l-1} \text{ and} \tag{2.31}$$

$$f_{l-1} = d_{l-1} + A_{l-1} v^*_{l-1} \tag{2.32}$$

as solution and right-hand side, respectively, the problem on level $l-1$ can be written as

$$A_{l-1} v_{l-1} = f_{l-1} \,.$$

The correction for the fine mesh can be determined from the solution on the coarse mesh using (2.31):

$$c_{l-1} = v_{l-1} - v^*_{l-1} \,.$$

As per (2.30) c_{l-1} is the approximation of $\overline{e}_l$ on the coarse mesh, it can be prolongated and used for correcting the current approximation on the fine mesh:

$$\tilde{v}_l = \overline{v}_l + P^l_{l-1} c_{l-1} \,.$$

These considerations are summarized in Algorithm 8, which formally describes the full approximation scheme. Like in the γ-cycle, the system on level $l-1$ is solved by applying the FAS-cycle recursively. v^*_{l-1} is used as initial guess for the first FAS-cycle on the coarse mesh.

Algorithm 8: The full approximation scheme, based on the γ-cycle defined in Algorithm 6.

8.1 **Algorithm** `FAScycle`$(A, v_l, f_l, R, \hat{R}, P, l, l_s, \gamma, \nu_1, \nu_2)$
8.2 **if** $l = l_s$ **then**
8.3 $\quad v_l$ = `solve`$(A_l v_l = f_l)$
8.4 **else**
8.5 $\quad \overline{v}_l$ = `smooth`(A_l, v_l, f_l, ν_1)
8.6 $\quad \overline{r}_l = f_l - A_l \overline{v}_l$
8.7 $\quad d_{l-1} = R_l^{l-1} \overline{r}_l$
8.8 $\quad v_{l-1} = v_{l-1}^* = \hat{R}_l^{l-1} \overline{v}_l$
8.9 $\quad f_{l-1} = d_{l-1} + A_{l-1} v_{l-1}$
8.10 $\quad$ **for** $i = 1 \ldots \gamma$ **do**
8.11 $\quad\quad v_{l-1}$ = `FAScycle`$(A, v_{l-1}, f_{l-1}, R, \hat{R}, P, l-1, l_s, \gamma, \nu_1, \nu_2)$
8.12 $\quad$ **end**
8.13 $\quad \tilde{v}_l = \overline{v}_l + P_{l-1}^l \left(v_{l-1} - v_{l-1}^* \right)$
8.14 $\quad v_l$ = `smooth`$(A_l, \tilde{v}_l, f_l, \nu_2)$
8.15 **end**
8.16 **return** v_l

Correspondence with correction scheme

For *linear* systems of equations the correction scheme and the full approximation scheme are analytically equal, i. e., for identical input parameters, the two algorithms return the same approximation to the solution. This property will be required to prove the correctness of the adaptive mesh refinement algorithm presented in Chapter 4. In order to prove the equality for the γ-cycle, it will first be proved for the two-grid cycle.

Theorem 2.3.1. *The iteration matrix of a two-grid cycle using the full approximation scheme (Algorithm 8) is equal to the iteration matrix of a two-grid cycle using the correction scheme (Algorithm 4).*

Proof. Since the compared numerical schemes are identical from the beginning until the residual computation (lines 4.3 and 8.6, respectively), and from the post-smoothing step (lines 4.8 and 8.14, respectively) until the end, it is sufficient to compare the iteration matrices for the steps in between.

For the correction scheme, expanding the formula for $\tilde{v}_l$ (line 4.7) by using lines 4.4–4.6 yields

$$\begin{aligned} \tilde{v}_l &= \overline{v}_l + P_{l-1}^l v_{l-1} \\ &= \overline{v}_l + P_{l-1}^l A_{l-1}^{-1} f_{l-1} \\ &= \overline{v}_l + P_{l-1}^l A_{l-1}^{-1} R_l^{l-1} r_l \,. \end{aligned} \tag{2.33}$$

For the full approximation scheme, expanding the formula for $\tilde{v}_l$ (line 8.13) yields

$$\begin{aligned} \tilde{v}_l &= \overline{v}_l + P_{l-1}^l \left(v_{l-1} - v_{l-1}^* \right) \\ &= \overline{v}_l + P_{l-1}^l \left(A_{l-1}^{-1} f_{l-1} - v_{l-1}^* \right) \\ &= \overline{v}_l + P_{l-1}^l \left(A_{l-1}^{-1} \left(R_l^{l-1} \overline{r}_l + A_{l-1} v_{l-1}^* \right) - v_{l-1}^* \right) . \end{aligned}$$

In the following step, linearity of A_{l-1} is assumed, i. e., the proof would fail for non-linear systems of equations at this point. The equation is further transformed into

$$\begin{aligned}\tilde{v}_l &= \overline{v}_l + P_{l-1}^l \left(A_{l-1}^{-1} R_l^{l-1} \overline{r}_l + A_{l-1}^{-1} A_{l-1} v_{l-1}^* - v_{l-1}^* \right) \\ &= \overline{v}_l + P_{l-1}^l A_{l-1}^{-1} R_l^{l-1} \overline{r}_l \,. \end{aligned} \tag{2.34}$$

Equations (2.33) and (2.34) are equal, i. e., the coarse-grid correction matrices of the correction and the full approximation scheme are equal for linear systems of equations. Since the schemes are identical in the remaining parts, they are equal altogether, and they yield the same results when employed in the two-grid algorithm. □

A γ-cycle over three levels ($l_f = l_s + 2$) consists of pre- and post-smoothing on level $l_s + 2$ and *two-grid* cycles over levels l_s and $l_s + 1$ for coarse-grid correction. The full approximation scheme and correction scheme have been proven equal for the two-grid cycle, and the schemes do not differ in the smoothing on level l_f. Therefore, the γ-cycle over three levels computes the same solution v_{l_f} with either of the two schemes. A γ-cycle over an arbitrary number of n levels with $n \geq l_s + 3$ consists of smoothing on the finest mesh and recursive γ-cycles on the $n-1$ coarser meshes. Therefore, the full approximation scheme is analytically equivalent to the correction scheme in γ-cycles over any number of levels.

In practice, however, it is possible that the two schemes do not produce numerically equivalent results. The full approximation scheme has more numerical operations than the correction scheme, and the linear system that has to be solved on the coarsest level is different. When running the algorithms on a computer, the typical effects of limited precision, like roundoff errors, will be present. The numerical scheme used for solving "exactly" on the coarsest mesh may work slightly better for one of the problems, introducing differences at this point. Therefore, full approximation scheme and correction scheme may still produce different results, even if they are implemented in a common software framework, e. g., HHG.

2.3.5 Convergence and computational complexity

Their computational complexity is the most important argument for using multigrid methods. It is astonishing to see how basic iterative schemes with quite bad computational complexity are employed to construct schemes with *optimal* complexity. This section highlights models for the convergence rates and the computational complexity of the relevant algorithms—from the Jacobi iteration up to the full multigrid scheme—in the style of a survey. The reader shall be provided with a summary of the arguments leading to the statement that full multigrid is an optimal solver, as claimed in Section 2.3.3. The required proofs and mathematical foundations are mostly not presented in detail, because they are not required in the remainder of this thesis. Instead, references to the detailed proofs in the literature are provided.

The computational complexity of an algorithm is the number of elementary numerical operations that are required to solve a problem, depending on the problem's size. It will not be given as an exact number of operations, but in $\mathcal{O}$ notation, which relates the number of unknowns N to the required number of elementary operations M by

$$M = \mathcal{O}(f(N)) \;\Rightarrow\; M \leq C \cdot f(N)\,, \text{ where } 0 < C < \infty\,.$$

The computational complexity is not to be confused with the actual computational performance on a computer, which will be discussed in Chapter 3.

Since the number of elementary operations per iteration is constant for the iterative schemes considered here, the decisive quantity for determining the computational complexity is the number of iterations required to solve the problem. The problem is considered *solved*, if the algebraic error of the current approximation is smaller than the discretization error. Clearly, the number of iterations depends on the difference between the initial error and the discretization error, but we can safely assume that it is only a few orders of magnitude. The number of iterations required to reduce the error by a factor of ten shall be denoted by ν. The relationship between the error after ν iterations $e^{(\nu)}$ and the initial error $e^{(0)}$ is then given by

$$\frac{\|e^{(\nu)}\|}{\|e^{(0)}\|} \leq \frac{1}{10}, \tag{2.35}$$

where $\|\cdot\|$ is the L_2 norm.

In order to determine ν, the error reduction per iteration must be known. This quantity, which is called *convergence rate*, is a characteristic number of an iterative scheme. In the following, convergence rates for the schemes introduced above will be derived.

Convergence and complexity of basic iterative schemes

Each of the basic iterative schemes is represented by its individual *iteration matrix* M, which represents one iteration of the algorithm. It connects an approximation $v^{(i)}$ with the approximation $v^{(i+1)}$ of the next iteration via

$$v^{(i+1)} = Mv^{(i)}.$$

Since the schemes are fix-point iterations, they do not alter the solution, i. e.,

$$u = Mu.$$

Subtracting the two equations shows how the algebraic error is affected by the iteration matrix:

$$\begin{aligned} & u - v^{(i+1)} = M(u - v^{(i)}) \\ \Leftrightarrow\ & e^{(i+1)} = Me^{(i)} \\ \Leftrightarrow\ & e^{(i+1)} = M^{i+1}e^{(0)} \end{aligned}$$

For the error in the L_2 norm,

$$\|e^{(i+1)}\| = \|M^{i+1}\|\|e^{(0)}\|$$

holds, if the corresponding matrix 2-norm

$$\|M\| = \sqrt{\rho(M^T M)},$$

is used, The matrix 2-norm is based on the *spectral radius*, which is given by

$$\rho(A) = \max|\lambda(A)|,$$

where $\lambda(A) = (\lambda_1(A), \ldots, \lambda_N(A))^T$ are the eigenvalues of A.

Since the iteration matrix is symmetric, its 2-norm can be simplified to

$$\|M\| = \sqrt{\rho(M^T M)} = \sqrt{\rho(M^2)} = \rho(M).$$

Since $\rho(M)$ does not depend on $e^{(i)}$, the error norm after ν iterations can be written as

$$\|e^{(\nu)}\| = \big(\rho(M)\big)^\nu \|e^{(0)}\|. \tag{2.36}$$

Using this equation in (2.35) yields

$$\big(\rho(M)\big)^\nu \leq \frac{1}{10},$$

and, thus, the number of iterations necessary to reduce the error by an order of magnitude is

$$\nu \geq \frac{1}{\log_{10}\big(\rho(M)\big)}. \tag{2.37}$$

In order to determine ν, the spectral radius of M has to be calculated. $\lambda(M)$ will be derived from $\lambda(A)$, the eigenvalues of the system matrix of the model problem. Recall from (2.26) that the iteration matrix of the Jacobi iteration is basically just a re-ordering of A, the system matrix of the model problem. Therefore, the iteration matrix has the same eigenvalues as the system matrix. For the damped Jacobi iteration, the iteration matrix is

$$\begin{aligned}&(1-\omega)I + \omega D^{-1}(L+U)\\ &= I - \omega\big(I - D^{-1}(L+U)\big),\end{aligned}$$

and, therefore, the eigenvalues of the iteration matrix are

$$\lambda_k(M) = 1 - \omega(1-\lambda_k(A)), \quad k = 1,\ldots,N.$$

As shown in [48], the system matrix of the model problem (2.1) has the eigenvalues

$$\lambda_k(A) = \cos\left(\frac{k\pi}{N}\right), \quad k = 1,\ldots,N-1.$$

Large values of $\lambda_k(M)$ are associated with small k, because then $\cos(k\pi/N)$ is close to 1. For small k the approximation

$$\cos\left(\frac{k\pi}{N}\right) \approx 1 - \frac{1}{2}\left(\frac{k\pi}{N}\right)^2$$

holds, and the largest eigenvalue is

$$\begin{aligned}\lambda_1(M) &\approx 1 - \omega\left(1 - 1 - \frac{1}{2}\left(\frac{\pi}{N}\right)^2\right)\\ &= 1 - \frac{\omega\pi^2}{2N^2}.\end{aligned}$$

Hence, for the damped Jacobi iteration the error reduction after ν iterations is, according to (2.36),

$$\begin{aligned}\big(\rho(M)\big)^\nu &= \left(1 - \frac{\omega\pi^2}{2N^2}\right)^\nu\\ &\approx e^{-\frac{\omega\pi^2\nu}{2N^2}}.\end{aligned}$$

That means that ν has to increase quadratically with N in order to maintain the desired convergence rate. Thus, the computational complexity of the damped Jacobi iteration is in $\mathcal{O}(N^2)$.

The computational complexity of the Gauss-Seidel and the SOR iterations is also in $\mathcal{O}(N^2)$. The proof follows the same lines (see [48], Section 3.3), but deriving the eigenvalues of the respective iteration matrices is a bit more involved. Generally, the computational complexity can be determined in this way for many iterative schemes, among them the two-grid cycle and the γ-cycle.

Convergence and complexity of the γ-cycle

For more complex schemes, however, a rigorous eigenvalue analysis of the corresponding iteration matrices quickly becomes very challenging. For the two-grid cycle (Algorithm 4), the iteration matrix M_{2G} is composed of the smoother's iteration matrix M_S, the restriction and prolongation matrices R_l^{l-1} and P_{l-1}^l, the operator matrix A_{l-1}, and the identity matrix I_l:

$$M_{2G} \;=\; (M_S)^{\nu_2}\left(I_l - P_{l-1}^l (A_{l-1})^{-1} R_l^{l-1} A_l\right)(M_S)^{\nu_1}\,.$$

Computing the eigenvalues, and, thus, the convergence rate of M_{2G} is possible with *rigorous Fourier analysis*, but more elegant and versatile is the *local Fourier analysis* (LFA). Like the method described for the basic iterative schemes above, the LFA is also based on determining the eigenmodes of the iteration matrix. The decisive difference is that it assumes periodic boundary conditions. Due to this assumption all considerations can be restricted to a single interior mesh node, or, in terms of the linear system, to a single matrix row.

For problems like (2.1) it can be shown that the two-grid convergence rate is *independent* of the number of unknowns (see [48], Section 3.3):

$$\|M_{2G}\| \;=\; \text{const. in } N\,. \tag{2.38}$$

This does not say anything about the computational complexity, though, because the cost for solving the coarse-grid problem—abstracted as $(A_{l-1})^{-1}$ in the iteration matrix—may very well depend on the number of unknowns. Therefore, the computational complexity of the two-grid algorithm will not be analyzed further.

Anyway, the result (2.38) is an important building block in the analysis of the γ-cycle, though. The γ-cycle's iteration matrix M_γ is similar to the two-grid matrix, but the term $(A_{l-1})^{-1}$ is replaced by the recursive term $I_{l-1} - (M_{\gamma,l-1})^\gamma (A_{l-1})^{-1}$:

$$M_{\gamma,l} \;=\; (M_S)^{\nu_2}\left(I_l - P_{l-1}^l \left(I_{l-1} - (M_{\gamma,l-1})^\gamma\right)(A_{l-1})^{-1} P_l^{l-1} A_l\right)(M_S)^{\nu_1}\,.$$

Note that recursive expansion of the term cancels out all matrix inversions except for $(A_{l_s})^{-1}$ on the coarsest level.

The similarity of M_γ and M_{2G} allows for a relatively simple proof that the γ-cycle's convergence rate depends linearly on the two-grid cycle's convergence rate. Given that

$$\|M_{2G,l}\| \;\leq\; \sigma^*\,,$$

where σ^* is a (small) constant independent of N (see (2.38)), one can show that

$$\|M_{\gamma,l}\| \;\leq\; \eta_l\,,\ \text{where } \eta_l = \sigma^* C(\eta_{l-1})^\gamma \text{ and } \eta_1 = \sigma^*\,,$$

where C is a (small) constant independent of N. For $\gamma \geq 2$ one can show that

$$\eta \leq 2\sigma^*$$

by using the limit equation for

$$\eta = \sigma^* + C\eta^\gamma .$$

The detailed proof can be found in [48], Section 3.2. For $\gamma = 1$ the proof is not that simple, but it can also be shown that $\eta \in \mathcal{O}(\sigma^*)$ (see [48], Appendix B).

Thus, the γ-cycle has the same convergence order as the two-grid cycle. In contrast to the two-grid cycle, for the γ-cycle also an upper bound on the number of operations can be given, because the solving step on level $l-1$ with uncertain complexity is replaced by a well-defined recursion. Concretely, an upper bound on the number of operations is determined as follows. The given line numbers refer to Algorithm 6. The steps on level l are smoothing (lines 6.5 and 6.13), residual calculation (line 6.6), recursion (line 6.10), and prolongation (line 6.12). All these steps have linear complexity in N_l. The steps on level $l-1$ are initialization of the approximation (line 6.8) and the recursive γ-cycle (line 6.10). The initialization is linear in N_{l-1}, the complexity of the recursion is defined by recursion. The effort for solving on level $l_s = 0$ (line 6.3) is constant, because the linear system is trivial. If $l_s > 0$, a higher cost for solving has to be assumed. For the following estimation to hold it is necessary that the effort for solving on the coarsest level can be assumed to be constant or negligible compared to the effort on the finest level. The number of operations $O(l)$ on level l can be estimated as

$$O(l) \leq CN_l + O(l-1), \text{ where } 1 \leq C < \infty .$$

Expanding the equation yields

$$O(l) \leq CN_l + CN_{l-1} + \ldots + CN_{l_s+1} + C .$$

Now an essential assumption has to be made. The number of unknowns shall increase exponentially with the level. For the hierarchy in Fig. 2.7 $N_l \approx 2^l$. For such regular refinements in d dimensions one would get $N_l \approx 2^{dl}$. In other application scenarios, e. g., if the mesh hierarchy is constructed by successively coarsening a fine initial mesh, this condition may be harder to fulfill. If the assumption holds, the number of operations is

$$\begin{aligned} O(l) &\leq C2^l + C2^{l-1} + \ldots + C2^0 \leq 2C2^l \\ \Rightarrow O(l) &\in \mathcal{O}(N_l) . \end{aligned}$$

Since the convergence rate is bounded by a constant independent on N, as shown above, the γ-cycle's computational complexity for reducing the error by an order of magnitude is $\mathcal{O}(N)$. This is already a good result, but the number of γ-cycles that is required to drive the error below the discretization error still depends on the initial error. Thus, in order to obtain a solver that only needs a small, *fixed* number of γ-cycles, it is necessary to provide an initial guess with a small error. This is achieved with the full multigrid algorithm.

Convergence of full multigrid

In the full multigrid algorithm a fixed, N-independent number of γ-cycles is executed on each level. Therefore, the total number of operations for this algorithm is also in $\mathcal{O}(N)$.

It can be shown that the algorithm computes an approximation with discretization accuracy, if the operator $\hat{P}$ used for interpolating the approximation is good enough. Concretely, the order of the interpolation has to be better than the order of the discretization. Then, the initial approximation on level l cannot become arbitrarily large, if the final approximation on level $l-1$ was within discretization accuracy. Then, a fixed number of γ-cycles on level l reduces the error below discretization accuracy again. A concise proof of this statement can be found in [48], Section 3.2.2. The bottom line is that full multigrid requires only $\mathcal{O}(N)$ operations to solve a linear system in N unknowns with discretization accuracy. In other words, full multigrid is a solver with *optimal* computational complexity.

2.4 Hierarchical Hybrid Grids

The concepts of finite elements and multigrid, which were described in the previous sections, are combined in the *Hierarchical Hybrid Grids* (HHG) software library. HHG strives to provide highly efficient algorithms for numerical simulations on the largest scale, using supercomputers.

The scale of a numerical simulation is determined by the size of the domain and the desired accuracy of the solution. Recall the example given in the introduction: the acoustical simulation of a concert hall. The desired accuracy in this scenario is determined by the sound frequencies that shall be simulated. If the full audible spectrum shall be covered, then the required mesh resolution is in the millimeter range—assuming that the sound propagation is simulated by computing the oscillations of the air pressure that cause the sound waves. Note that at this point it does not matter which discretization method is used—finite differences, finite elements, or some other technique. Only the choice of the physical model—i. e., the PDE—makes a difference. An indirect simulation in the *frequency domain*, for example, would likely require a lower mesh resolution, however, at the cost of not being able to precisely simulate acoustical phenomena like interference. The details of the domain's geometry also influence the required mesh resolution. If the mesh does not resolve details in the concert hall's wall that influence the sound propagation, the accuracy of the simulation suffers. Here, the choice of the discretization method does make a difference. If the domain is regular, then a discretization with finite differences may be most suitable, because it achieves the required mesh resolution with the smallest number of nodes. If the domain is irregular, a finite element discretization is more suitable, because it can achieve a homogeneous mesh resolution with the smallest number of nodes by adapting the elements to the domain's shape.

In relation to the required resolution, the size of the domain in the concert hall example is large. A regular mesh that covers a domain with a size of 10 m in each dimension at a resolution in the millimeter range (i. e., 10^{-3} m) consists of $(10^4)^3 = 10^{12}$ nodes. Assuming that storing the physical values at the nodes in a computer's memory requires 8 bytes per value, the solution alone would already occupy 8×10^{12} bytes (8 terabytes) of memory. It is obvious that supercomputers are required, if simulations at this scale and beyond shall be performed.

Besides the memory demand, the challenge of such a simulation is the time needed to compute the solution. If a solver with non-optimal complexity is used, the huge number of nodes (i. e., unknowns) easily causes the time to solution to become unacceptably long, even if the solver is implemented efficiently. Assuming, for example, a solver that can "quickly" solve a linear system with $N = 10^9$ unknowns in 0.1 s but has a complexity of $\mathcal{O}(N^2)$, the solution of a system with 10^{12} unknowns would take $(10^{12}/10^9)^2$ times longer, i. e., 10^5 s. Thus, when having to choose between a solver that is very fast for systems of moderate size and a solver with optimal complexity, for the simulations we are interested in the latter is clearly preferable.

The described requirements lead to the three main goals that guided the development of HHG:

- to discretize the domain with the *flexible* finite element method,
- to solve the linear systems with multigrid methods that have *optimal complexity*, and
- to make the implementation *portable* to a wide range of supercomputer architectures.

HHG was initially designed and implemented by Benjamin Bergen. The basic concepts and data structures, which result naturally from the above goals, have remained the same up to now; they are described in detail in [10]. Since a basic understanding of these concepts will be necessary in the remainder of this thesis, the following Sections recapitulate them briefly. The software architecture, which has evolved from the original version in order to support adaptive mesh refinement (see Chapter 4), is described in Section 2.4.4. The major improvements that have been implemented within the scope of this dissertation are summarized in Section 2.4.5. Section 2.4.6 presents sample code that shows how to implement a full multigrid solver with the HHG library.

2.4.1 Concepts

This section presents the main concepts that HHG's design follows in order to meet the described goals. Numerical as well as software-architectural design decisions are influenced by these goals.

Numerical concepts

In order to have flexibility for complex domains, HHG uses the finite element method. In order to perform simulations with optimal complexity, HHG uses the full multigrid method. The main challenge that arises from these choices is to combine both methods in a way that retains computational efficiency on massively parallel computers. This is achieved by the central design concept, which defines how the multigrid mesh hierarchy is created: instead of *coarsening* an initial fine mesh, HHG *regularly refines* the elements of an initial coarse mesh. This approach produces many *similar* finite elements on the higher multigrid levels.

Fig. 2.10 shows an example for a mesh in 2D with triangles and quadrilaterals. The initial coarse mesh is refined twice. The shading of the elements in one of the triangles highlights their similarity. The sub-elements of each coarse triangle fall into two similarity classes: one is a scaled version of the coarse element (gray); the other

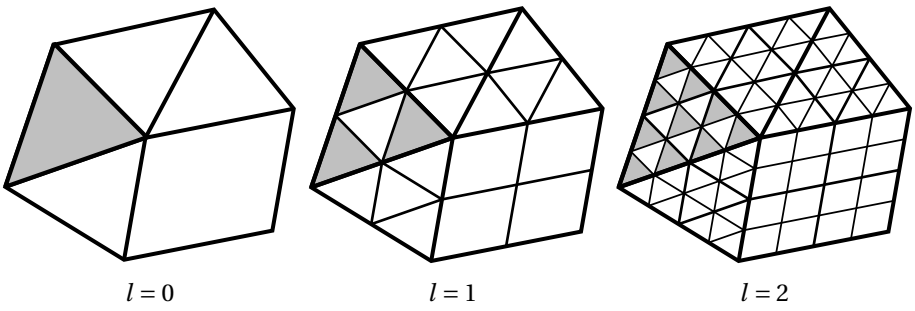

Figure 2.10: Regular refinement of a 2D mesh with three triangles and one quadrilateral.

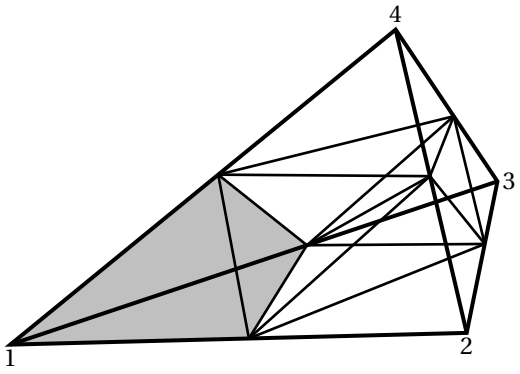

Figure 2.11: Regular refinement of a tetrahedron.

one is a scaled and rotated version of the coarse element (white). The sub-elements of the quadrilateral are all in a single similarity class.

Fig. 2.11 shows a regular tetrahedron refinement according to Bey [11]. The refinement is constructed in a similar way as the triangle refinement: the mid-points of adjacent edges (e. g., $\overline{12}$ and $\overline{14}$) are connected with new edges; additionally, one pair of non-adjacent edges (here: $\overline{13}$ and $\overline{24}$; also other pairs could be chosen) is connected with a new edge. The resulting fine mesh consists of eight tetrahedra, which fall into five similarity classes: one class for the tetrahedra at the nodes of the original tetrahedron and four classes for the tetrahedra in the interior. The tetrahedra at the nodes of the original tetrahedron (one of them is highlighted) are scaled versions of the original tetrahedron. An important property of this refinement technique is that the number of similarity classes is stable; when refining the sub-elements again, the resulting sub-elements will belong to the same five similarity classes again.

This regular refinement technique has several advantages. All are connected to the fact that identical elements have the same stiffness matrix. The stiffness matrices of elements on the higher multigrid levels can be calculated simply by scaling the matrices of the corresponding similar elements in the coarse mesh. The first benefit of this property is that the time required for computing these stiffness matrices is just a fraction of the time that would be required for computing each element's matrix individually. Exploiting this property greatly speeds up the construction of the system matrix.

The second benefit is that the system matrix does not even have to be constructed. Due to the regularity of large parts of the meshes on the higher levels—they are irregular only at the boundaries between the coarse elements—the numerical algorithms can be implemented as *stencil codes*. In this technique, for each mesh node the non-zero entries of the row of the operator matrix that corresponds to the node's unknown in the linear system is stored. The technique is called stencil code, because each row of the operator matrix defines how the corresponding node is coupled to its neighboring nodes. This coupling can be visualized as a stencil at the node. The numerical algorithms, instead of performing matrix-vector multiplications, iterate over the nodes and apply the stencil. Due to the regularity of the mesh, many nodes have identical stencils. Of course, each unique stencil has to be stored only once. Therefore, the memory required for storing the operators is minimal, compared to the memory required for storing the variables.

This is the main benefit, which enables HHG to solve extremely large simulations very quickly. Due to the minimal memory demand of the operators, almost the complete main memory (RAM) of the supercomputer can be filled with variables (e. g., unknowns, right-hand side, and residual), and the number of unknowns (i. e., mesh nodes) can therefore be very large. When executing the numerical algorithms, loading the stencils from RAM takes almost no time, and almost the complete memory bandwidth is available for transferring the data stored in the variables. In other words, the number of bytes transferred from/to memory per unknown is very low. Thus, the number of unknowns that can be processed by a numerical algorithm per second is very high.

Software engineering concepts

While the numerical concepts are the basis for solving large simulations quickly, proper software engineering has to ensure that the implementation uses current computer architectures efficiently. Another requirement is to produce software that is portable to the relevant supercomputer systems.

Practical experience shows that the performance of finite element solvers is usually limited by a computer's data transfer bandwidth rather than by its processor speed. This is applies to stencil codes as well as to codes that store the system matrix in other formats. Therefore, an important concept of the HHG implementation is to optimize the data transfer. Note that this is equally important for local memory access and for remote communication. With the tremendous increase in network performance in recent years at a near-stagnation of local memory performance, the difference in data transfer speed between local memory access and remote communication has become small, in terms of both latency and bandwidth.

HHG maximizes transfer bandwidth and minimizes latency-induced losses by storing and transferring data in large blocks. For transferring a block of data that is stored as a contiguous array in the RAM, latency occurs only at the start of the transfer. This is true for both local memory access and remote communication: for the transfer between local memory and processors, the hardware provides automatic prefetching of consecutively stored data; the transfer over the network can be grouped into a single communication event with latency occurring only at its beginning.

When designing software for portability, the two key criteria are to use a common programming language, e. g., C++, and to refrain from using non-standardized language features or additional software libraries. While this approach is most portable,

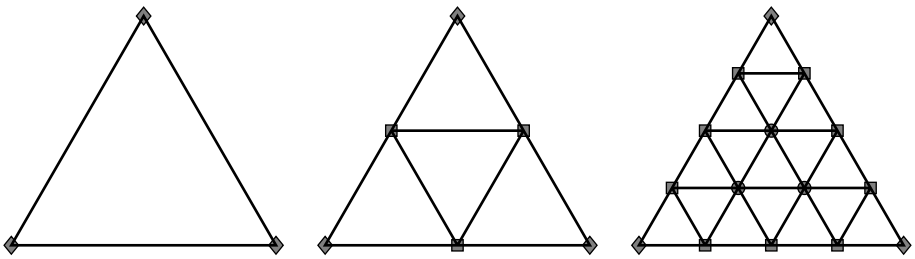

Figure 2.12: Triangle on refinement levels 0, 1, and 2, with mesh nodes assigned to vertices (◊), edges (□), and the triangle's interior (◯).

the resulting software will not be fast on all computer architectures that play a role in today's supercomputing scene. Some systems draw their performance from special hardware, like *graphics processing units* (GPUs). The usage of special libraries is unavoidable in order to achieve feasible performance on these systems. Moreover, the ideal performance optimization technique depends on the hardware, i. e., one implementation will never run efficiently on all computers. In order to produce software that is portable and even capable of accommodating to such rapid environmental changes as in the field of GPU computing, HHG strictly separates the high-level algorithms from the kernel routines that perform computation and communication. The high-level framework, which contains, e. g., basic functions for handling finite element meshes and performing various multigrid algorithms, is written with special attention to portability, so it can easily be compiled on any supercomputer host system. The kernels, which perform the low-level linear algebra operations and the inter-process communication, can easily be replaced with optimized versions.

2.4.2 Primitives and data structures

As argued above, it is important to organize the data into as large as possible contiguous arrays in the RAM. HHG achieves this goal by breaking down each finite element into the basic geometric entities that constitute the element—the element's *primitives*. A three-dimensional finite element consists of the following primitives:

- the element's interior (a three-dimensional primitive),
- the element's faces (two-dimensional primitives),
- the element's edges (one-dimensional primitives), and
- the element's vertices (zero-dimensional primitives).

A tetrahedron, for example, consists of its interior volume, four faces, six edges, and four vertices. The mesh nodes on each refinement level are assigned to one of these primitives. Fig. 2.12 shows an example, for simplicity in 2D: a triangle that is refined up to level two and the assignment of the mesh nodes to the triangle's primitives.

Organizing the nodes into primitives separates the structured from the unstructured parts of a mesh. Observing the couplings of the elements' nodes in Fig. 2.10 reveals that the nodes in the elements and on the edges are coupled to their neighboring nodes in a regular way. Only the nodes on the vertices have different numbers of

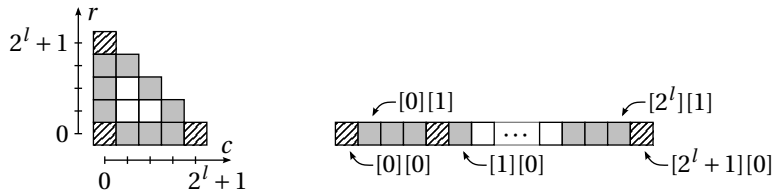

Figure 2.13: Linear memory array for a triangle at refinement level l (here: $l = 2$). The labels $[r][c]$ indicate the mapping of the triangle's nodes (left) to entries in the linear memory array (right).

neighbors, depending on how many elements they are adjacent to. For HHG's tetrahedral meshes, the same principle holds. The interior nodes of a tetrahedron always have 14 neighbors, also the nodes on faces between two tetrahedra. The nodes on boundary faces always have 11 neighbors. Only for the nodes on the edges and vertices the number of neighbors is variable, because these primitives can be adjacent to arbitrarily many elements.

This finding can be used to implement data structures that are beneficial for the computational efficiency of the numerical algorithms. Fig. 2.13 shows how the nodes in a triangle can be mapped to the entries of a linear memory array. The nodes of any regularly refined triangle—independent of its shape—can be organized with a *row* and *column* indexing scheme. The nodes' values are then stored row-wise in the memory array. Note that the memory array does not only contain the triangle's interior nodes, but also the nodes on the edges and vertices, although the numerical algorithms update only the interior nodes. For a description of all the data structures that are needed in HHG with uniform refinement, see [10]. For the implementation of adaptive refinement, additional data structures were developed; they will be introduced in Chapter 4.

By storing also the values on the triangle's boundary nodes, the memory array contains all values that are necessary for the algorithm to update the interior nodes; it does not have to access memory arrays of other elements. The values on the boundary are called *ghost values*. They have to be fetched from the primitive that owns the corresponding nodes when they have been updated there. What is not immediately visible in Fig. 2.13, because l is quite small: for larger values of l, the number of interior primitives dominates the number of ghost primitives, and the rows become very long. Thus, a numerical algorithm should update the nodes in row-wise order. Then the memory array is traversed almost completely linearly. On current computer architectures, linear memory access is much faster than irregular access, because the computer assumes that the access is linear and reads ahead data of the locations that follow the currently accessed one.

2.4.3 Programming languages and standards

For the software framework, the C++ programming language [43] is used. Programs written in C++ are portable to all supercomputer host systems. The object-oriented programming features of C++ allow for flexible and error-proof software engineering. The resulting programs are competitively fast, because well-optimizing compilers exist for all important processor types. Since C++ is widely used in the software engineering community, the obstacles for other scientists who want to start con-

tributing to HHG are small.

Most computational kernels are implemented in FORTRAN-77 (for simplicity referred to as FORTRAN, below) and ANSI C [37, 27]. The historical reason for FORTRAN is that on the Hitachi SR8000 computer, which was the first target architecture of HHG, optimized FORTRAN programs yielded a much higher performance than optimized C/C++ programs. Recent, not yet published experiments indicate that on current hardware HHG reaches the same performance with computational kernels written in C.

For inter-process communication, HHG relies on the *message passing interface* MPI standard [30], using the MPI libraries provided by the software infrastructure on the host system. MPI is the leading standard for inter-process communication on distributed-memory systems and is available on virtually all supercomputers. Thread-level parallelism, e. g., via OpenMP [12], is not implemented in HHG. When the development of HHG started, processors had at most four cores, and at that time hardly any application that tried to mix MPI and thread-level communication had any performance gain over using MPI only. Since HHG targets very large simulations running on thousands of processors, providing a version using only thread-level communication does not make sense for HHG, either. Currently, the number of cores per processor is growing, though. Therefore, hybrid communication strategies using both MPI and OpenMP may become necessary in the future.

2.4.4 Software architecture

The HHG library is organized into several components. Each of them consists of several C++ classes.

- The central *mesh* classes store the mesh topology, handle the multigrid levels and the adaptive mesh refinement.
- The *primitive* classes provide a class for every implemented finite element type (currently, tetrahedral elements are implemented) and for their lower-dimensional neighbors (faces, edges, vertices). They provide interfaces to routines that are specific for each primitive type, e. g., memory allocation, numerical functions, and communication.
- The *memory handling* subsystem allocates and de-allocates the memory for the variables and operators defined on the mesh. Every primitive type has a corresponding memory handler that knows about the number of unknowns—and, therefore, the memory demand—of the primitive, depending on the refinement level.
- The *compute* kernels, core functions for basic numerical operations like adding variables or applying a differential operator, are implemented specifically for each primitive type. They can easily be replaced by optimized versions for a specific type of hardware.
- The *communication* subsystem handles the communication of data between neighboring mesh primitives, for local memory transfers as well as for remote communication.
- The *finite element setup* classes perform the numerical integration on the finite elements and take care of the discretization of differential operators (currently, the Laplace operator is implemented) for each finite element type.

- A collection of *algorithms* performs high-level numerical operations with the variables and operators defined on the mesh. The algorithms reach from basic linear algebra like norms or vector sums to advanced multigrid like the full approximation scheme or the full multigrid algorithm.

These building blocks are described in detail, below. The classes that are required for adaptive mesh refinement and some other software components, e. g., the performance analysis tools, are described in separate chapters. Note that the following paragraphs and class diagrams do not intend to provide a complete description of all classes, since that would not match the majority of the readers' interest. Instead, they shall show in a very condensed form how the HHG concepts are cast into software.

The mesh classes

Fig. 2.14 gives an overview of the classes that are responsible for handling the finite element mesh topology. The central class of this compound—and of the whole HHG library—is `hhgMesh`. It acts as a relay station for many other classes to interact with each other; these classes have references to `hhgMesh` and can in this way access other components of the library. `hhgMesh` itself does not contain any major algorithms, it is mostly a storage space for references to other classes handling the finite element mesh and to other library components like the communication subsystem. Its main attributes are references to the `hhgPrimitiveStore` class described below and to the `hhgMPIController` class described later on in the communication subsystem part. The implementation of the `hhgMesh` class has methods, too, of course, and many more attributes, but none of them are important in this compact overview. Also for the other classes only the necessary members are depicted, in order to keep the figures readable.

Mesh primitives are not stored in `hhgMesh`, directly, but in a registry with references to all primitives on all refinement levels, called `hhgPrimitiveStore`. It provides two ways of accessing mesh primitives, matching the different needs of the algorithms working with the mesh. The first possibility is to access a mesh primitive by its global identifying number (*ID*). The function `getPrimitive(...)` takes the refinement level, the primitive dimension, and the primitive ID as arguments and returns a pointer to the requested primitive. It uses the attribute `allPrims`, which is implemented with vectors and maps from the C++ standard template library (*STL*). The second possibility is to access groups of primitives. Therefore, the type `hhgPrimitiveGroup` is defined; a variable of this type can have one of the following values:

- `hhgPgWorkingSet`: primitives on which the solution has to be computed,
- `hhgPgDirichletBndry`: primitives on the Dirichlet boundary,
- `hhgPgNeumannBndry`: primitives on the Neumann boundary,
- `hhgPgProcBndry`: primitives that are also present on another process.

Primitives are automatically added to these groups when the mesh is set up in the beginning. They are stored in the attribute `grpPrims`, which is implemented with nested STL vectors.

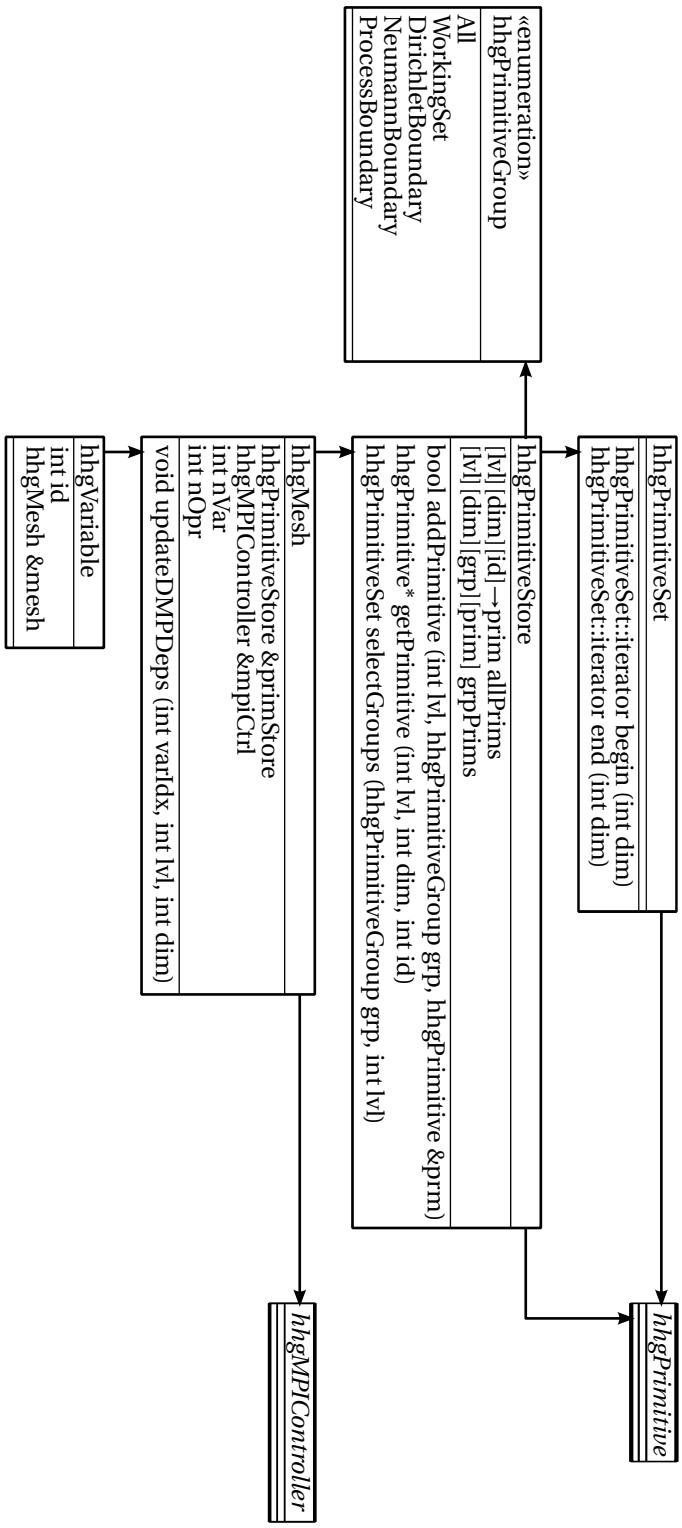

Figure 2.14: The mesh classes.

Listing 2.1: Iteration over Dirichlet and Neumann boundary primitives on level 5.

```
hhgPrimitiveStore pstore;
hhgPrimitiveSet pset = pstore.selectGroups
  (DirichletBoundary|NeumannBoundary, 5);
for (dim_t d=0; d<=3; d++) {
  for (pset::iterator it=pset.begin(d), ie=pset.end(d);
       it!=ie; it++) {
    // Work on primitive *it.
  }
}
```

The function `selectGroups(...)` can be used to select one *or more* groups on a certain level. The selection is returned as an instance of the class `hhgPrimitiveSet`, which hides the data structures in `hhgPrimitiveStore`. `hhgPrimitiveSet` provides access to the selected primitives via an STL-conforming iterator; the functions `begin(dim)` and `end(dim)` return iterators delimiting the user's selection. This concept is most easily explained with the example code in Listing 2.1, which accesses all primitives on the Dirichlet and Neumann boundaries on level 5.

The mesh primitives

The finite element mesh on the coarsest level is made up of mesh primitives that can be refined in a structured way to create the level hierarchy needed in multigrid computing. The dimension of the finite element mesh determines the primitive types used in the construction of the mesh. Each of the primitive types described in Section 2.4.2 is represented by a C++ class. All the primitive classes are derived—directly, or via other classes—from a common abstract class, `hhgPrimitive` (see Fig. 2.15). This class contains all attributes and functions that are independent of the primitive dimension or do not need to be tuned to the primitive's dimension. The class, as well as the derived classes, is templated with the parameter `T`, which defines the data type of the numerical functions. Therefore, HHG can be instantiated for simulations with different numerical precision (e. g., `float` or `double`) with relatively low effort.

Every instance of `hhgPrimitive` can be identified uniquely by the attributes `dim` and `id`. Every primitive has pointers to its neighbor primitives in the `neighbors` data structure. `hhgPrimitive` keeps track of memory allocated for variables on the mesh in the `memArrs` (*"memory arrays"*) data structure; pointers into MPI buffer memory are stored in `mpiDeps` (*"MPI dependencies"*). The memory handling is realized via the class `hhgVariableMemory`, which will be explained below.

Four child classes are derived from `hhgPrimitive`: `hhgVertex`, `hhgEdge`, `hhgFace`, and `hhgElement`. They are the base classes for the zero-, one-, two-, and three-dimensional primitive classes and contain attributes and functions that are independent of the global mesh dimension but dependent on of the primitive dimension. The classes are still abstract, except for `hhgVertex`, whose implementation is the same for meshes in all dimensions. In the third and final tier of classes, the primitive types are implemented specifically for each primitive dimension and mesh dimension. For three-dimensional meshes, these are `hhgVertex`, `hhgVol-`

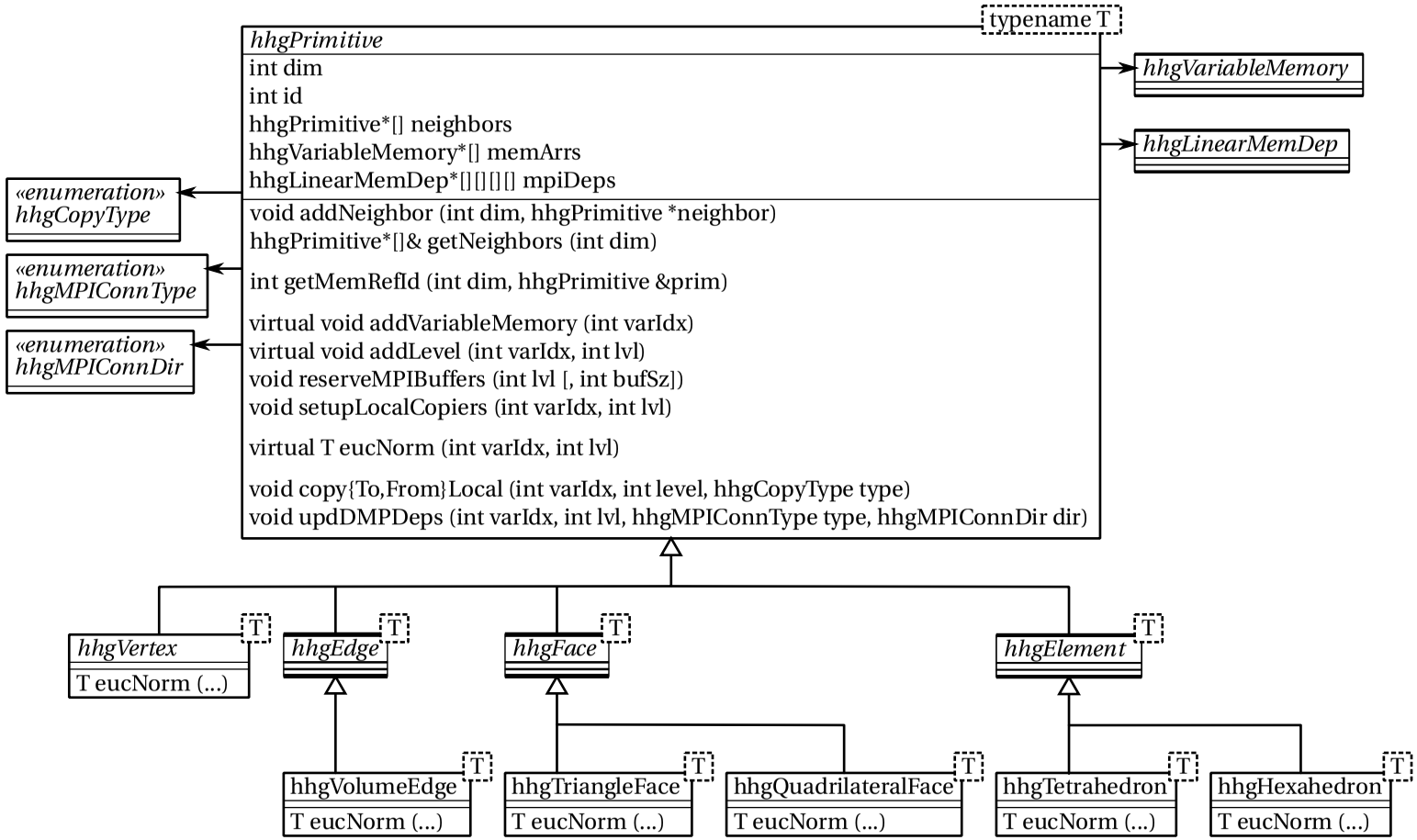

Figure 2.15: The primitive classes with select attributes and functions.

umeEdge, hhgTriangleFace, hhgQuadrilateralFace, hhgTetrahedron, and hhg-Hexahedron. HHG has currently only implemented full support of 3D meshes; for 2D meshes, another set of primitive classes—analogous to the existing ones—would have to be implemented. Note that hexahedral elements and quadrilateral faces are only rudimentarily implemented in the current version of HHG; the diagram includes them, anyway, to show how they fit into the class hierarchy in principle.

Memory handling

The class hhgVariableMemory is the central class when it comes to handling physical memory. For each primitive type, it has a derived class that knows how to efficiently map the primitive type's individual data structures into arrays in physical memory. The classes are shown in Fig. 2.16. They reference specialized classes for copying data between memory arrays; these will be explained below.

The compute kernels

The kernels of the primitives' numerical functions (like eucNorm in Fig. 2.15) are outsourced to simple C or FORTRAN functions that do not use any advanced language features like object-oriented programming. This allows the compiler to create highly optimized machine code. The strict separation from the bulk of the library code also allows for easy replacement of the numerical kernels by versions tuned to special hardware. As a tribute to C/FORTRAN mixed-language programming, data is exchanged between the primitives' numerical functions and the compute kernels via pointers only.

Local and remote communication

The term *local communication* subsumes all data transfers between primitives within the same process, i. e., when primitives can access each other's physical memory addresses. For every combination of two primitive types there is an abstract class hhgP1P2Copier, where P1 is the name of the higher-dimensional primitive type, and P2 the name of the lower-dimensional type. These classes provide the virtual functions copyToOther and copyFromOther for data transfer to and from the neighbor, from the view-point of the *higher*-dimensional primitive.

Each of the abstract classes hhgP1P2Copier has several realizations, one for each type of adjacency that can occur between two primitives of types P1 and P2. Each of these concrete classes implements the data transfer for a specific alignment of two primitives in the most efficient way. Upon mesh setup, an appropriate subclass of hhgP1P2Copier is instantiated for every adjacency of two primitives. When one of the higher-dimensional primitives' copy{To,From}Local functions (see Fig. 2.15) is called, it calls, for each lower-dimensional neighbor, the corresponding copier's copy{To,From}Other function.

An example is shown in Fig. 2.16. The abstract base class, hhgVEVertexCopier, is used for an adjacency between primitives of type hhgVolumeEdge and hhgVertex. The vertex can either be attached to the front or the rear end of the edge; depending on that, the copy functions have to select the correct entry in the edge memory array. Therefore, hhgVEVertexCopier has two realizations, one for each of the adjacency types. Fig. 2.17 shows which data hhgVEVertexZero and hhgVEVertex-One copy between an edge and its adjacent vertices.

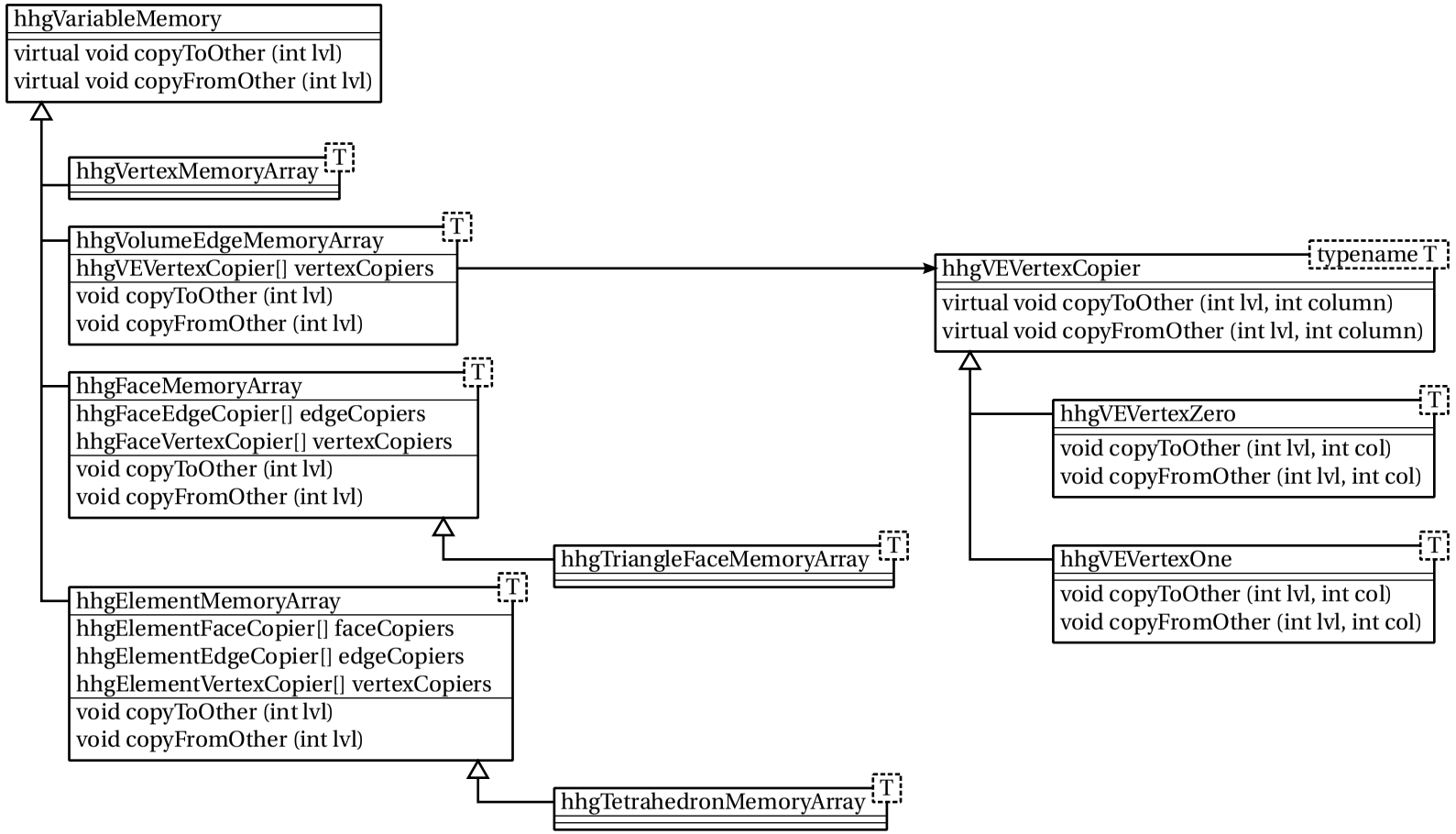

Figure 2.16: The classes for handling physical memory and for copying data between primitives.

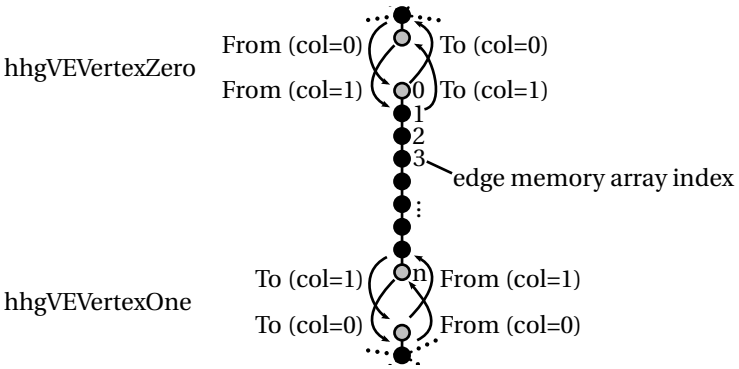

Figure 2.17: The semantics of the copy functions.

The edge-vertex example has been chosen for its simplicity: only two sub-classes are necessary to cover all adjacency cases. For higher-dimensional primitives, adjacent primitives are not only characterized by their location but also by their orientation. For an edge at a triangular face, for example, there are three positions the edge can have, and, for each position, there are two possible orientations. To cover all the cases for this combination of primitive types, six classes and, consequently, twelve different copy functions have to be implemented.

If two primitives that do not reside in the same process, i. e., whose physical memory addresses belong to different processes, need to exchange data, *remote communication* is necessary. In the current HHG version, this type of communication uses the MPI standard, but the modularity of the implementation would allow switching to a different communication paradigm, e. g., OpenMP or MPI mixed with OpenMP, without too much effort.

The remote communication infrastructure uses MPI, but it also re-uses the local communication infrastructure. This has the advantage that the functions performing the actual data transfers between processes are comparably straightforward, thus, and easy to debug—an extremely valuable asset of software that has to run on thousands of processes in parallel. The re-use of the local communication functions is made possible by using *ghost primitives*.

The central class of the remote communication infrastructure is `hhgMPIController` (see Fig. 2.18). A single instance of this class exists in each process; it can be obtained via the `instance` function. The `hhgMPIController` instance contains a set of `hhgMPIChannel` objects, one for each process with which data has to be exchanged The channels contain instances of the `hhgMPIMemoryArray` class. A channel has one `hhgMPIMemoryArray` instance for every mesh variable (i. e., for an instance of `hhgVariable`, see Fig. 2.19). `hhgMPIMemoryArray` stores all the data that has to be communicated with the channel's corresponding process in the `data` vector, where every vector entry corresponds to a multigrid level. The data itself is stored in a contiguous array in physical memory. The array contains the ghost values from *all* primitives adjacent to the channel's process. This way, only one message between a pair of processes is required to communicate all ghost values of a specific level.

Remote communication is initiated by calling `hhgMesh::updDMPDeps` ("update distributed memory processing dependencies") function for a certain level. This function makes sure that all ghost values in `hhgMPIMemoryArray`'s `data` field of that level are up to date by calling `hhgPrimitive::updDMPDeps` of every primitive on a

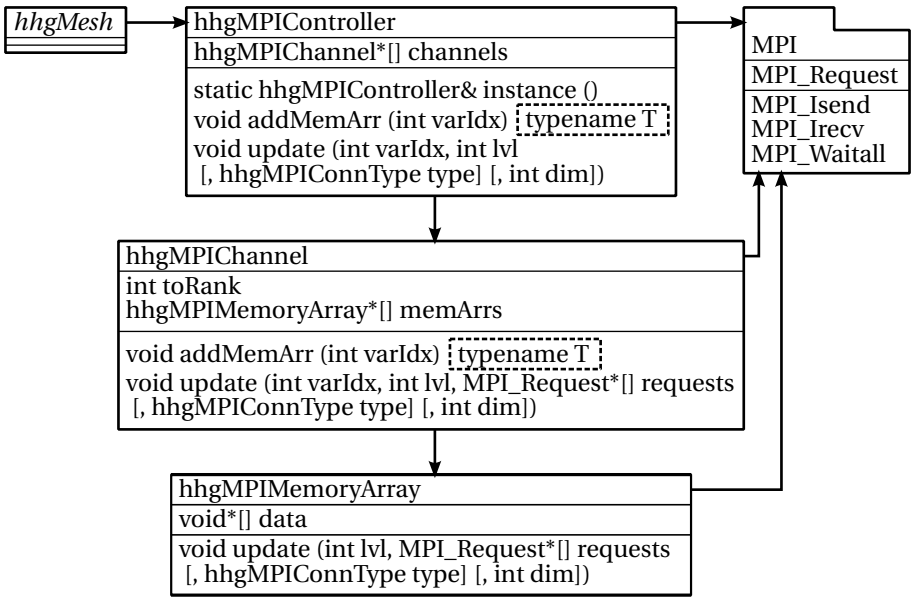

Figure 2.18: Remote communication classes.

process boundary. Then, `hhgMPIController::update` is called, which in turn calls `update` for all channels. The channels pass the update call on to their `hhgMPIMemoryArray` objects, which initiate immediate send and receive (`MPI_Isend` and `MPI_Irecv`) operations. After all MPI memory arrays have initiated the send and receive operations, `hhgMPIController::update` calls `MPI_Waitall` to wait for the completion of the data transfers.

Note that the remote communication classes do not have type-name template parameters like, e. g., the primitive classes (see Fig. 2.15), but the data is stored in pointers of type `void`. Knowing about the type of the communicated data is not necessary, because the communication functions in the MPI standard only deal with non-typed pointers. The advantage for the HHG implementation is that it gets much simpler, because a single `hhgMPIChannel` instance can handle the communication of ghost values from mesh variables with different data types. Type-safety at compile-time is still ensured, because every `hhgMPIMemoryArray` instance is generated via the `hhgMPIController::addMemoryArray` function, which is templated with the data type.

Finite element setup

The finite element setup phase comprises the assembly of the operator stencils and the initialization of the load vector. Therefore, the integrals derived in Section 2.2.7 have to be computed. The HHG classes involved with finite element setup are shown in Fig. 2.19.

The class `hhgOperator` is the generic class for all numerical operators in HHG. A specific operator is obtained by instantiating the concrete class that corresponds to the differential operator in the analytic equation. The only operator currently implemented in HHG is `hhgDiffusionOperator`, which resembles the Laplace operator (also known as “diffusion operator”). The operator assembly is initiated by `hhgOp-`

erator::setup. For every finite element in the input mesh the setup function calls a compute appropriate for the type of the element. Currently implemented, and shown in Fig. 2.19, is hhgTetrahedronStencil::compute. The method uses the virtual method hhgTetrahedronOperator::matrix to obtain a stiffness matrix on the coarsest level, which is distributed to the stencils of the element's mesh points on the coarsest level and—multiplied with an appropriate scaling factor—on the finer levels. Depending on which concrete operator class has been instantiated, the matrix method of the appropriate concrete class of hhgTetrahedronOperator is called; e. g., hhgTetrahedronDiffusion corresponds to hhgDiffusionOperator. The stiffness matrix is computed according to (2.24), depending on the coordinates of the element's corners. The coordinates are passed as arguments in the hhgPoint format, which will not be explained in detail, here. The stiffness matrix is returned in an hhgTetStiffnessMatrix object, a simple container for the matrix entries, which will not be explained further, either.

The classes and functions employed in the initialization of the load vector are shown in the lower part of Fig. 2.19. The setupLoad function of the FinitElements class, which is located in the hhgLinAlg namespace, is used to compute the load vector on a certain multigrid level (lvl) and store it in the variable var. Analogous to the hhgOperator::setup function, it employs different compute functions specifically implemented for the different element types to compute the integral (2.23). The compute functions (currently implemented an depicted in Fig. 2.19: hhgTetrahedronLoad::compute) compute the integral for every element on the desired level, using hhgVolumeFunction to obtain the (analytic) load function values at the integration points p. The load function is actually an instance of a sub-class of the abstract hhgVolumeFunction, which must be provided by the user in the initFun attribute of the variable var before calling setupLoad. As a generic load function, hhgStringVolumeFunction can be used, which is able to evaluate a variety of mathematical equations, but which is rather slow. New derived classes of hhgVolumeFunction that evaluate the load function of the user's specific mathematical models can be implemented quite easily even with basic programming skills.

Algorithms

The numerical algorithms provided for the HHG user are organized in the C++ namespace hhgLinAlg. The namespace contains three classes: hhgLinAlg::Basic, the already mentioned hhgLinAlg::FiniteElements, and hhgLinAlg::Multigrid.

- Basic comprises basic linear algebra algorithms for
  - taking vector norms,
  - copying variables,
  - scaling, adding, and subtracting variables,
  - applying operators to variables.
- FiniteElements handles finite-elements-related tasks. Currently, the only function in this class is setupLoad.
- Multigrid contains all the multigrid algorithms, namely
  - smoothing,

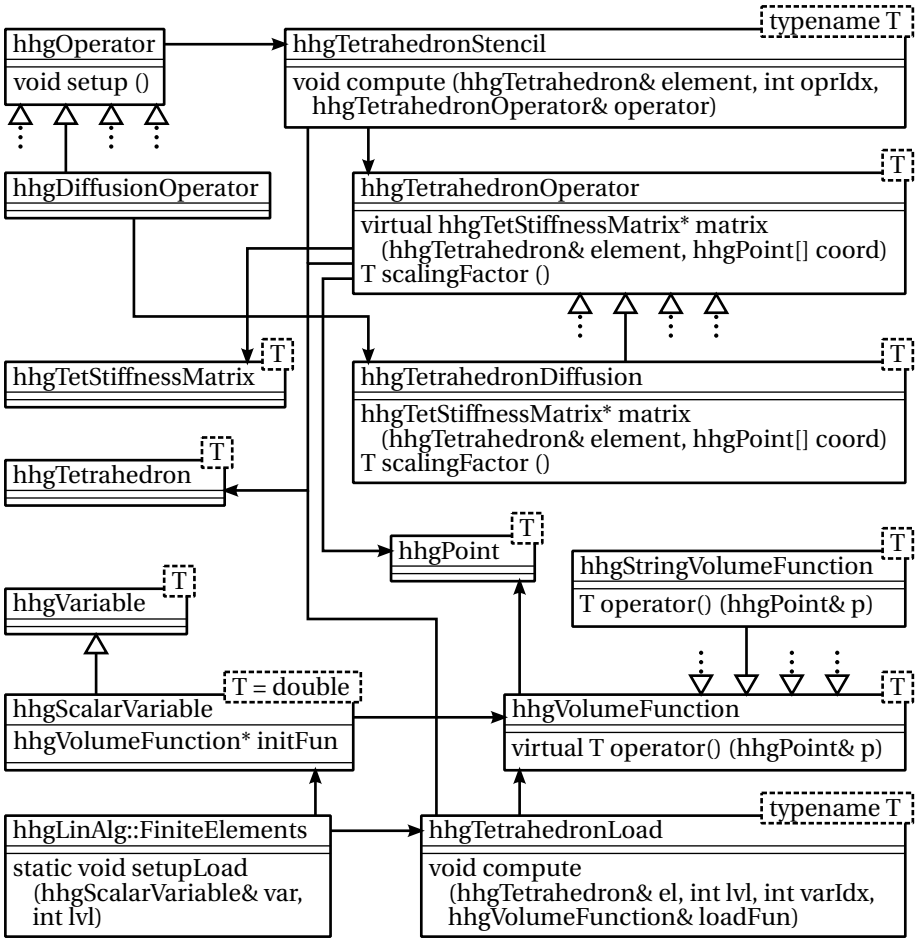

Figure 2.19: Finite element setup classes.

- residual computation,
- inter-grid transfer operations,
- multigrid cycles using the correction and full approximation schemes,
- the full multigrid algorithm.

The algorithms are provided with an `hhgMesh`, an `hhgOperator`, and one or more—depending on the algorithm—instances of `hhgVariable`. Besides that, several other parameters are usually necessary to define the algorithm's behavior. An example showing the usage of the `fullMG` algorithm is presented in Section 2.4.6.

2.4.5 Changes implemented within the scope of this thesis

The work with HHG that lead to the results presented in this thesis brought many changes to HHG's implementation. The major ones are summarized below.

- Additional multigrid algorithms were implemented, among them FAS and full multigrid.
- HHG's build system was migrated to SCons in order to support more supercomputer systems (see Chapter 3).
- For evaluating and optimizing HHG's performance on different supercomputers, the code was instrumented with a new timing framework (see Chapter 3).
- Adaptive refinement was implemented (see Chapter 4). This required changes in HHG's architecture, e. g., for supporting variables with integer data and new primitive types, the *hyperplanes*.
- An algorithm for optimizing the multigrid cycle structure at run-time was implemented, including mechanisms for persisting optimization data (see Chapter 5) between multiple program runs.

2.4.6 Usage example

Listing 2.2 shows a complete example program demonstrating the usage of the HHG library. The program executes the full multigrid algorithm with the full approximation scheme and a Gauss-Seidel smoother on three multigrid levels. The code is described in the following paragraphs.

All programs using HHG must include the `hhg.h` header (line 1). After the MPI initialization, the minimum and maximum multigrid levels are set in line 7.

Then, a 3D volume mesh is created and initialized. The initialization includes passing the name of the file that contains the input finite element mesh ("example.ugm", line 15) and setting a mesh partitioner (line 17). The partitioner, which has the task of assigning the finite elements of the input mesh to the available MPI processes, is, in this example, a very simple one (`uTestPartitioner`), which distributes the elements equally over the processes in the order of their IDs. Concluding the initialization phase, a refiner is added to the mesh. In conjunction with adaptive mesh refinement (Chapter 4), `hhgRefiner` will be explained in more detail; for now, `hhgUniformRefiner` is used, which refines all elements to the same level.

After the mesh has been initialized, variables and operators can be created. The full multigrid algorithm needs three variables: the unknowns vector (`unk`), the right-hand side (`rhs`), and the residual (`res`). All variables are created as `hhgScalarVariable` objects. Both the unknowns and the right-hand side get an initialization function, but HHG's usage of these functions is different for the two variables. The initialization function passed to `unk` (line 27) will be used by the library to initialize the Dirichlet boundaries. The initialization function of `rhs` (line 31), on the other hand, will be used to compute the load vector as explained above and shown in Fig. 2.19. The Dirichlet boundaries are initialized with the function $u(x,y,z) = x \cdot y \cdot z$, the right-hand side is set to a constant value of 0. In combination with the diffusion operator created in line 35, the PDE that will be solved is

$$\Delta u(x,y,z) = 0 \text{ in } \Omega \text{ with } u(x,y,z) = x \cdot y \cdot z \text{ on } \partial\Omega\,,$$

where $\partial\Omega$ is the Dirichlet boundary of Ω.

Some more parameters for the full multigrid algorithm and the underlying multigrid cycles are set in lines 37–48. The number of multigrid cycles performed by the full multigrid algorithm on each level (`cycles`) is set to 1. The full approximation scheme (`fas`) is activated. The Gauss-Seidel smoother (`GS`) is specified, which does not take extra arguments (therefore, `smootherArgs` is not needed). No particular coarse-grid solver is selected, thus the Gauss-Seidel smoother will be used as coarse-grid solver by HHG. The number of smoothing steps on the coarse mesh is set to -10, which means, that after every 10 smoothing steps the residual is checked, until convergence is reached. The number pre- and post-smoothing steps on the finer meshes is set to 2, each. Multigrid cycle optimization (which will be covered in Chapter 5), is switched off.

The unknowns vector has to be passed to the full multigrid algorithm in an STL `vector` container, because some algorithms (e. g., the Jacobi smoother) need more than one unknowns vector. This is taken care of in line 50. Finally, the variable `resmax` is defined, which will contain the maximal residual value upon return from the `fullMG` function. This function is in the `Multigrid` class of the `hhgLinAlg` package; it uses the mesh variables, the operator, and the other parameters to execute a full multigrid algorithm. When the algorithm is finished, the unknowns vector is shown with the `hhgVariable::print` function.

Listing 2.2: HHG usage example.

```
1  #include <hhg.h>
2
3  int main (int argc, char *argv[])
4  {
5    MPI_Init (&argc, &argv);
6
7    lvl_t lvlMin = 2, lvlMax = 4;
8
9    hhgMesh *mesh = new hhgVolumeMesh;
10
11   mesh->setCoarsestLevel (lvlMin);
12   mesh->setFinestLevel (lvlMax);
13
14   mesh->setGeometryInterface
```

```
15      (new hhgUGLiInterface ("example.ugm"));
16
17    mesh->setPartitioner (new uTestPartitioner);
18
19    mesh->initialize ();
20
21    hhgRefiner *refiner = new hhgUniformRefiner (*mesh);
22    refiner->initialize ();
23    mesh->setRefiner (refiner);
24
25    hhgScalarVariable unk ("unk", *mesh, lvlMin, lvlMax);
26    unk.setInitFunction
27      (new hhgStringVolumeFunction<double> ("x*y*z"));
28
29    hhgScalarVariable rhs ("rhs", *mesh, lvlMin, lvlMax);
30    rhs.setInitFunction
31      (new hhgConstantVolumeFunction<double> (0.));
32
33    hhgScalarVariable res ("res", *mesh, lvlMin, lvlMax);
34
35    hhgOperator *opr = new hhgDiffusionOperator (*mesh, 0, lvlMax);
36
37    unsigned cycles = 1;
38    bool fas = true;
39
40    hhgSmoother smoother = GS;
41    double *smootherArgs = 0;
42
43    void *csolver = 0;
44    int nuC = -10;
45
46    int nu1 = 2, nu2 = 2;
47
48    bool cycleopt = false;
49
50    std::vector<hhgScalarVariable*> unks;
51    unks.push_back (&unk);
52    double resmax;
53
54    hhgLinAlg::Multigrid::fullMG
55      (unks, rhs, res, *opr, csolver, nuC, nu1, nu2, lvlMin, lvlMax,
56       lvlMax, cycleopt, cycles, smoother, smootherArgs, fas,
57       &resmax, 0);
58
59    unk.print (2, 4, hhgPrintBoundary);
60
61    MPI_Finalize ();
62    return 0;
63  }
```

Chapter 3

Towards petaflop performance

Contents

3.1 Introduction

The design of HHG has started in 2002 with the HLRB I computer in mind. HLRB I was the first national supercomputer operated by the Leibniz Rechenzentrum (LRZ) in Munich, Germany. The Hitachi SR8000 started serving supercomputing demands of research groups throughout Germany in the year 2000. Upon installation, HLRB I ranked fifth on the June 2000 TOP500 list of supercomputers[1] with a Linpack performance of 1.0 TFlop/s. HHG proved to scale very well on this system, which had 1344 CPU cores in total [10].

At the time of this writing, over 40 supercomputers are available that have a Linpack performance of 1 PFlop/sor more, and the number of CPU cores in the largest computers is passing a million[2]. The exponentially increasing performance of supercomputers can only be exploited by software that can cope with this enormous degree of parallelism, which is increasing exponentially, as well. How does HHG cope with this development? Does it still scale well on such large numbers of CPUs? Section 3.4 answers these questions with performance measurements on computers with up to 16384 CPU cores.

[1] http://top500.org/list/2000/06
[2] http://top500.org/list/2014/06

A performance measurement can be as simple as taking the run-time of a program, from start to finish. If the performance is not as good as expected, however, the software engineer needs to perform more detailed measurements in order to find the bottleneck. Massively parallel programs provide a special challenge, here, because if one process in thousands takes longer, it will slow down all other processes. Therefore, analyzing the performance of software for high performance computing (*HPC*) requires special tools. The tool employed to analyze HHG's performance for this thesis is presented, along with some other state of the art tools, in Section 3.3.

Installing software on supercomputers is another task that requires special tools. These computers usually have special hardware for which the software needs to be tuned, and the operating system and software tools may differ significantly from what is standard on personal computers. HHG's build system proved to make the installation on various different supercomputers very convenient. Section 3.2 describes the key factors of a build system that is well-suited for HPC software.

3.2 Software engineering

HPC software is required to be portable across different computer architectures and operating systems. For example, we observe a much larger variety in operating systems than in the segment of workstation computers. The requirements for the portability of HPC software are different from those of workstation software, though. As an example, examine the task of installing the software on the computer, one among many aspects of portable software engineering.

The installation of workstation software, like office applications, mathematical toolkits for scientists, etc., has to be entirely automatized, because the distributor cannot assume much about the users' skills in dealing with computers and operating systems. Therefore, all the parameters of the host system, like installation location and system libraries, have to be detected automatically. On HPC systems, in contrast, it is often hard—if not impossible—to determine all these parameters automatically. Usually, automatism at that point is not even desired by the user. Take the choice of the compiler as an example. HPC systems usually offer several different compilers. The users will want to choose the correct compiler themselves instead and even experiment with different choices.

Another aspect in which workstation and HPC software differ is configurability. Behavior and range of features of workstation programs are rather fixed. They provide all their features to all users, may the individual user need them or not, and in these precincts that is not even a bad strategy. In the HPC environment, however, configuring an application one or the other way can have a large impact on its performance. Should the software perform error checks? That may be necessary, but it will slow down the program. Should it use shared- or distributed-memory parallelism, or a combination of both? That does not only depend on the hardware, and blindly using both may make the program run slower than including only what is necessary.

To simplify the process of building and installing large software packages, which is equally important for programmers and users, special tools have been developed. These tools, denoted *build systems* in this thesis, have to meet the above requirements, and many others, in order to be useful. The most used build systems are described in Section 3.2.1. They have all been designed with the task of building workstation software in mind. Therefore, they are not perfect for building HPC software.

Section 3.2.2 presents a build system designed especially for HPC applications.

3.2.1 Available build systems

The most common build system is the software suite consisting of *autoconf*, *automake*, and *libtool*, which are commonly subsumed under the term *Autotools* [21]. For the sake of simplicity, this toolkit will be referred to as *autoconf* in this thesis. Two more build systems which are quite popular are *CMake* and *SCons* [34, 17].

The Autotools suite has become the quasi-standard build system, and it was initially also used for HHG. It is widely used also in HPC applications. Programmers and users can expect it to be available on the login nodes of any compute server. Therefore, it is a safe bet for developers of new software to stick to the Autotools. The toolkit has built-in support for many standard tasks. The output is a description, the so-called *Makefile*, that can be used by the *make*[3] tool to compile the source-code into an executable. That being said, creating a build-system with functionality deviating from the standard tasks that were thought of by the developers of the Autotools is cumbersome. Extensions can be implemented as macros in the *M4*[4] language, which has a rather restricted functionality. For most tasks that require interaction with the operating system, external tools—which may not be available on all computers—have to be started, which makes it hard to create a build process that is portable onto various operating systems and computer architectures.

The CMake build system is implemented in C++. It defines its own language for describing the build process. Like for the Autotools, the output is a *Makefile*. CMake is a relatively new build system, therefore it is not as widely-spread than the Autotools. If developers use CMake for their software projects, they have to consider that the users may not be able to immediately install the software on their computers, but may first have to install CMake.

SCons, the build system used by HHG, is implemented in the programming language *Python*[5]. Python has built-in support for virtually any task related to interaction with the operating-system, so it is not necessary to resort to external tools for implementing special features of the build system. Like CMake, Scons is much newer than the Autotools, and still under development. Therefore, not as many standard tests for libraries, operating system parameters, etc., as in the Autotools are implemented, yet. Another consequence is that, like for CMake, the user of an HPC system cannot expect to find an SCons installation readily available on the login nodes.

3.2.2 Adapting SCons for HHG

The preceding paragraphs have already given an idea why SCons might be preferable for building scientific software, but some disadvantages of SCons have been mentioned, too. Thus, it is necessary to clearly point out the arguments for switching from HHG's existing build scripts to SCons.

Installing HHG on the JUGENE supercomputer [28] made clear that a high performance file system does not necessarily have to fulfill all the properties one would naively affiliate with the term "high performance". The *GPFS* file system is designed for fast throughput of very large files on parallel computers, but it does not handle

[3] `http://www.gnu.org/software/make/`
[4] `http://www.gnu.org/software/m4/`
[5] `http://www.python.org/`

the creation of many small files in short time very well. The latter is exactly what a build system attempts to do when running all the checks. It creates a small test program and tries to execute it. This is done for, usually, dozens of checks. Because of this, on JUGENE the run time of the autoconf script was around half an hour instead of, as usual, a few minutes.

Another problem with these checks is that they would ideally have to be run on the compute nodes of the HPC system, but not on the login nodes. Hardware and operating system are often entirely different on the login and compute nodes. Thus, checking for a certain property a the login node is useless; all the checks would have to be submitted to the computer's job queuing system in order to be run on the compute nodes, which would delay the configuration process even more. These difficulties with the automated checks suggest to abstain from them entirely. Not relying on them has the additional advantage of becoming independent from autoconf. The aforementioned disadvantage of SCons, that it does not have as many standard checks as autoconf, is not important, any more.

Now that we are free in the choice of our build system, SCons wins for mainly one argument: Python. Being implemented completely in Python, SCons can easily be extended using that powerful language. Build scripts for scientific software need many features that are not already available in the standard portfolio of the build systems. For this application domain, simple and powerful extension mechanisms are an invaluable asset of a build system.

Besides all the theoretical arguments for SCons that have been mentioned, the last, but not least important one is practical experience. All three of the considered build systems have been used in large software projects at the System Simulation Group in Erlangen. SCons is the one that users were most satisfied with, for the above arguments, but also for the small details that are important in day-to-day work.

The two characteristic properties of the HHG build system are, first, the need to specify the environment parameters of the system manually, and, second, the possibility to easily maintain different variants of the software. The first property is a natural result of the above discussion: it is nearly impossible to determine all the environment parameters automatically on an HPC system. It turns out, though, that the number of parameters that have to be specified is small, compared to some workstation software packages. The user has to specify the names of the compilers, the optimization flags that should be used, and the locations of some libraries. The biggest chunk are the compiler-related settings. To facilitate this task, the settings are grouped into a Python `map` data structure, and a function that defines the settings exists for each architecture. An example for such a function is shown in Listing 3.1. Functions defining specific settings for some architecture can, of course, make use of functions defining more general settings. Listing 3.2 shows the function setting the parameters for a GNU Linux system running on an Intel Core 2 processor and using version 4.2 of the GNU C++ compiler. The function `gnu_linux_core2_gcc4_1` uses the general settings for GNU Linux defined in `gnu_linux` and specifies only the settings specific to Core 2 and GCC 4.2.

The need to build different variants is also common for HPC software. For example, a program may support both shared- and distributed-memory parallelism. When the program is deployed on a system that requires only distributed-memory parallelism, the shared-memory support of the program may deteriorate the performance, even if it is not used. In this case, it is beneficial to completely disable the shared-memory support already at compile time. SCons makes building differ-

Listing 3.1: General settings for GNU Linux systems.

```
def gnu_linux (env):
    env['CC'] = 'gcc'
    env['CXX'] = 'g++'
    env['FORTRAN'] = 'gfortran'
    env['LD'] = 'g++'
    env['MPICC'] = 'mpicc'
    env['MPICXX'] = 'mpic++'
    env['MPIFORTRAN'] = 'mpif77'
    env['MPILD'] = 'mpic++'
    env['CFLAGS_ARCH'] = ''
    env['CXXFLAGS_ARCH'] = ''
    env['FORTRANFLAGS_ARCH'] = ''
    env['LINKFLAGS_ARCH'] = ''
    env['CFLAGS_TUNE'] = '-O2'
    env['CXXFLAGS_TUNE'] = '-O2'
    env['FORTRANFLAGS_TUNE'] = '-O2'
    env['LINKFLAGS_TUNE'] = '-O2'
    env['CFLAGS_DEBUG'] = '-O0 -ggdb'
    env['CXXFLAGS_DEBUG'] = '-O0 -ggdb'
    env['FORTRANFLAGS_DEBUG'] = '-O0 -ggdb'
    env['LINKFLAGS_DEBUG'] = '-O0 -ggdb'
    env['FORTRAN_NAME'] = 'lcname ## _'
```

Listing 3.2: Specific settings for Intel Core 2 and GCC 4.2.

```
def gnu_linux_core2_gcc4_2 (env):
    gnu_linux(env)
    env['CFLAGS_ARCH'] = '-march=nocona'
    env['CXXFLAGS_ARCH'] = '-march=nocona'
    env['FORTRANFLAGS_ARCH'] = '-march=nocona'
    env['LINKFLAGS_ARCH'] = '-march=nocona'
    env['CFLAGS_TUNE'] = '-mtune=generic -O2'
    env['CXXFLAGS_TUNE'] = '-mtune=generic -O2'
    env['FORTRANFLAGS_TUNE'] = '-mtune=generic -O2'
    env['LINKFLAGS_TUNE'] = '-mtune=generic -O2'
```

Listing 3.3: SCons code (simplified) for building different variants of HHG.

```
for target in COMMAND_LINE_TARGETS:
    if target == 'ser':
        build_root = build_prefix + '-ser'
        ugli_root = env['UGLI_SER']
        env_clone = env.Clone()

    elif target == 'par':
        build_root = build_prefix + '-par'
        ugli_root = env['UGLI_PAR']
        env_clone = env.Clone()
        env_clone['CC'] = env['MPICC']
        env_clone['CXX'] = env['MPICXX']
        env_clone['FORTRAN'] = env['MPIFORTRAN']
        env_clone['LD'] = env['MPILD']
        env_clone.Append (CPPDEFINES = ['DM_PARALLEL'])

    Export('env_clone')
    SConscript ('SConscriptLib', variant_dir=build_root+'/lib',
                duplicate=0)
```

ent variants easy. An existing configuration environment, like the one set up in Listings 3.1 and 3.2, can be *cloned* for each variant and then be supplemented with the variant-specific configuration. Listing 3.3 shows how a serial and an MPI-parallel variant of HHG are set up. It would even be straightforward to refine variants into sub-variants. For example, if shared-memory parallelism was added to HHG, the environment for the parallel variant could simply be cloned again, into sub-variants for shared-memory, distributed-memory, and mixed parallelism.

Summarizing the advantages of the HHG build system, two points are worth being highlighted. Most notable in every-day use are the reduced times for rolling out the software on high performance computers, caused by a reduced number of files that have to be written to the parallel file system. The second point, improved transparency, is especially important for users who do not want to dig into the internals of the build system. All build settings are defined explicitly in a few well-readable files.

3.3 Performance analysis

In a cooperation with the Future Technology Group at Lawrence Berkeley National Laboratory[6], HHG's performance on different computer architectures was analyzed in detail. Completing this analysis required a survey of available performance analysis tools, with the goal to identify the ones that are suited for the specific needs of analyzing HHG's performance. The findings are presented in the first part of this section.

As part of the project, also HHG's own performance measurement framework

[6] http://ftg.lbl.gov

was extended, and a toolkit for evaluating the performance measurements was created. The tools were successfully used in finding a performance bottleneck on one of the architectures, which was caused by a problem in the computer's operating system configuration. HHG's performance measurement framework and the performance evaluation toolkit are presented in the second part of this section.

3.3.1 State of the art

Three tools were assessed regarding their usability with HHG: *IPM*, *Vampir*, and *TAU*. The tools have to fulfill three main requirements in order to be useful for our purposes. The first, and most important one, is the portability to many platforms. In particular, SGI Altix 4700, Cray XT4, and IBM BlueGene/P had to be supported for this project, as these platforms were mainly used for evaluating HHG. Second, since multigrid algorithms have a recursive structure (completing a V-cycle on a fine mesh requires recursively completing a V-cycle on a coarser mesh), the performance analysis tools have to resolve and display recursive function calls in some way. The last requirement arises from HHG's software structure. Some functions consist of several parts that have to be timed individually. Timing of each of the parts must be triggerable separately. The effort for refactoring the complete software structure just for timing purposes would have been disproportionally high, let alone refactoring would affect the performance. Therefore, the performance analysis tools have to support timing of sub-function blocks.

The gathering of performance data can be divided into two main strategies: profiling and tracing. A profile sums up the values of recurring performance events, but does not store information about when the events occurred or how they were ordered in time. For example, if a program repeatedly calls two functions, a time profile of the program can tell how much time was spent in each function in total, but not how much time was spent in every single function call. A trace, in contrast, records the values of individual performance events, and stores their absolute position in time or their relative order of occurrence.

While a profile can also be created from a trace in a post-processing step by summing up the values of individual performance events, pure profilers are still necessary. A trace generates much more data than a profile—data that has to be stored in intermediate buffers in memory. These buffers can not be very large, in order not to hinder the execution of the program, and their contents have to be written to disk whenever they are full. These disk accesses occur at unforeseeable times and have a significant impact on the measured performance. Pure profile data volumes are usually so small that they do not have to be flushed to disk before the end of the program run. The following paragraphs will focus on three performance analysis tools and highlight their advantages and restrictions regarding our requirements.

Integrated Performance Monitoring (IPM)

The Integrated Performance Monitoring (*IPM*) tool is portable to all of the architectures mentioned above[42]. It creates profiles in the HTML format that can be viewed with an internet browser. The performance data is presented in a variety of summary charts. The focus of IPM is on profiling. The so-called "snapshot tracing" feature collects tracing data during user-specified time intervals, until a buffer of predefined size is filled. While IPM does not offer a natural way of dealing with recursive function calls, it allows for defining regions using the `MPI_Pcontrol` directive.

A string passed to this directive is interpreted as the region name, and the profiling output generated by IPM can be filtered by the region name. The region names can be set dynamically at program run time. This allows for the definition of recursive regions by including the recursion depth into the name (e. g., the level, for a multigrid algorithm). The `MPI_Pcontrol` statements can be placed at arbitrary locations in the code, enabling the performance measurement in sub-function regions. Thus, IPM fulfills all the requirements on a profiler that we stated initially. Its profiling features have proven highly useful in the performance analysis of HHG, but the tool lacks sufficient tracing support.

Vampir

Vampir[7] is a partly commercial tool. It is split into a freely available trace generator (called "vampirtrace") and a commercially distributed trace visualization GUI (called "vampir"). Like TAU, Vampir records traces and it generates and visualizes profiling information using these traces. At the time the performance analysis project was carried out, Vampir had been ported to SGI Altix 4700 and Cray XT4, but not to IBM BlueGene/P. Recursive function calls were partly supported by the GUI. When viewing the trace of a selected process, the time is assigned to the x axis of the two-dimensional display, and the recursion depth is assigned to the y axis. Filtering traces or profiles for a certain recursion depth is not supported, though. If this functionality is required (as in the case of HHG), manual instrumentation becomes necessary again. The vampirtrace API supports functions similar to IPM's `MPI_Pcontrol`, which was explained above. Using this API also allows for manually defining sub-function regions, which was the third of our requirements. The mature and versatile vampir GUI proved very helpful in supporting our performance analysis work, but due to the lack of BlueGene/P support it was not used as our main analysis tool.

Tuning and Analysis Utilities (TAU)

Tuning and Analysis Utilities (TAU)[33] is, like IPM, portable to the required architectures. TAU creates traces during program run time. It can generate profiles from the traces in a post-processing step, if desired. The graphical user interface (GUI) offers functions for visualizing both traces and profiles. TAU can also store the recorded data in formats readable by other tools, e. g., Vampir.

Due to complications during the roll-out of the tool on some of the computers that could not be resolved within the available time frame, we could unfortunately not use TAU for our performance measurements. The tool has strongly evolved, since, and the support for new computer architectures and systems is continuously being extended. In any assessment of performance analysis tools today, TAU has to be considered.

3.3.2 HHG's performance analysis toolkit

For basic profiling purposes, IPM was suited well and fulfilled our requirements. For tracing purposes, none of the considered tools fully satisfied our needs. Therefore, a toolkit that fulfills all the requirements was developed in order to support the performance study presented in Section 3.4. Its design concepts shall be outlined in the following.

[7] `http://www.vampir.eu/`

The toolkit consists of two components,

- a framework for recording timings and other parameters during program execution, and
- a set of tools for generating timing and profiling graphs from the recorded data.

Both components fulfill two important requirements. First, they allow for the instrumentation of sub-function blocks. Second, they are aware of recursivity, i. e., the recording and evaluation of measurements can distinguish between different levels of the same function (or sub-function block).

In comparison to other toolkits, HHG's toolkit also has some limitations. As the following paragraphs will explain, the code instrumentation is not done automatically, but the measurement points have to be inserted manually by the programmer. If used heavily, he measurement directives can also clobber the code to some extent, making it less readable. Besides that, a pure profiling mode is currently not available.

Measuring time and other parameters

Timing data is recorded using the available logging functionality. This component was already used previously for writing data into log files. The component also provides a feature to avoid measurement aberrations caused by flushing log data to disk: a memory buffer of configurable size can be reserved for the log data, and the flushing of the buffer can be triggered manually at points where it does not harm the measurements, e. g., between V-cycles.

The timing instructions have to be inserted into the code by the programmer. They are configurable with a name and a parameter that indicates the current recursion level. Each unique combination of these parameters defines a *monitor*. During program execution, the times measured for each monitor are written into the log file in a concise plain-text format.

The times are measured using the `gettimeofday` function, which is defined in the POSIX standard and, therefore, available on all systems. It returns the current system time with a theoretical accuracy of 10^{-6} s. Preliminary measurements on HLRB II showed that, taking the overhead of function calls into account, the resolution of this method is approximately 10^{-4} s in reality. It is straightforward to extend the timing framework to measure other parameters, besides the time. Currently, recording of CPU counters retrieved through the *Performance Application Programming Interface* (PAPI, [46]) is supported.

The detailed instrumentation of HHG results in quite a large number of timing directives. In order to keep the log file sizes moderate, the timing of individual components can be enabled via the SCons configuration. This has the additional benefit that unused timing directives do not add any overhead to the program's execution times. Comparing measurements with and without timing enabled proved that the timing framework, if configured reasonably, does not significantly alter the execution times.

Evaluating measurements

For the evaluation of the recorded measurements, three command-line tools are available.

Listing 3.4: Sample output (shortened) of the `profile` tool for a HHG run performing four V-cycles on 4096 processes. 57 monitors were enabled.

```
# 4096 processes, 57 timers
# 0: vcycle l7 presmooth
# 1: vcycle l7 prolongate
# 2: vcycle l7 residual
...
# 56: vcycle l4 residual

# 0: vcycle l7 presmooth
0 1.15306e-01 1.10866e-01 1.16576e-01 1.18248e-01
1 1.16738e-01 1.12770e-01 1.16662e-01 1.15959e-01
...
4095 1.21739e-01 1.23102e-01 1.18604e-01 1.16803e-01
4096 1.07166e-01 1.06388e-01 1.05857e-01 1.06970e-01 # avg
4097 9.21860e-02 9.20310e-02 9.08080e-02 9.16770e-02 # min
4098 1.48383e-01 1.40133e-01 1.45079e-01 2.21535e-01 # max
4099 4.38954e+02 4.35764e+02 4.33590e+02 4.38149e+02 # sum
```

The `profile` tool reads the log file of an HHG run and collects all the monitors that occur in the file. For each monitor, it creates a summary of the values reported by each process. For each monitor, the summary contains

- all values (from all processes, from all iterations) reported for the monitor,
- for each iteration, the maximum, the average, and the sum over all processes.

Note that not all items of the profile make sense for all kinds of measurements. The sum, for example, is not a useful item for timing data; however, HHG also has a monitor that counts the number of unknowns in each process's part of the mesh, and for this monitor the sum is the total number of unknowns in the simulation. An example of the output of `profile` is shown in Listing 3.4.

The output of `profile` can be used by the `scalingplot` tool to create scaling plots with *Gnuplot*. An output file generated by `profile` contains data for a single HHG run on a specific number of processes. `scalingplot` requires an additional file that lists the `profile` output files for different numbers of processes. With this input data, `scalingplot` can generate a graph like in Fig. 3.1. The plot shows the scaling of two monitors from 4 to 4096 processes. The lines follow the average time over all processes during one V-cycle; error bars indicate the minimum and maximum values measured.

The third tool, `histplot`, is useful when measurements of different processes shall be compared. In tightly coupled algorithms, like the V-cycle, it is critical for the performance that all processes take about the same time for the same step. If only one process takes longer, all others have to wait before starting with the next step. This means, one outlier will slow down the whole program. Exactly this problem occurred when testing HHG on a Cray XT4 computer (the same system as HECToR, but with a slightly different configuration). Due to a misconfiguration of the system library that is responsible for managing RAM pages, the residual calculation on

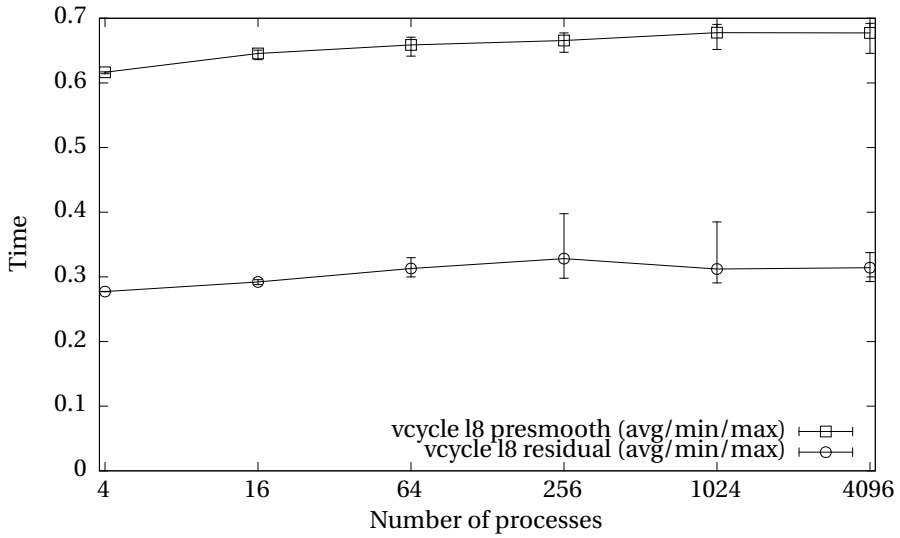

Figure 3.1: Sample output of the `scalingplot` tool.

a few processes was 25 % slower than average. This caused the measured performance to be significantly lower than expected. A plot created by the `histplot` tool immediately unveiled this artifact. An excerpt of the plot is shown in Fig. 3.2. The x-axis has been trimmed for demonstration purposes; the complete plot includes the measurements for all processes. The stepped line shows the time of the "vcycle l8 residual" monitor measured on each process. In addition, horizontal lines show the average, minimum, and maximum over all processes. The graph shows at first glance that there is a problem with outliers. Zooming into the plot allowed for precisely identifying the process IDs of the outliers.

3.4 Performance of HHG on different architectures

A project funded by the *Distributed European Infrastructure for Supercomputing Applications* (DEISA) gave us the chance to test HHG on three of Europe's leading supercomputers [29]. This section contrasts the characteristics of the evaluated computer architectures, describes the measurement setup that was used in the study, and compares the performance and scalability of HHG on the three computers.

3.4.1 Architectures

In order to see the influence of hardware parameters on HHG's performance, measurements were carried out on three computers with quite different architectures. The SGI Altix system *HLRB II* at the Leibniz Supercomputing Centre in Munich provided the largest amount of memory, both per core and in total. The Cray XT4 system *HECToR* at the Edinburgh Parallel Computing Centre had the most powerful processors. The IBM BlueGene/P system *JUGENE* at the Jülich Supercomputing Centre provided the largest number of cores. The following paragraphs describe the other two computers in detail. Table Table 3.1 summarizes their key facts.

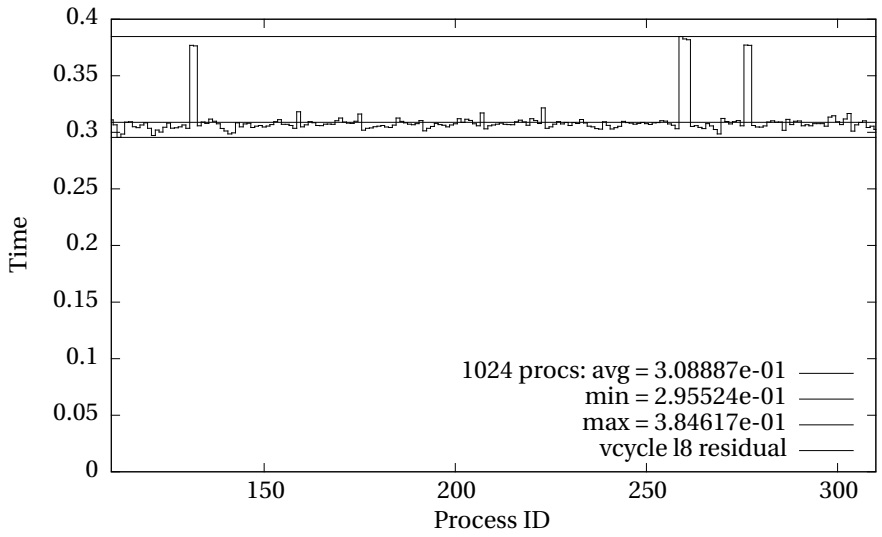

Figure 3.2: Sample output (x-axis trimmed) of the `histplot` tool.

Table 3.1: Key facts of the evaluated computers, given for the installation phases evaluated in this thesis: phase 2 (from 04/2007) of HLRB II, phase 1 (until 06/2010) of HECToR, and phase 1 (until 03/2009) of JUGENE. The data is taken, if not stated otherwise, from the references cited in the text.

Property	HLRB II	HECToR	JUGENE
Model	SGI Altix	Cray XT4	IBM BlueGene/P
Type of CPU	Intel Itanium 2	AMD Opteron	IBM PowerPC 450
Clock frequency	1.6 GHz	2.8 GHz	850 MHz
# of cores per proc.	2	2	1
# of cores in system	9 728	11 328	65 536
Theor. perf. per core	6.4 GFlop/s	5.2 GFlop/s	3.4 GFlop/s
Theor. perf. of system	62 TFlop/s	59 TFlop/s	222 TFlop/s
Linpack perf.[8]	56 TFlop/s	54 TFlop/s	180 TFlop/s
Memory per core	4 GB	6 GB	2 GB
Memory per SMP node	8 or 16 GB	6 GB	2 GB
Memory of system	39 TB	33.2 TB	32 TB
Network type(s)	fat tree, 2D mesh	3D torus	3D torus, tree
Point-to-point bandw.	6.4 GB/s	2.2 GB/s	5.1 GB/s
Bisection bandw.	<1.9 TB/s[9]	≥4.1 TB/s	*data missing*

HLRB II

The installation of HLRB II was completed in two phases. Phase 1, described in detail in [4], had a total of 4096 CPUs and 17.2 TB of main memory (*RAM*). The computer was comprised of 16 shared-memory partitions, each holding 256 node blades. Each blade hosted a 1.6 GHz single-core Intel Itanium 2 CPU with a theoretical peak performance of 6.4 GFlop/s, as well as 4 GB of RAM.

In 2007, HLRB II reached its final assembly stage with installation phase 2 [5]. The measurements presented in this thesis were performed on this installation. The single-core CPUs were replaced by dual-core Intel Itanium 2 CPUs. The clock frequency did not change, so every core of the new processors still had a theoretical peak performance of 6.4 GFlop/s. A subset of the blades was equipped with two CPUs (i. e., four cores). A total of 9728 cores were available on HLRB II in phase 2. The *high density* blades, hosting two CPUs each, were upgraded to 16 GB RAM, the *high bandwidth* blades with one CPU each were equipped with 8 GB. Thus, each core still had access to 4 GB of local memory, and the total amount of RAM in phase 2 was 39 TB. In terms of available memory per core, HLRB II was hardly matched by any other computer of comparable size. Also in the total amount of memory, both installation phases of HLRB II were outstanding at the respective times of commissioning. Even computers ranked higher in the TOP500 lists of November 2006 and June 2007 due to higher Linpack performance numbers did not beat HLRB II in memory size.

Besides the amount of memory per core, also the amount of memory accessible in a single shared-memory address space was exceptionally high. HLRB II was equipped with a *NUMALink 4* network. Within a partition, the routers provided a fat-tree topology. The inter-partition network had a mesh topology. The network synchronized all the memory within a partition in a cache-coherent, non-uniform memory access (*ccNUMA*) style, allowing applications to use up to 2 TB of RAM in a purely shared-memory parallel programming style, i. e., without having to resort to MPI parallelization.

HECToR

HECToR was installed in three phases. Each of the installations used AMD Opteron processors, but the number of cores per processor increased from 2 over 12 to 16. The measurements presented in this thesis were performed on the first installation, "Phase 1" [32]. Its compute power came from AMD Opteron dual core processors that were clocked at 2.8 GHz and had a theoretical peak performance of 5.2 GFlop/s per core. The installation was organized into 60 cabinets, which held 1416 compute blades with 4 processors, each. Thus, the total number of cores amounted to 11328, which lead to a theoretical peak performance of 59 TFlop/s for the entire system.

Each processor controlled its own memory chip, which means that sharing memory between processors on the same blade—as on HLRB II—was not possible. Each processor had 6 GB of memory, which sums up to 33.2 TB for the whole system.

The processors each had their own network router, which connected them to a 3D-torus network. The system's point-to-point bandwidth was 2.17 GB/s, its bisection bandwidth was 4.1 TB/s.

[8] According to `http://www.top500.org/system/174855` (HLRB II), `http://www.top500.org/system/175159` (HECToR), and `http://www.top500.org/system/176501` (JUGENE).

[9] Hager et. al. report a bisection bandwidth of 0.8 GB/s per direction, per socket [26].

JUGENE

JUGENE's lifetime split into two phases. The system commenced operation with 16 racks; for the second phase it was upgraded to 72 racks. The measurements presented in this thesis were performed on the initial installation [28]. The system employed single core IBM PowerPC 450 processors clocked at 850 MHz, which had a theoretical peak performance of 13.9 GFlop/s each. The resulting number of 16384 processors (respectively: cores) in total helped the system to a theoretical peak performance of 223 TFlop/s.

Each compute node was equipped with 2 GB of memory, thus, the amount of memory available to the complete system was 32 TB.

A specialty of JUGENE was that it had two networks. A 3D-torus network provided a high bandwidth of 5.1 GB/s for point-to-point communication. A second network connected the cores in a tree structure, which allowed to execute computations involving collective communication (e. g., vector norms) with very low latency.

3.4.2 Measurement setup for scaling tests

The details of the finite element simulation used for the scaling experiments were chosen to represent a compromise between a realistic setup resulting from a real-world problem and, the other extreme, a setup tailored to get maximum performance out of HHG. A simulation designed for the highest possible performance, in terms of the number of unknowns solved for per time interval, is characterized by a low communication-to-computation ratio and by a mesh layout that subserves an efficient execution of the necessary communication and computation. The desired compromise is achieved by appropriately choosing the PDE to be solved, as well as the finite element mesh on which the PDE is solved.

Model problem (2.1) is a good choice for the PDE. On the one hand, as a linear, scalar, elliptic PDE with Dirichlet boundary conditions it can be solved efficiently with multigrid methods. On the other hand it appears as a building block in numerous more advanced equations, e. g., in acoustics simulations.

The choice of the finite element mesh influences all the important simulation characteristics listed above. The first important decision is to perform a *weak-scaling* experiment, in which the problem size is proportional to the number of processes in every run of the scaling series. A *strong-scaling* experiment, in contrast, would use the same problem size for varying number of processes. The results of weak- and strong-scaling experiments usually differ significantly, but also the intentions are different. The strong-scaling experiment answers the question, "by how much can I speed up the solution of a *particular* problem by running it on a parallel computer?" The speed-up will be good at a moderate degree of parallelism, but for larger numbers of processes the communication overhead will become dominant, and adding more processes will not reduce the execution times any further. The weak-scaling experiment answers the question, "how large a problem of the *same type* can I solve by using a computer with more CPUs?" The problem size is given by the number of unknowns, respectively mesh points. In a *strong-scaling* experiment, in contrast, the same mesh would be used for all runs. Both types of scaling experiments are valuable, and it depends on the application which of them to choose. In the context of HHG, the weak-scaling experiment is of more interest, because HHG's intention is to push the precision of physical simulations to new limits, and not to speed up the simulation of a particular problem with a given precision.

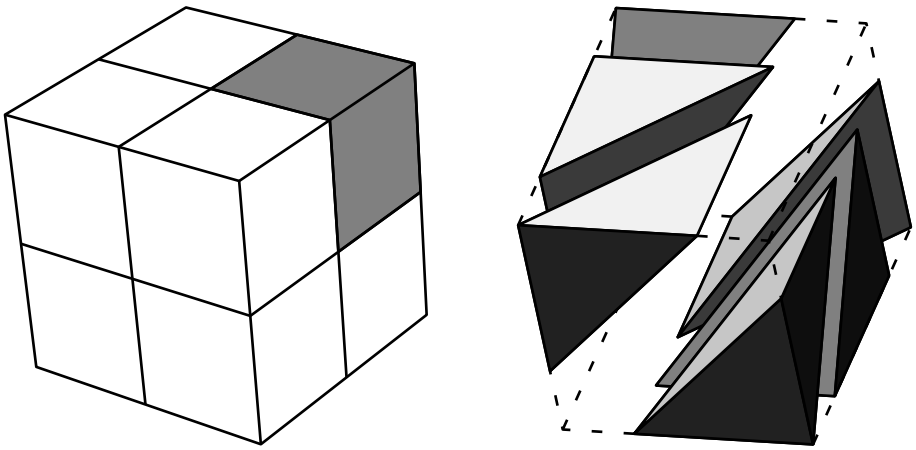

Figure 3.3: Finite element meshes for weak-scaling experiments. The left image shows a cubic domain that is split into eight sub-cubes. The right image illustrates how a sub-cube is split into six tetrahedra.

Conducting a weak-scaling experiment requires a series of meshes with increasing size. Our approach to construct such a series of meshes is simple, but nevertheless yields meshes that could as well come from a real-world simulation. It is illustrated in Fig. 3.3. A mesh for a run with p processes consists of p cubic sub-domains. Since HHG can currently only deal with tetrahedral meshes, each of the cubes is split into tetrahedra. The sub-domains are arranged to form a hexahedron that is shaped as close to cubic as possible by setting the number of sub-domains in each space direction to be close to $\sqrt[3]{p}$.

The described mesh creation technique leads to meshes with a low communication-to-computation ratio and, therefore, fulfills the requirements stated initially. The communication-to-computation ratio is defined by the amount of inter-process communication necessary for solving a given PDE on a given mesh, divided by the amount of computation. While the amount of computation is, in our case, accurately represented by the number of arithmetic operations executed, the cost of communication is determined not only by the transferred data volume, but also by the number of messages sent between the processes. At this point, we focus purely on the data volume. Multigrid can solve (2.1) with a number of arithmetic operations proportional to the number of unknowns, if the finite element mesh looks as described above [48, Chapter 3.2.2]. Therefore, the amount of computation can be assumed as proportional to the number of processes. The amount of communication is determined by the number of mesh points on the internal boundaries between the sub-domains. These points hold data that is needed by several processes adjoining the boundaries and, therefore, has to be communicated. Clearly, the arrangement of the sub-domains determines the area of the internal boundary faces. So, why not put them all in a row, so each cube has at most two neighbors, or even disconnect them entirely, eliminating all need of communication? This would, of course, contradict the goal of having a physically realistic simulation. We have to pick some reasonable shape, and we choose to cast the sub-domains into a shape as close to cubic as possible.

The cubic sub-domains are split into six tetrahedra each, because this is the

Table 3.2: Measurement setup for each computer.

Property	HLRB II	HECToR	JUGENE
Levels of refinement	8	8	7
Cubes per process	2	1	1
Unknowns per process	33.16×10^6	16.58×10^6	2.05×10^6

smallest possible number. It is beneficial for the computational efficiency of HHG to have an input mesh with a small number of elements per process. If there are few elements in the input mesh, they can be refined many times until the computer's memory is filled. Then, the structured regions on the finest levels are large, and the data (unknowns, right hand side, etc.) can be stored in long, contiguous arrays in memory. The loops of the computational kernels (smoothing, interpolation, etc.), which traverse these arrays, have many iterations, and optimization techniques like loop unrolling, pipelining, and SIMD vectorization are very effective. This leads to a Flop/s rate close to the processor's peak performance, and the arithmetic operations, called "amount of computation" above, can be worked off quickly.

Table 3.2 summarizes the measurement setup key facts for all computers. On each computer the configuration was chosen such that the simulation fills as much as possible of the computer's RAM. The variables (unknowns, right-hand side, residual, and, optionally, error) contribute most to the simulation's memory consumption; the memory needed for the operator stencils is negligible. The main control to influence the memory consumption is the number of refinement levels. With each level, the memory demand increases by a factor of eight, because the number of mesh points doubles in each spatial dimension. Another tuning parameter is the number of cubes per process. It allows for more a precise adjustment of the memory demand. However, it also changes the geometries of the process sub-domains and the number of inter-process boundary primitives, and, thus, the communication-to-computation ratio.

On JUGENE seven levels of refinement were possible, resulting in 2.05×10^6 unknowns per process. On HECToR, due to the larger memory per process, the mesh could be refined one level further, resulting in 16.58×10^6 unknowns per process. HLRB II has even more memory per process than HECToR, but not enough to go to nine levels of refinement. It is possible to increase the number of cubes per process to two, though, so the number of unknowns per process increases to 33.16×10^6.

On JUGENE and HECToR a higher number of unknowns per process could probably have been achieved by experimenting further, trying other combinations of number of levels and number cubes per process. Experiments are inevitable, here, because the real memory demand of the actual application running on the computer cannot be predicted precisely. Even though the memory required for storing the variables and operators can be calculated, the actual memory demand of the complete application will be higher due to, for example, overheads of the communication library. Besides that, the memory occupied by the operating system is not known exactly. The experiments necessary to determine the optimal combination of parameters are rather time-consuming. For the measurements on HLRB II that effort was spent, because the main goal was to set a new record in number of unknowns solved with the finite element method [24]. That was not the focus of the tests on HECToR and JUGENE, however. On these computers the goal was to show

that HHG can easily be deployed onto a variety of systems, and that it achieves good scalability out of the box, i. e., without modifications of the implementation. For showing that, it was not necessary to drive the simulation's size to the absolute maximum.

On HLRB II, the scaling runs were performed up to the complete system size. On HECToR, the CPU core count was increased in powers of two between scaling runs, because this simplifies the creation of the corresponding finite element meshes. Therefore, the largest run was done with 8192 cores. On JUGENE, measurements for up to 16384 cores are reported. At the time of the study, a part of HHG's initialization phase—the distribution of elements to processes and the identification of inter-process boundaries—was not parallelized well. Every process had to work on the complete input mesh, and the algorithm's complexity was quadratic in the number of elements in the input mesh. On HLRB II and HECToR that was not a problem. On JUGENE, however, with eight times the number of cores, and a comparably low performance of the individual cores, the duration of the initialization became infeasibly long—by magnitudes longer than the solution phase. In the meantime, this weakness has been eliminated, and HHG has been run at much larger scales [20].

3.4.3 Results

Table 3.3 shows the results of the weak scaling experiments on HLRB II, HECToR, and JUGENE. In case of perfect scalability the times per V-cycle stay constant for each computer, because the amount of computation per process stays constant. Perfect scalability is, of course, not achievable for an algorithm that involves communication—especially, collective communication (for computing the residual norms). So, let us have a look how close to perfect scaling HHG is on each computer. For practical purposes, good scalability is nice, but what really counts at the end of the day is whether the simulation has finished or not. Therefore we also analyze how many unknowns HHG can solve for per second on each computer.

Parallel scalability

Fig. 3.4 depicts HHG's parallel scalability $S(p)$ vs. the number of processes p. The parallel scalability is defined as the time per V-cycle on p processes divided by the time per V-cycle on p_0 processes,

$$S(p) = \frac{t(p)}{t(p_0)}. \tag{3.1}$$

The reference point p_0 can be chosen arbitrarily. We set p_0 to 4, the smallest number of processes for which timings were taken on all computers. The scaling diagram reveals significant differences between the computers.

The simulation on JUGENE, on first glance, seems to have a severe scalability problem. It is not as bad as it looks, though. First of all, on the maximum number of processes the V-cycle takes only 56 % longer than on four processes ($S(16384) = 1.56$), which is quite good, considering the extreme span in the number of processes. Second, the scaling is deteriorating quickly only until 128 processes. From there on, S remains almost constant: $S(16384)/S(128) = 1.07$. On HECToR and HLRB II the scalability is even better, and it follows a more linear curve. At 8152, respectively 8192, processes, the scaling is approximately 1.16 for both computers. When we used the full HLRB II system (9170 processes), the scaling jumped to 1.21. HHG's

Table 3.3: Results from performance measurements on three computers. For each run, the table shows the number of unknowns and the run-time of one V-cycle in seconds.

Cores	HLRB II		HECToR		JUGENE	
	Unk. $\times 10^6$	Time	*Unk.* $\times 10^6$	Time	*Unk.* $\times 10^6$	Time
4	*134.2*	6.38	*66.6*	2.72	*8.3*	3.05
8	*268.4*	6.67			*16.6*	3.33
16	*536.9*	6.75			*33.2*	3.48
32	*1 073.7*	6.80			*66.6*	3.67
64			*1 070.6*	3.00	*133.4*	4.04
128	*4 295.0*	7.06			*267.1*	4.44
252	*8 455.7*	7.39				
256			*4 286.6*	2.97	*534.8*	4.46
512					*1 070.6*	4.60
1024			*17 158.9*	3.12	*2 142.2*	4.60
2048					*4 286.6*	4.56
4096			*68 669.2*	3.09	*8 577.4*	4.60
8152	*273 535.7*	7.43				
8192			*137 355.0*	3.13	*17 158.9*	4.70
9170	*307 694.1*	7.75				
16384					*34 326.2*	4.75

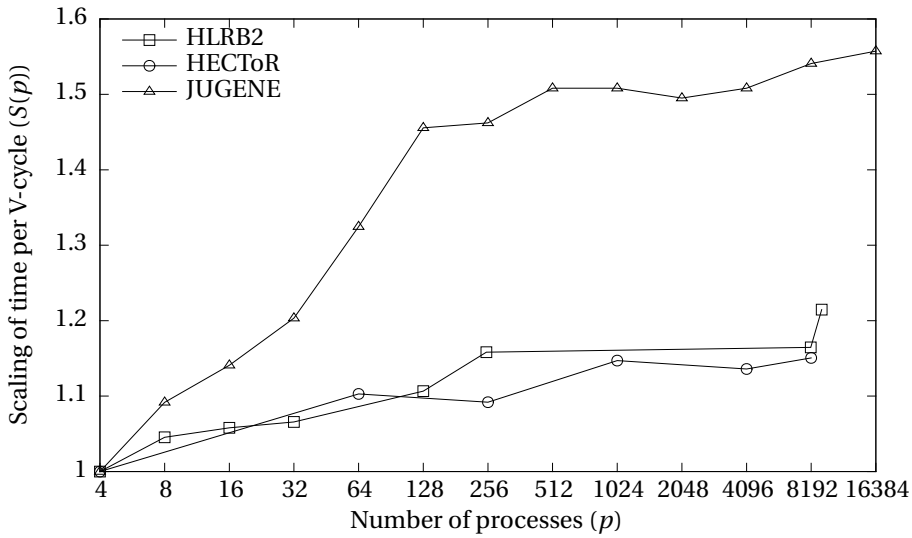

Figure 3.4: Scaling of V-cycle run-times.

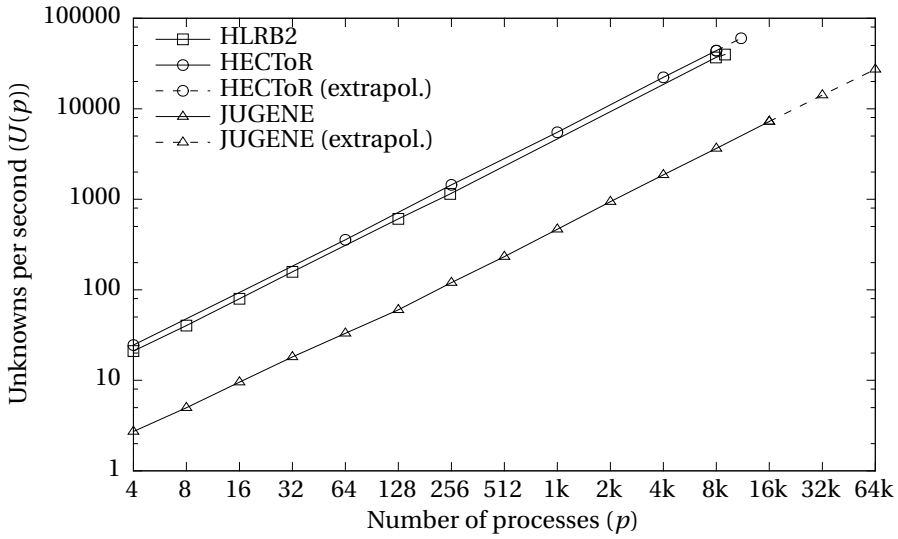

Figure 3.5: Unknowns processed by the V-cycle per second vs. number of processes. The unit k in the x-axis labels stands for a factor of 1024. Extrapolations for HECToR and JUGENE show theoretical values for the complete systems.

timing methods were not detailed enough at the time of the experiment to find out the root cause of the jump. It is likely, though, that the operating system was not configured optimally, because HLRB II needed to be re-configured especially for this measurement; in normal operation, only 8152 cores were available.

Unknowns per second

Fig. 3.5 puts the run-times $t(p)$ of the V-cycles into relation with the numbers of unknowns $N(p)$ in the linear system by plotting the ratio

$$U(p) = \frac{N(p)}{t(p)} \tag{3.2}$$

vs. the number of processes p.

In the comparison of the three computers, HECToR comes out first, with a small margin of 11 % in front of HLRB II, when comparing $U(p)$ for same numbers of p. We should not forget, though, that HECToR was larger than HLRB II. Since a run on the complete HECToR system was not performed, the theoretical performance on 11328 processes is shown by extrapolating the scaling from 4096 to 8192 processes. Wit that value ($U(11328) = 6 \times 10^{11}$) the computer is clearly ahead of the others.

For equal numbers of p, JUGENE is clearly behind the other computers. That is not surprising, though, because the PowerPC processor has a much smaller Flop/s rate than the Itanium or Opteron processor. However, JUGENE has a much larger total number of cores. Extrapolating the performance measured on 8192 and 16384 cores shows that the full JUGENE system would be on a similar performance level as one the other two systems. Measurements of HHG on up to 294912 cores of JUGENE have been reported in [20].

Chapter 4

Adaptive mesh refinement

Contents

4.1 Introduction

Just as important as solving for many unknowns in short time is reducing the number of unknowns. *Adaptive mesh refinement* (AMR) can significantly reduce the problem size in many cases. The technique is known to impede the use of regular data structures, though, which drastically restricts the possible performance of the solver. This chapter shows how AMR is implemented in HHG without destroying the regularity of the data structures, thus maintaining its computational efficiency.

The term *adaptive mesh refinement* refers to techniques for adjusting the mesh resolution to the properties of the problem that has to be solved. A physics simulation may require a higher mesh resolution in places where the solution is not as smooth as in other areas. Typical examples are fluid simulations, which need

high mesh resolutions where vortices or shocks are formed. The solver has to provide *spatial adaptivity* for dealing with these problems. If the required adaptivity is known a priori to the computation, pure spacial adaptivity is sufficient. However, often the areas that require higher mesh resolutions are not known in advance. In time-dependent simulations these regions may even move over time (moving vortices are an example). In these cases, *temporal adaptivity* is necessary. Adaptivity in space and time are orthogonal, as they can be used independently, but they can also be combined.

Related work

For more details on AMR we refer to an introductory article about multigrid on adaptively refined meshes by Bastian and Wieners, which includes references to related work [8]. A paper by Lang and Wittum describes the building blocks of a parallel adaptive multigrid solver in detail [31]. An in-depth analysis of the mathematical aspects of multilevel methods on adaptively refined meshes is available in a book by McCormick [35]. For both spatial and temporal adaptivity, error indicators deciding where and when to refine the mesh are necessary. There is extensive literature on the topic of error estimation, e. g., by Verfürth [49], Ainsworth and Oden [1].

Outline

This chapter presents the implementation of spatial adaptivity in HHG. First, Section 4.2 contrasts two popular refinement techniques, red-green refinement and refinement with hanging nodes, the latter of which is used in HHG. In the main part of the chapter, Sections 4.3 and 4.4, the theoretical foundation for implementing the full multigrid algorithm with adaptive refinement is constructed. Section 4.5 describes the HHG data structures that are necessary for realizing an efficient implementation. An efficient mesh setup algorithm and its implementation are also described in this section. Finally, Section 4.6 presents results from a practical example.

4.2 Refinement techniques

We have already met a non-adaptive mesh refinement technique in this thesis: the HHG mesh hierarchy introduced in Section 2.4 was created by refining an initial set of coarse finite elements. It was implicitly assumed that all elements are refined to the same level. If we give up this assumption, we have several possibilities of adaptive mesh refinement. Two of them are compared in this section. For simplicity, the discussion refers to 2D meshes, which is sufficient for showing all the relevant differences.

Refinement with hanging nodes

Instead of refining all coarse-grid elements to the same level, each element can be refined individually. An example is shown in the left part of Fig. 4.1. The mesh on the coarsest level consists of four triangles and one quadrilateral. All elements are refined once, except for the lower left triangle, which is refined twice. The effect is the desired one: the mesh width is adapted locally.

However, there is also an undesired effect: some of the corners of the triangles on the second refinement level are located on edges of other elements; they are

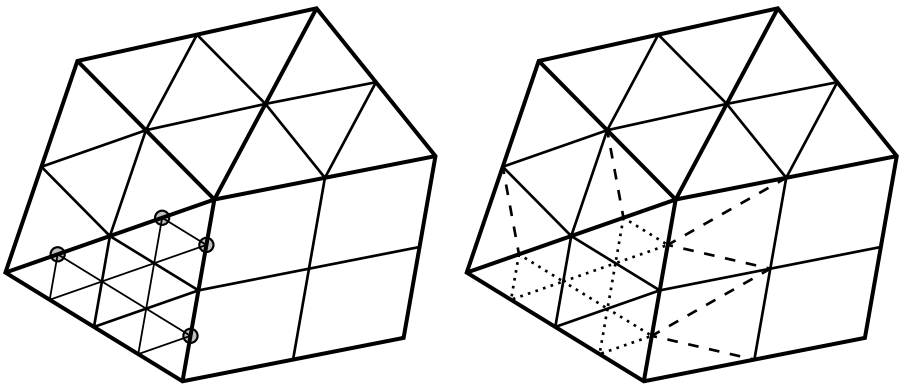

Figure 4.1: Refinement with hanging nodes (left) and red-green refinement (right)

marked with circles in the figure. Thus, the mesh has become *non-conforming* according to Definition 1. The problematic points are called *hanging nodes*. This artifact challenges the previous understanding of the finite element method. Assuming that the mesh nodes—i. e., the locations where the weights for the basis functions are defined—are located at the element corners, there are now locations at which a local basis function is only valid for one of the adjacent elements. How to deal with this situation will be discussed in Section 4.3.

Red-green refinement

In order to get rid of the hanging nodes, red-green refinement can be applied [7]. The two colors refer to the two steps that are necessary to restore the conformity of the mesh. The steps and the resulting mesh are illustrated in the right part of Fig. 4.1. The first ("red") step is simply the local refinement of individual elements described above. It is indicated by dotted lines in the figure. The resulting hanging nodes are taken care of by applying the second ("green") step, indicated by dashed lines. Each element that is adjacent to a hanging node is split into two or more elements by connecting the hanging node with one or more of the element's already existing nodes. After that, the mesh is conforming again. If further refinement is necessary, the red and green steps can be applied iteratively.

Two—potentially adverse—side-effects of this technique have to be mentioned. First, the type of the green-refined elements may change. For example, some of the quadrilaterals in Fig. 4.1 are converted to triangles. Thus, red-green refinement cannot be used, if meshes with only quadrilateral or hexahedral elements are desired. Second, if red-refinement were successively applied only to the lower left triangle, the triangles resulting from the green-refinement would quickly *degrade*. Some of their angles would become more and more acute, making the resulting linear system harder to solve.

Combining both techniques

While each of the two refinement methods can alone be used to implement adaptive refinement, combining them results in additional advantages. A solver that is able to perform red-green refinement as well as structured refinement with varying levels

can trade off the advantages and disadvantages of both methods to obtain an optimal compromise between adaptivity and performance. One of the goals when setting up a simulation for HHG is to have as few coarse grid elements as possible, because then the structured areas are large and can be treated with high performance. Refinement with hanging nodes creates less elements on the coarsest level than red-green refinement and should thus be preferred in the HHG framework. However, not exploiting red-green refinement can also lead to bad performance. For example, the domain may have only very few geometric features, so that it can be resolved with a very coarse initial HHG mesh. Then, if it turned out during the solution process that the mesh had to be refined only in a small part of an element on the coarsest level, for instance due to the characteristics of the solution, still the complete element would have to be refined regularly, leading to a fine mesh resolution also in areas where it is not needed. This can be avoided by red-green-refining the initial coarse mesh.

4.3 Full multigrid on meshes with hanging nodes

Refinement hierarchies

The introduction into adaptive mesh refinement techniques omitted some details that become relevant when thinking about multigrid methods. First of all, only one adaptively refined level was considered, up to now. Exploiting the full potential of a multigrid solver requires a hierarchy of adaptively refined meshes over several levels. Extending the refinement process to a hierarchy of levels is straightforward, though. Still, the implications for the hierarchical structure of the meshes are interesting, because they allow for some simplifications during the construction of the algorithms, later on. Thus, what is up to now only an idea of an adaptive mesh hierarchy is formalized in the following definition.

Definition 2 (Hierarchy of adaptively refined meshes)**.** *This definition assumes a hierarchy of regularly refined meshes*[1]*, i. e.,*

$$\Omega_{l-1} \subset \Omega_l \text{ for all } l > 0\,.$$

Let $\mathscr{C}_l$ denote the set of mesh points on which the solution is computed on level l. It is a subset of all mesh points on level l, excluding the Dirichlet boundary, i. e.,

$$\mathscr{C}_l \subseteq \Omega_l \cup \Gamma_l^N,.$$

In particular, $\mathscr{C}_l$ may be the empty set or the complete set Ω_l.

For two consecutive meshes, the area in which the solution is computed on the coarse mesh shall cover at least the area in which the solution is computed on the fine mesh, i. e.,

$$\mathscr{C}_l \cap \left(\Omega_{l-1} \cup \Gamma_{l-1}^N\right) \subseteq \mathscr{C}_{l-1} \text{ for all } l > 0\,.$$

Hierarchical bases

The ideas that have lead to the construction of the multigrid methods for uniformly refined meshes in Section 2.3 were

[1] The HHG meshes fulfill this property, see Section 2.4.1.

- fine-grid error smoothing,
- coarse-grid approximation of the fine-grid error, and
- coarse-grid approximation of the fine-grid solution (for FAS).

These concepts, and the multigrid methods built upon them, are also valid for adaptively refined meshes with hanging nodes. For an intuitive approach to this mesh type, an alternative view on adaptively refined meshes is endorsed. Technically, an adaptively refined mesh with hanging nodes can be interpreted as a uniformly refined mesh, but with limited degrees of freedom for the variables in the "unrefined" areas.

This dual point of view—well-known to those acquainted with *hierarchical bases* [25]—shall not be covered in depth, here, but the basic idea is outlined in Fig. 4.2. In mesh-based discretization techniques, continuous functions are approximated by weighted sums of locally defined basis functions. The weights are in the unknowns vector of the discrete algebraic system of equations, the basis functions influence the matrix entries and possibly the right-hand side. The basis functions appear very prominently in the finite element method (see Section 2.2), but also other methods, like the finite difference method, can be interpreted in that way. Fig. 4.2 compares two types of basis functions. The illustration shows a section of an equidistant one-dimensional mesh; the mesh boundaries do not have to be considered for a basic understanding. The top-most picture shows a coarse mesh with three nodes (labeled 0, 2, and 4). The continuous curve $f(x)$ is approximated by the piecewise linear curve $g_0(x)$, which is constructed by multiplying the depicted basis functions with appropriate weights (indicated with dashed lines). For approximating $f(x)$ on a finer mesh (with five mesh points, labeled from 0 to 4), two possibilities are depicted in the pictures below, both leading to the *same* approximation $g_1(x)$. The easiest way (shown in the left picture) is to use the same approach as for the coarse mesh and construct an entirely new set of basis functions adjusted to the new mesh width. The *hierarchical bases* approach (shown in the right picture), in contrast, re-uses the basis functions from the coarse mesh (shown in gray) and introduces new basis functions only at the new points of the fine mesh. The weights at the new nodes (indicated with dashed lines again) are now calculated relative to the approximation on the coarse mesh, $g_0(x)$: a weight of zero yields $g_1(x) = g_0(x)$ at the corresponding node of the fine mesh.

Because of this property, the concept of hierarchical bases is useful in the initial formulation of adaptive algorithms. The notion of an adaptively refined *mesh* with hanging nodes is not even necessary. The refinement instructions of the error estimator can be fulfilled on a standard—uniformly refined—mesh by distinguishing between two *sub-meshes* on the fine level:

(1) at those mesh points that do not need to be considered on the fine level, because the approximation to the solution on the coarse mesh is already good enough, the weights of the basis functions on the fine mesh are fixed to zero,

(2) all actual computations—i. e., changes to weights of basis functions—on the fine level are restricted to the remaining mesh points.

When working on the fine mesh, the approximation from the coarse mesh is still available in sub-mesh (1) and is automatically used by the computations in sub-mesh (2).

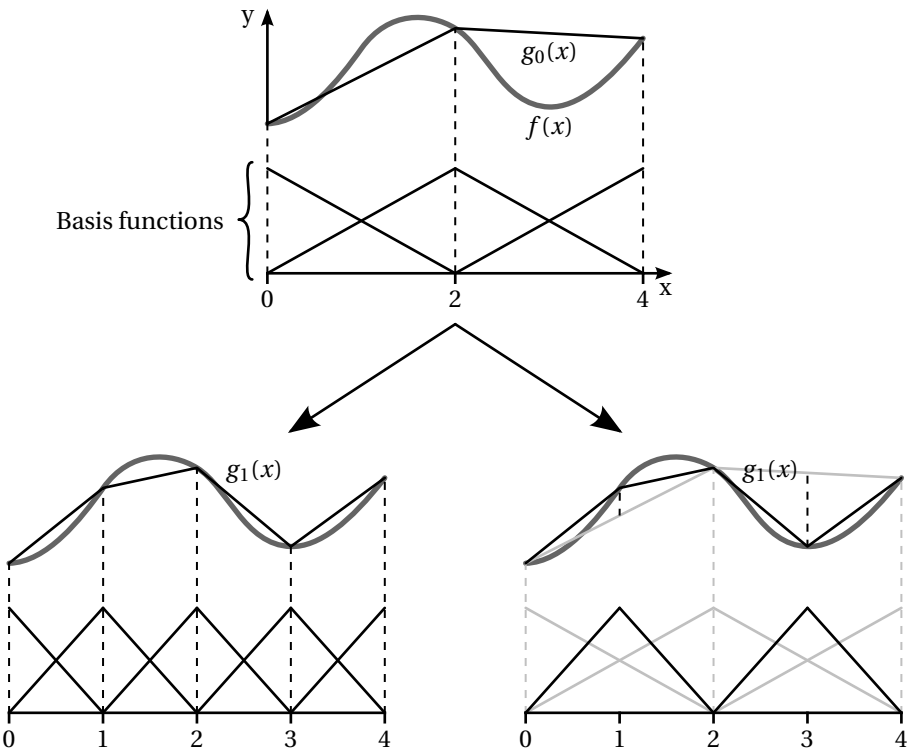

Figure 4.2: Hierarchical basis functions.

Of course, the plan is not to re-engineer HHG to use hierarchical bases. Rather, the understanding that a multigrid implementation using hierarchical bases would remain correct even if the changes to the unknowns vector were restricted to certain sub-meshes will guide the implementation of adaptivity in HHG. The algorithms have to be designed such that the computed results coincide with the results the above scheme would yield.

Outline of this section

The algorithmic steps that have to be performed at the hanging nodes can be deduced in a straightforward way from the original algorithms, in principle. To obtain algorithms that can be implemented efficiently, several more—and mathematically more involved—steps are necessary, though.

The description of the algorithmic transformations reflects these two aspects. In Section 4.3.1 the basic transformations leading to a working adaptive multigrid method are derived. In Sections 4.3.2 and 4.3.3 the modifications leading to an algorithm that can be implemented efficiently in HHG are introduced.

4.3.1 The basic algorithm

The starting point for developing an adaptive multigrid algorithm will be the full multigrid algorithm (Algorithm 7 from Section 2.3). For the first analysis of the adaptive full multigrid algorithm, it is not even necessary to have the notion of *hanging nodes*, at all. We assume a series of uniformly refined meshes Ω_l ($l_s \le l \le l_f$) with a hierarchical structure ($\Omega_{l-1} \subset \Omega_l$). This is the standard mesh hierarchy of HHG with uniform refinement.

The first step away from standard full multigrid is to let a refinement criterion determine, for every level l, a sub-mesh $\mathscr{C}_l$ on which the solution u_l has to be computed (see Definition 2). $\mathscr{C}_l$ may be computed statically, before the full multigrid algorithm is started, or dynamically, for each level l after u_{l-1} has been computed.

Depending on $\mathscr{C}_l$, two more sub-meshes can be defined.

- $\overline{\mathscr{C}}_l = \left(\Omega_l \cup \Gamma_l^N\right) \setminus \mathscr{C}_l$ denotes the sub-mesh on which u_l does not have to be computed.
- $\mathscr{R}_l$ denotes the points in $\overline{\mathscr{C}}_l$ that are coupled to points in $\mathscr{C}_l$ by the operator A_l. This set of points is called *refinement boundary*.

Fig. 4.3 gives an example of the hierarchical discretization of a two-dimensional rectangular domain Ω, and the partitioning of the resulting meshes Ω_l. Like this figure, most illustrations and examples in this chapter use 2D meshes; the algorithms do not depend on the mesh dimension, though, and work just as well in 3D. The boundaries on the top and on the right of Ω are of Dirichlet type (Γ^D); the boundaries on the bottom and on the left are of Neumann type (Γ^N). On the coarsest level, $l = 0$, Ω is discretized into the rectangular mesh Ω_0 with 1×5 interior points. Two steps of regular refinement yield the meshes Ω_1 and Ω_2; the finest mesh has 7×23 interior points.

On level 0, since this is the coarsest level, the solution has to be computed on all points. Therefore, the set $\mathscr{C}_0$ has to include all points in Ω_0, except the Dirichlet boundary points:

$$\mathscr{C}_0 = \Omega_0 \cup \Gamma_0^N .$$

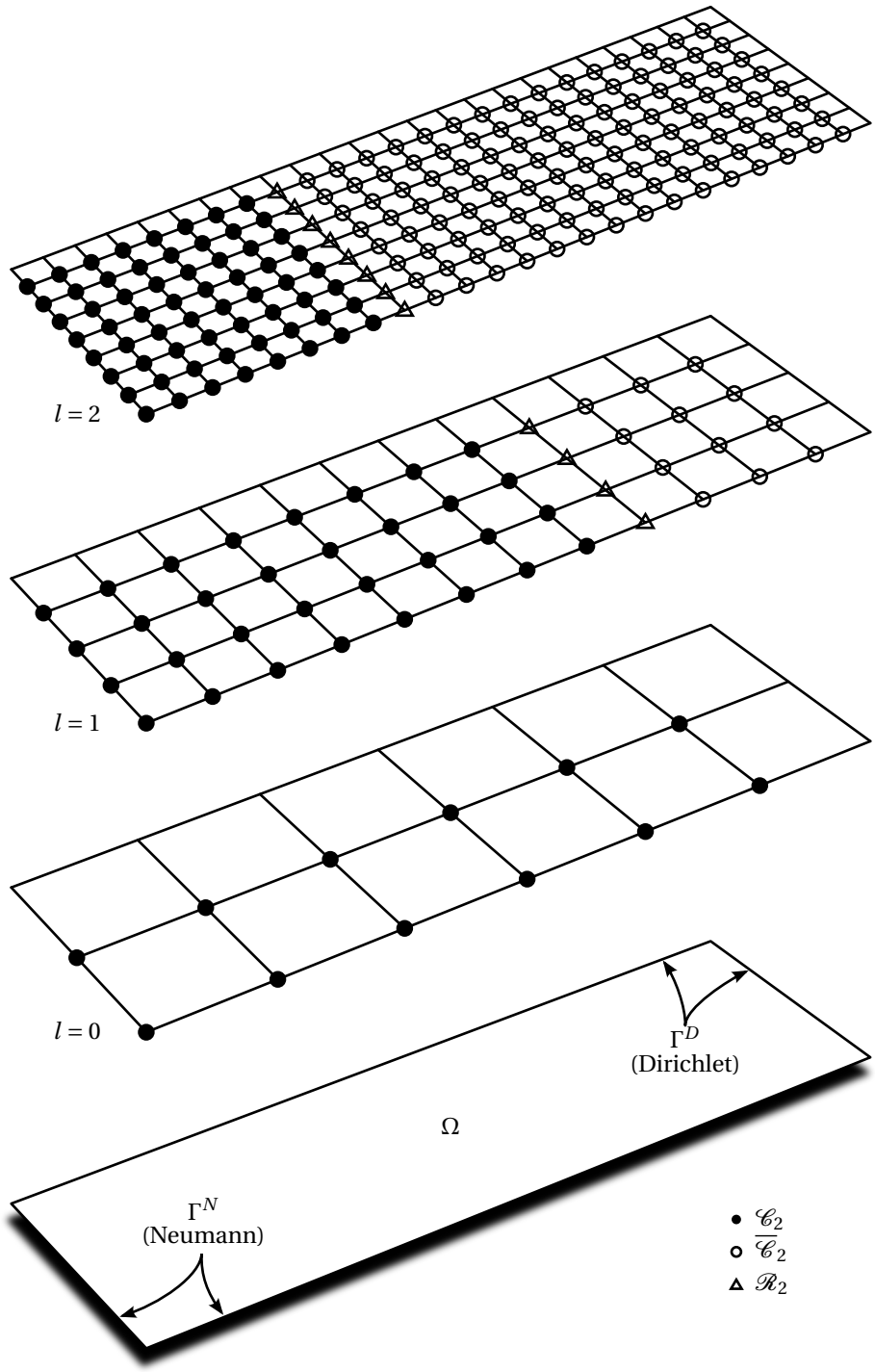

Figure 4.3: 2D domain with the sub-domains used in adaptive refinement.

For level 1, we arbitrarily assume that the refinement criterion has selected a sub-domain comprising the left two thirds of Ω_1. $\mathscr{C}_1$ consists of 32 points; they are marked with bullet points (•). The refinement boundary $\mathscr{R}_1$ is the row of points next to $\mathscr{C}_1$; the points on $\mathscr{R}_1$ are marked with triangles (△). The rest of the points in $\overline{\mathscr{C}}_1$ are marked with circles (◯). On level 2, the left third of Ω_2 (64 points) is in $\mathscr{C}_2$. Like on Ω_1, we find $\mathscr{R}_2$ and $\overline{\mathscr{C}}_2$ to the right of $\mathscr{C}_2$.

On all levels, the partitioning is based on the assumption that the operators A_l couple mesh points only to their nearest neighbors. If the operators coupled a greater range of points, the refinement boundary would have to be wider, and some additional points of $\overline{\mathscr{C}}_l$ would also be on $\mathscr{R}_l$.

For the following considerations it is sufficient to observe the algorithm on two levels, l and $l-1$, between l_s and l_f:

$$l-1 \geq l_s \text{ and } l \leq l_f\,.$$

The full approximation scheme will be used for the individual multigrid cycles embedded in the full multigrid solver. While FAS is more complicated than the correction scheme, it can be transformed into an adaptive scheme more easily. In particular, the initialization of f_{l-1} in the recursion is not straightforward in an adaptive correction scheme.

Adaptive smoothing

As stated above and as illustrated in Fig. 4.3, no hanging nodes are considered, for now. The goal of the first step is merely to incorporate $\mathscr{C}_l$ into the standard full approximation scheme. To accomplish this, the smoother (Algorithm 1) is confined to updating points in $\mathscr{C}_l$ only. In theory, this is only a slight modification; see Algorithm 9. Implementing this modification may be quite challenging, though. The implementation will be discussed in Section 4.5.

Algorithm 9: The adaptive smoother.

9.1 **Algorithm** `smooth`$(A_l, v_l, f_l, \mathscr{C}_l, \nu)$
9.2 Perform ν iterations of an iterative solver with good error smoothing properties on the linear system $A_l v_l = f_l$ on mesh Ω_l, only updating points in $\mathscr{C}_l$
9.3 **return** v_l

Algorithm 9 can now be employed by the full multigrid algorithm. Algorithm 10 is the full approximation scheme with adaptive smoothing, Algorithm 11 is the corresponding full multigrid algorithm. Compared to the original versions, Algorithms 7 and 8, some of the equations have been expanded, introducing additional variables. They will be useful in the detailed analysis of the algorithm, but they do not need to be computed explicitly in the implementation. The newly introduced variables are

- r_l, the initial residual on level l,
- d_{l-1}, the restricted residual (the "defect") on level $l-1$,
- c_{l-1}, the correction computed in the recursion on level $l-1$, and
- $\tilde{e}_l$, the approximation to the error on level l, computed from the coarse grid correction.

The parameter $\mathscr{C}$ stands for the set of all $\mathscr{C}_l$ ($l_s < l \leq l_f$), in analogy to the parameter A, which contains the operators on all levels.

Algorithm 10: The full approximation scheme with adaptive smoothing.

Algorithm `FAScycle`$(A, v_l, f_l, R, \hat{R}, P, l, l_s, \mathscr{C}, \nu_1, \nu_2)$
if $l = l_s$ **then**
  $v_l =$ `solve`$(A_l v_l = f_l)$
else
  $\overline{v}_l =$ `smooth`$(A_l, v_l, f_l, \mathscr{C}_l, \nu_1)$
  $\overline{r}_l = f_l - A_l \overline{v}_l$
  $d_{l-1} = R_l^{l-1} \overline{r}_l$
  $v_{l-1}^* = \hat{R}_l^{l-1} \overline{v}_l$
  $f_{l-1} = d_{l-1} + A_{l-1} v_{l-1}^*$
  $v_{l-1} =$ `FAScycle`$(A, v_{l-1}^*, f_{l-1}, R, \hat{R}, P, l-1, l_s, \mathscr{C}, \nu_1, \nu_2)$
  $c_{l-1} = v_{l-1} - v_{l-1}^*$
  $\tilde{e}_l = P_{l-1}^l c_{l-1}$
  $\tilde{v}_l = \overline{v}_l + \tilde{e}_l$
  $v_l =$ `smooth`$(A_l, \tilde{v}_l, f_l, \mathscr{C}_l, \nu_2)$
end
return v_l

Algorithm 11: The full multigrid algorithm, using the full approximation scheme and adaptive smoothing (Algorithm 10).

Algorithm `fullmg`$(A, v, f, R, P, \hat{P}, l_s, l_f, \mu, \nu_1, \nu_2)$
$u_{l_s} =$ `solve`$(A_{l_s} u_{l_s} = f_{l_s})$
for $l = l_s + 1 \ldots l_f$ **do**
  Initialize $\mathscr{C}_l$ according to refinement criteria
  $v_l = \hat{P}_{l-1}^l u_{l-1}$
  $r_l = f_l - A_l v_l$
  for $i = 1 \ldots \mu$ **do**
    $v_l =$ `FAScycle`$(A, v_l, f_l, R, \hat{R}, P, l, l_s, \mathscr{C}, \nu_1, \nu_2)$
  end
  $u_l = v_l$
end
return v_l

A detailed analysis

While the adaptive full approximation scheme can be implemented just as it is formulated in Algorithm 10, such an implementation would be of very limited use in reducing the computational effort for solving the linear system. Unknowns vector, right hand side, and residual are still defined on the entire mesh Ω_l, and besides the trimming of the smoothing step no numerical operations are saved, either. In order to identify further possibilities of reducing computational operations, we will analyze how each step of the algorithm affects the different regions of Ω.

This analysis is illustrated in Fig. 4.4. The diagram shows the partitioning of a 1D mesh with 17 points on level l and 9 points on level $l-1$. The points in Ω_l are marked with vertical bars (|). The points in $\mathscr{C}_l$ are marked with x-shaped crosses (×). The points in Ω_{l-1} are marked with with circles (◯). $\mathscr{C}_{l-1}$ spans the whole of Ω_{l-1} and is therefore not indicated separately. The boundary points are marked with squares (□); Dirichlet boundary conditions are imposed on both points. Neumann boundary conditions are not covered in this part of the analysis, because they would make no difference at this point, and including them would unnecessarily bloat the following formulations.

The diagram shows the simplest example of an adaptively refined mesh, but at the same time it is very general. Every other partitioning for adaptive refinement—whether it were applied to higher-dimensional or to more complicated domains—can be broken down to this abstract structure: two disjoint groups of mesh points, $\mathscr{C}_l$ and $\overline{\mathscr{C}}_l$, that are separated by a refinement boundary, $\mathscr{R}_l$. Only when it comes to domain boundaries, a 1D picture is not enough to illustrate all cases that are possible in two or more dimensions. In 1D $\mathscr{R}_l$ can never be adjacent to Γ_l, because it is enclosed by $\mathscr{C}_l$ and $\overline{\mathscr{C}}_l$. In higher dimensions this is possible. However, since Algorithms 11 and 10 are not concerned with setting boundary conditions (the boundaries are assumed to be initialized correctly), it is possible at this point to abstain from a precise formulation in favor of simplicity.

The following line by line analysis starts at line line 11.5 of Algorithm 11. We assume that $A_{l-1}u_{l-1} = f_{l-1}$ has been solved for u_{l-1} and $\mathscr{C}_l$ has been initialized. The analysis covers the prolongation of the solution from level $l-1$ to level l and the first FAS-cycle (i. e., $i = 1$ in Algorithm 11) on level l. The analysis would be the same for the subsequent FAS-cycles ($i = 2 \dots \mu$), except for a few details that will be highlighted at the end of this section.

Line 11.5. The prolongation of the solution from level $l-1$ to level l affects points in $\mathscr{C}_l$ and in $\overline{\mathscr{C}}_l$ in the same way. This is reflected in the diagram by a box covering both partitions. The diagram also shows that on the boundary v_l is not obtained by prolongation, but that it is initialized with the problem's boundary values.

Line 11.6. The initial residual is determined from the initial prolongation of the solution by

$$r_l \;=\; f_l - A_l v_l \;=\; f_l - A_l \hat{P}^l_{l-1} u_{l-1} \quad \text{in } \mathscr{C}_l \cup \overline{\mathscr{C}}_l \,. \tag{4.1}$$

On the boundary the residual is not defined.

Line 10.5. Pre-smoothing acts only on points in $\mathscr{C}_l$ now. Therefore,

$$\overline{v}_l \;=\; v_l \quad \text{in } \overline{\mathscr{C}}_l \cup \Gamma_l \,. \tag{4.2}$$

Line 10.6. The general formula for computing the residual after the pre-smoothing step is $\overline{r}_l = f_l - A_l \overline{v}_l$. However, since $\overline{v}_l = v_l$ in $\overline{\mathscr{C}}_l$, there is an area in which the residual is not changed, i. e., $\overline{r}_l = r_l = f_l - A_l v_l = f_l - A_l \hat{P}^l_{l-1} u_{l-1}$. This formulation does not look like a simplification at this point, but it will soon become useful. Therefore, we want to locate precisely in which part of $\overline{\mathscr{C}}_l$ it is valid.

$\overline{v}_l$ influences $\overline{r}_l$ only through the multiplication with A_l, so the shape of A_l determines how far into $\overline{\mathscr{C}}_l$ a change of v_l in $\mathscr{C}_l$ will spread. If the stencil of A_l couples mesh points only to their nearest neighbors, changes to v_l will spread into $\mathscr{R}_l$, but not any further. Thus, it is valid to use

$$\overline{r}_l \;=\; r_l \;=\; f_l - A_l \hat{P}^l_{l-1} u_{l-1} \quad \text{in } \overline{\mathscr{C}}_l \setminus \mathscr{R}_l \,. \tag{4.3}$$

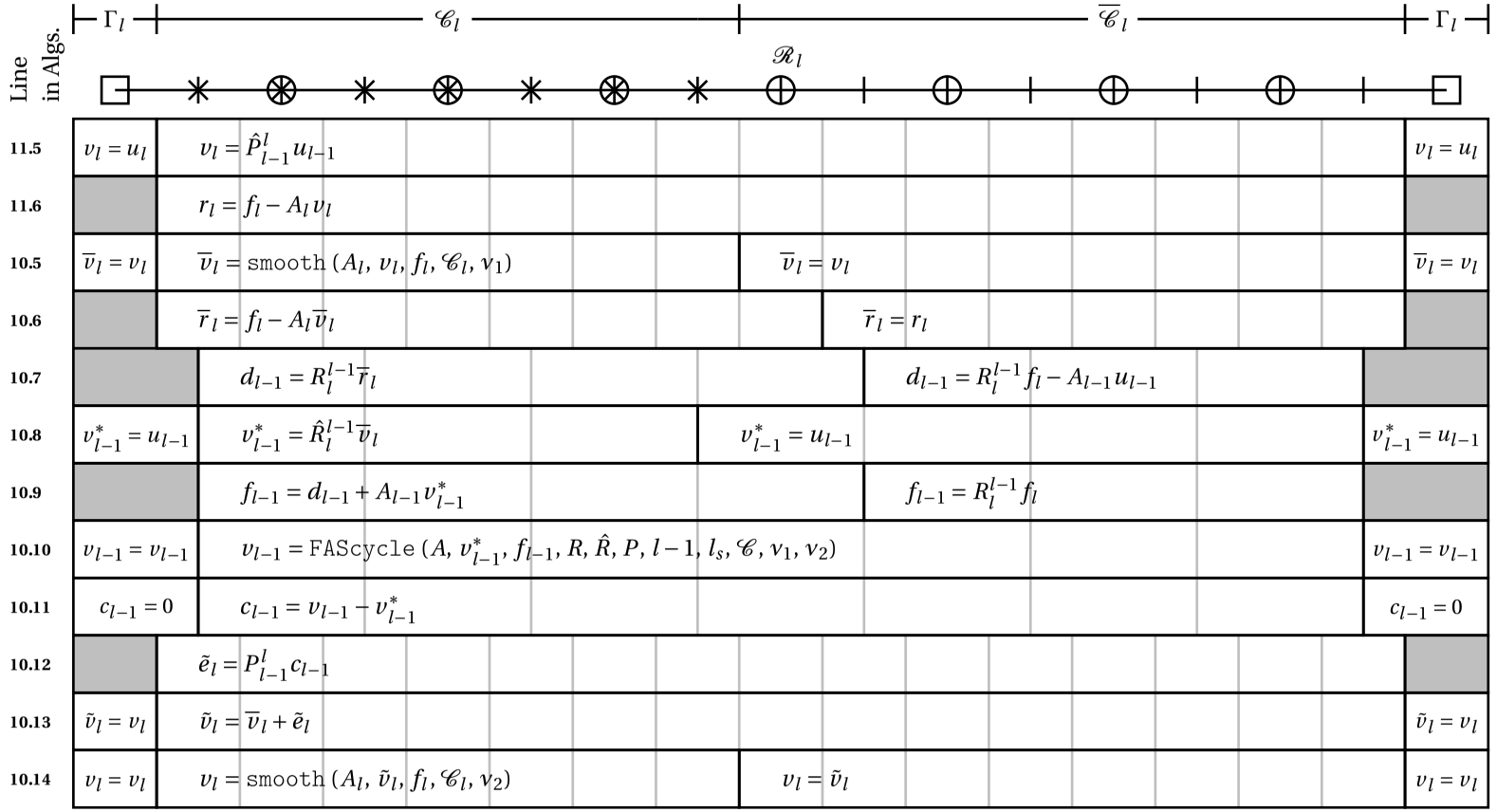

Figure 4.4: Diagram of the full multigrid algorithm with the full approximation scheme and adaptive smoothing. The diagram shows the propagation from level $l-1$ to level l and the first FAS-cycle on level l.

This "lucky" choice of $\mathscr{R}_l$ is no coincidence, but it is in fact founded on this property of A_l. The width of the refinement boundary layer should be chosen such that it shields changes in $\mathscr{C}_l$ from the rest of Ω_l.

Line 10.7. The defect on level $l-1$ is, in general, computed as $d_{l-1} = R_l^{l-1}\overline{r}_l$. In a part of Ω_{l-1} (4.3) can be used to modify the formula for d_{l-1}. We want to identify that part, because the modification will be used in the computation of f_{l-1}, later on.

According to (4.3), the necessary property, $\overline{r}_l = r_l$, is fulfilled in $\overline{\mathscr{C}}_l \setminus \mathscr{R}_l$, but not in $\mathscr{C}_l \cup \mathscr{R}_l$. The shape of the restriction operator R_l^{l-1} determines how this partitioning is transferred to the coarse mesh. Equation (4.4) can be used on all points of the coarse mesh that are not connected to points in $\mathscr{C}_l \cup \mathscr{R}_l$ by the restriction operator. The standard restriction operator (introduced in Section 2.3.2) uses only nearest neighbor points on the fine mesh to compute the values on the coarse mesh. For example, the value of d_{l-1} at point 10 is computed from the values of $\overline{r}_l$ at points 9, 10, and 11. They are all in $\Omega_l \setminus \mathscr{R}_l$, thus, (4.4) can be used at point 10.

Based on these considerations, the formula for the defect can be expanded to

$$\begin{aligned} d_{l-1} &= R_l^{l-1}\overline{r}_l = R_l^{l-1} r_l \\ &= R_l^{l-1}\left(f_l - A_l \hat{P}_{l-1}^l u_{l-1}\right) \\ &= R_l^{l-1} f_l - R_l^{l-1} A_l \hat{P}_{l-1}^l u_{l-1} \\ &= R_l^{l-1} f_l - A_{l-1} u_{l-1} \qquad \text{in } \Omega_{l-1} \setminus (\mathscr{C}_l \cup \mathscr{R}_l)\,. \end{aligned} \tag{4.4}$$

If the restriction operator used a wider range of points, including also point 8, (4.4) could not be used at point 10. Here, and in several occasions below, we assume a compact operator that has only nearest-neighbor dependencies. The theory of adaptive refinement, as it is developed here, is also valid for wider stencils; the implementation in HHG, however, will heavily exploit the compactness of the operators, because its computational efficiency—especially on parallel computers—depends strongly on the compactness of the operators.

Line 10.8. After the defect, the unknowns are initialized on the coarse mesh. In the sub-domain $\Omega_{l-1} \cap \mathscr{C}_l$ the restriction $v_{l-1}^* = \hat{R}_l^{l-1}\overline{v}_l$ is computed. In $\Omega_{l-1} \cap \overline{\mathscr{C}}_l$ this restriction does not have to be computed. Here $\overline{v}_l = v_l$, because no smoothing took place in $\overline{\mathscr{C}}_l$. Therefore, v_{l-1}^* can be calculated as

$$\begin{aligned} v_{l-1}^* &= \hat{R}_l^{l-1}\overline{v}_l = \hat{R}_l^{l-1} v_l \\ &= \hat{R}_l^{l-1}\hat{P}_{l-1}^l u_{l-1} \qquad \text{in } \Omega_{l-1} \cap \overline{\mathscr{C}}_l\,. \end{aligned}$$

Recall from Section 2.3.4 that injection is used for the restriction of the unknowns (the operator $\hat{R}_l^{l-1}$). Furthermore, the interpolation $\hat{P}_{l-1}^l$ is also an injection—from the coarse to the fine mesh—at the points that coincide on coarse and fine mesh. Therefore, the operator product $\hat{P}_{l-1}^l \hat{R}_l^{l-1}$ does not alter u_{l-1}, and the unknowns on the coarse mesh can be initialized as

$$v_{l-1}^* = u_{l-1} \quad \text{in } \Omega_{l-1} \cap \overline{\mathscr{C}}_l\,. \tag{4.5}$$

Line 10.9. With d_{l-1} and v_{l-1}^* set up, the right hand side can be initialized. In $\Omega_{l-1} \cap (\mathscr{C}_l \cup \mathscr{R}_l)$, where (4.4) and (4.5) can not be used, $f_{l-1} = d_{l-1} + A_{l-1} v_{l-1}^*$. In $\Omega_{l-1} \setminus (\mathscr{C}_l \cup \mathscr{R}_l)$, these equations can be used, and the right-hand side on the coarse mesh can simply be initialized with the restricted right-hand side of the original fine

problem on the fine mesh

$$\begin{aligned} f_{l-1} &= d_{l-1} + A_{l-1} v_{l-1}^* \\ &= R_l^{l-1} f_l - A_{l-1} u_{l-1} + A_{l-1} u_{l-1} \\ &= R_l^{l-1} f_l \qquad\qquad \text{in } \Omega_{l-1} \setminus (\mathscr{C}_l \cup \mathscr{R}_l)\,. \end{aligned} \tag{4.6}$$

This is already a helpful result, because it eliminates the need to compute d_{l-1} in this region. On the other hand, f_l would be required in all of Ω_l to compute f_{l-1}, which would make all further effort to reduce computational complexity a vain endeavor. Stated more optimistically, it is worthwhile to find out if $R_l^{l-1} f_l$ can be replaced by f_{l-1}.

There is no general answer to this question, because the function f of the application's mathematical model, which is used to initialize f_l, may be of any type. The full-weighting restriction used in HHG, for example, can only restrict polynomials up to second degree exactly.

However, there is still reason to assume that, in practice, $f_{l-1} = R_l^{l-1} f_l$ is—almost and most of the time, at least—fulfilled: if the right-hand side is so rough that $R_l^{l-1} f_l$ is not even approximately equal to f_{l-1}, it is questionable whether solving on level $l-1$ would be sufficiently accurate, at all. In other words, in a region where the right-hand side is discontinuous or very oscillating, the solution must probably be computed on the fine mesh, anyway. The literature also endorses initializing f_{l-1} in $\Omega_{l-1} \setminus (\mathscr{C}_l \cup \mathscr{R}_l)$ by discretizing f on level $l-1$ (see, e. g., [35]). With the knowledge that the implementation of this algorithm will not be a black-box solver, anyway, we choose this pragmatic approach, keeping in mind that it will not work in all cases one can think of.

Line 10.10. Now all the variables are initialized and the adaptive FAS-cycle (Algorithm 10) can be executed recursively on Ω_{l-1}. It solves the linear system $A_{l-1} v_{l-1} = f_{l-1}$ for v_{l-1}. Like in the standard FAS algorithm, v_{l-1} may be only a good approximation, not the exact solution, to $A_{l-1}^{-1} f_{l-1}$, but like in the standard algorithm, this does not pose a problem here, either. Note that the recursion provides a solution on the complete mesh Ω_{l-1}, even if $\mathscr{C}_{l-1}$ is not, as in Fig. 4.4, identical with Ω_{l-1}. Whether the v_{l-1} is *needed* on all of Ω_{l-1} will be analyzed later on in this section.

Line 10.11. Since the recursion has potentially modified the unknowns vector everywhere on Ω_{l-1}, the correction $c_{l-1} = v_{l-1} - v_{l-1}^*$ has to be computed on the complete mesh Ω_{l-1}, too. On the boundary, $c_{l-1} = 0$.

Line 10.12. $\tilde{e}_l$, the approximation to the error on the fine mesh, is computed by prolongating c_{l-1}. The formula $\tilde{e}_l = P_{l-1}^l c_{l-1}$ holds on all of Ω_l and cannot be simplified anywhere.

Line 10.13. The same applies to $\tilde{v}_l = \overline{v}_l + \tilde{e}_l$. In $\overline{\mathscr{C}}_l$, where no pre-smoothing took place, $\overline{v}_l$ could be replaced by v_l, but this would not be of any use in improving the algorithm.

Line 10.14. Post-smoothing, like pre-smoothing, changes $\tilde{v}_l$ only in $\mathscr{C}_l$. Therefore,

$$v_l = \tilde{v}_l \quad \text{in } \overline{\mathscr{C}}_l \cup \Gamma_l\,.$$

With the post-smoothing step, the FAS-cycle is complete. The diagram in Fig. 4.4 gives a complete picture of how the algorithm actually affects the different parts of the meshes. The global picture reveals: there is no single dividing line between the refined and non-refined areas. While the transformations of the formulas that have

been derived above are straightforward, it is still not immediately visible which calculations can be saved in the implementation. Therefore, the next step is to analyze the data dependencies of each algorithmic step. With this information it will be possible to create an efficient implementation stripped of unnecessary calculations.

More FAS-cycles on level l

The discussion up to now made use of the observation that $\overline{v}_l = v_l = \hat{P}^l_{l-1} u_{l-1}$ in regions where no smoothing is done, i. e., in $\overline{\mathscr{C}}_l$. For the first FAS-cycle on level l this is obvious. Is this assumption still correct for subsequent FAS-cycles, though?

Expanding the formula for v_l in $\overline{\mathscr{C}}_l$ at the end of the first cycle using line 10.8 and lines 10.11–10.14 yields

$$\begin{aligned} v_l &= \tilde{v}_l && \text{(line 10.14)} \\ &= \overline{v}_l + \tilde{e}_l && \text{(line 10.13)} \\ &= \overline{v}_l + P^l_{l-1} c_{l-1} && \text{(line 10.12)} \\ &= \overline{v}_l + P^l_{l-1}(v_{l-1} - v^*_{l-1}) && \text{(line 10.11)} \\ &= \overline{v}_l + P^l_{l-1} v_{l-1} - P^l_{l-1} \hat{R}^{l-1}_l \overline{v}_l && \text{(line 10.8)} \\ &\approx P^l_{l-1} v_{l-1} && \text{in } \overline{\mathscr{C}}_l . \end{aligned} \tag{4.7}$$

In general, $v_{l-1} \neq u_{l-1}$, as the following reasoning shows. In the FAS-cycles starting from level l, f_{l-1} is initialized using information computed on level l. Thus, f_{l-1} will be different than in the FAS-cycles up to level $l-1$, where it was initialized by discretizing f. Thus, since f_{l-1} *globally* influences the solution of $A_{l-1} v_{l-1} = f_{l-1}$, the assumption that $v_l = \hat{P}^l_{l-1} u_{l-1}$ in $\overline{\mathscr{C}}_l$ does generally not hold for subsequent FAS-cycles.

The above line by line analysis of the algorithm is not invalidated by this change, though. None of the derivations introduced depends on the assumption that u_{l-1} contains the approximation computed by the last FAS-cycle on level $l-1$. The relevant observations, e. g., that u_{l-1} cancels out in the calculation of f_{l-1} in (4.6), hold independently of the value of u_{l-1}.

One more interesting fact shall be highlighted at this point. Equation (4.7) shows that the FAS-cycle on level l changes v_l also in $\overline{\mathscr{C}}_l$ (i. e., v_l is not simply an interpolation of u_{l-1} there), even though Algorithm 10 does not do any real work in $\overline{\mathscr{C}}_l$. In other words, the FAS-cycle with adaptive smoothing improves the approximation on the coarser meshes also in areas that are not refined.

4.3.2 The improved algorithm

The way towards computational efficiency is described in two parts. The first set of modifications is straightforward and does not require any additional assumptions about the underlying problem or the numerical method. In the second part, the algorithm is optimized further to suit an implementation that uses HHG's data structures and its communication infrastructure. Two diagrams accompany the remarks and highlight the changes of each step.

The first diagram is shown in Fig. 4.5. It has the same structure and uses the same notation as the diagram in Fig. 4.4. Compared to the original one, gray boxes have been inserted where computations are not necessary. A gray box in the diagram means that the corresponding formula does not have to be computed in the

sub-mesh that is grayed out. In the following paragraphs, the rationale for the locations of these areas will be established. In the analysis, two additional refinement boundary layers will come into the focus of attention. They are marked with $\mathscr{R}_l^E$ and $\mathscr{R}_l^{EE}$ in the diagram and are defined as follows.

- $\mathscr{R}_l^E$, the *external refinement boundary*, is the set of all points in $\overline{\mathscr{C}}_l \setminus \mathscr{R}_l$ that are coupled to points in $\mathscr{R}_l$.
- $\mathscr{R}_l^{EE}$, the *second external refinement boundary*, is the set of all points in $\overline{\mathscr{C}}_l \setminus \left(\mathscr{R}_l \cup \mathscr{R}_l^E\right)$ that are coupled to points in $\mathscr{R}_l^E$.

The one line that determines the shape of the whole diagram does not stand out at first glance: it is the defect calculation (line 10.7). Starting from here, it is straightforward to derive in which parts of Ω_l and Ω_{l-1} the other variables need to have their correct values.

As the diagram shows, d_{l-1} depends on $\overline{r}_l$ in $\mathscr{C}_{l-1} \cap (\mathscr{C}_l \cup \mathscr{R}_l)$. Outside that area, d_{l-1} does not depend on values from the fine mesh (except for f_l, which, as argued above, can be replaced with f_{l-1}). So, where in Ω_l does $\overline{r}_l$ have to be computed? The answer depends, as in several occasions before, on the scope of R_l^{l-1}, and, as usually, R_l^{l-1} is assumed to have only nearest-neighbor couplings. Under that premise, $\overline{r}_l$ needs to be computed in $\mathscr{C}_l \cup \mathscr{R}_l \cup \mathscr{R}_l^E$. The area is shown in the diagram (line 10.6).

Two different formulas are employed for obtaining $\overline{r}_l$ in this area. In $\mathscr{C}_l \cup \mathscr{R}_l$ $\overline{r}_l$ depends on $\overline{v}_l$; in $\mathscr{R}_l^E$ it depends only on r_l. The dependencies are fulfilled by the two lines above line 10.6 in the diagram. Assuming that A_l has only nearest-neighbor couplings, $\overline{v}_l$ has to be provided on $\mathscr{C}_l \cup \mathscr{R}_l \cup \mathscr{R}_l^E$ and on the part of Γ_l adjacent to $\mathscr{C}_l$ (see line 10.5). r_l has to be computed only on $\mathscr{R}_l^E$ (see line 11.6), because it is simply assigned—without applying any operator—to $\overline{r}_l$ in line 10.6.

On $\mathscr{C}_l$ $\overline{v}_l$ is created by smoothing, outside $\mathscr{C}_l$ it is equal to v_l. Therefore, v_l must be available in $\mathscr{C}_l \cup \mathscr{R}_l \cup \mathscr{R}_l^E$ and on Γ_l at $\mathscr{C}_l$. The formula for r_l also depends on v_l, multiplied with A_l. The latter dependency is also the one that creates the need for $\mathscr{R}_l^{EE}$, the second external refinement boundary: for computing r_l on $\mathscr{R}_l^E$, v_l is needed on $\mathscr{R}_l$, $\mathscr{R}_l^E$, and $\mathscr{R}_l^{EE}$. Line 11.5 initializes v_l by prolongating the solution on the coarse mesh, u_{l-1}, onto $\mathscr{C}_l \cup \mathscr{R}_l \cup \mathscr{R}_l^E \cup \mathscr{R}_l^{EE}$. On the mesh boundary, v_l is initialized with the problem's boundary values.

Now, the first part of Fig. 4.5 (lines 11.5–10.10) is complete. To complete the second part, another anchor for deriving the data dependencies has to be found. The argument which comes to mind first is to restrict line 10.14 to $\mathscr{C}_l$, because that is the area where the solution on level l has to be computed, according to the refinement criterion. On second glance, this is sufficient only in the first FAS-cycle on level l. In the likely case that more cycles follow, line 10.14 has to cover the same area as line 11.5 in order to fulfill the dependencies of the following cycle. Therefore, we require v_l to be present on $\mathscr{C}_l \cup \mathscr{R}_l \cup \mathscr{R}_l^E \cup \mathscr{R}_l^{EE}$ and on the part of Γ_l adjacent to that area. On $\mathscr{C}_l$ v_l is computed by smoothing, outside $\mathscr{C}_l$ the variable is equal to $\tilde{v}_l$.

$\tilde{v}_l$ is computed in line 10.13 by adding the error approximation $\tilde{e}_l$ to the presmoothed unknowns. The addition does not expand the data dependencies, thus, $\tilde{e}_l$ is also required on $\mathscr{C}_l \cup \mathscr{R}_l \cup \mathscr{R}_l^E \cup \mathscr{R}_l^{EE}$ and on the appropriate part of Γ_l.

Since $\tilde{e}_l$ is computed from c_{l-1} via the prolongation operator, c_{l-1} may need to be available on a larger area than $\tilde{e}_l$. Let us examine the boundaries of the active domain in line 10.12. The leftmost point only exists on the fine mesh. Therefore, the values of c_{l-1} on the adjacent points on the coarse mesh are used for interpolating $\tilde{e}_l$

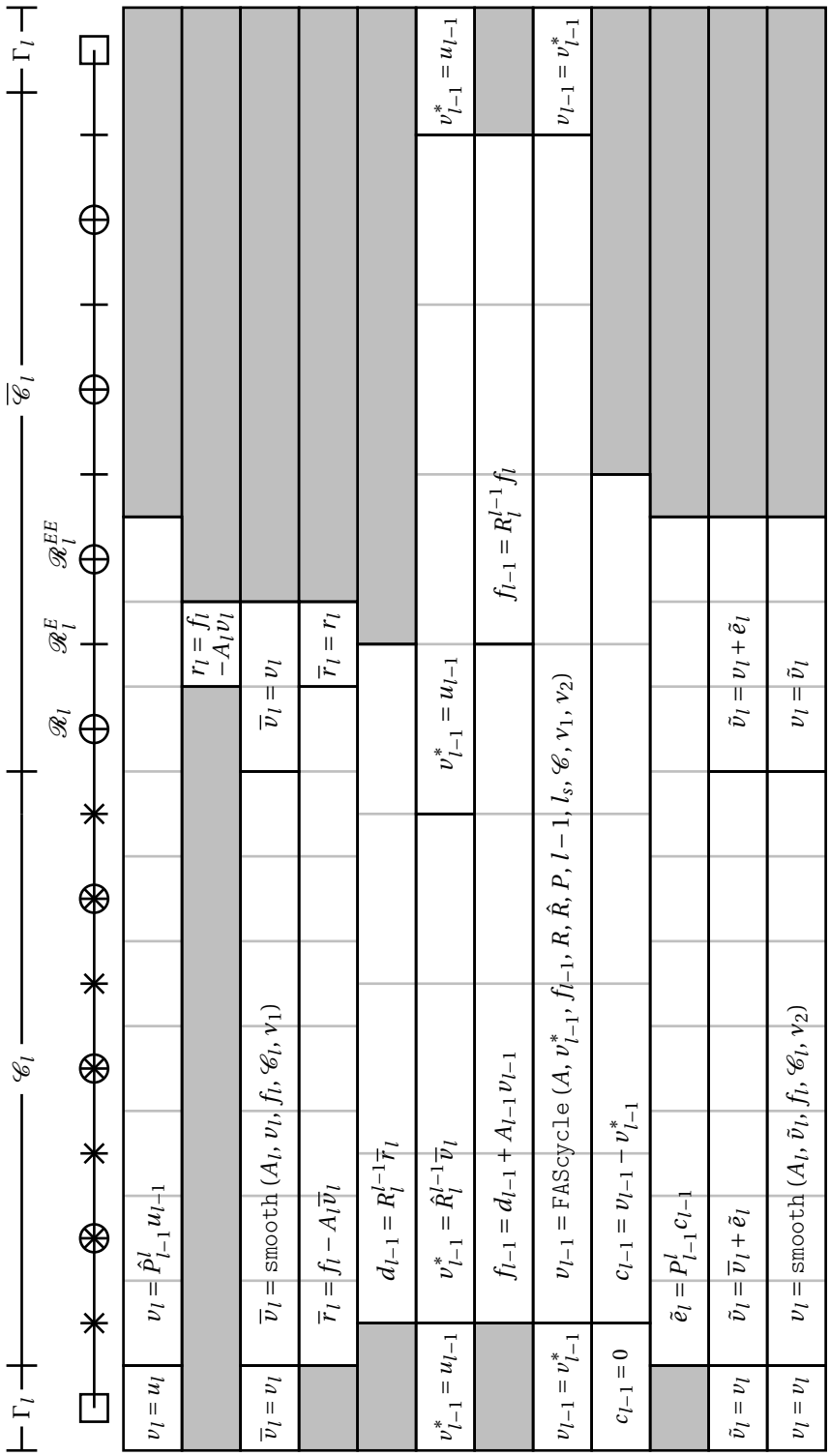

Figure 4.5: Diagram of the adaptive full multigrid algorithm with the improved full approximation scheme.

on the fine mesh point. One of them is on the boundary Γ_{l-1}, the other one is on $\mathscr{C}_{l-1}$ (see line 10.11). The rightmost point of the active domain in line 10.12, $\mathscr{R}_l^{EE}$, exists on both Ω_l and Ω_{l-1}. Thus, it can be interpolated from the coarse mesh by injection, and the area for computing c_{l-1} does not have to be expanded on that end. All in all, c_{l-1} has to be computed on $\mathscr{C}_{l-1} \cap \left(\mathscr{C}_l \cup \mathscr{R}_l \cup \mathscr{R}_l^E \cup \mathscr{R}_l^{EE}\right)$ and initialized to zero on Γ_{l-1} adjacent to that area.

Implementation issues

Now that the individual lines have been analyzed, let us take a step back and examine the diagram in its entirety. Data on the fine mesh has to be computed on the points in $\mathscr{C}_l$ and on the three refinement boundary layers $\mathscr{R}_l$, $\mathscr{R}_l^E$, and $\mathscr{R}_l^{EE}$. At first glance, this observation is not problematic, but how about the implementation of this algorithm? If it were for an implementation using regular grids with global indexing, no parallelization, and no interest in efficient data structures, it still would not be a problem. Taking away one of these advantages—HHG can not rely on a single one of them—makes the implementation much more challenging. As Section 4.5.2 will show, treating $\mathscr{R}_l^E$ efficiently in the implementation requires additional data structures and program code. Additionally handling $\mathscr{R}_l^{EE}$ would make the implementation even more complex. Therefore, the goal for the final version of the algorithm is to eliminate the need for $\mathscr{R}_l^{EE}$. The resulting algorithm will be much easier to implement.

4.3.3 The final algorithm

To eliminate the second external refinement boundary ($\mathscr{R}_l^{EE}$), the defect d_{l-1} at the refinement boundary will be computed with a modified formula. Fig. 4.5 shows that the residual calculation (line 10.6) partly accesses entries in the unknowns vector that have not been altered since the initial prolongation (in line 11.5). This observation endorses the question whether data that is not modified is affecting the results of the computation at all. We will show that it is possible to segregate this data and eliminate it from the computation.

For analyzing the effect of computations on sub-meshes, the index notation $x_l[i]$ will be used. $x_l[i]$ refers to the value of variable x on an individual point i of the mesh on level l. All mesh points adjacent to point i on level l, the *neighborhood* of point i, will be denoted by $N_l(i)$.

Using this notation, the restriction can be written as a weighted sum. The defect at each coarse mesh point (line 10.7) is the weighted sum of the residuals at the neighboring fine mesh points, i. e.,

$$d_{l-1}[i] = \sum_{j \in N_l(i)} \left(w_{ij} \overline{r}_l[j] \right) \tag{4.8}$$

with the weights w_{ij}. The neighborhood of point i can generally be split into two disjoint sets $N_l^S(i)$ and $N_l^P(i)$. The sub-set $N_l^S(i)$ contains all points of $N_l(i)$ at which $\overline{r}_l$ depends on smoothed values of $\overline{v}_l$:

$$N_l^S(i) = N_l(i) \cap (\mathscr{C}_l \cup \mathscr{R}_l) .$$

The sub-set $N_l^P(i)$ contains the remaining points,

$$N_l^P(i) = N_l(i) \setminus N_l^S(i) ,$$

where the residual depends *only* on prolongated (i. e., non-smoothed) values of $\overline{v}_l$. One of the two sub-sets may be empty, depending on the location of point i.

This splitting is useful in the elimination of $\mathscr{R}_l^{EE}$, because $N^S(i) \cap \mathscr{R}_l^E = \emptyset$ for any point i. Splitting the sum in (4.8) according to this partitioning yields

$$d_{l-1}[i] = \sum_{j \in N_l^S(i)} \left(w_{ij} \overline{r}_l[j] \right) + \sum_{j \in N_l^P(i)} \left(w_{ij} \overline{r}_l[j] \right). \tag{4.9}$$

If the sum over $j \in N_l^P(i)$ could be eliminated from this equation, $\overline{r}_l$ would not have to be computed on $\mathscr{R}_l^E$, and, consequently, v_l would not have to be provided on $\mathscr{R}_l^{EE}$.

The residual $\overline{r}_l$ at points in $N_l^P(i)$ is the same as if it had been calculated right after the prolongation (line 11.5) (i. e., before any smoothing steps), because points in $N_l^P(i)$ are not coupled to any points that are affected by smoothing. To examine this relation, we call the residual after prolongation r_l and its restriction to the coarse mesh $\hat{d}_{l-1}$. The variables are defined as

$$\begin{aligned} r_l &= f_l - A_l v_l \quad \text{and} \\ \hat{d}_{l-1} &= R_l^{l-1} r_l . \end{aligned}$$

$\hat{d}_{l-1}$ at point i is, analogous to (4.9), a weighted sum of neighboring entries of r_l:

$$\hat{d}_{l-1}[i] = \sum_{j \in N_l^S(i)} \left(w_{ij} r_l[j] \right) + \sum_{j \in N_l^P(i)} \left(w_{ij} r_l[j] \right). \tag{4.10}$$

The residual after smoothing is equal to the residual before smoothing at mesh points which are not coupled to $\mathscr{C}_l$, i. e.,

$$\overline{r}_l[j] = r_l[j] \text{ for all neighbors } j \in N_l^P(i) \text{ of any point } i \in \Omega_l .$$

Thus, r_l can be substituted for $\overline{r}_l$ in the corresponding sum in (4.9):

$$d_{l-1}[i] = \sum_{j \in N_l^S(i)} \left(w_{ij} \overline{r}_l[j] \right) + \sum_{j \in N_l^P(i)} \left(w_{ij} r_l[j] \right). \tag{4.11}$$

Subtracting (4.10) from (4.11) eliminates the sum over the points in $N_l^P(i)$, leaving

$$d_{l-1}[i] = \hat{d}_{l-1}[i] + \sum_{j \in N_l^S(i)} \left(w_{ij} \left(\overline{r}_l[j] - r_l[j] \right) \right). \tag{4.12}$$

This means that d_{l-1} can easily be computed without using residual values from $\mathscr{R}_l^E$, provided that $\hat{d}_{l-1}$ and r_l are known. While r_l can be acquired on the necessary mesh points simply by computing it after the prolongation, the attempt to compute $\hat{d}_{l-1}$ from r_l would create a "chicken and egg" dilemma: in order to compute it (via (4.10)), the values of r_l at points on $\mathscr{R}_l^E$ are required, yet this is precisely what we want to get rid of.

The goals is thus to compute $\hat{d}_{l-1}$ in a different way. Expanding the formula for $\hat{d}_{l-1}$ yields

$$\begin{aligned} \hat{d}_{l-1} &= R_l^{l-1} r_l \\ &= R_l^{l-1} \left(f_l - A_l v_l \right) \\ &= R_l^{l-1} f_l - R_l^{l-1} A_l \hat{P}_{l-1}^l u_{l-1} . \end{aligned}$$

As already discussed in Section 4.3.1, $R_l^{l-1} f_l$ can be replaced with f_{l-1}.

In order to simplify the remainder of the equation, one approach is to make use of the fact that HHG's operators fulfill the *Galerkin condition*,

$$A_{l-1} = R_l^{l-1} A_l P_{l-1}^l .$$

Unfortunately, $\hat{P}$ appears instead of P in the equation. Thus, we have to make the somewhat painful restriction that $\hat{P} = P$ in Algorithm 11. Recall from Section 2.3.3 that for $\hat{P}$ generally a higher-order interpolation is required than for P, because it needs to interpolate the solution, which is not necessarily a smooth function. Using a first-order operator here may have the consequence that more FAS-cycles are required on level l, and, in the worst case, full multigrid does not have optimal complexity, any more. That being said, the winning argument will once more be a practical one. In the current HHG implementation, only the linear interpolation is available, because HHG's data structures and communication algorithms only allow for data exchange between directly adjacent mesh points. A higher-order interpolation operator creates data dependencies between points that are not nearest neighbors. In order to implement it, new data structures would have to be added to HHG. If they are available, they will also simplify handling a second external refinement boundary, which this section attempts to get rid of. To summarize: the restriction $\hat{P} = P$ does not deteriorate HHG's current convergence or the quality of the solution; when a higher-order interpolation is implemented, the restriction will not be needed, any more.

Thus, the formula for $\hat{d}_{l-1}$ can be simplified further to

$$\begin{aligned} \hat{d}_{l-1} &= f_{l-1} - R_l^{l-1} A_l P_{l-1}^l u_{l-1} \\ &= f_{l-1} - A_{l-1} u_{l-1} . \end{aligned}$$

This is the residual of the final approximation to the solution on level $l-1$ (line 11.10 in Algorithm 11), which will be denoted with r_{l-1}:

$$r_{l-1} = f_{l-1} - A_{l-1} u_{l-1} . \tag{4.13}$$

The relation

$$\hat{d}_{l-1} = r_{l-1}$$

can now be substituted in (4.12). The resulting formula,

$$d_{l-1}[i] = r_{l-1}[i] + \sum_{j \in N_l^S(i)} \left(w_{ij} \left(\overline{r}_l[j] - r_l[j] \right) \right) , \tag{4.14}$$

is easy to compute and can thus be built into the adaptive full multigrid algorithm, provided that r_{l-1} is computed.

The final versions of the algorithms are shown in Algorithm 12 (the full approximation scheme) and Algorithm 13 (the full multigrid solver). The computations are now restricted to the regions derived in Section 4.3.2. For every line, Algorithms 12 and 13 state on which part of the mesh it has to be computed. If the formula for a statement varies among the mesh regions, the different versions are merged into one algorithmic line. The set $\mathscr{R}_l^I$ is new, it will be introduced below.

The full multigrid algorithm has been expanded by the calculation of r_{l-1} at the beginning of each level (line 13.5) and the calculation of r_l after prolongation

(line 13.7). The full approximation scheme has been expanded by residual calculations after the recursion (line 12.11) and after the coarse grid correction (line 12.15). Both lines are required for the computation of $d_{l-1} = r_{l-1} + R_l^{l-1}\left(\overline{r}_l - r_l\right)$ in the subsequent multigrid cycle: since line 12.10 possibly changes v_{l-1} in all of $\mathscr{C}_{l-1}$, r_l from line 13.7 and r_{l-1} from line 13.5, which are based on $v_{l-1}^* \neq v_{l-1}$, are not correct, any more.

One question remains to be answered before the algorithm is complete: on which parts of Ω_l and Ω_{l-1} do the newly introduced residuals r_l and r_{l-1} have to be computed? Since d_{l-1} can be computed in the traditional way in most parts of $\mathscr{C}_{l-1}$, it is clearly not necessary to compute r_l and r_{l-1} everywhere on $\mathscr{C}_l$ and $\mathscr{C}_{l-1}$, respectively.

The first step is to find out where d_{l-1} can be computed the traditional way, and where d_{l-1} has to be computed via (4.14). Provided that $\overline{r}_l$ is available on $\mathscr{C}_l \cup \mathscr{R}_l$, but not on $\mathscr{R}_l^E$, $d_{l-1} = R_l^{l-1}\overline{r}_l$ can be computed on $\mathscr{C}_{l-1} \cap \mathscr{C}_l$, but not on $\mathscr{C}_{l-1} \cap \mathscr{R}_l$. In order to make (4.14) more conveniently usable in Algorithm 12, we define

$$r_l \;=\; \overline{r}_l \;=\; 0 \quad \textbf{on}\; \Omega_l \setminus (\mathscr{C}_l \cup \mathscr{R}_l)\,,$$

i. e., in those regions where $\overline{r}_l$ is not used, anyway. Now $\overline{r}_l[j] - r_l[j] = 0$ for all j in $N_l^P(i)$, and the sum over $j \in N_l^P(i)$ vanishes for all points i on Ω_l, and thus (4.14) can be written as

$$d_{l-1}[i] \;=\; r_{l-1}[i] + \sum_{j \in N_l^S(i)} \left(w_{ij}\left(\overline{r}_l[j] - r_l[j]\right)\right) + \sum_{j \in N_l^P(i)} \left(w_{ij}\left(\overline{r}_l[j] - r_l[j]\right)\right),$$

This variant formally includes all couplings of the restriction operator again. The two separate sums can be combined to a single sum over all $j \in N_l(i)$, which is equivalent to the restriction operator R_l^{l-1}. The indices can be removed, and we obtain

$$d_{l-1} \;=\; r_{l-1} + R_l^{l-1}\left(\overline{r}_l - r_l\right).$$

The defect computation in Algorithm 12 (line 12.7) can now be written concisely as

$$d_{l-1} \;=\; \begin{cases} R_l^{l-1}\overline{r}_l & \textbf{on}\; \mathscr{C}_{l-1} \cap \mathscr{C}_l \\ r_{l-1} + R_l^{l-1}\left(\overline{r}_l - r_l\right) & \textbf{on}\; \mathscr{C}_{l-1} \cap \mathscr{R}_l\,, \end{cases}$$

provided that the residuals r_l and $\overline{r}_l$ are initialized to zero on $\mathscr{R}_l^E$ in lines 13.7 and 12.6.

For the residual on the coarse mesh, r_{l-1}, this means that it has to be computed on $\mathscr{C}_{l-1} \cap \mathscr{R}_l$ (see lines 13.5 and 12.11). The residuals on the fine mesh, r_l and $\overline{r}_l$, have to be initialized to zero on $\mathscr{R}_l^E$. In order to specify where their actual values have to be computed, we define the *internal refinement boundary* $\mathscr{R}_l^I$ as the set of all points in $\mathscr{C}_l$ that are coupled to points in $\mathscr{R}_l$. Using this definition, the required sub-mesh for computing r_l can be specified as $\mathscr{R}_l^I \cup \mathscr{R}_l$ (lines 13.7 and 12.15). $\overline{r}_l$ (line 12.6) has to be computed on $\mathscr{C}_l \cup \mathscr{R}_l^I \cup \mathscr{R}_l$, which can be reduced to $\mathscr{C}_l \cup \mathscr{R}_l$, because $\mathscr{R}_l^I \subset \mathscr{C}_l$.

Fig. 4.6 shows Algorithms 12 and 13 in the familiar diagram form. Coming from Fig. 4.5, the sub-mesh $\mathscr{R}_l^{EE}$ has been removed, and $\mathscr{R}_l^I$ has been added. Also the new residual computations have been added. The diagram completes the presentation of the adaptive full multigrid algorithm with full approximation in its final version. The implementation in HHG, described in Section 4.5, is based on this version.

Algorithm 12: The adaptive full approximation scheme.

Algorithm $\texttt{FAScycle}(A, v_l, f_l, R, \hat{R}, P, l, l_s, \nu_1, \nu_2)$
if $l = l_s$ **then**
  $v_l = \texttt{solve}(A_l v_l = f_l)$ **on** $\mathscr{C}_l = \Omega_l$
else
  $$\overline{v}_l = \begin{cases} \texttt{smooth}(A_l, v_l, f_l, \mathscr{C}_l, \nu_1) & \textbf{on } \mathscr{C}_l \\ v_l & \textbf{on } \mathscr{R}_l \cup \mathscr{R}_l^E \end{cases}$$
  $$\overline{r}_l = \begin{cases} f_l - A_l \overline{v}_l & \textbf{on } \mathscr{C}_l \cup \mathscr{R}_l \\ 0 & \textbf{on } \mathscr{R}_l^E \end{cases}$$
  $$d_{l-1} = \begin{cases} R_l^{l-1} \overline{r}_l & \textbf{on } \mathscr{C}_{l-1} \cap \mathscr{C}_l \\ r_{l-1} + R_l^{l-1}\left(\overline{r}_l - r_l\right) & \textbf{on } \mathscr{C}_{l-1} \cap \mathscr{R}_l \end{cases}$$
  $$v_{l-1}^* = \begin{cases} \hat{R}_l^{l-1} \overline{v}_l & \textbf{on } \mathscr{C}_{l-1} \cap \mathscr{C}_l \\ u_{l-1} & \textbf{on } \mathscr{C}_{l-1} \cap \overline{\mathscr{C}}_l \end{cases}$$
  $$f_{l-1} = \begin{cases} d_{l-1} + A_{l-1} v_{l-1}^* & \textbf{on } \mathscr{C}_{l-1} \cap (\mathscr{C}_l \cup \mathscr{R}_l) \\ R_l^{l-1} f_l^{\Omega} & \textbf{on } \mathscr{C}_{l-1} \cap \left(\overline{\mathscr{C}}_l \setminus \mathscr{R}_l\right) \end{cases}$$
  $v_{l-1} = \texttt{FAScycle}(A, v_{l-1}^*, f_{l-1}, R, \hat{R}, P, l-1, l_s, \mathscr{C}, \nu_1, \nu_2)$ **on** $\mathscr{C}_{l-1}$
  $r_{l-1} = f_{l-1} - A_{l-1} v_{l-1}$ **on** $\mathscr{C}_{l-1} \cap \mathscr{R}_l$
  $c_{l-1} = v_{l-1} - v_{l-1}^*$ **on** $\mathscr{C}_{l-1} \cap (\mathscr{C}_l \cup \mathscr{R}_l)$
  $$\tilde{e}_l = \begin{cases} P_{l-1}^l c_{l-1} & \textbf{on } \mathscr{C}_l \cup \mathscr{R}_l \\ P_{l-1}^l \left(v_{l-1} - v_{l-1}^*\right) & \textbf{on } \mathscr{R}_l^E \end{cases}$$
  $\tilde{v}_l = \overline{v}_l + \tilde{e}_l$ **on** $\mathscr{C}_l \cup \mathscr{R}_l \cup \mathscr{R}_l^E$
  $r_l = f_l - A_l \tilde{v}_l$ **on** $\mathscr{R}_l^I \cup \mathscr{R}_l$
  $$v_l = \begin{cases} \texttt{smooth}(A_l, \tilde{v}_l, f_l, \mathscr{C}_l, \nu_2) & \textbf{on } \mathscr{C}_l \\ \tilde{v}_l & \textbf{on } \mathscr{R}_l \cup \mathscr{R}_l^E \end{cases}$$
end
return v_l

Algorithm 13: The adaptive full multigrid algorithm, using the adaptive full approximation scheme (Algorithm 12).

Algorithm $\texttt{fullmg}(A, v, f, R, \hat{R}, P, \hat{P}, l_s, l_f, \mu, \nu_1, \nu_2)$
$u_{l_s} = \texttt{solve}(A_{l_s} u_{l_s} = f_{l_s})$ **on** $\mathscr{C}_{l_s}$
for $l = l_s + 1 \ldots l_f$ **do**
  Initialize $\mathscr{C}_l$ according to refinement criteria
  $r_{l-1} = f_{l-1} - A_{l-1} u_{l-1}$ **on** $\mathscr{C}_{l-1} \cap \mathscr{R}_l$
  $v_l = \hat{P}_{l-1}^l u_{l-1}$ **on** $\mathscr{C}_l \cup \mathscr{R}_l \cup \mathscr{R}_l^E$
  $$r_l = \begin{cases} f_l - A_l v_l & \textbf{on } \mathscr{R}_l^I \cup \mathscr{R}_l \\ 0 & \textbf{on } \mathscr{R}_l^E \end{cases}$$
  for $i = 1 \ldots \mu$ **do**
    $\texttt{FAScycle}(A, v_l, f_l, R, \hat{R}, P, l, l_s, \nu_1, \nu_2)$
  end
  $u_l = v_l$
end
return u_l

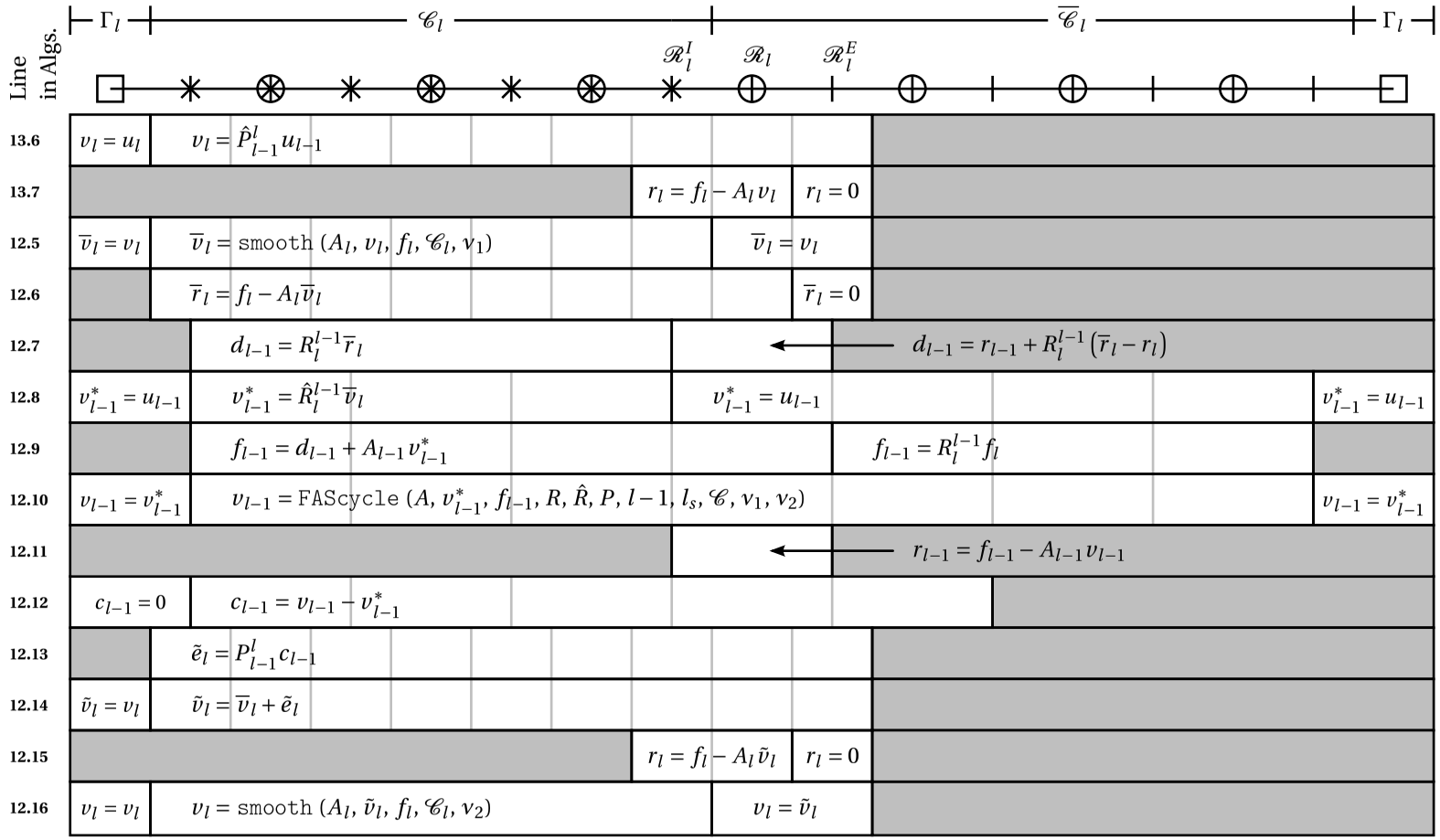

Figure 4.6: Diagram of the adaptive full multigrid algorithm with only one external refinement boundary layer.

4.4 An efficient adaptive refinement algorithm

The central goal for implementing adaptive mesh refinement in HHG is to maintain the good performance and parallel scalability of the implementation. Therefore, the design guidelines of the original implementation (see Section 2.4)—hardware-friendly data structures, communication grouping, and communication locality—are maintained in the adaptive implementation. The basis for the implementation are the full multigrid and FAS algorithms that are defined in Algorithms 13 and 12 and visualized in Fig. 4.6.

4.4.1 Error estimation

Reliably estimating the error of the computed solution is a highly application-specific task, and so is its implementation. Generally reliable methods like measuring the curvature of the solution are computationally expensive; cheap methods like evaluating the residual norm deliver entirely unfeasible results in many cases. Since the adaptive implementation presented here does not target a specific application, the error estimation was not in the focus of the work.

4.4.2 Mesh refinement

In order to understand the implementation, it is necessary to think about what it actually means to "refine a mesh". What are the implications of a finite element getting refined? Which components of the solver have to be informed about a primitive's refinement, and which data structures need to be allocated on the refined mesh?

Fig. 4.6 gives an idea of what adaptive mesh refinement means: the operators and some (not necessarily all) variables have to be defined on the fine level on different subsets of the mesh. The matter cannot be reduced to a simple statement like "a primitive is refined to a certain level." In fact, it will turn out that the *primitive* classes (listed in Fig. 2.15) do not need any level information about themselves, at all.

The main concept of HHG's AMR implementation is that the individual primitives do not know "their" refinement levels, but, instead, the mesh has a registry with pointers to the primitives in its sub-meshes on each level. This registry, which is called primitive store in HHG (the name of the corresponding class is `hhgPrimitiveStore`), has already been presented in Section 2.4.4 (see Fig. 2.14). The only extension to the implementation described there is the introduction of new primitive groups:

- `RefinementBoundary`: primitives on the refinement boundary of a certain level (denoted by $\mathscr{R}_l$ in Section 4.3).
- `InternalRefinementBoundary`: primitives in the working set (i. e., they are also in the `WorkingSet` group) and next to the refinement boundary (denoted by $\mathscr{R}_l^I$ in Section 4.3).
- `ExternalRefinementBoundary`: primitives not in the working set but next to the refinement boundary (denoted by $\mathscr{R}_l^E$ in Section 4.3).

The first step in adaptive mesh refinement is to determine for each primitive the level on which the solution has to be computed. This is done—either statically in

the beginning or dynamically after a multigrid cycle—by the error estimator. Therefore, the `hhgPrimitive` class provides the method `triggerRefn (int varIdx, int lvl)` ("trigger refinement"), which notifies the primitive that the variable with ID `varIdx` has to be refined to level `lvl`. Since the error estimator's statement is that the *solution* has to be computed on a certain level, it will use the ID of the unknowns vector as `varIdx`. Note that `triggerRefn` only *marks* the primitive for refinement, but it neither evokes any action by the primitive, nor is the information used by the primitive in any way. It may even happen, due to other constraints, that the unknowns vector on this primitive will be refined to a higher level than the one triggered. Thus, this concept does not contradict the above statement that the primitives do not know their refinement levels. The information provided with `triggerRefn` is only stored with the primitive, but used by the adaptive refinement algorithm.

The adaptive refinement algorithm is implemented in the `refine` method of the `hhgAdaptiveRefiner` class. The algorithm's two tasks are

- inserting the primitives into the appropriate groups of the primitive store, using the refinement information given to `triggerRefn`,
- maintaining refinement consistency among neighboring primitives.

While inserting the primitives into `hhgPrimitiveStore` is achieved simply by calling the store's `addPrimitive` method (see Fig. 2.14), the complicated task is maintaining consistency among the primitives—especially when the mesh is distributed over several processes.

Refinement consistency among primitives

The algorithms in Section 4.3 are based on certain properties of the mesh, which have been defined in Definition 2. In order to implement these algorithms in HHG, we have to ensure that the HHG mesh complies to these properties. Additionally, after the theoretical part, for the sake of simplicity, assumed the boundary values to be available "where needed", it now becomes necessary now to precisely identify at which boundary points Dirichlet or Neumann boundary conditions have to be computed. An HHG mesh that fulfills these properties will be called *consistent*.

The following concise set of rules will be used to construct consistent meshes in HHG. Some of the rules go beyond the requirements of Definition 2. They have been added, because they make the implementation much simpler by eliminating many special cases that would otherwise have to be considered. The rules are formulated such that they can be understood intuitively, but they are not precise enough for directly deriving a refinement algorithm from them. A more formal representation of the rules will follow in Definition 3. The term "a primitive is refined to a certain level" in the rules means that the solution is computed on the primitive at that level.

1. If the solution is computed at a certain level on an interior primitive, then it is also computed at all coarser levels on that primitive. The same applies to the computation of boundary conditions on boundary primitives.
2. Every mesh boundary primitive is refined to the maximum refinement or refinement boundary level of all neighboring interior primitives.
3. Every interior primitive is refined to the maximum common refinement level of all neighboring higher-dimensional primitives.

4. If two neighboring interior primitives are refined to different levels, the one with the lower refinement level (which is also the lower-dimensional one) is part of the refinement boundary on the higher level.
5. At a refinement boundary the degree of refinement jumps by exactly one level.
6. If two neighboring interior primitives are refined to different levels, the one with the higher refinement level is on the internal refinement boundary.
7. If an interior primitive has the same refinement level as an adjacent primitive on the refinement boundary, but is not on the refinement boundary itself, it is on the external refinement boundary.

Rule 1 states that no refinement levels are skipped. It is one of the rules that are not strictly necessary to establish the mesh properties claimed in Section 4.3. Its purpose is to simplify the mesh refinement process and the implementation of the algorithms.

Rule 2 ensures that the boundary primitives are sufficiently refined. If an interior primitive is refined to level l, it requires the adjacent boundary primitives to be refined to level l, too, namely in the smoothing and residual calculation steps of the multigrid algorithms. Interior primitives that are on the refinement boundary of level l also need data on level l from their adjacent boundary primitives during the residual calculation.

Note that the formulation "all neighboring interior primitives" can be restricted to "all higher-dimensional neighboring interior primitives", because boundary primitives do not have lower-dimensional neighbors in the interior of the mesh. Considerations like this one will be used later on to improve the efficiency of the refinement algorithm's implementation.

Rule 3 defines the location of the refinement boundary. It states that, for example, a face between two elements that are refined to level l and $l-1$, respectively, will be refined to level $l-1$. In other words, primitives on the refinement boundary between $\mathscr{C}_l$ and $\mathscr{C}_{l-1}$ are in $\mathscr{C}_{l-1}$, but not in $\mathscr{C}_l$.

Implicitly, the rule also makes an important statement about the primitives *inside* a refined area: if all higher-dimensional neighbors of a primitive are refined to level l, the primitive itself will also be refined to l. This means that there are no infinitesimally small "holes" in a refined area, for example, a plane refined to $l-1$ within a volume refined to l. That makes sense both in theory and in practice. Mathematically, a small error in an infinitesimally small region within a region with large error is likely an artifact, and it is not a sufficient reason for treating that region on a coarser resolution. In the implementation, the low refinement of a plane within a volume of high refinement would not save a significant amount of floating point operations—quite the contrary: it would unnecessarily increase the number of primitives that need expensive special treatment.

We read the rule as a logical equivalence ("if and only if"). Therefore, it has another implication: the refinement of higher-dimensional primitives also depends on their lower-dimensional neighbors' refinement. Assume, for example, that a high error is measured at a point on a face, inducing an increase in refinement from level $l-1$ to level l on that face. Then, since the face is, according to the rule, refined to the "maximum common level of all higher-dimensional neighbors", all the elements around the face will have to be refined to level l, too, in order to make the mesh consistent. More concisely: refinement of a primitive always induces refinement of its higher-dimensional neighbors.

Rule 4 identifies the primitives on the refinement boundary, i. e., on $\mathscr{R}_l$. In compliance with Section 4.3, primitives on $\mathscr{R}_l$ are those neighbors of primitives in $\mathscr{C}_l$ that are not in $\mathscr{C}_l$, themselves. As explained in the discussion of rule 3, they also have a lower dimension than their neighbors in $\mathscr{C}_l$.

Rule 5 forbids differences in refinement of more than one level between neighboring primitives. This property is not enforced by the adaptive refinement theory. However, in order to treat the refinement boundaries efficiently, every jump size would have to be implemented explicitly as a special case, and error-prone manual calculations of the involved prolongation and restriction functions would have to be done. Therefore, we restrict the jump size to one.

Rule 6 defines the location of the internal refinement boundary. Note that, in contrast to the refinement boundary, the internal refinement boundary does not protrude into the mesh boundary.

Rule 7 defines the location of the external refinement boundary. It relies on the refinement boundary primitives already being identified, i. e., rules 4 and 5 being fulfilled. Like the internal refinement boundary, the external refinement boundary does not protrude into the mesh boundary.

To establish a consistent HHG mesh, primitives have to be added to groups in the primitive store. To keep the above rules readable, the HHG primitive groups were not explicitly be mentioned, there. The formulations used in the rules have to be interpreted in the following way.

- An *interior* primitive is a primitive with the flag `InteriorPrimitive` set. A primitive on the *mesh boundary* has the flag `BoundaryPrimitive` set.
- An interior primitive *refined to* level l is in `WorkingSet` of level l, but not in `WorkingSet` of any level $l^* > l$. A primitive having the *refinement level* l means the same. A mesh boundary primitive *refined to* level l is in `DirichletBoundary` or `NeumannBoundary` of level l, depending on its boundary condition type, but not in `DirichletBoundary` or `NeumannBoundary` of any level $l^* > l$.
- A primitive on the *refinement boundary* of level l is in `RefinementBoundary` of level l.
- A primitive on the *internal* or *external refinement boundary* of level l is in `InternalRefinementBoundary` or `ExternalRefinementBoundary` of level l, respectively.

Since the rules are too informal to serve as a basis for creating a correct mesh refinement algorithm, we formalize them in the following definition.

Definition 3 (Consistent refinement of HHG meshes)**.** *Let p denote a mesh primitive. Let $N(p)$ denote the "neighborhood" of p, i. e., the set of all primitives adjacent to p. Let $N^-(p)$ and $N^+(p)$ denote the subsets of $N(p)$ that contain only primitives with a lower or, respectively, higher dimension than p.*

A refinement of a mesh is consistent, if the following criteria are fulfilled for all levels $l > 0$.

1. (a) $\forall\ p \in \mathscr{C} = \bigcup_{l \geq 0} \mathscr{C}_l : \quad p \in \mathscr{C}_l \quad \Rightarrow \quad p \in \mathscr{C}_{l-1}$

   (b) *The same applies to* Γ, Γ^D, *and* Γ^N.

2. (a) $\forall\ p \in \Gamma : \quad N(p) \cap (\mathscr{C}_l \cup \mathscr{R}_l) \neq \emptyset \quad \Rightarrow \quad p \in \Gamma_l$

(b) *The same applies to* Γ^D *and* Γ^N.

3. (a) $\forall\, p \in \mathscr{C} : \quad p \in \mathscr{C}_l \quad \Rightarrow \quad \forall\, q \in N^+(p) : q \in \mathscr{C}_l$
   (b) $\forall\, p \in \mathscr{C} : \quad p \notin \mathscr{C}_l \quad \Rightarrow \quad \exists\, q \in N^+(p) : q \notin \mathscr{C}_l$
4. $\forall\, p \in \mathscr{C} : \quad N^+(p) \cap \mathscr{C}_l \neq \emptyset \;\wedge\; p \notin \mathscr{C}_l \quad \Rightarrow \quad p \in \mathscr{R}_l$
5. $\forall\, p \in \mathscr{R} = \bigcup_{l \geq 0} \mathscr{R}_l : \quad p \notin \mathscr{C}_l \quad \Rightarrow \quad \forall\, q \in N(p) : q \notin \mathscr{C}_{l+1}$
6. $\forall\, p \in \mathscr{C} : \quad N^-(p) \cap \mathscr{R}_l \neq \emptyset \;\wedge\; p \notin \mathscr{R}_l \quad \Rightarrow \quad p \in \mathscr{R}_l^I$
7. $\forall\, p \in \mathscr{C} : \quad N(p) \cap \mathscr{R}_l \neq \emptyset \;\wedge\; p \notin (\mathscr{C}_l \cup \mathscr{R}_l) \quad \Rightarrow \quad p \in \mathscr{R}_l^E$

Definition 3 gives the properties of a consistent adaptive mesh refinement, but it does not provide a construction scheme for a refinement algorithm. An implementation of adaptive refinement must translate the above criteria into algorithmic statements. It would be possible to construct a refinement algorithm from the properties in the definition in a straightforward—although not trivial—way by using loops to implement the "$\forall$" quantifiers and conditional branches to implement the "$\exists$" quantifiers and "$\Rightarrow$" inferences. Such a straightforward implementation would be neither computationally efficient nor easy to understand or even verify. Therefore, the defined criteria will be transferred into a set of formal rules that represent the criteria exactly but provide a more suitable foundation for a refinement algorithm.

Rules for the refinement algorithm

One central property of the refinement algorithm is already clear with Definition 3: just like the numerical algorithms described in Section 2.4.4, it will have to iterate over groups of mesh primitives. Therefore, the same problems concerning local and remote data dependencies between primitives arise: if a primitive changes its refinement level, how are the neighboring primitives—which may even belong to a different process—notified about that change? The established mechanisms for handling these data dependencies can be re-used. Like in the numerical algorithms, updates between different processes have to be grouped in order to use the communication infrastructure efficiently.

In order to re-use the communication infrastructure that is already available in HHG, we choose the same algorithmic structure for the refinement algorithm as for the numerical algorithms. This structure (explained in detail in [10], Section 8.11) has two central concepts.

- Before and after an operation is performed on a primitive, the local data dependencies are updated.
- The operations and local updates are performed for all primitives with the same dimension. Then, the remote dependencies are updated for all process boundary primitives of that dimension. This pattern is repeated for all primitive dimensions.

By convention, higher-dimensional primitives are responsible for updating data dependencies with their lower-dimensional neighbors. Algorithm 14 implements this structure in its very basic form for adaptive mesh refinement. It takes the following arguments:

D The mesh dimension, i. e., the dimension of the highest-dimensional primitives.

$\mathscr{G}$ A group of primitives. $\mathscr{G}|_d$ indicates the group's primitives with dimension d.

Algorithm 14: Basic structure used for modifying and propagating refinement information.

```
14.1  Algorithm Basic refinement sweep(D, G)
14.2  Select primitive group G.
14.3  for d = 0...D do
14.4      foreach p ∈ G|_d do
14.5          Gather refinement information from all q ∈ N⁻(p).
14.6          Modify refinement of p, depending on neighborhood refinement.
14.7          Copy modified refinement information to all q ∈ N⁻(p).
14.8      end
14.9      Update remote dependencies between process boundary primitives
          with dimension d and their higher-dimensional neighbors
14.10 end
14.11 Update remote dependencies between process boundary primitives with
      dimension 1...D and their lower-dimensional neighbors
```

Several characteristics of the rules in Definition 3 conflict with their efficient implementation in the style of Algorithm 14. There are four categories of problems.

- The primitive groups that have to be iterated over are too large for maintaining an optimal complexity of the refinement algorithm.
- Splitting iterations over groups by primitive dimension is not possible.
- Rules modify primitives that could belong to a different process.
- Some inferences are redundant.

The following paragraphs show how to transform the rules in Definition 3 in order to resolve these problems. The transformed rules, which will then be used to construct an efficient refinement algorithm, are summarized in Lemma 1.

An iteration over all primitives in a group (e. g., "$\forall\ p \in \mathscr{C}$"), as it occurs in several rules, can be considered inefficient in general, but in some cases it even leads to the full multigrid solver dropping out of the class of optimal algorithms, which would completely annihilate the advantage of adaptive refinement. One prerequisite for the optimal complexity of full multigrid is that the number of mesh points on a refinement level is an exponential function of the level, since the amount of numerical work is proportional to that number [39]. If the mesh refinement algorithm needed to iterate over all primitives, independent of their refinement levels, the amount of work performed in refinement would not be bound to the decreasing number of mesh points on the finer levels, but would grow faster than the amount of numerical work. To avoid this, the iterations over all primitives in a group are replaced by iterations over subsets of the groups that contain only primitives on specific levels. For example, "$\forall\ p \in \mathscr{C}$" in rule 1a of Definition 3 is replaced with "$\forall\ p \in \mathscr{C}_l$" in rule 1 of Lemma 1. Additional logic in the refinement algorithm has to identify the levels l

on which the rule may not be fulfilled and, thus, over which groups $\mathscr{C}_l$ the algorithm has to iterate.

Iterating over all the neighbors of a primitive ("$\forall\ q \in N(p)$", in rule 5) prevents the grouping of primitives by dimension, which is necessary for an efficient implementation. Even though the rule may be fulfilled for primitive p with dimension d at the point of iterating over it, at a later point in the iteration some neighbor $q \in N(p)$ with a dimension $d^* > d$ may change its status, leading to the rule being violated for p, again. Thus, the entire iteration would have to be repeated. The same problem exists for statements that check for the existence of primitives in groups ("$N(p) \cap \mathscr{G} \neq \emptyset$"). The solution is to transform or to split rules such that instead of all neighbors only lower- or higher-dimensional neighbors ($N^-(p)$ or $N^+(p)$) are referred to. Then, the refinement algorithm can iterate over the primitives in dimension-ascending or -descending order, and the rules refer only to primitives that have already been passed in the iteration.

Due to parallel computing constraints, it may be impossible in practice for the process owning a primitive p to affect all neighboring primitives' refinement statuses ("$\forall\ q \in N^+(p) :\ q \in \ldots$"), like, e. g., in rule 3. In parallel computing, a neighbor q may belong to a different process, and only that process can add it to its local primitive sets. Therefore, the affected rules have to be re-written such that only the refinement status of the primitive currently iterated over is modified, but not the refinement status of its neighbors.

Since the rules usually require iterating over all the neighbors of a—possibly large—set of primitives, they cause a lot of computational work that can not even be reduced by the usual software optimization techniques. Therefore, it is important to eliminate duplicate or redundant inferences, not necessarily for obtaining a correct algorithm, but necessarily for making it fast. In rule 2, for example, Γ_l^D and Γ_l^N are set according to the same inference as Γ_l. Instead of computing the same inference three times for each primitive, it is sufficient to compute rule 2a and replace rule 2b with

$$\begin{aligned} &\forall\ p \in \Gamma^D : \quad p \in \Gamma_l \quad \Rightarrow \quad p \in \Gamma_l^D \text{ and} \\ &\forall\ p \in \Gamma^N : \quad p \in \Gamma_l \quad \Rightarrow \quad p \in \Gamma_l^N . \end{aligned}$$

A set of rules that respects these constraints is provided in Lemma 1. Since there is no compulsory scheme on how to transform the rules in Definition 3, it is not the only feasible rule set that one could construct. However, this one fits into the HHG framework especially well.

Lemma 1. *The notation of Definition 3 applies. The term $\mathscr{G}|_d$ denotes all primitives with dimension d in group $\mathscr{G}$.*

A mesh refinement is consistent according to Definition 3, if the following rules are fulfilled on all levels l in the mesh.

1. $\forall\ p \in \mathscr{C}_l : \quad p \in \mathscr{C}_{l-1}$

2. *(a)* $\forall\ p \in \Gamma_l : \quad N^+(p) \cap (\mathscr{C}_{l+1} \cup \mathscr{R}_{l+1}) \neq \emptyset \quad \Rightarrow \quad p \in \Gamma_{l+1}$

   (b) $\forall\ p \in \Gamma_l : \quad p \in \Gamma^D \quad \Rightarrow \quad p \in \Gamma_l^D$

   The same applies to Γ^N.

3. *(a)* $\forall\ p \in \mathscr{C}_l|_{d>0} : \quad N^-(p) \cap \mathscr{C}_{l+1} \neq \emptyset \quad \Rightarrow \quad p \in \mathscr{C}_{l+1}$

(b) $\forall\ p \in \mathscr{C}_l|_{d<D}: \quad \left(\forall\ q \in N^+(p):\ q \in \mathscr{C}_{l+1}\right) \quad \Rightarrow \quad p \in \mathscr{C}_{l+1}$

4. *(a)* $\forall\ p \in \mathscr{C}_l|_{d<D} \setminus \mathscr{C}_{l+1}: \quad N^+(p) \cap \mathscr{C}_{l+1} \neq \emptyset \quad \Rightarrow \quad p \in \mathscr{R}_{l+1}$
   (b) $\forall\ p \in \mathscr{C}_l|_{d<D-1}: \quad N^+(p)\ \cap \mathscr{R}_{l+1} \neq \emptyset \quad \Rightarrow \quad p \in \mathscr{R}_{l+1}$

5. $\forall\ p \in \mathscr{C}_l|_{d<D}: \quad N^+(p) \cap \mathscr{C}_{l+2} \neq \emptyset \quad \Rightarrow \quad p \in \mathscr{C}_{l+1}$

6. $\forall\ p \in \mathscr{C}_l|_{d>0} \setminus \mathscr{R}: \quad N^-(p) \cap \mathscr{R}_l \neq \emptyset \quad \Rightarrow \quad p \in \mathscr{R}_l^I$

7. $\forall\ p \in \mathscr{C}_l|_{d>0} \setminus \mathscr{R}: \quad N^-(p) \cap \mathscr{R}_{l+1} \neq \emptyset \quad \Rightarrow \quad p \in \mathscr{R}_{l+1}^E$

Some of the transformations made to obtain a rule in Lemma 1 from a rule in Definition 3 are straightforward and intuitive. For other rules the origins are not so clear or the transformations need justification beyond the arguments mentioned above.

The new rules (except rule 1, which will be dealt with in the next paragraph) are set up such that the same group is modified only on levels higher than the current iteration levels and that other groups are modified on levels at least as high as the iteration level. Thus, it is sufficient to set the coarsest iteration level, let us call it L_c, to the finest level on which no primitive groups have been modified by the refinement algorithm.

The only exception is rule 1, which can have the effect of primitives being added to some $\mathscr{C}_l$ with $l < L_c$. If this happens, L_c will have to be set to the coarsest level that has been affected by rule 1, and all the rules that depend on $\mathscr{C}_l$ will have to be re-applied.

In Lemma 1, an explicit match for rule 1b in Definition 3 is missing. As a contribution to eliminating redundant inferences, this rule has been integrated into rule 2 of Lemma 1. Rule 2a states that boundary primitives are refined to the same level as neighboring interior primitives. Therefore, if $p \in \mathscr{C}_l \Rightarrow p \in \mathscr{C}_{l-1}$ is valid for all interior primitives, then $p \in \Gamma_l \Rightarrow p \in \Gamma_{l-1}$ is valid for all boundary primitives. Rule 2b ensures the same for the groups Γ^D and Γ^N.

In rule 3a the infeasible changing of neighbors' refinement statuses has been eliminated. With L_c available, this is easy: instead of adding the $q \in N^+(p)$ to $\mathscr{C}_l$ once p gets added to $\mathscr{C}_l$, it is equivalent to wait until q is the current iteration primitive and add q to $\mathscr{C}_l$ if any of its lower-dimensional neighbors are in $\mathscr{C}_l$. A necessary precondition for this equivalence is that the q that have to be added are actually included in the iteration. If L_c is set as described above, this condition is fulfilled.

In the transformation of rule 3b we replaced the argument "$A \Rightarrow B$" with the converse argument "$\neg B \Rightarrow \neg A$". In our case,

$$\left(p \notin \mathscr{C}_l \Rightarrow \exists\ q \in N^+(p) : q \notin \mathscr{C}_l\right) \Leftrightarrow \left(\neg\left(\exists\ q \in N^+(p) : q \notin \mathscr{C}_l\right) \Rightarrow p \in \mathscr{C}_l\right), \qquad (4.15)$$

where

$$\neg\left(\exists\ q \in N^+(p) : q \notin \mathscr{C}_l\right) \Leftrightarrow \forall\ q \in N^+(p) : q \in \mathscr{C}_l\,.$$

The scope of the refinement boundary has been expanded beyond the scope of Definition 3. Lemma 1 contains an additional rule, 4b, which makes *all* lower-dimensional neighbors of a refinement boundary primitive also refinement boundary primitives. With the additional rule, every refinement boundary is ensured to be a continuous $(D-1)$-dimensional surface within the computational domain. While that property is not strictly necessary for refinement consistency, it makes the implementation much easier, because it eliminates special cases that would otherwise have to be considered.

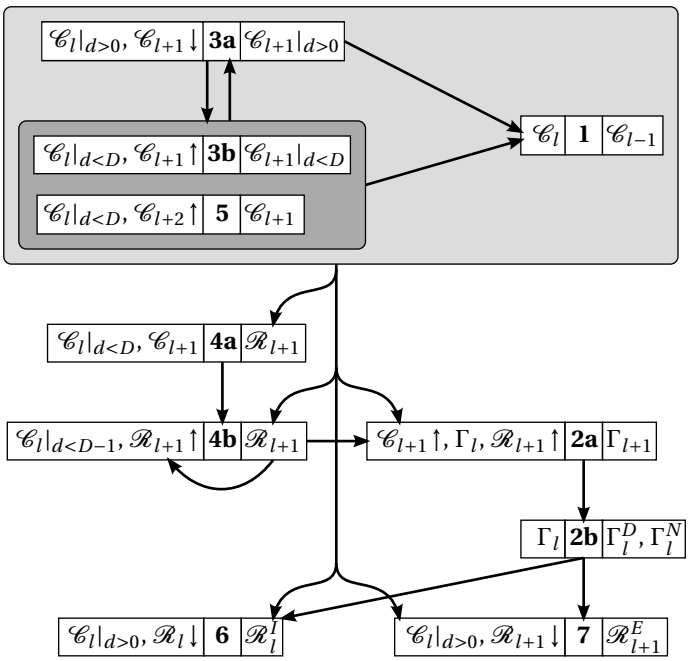

Figure 4.7: Dependencies between refinement rules.

The refinement algorithm

The requirements for an efficient parallel implementation are fulfilled by the rules in Lemma 1. However, the rules are interdependent. This means that the order of applying them to the mesh cannot be chosen arbitrarily, but has to take these dependencies into account. For example, rule 2a for setting up Γ_{l+1} relies on $\mathscr{C}_{l+1}$ and $\mathscr{R}_{l+1}$ being set up correctly by rules 3 and 4.

The dependencies are visualized in Fig. 4.7. The figure annotates each rule with the primitive groups it depends on (shown in the left box of each rule) and the primitive groups it modifies (shown in the right box). Not shown are dependencies on the immutable, level-independent groups $\mathscr{C}$, Γ, Γ^D, and Γ^N. For a compact notation, arrows are used to indicate dimensions. An arrow pointing downwards ($\downarrow$) means that the rule only depends on lower-dimensional primitives of the respective group, an arrow pointing upwards ($\uparrow$) indicates higher-dimensional primitives of the respective group. The arrows between the rules indicate dependencies. An arrow pointing from rule A to rule B means that rule B depends on modifications of primitive groups caused by rule A. Thus, whenever rule A is applied to the mesh, rule B will have to be applied afterwards. On the other hand, if no arrow or sequence of arrows leads from rule A to rule B, then rule B does not have to be applied after rule A.

Some rules have circular dependencies, though.

- Rules 3a and 3b depend on each other.
- Rule 5 is tightly coupled with rule 3b.
- Rule 4b depends on itself being established for all primitives with a higher dimension.

These rules have to be applied repeatedly. Rules 3a, 3b, and 5 have to be applied alternately until neither of the rules causes changes to any primitive groups, any more. During this process, L_c may change, too. Rule 4b has to be applied first to primitives of dimension $D-2$ and then to primitives with decreasing dimension, until the lowest dimension is reached.

Algorithms 15–20 implement the rules in an imperative programming style. They are kept as simple as possible, at this point, and do not contain any parallel programming constructs. Algorithm 15 is the central function. It takes the following arguments:

- D The mesh dimension, i. e., the dimension of the highest-dimensional primitives.
- L_t Target refinement levels for all primitives, determined by the error estimator. $L_t(p) = l$ means that on primitive p the solution has to be computed on level l.
- L_r Refinement boundary target levels, used internally by the refinement algorithm. $L_r(p) = l$ means that primitive p is on the refinement boundary of level l.
- $\mathscr{C}$ The set of primitives on which the solution has to be computed. The subset $\mathscr{C}_l$ contains all primitives on which the solution has to be computed on level l. The same applies for the following primitive sets.
- Γ The mesh boundary primitives.
- Γ^D The Dirichlet boundary primitives.
- Γ^N The Neumann boundary primitives.
- $\mathscr{R}$ The refinement boundary primitives.
- $\mathscr{R}^I$ The internal refinement boundary primitives.
- $\mathscr{R}^E$ The external refinement boundary primitives.

When the algorithm is called, the mesh (represented by the sets $\mathscr{C}$, Γ, Γ^D, Γ^N, $\mathscr{R}$, $\mathscr{R}^I$, and $\mathscr{R}^E$) must be refined consistently (e. g., uniformly to some level). Then, the algorithm returns a consistent refinement with all primitives refined to the levels given by L_t. It applies the above rules in the order induced by the dependencies shown in Fig. 4.7 by calling sub-algorithms that apply one or more rules at a time. In the beginning, all refinement boundary primitive groups at levels greater than L_c are cleared. They will be set up from scratch in `AddRefnBndry` (Algorithm 18). This rather radical approach is taken despite an increase in add operations on primitive groups, because in practice it is in fact quicker than having to check whether a primitive is already contained in a group before adding it. The loop over `InteriorUpSweep` and `InteriorDownSweep` (Algorithms 16 and 17) repeats as long as any of the two algorithm makes changes to the primitive groups ($b_1 \vee b_2 =$ **True**). After that, the algorithms for setting up the refinement boundaries (`AddRefnBndry` and `AddSecondaryRefnBndry`, i. e., Algorithms 18 and 20) and for adding the mesh boundary (`AddMeshBndry`, i. e., Algorithm 19) are called in the order shown in Fig. 4.7.

The first sub-algorithm, `InteriorUpSweep` (Algorithm 16), adds primitives to the sets $\mathscr{C}_l$ according to their target refinement levels given by L_t. It only applies the

Algorithm 15: The basic refinement algorithm.

```
15.1  Algorithm BasicRefiner(D, L_t, L_r, 𝒞, Γ, Γ^N, ℛ, ℛ^I, ℛ^E)
15.2  do
15.3      foreach l | l > L_c ∧ 𝒞_l ≠ ∅ do
15.4          ℛ_l = ∅, ℛ_l^I = ∅, ℛ_l^E = ∅
          // Apply rules 1 and 3a
15.5      (b_1, L_c, 𝒞, L_r) = InteriorUpSweep(D, L_c, L_t, 𝒞)
          // Apply rules 1, 3b, and 5, prepare rule 4
15.6      (b_2, L_c, 𝒞, L_r) = InteriorDownSweep(D, L_c, L_t, L_r, 𝒞)
15.7  while b_1 ∨ b_2 = True
      // Apply rules 4a and 4b.
15.8  ℛ = AddRefnBndry(D, L_c, 𝒞, ℛ)
      // Apply rules 2a and 2b.
15.9  (Γ, Γ^D, Γ^N) = AddMeshBndry(D, L_c, Γ, Γ^D, Γ^N)
      // Apply rules 6 and 7
15.10 (ℛ^I, ℛ^E) = AddSecondaryRefnBndry(D, L_c, 𝒞, ℛ, ℛ^I, ℛ^E)
15.11 return (𝒞, Γ, Γ^N, ℛ, ℛ^I, ℛ^E)
```

first part of rule 3, in which the primitives' refinement levels depend on their lower-dimensional neighbors' levels. For this property the algorithm is given the name "up sweep". It is separated from the application of rule 3b, the "down sweep", in view of a parallelization according to Algorithm 14. If the refinement level of a primitive increases ($L_t(p) > L_m$), the algorithm ensures that rule 1 is satisfied on that primitive. In this case it is also necessary to reset $L_r(p)$, because it is unknown at this point whether the primitive will be on a refinement boundary. If necessary, L_c is decreased, so all primitives that have to be modified will be included in the iteration. In this case, `InteriorUpSweep` has to be re-run, which is signaled to Algorithm 15 by setting $b_1 =$ **True**.

`InteriorDownSweep` (Algorithm 17) is similar to Algorithm 16, but it iterates over the dimensions from $D-1$ to 0, because rule 3b depends on higher-dimensional neighbors. By including $L_m^+ - 1$ in the calculation of $L_t(p)$ the algorithm also ensures that rule 5 is observed.

At this point it is already possible to prepare the setup of $\mathscr{R}$ (which will be finalized in Algorithm 18) by initializing $L_r(p)$ according to rules 4a and 4b. Note that even if $L_r(p)$ is set in line 17.17, it may be reset again in a later iteration. Still, having possibly several sets and resets of $L_r(p)$ is more efficient than spending an entire iteration over many primitives afterwards only for initializing $L_r(p)$.

The remaining sub-algorithms are, thanks to the work done in the first part, much simpler. For adding the primitives to $\mathscr{R}$, `AddRefnBndry` (Algorithm 18) can use the L_r already set up in the preceding sub-algorithms. `AddMeshBndry` (Algorithm 19) sets up Γ_l, Γ_l^D, and Γ_l^N according to rule 2. Last but not least, the internal and external refinement boundaries are set up in `AddSecondaryRefnBndry` (Algorithm 20) according to rules 6 and 7.

Algorithm 16: Refine interior primitives, up sweep: apply rules 1 and 3a.

```
16.1  Algorithm InteriorUpSweep(D, L_c, L_t, 𝒞)
16.2  b_1 = False
16.3  for d = 0...D do
16.4      foreach p ∈ 𝒞_l|_d do
16.5          if d = 0 then
16.6              L_m^- = 0
16.7          else
                  // Rule 3a
16.8              L_m^- = max(l | N^-(p) ∩ 𝒞_l ≠ ∅)
16.9              L_a^- = max(l | ∀ q ∈ N^-(p): q ∈ 𝒞_l)
16.10             if L_a^- < L_c then
16.11                 L_c = L_a^-
16.12                 b_1 = True
16.13         L_t(p) = max(L_m^-, L_t(p))
16.14         L_m = max(l | p ∈ 𝒞_l)
16.15         if L_t(p) > L_m then
                  // Rule 1
16.16             for l = L_m + 1...L_t(p) do
17                    𝒞_l = 𝒞_l ∪ {p}
18                L_r(p) = 0
19 return (b_1, L_c, 𝒞, L_r)
```

Algorithm 17: Refine interior primitives, down sweep: apply rules 1, 3b, and 5, prepare rule 4.

17.1 **Algorithm** `InteriorDownSweep`$(D, L_c, L_t, L_r, \mathscr{C})$
17.2 $b_2 =$ **False**
17.3 **for** $d = D-1 \ldots 0$ **do**
17.4 **foreach** $p \in \mathscr{C}_l|_d$ **do**
  `// Rules 3b and 5`
17.5 $L_m^+ = \max\big(l \mid N^+(p) \cap \mathscr{C}_l \neq \emptyset\big)$
17.6 $L_a^+ = \max\big(l \mid \forall\ q \in N^+(p) :\ q \in \mathscr{C}_l\big)$
17.7 $L_t(p) = \max\big(L_a^+,\ L_m^+ - 1,\ L_t(p)\big)$
17.8 $L_m = \max\big(l \mid p \in \mathscr{C}_l\big)$
17.9 **if** $L_t(p) > L_m$ **then**
   `// Rule 1`
17.10 **for** $l = L_m + 1 \ldots L_t(p)$ **do**
17.11 $\mathscr{C}_l = \mathscr{C}_l \cup \{p\}$
17.12 $L_r(p) = 0$
17.13 $b_2 =$ **True**
17.14
17.15 $L_r^+ = \max\big(l \mid \big(\exists\ q \in N^+(p) \mid L_r(q) = l\big)\big)$
17.16 **if** $L_m^+ > L_t(p)\ \vee\ L_r^+ > 0$ **then**
17.17 $L_r(p) = \max\big(L_m^+,\ L_r^+\big)$
17.18 **if** $L_t(p) < L_r(p) - 1$ **then**
17.19 $L_t(p) = L_r(p) - 1$
17.20 $b_2 =$ **True**
17.21 **return** $(b_2,\ L_c,\ \mathscr{C},\ L_r)$

Algorithm 18: Add refinement boundary primitives, i. e., apply rules 4a and 4b.

18.1 **Algorithm** `AddRefnBndry`$(D, L_c, \mathscr{C}, \mathscr{R})$
18.2 **for** $d = 0 \ldots D-1$ **do**
18.3 **foreach** $p \in \mathscr{C}_{L_c}|_d$ **do**
18.4 **if** $L_r(p) > L_c$ **then**
18.5 $\mathscr{R}_{L_r(p)} = \mathscr{R}_{L_r(p)} \cup \{p\}$
18.6 **return** $\mathscr{R}$

Algorithm 19: Add mesh boundary primitives, i. e., apply rules 2a and 2b.

19.1 **Algorithm** `AddMeshBndry`$(D, L_c, \Gamma, \Gamma^D, \Gamma^N)$
19.2 **for** $d = 0 \dots D-1$ **do**
19.3 **foreach** $p \in \Gamma_{L_c}|_d$ **do**
19.4 $L_m = \max\left(l \mid p \in \Gamma_l\right)$
19.5 $L_m^+ = \max\left(l \mid N^+(p) \cap \mathscr{C}_l \neq \emptyset\right)$
19.6 $L_t(p) = L_m^+$
19.7 **if** $L_t(p) > L_m$ **then**
19.8 **if** $p \in \Gamma^D$ **then** $\Gamma_l^* = \Gamma_l^D$
19.9 **else if** $p \in \Gamma^N$ **then** $\Gamma_l^* = \Gamma_l^N$
19.10
19.11 **for** $l = L_m + 1 \dots L_t(p)$ **do**
19.12 $\Gamma_l = \Gamma_l \cup \{p\}$
19.13 $\Gamma_l^* = \Gamma_l^* \cup \{p\}$
19.14 **return** $\left(\Gamma,\ \Gamma^D,\ \Gamma^N\right)$

Algorithm 20: Add internal and external refinement boundary primitives, i. e., apply rules 6 and 7.

20.1 **Algorithm** `AddSecondaryRefnBndry`$(D, L_c, \mathscr{C}, \mathscr{R}, \mathscr{R}^I, \mathscr{R}^E)$
20.2 **for** $d = 1 \dots D$ **do**
20.3 **foreach** $p \in \mathscr{C}_{L_c}|_d$ **do**
20.4 **if** $\nexists\ l \mid p \in \mathscr{R}_l$ **then**
20.5 **if** $N^-(p) \cap \mathscr{R}_{L_t(p)} \neq \emptyset$ **then**
20.6 $\mathscr{R}^I_{L_t(p)} = \mathscr{R}^I_{L_t(p)} \cup \{p\}$
20.7 **if** $N^-(p) \cap \mathscr{R}_{L_t(p)+1} \neq \emptyset$ **then**
20.8 $\mathscr{R}^E_{L_t(p)+1} = \mathscr{R}^E_{L_t(p)+1} \cup \{p\}$
20.9 **return** $\left(\mathscr{R}^I,\ \mathscr{R}^E\right)$

4.5 Implementation in HHG

The implementation of Algorithms 15–20 in HHG requires new data structures and program code that uses them efficiently. Section 4.5.1 introduces the data structures for storing the refinement information and shows how they are used for communicating refinement information in a parallel refinement algorithm. Section 4.5.2 introduces the data structures for storing and communicating linear algebra data at refinement boundaries. Section 4.5.3 describes how the full multigrid implementation from Section 2.4 has to be transformed in order to use an adaptively refined mesh. The complete source code of the implementation is printed in Appendix A for reference.

4.5.1 The adaptive refinement algorithm

The challenges in implementing adaptive refinement in HHG efficiently are in storing the refinement information and in communicating it between processes. Since the data dependencies in the algorithms will play an important role in the following efficiency considerations, it is important to point them out clearly.

Note that the relation

$$L_t(p) = \max\left(l \mid p \in \mathscr{C}_l\right) \tag{4.16}$$

is ensured for every primitive p throughout all parts of the refinement algorithm (Algorithms 15–20). Whenever a primitive for which (4.16) does not hold is encountered, the relation is enforced immediately, before the iteration continues with the next primitive. This means that the terms in which refinement levels of neighbors are involved can be reformulated as follows.

$$\begin{aligned} L_m^- &= \max\left(l \mid N^-(p) \cap \mathscr{C}_l \neq \emptyset\right) = \max\left(l \mid \left(\exists q \in N^-(p) \,:\, L_t(q) = l\right)\right), \\ L_a^- &= \max\left(l \mid \forall\, q \in N^-(p) :\, q \in \mathscr{C}_l\right) = \min\left(l \mid \left(\exists q \in N^-(p) \,:\, L_t(q) = l\right)\right). \end{aligned}$$

The same applies analogously to L_m^+ and L_a^+. Note that the formula for calculating L_r^+ in Algorithm 17 is already in that form, i. e., it makes use of $L_r(q)$.

These transformations help in grasping the data dependencies of the refinement algorithm. They show that communicating L_t and L_r between processes is sufficient to satisfy all data dependencies of neighboring primitives.

The following actions of the refinement algorithm are performance-critical:

- determining the maximum level of a primitive in a group ($\max\left(l | p \in \mathscr{G}\right)$),
- determining the refinement status of a primitive's neighborhood (e. g., $N^-(p) \cap \mathscr{G} \neq \emptyset$ and $\forall q \in N^-(p) : q \in \mathscr{G}$),
- synchronizing the refinement information L_t and L_r of neighboring primitives among processes, and
- synchronizing the stop criterion $c_1 \vee c_2 =$ **True** of the **do**...**while** loop in Algorithm 15 among processes.

In the following, we will show how these actions are implemented in HHG.

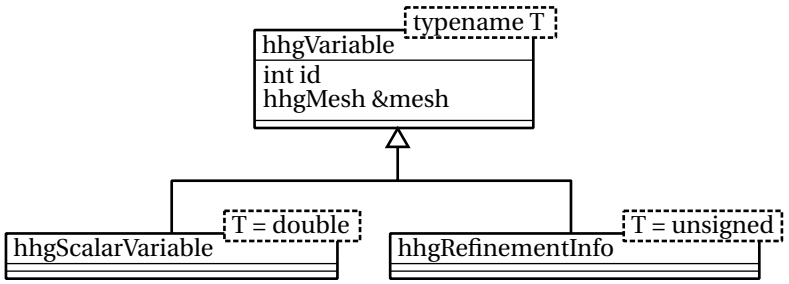

Figure 4.8: The class `hhgRefinementInfo` within its context.

Refinement information data structures

Both for accessing refinement information locally and for synchronizing it with remote processes, efficient data structures are necessary. In order to re-use HHG's existing communication infrastructure, the refinement information is interpreted as a variable on the mesh. Analogous to a variable in the linear system of equations, like the vector of unknowns, every primitive holds a part of the global refinement information. Instead of real numbers, which are usually stored in a floating point format, the variable has to store refinement levels in an unsigned integer format. Another difference is that the refinement information is not level-dependent; the primitives have to provide memory for the refinement information only once, but not for each multigrid level individually. For these reasons, a variable for refinement information consumes much less memory than a regular variable.

The new type of variable is implemented in the class `hhgRefinementInfo`. Like `hhgScalarVariable`, which was introduced in Section 2.4, it is derived from `hhgVariable`. The class diagram is shown in Fig. 4.8.

The handling of physical memory is also implemented analogous to the memory handling of the linear algebra variables. The class `hhgRefnInfoMemory`, which is derived from `hhgVariableMemory` (see Fig. 2.16), takes care of providing memory for storing a primitive's refinement information. In contrast to the linear algebra variables, the amount of memory required does not depend on the type of the primitive; every primitive type has the same kind of refinement information. Thus, `hhgRefnInfoMemory` is used directly by all primitive types.

The refinement information must contain two types of levels. The level at which the solution has to be computed on a primitive has been denoted with $L_t(p)$ in the previous section. The refinement boundary level of a primitive has been denoted with $L_r(p)$. Both $L_t(p)$ and $L_r(p)$ are accessed by neighboring primitives in the algorithms and, therefore, have to be included in `hhgRefinementInfo`. The information is spread over three class members, which are shown in Fig. 4.9.

- `myLvl` is an unsigned integer array of length 2. It contains $L_t(p)$ and $L_r(p)$.
- `lowerNeighLvls` is a vector of unsigned integer arrays of length 2. It contains $L_t(q)$ and $L_r(q)$ for all lower-dimensional neighbors q of p.
- `higherNeighLvls` has the same structure as `lowerNeighLvls` and contains $L_t(q)$ and $L_r(q)$ for p's higher-dimensional neighbors.

The computational complexity of the functions for computing L_m^-, L_a^-, L_m^+, L_a^+, and L_r^+ (see Algorithms 16–19) depends on this data layout. For computing max-

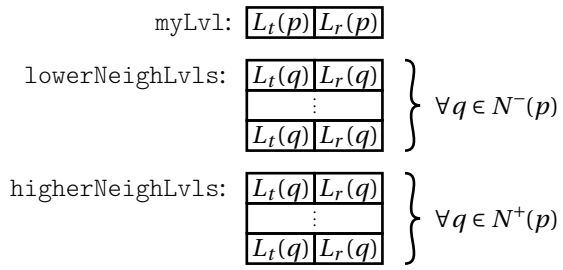

Figure 4.9: Refinement information memory layout.

ima and minima of higher and lower neighbor refinement levels, iterating over the corresponding vectors is necessary. These iterations could be eliminated by additionally storing the extrema of L_t and L_r for each vector. On the other hand, if the extrema were stored, they would have to be updated upon every modification of a vector entry. If the modification increased the maximum or decreased the minimum, this would only cost a small amount of computational effort. If, however, the modification potentially decreased the maximum or increased the minimum, the whole vector would have to be searched for the new extremum, again. Since modification of vector entries occurs much more often than computation of the above values, storing the extrema would probably not make the algorithms more efficient. A performance study concerning this topic has not yet been done; it is left to future performance optimization work.

Parallelization

The refinement algorithms have already been designed in a way that allows for reusing the established communication strategies. Both global and nearest neighbor communication is required at different points of the algorithms. The **do**...**while** loop in Algorithm 15 requires global synchronization of the stop criterion. The global synchronization step in Algorithm 15 is necessary, because a process cannot determine from locally available information whether the refinement process is still active in remote parts of the computational domain. The domain owned by a certain process may not be affected by refinement in the beginning (i. e., $c_1 \wedge c_2 =$ **False**, locally), but in a later iteration of the **do**...**while** loop the refined region may spread into the process's domain and require the process to refine its domain. Thus, a process must not exit the loop and proceed in the algorithm before the stop criterion is fulfilled globally. For the global synchronization, the `MPI_Allreduce` function is used with its *logical or* (`MPI_LOR`) operator. This way, all the processes repeat the loop as long as any of the interior sweeps on any of the processes makes changes to the mesh.

Refinement information has to be exchanged among nearest neighbors after L_t or L_r have been modified, or—seen from the opposite perspective—before this information is accessed by the neighbors. Nearest neighbor communication of refinement information is necessary at three points in the algorithm. Two of them are obvious:

- in Algorithm 16 at the end of each iteration of the loop over d, because $L_t(p)$ is modified in line 16.13 and required by higher-dimensional primitives in lines 16.8 and 16.9, and

- in Algorithm 17 at the beginning of each iteration of the loop over d, because $L_t(p)$ and $L_r(p)$ are modified in lines 17.7, 17.12, 17.19, and 17.17, and the levels of higher-dimensional primitives are required in lines 17.5, 17.6, and 17.15.

The third point is not necessary for the functionality of the refinement algorithm itself, but it has to do with the HHG way of nearest neighbor communication. In order to exchange data between processes, it is necessary that the primitives on the process boundaries are in the primitive group `hhgPgProcBndry` (see Section 2.4.4). Therefore, if refinement boundary primitives get refined (i. e., get added to some primitive group on a level l), they also have to be added to `hhgPgProcBndry` on level l. This was not included in the generic algorithms in the previous section, because it is an HHG-specific requirement. Due to this requirement, also ghost primitives that do not belong to the working set of a process have to be added to `hhgPgProcBndry` of that process. Therefore, the third point requiring nearest neighbor communication is in Algorithm 19 at the end of each iteration of the loop over d, because $L_t(p)$ is modified in line 19.6, and this information is needed by neighboring processes that have p as a ghost primitive in order to add p to `hhgPgProcBndry` of $L_t(p)$.

4.5.2 Data structures for the refinement boundary

The mesh points on the internal and external refinement boundary present a new challenge in the implementation of efficient data structures, numerical operators, and communication schemes, because they are part of large, structured regions (e. g., the interior of an element), but they have to be updated separately. Up to now, all building blocks of HHG have been designed with large structured regions in mind. Now, it becomes necessary to process only parts of these regions.

In a mesh mesh of dimension D that is consistently refined according to Definition 3, the refinement boundaries are $D-1$-dimensional *hyperplanes*. Since the interior of a primitive is always refined uniformly, these hyperplanes do not cross the interior of the D-dimensional primitives. Thus, the refinement boundary always is always comprised of primitives with dimension 0 to $D-1$. In a tetrahedral mesh, for example, the refinement boundaries always consist of vertices, edges, and faces. Also, according to the definition, there are no "holes" in the refinement boundaries, because all lower-dimensional neighbors of a primitive on a refinement boundary are also on the refinement boundary. Due to these properties, the mesh points on the internal and external refinement boundaries ($\mathscr{R}_l^I$ and $\mathscr{R}_l^E$) are always located in the interior of a primitive, directly adjacent to the primitive's *halo*.

An example is shown in Fig. 4.10. It depicts the data structure of a triangle face at refinement level 3, which has been described in detail in [10], Section 8.4.2. The figure shows only the middle plane of the face. Each box represents the memory array entry for one mesh point. The memory required for the plane is allocated as a contiguous block at the address `tri_[1][0]` in physical memory, (i. e., the memory locations of, e. g., `tri_[1][0][8]` and `tri_[1][1][0]` are adjacent in physical memory). The gray fields in the figure show the possible locations of internal or external refinement boundaries. They are comprised of the mesh points—or array entries, respectively—adjacent to the triangle's edges and vertices.

In the HHG implementation, these sets of array entries are called *hyperplanes*. The triangle in the example has one-dimensional hyperplanes (next to the triangle's edges) and zero-dimensional hyperplanes (next to its vertices). In particular,

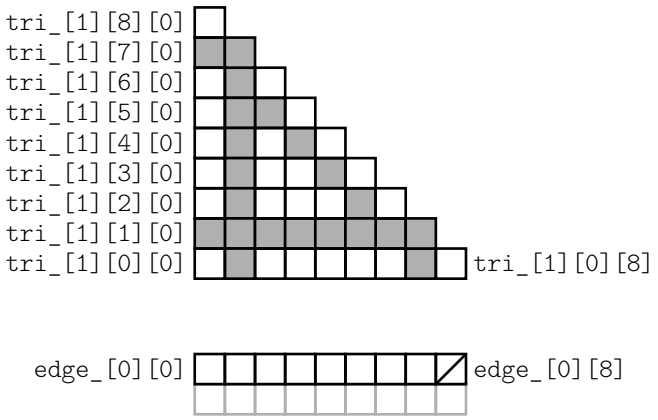

Figure 4.10: Hyperplanes in the triangle face data structure.

the one-dimensional hyperplanes are

- for edge 0 between `tri_[1][0][0]` and `tri_[1][0][8]`: the array entries `tri_[1][1][`j`]` with $0 \le j \le 7$,
- for edge 1 between `tri_[1][0][0]` and `tri_[1][8][0]`: the array entries `tri_[1][`i`][1]` with $0 \le i \le 7$, and
- for edge 2 between `tri_[1][0][8]` and `tri_[1][8][0]`: the array entries `tri_[1][`i`][`j`]` with $0 \le i \le 7$ and $j = 7 - i$.

The zero-dimensional hyperplanes are

- for vertex 0 at `tri_[1][0][0]`: the array entry `tri_[1][1][1]`,
- for vertex 1 at `tri_[1][0][8]`: the array entry `tri_[1][1][6]`, and
- for vertex 2 at `tri_[1][8][0]`: the array entry `tri_[1][6][1]`.

If a variable has to be refined to a certain level only at one or more hyperplanes, the values of the variable need only be computed at the mesh points of these hyperplanes, but not on at the bulk of the mesh points on that level. The savings in computational effort are large, because the number of mesh points on a hyperplane is always at least an order of magnitude smaller than the total number of interior mesh points of a primitive. The same applies to the amount of memory that can be saved by only allocating storage for the required hyperplanes. Doing any more than that would be a waste of computational resources.

Hyperplane data structures

If a primitive is supposed to store hyperplane data compactly, it cannot use the usual data structures. However, it is not necessary to invent entirely new data structures just for the hyperplanes. Instead, the data structures of lower-dimensional primitives can be used. Clearly, a triangle's one-dimensional hyperplane data fits perfectly into an edge data structure. This applies to all other primitive types, too, and the data structures for all types of hyperplanes a primitive can have are already implemented

for handling the data of lower-dimensional primitives. Thus, no new code for data structures has to be written. Only the algorithms for numerical and communication operations have to be extended for using the data structures in the intended ways.

Numerical operations on hyperplanes

As we will see in Section 4.5.3, the operations that need to work with hyperplanes are the prolongation, the restriction, and the residual calculation. All of these operations can use their standard equations for computing variable values at the hyperplane mesh points. Additionally, they only need to write results of calculations to hyperplanes; reading hyperplane values as input is not required. These properties reduce the number of special cases that need to be implemented for different types of hyperplanes. The operators only need to know,

- which mesh points belong to a hyperplane and, therefore, have to be iterated over in the computation, and
- how the array indices of a primitive's standard data structure are mapped to the array indices in its hyperplane data structure.

Both points depend on the location of a hyperplane within a primitive.

Consider, for example, the hyperplane at edge 1 in Fig. 4.10. The mapping between the (not actually allocated) original data structure `tri_` and the edge data structure `edg_` allocated for the hyperplane is

$$\texttt{edge\_[0][i]} \mathrel{\hat{=}} \texttt{tri\_[1][i][0]} \text{ for } 0 \le i \le 7.$$

The array entries indicated by i in the above equation are also the ones that have to be iterated over by the operators.

Local and remote communication

Smooth integration into the established communication schemes requires functions for copying data between hyperplanes and their adjacent mesh primitives. Since the hyperplanes correspond to mesh points in a primitive's interior, the primitive is responsible for copying the data to the adjacent lower-dimensional neighbors. Furthermore, it is advantageous to make the copy process transparent for the lower-dimensional neighbor, i. e., the neighbor does not need to know whether the data comes from a fully refined interior or from a hyperplane.

These demands are met by a new set of local communication functions that copy the complete content of a primitive to another primitive of the same type. As for the traditional local communication functions (see Fig. 2.17), there is a hard-coded implementation for every possible alignment of primitives. The functions are organized into the classes

- `hhgVertexVertexCopier` for copying between vertices,
- `hhgVEFullEdgeCopier` for copying between volume edges, and
- `hhgFaceFullFaceCopier` for copying between faces.

With the new operators for local communication a good basis for remote communication is available. In fact, no new data structures or algorithms are necessary

for communicating hyperplane data between processes. Recall that every primitive on the process boundary has its representative ghost primitive on the process on the other side of the boundary. In the HHG concept before the introduction of adaptive refinement, remote communication always transmitted the complete interior data of the primitive or complete halo layers between processes. Partial transfers of values from a primitive's interior (i. e., of a hyperplane) did not occur. Thus, they do not need to be considered in the adaptive refinement implementation.

4.5.3 The adaptive full multigrid algorithm

In order to get an overview over the additional functionality required for the implementation of the adaptive full multigrid algorithm in HHG, we compare the algorithm in pseudo-code (Algorithms 13 and 12) with its original form (Algorithm 7 using the FAS-cycle defined in Algorithm 8 instead of the V-cycle). From the differences we deduce how the existing functions have to be changed and which new functions have to be implemented. Another important question is how much memory the adaptive full multigrid algorithm uses, compared to the non-adaptive version. An analysis of the memory demand concludes this section.

Additional variables

In order to be easily comprehensible, Algorithms 12 and 13 introduced several new variables. Not all of them are necessary in the implementation, though. Some variables can be mapped to a single variable, because they do not have overlapping data dependencies. For the memory consumption it is beneficial to keep the number of variables as small as possible. However, a small increase in the number of variables, compared to the full-multigrid algorithm on uniformly refined meshes, is not avoidable.

The original implementation of uniform refinement needs three variables on the finest level (v, f, r) and four variables on the coarser levels (v, f, r, v^*). The HHG implementation of the full multigrid algorithm with adaptive refinement is shown in Algorithms 21 and 22. It needs one additional variable on the finest level (r^*) and one additional variable on the coarser levels (f^*).

Additional functionality

Algorithm 13 differs from Algorithm 7 in several points. A general difference is that all numerical operations are annotated with the mesh regions in which they have to be carried out. For their implementation, this means that all of them have to be extended such that they accept a parameter of type `hhgPrimitiveGroup`, so that they only iterate over primitives in the given groups. Besides accepting this parameter, the `solve` function (line 13.2) and the residual calculation on level $l-1$ (line 13.5) do not need any further changes. A real extension has to be implemented for the prolongation

$$v_l = \hat{P}^l_{l-1} v_{l-1} \qquad \textbf{on } \mathscr{R}^E_l$$

in line 13.6, because the target data structure is a hyperplane. The residual calculation

$$r_l = f_l - A_l v_l \quad \textbf{on } \mathscr{R}^I_l$$

Algorithm 21: Implementation of the adaptive full multigrid algorithm, using the adaptive full approximation scheme (Algorithm 22).

21.1 **Algorithm** `fullmg`$(A, v, f, g, R, \hat{R}, P, \hat{P}, l_s, l_f, \mu, \nu_1, \nu_2)$
21.2 Initialize f_{l_s} **on** $\mathscr{C}_{l_s}$
21.3 $v_{l_s} =$ `solve`$(A_{l_s} v_{l_s} = f_{l_s})$ **on** $\mathscr{C}_{l_s}$
21.4 **for** $l = l_s + 1 \dots l_f$ **do**
21.5 Initialize $\mathscr{C}_l$ according to refinement criteria
21.6 $u_{l-1} = v_{l-1}$ **on** $\mathscr{C}_{l-1} \cup \Gamma_{l-1}$
21.7 $f^*_{l-1} = f_{l-1} - A_{l-1} v_{l-1}$ **on** $\mathscr{C}_{l-1} \cap \mathscr{R}_l$
21.8 $v_l = \begin{cases} \hat{P}^l_{l-1} v_{l-1} & \textbf{on } \mathscr{C}_l \cup \mathscr{R}_l \cup \mathscr{R}^E_l \\ g_l & \textbf{on } \Gamma^D_l \end{cases}$
21.9 Initialize f_l **on** $\mathscr{C}_l$
21.10 $r^*_l = \begin{cases} f_l - A_l v_l & \textbf{on } \mathscr{R}^I_l \cup \mathscr{R}_l \\ 0 & \textbf{on } \mathscr{R}^E_l \end{cases}$
21.11 **for** $i = 1 \dots \mu$ **do**
21.12 `FAScycle`$(A, v_l, f_l, f^*, r^*, R, \hat{R}, P, l, l_s, \nu_1, \nu_2)$
21.13 **end**
21.14 **end**
21.15 **return** v_l

Algorithm 22: Implementation of the adaptive full approximation scheme.

22.1 **Algorithm** `FAScycle`$(A, v_l, f_l, f^*, r^*, R, \hat{R}, P, l, l_s, \nu_1, \nu_2)$
22.2 **if** $l = l_s$ **then**
22.3 $v_l =$ `solve`$(A_l v_l = f_l)$ **on** $\mathscr{C}_l$
22.4 **else**
22.5 $v_l =$ `smooth`$(A_l, v_l, f_l, \mathscr{C}_l, \nu_1)$
22.6 $r_l = f_l - A_l v_l$ **on** $\mathscr{C}_l \cup \mathscr{R}_l$
22.7 $u_{l-1} = \hat{R}^{l-1}_l$ **on** $\mathscr{C}_{l-1} \cap (\mathscr{C}_l \cup \mathscr{R}_l \cup \Gamma_l)$
22.8 $f_{l-1} = \begin{cases} R^{l-1}_l r_l & \textbf{on } \mathscr{C}_{l-1} \cap \mathscr{C}_l \\ f^*_{l-1} + R^{l-1}_l \left(r_l - r^*_l \right) & \textbf{on } \mathscr{C}_{l-1} \cap \mathscr{R}_l \end{cases}$
22.9 $f_{l-1} = f_{l-1} + A_{l-1} v_{l-1}$ **on** $\mathscr{C}_{l-1} \cap (\mathscr{C}_l \cup \mathscr{R}_l)$
22.10 $v_{l-1} = u_{l-1}$ **on** $\Omega_{l-1} \cup \Gamma_{l-1}$
22.11 $v_{l-1} =$ `FAScycle`$(A, v_{l-1}, f_{l-1}, f^*, r^*, R, \hat{R}, P, l-1, l_s, \nu_1, \nu_2)$
22.12 $f^*_{l-1} = f_{l-1} - A_{l-1} v_{l-1}$ **on** $\mathscr{C}_{l-1} \cap \mathscr{R}_l$
22.13 $v_{l-1} = v_{l-1} - u_{l-1}$ **on** $\mathscr{C}_{l-1} \cap (\mathscr{C}_l \cup \mathscr{R}_l)$
22.14 $v_l = \begin{cases} v_l = P^l_{l-1} v_{l-1} & \textbf{on } \mathscr{C}_l \cup \mathscr{R}_l \\ P^l_{l-1} (v_{l-1} - u_{l-1}) & \textbf{on } \mathscr{R}^E_l \end{cases}$
22.15 $u_{l-1} = v_{l-1}$ **on** Ω_{l-1}
22.16 $r^*_l = f_l - A_l v_l$ **on** $\mathscr{R}^I_l \cup \mathscr{R}_l$
22.17 $v_l =$ `smooth`$(A_l, v_l, f_l, \mathscr{C}_l, \nu_2)$
22.18 **end**
22.19 **return** v_l

and the initialization

$$r_l = 0 \qquad \textbf{on } \mathscr{R}_l^E$$

in line 13.7 are also actions on hyperplanes, which have not been implemented, so far.

Even more differences can be found in the full approximation scheme (Algorithm 12 vs. Algorithm 8). The first one is spotted in the pre-smoothing step, where v_l is not smoothed, but only copied to $\overline{v}_l$ on $\mathscr{R}_l$ and $\mathscr{R}_l^E$ (line 12.5). When Gauss-Seidel is used as a smoother, this operation can be omitted, because the smoother works in-place, i. e., v_l and $\overline{v}_l$ are identical. For the Jacobi smoother, v_l and $\overline{v}_l$ are stored in different memory arrays. Thus, the values have to be copied. This is not tough to implement, but it is mentioned here for completeness.

The calculation $\overline{r}_l = 0$ on $\mathscr{R}_l^E$ (line 12.6) already occurred in Algorithm 13. A new change is the calculation of the defect

$$d_{l-1} = r_{l-1} + R_l^{l-1}\left(\overline{r}_l - r_l\right) \qquad \textbf{on } \mathscr{C}_l \cap \mathscr{R}_l$$

in line 12.7, which will need a special implementation.

In the initialization of v_{l-1}^* outside the refined area,

$$v_{l-1}^* = u_{l-1} \qquad \textbf{on } \mathscr{C}_{l-1} \cap \overline{\mathscr{C}}_l$$

in line 12.8, is again a simple copy operation.

The right-hand side calculation

$$f_{l-1} = R_l^{l-1} f_l \qquad \textbf{on } \mathscr{C}_{l-1} \cap \left(\overline{\mathscr{C}}_l \setminus \mathscr{R}_l\right)$$

in line 12.9 was already a topic of discussion in Section 4.3.1. The conclusion there was that this calculation can usually avoided by initializing f_{l-1} with the model's right-hand side. In HHG it is implemented this way; therefore, there is no need to implement additional functionality, at this point.

The calculation

$$\tilde{e}_l = P_{l-1}^l\left(v_{l-1} - v_{l-1}^*\right) \qquad \textbf{on } \mathscr{R}_l^E$$

in line 12.13 cannot be avoided and the equation has not been encountered, yet. Thus, it has to be implemented.

The addition $\tilde{v}_l = \overline{v}_l + \tilde{e}_l$ does not have to be considered, because $\tilde{e}_l$ is not explicitly computed in Algorithm 22. The subsequent residual computation on the interior refinement boundary has already occurred in Algorithm 13. For post-smoothing the same statements as for pre-smoothing apply.

Memory demand

As stated above, the adaptive full multigrid algorithm needs one variable more than the algorithm for uniform meshes. However, the variables do not have to be allocated on the entire mesh. The additional variable on the finest level, r^*, is needed on $\mathscr{R}_l^I \cup \mathscr{R}_l \cup \mathscr{R}_l^E$. The additional variable on the coarser levels, f^*, is needed on $\mathscr{C}_{l-1} \cap \mathscr{R}_l$. Thus, both variables are only needed on hyperplanes of the mesh.

For estimating the memory demand of the additional variables, we assume that the memory demand of a variable depends linearly on the number of mesh points

on which it is defined. For estimating the number of mesh points, we can assume a regular grid in D dimensions, because the HHG meshes on the higher levels are regular to a large extent. The number of points in a regular grid Ω_l on level l is

$$N(\Omega_l) \approx 2^{Dl}.$$

The number of points in a hyperplane Ω_l^*, which is a mesh of dimension $D-1$, is

$$N(\Omega_l^*) \approx 2^{(D-1)l}.$$

Thus, a variable that is only defined on a hyperplane needs 2^l times less memory than a variable that is defined on the complete mesh:

$$\frac{N(\Omega_l^*)}{N(\Omega_l)} \approx 2^{-l}.$$

Now the additional memory demand on the finest level can be calculated. Since r^* is defined on three hyperplanes, its memory demand is $3 \cdot 2^{-l}$ the demand of a fully allocated variable. Since there are three other variables defined on the mesh, a fourth fully allocated variable would increase the total memory demand by 1/3. Thus, the need for r_l^* increases the total memory demand by a factor of $1/3 \cdot 3 \cdot 2^{-l} = 2^{-l}$. The additional memory demand for f_{l-1}^* can be calculated in a similar way. However, since the number of mesh points scales with a factor of 2^D between the levels, it is small compared to the demand for r_l^* and will, for simplicity, be neglected in the following.

The question that is relevant in practice is: when does adaptive mesh refinement pay off, in terms of memory demand? Adaptive refinement uses less memory than uniform refinement as soon as the savings due to a reduced set $\mathscr{C}_l$ exceed the expenses caused by the variable r^*. The variables v, f, and r are defined on $\mathscr{C}_l \cup \mathscr{R}_l \cup \mathscr{R}_l^E$. Thus, the memory demand for these variables is

$$\begin{aligned} N(v,f,r) &= 3N(\mathscr{C}_l) + 3\left(N(\mathscr{R}_l) + N(\mathscr{R}_l^E)\right) \\ &\approx 3N(\mathscr{C}_l) + 6N(\mathscr{R}_l), \end{aligned}$$

assuming that $N(\mathscr{R}_l) \approx N(\mathscr{R}_l^E)$. Including r^*, the total memory demand for adaptive refinement is

$$\begin{aligned} N(v,f,r)_{\text{adaptive}} &= 3N(\mathscr{C}_l) + 9N(\mathscr{R}_l) \\ &\approx 3\left(2^{Dl} + 3 \cdot 2^{(D-1)l}\right). \end{aligned}$$

For uniform refinement, $\mathscr{R}_l$ and $\mathscr{R}_l^E$ are empty and $\mathscr{C}_l = \Omega_l$. Thus,

$$\begin{aligned} N(v,f,r)_{\text{uniform}} &= 3N(\Omega_l) \\ &\approx 3 \cdot 2^{Dl}. \end{aligned}$$

Thus, $N(v,f,r)_{\text{adaptive}} < N(v,f,r)_{\text{uniform}}$, if

$$\begin{aligned} & \rho 2^{Dl} + 3 \cdot 2^{(D-1)l} < 2^{Dl} \\ \Leftrightarrow\ & \rho < 1 - 3 \cdot 2^{-l}, \end{aligned}$$

where ρ is the fraction of Ω_l that is covered by $\mathscr{C}_l$.

Note that ρ depends only on l, but not on D. For typical values of l, the limits are:

- for $l = 1$: $\rho < -0.50$,
- for $l = 2$: $\rho < 0.25$,
- for $l = 5$: $\rho < 0.90$,
- for $l = 9$: $\rho < 0.99$.

This means, for only one level of refinement, adaptive refinement never pays off. However, for two or more levels of refinement, it quickly starts paying off; e. g., for five levels it pays off even if $\mathscr{C}_l$ covers almost 90% of the domain.

4.6 Numerical results

Now that the theory is complete, some examples that show the quality of the approximation produced by the adaptive refinement algorithm shall be presented. Polynomials of different degrees and trigonometric functions are used as boundary conditions. Since the underlying finite element method is not exact for polynomials of higher degrees and non-polynomial functions, it would not be useful to compare the approximation with the exact solution. Therefore, in order to determine its quality, the approximation is put into relation with the approximation obtained by uniform refinement.

4.6.1 Model problems and geometries

The unit cube is used as computational domain for all numerical experiments. Since HHG currently knows only tetrahedral elements, the cube is split into tetrahedra. At least six tetrahedra are needed to build up a cube. The splitting of a cube into six tetrahedra was already shown in Chapter 3 (see Fig. 3.3). Since the adaptive refinement algorithm does not allow for varying the refinement level within coarse-grid elements, but only between them, a mesh with only six coarse-grid elements is not very flexible in terms of adaptive refinement. Therefore, the unit cube is split into 8 sub-cubes, which are then split into six tetrahedra, each. The resulting mesh with 48 elements is sufficient for testing adaptive refinement. At the same time, it is regular enough for the refinement to be defined manually. This is important, because an automatic, dynamic mesh refiner is not yet available for HHG.

The boundary conditions in all experiments are chosen such that the dominating term of the solution increases along the diagonal from $x = (0,0,0)^T$ to $x = (1,1,1)^T$, which is shown in Fig. 4.11. If that term dominates strongly, the solution "concentrates" in the corner at $x = (1,1,1)^T$. Together with proper tuning of the solution's curvature, this setup induces the demand for adaptively refining the sub-cube at $x = (1,1,1)^T$.

Another advantage of this setup is that the numerical values (solution, error, etc.) along the diagonal of the unit cube are representative for the values in the entire domain. Therefore, the diagrams for showing the numerical results in the following sections plot the values along the diagonal of the unit cube. For these plots, we define the auxiliary one-dimensional coordinate system (x'). The coordinate axis is aligned with the diagonal of the unit cube. The system's offset and scaling relative to the original system are chosen such that

- $x = (0,0,0)^T$ is mapped to $x' = 0$, and

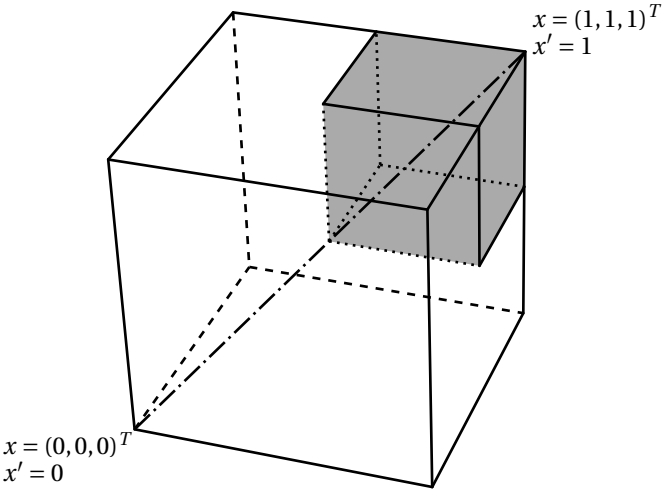

Figure 4.11: Computational domain used for numerical experiments.

- $x = (1,1,1)^T$ is mapped to $x' = 1$.

The auxiliary coordinate system is also indicated in Fig. 4.11.

In the following, we will analyze three model problems.

Problem 1. A linear polynomial:

$$\begin{aligned} u &= \frac{1}{3}(x+y+z), \\ f &= 0. \end{aligned}$$

The solution increases linearly with x'. It is scaled such that $u(x') = 1$ for $x' = 1$. Since u is a polynomial of first order, a correctly implemented a linear finite element method must solve this problem exactly, also on an adaptively refined mesh.

Problem 2. A quadratic polynomial:

$$\begin{aligned} u &= \frac{1}{3}(x+y+z)^2 - \frac{1}{2}(x-y)^2, \\ f &= 0. \end{aligned}$$

The function u is obtained by rotating and scaling the function $x^2 - y^2$ such that the x-axis is mapped onto the diagonal of the unit cube and the point $(1,0,0)^T$ is mapped onto $(1,1,1)^T$.

The solution increases quadratically with x'. Note, though, that $(x+y+z)^2$ is not the dominant term; the function is also quadratic along the $(x-y)$ axis. The linear finite element method does not necessarily solve this problem exactly, any more. Therefore, it is a good tool for studying subtle differences between uniform and adaptive refinement.

Problem 3. A transcendental function:

$$\begin{aligned} u &= s \cdot \sinh\left(\frac{2}{\sqrt{6}} k\pi(x+y+z)\right) \cos\left(\frac{1}{\sqrt{2}} k\pi(y-x)\right) \cos\left(\frac{1}{\sqrt{6}} k\pi(2z-x-y)\right), \\ &\quad \text{where } s = \left(\sinh(\sqrt{3})\sqrt{2}k\pi\right)^{-1}, \\ f &= 0. \end{aligned}$$

The function u is obtained by transforming the function $\sinh(\pi x)\cos(\pi y)\cos(\pi z)$ in the way described in Problem 2.

For this function, $u(x') = \sinh(x')$; in the orthogonal directions $(x-y)$ and $(z-x-y)$ the sinh is "distorted" by cos waves. In contrast to the previous functions, the gradient of this function is strong and directed primarily along the diagonal of the unit cube, making the function a natural candidate for adaptive refinement. The parameter k modifies the "steepness" of u along x'; a higher value of k causes the solution to concentrate more at $x' = 1$. The scaling parameter s ensures that $u = 1$ at $x' = 1$ for any choice of k.

4.6.2 Expected results

In general, the solution of a BVP is not computed exactly by the linear finite element method, even on regular grids. Examples are Problems 2 and 3 above. Unfortunately, it is, in general, not possible to precisely state the expected error of the computed approximation. Generally only the order of the error can be given—for HHG's finite element discretization, the error is in $\mathcal{O}(2^{-2l})$. Exact error bounds can only be derived for regular grids. Therefore, we will formulate properties that we can expect the numerical results to fulfill, even though strict error bounds are not available for the HHG meshes.

The linear polynomial is solved exactly

Both the quadrature rules (cf. Section 2.2.2) and the basis functions (cf. Section 2.2.6) used in HHG approximate linear polynomials exactly. The quadrature rules approximate even polynomials up to second degree exactly, by the way. Therefore, HHG must solve Problem 1 exactly. The adaptive refinement method does not introduce any additional errors in this case, so the problem must also be solved exactly on an adaptively refined mesh.

The exact solution is invariant

If a multigrid cycle is initialized with the exact solution (e. g., $v_l = u_l$ in Algorithm 6), this initial approximation must not be changed. This must also hold for adaptively refined meshes. Note that the full multigrid algorithm does generally not have this property, because the interpolation of the solution to the next higher level is not exact.

Adaptive refinement is not worse than uniform refinement on a lower level

Adaptively refining a mesh in HHG means starting with a mesh that is refined uniformly or adaptively to level l and refining some elements from level l to level $l+1$. This means that all nodes of the original mesh are also contained in the refined

mesh. In the numerical experiments, we will examine the particular case that all elements in the adaptively refined mesh are refined to either level l or $l+1$. When solving the problem on the adaptively refined mesh, we expect that—although the computations at the refinement boundaries may be affected by inaccuracies, see Section 4.3—the error on all nodes is equal to or lower than the error obtained by solving on a mesh refined uniformly to level l.

4.6.3 Observed results

The analysis of the actual HHG results starts with the polynomial problem and works its way up over the slightly harder quadratic problem to the hardest one, the non-polynomial problem. The following solver configuration was used in all computations:

- Multigrid algorithm: full multigrid with FAS-cycles.
- Smoother: red-black Gauss-Seidel.
- Number of pre-/post-smoothing steps: $\nu_1 = \nu_2 = 2$.
- Restriction operator: full weighting.
- Prolongation operator: linear interpolation.
- Coarse-grid solver: red-black Gauss-Seidel.
- Start level for full multigrid and coarsest level for FAS-cycles: $l_s = 2$.
- Initial approximation on start level: $\nu_2 = 0$.
- Mesh hierarchy for adaptive refinement: uniform refinement up to $l_f - 1$.
- Number of FAS-cycles on each level: $\mu = 20$.

The high number of FAS-cycles per level ensures that the problems are always solved to discretization accuracy. Thus, the errors that are shown in the following graphs are the discretization errors of the numerical schemes.

Fig. 4.12 shows the solution and the errors along x' for solving Problem 1 on $l_f = 4$. The error curves are plotted using a logarithmic y axis. Therefore, the Dirichlet boundary points at $x' = 0$ and $x' = 1$ (where the error is 0) are excluded. With uniform refinement, the number of interior mesh points along x' on level l_f is $2^{l_f+1} - 1$, i. e., 31 for $l_f = 4$. For $x' \in [0, 0.5]$ the error graph for adaptive refinement is defined on every other point only, because there the mesh is refined to one level less—see Fig. 4.11.

Both uniform and adaptive refinement solve the problem exactly; for both algorithms the error maxima are in the order of 10^{-14}. Slight differences in the error curves can be observed, but they are not surprising. The two algorithms perform quite different computations, which are both numerically correct, but, due to round-off errors, have slightly different results when implemented with floating-point numbers. These round-off errors show up most prominently, if the numerical errors are small, as in the example at hand.

Fig. 4.13 shows solution and errors for Problem 2, again for $l_f = 4$. As stated in the presentation of the problems, we cannot assume that HHG solves this problem

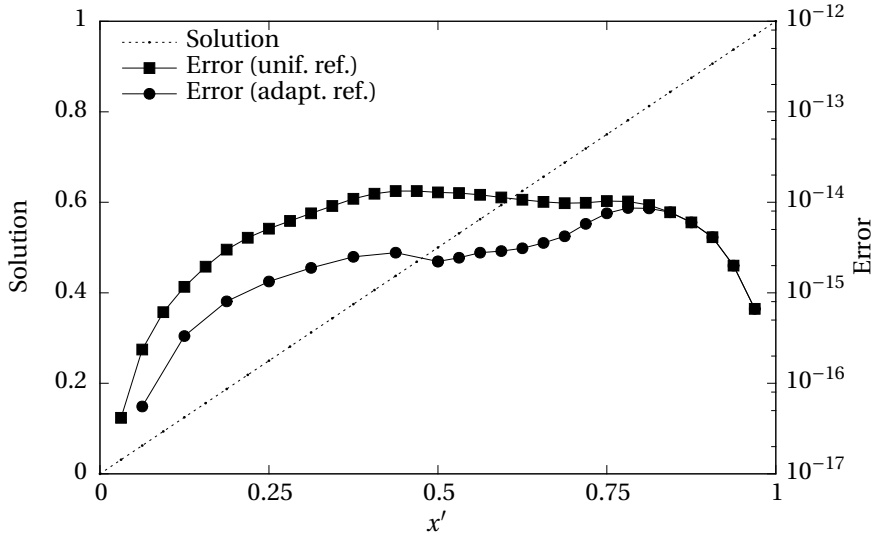

Figure 4.12: Solution and errors for Problem 1 on level 4.

exactly, because it implements a linear finite element method. However, on completely regular grids even a linear method can reach quadratic accuracy. This is what happens in this example, as the error curve for uniform refinement shows. The approximation on the uniformly refined mesh is still exact.

The mesh's regularity can easily be destroyed by moving one of the vertices of the coarse mesh. For comparison, a "skewed" mesh was created by moving the inner vertex (the one that connects all eight cubes) from $(0.5, 0.5, 0.5)^T$ to $(0.4, 0.4, 0.4)^T$. The resulting mesh is topologically equal to the original mesh, but the refined meshes are not completely regular, any more. The skew does not influence the finite elements' anisotropy significantly, by the way. The error obtained when solving Problem 2 on the skewed mesh with uniform refinement is also shown in Fig. 4.13; it is in the order of 10^{-4}.

The regularity is also lost, if the regular mesh is refined adaptively, because then the couplings at the refinement boundary are not symmetric, any more. That even applies for adaptive smoothing (Algorithm 9), although that algorithm works on the uniformly refined mesh! Not *using* some of the degrees of freedom on the finest mesh is the same as not having them available in the first place. Fig. 4.13 shows the errors for adaptive refinement and adaptive smoothing. They are in the range of the error for the skewed mesh. The question why adaptive refinement and adaptive smoothing are not exactly equal was not analyzed in detail.

A logarithmic plot (Fig. 4.14) of the error at $x' = 0.5$ for varying l_f shows that for adaptive refinement, adaptive smoothing, and uniform refinement of the skewed mesh the error decreases exponentially with l_f. This is the expected behavior of a linear finite element method with a discretization accuracy of $\mathcal{O}(h^2)$. The error for uniform refinement of the regular mesh actually increases with l_f, because, as for Problem 2, it is dominated by round-off errors, which depend on the number of unknowns on the finest level.

This observed result contradicts the expected result that adaptive refinement is not worse than uniform refinement on a lower level. In this example, the regularity of

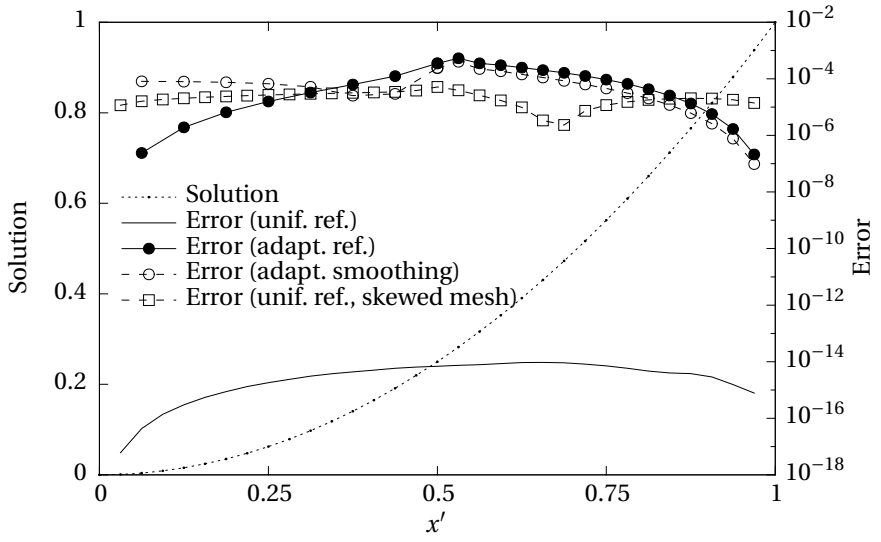

Figure 4.13: Solution and errors for Problem 2 on level 4.

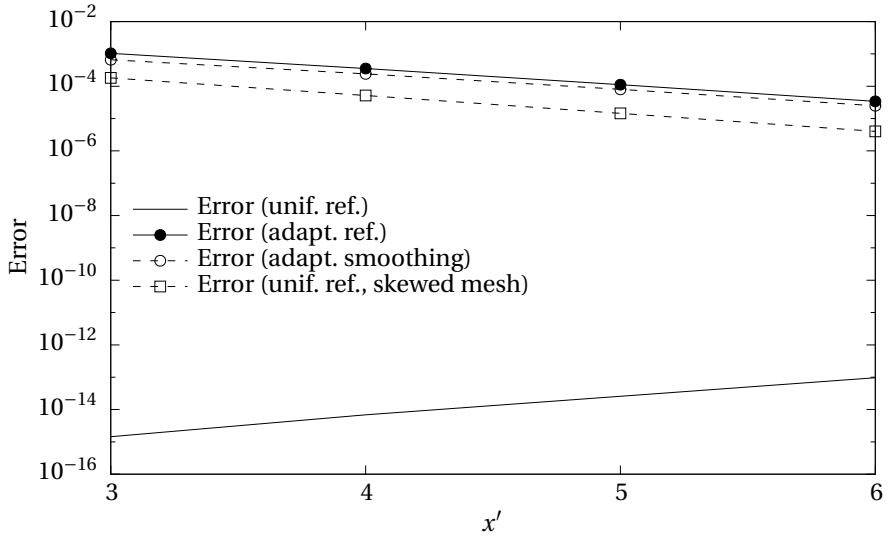

Figure 4.14: Errors on different levels for for Problem 2.

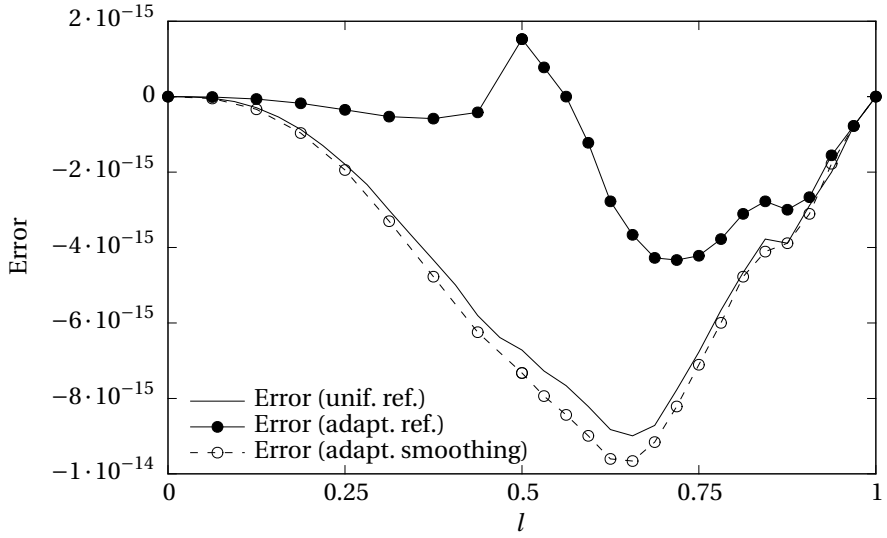

Figure 4.15: Errors for Problem 2 on level 4 when initializing the solver with the exact solution.

the mesh, which can only be achieved with uniform refinement, boosts the accuracy of the finite element method beyond the usual $\mathcal{O}(h^2)$ accuracy, and the problem is always solved exactly, even on a very coarse mesh. Although the example is rather academic, it shows that adaptive refinement should generally be used with caution.

A test for verifying that the error is not introduced through the multigrid algorithm is to initialize the FAS cycle on the finest level with the exact solution. (Then, of course, it is not necessary—nor useful—to use full multigrid.) The full approximation scheme, like any stationary iteration, is invariant to the exact solution, and this must also hold for FAS with adaptive refinement. Thus, when conducting this experiment, the error must remain zero after any number of FAS cycles. Fig. 4.15 shows that this is in fact the case. 20 FAS cycles on a mesh refined up to level 4 were performed, where the first cycle was initialized with the exact solution. For adaptive refinement, the elements that were only refined to level 3 were initialized with the exact solution on that level. As the figure shows, the error is in the order of 10^{-14} in all cases, which is the round-off error that we already observed in some of the previous examples.

The non-polynomial Problem 3 can not be solved exactly with a linear finite element method, any more, even on a regular grid. And, as we will see, the steepness k of the solution has a high influence on how well it is approximated. Fig. 4.16 depicts the solution for two different values of k: $k = 1$ and $k = 2$.

For $k = 1$, Fig. 4.17 shows the errors achieved with uniform and adaptive refinement on level 4 ($l_\mathrm{f} = 4$). For comparison, it also shows the error achieved with uniform refinement on level 3.

With uniform refinement the error is always positive. With adaptive refinement (both adaptive smoothing and real adaptive refinement) the error drops strongly into the negative at the refinement boundary. The peak is located at the layer of nodes inside the refined area that is directly adjacent to the refinement boundary (at $\mathscr{R}_l^I$, as defined in Section 4.3). As Fig. 4.18 shows, the artifact does not occur because

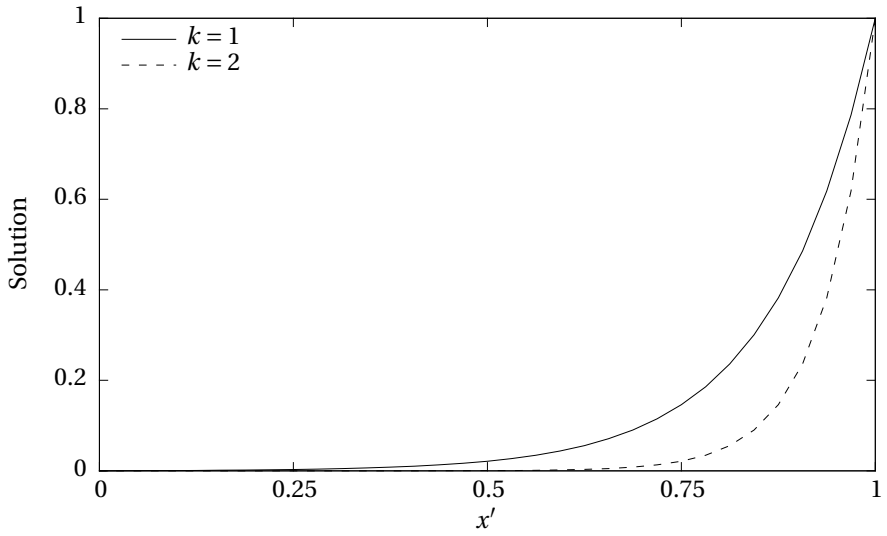

Figure 4.16: Solutions for different values of k in Problem 3.

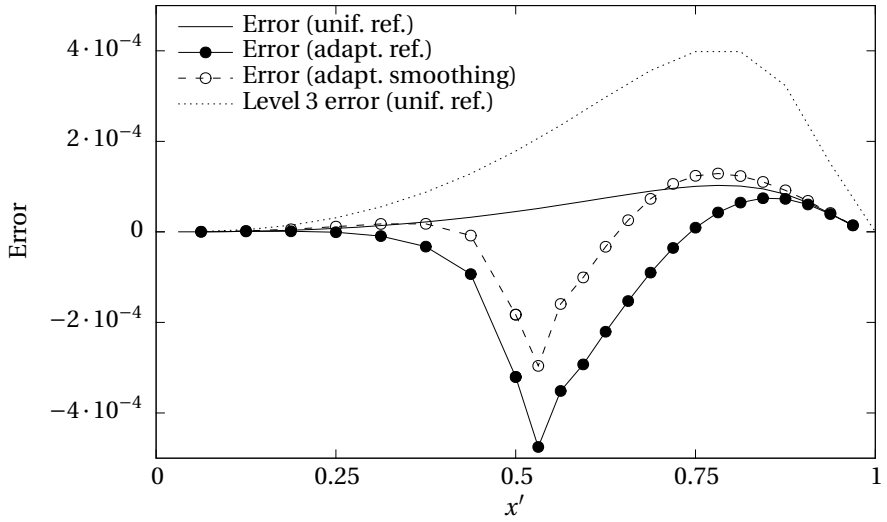

Figure 4.17: Errors for Problem 3 with $k = 1$ on level 4.

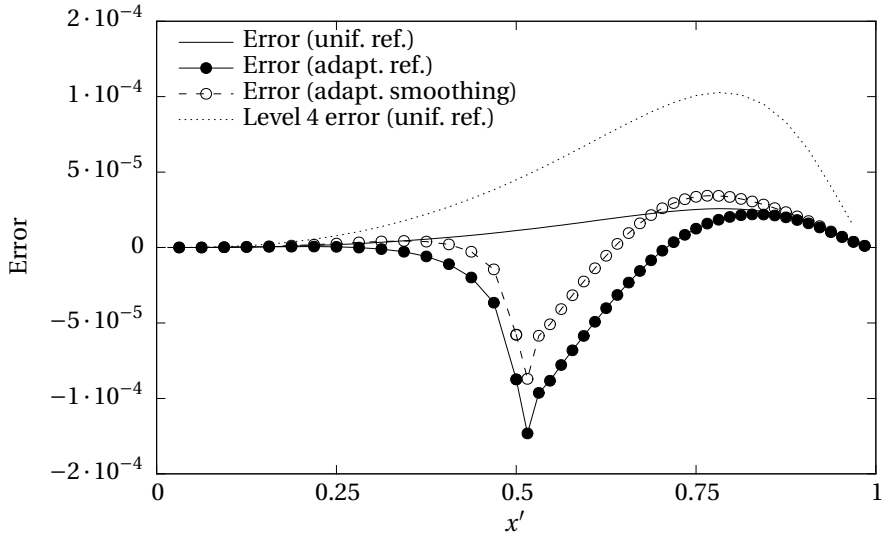

Figure 4.18: Errors for Problem 3 with $k = 1$ on level 5.

the mesh resolution is too small. At one refinement level higher ($l_\mathrm{f} = 5$), the artifact in the adaptive refinement error is present at a smaller scale, but in the same relation as for $l_f = 4$. In both cases the maximum of the absolute error is slightly higher than the maximum of the uniform refinement error on the coarser level.

The root cause for this artifact is that the curvature of the solution is too high at the refinement boundary. In consequence, when prolongating the solution from $l_\mathrm{f} - 1$ to l_f (line 13.6 in Algorithm 13), the initial error is quite high, because in HHG only linear interpolation is used for prolongating the solution (i. e., $\hat{P} = P$). With uniform refinement, that is not a big problem, because the FAS cycles on l_f apply smoothing and coarse grid correction to all nodes on l_f. With adaptive refinement, however, the FAS cycles do not apply smoothing to nodes in $\mathscr{R}_l$ and $\mathscr{R}_l^E$ (only coarse grid correction).

To alleviate the problem, two options are available. First, a higher-order interpolation $\hat{P}$ can be used. However, as explained in Section 4.3.3, that would require a significant implementation effort in HHG. While it is an interesting topic for future research, it is beyond the scope of this thesis. The other option is to extend the refined area of the mesh, so that the refinement boundary is located in an area of lower solution curvature. For our experiments, we can analogously change the characteristics of the problem, so that the parts of the solution with high curvature are all inside the refined are a of the given mesh. This can be achieved by changing the problem's steepness parameter k. As shown in Fig. 4.16, for $k = 2$ the solution concentrates more at $x' = 1$, and the curvature in the unrefined area ($x' \leq 0.5$) decreases. The errors for this configuration are shown in Fig. 4.19. The y-axis of the graph is scaled such that the errors on level 4 can be observed well, at the expense of cutting off the level 3 error curve, which has its maximum of $1.5 \cdot 10^{-3}$ at $x' = 0.875$. The error for adaptive smoothing is not shown, because it is so close to the error for real adaptive refinement that it would clobber the plot. The artifact of the adaptive refinement error dropping into the negative at the refinement boundary, which we know from Figures 4.17 and 4.18, is still observable, but it is much less pronounced.

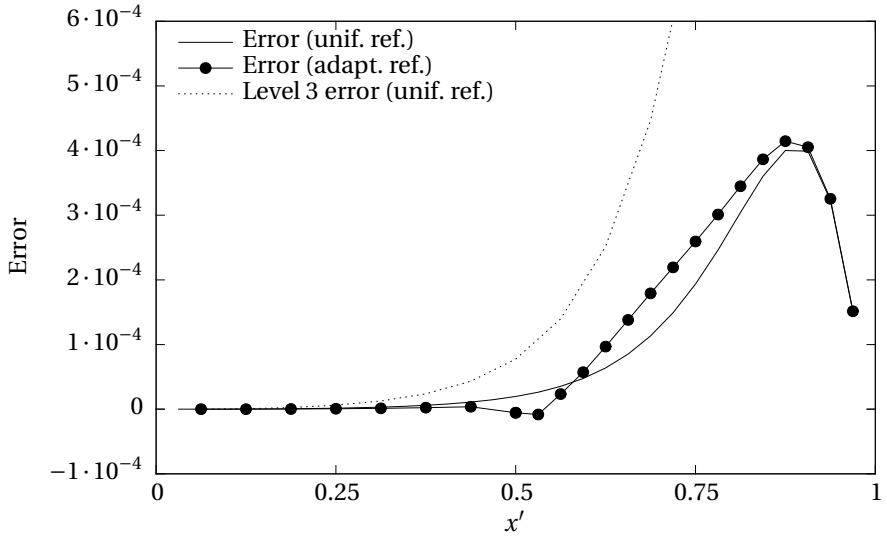

Figure 4.19: Errors for Problem 3 with $k = 2$ on level 4.

The error is now close to the error of uniform refinement. In particular, it is everywhere smaller than the uniform refinement error on the coarser level, which is the expected result claimed in Section 4.6.2.

Chapter 5

Optimization of multigrid cycles

Contents

This chapter is based on the article "Optimizing the number of multigrid cycles in the full multigrid algorithm" by Alexander Thekale[1] and the author of this work, which was published in the proceedings of the 14th Copper Mountain conference on multigrid methods, 2009 [47]. Its quite different approach to improve the efficiency of multigrid algorithms provides an interesting contrast to the previous chapter. While adaptive mesh refinement aims at distributing the computational work in an optimal way by exploiting the *properties of the PDE*. The method presented in this chapter looks at the *structure of the multigrid algorithm*. Since the method does not consider *what* is solved, but *how* it is solved, it is even applicable to multilevel methods that are not concerned with the solution of PDEs.

The method minimizes the execution time of full multigrid by optimizing the number of multigrid cycles performed on each level. In Algorithm 7, the parameter μ denoted the number of $\gamma-cycle$s performed on each level. The introduction did, however, not specify how to choose μ. It is common practice to set μ to the same value on all levels, although no one prevents us from using a different value on each level. The number of cycles per level that is necessary to reach the desired accuracy of the solution on the finest level is usually determined by trial and error. We will show how a *branch and bound* (B&B) algorithm can be used to find the optimal number of cycles per level. Characteristics of the PDE, its discretization, and the solver are used as parameters in the optimization.

A model that describes the error reduction and the cost of full multigrid is constructed in Section 5.1. This model is required for a B&B optimization algorithm,

[1] at that time doctorate student at the Department of Mathematics, Applied Mathematics II, University of Erlangen-Nuremberg, Germany

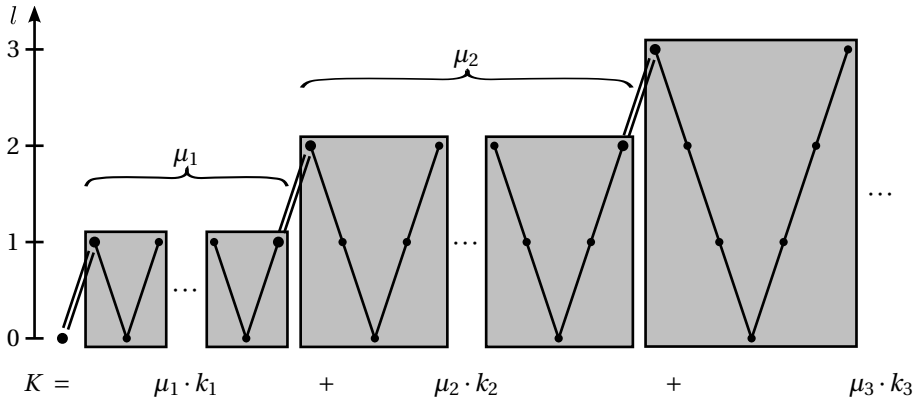

Figure 5.1: Illustration of the full multigrid method.

which is derived in Section 5.2. In Section 5.3 some extensions that make the initial model more applicable in practice are added. The software library *cycleopt*, which implements this optimization algorithm, is introduced in Section 5.4. The chapter is concluded with examples in Section 5.5 and an outlook to further and related work in Section 5.6.

5.1 An error and cost model for the full multigrid algorithm

The goal of this section is to set up a model for the cost and the accuracy of the full multigrid algorithm. In order to remain easily comprehensible, the basic model presented in this section excludes details necessary for representing real-life multigrid algorithms. A more complete model that can be used with HHG is described in Section 5.3.

Definitions and assumptions

In order to set up the model, we need to make some assumptions about the solver, as well as about the PDE to be solved, and its discretization. Fig. 5.1 shows a schematic representation of the full multigrid algorithm. In the first step, the linear system of equations is solved on level $l = 0$. The solution is propagated to level 1, where μ_1 multigrid cycles are performed. The result (not the exact solution, in general, but only an approximation) is propagated to level 2, where μ_2 multigrid cycles are performed, and so on. The algorithm terminates upon reaching a predefined finest level. The cost of a full multigrid run, denoted by K, depends on the number of cycles μ_l performed on each level l and on the cost k_l of a multigrid cycle on that level.

The discretization technique that is used for constructing the mesh hierarchy influences both the cost of the multigrid cycles and the accuracy they can achieve. The error and cost model assumes that the number of mesh points in each space dimension doubles when going from one mesh to the next finer one. A doubling of the mesh points means that the distance h between the points is halved:

$$h_l = \frac{1}{2} h_{l-1}. \tag{5.1}$$

Using regular meshes, it is easy to construct such a mesh hierarchy. Recall from Section 2.4 that the Bey tetrahedral refinement used in HHG satisfies this property. For other mesh generation techniques, like adaptive refinement, or when coarsening an initial fine mesh, this property may not hold. In this case, the cost model would have to be adapted.

In general, the analytic solution of a PDE cannot be reproduced exactly by solving the corresponding discrete system of equations. The difference between analytic and discrete solution, the *discretization error*, will be denoted with e^*. Its magnitude depends on the discretization technique, the smoothness of the analytic solution, and the mesh resolution. In the basic model we assume that e^* follows perfectly an $\mathcal{O}(h^2)$ behavior, i. e., with (5.1),

$$e_l^* = \frac{1}{4} e_{l-1}^* \text{ for all } l > 0 .$$

Imposing, without loss of generality, a normalization of the initial discretization error on level 0 as $e_0^* = 1$, the discretization error on level l is given by

$$e_l^* = 2^{-2l} = 4^{-l} . \tag{5.2}$$

While full multigrid can be considered a direct solver, its building blocks, the multigrid cycles, are iterative solvers. Starting from an initial guess, iterative solvers approach the discrete solution not in one step, but in several iterations. The difference between the discrete solution and the approximate solution after an iteration is called *algebraic error*. The *total error*, denoted by e, is the sum of discretization and algebraic error.

The cost of a multigrid cycle on level l is the sum of the costs of its basic building blocks on level l (smoothing, residual calculation, restriction, and prolongation) plus the cost of a multigrid cycle on level $l-1$. The costs of the building blocks are assumed to be proportional to the number of mesh points on that level. To describe the cost of a multigrid cycle, we use abstract *work units* that are proportional to the number of mesh point updates during that cycle. The cycleopt library can also use the actual execution times for the optimization.

With (5.1), the number of mesh points on level l of a mesh in d spatial dimensions is given by 2^{dl}. With these assumptions, the cost k_l of a multigrid cycle on level l is given by $k_l = 2^{dl} + k_{l-1}$. Setting, without loss of generality, the cost for solving the linear system on the coarsest mesh to 1 ($k_0 = 1$), we can eliminate the recursion, and

$$k_l = \sum_{i=0}^{l} 2^{di} . \tag{5.3}$$

The factor by which the algebraic error (*not* the total error) is reduced by an iterative solver in each iteration is called *convergence rate* and denoted by ρ. Section 5.3 will show that the convergence rate is usually different for each level and for each multigrid iteration at that level. For our basic error and cost model we assume that the convergence rate is constant.

The basic model will assume that the full multigrid starts at level 0 with solving the linear system. In general, full multigrid can start with solving at an arbitrary level $l_s \geq 0$. This will be covered by the extended model. The finest level $l_f > l_s$ depends on the desired accuracy of the solution. The desired accuracy is specified by the user as an *error target*, denoted by $e^\dagger$, that must not be exceeded upon termination of the

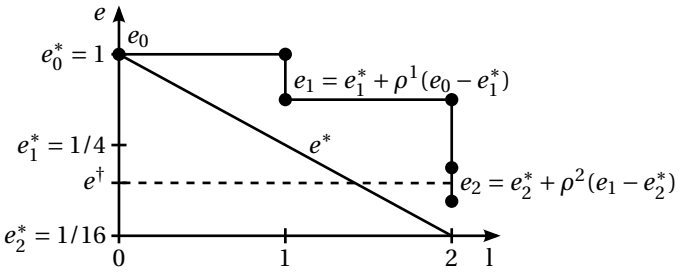

Figure 5.2: Error reduction, according to the basic error model, for a full multigrid run with $\mu_1 = 1$ and $\mu_2 = 2$.

full multigrid run. If the discretization error on the finest mesh was larger than the error target, the accuracy goal could not be reached. Therefore, l_f can be any integer that is large enough to ensure

$$e^*_{l_\mathrm{f}} < e^\dagger . \tag{5.4}$$

The case $e^*_{l_\mathrm{f}} = e^\dagger$ is not allowed, because, except for special cases, ρ is larger than 0. This means that the algebraic error will never vanish completely, and the total error will always be larger than the discretization error.

The basic model

Fig. 5.2 depicts the total error e, as it is reduced in the course of a full multigrid run. Assuming a constant convergence rate, every multigrid cycle on level l reduces the algebraic error $e_l - e^*_l$ by a factor of ρ. The initial error e_l (after prolongating from level $l-1$, but before performing any multigrid cycles) is assumed to be equal to e_{l-1}. Therefore, after μ_l iterations on level l, the total error is

$$e_l = e^*_l + \rho^{\mu_l} \left(e_{l-1} - e^*_l \right) . \tag{5.5}$$

Recursively using this equation for e_{l-1} yields

$$\begin{aligned} e_l &= e^*_l + \rho^{\mu_l} \left(e^*_{l-1} + \rho^{\mu_{l-1}} \left(e_{l-2} - e^*_{l-1} \right) - e^*_l \right) \\ &= e^*_l + \rho^{\mu_l} \left(e^*_{l-1} - e^*_l \right) + \rho^{\mu_l + \mu_{l-1}} \left(e_{l-2} - e^*_{l-1} \right) . \end{aligned} \tag{5.6}$$

The total error on level 0 is equal to the discretization error ($e_0 = e^*_0$), because the linear system is solved exactly on this level. Using this terminal and (5.2), the recursive equation (5.6) can be written as a sum to describe the total error on the finest level:

$$\begin{aligned} e_{l_\mathrm{f}} &= e^*_{l_\mathrm{f}} + \sum_{l=1}^{l_\mathrm{f}} \left(e^*_{l-1} - e^*_l \right) \rho^{\mu_{l_\mathrm{f}} + \ldots + \mu_l} \\ &= 4^{-l_\mathrm{f}} + \sum_{l=1}^{l_\mathrm{f}} 3 \cdot 4^{-l} \rho^{\mu_{l_\mathrm{f}} + \ldots + \mu_l} . \end{aligned} \tag{5.7}$$

The total cost K of the full multigrid algorithm is the sum of the costs of all multigrid cycles on all levels.

$$K = \sum_{l=0}^{l_\mathrm{f}} \mu_l k_l \quad \text{with } \mu_0 = 1 \text{ and } k_0 = const . \tag{5.8}$$

5.2 Branch and bound optimization

In order to find and apply a suitable optimization algorithm, we cast our cost optimization problem into the form of a standard minimization problem. The aim of the optimization is to minimize the total cost K of a full multigrid run. The variables that can be modified in order to find a minimal solution are the numbers of multigrid cycles, μ_l. The error target $e^\dagger$ acts as a constraint to the minimization, because it determines the feasible values for the μ_l.

The optimization problem

We can formally write this optimization problem as

$$\begin{array}{lll} \min\limits_{(\mu_1,\ldots,\mu_{l_\mathrm{f}})^T} & \sum\limits_{l=1}^{l_\mathrm{f}} \mu_l k_l & \\ \text{such that} \quad e_{l_\mathrm{f}} & \leq & e^\dagger \\ \mu_1,\ldots,\mu_{l_\mathrm{f}} & \in & \mathbb{N}_0 = \{0,1,2,3,\ldots\}. \end{array} \tag{5.9}$$

Note that for $l = 0$ we have $\mu_0 = 1$ and, thus, the leading term of the sum in the objective function was omitted.

Replacing e_{l_f} by its expression in (5.7) shows that (5.9) is a nonlinear programming problem, because the μ_l appear in the exponent of ρ:

$$\begin{array}{lll} \min\limits_{(\mu_1,\ldots,\mu_{l_\mathrm{f}})^T} & \sum\limits_{l=1}^{l_\mathrm{f}} \mu_l k_l & \\ \text{such that} \quad \sum\limits_{l=1}^{l_\mathrm{f}} 3\cdot 4^{-l}\rho^{\mu_{l_\mathrm{f}}+\ldots+\mu_l} & \leq & e^\dagger - 4^{-l_\mathrm{f}} \\ \mu_1,\ldots,\mu_{l_\mathrm{f}} & \in & \mathbb{N}_0. \end{array}$$

Since the μ_l can only be integers, the problem falls into the category of *nonlinear integer programming* problems.

Branch and bound algorithms

Since the number of integer variables in problem (5.9) is rather small, an exact solution method based on a B&B procedure appears to be appropriate. In the following, we will briefly explain how the B&B algorithm works. For a detailed discussion we refer to [36].

As depicted in Fig. 5.3, the B&B algorithm spans a search tree, evaluating the cost of possibly every feasible assignment of values to the variables. The first level of the tree contains nodes for all feasible values of the first variable. Branches lead from each node on the first level to nodes on the second level, connecting each feasible value for the first variable with all feasible values for the second variable. The number of levels in the tree is equal to the number of variables. Each path through the tree represents exactly one configuration, i. e., an assignment of values to all variables.

If there are m_i feasible values for variable i, then there are $m_1 m_2$ nodes on the second level of the tree. If there are n variables, the complete tree has n levels, and the number of nodes on the last level is $\prod_{i=1}^{n} m_i$. However, if the set of feasible values is infinite for one or more variables the search tree is infinitely large, and the B&B algorithm does not terminate. Even if the tree has a finite size, it has usually too

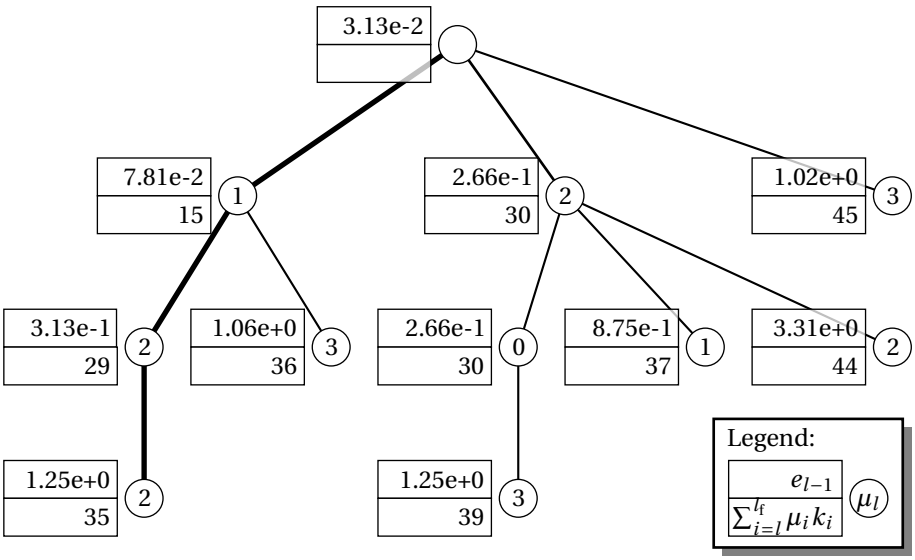

Figure 5.3: Example of a branch and bound tree.

many nodes to be searched completely in reasonable time. Therefore, an essential part of an efficient B&B algorithm is a set of bounds narrowing the range of feasible values for each variable. Bounds may result from the constraints of the optimization problem, but they may also be constructed using deeper knowledge about the optimization problem that is not explicitly expressed in the constraints. Bounds can either be static for all nodes in the search tree or depend dynamically on the branch that is currently evaluated. No matter what type the bounds are of, their purpose is to cut off branches that will not lead to optimal solutions.

A B&B tree can be traversed in breadth-first or depth-first order. Breadth-first means that all nodes on one level are evaluated regarding their cost, before any nodes on the next level are evaluated. Depth-first search means that for every node all the connected nodes on subordinate levels are examined, before more nodes on the same level are examined. Depending on the situation, one or the other search method may be advantageous. As elaborated below, depth-first search will be preferable for problem (5.9).

A B&B algorithm for our optimization problem

To adapt the generic B&B algorithm to our problem, we need to decide in what order to map the variables to the levels of the search tree, and find good bounds to cut down on the search tree. In order to get good bounds early in the B&B tree, we will branch on the μ_l in the order of decreasing l, i. e., beginning with variable μ_{l_f}, continuing with variable μ_{l_f-1}, and so on. This order is preferable, because, as will become clear below, for determining upper and lower bounds for μ_l all μ_L on the finer levels $L > l$ have to be fixed.

Bounds for μ_l can be derived from the error model. Solving (5.5) for μ_l yields

$$\mu_l = \log_\rho \frac{e_l - e_l^*}{e_{l-1} - e_l^*}. \tag{5.10}$$

Assuming values or bounds for the errors e_{l-1} and e_l, we can calculate bounds for the number of cycles μ_l. For the upper and the lower bound on μ_l common assumptions about e_l can be used. On l_f, the finest level, we set e_{l_f} to the highest admissible error, $e^\dagger$. This means that the optimization will make sure that the final error is smaller or equal to $e^\dagger$, but it will not attempt to reduce the error even further. For the levels $l < l_\mathrm{f}$ the error e_l is assumed to equal e_{l+1} before any multigrid cycles on level $l+1$ are performed. By rearranging (5.5), e_l can be computed from e_{l+1} and μ_{l+1}:

$$e_l \;=\; e^*_{l+1} + \rho^{-\mu_{l+1}} \left(e_{l+1} - e^*_{l+1} \right). \tag{5.11}$$

If e_{l_f} is known, e_l can be calculated for any $l < l_\mathrm{f}$ by recursively evaluating (5.11) for all levels from l to $l_\mathrm{f} - 1$. However, the μ_L for all levels $L > l$ have to be known, too. That is why the variables have to be assigned to the search tree levels in the order of descending full multigrid level.

Now that e_l has been fixed, finding upper and lower bounds for e_{l-1} will provide us with upper and lower bounds for μ_l. ρ is between 0 and 1, and both e_{l-1} and e_l are larger than e^*_l. The logarithm is a monotonically decreasing function for a basis in $(0,1)$ and an argument larger than 0. Therefore, μ_l in (5.10) is maximal if e_{l-1} is maximal. e_{l-1} achieves its maximal value if no multigrid cycles are performed on any of the coarser levels. In this case $e_{l-1} = e^*_0$, because the linear system has been solved exactly on level 0. Therefore, the upper bound on μ_l, denoted by μ^+_l, is the smallest integer in $\mathbb{N}_0$ such that

$$\mu^+_l \geq \log_\rho \frac{e_l - e^*_l}{e^*_0 - e^*_l}. \tag{5.12}$$

For finding a lower bound on μ_l we use the fact that the total error can not be smaller than the discretization error. Hence, $\mu^-_l \in \mathbb{N}_0$ is the smallest integer satisfying

$$\mu^-_l > \log_\rho \frac{e_l - e^*_l}{e^*_{l-1} - e^*_l}. \tag{5.13}$$

The B&B algorithm is initiated by branching on the variable μ_{l_f}, generating child nodes for $\mu_{l_\mathrm{f}} \in \{\mu^-_{l_\mathrm{f}}, \ldots, \mu^+_{l_\mathrm{f}}\}$. Three values are associated with each node:

- μ, the value of μ_{l_f} at this node,
- $\mu_{l_\mathrm{f}} k_{l_\mathrm{f}}$, the cost that is added to the total cost K if $\mu_{l_\mathrm{f}} = \mu$,
- $e_{l_\mathrm{f}-1} = e^*_{l_\mathrm{f}} + \rho^{-\mu_{l_\mathrm{f}}}(e_{l_\mathrm{f}} - e^*_{l_\mathrm{f}})$, the error that will be needed for determining $\mu_{l_\mathrm{f}-1}$ when branching from this node.

The node with the lowest cost is expanded recursively in a depth-first search strategy. On each level l the node with the smallest accumulated cost $\sum_{i=l}^{l_\mathrm{f}} \mu_i k_i$ is selected for further branching. The number of multigrid cycles on level 1 is then uniquely determined by $\mu_1 = \mu^+_1$. As soon as level 1 is reached for the first time, a feasible solution $(\mu_1, \ldots, \mu_{l_\mathrm{f}})^T$ is obtained with the currently best known cost value K_best.

With K_best, the algorithm has found an upper bound on the optimal solution value of problem (5.9). This value may provide a new upper bound for μ_l that is better than the one provided by (5.12). Since k_l is always positive, it does not make

sense to branch on nodes that have already accumulated a cost higher than K_{best}. Therefore, if the maximum $\mu_l^+ \in \mathbb{N}_0$ that fulfills

$$\mu_l^+ \leq \frac{1}{k_l}\left(K_{\text{best}} - \sum_{i=l+1}^{l_\text{f}} \mu_i k_i\right) \tag{5.14}$$

is smaller than the μ_l^+ provided by (5.12), then it is used as upper bound.

The depth-first search is continued until there are no more nodes to investigate. The best known solution vector $(\mu_1, \ldots, \mu_{l_\text{f}})^T$ and K_{best} are updated whenever another, better solution is found during the search. If several solutions achieve the same optimal cost value, the solution with the smallest total error e_{l_f} according to (5.7) is preferred.

Fig. 5.3 shows a theoretical example for a B&B tree. It assumes a 3D computation ($d = 3$) on up to four levels ($l_\text{f} = 3$). The assumed convergence rate of a multigrid cycle is $\rho = 1/4$. The error target is set to $e^\dagger = 2e_3^* = 1/32$. The optimal solution for this problem is $(2,2,1)^T$ with a cost of 35. None of the other branches has to be evaluated all the way down to level 1, except for the branch $(3,0,2)^T$, which has still got a quite promising cost of 30 on level 2.

5.3 Integrating model extensions into the optimization

The model derived in the last section includes all the basic concepts, but it is missing some details that are necessary to represent a "real life" full multigrid algorithm. These details will be integrated into the model in the following. The final model including all the extensions is presented at the end of the section.

Coarsest level is not level 0

The linear system is not necessarily solved on level 0. For example, in HHG the coarsest refinement level at which a tetrahedron has interior unknowns is level 2. Thus, $l_\text{s} \geq 2$ for a full multigrid algorithm within HHG; levels 0 and 1 are not used. Therefore, the optimizer has to allow the user to specify a coarsest level $l_\text{s} \neq 0$. This changes (5.7) and (5.8) into

$$e_{l_\text{f}} = 4^{-l_\text{f}} + \sum_{l=l_\text{s}+1}^{l_\text{f}} 3 \cdot 4^{-l} \rho^{\mu_{l_\text{f}} + \ldots + \mu_l} \quad \text{and} \tag{5.15}$$

$$K = \sum_{l=l_\text{s}+1}^{l_\text{f}} \mu_l k_l \,. \tag{5.16}$$

The optimization algorithm is not changed in principle by this modification, it just has to terminate the recursive search at level $l_\text{s} + 1$. The cost for solving the linear system on level l_s can safely be ignored in the optimization, because l_s is fixed throughout the optimization, and our aim is to find μ_l for $l = l_\text{s} + 1, \ldots, l_\text{f}$.

Arbitrary order of the discretization error

The discretization error depends on the selected discretization method, the smoothness of the solution, and other factors. Usually, only the order of the discretization

error is known. e^* is of order D, if there exists a constant $c_* > 0$ such that, for all levels $l = l_s, \ldots, l_f$,

$$e_l^* \leq c_* 2^{-Dl}. \tag{5.17}$$

D and c_* are assumed to be independent of the level.

Equation (5.5) can be reordered into

$$e_l = \left(1 - \rho^{\mu_l}\right) e_l^* + \rho^{\mu_l} e_{l-1}.$$

Since $\rho^{\mu_l} < 1$ and $e_l^* > 0$, plugging (5.17) into this equation yields

$$\begin{aligned} e_l &\leq \left(1 - \rho^{\mu_l}\right) c_* 2^{-Dl} + \rho^{\mu_l} e_{l-1} \\ &= c_* 2^{-Dl} + \rho^{\mu_l} \left(e_{l-1} - c_* 2^{-Dl}\right). \end{aligned} \tag{5.18}$$

Additional interpolation error

In practice, the error after interpolating the approximate solution from level $l-1$ to level l is not equal to the error on the coarse level. The interpolation error $e_l^\uparrow$ depends on the selected interpolation method. Like the discretization error, it also depends on other characteristics of the PDE, and it is only known approximately. The interpolation error is defined by its order P and a bounding constant $c_\uparrow$, such that

$$e_l^\uparrow \leq c_\uparrow 2^{-Pl}. \tag{5.19}$$

The interpolation error $e_l^\uparrow$ is added to the error e_{l-1}, before the multigrid cycles are performed on level l. Therefore, (5.5) becomes

$$\begin{aligned} e_l &= e_l^* + \rho^{\mu_l} \left(e_{l-1} + e_l^\uparrow - e_l^*\right) \\ &\leq e_l^* + \rho^{\mu_l} \left(e_{l-1} + c_\uparrow 2^{-Pl} - e_l^*\right). \end{aligned} \tag{5.20}$$

Level- and iteration-dependent convergence rates

If a multigrid cycle is run repeatedly, the convergence rates are usually very good in the beginning, but then they deteriorate and approach an asymptotic convergence rate [48]. An example of this behavior is shown in Fig. 5.4. The figure also points out that the level influences the value of the asymptotic convergence rate and the characteristics of approaching that value. On the coarser levels the influence of the boundary is more prominent than on the fine levels, and the type of boundary conditions influences the convergence rates.

To account for varying convergence rates in the error model, we replace the parameter ρ by a set of parameters $\rho_{l,i}$ containing the convergence rate of each multigrid cycle i on each level l. Now, the total convergence rate on level l can not be written simply as ρ^{μ_l}, any more, but has to be written as a product of individual $\rho_{l,i}$. Equation (5.5) becomes

$$e_l = e_l^* + \left(e_{l-1} - e_l^*\right) \prod_{i=1}^{\mu_l} \rho_{l,i}.$$

The convergence rates $\rho_{l,i}$ are often not known in as much detail as depicted in Fig. 5.4, before the solver is actually started. Fortunately, though, the optimization algorithm is fast enough not to harm the overall program run time, even if it is re-run several times during a full multigrid run. Therefore, the optimization should be re-run, whenever new convergence rates are available.

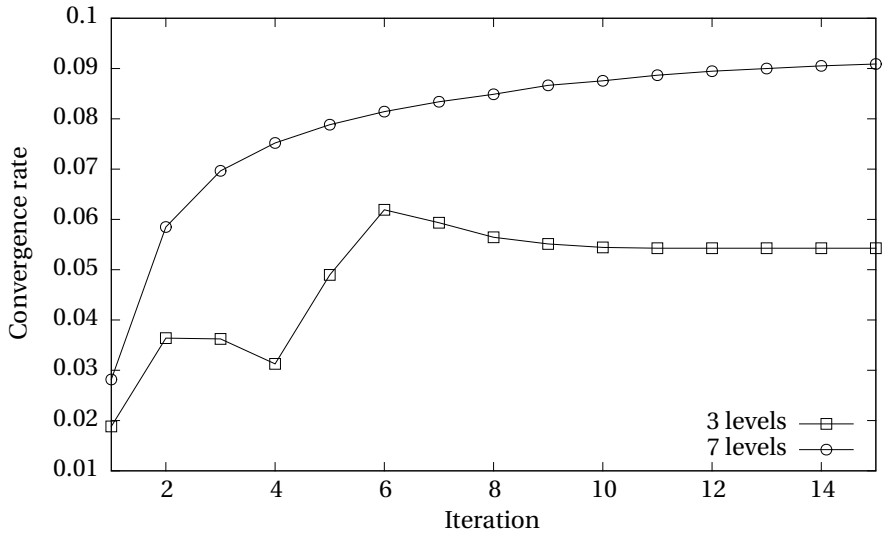

Figure 5.4: Level- and iteration-dependent convergence rates of HHG. The convergence rates are reported for the first 15 iterations on 3 and 7 levels.

Modifications of the B&B algorithm

The extensions discussed in this section do not affect the B&B algorithm in general, but the only modification is the calculation of μ_l^- and μ_l^+ for $l = l_s + 1, \ldots, l_f$. If all extensions from above are included, (5.5) becomes

$$e_l \leq c_* 2^{-Dl} + \left(e_{l-1} + c_\uparrow 2^{-Pl} - c_* 2^{-Dl} \right) \prod_{i=1}^{\mu_l} \rho_{l,i} \,. \tag{5.21}$$

For the lower bound on μ_l the arguments from Section 5.2 still hold. Thus, μ_l^- is the smallest $\mu_l \in \mathbb{N}$ such that

$$\prod_{i=1}^{\mu_l} \rho_{l,i} < \frac{e_l - c_* 2^{-Dl}}{c_* 2^{-D(l-1)} - c_* 2^{-Dl} + c_\uparrow 2^{-Pl}} \tag{5.22}$$

for $l = l_s + 1, \ldots, l_f$.

For determining the upper bound on μ_l, Section 5.2 assumed that no multigrid cycles are performed on the levels from $l_s + 1$ to $l - 1$. The solution obtained on l_s has to be prolongated all the way up to l. The error of a prolongation across more than one level is not included in the present model, neither is such a prolongation implemented in most multigrid solvers. In HHG, a prolongation across several levels is achieved via a sequence of prolongations between subsequent levels. Therefore, we model the prolongation error in that case as a sum of the individual prolongation errors. With this assumption, μ_l^+ is the smallest $\mu_l \in \mathbb{N}$ such that

$$\prod_{i=1}^{\mu_l} \rho_{l,i} \leq \frac{e_l - c_* 2^{-Dl}}{c_* 2^{-Dl_s} - c_* 2^{-Dl} + c_\uparrow \sum_{i=l_s+1}^{l} 2^{-Pi}} \tag{5.23}$$

for $l = l_s + 1, \ldots, l_f$.

Note that (5.22) and (5.23) do not provide, in comparison to (5.13) and (5.12), a direct formula for μ_l^- and μ_l^+. The values can be computed by iteratively multiplying $\rho_{l,i}$ until the bounds are satisfied. In practice, the μ_l are small (below 10), so computing these products and storing them for recurring use does not pose efficiency problems.

5.4 Implementation

The presented multigrid cycle optimization is available in the cycleopt library [23]. It provides interfaces to the C and C++ programming languages; it can also be used from FORTRAN programs via the C interface.

Features

The cycleopt library uses the extended model from Section 5.3 by default. Users can also provide their own error and/or cost models in the form of call-back functions. Instead of using a cost model, it is also possible to specify the actual—level- and iteration-dependent—run times of the multigrid cycles. Doing so captures algorithmic and hardware influences on run time more accurately and yields better optimization results.

It is possible to choose between constant and variable convergence rates. If only one convergence rate is provided, cycleopt uses it for all levels and iterations. For best optimization results it is advisable, though, to specify individual convergence rates for each level and iteration.

The more accurately the convergence rates and costs are known, the better will be the results of the optimization. Therefore, individual $\rho_{l,i}$ and $k_{l,i}$ can be provided to cycleopt once they become available. Cycleopt will then use them in subsequent optimization runs. Ideally, the accurate numbers are available right from the start. That is not the case in reality, but when multigrid cycles are applied to similar linear systems, the convergence rates will be similar. Analogously, as long as the full multigrid solver is run on the same hardware, run times will be similar. Therefore, convergence rates and costs measured during previous full multigrid runs should be used right from the start in future runs on similar systems and/or hardware, allowing the optimizer to return realistic results already in the beginning of a full multigrid run. To facilitate this, the cycleopt library provides functionality for saving the $\rho_{l,i}$ and $k_{l,i}$ in files and loading them in subsequent runs.

Usage

Detailed usage instructions with code examples are available in the library's documentation. Here, we provide some general remarks on using the optimizer in practice.

The convergence rate has been defined as the factor by which the algebraic error is reduced per iteration. In practice, however, the error not known, because the exact solution is not known. Therefore, the convergence rate must be computed using the residual, which is directly connected to the error through the operator of the linear system (see Section 2.3).

Computing the residual norm is not for free, of course. In order to determine all convergence rates, the residual norm has to be computed after every multigrid cycle. To obtain the convergence rate of the first cycle on a level, the norm does even

have be computed after the prolongation from the coarser level. Considering that often only one or two multigrid cycles are performed on each level, the additional cost for obtaining the convergence rates can be quite high. In HHG, a multigrid cycle for a linear system with about $2 \cdot 10^{10}$ unknowns, executed on 1024 cores of a Cray XT4 computer, takes about 3.2 s, while computing the residual norm takes about 1.0 s. This example shows that it would be very valuable to have a method for predicting convergence rates. Comparing the curves for 7 and 3 levels in Fig. 5.4 shows that such predictions may be quite simple in some cases, but will be unreliable in other cases. Therefore, a convergence rate prediction is not yet implemented in cycleopt. Developing good models for estimating convergence rates in different situations would, however, be a worthwhile research topic. The results could easily be included into cycleopt.

5.5 Examples

This section presents an example of a real full multigrid run with HHG, in which the execution time was reduced by 35 % using the presented cycle optimization. Since some interesting optimization results can be better presented with artificial constellations, we first show some theoretical examples.

5.5.1 Theoretical examples

Choosing from multiple solutions with the same cost

For some combinations of parameters the optimization yields multiple solutions. They all have the same cost, but they may lead to different e_{l_f}, because minimizing the final error is not an optimization criterion. In the following example a full multigrid run with the parameters $d = 1$, $\rho = 0.35$, $D = 2$, $c_* = 1$, and $c_\uparrow = 0$ is optimized. The table shows the three optimal solutions for $e^\dagger = 8.6\text{e-}5$, $l_\mathrm{s} = 0$, and $l_\mathrm{f} = 7$. They all have a cost of 596.

μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	e_{l_f}	K
4	3	2	1	3	1	3	8.5928e-5	596
4	1	3	1	3	1	3	8.5524e-5	596
4	1	1	2	3	1	3	8.5897e-5	596

If several optimal solutions are found, the cycleopt library will select the one with the smallest e_{l_f}. From the above solutions, $(4,1,3,1,3,1,3)^T$ would thus be selected.

Skipping levels

The second example reviews an observation we already made in the introduction of the B&B algorithm: the optimal solution may have $\mu_l = 0$ for some l. In practice, however, it may not be feasible to skip levels. Therefore, the cycleopt library can be forced to generate only solutions with $\mu_l > 0$ for all l. For the parameters $d = 1$, $\rho = 0.1$, $D = 2$, $c_* = 1$, $c_\uparrow = 0$, $e^\dagger = 7.2\text{e-}5$, $l_\mathrm{s} = 0$, and $l_\mathrm{f} = 7$, the optimal solution is shown in the table below. It is compared to the optimal solution with the constraint that $\mu_l > 0$, which is more expensive, of course.

μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	e_{l_f}	K
1	0	2	0	2	0	2	7.1964e-5	338
1	1	1	1	1	1	2	6.4081e-5	382

Increasing the finest level

Sometimes the cheapest solution is found for an l_f that is larger than the one given by (5.4). Recall that a multigrid cycle on level l_f reduces only the difference $(e_{l_\mathrm{f}} - e^*_{l_\mathrm{f}})$, but not e_{l_f} itself, by a factor of ρ. Thus, if $e^\dagger$ is very close to $e^*_{l_\mathrm{f}}$, many multigrid cycles are required on l_f to push e_{l_f} below $e^\dagger$. In this case, it may be cheaper to increase l_f, because then the difference $(e_{l_\mathrm{f}} - e^*_{l_\mathrm{f}})$ increases, and the multigrid cycles are more effective in reducing e_{l_f}. The following table shows the optimal solutions for $l_\mathrm{f} = 6$ and $l_\mathrm{f} = 7$ for a full multigrid run with the parameters $d = 1$, $\rho = 0.3$, $D = 2$, $c_* = 1$, $c_\uparrow = 0$, and $e^\dagger = 1.01 e^*_6 = 2.4658\text{e-}4$.

μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	e_{l_f}	K
2	1	2	1	3	5	0	2.4656e-4	456
2	1	1	2	2	1	1	2.4114e-4	304

In practice, it is not always possible to go to the next finer level, e. g., due to storage space restrictions on the computer. The advice to users of the cycleopt library is to set l_f to at least one level higher than suggested by (5.4), if the hardware configuration permits it. The additional level will only be used if it gives a cost reduction, otherwise μ_{l_f} will be set to 0. Increasing l_f increases the run time of the optimization only marginally, because infeasible branches with $\mu_{l_\mathrm{f}} > 0$ will be detected early in the B&B algorithm.

5.5.2 Optimization of an HHG full multigrid run

The efficacy of cycleopt under real-life conditions is demonstrated with a full multigrid run of HHG solving model problem (2.1). Since the analytic solution to this problem is known, the discretization error is accessible, which enables us to document the feasibility of the optimization results. We perform the optimization with the extended error and cost model that was derived in Section 5.3. By comparing the discrete solution found by HHG with the analytic solution, the error bounding constants were identified as $c_* = 91$ and $c_\uparrow = 229$. The orders of the discretization and interpolation operators are $D = P = 2$. Usually, these constants can, for the lack of an analytic solution, not be calculated that exactly, but they can be estimated, e. g., by evaluating the curvature of the solution.

We choose to start the full multigrid run on the coarsest possible level, $l_\mathrm{s} = 2$, and proceed up to $l_\mathrm{f} = 6$. Furthermore, we demand an error target of $e^\dagger = 3.5 e_6 = 0.078$.

Fig. 5.5 compares three full multigrid runs that achieve the given error target, which is marked with a horizontal line. The first run uses an experimentally determined configuration with two V-cycles on every level. Using only one cycle per level is not sufficient; the final error is $e_6 = 1.1$. Two cycles per level yield an acceptable solution with $e_6 = 0.039$ after an execution time of 1.3 s. The figure shows how the full multigrid algorithm reduces the error over time. After solving the linear system

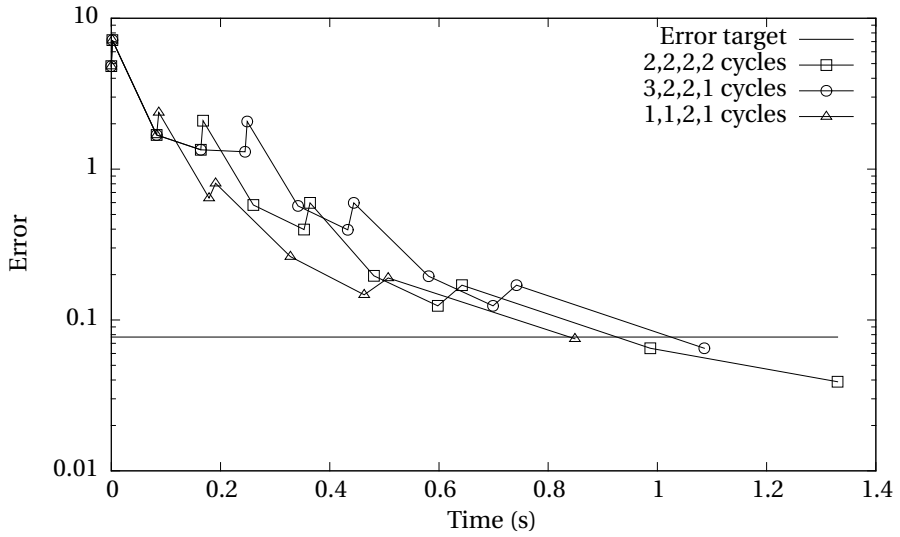

Figure 5.5: Optimization of a full multigrid run in HHG.

on level 2, the initial error is 4.8 (this is the discretization error on level 2). The four prolongations are visible as peaks in the error curve.

The configuration for the second run was determined by the B&B optimization, assuming a constant convergence rate of $\rho = 0.3$. The optimization finds $(3,2,2,1)^T$ on levels 3 to 6 to be the cheapest pattern that fulfills the error bound. A full multigrid run with this cycle pattern takes 1.1 s, which is an improvement of 15 % over the experimental configuration.

For finding the third pattern, $(1,1,2,1)^T$, the optimizer was provided with the convergence rates measured during the previous runs. With this more realistic estimate of the convergence rates at hand the optimizer is able to reduce the execution time by another 0.24 s. The third setup takes 0.85 s, 35% less than the experimental configuration.

5.6 Further model extensions and related work

As the examples show, The optimization yields valuable run time reductions already in its present form. Some ideas for further model extensions that could reduce the run times even more are presented here. Other research groups have made similar efforts to optimize parameters of their multigrid algorithms. We collect their ideas in this section and analyze how they connect to our optimization approach.

The level on which the linear system is solved up to the discretization error, l_s, has always been assumed to be a parameter set by the user. Provided that the costs for solving the linear system on different levels can be provided by the user, l_s could also be treated as a variable in the optimization. On the coarse levels the cost does usually not follow the 2^{-dl} model, any more. In parallel multigrid methods, for example, communication latency becomes more dominant as the number of unknowns per process decreases, which leads to longer run times than predicted by the model. Therefore, it makes sense to start the full multigrid algorithm at a finer

level than the coarsest one possible.

As shown in the examples, it can be advantageous to set l_f to a higher level than the coarsest one required by the error target. Currently, it is left to the user to check for that, but finding the best l_f could also be included in the optimization.

Since the early days of multigrid, researchers have been trying to optimize the *spatial* distribution of computational work, e. g., Brandt [13] and Bai [6]. More recently, De Sterck et al. have pursued research in that direction, too [16]. Their optimization methods aim at reaching a specified error target with minimal work by shifting computational work to regions with a high local error.

Research by Chan et al. on automatically tuning multigrid cycles has been presented at the 2009 Copper Mountain multigrid conference [15]. Their method automatically adapts the recursive structure of W-cycles to given hardware and problem characteristics in order to maximize the cycle's error reduction per run time.

Chapter 6

Conclusion

The main focus of this work was the integration of adaptive refinement into HHG. In order to maintain performance and scalability, it was crucial for the implementation to respect HHG's core concepts, the patch-wise regularity of the meshes and the locality of communication.

The adaptive refinement scheme was developed out of HHG's existing finite element solver. Interpreting the finite element method's basis functions in a hierarchical way allowed for eliminating computation and communication in parts of the finite element mesh without sacrificing the mathematical correctness of the multigrid algorithms. Retaining HHG's locality of communication however required imposing some restrictions on the smoothness of the PDE's right-hand side, which may not always be given. The resulting adaptive refinement scheme demanded for the implementation of special numerical building blocks in HHG. The concept of hyperplanes was used for modeling and implementing new types of data dependencies between mesh primitives at different refinement levels. The new solver is completed by a distributed adaptive mesh refiner. Its scalability to large numbers of processes is ensured by using local communication whenever possible.

As a primer for the work on the adaptive refinement algorithm, HHG's performance and scalability was assessed on different architectures. Therefore, HHG was run on several supercomputers, which were among the largest and fastest available at the time of the assessment. An SCons-based build system was developed for that purpose. It allowed for a quick roll-out on all platforms, including tailoring of build parameters and integration of platform-specific code (e. g., for profiling). Performance data (e. g., execution times) was measured and evaluated with a newly implemented performance analysis toolkit, which is tailored towards the special requirements of multigrid algorithms.

The new adaptive refinement solver was evaluated in experiments with different numerical problems. The tests showed that adaptive refinement is effective and has the potential for drastically reducing the computation and communication effort for solving numerical problems. However, an important finding is also that applying adaptive refinement blindly to a problem with unknown characteristics is dangerous. Under some circumstances (for polynomial problems on highly structured meshes) adaptive refinement can deteriorate the quality of the solution. Even for the problems that are well-suited for adaptive refinement it is still crucial to choose the refined parts of the mesh properly in order to avoid artifacts at the refinement boundaries.

In a separate project, which was conducted together with Alexander Thekale from the Department of Applied Mathematics at the University of Erlangen-Nürnberg, an approach for reducing the computational effort of full multigrid was developed. It uses mathematical optimization to determine the optimal number of V-cycles in each step of the full multigrid algorithm. In practical experiments the approach reduced the computational effort of full multigrid by up to 35 %. The project would probably not have been conducted without the contact to Alexander, which was founded in the interdisciplinary doctorate program *Identification, Optimization, and Steering for Technical Applications* by the *Elite Network of Bavaria* [9]. Thus, the project is a good example that bringing together scientists from different disciplines in such programs is helpful and necessary.

As recent publications by Gmeiner et al. showed, HHG still provides interesting topics for further research (see, e. g., [18]). In particular, the adaptive refinement algorithm provides several new topics. The most important one at the time of this writing is to conduct extended performance and scalability measurements on current hardware. A further task, highly demanding in both the mathematical and the software engineering components, will be to analyze whether higher-order interpolation can be implemented efficiently in HHG. This would help in making the adaptive refinement solver less susceptible to numerical errors at the refinement boundaries.

In the area of algorithm development, one of the most important next steps will be the integration of a (semi-)automatic mesh refiner that uses, e. g., the residual or the solution curvature for determining which parts of the mesh need to be refined. This will enable HHG to be used with adaptive refinement in “real-world” applications, which generally have meshes that are too complicated to be refined manually. Acoustics simulations (e. g., room acoustics) and geophysical simulations (e. g., earth mantle convection) are good candidates, because they combine exactly the requirements that HHG meets especially well: basically irregular geometries that, however, contain large regular areas, and the need for—compared to the domain sizes—extremely fine mesh resolutions.

Appendix A

The complete adaptive refinement algorithm

This appendix collects the functions that implement the adaptive refinement algorithms (Algorithms 15–20). The algorithms are distributed among the following member functions of the class `hhgAdaptiveRefiner`.

- Listing A.1: `refine` (Algorithm 15).
- Listing A.2: `refineInterior` (the *while* loop in Algorithm 15).
- Listing A.3: `refineInteriorUp` (Algorithm 16).
- Listing A.4: `refineInteriorDown` (Algorithm 17).
- Listing A.5: `addRefnBndry` (Algorithm 18).
- Listing A.6: `addMeshBndry` (Algorithm 19).
- Listing A.7: `addSecondaryRefnBndry` (Algorithm 20).

Listing A.1: Implementation of Algorithm 15

```
1  void hhgAdaptiveRefiner::refine
2  (std::vector<hhgScalarVariable*> &unks, hhgScalarVariable &sol,
3  hhgScalarVariable &rhs, hhgScalarVariable &rhb, hhgScalarVariable &res,
4  hhgScalarVariable &reb, hhgOperator &opr, lvl_t lvlStart, lvl_t lvlMax)
5  {
6    lvl_t lvlBase = mesh_.getCoarsestLevel();
7
8    if (lvlStart < lvlBase) {lvlStart = lvlBase;};
9    if (lvlMax == lvlStart) {return;}
10
11   uint nu = unks.size();
12   std::vector<int> unkIdcs(nu); std::vector<bool> unkDMDeps(nu);
13   for (uint iu=0; iu<nu; iu++) {
14     unkIdcs[iu] = unks[iu]->getId(); unkDMDeps[iu] = unks[iu]->hasDMDeps();
15   }
16   varidx_t solIdx = sol.getId(); bool solDMDeps = sol.hasDMDeps();
17   varidx_t rhsIdx = rhs.getId(); bool rhsDMDeps = rhs.hasDMDeps();
18   varidx_t rhbIdx = rhb.getId(); bool rhbDMDeps = rhb.hasDMDeps();
19   varidx_t resIdx = res.getId(); bool resDMDeps = res.hasDMDeps();
```

```
20    varidx_t rebIdx = reb.getId(); bool rebDMDeps = reb.hasDMDeps();
21    varidx_t oprIdx = opr.getId();
22    dim_t dimMax = mesh_.getDimension();
23    hhgPrimitiveStore<double>& store = mesh_.getPrimitiveStore();
24
25    hhgMPIController &mpiCtrl = hhgMPIController::instance();
26    hhgPrimitiveSet<double> pb;
27
28    lvl_t lvlCoar = refineInterior (lvlStart);
29    addRefnBndry (lvlCoar);
30    addMeshBndry (lvlCoar);
31    addSecondaryRefnBndry (lvlCoar);
32  }
```

Listing A.2: Implementation of the *while* loop in Algorithm 15

```
1   lvl_t hhgAdaptiveRefiner::refineInterior (lvl_t lvlCoar)
2   {
3     hhgPrimitiveStore<double>& store = mesh_.getPrimitiveStore();
4
5     for (lvl_t ll=lvlCoar+1, lvlStore=store.getLMax(); ll<=lvlStore; ll++) {
6       store.removePrimitives (hhgPgRefnBndry, ll);
7       store.removePrimitives (hhgPgIntRefnBndry, ll);
8       store.removePrimitives (hhgPgExtRefnBndry, ll);
9     }
10
11    bool changes = true;
12    while (changes == true)
13    {
14      lvl_t lvlCCur = lvlCoar;
15
16      bool changesUp = refineInteriorUp (lvlCCur, lvlCoar);
17      bool changesDown = refineInteriorDown (lvlCCur);
18
19      changes = changesUp || changesDown;
20
21      char changesGlob;
22      char changesChar = changes;
23      MPI_Allreduce
24      (&changesChar, &changesGlob, 1, MPI_SIGNED_CHAR, MPI_LOR, MPI_COMM_WORLD);
25      changes = changesGlob;
26    }
27
28    return lvlCoar;
29  }
```

Listing A.3: Implementation of Algorithm 16

```
1   bool hhgAdaptiveRefiner::refineInteriorUp (lvl_t lvlCCur, lvl_t &lvlCoar)
2   {
3     varidx_t refnIdx = this->refnInfo_->getId();
4     dim_t dimMax = mesh_.getDimension();
5     lvl_t lvlRi = mesh_.getCoarsestLevel();
6     hhgPrimitiveStore<double>& store = mesh_.getPrimitiveStore();
7     lvl_t lvlStore = store.getLMax();
8     hhgMPIController &mpiCtrl = hhgMPIController::instance();
9     bool changes = false;
10
11    hhgPrimitiveSet<double> ws = store.selectGroups (hhgPgWorkingSet, lvlCCur);
12    hhgPrimitiveSet<double> pb = store.selectGroups (hhgPgProcBndry, lvlCCur);
13
14    for (dim_t dim=0; dim<=dimMax; dim++)
```

```
15    {
16      for (psit_t ip=ws.begin(dim), pe=ws.end(dim); ip!=pe; ++ip)
17      {
18        hhgPrimitive<double> &prim = *(*ip);
19
20        lvl_t lvlLower;
21
22        if (dim == 0)
23        {
24          lvlLower = 0;
25        }
26        else
27        {
28          prim.copyFromLocal (refnIdx, lvlRi, hhgCptInteriorToGhost);
29
30          lvlLower = prim.maxLowerNeighLevel (refnIdx);
31
32          lvl_t lvlMaxCommonLower = prim.minLowerNeighLevel (refnIdx);
33          if (lvlMaxCommonLower < lvlCoar) {
34            for (lvl_t ll=lvlMaxCommonLower+1; ll<=lvlCoar; ll++) {
35              store.removePrimitives (hhgPgRefnBndry, ll);
36              store.removePrimitives (hhgPgIntRefnBndry, ll);
37              store.removePrimitives (hhgPgExtRefnBndry, ll);
38            }
39            lvlCoar = lvlMaxCommonLower;
40            changes = true;
41          }
42        }
43
44        lvl_t lvlTrg = hhgMax (lvlLower, prim.refnTriggered (refnIdx));
45        prim.triggerRefn (refnIdx, lvlTrg);
46
47        lvl_t lvlOld = prim.getRefnLevel();
48        if (lvlTrg > lvlOld)
49        {
50          lvlStore = store.provideLevel (lvlTrg);
51
52          if (prim.testFlag (hhgPfProcBndry)) {
53            for (lvl_t ll=lvlOld+1; ll<=lvlTrg; ll++) {
54              if (store.addPrimitive (ll, &prim)) {
55                store.addPrimitive (hhgPgProcBndry, ll, &prim);
56              }
57            }
58          } else {
59            store.addPrimitive (lvlOld+1, lvlTrg, &prim);
60          }
61
62          store.addPrimitive (hhgPgWorkingSet, lvlOld+1, lvlTrg, &prim);
63
64          lvl_t lvlRB = prim.refnBndryTriggered (refnIdx);
65          if (lvlRB) {
66            prim.clearFlag (hhgPfRefnBndry);
67            prim.triggerRefnBndry (refnIdx, 0);
68          }
69        }
70
71        if (dim > 0) {prim.copyToLocal (refnIdx, lvlRi, hhgCptInteriorToGhost);}
72      }
73
74      if (dim < dimMax) {
75        for (psit_t ip=pb.begin (dim), pe=pb.end (dim); ip!=pe; ++ip) {
76          ip->updateDmpDeps (refnIdx, lvlRi, hhgCtDirect, hhgCdSend);
```

```
77        }
78
79        mpiCtrl.updateDim (refnIdx, lvlRi, hhgCtDirect, dim);
80
81        for (psit_t ip=pb.begin (dim), pe=pb.end (dim); ip!=pe; ++ip) {
82          ip->updateDmpDeps (refnIdx, lvlRi, hhgCtDirect, hhgCdReceive);
83
84          if (ip->testFlag (hhgPfGhost)) {
85            lvl_t lvlTrg = ip->refnTriggered (refnIdx);
86            lvlStore = store.provideLevel (lvlTrg);
87            for (lvl_t ll=lvlCCur+1; ll<=lvlTrg; ll++) {
88              if (store.addPrimitive (ll, *ip)) {
89                store.addPrimitive (hhgPgProcBndry, ll, *ip);
90              }
91            }
92          }
93        }
94      }
95    }
96
97    return changes;
98  }
```

Listing A.4: Implementation of Algorithm 17

```
1   bool hhgAdaptiveRefiner::refineInteriorDown (lvl_t lvlCCur)
2   {
3     varidx_t refnIdx = this->refnInfo_->getId();
4     dim_t dimMax = mesh_.getDimension();
5     lvl_t lvlRi = mesh_.getCoarsestLevel();
6     hhgPrimitiveStore<double>& store = mesh_.getPrimitiveStore();
7     lvl_t lvlStore = store.getLMax();
8     hhgMPIController &mpiCtrl = hhgMPIController::instance();
9     bool changes = false;
10
11    hhgPrimitiveSet<double> ws = store.selectGroups (hhgPgWorkingSet, lvlCCur);
12    hhgPrimitiveSet<double> pb = store.selectGroups (hhgPgProcBndry, lvlCCur);
13
14    for (dim_t dim=dimMax-1, dimp=dim+1; dimp>0; dimp--, dim--)
15    {
16      for (psit_t ip=pb.begin (dim), pe=pb.end (dim); ip!=pe; ++ip) {
17        ip->updateDmpDeps (refnIdx, lvlRi, hhgCtAdjacency, hhgCdSend);
18      }
19
20      mpiCtrl.updateDim (refnIdx, lvlRi, hhgCtAdjacency, dim);
21
22      for (psit_t ip=pb.begin (dim), pe=pb.end (dim); ip!=pe; ++ip) {
23        ip->updateDmpDeps (refnIdx, lvlRi, hhgCtAdjacency, hhgCdReceive);
24      }
25
26      for (psit_t ip=ws.begin(dim), pe=ws.end(dim); ip!=pe; ++ip)
27      {
28        hhgPrimitive<double> &prim = *(*ip);
29
30        lvl_t lvlMaxHigher = prim.maxHigherNeighLevel (refnIdx);
31        lvl_t lvlTrg = hhgMax (hhgMax
32        (prim.minHigherNeighLevel (refnIdx),
33        lvlMaxHigher - 1),
34        prim.refnTriggered (refnIdx));
35        prim.triggerRefn (refnIdx, lvlTrg);
36
37        lvl_t lvlOld = prim.getRefnLevel();
```

```
38        if (lvlTrg > lvlOld)
39        {
40          lvlStore = store.provideLevel (lvlTrg);
41
42          if (prim.testFlag (hhgPfProcBndry)) {
43            for (lvl_t ll=lvlOld+1; ll<=lvlTrg; ll++) {
44              if (store.addPrimitive (ll, &prim)) {
45                store.addPrimitive (hhgPgProcBndry, ll, &prim);
46              }
47            }
48          } else {
49            store.addPrimitive (lvlOld+1, lvlTrg, &prim);
50          }
51
52          store.addPrimitive (hhgPgWorkingSet, lvlOld+1, lvlTrg, &prim);
53
54          lvl_t lvlRB = prim.refnBndryTriggered (refnIdx);
55          if (lvlRB) {
56            prim.clearFlag (hhgPfRefnBndry);
57            prim.triggerRefnBndry (refnIdx, 0);
58          }
59
60          changes = true;
61        }
62
63        lvl_t lvlMaxRefnHigher = prim.maxHigherNeighRefnLevel (refnIdx);
64        if ((lvlMaxHigher > lvlTrg) || (lvlMaxRefnHigher > 0))
65        {
66          prim.setFlag (hhgPfRefnBndry);
67
68          lvl_t lvlRB = hhgMax (lvlMaxHigher, lvlMaxRefnHigher);
69          prim.triggerRefnBndry (refnIdx, lvlRB);
70
71          if (lvlTrg < lvlRB-1) {
72            prim.triggerRefn (refnIdx, lvlRB-1);
73            changes = true;
74          }
75        }
76
77        if (dim > 0) {prim.copyToLocal (refnIdx, lvlRi, hhgCptInteriorToGhost);}
78      }
79    }
80
81    return changes;
82  }
```

Listing A.5: Implementation of Algorithm 18

```
1   void hhgAdaptiveRefiner::addRefnBndry (lvl_t lvlCoar)
2   {
3     varidx_t refnIdx = this->refnInfo_->getId();
4     dim_t dimMax = mesh_.getDimension();
5     lvl_t lvlRi = mesh_.getCoarsestLevel();
6     hhgPrimitiveStore<double>& store = mesh_.getPrimitiveStore();
7     lvl_t lvlStore = store.getLMax();
8     hhgMPIController &mpiCtrl = hhgMPIController::instance();
9
10    hhgPrimitiveSet<double> ws = store.selectGroups (hhgPgWorkingSet, lvlCoar);
11    hhgPrimitiveSet<double> pb = store.selectGroups (hhgPgProcBndry, lvlCoar);
12
13    for (dim_t dim=0; dim<dimMax; dim++)
14    {
```

```
15      for (psit_t ip=ws.begin(dim), pe=ws.end(dim); ip!=pe; ++ip)
16      {
17        hhgPrimitive<double> &prim = *(*ip);
18
19        lvl_t lvlRB = prim.refnBndryTriggered (refnIdx);
20
21        if (lvlRB > lvlCoar)
22        {
23          lvlStore = store.provideLevel (lvlRB);
24
25          bool added = store.addPrimitive (lvlRB, &prim);
26
27          if (added && prim.testFlag (hhgPfProcBndry)) {
28            store.addPrimitive (hhgPgProcBndry, lvlRB, &prim);
29          }
30
31          store.addPrimitive (hhgPgRefnBndry, lvlRB, &prim);
32        }
33      }
34
35      for (psit_t ip=pb.begin(dim), pe=pb.end(dim); ip!=pe; ++ip)
36      {
37        hhgPrimitive<double> &prim = *(*ip);
38        if (! prim.testFlag (hhgPfGhost)) {continue;}
39
40        lvl_t lvlRB = prim.refnBndryTriggered (refnIdx);
41
42        if (lvlRB > lvlCoar)
43        {
44          lvlStore = store.provideLevel (lvlRB);
45
46          if (store.addPrimitive (lvlRB, &prim)) {
47            store.addPrimitive (hhgPgProcBndry, lvlRB, &prim);
48          }
49        }
50      }
51    }
52  }
```

Listing A.6: Implementation of Algorithm 19

```
1   void hhgAdaptiveRefiner::addMeshBndry (lvl_t lvlCoar)
2   {
3     varidx_t refnIdx = this->refnInfo_->getId();
4     dim_t dimMax = mesh_.getDimension();
5     lvl_t lvlRi = mesh_.getCoarsestLevel();
6     hhgPrimitiveStore<double>& store = mesh_.getPrimitiveStore();
7     lvl_t lvlStore = store.getLMax();
8     hhgMPIController &mpiCtrl = hhgMPIController::instance();
9
10    hhgPrimitiveSet<double> mb = store.selectGroups (hhgPgBndry, lvlCoar);
11    hhgPrimitiveSet<double> pb = store.selectGroups (hhgPgProcBndry, lvlCoar);
12
13    for (dim_t dim=0; dim<dimMax; dim++)
14    {
15      for (psit_t ip=mb.begin(dim), pe=mb.end(dim); ip!=pe; ++ip)
16      {
17        hhgPrimitive<double> &prim = *(*ip);
18
19        if (dim > 0) {prim.copyFromLocal (refnIdx, lvlRi, hhgCptInteriorToGhost);}
20
21        lvl_t lvlOld = prim.getRefnLevel();
```

```
22        lvl_t lvlMaxHigher = prim.maxHigherNeighLevel (refnIdx, true);
23
24        lvlStore = store.provideLevel (lvlMaxHigher);
25        prim.triggerRefn (refnIdx, lvlMaxHigher);
26
27        bool pb = prim.testFlag (hhgPfProcBndry);
28        for (lvl_t ll=lvlOld+1; ll<=lvlMaxHigher; ll++)
29        {
30          if (store.addPrimitive (ll, &prim))
31          {
32            if (pb) {store.addPrimitive (hhgPgProcBndry, ll, &prim);}
33
34            if (prim.testFlag (hhgPfDirichletBndry)) {
35              store.addPrimitive (hhgPgDirichletBndry, ll, &prim);
36            }
37            else if (prim.testFlag (hhgPfNeumannBndry)) {
38              store.addPrimitive (hhgPgNeumannBndry, ll, &prim);
39            }
40          }
41        }
42
43        if (dim > 0) {prim.copyToLocal (refnIdx, lvlRi, hhgCptInteriorToGhost);}
44      }
45
46      if (dim < dimMax) {
47        for (psit_t ip=pb.begin (dim), pe=pb.end (dim); ip!=pe; ++ip) {
48          ip->updateDmpDeps (refnIdx, lvlRi, hhgCtDirect, hhgCdSend);
49        }
50
51        mpiCtrl.updateDim (refnIdx, lvlRi, hhgCtDirect, dim);
52
53        for (psit_t ip=pb.begin (dim), pe=pb.end (dim); ip!=pe; ++ip) {
54          ip->updateDmpDeps (refnIdx, lvlRi, hhgCtDirect, hhgCdReceive);
55
56          if (ip->testFlag (hhgPfGhost)) {
57            lvl_t lvlTrg = ip->refnTriggered (refnIdx);
58            lvlStore = store.provideLevel (lvlTrg);
59            for (lvl_t ll=lvlCoar+1; ll<=lvlTrg; ll++) {
60              if (store.addPrimitive (ll, *ip)) {
61                store.addPrimitive (hhgPgProcBndry, ll, *ip);
62              }
63            }
64          }
65        }
66      }
67    }
68  }
```

Listing A.7: Implementation of Algorithm 20

```
1   void hhgAdaptiveRefiner::addSecondaryRefnBndry (lvl_t lvlCoar)
2   {
3     varidx_t refnIdx = this->refnInfo_->getId();
4     dim_t dimMax = mesh_.getDimension();
5     lvl_t lvlRi = mesh_.getCoarsestLevel();
6     hhgPrimitiveStore<double>& store = mesh_.getPrimitiveStore();
7     lvl_t lvlStore = store.getLMax();
8     hhgMPIController &mpiCtrl = hhgMPIController::instance();
9
10    hhgPrimitiveSet<double> ws = store.selectGroups (hhgPgWorkingSet, lvlCoar);
11    hhgPrimitiveSet<double> pb = store.selectGroups (hhgPgProcBndry, lvlCoar);
12
```

```
13    for (dim_t dim=1; dim<=dimMax; dim++)
14    {
15      for (psit_t ip=ws.begin(dim), pe=ws.end(dim); ip!=pe; ++ip)
16      {
17        hhgPrimitive<double> &prim = *(*ip);
18
19        prim.copyFromLocal (refnIdx, lvlRi, hhgCptInteriorToGhost);
20
21        if (prim.refnBndryTriggered (refnIdx) > 0) {continue;}
22
23        lvl_t lvlPrim = prim.refnTriggered (refnIdx);
24
25        if (prim.hasLowerNeighRefnLevel (refnIdx, lvlPrim)) {
26          prim.setFlag (hhgPfIntRefnBndry);
27          store.addPrimitive (hhgPgIntRefnBndry, lvlPrim, &prim);
28        }
29
30        if (prim.hasLowerNeighRefnLevel (refnIdx, lvlPrim+1)) {
31          prim.setFlag (hhgPfExtRefnBndry);
32          store.addPrimitive (lvlPrim+1, &prim);
33          store.addPrimitive (hhgPgExtRefnBndry, lvlPrim+1, &prim);
34        }
35      }
36    }
37
38    for (dim_t dim=1; dim<dimMax; dim++)
39    {
40      for (psit_t ip=pb.begin(dim), pe=pb.end(dim); ip!=pe; ++ip)
41      {
42        hhgPrimitive<double> &prim = *(*ip);
43
44        if (! prim.testFlag (hhgPfGhost)) {continue;}
45
46        lvl_t lvlPrim = prim.refnTriggered (refnIdx);
47
48        if (prim.hasLowerNeighRefnLevel (refnIdx, lvlPrim+1)) {
49          if (store.addPrimitive (lvlPrim+1, &prim)) {
50            store.addPrimitive (hhgPgProcBndry, lvlPrim+1, &prim);
51          }
52        }
53      }
54    }
55  }
```

Bibliography

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

[2] T. M. Apostol. *Mathematical Analysis*. Pearson, 1974.

[3] O. Axelsson and V. A. Baker. *Finite Element Solution of Boundary Value Problems*. Vol. 35. Classics in Applied Mathematics. SIAM, 2001.

[4] R. Bader. "New National Supercomputing System at LRZ: SGI Altix 4700". In: *Innovatives Supercomputing in Deutschland (inSiDE)* 4.2 (2006), pp. 30–33.

[5] R. Bader. "Upgrading the SGI Altix 4700 at LRZ". In: *Innovatives Supercomputing in Deutschland (inSiDE)* 5.1 (2007), pp. 6–7.

[6] D. Bai and A. Brandt. "Local Mesh Refinement Multilevel Techniques". In: *SIAM Journal on Scientific and Statistical Computing* 8.2 (1987), pp. 109–134.

[7] R.E. Bank, A. H. Sherman, and A. Weiser. "Some refinement algorithms and data structures for regular local mesh refinement". In: *Scientific Computing, Applications of Mathematics and Computing to the Physical Sciences, Volume I*. Ed. by R. Stepleman et al. IMACS. North-Holland, 1983.

[8] P. Bastian and C. Wieners. "Multigrid Methods on Adaptively Refined Grids". In: *Computing in Science & Engineering* 8 (6 Nov. 2006), pp. 44–54.

[9] Elite Network of Bavaria, ed. *Identification, Optimization and Control with Applications in Modern Technologies*. Dec. 22, 2014. URL: https://www.elitenetzwerk.bayern.de/doktorandenkollegs/doctorate-programs-according-to-fields-of-study/identification-optimization-and-control-with-applications-in-modern-technologies/.

[10] B. Bergen. "Hierarchical Hybrid Grids: Data Structures and Core Algorithms for Efficient Finite Element Simulations on Supercomputers". Dissertation. July 2006.

[11] J. Bey. "Tetrahedral grid refinement". In: *Computing* 55.4 (1995), pp. 355–378.

[12] OpenMP Architecture Review Board, ed. *The OpenMP API specification for parallel programming*. Nov. 22, 2014. URL: http://openmp.org/.

[13] A. Brandt. *Multigrid Techniques: 1984 Guide with Applications to Fluid Dynamics*. Vol. 85. GMD-Studien. GMD, 1984.

[14] A. Brandt. "Multi-level adaptive solutions to boundary-value problems". In: *Math. Comp.* 31 (1977), pp. 333–390.

[15] C. P. Chan et al. "A Dynamic Programming Approach to Autotuning Multigrid". In: *Numerical Linear Algebra with Applications* (to be published in 2009).

[16] H. De Sterck et al. “Efficiency-based h- and hp-refinement strategies for finite element methods”. In: *Numerical Linear Algebra with Applications* 15 (2008), pp. 89–114.

[17] The SCons Foundation, ed. *SCons*. Nov. 4, 2014. URL: http://www.scons.org/.

[18] B. Gmeiner. “Design and Analysis of Hierarchical Hybrid Multigrid Methods for Peta-Scale Systems and Beyond”. Dissertation. 2013.

[19] B. Gmeiner. “Extension of a Software Package for Hierarchical Hybrid Grids”. Master’s thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2009.

[20] B. Gmeiner et al. “Parallel multigrid on hierarchical hybrid grids: a performance study on current high performance computing clusters”. In: *Concurrency and Computation: Practice and Experience* 26.1 (2014), pp. 217–240.

[21] GNU, ed. *Autotools*. Nov. 4, 2014. URL: http://www.gnu.org/software/.

[22] M. S. Gockenbach. *Understanding and Implementing the Finite Element Method*. SIAM, 2006.

[23] T. Gradl, ed. *Cycleopt*. Nov. 16, 2014. URL: http://www.sourceforge.net/projects/cycleopt/.

[24] T. Gradl and U. Rüde. “Massively Parallel Multilevel Finite Element Solvers on the Altix 4700”. In: *Innovatives Supercomputing in Deutschland (inSiDE)* 5.2 (2007), pp. 24–29.

[25] M. Griebel. “Zur Lösung von Finite-Differenzen- und Finite-Element-Gleichungen mittels der Hiearchischen-Transformations-Mehrgitter-Methode”. Dissertation. TU München, 1990.

[26] G. Hager et al. “RZBENCH: Performance Evaluation of Current HPC Architectures Using Low-Level andăApplication Benchmarks”. In: *High Performance Computing in Science and Engineering, Garching/Munich 2007*. Ed. by S. Wagner et al. Springer, 2009, pp. 485–501.

[27] ISO, ed. *ISO/IEC 9899:1999. Programming languages—C*. Dec. 7, 2014. URL: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=29237.

[28] Forschungszentrum Jülich, ed. *JUGENE - Configuration*. Nov. 9, 2014. URL: http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUGENE/Configuration/Configuration_node.html.

[29] A. Kennedy and H. Lederer. “DEISA Extreme Computing Initiative (DECI) and Science Community Support”. In: *Parallel Computing: From Multicores and GPU’s to Petascale*. Ed. by B. Chapman et al. Vol. 19. Advances in Parallel Computing. 2010, pp. 482–491.

[30] Argonne National Laboratory, ed. *The MPI Standard*. Nov. 22, 2014. URL: http://www.mcs.anl.gov/research/projects/mpi/standard.html.

[31] S. Lang and G. Wittum. “Large-scale density-driven flow simulations using parallel unstructured Grid adaptation and local multigrid methods”. In: *Concurrency and Computation: Practice and Experience* 17 (11 2005), pp. 1415–1440.

[32] UoE HPCX Ltd, ed. *HECToR Phase 1 Hardware Configuration*. Nov. 9, 2014. URL: http://www.hector.ac.uk/service/hardware/phase1.php.

[33] A. Malony et al. “Advances in the TAU Performance System”. In: *Tools for High Performance Computing 2011*. Ed. by H. Brunst et al. Springer, 2012, pp. 119–130.

[34] K. Martin and B. Hoffman. *Mastering CMake*. Kitware, Inc., 2013.

[35] S. F. McCormick. *Multilevel Adaptive Methods for partial Differential Equations*. Frontiers in Applied Mathematics. SIAM, 1989.

[36] G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization*. Wiley, 1988.

[37] W. H. Press et al. *Numerical Recipes in FORTRAN 77*. Cambridge Press, 1993.

[38] D. Ritter and U. Rüde. “Experimental analysis of a Fast Adaptive Composite-based grid expansion scheme for open boundary problems”. In: *Numerical Linear Algebra with Applications* 19.2 (2012), pp. 268–278. DOI: 10.1002/nla.1807.

[39] U. Rüde. *Mathematical and Computational Techniques for Multilevel Adaptive Methods*. Vol. 13. Frontiers in Applied Mathematics. SIAM, 1993.

[40] H. R. Schwarz. *Numerische Mathematik*. B. G. Teubner, 1997.

[41] V. V. Shaidurov. *Multigrid Methods for Finite Elements*. Vol. 318. Mathematics and Its Applications. Kluwer Academic Publishers, 1995.

[42] D. Skinner et al. *Integrated Performance Monitoring*. Nov. 6, 2014. URL: http://ipm-hpc.sourceforge.net/.

[43] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 2013.

[44] K. Stüben and U. Trottenberg. “Multigrid methods: Fundamental algorithms, model problem analysis and applications”. In: *Multigrid Methods*. Ed. by W. Hackbusch and U. Trottenberg. Vol. 960. Lecture Notes in Mathematics. Springer, 1982, pp. 1–176.

[45] E. Süli and D. F. Mayers. *An Introduction to Numerical Analysis*. Cambridge Press, 2000.

[46] University of Tennessee, ed. *Performance Application Programming Interface (PAPI)*. Dec. 21, 2014. URL: http://icl.cs.utk.edu/papi/index.html.

[47] A. Thekale et al. “Optimizing the number of multigrid cycles in the full multigrid algorithm”. In: *Numerical Linear Algebra with Applications* 17.2–3 (2010), pp. 199–210.

[48] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Elsevier, 2001.

[49] R. Verfürth. *A Review of A Posteriori Error Estimation and Adaptive Mesh-Refinement Techniques*. Wiley-Teubner, 1996.