Design and Analysis of Hierarchical Hybrid Multigrid Methods for Peta-Scale Systems and Beyond

Design und Analyse für hierarchisch hybride Mehrgittermethoden auf Peta-Scale Systemen und darüber hinaus

Der Technischen Fakultät
der Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur
Erlangung des Grades

Doktor-Ingenieur

vorgelegt von

Björn Gmeiner, M. Sc.
aus München
Als Dissertation genehmigt von
der Technischen Fakultät der
Friedrich-Alexander-Universität Erlangen-Nürnberg

Tag der mündlichen Prüfung: 20.09.2013
Vorsitzende des Promotionsorgans: Prof. Dr. Marion Merklein
Gutachter: Prof. Dr. Ulrich Rüde
Prof. Dr. Hans-Joachim Bungartz
Abstract

Earth mantle convection is the driving mechanism for plate tectonics, continental drift, volcanism, mountain formations, oceans, and genesis of earthquakes. Since the inside of the planet is physically inaccessible, observations and simulations are the only way to derive knowledge of the ongoing physics. The mantle itself behaves like an extremely viscous fluid (Stokes flow) which is driven by a heat flux through thermal advection and secular cooling.

To simulate fine scale effects of such physical processes, we have to resolve the domain with several thousands of grid points per dimension. This eventually leads to huge (sparse) systems of equations given by elliptic partial differential equations. Due to their optimal algorithmic complexity, multigrid methods would be an obvious choice to solve such large systems. Hierarchical Hybrid Grids construct a geometric multigrid procedure on a regularly refined (unstructured) mesh. Moreover, the resulting semi-structured meshes allow a matrix-free implementation with reduced memory consumption and high efficiency in terms of number of processed grid points per time. Additionally, this kind of meshes naturally provides a clear domain decomposition between the structured blocks for a parallel execution, but allows at the same time a certain geometric flexibility.

One part of this thesis covers the computational and numerical performance of this approach. This includes the validation of the convergence of the multilevel approach as well as the tuning for degenerated finite element grids. The generation of the coarsest mesh is revised, and hybrid parallelization is introduced to the framework in order to allow scalability on current peta-scale clusters. To this extent, scalability studies are presented to determine and analyze the achieved performance of the software.

In a second step, Hierarchical Hybrid Grids are driven towards applications i.e. to the Earth mantle convection problem, as motivated before. Therefore, the software was extended from a solver for the Poisson equation to a solver for variable coefficient problems and the Stokes system. One important issue here is to maintain the matrix-free, stencil-based approach leading to on-the-fly calculations for the stencils. The Stokes system is solved by a pressure correction scheme on a stabilized finite element discretization. It is shown on a shell geometry, that it is possible to resolve the Earth mantle uniformly with a resolution in excess of one kilometer. This leads to the solution of a linear saddle point problem of a trillion ($10^{12}$) unknowns. Such problem sizes pose a challenge for current clusters by requesting hundreds of thousands of compute cores and millions of threads. For the parallel execution, a detailed predictive performance model is developed for computation and communication.
to validate the achieved performance and compare it to the best possible performance. Overall, a prototype of fast and highly scalable Earth mantle convection software is designed and analyzed.
Acknowledgements

First of all, I would like to thank Prof. Dr. Ulrich Rüde for supervising my work, the many valuable discussions, support, and making different collaborations possible. I wish to thank Prof. Dr. Hans-Joachim Bungartz for taking his time to review my thesis.

Thanks to Harald Köstler for all the intensive discussions and fun, including a bit offside research direction. I want to thank my roommate Dominik Bartuschat for the nice time during work and beyond. Moreover, I appreciate the work and spare time with Armin Talai. In addition, I want to thank Gaby Fleig, Kristina Pickl, and Daniel Ritter for proofreading, as well as all of my colleagues at LSS for entertaining and assistance.

Thank you, Johann Rudi, Carsten Burstedde, and Omar Ghattas for the interesting three months in Austin. Further thanks to Carmen Rodrigo and Francisco Gaspar for the great weeks in Zaragoza.

A lot of thanks to Jens Weismüller, Matthias Hofmann, Marcus Mohr, Jan Treibig, Christian Waluga, and Barbara Wohlmuth, Hans-Peter Bunge, Prof. Dr. Gerhard Wellein, and the whole Terra-Neo team for the first very productive months.

Finally, I would like to thank my girlfriend Krasimira Ilieva and my family for their support and help.

The work was supported by the International Doctorate Program (IDK) within the Elite Network of Bavaria. I gratefully acknowledge the Gauss Centre for Supercomputing (GCS) for providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS share of the supercomputer JUGENE and JUQUEEN at Jülich Supercomputing Centre (JSC). GCS is the alliance of the three national supercomputing centres HLRS (Universität Stuttgart), JSC (Forschungszentrum Jülich), and LRZ (Bayerische Akademie der Wissenschaften), funded by the German Federal Ministry of Education and Research (BMBF) and the German State Ministries for Research of Baden-Württemberg (MWK), Bayern (StMWFK) and Nordrhein-Westfalen (MIWF).
## Contents

1 Introduction ........................................ 1
   1.1 Contributions ..................................... 2
   1.2 Outline ........................................... 2

2 Hierarchical Hybrid Grids .......................... 5
   2.1 Finite Elements ...................................... 7
      2.1.1 Variational Formulation ...................... 8
      2.1.2 Approximation ................................ 9
      2.1.3 Stokes System ................................. 10
   2.2 Multigrid ........................................... 12
      2.2.1 Multigrid Schemes ............................ 12
      2.2.2 Multigrid Cycles ............................. 13
      2.2.3 Transfer Operators ............................ 15
      2.2.4 Coarse Grid Operators ....................... 16
      2.2.5 Parallel Multigrid ............................ 17

I Convergence Rate Optimization by Local Fourier Analysis 19

3 Local Fourier Analysis on Structured Grids ....... 21
   3.1 Setup and Assumptions ............................. 22
   3.2 Regular Tetrahedron ............................... 23
   3.3 Optimized Tetrahedron ............................. 24
   3.4 Degenerated Tetrahedra ............................ 26
      3.4.1 Wedge ........................................ 28
      3.4.2 Spade ......................................... 28
      3.4.3 Needle ........................................ 29
      3.4.4 Spindle ....................................... 29
      3.4.5 Sliver and Cap ................................ 29

4 Numerical Experiments on Semi-Structured Meshes 31
   4.1 Computational Cost for the Implementation ....... 31
   4.2 Collective Smoothing ............................... 32
      4.2.1 Anisotropic Smoothing ........................ 33
      4.2.2 Locally Adaptive Smoothing ................... 34
Contents

4.3 Locally Adaptive Damping and Smoothing Steps . . . . . . . . . . . . . . 35
  4.3.1 Unoptimized Version and Exact Block-wise Solving . . . . . . . . 36
  4.3.2 Locally Adaptive Number of Smoothing Steps . . . . . . . . . . . 37
  4.3.3 Locally Adaptive Damping . . . . . . . . . . . . . . . . . . . . . . 39
  4.3.4 Combined Methods . . . . . . . . . . . . . . . . . . . . . . . . . . 39

II Framework Extensions 41

5 Variable Coefficients 43
  5.1 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
  5.2 $\tau$-Extrapolation . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
    5.2.1 Smooth, Variable Dielectricity . . . . . . . . . . . . . . . . . . . 46
    5.2.2 Jumping, Piecewise Constant Dielectricity . . . . . . . . . . . . . 46
  5.3 Simulation of Porosified Structures . . . . . . . . . . . . . . . . . . . . 48
    5.3.1 Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
    5.3.2 Simulation and Post-processing . . . . . . . . . . . . . . . . . . . 49

6 Stokes System 53
  6.1 Galerkin/Least-squares Stabilization . . . . . . . . . . . . . . . . . . . . 53
  6.2 Pressure Correction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
  6.3 Boundary Conditions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
  6.4 Lid-driven Cavity Example . . . . . . . . . . . . . . . . . . . . . . . . . 56
  6.5 Mantle Convection-Diffusion Simulation . . . . . . . . . . . . . . . . . . 56

III Large Scale Experiments 63

7 Scalability 65
  7.1 Architectures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
  7.2 Unstructured Input Meshes . . . . . . . . . . . . . . . . . . . . . . . . . 67
    7.2.1 Gmsh Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
    7.2.2 Validation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
  7.3 Massively Parallel Mesh Treatment . . . . . . . . . . . . . . . . . . . . 69
    7.3.1 Hybrid Parallelization . . . . . . . . . . . . . . . . . . . . . . . 70
    7.3.2 Weak Scaling on JUGENE . . . . . . . . . . . . . . . . . . . . . 72
    7.3.3 Weak Scaling on JUQUEEN . . . . . . . . . . . . . . . . . . . . 75
    7.3.4 Strong Scaling . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
  7.4 Comparison with Other Peta-Scale Clusters . . . . . . . . . . . . . . . . . 76
  7.5 Icosahedral Mesh . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
    7.5.1 General . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
    7.5.2 Tetrahedral Decomposition . . . . . . . . . . . . . . . . . . . . . 82
    7.5.3 Diffusion Problem . . . . . . . . . . . . . . . . . . . . . . . . . . 83
1 Introduction

Convection in the Earth’s mantle is the driving force behind large-scale geologic activity such as plate tectonic and continental drift. Phenomena like e.g. earthquakes, mountain and ocean formations, and hot-spot volcanism are direct consequences of this process. The domain reaches from tens of kilometers below the Earth surface to a depth of 2,900 km to the core. Driven by temperature differences between core and surface as well as (internal) radioactive heating by the rock, thermal advection transports the heat predominantly in vertical direction.

The very viscous and slow (Stokes) flow in the mantle is modeled by partial differential equations in space and time. Paleomagnetic methods [12] are able to reconstruct movements of the plates back to about 130 million years and thus help to provide boundary conditions. Compositional models of the Earth give information about current seismic velocities and as such act as terminal conditions. This eventually leads to an inverse problem. Simulations require very high resolutions to capture pressure and temperature effects accurately. During this process, the two-sphere geometry is mapped to a computational mesh. The elliptic nature of the underlying differential equations leads to a large system of linear equations that have to be solved. The general process is well known [13, 14] and can be captured by moderate resolution. However, it turns out that regional deviations of the rheology have significant impact [15, 16], too. While most state of the art mantle convection codes reach their limit of scalability at resolutions of about 10 km [17, 18], we aim for resolutions down to 1 km.

Such resolutions result in linear systems that consist of over $10^{11}$ finite elements and unknowns. The solution of those can only be achieved by an optimal solver, which means that the workload has to be proportional to the number of unknowns. Geometric multigrid methods combine this optimality with a high computational efficiency. On the downside these algorithms have to be tuned to the underlying problem. Here, e.g. degeneration of the mesh or strong variations in the viscosity field deteriorate the convergence. To accomplish extreme resolutions computationally, one has to program highly parallel supercomputers. The increasing parallelism forces us to partition the domain into very many sub-problems, where performance analysis and debugging are challenging tasks in themselves. This thesis is driven to develop and investigate a geometric multigrid solver for such kind of problems. In addition, we enrich the Hierarchical Hybrid Grids (HHG) software framework by algorithms such that it can now serve as a prototype for an upcoming mantle-convection program Terra-Neo.
1.1 Contributions

The HHG framework implements a parallel geometric multigrid algorithm on refined semi-structured finite element meshes. Moreover, it exploits the geometric structure of the underlying mesh allowing matrix-free stencil-based finite element operator evaluations.

One goal within this thesis is to transfer the HHG approach to more general equations. These extensions are motivated by applications such as geophysics, molecular dynamics, chemical etching, or high dynamic range compression. Extending the framework is not always straightforward, since we want to retain a high computational, as well as parallel, efficiency. In this context, performance modeling is a key to understand and improve the achieved performance. The contributions are specifically to:

- Construct a multigrid method that block-wise adapts the smoother to anisotropies in the discretization by local Fourier analysis predictions.
- Show that the HHG approach is well suited to efficiently solve variable coefficient differential equations or the Stokes equations within applications.
- Adapt the HHG software package and our multigrid method such that it scales up to hundred thousands of cores on supercomputing machines like JUGENE, JUQUEEN, or SuperMUC. Some fundamental changes to HHG were necessary, e.g., the way to create the data structures for tetrahedral input grids efficiently in parallel.
- Develop diagnostic performance models for our multigrid solver on different clusters via suitable performance and communication models.

While the focus of the first part is more on numerical and application oriented topics, the second part is devoted to high performance computing.

1.2 Outline

The next section introduces the Hierarchical Hybrid Grids approach including the finite element method and the single components of the multigrid algorithm. In addition, the Stokes system is set up in Schur complement form, as it is often done for Earth mantle convection problems.

The first part shows the effectiveness of different smoothers for degenerated tetrahedral elements. For those smoothers, convergence rates of HHG and predictions resulting from local Fourier analysis are compared. With these smoothers and smoothing parameters, a multigrid algorithm is constructed for semi-structured meshes.

The next part introduces a matrix-free variable coefficient implementation for HHG. It is experimentally shown that $\tau$-extrapolation can improve the consistency to fourth order for
jumping as well as for variable coefficient problems. A model motivated by an etching process is developed and parameterized to generate irregular porousified meshes for electromagnetic simulations. Furthermore, the framework extensions are presented to solve mantle convection-diffusion equations.

The third part presents changes of the program which are necessary to achieve excellent computational and parallel efficiency. This is demonstrated and compared on different supercomputers. We split an icosahedral mesh into tetrahedral elements to enable computations on the shell of a planet. On this geometry, large scale runs are performed. The last section analyzes the performance of the previous experiments. Therefore, we develop performance models for the computation as well as the communication of our multigrid algorithm. Finally, we compare HHG and the mantle-convection framework TERRA in terms of operator performance.
2 Hierarchical Hybrid Grids

Hierarchical Hybrid Grids (HHG) is an Message Passing Interface (MPI)\(^1\) parallel software package [19] that combines the advantages of structured and unstructured grids in a block-structured approach. The basic idea behind the data layout is explained with a focus on two-dimensions in [20, 21]. The basic framework is explained in detail in the dissertation of Benjamin Bergen [22].

In 2006 the used principles led to solving a finite element system with \(1.7 \cdot 10^{10}\) unknowns on an SGI Altix supercomputer using 1 024 nodes with an overall performance of 0.96 TFLOP/s [23]. Two years later, it was shown that it is possible to solve a linear system with \(10^{11}\) unknowns in about 1.5 minutes on almost 10 000 processors on Itanium 2 cores of the HLRB II at Leibnitz-Rechenzentrum (LRZ) in Garching [24, 25] by Tobias Gradl. Further, he determined the optimal number of MG cycles on each level to reduce the computational cost of the full MG method by a Branch and Bound algorithm [26] within HHG.

In HHG, an unstructured finite element input grid is split into macro-elements: vertices, edges, faces, and volumes. These macro-elements are refined in a structured way resulting in a block-structured or semi-structured grid (see figure 2.1 for the two-dimensional case) and are stored contiguously in memory.

In the case of tetrahedral grids, the input grid is refined by Bey’s refinement strategy [27]. Here, each input tetrahedron is subdivided into eight child tetrahedra of equal volume in such a way that each corner of a child coincides with either a corner or an edge midpoint of the parent. An example of a tetrahedral macro-element is given in figure 2.2.

The HHG data layout preserves the flexibility of unstructured meshes, while the regular internal structure of the primitives allows for an efficient implementation on current computer architectures, especially on parallel computers.

The regularity of the refined grids can be exploited in such a way that it is no longer necessary to explicitly assemble and store any discretization matrix. Instead, the matrix is implicitly stored by stencils that are constant for each macro-element. Semi-structured meshes also support local tuning of the smoothing parameters which will be discussed in the first part of this work.

\(^{1}\)www.mcs.anl.gov/mpi
Figure 2.1: Splitting of two triangle input elements into HHG macro-elements after two steps of refinement. Additionally, the memory representation of a refined triangle with a 7-point stencil for the lower left inner point is sketched.

Figure 2.2: Left: one refinement step of a single tetrahedral macro-element, middle: two refinement steps creating an inner node, right: the outermost faces of three refinement steps
The resulting hierarchy of meshes is the basis for a parallel geometric multigrid algorithm (MG). Here, we distribute the unstructured tetrahedral HHG input grid to different processes and introduce one ghost layer at each boundary between neighboring processes. Other parallel MG approaches are outlined in chapter 7.

For communication of the ghost layers, the ghost points of the vertices and edges are copied from local memory to MPI buffers. The ghost layers of the faces are stored directly inside MPI buffers.

HHG is written in object oriented C++ in order to provide maintainability. The software structure is illustrated in figure 2.3. Only the representative layers of the core classes are shown. In total, HHG consists of around 150 classes that are distributed over more than 350 files.

The algorithm layer determines the flow control of the MG algorithm as well as basic operations, e.g. norms or arithmetic operations on variables. The next layer includes the assembly of the differential finite element operators and describes (scalar) field variables.

The mesh class and associated classes like the primitiveStore manage the macro-primitives of the current process and the connectivity to adjacent primitives. Each primitive is represented by a vertex, edge, face, or volume object in the primitive layer. The storage of the primitive objects references to its memory representations for each variable of the differential equation.

The memory storage itself, communication features like ghost layer exchanges and MPI control, as well as the actual execution of operations on the memory are swapped out. The computational kernels are called in a low-level C function-oriented fashion. This allows to integrate other programming languages easily and e.g. use Fortran for the compute-intensive kernels, if this is advantageous for the performance on a specific computer system. Further, these interfaces could be beneficial to incorporate hardware accelerators on heterogeneous architecture.

### 2.1 Finite Elements

The finite element (FE) method [28, 29, 30, 31] is a numerical technique to find an approximate solution of a partial differential equation (PDE). It splits the space (and possibly time) into small elements and defines dependencies to each other. FEs allow to discretize complicated domains, and local refinement of the underlying mesh. Often it is the method of choice, if the solution changes w.r.t. time or the solution lacks smoothness.

Originally invented in the field of elasticity and structural mechanics problems, a lot of mathematical theory was developed over the past decades. Further, the method was transferred to other application fields e.g. fluid dynamics [32] or electromagnetics [33].
2.1.1 Variational Formulation

We define the second-order linear scalar elliptic PDE

\[-\nabla \cdot (\epsilon(x) \nabla u(x)) = f(x) \text{ in } \Omega \]
\[u(x) = g(x) \text{ on } \partial \Omega\]

(2.1)

(2.2)

with a bounded Lipschitz domain \( \Omega \subseteq \mathbb{R}^d \), and its boundary \( \partial \Omega \), the coefficient \( \epsilon(x) \in L^2(\Omega) \), and the right hand side \( f(x) \in L^2(\Omega) \). Furthermore \( \Omega \) must be simply connected, open and bounded.

FEs work on the variational or weak formulation of the classical problem (2.1). In a first step, a test function \( v \in V \) is chosen. For the presented problem we choose the Sobolev space with compact support \( V := H^1_0 \) in the following way:

\[H^1(\Omega) := \{u : \Omega \rightarrow \mathbb{R} | u \in L^2(\Omega)\} \]

(2.3)

with weak derivatives \( \partial u \in L^2(\Omega) \) in each direction of \( d \)}

(2.4)

\[H^1_0(\Omega) := \{u \in H^1(\Omega) | u = 0 \text{ at } \partial \Omega\}.\]

(2.5)

A definition of the weak derivative is given e.g. in [30]. Multiplying with the test function and integrating over \( \Omega \) yields:

\[- \int_\Omega \nabla \cdot (\epsilon(x) \nabla u(x)) v(x) dx = \int_\Omega f(x) v(x) dx \text{ for } \forall v \in V.\]

(2.6)

Applying Green’s formula gives

\[\int_\Omega \epsilon(x) \nabla u(x) \cdot \nabla v(x) dx - \int_{\partial \Omega} \frac{\epsilon(x) \partial u(x)}{\partial n} \cdot v(x) dx = \int_\Omega f(x) v(x) dx \text{ for } \forall v \in V.\]

(2.7)
The boundary integral vanishes due to the compact support of the test function on the Dirichlet boundary:

$$
\int_{\Omega} \varepsilon(x) \nabla u(x) \cdot \nabla v(x) \, dx = \int_{\Omega} f(x) v(x) \, dx \text{ for } \forall v \in V.
$$

We define the bilinear form \( a(\cdot, \cdot) \) and the linear form \( b(\cdot) \) by

$$
a(u, v) := \int_{\Omega} \varepsilon(x) \nabla u(x) \cdot \nabla v(x) \, dx \quad (2.9)
$$

$$
b(v) := \int_{\Omega} f(x) v(x) \, dx. \quad (2.10)
$$

For constant coefficients \( \varepsilon(x) = \text{const.} \) we define Poisson’s equation to

$$
a_c(u, v) := \varepsilon \int_{\Omega} \nabla u(x) \cdot \nabla v(x) \, dx. \quad (2.11)
$$

### 2.1.2 Approximation

The discretization replaces the infinite dimensional space \( V \) by the finite dimensional subspace \( V_h \). Conforming FE even assume that \( V_h \subset V \). The Galerkin approach is to find a \( u_h \in V_h \) such that

$$
a(u_h, v) = b(v) \text{ for } \forall v \in V_h. \quad (2.12)
$$

Some additional requirements are assumed for the domain \( \Omega \), with \( K \) being an element, and \( \tau_h \) the triangulation of \( \Omega \) [30]:

- \( \Omega \in \mathbb{R}^d \) has a polygonally shaped boundary,
- \( \Omega \cup \partial \Omega \) are decomposed in closed elements (with boundary), such that \( \Omega \cup \partial \Omega = \bigcup_{K \in \tau_h} K \),
- For \( K, K' \in \tau_h : \text{int}(K) \cap \text{int}(K') \neq \emptyset \) with \( \text{interior}(K) \) as an open element (without boundary),
- If \( K \cap K' \neq \emptyset \) then \( K \cap K' \) is either a point, a common edge, or a common face of \( K \) and \( K' \).

The last three requirements are necessary to guarantee a conforming triangulation. Since we deal with linear tetrahedral FE throughout the thesis, we define the space of the test functions \( V_h \) as follows

$$
V_h := \{ u \in C(\Omega \cup \partial \Omega) \mid u|_K \in \mathbb{P}_1(K) \text{ for } \forall K \in \tau_h \}, \quad (2.13)
$$

where \( \mathbb{P}_1(K) \) are polynomials of first order and \( u|_K \) is \( u \) on the element \( K \). The accuracy of tetrahedral finite elements w.r.t. their maximal angle is discussed e.g. in [34].
Next, we write the approximation $u_h$ as a linear combination of the basis vectors $\phi_i$

$$u_h(x) = \sum_{i=1}^{N} u_i \phi_i(x), \quad (2.14)$$

with node indices $i = 1..n$ given by the triangulation. Naturally, one uses the nodal basis functions $\phi_j$ for $v$.

Inserting the approximation $u_h(x)$ and the nodal basis functions for $v$ in (2.12) gives

$$\sum_{i=1}^{n} u_i a(\phi_i, \phi_j) = b(\phi_j) \text{ for } \forall j = 1, ..., n. \quad (2.15)$$

We define the stiffness matrix $A_h$ by

$$A_h = a_{ij} = a(\phi_i, \phi_j), \quad (2.16)$$

and the right hand side $f_h$ by

$$f_h = f_j = b(\phi_j). \quad (2.17)$$

One can assemble the global stiffness matrix by summing over all nodes $n$ the local stiffness matrices $a_{ij}^{(m)}$ of each element $K_m$:

$$a_{ij} = a(\phi_i, \phi_j) = \sum_{m=1}^{n} a_{ij}^{(m)} = \sum_{m=1}^{n} \int_{K_m} \varepsilon(x) \nabla \phi_i \cdot \nabla \phi_j dx. \quad (2.18)$$

The elements are mapped by an affine linear mapping $M$ and its determinant $det(M)$ to a reference element $K_{ref}$ with the local nodal basis functions $N$:

$$\int_{K_m} \varepsilon(x) \nabla \phi_i \cdot \nabla \phi_j dx = \int_{K_{ref}} \varepsilon(x) M^{-T} \nabla N_j \cdot M^{-T} \nabla N_i dx \cdot |det(M)|. \quad (2.19)$$

On the reference element a quadrature rule can be applied to evaluate the integral numerically.

### 2.1.3 Stokes System

In this section we progress from scalar differential equations to a system of differential equations. Our motivation are models for the simulation of convection in the Earth’s mantle. These are derived from the basic equations for conservation of energy, mass and the balance of forces. For mass conservation and the balance of forces, the Stokes equations are used since the diffusive forces are much larger than the convective forces. An energy equation
might be coupled through the temperature $T$ into the Stokes equations via a buoyancy term $\rho(T)g$ on the right hand side

$$\nabla \cdot (2\mu s(u)) - \nabla p = \rho(T)g,$$  \hspace{1cm} (2.20)

$$\nabla \cdot (\rho u) = 0.$$  \hspace{1cm} (2.21)

In the equations, $u$ denotes the three velocity components, $p$ the pressure, $\mu$ the dynamic viscosity, $s(u)$ the strain rate tensor, $g$ the gravity vector, and $\rho$ the density of the fluid. In geophysics applications, the viscosity $\mu$ has variations of several orders of magnitude in radial direction.

In case of a Newtonian fluid, the second order elliptic operator has the form

$$\nabla \cdot (2\mu s(u)) = \nabla \cdot \mu (\nabla u + (\nabla u)^T) - \frac{2}{3} \nabla (\mu \nabla \cdot u).$$  \hspace{1cm} (2.22)

For a constant viscosity, the operator further reduces to the vector Laplacian.

A weak formulation of the Stokes problem for the FE framework for an incompressible flow yields

$$(s(\phi^u_i), 2\mu s(u_h)) - (\nabla \cdot \phi^u_i, p_h) = (\phi^u_i, \rho(T)g),$$  \hspace{1cm} (2.23)

$$(\phi^p_i, \nabla \cdot u_h) = 0.$$  \hspace{1cm} (2.24)

Different FE spaces for $\phi^u_i, \phi^p_i$ may be chosen for velocity and pressure leading to a mixed FE discretization.

The system can be written in matrix notation with a zero diagonal block as a linear equation system

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

with the matrices

$$A_{ij} = (s(\phi^u_i), 2\mu s(\phi^u_j)),$$  \hspace{1cm} (2.25)

$$B_{ij} = - (\phi^u_i, \nabla \cdot (\phi^u_j)),$$  \hspace{1cm} (2.26)

$$F_i = (\phi^u_i, \rho(T)g).$$  \hspace{1cm} (2.27)

One way to solve the Stokes system is the reformulation of the system by using the Schur complement $-BA^{-1}B^T$ to

$$\begin{pmatrix} A & B^T \\ 0 & -BA^{-1}B^T \end{pmatrix} \begin{pmatrix} U \\ P \end{pmatrix} = \begin{pmatrix} BA^{-1}F \\ 0 \end{pmatrix}.$$  \hspace{1cm} (2.28)

On this system, a pressure correction approach [35] can be performed on the decoupled variables, as we discuss in section 6.2. The pressure correction scheme is used in TERRA, CitComS, and other mantle convection codes [36].
2.2 Multigrid

In order to solve large linear systems of equations, as they are constructed via FE, Multigrid methods [37, 38] are attractive because they achieve an asymptotically optimal complexity of $O(N)$ [39, 40, 41].

Algebraic multigrid (AMG) methods [42, 43, 44, 40] inherently support unstructured meshes by construction. BoomerAMG from the Hypre package is a popular implementation of AMG [45]. In contrast, geometric multigrid (MG), relies on a given hierarchy of nested meshes. Geometric MG may achieve a significantly higher performance than AMG in terms of unknowns computed per second, since e.g. the coarse grid meshes are known a-priori.

In this thesis, we will restrict ourselves to geometric multigrid methods. They split up the frequencies of the error by choosing different subspaces. Usually, additional subspaces to the finest grid are constructed by defining coarser grids. Basically it is the other way around within the HHG approach: The input mesh defines the original space, and additional spaces are generated by the grid refinement.

While high-error frequencies can easily be damped by smoothing on a certain grid level, low-frequency errors are reduced by corrections interpolated from the coarser grids. The idea of the coarse grid correction is that frequencies of discrete functions depend on the mesh size. Thus the whole spectrum of frequencies can be captured by transferring the algebraic error to different coarse meshes and eliminating its high-frequency modes by smoothing respectively.

2.2.1 Multigrid Schemes

This section presents an overview, how the coarse grid correction can be achieved, while the next section presents different traversal strategies.

Correction Scheme

In the case of a linear operator $A$, direct access to the error is given by the residual equation on a coarser mesh size $H$

$$A_H e_H = r_H = f_H - A_H u_H.$$ (2.29)

The MG cycles in the next sub-section show how the residual equation is incorporated into a correction scheme (CS).
Full Approximation Scheme

In the non-linear case, one can locally or globally linearize the equation, or apply the MG full approximation scheme (FAS). This section briefly motivates a FAS. Although one technically works with a coarse grid correction, full approximations to the discrete solutions instead of errors are stored on the coarser grids.

The truncation error $\tau_h(u^{ex})$ gives a motivation for the FAS scheme. For the discrete problem of equation (2.9), $\tau_h(u^{ex})$ is given by the residual with injected exact solution $u^{ex}$

$$
\tau_h(u^{ex}) = A_h R^{inj} u^{ex} - f_h = A_h R^{inj} u^{ex} - R a(u^{ex}, v),
$$

with injection $R^{inj}$, and restriction $R$ as transfer operators.

In the FAS, the coarse grid problem discretized with mesh size $H$ can be re-written as

$$
A_H u_H = f_H + \tau^H_h(u_h)
$$

with the correction term

$$
\tau^H_h(u_h) = A_H P^{inj} u_h - P A_h u_h.
$$

The correction term $\tau^H_h(u_h)$ is an estimation of the discretization error based on the fine grid solution. After an approximation of the solution by equation (2.31), $u_H$ is an improved approximation for the finer grid $h$. For a linear operator $A$ both MG schemes are equivalent. A detailed explanation of the algorithm is given in e.g. Trottenberg et al. [40].

2.2.2 Multigrid Cycles

Traversing the grids in different manner leads to different cycles which can be described as follows. Figure 2.4 depicts an overview of different MG cycles within a grid hierarchy of four levels.

Two-grid cycle The simplest variant of a MG method is the two-grid cycle (see algorithm 1). This cycle does not lead to a MG method with optimal complexity for a direct or Conjugate Gradient (CG) type coarse grid solver. However, the two-grid cycle is often used for analysis or testing purposes.

V-cycle The V-cycle is a recursive extension of the two-grid cycle. Here, line 6 of algorithm 1 is only done on the coarsest grid. On all other grids the coarser system is solved by calling $e_H = twoGrid(0,A_H,r_H,v_1,v_2)$ instead.
Algorithm 1 Two-grid cycle: $u_h^{(k+1)} = \text{twoGrid}_h(u_h^{(k)} ,A_h,f_h,\nu_1,\nu_2)$

1: // Presmoothing
2: $\bar{u}_h^{(k)} = S_{h}^{\nu_1}(u_h^{(k)} ,A_h,f_h)$
3: // Compute residual
4: $r_h = f_h - A_h \bar{u}_h^{(k)}$
5: // Restrict residual
6: $r_H = R r_h$
7: Solve $A_H e_H = r_H$ by a (parallel) direct or iterative solver
8: // Prolongate error and do coarse grid correction
9: $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + Pe_H$
10: // Postsmoothing
11: $u_h^{(k+1)} = S_{h}^{\nu_2}(\tilde{u}_h^{(k)} ,A_h,f_h)$

Figure 2.4: Different MG cycles within a grid hierarchy of four levels.
A W-cycle differs from a V-cycle by a second recursive call directly after the first call. This leads to a more accurate coarse grid correction, which can lead to a stabilization of the MG method for difficult problems.

During a F-cycle [40] one first steps down the MG hierarchy until one reaches the coarsest grid. Then, one performs an additional V-cycle on each grid level when moving upwards the MG hierarchy again.

2.2.3 Transfer Operators

During the MG algorithm, functions are transferred by interpolation and restriction between the grid levels. Often one defines only the interpolation, and then the restriction is defined by the transposed of the interpolation ($R = P^T$) or vice versa. In the following, some basic operators are described for triangular grids. The corresponding node numbering is given in figure 2.5. The extension to tetrahedral elements is straightforward, since by construction of the coarse grids fine grid point are also only located on coarse grid points, or on edges between two coarse grid points.
Injection

The injection $R^{inj}$ is a restriction that directly maps the fine grid points to coarse grid points without taking the neighboring fine grid points into account.

$$R^{inj} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0
\end{bmatrix}$$

(2.33)

Linear Interpolation

We define the linear restriction $R^{lin}$ by

$$R^{lin} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0
\end{bmatrix},$$

(2.34)

and the corresponding linear interpolation $P^{lin}$ as its transposed.

2.2.4 Coarse Grid Operators

Direct Coarsening

Direct coarsening defines its coarse grid operators by applying a direct discretization of the PDE on the coarser grids. Advantages are that this method is usually very cheap and the discrete operators naturally have the same size. On the downside, one needs a geometric representation of the problem and features on the finest grid are possibly not captured on the coarser grids.

Galerkin Coarsening

Galerkin coarsening is an algebraic way to define the coarse grid operator:

$$A_H = RA_hP.$$  

(2.35)

When using injection as restriction and linear interpolation the Galerkin operator is defined

$$R^{inj} a_{ij} P^{lin} = \begin{bmatrix}
a_{11} + \frac{1}{2}a_{15} + \frac{1}{2}a_{16} & \frac{1}{2}a_{16} & \frac{1}{2}a_{15} \\
\frac{1}{2}a_{26} & a_{22} + \frac{1}{2}(a_{24} + a_{26}) & \frac{1}{2}a_{24} \\
\frac{1}{2}a_{35} & \frac{1}{2}a_{34} & \frac{1}{2}a_{35} + \frac{1}{2}(a_{34} + a_{35})
\end{bmatrix}.$$  

(2.36)
For linear interpolation and linear restriction, the first row of the Galerkin operator sums up to

\[
R^{lin} a_{1j} P^{lin} = \begin{pmatrix}
    a_{11} + \frac{1}{2}(a_{51} + a_{61}) + \frac{1}{4}(a_{15} + a_{16} + a_{55} + a_{56} + a_{66}) \\
    \frac{1}{4}(a_{16} + a_{62}) + \frac{1}{4}(a_{54} + a_{55} + a_{64} + a_{66}) \\
    \frac{1}{4}(a_{15} + a_{53}) + \frac{1}{4}(a_{54} + a_{55} + a_{64} + a_{66})
\end{pmatrix}^T.
\] (2.37)

The connectivity is the same for both coarse grid points which are represented by the two other rows.

### 2.2.5 Parallel Multigrid

Within this section two main ingredients of a parallel MG algorithm are motivated: a typical ghost layer exchange within the HHG approach, and one issue arising through the coarsest MG-levels.

**Ghost Layer Exchanges**

B. Bergen implemented a ghost layer exchange for the resulting block-structured HHG grid hierarchy [19]. One parallel MG cycle (see algorithm 1) has to communicate data at the ghost points during smoothing, residual computation, and prolongation on each grid level.

For example, the parallel smoothing procedure works as follows:

1. Smooth the inner vertex points.
2. Update and communicate the vertex ghost points of the edges, faces, and volumes.
3. Smooth the inner edge points.
4. Update and communicate the edge ghost points of the vertices, faces, and volumes.
5. Smooth the inner face points.
6. Update and communicate the face ghost points of the vertices, edges, and volumes.
7. Smooth the inner volume points.
8. Update and communicate the volume ghost points of the vertices, edges and faces.

A detailed explanation of each single step is given in the dissertation of B. Bergen [22]. By the sequential smoothing of the different types of macro-elements we introduce an implicit coloring. The processing order of the points and the ghost layer exchange works similar also within residual computation and prolongation.
Coarsest Grid Problem

From the theoretical point of view, MG has a linear complexity with respect to the number of unknowns for sparse systems. For serial runs, this observation can also be made in practical use. In parallel settings we are able to achieve this behavior up to a certain degree that seems to be limited by the increasing amount of communication cost for the coarsest levels. Estimates for the decreasing volume to surface ratio for MG hierarchy are given in the work of Hülsemann et al. [46].

As an example, let us assume we want to utilize 100000 cores, having one process per core. With static grid partitioning the coarsest grid would consist of at least 100000 elements. For a regular tetrahedral grid it is even too drastic to end up with one element per process, like in our example. This would lead to up to four local unknowns per process and at least 44 ghost points in three dimensions. HHG refines each input element twice to generate the coarsest MG level such that the minimum size per process is 64 elements (up to 35 local unknowns) which provide a reasonable volume to surface ratio.

In general, there are two possibilities to treat this grid: Proceed with the construction of new coarser grid levels by collocating elements on fewer number of cores (agglomeration) [47], or stop at this stage and apply any iterative or direct solver [48].

Since for many problems the very coarse levels do not improve the convergence so much, and due to a simple implementation, we decided not to coarsen any further. This strategy is referred as flat MG, truncated cycle, or U-cycle. Otherwise, an AMG would be necessary since the coarsest grid is unstructured. On the remaining finer grids, we can perform a truncated MG cycle. Here, the Poisson problem remains a hard test case for MG on parallel systems using such truncations [40]. We do not directly solve the coarsest level but apply an iterative scheme to get the error below a specified bound.
Part I

Convergence Rate Optimization by Local Fourier Analysis
3 Local Fourier Analysis on Structured Grids

Local Fourier analysis (LFA) [37] is a very useful tool to predict the asymptotic convergence factors of MG methods with high accuracy quantitatively. Therefore it is widely used to design efficient MG algorithms. In the LFA, an infinite regular grid is considered and boundary conditions are ignored. On an infinite grid, the discrete solution and the corresponding error can be represented by linear combinations of certain complex exponential functions, the Fourier modes. Fourier modes form a unitary basis of the space of grid functions with bounded $l_2$–norm.

The LFA monograph of Wienands and Joppich [49] provides an excellent background for experimenting with Fourier analysis. Over the last years, the LFA was extended to e.g.:

- hexagonal meshes [50],
- MG as a preconditioner [51],
- optimal control problems [52],
- discontinuous Galerkin discretizations [53],
- MG for the FE discretization of the two-dimensional curl-curl equation on a quadrilateral grid [54],
- and generalized to triangular grids [55].

The key of the last mentioned generalization is to express the Fourier transform in a special coordinate system. It is possible to perform an LFA also on 3-dimensional tetrahedral grids such that we can use it to design efficient MG solvers in the HHG framework. A novel partitioning of the Fourier space is proposed to analyze the four-color smoother in [1]. In this work, the four-color smoother is defined for which smoothing and two-grid analysis are performed. An important aspect is to show the decomposition of the Fourier space into minimal invariant subspaces. These are four-dimensional for the smoothing analysis and have sixteen dimensions for the two-grid analysis. The following part is mainly published in [1].

This chapter is devoted to show some LFA results in order to design efficient MG methods for different shapes of tetrahedra on the Poisson’s equation (2.11). We mainly focus on the
choice of good smoothers for poorly shaped tetrahedra which are inevitably produced by
commonly used mesh generators.

Recall, in an MG algorithm, the smoother has the task to reduce high-frequency error
components, while the coarse-grid correction reduces low-frequency components. Both,
smoother and coarse-grid correction can be tuned to optimally suit the differential equation
and its discretization [56, 57]. Besides choosing the smoother type, two more parameters
can be changed to achieve more efficient smoothing behavior: the under-/over-relaxation
parameter (ω) and the number of pre-/post-smoothing steps per MG cycle (ν1, ν2). We use
the notation V(ν1, ν2) in order to additionally specify the MG V-cycle. All parameters can
even be adapted to the problem locally, if the numerical properties differ strongly across the
domain. Next, we introduce how multi-coloring and block-wise smoothers can be created
for our tetrahedral refinement.

The Gauss-Seidel (GS) algorithm often is a more efficient smoother than the Jacobi algo-
rithm but due to its data dependencies it is potentially slower. The data dependencies can
be eliminated by coloring the grid points such that points of the same color are not directly
connected to each other, and thus can be updated in parallel. In a 2D rectangular grid with a
5-point discretization stencil, e.g. two colors are needed, whereas the 15-point stencil within
HHG solver requires four colors, see figure 3.1. Note that a relaxation parameter ω can be
chosen individually for each of the colors ωi = (ω1, ω2, ω3, ω4).

As we will see, four-color relaxation results in a very efficient smoother for regular tetrahe-
dral grids. However, when poorly shaped tetrahedra occur, then point-wise smoothers are
not efficient anymore. In this case we will consider block-wise smoothers. In particular,
line- and plane-wise smoothers will be used. Note that for Bey refinement, seven different
line smoothers can be defined that correspond with the seven directions that appear in the
connections between each pair of stencil entries, see figure 3.1. Similarly, seven different
plane smoothers can be considered. Four faces have the orientation of the un-refined tetra-
hedron face. Three other face orientations can be spanned up by two vectors connecting two
opposing edges.

3.1 Setup and Assumptions

In order to compare the convergence factors predicted by LFA, we use a W-cycle since a two-
grid cycle requires a high computational effort on the coarse grid in practice. However, the
convergence factors of a W-cycle are very similar to a two-grid cycle. The LFA assumes an
infinitely large regular domain. In practice, such a domain can be approximated by a regular
grid with a very large number of unknowns. In experiments, structured meshes of 2.1 · 10^6
elements (129 grid points in each direction) easily turned out to be sufficient to exhibit the
asymptotic behaviour. Similar number of grid points in each directions are necessary in
2D. In the experiment, the solution is initialized with random values between −1.0 and
Figure 3.1: Coloring of a 15-point stencil. Four different colors (and shapes accordingly) are used to break all dependencies between each connected pair of vertices. Planes between the elements are introduced to improve readability. Exemplary two tetrahedra are shown with thicker edges [1].

The errors for determining the convergence rates are measured with the discrete L2 norm, using the observed rate of error reduction in the 151st cycle as approximation to the asymptotic value. The remaining MG components will be standard, i.e. we will use linear interpolation and its adjoint as transfer operators and will use direct coarsening for defining the coarse-grid operators.

If not stated differently, all relaxation parameters are found by LFA for the remaining part. At first we are interested to find optimal relaxation parameters w.r.t. the convergence rate. To cover the complete search space, all different relaxation parameters up to $\omega = 2.0$ are predicted with a resolution $d\omega = 0.05$ for each different color and the best is chosen. In section 4.3.3, a downhill simplex method [58] is used that turned out to be stable and additionally much more computationally efficient. Both methods were compared and achieved very similar convergence rates.

3.2 Regular Tetrahedron

We first turn our attention to the $\omega$–Jacobi, the lexicographic Gauss-Seidel and the four-color smoother for a regular tetrahedron, that is characterized by having edges of equal length. In the case of Jacobi relaxation, we use the optimal parameter $\omega = 0.8$ as predicted by LFA. For the Gauss-Seidel and 4-color smoothers no damping is applied. In table 3.1 we show for different number of pre-smoothing ($v_1$) and post-smoothing ($v_2$) steps the smoothing factor $\mu$, the two-grid convergence factor $\rho$ as predicted by LFA, and the experimentally observed convergence factor obtained by using $W$–cycles. Considering Fourier
smoothing analysis, the smoothing factor describes the influence of a smoothing operator to the high-frequency error components [49]. These components are important to be smoothed out in a multigrid algorithm, since lower frequencies are reduced by coarse grid correction. Clearly the two-grid convergence factors are very well predicted by LFA in all cases. We also observe that the four-color smoother provides the best convergence factors among the considered point-wise smoothers.

It should be noted that the smoothing factors on tetrahedra are somewhat worse than one would expect for triangles in 2D. For example, the smoothing factor of Gauss-Seidel on an equilateral triangle is 0.416 in 2D, while it is 0.521 for a regular tetrahedron in 3D. This behaviour is similar to what is observed for standard finite difference discretizations of the 3D Poisson equation (see, for example [40]).

In passing, note that it is possible to improve the convergence factors of the four-color smoother by using relaxation parameters. For instance, in the case of \((\nu_1, \nu_2) = (2, 1)\), the convergence factor for four-color smoother can be improved from 0.153 to 0.090 by using overrelaxation parameters \(\omega_i = 1.15, 1.20, 1.25, 1.25\). Similarly for lexicographic Gauss-Seidel, the convergence factor can be reduced from 0.176 to 0.141 with the relaxation parameter \(\omega = 1.2\).

### 3.3 Optimized Tetrahedron

A regular input tetrahedron is not the best shape for our regular refinement with respect to the convergence rate. Inside a structured region of a refined coarse grid element, one pair of stencil entries relates to an edge of the element. After one refinement step, four child elements are similar to the parent element, only translated and scaled by the factor 0.5. The remaining four child elements contain five edges of its parent element, which are translated and scaled by the factor 0.5. The remaining sixth edge is introduced by Bey’s refinement [27]. This edge is located between the midpoint of edge \(e_{02}\) and the midpoint of edge \(e_{13}\) (see figure 3.2) of the parent element. The subscripts denote the vertex indices of the refined element. In the case of a regular tetrahedron, all edges except of the additionally introduced edge have the same length. In contrast to the six other entry pairs, which are typically negative, the seventh pair has positive entries in the stencil. Thus, compared to the other edges there is a weak connection, if this edge is short. For example by setting the edge lengths of \(e_{02}\) and \(e_{13}\) equal to 1.2 and keeping all other four edge sizes equal to 1.0, we receive an optimized tetrahedron concerning the convergence rate (see figure 3.3).

The exceptional convergence factors associated with the four-color smoother for this optimized tetrahedron are pointed out in table 3.2. On this type of tetrahedra, the convergence factor of the four-color smoother is approximately \(\rho = 0.106\) for one pre- and post-smoothing steps, having at the same time the desirable property of high parallelism. This
Figure 3.2: Two times refined regular tetrahedral grid [1].

Figure 3.3: Optimized tetrahedron with two non-unit edge lengths [1].
Damped Jacobi Gauss-Seidel Four-color

<table>
<thead>
<tr>
<th>( \nu_1, \nu_2 )</th>
<th>( \mu^{\nu_1+\nu_2} )</th>
<th>( \rho )</th>
<th>( \rho_h )</th>
<th>( \mu^{\nu_1+\nu_2} )</th>
<th>( \rho )</th>
<th>( \rho_h )</th>
<th>( \mu^{\nu_1+\nu_2} )</th>
<th>( \rho )</th>
<th>( \rho_h )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 0</td>
<td>0.741</td>
<td>0.640</td>
<td>0.637</td>
<td>0.521</td>
<td>0.434</td>
<td>0.427</td>
<td>0.500</td>
<td>0.407</td>
<td>0.389</td>
</tr>
<tr>
<td>1, 1</td>
<td>0.550</td>
<td>0.409</td>
<td>0.407</td>
<td>0.272</td>
<td>0.223</td>
<td>0.219</td>
<td>0.250</td>
<td>0.195</td>
<td>0.197</td>
</tr>
<tr>
<td>2, 1</td>
<td>0.406</td>
<td>0.298</td>
<td>0.297</td>
<td>0.141</td>
<td>0.176</td>
<td>0.174</td>
<td>0.125</td>
<td>0.153</td>
<td>0.153</td>
</tr>
<tr>
<td>2, 2</td>
<td>0.301</td>
<td>0.250</td>
<td>0.250</td>
<td>0.074</td>
<td>0.143</td>
<td>0.141</td>
<td>0.062</td>
<td>0.123</td>
<td>0.125</td>
</tr>
</tbody>
</table>

Table 3.1: LFA smoothing factors, \( \mu^{\nu_1+\nu_2} \), LFA two–grid convergence factors, \( \rho \), and measured \( W \)–cycle convergence rates, \( \rho_h \), for a regular tetrahedron [1]

<table>
<thead>
<tr>
<th>( \nu_1, \nu_2 )</th>
<th>( \mu^{\nu_1+\nu_2} )</th>
<th>( \rho )</th>
<th>( \rho_h )</th>
<th>( \mu^{\nu_1+\nu_2} )</th>
<th>( \rho )</th>
<th>( \rho_h )</th>
<th>( \mu^{\nu_1+\nu_2} )</th>
<th>( \rho )</th>
<th>( \rho_h )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 0</td>
<td>0.720</td>
<td>0.602</td>
<td>0.598</td>
<td>0.492</td>
<td>0.401</td>
<td>0.392</td>
<td>0.442</td>
<td>0.345</td>
<td>0.331</td>
</tr>
<tr>
<td>1, 1</td>
<td>0.517</td>
<td>0.362</td>
<td>0.360</td>
<td>0.243</td>
<td>0.151</td>
<td>0.145</td>
<td>0.196</td>
<td>0.106</td>
<td>0.105</td>
</tr>
</tbody>
</table>

Table 3.2: LFA smoothing factors, \( \mu^{\nu_1+\nu_2} \), LFA two–grid convergence factors, \( \rho \), and measured \( W \)–cycle convergence rates, \( \rho_h \), for an optimized tetrahedron [1]

factor can also be further enhanced to 0.080 using damping parameters of values \( \omega = (1,1,1,1.2) \).

### 3.4 Degenerated Tetrahedra

On the other hand, the highly satisfactory convergence factors obtained for regular and optimized tetrahedra, can deteriorate quickly when poorly shaped tetrahedra are treated. In two dimensions, there are only two types of failure, angles close to 0\(^0\) and angles close to 180\(^0\), and no failures of the first kind implies no failures of the second. In three dimensions, we can classify poorly shaped tetrahedra according to both dihedral and solid angles. Dihedral is the interior angle between two face planes, while solid angles are formed by the intersection of three planes at one vertex. There are then six types of degenerated tetrahedra, as shown in figure 3.4.

Table 3.3 shows the characteristics of differently shaped tetrahedra. Besides the number of solid and dihedral angles, the edge ratio, minimal and maximal angles between two edges are provided for an exemplary tetrahedron of this type. For these elements, the smoothing factor and the two-grid convergence factor are given for one pre- and one post-smoothing step (\( \nu_1 = 1, \nu_2 = 1 \)). Poor behavior of the four-color smoother for these type of tetrahedra is shown together with the good performance that is observed for the regular and optimized tetrahedra.
Figure 3.4: Different classes of degenerated elements [1].

<table>
<thead>
<tr>
<th></th>
<th>Regular</th>
<th>Optimized</th>
<th>Needle</th>
<th>Wedge</th>
<th>Spindle</th>
<th>Spade</th>
<th>Sliver</th>
<th>Cap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small solids</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Large solids</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Small dihedrals</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Large dihedrals</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$\mu^{\nu_1+\nu_2}$</td>
<td>0.250</td>
<td>0.196</td>
<td>0.982</td>
<td>0.786</td>
<td>0.980</td>
<td>0.590</td>
<td>0.896</td>
<td>0.872</td>
</tr>
<tr>
<td>$\rho$</td>
<td>0.195</td>
<td>0.106</td>
<td>0.982</td>
<td>0.740</td>
<td>0.980</td>
<td>0.500</td>
<td>0.889</td>
<td>0.739</td>
</tr>
<tr>
<td>Edge ratio</td>
<td>1.0</td>
<td>1.15</td>
<td>10.0</td>
<td>4.0</td>
<td>10.0</td>
<td>1.67</td>
<td>1.40</td>
<td>1.71</td>
</tr>
<tr>
<td>Min angle</td>
<td>60$^0$</td>
<td>54.9$^0$</td>
<td>5.7$^0$</td>
<td>14.3$^0$</td>
<td>5.7$^0$</td>
<td>33.6$^0$</td>
<td>45.6$^0$</td>
<td>31.4$^0$</td>
</tr>
<tr>
<td>Max angle</td>
<td>60$^0$</td>
<td>70.2$^0$</td>
<td>87.1$^0$</td>
<td>82.8$^0$</td>
<td>87.1$^0$</td>
<td>112.9$^0$</td>
<td>88.9$^0$</td>
<td>117.2$^0$</td>
</tr>
</tbody>
</table>

Table 3.3: LFA smoothing factors, $\mu^{\nu_1+\nu_2}$, and two–grid convergence factors, $\rho$, of the four-color smoother for different tetrahedra [1]
To overcome this difficulty, we propose specific smoothers for some of the poorly shaped tetrahedra. The first type of such tetrahedra is the wedge type, which has one edge much smaller than the others. This causes two entries of the stencil to be much larger than the others, producing the strongest connection in one direction. Since the common lore states that the good smoothing of errors is obtained when the strongly coupled unknowns are collectively relaxed, a line smoothing in the direction of the smaller edge is expected to improve the performance. In fact, a two-grid convergence factor $0.122$ is obtained by using the corresponding line smoother with one pre- and one post-smoothing steps. For very anisotropic wedge-type tetrahedra improved convergence rate easily compensated the somewhat higher computational cost of the tridiagonal solvers in the line smoother. Table 3.4 presents an overview of the effect of line-wise and plane-wise smoothers onto the convergence rate for this and the following tetrahedral shapes.

### Table 3.4: Smoothing properties of line-wise and plane-wise smoothers for different tetrahedra [1]

<table>
<thead>
<tr>
<th></th>
<th>Wedge</th>
<th>Spade</th>
<th>Needle</th>
<th>Spindle</th>
<th>Sliver/Cap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line-wise</td>
<td>good</td>
<td>middle</td>
<td>poor</td>
<td>poor</td>
<td>poor</td>
</tr>
<tr>
<td>Plane-wise</td>
<td>good</td>
<td>good</td>
<td>good</td>
<td>good</td>
<td>poor</td>
</tr>
</tbody>
</table>

#### 3.4.1 Wedge

Another tetrahedral shape is generated by assuming two short edges, which are connected to each other. Keeping the other edge lengths similar but larger results in a spade. In comparison to the needle-type, discussed next, only two stencil-pairs of three in a plane are large. Already an improvement is visible, choosing a line-wise smoothing in the direction of one short edge. With two smoothing steps in total, the computational work is comparable to point-wise smoothing, but results in a better convergence rate of $0.23$. Hereby it yields to similar convergence, regardless of smoothing alternatively in both directions of the short edges, or repeatedly along one of them. Point-wise smoothing only provides a convergence rate of $0.59$ (see table 3.3).

Another possibility is applying a zebra-plane smoothing about the faces spanned by those edges. With only one smoothing step, a plane-zebra smoother yields convergence factors about $0.105$. Whether or not this reduces the overall cost depends on the implementation of the plane-smoother and the degree of anisotropy. Indeed, because of significant higher costs of plane-wise smoothing in our current implementation, discussed in the next section, we would choose line-wise smoothing for that example.
3.4.3 Needle

The next degenerated tetrahedron is the needle-type, which is characterized by the fact that one vertex is far away from the opposing face. This feature produces a stencil with stronger connections in the direction of the plane defined by the face opposite to such vertex. Therefore, the corresponding plane relaxation will result in excellent smoothing. In fact, a very good convergence factor about 0.121 is reached with only one smoothing step for a zebra-plane smoother. We want to emphasize the fact of the use of zebra-plane smoothers rather than lexicographic plane relaxation. This is due to the significant improvement of the convergence factors when zebra-type smoothers are used. For example, in this case a factor of 0.330 is provided by lexicographic plane relaxation. Thus one plane-zebra sweep is roughly as effective as two lexicographic plane sweeps ($(0.330)^2 = 0.1089$). One might think that alternating line relaxation with respect to the two directions defining the corresponding face could be a suitable choice, but very poor convergence factors are provided by this strategy. For instance, only a factor of about 0.942 is obtained when using two smoothing steps. We remark that all strongly connected unknowns should be relaxed collectively and therefore a plane smoother must be used.

3.4.4 Spindle

Analogously, the behavior of the spindle-type is similar to that of needle type. In this case the strong connection also corresponds to the direction of a face of the tetrahedron, and therefore a plane smoothing will be a good choice. Again, very satisfactory convergence properties are provided by using such smoother. LFA predicts a two-grid convergence factor about 0.124 with only one smoothing step if the adequate zebra-plane smoother is activated, that is, the zebra-plane relaxation updating all strongly connected unknowns.

3.4.5 Sliver and Cap

According to Shewchuk [59] "good" tetrahedra are of two types: those that are not flat, and those that can grow arbitrarily flat without having a large planar or dihedral angle. The "bad" tetrahedra result in error bounds that explode, and a dihedral angle or a planar angle that approaches $180^\circ$, as they are flattened. Sliver as well as cap type can grow arbitrary flat and have in addition large planar or dihedral angles and thus have bad interpolation properties.

For a different reason, we also were not able find an optimal collective relaxation method for sliver and cap type. Here, the strongly connected points are not located on an edge or plane; therefore the smallest block would already include all grid points. In principle alternating plane relaxations, or alternating line relaxations improve the convergence rate compared to
a point-wise relaxation like shown for a spade. However, similarly in this case the convergence rate deteriorates when the degree of anisotropy increases.

However, this is not a limitation from the practical point of view. Many Delaunay mesh generators include a post-processing to find and remove sliver and cap type tetrahedra, because of the poor approximation quality of the FE methods using these types.

This section showed, how the smoothing process might be improved on purely structured degenerated grids. The next step is to extend the ideas to semi-structured meshes, as they are appear in HHG. We present some approaches come along with these more general meshes in the next chapter.
4 Numerical Experiments on Semi-Structured Meshes

In this chapter we will discuss numerical experiments to illustrate two different strategies in section 4.2 and 4.3. Firstly, we will demonstrate how to use LFA to design efficient MG solvers for semi-structured tetrahedral grids. We will use the block-wise processing to choose different smoothers depending on the shape of the tetrahedra. This kind of strategy has already been used successfully for two-dimensional linear-elasticity problems in [60]. Here, we will apply it for three-dimensional tetrahedral meshes. The second strategy consists of applying LFA to improve the overall convergence rates by adapting smoothing parameters. This includes the damping parameters, number of pre- and post-smoothing steps, and smoother types individually for each block, in a complex domain.

4.1 Computational Cost for the Implementation

Especially for the more sophisticated smoothers it is important to keep their computational cost in mind. The computational costs of an algorithm is often expressed in arithmetic operations. An advantage of this measure is its platform independence. Unfortunately, it often cannot give realistic predictions of performance for an implementation on specific hardware. The runtime on many modern architectures is dictated by memory bandwidth, cache effects, instruction level parallel execution, or memory alignment issues. Here we will present an overview of the smoothers that are available in HHG. To allow a fair comparison, all algorithms are implemented in an efficient way, but are not yet optimized specifically for each architecture. The performance is evaluated on one core of a Xeon 5550 Nehalem chip with 8 MB shared cache per chip. The code was produced by the Intel 10.1 compiler (flags: -O3). We measure the required number of clock cycles per unknown for three different block sizes for a structured region: 65, 129 and 257 unknowns in each direction (see table 4.1). Most differences can be attributed to cache-effects. Applying damping parameters cost additional 3 clock cycles or less. This is not really severe for any of the schemes introduced. Thus it should be applied, if it leads to an improved convergence. A much more detailed performance analysis for a point-wise smoother will be presented in section 8.2.1.
for a PowerPC 450 quadcore processor and 8.3.1 for a Intel Xeon 5650 Westmere processor.

The compiler is basically able to utilize SSE SIMD instructions (small vector processing) for the stencil evaluations. For large vectors, a 4-color smoother with a splitting of the colors to different arrays would be necessary to achieve a good performance. Further, the data layout on the structured regions would also fit well to a parallel execution on GPUs. Here, especially smoothers with a low degree of dependency (like a 4-color smoother, zebra-line smoother, or a parallel plane-wise smoother) would be most suitable.

Especially to perform line-wise or plane-wise smoothing over interface boundaries, using a MG in one or two dimensions seems to be a promising approach. However it not always obvious to define a line or plane for the whole domain, since the coarsest mesh is unstructured.

**Line-wise Smoother**  The line-wise smoother is implemented by Thomas’ algorithm, or tridiagonal matrix algorithm, a simplified version of Gaussian elimination. For constant coefficients, terms involving only the coefficients of the stencil are calculated once per stencil. Thus the required operations per unknown are comparable to a Gauss-Seidel smoothing. The number of clock cycles is only slightly higher. A zebra-line fashion has a high degree of parallelism in two directions. A parallel execution in the line-wise direction can be achieved by cyclic reduction (see e.g. [61]).

**Plane-wise Smoother**  The advantage of an iterative method for a plane-wise smoother is, that the planes do not have to be solved exactly. A good approximation is enough to obtain an optimal convergence rate of the whole V-cycle. This behavior was observed e.g. in [62], where a 2-dimensional MG is used for each plane.

For tetrahedral structured regions, it is not straightforward to apply plane-wise MG, since each plane sizes has a different size. Therefore, we solved the plane system by a CG algorithm. Since the number of CG iterations increases with respect to a growing plane-size, also the number arithmetic operations and thus solving time are effected.

### 4.2 Collective Smoothing

In the previous section we presented the impact of strongly anisotropic meshes on the convergence rate for structured tetrahedral meshes. Furthermore we found appropriate line- and plane-smoothing strategies for different types of element degeneration. In a smoothing step of HHG, we first smooth the interface points (macro vertices, edges, and faces) between all structured regions. Afterwards we smooth the structured regions themselves. While for point-wise smoothing this proceeding has nearly no influence on the convergence [63], we
Table 4.1: Arithmetic costs per unknown and measured clock cycles for a Gauss-Seidel, four-color, line-wise and plane-wise smoothers [1]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Operations</th>
<th>Clock cycles (line sizes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>65</td>
</tr>
<tr>
<td>lex. Gauss-Seidel</td>
<td>29</td>
<td>35</td>
</tr>
<tr>
<td>Four-color</td>
<td>29</td>
<td>40</td>
</tr>
<tr>
<td>Line-wise</td>
<td>29</td>
<td>40</td>
</tr>
<tr>
<td>Plane-wise</td>
<td>270-600</td>
<td>360</td>
</tr>
</tbody>
</table>

study its influence for plane- or line-smoothing in the first numerical experiment, where we measure the impact of the interface points between the structured regions using semi-structured meshes. In the second subsection, the advantages of using different smoothers for different shaped tetrahedra composing the domain are displayed.

### 4.2.1 Anisotropic Smoothing

In the following two numerical experiments we construct our computational domain out of the wedge type elements. Figure 4.1 (a) shows a setup where six structured regions are sharing anisotropic triangular faces. In figure 4.1 (b) the same type of tetrahedra are used, but are here in contrast connected at their regular triangular face. In the following part, we apply two smoothing steps for all smoothers. The convergence factors are measured after 50 W–cycles. We point out that the used elements have a edge ratio of 10 between the shortest and the largest edge length and dihedral angles of 5.7°. Otherwise the following discussed effects are not so severe. On the fine grid we end up with $1.73 \cdot 10^7$ elements in case (a) and $3.46 \cdot 10^7$ elements in case (b).

For one single region, a lexicographic Gauss-Seidel iteration achieves a convergence factor of 0.934. This factor is nearly the same for both domains (a) and (b). In contrast a line smoother yields a convergence factor of 0.072 for a single region. For domain (a) the interfaces slightly effect the convergence rate to 0.094. However it is worse if we cannot solve the full line of unknowns along the strong coupling, but only parts of the line. This is case for domain (b) since two interfaces are breaking these lines and do not allow a fully collective line-smoothing. This explains a bad convergence factor of 0.847 for (b). Damping parameter of 1.05 inside the structured regions and 1.15 for the boundary points help a bit to decrease the convergence factor to 0.796. This problem has been reported in some papers about grid-partitioning, see for example [64]. If possible, this case should be avoided in the original partitioning of the domain.

The same behavior is observed by applying plane smoothing to adjoined needle or spindle type regions. In these cases lines of unknowns, which are not collectively updated since
they are interface points, increase the convergence rate.

### 4.2.2 Locally Adaptive Smoothing

Contrary to the previous examples, we apply different smoothers according to the geometry of the structured region. A model domain consisting of four different tetrahedral types is presented in figure 4.2. Wedge and needle elements share some of their faces with regular elements. The whole domain consists of $1.73 \cdot 10^7$ tetrahedron elements on the finest level. Table 4.2 shows some properties of the single structured regions without its interfaces to any other regions.

In order to get a similar convergence rate for the composed domain, we choose different smoothers according to the underlying mesh geometry. The regular tetrahedra are smoothed by using four-color relaxation with damping parameters $\omega = (1.15, 1.20, 1.25, 1.25)$, and two pre- and one post-smoothing steps. From section 3, we know that a convergence factor of about 0.09 is achieved for these regular tetrahedra. For the wedge element, again a value of 0.09 is predicted by LFA, if a line-smoother is used with two pre- and one-smoothing steps. By the other hand, for the needle tetrahedron a zebra-plane smoother with only one smoothing step is enough to reach a convergence factor about 0.12. Taking into account these local convergence factors and considering the worst of them, we can predict a global factor of 0.12 for the composed domain.

By performing the MG algorithm with the corresponding local components chosen above, a convergence factor of 0.17 is obtained for the whole computational domain. An additional smoothing step of the macro faces after smoothing the interior was applied. Using damping parameter at the interface points and in the structured regions, the convergence factor improves to 0.14, which is very close to the expected value of 0.12. As a reference, a standard lexicographic Gauss-Seidel with $\nu_1, \nu_2 = 2, 1$ provides a convergence factor of 0.95. It turns out that using different smoothing strategies depending on the sub-domains is very advantageous and following this rule, very efficient MG solvers can be designed for quite complicated three-dimensional domains. However to be fair, one has to consider the...
additional computational overhead for line- or plane-smoothing. If and how much it pays out can also depend much on the implementation of the smoothers.

### 4.3 Locally Adaptive Damping and Smoothing Steps

In order to achieve a good overall convergence of the solver, we try to distribute the computational work such that all regions of the domain receive a similar error reduction in each smoothing step. In our approach we use LFA to control this process. Partial smoothing has already been promoted, e.g., in [65]. In addition to that, we apply individual damping parameters to each structured region.

With HHG’s semi-structured meshes it is relatively easy to locally adapt smoothing parameters to mesh-induced convergence rate variations. Since each coarse mesh element gets refined in a regular way, all its sub-elements are similar, and the operator stencils of all interior grid points are the same. Therefore, one can determine the optimal smoothing parameters for each coarse grid element and use them for all its sub-elements. We will use LFA prediction to optimize the convergence rate of the structured regions by choosing suitable the smoother parameters. Further we will provide suggestions how to deal with interface points between those regions. Step by step we will apply different modifications to the smoothing procedure and observe its influence. Thus we end up with an asynchronous or chaotic smoothing [66, 67, 68]. However, this is usually a result of an efficient parallel implementation anyway.

As a domain for testing the optimizations we use a half ball with a small box cut out at its bottom. The diameter of the half ball is 6.0, the box has a size of $1.0 \times 2.0 \times 0.3$ (see

<table>
<thead>
<tr>
<th>Shape</th>
<th>edge ratio</th>
<th>max angle</th>
<th>min angle</th>
<th>smoothing</th>
<th>$\nu_1$, $\nu_2$</th>
<th>$\rho$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Needle</td>
<td>10</td>
<td>87°</td>
<td>5.7°</td>
<td>plane-wise</td>
<td>1.0</td>
<td>0.12</td>
</tr>
<tr>
<td>Regular</td>
<td>1</td>
<td>60°</td>
<td>60°</td>
<td>4-color</td>
<td>2.1</td>
<td>0.09</td>
</tr>
<tr>
<td>Wedge</td>
<td>5</td>
<td>110°</td>
<td>11°</td>
<td>line-wise</td>
<td>2.1</td>
<td>0.09</td>
</tr>
</tbody>
</table>

Table 4.2: Convergence factor for the single structured regions [1]
The domain is discretized with 293 elements on the coarsest mesh, generated by Gmsh [69].

These elements are refined 8 times, which leads to $1.02 \cdot 10^8$ unknowns on the finest level. A well-known degeneracy measure for FE meshes is the ratio between inscribed and circumscribed radius $\alpha$, another one is the ratio between the shortest and the longest edge $\beta$ of an element. The average degeneracy measures of our test mesh are $\alpha = 0.610$ and $\beta = 0.537$. The minimal and maximal values are $\alpha_{\text{min}} = 0.313$, $\alpha_{\text{max}} = 0.900$, $\beta_{\text{min}} = 0.214$, and $\beta_{\text{max}} = 0.830$.

### 4.3.1 Unoptimized Version and Exact Block-wise Solving

As a reference a four-color smoother is considered in the domain. For the unoptimized version $\nu = (4, 4)$ yielded the best time to solution. With this number of smoothing steps, 21 W- or V-cycles are required to reduce the algebraic error of a random initial solution below $10^{-7}$. On a single core, one V-cycle takes approx. 16 seconds. For all following experiments, we measure the convergence rate of the last cycle (see table 4.3). The single lines of the table, corresponding to different optimization strategies are explained step by step in the remaining section.

Further we impose two conditions:

1. The required amount of communication is the same for all the methods. We are exchanging eight times all ghost boundary points at the interfaces during the smoothing procedure.
Table 4.3: Measured convergence factors for different smoothing strategies [1]

<table>
<thead>
<tr>
<th>Smoothing strategy</th>
<th>W-cycle</th>
<th>V-cycle</th>
<th>$\omega$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unoptimized</td>
<td>0.64</td>
<td>0.65</td>
<td>1.0</td>
</tr>
<tr>
<td>Exact block-wise solving</td>
<td>0.11</td>
<td>0.18</td>
<td>-</td>
</tr>
<tr>
<td>Adaptive smoothing steps</td>
<td>0.51</td>
<td>0.53</td>
<td>1.0</td>
</tr>
<tr>
<td>+ Additional interface smoothing</td>
<td>0.34</td>
<td>0.34</td>
<td>1.0</td>
</tr>
<tr>
<td>Full damping</td>
<td>0.59</td>
<td>0.56</td>
<td>1.15</td>
</tr>
<tr>
<td>Interior damping</td>
<td>0.44</td>
<td>0.49</td>
<td>1.55 / 1.0</td>
</tr>
<tr>
<td>+ Additional interface smoothing</td>
<td>0.30</td>
<td>0.30</td>
<td>1.55 / 1.0</td>
</tr>
<tr>
<td>Adaptive interior damping</td>
<td>0.46</td>
<td>0.51</td>
<td>variable</td>
</tr>
<tr>
<td>+ Additional interface smoothing</td>
<td>0.31</td>
<td>0.34</td>
<td>variable</td>
</tr>
<tr>
<td>Combined Methods</td>
<td>0.15</td>
<td>0.19</td>
<td>1.55 / 1.0</td>
</tr>
</tbody>
</table>

2. The total smoothing workload has to be comparable to the unoptimized version in the solving phase. The additional time for the setup phase strongly depends on the implementation of a LFA evaluation in some of the following smoothing strategies.

The first condition provides an estimate of a tight lower limit for the global convergence rate for our experiments. For standard iterative smoothers, we can estimate this limit by solving each block exactly (see table 4.3 exact block-wise solving). However this is just to provide a comparison, since it requires much more smoothing work than we need for the unoptimized version. The interface points (macro vertices, edges and faces) between the structured regions are also considered as small blocks in themselves.

### 4.3.2 Locally Adaptive Number of Smoothing Steps

LFA predictions can help to adjust the number of pre- and post-smoothing steps according to the element. Since the communication overhead should remain the same, we adjust the smoothing procedure locally. In order to individually increase the amount of smoothing work for a structured region, we add 4-color smoother iterations during smoothing steps. During execution of those iterations, the ghost points of the element are not updated. Thus, it is like smoothing a smaller Dirichlet boundary value problem. We decrease the smoothing work for a structured region by skipping smoothing during a (global) smoothing step.

For our experiment, the overall smoothing work should stay constant, thus it is more a redistribution of smoothing work. We start by setting the number of smoothing iterations for all structured regions to one. At this stage there is one real update of the ghost layers,
but seven (three at pre- and four at post-smoothing) which would not be needed at all. Then all convergence rates are predicted by LFA and the remaining smoothing iterations are successively distributed to the element with the highest convergence rate. After each step the convergence rate of the updated element has to be recalculated. The smoothing iterations are distributed as equally as possible amongst between the updating steps of the ghost boundaries. The LFA evaluation took around 5 seconds for our mesh in the setup time. But the advantage easily outweighs the required time. In table 4.3, adaptive smoothing steps shows the effect of this optimization. For our example, figure 4.4 gives an impression of the smoothing work distribution. For most elements \( \nu_1 + \nu_2 = 4 \) steps are predicted to be sufficient, in an extreme cases even \( \nu_1 + \nu_2 = 51 \) are predicted for an upper bound of 0.18 on the convergence factor for each single structured region.

Moreover, we can add (local) smoothing iterations for the interface points, too. Since we do not save this additional work anywhere else, it is some overhead. However, it is negligible compared to its possible effects (see + additional interface smoothing in table 4.3), since there are many more inner points than interface points.
4.3.3 Locally Adaptive Damping

In a first step we want to optimize the damping parameter. A simple choice is a constant damping parameter \( \omega_{\text{const}} = \omega_1 = \omega_2 = \omega_3 = \omega_4 \) for all points (see full damping in table 4.3). By experiment \( \omega_{\text{const}} = 1.15 \) showed up to be a good choice. Damping parameters around 1.25 and more lead to divergence.

These damping parameters are low compared to LFA predicted optimal damping values in the interior of the structured regions. Indeed, the interface points cause a problem in this case. Without damping the interface points we observe improved convergence using a larger damping factor of 1.55 (see interior damping, table 4.3). Increasing the experimentally found damping parameters on the macro vertices, edges and faces to 1.1 nearly does not improve the convergence rate. Similarly to the optimization in subsection 4.3.2, additional interface smoothing is necessary to obtain the full potential of this optimization.

To further improve the damping parameters, we tried to find optimal relaxation parameters for each structured region. The four-color damping parameters are determined by applying the downhill simplex method [58] to LFA individually for each of the 293 elements on the coarsest mesh. This is computationally quite expensive. The optimal damping parameter can vary quite strongly amongst the different stencils. Inside the domain, quite low damping parameters e.g. \( \omega = (1.43, 1.26, 1.17, 1.33) \) and large damping parameters e.g. \( \omega = (1.75, 1.47, 1.46, 1.83) \) or \( \omega = (1.51, 2.00, 1.44, 1.46) \) are applied. However it did not improve the time to solution (see adaptive interior damping in table 4.3). Exactly the same number of V-cycles like using the global damping parameter are required. The convergence factors are even a bit worse than using interior damping. LFA predicts for some structured regions convergence factors down to 0.03, while worst convergence factors are up to 0.51 and 0.36 with and without damping, respectively.

Summarizing, the interface points benefit from additional smoothing. Further, the interior points can be damped more extremely, as long as damping is avoided on the interface points. It was not favorable to choose different damping parameter for the interior points.

4.3.4 Combined Methods

In this section, we combine the various methods. As we will see, the positive effects w.r.t. the convergence factor will be additive to some extent. We chose the best variant of each previous subsection 4.3.3 and 4.3.2. Thus, the combined methods of table 4.3 include adaptive smoothing steps, interior damping, and additional interface smoothing. For this example the convergence rate is even similar to an exact block-wise solving.
Part II

Framework Extensions
5 Variable Coefficients

According to B. Bergen, the HHG approach at the time has three limitations [22], namely that the

• input grids must have patch-wise constant material parameters,
• refinement of the input elements leads to child elements of equal volumes,
• finite element discretization uses only piece-wise linear basis functions, and
• HHG framework uses regular refinement.

The first two points are motivated as follows: "Both of these restrictions are necessary to ensure that the resulting grid hierarchy can be discretized using constant-coefficient stencils. As stated previously, this is an important factor in allowing us to solve extremely large problems, because it reduces the amount of memory needed to represent the global discretization matrix. Completely easing these restrictions to include grids with continuously variable coefficients and arbitrary hexahedral and prismatic elements would mean that the HHG data structures would require substantially more memory, although still less than a CRS format. A possible compromise would be to allow certain elements in the input grid to have continuously variable coefficients while keeping the rest constant."

This chapter will show how it is possible to relax the first limitation and retain a matrix-free implementation. Extensions to support PDEs with variable coefficients (equation (2.9)) instead of constant coefficients (equation (2.11)) are provided. However, this requires to introduce an element-wise view before the stencil-based operator evaluation. For linear simplex element types, like tetrahedral FE, $\nabla N_i$ and $M$ of equation (5.1) are constant terms. Therefore, the local stiffness matrices without the variable coefficients $\varepsilon(x)$ can still be computed exactly and stored.

$$\int_{K_{\text{ref}}} \varepsilon(x)M^{-T}\nabla N_j \cdot M^{-T}\nabla N_i dx |\text{det}(M)| = M^{-T}\nabla N_j \cdot M^{-T}\nabla N_i \int_{K_{\text{ref}}} \varepsilon(x) dx |\text{det}(M)|$$

(5.1)

We approximate only the integral $\int_{K_{\text{ref}}} \varepsilon(x) dx$ numerically in each operator evaluation. For linear elements, one evaluation at the barycentric coordinate $(\frac{1}{4}, \frac{1}{4}, \frac{1}{4})$ with weight $\frac{1}{6}$, or a sum over the four nodes with the weights $\frac{1}{24}$ are sufficient [70]. Since tetrahedral macro-elements lead to complicated element-wise data structures in case of a non-redundant storage, we decided to use a node-wise storage. In contrast to constant coefficients, variable coefficients require the stencil to be assembled for every grid point. For non-primitive element types,
or when $\varepsilon$ is extended to a tensor, a possible implementation approach might be to store the vectors $M^{-T} \nabla N_j$ instead.

### 5.1 Implementation

Our integration rule of the coefficient $\varepsilon$ requires three additions per element, and consequently 72 additions per stencil in a straight-forward implementation. To reduce the computational work, we apply a local (i.e. stencil-wise) common subexpression elimination: First, two nodal coefficient values on the red edges are averaged, then the coefficients of each element are calculated by averaging the corresponding two opposing red edges. Figure 5.1 illustrates the 16 used edges. By this optimization, the number of operations for the averaging reduces from 72 to 40 floating point operations.

The averaged coefficients are multiplied by the pre-computed local stiffness matrices $\text{stiff}_{k,j,i}$ for all element groups (types) $k$ and positions $j$. Details of the exploited regularities leading to this assembling procedure are presented in [22]. Algorithm (2) gives an idea of the arithmetic procedure for one grid point with the variable coefficient $k$, and the unknown vector $u$ for one element. A complete smoothing procedure can be found in appendix C.1 of this work. The integration weights $\frac{1}{24}$ is contained in the local stiffness matrices in the implementation.
Algorithm 2 Stencil assembling of one element for one grid point $x_1$. See figure 5.1 for illustration.

1: $edge_a = k(x_1) + k(x_2)$
2: $edge_b = k(x_4) + k(x_8)$
3: ... accordingly for 14 additional edges
4: $element_{(1)} = \frac{1}{24}(edge_a + edge_b)$
5: $stencil_{(1)} = stencil_{(1)} + element_{(1)} \cdot \text{stiff}_{(2,3,1)}$
6: $stencil_{(2)} = stencil_{(2)} + element_{(1)} \cdot \text{stiff}_{(2,3,2)}$
7: $stencil_{(4)} = stencil_{(4)} + element_{(1)} \cdot \text{stiff}_{(2,3,3)}$
8: $stencil_{(8)} = stencil_{(8)} + element_{(1)} \cdot \text{stiff}_{(2,3,4)}$
9: ... accordingly for 23 additional elements

The vertex, edge, and face primitives are more technical and require indirect addressing, since they are basically unstructured or at least are composed of rotated stencil parts. Next, two applications motivate the implemented extension.

5.2 $\tau$-Extrapolation

The $\tau$-extrapolation (see [38, 42]) is a modification of the MG method to improve the convergence order of the solution by an extrapolation process. In contrast to a classical Richardson extrapolation, the extrapolation works not explicitly on the solution, but implicitly on the residual [71]. However, both methods have in common that they are motivated by an expansion of the truncation error. In two dimensions, the $\tau$-extrapolation was applied to the finite element framework [72] and to variable coefficient problems [73]. Investigations for one-dimensional linear and non-linear problems and for the Navier-Stokes equations are presented in [74]. In this work, the $\tau$-extrapolation was implemented and parallelized in the current version of HHG.

In the following, we describe a numerical example computation originating from a molecular dynamics simulation. One essential step is the solution of the potential equation that reduces to Poisson’s equation in the case of a homogeneous dielectricity coefficient [75, 76]. However, an ionic solvent with varying dielectricity leads to variable coefficient problems. We use this application to evaluate and discuss the convergence features of our method. Since our goal is to study the approximation order of our discretization, we choose a test problem that is complex enough to exhibit many of the relevant characteristics of the actual application. However, it is constructed such that there exists an analytical solution.
Table 5.1: Error and order of consistency with and without \( \tau \)-extrapolation for a variable coefficient problem.

### 5.2.1 Smooth, Variable Dielectricity

In the first experiment, the dielectricity is chosen variable, i.e.:

\[
\varepsilon(x, y, z) = \sin(6x) \sin(6y) \sin(6z) + 2. \tag{5.2}
\]

The offset excludes a zero value in the coefficient \( \varepsilon \). The solution \( u(x, y, z) \) and corresponding right hand side \( f(x, y, z) \) are:

\[
u(x, y, z) = \sin(6x) \sin(6y) \sin(6z), \tag{5.3}
\]

and

\[
f(x, y, z) = -36 \cos^2(6x) \sin^2(6y) \sin^2(6z) \\
+ 108(\sin(6x) \sin(6y) \sin(6z) + 2) \sin(6x) \sin(6y) \sin(6z) \\
- 36 \sin^2(6x) \cos^2(6y) \sin^2(6z) - 36 \sin^2(6x) \sin^2(6y) \cos^2(6z). \tag{5.4}
\]

We consider the cubic domain \( \Omega = (0, 1) \times (0, 1) \times (0, 1) \). For the right hand side \( f \) a quadratic quadrature rule according to [70] is used. The element-wise integrals over the coefficients \( \varepsilon \) for the operator are evaluated with a lower approximation formula at the barycentric coordinates \((\frac{1}{4}, \frac{1}{4}, \frac{1}{4})\) with weight \(\frac{1}{6}\). We use no post-smoothing steps to keep the improved solution on the finest grid, which is done typically for \( \tau \)-extrapolation. The coefficients \( k \) are linearly averaged for a direct coarse grid correction. Table 5.1 shows that the consistency is enhanced from second to fourth order by \( \tau \)-extrapolation.

### 5.2.2 Jumping, Piecewise Constant Dielectricity

In contrast to the last paragraph, we assume that

\[
\varepsilon(x, y, z) = \begin{cases} 
\varepsilon_1 = 1, & x < 0, \\
\varepsilon_2 = 100, & x \geq 0.
\end{cases} \tag{5.5}
\]
In a typical setup from molecular dynamics (see [77]), the jump in $\varepsilon$ is aligned with one of the coordinate axes which simplifies the difficulty of the problem significantly. Therefore, we directly inject the coefficients $\varepsilon$ for $\tau$-extrapolation.

We consider the cubic domain $\Omega = (-0.5, 0.5) \times (0, 1) \times (0, 1)$, and harmonic functions (see figure 5.2)

$$u(x, y, z) = (a \cdot \sinh(\pi \sqrt{2}x) + b \cdot \cosh(\pi \sqrt{2}x)) \cdot \sin(\pi y) \cdot \sin(\pi z).$$  \hspace{1cm} (5.6)

These functions satisfy the Laplace equation $\Delta u = 0$. The parameters $a$ and $b$ are chosen such that they require continuity of $u$ across the interface and the continuity of $\varepsilon \mathbf{j} \cdot \mathbf{n}$ leading to the conditions

$$a = \frac{\cosh(\sqrt{2}\pi)}{100}, \quad b = \sinh(\sqrt{2}\pi).$$  \hspace{1cm} (5.7)

Similarly to the variable dielectricity test case, table 5.2 shows that the consistency is enhanced to fourth order by $\tau$-extrapolation.

In addition to improve the resolution by increasing the number of elements, $\tau$-extrapolation shows up as a HPC-confirm MG method to enhance the approximation order for variable and jumping coefficient test-cases.

In a next step for the Stokes problem, we plan to extrapolate only the velocity on the fine grid and let the pressure live on the coarser grid in order to end up with something similar to a stable mixed (Taylor-Hood) FE pair. However, this is not straightforward for an arbitrary domain, since first experiments show larger errors on the interfaces between the structured blocks.
Table 5.2: Error and order of consistency with and without \( \tau \)-extrapolation for a jumping piecewise constant coefficient problem.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Error (inf.) without ( \tau )-extr.</th>
<th>Error (inf.) with ( \tau )-extr.</th>
<th>Order of consistency without ( \tau )-extr.</th>
<th>Order of consistency with ( \tau )-extr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>9(^3)</td>
<td>1.77 ( \cdot ) 10(^9)</td>
<td>6.41 ( \cdot ) 10(^{-1})</td>
<td>1.98</td>
<td>3.53</td>
</tr>
<tr>
<td>17(^3)</td>
<td>4.48 ( \cdot ) 10(^{-1})</td>
<td>5.53 ( \cdot ) 10(^{-2})</td>
<td>1.99</td>
<td>3.79</td>
</tr>
<tr>
<td>33(^3)</td>
<td>1.13 ( \cdot ) 10(^{-1})</td>
<td>4.00 ( \cdot ) 10(^{-3})</td>
<td>2.00</td>
<td>3.90</td>
</tr>
<tr>
<td>65(^3)</td>
<td>2.82 ( \cdot ) 10(^{-2})</td>
<td>2.69 ( \cdot ) 10(^{-4})</td>
<td>2.00</td>
<td>3.90</td>
</tr>
<tr>
<td>129(^3)</td>
<td>7.05 ( \cdot ) 10(^{-3})</td>
<td>1.73 ( \cdot ) 10(^{-5})</td>
<td>2.00</td>
<td>3.96</td>
</tr>
</tbody>
</table>

5.3 Simulation of Porosified Structures

For electronic applications, the permittivity of Low Temperature Cofired Ceramics (LTCC) can be locally decreased. The resulting porosified LTCCs are e.g. suitable for direct antenna integration on glass-ceramic substrates. This section describes a computational mesh generation approach and an importing procedure to the CST electromagnetic simulation software\(^1\) of such a porous structure. This offers a way to explore the impact of fully inhomogeneous materials in comparison to a one-dimensional reduction to the electromagnetic field distribution.

5.3.1 Approach

Porosified LTCC is based on a chemical etching process that results in locally irregular material structures. The creation of these porosified structures can be accomplished by partial differential equations [78, 79]. We choose the time-dependent diffusion equation (5.8) [3]

\[
\frac{\partial c(x,t)}{\partial t} - \nabla \cdot (\varepsilon(x) \nabla c(x,t)) = 0 \text{ in } \Omega, \\
\frac{\partial c(x)}{\partial n} = c_0 \text{ on } \partial \Omega_D, \\
c(x) = 0 \text{ on } \partial \Omega_N, 
\]

(5.8)

where \( c(x,t) \) is the relative liquid concentration of acid as a function of space \( x \in \mathbb{R}^3 \) and time \( t \) on the domain \( \Omega \). Consequently, the relative material concentration is given as \( 1 - c \).

On one face of the cubic domain, a Dirichlet boundary condition \( \Omega_D \) is applied, whereas all other boundaries are Neumann type \( \Omega_N \). The initial condition \( c(x,0) \) is set to zero in the whole domain. The alumina grains in the material are modeled by a lower dissolving coefficient \( \varepsilon(x) \) than in the rest of the domain.

\(^1\)www.cst.com
Figure 5.3: Solution of equation (5.8) at various time steps $t$.

The domain $\Omega$ is resolved with 1 µm per grid point. This resolution is nearly in the range required to capture the smallest measured grains. Higher resolutions result in too many mesh cells for the subsequent electromagnetic simulations. The distribution of relative material in etching direction is mainly controlled by the number and size of the grains. Since the number and distributions of the dissolving coefficients of the grains are not known, they enter the system as free parameters. $5^3$ spherical grains per $10^3$ µm are randomly distributed with a radius between 1 – 4 µm and may overlap. The dissolving coefficient $\varepsilon(x)$ is set to 0.001 for alumina and 1 for the remaining space. The size of the grains is experimentally determined, which controls both, the characteristics of the material structure and the relative material gradient in the depth.

5.3.2 Simulation and Post-processing

For the time discretization an implicit Euler scheme is chosen. Figure 5.3 shows the solution of the differential equation at various time steps. The output file is imported in Paraview [80] and further post-processed.

A threshold is applied to the material concentration (figure 5.4 top, left) to filter the elements without material. Then, a small amount of detached material is removed by its missing mesh connectivity. The surface of the remaining block is exported in STL format. Our first attempt was to export contour-surfaces of the material concentration (figure 5.4 top, right) instead.
of a threshold filter using different free visualization tools including Paraview. However, none of them was able to export the mesh in a way that could be imported and converted to a 3D CAD structure by the software Microwave Studio within the Computer Simulation Technology environment. In addition the meshes are much larger and have degenerated triangles.

Figure 5.4 middle shows a comparison between the measured material-air 2D cuts of porosified DP951 and the generated model. In figure 5.5, the material distribution is compared to the generated structure in dependence of penetration depth and shows good agreement to the measured data. Deviations in the exemplary shown 2D cuts are consequences of the reduced information by taking a single cut through the material.

Clearly, the work within this thesis was to create the porosified material. However, colleagues obtained and concluded the following [3]:

"On this structure, performed finite 3D field simulations provided both, new insights in the resulting local decrease of permittivity and a worst case approximation of the reflection coefficients due to refractive index transitions.

Furthermore, the electromagnetic behavior of conductors vertically penetrating a porosified structure are investigated. The results of these studies show a surprisingly large impact of the vertical conductor position, since even a penetration of 4 µm results in an almost 50% decreased effectiveness in permittivity reduction. This newly gained knowledge highlights the importance of dealing with this matter, e.g. by sophisticated surface sealing techniques.
The reduction of effective relative permittivity $\varepsilon_{r,\text{eff}}$ due to porosification increases with frequency, so this technology offers the possibility of compensating a substrate inherent increase in $\varepsilon_{r,\text{eff}}$ with frequency. Lower variations of permittivity over frequency are of considerable interest for broadband applications.

However, it has to be considered that the measured data represents a porosification gradient and not a real permittivity gradient. In reality, the phosphoric acid predominantly dissolves certain material phases out of a composite. The real permittivity gradient has to be examined in further studies and is expected to vary with every other substrate material.

Currently, we aim to perform an electromagnetic analysis of frayed conducting structures in thick film technology. To allow this, a realistic 3D model of a frayed microstrip line on top of the porosified LTCC has to be generated and compared to measurements. In addition, we have to increase the resolution of material modeling to capture all additional effects. Therefore, we will have to increase the degree of parallelism, which is possible as illustrated in the next part of this thesis. The influence of its fringed edges on the electric field distribution, effective permittivity, and scattering parameters are planned to be investigated on the (coarsened) geometry. Since the computational effort is high, a corrected, simplified model shall be derived from the simulations.

Figure 5.5: Measured and modeled relative amount of material.
6 Stokes System

In this thesis, HHG was extended to solve the Stokes system in addition to scalar differential problems. Therefore, additional gradient operators, a CG solver on the Schur complement equation, stabilization techniques, and the separate treatment of boundary conditions for pressure and velocity were implemented. This chapter provides an overview of these extensions, which allows to calculate viscosity-dominated flow fields in extremely high resolutions. The Stokes solver is a central component for mantle convection codes.

6.1 Galerkin/Least-squares Stabilization

One issue which makes the discretization of the Stokes system interesting is that the spaces for velocity and pressure cannot chosen arbitrarily. Otherwise, the pressure field shows a spurious and oscillatory behavior. While standard equal order FE discretizations are not stable, mixed FE methods can lead to a stable scheme. One popular choice is to choose quadratic FE for the velocity and linear for the pressure, known as Taylor-Hood elements. In general, one necessary condition is that the dimension of the pressure space has to be smaller than or equal to the dimension of the velocity space. Moreover, the LBB (or inf-sup) condition provides an sufficient criterion. Besides introducing bubble functions, discontinuous Galerkin methods, or introducing mini elements (see [81] for an overview), one popular stabilization technique for non-stable standard Galerkin formulations is the Galerkin/Least-squares (GLS) approach.

The GLS stabilization [82] proposed by Hughes and Franca adds terms from the minimization of the square of the residual of the momentum equation to the weak formulation of the problem. An additional condition is provided by stationarity arguments. The GLS stabilization only affects the continuity equation for linear elements. The resulting Stokes system changes from equation (2.24) to

\[
\begin{align*}
(s(\phi^{\mu}_i), 2\mu s(u_h)) - (\nabla \cdot \phi^{\mu}_i, p_h) & = (\phi^{\mu}_i, \rho(T_h)g), \\
(\phi^{\rho}_i, \nabla \cdot u_h) - \sum_{e=1}^{n_{el}} \tau_e (\nabla \phi^{\rho}_i, \nabla p_h)_e & = 0.
\end{align*}
\]

(6.1) (6.2)
The chosen stabilization parameter $\tau_e = \frac{h^2}{12\mu}$ is optimal for linear elements, as they are used in HHG. The additional term is implemented in form of a patch-wise constant 15-point stencil.

### 6.2 Pressure Correction

The discrete Stokes equations are frequently solved using a pressure correction approach on equation (2.28). The first step is to calculate the velocity with an initial guess for the pressure by the momentum equations. In a second step, the transformed pressure equation $-BA^{-1}B^TP = BA^{-1}F$ (Schur complement system) is solved by a CG method. Algorithm 3 presents the single steps without any stopping criteria. The implementation in HHG includes stopping criteria for the number of pressure correction iterations $n_{PC}$, as well as for the MG solver in the lines 2 and 12. The threshold for the number of pressure correction iterations is determined by the scalar product $(r_i, r_i)$. The pressure correction algorithm is restarted every 10 iterations. Algorithm 4 shows the pressure correction algorithm with the preconditioner $M$ and the stabilization operator $C$.

We choose a lumped mass matrix preconditioner for $C$ that is known to be spectrally equivalent to the Schur complement. This preconditioning reduces effects of the gradually changing element sizes and turns out to be beneficial in our computations. Note that in case of strongly varying viscosity, we can weight the mass matrix with the inverse viscosity $\mu^{-1}$ in order to capture conditioning problems introduced by these variations [83].
Algorithm 4 Preconditioned Pressure Correction

1: // Solve \( u_0 \) by MG
2: \( Au_0 = f - B^T p_0 \)
3: \( r = Bu - Cp \)
4: for \( i = 1..n_{PC} \) do
5: if \( i = 1 \) then
6: \( s_1 = M^{-1}r_0 \)
7: else
8: \( z = M^{-1}r \)
9: \( \gamma = (r_{i-2}, r_{i-3}) / (r_{i-1}, r_{i-2}) \)
10: \( s_i = z_{i-1} + \gamma s_{i-1} \)
11: end if
12: // Solve \( v_i \) by MG
13: \( Av_i = B^T s_i \)
14: \( \alpha = (r_{i-1}, r_{i-2}) / (s_i, Bv_i - Cs) \)
15: \( p_i = p_{i-1} + \alpha s_i \)
16: \( u_i = u_{i-1} - \alpha v_i \)
17: \( r_i = r_{i-1} - \alpha Bv_i \)
18: end for

6.3 Boundary Conditions

For the Stokes system, the boundary conditions are not purely defined by the geometry anymore as before. Instead, it is necessary to define them separately on each specific variables or during execution of an operator. Even in easy benchmark problems like the lid-driven cavity, the velocity components are given by Dirichlet conditions, while the pressure is defined by free or homogeneous Neumann conditions on the boundaries. Therefore, conceptually all boundary primitives in HHG are now more similar to Neumann type conditions, since they are more flexible in terms of ghost layer exchanges and the availability of stencils. Of course, they still include all functionality necessary for Dirichlet conditions. The choice, which boundary conditions are actually applied to the variable, is now implicitly passed when applying an operator.

On the boundaries, it is often required to set the velocity vector in normal or tangential direction to a fixed value in order to impose e.g. acceleration or impermeable conditions. While this might be straightforward for geometries that are aligned with the coordinate axis, it might seems more elaborate from an implementation point of view for more complex geometries. However, fortunately an easy access to the normals is available by the FE formulation itself. A review of the implementation of normal and tangential boundary conditions in FE for incompressible flow are provided e.g. in the work of Engelman and Sani [84].

55
6.4 Lid-driven Cavity Example

We use the well-known cavity benchmark as a validation test case for our solver. In this setup, one face of a box moves with constant speed, while the other faces remain fixed. The discontinuity in the velocity, where the moving faces are adjacent to the non-moving faces leads to high and low pressure values at the corners. The results obtained by HHG are compared to a two-dimensional cavity flow presented in [81]. Figure 6.1 shows a good visual agreement of the streamlines, as well as comparable location of the vortex. Furthermore, the velocity profile between the two-dimensional case from [81] and the three-dimensional cavity in HHG are in good agreement as depicted in figure 6.2. Surprisingly, the no-slip conditions have nearly no influence on the velocity at the center-line in the cavity.

6.5 Mantle Convection-Diffusion Simulation

As stated in the motivation, mantle-convection simulations require in addition to the Stokes equations a convective dominated temperature equation. The mantle geometry is approximated as a thick spherical shell. The discretization itself is shown in section 7.5 in detail. To quantify the convective nature of the temperature equation compared to its diffusivity,
we introduce the Peclet number \( Pe \). In addition, the Rayleigh number \( Ra \) is associated with buoyancy driven flow and is the ratio of buoyancy and viscosity forces times the ratio of momentum and thermal diffusivity. For the following experiments, the fluid is solely driven by the temperature differences between the core temperature and the reference mantle temperature. Thus, we do not consider any internal heating within the mantle. Finally, the simplified system can be written as

\[
-\Delta u + \nabla p = -RaT e_r \\
\text{div} u = 0 \\
\partial_t T + u \cdot \nabla T = Pe^{-1} \Delta T.
\]  

(6.3a)  
(6.3b)  
(6.3c)

with temperature \( T \), and the radial unit vector \( e_r \) for a shell defined around the coordinate origin. For these equation, the product of \( Pe \) and \( Ra \) number determines the characteristics of the physics.

Before our implementation of the convection-diffusion equation is described, we briefly review other recent mantle-convection solvers. Geenen et al. [85] solve the Stokes equations with Taylor-Hood elements. They use a block-triangular-preconditioner to accelerate the Generalized Conjugate Residual (GCR) Krylov-subspace-method, where the preconditioner is applied by a limited accuracy solve. AMG act as a preconditioner for the arising sub-systems. The approach is closely related to that of [86]. They observe optimal scaling characteristics and robustness with respect to large viscosity variations.

Burstedde et al. [86, 87] implemented an adaptive projection stabilized Stokes solver using MINRES with an inner AMG solver on the mass matrix preconditioned system. The
energy equation is treated with a predictor-corrector scheme and streamline-upwind Petrov-Galerkin method (SUPG) [88] stabilization.

Kronbichler et al. [89] also use adaptive mesh refinement for their Taylor-Hood, or $Q_{k+2} - P_{k}$ for an improved mass conservation, discretized Stokes system. They basically follow the approach of Geenen et al. in their Stokes solver [85]. The advective terms of the energy equation are stabilized using the entropy-viscosity method [90].

In our case, the energy equation is semi-discretized by a stabilized Galerkin finite element method. In the following, we consider a simple artificial diffusion approach as it is one of two limiters for entropy-based stabilization techniques for high-resolution mantle convection [89]. We choose the artificial diffusion $v_{h,e} = 0.1 \cdot h_e |u_e|_{e}$.

However, in principle one might also implement the SUPG as it was done for HHG by colleagues via automatic stencil generation through the FEniCS Form Compiler (FFC) [91]. Clearly, the whole current discretization leaves room for improvement. However, we will have a chance for major adaption in the new Terra-Neo mantle convection code which will be written from scratch.

To illustrate typical output coming temperature fields for these kinds of equations, we solve them on a shell. Dirichlet conditions are applied for the velocities and temperature. While homogeneous no-slip conditions are applied for the velocities, initial and boundary for the temperature is set to one at the core, 0.5 inside the mantle, and zero on the surface. The time dependency of the energy equation is discretized by an explicit Euler scheme. We approximate the necessary inverse of the mass matrix by mass lumping. The energy equation shows up not to be the computationally expensive part compared to the solution of the Stokes system in at most every few time-steps. Due to the explicit scheme, the Courant-Friedrichs-Lewy (CFL) condition determines the time-step size for high $Ra$ and $Pe$ numbers.

Figure 6.3 - 6.5 shows three different convection-diffusion simulations with $Ra = 10^5$. In figure 6.3, the temperature field at different time-steps using a (low) Peclet number equal to one is visualized. The energy is transported by both, convection and diffusion. The resulting plumes are relatively thick, structured, and equally distributed. However, in reality they are assumed to be more advective. Figure 6.5 shows a series of time-steps for a Peclet number of 75, where the patterns of the plumes are more irregular and thinner. This is a setup which still can be handled without a stabilization of the convective term for this resolution.

However so far, all computations are of moderate resolutions and could be carried out on a workstation. The investigation of the scaling behaviour of the Stokes solver and the underlying MG algorithm is the focus of the next chapter in order to accomplish finer resolutions for our simulations.
Figure 6.3: Temperature (iso-surfaces) and velocities (glyphs) of convection-diffusion simulations on 2.5 mio grid points with $Ra = 10^5$ and $Pe = 1$ at various time-steps.
Figure 6.4: Temperature (iso-surfaces) and velocities (glyphs) of convection-diffusion simulations on 2.5 mio grid points with $Ra = 10^5$ and $Pe = 10$ at various time-steps.
Figure 6.5: Temperature (iso-surfaces) and velocities (glyphs) of convection-diffusion simulations on 2.5 mio grid points with $Ra = 10^5$ and $Pe = 75$ at various time-steps.
Part III

Large Scale Experiments
7 Scalability

If one has a look at upcoming hardware, parallel execution is becoming increasingly important. This trend has started earlier in Scientific Computing leading to massive parallelism at that time. This section introduces three different European petaflop machines. We discuss necessary changes to cope with hundreds of thousands of threads within the HHG framework including parallel mesh generation and hybrid parallelization. Furthermore, weak and strong scaling MG experiments are investigated on different meshes within the HHG approach. For weak scaling, the solution time is measured with increasing number of processors for a constant problem size per processor. However, in case of a strong scaling the problem size is fixed while the number of processors are varied.

Other parallel MG approaches were implemented in various frameworks, e.g., in waLBerla [92, 93] for finite difference discretizations on fully structured grids, Boomer AMG [94, 95] for unstructured grids and general matrices, Peano [96, 97] that is based on space-filling curves, or DUNE [98, 99] that is a general software framework for solving PDEs. In [100] an unstructured FE elasticity and plasticity problem was solved using geometric MG on an IBM PowerPC 4-way SMP cluster with up to 960 processors and 60% efficiency. Weak and strong scaling for a variable-coefficient Poisson problem with a proposed matrix-free distributed octree geometric MG algorithm up to 4096 processors on a Cray XT3 MPP system is demonstrated in [101]. Here, also a comparison with the algebraic multigrid scheme BoomerAMG is done.

Our MG sub-algorithms and parameters are standard. If not mentioned differently, they are specified in this chapter as

- three Gauss-Seidel steps for pre- and post-smoothing steps,
- linear interpolation,
- six MG levels,
- parallel conjugate gradient (CG) algorithm for solving on the coarsest grid, and
- standard coarse grid approximation.
7.1 Architectures

Within this chapter we compare the performance of HHG on three European supercomputers, namely JUGENE, JUQUEEN both located at FZ Jülich\(^1\), and SuperMUC located in the LRZ supercomputing center in Garching\(^2\). Further specification details can be found in A.2, A.3, and A.5, respectively. Table 7.1 presents a system overview of these clusters.

JUGENE, which is not in operation anymore, was the largest BlueGene/P installation with 294912 compute cores. The three year older performance results obtained on JUGENE serve as reference for the new results. Each node was equipped with a PowerPC 450 quadcore processor running at a low clock frequency of 850 MHz to achieve a low power consumption. The architecture provided a very high main memory performance, as measured with the stream benchmark. The overall peak performance was 1 petaflop/s. A three-dimensional torus network in combination with a tree-based collective network was available for parallel communication.

The BlueGene/Q system JUQUEEN is the successor of the JUGENE supercomputer with a peak performance of more than 5.0 petaflop/s. Although the clock-frequency still remains relatively low, it is nearly doubled and the number of processors per node quadrupled. Beneath the 16 cores available for user applications an additional 17th core is reserved for the operation system. Each core has four hardware (HW) threads which have to be utilized for best execution performance. The memory bandwidth has not scaled up accordingly but in order to compensate this disadvantage in part, e.g. the prefetching and speculative execution facilities have been improved. The torus network is extended to five-dimensions for shorter paths, and the collective network is now implemented within the torus network. The ratio of peak network bandwidth node performance and peak floating point performance is only

\(^1\)www.fz-juelich.de
\(^2\)www.lrz.de

<table>
<thead>
<tr>
<th></th>
<th>JUGENE</th>
<th>JUQUEEN</th>
<th>SuperMUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>IBM BlueGene/P</td>
<td>IBM BlueGene/Q</td>
<td>IBM System x iDataPlex</td>
</tr>
<tr>
<td>Processor</td>
<td>IBM PowerPC 450</td>
<td>IBM PowerPC A2</td>
<td>Intel Xeon E5-2680 8C</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>0.85 GHz</td>
<td>1.6 GHz</td>
<td>2.8 GHz</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>73728</td>
<td>24576</td>
<td>9216</td>
</tr>
<tr>
<td>Cores per Node</td>
<td>4</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>HW Threads per Core</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Memory per HW Thread</td>
<td>0.5 GB</td>
<td>0.25 GB</td>
<td>1 GB</td>
</tr>
<tr>
<td>Network Topology</td>
<td>3D Torus</td>
<td>5D Torus</td>
<td>Tree</td>
</tr>
<tr>
<td>Gflop/s per Watt</td>
<td>0.44</td>
<td>2.54</td>
<td>0.94</td>
</tr>
</tbody>
</table>

Table 7.1: System overview of the supercomputers JUGENE, JUQUEEN, and SuperMUC.
50% of that of BlueGene/P. On the other hand, the cores within each node and consequently the intra-node communication performance has drastically increased.

SuperMUC is a 3.2 petaflop/s IBM x iDataPlex cluster. This machine consists of 18 thin islands carrying 97.5% of total performance, and one fat island for memory intensive but only moderately parallel applications. Each thin island is equipped with 512 compute nodes. Two sockets with Sandy Bridge-EP Intel Xeon E5-2680 8C provide 16 physical cores (32 logical cores with hyperthreading). The Intel Xeon processors deliver a significantly higher core and node performance than the PowerPCs in the IBM architectures, however, at the price of higher power consumption. The nodes within an island are linked by an Infiniband non-blocking tree, whereas a pruned 4:1 tree connects all islands.

7.2 Unstructured Input Meshes

Before the discussion of experiments on the described architectures are done, some experiences on mesh generations are reflected. Mesh generators deliver a time saving and comfortable way to discretize an arbitrary geometry. A brief validation example will show that HHG is able to treat a fully unstructured grid generated via a 3D model, within a parallel setup.

7.2.1 Gmsh Interface

Gmsh [69] is a three-dimensional finite element mesh generator. Its design goal is to provide a fast, light and user-friendly meshing tool with parametric input and advanced visualization capabilities. A small Python script was developed to convert the generated MSH 2.0 output mesh files from Gmsh to HHG’s UGLi input files. This interface allows to import e.g. CAD mesh formats, like visualized in figure 7.1. After a coarse Gmsh discretization, the resulting mesh can be converted and read from HHG. The regularly refined mesh with a plotted trigonometric function is depicted in figures 7.2 and 7.3.

7.2.2 Validation

This section contains a basic validation case of the parallel implementation. The initial mesh is depict in figures 7.2 and 7.3. It is a benchmark geometry generated by the electromagnetic code consortium (EMCC) [102]. The coarse grid mesh generation was done by Gmsh, leading to 16 028 tetrahedral elements. Each element was distributed to one compute core on JUGENE and refined up to seven times.

\[\text{http://geuz.org/gmsh/}\]
Figure 7.1: Visualization of a simple 3D model in Gmsh.

Figure 7.2: Top view of a two times refined HHG mesh.

Figure 7.3: Side view of a two times refined HHG mesh.
Table 7.2: Discretization error when doubling the resolution in every row. Domain is chosen according to figure 7.2.

<table>
<thead>
<tr>
<th>Levels</th>
<th>Unknowns</th>
<th>Discretization Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$1.59 \cdot 10^5$</td>
<td>$9.74 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>2</td>
<td>$1.32 \cdot 10^6$</td>
<td>$2.53 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>3</td>
<td>$1.00 \cdot 10^7$</td>
<td>$6.38 \cdot 10^{-6}$</td>
</tr>
<tr>
<td>4</td>
<td>$8.68 \cdot 10^7$</td>
<td>$1.59 \cdot 10^{-6}$</td>
</tr>
</tbody>
</table>

We solve as a test problem equation (2.11) with the function $\sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$ as right hand side $f$. The maximal point-wise discretization error is shown in table 7.2. In each refinement step the resolution is doubled resulting in a quadratic convergence according to the theory for linear FEs.

### 7.3 Massively Parallel Mesh Treatment

When running in parallel one process reads the input grid file including the partitioning and distributes the grid information to all other processes. Then, each process has to build up its own part of the unstructured FE grid. All input elements are refined equal times, leading to conforming grids and allowing a static grid partitioning.

Nevertheless, during construction of the refined grids in the setup phase global acting algorithms have to be used, e.g. to identify the neighboring processes or to perform a consistent global numbering over all processes. HHG has to number the edge and face macro-elements, which are not given by the input grid.

Before adapting HHG to JUGENE, each parallel process assembles the full unstructured input grid locally. For each input element, HHG loops over all other elements to identify its neighbors. However, this is a very time and memory consuming procedure for a large number of processes. Especially on relatively slow cores with a small main memory, one alternative could be to relocate the assembly into a pre-processing step and store all important information in the input grid file. In order to maintain flexibility, it was decided to build up a local grid instead. This grid contains only macro-volumes that belong to the local process and neighboring macro-volumes. While the mapping of the volumes to the processes is given by the input grid file, the points belonging to vertices, edges, and faces could be mapped arbitrarily. To keep this procedure simple, local, and consistent over all processes such macro-elements are assigned to the same process as the adjacent volume with the largest volume ID given by the input grid file.

While the IDs of vertices and volumes are given by the input grid file, process-consistent IDs for the edges and faces have to be chosen by ourselves. The macro-elements are uniquely characterized by the IDs of its vertices. These tuples of two vertices for the edges, and
three vertices for the triangle faces are sorted in a list. Then IDs are assigned to the tuple according to the index of the macro-element in the list. Algorithm 5 presents the structure of the grid generation in more detail. Thus, in HHG the complexity for the grid generation is limited by the sorting algorithm.

Although it would be best to solve the problem on the coarsest grid by a parallel direct solver, this is inefficient for a larger number of unknowns on the coarsest grid. Therefore, a CG method is chosen as a solver for the coarsest grid. To estimate the required number of CG iterations, we assume that

- the required number of CG iterations is proportional to the diameter (in FE nodes) of the computational domain for finite differences or finite elements (see e.g. [103]),

- high frequency errors are eliminated by smoothing, and

- one CG iteration for the diameter one with one coarsest grid point is sufficient.

Consequently, a rough estimate for the required number of CG-steps $n_{cg}$ for $l$ grid levels and $d = 3$ dimensions is

$$n_{cg} \approx c \cdot \frac{\sqrt[3]{N}}{2^{l-1}},$$  \hspace{1cm} (7.1)

with a constant $c \approx 1$.

### 7.3.1 Hybrid Parallelization

The new system architectures with more powerful and complex compute nodes make a hybrid parallelization approach especially attractive and potentially profitable, since they provide better opportunities for a shared memory parallelization via OpenMP\textsuperscript{4}. Thus a hybrid parallelization strategy, including message passing for coarse grain parallelism, and shared memory parallelism within a node for finer scale parallel execution, has been found essential for exploiting the full potential of architectures like JUQUEEN or SuperMUC.

- In a pure MPI parallel setting, the available main memory per process is only 256 MB per process on JUQUEEN. This is too small for the three largest runs described in the next sections. In contrast, a hybrid parallelization increases the available main memory for each process.

- On SuperMUC, the scaling breaks down when too many MPI processes are being used. Here, a hybrid parallelization helps to limit the total number of MPI processes and this helps to maintain scalability when going to simulations of extremely large size.

\textsuperscript{4}www.openmp.org
Algorithm 5 Parallel Grid Generation [4]

1: Build the local mesh on each process
2: for all input volumes $v$ in the mesh file do
3: if $v$ is mapped to current process then
4: create local macro-vertices, edges, faces, and volume objects belonging to $v$
5: end if
6: end for
7:
8: Build a ghost layer of direct adjacent macro-elements on each process. Memory for the grid points is not allocated for those primitives.
9: for all input volumes $v$ in the mesh file do
10: if $v$ is not mapped to the current process then
11: for all local faces $f$ do
12: if $f$ shares all vertices with any face of $v$ then
13: create macro-vertices, edges, faces, and volume objects belonging to $v$
14: flag new objects as process boundary element or ghost elements
15: end if
16: end for
17: end if
18: end for
19:
20: Assign global IDs to macro-edges and faces, process mapping for all elements
21: create edge/face tuple list $l$ (one tuple entry per vertex ID)
22: initialize edge/face ID-counters
23: for all input volumes $v$ in the mesh file do
24: for all vertex, edge, and face $p$ belonging to $v$ do
25: create a tuple $t$ of $p$ (one tuple entry per vertex ID)
26: if $t$ is not contained in $l$ then
27: add $t$ to tuple list $l$
28: if $t$ identifies an existing macro-element object then
29: assign current edge/face ID-counters to a macro-edge/face object $t$
30: update process mapping of a macro-vertices, edge, volume object belonging to $v$ by the process-ID of $v$ (always the largest process-ID holds)
31: end if
32: increment edge/face ID-counters
33: end if
34: end for
35: end for
<table>
<thead>
<tr>
<th>MPI-Processes</th>
<th>Threads/Core</th>
<th>OpenMP-Threads</th>
<th>Time</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>4</td>
<td>1</td>
<td>3.09</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>4</td>
<td>2</td>
<td>3.10</td>
<td>99%</td>
</tr>
<tr>
<td>1024</td>
<td>4</td>
<td>4</td>
<td>3.21</td>
<td>96%</td>
</tr>
<tr>
<td>512</td>
<td>4</td>
<td>8</td>
<td>3.45</td>
<td>90%</td>
</tr>
<tr>
<td>256</td>
<td>4</td>
<td>16</td>
<td>3.95</td>
<td>78%</td>
</tr>
<tr>
<td>128</td>
<td>4</td>
<td>32</td>
<td>4.33</td>
<td>71%</td>
</tr>
<tr>
<td>64</td>
<td>4</td>
<td>64</td>
<td>5.22</td>
<td>59%</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>32</td>
<td>5.77</td>
<td>54%</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>16</td>
<td>8.36</td>
<td>37%</td>
</tr>
</tbody>
</table>

Table 7.3: Hybrid parallelization on JUQUEEN for moderate problem sizes with 4096 threads.

<table>
<thead>
<tr>
<th>MPI-Processes</th>
<th>OpenMP-Threads</th>
<th>Time</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>1</td>
<td>1.29</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>2</td>
<td>1.32</td>
<td>98%</td>
</tr>
<tr>
<td>256</td>
<td>4</td>
<td>1.39</td>
<td>93%</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
<td>1.49</td>
<td>86%</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>2.02</td>
<td>64%</td>
</tr>
</tbody>
</table>

Table 7.4: Hybrid parallelization on SuperMUC for moderate problem sizes with 2048 threads without Hyperthreading.

The current OpenMP implementation in HHG supports parallelism inside kernel executions and copying of ghost layers on several primitives. However, the MPI instructions are executed asynchronously, but not explicitly OpenMP-parallel. Further, OpenMP introduces an additional overhead for spawning threads, which is especially critical on the coarsest grids, where the workloads per thread are small. The quality of the MPI/OpenMP parallel execution is reflected in table 7.3 for JUQUEEN. All runs up to the last two are executed by four threads per compute core. The timings conclude that the serial fraction of the code is still between 1 – 2%. We will use a hybrid parallelization with up to eight OpenMP-threads for the largest parallel run on JUQUEEN in the following scaling experiment as the performance loss is still not too high. Table 7.4 shows the experiment for SuperMUC, respectively. The serial fraction turns out to be similar to JUQUEEN.

### 7.3.2 Weak Scaling on JUGENE

In order to validate our choice of the MG components for a large number of compute nodes, weak scaling experiments were done for a complete MG V-cycle on JUGENE with 12 tetrahedral input volumes per core.
<table>
<thead>
<tr>
<th>Cores</th>
<th>Input volumes</th>
<th>Unknowns (-10^6)</th>
<th>CG it.</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1 536</td>
<td>535</td>
<td>15</td>
<td>5.64</td>
</tr>
<tr>
<td>256</td>
<td>3 072</td>
<td>1 071</td>
<td>20</td>
<td>5.66</td>
</tr>
<tr>
<td>512</td>
<td>6 144</td>
<td>2 142</td>
<td>25</td>
<td>5.69</td>
</tr>
<tr>
<td>1024</td>
<td>12 288</td>
<td>4 287</td>
<td>30</td>
<td>5.71</td>
</tr>
<tr>
<td>2048</td>
<td>24 576</td>
<td>8 577</td>
<td>45</td>
<td>5.75</td>
</tr>
<tr>
<td>4096</td>
<td>49 152</td>
<td>17 159</td>
<td>60</td>
<td>5.92</td>
</tr>
<tr>
<td>8192</td>
<td>98 304</td>
<td>34 326</td>
<td>70</td>
<td>5.86</td>
</tr>
<tr>
<td>16384</td>
<td>196 608</td>
<td>68 669</td>
<td>90</td>
<td>5.91</td>
</tr>
<tr>
<td>32768</td>
<td>393 216</td>
<td>137 355</td>
<td>105</td>
<td>6.17</td>
</tr>
<tr>
<td>65536</td>
<td>786 432</td>
<td>274 744</td>
<td>115</td>
<td>6.41</td>
</tr>
<tr>
<td>131072</td>
<td>1 572 864</td>
<td>549 555</td>
<td>145</td>
<td>6.42</td>
</tr>
<tr>
<td>262144</td>
<td>3 145 728</td>
<td>1 099 176</td>
<td>180</td>
<td>6.52</td>
</tr>
<tr>
<td>294912</td>
<td>294 912</td>
<td>824 365</td>
<td>110</td>
<td>3.80</td>
</tr>
</tbody>
</table>

Table 7.5: Weak scaling for HHG MG V(3,3)-cycles on JUGENE. Twelve tetrahedral input volumes are distributed to each compute core. The number of CG iterations to approximate the coarsest grid increases proportional to the diameter of the domain (see equation (7.1)) [9],[4].

As a test case, the material has a piecewise constant dielectricity, and thus constant stencils within each HHG block and each geometric primitive are sufficient. Consequently, the numerical efficiency is extremely high and in a relative sense, the communication is very intensive. Therefore, this is quite a challenging setup for maintaining parallel scalability as the performance studies will show.

One goal is to sustain the MG convergence rate of 0.1 that can be predicted by local Fourier analysis [49] for a V(3,3)-cycle up to the full JUGENE with its 294 912 compute cores. Since the number of grid levels is fixed and the problem size increases the number of required CG iterations on the coarsest grid grows from 15 to 180 (see table 7.5).

The full machine run uses seven instead of six levels. In the semi-structured approach, a structured region cannot be shared by multiple processors. Thus, one cannot use the full main memory. The additional level reduces the number of CG iterations. Overall, these numbers fit to the predicted number of CG iterations with $c \approx 0.5$ (see equation (7.1)). A more comprehensive picture is given in figure 7.4. Here, the V-cycles are compared to F-cycles that come close to $c \approx 1$. Note that F-cycles require a better coarse grid approximation than V-cycles [40].

Although more CG iterations are required for larger problem sizes, one finds that for the largest runs in table 7.5 the time spent on the coarsest grid is around only 12.5% of the total time for one V-cycle. The resulting parallel efficiency compared to a serial core runtime of 4.25 s is still at 65.2% when solving for $10^{12}$ unknowns on 262 144 cores. About $60 \cdot 10^{12}$

73
Figure 7.4: Required number of CG iterations on the coarsest grid per cycle to sustain the overall MG convergence rate with increasing numbers of compute cores. The prediction is computed via equation (7.1) [9],[4].

Floating point operations per second (TFLOP/s) are reached in the MG solver. This is way below 10% of the peak performance and can be roughly explained as follows: Without using SIMD (double hammer) a potential factor of two compared to peak performance is lost due to memory alignment issues. Additionally, a factor of five is caused by instruction level serialization as it is described in section 8.2.1. This means that one drops down to at most 10% of the peak performance when neglecting parallelization.

MG convergence also depends on the type of cycle as observed in figure 7.4. W-cycles for example have a very similar convergence like F-cycles but are computationally nearly two times more expensive. Algorithmically the difference between V-cycles and W-cycles is the double recursive call in case of W-cycles. As an example in our application setups typical asymptotic convergence rates are 0.21 for a V-cycle and 0.18 for a W-cycle. Thus W-cycles instead of V-cycles only pays off, if a W-cycle is less then \( x \approx 10\% \) \( (0.21^{(1+x)} = 0.18) \) slower.

For strong scaling as it is done e.g. in section 7.3.4, communication has a large impact on the run-time. The number of grid points that have to be communicated decreases by a factor of four in 3D on each level. This means the coarser grids require for W-cycles \( \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \ldots \approx 100\% \) additional communication compared to the communication on the finest level. In contrast, a V-cycle requires only \( \approx 33\% \) additional amount of communication. Therefore, only V-cycles are considered in the remaining work.
7.3.3 Weak Scaling on JUQUEEN

This section shows the scalability of the HHG approach on a state of the art cluster. The program is compiled with the IBM XL compiler suite the BlueGene cluster JUQUEEN. Only basic optimization levels (-O2 or -O3 -qstrict) are possible to maintain correct program execution. More aggressive optimization (-qhot) causes a performance drop of around a factor of two.

Table 7.6 shows the run-time results of a scaling experiment. The smallest test run already solves a system of slightly more than $10^8$ unknowns and one V-cycle takes approximately 2.3 seconds. Note that this is performed on a single compute node on JUQUEEN, demonstrating the high efficiency of the HHG approach.

In each further row of the table, the problem size is doubled as well as the number of nodes. The full machine could eventually solve a linear system with $3.3 \cdot 10^{12}$ unknowns corresponding to more than $10^{13}$ tetrahedral finite elements. In total, this computation uses 300 out of the almost 400 terabytes of main memory during the solution process.

Four hardware threads are necessary to saturate the performance of one processor core leading to a parallel execution of more than one million threads. Although the computational time increases only moderately we note that the coarse grid solver is only a straightforward CG iteration. Therefore in large runs, more than half a second of the V-cycle execution time is spent in the increasing number of CG iterations on the coarsest grid, that is caused by larger and larger coarse grids. This shows clearly, that for perfect asymptotic scalability a better coarse grid solver would be necessary. Nevertheless, we believe that the results with the CG solver indicate clearly that the coarse grid solver performance is not as critical for scalability, as has been discussed in the older literature on parallel MG methods. In our experience, a careful and well-designed implementation of the communication routines can reduce the negative effect of the coarse grid bottleneck significantly. Based on these results, we see no reason why MG should be considered a-priory inferior to other iterative methods in terms of parallel efficiency.

7.3.4 Strong Scaling

Next, we perform strong scaling experiments on the clusters JUGENE and lima (cluster A.4). In contrast to JUGENE, lima has only 500 compute nodes, but each of them consists of two sockets equipped with fast hexa-core Xeon 5650 Westmere 2.66 GHz processors ($P_{cores} = 12$). All cores are running at 2.93 GHz due to Intel Turbo Boost Technology in combination with water cooling. The overall peak performance of the cluster is about 56.7 TFlop/s. On lima we use the Intel MPI compiler 4.0.

For strong scaling between one and 768 cores on both clusters are used. On the coarsest grid 30 CG iterations are sufficient to resolve the problem well enough. The problem size
Table 7.6: Weak scaling experiment on JUQUEEN solving a problem on the full machine with up to $3.29 \cdot 10^{12}$ unknowns.

for each cluster is chosen such that the time to solution for one V-cycle on a single node is similar, i.e. 0.51 seconds for lima and 0.59 seconds for JUGENE.

On JUGENE in total a linear system of $4.11 \cdot 10^6$ unknowns on four grid levels is solved. Figure 7.5 visualizes strong scaling behavior of HHG for the test case. In between one and 64 cores the parallel efficiency stays above 80%. With increasing number of cores the parallel efficiency drops to roughly 30% for 768 compute cores. The runtime reduces from 8.5 to 0.036 seconds per V-cycle from one to 768 cores, what corresponds to a speedup of 236.

On lima a larger problem with in total $3.33 \cdot 10^7$ unknowns on five grid levels is solved. Figure 7.6 shows the performance of HHG for this setup. In the first part of the graph up to 12 cores reflect the intra-node behavior. The parallel efficiency drops to $\approx 60\%$ mainly due to shared cache effects. Similar to JUGENE the parallel efficiency for 768 cores is around 30%. The time reduces from 3.79 to 0.015 seconds per V-cycle, what corresponds to a speedup of 252.

For the same problem size like on JUGENE, one ends up with 0.01 seconds on 768 cores. This already indicates that there is a point when decreasing the problem size per core does not help any more to increase the overall performance. In the next chapter this point is predicted via performance models.

### 7.4 Comparison with Other Peta-Scale Clusters

Now we compare the performance of HHG on JUGENE, JUQUEEN, and SuperMUC. Differently from the previous part, the focus lies more on the different architectures than on the
Figure 7.5: Strong scaling for HHG on JUGENE from one to 768 compute cores. $4.11 \cdot 10^6$ unknowns are solved with $V(3,3)$ cycles. The ideal curve prescribes the optimal speedup, where the runtime on one core is divided by the number of compute cores [4].

Figure 7.6: Strong scaling for HHG on lima from one to 768 compute cores. $3.33 \cdot 10^7$ unknowns are solved with $V(3,3)$ cycles. The ideal curve prescribes the optimal speedup, where the runtime on one core is divided by the number of compute cores [4].
code itself. This section contains parts of the proceedings article [5], which is accepted for publication.

In contrast to the BlueGene systems the program is compiled with the Intel compiler suite and IBM MPI for SuperMUC with -O3 -xavx compiler flags. As a reference, one V-cycle takes 4.25 s for JUGENE, and 1.18 s on SuperMUC on one compute node.

Figure 7.7 shows strong efficiency drops when advancing from one node to several nodes. This is especially prominent on both BlueGene systems. However, from then onwards the parallel efficiency stays nearly constant. Only the transition from a single Midplane on BlueGene/P, or one Node Card on BlueGene/Q to larger subportions of the architecture induce more significant performance drops again. On SuperMUC the efficiencies up to a quarter of an island ($2.6 \cdot 10^{10}$ unknowns) differ between the MG cycles. We believe that this is caused by perturbations due to other applications running simultaneously on the same island. From a quarter of an island to half of an island ($5.2 \cdot 10^{10}$ unknowns) the performance even improves. However, when leaving a single island of the architecture, the parallel efficiency drops significantly. This is likely caused by the reduced communication performance beyond each island in the pruned 4:1 tree. For more than two islands we also disable hyperthreading to obtain substantially more reproducible run-times. In contrast to this observation, the run-times of both BlueGene machines remain more stable for all
Table 7.7: Single node and parallel efficiencies (scaling), as well as power consumptions of used parts of the clusters while running HHG [5].

<table>
<thead>
<tr>
<th></th>
<th>JUGENE</th>
<th>JUQUEEN</th>
<th>SuperMUC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single Node</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peak flop/s (constant dielectricity)</td>
<td>6%</td>
<td>7%</td>
<td>12%</td>
</tr>
<tr>
<td>Peak flop/s (variable dielectricity)</td>
<td>9%</td>
<td>10%</td>
<td>13%</td>
</tr>
<tr>
<td>Peak bandwidth (constant dielectricity)</td>
<td>11%</td>
<td>53%</td>
<td>60%</td>
</tr>
<tr>
<td><strong>Parallel Efficiencies</strong> (at ≈ 0.8 Pflop peak)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scaling (constant dielectricity)</td>
<td>65%</td>
<td>64%</td>
<td>72%</td>
</tr>
<tr>
<td>Scaling (variable dielectricity)</td>
<td>94%</td>
<td>93%</td>
<td>96%</td>
</tr>
<tr>
<td>Scaling – without CG (constant dielectricity)</td>
<td>75%</td>
<td>70%</td>
<td>79%</td>
</tr>
<tr>
<td>Number of processes</td>
<td>262144</td>
<td>262144</td>
<td>32768</td>
</tr>
<tr>
<td>Energy improvement compared to JUGENE (constant dielectricity)</td>
<td>1</td>
<td>6.6</td>
<td>4.7</td>
</tr>
<tr>
<td>Energy improvement compared to JUGENE (variable dielectricity)</td>
<td>1</td>
<td>6.4</td>
<td>3.2</td>
</tr>
</tbody>
</table>

problem sizes. The parallel efficiency is comparable to be the best results on SuperMUC. However, we could not map our mesh onto the torus networks, since the coarsest mesh is basically unstructured.

First scaling experiments on SuperMUC showed a breakdown at 65,536 MPI processes, resulting in roughly four times longer run-times, as well as fluctuations in the timings of up to 15 seconds between the single V-cycles compared to runs with 32,768 MPI processes. Figure 7.7 displays that there have already been problems on 32,768 MPI processes (corresponding to $4.1 \cdot 10^{11}$ unknowns or four islands). The results on larger machine sizes use the hybrid parallelization that allows us to execute the largest two runs with only 16,384 MPI processes, leading to a significantly improved parallel efficiency. The largest run was carried out on 16 islands of the cluster.

Different from the behavior of SuperMUC the hybrid parallelization on JUQUEEN, as used for the largest runs, clearly decreases the parallel efficiency. However a hybrid parallelization is still necessary as explained above in order to have enough main memory available.

Table 7.7 shows the single node performance, parallel efficiencies, and energy consumptions relatively to JUGENE. The runs utilized an node allocation providing ≈ 0.8 Pflop/s nominal peak. Even though a major design goal of the BlueGene/P was to have a low energy consumption, the next generation could improve the energy consumption by a factor between six to seven for our application. The SuperMUC turns out not to be as energy efficient as JUQUEEN, however it does not require such a high degree of parallelism from the application.
We now analyze the single node performance as given in table 7.7. This is for the case of constant dielectricity.

On JUGENE one MPI process is assigned to each compute core. Since the processors provide high memory bandwidth, codes tend to be more limited by instruction throughput than by memory bandwidth. However, the kernel that applies the stencil is affected on JUGENE from a serialization within the PowerPC multiply-add instructions. Additionally, a correct memory alignment for vectorized loads (for the SIMD units) is not assured due to the varying loop sizes that are caused by the tetrahedral macro elements. The limited-issue width and the in-order architecture of the processor is leading to further performance limitations and eventually result in a node performance of only 6.1% of the peak performance (see table 7.7). This is for a complete MG cycle.

For a comparison, we refer to results of Datta et al. [104, 105] who present auto-tuning results for an averaging 29-point stencil on this architecture. Their baseline implementation achieves about 0.035 GStencil/s updates which corresponds to 7.7% of the peak performance for a reference (in-cache) implementation. Basically two of eleven optimization techniques (including e.g. padding, core blocking, software pre-fetching) techniques can achieve a significant speedup. These are common subexpression elimination and register blocking. While the first inherently cannot be applied for our stencil, since we do not have redundant calculations, the register blocking results in our case roughly in a speedup of two. In principle we could use this code optimization but it leads to very small sub-blocks that will suffer from non-constant loop sizes. Moreover, the issue with serialization remains a bottleneck, which is not the case for the averaging stencil.

On JUQUEEN, we assign one MPI process to each thread (64 per node). The stream benchmark shows that it is possible to run at a high fraction of $\approx 85\%$ of the effective maximal memory bandwidth of 27.8 GB/s by using one process per node. Two or four threads per node saturate the effective bandwidth completely. Going from one to two threads per core, HHG gives a factor of two improvement in performance. In these cases, the code is still instruction bound like on BlueGene/P. Going from two to four threads per core, the additional speedup is only a factor 1.3. Overall, in this case, a MG cycle utilizes in average about 18.1 GB/s of the main memory bandwidth. Only by reducing the main memory footprint and possibly improving the core performance itself, there is a chance for further significant reductions of the execution time.

Similarly to the situation on JUQUEEN, the code is mainly memory bandwidth limited on the SuperMUC node architecture. However, the nodes can saturate the bandwidth better and its machine balance suits better the characteristics of this code. Thus it was possible to achieve a better flop/s performance than on JUQUEEN. However, hyperthreading for single node improves the performance only insignificantly by at most a few percent.
7.5 Icosahedral Mesh

In the remaining and the following chapter, we want to predict the performance of a mantle convection code utilizing HHG’s data structures. For this we consider the same mesh generation approach as in the mantle convection simulation program TERRA (see Baumgardner [106, 107] and section 8.5). The latter is based on the discretization of the unit sphere by means of an icosahedral mesh, an approach commonly used also in meteorology. For details see e.g. [108, 109, 110].

Parts of the following texts concerning the generation of the icosahedral mesh, as well as simulations carried out on the geometry were published in [6].

7.5.1 General

In general for implementation and especially for performance reasons the use of structured meshes is preferable in PDE-based applications. However, in the case of the sphere there is no known way to generate a subdivision into more than 120 equivalent domains\textsuperscript{5}, see e.g. [111]. One possible compromise is the use of an icosahedral mesh which leads to an at least logically block-structured grid. Further other possible discretization of the sphere are often based on the Yin-Yang grid [112] or the cubed-sphere [113].

The initial mesh is an icosahedron, which is mapped onto the sphere. This results in 12 vertices and 20 spherical triangles whose 30 edges are formed by geodesic arcs. This and the then following refinement process are illustrated in figure 7.8. The icosahedral base grid is

\textsuperscript{5}The exception are lunes, which finally degenerate to arcs.
iteratively refined by midpoint subdivision. In each refinement step each spherical triangle is subdivided into four new triangles by adding additional vertices at the midpoints of the three edges of the base triangle and connecting them by three new geodesic arcs. This process is repeated recursively to produce a two-dimensional spherical discretization up to the resolution required. Scaling this grid to different radii produces the final three-dimensional mesh. Connecting vertices with the same inclination and azimuth angles constructs prismatic elements with spherical triangles as bottom and top surface (see figure 7.9). On these the FE ansatz functions of TERRA are then defined.

### 7.5.2 Tetrahedral Decomposition

In the following benchmarks, we provide to HHG a coarse input mesh generated by the approach described in section 7.5.1. In contrast with TERRA we only go to a moderate resolution in this step. We then replace the spherical prisms by meshes with planar triangles on top and bottom. Each of these prisms is then further decomposed into three tetrahedra. Successive refinement of these tetrahedra, as shown in figure 2.2 in the introduction, leads to the mesh with the final resolution.

While a simple decomposition of one prismatic element into three tetrahedra exists, the situation is more involved for a complete mesh. We will illustrate this issue and suggest a conforming triangulation. Figure 7.10 shows two prisms, decomposed into three tetrahedral elements each. For example, the left prism consists of tetrahedra \( abce \), \( acde \), and \( cdef \). Both prisms are connected by the quadrilateral face \( bcef \), which consists of two triangle faces. The illustrated discretization is consistent. However, if edge \( ce \) were flipped to be edge \( bf \) in one of the prisms, then the FE mesh would become non-conforming. Thus the subdivision of all prisms in the icosahedral mesh must be designed carefully to avoid such inconsistencies. To provide a conforming triangulation between two icosahedral layers, one
might proceed as follows: We note that there are only two points with the same inclination and azimuth angles. A simplified representation of figure 7.10 is displayed in figure 7.11(a). With this notation, we produce a valid triangulation, if all triangles (representing prisms) do

1. not share one edge with arrows pointing to opposite directions,
2. not form a cycle with the arrows.

Figure 7.12 shows a possible conforming triangulation of an icosahedral mesh. Note that the first condition has to be also fulfilled on the boundaries of the mesh. To refine the icosahedral layers in a consistent way, a refinement like in figure 7.11(b) is used. For parallel computing on distributed memory systems, this mesh is partitioned suitably onto the processors of the computer cluster.

In terms of physical dimensions the inner boundary of the spherical shell is set at \(3.48 \cdot 10^3\) km which is the location of the core-mantle boundary. The outer boundary and uppermost layer is placed at \(6.37 \cdot 10^3\) km and thus is at the level of Earth’s surface. Figure 7.13 presents a cut through a mesh that is four times refined spherical, as well as in radial, direction. Here the radial refinement is graded both towards the Earth’s surface and the core-mantle boundary.

### 7.5.3 Diffusion Problem

First, we solve an elliptic diffusion problem on the icosahedral geometry, which is a core component of the Stokes system for the next sub-section. The test machine used for the
(a) Simplified notation of figure 7.10.
(b) Possible refinement of a prismatic element.

Figure 7.11: Simplified notation of a prismatic mesh discretised with tetrahedra assuming only two grid points on top of each other in the third dimension. Only quadrilateral cutting edges are shown [6].

Figure 7.12: Simplified notation of possible conforming triangulation of an icosahedral mesh, assuming only two grid points on top of each other in the third dimension. Only quadrilateral cutting edges are shown [6].
The experimental setup starts with an icosahedron that is refined four times, such that each triangle of the initial coarse mesh on the Earth’s surface represents an area of $5.10 \cdot 10^8 \text{km}^2 / (20 \cdot 4^4) \approx 10^5 \text{km}^2$. For our test case, this mesh is then further refined seven times, however, this is performed without using curved boundaries. The final mesh then contains triangles that cover an area of $6 \text{km}^2$ of the Earth’s surface. In radial direction, we obtain a resolution of approximately $2.8 \text{km}$ between two grid points. Altogether $5.15 \cdot 10^{11}$ elements are used in this discretization.

Table 7.8 shows results for a strong scaling from 20480 to 81920 cores solving a scalar Poisson equation with constant coefficients on this mesh using HHG’s parallel MG algorithm. The quantity GStencil/s represents the number mesh nodes in units $10^9$ that can be processed per second. Here, this is equivalent to the degrees of freedoms that are processed.

Most of the memory is consumed by four variables per grid point that are used in our MG
implementation. We use the MG components of chapter 7 in this setup. A 15-point stencil is applied seven times for each fine grid variable per V-cycle, thus the stencil evaluation for one grid point requires 29 flops.

Finally, we run the same setup, but with one additional refinement step for the input mesh and using 163840 compute cores. Thus the resolution doubles in every dimension. The time for one V-cycle for this size is 6.87 s. Note that for a perfect weak scaling this time should be the same as for the smaller problem with 20480 compute cores, which is approximately also obtained. Note that for this problem, the mesh contains now $4.1 \times 10^{12}$ finite elements and that the surface resolution is now down to $1.5 \text{km}^2$ per surface triangle.

Figure 7.14 shows a smaller strong scaling experiment from 2560 to 10240 processes in addition. Strong scaling in both scalings reduce the run-time from about six to two seconds, while increasing the number of compute cores by a factor of four. Here, the total main memory consumption increases weakly, since additional ghost layers are introduced. Further, utilizing more cores leads to additional amount of memory consumed by the additional program instances itself.

For the larger strong scaling experiment, V-cycles with a constant and variable (on-the-fly) stencil evaluation are compared in figure 7.15. The memory consumption mainly increases because one additional variable $\epsilon$ for the variable coefficient case has to be stored. Only the stencils on the faces are kept in main memory, since otherwise the run-time increases by a factor of three due to indirect addressing. All stencils in the volumes are assembled during each evaluation. This leads to a five times slower run-time than in case of the constant stencil. Since the communication is nearly the same, the strong scaling speed-up is better (x3.80) for the variable viscosity than for the constant viscosity (x3.43) case.
Figure 7.15: Comparison of run-time and memory for constant and variable (on-the-fly) stencil operator evaluation in a strong scaling setup. The finest grid has $5.1 \cdot 10^{10}$ elements. Speedup factors are given compared to the 20 480 cores run.

### 7.5.4 Stokes Problem

Next, we consider the Stokes solver on the icosahedral grid. Similarly, to the last experiments, a sequence of runs using increasingly finer resolution is performed. One approach to resolve the Earth up to one kilometer locally is to employ adaptive mesh refinement strategies in combination with algebraic multigrid methods see [86, 89]. Next, we will demonstrate that it is also possible to achieve this resolution globally.

To this end, we perform two experiments with different accuracy requirements. In a time stepping scenario, we expect that a reduction of the residual by $10^3$ will be sufficient (case A), while for a full solution (as may e.g. be required in the initial time step), we require a residual reduction by $10^8$ (case B). Table 7.9 and table 7.10 display these weak scaling results for JUQUEEN and SuperMUC, respectively. MG is used only in the inner loop of the conjugate gradient method for the pressure-Schur system, where it consumes the major portion of the overall solution time.

Recall that HHG uses uniformly refined tetrahedral blocks. Here these mesh blocks have a size of $129^3/6$ grid points. For this numerical experiment, two such mesh blocks are distributed to each thread on a single JUQUEEN core, while sixteen of these blocks are assigned to each core of SuperMUC. Using four hardware threads turns out to be necessary on JUQUEEN to reach full performance, but on SuperMUC hyperthreading did not lead to a performance improvement. The setup times (i.e. mesh generation, stencil computations) are in the range of 10s – 20s for JUQUEEN and somewhat less for SuperMUC.

The first rows of the table 7.9 and table 7.10 display the execution time spent for the solution on a single compute node. Initially, this corresponds to a global resolution of the Earth
mantle for a mesh with a resolution of 32 km. In each following row, the mesh resolution is halved and consequently the numbers of threads increases by factor eight.

The computations on JUQUEEN are executed using a hybrid parallel programming model where OpenMP and MPI are combined such that four OpenMP threads are used in the largest run. This is necessary to compensate for the increasing main memory requirements needed for MPI. In contrast, on JUQUEEN the execution times for the hybrid parallelization remain similar to an equivalent pure MPI parallelization. Experiments varying the number of threads for this machine are provided in the next sub-section.

Figure 7.16 displays a breakup of the execution time for the individual algorithmic components when run with 15 360 threads for case (A). As a diagnostic tool we use Scalasca [114]. Note that about one third of the overall solver time (pressure correction) is spent in communication. Within the Stokes solver, the implicit treatment of the Schur complement equation by MG consumes the major portion of time. The results also clearly display that the parallel efficiency of MG algorithms is best on the finest level, since the balance between computation and communication becomes worse on coarser levels of the hierarchy. The data in figure 7.16 show the time for the coarsest level (CL) separately (multigrid, CL). Note that the coarsest mesh solver is here implemented by a conjugate gradient method and thus ul-

Table 7.9: Weak scaling of the Stokes equation on the mantle geometry on JUQUEEN. The pressure residual is reduced by at least three orders of magnitude for (A), and seven orders of magnitude for (B).

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Threads</th>
<th>Grid points</th>
<th>Resolution</th>
<th>Time: (A)</th>
<th>Time: (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>2.1 \cdot 10^{37}</td>
<td>32 km</td>
<td>30 s</td>
<td>89 s</td>
</tr>
<tr>
<td>4</td>
<td>240</td>
<td>1.6 \cdot 10^{38}</td>
<td>16 km</td>
<td>38 s</td>
<td>114 s</td>
</tr>
<tr>
<td>30</td>
<td>1920</td>
<td>1.3 \cdot 10^{39}</td>
<td>8 km</td>
<td>40 s</td>
<td>121 s</td>
</tr>
<tr>
<td>240</td>
<td>15 360</td>
<td>1.1 \cdot 10^{10}</td>
<td>4 km</td>
<td>44 s</td>
<td>133 s</td>
</tr>
<tr>
<td>1 920</td>
<td>1 228 800</td>
<td>8.5 \cdot 10^{10}</td>
<td>2 km</td>
<td>48 s</td>
<td>153 s</td>
</tr>
<tr>
<td>15 360</td>
<td>983 040</td>
<td>6.9 \cdot 10^{11}</td>
<td>1 km</td>
<td>54 s</td>
<td>170 s</td>
</tr>
</tbody>
</table>

Table 7.10: Weak scaling of the Stokes equation on the mantle geometry on SuperMUC. The pressure residual is reduced by at least three orders of magnitude for (A), and seven orders of magnitude for (B).

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Threads</th>
<th>Grid points</th>
<th>Resolution</th>
<th>Time: (A)</th>
<th>Time: (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>2.1 \cdot 10^{37}</td>
<td>32 km</td>
<td>16 s</td>
<td>51 s</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>1.6 \cdot 10^{38}</td>
<td>16 km</td>
<td>20 s</td>
<td>63 s</td>
</tr>
<tr>
<td>15</td>
<td>240</td>
<td>1.3 \cdot 10^{39}</td>
<td>8 km</td>
<td>24 s</td>
<td>73 s</td>
</tr>
<tr>
<td>120</td>
<td>1 920</td>
<td>1.1 \cdot 10^{10}</td>
<td>4 km</td>
<td>27 s</td>
<td>80 s</td>
</tr>
<tr>
<td>960</td>
<td>15 360</td>
<td>8.5 \cdot 10^{10}</td>
<td>2 km</td>
<td>34 s</td>
<td>107 s</td>
</tr>
<tr>
<td>7 680</td>
<td>1 228 800</td>
<td>6.9 \cdot 10^{11}</td>
<td>1 km</td>
<td>41 s</td>
<td>131 s</td>
</tr>
</tbody>
</table>
Figure 7.16: Run-time breakup of the parallel Stokes solver on JUQUEEN with 15,360 threads.

timately is a non-optimal method. In total, 20 parallel CL iteration steps are performed in each V-cycle for the breakup and for the following strong-scaling experiment. During the weak scaling, the number of CL iterations increases from 10 to 70.

For the large problems 7 pressure correction iterations are required for case (A) and 22 for case (B) to reach the prescribed accuracy. This yields an average residual reduction per CG step of about 2.33 for the solve, and 2.7 for the time-stepping case. The last refinement steps in our grid generation are performed without curved boundaries. This negatively influences the problem condition with coarse resolutions and would lead to a higher number of pressure correction iterations. In order to have a comparable workload we limit the iteration number for small problems to that of the large problems. The timings, however, include the increasing solution time on the growing coarsest mesh. The pressure-correction scheme is restarted twice for (B) in agreement with [35]. Three V-cycles are performed before the first pressure-correction iteration, and one within each subsequent iteration for each velocity component. The solver clearly shows the mesh independent convergence that is predicted by theory.

We point out that the run-time increases only moderately when we increase the number of grid points by four orders of magnitude. The finest discretization includes $6.9 \cdot 10^{11}$ grid points corresponding to $4.1 \cdot 10^{12}$ linear tetrahedral finite elements. Although these scaling experiments are still performed for a simplified model and therefore eventually more memory and longer compute times will be needed for more advanced models and better discretizations, we are confident that mantle convection simulations with a uniform global resolution of 1 km are within reach with the upcoming generation of near-exa-scale supercomputers.

Since in practical mantle convection simulations, reducing the compute time for each time step may be more important than going to finer spatial resolutions, we investigate the strong
Figure 7.17: Run-times of strong scaling experiments on SuperMUC from 1 920 to 30 720 cores for cases (A) and (B).

Figure 7.18: Run-times of strong scaling experiments on JUQUEEN from 3 840 to 122 880 cores with different numbers of threads per core (right) for case (A).
scaling characteristics of our solver. The initial program run is determined by the main memory limits, defining the smallest number of nodes that can be used for this kind of simulation. We then perform a strong scaling, i.e., we fix the mesh size and consequently the problem size to a resolution of 4 km, while increasing the number of cores.

On SuperMUC, we do not need to reduce the initial block size, since each core already has sufficient memory available. This leads to an almost linearly decreasing run-time when going from 1920 to 30720 cores, as illustrated in figure 7.17. For the maximum number of cores, only one block is assigned to each compute core. Throughout this range, the parallel efficiency remains above 50% for both cases (A) and (B).

For JUQUEEN, the block size has to be decreased from \(129^3/6\) to \(65^3/6\) grid points after the first doubling of compute nodes. The smaller block sizes lead to a clearly visible performance decrease, as displayed in figure 7.18. Regarding the use of hardware threads on each compute core, several OpenMP/MPI combinations are displayed. While a significant speedup is observed when proceeding from one to two hardware threads, four threads only have a positive effect, if larger blocks can be used. The time reduces to 7.7 s for two threads having two blocks on each of the 122 880 cores.
8 Performance Analysis and Modeling

The previous experiments addressed practical issues of the solver including its convergence properties. Scope of this chapter is to provide a fundamental understanding of the performance of HHG on two current supercomputers. Therefore, a performance model including the run-times for the most time-consuming components of the MG algorithm is developed. Then, the model is validated for different problem sizes for a MRI voxel geometry. However, recently this model was extended to the Stokes problem and validated on the icosahedral mesh via the performance tool Scalasca [114] on JUQUEEN.

Related to this work are other hardware-adapted and parallel MG implementations mainly on fully structured grids to achieve the best possible performance e.g. found in [115, 116, 46, 117, 118, 119, 120] and performance models like in [121, 122].

While these performance models are mainly for one compute node and fully structured grids, the parallel inter-node performance is usually not taken into account. Because of its stencil-nature, existing knowledge about stencil-based performance analysis can be adapted to HHG. Further the communication depends not strongly on the geometry, like it is the case for e.g. fully unstructured meshes. Therefore, the messages between the nodes are reasonably well predictable. However, as we will show this MG approach has still different communication properties that are interesting to model and understand. Therefore, in my opinion HHG can give a particular good insight and understanding of a block-structured MG algorithm performance during highly parallel execution.

8.1 General Model

In order to predict the expected runtime of the MG algorithm and to estimate the quality of our implementation an diagnostic performance model is developed in this section. Basis for the model is the runtime $t_{\text{total}}$ of one V-cycle. Since we do not overlap computation and communication in our implementations,

$$ t_{\text{total}} = t_{\text{comp}} + t_{\text{comm}}. $$

with computation time $t_{\text{comp}}$ and communication time $t_{\text{comm}}$. Note that in principle overlapping would be possible but does not improve the performance drastically. One reason is that in the parallel smoothing procedure described in section 2.2.5 only step 5 and 6 and step
Table 8.1: Number of arithmetic, load, and store instructions for different MG components per (fine) grid point. The second value for the load instructions includes the loads of the stencil coefficients.

<table>
<thead>
<tr>
<th>Component</th>
<th>Add</th>
<th>Mult</th>
<th>Loads</th>
<th>Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smoothing</td>
<td>14</td>
<td>15</td>
<td>15(30)</td>
<td>1</td>
</tr>
<tr>
<td>Residual</td>
<td>15</td>
<td>15</td>
<td>16(31)</td>
<td>1</td>
</tr>
<tr>
<td>Restriction</td>
<td>$\frac{14}{8}$</td>
<td>$\frac{1}{8}$</td>
<td>$\frac{15}{8}$</td>
<td>$\frac{1}{8}$</td>
</tr>
<tr>
<td>ProlongationAdd</td>
<td>$\frac{14}{8}$</td>
<td>$\frac{8}{8}$</td>
<td>$\frac{24}{8}$</td>
<td>$\frac{8}{8}$</td>
</tr>
</tbody>
</table>

7 and 8 can be partially overlapped, because edges and vertices do only have one layer of inner points. Other aspects of overlapping will be addressed in more detail when explaining the contributions to $t_{comm}$.

### 8.1.1 Computation

For simplicity, exactly one HHG input volume is assigned to each parallel process. The coarsest grid ($l = 0$) is constructed by refining each tetrahedral input volume twice (see figure 2.2 (middle)). Thus, every process owns at least one inner grid point. Additional grid levels are created by refining each volume in eight child elements. Thus, a process owns for grid level $l$ in average

$$n(l) \approx \frac{(2^{(l+2)} + 1)(2^{(l+2)} + 2)(2^{(l+2)} + 3)}{6} - \frac{2^{(2l+2)} + 1)(2^{(l+2)} + 2)}{2}$$

inner grid points $n$, neglecting edges and vertices.

Table 8.1 lists the number of instructions per fine grid point for different MG components on one level. The most expensive parts of a MG algorithm (see algorithm 1) are matrix-vector operations like $A_I h u_h$ occurring, e.g., when computing the residual or smoothing. In our case this means the 15-point stencil representing $A_I$ is applied. Note that for smoothing the updated point does not have to be loaded. The usually necessary division by the center stencil coefficient counts as a multiplication, since our stencil is constant and the inverse of this stencil entry can be pre-computed. The number of grid points is reduced by a factor 8 in each grid level that is found in the linear interpolation and restriction instructions.

Basically two different factors that may limit the overall compute performance are taken into account in our model:

1. time for loading data from main memory to the caches and storing it back limited by memory bandwidth
2. time for processing all instructions limited by instruction throughput
Note that for simplicity the time for cache transfers that can be in principle a substantial part of the overall computing time [122] is neglected.

Depending on the concrete processor with clock frequency $f$, table 8.1 can help to estimate the number of clock cycles $n_{cycles,i}$, $i = \{1, 2\}$ necessary for updating one grid point within one V-cycle for all three factors. The highest number of clock cycles $n_{cycles} = \max_i n_{cycles,i}$ is the limiting factor in terms of performance and the overall time with $L$ grid levels is then

$$t_{comp} = \frac{n_{cycles} \cdot \sum_{l=1}^{L-1} n(l) + n_{cg cycles} \cdot n_{cg} \cdot n(0)}{f} \quad (8.3)$$

with $n_{cg}$ CG iterations on the coarsest grid, which is modeled as one stencil evaluation requiring $n_{cg cycles}$ cycles.

### 8.1.2 Communication

Network bandwidth as well as latency determine the performance of the communication. The overall time for communication $t_{comm}$ is composed of the delays generated by bandwidth $t_{bw}$, direct communication latencies $t_{lat}^{direct}$, and reduction latencies $t_{lat}^{coll}$

$$t_{comm} = t_{bw} + t_{lat}^{direct} + t_{lat}^{coll} \quad (8.4)$$

For our application the bottleneck turns out to be the available bandwidth for large and the latency for small problem sizes. More concretely, the reduction kernels on the coarsest grid required within the CG iterations are a limiting factor for small problem sizes.

#### Bandwidth

In HHG the number of communicated grid points on level $l$ for each process and one ghost layer exchange is roughly

$$n_{comm}(l) = (1 - \frac{\lambda}{4}) \cdot 4 \left( \frac{2^{(l+2)} + 1}{2} \cdot \frac{2^{(l+2)} + 2}{2} \right), \quad (8.5)$$

where $\lambda$ are the number of all four faces, which are located within the same compute node and do not require inter-node communication. The factor $(1 - \frac{\lambda}{4})$ determines how many points have to be communicated via network. The remaining term are the number of grid points of the four triangle faces, which are roughly half of a quadrilateral face.

One ghost layer has to be exchanged after smoothing, residual computation, and prolongation, this sums up to $\nu_1 + \nu_2 + 2$ transfers on each grid level for one $V(\nu_1, \nu_2)$-cycle and

$$n_{comm}^{mg}(l) = (\nu_1 + \nu_2 + 2) \cdot n_{comm}(l). \quad (8.6)$$
On the coarsest grid with equation (7.1) are
\[ n_{comm}^{mg}(0) = n_{cg} \cdot n_{comm}(0) \quad (8.7) \]
and in total
\[ n_{comm} = \sum_{l=0}^{L-1} n_{comm}^{mg}(l) \quad (8.8) \]
grid points to send and receive.

The transfer time for sending double precision values on \( p \) cores per node is
\[ t_{bw} = \frac{n_{comm} \cdot p_{cores} \cdot 8 \text{ Bytes}}{bw_{comm}}, \quad (8.9) \]
where \( bw_{comm} \) is effective network bandwidth.

**Latency**

Latency is especially interesting on coarser grids, where the messages become shorter. Direct as well as collective communication requests cause latencies.

**Direct communication** On all grid levels except on the coarsest one, \( \nu + 2 \) times a ghost layer exchange takes place. On the coarsest grid one stencil evaluation within the CG iterations has to be done that requires one ghost layer exchange. The total number of latency delays is
\[ n_{lat}^{direct} = \gamma \cdot ((\nu + 2) \cdot (L - 1) + n_{cg}). \quad (8.10) \]
with \( \gamma \) describing the number of latency delays for one ghost layer exchange.

HHG has to update all macro-elements when applying a stencil. One additional exchange of adjacency information is required at the end of such an operation. Even if the communication of the same class of macro-elements are overlapped, the latency occurs \( \gamma = 4 \) times per ghost layer exchange.

If overlapping of the latencies of communication of the same class is not possible, the number of messages in HHG that have to be sent and received is roughly \( \gamma = 24 + 3 + 6 + 4 = 37 \) for a process owning one macro-vertex: a macro-vertex in our grid is adjacent to maximal 24 other processes, and one additional messages for every other three vertices, six edges, and four faces have to be received.

Hence, the latency due to direct communication is
\[ t_{lat}^{direct} = n_{lat}^{direct} \cdot t_{hop} \cdot n_{hops}. \quad (8.11) \]
Here, \( n_{hops} \) is the number of hops and \( t_{hop} \) the time for one hop within the network. A hop is a packet jump from one compute node to a neighboring one.
Collective Communication Collective operations like a global reduction via MPI_Allreduce are only occurring when computing two scalar products within the CG iteration on the coarsest grid. Usually a binomial algorithm with complexity $\log_2(N)$ is used on the network.

For $p_{nodes}$ compute nodes and two scalar products per CG iteration we assume a total latency $t_{lat}^{coll}$ of

$$t_{lat}^{coll} = 2 \cdot \log_2(p_{nodes}) \cdot t_{red} \cdot n_{cg},$$

(8.12)

with the time $t_{red}$ for one reduction.

8.2 Model on JUGENE

8.2.1 Computation

As introduced before, each compute node of JUGENE consists of one socket that is equipped with a PowerPC 450 quadcore processor ($p_{cores} = 4$). The cores run at a quite low clock frequency of 850 MHz to achieve a very low power consumption. With two multiply-add instructions per cycle, one core has an theoretical peak performance of 3.4 GFlop/s in double precision, one compute node 13.6 GFlop/s. The L1 data cache has a size of 32 KB per core, while the L2 cache is shared between four cores with a total size of 8 MB. One compute node has a total of 2 GB main memory and is capable of delivering 13.6 GB/s bandwidth. The double precision flop to byte ratio is thus 1.0.

Detailed information about the hardware can be extracted from the IBM Redbook [123]. We refer to certain pages in the IBM Redbook later on in this section. Most of its data are theoretical specifications, what leads to an optimistic prediction in our performance model. Next we check which of the three factors from subsection 8.1.1 limits the single core compute time of one smoothing step.

Memory bandwidth One can assume that several lines of the arrays for the right hand side and the unknowns fit into cache. Thus instead of 15 double values one only has to load three and store one double value for the unknown vector, and load one double value for the right hand side. In average one compute core can load 4 Bytes per cycle from main memory. Thus one smoothing step takes

$$n_{cycles,1} = \frac{8 \text{ Bytes} \cdot 5}{4 \text{ Bytes/cycle}} = 10$$

(8.13)

processor cycles.
**Instruction throughput**  

The PowerPC 450 is capable of doing one SIMD instruction per processor cycle. However, the compiler is not able to vectorize the HHG code automatically and thus does only one multiply-add operation per cycle. One reason for that are memory alignment constraints, which are difficult to accomplish in the current implementation of the tetrahedral data structures. Therefore, SIMD vectorization is not supported in HHG, and the estimate for one smoothing step becomes

\[
 n_{\text{cycles},2} = \frac{15 \text{ FMADD}}{1 \text{ FMADD/cycle}} = 15 \tag{8.14}
\]

per grid point. The multiply-add instructions can be done in parallel to loads and stores, which take both also one processor cycle, thus

\[
 n_{\text{cycles},2} = \frac{30 \text{ load instr.}}{1 \text{ load instr./cycle}} = 30. \tag{8.15}
\]

The assumption here is (see table 8.1) that also the stencil coefficients have to be load. These numbers suggest that instruction throughput can be the limiting factor on JUGENE.

There exists even sharper bounds for the performance. The next considered operations are multiply-add instructions, since residual and smoothing calculations spend most of the time doing operations of the type

\[
c_0 u_0 + c_1 u_1 + \ldots + c_{15} u_{15} \tag{8.16}
\]

which form seems to be optimally suited for fused multiply-add instructions. Unfortunately, the evaluation of one pair has to wait until the previous pair is completed. This takes approximately five cycles (p. 115), thus the residual and \( \nu \) smoothing iterations sum up to

\[
 n_{\text{cycles},3} = 15 \cdot 5 \cdot (\nu + 1) \tag{8.17}
\]

cycles per grid point.

One can question the estimate of \( 15 \cdot 5 = 75 \) cycles for one smoothing iteration, since in principle the -O4 optimization level of the IBM compiler allows to reorder floating point operations to break dependencies. But looking at the assembler code tells us that the compiler is not able to do so within the HHG code. If one tries to help the compiler by parenthesizing, e.g.

\[
((c_0 u_0 + c_1 u_1) + (c_2 u_2 + c_3 u_3)) + \ldots + ((c_{14} u_{14} + c_{15} u_{15}) \tag{8.18}
\]

or

\[
(c_0 u_0 + \ldots + c_4 u_4) + (c_5 u_5 + \ldots + c_9 u_9) + \ldots \tag{8.19}
\]

the brackets remove the instruction level serialization, but also decrease the number of fused multiply-add instructions. Additionally, it forces the compiler to insert \texttt{NOP} instructions. The assembler code for one red-black GS iteration implemented as in (8.16) takes 75 cycles per point. Version (8.18) and (8.19) take 72 and 80 cycles including 5 and 12 \texttt{NOP} instructions per grid point, respectively.
Summarizing, neglecting all other operations, equation (8.3) can be refined for JUGENE as

\[
t_{\text{comp}} = \frac{n_{\text{cycles},3} \cdot \sum_{l=1}^{L-1} n(l) + 15 \cdot 5 \cdot n_{cg} \cdot n(0)}{850 \text{MHz}}. \tag{8.20}
\]

### 8.2.2 Communication

The JUGENE has a three-dimensional torus network with two links of 425 MB each per direction of the torus. This torus network connects each processor with two neighboring in every Cartesian direction. Further the compute nodes are connected by a global tree network, which can be used for collective and I/O operations.

**Bandwidth**

For JUGENE, the effective bandwidth between two nodes is indirectly proportional to the number of hops (p. 355). The torus network has an average edge length of \( \sqrt[p_{\text{nodes}}]{p_{\text{nodes}}} \) nodes in each Cartesian direction. One message has to traverse this edge length for all three dimensions in the worst case to get from one corner to the opposite corner of a Cartesian grid. In the average case for irregular communication only half of the hops are assumed. An additional factor of two distinguishes a torus network from a mesh network [124]. Thus the number of hops using \( p_{\text{nodes}} \) nodes are roughly in average

\[
n_{\text{hops}}^{avg} = 3 \cdot \frac{\sqrt[p_{\text{nodes}}]{p_{\text{nodes}}}}{4}. \tag{8.21}
\]

Our grid partitioning on JUGENE assures that every tetrahedral volume shares at least one face with another volume that lies on the same compute node, therefore \( \lambda = 1 \) for equation (8.5).

Most messages are relatively short and usually the MPI short or eager protocol is active (p. 69f.). This protocol uses deterministic routing to reduce latency on BlueGene/P. This fits to a strong scaling setup, where latency is critical. One Blue Gene/P node is connected by two channels with 4 bits per cycle bandwidth to the network in every three-dimensional Cartesian direction. One node is able to use six of the channels (p. 9). In our case one HHG process has only three neighbors with a large communication face. Furthermore, it is possible that all neighbors of a process are mapped in one line on the torus, since the coarsest grid is logically unstructured. The slowest process limits the overall performance, consequently this situation has only to occur for one process. Under this assumption, the effective bandwidth \( bw \) per node utilizing two channels is

\[
bw_{\text{comm}} = \frac{0.5 \text{ Bytes} \cdot 850 \text{ MHz} \cdot 2}{n_{\text{hops}}^{avg}} = \frac{850 \text{ MB/s}}{n_{\text{hops}}^{avg}} \tag{8.22}
\]
and the transfer time on JUGENE is thus with \( p_{\text{cores}} = 4 \)

\[
t_{bw} = \frac{n_{\text{comm}} \cdot 4 \cdot 8 \text{ Bytes} \cdot n_{\text{avg}}}{850 \text{ MB/s}}.
\]  (8.23)

**Latency**

**Direct communication** The latency is proportional to number of hops. Since the coarsest grid is logically unstructured, the worst case is that two communicating processes have the maximal point-to-point Manhattan distance. The number of hops is in worst case (see section 8.2.2)

\[
n_{hops} = 2 \cdot n_{\text{avg}}.
\]  (8.24)

The hardware latency of a package with message descriptor of 32 Bytes is \( t_{hop} = 100 \text{ ns} \) per hop (p. 8). Since not differently mentioned in the IBM Redbook, the transfer of small messages are assumed to overlap (\( \gamma = 4 \) in equation (8.10)), and usually does not have a significant influence to the performance in our application. Hence, the latency due to direct communication is

\[
t_{\text{direct}} = n_{\text{lat}} \cdot 100 \text{ ns} \cdot 2 \cdot n_{hops}.
\]  (8.25)

**Collective Communication** BlueGene/P has a separated network for collective communication. Unfortunately, the HHG grid cannot utilize all allocated nodes with our meshing strategy. Therefore, a binomial algorithm with complexity \( \log_2(N) \) is used on the torus network.

MPI_Allreduce takes 37.96 \( \mu s \) for 512 compute nodes (p. 86). Hence, a single reduction step lasts

\[
t_{\text{red}} = \frac{37.96 \mu s}{\log_2(512)}
\]  (8.26)

in equation (8.12).

**8.3 Model on lima**

**8.3.1 Computation**

On the Xeon architecture within the lima cluster, one can make a similar analysis as in the last section in order to see that one is also limited by instruction throughput and not by memory bandwidth.
In case of the row-wise red-black 15-point stencil within the smoother the compiler produces SIMD instructions for additions and multiplications, but no packed load instructions due to memory alignment issues. The stencil coefficients are kept in registers using the full SSE register size and two unknowns can be updated instruction-level parallel. The additional prize are additional SSE unpack and move instructions between registers.

Intel provides a reference summarized in the manuals of Agner Fog \(^1\) for an in-depth scheduling analysis, which we did with extracted information from the assembler code. For the Xeon architecture all instructions can be split into μops. These μops are assigned to one of three execution ports, a memory read port p2, a memory write address port p3, and a memory write data port p4. The limit in our code is the instruction throughput for the execution port p5. More precisely, 26 move, 12 unpack, and one jump instructions spend one μop on port p5 each for updating two unknowns. An update of an unknown lasts at least

$$n_{cycles,2} = (v + 1) \cdot \frac{39}{2}$$  \( (8.27) \)

cycles per grid point with the HHG stencil. Thus, for lima equation (8.3) reads

$$t_{comp} = \frac{n_{cycles,2} \cdot \sum_{l=1}^{L-1} n(l) + \frac{39}{2} \cdot n_{eg} \cdot n(0)}{2930MHz}.$$  \( (8.28) \)

### 8.3.2 Communication

The nodes of lima are connected by a 40 GBit/s bandwidth network in each direction. This quad data rate InfiniBand network is fully non-blocking with a backbone which is able to provide the maximal traffic produced by a node. In contrast to JUGENE the fat tree network uses static routing for all message sizes.

**Bandwidth**

On lima \( p_{cores} = 12 \) processes are running on one node. Therefore more tetrahedral faces lie between processes that are located on the same node than on JUGENE. By our grid partitioning on lima we assume two of the four faces are transferred intra-node, therefore \( \lambda = 2 \) in equation (8.5).

Furthermore, the effective bandwidth on lima does not strongly depend on the number of hops, but on the message sizes. Figure 8.1 shows a ping-pong benchmark on lima. Both processes are located on different nodes and are sending messages to each other. In order to utilize both links of the lima cluster the message was sent in both directions via MPI.

\(^1\)http://www.agner.org/optimize/
On lima the effective bandwidth for a node is

\[ bw_{\text{eff}} = \sum_{l=0}^{L-1} \frac{bw(n_{\text{comm}}(l)) \cdot n_{\text{msg}}^{\text{comm}}(l)}{n_{\text{comm}}}, \]  

(8.29)

with measured data from the ping-pong benchmark \( bw(n_{\text{comm}}(l)) \), which depends on the message size \( n_{\text{comm}}(l) \) of a given level.

Note that one introduces an error here. Measurements showed that the bandwidth further increases when sending more messages of smaller size, or decreases when sending messages of larger sizes concurrently utilizing more cores per process.

**Latency**

**Direct communication** To check whether the average latency per message decreases when sending more messages concurrently the following benchmark was done. We implemented a straightforward all-to-all communication using \( MPI\_\text{Isend} \). To determine latency only one double value per message is sent. Figure 8.2 shows the average time per message with increasing number of processed messages per compute core. It starts with two processes on two different nodes. For each measurement point the number of processes, and consequently the number of compute cores, increases by two. We conclude that nearly no overlapping is possible. While below approximately 34 processed messages per compute core the average time per message stays constant, the time even increases linearly.
when trying to send more messages concurrently. In HHG one executes less MPI_Isend operations without waiting, thus one does not get an additional penalty for each message. Thus $\gamma = 37$ in equation (8.10). In equation (8.11) one has $t_{hop} = 2 \mu s$ and $n_{hops} = 1$ for lima.

Collective Communication For lima we assume the latency $t_{red} = 2 \mu s$ from our ping-pong benchmark for one reduction.

8.4 Model Validation

In this section first a performance study of the MG smoother on single compute nodes is done and then and strong scaling results for the whole MG algorithm on JUGENE and lima are presented in order to validate our performance models. Furthermore, we are able to predict for a given problem size which number of compute nodes should be used in order to minimize the overall solution time.

8.4.1 Single Node Smoother Performance

JUGENE Since most of the runtime within the MG solver is spent in smoothing and residual calculation that are both dominated by stencil evaluations, we will analyze the
smoother performance and validate our performance model for it first. As an example, if one chooses the computational kernels according to table 8.1 within a serial setup and the finest grid has approximately $2.7 \cdot 10^6$ grid points, approximately 70\% of the runtime of a V(2,2)-cycle is spent in the smoother and 19\% in the residual calculation.

Figure 8.3 depicts the measured number of unknowns per core that can be updated within different kinds of Gauss-Seidel iterations on JUGENE. The performance is often measured in unknowns per second when solving linear systems, because it allows us to compare different problem types and sizes. Since the algorithm is not memory bandwidth limited, it makes no difference for the single core performance, if we run one or four processes per compute node.

For a row-wise red-black Gauss-Seidel 15-point stencil (RB 15-point) used within HHG or a lexicographic Gauss-Seidel 15-point stencil (GS 15-point) one expects a lower bound of 75 cycles per unknown (see subsection 8.2.1), thus

$$\frac{850 \text{MHz}}{75 \text{cycles}} \approx 11.3 \text{MUnknowns/s}. \quad (8.30)$$

This number is close to the measured performance for small array sizes. For larger array sizes the performance drops below 10 MUnknowns/s what could be explained by additional cache effects resulting from several input streams for the right hand side and the solution.

The serialization can be avoided by a row-wise red-black Gauss-Seidel (RB 15-point, disjoint). Here, we added disjoint pragmas for the innermost loop in order to allow the com-
piler to assume that the arrays are not overlapping in memory. A simple trick is to tell the compiler that input and output stream of the unknown vector \( u^h \) are independent of each other. A second pointer is introduced which points also to the memory location of the unknown vector and both are marked as disjoint. During stencil evaluation, one of the two pointers is used as the source array, the other one as the destination array. This trick is possible, because red and black unknowns are updated independently of each other, and it improves the throughput significantly. For the RB 7-point stencil half of the work is necessary, what results in roughly a speedup of two in performance.

**Lima** In contrast to JUGENE the single node smoother performance on the lima cluster depends on the number of cores used within one compute node, therefore we split our results in two graphs for single core runs in figure 8.4 and runs on one compute node in figure 8.5. Shared cache effects inside the L3 cache are responsible for the differences in performance between figure 8.4 and figure 8.5 for both 15-point stencils. In detail, the RB 15-point stencil used within HHG is bound by

\[
\frac{2930 \text{ MHz}}{\frac{39}{2} \text{ cycles}} \approx 150.2 \text{ MUnknowns/s} \quad (8.31)
\]

estimated via our performance model in section 8.3.1. The relatively big difference between estimate and measurements in this case stems from the fact that cache transfers are neglected in our performance model as mentioned in section 8.1. For a Gauss-Seidel 15-point stencil (GS 15-point) running on one single core, the 30 load instructions for stencil coefficients and unknowns (see table 8.1) limit the performance to \( n_{cycles,3} = 30 \cdot (\nu + 1) \) cycles per grid point for the residual and \( \nu \) smoothing steps.

In case of red-black Gauss-Seidel (RB 7-point) iterations, the limiting factor is the instruction throughput on one core (figure 8.4). On one compute node the limit is main memory bandwidth, what can be explained as follows: By using spatial blocking techniques the cache is large enough to store three grid planes of unknowns and one of the right hand side. Thus, one has to load one unknown value, one right hand side value, and store one unknown value per grid point. With the peak bandwidth of 64 GB/s on one lima node theoretically

\[
\frac{1}{8} \frac{64 \text{ GB/s}}{8 \text{ Bytes} \cdot 3} \approx 222 \text{ MUnknowns/s}. \quad (8.32)
\]

can be achieved per core. In practice we measured for stream triad benchmarks between 10-15 GB/s on a single core and about 18-21 GB/s on all six cores of a lima socket. For larger data sizes that do not fit into cache any more our measured MUnknowns/s for RB 7-point correspond to a bandwidth of about 6 GB/s on a single core in figure 8.4 and 18 GB/s on all six cores in figure 8.5. The performance drop at 50 points comes from the data not fitting in the L3 cache any more. Six compute cores share a L3 cache of 12 MB. The memory requirement for unknowns and right hand side for \( 50^3 \) grid points per core is thus

\[
6 \cdot 50^3 \cdot 8 \text{ Bytes} \cdot 2 = 12 \text{ MB}.
\]
Figure 8.4: Single compute core performance in MUnknowns/s for different smoother variants on lima. The size specifies the number of grid points in each dimension [4].

Figure 8.5: Single compute node (two sockets, without communication between the cores) performance in MUnknowns/s per core for different smoother variants on lima. The size specifies the number of grid points in each dimension [4].
Figure 8.6: Single (half) compute node (two sockets, without communication between the cores) performance in MUnknowns/s per core for different smoother variants on lima. The size specifies the number of grid points in each dimension.

The comparison in figure 8.6 to a 27-point stencil confirms that the 15-point stencil is instruction throughput limited on that machine. Both consume the same amount of bandwidth, but the 27-point stencil consumes a factor of about 1.8 more arithmetic operations. This is in good agreement with the ratio of the performance ($\approx \frac{95 \text{MUnknowns}}{55 \text{MUnknowns}} \approx 1.7$) of both stencils. This is not longer the case between the 15-point stencil and the 7-point stencil. Here, the ratios are 2.2 and $1 - 1.2$ for larger loop sizes accordingly. The figure shows that using only half of the cores on each node, and therefore doubling the effective bandwidth, the performance per core nearly doubles. For this stencil, blocking or a shared memory approach is necessary to improve the performance. However, for the 15-point stencil we are in the interesting situation that instruction throughput as well as main memory bandwidth are both influencing factors.

### 8.4.2 High Dynamic Range Compression

High dynamic range (HDR) compression tries to allow a wide dynamic range of luminance between the lightest and darkest areas within an image. Although doing HDR compression in the gradient space [125] is beyond the scope of this dissertation, we use it as a motivating application for the strong scaling performance studies of our MG solver. However, in order to motivate the application figure 8.7 provides slices of a 3D MRI image of a human head (see figure 8.8) with different dynamic gray value ranges. The HDR compressed image exhibits an improved contrast.
In general, one has to transform an input image $I : \Omega \rightarrow \mathbb{R}$ defined in the domain $\Omega \subset \mathbb{R}^3$ to gradient space and back. While the forward transformation to gradient space is fast by using simple finite differences to obtain the image gradient $\nabla I = \begin{pmatrix} I_x \\ I_y \\ I_z \end{pmatrix}$, the backward transformation requires the solution of Poisson’s equation (2.11) typically assuming homogeneous Dirichlet boundary conditions. Here,

$$f = \text{div}(\Phi \nabla I),$$

where $\Phi \nabla I$ are compressed dynamic range image derivatives and $\Phi : \mathbb{R}^3 \rightarrow \mathbb{R}$ is a position-dependent attenuating function (see [125] for more details). The solution $u : \Omega \rightarrow \mathbb{R}$ is the HDR compressed image.

Figure 8.7: 2D image slices from MRI data set (see figure 8.8) with different gray value ranges. The left slice (ground truth) has a gray value contrast 920:1, the middle slice a high dynamic range contrast 2800:1, and the right slice is the resulting HDR compressed image with contrast 80:1 [4].

Most of the overall runtime for HDR compression is spent in the numerical solution of equation (2.11), therefore we restrict ourselves to this part in the following. As example image an MRI data set (see figure 8.8) is used. Often, HDR compression is only one step within the image acquisition pipeline and there are hard time constraints that have to be met in practical applications.

### 8.4.3 MRI Voxel Geometry

As a test case we consider 3D MRI head scans. In order to save memory the head geometry is extracted from the fully structured 3D voxel grids and approximated with an unstructured tetrahedral grid (see figure 8.8). Therefore, the cubic voxel grid is divided into a number of equally sized cubic blocks consisting of six tetrahedral elements. Then, we remove all blocks that do not contain at least one voxel belonging to the head.
Table 8.2: Grid coverage for the MRI data set in figure 8.8 (scaled to size 1024x1024x576) with increasing number of blocks. [4]

<table>
<thead>
<tr>
<th>No. of blocks</th>
<th>Grid points per block</th>
<th>Grid coverage</th>
<th>Input volumes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4^3$</td>
<td>$257^3$</td>
<td>83%</td>
<td>180</td>
</tr>
<tr>
<td>$8^3$</td>
<td>$129^3$</td>
<td>55%</td>
<td>954</td>
</tr>
<tr>
<td>$16^3$</td>
<td>$65^3$</td>
<td>41%</td>
<td>5 646</td>
</tr>
<tr>
<td>$32^3$</td>
<td>$33^3$</td>
<td>33%</td>
<td>371 58</td>
</tr>
</tbody>
</table>

The resulting input grid is refined until the resolution of the original voxel grid is reached, i.e., one voxel contains six tetrahedral elements. Table 8.2 gives the grid coverage, i.e. the ratio of the number of grid points in the tetrahedral input grid and the original voxel grid, for different block sizes.

(a) Volume rendering from 3D-MPRAGE MRI slices (size 256x256x144) of a patient undergoing presurgical evaluation for epilepsy surgery.

(b) Twice refined tetrahedral finite element grid. The input grid consists of 5 646 tetrahedral volumes.

Figure 8.8: MRI data provided by Dr. S. Rampp from Neurologische Klinik Erlangen and extracted HHG input grid [4].

### 8.4.4 Full Performance Model

In medical applications it is important to achieve close to real time performance for moderate problem sizes. In addition to that we want to find the optimal number of nodes to use on a given cluster in order to minimize the overall runtime. At best this choice can be done based on our performance model. To show this, strong scaling experiments are
Table 8.3: Time per V-cycle (in seconds) and total time to solution for different number of smoothing steps and problem size 2048x2048x1152.

<table>
<thead>
<tr>
<th>Cycle type</th>
<th>No. of cycles</th>
<th>Time per cycle</th>
<th>Time to solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>V(1,1)</td>
<td>11</td>
<td>0.079</td>
<td>0.87</td>
</tr>
<tr>
<td>V(2,1)</td>
<td>8</td>
<td>0.097</td>
<td>0.77</td>
</tr>
<tr>
<td>V(2,2)</td>
<td>6</td>
<td>0.11</td>
<td>0.66</td>
</tr>
<tr>
<td>V(3,2)</td>
<td>6</td>
<td>0.13</td>
<td>0.78</td>
</tr>
<tr>
<td>V(3,3)</td>
<td>5</td>
<td>0.15</td>
<td>0.75</td>
</tr>
</tbody>
</table>

performed with the input grid generated from the MRI data set (see figure 8.8) and the settings:

- scaled up MRI data set to resolutions of 512x512x288, 1024x1024x576, and 2048x2048x1152
- The tetrahedral input grids are created according to section 7.3,
- doing V-cycles until the initial residual is reduced by six orders of magnitude

On JUGENE four grid levels are used for 512x512x288 and 1024x1024x576, and five grid levels for 2048x2048x1152. On lima five grid levels are taken for size 512x512x288, six grid levels for sizes 1024x1024x576 and 2048x2048x1152.

Depending on the grid, four to six smoothing steps are optimal concerning time to solution. For most setups six V(2,2)-cycles are enough to solve the linear system up to the prescribed accuracy. For all three resolutions, various numbers of compute cores and smoothing steps were tested. In table 8.3 the overall time to solve the biggest problem with a varying number of smoothing steps is listed. Figure 8.9 depicts the optimal settings and runtime for the three different grid sizes on both clusters.

It is interesting to compare our experimental results to the theoretical predictions for the optimal number of cores. As examples the problem size 1024x1024x576 for JUGENE and size 512x512x288 for lima are considered. Depending on the number of cores different parts of the code dominate the runtime (see figure 8.10 for JUGENE and figure 8.11 for lima). Direct communication latency for JUGENE is predicted very small and therefore does not appear in the figure. Here for large number of cores, the scalar products in the CG algorithms become time consuming. In contrast the latency is dictated by direct communication latency for lima. The model becomes more accurate, if communication is dominating. There is mainly the discrepancy between model and measurements of the single node performance from section 8.4.1 for both clusters. The model predicts a minimal runtime for around 37158 cores what matches quite well to our results in figure 8.9 and one can see that for a very large number of cores the coarsest grid influences the overall runtime most. For lima 5646 seems to be the best choice, which differs from figure 8.9. The times per V-cycle vary especially for large runs on this cluster. For the prediction, the best V-cycle concerning time
Figure 8.9: Average runtime and settings for HHG on JUGENE and lima for one V(2,2)-
cycle in the best setups found by our experiments using different core numbers [4].

Figure 8.10: Prediction of the runtime distribution from our performance model for JU-
GENE (see section 8.2) compared to the measured performance for an increasing number of compute cores (problem size 1024x1024x576) [4].
Figure 8.11: Prediction of the runtime distribution from our performance model for lima (see section 8.3) compared to the measured performance for an increasing number of compute cores (problem size 512x512x288) [4].

is taken, while for figure 8.9 an average over six cycles is taken. The last two measurements for both machines are quite similar concerning time to solution.

8.5 Operator Run-time Comparison with TERRA

Finally, we compare the HHG performance with TERRA, which is a package for simulating the convection in the mantle of Earth or other planetary bodies. It is well-known in the Geodynamics community and together with CitcomS, see [126, 127], one of the two packages most commonly employed for this purpose worldwide. TERRA was originally developed in the mid-1980ies by J. Baumgardner, see [106, 107]. It was later parallelised, first with PVM and then with MPI, see e.g. [128, 129], and extended in various ways, e.g. for allowing strong viscosity variations, see [130, 131]. Apart from some IO-routines, TERRA is written in a mix of classical and modern Fortran.

At the numerical core of TERRA is the solution of the Stokes problem given by equations (2.20) together with appropriate boundary conditions for the velocity field. In TERRA this problem is posed on a thick spherical shell representing Earth’s mantle. This domain is discretised with an icosahedral mesh as presented in section 7.5. The differential equations are discretised by nodal first order finite elements. The resulting linear system of equations is then solved by a pressure-correction scheme, similarly to it is now done within HHG.
This involves an outer CG iteration on (basically) the Schur complement of the system. In each CG iteration step a (conceptual)\(^2\) matrix-vector product must be computed where the matrix involves the inverse of the matrix in the FE-representation of the elliptic operator \(\nabla \cdot (2\mu s(u))\) from equation (2.22). The latter is performed by solving the associated linear system with a MG iteration. For further technical details see e.g. [132, 131]. Performance analysis has shown that the application of the operator \(\nabla \cdot (2\mu s(u))\) to a 3D vector field, occurring e.g. as part of the smoothing step, is the most time-consuming component of TERRA.

Following, we compare the performance of HHG and TERRA with respect to the application of an algorithmic application operator \(\nabla \cdot (2\mu s(u))\). Algebraically, this is equivalent to a matrix vector product with the stiffness matrix of the FE discretization of \(\nabla \cdot (2\mu s(u))\). For comparison we use two different computer architectures. First, we measure the run-times on BorgCube (cluster A.1) of the Geophysics section of LMU. This represents a smaller cluster as it is typically used in the GeoComputing community today. It consists of four HP Proliant DL785 G5 machines each equipped with eight AMD QuadCore Opteron 8384 (Shanghai) CPUs.

HHG and TERRA use different approaches to deal with data dependencies across sub-domain boundaries. While HHG employs a node-centered decomposition in combination with ghost nodes, TERRA uses an element-centered decomposition in combination with additive and overlapping storage schemes. For details see e.g. [133]. In both cases, communication is required after the application of the operator. In HHG, ghost node values must be updated between neighboring processors, and in TERRA a conversion from additive to overlapping storage needs to take place. Our timings include these data exchanges.

Both programs do not store the operator \(\nabla \cdot (2\mu s(u))\) in a matrix-based format. For HHG there is a data structure \textit{HHG-constant} for representing the operator when the coefficients are constant and thus the stiffness matrix contains the same entries in each row. Here, each inner point of a macro-element is directly connected to 14 other nodes (figure 2.2 middle). Thus we can assemble the matrix in terms of a single 15-point stencil (representing a row of the stiffness matrix) per macro-element in the setup phase, provided that the viscosity is constant in each macro-element.

More generally, when the viscosity is variable (as is usually the case in earth mantle simulations), then both TERRA and in HHG the \textit{HHG-variable} data structure provide mechanisms to assemble the corresponding entries of the stiffness matrix on-the-fly whenever the operator is applied for a grid point. This makes the application of the operator (i.e. the matrix-vector-multiplication) computationally significantly more expensive in terms of computational cost. However, compared to the usual assembly of the stiffness matrix it saves a significant amount of memory and, depending on the computer architecture, may

\(^2\)By conceptual we want to indicate that TERRA employs a grid-based, matrix-free implementation and does not setup the problem in matrix–vector form with corresponding data objects.
even be faster when memory bandwidth is so limited that recomputing the coefficients is cheaper than loading them from memory.

Figure 8.12 illustrates the run-times for a problem size of $8.3 \cdot 10^7$ grid points. This mesh has a minimal resolution of 30km on the Earth surface and 22.6km in radial direction for realistic Rayleigh numbers of $10^9$.

Here, TERRA and HHG-variable require a comparable number of of 595 and 678 floating point operations per grid point, respectively. In the case of constant viscosity, HHG-constant is computationally much cheaper with only 87 flops per grid point. However, in routine for the HHG-constant operator, most time for such a small grid is consumed by communication. Additionally, in the strong scaling scenario, the communication to computation ratio on each process grows when increasing number of processes. Therefore the time decreases only weakly when using more processors. Note however, that in terms of absolute time, this routine is significantly faster than the variable-coefficient versions. The two more compute intensive kernels scale better due to a better computation to communication ratio. Here we find that the HHG operator with on-the-fly assembly is still faster by roughly a factor of two than the TERRA operator implementation. Additionally, HHG-constant demonstrates the significant performance potential, if a prismatic discretisation using piecewise constant stencils can be developed. Even when the viscosity is physically variable this can possibly be exploited in a defect correction approach or in a MG method on the coarser grids.
Conclusion and Future Work
9 Conclusion and Future Work

After an introduction of software and numerical methods, asymptotic MG convergence rate predictions by local Fourier analysis were compared to measurements from HHG for point-wise, line-wise, and plane-wise smoothers for regular and degenerated elements with a deviation of below 10%. The local Fourier analysis predictions stem from a two-grid analysis for the Laplace operator for the tetrahedral 15-point stencil of HHG. Within the investigated smoothers, the four-color smoother was the best choice for regular-like tetrahedra. An optimal choice of individual damping parameters for the different colors has a stronger impact to improve the convergence factor than it is the case for a lexicographic Gauss–Seidel. Line-smoothing shows an especially good convergence for some tetrahedral shapes, e.g. for the wedge type. Plane-smoothing turned out to be very efficient for needle or spindle shaped elements. For the multi-relaxation schemes, zebra-wise updating proved to be more effective than a lexicographic updating.

Because of the refinement strategy in HHG, child elements of a tetrahedron have the same anisotropy as the parent element. Thus, it can be worthwhile doing some calculations in order to find out good smoothing parameters, since these parameters are applied to a huge number of tetrahedra on the finer grids. Following, we focused on the number of pre-/post-smoothing steps and local adaptively chosen damping parameters. It turned out that by redistributing the number of smoothing steps and applying different local damping parameters over a semi-structured domain can significantly reduce the convergence.

It was shown how the HHG concept can be extended to solve problems in geophysics, molecular dynamics, or chemical etching. To accomplish these tasks, we introduced methods such as variable coefficient diffusion operators, linear saddle point solver, or $\tau$-extrapolation for variable coefficient problems. Although clearly, more sophisticated physics has to be included, the framework has gained several new features for further research in those directions.

In the third part, extensions such as the treatment of large semi-structured meshes and hybrid parallelization for HHG are presented to enable an ultra-scalable MG method for beyond one million threads. Weak and strong scalability are proven on the three different petaflop clusters JUGENE, JUQUEEN, and SuperMUC. Driven by geophysics a tetrahedral decomposition of a thick sphere was developed, which is based on an icosahedral mesh. First, I investigated the scalability of a diffusion problem on this geometry. This is then the core component to solve large Stokes systems to resolve the Earth mantle uniformly with 1 km
resolution. For one time-step a linear saddle point system with $10^{12}$ unknowns could be solved in less than one minute.

A predictive performance and communication model has been designed for the approach that accurately estimates the parallel runtime. Besides memory bandwidth, instruction throughput can easily become the bottleneck for stencil-based code depending on the architecture and problem size. Depending on the problem size per core, network bandwidth (predominantly on the finer levels) as well as latency for the coarsest level limit the performance of the semi-structured MG approach.

Now the code is well prepared to explore different highly parallel MG strategies. Within the next year, HHG has a central place in one of two test-beds for the upcoming Terra-Neo framework. The plan within this project is to investigate different software and numerical issues, e.g.:

- Stencil-based MG correction for strongly varying coefficients,
- Communication avoiding asynchronous MG,
- MG check pointing strategies, and
- Performance modeling and prediction of different stencil-based discretization approaches.

The latter could derive predictions for other FE discretization within an HHG approach to narrow down the number of choices before performing an actual implementation. One major goal for the next-generation framework is to develop further improvements for upcoming hardware, but still deliver spatial resolutions in the scale of 1 km and time-steps below one minute.
Reviewed Publications


Selected Other Publications


Selected Presentations

B. Gmeiner and U. Rüde. PetaScale Finite Element Methods for Hierarchical Hybrid Grids using Hybrid Parallelization, *16th Copper Mountain Conference on Multigrid Methods, Copper Mountain, Colorado, USA, 19.03.2013*


B. Gmeiner and U. Rüde. Multigrid for Finite Elements on 290,000 Compute Cores: Scalability Experiments, *20th International Conference on Domain Decomposition Methods, UC San Diego, La Jolla, California, USA, 11.02.2011*


B. Gmeiner, H. Köstler and U. Rüde. Towards real-time image processing with Hierarchical Hybrid Grids, *IDK Summer School, Pommersfelden, 03.08.2011*


B. Gmeiner, T. Gradl, F. Gaspar and U. Rüde. Validation and Optimization of the Convergence Rate on Semi-structured Meshes Using the LFA, *Eleventh Copper Mountain Conference on Iterative Methods, Copper Mountain, USA, 08.04.2010*
Part IV

Appendix
A Cluster Specifications

A.1 BorgCube

HP Proliant DL785 G5

- Processor: AMD QuadCore Opteron 8384
- Clock Frequency: 2.7 GHz
- Number of Nodes: 4
- DP Performance per Node: 345 GFLOP/s
- Cores per Node: 8x4
- Hardware-Threads per Core: 1
- Main Memory per Hardware-Thread: 8 GB
- Network Topology: Tree
- Location: Geophysics section of LMU Munich, Germany

A.2 JUGENE

IBM BlueGene/P

- Processor: IBM PowerPC 450
- Clock Frequency: 0.85 GHz
- Number of Nodes: 73 728
- DP Performance per Node: 13.6 GFLOP/s
- Cores per Node: 4
- Hardware-Threads per Core: 1
- Main Memory per Hardware-Thread: 0.5 GB
- Network Topology: 3D Torus
- Location: Forschungszentrum Jülich (FZJ), Germany
A.3 JUQUEEN

IBM BlueGene/Q
- Processor: IBM PowerPC A2
- Clock Frequency: 1.6 GHz
- Number of Nodes: 8192
- DP Performance per Node: 204.8 GFLOP/s
- Cores per Node: 16
- Hardware-Threads per Core: 4
- Main Memory per Hardware-Thread: 0.25 GB
- Network Topology: 5D Torus
- Location: Forschungszentrum Jülich (FZJ), Germany

A.4 lima

NEC LX-2400
- Processor: Intel Xeon X5650
- Clock Frequency: 2.7 GHz
- Number of Nodes: 500
- DP Performance per Node: 128 GFLOP/s
- Cores per Node: 2x6
- Hardware-Threads per Core: 2
- Main Memory per Hardware-Thread: 1 GB
- Network Topology: Tree
- Location: Regionales Rechenzentrum Erlangen (RRZE), Germany
A.5 SuperMUC

IBM System x iDataPlex
- Processor: Intel Xeon E5-2680 8C
- Clock Frequency: 2.8 GHz
- Number of Nodes: 9,216
- DP Performance per Node: 345.6 GFLOP/s
- Cores per Node: 2x8
- Hardware-Threads per Core: 2
- Main Memory per Hardware-Thread: 1 GB
- Network Topology: Tree
- Location: Leibniz-Rechenzentrum (LRZ) Munich, Germany
Erdmantelkonvektion ist die treibende Kraft hinter Plattentektonik, Kontinentalverschiebung, Vulkanismus, Bergformationen, Ozeane und der Erzeugung von Erdbeben. Weil das Innere des Planeten physikalisch unzugänglich ist, verbleiben Beobachtungen und Simulationen der einzige Weg um Kenntnisse über die ablaufende Physik zu erlangen. Der Mantel verhält sich hierbei wie ein extrem zähes Fluid (Stokes Strömung), der mittels eines Wärmeabflusses durch Thermische Advektion und Abkühlung über lange Zeitskalen angetrieben wird.


C Extended Code Example

Listing C.1: Macro-element stencil assembling for one grid point

```c
#define PLAINSIZE(size) (((size) * ((size) + 1))/2)

// lex. Gauss-Seidel variable coefficient
// smoother for tetrahedra
const int tet_mcq = 0;
const int tet_me = 1;
const int tet_mnw = 2;
const int tet_mn = 3;
const int tet_ts = 4;
const int tet_tse = 5;
const int tet_tw = 6;
const int tet_tc = 7;
const int tet_bc = 8;
const int tet_be = 9;
const int tet_bnv = 10;
const int tet_bn = 11;
const int tet_ms = 12;
const int tet_mse = 13;
const int tet_mw = 14;
const int tet_mc = 15;

void tet_gs_coeff(double *u, double *f, double *koe, double *stiff, int *p_tsize) {
  int tsize = p_tsize[0];
  int mp, tp, bp;
  int mp_mr, mp_tr, mp_br;
  int tp_mr, tp_br;
  int bp_mr, bp_tr;
  bp = 0;
  mp = PLAINSIZE(tsize);
  tp = mp + PLAINSIZE(tsize - 1);
  double stencil[16];
  double c[6*4*4];
  double k_tw_tc, k_tc_tse, k_tc_mn, k_ts_mc, k_tse_mse, k_tw_mw,
         k_mnw_mc, k_mc_me, k_ms_mse, k_mw_ms;
  double k_mc_bc, k_nn_bn, k_mse_be, k_mw_bnv, k_bnv_bn, k_bn_be;
  double k_el;
  for (int i=0; i<6*4*4; ++i) {
    c[i] = 0.25 * stiff[i];
  }
  for (int k=1; k < (tsize - 3); ++k) {
    bp_mr = bp + tsize-k+1;
    bp_tr = bp + 2*(tsize-k+1) - 1;

```
mp_br = mp;
mp_mr = mp + tsize - k;
mp_tr = mp + 2*(tsize - k) - 1;

tp_br = tp;
tp_mr = tp + tsize - k - 1;

for (int j=1; j < (tsize-k-2); ++j) {
    for (int i=1; i < (tsize-j-k-1); i=i+1) {
        k_tw_tc = koe[tp_mr+i-1] + koe[tp_mr+i];
        k_tc_tse = koe[tp_mr+i] + koe[tp_br+i+1];
        k_ts_nc = koe[tp_br+i] + koe[mp_tr+i];
        k_tc_mn = koe[tp_mr+i] + koe[mp_tr+i];
        k_ms_mse = koe[tp_br+i] + koe[mp_br+i+1];
        k_ts_mse = koe[tp_br+i+1] + koe[mp_br+i+1];
        k_tw_mw = koe[tp_mr+i-1] + koe[mp_mr+i];
        k_mnw_mc = koe[mp_tr+i-1] + koe[mp_mr+i];
        k_mc_ne = koe[mp_mr+i] + koe[mp_mr+i];
        k_ms_ne = koe[mp_br+i] + koe[mp_br+i+1];
        k_mw_ms = koe[mp_mr+i-1] + koe[mp_br+i];
        k_mc_bc = koe[mp_mr+i] + koe[bp_mr+i];
        k_mn_bn = koe[mp_tr+i-1] + koe[bp_tr+i];
        k_mse_be = koe[mp_br+i+1] + koe[bp_mr+i+1];
        k_mw_bnw = koe[mp_mr+i -1] + koe[bp_tr+i -1];
        k_bnw_bn = koe[bp_tr+i -1] + koe[bp_tr+i];
        k_bn_be = koe[bp_tr+i] + koe[bp_mr+i+1];
    }

    // ///////////////////////////////////////
    // group zero element
    // ///////////////////////////////////////
    k_el = k_tc_mn + k_mc_ne;
    // position zero
    stencil[tet_mc] = k_el*c[0*4*4+0*4+0];
    stencil[tet_me] = k_el*c[0*4*4+0*4+1];
    stencil[tet_mn] = k_el*c[0*4*4+0*4+2];
    stencil[tet_tc] = k_el*c[0*4*4+0*4+3];
    k_el = k_mnw_mc + k_tw_mw;
    // position one
    stencil[tet_mw] = k_el*c[0*4*4+1*4+0];
    stencil[tet_mc] = k_el*c[0*4*4+1*4+1];
    stencil[tet_mnw] = k_el*c[0*4*4+1*4+2];
    stencil[tet_tw] = k_el*c[0*4*4+1*4+3];
    k_el = k_ts_nc + k_ms_mse;
    // position two
    stencil[tet_ms] = k_el*c[0*4*4+2*4+0];
    stencil[tet_tse] = k_el*c[0*4*4+2*4+1];
    stencil[tet_tnc] = k_el*c[0*4*4+2*4+2];
    stencil[tet_ts] = k_el*c[0*4*4+2*4+3];
    k_el = k_mc_bc + k_bn_be;
    // position three
    stencil[tet_bc] = k_el*c[0*4*4+3*4+0];
    stencil[tet_be] = k_el*c[0*4*4+3*4+1];
    stencil[tet_bn] = k_el*c[0*4*4+3*4+2];
    stencil[tet_tnc] = k_el*c[0*4*4+3*4+3];

    // ///////////////////////////////////////
    // group one element
    // ///////////////////////////////////////
    k_el = k_tw_tc + k_ts_mse;
    // position zero
stencil[tet_mc] += k_el * c[1*4*4+0*4+0];

stencil[tet_ts] += k_el * c[1*4*4+0*4+1];

stencil[tet_tw] += k_el * c[1*4*4+0*4+2];

stencil[tet_tc] += k_el * c[1*4*4+0*4+3];

k_el = k_mnw_mc + k_mn_bn;

// position one

stencil[tet_bn] += k_el * c[1*4*4+1*4+0];

stencil[tet_mc] += k_el * c[1*4*4+1*4+1];

stencil[tet_mnw] += k_el * c[1*4*4+1*4+2];

stencil[tet_mn] += k_el * c[1*4*4+1*4+3];

k_el = k_mc_me + k_mse_be;

// position two

stencil[tet_be] += k_el * c[1*4*4+2*4+0];

stencil[tet_mse] += k_el * c[1*4*4+2*4+1];

stencil[tet_mc] += k_el * c[1*4*4+2*4+2];

stencil[tet_me] += k_el * c[1*4*4+2*4+3];

k_el = k_mw_ms + k_mc_bc;

// position three

stencil[tet_bc] += k_el * c[1*4*4+3*4+0];

stencil[tet_ms] += k_el * c[1*4*4+3*4+1];

stencil[tet_mw] += k_el * c[1*4*4+3*4+2];

stencil[tet_mc] += k_el * c[1*4*4+3*4+3];

// ////////////////////////////////////////////////////////////////////////////////////
// group two element
// ////////////////////////////////////////////////////////////////////////////////////

k_el = k_tw_tc + k_mnw_mc;

// position zero

stencil[tet_mc] += k_el * c[2*4*4+0*4+0];

stencil[tet_mnw] += k_el * c[2*4*4+0*4+1];

stencil[tet_tw] += k_el * c[2*4*4+0*4+2];

stencil[tet_tc] += k_el * c[2*4*4+0*4+3];

k_el = k_ts_mc + k_tse_mse;

// position one

stencil[tet_mse] += k_el * c[2*4*4+1*4+0];

stencil[tet_mc] += k_el * c[2*4*4+1*4+1];

stencil[tet_ts] += k_el * c[2*4*4+1*4+2];

stencil[tet_tse] = k_el * c[2*4*4+1*4+3];

k_el = k_mc_me + k_bn_be;

// position two

stencil[tet_be] += k_el * c[2*4*4+2*4+0];

stencil[tet_bn] += k_el * c[2*4*4+2*4+1];

stencil[tet_mc] += k_el * c[2*4*4+2*4+2];

stencil[tet_me] += k_el * c[2*4*4+2*4+3];

k_el = k_mc_bc + k_mw_bnw;

// position three

stencil[tet_bc] += k_el * c[2*4*4+3*4+0];

stencil[tet_bnw] = k_el * c[2*4*4+3*4+1];

stencil[tet_mw] += k_el * c[2*4*4+3*4+2];

stencil[tet_mc] += k_el * c[2*4*4+3*4+3];

// ////////////////////////////////////////////////////////////////////////////////////
// group three element
// ////////////////////////////////////////////////////////////////////////////////////

k_el = k_tc_tse + k_mc_me;

// position zero

stencil[tet_mc] += k_el * c[3*4*4+0*4+0];
stencil[tet_ne] += k_el*c[3*4*4+0*4+1];
stencil[tet_tse] += k_el*c[3*4*4+0*4+2];
stencil[tet_tc] += k_el*c[3*4*4+0*4+3];

k_el = k_tw_nw + k_ts_mc;
// position one
stencil[tet_nw] += k_el*c[3*4*4+1*4+0];
stencil[tet_mc] += k_el*c[3*4*4+1*4+1];
stencil[tet_ts] += k_el*c[3*4*4+1*4+2];
stencil[tet_tv] += k_el*c[3*4*4+1*4+3];

k_el = k_mnw_mc + k_bnw_bn;
// position two
stencil[tet_bnw] += k_el*c[3*4*4+2*4+0];
stencil[tet_bn] += k_el*c[3*4*4+2*4+1];
stencil[tet_mc] += k_el*c[3*4*4+2*4+2];
stencil[tet_mnw] += k_el*c[3*4*4+2*4+3];

k_el = k_mse_be + k_mc_bc;
// position three
stencil[tet_bc] += k_el*c[3*4*4+3*4+0];
stencil[tet_be] += k_el*c[3*4*4+3*4+1];
stencil[tet_mse] += k_el*c[3*4*4+3*4+2];
stencil[tet_mc] += k_el*c[3*4*4+3*4+3];

//////////////////////////////////////////////////////////////////////////////////////
// group four element
////////////////////////////////////////////////////////////////////////////////////////

k_el = k_tc_tse + k_ts_mc;
// position zero
stencil[tet_mc] += k_el*c[4*4*4+0*4+0];
stencil[tet_ts] += k_el*c[4*4*4+0*4+1];
stencil[tet_tse] += k_el*c[4*4*4+0*4+2];
stencil[tet_tc] += k_el*c[4*4*4+0*4+3];

k_el = k_mc_ne + k_mn_bn;
// position one
stencil[tet_bn] += k_el*c[4*4*4+1*4+0];
stencil[tet_mc] += k_el*c[4*4*4+1*4+1];
stencil[tet_ne] += k_el*c[4*4*4+1*4+2];
stencil[tet_mn] += k_el*c[4*4*4+1*4+3];

k_el = k_mnw_mc + k_mw_bnw;
// position two
stencil[tet_bnw] += k_el*c[4*4*4+2*4+0];
stencil[tet_nw] += k_el*c[4*4*4+2*4+1];
stencil[tet_mc] += k_el*c[4*4*4+2*4+2];
stencil[tet_mnw] += k_el*c[4*4*4+2*4+3];

k_el = k_mse_mse + k_mc_bc;
// position three
stencil[tet_bc] += k_el*c[4*4*4+3*4+0];
stencil[tet_mse] += k_el*c[4*4*4+3*4+1];
stencil[tet_mse] += k_el*c[4*4*4+3*4+2];
stencil[tet_mc] += k_el*c[4*4*4+3*4+3];

//////////////////////////////////////////////////////////////////////////////////////
// group five element
////////////////////////////////////////////////////////////////////////////////////////

k_el = k_tc_mn + k_mnw_mc;
// position zero
stencil[tet_mc] += k_el*c[5*4*4+0*4+0];
stencil[tet_mn] += k_el*c[5*4*4+0*4+1];
stencil[tet_mn] += k_el *c[5*4*4+0*4+2];
stencil[tet_tc] += k_el *c[5*4*4+0*4+3];

k_el = k_tse_mse + k_mc_me;

// position one
stencil[tet_mse] += k_el *c[5*4*4+1*4+0];
stencil[tet_mc] += k_el *c[5*4*4+1*4+1];
stencil[tet_me] += k_el *c[5*4*4+1*4+2];
stencil[tet_tse] += k_el *c[5*4*4+1*4+3];

k_el = k_ts_mc + k_mw_ms;

// position two
stencil[tet_ms] += k_el *c[5*4*4+2*4+0];
stencil[tet_mw] += k_el *c[5*4*4+2*4+1];
stencil[tet_mc] += k_el *c[5*4*4+2*4+2];
stencil[tet_ts] += k_el *c[5*4*4+2*4+3];

k_el = k_mc_bc + k_bnw_bn;

// position three
stencil[tet_bc] += k_el *c[5*4*4+3*4+0];
stencil[tet_bnw] += k_el *c[5*4*4+3*4+1];
stencil[tet_bn] += k_el *c[5*4*4+3*4+2];
stencil[tet_mc] += k_el *c[5*4*4+3*4+3];

// pre-compute center weight quotient
stencil[tet_mcq] = 1.0/stencil[tet_mc];

u[mp_mr+i] = stencil[0] * (f[mp_mr+i] -

bp_mr = bp_tr;
bp_tr = bp_tr + tsize-j-k;

mp_br = mp_mr;
mp_mr = mp_tr;
mp_tr = mp_tr + tsize-j-k-1;

tp_br = tp_mr;
tp_mr = tp_mr + tsize-j-k-1;

bp = mp;
mp = tp;
tp = tp + PLAINSIZE(tsize-k-1);
Bibliography


