Solving partial differential equations (PDEs) is a fundamental challenge in many application domains in industry and academia alike. With increasingly large problems, efficient and highly scalable implementations become more and more crucial. Domain-specific languages (DSLs) and code generation techniques hold the power to automate the application of domain-specific optimizations as well as mapping to the increasingly heterogeneous hardware landscape of today. This work aims to further the state of the art in this field, in particular for PDE solvers based on geometric multigrid methods operating on (patch-)structured grids.

We begin by developing our multi-layered external DSL ExaSlang: Layer 1 is designed to resemble LaTeX and allows inputting continuous equations and functions. Their discretization is expressed on layer 2. It is complemented by algorithmic components implemented in a Matlab-like syntax on layer 3. All information provided to this point is summarized on layer 4, enriched with particulars about data structures and the employed parallelization. For convenience, we support automated progression between the different layers. All ExaSlang input is processed by our jointly developed Scala code generation framework to ultimately emit C++ code. Generated applications are automatically parallelized with MPI, OpenMP and CUDA to run on platforms ranging from workstations to large-scale clusters.

We showcase the applicability of our approach by implementing simple test problems, like Poisson's equation, as well as relevant applications from the field of computational fluid dynamics. In particular, we implement scalable solvers for the Stokes, Navier-Stokes and shallow water equations discretized with finite differences and finite volumes. We also extend our implementation towards non-uniform grids and advanced effects such as the simulated fluid being non-Newtonian and non-isothermal.
Sebastian Kuckuk

Automatic Code Generation for Massively Parallel Applications in Computational Fluid Dynamics
FAU Studien aus der Informatik

Band 10

Herausgeber der Reihe:
Björn Eskofier, Richard Lenz, Andreas Maier,
Michael Philippsen, Lutz Schröder,
Wolfgang Schröder-Preikschat, Marc Stamminger, Rolf Wanka
Automatic Code Generation for Massively Parallel Applications in Computational Fluid Dynamics

Automatische Codegenerierung für Massiv Parallele Applikationen in der Numerischen Strömungsmechanik

Der Technischen Fakultät
der Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur Erlangung des
Doktorgrades Dr.-Ing.
vorgelegt von
Sebastian Kuckuk
aus Berlin
Als Dissertation genehmigt
von der Technischen Fakultät
der Friedrich-Alexander-Universität Erlangen-Nürnberg

Tag der mündlichen Prüfung: 13.08.2019

Vorsitzender des Promotionsorgans: Prof. Dr. Reinhard Lerch

Gutachter: Prof. Dr. Harald Köstler
       Prof. Dr. Matthias Bolten
Abstract

Solving partial differential equations (PDEs) is a fundamental challenge in many application domains in industry and academia alike. With increasingly large problems, efficient and highly scalable implementations become more and more crucial. Today, facing this challenge is more difficult than ever due to the increasingly heterogeneous hardware landscape. One promising approach is developing domain-specific languages (DSLs) for a set of applications. Using code generation techniques then allows targeting a range of hardware platforms while concurrently applying domain-specific optimizations in an automated fashion. The present work aims to further the state of the art in this field. As domain, we choose PDE solvers and, in particular, those from the group of geometric multigrid methods. To avoid having a focus too broad, we restrict ourselves to methods working on structured and patch-structured grids.

We face the challenge of handling a domain as complex as ours, while providing different abstractions for diverse user groups, by splitting our external DSL ExaSlang into multiple layers, each specifying different aspects of the final application. Layer 1 is designed to resemble LaTeX and allows inputting continuous equations and functions. Their discretization is expressed on layer 2. It is complemented by algorithmic components which can be implemented in a Matlab-like syntax on layer 3. All information provided to this point is summarized on layer 4, enriched with particulars about data structures and the employed parallelization. Additionally, we support automated progression between the different layers. All ExaSlang input is processed by our jointly developed Scala code generation framework to ultimately emit C++ code. We particularly focus on how to generate applications parallelized with, e.g., MPI and OpenMP that are able to run on workstations and large-scale cluster alike.
We showcase the applicability of our approach by implementing simple test problems, like Poisson’s equation, as well as relevant applications from the field of computational fluid dynamics (CFD). In particular, we implement scalable solvers for the Stokes, Navier-Stokes and shallow water equations (SWE) discretized using finite differences (FD) and finite volumes (FV). For the case of Navier-Stokes, we also extend our implementation towards non-uniform grids, thereby enabling static mesh refinement, and advanced effects such as the simulated fluid being non-Newtonian and non-isothermal.
Zusammenfassung


Wir begegnen der Herausforderung eine Domäne so komplex wie die unsere zu behandeln, indem wir unsere externe DSL ExaSlang in mehrere Ebenen aufteilen. Dies ermöglicht unter anderem Abstraktionen für verschiedene Anwendergruppen zur Verfügung zu stellen. Ebene 1 ist so entworfen, dass sie LaTeX ähnelt und erlaubt die Eingabe von kontinuierlichen Gleichungen und Funktionen, deren Diskretisierung dann auf Ebene 2 beschrieben werden kann. Ebene 3 komplementiert dies mit algorithmischen Komponenten, die in einer Matlab-ähnlichen Syntax angegeben werden können. Alle bis hierhin gemachten Angaben werden dann auf Ebene 4 zusammengefasst und mit Details über Datenstrukturen und der zu verwendenden Parallelisierung angereichert. Darüber hinaus sind wir in der Lage zwischen den Ebenen automatisiert Infor-

Wir belegen die Anwendbarkeit unseres Ansatzes indem wir nicht nur einfache Testprobleme wie die Poisson-Gleichung implementieren, sondern auch komplexe Anwendungen aus dem Gebiet der numerischen Strömungsmechanik. Genauer gesagt betrachten wir skalierbare Löser für die Stokes, Navier-Stokes und Flachwassergleichungen, die mittels finiten Differenzen und Volumen diskretisiert werden. Im Falle von Navier-Stokes erweitern wir darüber hinaus unsere Implementierung sodass sie nicht-uniforme Gitter unterstützt wodurch eine statische Gitterverfeinerung ermöglicht wird. Parallel dazu betrachten wir fortgeschrittene physikalische Effekte wie z.B. dass die simulierte Flüssigkeit temperaturabhängig und nicht-Newtonisch ist.
Acknowledgements

First of all, I would like to thank my advisor Prof. Dr. Harald Köstler for his constant support and for introducing me to the fascinating world of domain-specific languages and code generation. I am particularly grateful for being given the freedom to always follow my own research ideas and collaborations. I would also like to thank Prof. Dr. Matthias Bolten for not only agreeing to examine my thesis, but also for always being available for all mathematical questions.

This thesis would not have been possible without generous funding by the DFG and the provision of computing resources by the RRZE, the JSC and the CSCS. Working in the scope of the ExaStencils project has been an invaluable experience. I am particularly indebted to the (former) PhD candidates Stefan, Lisa, Christian, Hannah, Alexander and Georg for interesting discussions and fruitful collaborations as well as to our project speaker Prof. Dr. Lengauer. I also thank all partners from former collaborations, in particular Prof. Dr. Haase and Dr. Vasco for sharing their expertise in the field of computational fluid dynamics. I am also grateful for the possibility to further develop our software in an ongoing collaboration with Prof. Dr. Aizinger and Dr. Grosso. Special thanks to my colleagues Daniel and Sara – it continues to be a joy working with you!

I would also like to thank everyone at the chair for providing an enjoyable environment, especially Christoph, Chris, both Dominiks, Julian and Sebastian. I particularly enjoyed our occasional after-hours tastings, our LSS cinema and our sports activities. A big thank you also to Iris and the whole office and administration team for always lending a helping hand.

Finally, I am indebted to my family and, in particular, my beautiful and loving wife Jasmin. Thank you for always believing in me – even when I did not.
# Contents

1 Scope and Introduction  

2 Background  
  2.1 Discretization of Partial Differential Equations  
  2.1.1 Finite Difference Method  
  2.1.2 Finite Volume Method  
  2.2 Numerical Solvers  
  2.2.1 Point-Based Solvers  
  2.2.2 Block Solvers  
  2.2.3 Krylov Subspace Solvers  
  2.2.4 Geometric Multigrid  
  2.3 Domain-Specific Languages  
  2.4 Code Generation  
  2.4.1 Framework  

3 The ExaStencils Language (ExaSlang)  
  3.1 Our Layered Approach  
  3.2 Required Concepts  
  3.3 Layers 1 to 4  
  3.3.1 Layer 1  
  3.3.2 Layer 2  
  3.3.3 Layer 3  
  3.3.4 Layer 4  
  3.4 Inter-Layer Hints  
  3.4.1 Discretization Hints  
  3.4.2 Solver Hints  
  3.4.3 Application Hints  
  3.5 Configuration Files  
  3.5.1 Platform
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.2</td>
<td>Settings</td>
<td>60</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Knowledge</td>
<td>61</td>
</tr>
<tr>
<td>3.6</td>
<td>Contribution</td>
<td>61</td>
</tr>
<tr>
<td>4</td>
<td>Code Generator Part I: Workflow</td>
<td>63</td>
</tr>
<tr>
<td>4.1</td>
<td>Data Structures and Transformations</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>Workflow</td>
<td>65</td>
</tr>
<tr>
<td>4.3</td>
<td>Transition Between Different Layers</td>
<td>66</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Layer 1 to Layer 2</td>
<td>67</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Layer 2 to Layer 3</td>
<td>69</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Layer 3 to Layer 4</td>
<td>82</td>
</tr>
<tr>
<td>4.4</td>
<td>The Intermediate Representation (IR)</td>
<td>83</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Example</td>
<td>84</td>
</tr>
<tr>
<td>4.5</td>
<td>Contribution</td>
<td>90</td>
</tr>
<tr>
<td>5</td>
<td>Code Generator Part II: Extensions</td>
<td>93</td>
</tr>
<tr>
<td>5.1</td>
<td>Parallelization of Generated Applications</td>
<td>93</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Data Partitioning</td>
<td>94</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Data Synchronization</td>
<td>99</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Automatic Insertion of Communication Statements</td>
<td>106</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Results</td>
<td>108</td>
</tr>
<tr>
<td>5.2</td>
<td>New Grid Types</td>
<td>109</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Staggered Grids</td>
<td>109</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Non-Uniform Grids</td>
<td>110</td>
</tr>
<tr>
<td>5.3</td>
<td>Finite Volume Discretizations</td>
<td>113</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Evaluate Functions</td>
<td>114</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Integrate Functions</td>
<td>116</td>
</tr>
<tr>
<td>5.4</td>
<td>Automatically Applied Optimizations</td>
<td>117</td>
</tr>
<tr>
<td>5.5</td>
<td>Contribution</td>
<td>118</td>
</tr>
<tr>
<td>6</td>
<td>Applications</td>
<td>121</td>
</tr>
<tr>
<td>6.1</td>
<td>Stokes Equations</td>
<td>122</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Governing Equations</td>
<td>122</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Discretization</td>
<td>123</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Solver</td>
<td>131</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Results</td>
<td>137</td>
</tr>
<tr>
<td>6.2</td>
<td>Navier-Stokes Equations</td>
<td>150</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Governing Equations</td>
<td>150</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Discretization</td>
<td>152</td>
</tr>
</tbody>
</table>
1 Scope and Introduction

In the field of scientific computing, one of the recurring patterns is solving partial differential equations (PDEs). For accurate and meaningful results, often a large number of unknowns has to be solved for. This usually relates to a considerable number of compute resources required. In some cases, only super computers can provide the required compute power. They are, however, quite costly to use. One example for this is JUQUEEN, a now decommissioned supercomputer which was located at the JSC in Jülich, Germany. It required around 1.7 MW on average¹, roughly the same as a small town with 4,000 inhabitants. It is thus crucial to use the machine to its fullest potential and, for that, highly efficient and massively parallel solvers with optimal scaling behavior are vital. This goal led to two different approaches which can be considered state of the art nowadays. The first option is hand-crafting a highly specialized implementation for a very narrow problem specification. It is usually hand-tuned towards a specific target hardware platform. The second option is relying on large scale scientific libraries for which examples are discussed in detail in section 8.1. Both approaches, however, frequently suffer from severe draw-backs. By nature, specialized codes are hard to extend for anything they were not designed for. Furthermore, they are often so-called legacy codes written in older languages such as FORTRAN and using software engineering techniques that date back decades. Modernization of these codes is not feasible most of the time since the effort required rivals or even surpasses that of a full reimplementation. In contrast to that, high performance computing (HPC) frameworks hold the benefit of supporting a wide range of application scenarios which adds flexibility for the user. Instead of optimizing for

¹ fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/Configuration/Configuration_node
specific applications, general optimizations can be implemented that benefit all users of the framework. Being agnostic of the actual application implemented, however, a number of domain-specific optimizations may not be possible. This leads to sub-optimal implementations. HPC frameworks can also be notoriously complex and, in consequence, difficult to use. Especially for new users, the learning curve is often very steep.

One emerging technology aiming at mitigating the draw-backs just detailed is using domain-specific languages (DSLs) in conjunctions with code generation techniques. DSLs are custom languages that allow users to specify their problems (and solutions) in a way familiar to them. They often avoid including implementation particulars as their goal is to allow users to specify what to do and not how to do it. This usually results in representations that are more abstract and more concise than those done in general purpose languages. Additionally, this gives the flexibility to alter the final implementation thereby allowing to apply automated optimizations. This step is usually realized in a code generator. Its job is to take the DSL input and to translate it either to another representation, e.g. an implementation in another programming language, or directly to machine code. We focus on the former since it, on the one hand, still allows verifying the output of the generator and, on the other hand, facilitates coupling to existing code modules.

The main aim of this work is developing such a DSL and code generation technology for PDE solvers. While covering as many application scenarios as possible seems desirable (similar to HPC frameworks), a much more narrow domain is favorable. This allows implementing a more tailored DSL and more targeted domain-specific optimizations.

PDE solvers can be categorized into two groups: grid-based methods and meshfree methods also including particle methods. In this work, we focus on grid-based methods since we feel that they are most relevant in the field. They can be further subdivided according to the types of grids they employ. Traditionally, these are either structured or unstructured. However, it is also possible to use a hybrid approach where structured patches are connected in an unstructured fashion. These less commonly used grids are usually referred to as block-structured. They aim at combining the flexibility of unstructured grids with the performance potential of structured ones. We focus on these types of grids as well. Naturally, (fully) structured grids are a specialization of block-structured grids and thereby included in our scope. Figure 1.1 il-
1 Scope and Introduction

(a) structured (uniform)  (b) block-structured  (c) unstructured  (d) meshfree (particles)

Figure 1.1: Illustration of different types of grids in (a) through (c) and meshfree approaches in (d).

(a) uniform  (b) non-uniform  (c) non-uniform axis-aligned

Figure 1.2: Illustration of different types of structured grids.

lustrates the different approaches: from mesh-less to fully structured grid methods. In our context, structured grids are not required to be uniform, as figure 1.2 underlines by showing different grids also targeted by us. Moreover, each of them can be used as building blocks in block-structured grids.

After choosing the grid, the next step is discretizing the given PDEs on them. The predominant methods for this are

• finite differences (FD), also called the finite difference method (FDM),
• finite elements (FE), also called the finite element method (FEM),
• finite volumes (FV), also called the finite volume method (FVM), and
• discontinuous Galerkin (DG) methods which can be interpreted as a combination of FE and FV.

We aim at supporting all of these methods, although this work mainly focuses on FD and FV. More details on them can be found in section 2.1.

Next, the discretized equations have to be solved for. We further restrict our domain by allowing only certain classes of solvers at this stage. Here, only algorithms are allowed that exhibit an access pattern with a fixed and small neighborhood. This includes basic iterative solvers...
of the Jacobi and (multi-colored) Gauss-Seidel variants as well as many Krylov subspace methods such as conjugate gradient (CG). More details on the latter are given in section 4.3.2. These solvers often suffer from a less than optimal complexity, however. On possible alternative fixing this issue is given by the class of multigrid methods. A detailed introduction to them is given in section 2.2.4 as well as in [67, 100]. We deem using a suitable solver algorithm crucial for HPC applications which is why this work focuses on (geometric) multigrid solvers. That is, we specialize our language to naturally support grid hierarchies and functions operating on them. Of course, it is still possible to use only a single level and, in consequence, the other types of solvers referred to before.

In summary, the main aim of our work is to
1. design a DSL for geometric multigrid solvers on (block-)structured grids,
2. implement a code generator taking DSL specifications as input that is capable to
3. emit optimized and highly scalable implementations and
4. show real-world applicability in the field of computational fluid dynamics (CFD).

Geometric multigrid solvers, at least on the types of grids supported here, usually fall into the category of stencil codes. In consequence, our language and code generation framework is not only suitable to implement multigrid methods, but also general stencil codes. This, for instance, includes implementations of the lattice Boltzmann method (LBM).

The rest of this work is structured as follows: First, we summarize background information relevant for our work in chapter 2. This includes mathematical and numerical aspects of applicable discretizations and solvers, as well as particulars about DSLs and code generation techniques. Next we present our DSL as seen by users in chapter 3 and how it is processed by our code generator in chapter 4. Chapter 5 showcases how our code generation pipeline can be extended, e.g. to emit massively parallel codes and to support new grid types. Based on these concepts, we highlight CFD applications implemented with our approach in chapter 6. Chapter 7 discusses the ecosystem around our code generator and how it can be used to examine and compare multiple configurations. Finally, we discuss related work in chapter 8, before concluding in chapter 9.
In this chapter we summarize background information relevant to the present work. Sections 2.1 and 2.2 go into detail about the numerics side, more precisely discretization and solver approaches. This is complemented by sections 2.3 and 2.4, which discuss computer science focused topics, i.e. domain-specific languages (DSLs) and code generation technologies.

2.1 Discretization of Partial Differential Equations

One way to numerically solve partial differential equations (PDEs) is to discretize them such that a system of linear, or a non-linear in case of a non-linear PDE, equations is set up. This system can then be solved with a suitable technique, thereby yielding an approximate solution for the original problem. In this section, we summarize key concepts of discretization techniques used in the present work. Please note, that it is deliberately kept concise and focuses on the aspects most relevant to our approach. For more thorough introductions we refer to literature, e.g. [92, 97].

In general, the following steps are performed: Initially, the physical domain $\Omega$ is discretized using a grid or a grid-like structure to form the computational domain $\hat{\Omega}$. For this chapter, we restrict ourselves to Cartesian grids. On them, functions can be approximated. In the most basic case, this is done by assigning the values of the function evaluated at certain points of the grid to said points. These so-called discretization points are traditionally the nodes or cell centers of the grid, but can also be located at other positions such as, e.g., on the edges. The resulting values can then be stored in what we call fields – a combination of logical indexing and data. In more complicated approaches, functions
are not approximated by single values, but using special bases. Usually, these bases only have support on a limited portion of the grid, such as a single cell or all cells around a given node. Instead of storing the function values themselves, it is now necessary to store a set of coefficients that, multiplied with the basis functions, approximate the function. If the chosen bases only consist of constant unit functions, this approach can be seen as equivalent to the first one.

Next, operators and operator applications are approximated with calculations based on field values at neighboring grid locations. Dominant approaches for this are the finite difference method (FDM), finite element method (FEM) and finite volume method (FVM) as well as discontinuous Galerkin (DG). For the remainder of this work, we focus on finite differences (FD) and finite volumes (FV). Both approaches work with approximating functions with single values and, therefore, a basis function approach is not required.

2.1.1 Finite Difference Method

The One-Dimensional Case

The main idea of the FDM is finding combinations of values of neighboring grid values to approximate operators such as derivatives. In order to do so, one starts with setting up Taylor expansions for neighboring grid locations. Given a function \( u : \Omega \rightarrow \mathbb{R} \), which is evaluated at grid points \( x_i \) to give a set of discretized values \( u_i \), the expansions read

\[
\begin{align*}
  u(x_i + \Delta x) &= u_{i+1} = u_i + \Delta x \frac{\partial}{\partial x} u_i + \frac{\Delta x^2}{2} \frac{\partial^2}{\partial x^2} u_i \\
  &+ \frac{\Delta x^3}{6} \frac{\partial^3}{\partial x^3} u_i + \frac{\Delta x^4}{24} \frac{\partial^4}{\partial x^4} u_i + R(\Delta x^5), \\
  u(x_i - \Delta x) &= u_{i-1} = u_i - \Delta x \frac{\partial}{\partial x} u_i + \frac{\Delta x^2}{2} \frac{\partial^2}{\partial x^2} u_i \\
  &- \frac{\Delta x^3}{6} \frac{\partial^3}{\partial x^3} u_i + \frac{\Delta x^4}{24} \frac{\partial^4}{\partial x^4} u_i + R(\Delta x^5).
\end{align*}
\]  

(2.1)

Here, \( R(o) \) summarizes remainder terms in the order of \( o \) or higher and thereby represents the \textit{discretization error}.  


2.1 Discretization of Partial Differential Equations

Operators are now built by suitably combining the previously set up equations. By, e.g., subtracting them to obtain

\[ u_{i+1} - u_{i-1} = \left( u_i + \Delta x \frac{\partial}{\partial x} u_i + \frac{\Delta x^2}{2} \frac{\partial^2}{\partial x^2} u_i + R(\Delta x^3) \right) \]
\[ - \left( u_i - \Delta x \frac{\partial}{\partial x} u_i + \frac{\Delta x^2}{2} \frac{\partial^2}{\partial x^2} u_i + R(\Delta x^3) \right), \]

a second order accurate representation of the first derivative can be obtained and formulated as

\[ \Delta x \frac{\partial}{\partial x} u_i = \frac{u_{i+1} - u_{i-1}}{2\Delta x} + R(\Delta x^3). \] (2.3)

In literature, this is usually called central difference. Similar to this option, other variants can be found by subtracting \( u_i \) from one of the equations in eq. (2.1). This leads to the forward difference

\[ \Delta x \frac{\partial}{\partial x} u_i = \frac{u_{i+1} - u_i}{\Delta x} + R(\Delta x^2) \] (2.4)

and the backward difference

\[ \Delta x \frac{\partial}{\partial x} u_i = \frac{u_i - u_{i-1}}{\Delta x} + R(\Delta x^2). \] (2.5)

Both variants have a discretization error order worse than obtained using central differences.

Higher order derivatives can be constructed similarly or obtained by combining lower order derivative approximations. For the case of second order, this results in

\[ \frac{\partial^2}{\partial x^2} u_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} + R(\Delta x^3) \] (2.6)

which can either be found by adding both equations in eq. (2.1) and subtracting \( 2u_i \), or by constructing the central difference of two central differences evaluated a half step to the left and to the right.
A higher discretization order can be achieved by using more than two neighbors per dimension. Using two neighbors on each side results in

\[
\begin{align*}
\frac{4\Delta x^3}{3} \frac{\partial^3}{\partial x^3} u_i + \frac{2\Delta x^4}{3} \frac{\partial^4}{\partial x^4} u_i + R(\Delta x^5) \\
\frac{\Delta x^3}{6} \frac{\partial^3}{\partial x^3} u_i + \frac{\Delta x^4}{24} \frac{\partial^4}{\partial x^4} u_i + R(\Delta x^5) \\
\frac{\Delta x^3}{6} \frac{\partial^3}{\partial x^3} u_i - \frac{\Delta x^4}{24} \frac{\partial^4}{\partial x^4} u_i + R(\Delta x^5) \\
\frac{4\Delta x^3}{3} \frac{\partial^3}{\partial x^3} u_i + \frac{2\Delta x^4}{3} \frac{\partial^4}{\partial x^4} u_i + R(\Delta x^5).
\end{align*}
\]

(2.7)

Based on this, a fourth order accurate central difference can be derived as

\[
\Delta x \frac{\partial}{\partial x} u_i = \frac{u_{i-2} - 8u_{i-1} + 8u_{i+1} - u_{i+2}}{12\Delta x} + R(\Delta x^5).
\]

(2.8)

**Higher Dimensionalities**

In cases with more than one dimension, the general approach remains the same but is done for each dimension individually. For the negative Laplace operator in 2D, for instance, this yields

\[
\begin{align*}
-\Delta u_{i,j} &= -\frac{\partial^2}{\partial x^2} u_{i,j} - \frac{\partial^2}{\partial y^2} u_{i,j} \\
&= \frac{-u_{i+1,j} + 2u_{i,j} - u_{i-1,j}}{\Delta x^2} + \frac{-u_{i,j+1} + 2u_{i,j} - u_{i,j-1}}{\Delta y^2} + O(\Delta x^3) + O(\Delta y^3).
\end{align*}
\]

(2.9)
Relation to Stencils

Stencils are a compact notation for operations applied to a single point of a grid taking only neighboring values into account. As such, FD discretization of operators on a regular grid can be expressed as stencils as well. For the 1D case of the first order derivative, the central, forward and backward differences listen in eqs. (2.3) to (2.5) read

\[ \frac{1}{\Delta x} \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}, \tag{2.10} \]

\[ \frac{1}{\Delta x} \begin{bmatrix} 0 & -1 & 1 \end{bmatrix}, \tag{2.11} \]

and

\[ \frac{1}{\Delta x} \begin{bmatrix} -1 & 1 & 0 \end{bmatrix}. \tag{2.12} \]

The center element of a stencil is always the coefficient associated with the point it is applied to, that is the point with a zero offset. Likewise, neighbors to the left and right are linked with the coefficients left and right in the stencil.

In 2D, stencils can be expressed in a similar fashion. Take, for instance, the negative Laplace operator from eq. (2.9). It can either be expressed as the sum of two 1D stencils

\[ \begin{bmatrix} -\frac{1}{\Delta x} & \frac{2}{\Delta x} & -\frac{1}{\Delta x} \end{bmatrix} + \begin{bmatrix} \frac{-1}{\Delta y} \\ \frac{2}{\Delta y} \\ \frac{-1}{\Delta y} \end{bmatrix} \tag{2.13} \]

or as a combined 2D stencil

\[ \begin{bmatrix} 0 & -\frac{1}{\Delta y} & 0 \\ -\frac{1}{\Delta x} & \frac{2}{\Delta x} + \frac{2}{\Delta y} & -\frac{1}{\Delta y} \\ 0 & -\frac{1}{\Delta y} & 0 \end{bmatrix}. \tag{2.14} \]

Please note, that throughout this work stencils are depicted as they would be applied to a Cartesian grid. That is, right and top directions correspond to positive offsets while left and bottom directions correspond to negative offsets.
While analogous constructs are possible for higher dimensions, it is not as straightforward to depict them. A 3D version of the discretized negative Laplace operator would already look something like

\[
\begin{bmatrix}
0 & 0 & 0 \\
0 & -\frac{1}{\Delta z} & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
0 & \frac{1}{\Delta y} & \frac{2}{\Delta x} \\
-\frac{1}{\Delta x} & 0 & \frac{2}{\Delta y} \\
0 & \frac{2}{\Delta y} & 0 \\
\end{bmatrix}
\begin{bmatrix}
0 & 0 & 0 \\
0 & -\frac{1}{\Delta z} & 0 \\
0 & 0 & 0 \\
\end{bmatrix}.
\] (2.15)

In our framework, we therefore decide to not use this traditional structure, but to express stencils as a list of pairs of offsets and coefficients. The 1D central difference stencil for the first order derivative then reads

\[
\begin{bmatrix}
-1 \\
1 \\
\end{bmatrix} \Rightarrow -\frac{1}{\Delta x} \quad (2.16)
\]

and the 3D negative Laplace stencil

\[
\begin{bmatrix}
0 & 0 & 0 \\
-1 & 0 & 0 \\
1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 1 & 0 \\
0 & 0 & -1 \\
0 & 0 & 1 \\
\end{bmatrix} \Rightarrow \frac{2}{\Delta x} + \frac{2}{\Delta y} + \frac{2}{\Delta z} \quad (2.17)
\]

It is immediately evident that this approach naturally extends to higher dimensions as well.

### 2.1.2 Finite Volume Method

Similar to FD, the FVM uses a grid for discretization. Where, however, values are usually located at the grid nodes in the case of FD, cell centers
are used in the case of FV. Usually, it is then assumed that the value does not change across one cell and that, thus, the whole cell is represented by the central value. Contrary to FD, operators are not discretized separately but the whole equation is considered at once. We want to illustrate this process with the example of a simple two-dimensional equation

\[
\frac{\partial}{\partial x} u + \frac{\partial}{\partial y} u = \nabla \cdot u = f. \tag{2.18}
\]

First, the equation is integrated over the physical domain \(\Omega\). In our example this results in

\[
\iiint_{\Omega} \nabla \cdot u \, dx \, dy = \iiint_{\Omega} f \, dx \, dy. \tag{2.19}
\]

Instead of solving for the integral over \(\Omega\), we now solve for the same equation over the computational domain \(\hat{\Omega}\). Splitting \(\hat{\Omega}\) into multiple cells, we can then solve for the given equation in each cell. In this context, each cell is called a control volume and denoted with an index subscript, e.g. \(\Omega_{i,j}\) in 2D. Plugging this into eq. (2.19) yields

\[
\iiint_{\Omega_{i,j}} \nabla \cdot u \, dx \, dy = \iiint_{\Omega_{i,j}} f \, dx \, dy \quad \forall \Omega_{i,j} \in \Omega. \tag{2.20}
\]

Please note, that we we omit \(\forall \Omega_{i,j} \in \Omega\) from hereon, but it should be clear that the equation still has to be fulfilled for every cell in the grid.

Using the divergence theorem it is now possible to replace the left-hand side volume integral in eq. (2.20) with boundary integrals to obtain

\[
\int_{\partial\Omega_{i,j}} u \cdot \hat{n} \, dx \, dy = \iiint_{\Omega_{i,j}} f \, dx \, dy, \tag{2.21}
\]

where \(\hat{n}\) is the surface normal. For axis-aligned grids with uniform \(\Delta x\) and \(\Delta y\), this can be expanded to

\[
\Delta y u_{i+\frac{1}{2},j} - \Delta y u_{i-\frac{1}{2},j} + \Delta x u_{i,j+\frac{1}{2}} - \Delta x u_{i,j-\frac{1}{2}} = \Delta x \Delta y f_{i,j}. \tag{2.22}
\]

Here, half indices mark evaluation of the specified quantity at cell interfaces. Various methods for reconstructing that value are conceivable with the most straight-forward being to take the mean of the two neighboring cell values, e.g.

\[
u_{i+\frac{1}{2},j} = \frac{u_{i,j} + u_{i+1,j}}{2}. \tag{2.23}
\]
Plugging this, and equivalent formulations for the remaining interfaces, into eq. (2.22) yields

\[ \Delta y (u_{i+1,j} - u_{i-1,j}) + \Delta x (u_{i,j+1} - u_{i,j-1}) = \Delta x \Delta y f_{i,j}. \] (2.24)

This particular formulation could now be divided by \( \Delta x \Delta y \) to result in the formulation that would be obtained by applying (central) FD, and a similar transformation can be applied in the 3D case as well. However, this only works for grids with constant grid widths. For non-uniform grids, the discretizations obtained through applying FD and FV differs significantly.

Similar to before, setting up a stencil formulation for our model equation is possible and yields

\[
\begin{bmatrix}
0 & \Delta x & 0 \\
\Delta y & 0 & \Delta y \\
0 & \Delta x & 0
\end{bmatrix}
\begin{bmatrix}
u_i \\
u_j \\
u_{i,j}
\end{bmatrix}
=
\begin{bmatrix}
0 & 0 & 0 \\
0 & \Delta x \Delta y & 0 \\
0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
f_i \\
f_j \\
f_{i,j}
\end{bmatrix}.
\] (2.25)

Next, we examine a PDE with second order derivatives, namely Poisson’s equation (with negative sign):

\[ -\Delta u = -\nabla^2 u = f. \] (2.26)

For a given 2D grid we can again integrate and separate into control volumes to find

\[ -\iiint_{\Omega_{i,j}} \nabla^2 u \, dx \, dy = \iint_{\Omega_{i,j}} f \, dx \, dy. \] (2.27)

Using Green’s identities allows replacing the volume integral on the left-hand side with a surface integral once again:

\[ -\int_{\partial \Omega_{i,j}} \nabla u \cdot \hat{n} \, dx \, dy = \iint_{\Omega_{i,j}} f \, dx \, dy. \] (2.28)

At this point, however, instead of needing to evaluate quantities on the cell interfaces, evaluations of the derivatives of said quantities are required. The most straight-forward approach to obtain them is using FD approximations. One possible alternative would be introducing an auxiliary unknown that represents the gradient of \( u \). In this work, we will
focus on the FD approach. Applying it, and assuming a uniform grid, results in

\[
-\Delta y \frac{u_{i+1,j} - u_{i,j}}{\Delta x} + \Delta y \frac{u_{i,j} - u_{i-1,j}}{\Delta x} \\
-\Delta x \frac{u_{i,j+1} - u_{i,j}}{\Delta y} + \Delta x \frac{u_{i,j} - u_{i,j-1}}{\Delta y} = \Delta x \Delta y f_{i,j}.
\] (2.29)

Equation (2.29) can again be written in stencil notation as

\[
\begin{bmatrix}
0 & -\frac{\Delta x}{\Delta y} & 0 \\
\frac{\Delta y}{\Delta x} & 2\frac{\Delta u}{\Delta x} + 2\frac{\Delta x}{\Delta y} & -\frac{\Delta y}{\Delta x} \\
0 & -\frac{\Delta x}{\Delta y} & 0
\end{bmatrix}
\begin{bmatrix}
u_{i,j}
\end{bmatrix}
= \begin{bmatrix}
0 \\
0 \\
0
\end{bmatrix} f
\] (2.30)

which can be simplified if \( \Delta x = \Delta y \) to yield

\[
\begin{bmatrix}
0 & -1 & 0 \\
-1 & 4 & -1 \\
0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
u_{i,j}
\end{bmatrix}
= \begin{bmatrix}
0 \\
0 \\
0
\end{bmatrix} \Delta x^2 f.
\] (2.31)

### 2.2 Numerical Solvers

After applying a suitable discretization technique, such as discussed in the previous section, a system of linear equations in the form of \( Ax = b \) is obtained. Each line of the system represents the equation to be fulfilled for a specific unknown located at a single grid location, e.g. at one given grid node. In case there is a single operator in the PDE to be solved, and that its discretized counterpart can be represented by a stencil, each line of \( A \) can be constructed from said stencil. Please note, that a non-linear PDE may result in a system of non-linear equations \( A(x) = b \). This case is discussed in more detail in section 6.2.

In order to obtain the final solution, the system of equations needs to be solved. The most straight-forward approach is inverting the system matrix \( A \) and multiplying it with the right-hand side \( b \). Using such direct solvers in practice, however, is usually prohibitively expensive. On top of that, efficient parallelization, especially across multiple compute nodes, is extremely difficult.

Another class of methods, which will also be the focus of this work, is given by iterative solvers (or iterative methods). Here, an initial guess \( x^0 \) is improved during the course of multiple iterations, where \( x^k \) denotes
the guess in the $k$th step. The method is applied until the $x^k$ approximates the searched solution sufficiently good. This can be checked using the norm of the residual $r$ which is given by $||r^k|| = ||b - Ax^k||$ for the $k$th iteration.

In many iterative solvers, $A$ doesn’t have to be constructed explicitly since it suffices to be able to express the product $Ax$. If $A$ can then be represented through one or more stencils, simply applying the stencils directly is usually more efficient. In literature, this is usually referred to as matrix-free methods. This not only results in less data that needs to be stored, but also in less data that needs to be transferred in each solver iteration, thereby potentially speeding up implementations.

### 2.2.1 Point-Based Solvers

The most straight-forward class of iterative solvers is given by point-based solvers or sometimes matrix splitting methods. They build upon the relation $A = L + D + U$, where $L$, $D$ and $U$ are the strictly lower triangular, diagonal and strictly upper triangular of $A$, respectively.

The name point-based stems from the idea to calculate an update for an unknown at each grid point separately, such that it solves one line of the system of equations exactly. The resulting updated unknown vector can be stored in a separate field which is then swapped with the current one afterwards. Methods build on this idea are usually called Jacobi-type methods and can be formulated as

$$x^{k+1} = D^{-1} \left( b - (A - D)x^k \right).$$

(2.32)

Looking at a single unknown $x_i$, a corresponding element-based update is given by

$$x_i^{k+1} = \frac{1}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij}x_j^k \right).$$

(2.33)

Unknowns can also be updated in-place, resulting in a Gauss-Seidel method. In this case, the order in which the single unknowns are updated heavily influences the result. A common approach is using a lexicographical ordering and updating the unknowns according to their index in the governing system of equations, resulting in

$$x^{k+1} = (L + D)^{-1} \left( b - Ux^k \right).$$

(2.34)
In practice, inverting $L + D$ is usually not feasible, but an appropriate element-based update can be derived using forward substitution:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left( b_i - \sum_{j<i} a_{ij} x_j^{k+1} - \sum_{j>i} a_{ij} x_j^k \right). \quad (2.35)$$

This method is, however, implicitly serial due to the data dependencies between the updates.

Coloring can be added to allow parallelization. Here, the main idea is splitting the unknowns into sets that can be updated independently and such that there are no data dependencies between unknowns within any given set. In case of Poisson’s equation discretized on a uniform grid, such as discussed in section 2.1, two colors are sufficient. In 2D, unknowns are colored in a checker-board fashion which is why this coloring is often called red-black and the corresponding solver red-black Gauss-Seidel (RBGS). A similar 2-way coloring, still called red-black, may be applied for other numbers of dimension as well. All three cases are visualized in figure 2.1. Updating the unknowns is then split into multiple steps where the number of steps is equal to the number of colors, e.g.

$$\tilde{x}_i^k = \begin{cases} \frac{1}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij} x_j^k \right) & \text{if } i \in \text{red} \\ x_i^k & \text{else} \end{cases} \quad (2.36)$$

$$x_i^{k+1} = \begin{cases} \frac{1}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij} \tilde{x}_j^k \right) & \text{if } i \in \text{black} \\ \tilde{x}_i^k & \text{else} \end{cases}.$$

As evident, only unknowns of one color are updated in each step. Updates within one step can be done in-place and in parallel.

Another option to tune the already discussed solvers is blending the old and the updated solution instead of overwriting the former with the latter. This can be done using a linear interpolation with a factor lower than one, resulting in under-relaxation or damping. An example of using this technique is a damped Jacobi solver:

$$\mathbf{x}^{k+1} = (1 - \omega) \mathbf{x}^k + \omega \mathbf{D}^{-1} (\mathbf{b} - (\mathbf{A} - \mathbf{D}) \mathbf{x}^k) \quad (2.37)$$

which can be reformulated as

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \omega \mathbf{D}^{-1} (\mathbf{b} - \mathbf{A} \mathbf{x}^k) \quad (2.38)$$
Background

Figure 2.1: Illustration of the red-black (checkerboard) pattern in 1D, 2D and 3D.

with the corresponding element-based update

\[ x_i^{k+1} = x_i^k + \frac{\omega}{a_{ii}} \left( b_i - \sum_j a_{ij} x_j^k \right). \] (2.39)

Using a blending factor larger than one results in over-relaxation, such as used in successive over-relaxation (SOR) solvers which can be expressed as

\[ x_i^{k+1} = x_i^k + \frac{\omega}{a_{ii}} \left( b_i - \sum_{j<i} a_{ij} x_j^{k+1} - \sum_{j>i} a_{ij} x_j^k \right). \] (2.40)

As evident, they are a variant of Gauss-Seidel solvers.

2.2.2 Block Solvers

In the previous section, we discussed iterative methods solving for one unknown at a time. Similarly to these point-based solvers, it is also possible to solve for multiple unknowns at the same time to form so-called block-based solvers or simply block solvers. If groups of unknowns to be updated simultaneously are stored consecutively in the unknown vector \( \mathbf{u} \) it suffices to replace the diagonal \( D \) with the block-diagonal \( \tilde{D} \) to obtain the block Jacobi method

\[ \mathbf{x}^{k+1} = \tilde{D}^{-1} \left( \mathbf{b} - \left( \mathbf{A} - \tilde{D} \right) \mathbf{x}^k \right). \] (2.41)

Analogously, a damped variant can be set up as

\[ \mathbf{x}^{k+1} = \mathbf{x}^k + \omega \tilde{D}^{-1} \left( \mathbf{b} - \mathbf{A} \mathbf{x}^k \right). \] (2.42)
2.2 Numerical Solvers

Figure 2.2: Illustration of a red-black pattern for blocks of size 2 in 1D, $2 \times 2$ in 2D and $2 \times 2 \times 2$ in 3D.

Inverting the block-diagonal is, however, more costly than inverting the diagonal since a small matrix has to be inverted for each block. In case of groups with $b$ unknowns, these local matrices are of size $b \times b$, which is why $b$ is usually chosen to be rather small.

In practice, groups of unknowns are usually not stored consecutively since the grouping is chosen according to the grid neighborhood. For instance, considering $2 \times 2$ nodes at the same time is a popular choice for certain 2D problems. Conceptually, however, the method remains the same: a small system of $b \times b$ has to be solved for each group of size $b$ to update all associated unknowns.

Other variants, such as a block Gauss-Seidel and block SOR solvers can be constructed similarly taking the variants discussed in the previous section as base. Similarly, colored variants are easy to set up as well. This is also illustrated in figure 2.2 where a red-black coloring for $2^d$ blocks is shown, using $d$ as the dimensionality again. Lastly, it is also possible to overlap blocks to form overlapping block solvers.

2.2.3 Krylov Subspace Solvers

The main idea behind Krylov subspace solvers is constructing a sequence of approximate solutions

$$x^k \in x^0 + \mathcal{K}^k(A, r^0).$$

Here, $\mathcal{K}^k$ is the $k$th Krylov subspace given by

$$\mathcal{K}^k(A, y) = \text{span}\{y, Ay, AAy, ..., A^{n-1}y\}.$$  

Prominent examples for solvers from this class are conjugate gradient (CG), conjugate residual (CR), minimal residual (MINRES) and biconjugate
2 Background

Gradient stabilized (BiCGSTAB). All of them will be discussed together with their respective DSL implementations in section 4.3.2. In practice, they are often favored over point-based solvers due to them converging faster and over block solvers due to lower execution times.

2.2.4 Geometric Multigrid

Solving the governing system of equations $Ax = b$ means looking for an $x$ that satisfies the system. A guess in an iterative solver $x^k$ can be expressed as the sum of the actual exact solution $x_{exact}$ and an error $e$, i.e. $x^k = x_{exact} + e$. Please note, that even though $x_{exact}$ may solve the given system exactly indeed, a discretization error may still remain which is not included in $e$.

Plugging our definition of $x_{exact} = x^k - e$ into $Ax_{exact} = b$ yields

$$Ae = Ax^k - b = r,$$  \hspace{1cm} (2.45)

where $r$ is again the residual. We call eq. (2.45) the error equation.

Solving for $Ax = b$ is equivalent to reducing $e$ to zero. When applying point-based solvers, such as described before, it can be observed that only certain components of the error get eliminated efficiently, namely high-frequency ones. These methods are thus efficient in smoothing the error such that only low-frequency components remain. Reducing it to zero, however, takes usually a number of iterations so high that they are not suitable for practical use as a stand-alone solver.

Smooth errors, however, can be represented well on coarser grids. This restriction also shifts the frequency distribution of the error, thereby making it again a viable target for solvers with smoothing properties. Using the smoothing and coarse grid principles allows constructing an efficient iterative solver. First, the current solution is smoothed by applying a couple of iterations of a smoother, usually a cheap point-based solver. Then, the residual is evaluated and restricted to a coarser grid where it forms the right-hand side. After solving the error equation $Ae = r$ on the coarse grid, the obtained error is interpolated onto the original grid where it is applied as a correction. Finally, a smoother is applied again in order to reduce new high-frequency error components introduced by the correction.

This so-called two-grid method can be applied in a recursive fashion, thereby yielding the multigrid algorithm also shown in algorithm 1. In
it, \( S \) represents a smoother application and its power the number of times it is applied. \( R \) and \( P \) denote the restriction and prolongation operators, respectively. The variant shown is known as a v-cycle and can be visualized as in figure 2.3. In this figure, we also see the w-cycle which can be obtained by recursing more than one time per iteration.

**Algorithm 1:** Recursive \( v(s_1, s_2) \)-cycle to compute the approximate solution of the next iteration \( x^{k+1} = V_{\text{finest}}(x^k_{\text{finest}}, A_{\text{finest}}, b_{\text{finest}}) \).

```plaintext
if coarsest level then
    solve directly
else
    \( x^k \leftarrow S^{s_1}(x^k, A, b) \) {pre-smoothing}
    \( r \leftarrow b - Ax^k \) {residual computation}
    \( r_{\text{coarser}} \leftarrow Rr \) {restriction}
    \( e_{\text{coarser}} \leftarrow V_{\text{coarser}}(0, A_{\text{coarser}}, r_{\text{coarser}}) \) {recursion}
    \( e \leftarrow Pe_{\text{coarser}} \) {prolongation}
    \( x^k \leftarrow x^k + e \) {correction}
    \( x^{k+1} \leftarrow S^{s_2}(x^k, A, b) \) {post-smoothing}
end
```

On the coarsest level, one can employ direct solvers which is, however, usually not advisable in a high performance computing (HPC) context. Instead, a so-called coarse-grid solver (CGS) is employed instead,
2 Background

often a Krylov solver as introduced before and explained in detail in section 4.3.2.

In practice, multigrid methods are quite attractive compared to other solver variants discussed so far. The main reason for this lies in their theoretical complexity, which is linear in the number of unknowns to be solved for. Since the computational complexity is vital for scalability and, thus, on the road to exascale computing, it is the main focus of this work.

Finally, more in-depth introductions and reviews of multigrid methods can be found in literature, e.g. [67, 100].

2.3 Domain-Specific Languages

Programming languages traditionally employed in HPC, such as C/C++ and FORTRAN, expose a high degree of low-level functionality. As such, they are perfectly suited to create highly optimized implementations of numerical algorithms. This process, however, requires a lot of time and effort and can be almost impossible for scientists lacking proper training. Even implementing unoptimized versions can often be tedious and time-consuming. Scripting languages, such as Python, aim to mitigate this. They often do so, however, at the cost of run-time performance.

Domain-specific languages (DSLs) aim to resolve this conflict between ease of implementation and execution performance. In contrast to general-purpose languages (GPLs), that need to be able to cover a wide range of application scenarios, they are restricted to one specific application domain. The core idea is then to design a new language that caters to the needs of users stemming from said domain. This usually includes the ability to express concepts relevant to the domain in a concise and familiar fashion. Since the input is abstract, code generators can – ideally – transform it into a specialized implementation running optimally on a given hardware platform.

DSLs come in two flavors. Internal or embedded DSLs are an extension of an existing host language, usually a GPL. The extension itself can take different forms, e.g. the introduction of macros, new data types and/ or specialized functions. In C++, templates are also often used. Last but not least, annotations in the form of pragmas or comments are frequently employed as well. External DSLs on the other hand define a completely new language. This allows for more flexibility and the syntax of the DSL can be custom-tailored to match what domain specialists
are familiar with. Moreover, code generators using specifications from external DSLs as input have complete access to the whole specification, which potentially allows new and powerful optimizations. The main drawback of this approach, however, is the increased initial effort in conceptualizing the language and in setting up the processing pipeline. Nevertheless, we focus on external DSLs for the remainder of the this work since they are best suited to implement our vision.

## 2.4 Code Generation

This section reviews the general steps that need to be taken to produce code in a given target programming language from abstract input given in an (external) DSL.

At the start of the pipeline stands the **lexer** (or tokenizer). Its job is to mark keywords and delimiters of the input language. The output, a tokenized representation, is then given to a **parser**. The parser matches it to a given grammar to produce a so-called **parse tree** or **concrete syntax tree (CST)**. This tree contains a set of **nodes** with direct mapping to elements of the language. In some applications, the CST is equivalent to an **abstract syntax tree (AST)**.

Next, one or more transformations may be applied to the CST to produce the **intermediate representation (IR)**, in which more transformation may be carried out. Each of them receives an AST as input and emits a (usually different) AST as output. After all necessary transformations have been applied, the final AST is fed into a **pretty-printer**. Its output is usually source code in one or more target languages.

Let us consider a simple example. The input given in a fictional DSL could look like this:

```plaintext
repeat 5 times {
    print ( "Hello" )
}
```

**Listing 2.1:** Exemplary input in a fictional DSL.

The lexer will first mark delimiters such as spaces and brackets. This allows identifying and marking keywords in the input, in this case `repeat`, `times` and `print`. Concurrently, literals such as `5` and `Hello` are marked as well. The resulting representation can be used by the parser to match
its grammar and produce an AST similar to the one depicted in figure 2.4.

The types of nodes occurring the AST are usually also modeled as a hierarchy. At the root, there is a general Node type. Next, types to represent Expressions and Statements are introduced. They are followed by specialized types according to the DSL's scope. In our example, this would translate to adding Repeat and Print statements, as well as literal expressions such as StringLit. The result is illustrated in figure 2.5.

2.4.1 Framework

When implementing a custom programming language, and the attached code generation pipeline, one can choose from many existing technologies to base the project upon. We conducted a comparative study of such technologies which was published in [19]. Its result was the decision to set up a custom code transformation and generation framework using the popular programming language Scala\(^1\). The bulk part of the ef-

\(^1\)

\[\text{scala-lang.org}\]
2.4 Code Generation

forts in implementing the foundation of said framework, which we call Athariac, has been done by Christian Schmitt. Its core concepts can also be reviewed in [94] and the dissertation of Christian Schmitt. Note that Athariac is designed to provide the backbone for projects like Exa-Stencils and, as such, aims to be agnostic of the domain it is ultimately used for.

One reason for Scala is that a number of different lexers and parsers is already implemented and provided by default. Using an also provided parser combinator allows aggregating single parser variants to a bigger one. Its output is an AST which, after some transformations, only contains nodes of the IR. To emphasize this, all node types inherit from the IR_Node trait which, in turn, inherits from the general Node trait. Details on the intermediate steps are discussed in chapter 4. The node type hierarchy is modeled as actual type hierarchy in our Scala framework. Analogous to the previously discussed example, abstract classes such as IR_Expression and IR_Statement are used to group types. This also allows handling collections of a certain group of types, e.g. a list of statements. The leaves of the type hierarchy are, in most cases, Scala case classes to allow for matching them in transformations, as explained next.

The main purpose of Athariac is applying transformations to an AST. For this, it implements the visitor pattern to traverses the AST. At each node, a user-defined match is performed and, if successful, the current node is replaced with the return value of the match. An example for this mechanism is given in listing 2.2.

```scala
Transformation("Simplify", {
  case IR_Addition(n, IR_IntegerConstant(0)) => n
})
```

**Listing 2.2:** Example of a strategy in Athariac that simplifies additions with 0.

Although this example is rather simple and probably only of little use in practice, it demonstrates Scala's deep matching capabilities well. Our transformation looks for any node that adds 0 to something. If matched, it simply replaces the addition with its left-hand child. A default match is not required in transformations. By default, transformations will also descend into children of replaced nodes. Since this behavior can lead to complications, e.g. when wrapping one node with another, it can be deactivated by providing a flag when creating the transformation.
Multiple transformations, that are to be applied in succession, can be collected in a so-called strategy. When strategies are applied, the visitor starts at the root node by default. A custom starting node can, however, be provided to override this behavior.

The StateManager keeps track on applied strategies and the current AST. It optionally implements a transactional approach that allows rolling back if necessary. In practice, however, this increases the generation time by an unreasonable amount. It is, thus, switched off by default.

Up to now, information in transformations is only available if it comes directly from the matched node or its children. If data from further up the tree is required, so-called collectors can be used. They are registered to a strategy and allow implementing enter and leave methods. Inside them, one usually matches specific node types whose information is then stored in the collector. The functions are called whenever the visitor starts or finished handling an AST node. An example for using collectors is given in listing 2.3 where a simple strategy for variable resolution is implemented.

```scala
class VariableDeclarationCollector
  extends Collector {
    var declarations =
      HashMap[String, IR_VariableDeclaration]()

    override def enter(node : Node) = {
      node match {
        case decl : IR_VariableDeclaration =>
          declarations += ((decl.name, decl))
        case _ =>
      }
    }
}

object VariableResolution
  extends DefaultStrategy("Resolve variable accesses") {
    var declCollector =
      new VariableDeclarationCollector
    this.register(declCollector)
  }
```
2.4 Code Generation

```javascript
this += new Transformation("Resolve", {
  case access : IR_UnresolvedAccess
    if declCollector.declarations.
      contains(access.name) =>
      IR_VariableAccess(access.name,
        Duplicate(declCollector.
          declarations(access.name).datatype))
})
```

**Listing 2.3:** Example of a strategy resolving accesses to previously defined variables. A specialized collector keeps track of the declarations.

Please note that this is an abridged excerpt which would have to be extended in reality to, amongst others, handle scopes and implement a suitable reset mechanism. In this example, the enter method matches for variable declarations which are then kept (as reference) in a map where they can be accessed by their name. The connected strategy then looks for accesses with a fitting name and, if successful, injects suitable access nodes. For this, the data type of the original declaration is carried over.

Lastly, arbitrary information can be appended to any AST nodes in the form of annotations. The most prominent use case is annotating the originating line of DSL code for nodes to allow for more useful error messages. Annotations are preserved when replacing one node with another.
3 The ExaStencils Language (ExaSlang)

This chapter describes the ExaStencils language – or ExaSlang for short – from the perspective of users. That is, we discuss important language concepts and specifics, but do not go into detail on how our generator processes that input. Instead, we defer this discussion to chapter 4.

The main goal for ExaSlang was to create a language that can be used to easily describe geometric multigrid solvers in an abstract fashion. It should work for a sufficiently wide range of partial differential equations (PDEs) and allow for guided and automatic optimizations. Parallelization particulars should be exposed to users, but only optionally and on a very abstract level. As already discussed in chapter 1, we focus on a narrow domain. The main restriction are

- the solver algorithm should come from the class of geometric multigrid solvers,
- the grid needs to be composed of structured patches combined in a regular fashion and
- all kernels must be expressible as stencil codes.

While we designed ExaSlang with these specifications in mind, it is still embedded in the larger domain of general stencil codes. Thus, it is also possible to use it for different applications using a solver from a different class. One example for this is given in section 6.3 where an explicit time-stepping scheme is implemented for a system of hyperbolic PDEs.

The rest of the chapter is organized as follows: In section 3.1 we introduce our layered approach to ExaSlang. Next, we discuss required concepts in section 3.2, followed by a detailed tour of the language itself in section 3.3. We conclude by looking at how the generator can be controlled through providing hints and via configuration files in sections 3.4 and 3.5 respectively.
## 3.1 Our Layered Approach

When designing the language, we quickly figured out that the expectations of potential users differed vastly. This can partly be attributed to them stemming from different communities. To model our language, we classify four classes of users:

<table>
<thead>
<tr>
<th>Interest</th>
<th>Indifference</th>
<th>Language</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 specification of the (continuous) problem</td>
<td>particulars of the discretization, the solver composition or the parallelization approach</td>
<td>LaTeX</td>
<td>natural scientists</td>
</tr>
<tr>
<td>2 tweaking the discretized formulation or experimenting with new discretizations</td>
<td>actual implementation of the solver and its parallelization</td>
<td>—</td>
<td>domain experts, e.g. from CFD</td>
</tr>
<tr>
<td>3 experimenting with different solver implementations; includes fine-tuning parameters of the multigrid solver or replacing whole parts of it</td>
<td>particulars of the discretization, aspects of the parallelization</td>
<td>Matlab</td>
<td>(applied) mathematicians</td>
</tr>
<tr>
<td>4 implementing full applications and adapting the parallelization; e.g. changing communication patterns or overlapping communication and computation</td>
<td>particulars of the discretization and the solver</td>
<td>C/C++</td>
<td>computer scientists</td>
</tr>
</tbody>
</table>

Table 3.1: Envisioned classes of users for ExaSlang.
ExaSlang would need to be able to cater to the needs of all these user groups. Designing a language that covers the whole range from a very abstract and coarse-grained representation for group 1 to a detailed and fine-grained one for group 4 is extremely challenging. Packing all of the required functionality in one language would most likely fail. In consequence, we decided to design and implement ExaSlang as a hierarchical, multi-layered domain-specific language (DSL). Figure 3.1 underlines our concept: four distinct layers are stacked on one another. Each layer can be associated with one of our user groups. This gives us the flexibility to adapt the syntax of each language to be similar to what users on that layer are used to. Orthogonally, there is also description of the target hardware platform which can be used on each layer to apply optimizations. Ideally, this description would be given in its own language. Concepts for this target platform description language (TPDL), and how it could be used, have been published by our collaborators [93]. Implementation and integration in our framework, however, have unfortunately not been done so far. Instead we rely on a simple configuration files as detailed in section 3.5.1.

In our initial concept, lower layers could be automatically derived from predecessors. This still holds for layer 4 which is complete in the sense that the full program specification is given on this layer. Layer 1...
Figure 3.2: Revised concept of the four layers of ExaSlang.

can also still be used to generate lower layers, but layer 2 and layer 3 are a little different in this aspect. While layer 2 is only used to specify the problem to be solved, layer 3 harbors solely the solver implementation. As such, a full specification to be mapped to layer 4 can only be attained by combining the input of both layers. We illustrate this in figure 3.2 which spiritually replaces figure 3.1.

Our multi-layered DSL allows users to implement their problems and solutions in a form natural to them. Apart from this input, we also allow controlling the generator itself. This is necessary to set simple things like the output path, but also to control the generation process itself. Examples for the latter are the decision if CUDA code is to be generated or if vectorization is to be applied. We split this auxiliary input into multiple categories and allow users to provide separate files for each of them. All information concerning the hardware and software of the target system are collected in platform files. The settings files contain parameters that are used for the generation process, but don't directly change its output. This includes, e.g., output paths or the name of the target binary. Correspondingly, all parameters intended to change the output of the code generation pipeline are summarized in knowledge files. Partitioning the domain or adding support for MPI fall in this category for instance. More details about the three parameter collections are given in section 3.5.
3.2 Required Concepts

In this section, common elements of numerical solvers are discussed as well as how they are mapped to our DSL.

First and foremost, geometric multigrid solvers are of special interest for us. The idea of having a hierarchy of things, such as grids, should also be reflected in our language. We adopt this concept by allowing common language objects to be leveled. In this context, what is an object is not yet clearly defined. It could be something with a persistent state or something else, e.g. a function. In any case, being leveled translates to having multiple instances, where each is associated with exactly one level of the multigrid hierarchy. Using this concept, it is also possible to specialize entities for a given level. When accessing such a leveled object, it is necessary to provide the object’s identifier and an additional level specification. To avoid unnecessary code bloating, we support resolving levels implicitly. For this, each leveled object creates a level scope: each access without a level specification inside this scope will implicitly inherit the level of the surrounding scope.

Introducing leveled objects is very specific to our domain. It is, however, also necessary to support basic concepts existing in many other languages. We carry over these concepts, although they might not be available on every layer. First, we need literals to represent constant boolean and numerical values as well as strings. The usual arithmetic operations need to be available as well.

Next, we introduce functions. Layer 1 and layer 2 are not required to support defining functions which is why we only support it from layer 3 onwards. Functions in ExaSlang may be leveled, but they don’t have to. If they are, they create a level scope as introduced earlier. In contrast to function declarations, function calls are available on all layers. This is necessary to, amongst other things, express common mathematical operations such as sine and cosine.

Up to this point, it is not yet decided if the language is purely functional or not. This changes when we introduce variables. As per our domain, they can be leveled as well. We also support scoping, where scopes are created by, e.g., functions. For variables that need to be available globally, a globals block can be introduced. Shadowing of variables is supported in our language. Variables must also be explicitly typed. For this, we add basic data types such as Int, Real and Boolean.

The first specialized language object is given by domains. It is used to represent the physical or computational domain, the problem is
specified on. For now, we regard rectangular domains. The extension to other types of domains is discussed in section 5.1.1. Apart from an identifier, each domain requires an extent and a number of dimensions. As implied, we support multiple domains where all domains must be fully included in one global domain. It is, amongst other things, used when partitioning for multiple patches. Starting with layer 2, a grid hierarchy is imposed on each of the domains. Thus, domains are never leveled as such, but their encapsulated grids are. In this chapter we assume uniform grids, but suitable extensions are discussed in section 5.2.

Up to now, no quantities are associated with the domains or their grids. This is changed by introducing so-called fields. In our multilayered approach, the perception of fields changes from layer to layer. On layer 1, they can be interpreted as functions in the mathematical sense, i.e. something that maps from a domain to a (usually scalar) value. Discretizing a layer 1 field, and thereby tying it to specific locations of the grid, results in a layer 2 field. Conceptually it is similar to a Matlab vector which is why it is also identical to its layer 3 counterpart. On layer 4, fields are conceptually partitioned as detailed in section 5.1. Memory layout specifics such as information about added ghost layers are exposed on this layer as well. Each patch can be interpreted as being similar to a data container in C. On all layers, each field is always tied to exactly one domain.

For many algorithms relevant to us, such as time-stepping schemes and Jacobi-type smoothers, multiple versions of the same field are necessary. To avoid costly copy operations, and to facilitate the process for users, we allow fields to be slotted on the lower layers. Slotted fields have multiple instances of themselves which can be accessed through what we call slot modifiers, e.g. the current or next slot. Implementation-wise, this corresponds to a simple ring-buffer of field instances with a light-weight interface in our DSL.

To express PDEs, operators are required in addition to fields. Here, we regard operators in the mathematical sense and, as such, they are directly represented on layer 1. On layer 2 and layer 3, discretized variants are available. They can be interpreted as big matrices with the common restriction that they can be constructed from a fixed stencil. Note that the shape of the stencil has to be fixed, but the coefficients may vary. Conceptually, this matrix doesn't exist on layer 4 anymore, but instead the stencil is directly represented. One special case is given by stencil fields. They are, as the name suggests, a combination of a stencil with a
fixed shape and a field which stores the coefficients of the stencil. On layer 1, they are not distinguishable from regular operators, but starting from layer 2 onwards they require special handling.

The way operators are applied to fields changes conceptually depending on the layer. On layer 3, a matrix-vector multiplication is performed, while on layer 4 a stencil is applied to a single unknown at a specific grid location. Syntax-wise, however, we decide to represent both operations as simple multiplications in our language.

For a complete specification, boundary conditions are required. There are multiple options to represent and integrate boundary conditions. We choose to tie them to fields on layer 1 since it is the most natural variant for us. Thus, each layer 1 field has zero or one boundary condition attached. In the future, multiple boundary conditions may be required per field, but this is not supported yet. On the lower layers, we model boundary conditions as properties of target fields as well to promote consistency. Here, the biggest alternative would be modifying the (discretized) operators at the domain’s boundary when they are applied to a given field. This, however, can introduce conditional branching in kernels which has to be resolved thereby increasing effort that has to be invested in the code generator. For implemented boundary conditions, we currently support

- Dirichlet conditions which can be given as an arbitrary expression to be evaluated at the boundary,
- Neumann with an optional approximation order,
- a function call to a user-provided function which implements custom boundary handling, or
- none if no boundary conditions are required.

Last but not least, we need to introduce equations to express the relations between fields, other fields and operators. As usual, equations are leveled as well. This can be particularly important when implementing a multigrid solver since the equation to be solved or, more precisely the right-hand side of the equation, may change across levels. Equations work roughly the same on all layers, with the difference that they only contain the operator and field variants of their respective layer.

### 3.3 Layers 1 to 4

This section describes how the concepts discussed in the previous section can be mapped to the different layers of the ExaSlang language.
At the end of every subsection, we showcase a complete example for Poisson’s equation in 2D. For this, we use the following test problems described by their exact solution. In 2D, the exact solution is given by

$$u = \cos(\pi x) - \sin(2\pi y)$$  \hspace{1cm} (3.1)

with the corresponding right-hand side given by the negative Laplacian of $u$, i.e.

$$f = \pi^2 \cos(\pi x) - 4\pi^2 \sin(2\pi y).$$  \hspace{1cm} (3.2)

In 3D, we choose

$$u = \cos(\pi x) - \sin(2\pi y)$$  \hspace{1cm} (3.3)

as exact solution which yields

$$f = \pi^2 \cos(\pi x) - 4\pi^2 \sin(2\pi y)$$  \hspace{1cm} (3.4)

as the corresponding right-hand side. For both test problems, we impose Dirichlet boundary condition with the exact solution.

Note that all examples are stand-alone examples, that is they don’t require the specifications of previous layers. Nevertheless, they can often be generated from them. This requires some additional input in the form of so-called hints, which will be discussed in detail in section 3.4. We deliberately omit any hints in the stand-alone examples to keep them as brief as possible.

There are often cases, where one layer doesn’t support a certain specification, or is not able to derive a certain specification when mapping to a lower layer. In these cases, users may go to lower layers and implement the specific parts of their project manually.

One thing that is common to all layers is how level specifications are represented. For declarations, an `@` followed by a non-empty list of levels is used. Here, the simplest case is to explicitly enumerate all levels. In case of successive level lists, we support providing a range of levels separated by `to`. For the coarsest and the finest level, special aliases exist such that it becomes possible to write `@coarsest to finest`. This includes all levels which can be further shortened using the keyword `@all`. Lastly, single levels can be excluded from a list using `but`, e.g. `@all but finest`. Access to leveled objects is also done using an `@`, but followed by only a single level. This level can be represented by a constant, coarsest or finest, or a scope-dependent level. The latter is carried over the level of the surrounding object and can be accessed
with @current. Selecting the next-coarser and finer levels can be done using @coarser and @finer. For convenience, simple arithmetic options are allowed for level specification, e.g. @(current + 2). Lastly, if no level specification is given but one is expected by the compiler, an implicit @all is assumed for declaration and an implicit @current for accesses.

3.3.1 Layer 1

In this section, the language specifics of ExaSlang Layer 1 are discussed. It is conceptualized to be very close to LaTeX. Ideally it would ultimately be possible to simply copy-paste problem specifications from a given paper references without the need for further adaption to the DSL. One key aspect in reaching this goal is supporting Unicode symbols as identifiers and as part of certain language constructs (which are discussed below).

On layer 1, only a limited subset of domains are available. In detail, only such that can be modeled by the Cartesian product of intervals, i.e. rectangular and axis-aligned domains, are possible. Multiple domains are supported as long as each has its own identifier. Here, \Omega or Ω can be used as well to get a close match to a possible LaTeX reference. Moreover, the domain named Ω is assumed to be the global domain as introduced in section 3.2. Cartesian production can be modeled using \times or \times. Finally, the actual dimensionality of the domain is determined by the number of intervals. Listing 3.1 illustrates the creation of a new domain on layer 1.

```
Domain Ω = ( 0, 1 ) × ( 0, 1 )
```

**Listing 3.1:** ExaSlang 1 declaration of a unit square domain named Ω.

After at least one domain has been declared, fields can be created. As usual, each field has to be tied to exactly one domain. This can be done using \in or ∈ followed by the domain’s identifier. Optionally, an expression can be provided that is then used to initialize the field. Later on, an initialize kernel is generated that loops over the (then discretized) field and evaluates the given expression at every point. This also enables using geometric information, such as the nodal positions, in the initialize expression. Apart from the field declaration, boundary conditions can be added via a separate statement. This statement looks like a regular field declaration, but it is tied to the boundary of a
domain rather than to the whole domain. On layer 1 this is marked by using $\in \partial$ or $\in \partial$, followed by a domain identifier. Again, this is designed to be rather similar to specifications found in scientific publications. Lastly, the actual boundary condition has to be given. Here, None, Dirichlet, Neumann or user-provided functions are available as detailed in section 3.2. Not declaring any boundary conditions for a field is equivalent to using None.

All field declarations, and their associated boundary conditions, are leveled on layer 1. If no level specification is given, an implicit @all is assumed. This is also demonstrated in listing 3.2 where $u$ and its boundary conditions are implicitly defined on all levels. It is important, that leveling an object means that separate instances of said object are created for each level. That is, for each level one field and one boundary condition declaration is set up. In consequence, specializing, e.g., boundary conditions for given levels can be done easily.

```
1 Field u \in \Omega = 0.0
2 Field u \in \partial \Omega = \cos (\pi x ) - \sin (2 \pi y )
3 Field f \in \Omega = \pi^2 \cos (\pi x ) - 4 \pi^2 \sin (2 \pi y )
```

Listing 3.2: ExaSlang 1 declaration of two fields $u$ and $f$, as well as $u$’s boundary conditions. Both fields are implicitly defined on all levels, the same holds for $u$’s boundary conditions. Boundary conditions for $f$ are not required in this example.

Apart from fields, operators may be created. Operators are, just as most objects on layer 1, leveled and if no level specification is given in the declaration, an implicit @all is assumed. The declaration itself is done by providing an identifier and an expression for the operator. Partial derivatives can be expressed using $\partial$ or $\partial$ followed by an underscore and the derivative’s direction(s) in curly braces. For convenience, the Laplace operator is also available through $\Delta$ or $\Delta$. The operator’s dimensionality is deducted later at the discretization stage. Here, the operator is tied to a specific domain and, thus, its dimensionality. An example for operator declaration is given in listing 3.3 where the negative Laplace operator is specified. In 2D, an equivalent expression using partial derivatives is given. Cases for other numbers of dimension can be handled analogously.
3.3 Layers 1 to 4

Operator \( \text{op} = - \Delta \)

// in 2D this is equivalent to
Operator \( \text{op} = - ( \partial_{\{xx\}} + \partial_{\{yy\}} ) \)

Listing 3.3: ExaSlang 1 declaration of the negative Laplace operator. An equivalent representation using partial derivatives is given for the 2D case.

Finally, operators and fields can be used as parts of equations. In ExaSlang 1, each equation is named, that is it has an identifier it can be referenced with. Additionally, equations are leveled and as usual an implicit @all is assumed if no level specification is given in the declaration. The equation itself may consist of two arbitrary expressions separated by \( \text{==} \). For clarity, an optional colon may be added between the identifier and the equation. Listing 3.4 shows the declaration of Poisson’s equation.

Equation \( \text{uEq: op * u == f} \)

Listing 3.4: ExaSlang 1 declaration of Poisson’s equation.

On layer 1 all declaration types are optional since the form of the declaration identifies the type distinctly. Using this feature, listings 3.1 to 3.4 can be combined to form a complete layer 1 specification of Poisson’s equation as depicted in listing 3.5. At this point, the design of ExaSlang 1 really shows: apart from declaring the operator as a separate entity, the given specification closely matches something one would write in a scientific publication.

\[
\Omega = (0, 1) \times (0, 1) \\
u \in \Omega = 0.0 \\
u \in \partial \Omega = \cos (\pi x) - \sin (2\pi y) \\
f \in \Omega = \pi^2 \cos (\pi x) - 4\pi^2 \sin (2\pi y) \\
\text{op} = - \Delta \\
\text{uEq: op * u == f}
\]

Listing 3.5: ExaSlang 1 example for the complete specification of the 2D test problem from eqs. (3.1) and (3.2).

As initially stated, ExaStencils’ vision was to use input similar to the one illustrated in listing 3.5 – and nothing else. In the current state,
this is unfortunately not possible yet. In addition to a layer 1 problem specification, users also need to provide certain hints to the generator on how to set up the subsequent layers. The specifics of that mechanism are discussed later in section 3.4.

### 3.3.2 Layer 2

ExaSlang Layer 2 is the natural evolution of ExaSlang 1: the same complete problem specification is given, but now in a discretized version. As such, the basic objects provided by the language are relatively close to those on layer 1. For instance, domains, fields, operators and equations can be directly mapped to their discretized counterparts in most cases. In this section, we will only discuss the key differences and extensions of layer 2 when compared to layer 1.

Layer 2 is only used to specify the problem to be solved, but not how it should be solved exactly. Nevertheless, ExaSlang is conceptualized as a multigrid DSL and, as such, it allows preparing certain aspects of a multigrid solver. Most prominently, this includes the discretized operators on the coarser levels as explained in detail later.

The declaration of a domain on layer 2 is similar to the one done on layer 1. However, instead of providing a Cartesian product of intervals, the rectangular domain is specified via an axis-aligned bounding box (AABB). For this, giving only the lower left and upper right corners is sufficient, as illustrated in Listing 3.6.

```plaintext
Domain global from [ 0, 0 ] to [ 1, 1 ]
```

**Listing 3.6:** ExaSlang 2 declaration of a unit square domain `global`.

As on layer 1, fields are tied to a specific, now discretized domain and have an optional expression to initialize their contents. Moreover, they explicitly show their data type and their localization in the grid. Their associated boundary condition declarations don’t need to be enriched with this information since it can be carried over from the linked field. An example is shown in listing 3.7. It is evident, that the syntax is less mathematical and less influenced by LaTeX. In particular, all support for Unicode and expressions lent from LaTeX has been dropped.
Operators show the biggest change between layer 1 and layer 2. As discussed in chapter 1, ExaStencils is focused on operations that can be expressed by stencils or stencil-like constructs. This restriction is also imposed on all layer 2 operators. Nevertheless, different variations are possible. The simplest case is constructing an operator from a given stencil. In this case, the stencil is expressed as a list of offsets and coefficients. Offsets need to be constant and can either be given as an index, e.g. $[1, 0]$ for the right neighbor, or through specialized keywords, e.g. east. An example for declaring a simple stencil is given in listing 3.8. This is also similar to what is generated when automatically discretizing a layer 1 operator, such as the one shown in listing 3.3. More details on inter-layer mapping are given in chapter 4.

These simple stencil constructs are already quite powerful and can be used to implement various applications. They have one major drawback, however, which shows particularly in the context of multigrid solvers: since they are based on constant offsets, stencils mapping between grids of different sizes are not supported. We fix this by introducing what we call *mapping stencils*. Listing 3.9 shows an example for such a stencil – a (linear) restriction for node-based quantities. It can be interpreted in two ways. On the one hand, this simply specifies indices for writing and reading. As evident, the read index is usually a factor of two bigger than the read index. This stems from the fact, that there is...
a factor of two between the number of cells per dimension in two successive grids in the multigrid hierarchy. Of course, other relations are possible as well. On the other hand, this can also be read as a building plan for a matrix. Here, the indices give the row and the column on the entries from which the (non-square) matrix can be constructed.

```
Operator Restriction from Stencil {
    [i0, i1] from [2 * i0 - 1, 2 * i1 - 1]
       with 1.0 / 16.0
    [i0, i1] from [2 * i0 + 0, 2 * i1 - 1]
       with 1.0 / 8.0
    [i0, i1] from [2 * i0 + 1, 2 * i1 - 1]
       with 1.0 / 16.0
    [i0, i1] from [2 * i0 - 1, 2 * i1 + 0]
       with 1.0 / 8.0
    [i0, i1] from [2 * i0 + 0, 2 * i1 + 0]
       with 1.0 / 4.0
    [i0, i1] from [2 * i0 + 1, 2 * i1 + 0]
       with 1.0 / 8.0
    [i0, i1] from [2 * i0 - 1, 2 * i1 + 1]
       with 1.0 / 16.0
    [i0, i1] from [2 * i0 + 0, 2 * i1 + 1]
       with 1.0 / 8.0
    [i0, i1] from [2 * i0 + 1, 2 * i1 + 1]
       with 1.0 / 16.0
}
```

Listing 3.9: ExaSlang declaration of a restriction operator as mapping stencil.

Specifying mapping stencils in this fashion can be somewhat cumbersome. As a remedy, ExaSlang allows setting up such stencils for default cases. Using this mechanism, it is sufficient to specify three things: First, the type of operation, currently restriction or prolongation. Second, the location of the target quantity. This may be any valid grid localization as detailed in section 5.2. Third, an interpolation order. This can be given as a natural number or as a string such as 'linear'. Together with some fixed keywords, declarations such as the ones in listing 3.10 are now possible.
Operators can also be built based on other operators. The simplest case is taking one operator and scaling it with a scalar expression. Other basic operations, such as adding or multiplying two operators, are also supported. Two operators of low dimensionality can be combined into one operator of higher dimensionality using the Kronecker product. Lastly, transposing an operator is possible as well. The latter two options are also used in listing 3.11. Here, the Kronecker product is used to combine two 1D linear interpolation operators into a 2D one. The result of this operation is equivalent to the operator declared in listing 3.9. Moreover, the transpose method is used on this mapping operator and a constant scaling is applied. It is noteworthy, that the levels need to be managed more carefully in this case. This is, partly, the result of a convention in ExaSlang and ExaStencils: when operators are applied to fields, both should have the same level. Transposing an operator changes the size of fields it is applicable to and, in extension, the level of said fields. Thus, the level of the operator is changed as well, which is why Restriction@finer is used to construct Prolongation which is implicitly defined @current. Moreover, Prolongation must not be constructed on the finest level since the Restriction required for that is not available. This is reflected in the explicit declaration level specification @(all but finest). Using default operators, as shown in listing 3.10, doesn’t require this special handling even though the same transpose mechanism is used internally in the code generator. The reason for this is, that a fictional default operator for a finer level can be constructed easily. Nevertheless, the prolongation operator for the finest level can not be used in practice since it would map to an invalid level when used on an applicable field.
3 The ExaStencils Language (ExaSlang)

7 Operator Restriction from
8   kron ( LinInterpolation, LinInterpolation )
9
10 Operator Prolongation@all but finest
11   from 4.0 * transpose ( Restriction@finer )

Listing 3.11: ExaSlang 2 example of combining lower-dimensional stencils into higher-dimensional ones.

Listings 3.9 to 3.11 lead to equivalent code if listing 3.9 is completed with a suitable prolongation operator.

To prepare for the multigrid solver to be specified at the layer 3, variants of the (main) operator(s) are required on the lower levels of the grid hierarchy. This can be either done via re-discretization, which is also the default case coming from layer 1, or via Galerkin coarsening. The latter approach is also shown in listing 3.12 where it is used to replace the operator declaration given in listing 3.8 on all levels but the finest.

1 Operator Laplace@finest from Stencil { /* ... */ }
2 Operator Laplace@all but finest from ( 
3   Restriction@finer 
4     * Laplace@finer 
5     * Prolongation 
6   )

Listing 3.12: ExaSlang 2 example for applying a Galerkin coarsening approach.

Last but not least, we support stencils with variable coefficients that are to be stored separately. This is required if the coefficients are either expensive to calculate or if they depend on other fields which change during the solving process. Conceptually, such a stencil template or stencil field is later split into a coefficient field and a stencil with coefficients being accesses to said field. Listing 3.13 shows how such an operator can be declared. To be able to set up the coefficient field later, providing a localization and a domain is required at this point. In addition to that, a separate function calculating or updating the stencil coefficients has to be set up on layer 3 or layer 4 as detailed in section 3.3.4.
3.3 Layers 1 to 4

Operator Laplace from StencilTemplate
  on Node of global {
    [ 0, 0] =>
    [-1, 0] =>
    [ 1, 0] =>
    [ 0, -1] =>
    [ 0, 1] =>
  }

Listing 3.13: ExaSlang 2 example for setting up a stencil template.

Layer 2 Equations work virtually identical to their layer 1 counterparts. The only difference is that they now present the relations between discretized entities rather than continuous ones. This is also reflected in listing 3.14.

SolEq {
  Laplace * Solution == RHS
}

Listing 3.14: ExaSlang 2 declaration of Poisson’s equation.

Similar to layer 1, declaration types are optional on layer 2 as well due to the distinct syntax of the single declarations. Using this feature and taking the previously introduced listings as basis, a complete layer 2 specification of a finite differences discretization of Poisson’s equation can be formulated as shown in listing 3.15. Note that re-discretization is used in this example to construct the operators on the coarser grids.

global from [ 0, 0 ] to [ 1, 1 ]

Solution with Real on Node of global = 0.0
Solution on boundary = (cos (PI * vf_boundaryPos_x ) - sin ( 2.0 * PI * vf_boundaryPos_y ) )

RHS with Real on Node of global = (PI**2 * cos (PI * vf_nodePos_x ) - 4.0 * PI**2 * sin ( 2.0 * PI * vf_nodePos_y ) )

Laplace from Stencil {
  [ 0, 0] => 2.0 / ( vf_gridWidth_x**2 ) + 2.0 / ( vf_gridWidth_y**2 )

43
3 The ExaStencils Language (ExaSlang)

\[
[-1, 0] \Rightarrow -1.0 / (vf_{gridWidth_x}^2) \\
[ 1, 0] \Rightarrow -1.0 / (vf_{gridWidth_x}^2) \\
[ 0, -1] \Rightarrow -1.0 / (vf_{gridWidth_y}^2) \\
[ 0, 1] \Rightarrow -1.0 / (vf_{gridWidth_y}^2)
\]

\[
SolEq \{
    \text{Laplace} * \text{Solution} == \text{RHS}
\}
\]

**Listing 3.15:** ExaSlang 2 example for a complete specification of the 2D test problem from eqs. (3.1) and (3.2).

### 3.3.3 Layer 3

ExaSlang 3 can be seen as the counterpart of ExaSlang 2: whereas layer 2 is used to specify a problem, layer 3 tells the generator how to solve it. As such, specifying only one without the other is usually not useful. Nevertheless, we chose to honor the hierarchical nature of the language and, to do so, we support the declaration of objects conceptually belonging to layer 2 on layer 3 as well. This also allows the creation of new (auxiliary) fields, e.g. for the residual, and operators, such as prolongation and restriction, which might be necessary to implement a given solver algorithm.

Apart from creating a new field via the mechanisms introduced in the previous section, it is now also possible to clone an already existing field. Listing 3.16 illustrates one of the most common use cases which is creating a field for the residual with the same attributes as the solution field. Attributes include the data type, the domain association, the grid localization and boundary conditions. Conceptually, this is a declaration and, as such, leveled by default. If no level specification is given, the field is created for each level the source field is defined on. For each level, the specific attributes of the source field on the same level is carried over. In some cases, such as the example provided in listing 3.16, altering some aspects of the newly created field is necessary. Here, the boundary conditions are adapted to implement Dirichlet-0 boundaries, which is necessary for most multigrid solvers. When providing the target field, a level specification can be added. If none is given, an implicit @all is assumed and the field adaptation is then performed on all available levels.
Field Residual from Solution
override bc for Residual with 0.0

Listing 3.16: ExaSlang 3 example of introducing an auxiliary field by copying another one. Boundary conditions of the new field are overwritten afterwards.

On all layers of the hierarchy, function calls are supported. This is necessary to, amongst others, support math functions such as \( \sin \) and \( \cos \). Layer 3 is the first layer on which new functions can be defined by users. In ExaSlang, functions work similarly to other programming languages. That is, they have an explicit return type, a parameter list, and a body containing a list of statements that are executed when the function is called. The return type may be \texttt{Unit} and the parameter list empty, and both may be omitted in these cases. Each function parameter must have a name and a type. Different to most other languages, however, only simple data types are allowed here. That implies that we explicitly prohibit passing layer 3 objects as function arguments. This includes fields, operators and other functions, which are all accessible globally anyway. Another key difference to, e.g., C functions is that layer 3 function may be leveled. Contrary to most other objects, however, functions will \textit{not} be leveled with and implicit \texttt{@all} when no level specifier is given. This is necessary to also support regular, un-leveled functions which might serve as an interface to external code components.

Inside the function body, local declarations, if-conditions and fixed-length loops are allowed as motivated in section 3.2. Moreover, operations working on fields and operators may be implemented here. They are designed to be close to Matlab syntax, which is also demonstrated in listing 3.17 where a \( v(3, 3) \)-cycle is implemented. The coarse-grid solver implemented in \texttt{VCycle@coarsest} is omitted here for the sake of compactness, but possible implementations are given in section 4.3.2. As smoother, a red-black Gauss-Seidel (RBGS) is used. Operations only acting on a subset of available entries in a field are denoted by appending \texttt{where} and a suitable condition to any field assignment.

```
Function Smoother@all {
    repeat 3 times {
        Solution += ( diag_inv ( Laplace )
            * ( RHS - Laplace * Solution )
        ) where (i0 + i1) % 2 == 0
    }
```
3 The ExaStencils Language (ExaSlang)

```plaintext
Solution += ( diag_inv ( Laplace )
    * ( RHS - Laplace * Solution )
) where (i0 + i1) % 2 == 1
```

Function VCycle@coarsest {
    /* implementation of a coarse-grid solver */
}

Function VCycle@coarsest + 1 to finest {
    Smoother ( )
    Residual = RHS - Laplace * Solution
    RHS@coarser = Restriction * Residual
    Solution@coarser = 0.0
    VCycle@coarser ( )
    Solution +=
        Prolongation@coarser * Solution@coarser
    Smoother ( )
}

Listing 3.17: ExaSlang 3 implementation of a v(3,3)-cycle using a RBGS smoother. The implementation of the CGS has been omitted.

Using layer 3 functions to implement a specific solver allows for great productivity while key options are still exposed and adaptable by users. Often, however, a boilerplate solver would be sufficient and in this case setting up ones own solver can be an unnecessary burden. To address this, we provide a light-weight interface to configure and automatically generate suitable solver variants. The input is kept very concise – a pair of the unknown to be solved for and equation it is subject to. For systems of PDEs a list of such pairs may be provided. Taking this input, our generator will do its best to find a matching solver. For this, it takes properties of the posed problem, such as the localization of the unknowns, into account. Details can be found in section 4.3.2.

At this point, users can either fully specify their own solver or rely on a fully configured and generated variant. In reality, however, one would often like to have a standard solver, but with only a little change. Thus
we extend the previously introduced interface with the options of configuring the solver and modifying it during generation. First, we implement certain knowledge parameters that can be used to tweak options common to most geometric multigrid solvers. Currently, this includes the following list, which can easily be extended:

- the exit criterion in the form of a target residual reduction and the maximum number of iterations,
- smoother options such as the number of pre- and post-smoothing steps, the employed relaxation factor, if it should be a Jacobi-type smoother and the applied coloring, and
- the type of coarse-grid solver, its target residual reduction as well as its maximum number of iterations.

In general, all of these options can be set as knowledge parameters as well. Other parameters are discussed in the relevant chapters 4 and 6. An example for this mechanism is given in listing 3.18. Here, the convergence criteria are set, and the smoother as well as the coarse grid solver are configured. A v(3,3)-cycle with a damped red-black Gauss-Seidel smoother is generated from this input. On the coarsest grid, an iterative conjugate gradient (CG) solver is employed to reduce the L2 norm of the residual by three orders of magnitude. The resulting generated implementation is similar to the manually implemented one given in listing 3.17.

```plaintext
generate solver for Solution in SolEq with {
    solver_targetResReduction = 1e-10
    solver_maxNumIts = 100
    solver_smoother_jacobiType = false
    solver_smoother_numPre = 3
    solver_smoother_numPost = 3
    solver_smoother_damping = 1.0 // no damping
    solver_smoother_coloring = "red-black"
    solver_cgs = "ConjugateGradient"
    solver_cgs_maxNumIts = 128
    solver_cgs_targetResReduction = 1e-3
}
```

**Listing 3.18:** ExaSlang 3 example of a generate solver block.

Another mechanism is adapting certain parts of the solver directly. For this, we first introduce solver stages which can currently be smoother,
updateResidual, restriction and correction on all levels but the coarsest, and cgs on the coarsest level. The complete multigrid cycle is also modeled as a stage, as is the whole solver. At each stage, one or more layer 3 statements may be added to the front or the back of the stage. Replacing a whole stage with one or more statements is also possible. This also allows replacing a stage with a single function call to a specialized implementation of said stage. As usual, level-dependent construct, such as the solver stages at hand, are regarded separate for each each level. That is, modification can be done for specific levels or, by default, for all applicable ones. Listing 3.19 demonstrates how this approach could be used. Here, a call to the PrintError function on the finest level is injected after each multigrid cycle. Additionally, the restriction stage is fully replaced by a custom function myRestriction on all applicable levels. Note that even though @all is defined for the restriction replacement, the generator will recognize that there is no restriction to replace on the coarsest level and skip it.

```plaintext
1  generate solver for Solution in SolEq with {
2       /* ... */
3      modifiers {
4            append to 'cycle' @finest {
5                PrintError@finest ( )
6            }
7            replace 'restriction' @all {
8                myRestriction@current ( )
9            }
10        }

Listing 3.19: ExaSlang 3 example of modifying a generated solver by appending to the cycle stage and exachanging the generated restriction with a custom one.
```

During solver generation, usually some auxiliary fields are generated. Examples include the residual and the right-hand side on the coarser levels. If this is not desired, the solver generator can be advised to use existing fields in their stead. The same mechanism also works for operators such as the restriction or prolongation. An example for instructing the generator to use a user-defined field is given in listing 3.20. Here, the generated field to hold the residual is replaced with an already existing field. This is done on all levels of the multigrid hierarchy.
Field myResidual from Solution
override bc for myResidual with 0.0

generate solver for Solution in SolEq with {
    /* ... */
} modifiers {
    replace 'gen_ResidualSolution' @all with myResidual
}

Listing 3.20: ExaSlang 3 example of replacing a generated field in a generated solver with a custom one.

The smoother is often the most crucial part of a multigrid solver. Especially for systems of PDEs, it can be quite difficult to compose a suitable smoother. In consequence, it can also be difficult to derive it automatically. For these cases, we support the addition of smoother stages. At each stage, a loop base and a set of unknowns to be solved for at each iteration point is required. The loop base is usually a layer 3 field that determines which part of the grid forms the iteration space – cells, nodes or a given set of faces. Each unknown is given as a field access with an optional offset modifier. Accesses to the same field with different offsets are considered separate unknowns in the smoother stage. From each smoother stage, one kernel is generated. If necessary, coloring, damping or Jacobi-type updates are applied as configured via the knowledge. Multiple smoother stages are executed in sequence. An example for a simple scalar PDE is given in listing 3.21. The smoother runs over the Solution field and solves for only one unknown at each point. Incidentally, a smoother of this form would also be generated if no smoother stage would be provided. More complicated smoothers are required for more complex problems such as those stemming from the domain of CFD. Examples can be found in chapter 6.

smootherStage {
    loopBase Solution solveFor {
        Solution@[0, 0]
    }
}

Listing 3.21: ExaSlang 3 example of defining a smoother stage for a simple point smoother.
3.3.4 Layer 4

Layer 4 holds the complete specification of a problem and the application to solve it. That is, the original input from the other layers is no longer required once a layer 4 representation is generated. Thus, it is also possible to simply start on layer 4 and implement everything there. For this reason, and because layer 4 needs to be present to effectively use the other layers which ultimately map onto it, layer 4 is the most discussed layer in publications until now. The first publication to discuss the major concepts and language features of ExaSlang 4 is [17]. Other publications such as [11], [9] and [12] build on this and explain how the language can be extended to cater to the needs of more complex applications and use cases. They are discussed in detail in sections 5.1, 6.2 and 6.3 respectively.

Conceptually, layer 4 is an extension of layer 3, even though it was developed first. It aims to provide a more computer science focused view on the algorithms and data structures implemented in it. As such, parts of the used data layouts and the employed domain partitioning are exposed. Moreover, data synchronization can be tuned and global operations on vectors and matrices (represented by operators) are replaced by loops over given iteration spaces. All of these aspects are discussed in the following.

On layer 3, field declarations hold the data localization and a data type for each localization point. This is extended on layer 4 to also contain information about the number of points and how they are arranged. Region sizes, such as the number of ghost layers and the number of inner grid points per dimension can be given here. If nothing is provided, default values are derived from the localization. Details on the different regions and their default values are given in section 5.1.1. To avoid overly lengthy field declarations, we decide to split them into field declaration and field layout declaration. The field layout encapsulates information relevant to the underlying data structure, i.e. the base data type, the localization, and the region sizes. Please note that the latter is optional as default values can be derived in most cases. As usual, such declarations are tied to an identifier and are leveled, by default to all levels if no specification is given. In the final field declaration, a reference to the connected domain, a field layout and the boundary condition must be provided. The level for accessing the field layout is carried over from the field level here. An example is given in listing 3.22. Here, we can see that separating the field layout can be beneficial for code complexity if
field declarations must be specialized for certain levels. Moreover, field layouts are intended to be shared between different fields.

```
Layout NodeLayout< Real, Node >@all {
    /* region information */
}

Field Solution< global, NodeLayout,
    0.0 >@all but finest
Field Solution< global, NodeLayout,
    cos ( PI * vf_boundaryPosition_x )
    - sin ( 2.0 * PI * vf_boundaryPosition_y )
    @finest
```


Optionally, a number of field slots can be provided as illustrated in listing 3.23. This will advise the generator to set up multiple instances of the same field. Applications are in Jacobi-type updates and (explicit) time-stepping schemes. Accesses to a slotted field can be modified with a slot modifier such as active, next, previous or a fixed slot given by an integer constant. Advancing to the next slot is possible through advance followed by a field access.

```
Field Solution< global, NodeLayout, 0.0 >[2]@finest
```

Listing 3.23: ExaSlang 4 declaration of a slotted field with two slots.

Operator declarations on layer 4 are similar to their layer 2 counterparts. They have, however, a slightly adapted syntax. This stems from the fact, that the main concept on layer 4 is a stencil, which is applied to a given point. They replace layer 2 operators, which are applied to a whole field at once. Yet, this difference only shows a low impact in practice. This is also evident when comparing the layer 2 and layer 4 variants of three different stencil declarations as shown in listings 3.24 and 3.25.

```
Operator IdentityOffset from Stencil {
    [0, 0] => 1.0
}
Operator IdentityMapping from Stencil {
    [i0] from [i0] with 1.0
}
Operator Twice from IdentityOffset + IdentityMapping
```

Listing 3.24: ExaSlang 2 declaration of an offset stencil, a mapping stencil, and a stencil constructed from other stencils. Analogous to listing 3.25.
Stencil IdentityOffset {
  [0, 0] => 1.0
}
Stencil IdentityMapping {
  [i0] from [i0] with 1.0
}
Stencil Twice from IdentityOffset + IdentityMapping


Similar to layer 2, stencil fields are supported. One key difference, however, is that the declaration of the shape, i.e. the underlying stencil, the field to store the coefficients and their combination into a stencil field have to be done separately. This is illustrated in listing 3.26. As evident, coefficients are not required for the stencil template since the coefficient field will be used for that later. Access to the single entries of the stencil (field) is possible through
- accesses to the coefficient field with an array subscript, e.g. Coefficients[0],
- accesses to the stencil field with an array subscript, e.g. Laplace[0], or
- accesses to the stencil field with a direction modification, e.g. Laplace:[0, 0], which is the recommended way.

Layout CoeffLayout < Vector < 5, Real >, Node > {
  /* ... */
}
Field Coefficients < global, CoeffLayout, None >
Stencil StencilTemplate {
  [0, 0] =>
  [-1, 0] =>
  /* other directions */
}
StencilField Laplace <
  Coefficients => StencilTemplate >

Listing 3.26: ExaSlang 4 declaration of a coefficient field including its layout and a stencil template, as well as their combination into a stencil field.
Functions are declared and used as on layer 3. Inside them, most of the statement types introduced on layer 3 are available as well, as discussed in sections 3.2 and 3.3.3. This includes simple loops, conditionals, as well as variable and value declarations. One key difference is that (direct) assignments to fields are no longer supported. Instead, we switch to a loop centered approach in which kernels are always expressed as a loop over a given iteration space and a set of statements to be executed at each point. On layer 4, this concept is implemented using loops over fields. They allow iterating of a field, or, more precisely, over an underlying grid as specified by the field's layout. In the most basic variant, a simple loop over Solution would be sufficient. For more complex application scenarios, different loop modifications are available. Using where in combination with a suitable expression evaluating to a boolean allows restricting the iteration space to all points fulfilling the given condition. This allows, amongst other things, implementing colored kernels as demonstrated in listing 3.27.

```plaintext
Function RBGS@ all {
    loop over Solution where ( i0 + i1 ) % 2 == 0 {
        Solution += ( omega * diag_inv ( Laplace )
                    * ( RHS - Laplace * Solution ) )
    }
    loop over Solution where ( i0 + i1 ) % 2 == 1 {
        Solution += ( omega * diag_inv ( Laplace )
                    * ( RHS - Laplace * Solution ) )
    }
}
```

**Listing 3.27**: ExaSlang 4 definition of a (damped) RBGS smoother for the discretized Poisson's equation.

This example can be further compacted by using color with or repeat with statements. As shown in listing 3.28, color statements are parameterized with a modulo expression which has to have a constant right-hand side. The equivalent repeat statement simply receives a list of boolean expressions to be applied to enclosed statements.

```plaintext
Function RBGS@all {
    color with (i0 + i1) % 2 {
        loop over Solution {
            Solution += ( omega * diag_inv ( Laplace )
                        * ( RHS - Laplace * Solution ) )
        }
    }
}
```
Listing 3.28: ExaSlang 4 example of using color and repeat statement to implement colored kernels.

Other ways to modify the iteration space are given by starting and ending to extend or shrink the iteration space uniformly, as well as stepping to adapt the step size. Each of them is followed by a constant index with the dimensionality of the domain associated with the field that is looped over. Lastly, reductions need to be specified explicitly on layer 4 at the moment. They can be added to a loop using the with reduction keywords followed by a suitable reduction specification. Its syntax is roughly based on OpenMP and given by an operator and a variable to be reduced, e.g. 
\[(+ : \text{mySum})\]. Finally, other, more specialized modifications, such as attaching communication steps, are discussed in the corresponding sections.

Apart from specifying field updates via single assignments, we also support updating multiple values at once. This is realized via local solve statements, as illustrated in listing 3.29. As shown, they contain a list of tuples of field accesses to be updated and equations they are subject to. Each statement then sets up a linear system of equations from the given information and solves for it. Finally, the obtained results are written back to the specified field accesses. Additional damping can be specified using the relax keyword and Jacobi-type updates can be enforced by adding with jacobi.

// RBGS smoother
color with \((i0 + i1) \% 2\) {
  loop over Solution {
    solve locally {
      Solution => Laplace * Solution == RHS
    }
  }
}
3.4 Inter-Layer Hints

At this point, specifications on the different layers of ExaSlang are possible. They are, however, somewhat separated and it is now the task of the generator to bridge that gap. In order to do so, further information is necessary. Users are able, and in some cases required, to provide this information via so-called hints. In general, they are defined as a (scoped) block of single hints. They can be either specific hints, as discussed in the next paragraphs, or general parameters. In case of the latter, a key-value pair needs to be provided. The key is usually a knowledge parameter, as explained in section 3.5.3. Values need to be expressible in the data type of the parameter referenced. Mechanism-wise, this simply works like a deferred knowledge inlining. It allows, however, grouping information to promote modularization and to let users express their intent more clearly.

3.4.1 Discretization Hints

In order to get from layer 1 to layer 2, discretized counterparts of declared objects need to be set up. For this, discretization hints can be used, where the following variants are available: Fields can be discretized by providing a suitable localization. More information is not required, as they are already connected to a domain and have boundary conditions attached. Operators are discretized with respect to a provided domain. This infers the number of dimensions, e.g. required...
when handling the Laplace operator, and the grid spacing in case of uniform grids. Optionally, a discretization type may be provided, where currently only finite differences are supported. Moreover, an order and a direction for the finite differences can be added. Finally, equations only need to be given, with no further parameters. This will trigger setting up a discretized counterpart, that is an equation where all layer 1 components are replaced with their discretized layer 2 counterparts.

Optionally, renaming can be performed at this stage for all types of discretization. Domains do not need to be discretized by hand, but if a domain with the identifier $\Omega$ is present, it is automatically renamed to global. An example for a discretization hint block is given in listing 3.30.

```plaintext
DiscretizationHints {
  f => RHS on Node
  u   on Node

  op => Laplace with "FiniteDifferences" on $\Omega$ order 2

  uEq

  /* parameters */
}
```

Listing 3.30: ExaSlang 1 example for a discretization hint block for Poisson's equation. $f$ and $\text{op}$ are renamed to RHS and Laplace, respectively. Additional parameters may be added as required.

### 3.4.2 Solver Hints

Solver hints can be used to instruct the generator to automatically set up solver for one or more layer 2 equations. As such, this specification would most likely live on layer 2, but for convenience a specification on layer 1 is also possible. In this case, it is simply carried over when generating layer 2. Solver hints are basically a slimmed down version of layer 3 generate solver statements (c.f. section 3.3.3) in such that they also hold information about equations and the unknowns in them. However, advanced features like solver stage modification or smoother stages are not available here. A simple example for a solver hints block is given in listing 3.31.
3.5 Configuration Files

Listing 3.31: ExaSlang 1 or 2 example for a solver hint block. Additional parameters may be added as required.

```java
SolverHints {
    generate solver for u in uEq
    /* parameters */
}
```

3.4.3 Application Hints

Application hints are the last type of hints supported in ExaSlang. They can be provided on any layer but layer 4 and their main task is to control the generation of ExaSlang 4 code. As such, it is currently sufficient to support only setting knowledge parameters, as with the other types of hints before. One example also given in listing 3.32 is setting the `l4_genDefaultApplication` flag which, as the name suggests, instructs our framework to automatically generate an application function on layer 4 should none be present.

```java
ApplicationHints {
    l4_genDefaultApplication = true
}
```

Listing 3.32: ExaSlang example for an application hint block setting a single parameter.

3.5 Configuration Files

In this section, we discuss the auxiliary files that can be provided to the generator in order to steer the generation process. We give examples for parameters in the specific collections, but this is by no means a complete list.

Parameters always have a valid identifier and their value can be of type Boolean, Double, Int or String. Moreover, collections in the form of `ListBuffer` of the aforementioned types are supported as well. For scalar parameters, users can set them via assignments in the corresponding configuration file. In case of lists, three variants are possible:

- assigning a list in curly braces,
- assigning as single value which is translated to assigning a list with a single entry, or
- appending a single value via a compound assignment.
An example for these mechanisms is given in listing 3.33. Please note, that given (compound) assignments are executed in the order that they are give in. This implies that overwriting previously set parameters is possible, as is appending to the same list-parameter multiple times.

```
produceHtmlLog = true
buildfileGenerators  = { "MakefileGenerator", "ProjectGenerator" }
buildfileGenerators = "MakefileGenerator"
buildfileGenerators += "ProjectGenerator"
```

Listing 3.33: Examples of setting, overwriting and appending to configuration parameters.

Values of parameters holding one or more strings may also be constructed based on other parameters. As listing 3.34 illustrates, this can be done by simply enclosing the parameter to be injected in dollar symbols.

```
configName = "2D_FD_Poisson"
outputPath = ".../generated/$configName/$"
```

Listing 3.34: Example of building one parameter value based on another.

Configuration files might get quite big in some cases. Concurrently, sets of parameters might be shared between different applications. To allow modularization in these cases, we add a basic import functionality to all configuration files. Whenever our generator encounters an import statement, such as the one illustrated in listing 3.35, it recursively descends into that file before handling the rest of the current one. This also enables nested imports. Thus, it is essential that paths are tracked such that the locations of imports are always relative to the file they are imported from.

```
import '../lib/common.knowledge'
```

Listing 3.35: Example of an import statement in a knowledge file.

### 3.5.1 Platform

The platform file(s) summarize parameters specifying the hardware and software of the target computer. In the future, this is planned to be replaced with input in a dedicated TPDL [93] by our collaborators, but as of now there is no production-ready version.
3.5 Configuration Files

First, the name of the system to be executed on can be specified as `targetName`. This parameter can later be used to identify special clusters for which specialized job script generation can be triggered.

Next, the hardware is described. `hw_numNodes` and `hw_numNodesAvailable` are key parameters in cases where clusters are used. The former can be used to restrict execution of a generated program to a given number of nodes, that is it is later incorporated when setting up the job script. The total number of nodes available is merely an annotation and currently not used by our generator. For each node, or in case of a single node, other hardware characteristics can be given. This includes the `hw_cpu_name`, the `hw_cpu_numCPUs` and the `hw_cpu_numCoresPerCPU` as well as the `hw_cpu_frequency` or clock rate. `hw_cpu_bandwidth` can be used to specify the main memory bandwidth. Again, the name is a pure annotation and only intended to structure information in case multiple platform files are managed. Most of the information required can be extracted from the hardware’s manual or the vendor’s websites. In case of one or more GPUs, they can be specified using `hw_gpu_name` and `hw_gpu_numDevices`. Similar to the CPU variant, we support adding hardware characteristics such as `hw_gpu_numCores`, `hw_gpu_frequency` and `hw_gpu_bandwidth`. For certain applications, the compute capability is also important which is why it can be given via `hw_cuda_capability` and `hw_cuda_capabilityMinor`, e.g. 5 and 4, respectively, for compute capability 5.4.

Lastly, the software stack can be detailed as well starting with the `targetOS` (operating system), over the `targetCompiler` and the `targetCudaCompiler` down to the `mpi_variant`. As for compilers, we currently support

- Microsoft visual C++ (MSVC) compiler,
- GNU compiler collection (GCC),
- IBM XL compilers,
- Intel C++ compiler (ICC),
- Cray C++ compiler,
- PGI C++ compiler and
- CLANG.

Additionally, using the compiler version given by `targetCompilerVersion` and `targetCompilerVersionMinor`, our generator is able to deduce other required information such as the maximum version of the supported OpenMP standard or if certain features, such as initializer lists, are supported.
3.5.2 Settings

The main task of the settings file is to configure the in- and output of the generator. Here, at least one DSL input file must be provided using l1file, l2file, l3file or l4file. Each of these parameters is a collection, that is multiple DSL files for any given layer are supported. To facilitate navigating in a folder hierarchy, basePathPrefix can be used, which is automatically prepended to other file locations.

For the output, the most important parameter is outputPath and the name of the binary. To be able to compile the generated code easily, suitable Makefiles, project files for Visual Studio, or CMake files can be generated. This is triggered by adding buildfileGenerators, which, again, is a list to support multiple variants concurrently. Of course, this requires that the chosen target compiler is available on all chosen build systems.

In some cases, external code components such as legacy code and interfaced libraries need to be added to the build process of the generated code. This can be done by modifying pathsInc and pathsLib, as well as by adding additionalIncludes, additionalFiles or additionalLibs. The difference between the latter two is that files are added to the compilation stage whereas libs are only linked after compilation.

Some options also influence the generated code directly, such as parameterizing the timerOutputFile used to emit collected time measurement data to.

Yet another feature is enabling and configuring debugging capabilities. For instance, our generator can be advised to produce a log file in HTML format via produceHtmlLog, where the output file is given by htmlLogFile. Likewise, debugL1File through debugL4File can be specified to get access to snapshots from the generation pipeline at crucial important points. Rudimentary profiling is also available using timeStrategies which advises our framework to measure how expensive each code transformations is.

Last but not least, a user can be set to trigger user-specific code in the generator. One example where this is used is the generation of job scripts where the e-mail address for job notifications can be added automatically depending on the current user.
3.5.3 Knowledge

The knowledge file summarizes all parameters that configure the generated code. It contains various parameters for the different layers of ExaSlang, e.g. those that can be set in the hints. Other big blocks are taken by the automatically applied optimizations and the domain partitioning and parallelization. All in all there are currently well over 200 of them which is why we refrain from discussing them in detail at this point. Instead, we will introduce them in the sections in which they will be required.

One interesting feature of our knowledge handling system is the fact that it can automatically check chosen parameter configurations. This allows enforcing pre-defined constraints and – to a certain extent – self-healing. We support three different responses if a conflict between options is found. First, there are basic sanity checks which only emit a warning if triggered. An example would be a configuration where the number of threads differs from the number of patches they are distributed across. It will not break the generated code, but it might lead to a load imbalance and, thus, to a performance deterioration. Second, there are cases where a chosen configuration might break the generation process or the generated code, but fixing it is possible. Then, we adapt the parameters accordingly and issue a warning. One example for this is if the number of multigrid levels to be optimized with certain techniques is larger than the total number of levels. Fixing it is easy, by simply restricting the first parameter to the value of the second. Third, there are issues that can not be resolved. Here, we have the most severe respond which is exiting with an error. For instance, using multiple blocks without enabling MPI is currently unsupported.

At the moment, the described mechanism is only available after first reading the user-provided configuration files. That is, after setting parameters later, e.g. in hints or generate solver blocks, no checks are performed.

3.6 Contribution

In this chapter, we presented our novel DSL geared towards the specification of PDEs, their discretization and multigrid methods to solve them. Our multi-layered approach allows specializing each language part to the needs of different user groups. Relying on an external DSL gives us the freedom to design the different layers accordingly. Moreover,
we sketched how the automatic mapping between the layers can be guided by users from the DSL level via hints. Different configuration files complete the specification.

My main contributions are:

• Implementation and integration of ExaSlang 1. This work is partly based on the thesis of Ewald Flad [26], but the language specification has been redesigned to fit the other layers and to better reflect the presented core concepts. Christian Schmitt has implemented parser and lexer support for unicode symbols.

• Conceptualization, implementation and integration of ExaSlang 2 and ExaSlang 3.

• Conceptualization, implementation and integration of ExaSlang 4 in collaboration with Christian Schmitt. An early version of the language has been published [17].

• Conceptualization, implementation and integration of the hint system.

Efforts to realize the remaining items have mostly been done in collaboration with Christian Schmitt. All work has been supervised by Harald Köstler.
4 Code Generator Part I: Workflow

This chapter provides a detailed look on how ExaSlang input is processed by our generator. As such, it is the logical continuation of chapter 3 and discusses the concepts from section 2.4 in its context.

4.1 Data Structures and Transformations

The first step is setting up the node classes which can then be used to construct an abstract syntax tree (AST). Similar to what is discussed in section 2.4, we create a node type hierarchy with a general node type \texttt{Node} at the top. In our context, \texttt{Node} is modeled as abstract, that is it can not be instantiated. To appropriately reflect our layered language design, we create layer-specific nodes as subclasses of \texttt{Node}, namely \texttt{L1\_Node}, \texttt{L2\_Node}, \texttt{L3\_Node} and \texttt{L4\_Node}. For the sake of brevity, we abbreviate layer-dependent names from hereon by replacing the concrete layer index with an \texttt{X}, e.g. \texttt{LX\_Node}. Next, we introduce (also abstract) subclasses representing expressions and statements as \texttt{LX\_Statement} and \texttt{LX\_Expression} respectively. Based on this, more specialized classes may be added. Examples for node types that exist on all layers are literals such as \texttt{LX\_IntegerLiteral}, and binary operations such as \texttt{LX\_Addition}. Of course, there are also node types which only occur on specific layers. \texttt{LX\_IfCondition}, for instance, is only available on layer 3 and layer 4. Further node types will be discussed as required in the following sections.

It is especially important to take care that the AST is always a valid tree. That is, there must not be any cyclic dependencies and each node must only have exactly one parent (with the exception of the root node of course). This poses a challenge when modeling accesses to global objects, such as fields, as AST nodes. Usually one would simply add the global object as member of the node class, i.e. as a child of a node
instance. This, however, leads to incorrect behavior but there are multiple options to circumvent this issue. Storing only the identifier of the referenced object breaks the invalid link but accessing information of the object requires searching the AST for it. This can become quite costly. Alternatively, objects could not be referenced but instead be added as (deep copies. This leads to a strong increase in node counts, which, in turn, results in longer generation times. Moreover, changing a knowledge object becomes harder, since each change must be synchronized with each copy. Keeping these drawbacks in mind, we decided for an alternative approach in ExaStencils. Orthogonally to our forest of ASTs, we also keep track of what we call knowledge objects. Amongst them are domains, fields and operators as introduced in the previous chapter. For each type of object, a dedicated knowledge collection is managing all applicable knowledge object instances. Since knowledge objects are not subclasses of Node, and as such no valid targets for transformations, they can be freely referenced from within the AST. This also leads to a smaller ASTs and, subsequently, to accelerated generator execution times. The main drawback of this approach is that any actual Node instances inside knowledge objects are effectively hidden from the tree and, in consequence, from any visitor implemented. This has to be kept in mind when designing transformations and transformation pipelines.

Concerning transformations, we follow the concepts introduced in section 2.4.1. As before, one or more transformations are aggregated in a strategy. Due to our layered structure, transformations, and subsequently strategies, are also layer specific. Although this requires some code duplication, it ensures that only nodes of the current layer are targeted and that all replacements are associated with the current layer as well. To facilitate implementing and maintaining layer-dependent code, our Meta tool presented in detail in section 7.4 can be used. It allows duplicating existing implementations for other layers and synchronizing changes across multiple ones.

There are some exceptions to the presented transformation paradigm. They are necessary when the whole AST needs to be processed at once. This happens, e.g., when the AST from one layer is mapped to a counterpart representation on another layer. In this case, each node in the AST is responsible for recursively handling each of its children and ultimately itself. A more detailed description of these operations is given in the relevant sections.
4.2 Workflow

In ExaStencils, just as in most other source-to-source compilers, the start of the processing pipeline is given by a specialized lexer and parser. Separate implementations are required for each layer of ExaSlang since the language specifications are highly specialized towards the needs of respective user groups. The resulting parse tree is already a valid AST in our framework. Inside it, there are only node types associated with the current layer to ensure consistency. Likewise, transformations are not allowed to replace nodes from one layer with nodes from another.

For each layer, a specialized layer handler has to be implemented. Its job is to process the input provided by the parser via applying a series of transformations. Here, we discuss a typical pipeline in ExaStencils. In practice, this pipeline is heavily extended and adapted to the specifics of the target layer.

First, we perform some preparatory steps such as scanning the AST for inline knowledge nodes which are then processed and removed.

Next, we prepare handling declarations. For that, it is necessary to resolve some level aliases such as @all, @coarsest and @finest. Level specifications including not are also handled at this stage. Afterwards, all declarations hold either no level specification, a single level or a list of single levels. This allows applying an unfold strategy to duplicate declarations for multiple levels. In case of a single level, nothing is done in the actual strategy. In case of no specification, a default for the type of declaration is inserted. Currently, we assume an implicit @all. For this case, and for the case of a specified list, we create duplicates for each level the declaration is valid for.

At this point, we can prepare resolving accesses to knowledge objects. Since all (leveled) declarations are now tied to a specific level, they form a valid level scope. This is required in the next step where @current aliases are resolved in accesses. Relative level specifications such as @coarser and @finer are handled in the same go.

The next step is assembling a list of declared objects. For this, we keep a collection of identifier and level pairs – or just identifiers in case the knowledge type is not leveled – in each knowledge collection. Since each leveled declaration has exactly one level at this point, the strategies performing the registration are straightforward.

Once all objects have been registered, the resolution of unknown accesses can be started. A resolution to full accesses is not yet possible, since the knowledge objects to be referred to are only set up in the next
step. Instead, we replace unresolved accesses with matching identifiers and level specifications to what we call future accesses. If no level is specified in the resolution, but the identifier matches the registered objects, we assume an implicit @current. For this, we keep track of the level scope in each of the strategies.

Now, all accesses that will refer to knowledge objects in the future are marked accordingly. This allows starting the last step, which is a stepwise resolution of the declarations and the concomitant integration with the knowledge collections. In tandem with this, future accesses to already integrated objects can be promoted to full accesses. We apply this step in an iterative fashion until all declarations have been processed.

On layers where function declarations are available, more steps are necessary. First, leveled function declarations are unfolded such that each function has no or only a single level. In case no level modifier was given, no implicit @all is assumed. This step is usually done before knowledge objects are handled since resolution of accesses to them require the level scopes of unfolded functions. Afterwards, function calls can be resolved, where four different scenarios can occur:

<table>
<thead>
<tr>
<th></th>
<th>call without level specification</th>
<th>call with level specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>plain (un-leveled)</td>
<td>resolution is straight-forward</td>
<td>ignore the provided level and issue a warning</td>
</tr>
<tr>
<td>function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>leveled function</td>
<td>assume an implicit @current</td>
<td>resolution is straight-forward</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Possible scenarios when resolving function calls in ExaSlang.

At the end of the pipeline, a debug output of the current layer’s AST is emitted. This includes printing declarations of all existing knowledge objects as well. The last step is then progressing to the next layer, as discussed in the next section.

4.3 Transition Between Different Layers

Transitioning between different layers can not be implemented using transformations since new nodes must be associated to the same layer as the node they replace. To circumvent this limitation, we decide for
an alternative approach: Each node that may be mapped to the next layer implements the progressable trait. The only requirement of this trait is that a progress function is implemented. This function is responsible for setting up a counterpart of the current node on the next layer and return it. For this, it usually recursively calls the progress function on some or all of its children. Once implemented for all applicable nodes, this allows transitioning the whole AST in a single step. Before doing so, however, knowledge collections need to be progressed as well. Otherwise, progressing of accesses is not possible since they can not be mapped to valid knowledge objects on the new layer. Progression itself is straightforward since it only requires calling progress on each knowledge object and adding the returned object to the appropriate knowledge collection on the next level. As an optimization, we additionally store a reference to the progressed object in the original one. This facilitates the progression of nodes referencing the object considerably.

4.3.1 Layer 1 to Layer 2

Processing knowledge objects from layer 1 is somewhat special since they have to be discretized at this stage. For domains, this can be done easily since the imposed grid is controlled by knowledge parameters. Thus, processing corresponds to simply setting up a layer 2 domain with the same geometric coordinates. Identifier adaptation, e.g. from $\Omega$ to $\text{global}$ is done at this point as well.

More information is necessary for the discretization of the remaining knowledge objects. As discussed, it is given through the hints provided by users in the domain-specific language (DSL) file. We find them by scanning the AST for any discretization hint nodes, which are then processed one after another. In the current version, it is necessary for users to ensure the correct ordering of discretizations. For instance, an equation can only be discretized once the fields and operators it refers to have been handled. We recommend the following order:

First, fields are handled. Since fields are leveled, the discretization may be level-specific as well, although it usually makes little sense to have varying discretizations for the same field on different levels. If no level specification is provided, we assume all levels the field with the corresponding identifier is defined for. The only information missing for setting up the corresponding layer 2 fields is the localization, which is given by the hint. Other attached layer-specific members, such as
boundary conditions and initialization expressions, are progressed at this stage as well.

Next, operators can be handled. In the current version, only finite difference schemes are supported. They can, however, be applied in an automated fashion through functions provided by the generator. A prototype implementation of such an approach has been done by Ewald Flad in the scope of his bachelor thesis [26]. The thesis also discusses how to get from a single partial derivative to a constant stencil. Based on this, we have integrated the proposed method into our code generator. The resulting module works in three steps. As preparation, a simplify strategy is applied to the operator expression. The new expression is then scanned for partial derivatives. For each found node, we create a stencil representing the discretized derivative using the mechanisms discussed in [26]. In our context, however, directly replacing the node is not possible since stencils cannot be part of an AST. Instead, we set up an access node referring to the newly created stencil, thereby wrapping it. As the stencil is only temporary there is no need to add it to the stencil collection.

After the whole operator expression has been handled, we squash it to form a single stencil. For this, it is beneficial to first promote all scalars in the expression to stencils with a single entry and zero offset. This generates an expression only containing stencils and arithmetic operations on them. Additions and subtractions are straightforward to handle. Multiplications are resolved such that their result is the stencil that would describe a matrix which could also be obtained by multiplying the two matrices represented by the input stencils. Divisions between stencils are realized by first inverting the second stencil and then resolving the resulting multiplication.

The main limitations of the current implementation are that it only works for uniform grids and that non-linear equations are not supported.

Equations are the only remaining knowledge objects once fields and operators have been progressed. Given that their left- and right-hand sides can now be progressed, it is possible to set up a layer 2 equation with them. Its identifier is, if necessary, altered according to the discretization hint.

After handling discretization hints, they are consumed, i.e. removed from the AST. At this point, only very few nodes remain in the AST. This stems from the fact, that only declarations of knowledge objects
and hint specifications can be parsed on layer 1. And, similar to the
discretization hints, knowledge object declarations have already been
consumed as well. In consequence, only other types of hints remain in
the AST and, thus, progressing it to layer 2 is straight-forward.

4.3.2 Layer 2 to Layer 3

Conceptually, solver hints are realized between layer 2 and layer 3. In
practice, however, it is easier to progress the whole hint block to layer 3
where it becomes a generate solver statement. Using it, a suitable
solver can then be constructed, analogously to how it would be done
from user-defined solver specifications. We discuss this process in the
following. Please note that the discussion is restricted to the case of a
single, scalar partial differential equation (PDE). An extension for sys‐
tems of equations is, however, comparably straight-forward and shown
later in chapter 6.

To automatically set up multigrid solvers from the given information,
multiple steps are necessary. Fields need to be set up, boundary con‐
ditions and equations need to be adapted, and new operators need to
be introduced. Ultimately, functions implementing the algorithm itself
can be generated.

First, temporary fields are created. This includes residual fields for all
levels and right-hand sides for all levels but the finest. To better reflect
the concept of multigrid solvers, we add fields to represent the error on
the coarser levels. All fields are duplicates of the solution field, that is
they inherit localization, data type and boundary conditions.

Next, boundary conditions are adapted. For the generated right-hand
side fields, no boundary condition is required which is why it is elimi‐
nated. Residual and error fields, on the other hand, required boundary
conditions, but generally they need to be adapted. Otherwise, an artifi‐
cial error or residual would be enforced at the boundaries. Currently, we
support three cases: Dirichlet boundary conditions are replaced with
Dirichlet-0 ones. Neumann boundary conditions are kept, including
the specified order. Nothing needs to be done if no boundary condi‐
tions were specified. There is one case where this does not work yet – if
users provide their own functions to implement specialized boundary
handling.

Another vital part for every multigrid solver are the inter-grid oper‐
ators, namely restriction and prolongation. For setting them up auto‐
matically, two things need to be specified: the order of interpolation
and the localization of data to be interpolated. The latter can be carried over from the solution field. At this point, we assume that one can be created by transposing the other. The order itself is, if not specified otherwise, dependent on the localization of the data. For cell-centered values, a constant interpolation is chosen, whereas node-centered values are handled by linear interpolation. 1D variants of default operators are then combined via Kronecker products to build operators of the required dimensionality. This approach also allows easily incorporating face-centered values as they can be interpreted as cell-centered in all but one dimension in which they are then node-centered.

After adding auxiliary objects, the governing equations on the coarser levels are reformulated as error equations. For this, we first replace references to the solution fields with ones targeting the corresponding error fields. The equations are then transformed into a sum of products. This allows easily reordering them such that all terms depending on the unknowns are on the left side and all other terms on the right. Now it is easy to replace the original right-hand sides with accesses to the previously generated right-hand side fields. While not strictly necessary, we also reorder the equation on the finest level as described since this facilitates the residual computation which is required later on.

This concludes the preparatory steps and allows generation of the solver functions. On each level but the coarsest, we generate a default cycle function which follows the algorithm in algorithm 2.

**Algorithm 2:** Multigrid algorithm implemented by our automatic solver generation.

1. apply smoother
2. update residual
3. restrict residual to the right-hand side on the coarser level
4. initialize error on the coarser level to 0
5. recursively descend
6. correct the solution or error with the error on the coarser level
7. apply smoother

The single parts of the algorithm are realized as follows. Updating the residual is easy given that we reordered the equations before – simply subtracting the left-hand side from the right-hand side and assigning this to the generated residual field is sufficient. Setting up the coarser right-hand side is done with a convolution of the previously introduced
restriction operator and the residual field. Initializing the coarser error field and descending recursively is straight-forward. For the correction step, it is important to add the prolongated values to the right field: the unknown field on the finest level or the error field on the other ones. Otherwise, another convolution of the generated prolongation operator with the coarser error field is sufficient. Lastly, the smoother needs to be set up. If no smoother stages are given, the generator will set up a default smoother. For this, we rely on the solve locally mechanic introduced in section 3.3.4, which is applied to the governing equation. The smoother is also tuned depending on the chosen parameters for damping and coloring, and the specification of a Jacobi-type update. Lastly, an enclosing loop is added to realize the prescribed number of pre- or post-smoothing steps.

A slightly modified example for a generated cycle function for the test problem of Poisson's equation is given in listing 4.1. The cycles employ a damped red-black Gauss-Seidel (RBGS) applied three times.

```plaintext
Function gen_mgCycle@ finest 
  repeat 3 times { 
    color with { 
      ( i0 + i1 ) % 2, 
      solve locally relax 0.8 { 
        Solution => Laplace * Solution == RHS 
      } 
    } 
  } 

gen_residual_Solution = RHS - Laplace * Solution 
gen_rhs_Solution@coarser = ( 
  gen_restrictionForRes_Solution * gen_residual_Solution ) 
gen_error_Solution@coarser = 0.0 
gen_mgCycle@coarser ( ) 
Solution += ( 
  gen_prolongationForSol_Solution@coarser * gen_error_Solution@coarser 
) 
/* smoother as above */
```
Function gen_mgCycle@(all but (coarsest
and finest)) {
    repeat 3 times {
        color with {
            ( i0 + i1 ) % 2,
            solve locally relax 0.8 {
                gen_error_Solution =>
                Laplace * gen_error_Solution
                == gen_rhs_Solution
            }
        }
    }
    gen_residual_Solution = (  
        gen_rhs_Solution
        - Laplace * gen_error_Solution )
    gen_rhs_Solution@coarser = (  
        gen_restrictionForRes_Solution
        * gen_residual_Solution )
    gen_error_Solution@coarser = 0.0
    gen_mgCycle@coarser ( )
    gen_error_Solution += (  
        gen_prolongationForSol_Solution@coarser
        * gen_error_Solution@coarser )
    /* smoother as above */
}

Listing 4.1: ExaSlang 3 example of cycle functions on different levels generated from a generate solver statement. Implicitly deductable level specifications have been omitted and level aliases have been introduced to promote readability. The coarse-grid solver (CGS) has been omitted as well.

**Coarse Grid Solver**

Adding a coarse grid solver is the final step. Here, multiple variants are available to users. In the following, we assume that the equation to be solved is always given by $A \bar{x} = b$. Moreover, the generator will always set up a function that calculates the norm of the residual on the coarsest field, as illustrated in listing 4.2.
4.3 Transition Between Different Layers

Function NormResidual@coarsest : Real {
    return sqrt ( dot ( Residual, Residual ) )
}

Listing 4.2: ExaSlang 3 example to a function calculating the L2 norm of the residual.

Conjugate gradient (CG)

The default option is an implementation based on the conjugate gradient (CG) method which was originally published in [69]. It can be formulated as algorithm 3, where $A$ is required to be symmetric and positive-definite.

Algorithm 3: Conjugate gradient algorithm.

1. $r^0 = b - Ax^0$
2. $p^0 = r^0$
3. $k = 0$
4. repeat
5.   $\alpha^k = (r^k \cdot r^k) / (p^k \cdot Ap^k)$
6.   $x^{k+1} = x^k + \alpha^k p^k$
7.   $r^{k+1} = r^k - \alpha^k Ap^k$
8.   if $r^{k+1}$ is sufficiently small then exit loop
9.   $\beta^k = (r^{k+1} \cdot r^{k+1}) / (r^k \cdot r^k)$
10. $p^{k+1} = r^{k+1} + \beta^k p^k$
11. $k = k + 1$
end

For the test problem of Poisson’s equation, our generator will implement algorithm 3 to yield a specification similar to the one depicted in listing 4.3. Note, that this solver is more or less the same for all cases where a single, scalar PDE is solved for. To accelerate the solver, the product of $Ap^k$ is stored in $ap$. 
Field p @coarsest from Residual
Field ap@coarsest from Residual

Function VCycle@coarsest {
  // init residual
  Residual = RHS - Laplace * Solution
  Var curRes : Real = NormResidual ( )
  Var initRes : Real = curRes
  Var nextRes : Real = 0.0

  if ( curRes == 0.0 ) { return }

  // init variables and fields
  Var alpha : Real = 0.0
  Var beta : Real = 0.0

  p = Residual

  // main loop
  Var curStep : Int = 0
  repeat 128 times count curStep {
    ap = Laplace * p
    alpha = ( dot ( Residual, Residual )
              / dot ( p, ap ) )
    Solution += alpha * p
    Residual -= alpha * ap

    nextRes = NormResidual ( )
    if ( nextRes <= 1e-3 * initRes ) { return }

    beta = nextRes**2 / curRes**2
    p = Residual + beta * p
    curRes = nextRes
  }

  print ( "Maximum number of cgs iterations (", 128, ") was exceeded" )
}

Listing 4.3: ExaSlang 3 implementation of the CG algorithm shown in algorithm 3 for Poisson’s equation.
Conjugate Residual (CR)

Another solver variant, that is quite similar to the CG method, is given by the conjugate residual (CR) technique. Based on [92], we can derive the formulation given in algorithm 4. For this algorithm, \( A \) is required to be Hermitian.

**Algorithm 4:** Conjugate residual algorithm.

1. \( r^0 = b - Ax^0 \)
2. \( p^0 = r^0 \)
3. \( k = 0 \)
4. repeat
   5. \( \alpha^k = \frac{(r^k \cdot Ar^k)}{(Ap^k \cdot Ap^k)} \)
   6. \( x^{k+1} = x^k + \alpha^k p^k \)
   7. \( r^{k+1} = r^k - \alpha^k Ap^k \)
   8. if \( r^{k+1} \) is sufficiently small then exit loop
   9. \( \beta^k = \frac{(r^{k+1} \cdot Ar^{k+1})}{(r^k \cdot Ar^k)} \)
   10. \( p^{k+1} = r^{k+1} + \beta^k p^k \)
   11. \( Ap^{k+1} \) can be computed from \( Ar^{k+1} + \beta^k Ap^k \)
   12. \( k = k + 1 \)
5. end

Our generator will realize algorithm 4 for the discretized Poisson’s equation similar to the depiction in listing 4.4. As evident, temporary fields that store the products \( Ap \) and \( Ar \) are introduced as \( ap \) and \( ar \) respectively. To improve performance, the dot product between \( r \) and \( Ar \) is buffered since it is required twice, once to calculate \( \alpha \) and then again to update \( \beta \).

1. Field \( p \) @coarsest from Residual
2. Field \( ap \) @coarsest from Residual
3. Field \( ar \) @coarsest from Residual
4. Function VCycle@coarsest {
   // init residual
   Residual = RHS - Laplace * Solution
   Var curRes : Real = NormResidual ( )
   Var initRes : Real = curRes
   Var nextRes : Real = 0.0
if ( curRes == 0.0 ) { return }

// init variables and fields
Var alpha : Real
Var beta : Real

p = Residual
ap = Laplace * p
ar = Laplace * Residual

// main loop
Var curStep : Int = 0
repeat 128 times count curStep {
    // buffer r_ar
    Var r_ar : Real = dot ( Residual, ar )
    alpha = r_ar / dot ( ap, ap )
    Solution += alpha * p
    Residual -= alpha * ap
    nextRes = NormResidual ( )

    if ( nextRes <= 1e-3 * initRes ) { return }  

    ar = Laplace * Residual
    beta = dot ( Residual, ar ) / r_ar
    p = Residual + beta * p
    ap = ar + beta * ap
    curRes = nextRes
}

print ( "Maximum number of cgs iterations (", 128, ",") was exceeded" )
}

Listing 4.4: ExaSlang 3 implementation of the CR algorithm shown in algorithm 4 for Poisson’s equation.

Minimal Residual (MINRES)

For cases where neither a CG nor a CR is applicable, we provide a minimal residual (MINRES) variant as well. It was first published in [84], but we base our implementation on the variant presented in [72] to obtain our derivation shown in algorithm 5. For this solver, $A$ is only re-
4.3 Transition Between Different Layers

required to be symmetric. Checking at the algorithm definition, and considering the first iteration, a potential problem may arise: for certain parameters and fields, a version from the previous time step is required. Looking closely, however, one finds that all these values are multiplied with 0 in the first iteration and, thus, only need to be initialized with any (valid) values.

Algorithm 5: Minimal residual algorithm.

\[ r^0 = b - Ax^0 \]
\[ \zeta^0 = ||r^0|| \]
\[ v^0 = r^0 / \zeta^0 \]
\[ p^0 = 0 \]
\[ \beta^0 = 0 \]
\[ c^0 = 1 \]
\[ s^0 = 0 \]
\[ k = 0 \]

repeat

\[ v^{k+1} = Av^k - \beta^k v^{k-1} \]
\[ \alpha^k = (v^{k+1})^T v^k \]
\[ v^{k+1} = v^{k+1} - \alpha^k v^k \]
\[ \beta^{k+1} = \frac{||v^{k+1}||}{||v^{k+1}||} \]
\[ v^{k+1} = v^{k+1} / \beta^{k+1} \]
\[ \rho_1^k = s^{k-1} \cdot \beta^k \]
\[ \rho_2^k = c^k \cdot c^{k-1} \cdot \beta^k + s^k \cdot \alpha^k \]
\[ \rho_3^k = c^k \cdot \alpha^k - s^k \cdot c^{k-1} \cdot \beta^k \]
\[ \tau^k = |\rho_3^k| + |\beta^{k+1}| \]
\[ \nu^k = \tau^k \sqrt{(\nu_3^k / \tau^k)^2 + (\beta^{k+1} / \tau^k)^2} \]
\[ c^{k+1} = \rho_3^k / \nu^k \]
\[ s^{k+1} = \beta^{k+1} / \nu^k \]
\[ \rho_3^{k+1} = \nu^k \]
\[ p^{k+1} = (v^k - \rho_1^k p^{k-1} - \rho_2^k p^k) / \rho_3^k \]
\[ x^{k+1} = x^k + c^{k+1} \cdot \zeta^k \cdot p^{k+1} \]
\[ \zeta^{k+1} = -s^{k+1} \cdot \zeta^k \]

if \( \zeta \) is sufficiently small then exit loop

\[ k = k + 1 \]
end
When implementing algorithm 5, it is important to buffer some values and fields since they are required multiple times. For instance, $\beta^k$ is still required even after $\beta^{k+1}$ is computed. Versions of $c$, $s$, $v$ and $p$ must be available for the current, the previous and the next iteration concurrently. This introduces the need for intermediaries which we postfix with \texttt{Old} and \texttt{New} in our implementation. Otherwise, a translation of the algorithm is straight-forward and results in a representation similar to the one shown in listing 4.5.

```plaintext
Field pOld@coarsest from Residual
Field p @coarsest from Residual
Field pNew@coarsest from Residual
Field vOld@coarsest from Residual
Field v @coarsest from Residual
Field vNew@coarsest from Residual

Function VCycle@coarsest {
   // init residual
   Residual = RHS - Laplace * Solution
   Var curRes : Real = NormResidual()
   Var initRes : Real = curRes

   if ( curRes == 0.0 ) { return }

   // init variables and fields
   Var alpha : Real
   Var beta : Real
   Var betaNew : Real = 0.0
   Var cOld : Real
   Var c : Real = 1.0
   Var cNew : Real = 1.0
   Var sOld : Real
   Var s : Real = 0.0
   Var sNew : Real = 0.0

   Var rho1 : Real
   Var rho2 : Real
   Var rho3Tld : Real
   Var tau : Real
   Var nu : Real
   Var rho3 : Real
}
```
4.3 Transition Between Different Layers

\[ v = 0.0 \]
\[ \frac{\text{Residual}}{\text{initRes}} = v_{\text{New}} \]
\[ p = 0.0 \]
\[ p_{\text{New}} = 0.0 \]

// main loop
Var curStep : Int = 0
repeat 128 times count curStep {
    beta = beta_{\text{New}}
    v_{\text{Old}} = v
    v = v_{\text{New}}
    v_{\text{New}} = \text{Laplace} \ast v - \beta \ast v_{\text{Old}}
    alpha = \text{dot} ( v_{\text{New}}, v )
    v_{\text{New}} -= alpha \ast v
    beta_{\text{New}} = \sqrt{\text{dot} ( v_{\text{New}}, v_{\text{New}} )}
    v_{\text{New}} //= beta_{\text{New}}
    c_{\text{Old}} = c
    c = c_{\text{New}}
    s_{\text{Old}} = s
    s = s_{\text{New}}
    rho1 = s_{\text{Old}} \ast beta
    rho2 = c \ast c_{\text{Old}} \ast beta + s \ast alpha
    rho3_{\text{Old}} = c \ast alpha - s \ast c_{\text{Old}} \ast beta
    tau = \text{fabs} ( rho3_{\text{Old}} ) + \text{fabs} ( beta_{\text{New}} )
    nu = \tau \ast \sqrt{\frac{( rho3_{\text{Old}} / \tau )^{\ast 2} + ( beta_{\text{New}} / \tau )^{\ast 2}}{}}
    c_{\text{New}} = \frac{rho3_{\text{Old}}}{nu}
    s_{\text{New}} = \frac{beta_{\text{New}}}{nu}
    rho3 = nu
    p_{\text{Old}} = p
    p = p_{\text{New}}
    p_{\text{New}} = ( v - rho1 \ast p_{\text{Old}} - rho2 \ast p ) / rho3
    \text{Solution} += c_{\text{New}} \ast \text{curRes} \ast p_{\text{New}}
    \text{curRes} *= -s_{\text{New}}

    if ( \text{fabs} ( \text{curRes} ) <= 1e-3 \ast \text{initRes} ) {
        return
    }
}
Listing 4.5: ExaSlang 3 implementation of the MINRES algorithm shown in algorithm 5 for Poisson’s equation.

The present implementation could be optimized by using slotted fields which, however, is future work.

**Biconjugate Gradient Stabilized (BiCGSTAB)**

Lastly, we introduce a biconjugate gradient stabilized (BiCGSTAB) solver which was first published in [102]. After some minor modifications to the original definition we wind up with the specification in algorithm 6. For this case, $A$ may be non-symmetric.

**Algorithm 6: Biconjugate gradient stabilized algorithm.**

\begin{verbatim}
1 $r^0 = b - Ax^0$
2 $\hat{r}^0 = r$
3 $\rho^0 = 1$
4 $\omega^0 = 1$
5 $v^0 = 0$
6 $p^0 = 0$
7 repeat
8     $\rho^{k+1} = \hat{r}^0 \cdot r^k$
9     $\beta^k = (\rho^{k+1}/\rho^k) (\alpha^k/\omega^k)$
10    $v^{k+1} = Ap^{k+1}$
11    $\alpha^k = \rho^{k+1} / (\hat{r}^0 \cdot v^{k+1})$
12    $s^k = r^k - \alpha^k v^{k+1}$
13    $t^k = As^k$
14    $\omega^{k+1} = (t^k \cdot s^k) / (t^k \cdot t^k)$
15    $x^{k+1} = x^k + \alpha^k p^{k+1} + \omega^{k+1} s^k$
16    $r^{k+1} = s^k - \omega^{k+1} t^k$
17     if $r^{k+1}$ is sufficiently small then exit loop
18 end
\end{verbatim}
4.3 Transition Between Different Layers

When implementing algorithm 6, $\rho$ is the only value that needs to be buffered due to its old value being required after updating it. Doing so, our generator is able to emit a representation similar to the one depicted in listing 4.6.

```plaintext
Field v @coarsest from Residual
Field p @coarsest from Residual
Field h @coarsest from Residual
Field s @coarsest from Residual
Field t @coarsest from Residual
Field resHat@coarsest from Residual

Function VCycle@coarsest {
    // init residual
    Residual = RHS - Laplace * Solution
    Var curRes : Real = NormResidual()
    Var initRes: Real = curRes

    if ( curRes == 0.0 ) { return }

    // init variables and fields
    Var alpha : Real = 1.0
    Var beta : Real = 1.0
    Var rho : Real
    Var rhoNew : Real = 1.0
    Var omega : Real = 1.0

    resHat = Residual
    v = 0.0
    p = 0.0

    // main loop
    Var curStep : Int = 0
    repeat 128 times count curStep {
        rho = rhoNew
        rhoNew = dot ( resHat, Residual )
        beta = ( rhoNew / rho ) * ( alpha / omega )
        p = Residual + beta * ( p - omega * v )
        v = Laplace * p
        alpha = rhoNew / dot ( resHat, v )
        s = Residual - alpha * v
        t = Laplace * s
        omega = dot ( t, s ) / dot ( t, t )
    }
}
```
Solution = Solution + alpha * p + omega * s
Residual = s - omega * t
curRes = NormResidual()

if ( curRes <= 1e-3 * initRes ) { return }
}

print ( "Maximum number of cgs iterations (", 128, ") was exceeded" )

Listing 4.6: ExaSlang 3 implementation of the BiCGSTAB algorithm shown in algorithm 6 for Poisson’s equation.

Conclusion

In conclusion, the choice of the coarse grid solver most likely depends on the properties of $A$:
- if $A$ is symmetric and positive‐definite, a CG solver is suitable,
- if $A$ is Hermitian, a CR solver is suitable,
- if $A$ is only symmetric, a MINRES solver is suitable, and
- if $A$ is non‐symmetric, a BiCGSTAB solver is suitable.
Currently, our generator is not able to derive properties of the system matrix automatically.

4.3.3 Layer 3 to Layer 4

Layer 4 introduces and exposes concepts for data partitioning and parallelization. One prominent result is the usage of field layouts. Setting them up automatically is possible using our code generation pipeline.

On layer 3, field layouts do not exist yet. Thus, we first progress to layer 4 introducing some placeholder layouts which are then, on layer 4, replaced by actual ones. Their region extents of a layout are derived from the field they are attached to and how that field is used in defined kernels. The strategy implementing this behavior works as follows: First, we scan for loops over fields which mark kernels on layer 4. Next, we look for field accesses inside the body of such loops. We distinguish between left‐ and right‐hand sides of assignments which correspond to write and read accesses, respectively. Taking the boundaries of the enclosing loop into account, we are able to extract a minimum and maximum extent of such accesses. We additionally enhance this
mechanism to recognize cases where special care is required. For instance, multiplications of operators, i.e. stencils, and fields may modify the access pattern. In this case, we handle each stencil entry as an offset applied to the original field access. This information is collected for each kernel and stored in maps linking each field to its respective minimum and maximum read and write extents. Having the data on a per-kernel basis is required to automatically add communication statements, as described in section 5.1.3. Using the aggregated values, it is possible to derive the required number of ghost layers for each field. At this point, boundary conditions are also taken into account since the implementation of some of them requires a specific number of ghost layers to be present. Dirichlet or Neumann boundary conditions applied to cell-centered fields, for instance, require one ghost layer. In order to avoid polluting the generated layer 4 representation, we aggregate field layouts with common characteristics.

4.4 The Intermediate Representation (IR)

In our framework, the intermediate representation (IR) is the link between the (processed) layer 4 AST and the generated target source code. As such, it can be understood as another layer in our generation pipeline. Implementing it, however, is more complex compared to the other layers. This stems mainly from two challenges. On the one hand, the gap between the input, a processed layer 4 specification, and the ultimate output, source code in a chosen target language, is comparably wide. On the other hand, a high degree of configurability is required. In many cases, additional functionality has to be added or injected, e.g. when users request parallelization, as discussed in chapter 5. In other cases, parts of the AST have to be rewritten, e.g. when applying performance optimizations. We tackle these challenges by breaking the transformation pipeline into fine-grained parts that refine the AST step by step. Naturally, this also requires including new node types representing intermediate steps. For reference, our complete code generation framework contains about 170 transformations in 130 strategies and 380 node types in the IR alone.

One drawback of having a high number of specialized transformations is the increase in generation time, as well as the heightened code complexity. We mitigate this issue by identifying types of nodes that simply need to be evolved into nodes of other types. One example for
this are loops over fields coming from layer 4 that, ultimately, need to be converted into sets of nested for loops. We call this mechanism expansion since the complexity and/or number of new nodes is usually higher than before. In most cases, it is not particularly relevant when exactly this happens in the generation pipeline. Introducing a separate transformation for each of these node types consequently not only bloats the code base, but also increases generation times since the whole AST needs to be traversed each time a transformation is applied. As an alternative, we provide a specialized trait called IR_Expandable in our framework. Implementing it requires providing an expand method which needs to return either a valid IR_Node, a collection of them or None in case the node is to be eliminated. A straight-forward strategy can then be applied to expand all applicable nodes in the current AST in one go. By default, this strategy will also recursively apply the expansion: all results of an expand call are scanned for new instances of IR_Expandable and, if found, they are automatically expanded as well.

4.4.1 Example

In this section, we illustrate the necessary steps to be taken in refining a given layer 4 input until, ultimately, C++ code can be emitted. We do this using a simple layer 4 input solving a 2D finite differences (FD) discretization of Laplace’s equation, i.e. assuming a zero right-hand side, on the unit square by applying 1 000 iterations of a damped Jacobi. The full specification is given in listing 4.7. Please note, that the state of the code generator is constantly evolving and, thus, this description may be incomplete. Moreover, some steps may need to be executed in a specific order not discussed here.

```plaintext
Domain global< [ 0.0, 0.0 ] to [ 1.0, 1.0 ] >
Layout NodeLayout< Real, Node > { }
Field Solution< global, NodeLayout, 
  vf_boundaryPosition_x**2 
- vf_boundaryPosition_y**2 >[2]
Stencil Laplace {
  [ 0, 0] => 4
  [-1, 0] => -1
  [ 1, 0] => -1
```
\[ [0, -1] \Rightarrow -1 \]
\[ [0, 1] \Rightarrow -1 \]

Globals {
    Var omega : Real = 0.8
}

Function Solve@finest {
    apply bc to Solution[active]
    apply bc to Solution[next]

    repeat 1000 times {
        startTimer ("solver_iteration")

        loop over Solution {
            solve locally with jacobi relax omega {
                Solution => Laplace * Solution == 0
            }

            advance Solution
        }

        stopTimer ("solver_iteration")
    }

    print (getMeanFromTimer("solver_iteration"))
}

Listing 4.7: ExaSlang 4 example of an application solving Laplace’s equation using damped Jacobi.

At this point, we assume that the steps discussed in section 4.2 have already been done. That is, knowledge objects have already been constructed from the corresponding declaration statements. Likewise, accesses to such objects have been resolved. Last but not least, the processed input has been progressed to the IR as described before. Although not relevant in this example, objects and functions declared for multiple levels are usually duplicated and specialized for each of the levels before progressing. Our first step is resolving function calls to what we call special functions. These are functions that are neither specified by users in the DSL, nor will they exist in the final generated code. Instead, calls to them are replaced by specialized nodes implementing the required
behavior. The function call to \texttt{print}, for instance, will be replaced by a dedicated \texttt{IR\_Print} node that, in the end, can be pretty-printed building on \texttt{std::cout}. Another example are the timer function calls \texttt{startTimer}, \texttt{stopTimer} and \texttt{getMeanFromTimer}, which are mapped to statements implementing timing routines specific to the target platform as explained in detail in section 7.1. This mechanism also allows for the quick addition of new and experimental functionality to our generator. Please note that the call to \texttt{sin} is not regarded here since math functions have already been resolved on layer 4.

Next, apply \texttt{bc} statements are resolved. Since the concrete implementation of the required behavior can become quite lengthy, we encapsulate it in a separate function. The original statement can then be replaced by a call to the newly set up function. This mechanism also allows reusing the same implementation if boundary conditions are to be applied to the same field in multiple stages of the application.

At this point field loops can be handled – doing it before would lead to issues since boundary handling functions frequently introduce new loops. In the IR, the first loop type is \texttt{IR\_LoopOverPoints} which can directly be progressed from its layer 4 counterpart. Its loop base is a field and, by extracting the dimensions of the associated field layout, a multi-dimensional loop nest can be set up. We call this \texttt{IR\_LoopOverDimensions}. The next step is splitting it up into multiple nested instances of \texttt{IR\_ForLoop}, which can then be pretty-printed to C++.

Concurrent to the loop resolution, we take care of field and stencil accesses, as well as convolutions of them. In this simple example, it is not particularly relevant when the different access and loop types are introduced. For some extensions, however, this may change, e.g. when loop transformations are to be applied. As discussed, we assume that accesses to fields and stencils have already been identified on layer 4 and that they have been progressed to instances of \texttt{IR\_FieldAccess} and \texttt{IR\_StencilAccess}, respectively. This allows looking for multiplications between fields and stencils, where multiple scenarios are handled: Multiplications of two stencils are resolved by replacing the two involved stencil accesses with an access to a newly generated stencil. Multiplications between a stencil and a field are expanded to a sum of multiplications of the the stencil coefficient expressions and field accesses with accordingly mapped indices. The same happens for multiplications of stencil fields and fields, although in this case the stencil coefficients are replaced by accesses to the the underlying coefficient
4.4 The Intermediate Representation (IR)

![Diagram showing different node types representing accesses to fields.](image)

**Figure 4.1:** Logical indexing for different node types representing accesses to fields.

Field. Of course it can now happen, that the result of a stencil-stencil convolution is to be multiplied with another stencil or field. Thus, we apply the transformations in charge of the resolution multiple times in an iterative fashion until no more changes occur.

As now no more stencil accesses remain, we can continue handling the field accesses. By definition, an IR_FieldAccess is the direct progression of an L4_FieldAccess. As such, it assumes that the point with zero index is the first valid computation point, i.e. it is agnostic of ghost and padding layers. This is changed when converting to an IR_DirectFieldAccess as illustrated in figure 4.1. Linearization is now possible, yielding an IR_LinearizedFieldAccess which can, finally, be transformed into a pretty-printable IR_ArrayAccess.
As last part of handling fields, we take care of allocating and initializing the C++ arrays holding our field data. We do this by setting up two specialized functions which are represented by instances of \texttt{IR\_AllocateDataFunction} and \texttt{IR\_InitFieldsWithZero}. The first is filled with allocate statements wrapped in loops over all slots, as applicable. The second is populated with one \texttt{IR\_LoopOverDimensions} instance for each field and level setting a single element to 0. Loops are, again, wrapped with slot loops as required and, finally, processed as previously discussed.

Of course, processing field accesses and subsequent node types must be pipelined with other strategies emitting new instances of them. One example for this is the handling of solve locally statements, which is done as follows: First, the unknowns to be solved for are gathered from the provided information and marked in the also provided equations. This allows unfolding nested expressions containing multiple unknowns. Next, each equation is reordered according to terms depending on the unknowns, which are moved to the left-hand side, and other expressions, which are moved to the right-hand side. The newly formed right-hand sides can now be simplified to reduce AST complexity. After these preparatory steps, a linear system can be composed from the unknowns, which are assembled in the unknown vector, and the reformulated equations which provide the system matrix and the right-hand side vector. In the base case, that is in the inner part of the domain, this system can be solved directly. For small systems, symbolic inversion of the system matrix is feasible and, in this case, exploiting zero entries in the matrix is possible. However, for larger matrices, say larger than four by four, this becomes prohibitively expensive since the resulting symbolic expression for the inverted matrix grows too large. For these cases, we switch to performing the matrix inversion at runtime by generating corresponding code. Regardless of the chosen variant, new values for the unknowns can be obtained by multiplying the inverted matrix and the right-hand side vector. At this stage, we also incorporate user-provided modifiers such as specifying a Jacobi-type update or applying damping.

This base approach is now extended to also work at or near the boundaries of a field. In this case, one could reduce the sizes of the matrix and the unknown and right-hand side vectors, but this would then require generating different routines for inverting matrices of different sizes. Instead, we adapt the system of equations for unknowns that are
not part of the inner region. First, the entries in the row of the system matrix associated with the unknown to be ignored are set to zero everywhere but on the diagonal entry which is set to one. Next, the right-hand side expression is set to the field access to the unknown to be ignored. Lastly, we also wrap the assignments writing back the final values in corresponding conditions. As an optimization, we also check if the special case described can even occur, that is if accessed unknowns can be located outside the inner region. One case where this is relevant is given by a simple point-wise smoother applied to a cell-centered quantity. Then, we can remove the branching required for handling the near-boundary cases and, thereby, compact the generated kernel and improve performance.

Another part that needs to be resolved is the **Globals** block. It is directly progressed from layer 4 and contains a list of **IR_VariableDeclaration** instances. Mapping it to C++ is done in multiple parts. First, a header file is set up which holds `extern` declarations of all variables. The actual declarations are then done in a separate source file. At this points, variables are also initialized if an initialization expression was provided on layer 4.

Apart from these user-defined global variables, there are also those that are not directly visible in ExaSlang programs. They represent data that not only has to be available to the generated code, but also needs to be modifiable at run-time. We call them **internal variables**, or IV for short. One example for them would be an integer variable keeping track of the currently active slot for the Solution field. In most cases, however, IVs are more complex since they cannot be represented by a single variable as values may differ for each field, level and/or domain. Thus, we represent each IV through a separate, specialized class inheriting from a common abstract base class **IR_InternalVariable** which, in turn, is an **IR_Expression**. Our example of the currently active slot can be modeled using this mechanism as well by implementing an **IR_IV_ActiveSlot** class parameterized with the field and level it is to be used for.

IVs can be mapped to target code in different ways. Considering, e.g., an IV which is only level-dependent, one option would be constructing separate scalar variables with names including the target level, e.g. `foo_0, foo_1, foo_2` and so on. One the other hand, these single variables can also be agglomerated in an array, e.g. `foo[3]` for three levels. Our code generation framework supports both options and allows
tuning it via knowledge flags for the different distinction features such as the domain, the field or the level. For fields, this requires using the field’s index instead of its name such that an array access is possible.

All IVs used in the AST are processed in a multi-stage strategy. First, declarations for each IV are set up, filtered for duplicates and added to the priorly discussed globals section. Next, initialization code is generated according to the default values defined in the corresponding IV classes. Lastly, accesses to IVs are replaced by (possibly nested) array accesses or flat variable accesses factoring in the parameters of the original class instance.

Adding support for IVs such as the active slot also allows implementing other slot operations. This includes the advance statement which can be replaced by a simple increment of an IV. A suitable modulo operation with the field’s number of slots as divisor can be added as required.

Last but no least, a main function needs to be set up for stand-alone applications. This can be done in two ways: either users specify an un-leveled Application function on layer 4 or, if no function with this name is given, a default one is generated. In the latter case, we look for a Solve function defined on the finest level to be called in the main function. Alternatively, a gen_solve function, as automatically set up when using the generate solver mechanism on layer 3, is also valid. If neither of these functions is present, an error is raised.

This pipeline of different node types being expanded into different node types in multiple stages might seem overly complicated. In fact, the first prototype [4] didn’t use this mechanism and, instead, was based on a single-pass expansion. This, however, made it very difficult to extend, especially with extensions that should remain optional. Moreover, a modular implementation is near-impossible. In contrast, our approach allows compartmentalizing different extensions such as the performance optimizations discussed in section 5.4. It can also be extended easily, as demonstrated by introducing a new loop type, IR_LoopOverPointsInOneFragment, in section 5.1.1.

4.5 Contribution

We presented the workflow of our code generation framework while particularly focusing on the mapping between layers. Continuous formulations from layer 1 can be automatically discretized using FD, and
matching solvers can be set up automatically on layer 3. An example illustrating the processing from layer 4 onwards completes the chapter. My main contributions are:

- Conceptualization, implementation and integration of required transformations and data structures on all layers. Work regarding layer 4 and the IR has partly been done in collaboration with Christian Schmitt.

- Realization of the hint system introduced in chapter 3.

- Implementation and integration of strategies applying automatic discretization using FD based on the thesis of Ewald Flad [26].

- Conceptualization, implementation and integration of the guided multigrid solver generation on layer 3, including four Krylov subspace methods available as CGS.

All work has been supervised by Harald Köstler.
5 Code Generator Part II : Extensions

This chapter discusses how the basic workflow presented in the last two chapters can be extended. We show how parallelization capabilities can be added to our code generator such that generated solvers can be executed on cluster architectures. Moreover, we widen the support for different grid types as required for more advanced application scenarios.

5.1 Parallelization of Generated Applications

This section explains how a domain-specific language (DSL) and code generator, such as the one presented to this point, can be extended to generate massively parallel applications. In the ExaStencils project such an extension is vital since one of the most important goals was to enable generation of solvers that are highly scalable, ideally up to future exascale architectures. In [11], we discussed suitable parallelization approaches, how they can be categorized and how we implemented them in our code generator. In essence, two parts need to be addressed: First, data, which in our context is represented by fields, needs to be partitioned and distributed across the available compute resources. Second, parts of the distributed data need to be synchronized, usually at the interfaces of partitions. To effectively realize the latter, back ends are required that map to a set of given parallelization APIs.

Here, we currently support MPI, OpenMP and CUDA. All of them are conceptualized as thin interfaces with multiple levels of abstraction. This means, for instance, that a generic wait node can be mapped to an MPI-specific wait node that, ultimately, can be pretty-printed as call to `MPI_Wait`. This multi-stage abstraction will also allow extending to different back ends in the future.
5.1.1 Data Partitioning

When parallelizing an application to utilize multiple compute resources, each of them has to do a portion of the work. Modern-day clusters are virtually always based on a distributed memory architecture and, thus, data needs to be distributed as well. When partitioning data in the context of numerical solvers, there are usually multiple possible starting points. The most abstract approach would be dividing up the computational domain; this is available from layer 1 onwards. Starting on layer 2, it is also possible to neglect the domain to a certain extend and partition the computational grid in its stead. Finally, directly targeting the arrays holding field data is also viable, although this approach would only be possible post layer 4.

Looking at the solution applied by our code generator is possible from two different angles – what we do conceptually and how it is really implemented. Conceptually, we partition the computational domain into patches that we call fragments. Thereby we also implicitly partition the computational grid which is usually tied to the domain. To be a valid partition, fragments must not overlap and the union of all fragments must yield the full domain. In case of multiple domains, the former must hold for one global domain. For all other domains embedded in the global domain, the condition must hold for sub-set of the available fragments. All fragments must be either fully inside or fully outside any given domain.

We introduce the concept of fragments on layer 4, since it is the most computer science oriented and it would be difficult to introduce these concepts on more abstract layers. In order to do so, we impose a rather drastic restriction: all layer 4 definitions are regarded in the scope of a single fragment. Conceptually, this works analogous to MPI applications, where a single program is executed on multiple compute resources in parallel. In ExaSlang, this affects all functions and, in extension, any kernels which are executed in the iteration space of a given fragment. Likewise, we use field layouts to specify sub-grids specific to the current fragment. One big benefit of this approach is that the layer 4 representation is independent of the number of fragments, with a single fragment being the special case discussed in the previous chapters.

In most applications, access to data of neighboring fragments is necessary. We solve this problem by relying on halos, which is predominant in scientific computing. More precisely, we extend field layouts to include ghost layers. To distinguish between these parts of the grid
5.1 Parallelization of Generated Applications

Figure 5.1: 1D illustration of the global distribution of data points onto two fragments and their mapping to padding (P), ghost (G), duplicate (D) and inner (I) regions [11].

and the ones which are inside the fragment, we impose grid regions. Inner points have already been introduced and mark grid parts which are owned by a fragment. Copies of inner points from other fragments may be represented by ghost regions. Grid points which are shared between multiple fragments, usually because they are located exactly on the fragment interface, are present on all adjacent fragments. We mark these as duplicate points since they play a special role in, amongst others, data synchronization which will be explained next. All of these regions may be defined per dimension and all of them will be inferred if no specification was made on layer 4. The latter is based on the chosen localization, e.g. cell-centered values are usually not duplicated. In addition to these regions, additional padding may be defined. These regions are only used to prepare aligned vector operations and, thus, are usually not user-provided on layer 4 but instead added automatically later. Figure 5.1 illustrates the typical region layout for node-centered quantities across two fragments.

It is important to keep in mind, that field layouts are level specific. Thus, when adapting the size of the inner region it is vital to fulfill certain restrictions resulting from the way inter-grid operators are set up. In most cases, a coarsening factor of two is assumed, that is the number of cells is halved for each coarser level. Figure 5.2 illustrates this for a hierarchy with three levels.

So, in summary, fragments have multiple roles in our context. Conceptually, each of them represents a part of the computational domain. In extension, this means that they also represent the part of the computational grid that is included in that sub-domain. This holds true for every level of the multigrid hierarchy. Finally, they act as containers for the field data associated with the (leveled) grids.
For the perspective of object-oriented programming, it would now make sense to encapsulate all data associated with a fragment in a class or struct in the generated code. In our framework, however, this would introduce another layer of indirection and, thus, make things unnecessarily complex. Instead, we rely on using internal variables (IVs), as introduced in section 4.4. All applicable IVs are extended such that they take a fragment index as parameter and that they generate an additional array dimension for said index. This also allows fully eliminating the array dimension used to access the fragment if only one fragment (per block) is present. Additionally, for all variables that need to be duplicated for each possible neighbor, we add another dimension for the neighbor direction index. In this parameter category are the indices of neighboring fragments, their MPI rank in case of an MPI parallelization, and flags stating if the neighbor is (a) valid, (b) in the same block or not. The flag for the neighbor existence needs additionally to be distinguished by domain index. Other IVs are explained in the next section.

Today’s cluster architectures and, in extension, parallelization approaches to be employed are usually hierarchical. Thus, our approach should be as well which we realize be introducing blocks as coarser unit of partition. Each block holds multiple fragments, and each fragment must be associated with exactly one block. Moreover, each block must currently hold the same number of fragments.

In some cases, it may be advantageous to expose different levels of our partitioning hierarchy. One example for this is given by fragment loops which, as the name suggests, allow looping over all fragments in the current block. In other words, they move the scope one level up the primitive hierarchy, from fragment-centric to block-centric. In the DSL, it suffices to write loop over fragments followed by a block of statements enclosed in curly braces, as demonstrated in listing 5.1. However,
users are not forced to set up these loops by hand if it is not required by his application: If the code generator encounters a field loop which is not inside the body of a fragment loop, it wraps it in a default fragment loop. This outer loop also inherits possible reduction clauses. In the intermediate representation (IR), we represent the new functionality by adding `IR_LoopOverFragments` and `IR_LoopOverPointsInOneFragment` classes. The old `IR_LoopOverPoints` class can then be transformed into wrapped instances of the newly introduced node types.

```plaintext
Function Smoother@all {
  loop over fragments {
    loop over Solution {
      Solution[next] += (
        omega * diag_inv ( Laplace )
        * ( RHS - Laplace * Solution )
      )
    } // ...
  }
}
```

**Listing 5.1:** ExaSlang 4 example of using fragment loops in the definition of a (damped) Jacobi smoother for the discretized Poisson's equation.

If parallelization is enabled and the domain is to be partitioned, the required information, e.g. the connectivity between the fragments, must be set up in the generated code. In case of rectangular domains, this can be done in an automated fashion based on a set of knowledge flags. These are

- `domain_rect_numBlocks_x` for the number of blocks in x-direction,
- `domain_rect_numFrgsPerBlock_x` for the number of fragments in each block in x-direction,
- `domain_rect_periodic_x` for signaling that the domain is periodic in x-direction, i.e. that the last fragment should be connected to the first in a ring-like fashion, and
- `domain_fragmentLength_x` for elongating each fragment with a constant factor in x-direction.

For other dimensions analogous variants exist. Using these flags, only rectangular blocks are possible. The length of a fragment is used when inferring the default size of the inner region of a layer 4 layout. It is calculated by $2^l \times \text{fragmentLength}$ for the number of inner cells in a given dimension, where $l$ is the multigrid level it is evaluated for.
Figure 5.3: Example of a plus-shaped domain comprised of 4 blocks with 5 fragments each.

The total number of blocks and fragments is given in domain_numBlocks and domain_numFragmentsPerBlock, respectively. Since these numbers can be determined automatically in case of a rectangular domain and partition, it is not necessary to provide them in the knowledge file.

For other types of domains, such as the one depicted in figure 5.3a, the approach of setting up geometric and topological information at the run-time of the generated code is not feasible. Instead, it is required to switch to reading in the required data from one or more files. Jeremias Isnardy examined different possibilities to realize this approach in his master thesis [29]. In order to do so, he set up a suitable data format and devised concepts for parallel file IO. Moreover, the geometric properties of fragments that were set up from a given axis-aligned bounding box (AABB) can be modified. This is realized through specifying a suitable transformation matrix for each fragment. Figure 5.3 shows one simple test case – a plus shaped domain – which was implemented within the scope of his thesis. As evident, the new approach also allows having irregular blocks. Other, more complicated domains such as the one depicted in figure 5.4 have been examined as well but, unfortunately, didn’t show convincing results.
5.1 Parallelization of Generated Applications

5.1.2 Data Synchronization

After concepts for partitioning the data have been set up and implemented, the next required step is supporting synchronization of said data. In our context, this translates to mainly updating ghost layers with values from neighboring fragments. Depending on the application at hand, and the implementation specifics, duplicate layers may need to be synchronized more or less frequently as well. For both cases, code needs to be generated. Our main aim is giving users a tool to adapt and optimize communication behavior on layer 4, while still being as compact and concise as possible. We realize this by introducing `communicate` statements as the main control mechanism on layer 4. These statements advise our generator to set up code to synchronize the data of a given field for a given level. By default, all data marked accordingly in the attached field layout is synchronized. Marking is done by appending `communicate` to a region specification in a field layout declaration. This is illustrated in listing 5.2.

```plaintext
Layout DefaultNodeLayout < Real , Node > {
    duplicateLayers = [ 1, 1, 1 ] with communication
    ghostLayers = [ 1, 1, 1 ] with communication
}
```

Listing 5.2: ExaSlang 4 example of a field layout declaration with communicating ghost and duplicate layers.

Figure 5.4: Example of a ring-shaped domain comprising of 6 fragments [29].
In some cases, it is better to not target all applicable regions. For these cases, communicate statements can be extended with region restrictions. In the simplest case, this means just providing the type of region to be updated. An example would be `communicate ghost of solution` which targets only ghost layers but not, e.g., duplicate ones. In case of more than one layer of a given type, it is also possible to target specific layers. Selecting only the first two ghost layers, for instance, could be done using `communicate ghost [0, 1] of solution`.

By default, all communication is synchronous with respect to the control flow of layer 4. That is, layer 4 statements in front of any communicate statements will be fully executed before the actual communication starts. Likewise, all statements behind it will only be executed after the communication phase was completed. This, however, does not mean that the generated code relies on synchronous operations. Inside the communication phase, the code generator is free to generate synchronous and asynchronous operations as necessary. If layer 4 asynchronism is required, communication can be split into begin and finish phases. This is illustrated in listing 5.3.

```
begin communicate solution
/* actual work */
finish communicating solution
```

Listing 5.3: ExaSlang 4 example of asynchronous communication.

Additional restrictions may still be added, but they need to be added to both statements and they must be identical. As shown, we additionally allow using `communicating` in the place of `communicate` to enable representations closer to natural language. One caveat in the current generator version is, that it is not checked whether begin and finish statements match. This is due to the fact, that both statements could be implemented in different functions and, thus, a detailed call graph would be required for verification. Setting up such a call graph is future work.

When implementing the communication triggered by the communicate statements, variants for ghost layer exchange are quite straightforward since source and destination are clearly defined. For duplicate layers, it is more complicated since it is not clear how to unify possibly varying variants of the same data point. To resolve this conflict, we define the fragment with the lowest rank as owner of any duplicate data point. When communication is triggered, it sends the duplicate data
to the other fragments which use it to overwrite their versions. Other approaches such as taking the mean value of the different versions is possible as well, although currently not implemented.

Regardless of which regions are communicated, there are usually two possible communication patterns for regular fragment topologies. The straight-forward variant is exchanging data with all direct neighbors. In 2D and 3D, this would amount to 8 and 26 neighbors, respectively. It is illustrated in figure 5.5 for duplicate and ghost layers.

This approach, however, might incur some overhead due to the high number of messages and, additionally, might promote congestion. Relaying data for neighbors sharing only a corner or edge by using neighbors sharing a face can improve the situation. This approach is visualized in figure 5.6. On the downside, this method requires synchronizing the operations per coordinate dimension. Moreover, it only works straightforward if all fragments are oriented the same way whereas the first approach could be extended more easily to also support rotated or flipped fragments.

Next, we discuss how our generator resolves communicate statements. In some cases, certain communication operations reoccur throughout a Layer 4 program. We regard this as the case if the same layers of the same field on the same level are targeted. This can become an inconvenience since, usually, the generated code resulting from communication operations becomes quite lengthy. As a countermeasure, we decided to encapsulate the code implementing a communicate statement in a separate function. The communicate statement itself can then just be replaced with a call to said function. When performing this replacement, we check if the required function already exists and, if not, one is generated. The body of the function is then where the actual logic is implemented. Depending on the chosen communication pattern, i.e. full communication with all neighbors or relaying data via neighbors on the same coordinate axes, different versions are created.

We first discuss the full communication variant. One possible way to set up the communication function is shown in figure 5.7. First, some intermediary nodes are introduced. They are usually split into two parts, start and finish, and provide abstractions for sending and receiving data to and from other fragments. Additionally, we differentiate local and remote communication, where local is defined to take place within one block and remote between fragments associated with different blocks. Local communication can usually be
(a) Ghost layer updates from the perspective of the lower left fragment. (b) Duplicate layer updates for four fragments. Information is only propagated upstream, i.e., only values of neighbors in the right and upwards direction are overwritten.

**Figure 5.5:** Two sample communication patterns. Circles, squares and triangles correspond to inner, duplicate and ghost elements, respectively [11].

**Figure 5.6:** Two-stage synchronization of ghost values as seen by the lower left fragment. The horizontal exchange on the left is followed by the vertical exchange on the right thereby also propagating information in empty triangles incrementally [11].
done directly, that is without buffering and without asynchronous operations. That means, that depending on the actual implementation strategy chosen, data is directly read from or written to a neighboring fragment. We reflect this fact by summarizing the send and receive nodes into general communication nodes for local operations. In both cases, i.e. for remote and local operations, every intermediary node is generated once for each neighbor. At this stage, we also set up the index sets to be communicated by using the information stored in the layout attached to the target field and the restrictions from the originating communicate statement. By definition, the introduced nodes are specific for a single fragment. Thus, the next step is wrapping them in a loop over fragments.

After setting up the intermediary nodes, they are expanded (c.f. section 4.4) depending on their type:

- nodes starting a remote send operation are split into a copy to buffer node and a remote send node,
- nodes finishing a remote send operation are transformed into a wait node,
- nodes starting a remote receive are converted into remote receive nodes,
- nodes finishing a remote receive operation are split into a wait node and a copy from buffer node,
- nodes starting a local communication are converted into a pull data node or a push data node, depending on a knowledge flag, and
- nodes finishing local communication are not required to do anything in this example.

Additionally, we wrap nodes generated from remote operations in conditions checking if the neighboring fragment is located in a different block, i.e. remote. Otherwise, nothing is done. Similarly, nodes generated from local operations are only to be executed if the connection to the neighboring fragment is local.

Nodes representing copies from and to buffers are expanded to loops over the specified index set. Inside, they perform an assignment between the target field and the communication buffer. To be able to parallelize these loops, it is vital that the index of loop may be linearized to a buffer index for each iteration independently. The buffers themselves are represented as IVs and the buffer management is handled in a specialized strategy collecting required buffer sizes and setting up
Figure 5.7: Exemplary expansion of communication statements. Specialized data exchange functions are created through successive refinement and shared for identical communication statements. Local operations are performed for fragments within one block, while remote operations relate to fragments assigned to different blocks [11].
5.1 Parallelization of Generated Applications

allocation and deallocation nodes. Other nodes, such as those representing send and receive or wait operations are mapped according to the chosen parallelization back end. Last but not least, push or pull data operations are implemented via basic loops and assignments between the same field on different fragments.

All expanded nodes, or node lists, are additionally each wrapped in a condition checking two things. One, the current fragment has to be valid for the domain of the field. This is necessary if more than one domain is defined and fragments may be present but not valid for a given sub-domain. Second, the target neighboring fragment has to exist. This, for instance, is not the case at the boundary. In general, both conditions are checked at run-time of the generated code. If, however, one of the conditions can be simplified or fully eliminated, e.g. because only one domain is present, our generator does so automatically.

This pipeline is sufficient to realize a communication function based of a full communication pattern. For the reduced variant, it can be implemented similarly. The main difference is, how the generated parts of the communication are ordered. In essence, the same steps are performed, but only for two neighbors, e.g. left and right, at a time. This is then repeated for the second and, if applicable, the third dimension. In a final step, the resulting parts are concatenated to form the body of the communication function. Another, smaller, difference is that the size of the communication layer is extended as visualized in figures 5.5a and 5.6.

This pipeline can be used for most cases. One potential issue, however, is that it introduces fragment loops which leads to wrong code if a communicate statement is placed inside the scope of another fragment loop. We fix this problem by generating specialized communication functions that handle only a single fragment whose index is received as function parameter. Other than the changed function signature, and associated function call, the pipeline only has to be adapted at two places. First, the step wrapping the generated nodes in a loop over fragments has to be omitted. Second, and more complicated, the local communication has to be synchronized between fragments. In the default setting, the iteration points of a fragment loop, i.e. handling single fragments, is distributed across multiple threads. Inside the fragment loop, computation and communication can be mixed arbitrarily. Thus, it may happen that one fragment still performs computations while another already started the communication phase, which, in turn, leads to potential race
conditions. We avoid this issue by introducing flags that signal when a fragment is ready to communicate and when it is done. They are modeled as IVs and are set up for each fragment and for each neighbor. It is vital that the flags’ data type is declared as volatile to prevent deadlocks. We additionally set up one dedicated flag for each communication neighbor of a fragment to avoid unnecessary waiting. Additionally, this can further improve overlapping computation and communication in future implementations. For this, we envision flagging the communication with a given neighbor as available as soon as the region to be exchanged has been updated in the computation.

Based on the introduced flags it is now possible to extend the resolution of the intermediary local communication nodes. For the start of the local communication we adapt it such that it generates code doing the following. First, it sets flags for all neighbors signaling that communication is ready, i.e. that the current fragment has finished computations and entered the communication phase. Likewise it then waits for its neighbors to signal readiness as well. Once the corresponding flag is set, the communication from the original implementation can take place, independent of it being a pull or push variant. After finishing the data exchange the flag indicating completion is set as well. In the code expanded from finish local communication nodes, we simply wait that all neighboring fragments signal that they completed communicating with the current fragment. That is, in the pull variant no more data has to be read from the current fragment. In the push variant, this translates to all data of the current fragment being updated. Using this extension, asynchronous communication is also supported without the need for further adaptations.

5.1.3 Automatic Insertion of Communication Statements

In the envisioned workflow of ExaSlang, users are not required to touch layer 4 as it can be set up automatically based on the specifications given on the upper layers. This also extends to communication statements which is why our code generator needs to be equipped with capabilities to add them automatically.

To implement this extension, the following needs to be done for each kernel on layer 4: First, the fields read and written by the current kernel need to be elicited. This information has to be enriched with the corresponding extents, i.e. the minimum and maximum index positions per
5.1 Parallelization of Generated Applications

dimension read and written for any given field. These indices can then be mapped on the regions of the field to determine which duplicate and ghost layers are affected. Based on the gathered data, communication statements can be added based on two approaches. Either data to be read is communicated before the kernel or data written is communicated after the kernel. For the latter, it is generally necessary to know how many ghost layers will be added to a field since the written (inner) regions need to be mapped accordingly. Thus, only the first variant is implemented in the current version of the generator.

The main drawback of this approach is that data may be communicated unnecessarily. Listing 5.4 illustrates this issue. In this—admittedly slightly constructed—example the Solution field would be communicated before both kernels. The second communication is not required since Solution is not written in the first kernel. To catch these cases, however, would require a global view in the form of a, e.g., call graph which is currently not available.

```
1 loop over Residual {
   2   Residual = RHS - Laplace * Solution
}

5 loop over Solution {
   6   Solution += (omega * diag_inv (Laplace )
   7          * ( RHS - Laplace * Solution ) )
}
```

Listing 5.4: ExaSlang 4 example of two loops resulting in unnecessary communication.

The required steps to automatically add communication statements is quite similar to the automatic layout deduction explained in section 4.3.3. Thus, we decided to implement it as an extension of said strategy. Whenever a kernel is scanned for field accesses to deduce the field layouts of the involved fields, we also add communication based on the same information. One other extension, however, is still necessary. For the field layout, slotting is irrelevant since all slots of a given field are required to share the same layout. For the communication, however, slots need to be incorporated since they need to be added to the communication statements. In most cases this is trivial but one case where it is not obvious are local solve nodes with enabled Jacobi-type updates.
Then, write accesses to the unknowns defined in the node need to be modified such that they incorporate the correct slot.

### 5.1.4 Results

In previous publications, we examined the scalability of prototypical reference implementations [8] as well as of our generated solvers [11, 17]. We want to highlight results obtained on the JUQUEEN supercomputer (c.f. section 9.3) for our model problem given by Poisson’s equation. Figure 5.8a shows the scalability for an MPI-only variant using up to 1,835,008 threads running on 458,752 cores across 28,672 nodes. While this test was performed for roughly $10^6$ unknowns per core, we later also solved a similar problem with 16 times more unknowns for a total of up to $7.34 \cdot 10^{12}$ unknowns on the full machine. We additionally added a hybrid parallelization with 4 OpenMP threads for each MPI thread to finally obtain the results shown in figure 5.8b. In both cases, our generated solver exhibit satisfactory scalability. It is, moreover, expected to increase even further for more complicated applications where the
5.2 New Grid Types

5.2.1 Staggered Grids

When providing a localization for fields from layer 2 onwards, up to now only node-centered or cell-centered were available. For advanced applications, such as the ones from computational fluid dynamics (CFD) detailed in chapter 6, an extension is required. It is given by face-centered quantities which are ultimately used to implement staggered grids. An illustration of such a grid is given in figure 5.9.

To incorporate face-centered values, we first introduce a face dimension. In our context, it is equivalent to the stagger dimension, that is the dimension in which cell-centered values need to be shifted to coincide with cell interfaces. Using this definition, it is easy to derive the behavior and properties of staggered variables based on already implemented variants. In all dimension but the stagger one, they can be interpreted as cell-centered quantities. Conversely, they behave just like node-centered quantities in the stagger dimension. This also includes having one duplicate layer by default in this dimension.
In the DSL and code generator, the following steps need to be done. First, new localizations need to be introduced to ExaSlang in the form of keywords. We call them `{face_x}`, `{face_y}` and `{face_z}`. This also requires to adapt the parsers and lexers for all affected layers. Next, each part of the generator taking the localization into account needs to be modified to also handle the new options. We usually realize this by handling each dimension separately and then mapping to the already implemented variants for node and cell localizations. One example for places where an adaption is necessary is given by the automatic field layout deduction, as introduced in section 4.3.3. Another one can be found in generating code for different boundary conditions.

In two dimensions, our definition of face dimensions may be counter-intuitive since one could assume it to be the dimension in which an edge extends. This would, effectively, switch `{face_x}` and `{face_y}`. On the DSL level, this change could be incorporated without a problem since only the parser had to be adapted. Inside the generator, no further adaptations would be required since the parser could already take care of mapping the keywords correctly. We decided for the presented approach, however, since it has the advantage of also working for other numbers of dimensions.

5.2.2 Non-Uniform Grids

Up to now grids have been assumed to be uniform and potentially staggered. Some applications, however, can benefit by a large amount from incorporating non-uniform grids. On the one hand, more complex domains may become possible. On the other hand, this allows resolving regions of interest in the computational domain more finely. Likewise, regions with little variance can be approximated with a coarser grid. While the former has the potential to improve numerical properties such as stability, the latter may increase attainable performance. Please note, that non-uniform grids are still regular, at least for each fragment. That means, that any supported grid must be constructible from a uniform grid by moving the node positions.

Some discretizations, such as finite differences, will vary strongly based on the underlying grid being axis-parallel or not. We, thus, categorize grids according to three properties: uniform, axis-parallel and staggered. Naturally, uniform infers axis-parallel.

To incorporate the new types of grids with our generator multiple steps are necessary. First, geometric information about the grid, such
as nodal positions or cell widths, need to be exposed on the DSL level. For non-uniform grids, this information must additionally be stored in the generated program. In some cases, automatic initialization of the data is possible, in which case one or more functions implementing this must be generated.

We first consider how the geometric information can be exposed in ExaSlang. Ideally, users could write a single DSL representation which can then be used to generate code for uniform and non-uniform grids alike. In most cases, this corresponds to implementing the general case of non-uniform grids which is then specialized inside the generator to, e.g., optimize performance. This approach may be hard to realize in reality, however, since the discretization may vary substantially for different grid types. We still believe that it is worthwhile to try following this approach, even if it only works for a subset of our target applications.

Since geometric information may vary for each location in the grid, we find it most natural to base our extension on the concept of fields. Introducing actual fields, however, would be unnecessarily complicated for users and make later optimizations considerably harder. We, thus, introduce what we call virtual fields. Being firstly introduced in [9], we have extended them since. Currently, the following virtual fields are available:

- `vf_nodePosition` and `vf_cellCenter` to access node positions and cell center positions as vector.
- `vf_cellWidth` to access the widths of the current cell per dimension as vector.
- `vf_nodePosition`, `vf_cellCenter` and `vf_cellWidth` postfixed with an underscore and a number to access their single components, e.g. `vf_nodePosition_1` to access the y-coordinate of the node position.
- `vf_cellVolume` to access the cell volume in 3D and area in 2D.
- Staggered variants of cell widths and volumes, e.g. `vf_stag_1_cellWidth_0` for the cell width of y-staggered cells in x-direction.
- `vf_boundaryPosition` to access the position on the next interface inside boundary handling specifications. Postfixed variants to access single components are available as before.

For backwards compatibility and user friendliness, we support various aliases for these identifiers. One example is given by using `nodePosition_x` which may be used instead of `vf_nodePosition_0`. As
evident, we support omitting the \texttt{vf} \_ prefix and replacing number postfixes with strings. A part from these general simplifications, we also support short forms is some cases such as, e.g., \texttt{nodePos\_x}.

As with real fields, offset accesses are possible. This allows, for instance, calculating a cell width on the fly directly in ExaSlang, as illustrated in listing 5.5.

\begin{verbatim}
1 Expr width = vf_nodePosition_0@\[1, 0\] - vf_nodePosition_0@\[0, 0\]
\end{verbatim}

\textbf{Listing 5.5:} ExaSlang 4 example of calculating the cell width from node positions.

For this extension, neither the parser nor the lexer need to be adapted since all accesses are parsed as unresolved accesses anyway. In the generator, we model virtual fields similar to 'real' fields. That is, there is a knowledge collection and each virtual field is modeled as a separate knowledge object. One key difference, however, is that declarations are not required. By default all fields matching the grid characteristics, such as whether it is staggered, are created. It is important to do this step only on one layer, usually the first provided, to avoid introducing duplicate fields when progressing the knowledge collection.

Resolution of accesses to virtual fields is also done analogous to 'real' field accesses, as described in section 4.2. One difference, however, is that aliases need to be taken into account as well. We implement this as a separate, preparatory strategy which assembles an alias map for all virtual fields and performs a rename operation on all applicable unresolved accesses. After identifying all virtual field accesses, it is possible to already resolve some of them. An example for this is given by the \texttt{cellVolume} which can be replaced by a multiplication of the \texttt{cellWidth} in each dimension. It is vital to inherit level specifications and modifiers, such as offsets, at this point. For certain types of grids, additional simplifications are possible. In the case of uniform grids, for instance, the \texttt{cellWidth} is independent of the position in the grid and can thus be replaced by a constant value for the given level and dimension. This resolution is possible on all layers and can be controlled through the knowledge flags \texttt{lX\_resolveVirtualFields}.

If all virtual field accesses can be replaced by constants or other simple expressions, no additional data needs to be stored. This is, for instance, usually the case when uniform grids are employed. In all other cases, we inject fields on layer 4 to hold the required data. As usual, this
also requires adding matching field layouts. Each virtual field class implements a method setting up its own field layout. It is usually equipped with ghost layers and prepared for communication, such that geometric information is also available beyond the inner region.

In the case of axis-parallel grids, it is not necessary to store information for every grid point. Instead, only one line for each dimension suffices since the information can be shared across, e.g., a whole column. To support this on layer 4, we introduce the new localization types `edge_cell` and `edge_node`. As the names suggest, they behave as one-dimensional field layouts. They are, however, declared as multidimensional fields with one inner layer, zero ghost layers and duplicate layers in all dimensions but one. This allows distinguishing different types of edges and facilitates projecting the indices of accesses correctly. The latter is done in the IR in a strategy which transforms all remaining virtual field accesses to regular field accesses.

## 5.3 Finite Volume Discretizations

Finite volume discretizations, as introduced in section 2.1.2, are one of the focal points of our technology. They are particularly relevant to CFD applications, especially when coupled with staggered grids. We reflect their importance in equipping our DSL and code generator with specialized constructs to allow expressing parts of finite volumes (FV) discretizations more naturally. [9] was the first publication to introduce the extensions discussed next and to apply them to a complex application.

In the case of staggered grids, we first introduce staggered control volumes as illustrated in figure 5.10 for the 2D case. The different types of control volumes can then be given as

\[
\Omega_{i,j} = \left[ x_i^{\text{node}}, x_{i+1}^{\text{node}} \right] \times \left[ y_j^{\text{node}}, y_{j+1}^{\text{node}} \right],
\]

\[
\Omega_{i+1/2,j} = \left[ x_i^{\text{cell}}, x_{i+1}^{\text{cell}} \right] \times \left[ y_j^{\text{node}}, y_{j+1}^{\text{node}} \right],
\]

\[
\Omega_{i,j+1/2} = \left[ x_i^{\text{node}}, x_{i+1}^{\text{node}} \right] \times \left[ y_j^{\text{cell}}, y_{j+1}^{\text{cell}} \right].
\]
in 2D and

\[
\begin{align*}
\Omega_{i,j,k} &= [x_{i}^{\text{node}}, x_{i+1}^{\text{node}}] \times [y_{j}^{\text{node}}, y_{j+1}^{\text{node}}] \times [z_{k}^{\text{node}}, z_{k+1}^{\text{node}}], \\
\Omega_{i+1/2,j,k} &= [x_{i-1}^{\text{cell}}, x_{i}^{\text{cell}}] \times [y_{j}^{\text{node}}, y_{j+1}^{\text{node}}] \times [z_{k}^{\text{node}}, z_{k+1}^{\text{node}}], \\
\Omega_{i,j+1/2,k} &= [x_{i}^{\text{node}}, x_{i+1}^{\text{node}}] \times [y_{j-1}^{\text{cell}}, y_{j}^{\text{cell}}] \times [z_{k}^{\text{node}}, z_{k+1}^{\text{node}}], \\
\Omega_{i,j,k+1/2} &= [x_{i}^{\text{node}}, x_{i+1}^{\text{node}}] \times [y_{j}^{\text{node}}, y_{j+1}^{\text{node}}] \times [z_{k-1}^{\text{cell}}, z_{k}^{\text{cell}}],
\end{align*}
\]

in 3D, extending the previous definitions from section 2.1.2.

### 5.3.1 Evaluate Functions

First, we support evaluating simple expressions at certain points of our grid. In the current context, these points are usually the interfaces of (staggered) control volumes or grid cells. Integration into ExaSlang is done via specialized functions as illustrated in listing 5.6.

```plaintext
// evaluate at interfaces of regular grid cells
evalAtSouthFace ( u )

// evaluate at interfaces of x-staggered cells
evalAtXStaggeredEastFace ( rho )
```

**Listing 5.6:** Example of evaluate function calls on regular and staggered cell interfaces.
At the resolution stage, our generator checks the localization of the expression to be evaluated and one of three cases can occur:

- The expression is independent of grid localities, e.g. a constant or a variable. In this case, the expression replaces the function call.
- The expression is a field access localized at the evaluation interface, e.g. a x-staggered quantity is to be evaluated at the non-staggered east interface. Then, the field access replaces the function call.
- The expression is a field access not matching the above case. Then, an interpolation is generated.

For interpolations, the default case is simply using linear interpolation of the two neighboring quantities. Users can, however, also override this default behavior by specifying a different interpolation when calling the evaluate function. An example would be `evalAtXStaggeredEastFace ( rho, "harmonicMean" )` where the harmonic mean is used. It is given as

\[
\frac{(a_0 + a_1) \cdot (x_0 \cdot x_1)}{a_1 \cdot x_0 + a_0 \cdot x_1},
\]

where \(a_0\) and \(a_1\) are the quantities to be interpolated and \(a_0\) and \(a_1\) their respective distances to the evaluation point, in our case the cell interface. Other interpolation techniques can be easily added to our generator as required.

Additionally, we support offsetting evaluations. This can either be done by applying an offset to the argument or to the function call itself. Of course, both options can also be used simultaneously, as also demonstrated in listing 5.7

```c
// regular evaluate call
evalAtEastFace ( u )

// the same evaluation with an offset field access
evalAtWestFace ( u@[1, 0] )

// the same evaluation with an offset function call
evalAtWestFace@[1, 0] ( u )
```
// the same evaluation with both offset variants
evalAtWestFace@[-1, 0] ( u@[2, 0] )

Listing 5.7: ExaSlang example of evaluate function calls with offset modifiers.

5.3.2 Integrate Functions

For FV discretizations, it is usually required to evaluate integrals over certain boundaries of a given, possibly staggered, control volume. Since this is an integral part of many applications, we decide to introduce specialized constructs for said operations in ExaSlang. Listing 5.8 illustrates this.

// integrate over interfaces of regular grid cells
integrateOverEastFace ( evalAtEastFace ( u ) )

// integrate over interfaces of x-staggered cells
integrateOverXStaggeredWestFace ( u * rho )

Listing 5.8: ExaSlang example of integrate function calls on regular and staggered cell interfaces.

The name of the function to be called is always build from integrate-Over and the target interface given by EastFace, WestFace, NorthFace, SouthFace, TopFace or BottomFace, optionally prefixed with XStaggered, YStaggered or ZStaggered. As shown, the argument of the function call is no longer limited to constants, variables and field accesses. Instead, complex expressions are allowed, which also requires more complex resolution approach. We first define the integration interval given by any interface of a potentially staggered cell to be $I$. If necessary, this interval can be sub-divided into different $I_s$, where the division is done along interfaces of other cell types. The north interface of a non-staggered cell, for instance, can be split to cover the parts overlapping with neighboring x-staggered cells as shown in figure 5.11. In our application scope, we assume that values of expressions do not change across given $I_s$. This allows expressing the integral of an expression over $I_s$ to be replaced by an evaluation of said expression multiplied with the size of $I_s$.

We implement this approach in our code generation framework as follows. First, we identify all field accesses in the expression to be integrated. They are then wrapped in evaluate functions, if not done in the DSL already. Second, we determine how $I$ has to be partitioned by
taking the localizations of accessed fields into account. Third, the original expression is duplicated and multiplied with its respective size for each $I_s$. For each $I_s$, we also adapt offsets of the involved field accesses and evaluation functions to match the integration sub-interval. At this point, the original function call can be replaced with the assembled integration expression.

Offsets modifying the function call are supported for integrate functions as well.

### 5.4 Automatically Applied Optimizations

Our code generation framework provides a solid basis for implementing automatically applied optimizations tailored to the target hardware platform. The bulk part of this work has been done by Stefan Kronawitter and is discussed in detail in his dissertation as well as in various publications [73, 7, 6, 94]. Currently, the following optimizations are available in our generator:

- function inlining,
- arithmetic normalizations,
- address precalculation,
- common subexpression elimination (CSE) and loop-carried common subexpression elimination (LCCSE),
- explicit vectorization,
- polyhedral loop transformations such as tiling, and
- data layout transformations.
For our CUDA back end, Christoph Woller implemented a number of optimizations in the scope of his thesis [110]. They mostly include different options to use local and shared memory on the accelerator to improve performance of kernels bound by main memory bandwidth.

One optimization from the above list is particularly relevant in the scope of FV discretizations and, thus, this work – LCCSE. As discussed in the previous section, evaluations and integrations on and across interfaces is common when using finite volume discretizations. Upon closer inspection one finds that left- and right-hand side evaluations on the same interface mostly result in the same value or at least share some common expressions. The evaluation of a quantity at the east face of a regular cell, for instance, is often equal to the evaluation on the west face in the next iteration. In an unoptimized version of the code, the expression is evaluated multiple times which introduces unnecessary overhead. This issue is approached by detecting and eliminating such common sub-expressions across multiple loop iterations in any dimension. The detection itself is performed by applying a classical CSE on the original expression and the expression in the next loop iteration, i.e. a copy of the expression with offsets applied to each part of it. For the elimination, data needs to be buffered. The buffer size is dependent on the loop strides and temporary buffers are added automatically. Special care has to be taken when applying an OpenMP parallelization. Here, the buffers connected to the outer-most loop are duplicated for each OpenMP thread and a warm-up iteration is added, also for each thread. Last but not least, the automatic vectorization needs to be adapted as well. Details are, however, out of scope here but can be reviewed in the corresponding publication [7]. In it, we could also demonstrate the applicability and usefulness of this optimization in the context of CFD applications as also presented in the next chapter.

5.5 Contribution

In this chapter, we summarized major extensions for our code generation framework building on the core concepts and implementation discussed in the previous chapters. It is now able to emit massively parallel applications. We discussed the employed concepts for data partitioning and communication as well as their realization. Mapping to suitable back ends as well as strategies to automatically parallelize Exa-Slang programs complete the extension. Additionally, we added lan-
Contribution

language and generator support for new types of grids, namely staggered and non-uniform ones. On them, finite volume discretizations can be expressed in a natural way using specialized language features. Lastly, we gave a brief overview of implemented optimizations and looked into LCCSE in the scope of the aforementioned discretizations.

My main contributions are:

• Extension of the code generation framework to handle partitioned data and synchronize said data.

• Extension of ExaSlang to express parallel computations, data synchronization and relevant data layout aspects. The last two points have been published in [11].

• Extension of the DSL and code generator to handle staggered non-uniform grids.

• Extension of ExaSlang with specialized constructs for expressing finite volume discretizations and extension of the code generator to handle them. The last two points have been published in [9].

All work has been supervised by Harald Köstler.
6 Applications

In this chapter, we demonstrate the applicability of our approach and framework to applications relevant for academia and industry. We discuss the background and continuous formulations of prominent partial differential equations (PDEs) as well as possible discretizations and solvers. Based on this, we demonstrate how an implementation in ExaSlang can be realized.

While a large range of application scenarios is conceivable, we focus on those from the field of computational fluid dynamics (CFD). CFD is aimed at simulating the behavior of fluids and flows. One discerning criterion is the viscosity of the fluid in question. Highly viscous ones can be represented by the so-called Stokes equations which will be presented in section 6.1. The behavior of other fluids may be modeled using the Navier-Stokes equations discussed in section 6.2. In section 6.2.4, we show how the model can be extended to incorporate other effects such as temperature and fluids not being non-Newtonian. Lastly, we simulate the flow of oceans in section 6.3 by implementing a solver for the hyperbolic shallow water equations (SWE).

Complementary to these applications based on discretizing governing PDEs using finite differences and volumes, our technology is feasible to use for alternative approaches. This has, e.g., been demonstrated by Denis Ribica who implemented a fluid solver based on the lattice Boltzmann method (LBM) in ExaSlang [33].

Apart from CFD applications applicability to problems from other domains has been demonstrated as well. One of these domains is given by image processing. Here, we solved for the optical flow [18, 6] using multigrid solvers implemented in our domain-specific language (DSL). Max Gerecke researched how advanced denoising algorithms can be implemented in ExaSlang [27]. More towards the field of 'classical' PDE applications, Kelvin Loh implemented solvers for stochastic partial
differential equations (SPDE) [31]. Last but not least, solvers generated with our framework could be successfully integrated with a state-of-the-art quantum chemistry application. This collaboration with Rochus Schmid, Hannah Rittich and Christian Schmitt is, however, not published yet.

6.1 Stokes Equations

The first application we examine is the simulation of creeping flows usually occurring with highly viscous fluids. One prominent use-case is given by the simulation of earth mantle convection [106] which lays the basis for the simulation of plate tectonics. Target applications include the simulation of topography changes and prediction of natural disasters such as earthquakes and tsunamis. Industrially relevant applications include, e.g., the simulation of paint and certain types of polymers.

6.1.1 Governing Equations

In this work, we assume an incompressible Newtonian fluid. The Stokes equations can then be given by

\[ -\mu \nabla^2 u + \nabla p = f \]
\[ \nabla \cdot u = 0 \] (6.1)

on a given domain \( \Omega \) using the definitions from table 6.1 as well as suitable boundary conditions. As evident, they form a coupled system of linear PDEs.

We slightly alter eq. (6.1) by introducing a source term for the pressure to allow us running a broader scope of analytical test cases. Assuming \( \mu \) being equal to one, we wind up with

\[ -\nabla^2 u + \nabla p = f_u \]
\[ \nabla \cdot u = f_p. \] (6.2)
6.1 Stokes Equations

<table>
<thead>
<tr>
<th>$d$</th>
<th>dimensionality</th>
<th>2 or 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>velocity as vector</td>
<td>$\Omega \mapsto \mathbb{R}^d$</td>
</tr>
<tr>
<td>$u$</td>
<td>velocity component in x-direction</td>
<td>$\Omega \mapsto \mathbb{R}$</td>
</tr>
<tr>
<td>$v$</td>
<td>velocity component in y-direction</td>
<td>$\Omega \mapsto \mathbb{R}$</td>
</tr>
<tr>
<td>$w$</td>
<td>velocity component in z-direction</td>
<td>$\Omega \mapsto \mathbb{R}$</td>
</tr>
<tr>
<td>$f = f_u$</td>
<td>source term for the velocity as vector</td>
<td>$\Omega \mapsto \mathbb{R}^d$</td>
</tr>
<tr>
<td>$f_u$</td>
<td>x-component of velocity source term</td>
<td>$\Omega \mapsto \mathbb{R}$</td>
</tr>
<tr>
<td>$f_v$</td>
<td>y-component of velocity source term</td>
<td>$\Omega \mapsto \mathbb{R}$</td>
</tr>
<tr>
<td>$f_w$</td>
<td>z-component of velocity source term</td>
<td>$\Omega \mapsto \mathbb{R}$</td>
</tr>
<tr>
<td>$p$</td>
<td>pressure</td>
<td>$\Omega \mapsto \mathbb{R}$</td>
</tr>
<tr>
<td>$f_p$</td>
<td>pressure source term</td>
<td>$\Omega \mapsto \mathbb{R}$</td>
</tr>
<tr>
<td>$\mu$</td>
<td>dynamic viscosity</td>
<td>$\mathbb{R}$</td>
</tr>
</tbody>
</table>

Table 6.1: Symbols used for the Stokes application.

Equation (6.2) can also be rewritten in scalar form. For 2D this results in

$$-\nabla^2 u + \frac{\partial}{\partial x} p = f_u$$
$$-\nabla^2 v + \frac{\partial}{\partial y} p = f_v$$
$$\frac{\partial}{\partial x} u + \frac{\partial}{\partial y} v = f_p$$

(6.3)

and for 3D in

$$-\nabla^2 u + \frac{\partial}{\partial x} p = f_u$$
$$-\nabla^2 v + \frac{\partial}{\partial y} p = f_v$$
$$-\nabla^2 w + \frac{\partial}{\partial z} p = f_w$$
$$\frac{\partial}{\partial x} u + \frac{\partial}{\partial y} v + \frac{\partial}{\partial z} w = f_p.$$ 

(6.4)

6.1.2 Discretization

In this section, we discuss the discretization of eqs. (6.3) and (6.4) using finite differences (FD) and finite volumes (FV) approaches.
6 Applications

Staggered Grids

Locating all quantities at the same locations, e.g. at the cell centers, can lead to stability issues. A similar problem occurs for traditional low order finite elements (FE) discretizations where additional stabilization is required [63]. In our scope, however, a better suited countermeasure is using staggered grids instead, as introduced in section 5.2.

Finite Differences

For FD, the first operator to discretize is the (negative) Laplace operator applied to the velocity components ($\nabla^2$). Here, we can reuse the formulations derived in section 2.1.1, e.g.

$$\begin{bmatrix}
0 & 1/\Delta y & 0 \\
1/\Delta x & -2/\Delta x + 2/\Delta y & 1/\Delta y \\
0 & 1/\Delta y & 0
\end{bmatrix}.$$  \hspace{1cm} (6.5)

for the 2D case. The remaining first order derivatives can be implemented as forward or backward differences, e.g.

$$\begin{bmatrix}
0 & -1/\Delta x & 1/\Delta x \\
1/\Delta x & 0 & 1/\Delta x \\
-1/\Delta x & 1/\Delta x & 0
\end{bmatrix},$$  \hspace{1cm} (6.6)

and

and analogously in other dimensions. Due to the staggered grid, they can also act as a central difference if the value of the derivative is not required at the grid locations of the field they are applied to. For instance, using a forward difference in x-direction on $u$, which is staggered in x-direction, is equivalent to evaluating the central difference at the cell center. In the concrete case of the Stokes equations, this results
in using forward differences on the velocity components and backward differences for the pressure. Finally, we find

\[
\begin{bmatrix}
0 & \frac{1}{\Delta y} & 0 \\
\frac{1}{\Delta x} & \frac{2}{\Delta x} + \frac{2}{\Delta y} & \frac{2}{\Delta x} \\
0 & \frac{1}{\Delta y} & 0
\end{bmatrix} u + \begin{bmatrix}
-\frac{1}{\Delta x} & \frac{1}{\Delta x} & 0
\end{bmatrix} p = f_u
\]

\[
\begin{bmatrix}
1 & \frac{1}{\Delta y} & 0 \\
\frac{1}{\Delta x} & \frac{2}{\Delta x} + \frac{2}{\Delta y} & \frac{2}{\Delta y} \\
0 & \frac{1}{\Delta y} & 0
\end{bmatrix} v + \begin{bmatrix}
0 & \frac{1}{\Delta y} & 0
\end{bmatrix} p = f_v
\]

(6.8)

\[
\begin{bmatrix}
0 & -\frac{1}{\Delta x}
\end{bmatrix} u + \begin{bmatrix}
-\frac{1}{\Delta y}
\end{bmatrix} v = f_p
\]

for the 2D case. Formulations for the 1D and 3D cases can be obtained using the same technique but will be omitted here for the sake of brevity.

**Finite Volumes**

In the case of a FV discretization, we can follow the steps discussed in section 2.1.2. Integration of eq. (6.2) and applying the divergence theorem as well as Green’s identities yields

\[
\begin{aligned}
-\int_{\partial \Omega} \nabla u \cdot \hat{n} \, dx \, dy + \int_{\partial \Omega} p \hat{n} \, dx \, dy &= \int_{\Omega} f_u \, dx \, dy \\
\int_{\partial \Omega} u \cdot \hat{n} \, dx \, dy &= \int_{\Omega} f_p \, dx \, dy,
\end{aligned}
\]

(6.9)

where \( \hat{n} \) is once again the surface normal. Integrals over \( \Omega \) can now be transformed into integrals over the control volumes. Due to the staggered grid, however, control volumes may be staggered as well as introduced in section 5.3. With them, we can formulate the final discretization for the 2D case as

\[
\begin{aligned}
-\int_{\partial \Omega_{i+1/2,j}} \frac{\partial}{\partial x} u \cdot \hat{n} \, dx \, dy + \int_{\partial \Omega_{i+1/2,j}} p \hat{n} \, dx \, dy &= \int_{\Omega_{i+1/2,j}} f_u \, dx \, dy \\
-\int_{\partial \Omega_{i,j+1/2}} \frac{\partial}{\partial y} v \cdot \hat{n} \, dx \, dy + \int_{\partial \Omega_{i,j+1/2}} p \hat{n} \, dx \, dy &= \int_{\Omega_{i,j+1/2}} f_v \, dx \, dy \\
\int_{\partial \Omega_{i,j}} u \cdot \hat{n} \, dx \, dy &= \int_{\Omega_{i,j}} f_p \, dx \, dy,
\end{aligned}
\]

(6.10)
where $\Omega_{i,j}$ is the non-staggered control volume defined by cell $(i,j)$ again. Similarly, $\Omega_{i+1/2,j}$ and $\Omega_{i,j+1/2}$ are again the control volumes staggered in x- and y-direction, respectively.

**Mapping to ExaSlang**

The next step is mapping the Stokes equation to ExaSlang. This step can be done either for the continuous formulation or for the already discretized variant. In the case of the former, only finite differences are available for automatic discretization using the current state of the code generation framework. As discussed previously, using forward and backward differences on a staggered grid is equivalent to central differences under the right conditions. We prepare for this by setting up the operators on layer 1 accordingly, as demonstrated in listing 6.1 with $\text{dxBwd}$, $\text{dyBwd}$, $\text{dxFwd}$ and $\text{dyFwd}$. Additionally, fields for the unknowns and right-hand sides are required, as well as boundary conditions for the unknowns. Finally, the Stokes equations can be implemented.

```plaintext
1 Domain $\Omega = (0, 1) \times (0, 1)$

2 Field $u \in \Omega = 0.0$
3 Field $v \in \Omega = 0.0$
4 Field $p \in \Omega = 0.0$

5 Field $u \in \partial \Omega = /* \text{Dirichlet expression} */$
6 Field $v \in \partial \Omega = /* \text{Dirichlet expression} */$
7 Field $p \in \partial \Omega = \text{Neumann}$

8 Field $f_u \in \Omega = /* \text{initialization expression} */$
9 Field $f_v \in \Omega = /* \text{initialization expression} */$
10 Field $f_p \in \Omega = /* \text{initialization expression} */$

11 Operator $\text{Laplace} = -\Delta$
12 Operator $\text{dxBwd} = \partial_\{x\}$
13 Operator $\text{dyBwd} = \partial_\{y\}$
14 Operator $\text{dxFwd} = \partial_\{x\}$
15 Operator $\text{dyFwd} = \partial_\{y\}$
```
Equation uEq Laplace * u + dxBwd * p == f_u
Equation vEq Laplace * v + dyBwd * p == f_v
Equation pEq dxFwd * u + dyFwd * v == f_p

Listing 6.1: ExaSlang 1 example of specifying the Stokes equations prepared for a finite differences discretization.

Based on this continuous formulation, an FD discretization can be applied automatically. We trigger this by specifying suitable discretization hints as demonstrated in listing 6.2. First, the unknown and the right-hand sides are discretized at suitable grid locations of a staggered grid. Operators are discretized next using finite differences of order 2 in the case of the Laplace operator and backward/forward differences for the first order derivatives. At this stage, given parameters such as the discr_type have already been processed. Discretizing the equations is the final step and allows to complete the mapping to layer 2.

Listing 6.2: ExaSlang 1 example of the discretization hints corresponding to the specification of the Stokes equations from listing 6.1.
A layer 2 implementation can either be generated automatically from the previous layer 1 specification or set up by hand. One possible implementation is shown in listing 6.3, where the FD discretization of the negative Laplace operator is used as before (c.f. eq. (2.14) and listing 3.8).

```plaintext
Equation uEquation {
    LaplaceXStag * u
    + ( p@[0, 0] - p [@[-1, 0] ) / vf_gridWidth_x
    == rhs_u
}

Equation vEquation {
    LaplaceYStag * v
    + ( p@[0, 0] - p [@[0, -1] ) / vf_gridWidth_y
    == rhs_v
}

Equation pEquation {
    ( u@[1, 0] - u @[0, 0] ) / vf_gridWidth_x
    + ( v@[0, 1] - v @[0, 0] ) / vf_gridWidth_y
    == rhs_p
}
```

Listing 6.3: ExaSlang 2 example of specifying the Stokes equations discretized with finite differences.

The presented FD discretization only works as intended in the case of uniform grids. Switching to a FV formulation extends this scope by also being applicable to non-uniform but still axis-aligned grids. A suitable specification is given in listing 6.4 which implements eq. (6.10). Other components, such as field and boundary condition declarations can be taken from the FD case, with the exception of the right-hand sides which need to be integrated over the (staggered) control volumes. This is demonstrated in listing 6.5.

```plaintext
Equation uEquation {
    - integrateOverXStaggeredEastFace ( 1.0 )
    * ( u@[1, 0] - u@[0, 0] )
    / ( vf_nodePos_x@[1, 0]
        - vf_nodePos_x@[0, 0] )
    + integrateOverXStaggeredWestFace ( 1.0 )
    * ( u@[0, 0] - u@[1, 0] )
    / ( vf_nodePos_x@[0, 0]
        - vf_nodePos_x@[0, 0] )
```
6.1 Stokes Equations

Equation uEquation {
  - integrateOverXStaggeredNorthFace ( 1.0 )
    * ( u@[ 0, 1] - u@[ 0, 0] )
    / ( vf_cellCenter_y@[ 0, 1] - vf_cellCenter_y@[ 0, 0] )
  + integrateOverXStaggeredSouthFace ( 1.0 )
    * ( u@[ 0, 0] - u@[ 0, -1] )
    / ( vf_cellCenter_y@[ 0, 0] - vf_cellCenter_y@[ 0, -1] )
  + integrateOverXStaggeredEastFace ( 1.0 )
    * p@[ 0, 0]
  - integrateOverXStaggeredWestFace ( 1.0 )
    * p@[ -1, 0]
  == rhs_u
}

Equation vEquation {
  - integrateOverYStaggeredEastFace ( 1.0 )
    * ( v@[ 1, 0] - v@[ 0, 0] )
    / ( vf_cellCenter_x@[ 1, 0] - vf_cellCenter_x@[ 0, 0] )
  + integrateOverYStaggeredWestFace ( 1.0 )
    * ( v@[ 0, 0] - v@[ -1, 0] )
    / ( vf_cellCenter_x@[ 0, 0] - vf_cellCenter_x@[ -1, 0] )
  - integrateOverYStaggeredNorthFace ( 1.0 )
    * ( v@[ 0, 1] - v@[ 0, 0] )
    / ( vf_nodePos_y@[ 0, 1] - vf_nodePos_y@[ 0, 0] )
  + integrateOverYStaggeredSouthFace ( 1.0 )
    * ( v@[ 0, 0] - v@[ 0, -1] )
    / ( vf_nodePos_y@[ 0, 0] - vf_nodePos_y@[ 0, -1] )
  + integrateOverYStaggeredNorthFace ( 1.0 )
    * p@[0, 0]
  - integrateOverYStaggeredSouthFace ( 1.0 )
    * p@[0, -1]
  == rhs_v
}

Equation pEquation {
  integrateOverEastFace ( 1.0 ) * u@[1, 0]
  - integrateOverWestFace ( 1.0 ) * u@[0, 0]
6 Applications

+ integrateOverNorthFace (1.0) * v@[0, 1]
- integrateOverSouthFace (1.0) * v@[0, 0]
== rhs_p

Listing 6.4: ExaSlang 2 example of specifying the Stokes equations discretized using finite volumes.

Field rhs_u with Real on Face_x of global = ( 
    vf_xStagCellVolume 
    * ( /* initialization expression */ ) )
Field rhs_v with Real on Face_y of global = ( 
    vf_yStagCellVolume 
    * ( /* initialization expression */ ) )
Field rhs_p with Real on Cell of global = ( 
    vf_cellVolume 
    * ( /* initialization expression */ ) )

Listing 6.5: ExaSlang 2 example of right-hand side definitions in the scope of FV discretizations.

Independently of the employed discretization scheme, it usually is worthwhile to extract operator definitions from the discretized equations as this promotes readability of generated lower layers. An example for this is shown in listing 6.6. This advises our generator to extract single stencil operators for coupling the different unknowns and, concurrently, to modify the discretized equations such that they employ them.

generate operators @all { 
  equation for u is uEquation store in { 
    u => A11 
    p => B1 
  } 
}

equation for v is vEquation store in { 
  v => A22 
  p => B2 
}
### 6.1 Stokes Equations

Listing 6.6: ExaSlang 2 example of an operator extraction from the equations defined in listing 6.4.

#### 6.1.3 Solver

The next step is implementing a suitable solver for the discretized system of equations, in our case a geometric multigrid solver. As before, this requires three main components given by the smoother, the intergrid operators and the coarse-grid solver. All of them are discussed in this section.

**Smoother**

Simple point-wise smoothers, such as (colored) Gauss-Seidel and Jacobi, are usually not applicable when solving the Stokes equations. One reason for this is that the diagonal of the system, which needs to be inverted for such smoothers, is 0 in the part of the system handling the pressure \( p \). Block or Vanka smoothers solve for multiple unknowns at the same time and, thus, can circumvent this problem. In the case of Stokes, one can, e.g., solve for the pressure of a cell and the adjacent velocity components. This is also illustrated in figure 6.1.

The resulting smoother constructs a local system of equations which is then solved using matrix inversion to obtain update values for the unknowns to be solved for. More details on this resolution are given in section 4.4.1. In the special case of the Stokes equations, the used process can be optimized even further. For this, we refrain from inverting the whole system matrix directly, but instead use a Schur complement representation. In 3D, this is the case if the local system can be given by

\[
\begin{pmatrix}
A_{11} & B_1 \\
A_{22} & B_2 \\
A_{33} & B_3 \\
C_1 & C_2 & C_3 & D
\end{pmatrix}
\begin{pmatrix}
U_1 \\
U_2 \\
U_3 \\
V
\end{pmatrix}
= 
\begin{pmatrix}
F_1 \\
F_2 \\
F_3 \\
G
\end{pmatrix}.
\]  

(6.11)
Figure 6.1: Illustration of a block smoother for the Stokes equations on staggered grids. Bold symbols are read and written, transparent ones only read.

Then, using

\[ S = D - (C_1 A_{11}^{-1} B_1 + C_2 A_{22}^{-1} B_2 + C_3 A_{33}^{-1} B_3) \]  

(6.12)

and

\[ \tilde{G} = G - (C_1 A_{11}^{-1} F_1 + C_2 A_{22}^{-1} F_2 + C_3 A_{33}^{-1} F_3) \]  

(6.13)

we can directly formulate the updates as

\[ V = S^{-1} \tilde{G} \]  

(6.14)

as well as

\[ U_1 = A_{11}^{-1} (F_1 - B_1 V) \]  

(6.15)

\[ U_2 = A_{22}^{-1} (F_2 - B_2 V) \]  

(6.16)

and

\[ U_3 = A_{33}^{-1} (F_3 - B_3 V) \]  

(6.17)

An adaptation to 2D cases is straightforward.

For the smoother itself, different variations are possible. First, the set of unknowns to be solved for in a block can be chosen. Larger blocks potentially improve convergence behavior but also require solving larger local systems, which increases the execution time of a single smoother step. Next, the iteration space can be chosen. Here, blocks can be overlapped and/ or a suitable coloring can be applied. Lastly, updates from
the block smoother can be relaxed, analogously to a damped Jacobi or a successive over-relaxation (SOR) solver.

For the present case of the Stokes equations, we find that a straightforward 5-point block smoother for 2D cases, and a corresponding 7-point block smoother for 3D cases, works reasonably well while still being comparably fast. As iteration space, the cell centers of the grid are chosen, thereby overlapping blocks in the velocity components. Additionally, coloring is applied. For GPU variants, a 9-way coloring for 2D and a 27-way coloring in 3D are required to fully satisfy the dependencies. When computing on the CPU, it can often be beneficial to switch to a simple red-black coloring even though this does not guarantee that no race conditions occur. Last but not least, we apply some damping, usually with a factor of around 0.8.

**Inter-Grid Operators**

Restriction operators for face-centered quantities like the velocity components can be constructed by combining 1D forms of variants for cell- and node-centered quantities. For this, the Kronecker product is used. Listing 6.7 illustrates this for the linear case based on the concepts presented in section 3.3.2. Alternatively, a suitable shortcut is provided for users for default operators. The current example, for instance, could be replaced by Operator RestrictionFaceX from default restriction on Face_x with 'linear'. As usual, prolongation operators can be obtained by transposing the corresponding restriction operators or by using the appropriate shortcut.

```plaintext
Operator LinNodeInterpolation from Stencil {
    [i0] from [2 * i0 - 1] with 1.0 / 4.0
    [i0] from [2 * i0 + 0] with 1.0 / 2.0
    [i0] from [2 * i0 + 1] with 1.0 / 4.0
}

Operator LinCellInterpolation from Stencil {
    [i0] from [2 * i0 + 0] with 1.0 / 2.0
    [i0] from [2 * i0 + 1] with 1.0 / 2.0
}

Operator RestrictionFaceX from
    kron ( LinNodeInterpolation, LinCellInterpolation )
```
6 Applications

Operator RestrictionFaceY from
   kron ( LinCellInterpolation, LinNodeInterpolation )

Listing 6.7: ExaSlang 2 example of constructing restriction operators for face-centered values.

Coarse Grid Solver

As introduced in section 4.3.2, our generator is capable of setting up the following solver variants automatically:
   • conjugate gradient (CG)
   • conjugate residual (CR)
   • minimal residual (MINRES)
   • biconjugate gradient stabilized (BiCGSTAB)

Theoretically, all of them except CG are suitable. As before, simply applying smoother variants is possible as well, although it is usually expected that they perform comparably poor.

In practice, we frequently observe issues due to numerical instabilities which impacts convergence behavior. BiCGSTAB is, by far, the most severe case: reductions yield slightly varying values which can become a problem given that value ranges of intermediaries in the algorithm differ by a large amount. In extreme cases, the solver can diverge non-deterministically and non-reproducibly. As a remedy, we introduce a restart mechanism for all solver variants. It can be activated by settings `solver_cgs_restart` to true. The number of steps after which a restart is to occur can be set via `solver_cgs_restartAfter`.

Another issue we observed is that solvers are usually not able to solve for a residual that is too far below the floating point precision. This happens, for instance, when the initial solution is already a rather good approximation and the target reduction is too high. We address this issue by introducing an absolute threshold value for the residual alongside the target reduction threshold used until now. It can be set via `solver_cgs_absResThreshold`.

Mapping to ExaSlang

As demonstrated in section 3.3.3, solvers can be generated automatically on layer 3. An example configuration for the Stokes system is shown in listing 6.8.


6.1 Stokes Equations

```plaintext
generate solver for u in uEquation
    and v in vEquation
    and p in pEquation with {
        solver_targetResReduction = 1e-12
        solver_absResThreshold = 1e-12
        solver_maxNumIts = 30
        solver_smoother_numPre = 4
        solver_smoother_numPost = 4
        solver_smoother_jacobiType = false
        solver_smoother_damping = 0.8
        solver_smoother_coloring = "red-black"
        solver_cgs = "BiCGStab"
        solver_cgs_maxNumIts = 10000
        solver_cgs_targetResReduction = 1e-12
        solver_cgs_absResThreshold = 1e-12
        solver_cgs_restart = true
        solver_cgs_restartAfter = 256
    }
```

Listing 6.8: ExaSlang 3 example of a generate solver statement for the Stokes equations.

Since no smoother stages are defined, our code generation framework applies some heuristic to determine a suitable smoother implementation. In the present case, it will recognize the staggered grid and deduce that a block smoother is usually a good option. Since cell-centered values are present, it will choose the cells as iteration space. In each cell it will solve for all quantities that are part of the cell including the face-centered ones. Relaxation, coloring and whether a Jacobi-type update is to be used is specified via the corresponding parameters, just as the number of smoothing steps. Listing 6.9 shows an equivalent solver stage specification for the 2D case and listing 6.10 the corresponding generated layer 4 implementation.

```plaintext
smootherStage {
    loopBase p solveFor {
        u@[0, 0] u@[1, 0]
        v@[0, 0] v@[0, 1]
        p
    }
}
```

Listing 6.9: ExaSlang 3 example of a smoother stage for the 2D Stokes problem.
6 Applications

```plaintext
color with {
    ( i0 + i1 ) % 2,
    loop over p {
        solve locally relax 0.8 {
            u[0, 0] => A11[0, 0] * u[0, 0]
            + B1[0, 0] * p[0, 0]
            == rhs_u[0, 0]
            u[1, 0] => A11[1, 0] * u[1, 0]
            + B1[1, 0] * p[1, 0]
            == rhs_u[1, 0]
            v[0, 0] => A22[0, 0] * v[0, 0]
            + B2[0, 0] * p[0, 0]
            == rhs_v[0, 0]
            v[0, 1] => A22[0, 1] * v[0, 1]
            + B2[0, 1] * p[0, 1]
            == rhs_v[0, 1]
            p[0, 0] => C2[0, 0] * v[0, 0]
            + C1[0, 0] * u[0, 0]
            == rhs_p[0, 0]
        }
    }
}
```

**Listing 6.10:** ExaSlang 4 example of a solver implementation in accordance to listing 6.9.

Inter-grid operations are determined automatically based on the localization of the fields they need to be applied to. That is, in the present case operators for cell-centered quantities as well as for each type of face-centered ones are set up.

Lastly, the coarse-grid solver is composed according to user specifications.

Listing 6.8 shows a layer 3 specification which could also easily be converted into a SolverHints block for layer 1 or layer 2. One reason to utilize layer 3 is the possibility to inject further functionality into the solver. If, for instance, an error evaluation is required a corresponding function call can be easily added. This is demonstrated in listing 6.11 for the user-provided function EvaluateAndPrintError.
generate solver for u in uEquation
    and /* others */ with {
        /* parameters */
    } modifiers {
        append to 'cycle' @finest {
            EvaluateAndPrintError@finest ( )
        }
        append to 'solver' @finest {
            EvaluateAndPrintError@finest ( )
        }
    }

Listing 6.11: ExaSlang 3 example of solver modifiers. Here, function calls are injected at the end of each cycle and at the end of the solve process.

6.1.4 Results

Model Problem

We regard two different test problems, one in 2D and one in 3D. For the 2D case, the exact solution is given by

\[
\begin{align*}
    u &= \sin(2\pi x) - \cos(\pi y) \\
    v &= \cos(\pi x) - \sin(2\pi y) \\
    p &= \sin(4\pi x) + \sin(4\pi y),
\end{align*}
\]  

which results in the right-hand sides

\[
\begin{align*}
    f_u &= 4\pi^2 \sin(2\pi x) - \pi^2 \cos(\pi y) + 4\pi \cos(4\pi x) \\
    f_v &= \pi^2 \cos(\pi x) - 4\pi^2 \sin(2\pi y) + 4\pi \cos(4\pi y) \\
    f_p &= 2\pi \cos(2\pi x) - 2\pi \cos(2\pi y).
\end{align*}
\]  

For the 3D case, the exact solution is given by

\[
\begin{align*}
    u &= \sin(2\pi x) - \cos(\pi z) \\
    v &= \cos(\pi x) - \sin(2\pi y) \\
    w &= \sin(2\pi z) - \cos(\pi y) \\
    p &= \sin(4\pi x) + \sin(4\pi y) + \sin(4\pi z),
\end{align*}
\]
which results in the right-hand sides

\[
\begin{align*}
    f_u &= 4\pi^2 \sin(2\pi x) - \pi^2 \cos(\pi z) + 4\pi \cos(4\pi x) \\
    f_v &= \pi^2 \cos(\pi x) - 4\pi^2 \sin(2\pi y) + 4\pi \cos(4\pi y) \\
    f_w &= 4\pi^2 \sin(2\pi z) - \pi^2 \cos(\pi y) + 4\pi \cos(4\pi z) \\
    f_p &= 2\pi \cos(2\pi x) - 2\pi \cos(2\pi y) + 2\pi \cos(2\pi z).
\end{align*}
\] (6.21)

As boundary conditions we choose Neumann for the pressure and Dirichlet with the corresponding exact solutions for the velocity components.

The domain \( \Omega \) is given by the unit square in 2D and unit cube in 3D. When coloring, it is vital that colors match across fragment boundaries. While not being an issue for a simple red-black coloring, more complex approaches need to be handled with care. For avoiding any overlap, as required on GPUs, 9 colors are required in 2D and 27 in 3D. We can represent this in ExaSlang via coloring with multiple modulo expressions, as shown in listing 6.12.

Listing 6.12: ExaSlang 4 example of a 27-way colored 3D kernel.

This coloring, however, would lead to non-matching colors between fragments, as shown in figure 6.2. We avoid this issue by scaling all fragments with a factor of three in each dimension, thereby ensuring matches across boundaries. Implementation-wise, this can easily be realized in the knowledge file or in an inline knowledge block via `domain_fragmentLength_x = 3` and similarly for the other dimensions.

**Coarse-Grid Solver Choice**

Next, we need to choose a well fitting coarse-grid solver (CGS) for our multigrid solver. In order to do so, we first attempt solving our model problem with only the CGS on the finest level, that is no multigrid is employed. Available options are CG, CR, MINRES and BiCGSTAB as well as simply applying the smoother discussed previously. In case of
6.1 Stokes Equations

the latter, we perform ten smoother iterations (of all colors respectively) for each iteration we would perform using one of the other Krylov subspace methods. Restart variants are available as well, bringing the total number of available solvers to ten. Each is configured to stop once at least one of the following conditions is fulfilled:

- the L2-norm of the residual (over all components) is reduced by $10^{12}$ compared to the initial residual,
- the L2-norm of the residual falls below the threshold of $10^{12}$ or
- 100,000 iterations are exceeded.

Problem sizes are kept moderate by using grids ranging from $6^2$ to $768^2$ cells in 2D and from $6^3$ to $192^3$ cells in 3D. Correspondingly, the number of unknowns solved for lies between 96 and around 1.77 million in 2D and 756 and around 28.2 million in 3D. Each solver is parallelized using OpenMP and four threads. As a benchmark machine we use a representative workstation (c.f. section 9.3 for detailed characteristics).

For restart variants, the number of iterations between each restart is fixed to 256 for all configurations. While this value could be optimized to a specific application in practice, we find that it generally yields good enough results for our test problems. Each configuration, i.e. combination of solver and problem size, is executed ten times and the results are averaged.

Running our ten solver variants, our first finding is that CG, CR and MINRES without restart are not stable, even for comparably small grid sizes. They are, consequently, eliminated from further evaluation. For the remaining options, results in terms of number of iterations required are given in figures 6.3 and 6.4. They suggest that simply using the smoother might be a good option. Taking into account, however, that one iteration represents ten smoother applications, BiCGSTAB is quite attractive as well. This is further consolidated when taking the average time to solution into account, which is shown in figures 6.5 and 6.6. As evident, using the smoother as CGS is only efficient for (very) small
Applications

grid sizes. Otherwise, relying on a BiCGSTAB with restart yields the best overall results. It is, thus, used as default CGS for the remainder of this chapter.

Please note, that the results presented here are obtained using a FV discretization. Results for a FD variant are, however, comparable in general and, thus, omitted here.

Error Convergence

In this paragraph, we check the correctness of our implementation by evaluating the error for the test cases described above. Since the pressure has Neumann boundary conditions on all sides, the solution to the governing system is not unique. Thus, before actually evaluating the error, a pressure normalization step needs to be applied. We set up a suitable function and inject a call to it directly into the error evaluation function. Usually, it would be necessary to determine the average value of the exact solution and the current guess to calculate a suitable corrections. In our case, however, the exact solution is chosen such that the integral of the pressure over the given domain is zero. Using this, we can set up a function NormalizePressure as illustrated in listing 6.13.

```exaslang
1 Function NormalizePressure@finest {
2 Expr numCells = /* ... */
3 
4 Var meanP = 0.0
5 loop over p with reduction (+ : meanP) {
6   meanP += p
7 }
8 
9 meanP /= numCells
10 
11 loop over p {
12   p -= meanP
13 }
14 }
```

Listing 6.13: ExaSlang 4 example of a function normalizing pressure values around zero.

As solver we use a complete multigrid this time. It uses a \(v(4, 4)\)-cycle type with red-black colored Vanka smoothers. We regard the infinity norm of the error for varying grid sizes and distinguish between the different components \(p, u, v\) and, where applicable, \(w\).
Figure 6.3: Average number of iterations required to solve the Stokes equations using different solvers in 2D.
Figure 6.4: Average number of iterations required to solve the Stokes equations using different solvers in 3D.
Figure 6.5: Average time to solution for solving the Stokes equations using different solvers in 2D.
Figure 6.6: Average time to solution for solving the Stokes equations using different solvers in 3D.
Figure 6.7: Obtained error values and convergence for the Stokes equations in 2D.
Figure 6.8: Obtained error values and convergence for the Stokes equations in 3D.
Figures 6.7 and 6.8 summarize the obtained results. As evident, the error convergence for the pressure component is not satisfactory in 3D. Upon closer inspection, we find that the insufficient approximation of the volume integral on the right-hand side seems to be the source of this deterioration. Integration of the right-hand side is commonly done by evaluating the analytical function at the cell centers and multiplying the result with the cell volume, respective area in 2D. One alternative can be using analytical formulations of the integrals. For the case of axis-aligned staggered grids, this yields

\[
\iiint_{\Omega_{i+1/2,j}} f_u = (y_{j+1}^{\text{node}} - y_j^{\text{node}}) \left( -2\pi \cos(2\pi x_i^{\text{cell}}) + 2\pi \cos(2\pi x_{i-1}^{\text{cell}}) \right) \\
+ (x_i^{\text{cell}} - x_{i-1}^{\text{cell}}) \left( -\pi \sin(\pi y_{j+1}^{\text{node}}) + \pi \sin(\pi y_j^{\text{node}}) \right) \\
+ (y_{j+1}^{\text{node}} - y_j^{\text{node}}) \left( \sin(4\pi x_i^{\text{cell}}) - \sin(4\pi x_{i-1}^{\text{cell}}) \right)
\]

(6.22)

\[
\iiint_{\Omega_{i,j+1/2}} f_u = (y_j^{\text{cell}} - y_{j-1}^{\text{cell}}) \left( \pi \sin(\pi x_{i+1}^{\text{node}}) - \pi \sin(\pi x_i^{\text{node}}) \right) \\
+ (x_{i+1}^{\text{node}} - x_i^{\text{node}}) \left( 2\pi \cos(2\pi y_j^{\text{cell}}) - 2\pi \cos(2\pi y_{j-1}^{\text{cell}}) \right) \\
+ (x_{i+1}^{\text{node}} - x_i^{\text{node}}) \left( \sin(4\pi y_j^{\text{cell}}) - \sin(4\pi y_{j-1}^{\text{cell}}) \right)
\]

(6.23)

\[
\iiint_{\Omega_{i,j}} f_p = (y_{j+1}^{\text{node}} - y_j^{\text{node}}) \left( \sin(2\pi x_{i+1}^{\text{node}}) - \sin(2\pi x_i^{\text{node}}) \right) \\
+ (x_{i+1}^{\text{node}} - x_i^{\text{node}}) \left( \sin(2\pi y_{j+1}^{\text{node}}) - \sin(2\pi y_j^{\text{node}}) \right)
\]

(6.24)

for 2D and analogously for 3D. Using these analytical formulations, we can reduce the error and finally obtain the expected convergence rate, as shown in figures 6.9 and 6.10.

One last aspect that needs to be handled with care is how boundary conditions are implemented. For our application, there are three cases to consider. First, Neumann boundary conditions are assumed for the pressure. Since it is never read outside the inner region, however, nothing needs to be done here. Second, unknowns can be located directly on the physical boundary. This is the case for x-staggered values on the
Figure 6.9: Obtained error values and convergence for the Stokes equations using analytically integrated right-hand sides in 2D.
Figure 6.10: Obtained error values and convergence for the Stokes equations using analytically integrated right-hand sides in 3D.
east and west boundaries, for y-staggered values on the north and south boundaries, and so on. Here, the given Dirichlet expression can simply be evaluated at the corresponding position. Third, there are unknowns for which suitable values need to be written to the ghost layers beyond the physical boundary. Let us consider, as an example, \( u \) at the south boundary. The simplest option for reconstructing the required value for \( u_{i,j} \) is using \( 2b - u_{i,j+1} \), where \( b \) is the Dirichlet expression evaluated at the physical boundary. Doing so, however, deteriorates the error convergence order. A better alternative, which we also use in our case, is given by using a higher order interpolation, e.g. \( \frac{8}{3}b - 2u_{i,j+1} + \frac{1}{3}u_{i,j+2} \).

### 6.2 Navier-Stokes Equations

The behavior of highly viscous fluids can be simulated by solving the Stokes equations. In case of other fluids, the more general Navier-Stokes equations can be considered.

#### 6.2.1 Governing Equations

In this work, we regard incompressible flows of Newtonian and non-Newtonian fluids. For the Newtonian case, the Navier-Stokes equations can be given by

\[
\rho \left( \frac{\partial u}{\partial t} + (u \cdot \nabla) u \right) - \mu \nabla^2 u + \nabla p = \tilde{f}
\]

\[
\nabla \cdot u = 0.
\]

(6.25)

on a given domain \( \Omega \) using the definitions from table 6.2 and suitable boundary conditions. The right-hand side \( \tilde{f} \) includes external source terms, e.g. gravity as \( \rho g \). An extension to non-Newtonian fluids will be discussed later in section 6.2.4.

At this point, one can also see the connection to the Stokes equations discussed in the previous section. Neglecting the convective term

\[
(u \cdot \nabla) u
\]

(6.26)

and solving for the steady state, i.e. setting the time derivative term to zero, directly yields eq. (6.1).

Introducing the kinematic viscosity as

\[
\nu = \frac{\mu}{\rho}
\]

(6.27)
6.2 Navier-Stokes Equations

<table>
<thead>
<tr>
<th>d</th>
<th>dimensionality</th>
<th>2 or 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>time</td>
<td>( \mathbb{R} )</td>
</tr>
<tr>
<td>u</td>
<td>velocity as vector</td>
<td>( \Omega \mapsto \mathbb{R}^d )</td>
</tr>
<tr>
<td>u</td>
<td>velocity component in x-direction</td>
<td>( \Omega \mapsto \mathbb{R} )</td>
</tr>
<tr>
<td>v</td>
<td>velocity component in y-direction</td>
<td>( \Omega \mapsto \mathbb{R} )</td>
</tr>
<tr>
<td>w</td>
<td>velocity component in z-direction</td>
<td>( \Omega \mapsto \mathbb{R} )</td>
</tr>
<tr>
<td>f</td>
<td>source term for the velocity as vector</td>
<td>( \Omega \mapsto \mathbb{R}^d )</td>
</tr>
<tr>
<td>f_u</td>
<td>x-component of velocity source term</td>
<td>( \Omega \mapsto \mathbb{R} )</td>
</tr>
<tr>
<td>f_v</td>
<td>y-component of velocity source term</td>
<td>( \Omega \mapsto \mathbb{R} )</td>
</tr>
<tr>
<td>f_w</td>
<td>z-component of velocity source term</td>
<td>( \Omega \mapsto \mathbb{R} )</td>
</tr>
<tr>
<td>p</td>
<td>pressure</td>
<td>( \Omega \mapsto \mathbb{R} )</td>
</tr>
<tr>
<td>( \rho )</td>
<td>density</td>
<td>( \mathbb{R} )</td>
</tr>
<tr>
<td>( \mu )</td>
<td>dynamic viscosity</td>
<td>( \mathbb{R} )</td>
</tr>
<tr>
<td>( \nu )</td>
<td>kinematic viscosity</td>
<td>( \mathbb{R} )</td>
</tr>
<tr>
<td>g</td>
<td>gravity</td>
<td>( \mathbb{R}^d )</td>
</tr>
</tbody>
</table>

Table 6.2: Symbols used for the Navier-Stokes application.

and dividing by \( \rho \), eq. (6.25) can be reformulated as

\[
\frac{\partial u}{\partial t} + (u \cdot \nabla) u - \nu \nabla^2 u + \frac{1}{\rho} \nabla p = f
\]

\[
\nabla \cdot u = 0,
\]

(6.28)

where \( f = \tilde{f}/\rho \). Thus, if gravity is the only source term, \( f = g \).

For easier mapping to our DSL later on, we rewrite eq. (6.28) as a set of scalar equations. In 2D, this yields

\[
\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} - \nu \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) + \frac{1}{\rho} \frac{\partial p}{\partial x} = f_u
\]

\[
\frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} - \nu \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) + \frac{1}{\rho} \frac{\partial p}{\partial y} = f_v
\]

(6.29)

\[
\frac{\partial}{\partial x} u + \frac{\partial}{\partial y} v = 0
\]
and, in 3D,
\[
\begin{align*}
\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} + w \frac{\partial u}{\partial z} - \nu \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) + \frac{1}{\rho} \frac{\partial p}{\partial x} &= f_u \\
\frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} + w \frac{\partial v}{\partial z} - \nu \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2} \right) + \frac{1}{\rho} \frac{\partial p}{\partial y} &= f_v \\
\frac{\partial w}{\partial t} + u \frac{\partial w}{\partial x} + v \frac{\partial w}{\partial y} + w \frac{\partial w}{\partial z} - \nu \left( \frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2} \right) + \frac{1}{\rho} \frac{\partial p}{\partial z} &= f_w \\
\frac{\partial}{\partial x} u + \frac{\partial}{\partial y} v + \frac{\partial}{\partial z} w &= 0.
\end{align*}
\]

(6.30)

\subsection{Discretization}

As for the Stokes case, we rely on staggered grids on which we discretize using finite volumes. Using finite differences would also be possible, but is less advantageous for non-uniform grids where the aspect ratios of cells are comparably large or small. In this section, we will focus on the 2D case as the 3D case can be derived analogously. Most of the terms in eq. (6.29) can be discretized as before, with the exception of the time derivative and the convective term. For the former, we use a first-order backward difference, i.e.

\[
\frac{\partial u^{n+1}}{\partial t} \approx \frac{u^{n+1} - u^n}{\Delta t},
\]

(6.31)

which represents an implicit method using \(n\) once again as the current time step.

Handling the convective term (c.f. eq. (6.26)) requires more effort since it is non-linear. First, we apply the standard steps for a FV discretization to wind up with

\[
\int_{\partial \Omega_{i+1/2,j}} u (\mathbf{u} \cdot \mathbf{n}) \, dx \, dy
\]

(6.32)

and

\[
\int_{\partial \Omega_{i,j+1/2}} v (\mathbf{u} \cdot \mathbf{n}) \, dx \, dy
\]

(6.33)

for x- and y-staggered control volumes, respectively. However, in many algorithmic parts, such as parts of the solver, a linearized version is re-
quired. We thus discuss two prominent options to implement the required linearization, namely Picard and Newton. In general, the linearization can take place globally, that is for the whole system of equations at once, or locally, that is only for a single cell or for a small number of them. For reasons of compactness, we regard linearization of the continuous term eq. (6.26). The linearized terms can afterwards be discretized as discussed previously.

The most straightforward approach is using what is called a Picard linearization or sometimes also frozen coefficients or lagged coefficients. Its main idea, as the name suggest, is to freeze some parts of the nonlinear term, denoted by the superscript $k$, while the remaining parts can be updated or solved for. The latter are denoted by the superscript $k+1$. For eq. (6.26) this can be done as

$$ (u^{k+1} \cdot \nabla) u^{k+1} \approx (u^k \cdot \nabla) u^{k+1}. \quad (6.34) $$

Combining it with the discretization of the remaining terms of the Navier-Stokes equations yields a system of the form

$$
\begin{pmatrix}
A_{11} & B_1 \\
A_{22} & B_2 \\
C_1 & C_2 \\
\end{pmatrix}
\begin{pmatrix}
u \\
v \\
p \\
\end{pmatrix}
= 
\begin{pmatrix}
f_u \\
f_v \\
0 \\
\end{pmatrix}.
\quad (6.35)
$$

A more accurate approximation can be given by using a Newton linearization. It is given by

$$ (u^{k+1} \cdot \nabla) u^{k+1} \approx (u^k \cdot \nabla) u^{k+1} + (u^{k+1} \cdot \nabla) u^{k} - (u^k \cdot \nabla) u^k. \quad (6.36) $$

Due to the additional terms, there is however an additional coupling between the velocity components. The resulting system for the whole Navier-Stokes equations thus takes the form

$$
\begin{pmatrix}
A_{11} & A_{12} & B_1 \\
A_{21} & A_{22} & B_2 \\
C_1 & C_2 & \end{pmatrix}
\begin{pmatrix}
u \\
v \\
p \\
\end{pmatrix}
= 
\begin{pmatrix}
f_u \\
f_v \\
0 \\
\end{pmatrix}.
\quad (6.37)
$$

Mapping to ExaSlang

Mapping the discretization to ExaSlang is comparable to the Stokes case. First, we set up the discretized balance equations on layer 2 as shown...
in listing 6.14. Apart from now having variable values for \( \nu \) and \( \rho \), the diffusion and pressure parts are equivalent to the Stokes versions. Particularly interesting are the newly added time derivative in line 3 and the convective term starting in line section 6.2.2. The latter has been already linearized using a Picard linearization. Here, we use a new special function to mark field accesses as \texttt{frozen}. The concrete implications of that will be discussed later.

```plaintext
Equation uEquation {
  // time derivative
  ( u - uOld ) * vf_xStagCellVolume / dt

  // diffusion
  + ( u@[ 0, 0] - u@[ 1, 0] )
    * integrateOverXStaggeredEastFace ( nue )
    / ( vf_nodePos_x@[ 1, 0] - vf_nodePos_x@[ 0, 0] )
  + ( u@[ 0, 0] - u@[-1, 0] )
    * integrateOverXStaggeredWestFace ( nue )
    / ( vf_nodePos_x@[ 0, 0] - vf_nodePos_x@[-1, 0] )
  + ( u@[ 0, 0] - u@[ 0, 1] )
    * integrateOverXStaggeredNorthFace ( nue )
    / ( vf_cellCenter_y@[ 0, 1] - vf_cellCenter_y@[ 0, 0] )
  + ( u@[ 0, 0] - u@[ 0, -1] )
    * integrateOverXStaggeredSouthFace ( nue )
    / ( vf_cellCenter_y@[ 0, 0] - vf_cellCenter_y@[ 0, -1] )

  // convection
  + integrateOverXStaggeredEastFace ( u * frozen ( u ) )
  - integrateOverXStaggeredWestFace ( u * frozen ( u ) )
  + integrateOverXStaggeredNorthFace ( u * frozen ( v ) )
  - integrateOverXStaggeredSouthFace ( u * frozen ( v ) )

  // pressure coupling
  + integrateOverXStaggeredEastFace ( p / rho )
  - integrateOverXStaggeredWestFace ( p / rho )
```

154
6.2 Navier-Stokes Equations

Listing 6.14: ExaSlang 2 definition of the discretized Navier-Stokes equations with applied Picard linearization.

A Newton linearization can be expressed as well by adding additional terms, such as shown in listing 6.15, to the equation.

Listing 6.15: ExaSlang 2 example of additional terms to be added to the equation from listing 6.14 to switch to a Newton linearization.

As before, operators can be extracted from the given equations. Listing 6.16 shows one example for this, in this case the Newton linearization. For the Picard variant, line 4 is omitted since it captures the then not existent coupling between u and v in the operator A12.
6 Applications

```
generate operators @all {
    equation for u is uEquation store in {
        u => A11
        v => A12
        p => B1
    }

    /* analogous for vEquation */

equation for p is pEquation store in {
    u => C1
    v => C2
}
}
```

Listing 6.16: ExaSlang 2 example of extracting operators from the discretized Navier-Stokes equations as set up in listings 6.14 and 6.15.

At this point, frozen field accesses are handled in a specialized way for the first time. In essence, they are regarded as constant when sorting the equation terms into the left- and right-hand sides, and when extracting the stencil coefficients. As such, frozen field accesses may now become part of the stencil coefficients, thereby expressing a locally linearized operator.

Extracting the operators as shown here sets up stencils with stencil coefficients consisting of a high number of expressions, which are re-evaluated frequently. In some cases, it is more attractive to compute the stencil coefficients once and store them for repeated use. This is also required if a global linearization of the system is to be done. In ExaSlang, this is a two-stage process. On layer 2, the shapes of the stencil templates are defined as shown in listing 6.17.

```
Operator A11 from StencilTemplate
    on Face_x of global {
        [ 0,  0] =>
        [-1,  0] =>
        [ 1,  0] =>
        [ 0, -1] =>
        [ 0,  1] =>
    }
```

Listing 6.17: ExaSlang 2 definition of a stencil template to store the $A_{11}$ operator for the Navier-Stokes equations.
The actual coefficients can then be calculated and stored in a function on layer 4. An example for this is given in listing 6.18. Defining operators in this fashion is, as evident, more complicated as simply extracting them from the discretized equations. We, thus, plan to generate the assembly functions automatically in future versions of the code generator. Independent of that, stencils and stencil fields can be mixed as required. That is, for instance, the diffusion part could be modeled as before while only the convection part is implemented as a stencil field.

```plaintext
Function AssembleStencil@all {
  loop over A11 {
    // diffusion
    A11:[-1, 0] = - ( 
      integrateOverXStaggeredWestFace ( nue ) 
      / ( vf_nodePos_x@[ 0, 0] 
        - vf_nodePos_x@[ -1, 0] ) 
    )
    A11:[ 1, 0] = - ( 
      integrateOverXStaggeredEastFace ( nue ) 
      / ( vf_nodePos_x@[ 1, 0] 
        - vf_nodePos_x@[ 0, 0] ) 
    )
    A11:[ 0, -1] = - ( 
      integrateOverXStaggeredSouthFace ( nue ) 
      / ( vf_cellCenter_y@[ 0, 0] 
        - vf_cellCenter_y@[ 0, -1] ) 
    )
    A11:[ 0, 1] = - ( 
      integrateOverXStaggeredNorthFace ( nue ) 
      / ( vf_cellCenter_y@[ 0, 1] 
        - vf_cellCenter_y@[ 0, 0] ) 
    )

    // convection
    A11:[-1, 0] -= 0.5 * integrateOverXStaggeredWestFace ( u )
    A11:[ 0, 0] -= 0.5 * integrateOverXStaggeredWestFace ( u )
    A11:[ 1, 0] += 0.5 * integrateOverXStaggeredEastFace ( u )
    A11:[ 0, 0] += 0.5 * integrateOverXStaggeredEastFace ( u )
  }
}
```
6 Applications

Listing 6.18: ExaSlang 4 function assembling stencil (field) coefficients.

6.2.3 Solver

When solving for the Navier-Stokes equations as discretized in the previous section, there are multiple options. Most relevant for our domain is using multigrid approaches, which will also be the focus of this section. The first option is straightforward linearization on the finest level and then solving the linear system using a standard multigrid. Operators on the coarser grid can be constructed using a Galerkin approach or – rather crudely – by coarsening the stencil coefficients themselves. The main issue of this approach is, that comparably good initial guesses are required. Alternatively, heavily damping the solver is also possible. As this, however, increases the computational costs tremendously it is also not attractive for us. An alternative is given by using a non-linear multigrid method, also called full approximation scheme (FAS). Details are discussed in the next section. If a non-linear multigrid is not suitable, one can also use other approaches such as the semi-implicit method for pressure linked equations (SIMPLE) [85]. In essence, it works as follows: First, one solves for the single velocity components separately. This involves setting up a system for each of the equations, usually also linearized using, e.g., Picard. The resulting system(s) can then be solved using a suitable linear solver, such as a straight-forward geometric multigrid. Since this approach neglects all coupling between the velocity components and the pressure, a pressure correction is re-
quired. It is solved for next, before being applied. This step, again, can be tackled using a standard multigrid solver. More details on the mathematical background, and also on the implementation in ExaSlang, can be found in [9].

Full Approximation Scheme (FAS)

The key idea of the FAS is not only restricting the residual to the coarser right-hand side, but also the current solution to obtain an approximate solution $\tilde{x}$ on the coarser level. It can then be used as an initial guess on the coarser level and to change the equation to be solved. Instead of our original error equation eq. (2.45), which is given by

$$A(e) = r$$

for non-linear operators $A$, we now solve for

$$A(x) = A(\tilde{x}) + r$$

where $x = e + \tilde{x}$. It can easily be seen, that eqs. (6.38) and (6.39) are equivalent if $A$ is linear. After solving, the correction to be applied to the finer level can be calculated by subtracting the approximate solution from the obtained one. The final method can finally be set up as shown in algorithm 7.

Before implementing the FAS in ExaSlang, some more points need to be addressed. First, the correct restriction and interpolation operators need to be chosen. In the context of FV, one has to take care if discretized functions or integrals of them are to be transferred. The solution and the obtained correction fall in the former category while the residual falls in the latter. Consider a constant solution and residual as example. Transfer operators for discretized functions, such as the solution, must be able to preserve values. That is, in our example the value on the coarser and finer levels must match after applying the operator. In contrast, integrated values must adapted with respect to the area they are integrated over. For a 2D uniform grid, this translates to the value on the coarser grid having to be four times as large as on the finer level. Second, a non-linear smoother would usually be required. While we do not directly support them, we allow using linearized operators as, e.g., obtained from listing 6.15, in block smoothers. This effectively triggers a linearization at the grid locations contained in the block to be solved for each time the solver is applied. If, instead, the system is linearized
Algorithm 7: Extension of algorithm 1 to the full approximation scheme (FAS) for non-linear $A$.

```plaintext
if coarsest level then
    solve directly
else
    $x^k \leftarrow S^{s_1}(x^k, A, b)$ \{pre-smoothing\}
    $r \leftarrow b - A(x^k)$ \{residual computation\}
    $r_{coarser} \leftarrow R r$ \{residual restriction\}
    $\tilde{x}_{coarser} \leftarrow R x^k$ \{solution restriction\}
    $b_{coarser} \leftarrow A_{coarser}(\tilde{x}_{coarser}) + r_{coarser}$ \{coarser rhs setup\}
    $x_{coarser} \leftarrow V_{coarser}(0, A_{coarser}, b_{coarser})$ \{recursion\}
    $e_{coarser} = x_{coarser} - \tilde{x}_{coarser}$ \{error computation\}
    $e \leftarrow P e_{coarser}$ \{prolongation\}
    $x^k \leftarrow x^k + e$ \{correction\}
    $x^{k+1} \leftarrow S^{s_2}(x^k, A, b)$ \{post-smoothing\}
end
```

globally to be stored in stencil fields, our default linear smoothers are applicable without further changes. Last, whenever evaluating the residual or its norm, it is crucial to linearize involved operators either locally on-the-fly or globally directly beforehand. Otherwise, wrong values are obtained and, as one effect, convergence can not be detected correctly.

Mapping to ExaSlang

When implementing a solver for the discretization described previously, there are two options. The first, and most straightforward, is using our `generate solver` interface. For non-linear problems, we implement two extensions. Using the newly introduced parameter `solver_useFAS` instructs our framework to implement algorithm 7. If locally linearized operators are used, no more work is required. Otherwise, calls to the function setting up the linearized operator need to be injected at appropriate places. At the bare minimum, this means either appending it to the `setCoarseSolution` stage or prepending to the `restriction` stage, as well as prepending it to the `updateResidual` stage.

The second option is implementing the solver by hand in ExaSlang 3. While this option requires more coding, it is also more flexible. List-
6.2 Navier-Stokes Equations

Listing 6.19 shows a shortened version of a possible implementation for the Picard case.

```plaintext
Function Cycle@(all but coarsest) {
    Smoother ( )

    AssembleStencil ( )

    residual_u = rhs_u - ( A11 * u + B1 * p )
    /* residual_v and residual_p */

    approx_u@coarser = RestrictionFaceX * u
    u@coarser = approx_u@coarser
    /* approx_v, v, approx_p and p */

    AssembleStencil@coarser ( )

    rhs_u@coarser = ( RestrictionFaceXIntegral * residual_u
    + A11@coarser * approx_u@coarser
    + B1 @coarser * approx_p@coarser )
    /* rhs_v and rhs_p */

    Cycle@coarser ( )

    u@coarser -= approx_u@coarser
    u += CorrectionFaceX * u@coarser
    /* v and p */

    Smoother ( )
}
```

Listing 6.19: ExaSlang 3 example of a FAS solver for the Navier-Stokes equations. Parts concerning $v$ and $p$ have been omitted to the sake of compactness.

6.2.4 Extensions

At this point, extending our base model and implementation to be able to simulate a wider range of effects is possible. This includes adding temperature as a quantity to be solved for and lifting the restriction of fluids having to be Newtonian. As a reference, we consider research done in collaboration with Gundolf Haase and Diego Vasco. During
Figure 6.11: Illustration of the SIMPLE algorithm extended for non-Newtonian non-isothermal fluids.

In this work, we focus on implementing the same extensions, but incorporated in our FAS solver. For this, multiple changes are required. First, we replace the formerly constant viscosity $\mu$ and density $\rho$ with scalar fields localized at the cell centers. Boundary conditions are not required, since their values are not read outside the inner region. Please note, that even though the density is not fixed anymore, the fluid is still considered to be incompressible in the model provided by our collaboration partners. As such, it is not necessary to change the governing equations eq. (6.25). Instead, they are simply extended to incorporate temperature and, using the updated definitions from table 6.3, we end up with

$$
\rho \left( \frac{\partial u}{\partial t} + (u \cdot \nabla) u \right) - \mu \nabla^2 u + \nabla p = \tilde{f} \\
\nabla \cdot u = 0
$$

(6.40)

$$
\rho \left( \frac{\partial \phi}{\partial t} + (u \cdot \nabla) \phi \right) - \gamma \nabla^2 \phi = 0.
$$
Table 6.3: Symbols used for the Navier-Stokes application for non-Newtonian and non-isothermal fluids (replaces table 6.2).

The newly introduced quantities $\phi$ and $\gamma$ are added as scalar cell-centered fields as well.

Next, we model the temperature dependency in the velocity equations. One often used option is adding an additional term to $\tilde{f}$ in order to model buoyancy effects. Another option, which we choose to implement, is adapting the density and viscosity directly. This also allows incorporating non-Newtonian properties in the same step. Implementation-wise, we set up a dedicated function updating $\rho$ and $\mu$ as well as $\gamma$. It is then called each time a linearization is triggered, directly before the stencil coefficients are calculated. An example for Newtonian fluids is given in listing 6.20 where we model the density according to [62], and the remaining quantities as proposed in [111]. Extending this function for non-Newtonian fluids is straight-forward by adding a suitable model for $\mu$, e.g. [82], which then replaces the update in the shown listing. Please note that approach only works if a global linearization scheme is implemented. Otherwise, integrating the dependencies directly into the layer 2 equations would be possible as well, but usually not advisable since computing them can be come quite costly.

```c
loop over phi {
    Expr phiCelsius = phi - 273.15
    rho = ( 999.840281
            + 0.0673268 * phiCelsius
            - 0.00894484 * phiCelsius**2
            + 8.7846287e-5 * phiCelsius**3
            - 6.6213979e-7 * phiCelsius**4 )
```
\[ \mu = 3.8208e-2 \left( \phi - 252.33 \right) \]

\[
\text{Expr } cp = \left( -3755.9 + 67.953 \phi - 0.19149 \phi^2 + 1.7850e-4 \phi^3 \right)
\]

\[
\text{Expr } k = \left( -0.63262 + 7.1959e-3 \phi - 1.144e-5 \phi^2 + 4.2365e-9 \phi^3 \right)
\]

\[ \gamma = \frac{k}{cp} \]

Listing 6.20: ExaSlang 4 implementation of an update routine for \( \rho, \mu \) and \( \gamma \) assuming a Newtonian fluid.

Apart from adding a call to the function updating the fluid properties, two more changes are required to the \texttt{AssembleStencil} function. First, it has to be adapted to implement eq. (6.40), that is the velocity equations have to be multiplied with the density. At this point, it is also vital to take care that all accesses to \( \rho \) and \( \mu \) are wrapped in correct evaluate or integrate functions. Second, an operator for the temperature equation has to be constructed. We choose to do a Picard linearization independent of the linearization approach chosen for the other equations. Doing so leads to a single operator, which we call \( E \).

Lastly, the time stepping scheme and the solver have to be adapted. Extending the former to include the new quantities is straight-forward. Generally, this is also the case for the FAS multigrid solver, with the exception of the smoother. Since we chose not to include the temperature dependency in the source term for the velocity equations, \( \phi \) is not directly included in them. Likewise, using a Picard linearization for the temperature equation eliminates the explicit velocity coupling. This allows splitting the smoother into two parts which can be applied independently. For the first part, we reuse the smoother already present. Afterwards, we apply a simple damped red-black point smoother for the temperature. This last step can be done multiple times to obtain a more accurate solution. In practice, we mostly use two temperature smoothing steps for each application of velocity and pressure smoothing. Listing 6.21 shows this in a possible implementation for three smoother iterations.
Function Smoother {
    repeat 3 times {
        color with {
            ( ( i0 + i1 ) % 2 ),
        solve locally at p relax 0.8 {
            u[0, 0] => A11[0, 0] * u[0, 0]
            + A12[0, 0] * v[0, 0]
            + B1[0, 0] * p[0, 0]
            == rhs_u[0, 0]
            u[1, 0] => A11[1, 0] * u[1, 0]
            + A12[1, 0] * v[1, 0]
            + B1[1, 0] * p[1, 0]
            == rhs_u[1, 0]
            v[0, 0] => A22[0, 0] * v[0, 0]
            + A21[0, 0] * u[0, 0]
            + B2[0, 0] * p[0, 0]
            == rhs_v[0, 0]
            v[0, 1] => A22[0, 1] * v[0, 1]
            + A21[0, 1] * u[0, 1]
            + B2[0, 1] * p[0, 1]
            == rhs_v[0, 1]
            p[0, 0] => C1[0, 0] * u[0, 0]
            + C2[0, 0] * v[0, 0]
            == rhs_p[0, 0]
        }
    }
    repeat 2 times {
        color with {
            ( ( i0 + i1 ) % 2 ),
        solve locally at phi relax 0.7 {
            phi => E * phi == rhs_phi
        }
    }
    AssembleStencil ( )
}

Listing 6.21: ExaSlang 3 implementation of a smoother for the extended Navier-Stokes problem.
6.2.5 Results

To assess the quality of the generated solver, and examine the impact of the different variations described before, we regard the classical test case of a lid-driven cavity. Here, velocity boundary conditions are set to zero everywhere, but for the x-component on the top boundary. We use a parabolic velocity profile implemented as a Dirichlet boundary condition with

\[ \tilde{u} \left( 4 \left( \bar{x} - \bar{x}^2 \right) \bar{y} \right) \]

and

\[ \tilde{u} \left( 16 \left( \bar{x} - \bar{x}^2 \right) \left( \bar{y} - \bar{y}^2 \right) \bar{z} \right) \]

in 2D and 3D, respectively, where \( \tilde{u} \) is the maximum velocity reached at the center of the boundary. \( \bar{x}, \bar{y} \) and \( \bar{z} \) denote normalized geometrical positions, i.e. \( \bar{x} = \frac{x}{l} \) with \( l \) as the edge length of the domain.

Using these symbols, and the kinematic viscosity \( \nu \) introduced before, one can compute the Reynolds number, which is frequently used in fluid mechanics to categorize different test cases, as

\[ Re = \frac{l \tilde{u}}{\nu}. \] (6.41)

In the Newtonian and isothermal case, we fix the viscosity and density such that our fluid mimics water at 25 Celsius, i.e. \( \mu = 8.9 \cdot 10^{-4} \) and \( \rho = 997.0479 \). In consequence, \( \nu \) is always approximately \( 8.9264 \cdot 10^{-7} \) and varying \( \tilde{u} \) allows studying test cases with different Reynolds numbers. Figure 6.12 displays the results obtained for such a series over several orders of magnitude. At low Reynolds numbers, the diffusive part dominates. With increasing lid velocities, and thus increasing Reynolds numbers, the convective parts increase in importance as does the turbulence in the fluid.

Next, we examine performance impacts of some choices in more detail. For all test cases, we implement a rudimentary adaptive time stepping. If the solver for a given time step has not reduced the residual sufficiently after a certain number of iterations, or diverges, we halve the time step size and try again. As an optimization we try doubling the time step size after a fixed number of steps. Where not denoted otherwise, we perform at most 20 solver iterations, i.e. multigrid cycles, to reduce the \( L2 \) norm of the residual below \( 10^{-10} \) and increase the time.
6.2 Navier-Stokes Equations

(a) \( Re = 1.12 \cdot 10^1, t = 250 \).

(b) \( Re = 1.12 \cdot 10^1, t = 500 \).

(c) \( Re = 1.12 \cdot 10^1, t = 1000 \).

(d) \( Re = 1.12 \cdot 10^2, t = 250 \).

(e) \( Re = 1.12 \cdot 10^2, t = 500 \).

(f) \( Re = 1.12 \cdot 10^2, t = 1000 \).
Figure 6.12: Results for the lid-driven cavity test case for varying Reynolds numbers at different points in time. Color represents the magnitude of velocity.
step size every 16 steps. For all global linearization approaches, we additionally allow the solver to perform up to 256 iterations in the first time step.

As domains, we choose a square and box with an edge length of 0.1 which we discretize using $256^2$ cells in 2D and $64^3$ cells in 3D. Solver-wise, we follow the ideas discussed so far and regard FAS multigrid solvers with three pre- and three post-smoothing steps using an $\omega = 1$ which corresponds to no damping. The coarse grid problem is solved using a BiCGSTAB solver with restarts after every 128 iterations. Evaluation is then performed on the workstation platform detailed in section 9.3 and all applications are parallelized using OpenMP and four threads. Instead of running the whole simulation, we restrict the maximum simulated time in our tests to keep execution times manageable. More precisely, we advance until we reach $t_{\text{max}} = 10$.

Table 6.4 summarizes the results for the Newtonian and isothermal case on uniform grids. The first thing that becomes obvious is that using a Newton linearization, whether local or global, is always more expensive than its Picard counter-part. One major factor for this is the time required in the inversion of small local systems as required by the smoothers. While a full inversion of a dense matrix is required in the Newton case, a more efficient inversion is possible in the Picard case due to the special matrix structure using a Schur complement representation. The latter can be automatically implemented by our generator without modification as discussed for the Stokes equations in section 6.1.3. It is also possible to apply the re-discretization and re-linearization fewer times, e.g. not for every smoother iteration but only for the first or the last. Although not shown in the table, we examined this possibility as well and found that this can indeed improve the performance in some cases, but does not change the overall trends observed. Another outcome is that, in our application, switching from a global to a local linearization approach is beneficial in most cases as the number of multigrid cycles required for each time step is approximately halved. One exception is the 2D Picard case where, additionally, the number of required time steps increases which in turn worsens overall performance.

Next, we examine the same test case on the same machine, but for non-uniform grids. Figure 6.13 illustrates the employed grids and how the regions close to the boundaries are resolved much finer. As table 6.5 shows, the general trends stay the same: Picard is superior to Newton
and switching from global to local linearization is beneficial for most cases due to the reduced number of solver iterations required. The exception to the latter is this time the 3D Picard case and the main reason for this behavior lies in the dramatically increased complexity of the stencil coefficients. In all cases, switching to non-uniform grids increases the time spent in a single multigrid cycle and the total time to solution compared to the uniform counter-parts. On the other hand, in some cases larger time step sizes are possible, effectively reducing the number of time steps required.

Finally, we examine the performance implications of the extensions discussed in the previous section. To promote comparability, we continue to regard our test case of a lid-driven cavity. On top of it, we add Dirichlet boundary conditions for the temperature similar to the ones

---

**Table 6.4:** Performance results for the Newtonian and isothermal test case on uniform grids.
6.2 Navier-Stokes Equations

Figure 6.13: Illustration of employed non-uniform grids with different sizes. Color-coded according to the magnitude of velocity in the lid-driven cavity problem.

used in a natural convection test case. They are again designed to give a parabolic profile and are given by

\[ \phi_0 + \phi_{off} \left( 4 \left( 2\bar{x} - 1 \right) \left( \bar{y} - \bar{y}^2 \right) \right) \]

and

\[ \phi_0 + \phi_{off} \left( 16 \left( 2\bar{x} - 1 \right) \left( \bar{y} - \bar{y}^2 \right) \left( \bar{z} - \bar{z}^2 \right) \right) \]

for the 2D and 3D cases, respectively. Here, \( \phi_0 \) is the initial fluid temperature, 278.15 Kelvin or 5.0 Celsius in our case, and \( \phi_{off} \) a maximum offset set to 5.0. Table 6.6 summarizes the obtained results for global linearization schemes. The overall performance penalty for additionally
Table 6.5: Performance results for the Newtonian and isothermal test case on non-uniform grids.

<table>
<thead>
<tr>
<th>$d$</th>
<th>linearization</th>
<th>total solver time in s</th>
<th>num. time steps</th>
<th># multi-grid cycles</th>
<th>avg. num. cycles per time step</th>
<th>avg. time per cycle in ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>global Picard</td>
<td>35.27</td>
<td>87</td>
<td>1694</td>
<td>19.5</td>
<td>20.82</td>
</tr>
<tr>
<td></td>
<td>local Picard</td>
<td><strong>21.86</strong></td>
<td><strong>72</strong></td>
<td>529</td>
<td>7.35</td>
<td>43.32</td>
</tr>
<tr>
<td>2</td>
<td>global Newton</td>
<td>119.1</td>
<td>88</td>
<td>1503</td>
<td>17.1</td>
<td>79.26</td>
</tr>
<tr>
<td></td>
<td>local Newton</td>
<td>60.39</td>
<td><strong>72</strong></td>
<td><strong>511</strong></td>
<td><strong>7.10</strong></td>
<td>118.2</td>
</tr>
<tr>
<td>3</td>
<td>global Picard</td>
<td><strong>15.58</strong></td>
<td><strong>13</strong></td>
<td>77</td>
<td>5.92</td>
<td><strong>202.3</strong></td>
</tr>
<tr>
<td>3</td>
<td>local Picard</td>
<td>38.49</td>
<td>19</td>
<td>61</td>
<td>3.21</td>
<td>630.9</td>
</tr>
<tr>
<td>3</td>
<td>global Newton</td>
<td>115.0</td>
<td>80</td>
<td>147</td>
<td>1.84</td>
<td>782.4</td>
</tr>
<tr>
<td>3</td>
<td>local Newton</td>
<td>84.66</td>
<td>40</td>
<td><strong>54</strong></td>
<td><strong>1.35</strong></td>
<td>1568</td>
</tr>
</tbody>
</table>

solving for the temperature, and for having variable densities, viscosities and thermal conductivities, is on average a factor of 6.5 across our test cases. Modeling the fluid as non-Newtonian adds another factor of almost two, again on average. Further comparing the results, we see that the mean time required for performing a multigrid cycle is almost doubled in 2D, but only moderately increased in 3D, independent of the simulated fluid being Newtonian or not. However, not only are smaller time step sizes necessary, but also the number of solver iterations per time step increases. This can hint to our solver being not as effective as before which could, in part, be attributed to the indirect coupling between the Navier-Stokes and temperature equations, and the complex non-linear models for the fluid properties.

After examining the performance of our generated applications, we go on to check the scalability. As a target cluster, we choose the JUWELS system located in Jülich, Germany, with characteristics as detailed in
### 6.2 Navier-Stokes Equations

<table>
<thead>
<tr>
<th>$d$</th>
<th>linearization</th>
<th>total solver time in s</th>
<th>num. time steps</th>
<th># multi-grid cycles</th>
<th>avg. num. cycles per time step</th>
<th>avg. time per cycle in ms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>non-isothermal Newtonian</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Picard</td>
<td>44.11</td>
<td>319</td>
<td>1456</td>
<td>4.56</td>
<td>30.29</td>
</tr>
<tr>
<td>2</td>
<td>Newton</td>
<td>126.7</td>
<td>319</td>
<td>1433</td>
<td>4.49</td>
<td>86.28</td>
</tr>
<tr>
<td>3</td>
<td>Picard</td>
<td>89.37</td>
<td>80</td>
<td>470</td>
<td>5.88</td>
<td>190.1</td>
</tr>
<tr>
<td>3</td>
<td>Newton</td>
<td>400.4</td>
<td>80</td>
<td>488</td>
<td>6.10</td>
<td>820.5</td>
</tr>
<tr>
<td></td>
<td>non-isothermal non-Newtonian</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Picard</td>
<td>82.10</td>
<td>328</td>
<td>2805</td>
<td>8.55</td>
<td>29.27</td>
</tr>
<tr>
<td>2</td>
<td>Newton</td>
<td>254.1</td>
<td>328</td>
<td>2800</td>
<td>8.54</td>
<td>90.75</td>
</tr>
<tr>
<td>3</td>
<td>Picard</td>
<td>194.6</td>
<td>112</td>
<td>994</td>
<td>8.88</td>
<td>195.8</td>
</tr>
<tr>
<td>3</td>
<td>Newton</td>
<td>823.9</td>
<td>122</td>
<td>994</td>
<td>8.11</td>
<td>833.1</td>
</tr>
</tbody>
</table>

**Table 6.6:** Performance results for the non-isothermal test cases on non-uniform grids.

section 9.3. At the point of writing this work, users are allowed to allocate up to 512 nodes, which is also the maximum in our presented results. When comparing the local and global linearization schemes, we quickly find that the local approach works very well on a single fragment, but its performance quickly deteriorates with an increasingly partitioned domain. We, thus, restrict ourselves to the best suited variant for this machine – global Picard linearization. Figure 6.14 shows a basic weak scaling for 2D and 3D where we evaluate our basic lid-driven cavity test case with a hybrid OpenMP parallelization. In 2D, we use four fragments per compute node, each handled by a distinct MPI thread and internally further parallelized using 12 OpenMP threads. In 3D, we change this to eight fragments per compute node and, consequently, eight MPI ranks with 6 OpenMP ranks each. While scalability seems to be satisfactory, it is important to not overlook the fact that changing the resolution of a problem such as ours usually also incurs changes for the time step size and overall convergence behavior. Thus, performing a strong scaling might give a more reliable assessment. For it, we start with $16384^2$ and $512^3$ unknowns in 2D and 3D, respectively, on
Figure 6.14: Weak scaling of generated Navier-Stokes solvers on JUWELS for \( Re \approx 1100 \). In 2D and 3D, \( 256^2 \) and \( 64^3 \) cells per node (48 cores) are regarded, respectively.

a single node using the same partitioning and parallelization as before. We then gradually introduce more fragments while reducing the number of multigrid levels per fragment such that the overall number of unknowns solved for remains constant. The results are summarized in figure 6.15. They show a very good scalability up to a certain point at which, in the 3D case, adding more compute resources is not amortizing any more. One reason for this is the coarse-grid problem becoming increasingly large and, thus, more and more costly to solve. Switching to alternative solvers could be a possible remedy here.

Finally, it should be said that the results presented in this work are specific to the problem and solver configurations presented. As such, they still hold some potential for optimization by tuning algorithmic components and parameters. This includes other types and shapes of smoothers, the number of smoothing steps, damping parameters and the parameters involved in the time step size adaptation routine. One option to tackle this challenge is using performance prediction and optimization techniques as discussed in section 7.1. Nevertheless, we can
Figure 6.15: Strong scaling of generated Navier-Stokes solvers on JUWELS for $Re \approx 1 \times 10^6$. In 2D and 3D, this corresponds to approximately $8 \cdot 10^8$ and $5 \cdot 10^8$ unknowns, respectively.

already show satisfactory performance and scalability for our generated multigrid solvers.

6.3 Shallow Water Equations

The last application discussed in this work is the simulation of ocean flows. It is a summary of the contributions and results also published in [12].

6.3.1 Governing Equations

In general, the behavior of ocean flows can be modeled using the Navier-Stokes equations as discussed before. In practice, however, one frequently uses the fact that the simulation domain is very large in the x- and y-dimensions while being very small in the z-dimension. This allows approximating the overall behavior by averaging the governing
equations over the z-dimension. Doing so results in the so-called shallow water equations (SWE) which can be formulated as

\[
\frac{\partial}{\partial t} \begin{bmatrix} h \\ hu \\ hv \end{bmatrix} + \frac{\partial}{\partial x} \begin{bmatrix} hu^2 + gh^2/2 \\ hv^2 \\ hvw \end{bmatrix} + \frac{\partial}{\partial y} \begin{bmatrix} hv \\ hvw \\ hv^2 + gh^2/2 \end{bmatrix} = \begin{bmatrix} 0 \\ -\frac{\partial}{\partial x} (ghb/2) \\ -\frac{\partial}{\partial y} (ghb/2) \end{bmatrix},
\]

(6.42)
on a given domain \( \Omega \) using the quantities detailed in table 6.7 and suitable boundary conditions. In our context, the bathymetry \( b \) is measured relative to the ocean line, as illustrated in figure 6.16. That means, \( b \) is usually negative and the water surface is level if \( h = -b \) everywhere.

Next, we define

\[
q = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}
\]

(6.43)
as the quantities to be solved for. Using

\[
F = \begin{bmatrix} q_1 \\ q_1^2/q_0 + gq_0^2/2 \\ q_1q_2/q_0 \end{bmatrix},
\]

(6.44)
\[
G = \begin{bmatrix} q_2 \\ q_1q_2/q_0 \\ q_2^2/q_0 + gq_0^2/2 \end{bmatrix}
\]

(6.45)
and
\[ S = S_h + S_v = \begin{bmatrix} 0 & 0 \\ -gg_0b/2 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ -gg_0b/2 & 0 \end{bmatrix} \]  \hspace{1cm} (6.46)

the SWE can be rewritten as
\[ \frac{\partial}{\partial t} q + \frac{\partial}{\partial x} F + \frac{\partial}{\partial y} G = \frac{\partial}{\partial x} S_h + \frac{\partial}{\partial y} S_v \]  \hspace{1cm} (6.47)

### 6.3.2 Discretization

We assume a uniform, non-staggered grid on which we discretize the SWE using the finite volume method (FVM) as introduced in section 2.1.2. Moreover, we approximate the time derivative with a first order explicit scheme. Doing so results in

\[
\begin{align*}
\Delta x \Delta y \left( q_{i,j}^{n+1} - q_{i,j}^n \right) \\
+ \Delta t \Delta y \left( F_{i+1/2,j}^n - F_{i-1/2,j}^n \right) \\
+ \Delta t \Delta x \left( G_{i,j+1/2}^n - G_{i,j-1/2}^n \right) \\
= \Delta t \Delta y \left( S_{h,i+1/2,j}^n - S_{h,i-1/2,j}^n \right) \\
+ \Delta t \Delta x \left( S_{v,i,j+1/2}^n - S_{v,i,j-1/2}^n \right)
\end{align*}
\]  \hspace{1cm} (6.48)

where \((i, j)\) is the cell index and \(n\) the time step. Here, half-indices are used to express evaluation at control volume interfaces which we call fluxes. The next step is finding suitable approximations for them. The most straight-forward approach is using a central flux approximation leading to

\[ F_{i \pm 1/2,j}^n = \frac{1}{2} \left( F_{i,j}^n + F_{i \pm 1,j}^n \right) \]  \hspace{1cm} (6.49)

and

\[ G_{i,j \pm 1/2}^n = \frac{1}{2} \left( G_{i,j}^n + G_{i,j \pm 1}^n \right). \]  \hspace{1cm} (6.50)

In practice, however, stability problems can be observed when using this approach. One possible remedy is using the Lax-Friedrichs method. In essence, its main idea is adding an artificial diffusion for \(q\) which can be done by replacing \(q_n\) with \(q_{i-1,j}^n + q_{i+1,j}^n + q_{i,j-1}^n + q_{i,j+1}^n\). The
corresponding adaption can also be moved into the flux representation, resulting in

\[
F_{i+1/2,j}^n = \frac{1}{2} \left( F_{i,j}^n + F_{i+1,j}^n \right) - \frac{\Delta x}{4\Delta t} \left( q_{i+1,j}^n - q_{i,j}^n \right)
\]

\[
F_{i-1/2,j}^n = \frac{1}{2} \left( F_{i,j}^n + F_{i-1,j}^n \right) - \frac{\Delta x}{4\Delta t} \left( q_{i,j}^n - q_{i-1,j}^n \right)
\]

(6.51)

and

\[
G_{i,j+1/2}^n = \frac{1}{2} \left( G_{i,j}^n + G_{i,j+1}^n \right) - \frac{\Delta y}{4\Delta t} \left( q_{i,j+1}^n - q_{i,j}^n \right)
\]

\[
G_{i,j-1/2}^n = \frac{1}{2} \left( G_{i,j}^n + G_{i,j-1}^n \right) - \frac{\Delta y}{4\Delta t} \left( q_{i,j}^n - q_{i,j-1}^n \right)
\]

(6.52)

The last step is incorporating the source term. Here, we use a basic central flux approximation yielding

\[
S_{h,i\pm1/2,j}^n = \frac{1}{2} \left( S_{h,i,j}^n + S_{h,i\pm1,j}^n \right)
\]

\[
S_{v,i\pm1/2,j}^n = \frac{1}{2} \left( S_{v,i,j}^n + S_{v,i\pm1,j}^n \right)
\]

(6.53)

Finally, we can combine the previously discussed components into

\[
\Delta x \Delta y \ q_{i,j}^{n+1} = \frac{\Delta x \Delta y}{4} \left( q_{i-1,j}^n + q_{i+1,j}^n + q_{i,j-1}^n + q_{i,j+1}^n \right)
\]

\[
+ \frac{\Delta t \Delta y}{2} \left( F_{i+1,j}^n - F_{i-1,j}^n \right)
\]

\[
+ \frac{\Delta t \Delta x}{2} \left( G_{i,j+1}^n - G_{i,j-1}^n \right)
\]

\[
= \frac{\Delta t \Delta y}{2} \left( S_{h,i+1,j}^n - S_{h,i-1,j}^n \right)
\]

\[
+ \frac{\Delta t \Delta x}{2} \left( S_{v,i,j+1}^n - S_{v,i,j-1}^n \right)
\]

(6.54)

to form a variant of the discretized SWE.

### 6.3.3 Solver

Since we employ an explicit time integration scheme, there is no need for a full-fledged solver. Instead, we simply advance time by solving eq. (6.54) for \( q^{n+1} \). For this, a suitable \( \Delta t \) has to be set. We support
using a predefined constant as well as adapting it at run-time. In the case of the latter, we impose the Courant-Friedrichs-Lewy (CFL) condition \[52\] given by

\[
C = \frac{\Delta t}{\Delta x} u + \frac{\Delta t}{\Delta y} v \leq C_{\text{max}} \quad (6.55)
\]

for 2D problems where \( C \) is the Courant number and \( C_{\text{max}} \) represents the chosen upper limit. For explicit methods, such as the one we are using, one usually chooses a value smaller or equal to 1 for said limit. In our concrete application, we find that 0.4 works reasonably well.

**Mapping to ExaSlang**

Next, we examine how our formulation of the SWE can be mapped to ExaSlang. Using layer 4 is most fitting in this case, since our layer 3 concept is not yet aware of time discretizations and how to compose time stepping algorithms from them. As soon as this changes, however, rewriting our application is possible without much additional work. Independent of the layer we start on, the first step is introducing some expressions to represent \( F, G \) and \( S \) from eqs. (6.44) to (6.46), as shown in listing 6.22. At this point, we also add shorthands for \( \Delta x \) and \( \Delta y \) as well as for the time step size \( \Delta t \). The latter is not modeled as a constant expression since it needs to be modifiable in case adaptive time stepping is enabled.

```
Globals {
    // scalar expressions can be used for components
    Expr F0 = hu
    Expr F1 = hu**2 / h + 0.5 * g * h**2
    Expr F2 = hu * hv / h
    Expr G0 = hv
    Expr G1 = hu * hv / h
    Expr G2 = hv**2 / h + 0.5 * g * h**2
    Expr S0 = 0.0
    Expr S1 = -0.5 * g * h * b
    Expr S2 = -0.5 * g * h * b

    // alternatively, vector expressions can be used
    Expr q = [ h; hu; hv ]
}
```

As evident, the components of $q$ have been replaced with the more meaningful $h$, $hu$ and $hv$. Each of them represents a single field access, that is, e.g., $h$ times $u$ is stored as a single quantity $hu$.

At this point, eq. (6.54) can be implemented in ExaSlang as shown in listing 6.23 using the definition of the Centering stencil from listing 6.24.

```exaslang
loop over h {
    solve locally {
        h[next] =>
            dx * dy * ( h[next] - Centering * h )
            + dt * dy / 2 * ( F0@east - F0@west )
            + dt * dx / 2 * ( G0@north - G0@south )
            == S0

        hu[next] =>
            dx * dy * ( hu[next] - Centering * hu )
            + dt * dy / 2 * ( F1@east - F1@west )
            + dt * dx / 2 * ( G1@north - G1@south )
            == dt * dy / 2 * ( S1@east - S1@west )

        hv[next] =>
            dx * dy * ( hv[next] - Centering * hv )
            + dt * dy / 2 * ( F2@east - F2@west )
            + dt * dx / 2 * ( G2@north - G2@south )
            == dt * dy / 2 * ( S2@east - S2@west )
    }
}
```
6.3 Shallow Water Equations

\[ + \text{dt} \times \text{dx} / 2 \times ( \text{G2@north} - \text{G2@south} ) \]
\[ = \text{dt} \times \text{dx} / 2 \times ( \text{S2@north} - \text{S2@south} ) \]

}\}

Listing 6.23: ExaSlang 4 implementation of an explicit time stepping scheme for eq. (6.54).

Stencil Centering {
  east => 0.25
  west => 0.25
  north => 0.25
  south => 0.25
}

Listing 6.24: ExaSlang 4 definition of an averaging stencil considering only direct neighbors.

This implementation allows for a close transcription of the governing (discretized) equations to our DSL. Users are not burdened with thinking about update rules and, yet, the resulting code is as fast as it would be without relying on our local solve mechanic. The main reason behind this is, that our generator can figure out two things. First, the resulting local system matrix is diagonal and can thereby be easily inverted without overhead. Second, there are no accesses to the next slots of the involved fields outside the inner region. This allows for eliminating all conditions in the kernel that would usually be required for handling special cases at or near the boundary. In consequence, the whole loop body comprises of only three assignments.

Alternatively, an explicit update rule can be specified for cases where users require to adapt it. Using the mathematical formulation

\[
q_{i,j}^{n+1} = \frac{1}{4} \left( q_{i-1,j}^n + q_{i+1,j}^n + q_{i,j-1}^n + q_{i,j+1}^n \right) + \frac{\Delta t}{2} \left( \frac{S_{h,i+1,j}^n - S_{h,i-1,j}^n}{\Delta x} + \frac{S_{v,i,j+1}^n - S_{v,i,j-1}^n}{\Delta y} \right) + \frac{\Delta t}{2} \left( \frac{F_{i+1,j}^n - F_{i-1,j}^n}{\Delta x} - \frac{G_{i,j+1}^n - G_{i,j-1}^n}{\Delta y} \right),
\]

we can easily set up a corresponding ExaSlang kernel as shown in listing 6.25. One possible adaptation, which would not have been possible before, is splitting the single loop updating all quantities into three single loops, each responsible for updating a single quantity.
loop over h {
    Expr qNew = Centering * q + dt / 2 * ( 
        ( Sh@east - Sh@west ) / dx 
        + ( Sv@north - Sv@south ) / dy 
        - ( F @east - F @west ) / dx 
        - ( G @north - G @south ) / dy 
    )
    h[next] = dot ( qNew, { 1, 0, 0 }T )
    hu[next] = dot ( qNew, { 0, 1, 0 }T )
    hv[next] = dot ( qNew, { 0, 0, 1 }T )
}

Listing 6.25: ExaSlang 4 counterpart to eq. (6.56).

In both cases, i.e. using our local solve mechanic or providing an explicit update kernel, advancing to the next time step can be done as shown in listing 6.26. If required, the time step size can be adapted in this function as well.

Function AdvanceTimestep@finest {
    advance h
    apply bc to h
    communicate h

    /* analogously for hu and hv */
}

Listing 6.26: ExaSlang 4 implementation of a function advancing to the next time step for the SWE.

6.3.4 Results

We evaluate the performance and scalability of our generated solvers on the state-of-the-art GPU cluster Piz Daint (c.f. section 9.3) sited at the Swiss national supercomputing centre (CSCS) in Lugano, Switzerland. As test case, we take a square domain and perform 1 000 time steps of fixed length. Since all results are also published in [12], we merely summarize them here and highlight the most important ones.

First, we perform a classical weak scaling on up to 2 048 nodes. Figure 6.17 displays the results and, as evident, scalability is satisfactory. To ensure that the global computational domain remains square, we adapt the fragment sizes accordingly: For cases where the total number of
6.3 Shallow Water Equations

![Graph showing weak scaling of generated SWE solvers on Piz Daint.](image)

**Figure 6.17:** Weak scaling of generated SWE solvers on *Piz Daint*. Aspect ratio of fragments is adapted to ensure a global quadratic domain. The largest test case solves for more than $8.2 \cdot 10^{11}$ unknowns.

fragments is a power of four, square fragments of 8 192 by 8 192 cells can be used. Otherwise, we increase the size of each fragment to 16 384 by 8 192. This also justifies the roughly doubled execution times in these cases.

In [12], we also discuss how computation and communication can be overlapped by only slightly adapting the ExaSlang implementation. Doing so, up to 58% of the time spent in communication can be saved. However, the necessary re-ordering of unknown updates in the compute kernels decreases overall performance by a factor of 1.4 to 2. In conclusion, we can show that overlapping works and is effective in general, but, unfortunately, is not attractive for the present combination of application and hardware.

A similar conclusion can be drawn from the examined hybrid parallelization utilizing the CPUs and GPUs of a compute node concurrently. While we can demonstrate that it is easy to implement, and works without further effort, load balancing on *Piz Daint* is extremely challenging. This is due to two factors. One is the way we realize the hybridization, that is we assign blocks and thereby fragments either fully to CPU or to
GPU. This not only requires that multiple MPI threads run on the same node, but also limits the possibilities of distributing the load. Usually, this would not be overly severe. On *Piz Daint*, however, a second factor comes into play, namely that the difference of performance capabilities between the CPU and GPU is around one order of magnitude. Thus, switching to a hybrid parallelization is not effective for us on this particular hardware configuration.

Last but not least, we evaluate the sustained peak performance. Being bound by main memory bandwidth, we apply a simple roofline model [109] to estimate upper bounds for cells updated per time. As reference, we can either take the theoretical bandwidth quoted by the manufacturer, which is 732 GB/s, or the one measured with simple benchmark applications. The latter is around 540 GB/s for copy operations and 560 GB/s for a triad kernel. Based on this, we achieve, on average, about 60% of the theoretical peak performance or around 80% of the measured one.

### 6.4 Contribution

In this chapter, we demonstrated how complex applications from the domain of CFD can be implemented in our DSL in an easily accessible fashion. This is enabled by allowing experts from the field to express concepts familiar to them as well as by supporting concise descriptions of complicated but necessary algorithmic components such as block smoothers. First, we implemented efficient solvers for the Stokes equations. Based on them, we studied the impact of different coarse-grid solver options and featured an in-depth analysis of the error and convergence behavior. Second, we switched from the linear Stokes equations to the non-linear Navier-Stokes equations. Here, different approaches to linearization are possible and, again, all of them are easily implementable in ExaSlang. Examining their performance characteristics allowed choosing the most suitable one for performing scaling experiments on *JUWELS*. They show that our generated solvers exhibit good weak and strong scalability up to the current maximum of 512 nodes. Extending our application to also simulate non-isothermal and non-Newtonian fluids demonstrated the extensibility of our approach. Finally, we regarded solvers for the hyperbolic SWE which we were able to scale up to 2 048 nodes on *Piz Daint*. Given that each node features
6.4 Contribution

3 584 CUDA cores, our simulation ran on more than 7.3 million threads for the largest configuration. My main contributions are:

- Implementation and evaluation of multigrid solvers for the Stokes equations.
- Implementation and evaluation of FAS multigrid solvers for the Navier-Stokes equations.
- Extension of said solvers to include temperature and non-Newtonian properties based on previous work [9].
- Implementation and evaluation of explicit solvers for the SWE. This has been published in [12].

All work has been supervised by Harald Köstler.
This chapter discusses the rich ecosystem around our code generator. It comprises of supporting tools and modules, partly integrated in the framework and partly stand-alone. Moreover, we highlight ways in which our generator is used in others types of research.

7.1 Measuring, Predicting and Tuning Performance

Once an application is implemented in ExaSlang, choosing the right combination of parameters – or configuration options – can be crucial for the ultimately attained performance. This is usually an extremely challenging task, which has the potential to completely overwhelm users. One major aim of the ExaStencils project was thus to automatically find optimal configurations. This requires three components: evaluating or measuring performance, predicting performance and, lastly, optimizing for the best configuration.

7.1.1 Performance Measurement

Most basically, performance measurement capabilities are required. We started work on it in collaboration with Oleg Kravchuk, who did some preparatory work in the scope of his thesis [30]. Based on this, we designed and implemented the current state of the timing module whose main function is to implement – as the name suggests – timing capabilities. The user interface is given by resolving special function on the domain-specific language (DSL) level. Currently, we support the following:
The Ecosystem

- `startTimer` and `stopTimer` to surround a region of an ExaSlang program to be timed; the function takes a single argument which holds the name of the timer; overlapping timers with different identifiers is supported.
- `getTotalFromTimer` and its alias `getTotalTime` to get the total time accumulated on a timer.
- `getMeanFromTimer` and its alias `getMeanTime` to get the mean time per execution of a timer.
- `getLastFromTimer` and its alias `getLastTime` to get the time measured in the last execution of a timer; the last three functions take the name of the timer to be evaluated as the only argument.
- `printAllTimers` to print the mean execution times for each timer used in the current ExaSlang program.
- `printAllTimersToFile` to print the total and mean values of all timers to a single file provided via the settings parameter `timerOutputFile`; in the case of multiple MPI ranks, timer data is reduced by default; this behavior can be overwritten using the `timer_printTimersToFileForEachRank` knowledge flag.

The function calls are then mapped to corresponding intermediate representation (IR) nodes. Of course, directly injecting these nodes without setting up the calls first is also possible. The nodes abstracting the timing behavior are finally mapped to one of multiple back ends, depending on the target system. Currently, we support

- the `std::chrono` library, which is also the default,
- the `QueryPerformanceCounter (QPC)` API,
- the less precise `gettimeofday` and `clock` functions on Linux and Windows systems respectively,
- the MPI timer interface, and
- directly accessing the time stamp counter (TSC) via the `__rdtsc` intrinsic function on Linux and Windows systems.

In addition to timer measurements, we also support emitting what we call characteristics, an arbitrary tuple of an identifier and one or more values. In ExaSlang, this is once again realized through special functions. Examples are given in listing 7.1 for single and multiple values. In contrast to simply printing to the command line, characteristics are emitted to a more easily processable comma-separated values (CSV) file which can be set using the `characteristicsFile` settings parameter.
7.1 Measuring, Predicting and Tuning Performance

```java
// log a single value
logCharacteristics ( "total_num_its", numIts )

// log multiple values
logCharacteristics ( "error_values", err_u, err_v, err_p )
```

Listing 7.1: ExaSlang 4 example of logging characteristics.

### 7.1.2 Performance Prediction

Measuring performance on target platforms usually yields very accurate values. There are cases, however, where predicting is preferred over measuring. This can be due to the target platform not existing (yet), not being accessible or being expensive to use. It is also beneficial if a high number of different configurations need to be evaluated, especially if the time required for generating and compiling overshadows the actual execution time of the application. We address this challenge on multiple levels.

**Analytical Performance Models**

First, we enhance our code generation framework with analytical performance models to be applied automatically in our IR. This works as follows: First, we scan for functions defined in the current abstract syntax tree (AST) and build up a collection containing them. At this stage, the execution time for each function is undetermined. Next, we scan for functions that do not contain calls to other functions without performance estimates. The cost of these functions can now be calculated by summing up the costs of each node inside. We discuss currently supported node types in the next paragraph. Once the cost of a function is known, other functions calling them internally can be estimated. These steps are repeated until all functions in the initial collection have been handled or if no further resolution is possible. The latter case can occur when either function calls to an external function are present or when there are infinite recursions in the program to be analyzed.

Inside functions, we specialize for the following node types: Function call statements are assigned the estimated execution time of the function called. For fixed length loops, the cost of one iteration is simply multiplied with the number of iterations. For kernels represented
by IR_Loop0verDimensions nodes, we apply an optimistic roofline model [109]. To do so, we scan for field accesses while distinguishing slotted accesses as well as read and write accesses. Being an optimistic model, we assume that all data associated with one slot of one field needs to be loaded only once and is then available in the cache throughout the remaining kernel iterations. Additionally, we count the number of arithmetic operations performed. Being optimistic once again, perfect vectorizability is assumed as well as that additions and multiplications can be fused perfectly. The resulting number of required operations as well as the number of loads and stores can then be combined with the hardware characteristics to end up with an execution time estimation. At this point it is important to factor in the employed configuration for OpenMP and MPI parallelization to use the correct hardware characteristics. For instance, we assume that the main memory bandwidth per socket is shared across all (OpenMP and MPI) threads operating on said socket. Likewise, oversubscribing cores can reduce the effectively possible operations per cycle per thread. Other AST node types are currently not modeled and simply ignored, i.e. assumed to have zero cost. Consequently, capturing the impact of other program parts such as, e.g., MPI communication is ongoing work.

All predictions are injected into the generated code as comments to be reviewed by users. This includes not only the final execution time estimate, but also the ratio of memory and arithmetic operations. Additionally, all predictions can be printed to a file if required for further analysis via external tools. Its filename can be controlled using the performanceEstimateOutputFile settings parameter.

**Convergence Prediction**

Analytical performance models have the power to (more or less) accurately predict the execution time for single solver iterations. They can, however, not factor in the number of iterations required since it is usually not known a-priori. The ExaStencils project planned on tackling this issue by relying on an automated mathematical analysis of the employed (discretized) operators and solvers. More precisely, local Fourier analysis (LFA) techniques were to be used to predict convergence behavior and, thereby, number of solver iterations required to meet given criteria. Hannah Rittich brought this vision into reality by implementing
7.1 Measuring, Predicting and Tuning Performance

the LFA Lab\textsuperscript{1} [91, 45]. While the software is up and running, an interface to our code generation framework is currently not operational.

**Performance Influence Models**

The prediction techniques detailed up to this point can be seen as white-box approaches since they require detailed knowledge of the application to be tuned. A contrary black-box approach was examined by Alexander Grebahn in the scope of his dissertation: Given only a set of knowledge parameters to be optimized, and their respective value ranges, a performance model is to be set up automatically. This is realized by modeling generated programs as parts of a software product line (SPL). The underlying configuration space is then sampled and a multidimensional higher-order function fitted to the obtained data. Restrictions imposed on the configuration space, as introduced in section 3.5.3, have been added to the input in later versions. The main benefits of the approach are twofold. First, once the model is set up, it is able to predict all possible configurations without the need for further code generation, compilation or execution. Second, users and domain experts can gain inside into how certain options, or combinations of them, influence performance.

The general applicability of this approach for our domain has been demonstrated [3, 2, 64]. However, for larger configuration spaces, which we typically deal with, a comparably large number of measurements is required incurring a heavy overhead. Moreover, there are some issues with the attained prediction accuracy, especially for corner cases.

### 7.1.3 Parameter Optimization

Once it is possible to measure or predict the performance of a given configuration, the next step is finding an optimally performing one. The core approach of the ExaStencils project in this respect is using the SPL approach to set up performance models to which an optimizer can be applied. There are, however, some drawbacks. First, the optimization is executed as a separate, decoupled step. This infers that the sampling used to gather the data fed into the performance model is not adapted to regions of interest, i.e. the regions around optimal configurations. It also means, that a separate optimizer is required. This (external) tool

\textsuperscript{1} hritich.github.io/lfa-lab
also needs to be able to deal with the restrictions imposed on the configuration space, which is a problem for many existing implementations. Second, a reasonably accurate performance model is required to find configurations that perform optimal or at least near-optimal. As mentioned in the last section, this requires a large number of configurations to be measured, i.e. generated, compiled and executed on the target platform. Thus, at least in our opinion, we deem this approach to be insufficient for our domain. One possible solution could be switching to a gray-box approach, e.g. by setting up performance influence models for smaller parts of an application, say single kernels and communication phases.

An alternative approach is given by using genetic algorithms (GAs), as examined by Lorenz Haspel in the scope of his thesis [28]. Using a similar input than for the SPL approach, i.e. a set of knowledge parameters to be tuned, their value ranges and existing constraints, finding well-performing solvers is possible. Advantages are the substantially lower number of measurements required and having the option of cutting the optimization short once a sufficiently good configuration has been found. The latter is also possible with the previous approach, but does not influence the main cost factor given by building up the full performance model. One disadvantage is that the performance of the optimization itself is highly dependent on chosen GA parameters and, thus, the optimizer needs to be tuned as well.

In some cases, optimization is also required for aspects that can not be modeled using configuration options. One example for this is given by the solver composition or implementation, which can be seen as an arbitrary AST. Jonas Schmitt is currently focusing on tackling this challenge by employing genetic programming (GP) techniques. In detail, the main aim is to learn solver algorithms for a given problem specification (discretization). Performance of an individual is assessed in a two-stage approach. First, a rough estimation is computed based on a simple roofline model, similar to the approach described before, and an LFA analysis provided by LFA Lab. Promising individuals are then converted into an ExaSlang 3 program, generated, compiled and executed. First results have already been published [22] and show that ExaSlang and our code generator provide a viable back end and evaluation platform.
7.1.4 Discussion

In conclusion, we find that measuring performance characteristics of generated solvers and applications works sufficiently well for our purposes. Predicting the same data is partly implemented, but needs more work for missing aspects such as the cost of MPI communication. Moreover, an autonomously working combination of analytical performance models and LFA analyses is missing. The most work, however, is still to be done in the field of optimization. For us, it is still an open question which approach will be the best fit for our domain and how it can be integrated such that users can use it without too much added effort.

7.2 Visualization of Simulation Results

Another important aspect of scientific computing is post-processing and visualizing the obtained simulation results. This is traditionally done in separate steps after running the actual simulation using external tools. We support a similar approach by allowing users to print fields to file from ExaSlang. Listing 7.2 illustrates this and, additionally, how to construct filenames on the fly to emit data series.

```
Var filename_vel : String
buildString ( filename_vel,
    "velocity_", curTime, ".txt" )
printField ( filename_vel, vel@finest )
```

Listing 7.2: ExaSlang 4 example of setting up consecutive filenames and using them when printing a field.

Post-processing can now be done externally, just as visualization using tools such as, e.g., ParaView\(^2\). Of course, performing certain data evaluation operations is also possible directly in ExaSlang. Calculating the magnitude of the cell-centered velocity from staggered velocity components, for instance, can be added as a simple kernel as shown in listing 7.3.

---

\(^2\) paraview.org
7 Ecosystem

Listing 7.3: ExaSlang 4 example of a simple post-processing kernel evaluating the velocity magnitude at cell centers.

While using external visualization toolkits such as ParaView is often considered state-of-the-art, rendering the results while the simulation is running is in our opinion more beneficial. These so-called in-situ visualizations not only allow direct feedback about the state of the simulation, but also allow for computational steering. We successfully demonstrated these benefits in early works for particle and fluid simulations [14, 10]. In the scope of our code generation framework, there are multiple options.

Most straight-forward, a specialized visualization can be implemented from scratch in C++ after generating the application to be visualized. This approach has been examined by Damian Swientek in the scope of his thesis [34], where he rendered the results of a generated shallow water equations (SWE) solver, as presented in section 6.3, at run-time. Two different implementations have been set up: one using a traditional rasterization approach and one performing ray-tracing. As figures 7.1 and 7.2 shows, the resulting images are generally of very high quality and the visualization is able to depict effects such as reflections and distortions. However, re-using the renderer for similar applications is cumbersome as it requires manual adaptation of the generated code.

This issue can be avoided by generating interfaces for a rendering back end, as evaluated by Markus Obereisenbuchner in his thesis [32]. In his case, a custom back end has been implemented based on BGFX\(^3\) aimed at being flexible enough to support a wide range of ExaSlang applications. It can be delivered as a library to be linked in the compilation stage of the generated application, or as source files to be compiled together with the generated application. Computational steering capabilities are available as well in the form of so-called tunable parameters. As listing 7.4 illustrates, marking global variables as tunable is straightforward, as is adding rendering views and fields to be visualized. Figure 7.3 shows an example visualization and the corresponding steering dialog (on the right) where tunables can be set as required.

\(^3\) github.com/bkaradzic/bgfx
7.2 Visualization of Simulation Results

![Example of an in-situ visualization for our SWE solver presented in section 6.3. Deeper water is shaded darker and the vertical wall (added for effect) is reflected on the water surface [34].](image)

**Figure 7.1:** Example of an in-situ visualization for our SWE solver presented in section 6.3. Deeper water is shaded darker and the vertical wall (added for effect) is reflected on the water surface [34].

```camera
1 // add a rendering view
2 Var viewId : Int =
3   renderer_addView ( 0.0, 0.0, 1.0, 1.0 )
4 PrepareVis ( viewId, h@finest, h@finest, 100, 100 )
5
6 // add tunable - gravity between -20 and 20
7 renderer_registerTunable ( g, "g", -20.0, 20.0 )
```

**Listing 7.4:** ExaSlang 4 example of using the BGFX based visualization interface.

This approach grants a high degree of flexibility which, however, comes at the price of another piece of software that needs to be maintained. One alternative is using a ready-to-use visualization toolkit with in-situ capabilities. VisIt\(^4\), which has a similar scope than ParaView, has been successfully used for this in the thesis of Richard Angersbach [23]. It is the spiritual successor of the thesis of Achim Däubler [25], who demonstrated general feasibility of this approach by coupling VisIt with the VisIt with the

\(^4\) visit.llnl.gov
Figure 7.2: Example of an in-situ visualization using ray tracing for our SWE solver presented in section 6.3. A plane cutting through the water surface has been added to enhance visual effects. Reflections at the circular wave are distorted [34].

Figure 7.3: Example visualization using a custom back end including computational steering capabilities [32].
7.2 Visualization of Simulation Results

Figure 7.4: Example visualization of generated solvers using VisIt [23].

waLBerla\textsuperscript{5} multiphysics framework. As before, necessary interfaces can be generated automatically. The biggest advantage is that users already proficient with VisIt can continue to work in a familiar environment. Figure 7.4 shows one example output.

All of the previous approaches share the potential issue that our code generator is not stand-alone anymore, but depends on external software to be delivered alongside or to be installed beforehand. We alleviate this by providing another, less powerful, visualization based on CImg\textsuperscript{6}, which only comprises of a single C++ header file. It can, if required, be directly emitted by our framework alongside the generated application. Listing 7.5 illustrates the lightweight ExaSlang interface and figure 7.5 one example output.

\begin{footnotesize}
\begin{itemize}
  \item\textsuperscript{5} walberla.net
  \item\textsuperscript{6} cimg.eu
\end{itemize}
\end{footnotesize}
Figure 7.5: Example visualization using CImg for the Navier-Stokes test case from section 6.2.

Listing 7.5: ExaSlang 4 example of using our CImg back end.

```cpp
// write field to file
writeMappedImage ( vel@finest, "velocity.png" )

// display field at run-time
showMappedImage ( vel@finest,
```

Discussion

Comparing the approaches presented so far, we find the following:

The solution based on CImg is the easiest to use and, since the dependencies comprise of only a single header file, it is also the one that can be integrated into code generators the quickest. On the down-side, it only supports 2D grids, which additionally have to be uniform to yield correct visualizations. Multiple fragments are supported, independent of them being in a single block or scattered across multiple ones, but get displayed as separate images.
Coupling with VisIt is the most flexible solution. It readily supports 2D and 3D grids that may be staggered and non-uniform, as well as multiple fragments and blocks. Frequently used visualization modules are available to users. They can, however, not be extended or adapted without considerable effort.

Using a custom back end, like the one presented based on BGFX, is exactly the opposite in this respect. Adding different kinds of visualizations is possible without incurring too much overhead, but also required more frequently. In our version, non-uniform grids are supported, as well as multiple fragments as long as they are within one block.

Writing a custom visualization for already generated applications yields the most visually pleasing results since advanced rendering techniques can easily be integrated. Adaptation to other generated applications, even if the underlying problem to be solved remains the same, is challenging, as is adding MPI support.

7.3 Running Multiple Configurations

When developing and testing ExaSlang programs, users frequently want to generate, compile and execute multiple different variants of the same application. We facilitate this process by providing a tool to do just that – the ConfigRunner. Its input are the usual knowledge, settings and platform files, and one additional file specifying the configuration space. This configs file contains three blocks as illustrated in listings 7.6 and 7.7. First, variabilities are specified. The left-hand side of the assignments may be any parameter from the knowledge, settings and platform collections. The right-hand side is either a list of possible values for the chosen parameter or an inline Scala code snippet evaluating to a sequence of values. Using the Cartesian product of all provided value ranges constructs the initial configuration space to be explored. It can, however, be filtered to eliminate unwanted combinations using the constraints block. Each constraint is again provided as a Scala code snippet, which, however, has to evaluate to a boolean value this time. If one of the constraints evaluates to false for a given parameter combination, it is removed from the configuration space. Before running the actual configurations, it is possible to adapt other knowledge, settings and platform parameters. This is done in the derived parameters block.
Parameters can be set as before (c.f. section 3.5). Moreover, using Scala code is available here as well.

Listing 7.6 shows an example for trying out different solver configurations. The number of pre- and post-smoothing steps is chosen between 0 and 6 each, with the added constraint of their sum having to be between 2 and 6. Additionally, two different coarse-grid solver (CGS) are evaluated.

Variables

```scala
Variabilities {
  solver_smoother_numPre = { 0, 1, 2, 3, 4, 5, 6 }
  solver_smoother_numPost = '(0 to 6)'
  solver_cgs = { "CG", "BiCGStab" }
}

Constraints {
  solver_smoother_numPre + solver_smoother_numPost >= 2
  && solver_smoother_numPre + solver_smoother_numPost <= 6
}

DerivedParameters {
  configName = 
    "v${solver_smoother_numPre}-${solver_smoother_numPost}-${solver_cgs}"
}
```

Listing 7.6: Configs file example for testing different combinations of pre- and post-smoothing, as well as two coarse-grid solvers.

Listing 7.7 illustrates how a classical weak scaling can be performed using our tool. First, the number of MPI threads to be used is set up as powers of two ranging from 4 to 128. As we want to execute 4 ranks per node, we derive the required number of nodes accordingly. The domain is then partitioned according to the total number of fragments: If the total number is a power of four, the number of fragments in each dimension can be chosen to be equal and fragments may be square. Otherwise, the domain can not be partitioned using square fragments. We resolve this by scaling all fragments by a factor of two in the x-dimension, which allows using twice as many fragments in the y-dimension as in the x-dimension.
7.4 The Meta Tool

In our code generation framework, node types need to be duplicated for each layer they are available on, as detailed in section 4.1. It is not surprising that at least in some cases these layer-dependent implementations share large portions of their code. To facilitate the implementation and maintenance of such classes, and the files they reside in, we implement the Meta tool. Its main job can be described as generating the generator. A typical implementation is shown in listing 7.8 for the case of a null statement, i.e. the empty statement. As evident, the generation

```scala
Variabilities {
    mpi_numThreads = ~'(2 to 7).map(1 << _)'~
    // equivalent to = { 4, 8, 16, 32, 64, 128 }
}

DerivedParameters {
    configName = "2D_mpi-$mpi_numThreads$"
    timerOutputFile = "./timings/$configName$.csv"
    hw_numNodes = ~'mpi_numThreads / 4'~
    domain_numBlocks = ~'mpi_numThreads'~
    domain_fragmentLength_x =
        ~'(if ((scala.math.log10(mpi_numThreads)
            / scala.math.log10(2.0)).round.toInt
            % 2 != 0) 2 else 1)'~
    domain_fragmentLength_y = 1
    domain_rect_numBlocks_x =
        ~'scala.math.pow(mpi_numThreads
            / domain_fragmentLength_x,
            1.0 / 2.0).round.toInt'~
    domain_rect_numBlocks_y =
        ~'scala.math.pow(mpi_numThreads
            * domain_fragmentLength_x,
            1.0 / 2.0).round.toInt'~
}

Listing 7.7: Configs file example for performing a weak scaling of a 2D application.
```
of the Scala classes and files is built on straightforward string manipulations. Layer-specific parts can be expressed using strings for the layer names in lower case (\texttt{LAYER\_LC}) and upper case (\texttt{LAYER\_UC}), as well as the names of the subsequent layers \texttt{NEXT\_LC} and \texttt{NEXT\_UC}. More complex changes can be realized via simple branching, as shown for the progress functionality which is available on all layers but the IR.

```scala
object ME_NullStatement extends Generatable {
  override def validLayers() =
  ListBuffer(L1, L2, L3, L4, IR)

  override def filenameForLayer(layer : Layer) =
  s"./Compiler/src/exastencils/base/" +
  s"\texttt{LAYER\_LC}/\texttt{LAYER\_UC}\_NullStatement.scala"

  override def generateForLayer(layer : Layer) = {
    val printer = new Printer
    printer <<<
    """package exastencils.base.\texttt{LAYER\_LC}"
    printer <<< """
    if (L1 == layer || L2 == layer
     || L3 == layer || L4 == layer) {
      printer <<< """import exastencils.base."
      printer <<< """ProgressLocation"
      printer <<< 
      """import exastencils.base.\texttt{NEXT\_LC}\_"""
    }
    printer <<<
    """import exastencils.prettyprinting._"
    printer <<< """
    printer <<<
    """case object \texttt{LAYER\_UC}\_NullStatement"
    printer <<< 
    """extends \texttt{LAYER\_UC}\_Statement {"
    printer <<<
    """override def prettyprint(out :"
    printer <<< """
    if (L1 == layer || L2 == layer
     || L3 == layer || L4 == layer) {
      printer <<<
      """override def progress = """
      printer <<< """ProgressLocation(\texttt{NEXT\_UC}\_"
      printer <<< """NullStatement)"
    }
```
7.5 Contribution

In this chapter, we presented the rich ecosystem around our core code generator. Tasks that need to be performed frequently, such as profiling...
generated applications, can now be automated easily. Based on these measurements, or on different types of performance predictions, parameters can be tuned. Similarly, our ConfigRunner tool allows effortlessly evaluating multiple configurations. We demonstrated how result data obtained from ExaSlang programs can be post-processed directly in our DSL and how it can be visualized. For the latter, different approaches are available. Last but not least, we make extending the code generator itself more comfortable by providing our Meta tool.

My main contributions are:

- Implementation and integration of performance measurement capabilities including a light-weight DSL interface.
- Implementation and integration of an automatically applied analytical performance model.
- Comparing different visualization approaches.
- Conceptualization and implementation of the ConfigRunner tool.
- Conceptualization and implementation of the Meta tool.

All work has been supervised by Harald Köstler.
8 Related Work

Our ultimate aim is researching how to enable automatic generation of massively parallel high-performance geometric multigrid solvers for structured grid applications. There are a number of approaches which, at least partly, follow the same goal. This section is intended to review them and provide an overview over the state of the art. Please note, that it is by no means complete as it only focuses on the most relevant projects and publications.

8.1 Frameworks for High Performance Computing

As already discussed in chapter 1, one of the most established approaches to tackling this challenge is utilizing large-scale frameworks. Over the last decades, a sizable number of them emerged.

One of the most popular ones is PETSc [40], short for portable extensible toolkit for scientific computation. Its focus lies in parallel solvers for linear and nonlinear equations, as well as suitable preconditioners. It supports execution on NVidia GPUs and includes automatic profiling capabilities.

HYPRE [58], short for high performance preconditioners, follows a similar path. Its main aim is providing scalable linear solvers and multigrid methods. In this context, it operates on structured and unstructured meshes alike. To our knowledge, there is no stable GPU back end at the time of writing this work.

Trilinos [68] is rather a collection of different modules – called packages – than a single framework. Each package is self-contained and managed and maintained by a distinct development team. Its name is derived from the Greek word ”Trilinos” which can be loosely translated
8 Related Work

to *string of pearls*. Trilinos also includes support for GPUs by coupling with Kokkos, which will be discussed in the next section.

Deal.II [41], the successor to the *differential equations analysis library*, is more focused on finite element codes, mainly on adaptive meshes. Within one node it relies on Intel threading building blocks (TBB) to implement a hierarchical parallelization. Moreover, it can interface with Trilinos and PETCs.

DUNE [43], short for the *distributed and unified numerics environment*, provides tools for solving partial differential equations (PDEs). It focuses on grid-based methods and provides discretization modules mainly for finite element method (FEM) applications. Nevertheless, support for finite difference method (FDM) and finite volume method (FVM) exists as well. More recently, the EXA-DUNE project [42] was aimed at preparing the DUNE framework for upcoming exascale platforms.

UG4 [105], short for *unstructured grids 4*, is a simulation framework aimed at solving PDEs on unstructured hybrid grids. It incorporates an interface to the LUA scripting language which can be used to control the whole program flow.

Chombo [36] is aimed at solving PDEs on adaptively refined rectangular block-structured grids. It is focused on FDM and FVM applications and supports complex boundaries using an embedded boundary approach. Its name stems from the Swahili word ”Chombo” which means *tool or container*.

8.2 Language Support for Parallel Applications

Many of the presented frameworks are challenged by supporting multiple levels of parallelism. All of them rely on MPI for the distributed memory parallelization. The best approach for parallelization within one node is, however, less clear. In recent years, many extensions to support GPUs have been added or at least been planned. This is mainly done via providing a CUDA back end. On the CPU level, incorporating OpenMP or Intel TBB seems suitable. Of course, vectorization has to be added in many use cases to attain ultimate performance.

It is evident, that this leads to a difficult challenge for developers implementing parallel applications and – in extension – high performance computing (HPC) frameworks. Apart from domain-specific languages
(DSLs) and code generation technologies, which will be discussed later, there are mainly two approaches that try to tackle this:

First, new programming languages are created that provide abstractions for parallel data structures as well as parallel execution. One examples for this is Chapel [48] which is now managed by Cray. For new applications and simple prototypes these languages are a viable alternative. In the case of legacy codes and frameworks, however, often a complete reimplementation would be required, which is infeasible most of the time.

Relying on language extensions might be a better approach since necessary changes are limited to a (hopefully) small subset of the framework’s code base. This can either be done by explicitly introducing abstractions for parallelization or by hiding it to a certain extend and thereby making it implicit. Examples for the former are, among others, CHARM++ [71] and Cilk [60], which was later acquired by Intel but is now marked as deprecated. One reoccurring concept in the scope of the latter option is given by partitioned global address space (PGAS). Here, data structures can be used as in the serial case, but under the hood they are distributed (semi-)automatically. Synchronization of data is then done automatically upon accessing a datum outside the local partition of the memory space. Examples include UPC [51] and, more recently, Dash [61]. The utilized distribution is traditionally determined statically which can lead to issues with load balancing, especially when trying to incorporate GPUs.

One alternative trying to mitigate this issue is given by task-based approaches. Here, the work to be done is defined as small-grained tasks that are scheduled by a given runtime. Examples for this approach are OmpSs [56] and StarPU [39].

If an MPI parallelization is already in place, such as it is the case with all of the discussed frameworks, looking at a solution specialized towards parallelization within one MPI rank might be worthwhile. One example for such an approach is Kokkos [57]. It provides abstractions for parallel data structures and parallel execution of kernels. CPUs as well as GPUs are supported via the provided CUDA, pthreads and OpenMP back ends. Data layouts can be adapted according to the target hardware. Kokkos understands itself as a templated C++ library. It can, however, also be viewed as an embedded DSL for parallel execution. More details on other DSLs are discussed later.
8.3 Auto-Tuning and Code Generation

The main aim of auto-tuning is finding a combination of parameters – usually called a configuration – that describes an implementation with ideal characteristics, usually a small execution time. This field of research is often tightly coupled with code generation approaches since the latter allows to quickly set up a high number of alternative implementations.

ATLAS [107] is one of the earliest approaches here. Its focus is providing automatically tuned linear algebra kernels that can be used analogous to BLAS routines. In a setup phase, multiple variants are generated and timed to determine the best-performing ones. Micro-benchmarks are used in this step to determine machine characteristics and, thereby, reduce the search space of potentially well performing variants.

FFTW [59], short for the fastest Fourier transform in the west, follows an approach similar to ATLAS, but for the domain of fast Fourier transforms. It also generates a multitude of implementation variants and choses optimal ones through tuning. Additionally, automatic vectorization of the emitted C code is available.

SPIRAL [86] focuses on the generation of kernels from abstract, mathematical representations. Similar to FFTW, it ultimately maps to C code and supports automatic vectorization. Its domain of general linear transforms is, however, much broader. This flexibility stems from accepting general input written in its own signal processing language. Alongside this specification, certain breakdown rules may be specified at the DSL level to instruct the compiler how to process certain expressions. To find optimal configurations, an automated empirical search can be performed. More recently, first steps towards implementing efficient multigrid solvers have been undertaken [44] and a partial reimplementation of SPIRAL in Scala has been examined under the name SPIRALS [83].

PetaBricks [38] provides an implicitly parallel language focused on describing algorithmic choices. Developers can implement different algorithmic variants solving the same problem and, through an autotuner, the best is chosen. The same tuner can also be used to determine optimal parameters which can be marked accordingly in the DSL using a special keyword. The DSL code is then fed into a source-to-source compiler ultimately emitting C++ code. In this step, an automatic parallelization and data distribution can be performed based on a task-based approach. It is then carried out by a runtime system that
includes a parallel work stealing dynamic scheduler. Apart from finding the best algorithmic variant through tuning, PetaBricks is also able to check the different implementations for consistency. It does neither support GPUs nor distributed memory parallelization. An application to multigrid solvers has been shown in [49] where the cycle shape was optimized.

8.4 General Code Generation

Optimizing implementations for a given hardware target can be tedious and error-prone. Some approaches aim at using code generation techniques to facilitate this process without being limited to a specific application domain.

SEJITS [47] advocates the approach of selective embedded just-in-time (JIT) specialization. As a prototype, stencil languages embedded in Ruby and Python have been implemented. They are able to generate C++ and CUDA code, where the former can be parallelized with OpenMP.

Mint [101] focuses on the transformation of general purpose code into GPU kernels. It is based on a source-to-source approach that takes annotated C++ code and emits CUDA code. Being based on ROSE, it is able to analyze and process general purpose code. That also allows implementing stencil computations, but there is no actual (language) support for stencil constructs.

Terra [54] focuses on facilitating optimizations employed in code generation phases. It is embedded in Lua, a popular scripting language, and can, through the use of LLVM, be JIT compiled to increase performance. GPUs are not available as targets at this point. An interesting feature is that domain-specific optimizations can be implemented in the same scope, i.e. directly in Lua. The build-in support for calling legacy C functions aims to facilitate coupling with other codes.

8.5 Code Generation for PDE Applications

In the domain of discretizing and solving PDEs, various code generation approaches have been researched in the past.

Liszt [55] focuses on solvers operating on unstructured meshes. Its language is embedded in Scala. Ultimately, C++ and CUDA code is generated where the former is additionally parallelized using pthreads.
In both cases, an automatic coloring may be added to prevent race-conditions. Partitioning of the grid is done using ParMETIS\(^1\) and, based on it, halo exchange functions are set up automatically. To our knowledge, multigrid solvers based on this technology have not been published and it is unclear if they can even be expressed in Liszt.

OP2 [80] focuses on computations performed on unstructured meshes as well. Based on ROSE, it implements a source-to-source compiler emitting C++ or FORTRAN. Shared and distributed memory parallelization using OpenMP and MPI are available, as is a CUDA back end to target GPUs. However, certain shortcomings have been reported such as the lack of AVX capabilities, the inability to utilize GPUDirect or MPI and OpenMP in tandem. For distributing computational data across multiple nodes, graph partitioning algorithms are used and halo exchanges are set up automatically based on their results. Meshes within one MPI rank are further broken up into mini-partitions which are automatically colored such that they can be executed in parallel using OpenMP or CUDA.

FEniCS [74] is focused on FEM applications for solving PDEs. Its main form of input is the unified form language (UFL) [37] which allows the specification of weak formulations of PDEs. Being embedded in Python allows its input to be fed into the FEniCS form compiler (FFC) which emits unified form-assembly code (UFC). UFC can then be mapped to optimized C++ kernel functions.

DOLFIN [75], short for \textit{dynamic object oriented library for finite element computation}, is a part of FEniCS and implements core components and interfaces to external software components. As such, it can be used to assemble linear systems from UFC specifications and solve them using back ends to external tools such as PETSc or Trilinos.

Firedrake [88] goes beyond the scope of FEniCS by introducing new abstractions and allowing users to implement operations that are not in the scope of pure variational formulations. An example for the latter is given by flux limiters. Still using UFL input, a new layer is introduced between the (local) discretization of mathematical operators and their execution over a given mesh. This layer is given by PyOP2 [89], an embedded DSL that implements the concepts of OP2 in Python. In contrast to OP2, it lacks a static analysis and source-to-source compilation capabilities, but instead it features JIT kernel compilation and a run-time

\(^1\)glaros.dtc.umn.edu/gkhome/views/metis
scheduler. It provides CUDA and OpenCL back ends or internally calls PETSc4py\(^2\), a Python interface to PETC\(s\), or Cusp\(^3\), a library for sparse linear algebra and graph computations based on Thrust. OpenMP and MPI capabilities have been reported to be under development. Domain-specific abstract syntax tree (AST) optimizations and automatic vectorization are available via COFFEE \([76]\), short for a compiler for fast expression evaluation. For this toolchain to work, the FFC is adapted to emit either a C++ kernel or an unoptimized AST which can be further processed. The required functionality is implemented in the two stage form compiler (TSFC) \([70]\).

**Stencil DSLs**

Last but not least, a large number of DSLs specifically targeting stencil applications have been published.

One of the earlier works \([53]\) focuses on the optimization of stencil computations for CPUs and GPUs. To this end, mainly blocking techniques and other loop transformations are applied. A four-level decomposition strategy of data is proposed to be able to efficiently map to the characteristics of a given hardware. In order to find optimal configurations, auto-tuning is employed. Ultimately, C code with added intrinsics is generated. An MPI extension is not available, however. A similar approach for the domain of lattice Boltzmann method (LBM) has been examined \([108]\).

SBLOCK \([46]\) focuses on stencil computations performed on block-structured grids. It consists of a runtime and a source-to-source compiler based on the Cheetah templating system\(^4\) which is usually used to insert dynamic content into predefined HTML templates for web pages. Automatic vectorization is available, but requires pycparser\(^5\) and constructing an AST. Parallelization can be done for GPUs and distributed memory machines. Multigrid implementations are possible. However, restriction and interpolation operations can not be expressed directly but have to be encoded as sparse matrix-vector multiplications (SpMVs). Boundary handling is not set up automatically either and has to be implemented manually by users.

---

\(^2\) bitbucket.org/petsc/petsc4py  
\(^3\) cusplibrary.github.io  
\(^4\) cheetahtemplate.org  
\(^5\) github.com/eliben/pycparser
PATUS [50], short for *parallel auto-tuned stencils*, focuses on optimizing and tuning stencil applications. Its DSL is split into two parts: First, the actual stencil computation can be encoded. Second, optimization strategies, such as blocking, can be expressed. Adding a specific keyword marks parameters for auto-tuning. Both parts of the DSL are parsed separately and then processed by a generator written in Java to ultimately emit code parallelized with OpenMP or CUDA. Internally, the Cetus framework\(^6\) is used and an interface to Maxima\(^7\) is available for symbolic simplifications. On the DSL level, vector fields are supported. For boundary conditions, only Dirichlet ones are available.

Pochoir [98] focuses on generating cache-oblivious implementations. Its DSL is embedded in C++ and being translated to Cilk by the Pochoir compiler which is written in Haskell. For debugging purposes, direct execution in C++, i.e. without translation, is possible. Cache-oblivious algorithms usually implement spatial and/or temporal blocking. As reported, this only pays off when many iterations of the same kernel are performed subsequently and without any other operations (such as communication phases) in-between. Specialized boundary handling is, however, possible. Neither a distributed memory parallelization nor a GPU back end are available in Pochoir.

Physis [77] is a general stencil DSL with focus on automatic parallelization. It is embedded in C and translated with a source-to-source compiler build on ROSE. Specialized towards regular multidimensional Cartesian grids, automatic parallelization using MPI and CUDA is available, as is automatic overlapping.

SDSLc [90] focuses on higher-order stencils. Target architectures include CPUs, GPUs and FPGAs, but limited to single nodes since a distributed memory parallelization is not available. Automatic vectorization is available as is polyhedral optimization of the generated code though external tools. Data layout optimizations are available as well, although they are somewhat limited. Interfaces to legacy Matlab code can be generated.

MODESTO [65], short for *model driven stencil optimization*, focuses mainly on stencil pipelines similar to some of the image processing DSLs presented next. To find optimal configurations, it introduces stencil algebra. Based on this, the automatic tuning of stencil programs is

\(^{6}\) engineering.purdue.edu/Cetus
\(^{7}\) maxima.sourceforge.net
formulated as a mathematical optimization problem which can then be solved using dynamic programming techniques. At this stage, performance models are used instead of actual run times to accelerate the tuning process. They are spiritually similar to the roofline model [109], although more sophisticated. MODESTO has been successfully applied to automatically tune programs written with STELLA which will be discussed later.

8.5.1 Image Processing DSLs

There are some approaches that can be attributed to an even narrower domain compared to the ones discussed previously. Image processing is one of these domains.

The Halide language and compiler [87] is one of the most influential ones. Its main focus lies in handling stencil pipelines, i.e. a succession of kernels that can be expressed by stencils. Automatic optimizations such as (overlapped) tiling and other loop transformations as well as vectorization are available. Optimal configurations can be found automatically by using the integrated auto-tuner which performs an offline stochastic search based on a genetic algorithm. Halide is mainly focused on 2D application and supports image pyramids which could be used to implement multigrid methods. As back-end, LLVM is used to also generate code for GPUs, but distributed memory parallelization is not supported. PolyMage [81] follows an approach similar to the one of Halide.

HIPACC [78], short for heterogeneous image processing acceleration, is a DSL embedded in C++ which relies on a source-to-source compiler built on top of Clang. It features optimizations such as loop unrolling and adapting the memory layout. As Halide, it supports image pyramids and a successful multigrid implementation has already been published [79]. Other than C++ code, it can also generate CUDA, OpenCL and RenderScript code to target a wide range of hardware platforms such as GPUs, FPGAs and other embedded devices. Distributed memory parallelization, however, is not supported and the main focus lies in 2D applications.

Chipotle [95] is an approach based on LISP and heavily inspired by HIPACC. It allows a more flexible syntax compared to the previous C++ approach while still being able to generate CUDA code for GPUs and variants with AVX intrinsics for CPUs. This results in achieving perfor-
mance comparable to HIPACC. Limitations, however, are the same as well.

RIPL [96], short for the *Rathlin image processing language*, is an external DSL. In contrast to the other approaches presented, it focuses on FPGAs by generating Verilog code. Another key difference is, that it is intended for applications handling multiple frames, i.e. that apply the same kernels to a succession of input images. In this context, latency hiding optimizations are crucial. They can be performed automatically after constructing a data flow graph.

Orion [54] focuses is 2D stencil computations performed on images. It is build on the previously discussed Terra project and designed after Halide. Orion mainly serves as a demonstration for the usefulness of Terra. As such, its functionality is quite limited, e.g. only zero boundary conditions are supported.

### 8.5.2 Climate and Weather DSLs

Another domain that is particularly attractive for stencil DSLs is given by climate and weather modeling.

ATMOL [104], short for the *atmospheric modeling language*, is one of the oldest works in this field. Input given in its language are translated to FORTRAN with the help of the code synthesis tool CTADEL [103]. Similar to our approach, it allows declaring fields and PDEs. Moreover, rules for processing derivatives and integrals can be given directly at the DSL level. As such, different abstraction levels are supported, a clear distinction is, however, missing. Implemented options include automated FDM application for derivatives and midpoint quadratures for integrals. Moreover, variables and fields may be declared with annotations of physical units. The framework can then use this information to automatically check for unit consistency. Distributed memory parallelization is available via MPI and the applicability has been demonstrated by implementing the HIRLAM (*high resolution limited area model*) system.

ICON [99] is a more recent project. In its scope, a DSL embedded in FORTRAN is created which is then fed into a source-to-source compiler to again emit FORTRAN code. After encountering multiple difficulties with a first approach based on the ANTLR parser generator, the project’s team decided to switch to building on top of the Rose compiler. Optimizations such as memory layout transformations are available, al-
though in this case limited to reordering index dimensions. An automatic parallelization using OpenMP, OpenACC and CUDA is planned.

STELLA [66], short for *stencil loop language*, is a DSL embedded in C++ focused on FDMs on structured grids. Code generation is realized via C++ template meta-programming based on the Boost MPL library. STELLA has been successfully applied to the COSMO (*consortium for small-scale modeling*) weather forecasting model in a full rewrite. Through this, support for GPUs was added with a CUDA back end, as well as a CPU parallelization using OpenMP and optimizations tailored to the domain at hand. The domain specialization also shows in the assumption that parallel execution is only possible in the horizontal i-j-plane. Along the vertical k-dimension, iterations are processed sequentially. Similarly, halo exchanges are supported, albeit only in two dimensions. Last but not least, overlapping computation and communication can be implemented via asynchronous data exchanges that need to be specified by users.

Grid Tools\(^8\) is planned as the successor to STELLA. Its main aim is generalizing STELLA to other grids, such as staggered ones, potentially also consisting of other primitive types such as triangles and prisms. Apart from a simplified syntax, user-experience is to be enhanced through a Python front-end.

CLAW\(^9\) is a project similar to STELLA, but embedded in FORTRAN. It is based on directives and implements a GPU back end using directives (OpenACC) as well. As such, it uses a source-to-source compilation approach.

### 8.6 Discussion

As already discussed in chapter 1, HPC frameworks are usually too general to efficiently implement domain-specific optimizations for multiple given hardware platforms. Moreover, they are often difficult to use for application scientists. Parallel languages and language extensions can help reducing the complexity of application code but still suffer from not being domain-specific. They can, however, provide viable back ends for code generation technologies. We decided against using such an approach since the long-term support is often not guaranteed. This holds especially true for software developed in an academic

\(^8\) pasc-ch.org/projects/2013-2016/grid-tools
\(^9\) github.com/claw-project/claw-compiler
background as such projects are notorious for being abandoned after some time. In our opinion, stencil DSLs are the most viable solution to the challenges faced. The number of DSLs and stencil frameworks proposed in the past clearly underlines this. One question, however, arises: does the world really need another stencil DSL? Or would it not be more productive to build on any of the already existing solutions? To answer this justified question we start by summarizing the requirements that need to be fulfilled to realize our ultimate goal – the automatic generation of massively parallel geometric multigrid solvers for real-world applications.

First, *multiple grids* must be supported, or at least the same grid with multiple resolutions. Most of the reviewed technologies are restricted to a single grid with multiple fields attached. Some image processing DSLs allow multi-resolution grids through the concept of image pyramids and a few general approaches support multiple grids. In our framework, we go one step further by directly providing DSL level language support for leveled data structures and functions operating on them.

Next, *operators mapping between grids* with different resolution are required. Some of the discussed approaches allow this, although often only through bypassing the language itself, or through unnecessarily complex expressions. We reduce this burden by introducing mapping stencils, as shown in section 3.3.2. They can, additionally, be constructed automatically as detailed in section 3.3.2 as well.

Considering stencils, many real-world applications need *variable coefficient stencils*. In many publications, stencils often have to be fully constant, that is apart from a constant shape they are required to have constant coefficients. Some approaches soften this restriction by allowing stencil coefficients to be expressions which are evaluated at each iteration. We support both these variants and, additionally, storing stencil coefficients in a separate field via stencil fields as introduced in section 3.3.4. The resulting operator can still be used like a 'regular' stencil, thereby reducing code duplication and complexity.

For some applications, *vector fields* can be a great help in keeping a DSL program concise and expressive. Most discussed approaches are, however, limited to scalar fields. Our framework has build-in support for such fields and also supports defining the necessary stencils to operate on them [18], i.e. stencils with small matrices as coefficients.

The last basic language requirement is that *solver logic* must be expressible. This usually includes an iteration loop with an exit criterion
8.6 Discussion

such as the residual falling under a certain threshold. In some DSLs only fixed length loops are allowed which is not applicable in our case where the number of required solver iterations is not known a priori. Additionally, reductions are required to evaluate the norm of the residual. While seemingly natural, often stencil DSLs have no support for this operation. We support both by providing traditional loop constructs developers are accustomed to and reduction operations, as demonstrated in section 3.3.4.

Fulfilling these requirements – in addition to what most presented technologies already are capable of – would suffice to implement simple solvers. They would, however, not be very effective and certainly would not work for more complicated problems. Thus, we also require that colored kernels are supported, to, e.g., express red-black Gauss-Seidel (RBGS) smoothers. Of course, more complicated patterns should be possible as well. We support this directly by allowing arbitrary conditions to be added to loops controlling their execution as shown in section 3.3.4. For many cases, including all simple coloring schemes, our generator can employ polyhedral loop transformations to generate an optimized loop nest similar to what experienced programmers would have implemented by hand. We further improve usability and expressiveness by introducing specialized color constructs, as introduced in section 4.3.2.

For more complex solvers, block smoothers are required. To our knowledge, no other project provides (language) support for these operations. Using simple syntax, we provide an intuitive way to implement such smoothers for a multitude of application scenarios. Examples are given in [12] and section 6.1.

Another important requirement is supporting 1D, 2D and 3D applications. While this seems to go without saying in the context of solving general PDEs, things are not as clear in other domains. Image processing DSLs, for instance, often only focus on 2D domains (images), since other scenarios are of lesser importance. Similarly, most climate DSLs are not applicable since they are only supporting 3D grids. Moreover, they usually restrict how far grids may extend in the third dimension, which also makes them less attractive for our domain. Our framework fully supports all required dimensionalities.

For some applications, simple uniform grids may not be sufficient. In computational fluid dynamics (CFD) applications, for instance, staggered grids are often regarded as state of the art. And even in cases
where uniform grids are not feasible, due to the nature of the solved PDEs, it is sometime still possible to exploit the regular topology of a grid, provided that it is non-uniform. Only very few DSLs can deal with staggered and/or non-uniform grids. Our framework, on the other hand, has not only build-in support for both, as detailed in section 5.2, but also demonstrated its applicability to real-world applications [9].

Restrictions discussed up to now are associated with the DSL, i.e. with how users interact with a given technology. There are, however, some more points which have to be addressed at code generation level.

For one, domain-specific optimizations need to be available. Most of the discussed capabilities have been implemented by Stefan Kronawitter or in collaboration with him. A certain set of optimizations such as vectorization and loop transformations can be considered state of the art at this point since most code generators support them in one form or the other. Our framework is capable of addressing this as well through emitting vector intrinsics for most relevant architectures [73, 94] and by maintaining an interface to the integer set library (ISL) to implement polyhedral transformations. Optimizing the data layout is done much less frequently, but in our opinion critical to attain optimal performance. Classical applications are color-splitting and switching between array of structures (AoS) and structure of arrays (SoA) layouts. We go beyond this by providing a concise interface that allows users to specify arbitrary affine transformations for fields [6]. One other domain-specific optimization that has been researched in the scope of our project is loop-carried common subexpression elimination (LCCSE). Here common sub-expressions are not only detected within one loop iteration but also between loop iterations. This technique is especially relevant in the context of finite volumes (FV) discretizations where we could also demonstrate its effectiveness [7]. At this point one could argue, that some of the optimization techniques implemented by other approaches are not available in our framework. This, however, is true to only a certain extent since often certain scenarios are assumed that do not match our domain. One of these scenarios is the repeated application of the same kernel, often hundreds or thousands of times. In our multigrid solvers, this only repeated operation (without other kernels in-between) is the smoother. It is, however, usually repeated only a couple of times, in our experience typically between one and five times. Another assumption is the presence of a stencil pipeline exhibiting further
parallelization opportunities. For classical multigrid solvers, however, this is not valid.

Last but not least, multiple back ends targeting a wide range of machines are required. While multiple compute nodes with CPUs and GPUs can be considered the base case, additional support for embedded architectures and FPGAs may be beneficial. This is where many stencil DSLs and frameworks fall short, since they provide no way of generating code that is able to run on more than one compute node. Solvers generated by our framework, in contrast, exhibit the required scalability \[11, 12, 17\]. Moreover, we have been able to present successful experiments on CPUs \[11, 9\], GPUs \[12, 6\], ARM-based architectures \[13\] and FPGAs \[20, 21\].

In conclusion, the set of requirements at hand present a unique combination which none of the existing solutions is able to satisfy. Many of the presented DSLs are, additionally, embedded. Implementing a multi-layered DSL, as proposed in this work, would be a though challenge with almost all of them. Moreover, they restrict the syntax in a way that makes the extensions we would require infeasible or even impossible in some cases.

Image processing DSLs are probably closest to the multigrid idea, but frequently lack the capabilities for distributed memory parallelization and, in some cases, 3D problems. Many general stencil DSLs claim that they are suitable to implement efficient PDE solvers. This is, however, often not demonstrated as examples are limited to simple Jacobi solvers which can not be considered efficient.

Development on the work presented in this thesis started already in 2013 and many other relevant approaches also reviewed in this chapter have only been published afterwards. However, most of the discussed shortcomings have still not been fully addressed which makes our proposed DSL and code generation framework somewhat unique. It can, thus, be seen as pioneering work that provides a meaningful addition to the state of the art.
9 Concluding Remarks

9.1 Conclusion

This work presents the complete language stack of ExaSlang, a multi-layered external domain-specific language (DSL) that caters to the needs of multiple user groups. Layer 1 allows working on highly abstract specifications in the continuous domain. Particulars about the discretization can then be expressed on layer 2. The layer 2 specification provides the basis for composing suitable solvers in a Matlab-like syntax on layer 3. All information provided on the upper layers can be combined into a complete program specification on layer 4. It is more closely related to what a programmer would implement in, e.g., C++. Relevant aspects of utilized data structures, their partitioning and the employed parallelization are exposed here as well and can be tuned by users as required.

Transition between the layers of ExaSlang is already highly automated while still allowing users to guide the process via providing hints. Between layer 1 and layer 2, automatic application of the finite difference method (FDM) is supported for a wide range of mathematical operators. For other types of discretization, providing an already discretized equation on layer 2 allows extracting contained operators as stencils or stencil fields. They can be used on subsequent layers – or our framework can be used to automatically compose geometric multigrid solvers from a set of parameters and equations on layer 3. Going to layer 4, we can automatically derive field layouts and add boundary handling methods as well as communication statements tailored to the application at hand.

From a single layer 4 program, a multitude of variants can be generated, thereby allowing specialization for the target hardware platform at hand. Our hierarchical parallelization supports anything between prototyping on a single workstation to doing production runs on the most powerful compute clusters available. Alternative architectures such as
ARMs and FPGAs are just as viable as targets as established technologies given by CPUs and GPUs.

A number of implemented applications from different fields of research demonstrate that our DSL and code generator are useful beyond simple model problems. Especially in the field of computational fluid dynamics (CFD), we can show meaningful results and collaborations with domain scientists. This is only possible by supporting crucial concepts in our language and framework. Among them are non-uniform and staggered grids as well as block smoothers, both not available with most competing approaches. Build-in support for expressing finite volumes (FV) discretizations further improves usability and user experience.

We showcase our capabilities by implementing efficient solvers for three of the most dominant partial differential equations (PDEs) in CFD: Stokes, Navier-Stokes and the shallow water equations (SWE). They not only require different discretization techniques and grids, but they also vary in being elliptic or hyperbolic and being linear or non-linear. Nevertheless, ExaSlang and our code generator are able to handle each of them and still exhibit satisfactory scaling behavior on state-of-the-art clusters such as Piz Daint and JUWELS.

Around our code generation framework, there is rich ecosystem providing assistance for reoccurring task – be it for users or for developers. Build-in support for measuring performance and logging other characteristics makes profiling generated applications straight-forward. Multiple configurations can be evaluated and compared easily using the ConfigRunner tool. The obtained execution times may be compared to automatically applied analytical performance models. Lastly, in-situ visualization capabilities can be added automatically. More on the developer side, the Meta tool allows generating the code generator itself, thereby facilitating tasks such as implementing and maintaining node types on multiple layers.

In conclusion, we could successfully demonstrate how numerical solvers for relevant CFD applications can be implemented in our DSL. Using abstract representations close to the ones domain scientists are familiar with allows for working in an intuitive fashion. Automated solver setup, guided domain partitioning and automatic parallelization further facilitate the process, as does our whole program generation. Generated applications are able to run on single workstations and massively parallel clusters alike while exhibiting the necessary scalability.
This is something that, to our best knowledge, could never be demonstrated before, making this work a relevant contribution to the state of the art.

9.2 Discussion

Implementing such a pipeline as sophisticated as ours requires without doubt an immense work effort. It is thus vital to critically evaluate if this effort pays off or whether other approaches, such as internal DSLs, could be more beneficial. It should go without saying, that this discussion merely reflects our opinions formed from the experiences gathered.

First, we compare the language design point of view. Designing internal and external languages requires all in all about the same amount of work. The key difference is that external languages are much more flexible which not only accelerates setting up prototype languages, but also allows implementing arbitrary language concepts. We heavily use this when mimicking parts of other languages like LaTeX on layer 1 and Matlab on layer 3. On the down-side, external languages require setting up dedicated parsers and lexers. This overhead is not required for internal DSLs. It comes, however, at the price of having to adhere to the host language confines which makes designing the language more cumbersome and time-consuming.

Second, from a users perspective, both approaches have their merits and which is better heavily relies on the target application at hand. If there already exists a large code base in the form of a framework or a legacy code, internal DSLs may be favorable since they allow reusing large parts of said code. For new implementations or prototype applications, however, external languages may be more suited since they allow whole program generation and since they can generally capture domain concepts more closely. The biggest hurdle for new users, in general, is being confronted with a new and unfamiliar programming language. Here, it is vital to provide a good error handling and comprehensible and meaningful error messages. In practice, this is unfortunately a difficult challenge for both approaches.

Lastly, the code generation portion is where external approaches are able to really show their power. By having full access and control over the whole application, sophisticated domain-specific optimizations can be carried out. In our project, these were mostly implemented by Stefan Kronawitter and we could show the applicability to problems from our
domain in joint publications. This includes adapting data structures as demonstrated when we use data-layout transformations for systems of PDEs [6] and using a specialized loop-carried common subexpression elimination (LCCSE) for finite volume discretizations [7]. Of course, this flexibility comes once again with a not to be underestimated overhead. This can be combated to a certain point by using auxiliary software components such as our Meta tool. For new undertakings, it would also be beneficial to reuse parts of the software architecture developed in this work and in the whole ExaStencils project.

In summary, external approaches offer more flexibility in almost all aspects at the price of a heightened initial work effort. If this flexibility can be used adequately, as in our case, then it is beneficial to use such an approach. Moreover, it would not have been possible to implement and evaluate a range of applications as wide as the one presented in this work without the developed technologies. And, last but not least, novel optimizations and parallelization approaches can be implemented directly in the generator. This way, the benefits are not limited to a single program but apply to all applications generated from our DSL.

9.3 Future Work

While we could already show a number of impressive results, there is still much work to be done. First and foremost, refactoring the language specification and enhancing error handling is necessary: Due to the nature of our exploratory work, the scope of functionality that needs to be supported was not clear from the beginning. Instead, we added required functionality as needed, leading to a partially non-uniform language. This shows particularly on the oldest part of ExaSlang – layer 4. A steady language specification also allows improving how erroneous user input is handled which, in turn, can make using ExaSlang more attractive. Also, adding new ways of coupling to external pieces of software, such as legacy codes and scientific frameworks, might be a good step towards this goal.

Another open topic, which needs to be addressed together with collaborators, is improving the performance of the code generator itself. Generation times for large applications are frequently in the range of minutes and can, in extreme cases, also reach multiple hours. Countermeasures could be parallelizing code transformation, optimizing the
underlying Athariac framework and summarizing transformations to reduce the required number of abstract syntax tree (AST) traversals.

On the scope of supported features, the most useful still missing is, in our eyes, support for implementing pre-conditioners, especially on layer 3. Another interesting aspect is given by other types of grids, e.g. triangles or tetrahedra. Embedding them in the types of grids we already provide, as illustrated in figure 9.1, may allow us to retain the performance benefits and reuse most of our generator implementation. Work on this has already begun in the scope of a DFG project for the domain of ocean modeling, more precisely solving the SWE on triangular meshes. We also aim to employ higher order discretizations such as discontinuous Galerkin (DG) which can be set up either directly in ExaSlang or in an external software module generating ExaSlang specifications [1]. This work also includes adding support for more general block-structured grids, i.e. those requiring data transformations in the communication step.

An even more ambitious goal is completing the ExaStencils vision: Providing only a layer 1 input and, without any hints, deriving a suitable solver. This not only requires a highly integrated coupling of the local Fourier analysis (LFA) and parameter tuning, but also being able to optimize algorithms. First steps in tackling the latter issue have already been done by Jonas Schmitt [22].

Figure 9.1: Non-uniform quadrilateral grid with embedded triangles.
9 Concluding Remarks

Last but not least, it would be interesting to see how well our approach translates to similar domains, such as unstructured grids or mesh-less methods.
Bibliography

(Co-)Authored Publications


Bibliography


Supervised Theses


Bibliography


Other References


Bibliography


mesh-based applications on multi-core and many-core architectures. In Proc. Innovative Parallel Computing (InPar), San Jose, California, May 2012. IEEE.


Bibliography


Appendix

Acronyms

AABB  axis-aligned bounding box
AoS   array of structures
AST   abstract syntax tree
BiCGSTAB biconjugate gradient stabilized
CFD   computational fluid dynamics
CFL   Courant-Friedrichs-Lewy
CG    conjugate gradient
CGS   coarse-grid solver
CR    conjugate residual
CSE   common subexpression elimination
CST   concrete syntax tree
CSV   comma-separated values
DG    discontinuous Galerkin
DSL   domain-specific language
FAS   full approximation scheme
FD    finite differences
FDM   finite difference method
FE    finite elements
FEM   finite element method
FFC   FEniCS form compiler
FV    finite volumes
FVM   finite volume method
GA    genetic algorithm
GCC   GNU compiler collection
GP    genetic programming
GPL   general-purpose language
HPC   high performance computing
## Appendix

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICC</td>
<td>Intel C++ compiler</td>
</tr>
<tr>
<td>IR</td>
<td>intermediate representation</td>
</tr>
<tr>
<td>ISL</td>
<td>integer set library</td>
</tr>
<tr>
<td>IV</td>
<td>internal variable</td>
</tr>
<tr>
<td>JIT</td>
<td>just-in-time</td>
</tr>
<tr>
<td>LBM</td>
<td>lattice Boltzmann method</td>
</tr>
<tr>
<td>LCCSE</td>
<td>loop-carried common subexpression elimination</td>
</tr>
<tr>
<td>LFA</td>
<td>local Fourier analysis</td>
</tr>
<tr>
<td>MINRES</td>
<td>minimal residual</td>
</tr>
<tr>
<td>MSVC</td>
<td>Microsoft visual C++</td>
</tr>
<tr>
<td>PDE</td>
<td>partial differential equation</td>
</tr>
<tr>
<td>PGAS</td>
<td>partitioned global address space</td>
</tr>
<tr>
<td>RBGS</td>
<td>red-black Gauss-Seidel</td>
</tr>
<tr>
<td>SIMPLE</td>
<td>semi-implicit method for pressure linked equations</td>
</tr>
<tr>
<td>SoA</td>
<td>structure of arrays</td>
</tr>
<tr>
<td>SOR</td>
<td>successive over-relaxation</td>
</tr>
<tr>
<td>SPDE</td>
<td>stochastic partial differential equation</td>
</tr>
<tr>
<td>SPL</td>
<td>software product line</td>
</tr>
<tr>
<td>SpMV</td>
<td>sparse matrix-vector multiplication</td>
</tr>
<tr>
<td>SWE</td>
<td>shallow water equations</td>
</tr>
<tr>
<td>TBB</td>
<td>threading building blocks</td>
</tr>
<tr>
<td>TPDL</td>
<td>target platform description language</td>
</tr>
<tr>
<td>TSC</td>
<td>time stamp counter</td>
</tr>
<tr>
<td>TSFC</td>
<td>two stage form compiler</td>
</tr>
<tr>
<td>UFC</td>
<td>unified form-assembly code</td>
</tr>
<tr>
<td>UFL</td>
<td>unified form language</td>
</tr>
</tbody>
</table>
## Benchmark Machines

<table>
<thead>
<tr>
<th>name</th>
<th># nodes</th>
<th>CPU</th>
<th>RAM</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>workstation</td>
<td>1</td>
<td>Intel Xeon E5-2623 v3, 4 cores, 3.0 GHz</td>
<td>64 GB</td>
<td>2 NVIDIA GeForce GTX 980 Ti, 2 816 CUDA cores, 1 000 MHz, 6 GB, 336 GB/s</td>
</tr>
<tr>
<td>JUQUEEN</td>
<td>28 672</td>
<td>IBM PowerPC A2, 16 cores, 1.6 GHz</td>
<td>16 GB</td>
<td>—</td>
</tr>
<tr>
<td>Piz Daint</td>
<td>5704</td>
<td>Intel Xeon E5-2690 v3, 12 cores, 2.6 GHz</td>
<td>64 GB</td>
<td>NVIDIA Tesla P100, 3 584 CUDA cores, 1 328 MHz, 16 GB, 732 GB/s</td>
</tr>
<tr>
<td>JUWELS (standard nodes)</td>
<td>2 271</td>
<td>2 Intel Xeon Platinum 8168, 24 cores each, 2.7 GHz</td>
<td>96 GB</td>
<td>—</td>
</tr>
</tbody>
</table>

*Table 2:* Hardware platforms utilized for performance evaluation.
Solving partial differential equations (PDEs) is a fundamental challenge in many application domains in industry and academia alike. With increasingly large problems, efficient and highly scalable implementations become more and more crucial. Domain-specific languages (DSLs) and code generation techniques hold the power to automate the application of domain-specific optimizations as well as mapping to the increasingly heterogeneous hardware landscape of today. This work aims to further the state of the art in this field, in particular for PDE solvers based on geometric multigrid methods operating on (patch-)structured grids.

We begin by developing our multi-layered external DSL ExaSlang: Layer 1 is designed to resemble LaTeX and allows inputting continuous equations and functions. Their discretization is expressed on layer 2. It is complemented by algorithmic components implemented in a Matlab-like syntax on layer 3. All information provided to this point is summarized on layer 4, enriched with particulars about data structures and the employed parallelization. For convenience, we support automated progression between the different layers. All ExaSlang input is processed by our jointly developed Scala code generation framework to ultimately emit C++ code. Generated applications are automatically parallelized with MPI, OpenMP and CUDA to run on platforms ranging from workstations to large-scale clusters.

We showcase the applicability of our approach by implementing simple test problems, like Poisson’s equation, as well as relevant applications from the field of computational fluid dynamics. In particular, we implement scalable solvers for the Stokes, Navier-Stokes and shallow water equations discretized with finite differences and finite volumes. We also extend our implementation towards non-uniform grids and advanced effects such as the simulated fluid being non-Newtonian and non-isothermal.