DOI: 10.2478/auom-2020-0038
An. Şt. Univ. Ovidius Constanţa **Vol. 28(3),2020, 123–152**

Code generation approaches for parallel geometric multigrid solvers

Harald Köstler, Marco Heisig, Nils Kohl, Sebastian Kuckuk, Martin Bauer, Ulrich Rüde

Abstract

Software development for applications in computational science and engineering has become complex in recent years. This is mainly due to the increasing parallelism and heterogeneity in modern computer architectures and to the more realistic physical and mathematical models that have to be processed. One idea to address this issue is to use code generation techniques. In contrast to manual implementations in a general-purpose computing language, they allow to integrate automatic code transforms to produce efficient code for different models and platforms. As an example the numerical solution of an elliptic partial differential equation via generated geometric multigrid solvers is considered. We present three code generation approaches for it and discuss their advantages and disadvantages with respect to performance, portability, and productivity.

1 Introduction

Developing simulation software is often done in interdisciplinary teams consisting of members from a particular application domain, applied mathematics, and computer science. Due to their varying background in modeling, mathematics, and programming, suitable software tools can make the work and especially the interfacing between the fields more productive. In the best case,

Key Words: code generation, domain-specific languages, multigrid solvers, elliptic PDEs.
2010 Mathematics Subject Classification: Primary 46G05, 46L05; Secondary 47A30.
Received: 10.07.2019.
Accepted: 16.12.2019.

there are specific tools for each user and there exists a toolchain that covers all steps from the model formulation over specification of numerical methods down to their concrete implementation on a certain platform. It should also require minimal intervention from users.

We discuss in this paper different approaches to achieve this goal, which are based on code generation techniques. This means that an abstract problem specification is first translated into a domain-specific representation that can then be processed successively. At the end one obtains simulation code that can be either compiled or executed directly. As a simple example, we will show how geometric multigrid solvers for elliptic partial differential equations can be generated.

Classically most people use available software libraries or frameworks that provide efficient multigrid solvers. Some of the popular ones are PETSc [2], HYPRE [15], or Trilinos [27] which understands itself as a collection of packages rather than a single framework, such as Deal.II [3], DUNE [4, 8], UG4 [55], Peano [56] and Chombo [1].

However, such frameworks are usually either too general to efficiently implement domain-specific optimizations for multiple hardware platforms or a high effort is required to adapt them. In practice, code generation techniques can be used to tackle this. As input they use external or embedded domain-specific languages (DSLs) and the output is usually either a complete program or single compute kernels that can be injected into existing software.

A prominent example for an external DSL is SPIRAL [46, 45, 9] that was originally developed for the domain of Fast Fourier transforms. It needs to provide less functionality compared to a general purpose language and thus is easier to maintain and use. Examples for DSLs in the domain of PDEs are Liszt [13] and OP2 [44] that focus on solvers operating on unstructured meshes, FEniCS [42] and Firedrake [47] that focus on finite element applications, SBLOCK [10] that focuses on block-structured grids, and PATUS [11], Pochoir [53], Physis [43], SDSLc [48], MODESTO [23], or STELLA [24], short for *stencil loop language*, that focus on structured grids. STELLA, e.g., is a DSL embedded in C++ for applications in weather and climate simulation.

Another approach that is popular in practice but out of scope in this work is given by *language extensions* that introduce abstractions for parallelization. Examples are, among others, CHARM++ [28], Cilk [16], UPC [12], Dash [17], or Kokkos [14]. Note that one reoccurring concept in many of these approaches is a partitioned global address space. 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.

One question arises now: what are the conceptional differences between existing DSLs for geometric multigrid solvers? To provide an answer to this, we selected three different DSLs developed in our research group and discuss their properties. The first is an multi-layered external DSL that covers several levels of abstraction as described above from model to code. The second is based on a classical software framework, which is extended by efficient compute kernels. These are generated with the help of an external tool. The third is an embedded DSL with special focus on just-in-time code generation.

The rest of the paper is structured as follows: Section 2 formulates the basic mathematical problem under consideration and introduces different abstraction layers for it. Challenges for the implementation of the numerical methods are pointed out and the requirements for a DSL that supports parallel geometric multigrid solvers are listed. In section 3 we introduce the three conceptionally very different approaches that enable the generation of software for numerical algorithms, use all of them to generate a multigrid solver, show performance results, and discuss the advantages and disadvantages of the approaches. Section 4 summarizes the paper and outlines possible directions for future work.

2 Abstraction Layers

As a simple test case, we consider Poisson's equation as an elliptic PDE that can be described by

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega & \text{(1a)} \\ u &= g \quad \text{on } \partial\Omega & \text{(1b)} \end{aligned}$$

where $\Omega = (0,1)^d, d \in \{2,3\}$, $u, f : \Omega \mapsto \mathbb{R}$, $g : \partial\Omega \mapsto \mathbb{R}$, and where we assume non-homogeneous Dirichlet boundary conditions.

Different layers of abstractions can describe the process of transforming such a continuous PDE model into concrete code. These correspond to the usual steps taken by a domain expert and are depicted in figure 1. Depending on the application, various choices are possible on each layer.

Layer 1 (Continuous Model) While (1) is a simple test case, a DSL on this layer should offer the following features to the user: definition of the computational domain for different dimensions like 1D, 2D, and 3D; support for common differential operators and boundary conditions; support for systems of equations and nonlinear problems. Once the problem specification is finished, certain *symbolic transformations* that do preserve the underlying model constraints can be applied to simplify or to manipulate the equations.

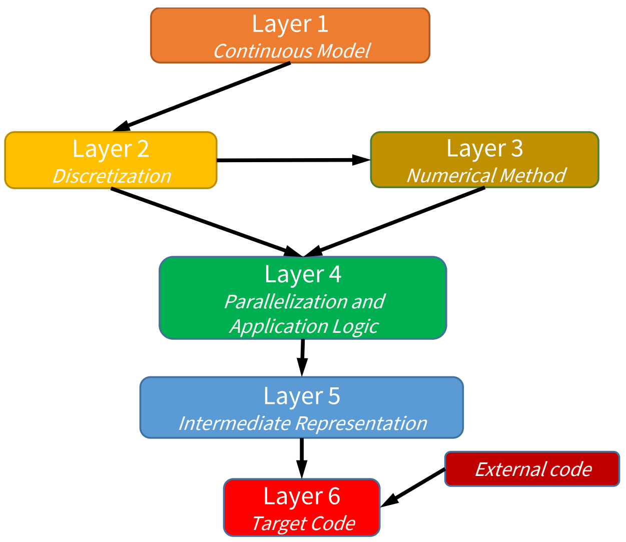

Figure 1: An abstract mathematical PDE model is transformed to concrete code using several layers of abstraction.

Layer 2 (Discretization) On this layer, many variants are possible to select the partition of the domain in elements, e.g., quadrangles, triangles, cubes, and to define discrete operators on them. Finite difference (FD), finite volume, finite element (FE) or discontinuous Galerkin discretizations are some of the most prominent methods. In this work, we restrict ourselves to uniform grids and discretize (1) by FD or FE.

This results in a sparse linear system of equations $A^h u^h = f^h$ with the discretization matrix $A^h \in \mathbb{R}^{N \times N}$, the vector of unknowns $u^h \in \mathbb{R}^N$ and the right hand side (RHS) vector $f^h \in \mathbb{R}^N$ on a discrete grid Ω^h. N denotes the degrees of freedom or number of unknowns in the linear system. In our case, A can be stored in a compact, matrix-free form using a constant stencil.

The DSL on this layer should offer the following features to the user: a *representation of discretized functions* for solution, right hand side and possible coefficients both for scalar and vector-values quantities; a *representation of the discrete operators* as constant or variable stencils or stencil fields; different *grid types* and *discretization strategies*. In the end continuous quantities are discretized on the specified computational domain.

Layer 3 (Numerical Method) This layer allows the user to express numerical methods like solvers for linear systems. It is complementary to layer 2, since different discretization and solver specifications can be coupled in various ways. To solve the above discrete linear system approximately, we apply a multigrid method [54]. As an iterative solver for large, sparse linear systems its major advantage is that it has a convergence rate independent of the mesh size. One multigrid iteration, the so-called *V-cycle*, is summarized in Algorithm 1.

Algorithm 1 Recursive V-cycle: $u_h^{(k+1)} = V_h(u_h^{(k)}, A^h, f^h, \nu_1, \nu_2)$

```
1: if coarsest level then
2:     solve $A^h u^h = f^h$ by a (parallel) direct solver or by CG iterations
3: else
4:     $\bar{u}_h^{(k)} = \mathcal{S}_h^{\nu_1}(u_h^{(k)}, A^h, f^h)$ {presmoothing}
5:     $r^h = f^h - A^h \bar{u}_h^{(k)}$        {compute residual}
6:     $r^H = R r^h$                {restrict residual}
7:     $e^H = V_H(0, A^H, r^H, \nu_1, \nu_2)$ {recursion}
8:     $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + P e^H$     {prolongate error and do coarse grid correction}
9:     $u_h^{(k+1)} = \mathcal{S}_h^{\nu_2}(\tilde{u}_h^{(k)}, A^h, f^h)$ {postsmoothing}
10: end if
```

For users it is convenient if the DSL on this layer is as close as possible to the high-level pseudo code used in Algorithm 1 for the chosen numerical method. It should offer the following features to the user: support for *multiple grids* with multiple resolutions; *operators that map between grids* with different resolutions (restriction operator R and prolongation operator P); support for various common relaxation schemes like Jacobi or red-black Gauß-Seidel (RBGS) smoothers; constructs like conditions or loops to be able to express the *solver logic*, this usually includes an iteration loop with a termination criterion such as the residual falling under a certain threshold; support for *reductions* to evaluate norms and dot products.

Layer 4 (Parallelization and Application Logic) On this layer, the information necessary to compose the complete specification is gathered.

A DSL on this layer should be close to a general purpose language with certain restrictions that allow easier domain-specific abstractions and code optimizations.

One part is the specification of the application logic. Here, input data, efficient *compute kernels* for numerical operations including algorithmic parameters, loops, and data output and visualization have to be expressed. At some point within the application, the solver specified on layer 3 can be executed. Another part is the parallelization concept that depends on the concrete platform on which the generated code will be executed. Suitable parallel data structures, domain partitioning, *parallelization strategies* for vectorization, shared-memory parallelization, distributed-memory parallelization, and *communication routines* for data exchange and synchronization can be tuned. On this layer, users merely provide the information required to actually realize these aspects on layer 5. Note that one of the main differences between code generation techniques is to which extent these requirements are contained in an existing HPC framework or must be generated. External code can be interfaced via, e.g., function calls on this layer. Embedding external code, however, is not possible, since this would require to support all features of a general purpose programming language.

Layer 5 (Intermediate Representation) This layer is internal, i.e., not exposed directly to the user, and usually implemented in a general-purpose programming language. Here, the main differences between the approaches are found. Here, most of the domain-specific optimizations are applied. Examples are *vectorization and loop transformations*, *data layout optimizations* like color-splitting and switching between AoS (array of structures) and SoA (structure of arrays) layouts, or *common sub-expression elimination* either only for expressions within one loop iteration or also between loop iterations.

Layer 6 (Target Code) This internal layer is the interface to the concrete platform the code will be executed on and thus should provide support for *multiple back ends* targeting a wide range of machines. While multiple compute nodes with CPUs and GPUs are found in many compute clusters currently, additional support for embedded architectures and FPGAs may be beneficial. In addition to that, *interfaces to existing HPC software frameworks or legacy codes* can be generated, if there are parts of the application that are already implemented and it is not necessary or possible to generate them again.

3 Code generation for geometric multigrid solvers

A lot of effort is necessary to maintain, adapt, and extend simulation frameworks that cover all six layers described above. Understanding the abstractions requires specialist skills, i.e., application, mathematical and computer science expertise. This is only possible when domain scientists and computer scientists collaborate over longer periods of time. But such a constellation is challenging to achieve and in many cases the effort is prohibitive. Therefore, in the last years, various approaches to automate the process to derive a simulation code from a mathematical model by code generation have been explored. Our group conducted research in several projects on both external and embedded domain-specific HPC languages in Python [5], Scala [30], AnyDSL [52], C++ and CommonLisp [26].

In this section, we show how the mathematical problem stated in (1) can be described in three different approaches that are currently in different stages of development.

3.1 Test Problem

To show the performance of the resulting implementations we solve (1) in 2D and 3D. The computational domain Ω is the unit square and unit cube, respectively. As a solver, we employ a geometric multigrid solver using a V(2,1)-cycle and Gauß-Seidel-type smoothers. Due to the specializations of the three approaches, we use different discretizations and possibly different orderings during the relaxation as well as appropriate interpolation and restriction routines. All components that are special to the individual implementations are described below. For 2D we set the right-hand side

$$f = \pi^2 \cos(\pi x) - 4\pi^2 \sin(2\pi y),\ g = \cos(\pi x) - \sin(2\pi y)$$

as boundary value function. In 3D, we regard

$$f = \pi^2 \cos(\pi x) - 4\pi^2 \sin(2\pi y) + 4\pi^2 \cos(2\pi z),$$

and

$$g = \cos(\pi x) - \sin(2\pi y) + \cos(2\pi z)$$

accordingly. Both test cases start from an initial guess $u = 0$ and stop as soon as the L_2-norm of the residual drops below 10^{-10}.

3.2 Per-node Performance Estimation

To estimate the expected per-node performance, we use an optimistic roofline model, since the performance is known to be limited by memory bandwidth in this test case [31]. As benchmark machine, we choose one workstation with two Intel Xeon Gold 5122 CPUs featuring four physical cores each. We measure the achieved copy stream bandwidth using likwid[*] resulting in 39 GB/s per CPU. Thus, the maximal throughput is $2 \cdot 39 = 78$ GB/s for both sockets. For each degree of freedom (DoF) we have to load the right-hand side and the solution and store the solution back to memory for one smoothing step. For computing the residual the same holds, but instead of the solution we

[*] `https://hpc.fau.de/research/tools/likwid/`

store the residual. In case of an RBGS smoother we assume that we store the different colors in separate fields and thus have to load and store each value only once. In addition to that, we count 1 load operation for the restriction and one load and one store for the prolongation and fused correction. Thus on the finest level, $8 \cdot (3 \cdot 3 \cdot 2 + 3 + 2 + 1) = 120$ bytes per degree of freedom are transferred for a V(2,1)-cycle and double accuracy. This is multiplied by a factor of $1+1/4+1/16+1/64+\cdots \approx 1.33$ in 2D resp. $1+1/8+1/64+\cdots \approx 1.14$ in 3D for the case of a multigrid solver operating also on coarser levels. These estimates provide a lower bound for the per-node performance, since they neglect any potential overhead including synchronization. They also assume perfect blocking, i.e., are independent of the concrete stencil shapes. In the 2D case the lower bound is then 487 MDoF/s and in the 3D case 568 MDoF/s. We can check now how far off the performance of generated code is in practice.

3.3 ExaStencils

General concept The main aim of project ExaStencils[†] is the generation of highly efficient and massively parallel geometric multigrid solvers from abstract descriptions given in its own external DSL *ExaSlang* – short for ExaStencils language [40, 41, 49].

ExaSlang is conceptualized as a multi-layered DSL, following the concepts introduced in Section 2. On each layer, different aspects of a given problem and associated solver can be specified. Ideally, it would be sufficient to state the problem to be solved on layer 1. This includes the computational domain, boundary conditions, unknowns, and the equation(s) to be solved. In practice, however, more domain knowledge is necessary, e.g., concerning the discretization or numerical methods. We address this by supporting a guided semi-automatic discretization that can be performed to generate layer 2. Here, the discretized counterparts of layer 1 can be reviewed before a suitable solver is composed on layer 3.

Information from layer 2 and 3 is then combined to form a complete program specification on layer 4. Additionally, communication can be added automatically for functions originating from layer 3. At this point, specifics about memory layouts such as the number of ghost layers required for distributed memory parallelization are deduced and exposed as well [38]. All in all, the (generated) layer 4 description is complete in the sense that it can be used to generate target code without requiring layers 1 through 3. This is also the layer where specialized data layout transformations, e.g. from an array of structures (AoS) to a structure of arrays (SoA), can be specified [32].

[†]`www.exastencils.org`

By design, users can intervene at any layer if they want to alter the generated parts of the code or if they want to express something that can not be generated automatically yet. Likewise, it is possible and indeed common to start at lower layers and to implement on multiple layers concurrently.

Features The ExaStencils code generator is implemented in Scala and is able to process ExaSlang input to set up representations in its own intermediate representation (IR). There, various code transformations are applied such that it can, ultimately, be mapped to C++ code parallelized with MPI, OpenMP and/or CUDA. Available transformations include automatic parallelization [38] for the previously named back ends as well as the automatic application of optimizations geared to the target hardware at hand. Examples for the latter are address pre-calculation, loop transformation, a sophisticated common sub-expression elimination (CSE) [34] and vectorization. The emitted code can be executed on various platforms ranging from traditional CPU [36, 35] and hybrid CPU-GPU [37] clusters, ARM-based architectures [39] to reconfigurable hardware [51, 50].

Layer 1 (Continuous Model) On layer 1, our benchmark problem can be described by the DSL code shown in listing 1. As evident, the syntax is kept close to the notation used in scientific publications. By providing so-called hints for the discretization and solver components, the framework is able to compose subsequent layer specifications automatically.

```
Ω = ( 0, 1 ) × ( 0, 1 )

u ∈   Ω = 0.0
u ∈ ∂ Ω = cos ( π x ) - sin ( 2 π y )
f ∈   Ω = π^2 cos ( π x ) - 4 π^2 sin ( 2 π y )

op = - Δ
uEq: op * u == f

DiscretizationHints {
  f on Node
  u on Node
  op on Ω

  uEq
  // parameters
  discr_type = "FiniteDifferences"
}

```

```
SolverHints {
  generate solver for u in uEq
}
```

Listing 1: ExaSlang layer 1 example for the complete specification of the 2D Poisson problem.

Layer 2 (Discretization) At this layer, the discrete domain and the discrete linear system have to be described. Listing 2 illustrates how a discretized variant of the previous specification can be set up on layer 2. It is similar to a version automatically generated using the provided discretization hints.

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

Solution with Real on Node of global = 0.0
Solution@finest on boundary =
  cos ( PI * x ) - sin ( 2.0 * PI * y )
Solution@(all but finest) on boundary = 0.0

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 )
  [-1,  0] => -1.0 / ( vf_gridWidth_x**2 )
  [ 1,  0] => -1.0 / ( vf_gridWidth_x**2 )
  [ 0, -1] => -1.0 / ( vf_gridWidth_y**2 )
  [ 0,  1] => -1.0 / ( vf_gridWidth_y**2 )
}

SolEq {
  Laplace * Solution == RHS
}
```

Listing 2: ExaSlang layer 2 example for a complete specification of the 2D Poisson problem.

Layer 3 (Numerical Method) While layer 2 is used to express the problem, layer 3 is geared towards algorithmic components of the numerical solver. Such a solver, which is again similar to the auto-generated variant, is displayed in listing 3.

```
Field Residual from Solution
override bc for Residual with 0.0

Operator Restriction  from default restriction
  on Node with 'linear'
Operator Prolongation from default prolongation
  on Node with 'linear'

Function Smoother@all {
    Solution += diag_inv ( Laplace ) * ( RHS -
      Laplace * Solution ) where (i0 + i1) % 2 == 0
    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 ( )
  Smoother ( )
  Residual = RHS - Laplace * Solution
  RHS@coarser = Restriction * Residual

  Solution@coarser = 0.0
  VCycle@coarser ( )

  Solution += Prolongation@coarser * Solution@coarser
  Smoother ( )
}
```

Listing 3: ExaSlang layer 3 implementation of a V(2, 1)-cycle using a RBGS smoother.

Layer 4 (Complete Specification) Based on the layer 2 and 3 specifications, a complete layer 4 program, comparable to the one in listing 1, can be assembled. Details about the data structures, such as the number of overlapping and ghost layers as well as which of them are to be communicated, can be adapted here. `communicate` statements provide an easy-to-use interface to fine-tune the communication behavior. Here, it would also be possible to manually overlap computation and communication.

```
Layout DefLayout<Real, Node >@all {
  duplicateLayers = [1, 1] with communication
  ghostLayers     = [1, 1] with communication
}

Field Solution< global, DefLayout, /* bc's */ >@finest
Field Solution< global, DefLayout, 0.0 >@(all but finest)
Field RHS      < global, DefLayout, None >
Field Residual< global, DefLayout, 0.0 >

/* operators as on layers 2 and 3 */

Function Smoother@all {
  color with (i0 + i1) % 2 {
    loop over Solution {
      Solution += omega * diag_inv ( Laplace ) *
                  ( RHS - Laplace * Solution )
    }
    communicate Solution
  }
}
/* VCycle functions */

Function Application {
  /* initialization */
  reapeat 10 times {
    VCycle@finest ( )
  }
  /* de-initialization */
}
```

Listing 4: ExaSlang layer 4 example of a full application applying a fixed number of V-cycles.

Runtime results Based on these representations, we benchmark our generated multigrid solvers of our benchmark problem from section 3.1. The finite difference discretization results in a 5-point (2D) and a 7-point (3D) stencil. We employ $V(2,1)$-cycles with a RBGS smoother and evaluate a serial version as well as an OpenMP parallel one using 16 threads. The results are summarized in table 1.

Using the performance model outlined in section 3.2, we obtain an optimistic performance prediction of about 2.3s in 2D and 1.9s in 3D per multigrid cycle for the case of $1.1 \cdot 10^9$ degrees of freedom (DoF).

Since the ExaStencils framework is able to apply a suitable color splitting [33], the measured times of 2.6s (423 MDoF/s) and 3.0s (314 MDoF/s) are very satisfactory. Of course one cannot expect to reach the estimate, e.g., the same color splitting makes the inter-grid kernels much more complex and thus likely more costly than the prediction. In 3D the stencil becomes bigger and the memory accesses are more complex what increases runtime. A more detailed analysis is beyond the scope of this work.

Discr.	Dim	DoFs	Iter	L^∞-error	Time / Cycle [s]	
					Num cores	
					1	8
FD	2D	$4.2 \cdot 10^6$	10	7.2e-07	0.054	0.007
		$1.7 \cdot 10^7$	10	1.8e-07	0.25	0.033
		$6.7 \cdot 10^7$	10	4.5e-08	0.96	0.15
		$2.7 \cdot 10^8$	10	1.1e-08	3.7	0.60
		$1.1 \cdot 10^9$	10	2.8e-09	19	2.6
FD	3D	$2.0 \cdot 10^6$	12	3.6e-04	0.025	0.004
		$1.7 \cdot 10^7$	12	9.0e-05	0.28	0.037
		$1.3 \cdot 10^8$	12	2.2e-05	2.3	0.36
		$1.1 \cdot 10^9$	12	5.6e-06	20	3.0

Table 1: Performance results for multigrid (V$(2,1)$-cycle, RGBS) solvers generated by the ExaStencils framework.

3.4 HyTeG

General concept HyTeG (Hybrid Tetrahedral Grids) [29] is a massively parallel finite element framework with a strong focus on matrix-free geometric multigrid solvers on block-structured tetrahedral grids. It is built around the concept of hierarchical hybrid grids as introduced in the early 2000s [6, 7]. As a re-design of the prototype framework HHG [18, 21], we aim to extend the core concepts of HHG to advanced discretizations and flexible data structures. The matrix-free multigrid solvers have proven their computational and numerical performance as well as their extreme scalability in geophysical applications [19, 22].

While the tetrahedral coarse-grid structure of domains in HyTeG offers more geometric flexibility than cuboid grids, the manual development of efficient compute and communication kernels is more challenging. We alleviate this problem by an extension to the *pystencils*[‡] framework that allows us to generate kernels that operate on simplicial data arrangements.

[‡]https://i10git.cs.fau.de/pycodegen/pystencils

Features HyTeG adopts a rather classical approach to simulation software. The discretization and solver algorithms are defined directly in the C++ framework. Application scientists are then able to modify and extend the source code themselves without having to learn a domain specific language. Moreover, interfacing with other C or C++ libraries is directly possibly.

However, due to the block-structured tetrahedral grids, the development of efficient kernels is to a large extend shifted to the code generation library *pystencils*. *pystencils* is a Python software library that automates the generation of numeric codes from a symbolic description. It extends the *sympy* library by data structures that represent the (discrete) computational domain. However, *pystencils*, opposed to the ExaStencils approach, generates compute kernels that are called from the C++ framework.

Apart from AST- and algebraic optimization techniques, HyTeG extends the *pystencils* library by a special field-access resolution for simplices. To operate on fields with different shapes (e.g. cuboid, tetrahedral, triangular) and different discretizations (FD, FE), indexing transformations have been developed as an abstraction from the underlying update pattern.

In HyTeG we require matrix-free compute and communication kernels that operate on finite-element discretizations of different numerical order. We avoid on-the-fly integration over the elements when possible and employ an automatically optimized stencil-based update pattern instead. For example for constant-coefficient operators, the integrals can be replaced with such a stencil-based kernel, where the resulting stencil is constant in every coarse-grid element. In this way, the block-structured domain partitioning into tetrahedrons is exploited, and the generator yields computationally efficient kernels.

Layer Formulations Similar to other classical frameworks, HyTeG does not offer an explicit representation for the continuous model. It is entirely defined outside of the framework by an application scientist. The finite element discretization and stencil assembly is done by the C++ framework. The tetrahedral meshes are implicitly defined through uniform refinement of the coarse grid elements. As for the operators, we partly implement our own C++ routines and also in some cases use the FEniCS [42] form compiler (FFC) to generate routines that compute the required local stiffness matrices. We assemble the stencils either in a setup phase, or on-the-fly as part of the compute kernels. The numerical methods are implemented as C++ templates in HyTeG. The individual kernels that are called by the framework are blackboxes and defined at layer 4.

In our showcase, we employ a geometric multigrid iteration (see listing 5).

```
if ( level == minLevel_ )
{
 coarseSolver_->solve( A, x, b, minLevel_ );
}
else
{
  for ( uint_t i = 0; i < preSmoothingSteps; ++i )
    smoother_->solve( A, x, b, level );

  A.apply( x, ax_, level, flag_ );
  tmp_.assign( {1.0, -1.0}, {b, ax_}, level, flag_ );

  restrictionOperator_->restrict( tmp_, level, flag_ );

  b.assign( {1.0}, {tmp_}, level - 1, flag_ );
  x.interpolate( 0, level - 1 );
  solveRecursively( A, x, b, level - 1 );

  tmp_.assign( {1.0}, {x}, level, flag_ );
  prolongationOperator_->prolongate(x,level - 1,flag_);
  x.add( {1.0}, {tmp_}, level, flag_ );

  for ( uint_t i = 0; i < postSmoothingSteps; ++i )
    smoother_->solve( A, x, b, level );
}
```

Listing 5: C++ implementation of a V-cycle in the HyTeG framework (stripped). The individual kernels (smoothers, restriction, etc.) are generated.

At this point, parallelization is already fully included in the C++ framework. The communication, performed via MPI, is hidden behind the low-level building blocks such as matrix-vector multiplication or grid transfer. In a last step, we generate the individual compute kernels that are called by the numerical algorithm. This happens at compile time using the *pystencils* library. In listing 6 we show a Python recipe that is used to generate a Gauß-Seidel with overrelaxation kernel for a 3D discretization. *sp* refers to sympy.

```
rhs = VertexTetrahedronField('rhs', const=True)
dst = VertexTetrahedronField('dst')

stencil_map = StencilMap('p1_stencil', [tuple], float)

relax = TypedSymbol('relax', float)

s = sum([stencil_map(d) * dst(d)
         for d in vertexdof_neighbors_at_vertexdof(
                             dim=3, with_center=False)])
```

```

update = sp.Eq(dst((0, 0, 0)),
               (1.0 - relax) * dst((0, 0, 0)) +
               relax * ((rhs((0, 0, 0)) - s) /
                        stencil_map((0, 0, 0))))
```

Listing 6: *pystencils* recipe for the Gauß-Seidel with overrelaxation algorithm applied to a 3D, scalar operator that is discretized with tetrahedrons and linear, conforming finite elements.

The *pystencils* recipes are transformed into an AST internally during the kernel generation. This AST undergoes various transformations and optimizations before the backend emits the final C++ code. The resulting application is compiled with a C++ compiler.

Runtime Results We discretize (1) using linear (P1) and quadratic (P2) conforming finite elements for both the 2D and 3D test case. The resulting stencils are much denser than those originating from the finite difference discretizations. For example, we end up with a 7-point stencil in 2D and 15-point stencil for the linear 3D case. For the 3D quadratic finite element discretization we end up with different stencil sizes roughly between 20 and more than 70 points.

We emphasize that the benchmark setting is unfavorable for the triangular and tetrahedral grids that are employed in HyTeG. The geometric flexibility that this approach offers is not at all exploited by a benchmark on a unit square or cube respectively. For all of these reasons, the results cannot be compared directly to the other approaches, but can only roughly classify the overall performance.

We employ a lexicographic Gauss-Seidel smoother for the linear discretizations and a multi-color ordering for the quadratic test cases that is tied to the orientation of the edges and the corresponding, constant stencils. See [29] for more details. The execution is performed MPI-parallel, without OpenMP on a single node. The results are listed in table 2 and in table 3. In 2D, we achieve up to 200 MDoF/s (P1) and 198 MDoF/s (P2), in 3D 142 MDoF/s (P1) and 87 MDoF/s (P2) for one V(2,1)-multigrid cycle. Due to the lexicographic iteration, no vectorization is performed in the smoother. Also, vectorization is harder to accomplish in general because of the triangular/tetrahedral memory layout.

The simplistic performance model in section 3.2 only partly fits to the HyTeG implementation, since both, the stencils and the memory layout are different. In the three-dimensional linear case and in the quadratic case, we cannot even assume that the kernels are bound by the bandwidth of the system since the stencils are much larger [20]. A detailed performance model is out of the scope of this article.

However, the HyTeG approach only loses about a factor 2 of performance comparing the linear finite element cases with the finite difference discretization of ExaStencils.

Discr.	DoFs	Iter	L^∞-error	Time / Cycle [s]			
				Num cores			
				1	2	4	8
P1	$8.4 \cdot 10^6$	9	2.8e-06	.31	.16	.09	.05
	$3.4 \cdot 10^7$	8	7.5e-07	1.17	.60	.32	.17
	$1.3 \cdot 10^8$	8	2.6e-07	4.57	2.32	1.25	.68
	$5.4 \cdot 10^8$	8	2.7e-07	19.71	9.15	4.94	2.70
P2	$8.4 \cdot 10^6$	9	1.4e-09	.34	.18	.11	.07
	$3.4 \cdot 10^7$	9	2.5e-09	1.20	.62	.35	.20
	$1.3 \cdot 10^8$	9	9.7e-09	4.55	2.35	1.32	.73
	$5.4 \cdot 10^8$	8	3.9e-08	19.34	9.14	5.17	2.73

Table 2: Performance results in 2D for the multigrid (V(2,1)-cycle) implementation for finite element discretizations in the HyTeG-framework.

Discr.	DoFs	Iter	L^∞-error	Time / Cycle [s]			
				Num cores			
				1	2	4	8
P1	$8.5 \cdot 10^6$	10	4.3e-04	.51	.28	.17	.09
	$6.8 \cdot 10^7$	9	1.5e-04	3.44	1.88	1.03	.55
	$5.4 \cdot 10^8$	7	5.7e-05	26.39	13.05	7.08	3.81
P2	$8.5 \cdot 10^6$	9	3.1e-06	2.01	1.11	.75	.42
	$6.8 \cdot 10^7$	9	6.0e-07	5.95	3.35	2.12	1.18
	$5.4 \cdot 10^8$	8	2.1e-06	38.87	20.35	11.52	6.22

Table 3: Performance results in 3D for the multigrid (V(2,1)-cycle) implementation for finite element discretizations in the HyTeG-framework.

3.5 Petalisp

General Concept Inspired by array programming languages like APL, and parallel programming languages like SISAL and SAC, Petalisp is a minimalistic embedded DSL for parallel array computations, written in Common Lisp. The goal of the project is to investigate the potential of generating and compiling kernels at runtime, based on simple data-flow graphs that are also assembled at runtime. At its core, Petalisp provides only a single data structure – the lazy, strided array – and four primitive operators. These operators are:

- **map** The operator `α` applies a pure function of arity n element-wise to the n supplied arrays of identical shape. It returns an array of the same shape, containing all results. If the shapes of the given input arrays are different, an attempt is made to broadcast smaller arrays to the shape of larger ones automatically.
- **reduction** The operator `β` reduces the elements an array with rank n along the first axis, using some pure, binary function. It returns an array of rank $n - 1$, containing the results of each individual reduction.
- **fusion** The operator `fuse` combines several smaller arrays of identical rank into a single, large array. An error is signaled if the given arguments overlap, or if the result cannot be expressed as a strided array, e.g., when there are large gaps, or when arrays of different strides are placed next to each other.
- **reference** The operator `reshape` takes an array, an array shape S, and an affine-linear transformation T from index tuples to index tuples, and returns an array of shape S, where each element with index tuple I has the same value as the entry of A with index tuple $T(I)$.

A more elaborate description can be found in [25].

Features Petalisp differs significantly from established HPC practices and even code generators in that it relies exclusively on just-in-time compilation. For that, a lot of analysis, optimization, code generation, and compilation is taking place under the hood. But since our implementation can carry out all these tasks in just a few microseconds, we can hide this process from the user and pretend that Petalisp is a purely interpreted scripting language.

Another peculiarity about Petalisp is that it is an untyped programming language. No restrictions are made on what objects are permissible in an array. Thanks to runtime type inference, this will not slow down carefully written numerical programs.

Petalisp employs a variety of optimization techniques, e.g., constant folding, folding of consecutive affine-linear indirections, hoisting of loop-invariant code, loop reordering, and automatic shared-memory parallelization.

Layer Formulations We describe how Petalisp relates into the previously introduced layer model. It does not provide any facilities for mathematical modeling or automatic discretization, so the tasks on layers 1 and 2 must either be resolved manually, or by using existing Lisp libraries such as the computer algebra system Maxima. A Petalisp user typically starts on layer 3, by solving a discrete problem using a mixture of standard Common Lisp code and Petalisp lazy arrays. Any set of arrays can be considered a full program that is invoked by explicit evaluation of these arrays. In this sense, the closest thing that Petalisp has to a layer 4 description is the data flow graph that describes the construction of one or more arrays. The layers 5 and 6 are canonical, except that the intermediate representation is highly optimized for compilation speed.

For our benchmark example, the separation into layers means that the grid generation functions in figure 2, the smoother in figure 4, and the final V-cycle in figure 3 for the layer 3 specification of the problem. The lazy array that is the result of one or more multigrid iterations is the root of a very large data flow graph. This graph is the layer 4 specification of the problem to be solved. Due to its size, we cannot show the entire multigrid data flow graph. But, for illustration, we present the subgraph corresponding to a single smoothing step in figure 5.

```
(defun make-unit-square (N fn)
  (flet ((coord-from-index (index)
           (/ (coerce index 'double-float)
              (coerce (1- N) 'double-float))))
    (let* ((shape (~ 0 (1- N) ~ 0 (1- N)))
           (x (α #'coord-from-index (indices shape 1)))
           (y (α #'coord-from-index (indices shape 0))))
      (values (α fn x y) x y))))

(defun make-grid (N fn)
  (fuse* (make-unit-square N fn)
         (reshape 0d0 (~ 1 (- N 2) ~ 1 (- N 2)))))
```

Figure 2: Discrete domain functions in Petalisp.

```lisp
(defun v-cycle (u f h v1 v2)
 (if (<= (range-size (first (shape-ranges (shape u)))) 3)
   (rbgs u f h 3) ; solve "exactly"
   (let* ((x (rbgs u f h v1))
          (r (restrict (residual x f h)))
          (c (v-cycle (reshape 0 (shape r)) r (* 2 h) v1 v2)))
     (rbgs (α #'+ x (prolongate c)) f h v2))))
```

Figure 3: V-cycle in Petalisp.

```lisp
(defun rbgs (u f h &optional (iterations 1))
 (labels ((update (&rest spaces)
            (setf u (apply #'fuse* u spaces)))
          (stencil (S)
           (α #'* (float 1/4)
             (α #'+
               (reshape (reshape u (τ (i j) ((1+ i) j))) S)
               (reshape (reshape u (τ (i j) ((1- i) j))) S)
               (reshape (reshape u (τ (i j) (i (1+ j)))) S)
               (reshape (reshape u (τ (i j) (i (1- j)))) S)
               (reshape (α #'* (* h h) f) S)))))
  (trivia:ematch (shape u)
   ((~ a b ~ c d)
    (let ((red-1 (~ (+ a 1) 2 (1- b) ~ (+ c 2) 2 (1- d)))
          (red-2 (~ (+ a 2) 2 (1- b) ~ (+ c 1) 2 (1- d)))
          (black-1 (~ (+ a 1) 2 (1- b) ~ (+ c 1) 2 (1- d)))
          (black-2 (~ (+ a 2) 2 (1- b) ~ (+ c 2) 2 (1- d)))))
    (loop repeat iterations do
      (setf u (fuse* u (stencil red-1) (stencil red-2)))
      (setf u (fuse* u (stencil black-1) (stencil black-2))))
    u))))
```

Figure 4: Red-Black Gauß-Seidel smoother in Petalisp.

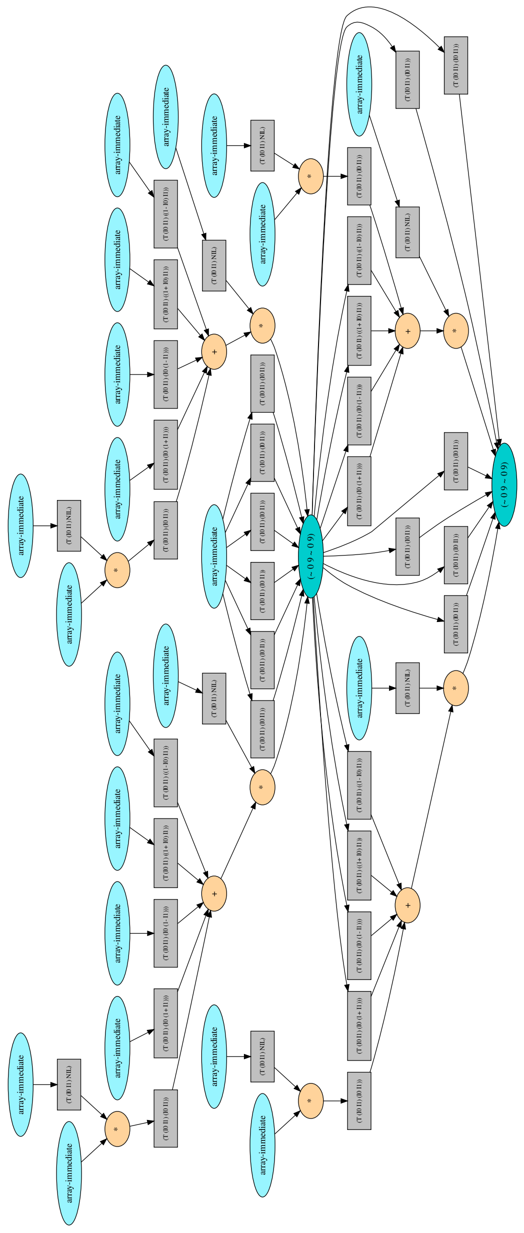

Figure 5: Data flow graph of one RBGS iteration in Petalisp, consisting of immediates (ellipses), references (boxes) and operations (circles).

Runtime Results Runtime results of using Petalisp to compute a multigrid V-cycle are shown in table 4. The shared-memory parallelization is based on operating system threads. We achieve 53 MDoF/s in 2D and 42 MDoF/s in 3D. Note that two crucial optimizations – vectorization and locality optimization in space and time – have not yet been implemented. Furthermore, the automatic parallelization is very recent work and does not yet fully saturate all cores.

Dim	DoFs	mesh width	Time / Cycle [s] Num cores 1	8
2D	$1.0 \cdot 10^6$	2^{-10}	0.07	0.07
	$4.2 \cdot 10^6$	2^{-11}	0.24	0.13
	$1.7 \cdot 10^7$	2^{-12}	0.95	0.32
3D	$2.0 \cdot 10^6$	2^{-7}	0.16	0.11
	$1.7 \cdot 10^7$	2^{-8}	1.59	0.40

Table 4: Performance results for multigrid (V(2,1)-cycle, RGBS) implementations on Petalisp.

Nevertheless, we see that for the serial case the execution time of Petalisp is just about a factor of four slower than an equivalent, optimized C++ program generated by the ExaStencils framework. Petalisp generates and compiles all its code at runtime and that the generated kernels are also written in Common Lisp. This teaches two valuable lessons. The first lesson is that it is feasible to generate and compile whole programs at runtime on modern computers. The second lesson is that modern Common Lisp compilers can be used to generate HPC code.

3.6 Discussion

We discuss advantages and disadvantages of the three presented approaches, namely external DSLs, internal DSLs and HPC frameworks with generated add-ins, regarding performance, portability and productivity.

In terms of performance, ExaStencils and HyTeG are able to deliver competitive results and to exhaust a good portion of the available hardware capabilities, at least for the examined benchmark problem. The Petalisp approach is promising, but still under development.

Portability to other hardware platforms is straight-forward for users of external DSLs since no changes to the implementation in the DSL are necessary. The same holds for all generated portions of the other two approaches, whereas caution and potentially more work is required for the remaining code or framework parts. For the developers of an embedded or external DSL adding a new platform requires to write potentially adapted code transforms and an additional backend in case of a different architecture. For framework developers the underlying code has to be extended.

Our last metric, productivity, is the hardest to quantify since it strongly depends on the users or developers prior knowledge and abilities. For users, learning a new language, even a compact one such as an DSL, can be challenging. This can be remedied to some extend by using internal DSLs, but only if the user is already proficient with the host language, which can be an issue when using less prevalent ones such as Lisp.

In the case of frameworks, users need to implement in two different languages as well, but since both of them are general purpose languages chances are high that users are already familiar with them.

For developers, DSLs are more demanding than classical frameworks, since they have to implement code transforms to manipulate certain representations on different levels of abstraction instead of directly implementing algorithms. One interesting point here is the amount of work that is required, if we implement a problem outside the originally intended scope. In our opinion, this is easiest in the framework case, already quite hard when using embedded DSLs, and virtually impossible for most cases when using external DSLs. If one regards the separation of concerns as an important feature, the multi-layered external DSL is beneficial since it explicitly defines different layers of abstraction. For the other two approaches part of the layers is contained in the DSL constructs and part in the underlying Lisp resp. C++ code.

4 Summary and Future Work

Currently, there is a trend to enhance the traditional way of implementing simulation codes to increase portability and productivity while at the same time keeping the expected high performance.

We have studied three different code generation approaches, an external DSL, an internal DSL, and an extended framework, which try to achieve this goal, and applied them to create a geometric multigrid solver for elliptic partial differential equations. With respect to performance code generation can compete with classical hand-tuned implementations. For portability and productivity, it depends on the role. Users profit from higher levels of abstractions and hardware-independent descriptions, but have to learn a new language for that. Developers can avoid code duplication, e.g. when switching to another platform or a similar algorithm, but need to have a deeper understanding of compiler techniques and potentially have to work with a more complex toolchain involving several programming languages.

In the next years, one can expect that the best of the arising approaches will be used and integrated by the framework and application developers. In case of ExaStencils, a Python frontend for ExaSlang will be provided to attract more users. Besides that, ExaStencils is already extending his own domain by, e.g., allowing further more general block-structured grid types. Tools like *pystencils* can be fully integrated also in other HPC frameworks. This enables the framework user to work more and more in Python instead of C++. PetaLisp is still ongoing research, but it already shows high potential for just-in-time compilation. However, it has to prove its efficiency and scalability on large distributed-memory machines, and since it requires knowledge of Common Lisp up to now, we are also working on a user-friendly Python interface to it.

5 Acknowledgements

This research has been funded partly by the Federal Ministry of Education and Research of Germany (BMBF) through the HPC2SE project and partly by the Deutsche Forschungsgemeinschaft (DFG) in its priority programme SPP 1648 “Software for Exascale Computing” through the projects TerraNeo and ExaStencils.

References

[1] M. Adams, P. Colella, D. T. Graves, J. N. Johnson, Keen, N. D., T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, P. Schwartz, T. Sternberg, and B. van Straalen. Chombo software package for AMR applications - design document. Technical Report LBNL-6616E, Lawrence Berkeley National Laboratory, Jan 2015.

[2] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[3] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

[4] P. Bastian, C. Engwer, D. Göddeke, O. Iliev, O. Ippisch, M. Ohlberger, S. Turek, J. Fahlke, S. Kaulmann, S. Müthing, and D. Ribbrock. EXA-DUNE: Flexible pde solvers, numerical methods and applications. In

Euro-Par 2014: Parallel Processing Workshops, volume 8806 of *Lecture Notes in Computer Science*, pages 530–541. Springer, 2014.

[5] M. Bauer, F. Schornbaum, C. Godenschwager, M. Markl, D. Anderl, H. Köstler, and U. Rüde. A python extension for the massively parallel multiphysics simulation framework walberla. *International Journal of Parallel, Emergent and Distributed Systems*, 31(6):529–542, 2016.

[6] B. Bergen, T. Gradl, F. Hülsemann, and U. Rüde. A massively parallel multigrid method for finite elements. *Computing in Science and Engineering*, 8(6):56–62, 2006.

[7] B. Bergen and F. Hülsemann. Hierarchical hybrid grids: data structures and core algorithms for multigrid. *Numer. Linear Algebra Appl.*, 11:279–291, 2004.

[8] M. Blatt, A. Burchardt, A. Dedner, C. Engwer, J. Fahlke, B. Flemisch, C. Gersbacher, C. Gräser, F. Gruber, C. Grüninger, D. Kempf, R. Klöfkorn, T. Malkmus, S. Müthing, M. Nolte, M. Piatkowski, and O. Sander. The distributed and unified numerics environment, version 2.4. *Archive of Numerical Software*, 4(100):13–29, 2016.

[9] M. Bolten, F. Franchetti, P. H. J. Kelly, C. Lengauer, and M. Mohr. Algebraic description and automatic generation of multigrid methods in SPIRAL. *Concurrency and Computation: Practice and Experience*, 29(17):4105:1–4105:11, 2017. Special Issue on Advanced Stencil-Code Engineering.

[10] T. Brandvik and G. Pullan. SBLOCK: A framework for efficient stencil-based PDE solvers on multi-core platforms. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1181–1188, Jun 2010.

[11] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 676–687, May 2011.

[12] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: Co-array Fortran and unified parallel C. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM.

[13] Z. DeVito, N. Joubert, F. Palaciosy, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12. ACM, 2011.

[14] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Special issue on Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[15] R. D. Falgout, J. E. Jones, and U. M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, pages 267–294, Berlin, Heidelberg, 2006. Springer.

[16] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[17] K. Fürlinger, C. Glass, A. Knüpfer, J. Tao, D. Hünich, K. Idrees, M. Maiterth, Y. Mhedheb, and H. Zhou. DASH: Data structures and algorithms with support for hierarchical locality. In *Euro-Par 2014 Workshops (Porto, Portugal)*, pages 542–552, 2014.

[18] B. Gmeiner, T. Gradl, H. Köstler, and U. Rüde. Highly parallel geometric multigrid algorithm for hierarchical hybrid grids. In K. Binder, G. Münster, and M. Kremer, editors, *NIC Symposium 2012*, volume 45 of *Publication series of the John von Neumann Institute for Computing*, pages 323–330, Jülich, Germany, 2012.

[19] B. Gmeiner, M. Huber, L. John, U. Rüde, and B. Wohlmuth. A quantitative performance study for Stokes solvers at the extreme scale. *J. Comput. Sci.*, 17(3):509–521, 2016.

[20] B. Gmeiner, H. Köstler, M. Stürmer, and U. Rüde. Parallel multigrid on hierarchical hybrid grids: a performance study on current high performance computing clusters. *Concurrency and Computation: Practice and Experience*, 26(1):217–240, 2014.

[21] B. Gmeiner, U. Rüde, H. Stengel, C. Waluga, and B. Wohlmuth. Performance and Scalability of Hierarchical Hybrid Multigrid Solvers for Stokes Systems. *SIAM J. Sci. Comput.*, 37(2):C143–C168, 2015.

[22] B. Gmeiner, U. Rüde, H. Stengel, C. Waluga, and B. Wohlmuth. Towards textbook efficiency for parallel multigrid. *Numer. Math. Theory Methods Appl.*, 8:2246, 2015.

[23] T. Gysi, T. Grosser, and T. Hoefler. MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 177–186, New York, NY, USA, 2015. ACM.

[24] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess. STELLA: A domain-specific tool for structured grid methods in weather and climate models. In *Proceedings International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 41:1–41:12. ACM, Nov 2015.

[25] M. Heisig. Petalisp: A common lisp library for data parallel programming. In *11th European Lisp Symposium*, page 4, 2018.

[26] M. Heisig and H. Köstler. Petalisp: run time code generation for operations on strided arrays. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, pages 11–17. ACM, 2018.

[27] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[28] L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. *SIGPLAN Notices*, 28(10):91–108, Oct 1993.

[29] N. Kohl, D. Thönnes, D. Drzisga, D. Bartuschat, and U. Rüde. The hyteg finite-element software framework for scalable multigrid solvers. *International Journal of Parallel, Emergent and Distributed Systems*, 0(0):1–20, 2018.

[30] H. Köstler, C. Schmitt, S. Kuckuk, F. Hannig, J. Teich, and U. Rüde. A scala prototype to generate multigrid solver implementations for different problems and target multi-core platforms. *Int. J. of Computational Science and Engineering*, 14(2):150–163, 2017.

[31] H. Köstler, M. Stürmer, and T. Pohl. Performance engineering to achieve real-time high dynamic range imaging. *Journal of Real-Time Image Processing*, pages 1–13, 2013.

[32] S. Kronawitter, S. Kuckuk, H. Köstler, and C. Lengauer. Automatic data layout transformations in the exastencils code generator. *Modern Physics Letters A*, 28(03):1850009, 2018.

[33] S. Kronawitter, S. Kuckuk, H. Köstler, and C. Lengauer. Automatic data layout transformations in the ExaStencils code generator. *Parallel Processing Letters*, 28(03):1850009, 2018.

[34] S. Kronawitter, S. Kuckuk, and C. Lengauer. Redundancy elimination in the ExaStencils code generator. In *Algorithms and Architectures for Parallel Processing*, pages 159–173, Cham, 2016. Springer International Publishing.

[35] S. Kuckuk, G. Haase, D. A. Vasco, and H. Köstler. Towards generating efficient flow solvers with the ExaStencils approach. *Concurrency and Computation: Practice and Experience*, 29(17):4062:1–4062:17, 2017. Special Issue on Advanced Stencil-Code Engineering.

[36] S. Kuckuk and H. Köstler. Automatic generation of massively parallel codes from ExaSlang. *Computation*, 4(3):27:1–27:20, 2016. Special Issue on High Performance Computing (HPC) Software Design.

[37] S. Kuckuk and H. Köstler. Whole program generation of massively parallel shallow water equation solvers. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 78–87, Sept 2018.

[38] S. Kuckuk and H. Kstler. Automatic generation of massively parallel codes from exaslang. *Computation*, 4(3):27:1–27:20, 2016. Special Issue on High Performance Computing (HPC) Software Design.

[39] S. Kuckuk, L. Leitenmaier, C. Schmitt, D. Schönwetter, H. Köstler, and D. Fey. Towards virtual hardware prototyping for generated geometric multigrid solvers. Technical Report CS 2017-01, Technische Fakultät, 2017.

[40] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, et al. Exastencils: Advanced stencil-code engineering. In *European Conference on Parallel Processing*, pages 553–564. Springer, 2014.

[41] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, and C. Schmitt. ExaStencils: Advanced stencil-code engineering. In L. Lopes et al., editors, *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science (LNCS)*, pages 553–564. Springer, 2014.

[42] A. Logg, K.-A. Mardal, and G. N. Wells. *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering (LNCSE)*. Springer, 2012.

[43] N. Maruyama, K. Sato, T. Nomura, and S. Matsuoka. Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2011.

[44] G. R. Mudalige, I. Reguly, M. B. Giles, C. Bertolli, and P. H. J. Kelly. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *Proc. Innovative Parallel Computing (InPar)*, San Jose, California, May 2012. IEEE.

[45] G. Ofenbeck, T. Rompf, and M. Püschel. Staging for generic programming in space and time. *SIGPLAN Not.*, 52(12):15–28, Oct 2017.

[46] M. Püschel, F. Franchetti, and Y. Voronenko. *Spiral*, volume 4, pages 1920–1933. Springer, 2011.

[47] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly. Firedrake: Automating the finite element method by composing abstractions. *ACM Trans. on Mathematical Software (TOMS)*, 43(3):24:1–24:27, 2016.

[48] P. Rawat, M. Kong, T. Henretty, J. Holewinski, K. Stock, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. SDSLc: A multi-target domain-specific compiler for stencil computations. In *Proc. 5th Int'l Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 6:1–6:10. ACM, Nov 2015.

[49] C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich. ExaSlang: A domain-specific language for highly scalable multigrid solvers. In *Proc. 4th Int'l Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 42–51. IEEE Computer Society, Nov. 2014.

[50] C. Schmitt, M. Schmid, F. Hannig, J. Teich, S. Kuckuk, and H. Köstler. Generation of multigrid-based numerical solvers for FPGA accelerators. In *Proc. 2nd Int'l Workshop on High-Performance Stencil Computations (HiStencils)*, pages 9–15, Jan. 2015.

[51] C. Schmitt, M. Schmid, S. Kuckuk, H. Köstler, J. Teich, and F. Hannig. Reconfigurable hardware generation of multigrid solvers with conjugate gradient coarse-grid solution. *Parallel Processing Letters*, 28(04):1850016, 2018.

[52] J. Schmitt, H. Köstler, J. Eitzinger, and R. Membarth. Unified code generation for the parallel computation of pairwise interactions using partial evaluation. In *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 17–24. IEEE, 2018.

[53] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 117–128. ACM, 2011.

[54] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, San Diego, CA, USA, 2001.

[55] A. Vogel, S. Reiter, M. Rupp, A. Nägel, and G. Wittum. UG 4: A novel flexible software system for simulating pde based models on high performance computers. *Computing and Visualization in Science*, 16(4):165–179, 2013.

[56] T. Weinzierl. The peano softwareparallel, automaton-based, dynamically adaptive grid traversals. *ACM Transactions on Mathematical Software (TOMS)*, 45(2):14, 2019.