

# Towards generating efficient flow solvers with the ExaStencils approach

Sebastian Kuckuk<sup>1\*</sup>, Gundolf Haase<sup>2</sup>, Diego A. Vasco<sup>3</sup>, Harald Köstler<sup>1</sup>

<sup>1</sup>*Friedrich-Alexander-Universität Erlangen-Nürnberg,  
Department of Computer Science 10 (System Simulation)*

<sup>2</sup>*Karl-Franzens Universität Graz, Institute for Mathematics and Scientific Computing*

<sup>3</sup>*Universidad de Santiago de Chile, Departamento de Ingeniería Mecánica*

## SUMMARY

ExaStencils aims at providing intuitive interfaces for the specification of numerical problems and resulting solvers, particularly those from the class of (geometric) multigrid methods. It envisions a multi-layered domain-specific language (DSL) and a sophisticated code generation framework ultimately emitting source code in a chosen target language. We present our recent advances in fully generating solvers applied to three-dimensional fluid mechanics for non-isothermal/non-Newtonian flows. In detail, a system of time-dependent, non-linear partial differential equations (PDEs) is discretized on a cubic, non-uniform and staggered grid using finite volumes. We examine the contained problem of coupled Navier-Stokes and temperature equations, which are linearized and solved using the SIMPLE algorithm and geometric multigrid solvers, as well as the incorporation of non-Newtonian properties. Furthermore, we provide details on necessary extensions to our DSL and code generation framework, in particular those concerning the handling of boundary conditions, support for non-equidistant staggered grids and supplying specialized functions to express operations reoccurring in the scope of finite volume discretizations. Many of these enhancements are generalizable and thus suitable for utilization in similar projects. Lastly, we demonstrate the applicability of our code generation approach by providing convincing performance results for fully generated and automatically parallelized solvers. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Domain-specific languages, code generation, multigrid methods, high-performance computing, computational fluid dynamics

## 1. INTRODUCTION

Partial differential equations (PDEs) are ubiquitous in many application fields. Traditionally, some of the most efficient and widely used methods to solve discretized PDEs are those from the class of multigrid methods [1, 2]. However, composing and tuning such a solver is highly non-trivial and usually dependent on a diverse group of parameters ranging from the actual problem description to the target hardware platform. To attain high performance, scalability and performance portability, new code generation techniques can be used in conjunction with domain-specific languages (DSLs), which provide the means of specifying salient features in an abstract fashion. ExaStencils<sup>†</sup> aims at realizing this vision for the domain of geometric multigrid solvers.

Many (real-world) applications are conceivable. One that is just as relevant as it is challenging is given by the application of simulating non-Newtonian fluids. Even without taking the extended use

\*Correspondence to: sebastian.kuckuk@fau.de

<sup>†</sup><http://www.exastencils.org>

case of non-Newtonian behavior into account, solving the underlying coupled fluid and temperature equations is non-trivial. Here, we present our recent accomplishments in generating highly optimized geometric multigrid solvers for this application. For this, we employ a finite volume discretization on a staggered grid with varying grid spacing which is solved using the SIMPLE algorithm [3, 4]. SIMPLE stands for Semi-Implicit Method for Pressure Linked Equations. It is a guess-and-correct procedure for the calculation of pressure on a staggered grid arrangement. The generated implementation is already OpenMP parallel and includes models for the incorporation of non-Newtonian properties.

The remainder of the paper is structured as follows: We briefly present related projects and approaches in section 2. Afterward, we give an overview of relevant concepts of ExaStencils and in particular its DSL ExaSlang in section 3. Next, the problem description is given in section 4, followed by our numerical solution approach in section 5. Since this application relies on some specialized concepts, extensions to the DSL as well as the code generator are highly beneficial. They are described in detail in section 6. A discussion of required coding work and a comparison with internal DSLs as well as libraries is given in section 7. Using the presented extensions, we generate corresponding solvers and present performance results in section 8. Lastly, we give a short conclusion and outlook in section 9.

## 2. RELATED WORK

Due to the nature of the regarded class of applications, many approaches for treating it have been developed. In the domain of scientific libraries and frameworks, many support multigrid solvers. Among the most prominent examples are the *hypr* software package [5], mainly focusing on numerical solvers and preconditioners, *Boomer AMG* [6], which is a part of *hypr*, for unstructured grids and general matrices, *Peano* [7], which is based on space-filling curves, and *DUNE* [8], which is a general software framework centered around the solution of PDEs featuring the *DUNE AMG* solver. More important frameworks from this area are *Trilinos* [9, 10], which focuses on solving large-scale, complex multi-physics problems, and *PETSc* [11], the portable, extensible toolkit for scientific computation.

In recent years, DSLs and DSL-like languages have constantly gained attention and importance as possible alternatives to library-based approaches. In the (broader) domain of stencil computations, a multitude of compilers and languages have been developed. Examples include *Mint* [12] and *STELLA* (STencil Loop Language) [13] which are DSLs embedded in C and C++, respectively. Both are geared towards stencil operations performed on structured grids and both are able to target accelerators using CUDA. *STELLA* additionally supports shared and distributed memory parallelizations, albeit the latter is restricted to the  $x$  and  $y$  dimensions for 3D problems. Additional projects are *PATUS* [14], which uses auto-tuning techniques to improve performance, *Pochoir* [15], which is built on top of the parallel C extension Cilk aiming at making stencil computations cache-oblivious, and *Liszt* [16], which handles stencil computations for unstructured problems by adding abstractions to Java. To our knowledge, neither of these latter projects provide language support for multigrid methods. *HIPAcc* [17] is a DSL centered around image processing applications. Based on the Clang/LLVM compiler infrastructure, it is capable of generating OpenCL and CUDA code from kernel specifications embedded into C++. In recent work, support for image pyramids was added [18], a kind of multi-resolution data structure which allows for operations similar to the ones used in multigrid methods. However, support for 3D problems and distributed-memory parallelizations based on, e.g., MPI is not available.

In the realm of finite element methods, *FEniCS* [19] features a DSL embedded in Python called unified form language (UFL). While *FEniCS* is mainly focused on providing discretizations, support for multigrid solvers is available via the *PETSc* library [11]. It supports parallelization based on pthreads and MPI as well as execution on GPUs.

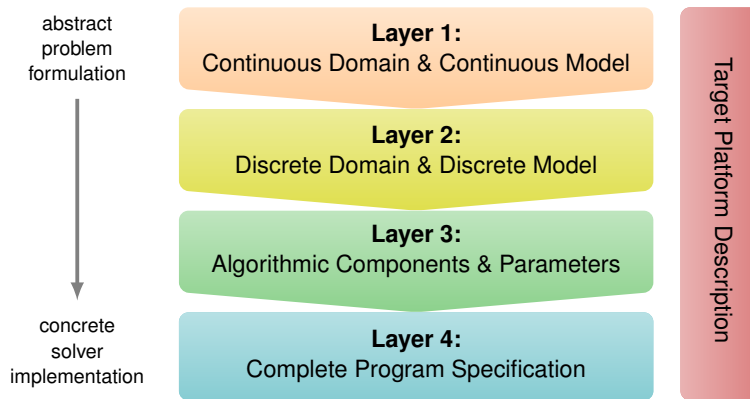


Figure 1. ExaSlang consists of four layers of abstraction [21]. Orthogonal to the algorithm specification, a description of the target hardware is provided to generate optimal code.

### 3. EXASTENCILS AND EXASLANG OVERVIEW

ExaStencils aims at generating highly efficient geometric multigrid solvers from an abstract description specified in its DSL ExaSlang [20, 21]. As illustrated in figure 1, ExaSlang is conceptualized as a multi-layered external DSL, where each layer addresses a different user group. In detail, the description of the continuous model is supposed to be provided by the application scientist (Layer 1). Mathematicians can review and adapt the discretized counterpart (Layer 2) and the algorithmic layout (Layer 3). The complete specification at Layer 4 is mostly relevant for computer scientists. Since Layer 4 is currently the most developed, it is also our focal point here. While a more thorough discussion of its features is given in [21], we still want to highlight the aspects most relevant for the application at hand.

**Control flow** Since ExaSlang’s Layer 4 is conceptually close to the target code, it makes sense to provide basic language elements computer scientists are accustomed to. These include loops, conditions and functions similar to the ones provided by, e.g., Scala, C++ and FORTRAN. Furthermore, certain basic mathematical functions, such as `sin`, `cos`, `exp` and `sqrt`, are available.

**Level specifications** ExaStencils, and consequently also ExaSlang, are geared towards solvers from the class of multigrid methods. One inherent concept in their scope is the utilization of multiple levels, with which data and functions are associated. We represent this concept in our DSL by allowing level assignments for most of the language objects discussed onwards. One example for this is given by listing 1. Here, the function `VCycle` is defined for all but the coarsest level of the multigrid hierarchy. Inside, the different steps of the algorithm are executed and the recursion is represented by calling the same function, albeit now at a different level. At the end of the recursion, the `VCycle` function is called at the coarsest level, which has a specialized instance implementing a designated coarse grid solver. As evident, the level index increases from coarser to finer levels. By default, the coarsest level has index 0 if not specified otherwise. Access to objects and functions at a specific level is provided via special identifiers such as `@current`, `@finer`, `@finest`, `@coarser` and `@coarsest`. For declaration purposes, single levels, lists of levels, e.g., `@(coarsest, 5, 6)`, ranges of levels, e.g., `@(0 to 5)`, or `@all` can be used. Furthermore, specific levels can be removed from sets of levels using `but`, e.g., `@(all but finest)`.

**Fields and layouts** Apart from the functions required to specify multigrid algorithms, representations of discretized variables, such as the unknown to be solved for, the residual and the right-hand side, are necessary. In ExaStencils, this is done through so-called *fields* and, in

```

1 Function VCycle@((coarsest + 1) to finest) ( ) : Unit {
2     repeat 3 times { Smoother@current ( ) }
3     UpdateResidual@current ( )
4     Restriction@current ( )
5     SetSolution@coarser ( 0 )
6     VCycle@coarser ( )
7     Correction@current ( )
8     repeat 2 times { Smoother@current ( ) }
9 }
10
11 Function VCycle@coarsest ( ) : Unit {
12     /* ... implementation of coarse grid solver ... */
13 }

```

Listing 1: Example of functions specifying a V-cycle. Solving at the coarsest level is implemented via a specialized coarse grid solver and by overloading the `VCycle` function at the corresponding multigrid level.

combination with them, *field layouts*. Broadly speaking, fields represent actual variables from, e.g., the PDEs while field layouts specify how associated data is stored. As listing 2 shows, fields are, just as most Layer 4 objects, level-specific in the sense described before. Furthermore, their declaration requires the specification of some parameters:

- a domain the field is tied to; this specifies the extents in physical coordinates and, in the case of multiple overlapping domains, on which processes memory is allocated and field operations are executed,
- a field layout, as described afterwards,
- boundary conditions, which may currently be `None`, `Neumann` or `Dirichlet` boundary conditions given by a constant value or an expression to be evaluated at the boundary, and
- optionally a number of slots, which describes how many instances of the field are required; slots facilitate the specification of, e.g., Jacobi-type updates or time stepping schemes.

In listing 2, the leveling mechanism is used again to implement some specialized behavior for one of the levels, in this case, the distinct boundary conditions at the finest level. Currently, all boundary conditions specified in that manner are translated to specialized routines setting or updating values at the boundary and/or in the ghost layers according to the type of boundary condition and data localization given by the field layout.

The memory layout required for the field declaration is specified in a similar fashion, this time providing the following options:

- the data type of the information stored at each of the discretization points,
- the localization of the data points, where currently `Cell` and `Node` are allowed for cell-centered and node-centered discretizations respectively, and
- specialized information about some regions of the field, e.g., ghost layers; a more complete description of regions is given in [22].

At the moment, combining multiple scalar values within one field is only possible if all values share the same localization. In this case, the data type of the field (layout) may be vector valued. Whether this field is represented as an array of structures (AoS) or a structure of arrays (SoA) in the generated code, however, is up to the compiler. Currently, arrays of structures are generated by default.

For easy interfacing with legacy codes, *external fields* can be used. As shown in listing 3, each external field is connected to a specific field layout describing how the data is stored in the external code parts as well as a (Layer 4) field it maps to. Based on this input, functions for copying data from and to external fields are set up automatically by our framework.

```

1 Domain global < [ 0.0, 0.0, 0.0 ] to [ 1.0, 1.0, 1.0 ] >
2
3 Layout DefNodeLayout < Real , Node >@all {
4   ghostLayers = [ 1, 1, 1 ] with communication
5 }
6
7 Field Solution < global, DefNodeLayout, 0.0 >[2]@(all but finest)
8 Field Solution < global, DefNodeLayout,
9   sin ( PI * vf_nodePosition_x@current ) >[2]@finest

```

Listing 2: Example of defining field layouts and using them to declare fields. The `Solution` field is given different boundary conditions on the finest level. Boundary conditions use virtual fields as described in section 6.2.

```

1 Layout ExternalNodeLayout < Real , Node >@finest {
2   innerPoints = [ 65, 65, 65 ]
3   ghostLayers = [ 0, 0, 0 ]
4 }
5
6 external Field ext_Solution < ExternalNodeLayout > => Solution@finest

```

Listing 3: Example for the declaration of an external field with  $65^3$  data points. Mapping to `Solution@finest` triggers generation of functions to copy from and to it.

```

1 Stencil Laplace@all {
2   [ 0, 0 ] => 2.0 / ( hx@current * hx@current )
3             + 2.0 / ( hy@current * hy@current )
4   [ 1, 0 ] => -1.0 / ( hx@current * hx@current )
5   [-1, 0 ] => -1.0 / ( hx@current * hx@current )
6   [ 0, 1 ] => -1.0 / ( hy@current * hy@current )
7   [ 0,-1 ] => -1.0 / ( hy@current * hy@current )
8 }

```

Listing 4: Sample specification of a stencil with pre-defined stencil coefficients. `hx` and `hy` have been initialized to hold the grid spacings per multigrid level. This example assumes a uniform grid. Alternatively, virtual fields as described in section 6.2 can be used.

**Stencils** The last component necessary for specifying simple solvers is a representation of discretized operators, i.e., matrices. For this, there are basically two options. If the matrix is expressible as a *stencil* with constant coefficients, it can be defined directly in ExaSlang 4. Apart from constant coefficients, simple expressions can be used as stencil coefficients as well, as demonstrated in listing 4. Ultimately, these calculations are simplified, if possible, and inlined when convolutions with stencils are resolved in the code generator. Since this might introduce some serious overhead, pre-calculation and storage in *stencil fields* is possible. Broadly speaking, stencils can be viewed as classical constant coefficient stencils while stencil fields represent the variable coefficient case. One use-case where stencil fields are highly desirable is the restriction of operators to coarser meshes, which is required if rediscrretization is not possible or desired (Galerkin approach). As shown in listing 5, stencil fields are set up by combining the storage of an (array) field with the usage information of a stencil, which in turn acts as a template on how to use the data stored in the field. In terms of using stencils and stencil fields in solver components such as the smoother, both behave essentially equally, as depicted in listing 6.

```

1 Domain global< [ 0.0, 0.0 ] to [ 1.0, 1.0 ] >
2
3 // definition of the field layout for the stencil field
4 Layout DefStencilLayout < Real[5], Node >@all {
5   ghostLayers = [ 0, 0 ]
6 }
7
8 // definition of the stencil template
9 Stencil LaplaceTemplate@all {
10  [ 0, 0 ] =>
11  [ 1, 0 ] =>
12  [ -1, 0 ] =>
13  [ 0, 1 ] =>
14  [ 0, -1 ] =>
15 }
16
17 // creation of a vector field to hold the stencil coefficients
18 Field LaplaceStencilData< global, DefStencilLayout, None >@all
19
20 // combination of vector field and stencil view
21 // the stencil field can now be used just as a stencil
22 StencilField Laplace< LaplaceStencilData => LaplaceTemplate >@all
23
24 // function initializing the stencil coefficients
25 Function InitStencil@all {
26   loop over Laplace {
27     Laplace@current:[ 0, 0 ] =
28       2.0 / ( hx@current * hx@current )
29       + 2.0 / ( hy@current * hy@current )
30       + kappa@current
31     Laplace@current:[ 1, 0 ] =
32       -1.0 / ( hx@current * hx@current )
33     Laplace@current:[ -1, 0 ] =
34       -1.0 / ( hx@current * hx@current )
35     Laplace@current:[ 0, 1 ] =
36       -1.0 / ( hy@current * hy@current )
37     Laplace@current:[ 0, -1 ] =
38       -1.0 / ( hy@current * hy@current )
39   }
40 }

```

Listing 5: Sample specification of a stencil field and its initialization. The coefficients are stored in `LaplaceStencilData`, applied as defined by the stencil template `LaplaceTemplate` and initialized by the function `InitStencil`. As in listing 4, `hx` and `hy` have been initialized to hold the grid spacings per multigrid level. Alternatively, virtual fields as described in section 6.2 can be used.

```

1 Function UpdateResidual@all ( ) : Unit {
2   loop over Residual@current {
3     Residual@current = RHS@current
4     - ( Laplace@current * Solution@current )
5   }
6 }

```

Listing 6: Sample usage of a stencil (listing 4) or stencil field (listing 5) in a stencil convolution.

#### 4. VISCOPLASTIC NON-NEWTONIAN FLUIDS

Viscoplastic fluids are those non-Newtonian fluids characterized by a yield-stress, defined as a threshold after which a fluid readily flows [23, 24]. In general, yield-stress fluids are suspensions

of particles or macromolecules, such as pastes, gels, foams, drilling fluids, food products and nanocomposites. Processes with viscoplastic fluids are of great importance in mining, chemical and food industry. For instance, several efforts have demonstrated that rheological properties of fruit juices [25, 26], mining pulps [27, 28] and nanofluids [29, 30] are described well by viscoplastic non-Newtonian models such as Herschel-Bulkley, Bingham and Casson. Nanofluids are colloidal dispersions of nanometric-sized (<100 nm) metallic or non-metallic particles in a base fluid. The addition of nanoparticles improves the thermal conductivity and increases the viscosity of the fluid. Nanofluids containing spherical nanoparticles are more likely to exhibit Newtonian behavior and those containing nanotubes show non-Newtonian behavior. Furthermore, nanofluids show non-Newtonian behavior at higher shear rate values [31]. The applications of interest in our numerical experiments are related to the flow of Bingham plastic fluids due to changes in the buoyancy forces caused by thermal effects and mixed convection effects. It has been found that under certain conditions water suspensions with nanoparticles such as SiO<sub>2</sub>/TiO<sub>2</sub> [29], BaTiO<sub>3</sub> [32], ITO [33] behave as Bingham fluids. The presentation of the application problem follows closely the description in [34] and we will focus especially on the natural convection example therein.

The formal mathematical description of the coupled problem is similar to the Navier-Stokes equations for incompressible fluids together with an additional equation regarding the (scaled) temperature  $\theta$ , an additional term in the Navier-Stokes equations and non-linear material coefficients regarding the non-Newtonian fluid behavior. Let us combine the three velocity components in one velocity column vector  $\vec{v} = (U, V, W)^T$  and the gradient operator  $\nabla$  is also considered as a column vector. Then, we get the steady-state flow equations

$$-\nabla^T (H(\dot{\Gamma})\nabla\vec{v}) + D \cdot (\vec{v}^T \cdot \nabla) \vec{v} + \nabla p \quad + D \begin{bmatrix} 0 \\ \theta \\ 0 \end{bmatrix} = 0 \quad (1)$$

$$\nabla^T \vec{v} = 0 \quad (2)$$

$$-\nabla^T (\nabla\theta) + G \cdot (\vec{v}^T \cdot \nabla) \theta = 0 \quad (3)$$

with

$$G(\theta) = \sqrt{Ra(\varrho(\theta)) \cdot Pr} \quad (4)$$

$$D(\theta) = \sqrt{\frac{Ra(\varrho(\theta))}{Pr}} \quad \text{with} \quad (5)$$

$$Ra(\varrho(\theta)) = \frac{\varrho(\theta)g\beta(T_H - T_C)L^3}{\eta_0\alpha} \quad (6)$$

$$Pr = \frac{\eta_0 C_p}{\lambda} \quad (7)$$

$$H(\dot{\Gamma}(\vec{v})) = \frac{\eta(\dot{\Gamma}(\vec{v}))}{\eta_0} \quad (8)$$

and the nomenclature from table I.

$\dot{\Gamma}(\vec{v})$  is related to the strain tensor and the viscosity  $H(\dot{\Gamma}(\vec{v}))$  is computed with respect to the Bingham model, which in its dimensional and tensorial form can be expressed as:

$$\bar{\dot{\gamma}} = 0 \quad \text{for } |\tau| \leq \tau_y \quad (9)$$

$$\bar{\tau} = \left(\frac{\tau_y}{\dot{\gamma}} + \eta_0\right) \dot{\gamma} \quad \text{for } |\tau| > \tau_y \quad (10)$$

The scalar  $\dot{\gamma}$  is a function of the second invariant of the strain tensor given by:

$$\dot{\gamma} = \sqrt{\left(\frac{1}{2}\bar{\dot{\gamma}} : \bar{\dot{\gamma}}\right)} \quad (11)$$

$C_p$	heat capacity	$\alpha$	thermal diffusivity
$g$	gravity acceleration	$\beta$	thermal expansion coefficient
$L$	characteristic length	$\Gamma$	diffusivity coefficient
$v$	velocity vector	$\dot{\gamma}$ ( $\dot{\Gamma}$ )	(dimensionless) deformation rate
$p$	pressure	$\tau$	stress tensor
$Ra$	Rayleigh number	$\varrho$	density
$Pr$	Prandtl number	$\eta$	viscosity of the fluid
$T_C$	cold temperature	$\eta_0$	plastic viscosity of the yielded fluid
$T_H$	hot temperature	$H$	dimensionless viscosity
$\lambda$	thermal conductivity	$\theta$	dimensionless temperature

Table I. Nomenclature of symbols

Broadly speaking, Bingham fluids behave as a Newtonian fluid under the influence of a shear stress higher than the yield stress ( $\tau > \tau_y$ ). When the yield-stress falls below  $\tau_y$  (unyielded region) a solid structure is formed. In the present work, the numerical implementation of the Bingham model is based on the bi-viscosity model [35], considering low values of  $\tau_y$  according to experimental observations [29].

## 5. NUMERICAL SOLUTION OF THE COUPLED PROBLEM

The coupled system of PDEs (1)-(3) is non-linear due to the material terms  $H, D, G$  that depend on the velocity  $\vec{v}$  and on the temperature  $\theta$ . Even in the case of constant material terms, the equations (1) and (2) represent the Navier-Stokes equations (non-linear because of  $\vec{v}^T \cdot \nabla$ ) with an additional term from the temperature. PDE (3) is only non-linear w.r.t. to the coupling via  $\vec{v}^T \cdot \nabla$ , not regarding the temperature  $\theta$ . The non-linear terms  $\mathcal{L}(x)$  are quasi-linear, i.e., they can be expressed as  $L(x) \cdot x$ . This allows the system of PDEs (1)-(3) to be rewritten as a block system with non-linear sub-blocks.

$$\begin{pmatrix} A(\theta, \vec{v}) & B & C(\theta) \\ B^T & 0 & 0 \\ 0 & 0 & T(\theta, \vec{v}) \end{pmatrix} \cdot \begin{pmatrix} \vec{v} \\ p \\ \theta \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \quad (12)$$

Fixing the temperature  $\theta^{\text{old}}$  we get the stationary Navier-Stokes equations

$$\begin{pmatrix} A(\theta^{\text{old}}, \vec{v}) & B \\ B^T & 0 \end{pmatrix} \cdot \begin{pmatrix} \vec{v} \\ p \end{pmatrix} = \begin{pmatrix} -C \cdot \theta^{\text{old}} \\ 0 \end{pmatrix} \quad (13)$$

and the additional temperature equation

$$T(\theta, \vec{v}) \cdot \theta = 0 \quad (14)$$

In our case it is sufficient to solve (13) and (14) in an alternating fashion. Thus, the next subsection concentrates on handling the Navier-Stokes system.

### 5.1. Linearization

The non-linear system (13) is transferred into a linear system by also using the old velocity  $\vec{v}^{\text{old}}$  from previous iterations in the operators. Operator  $A(\theta^{\text{old}}, \vec{v}^{\text{old}})$  from (13) can be expressed as a sum of diffusive and convective parts  $A^{\text{diff}}(\vec{v}^{\text{old}}) + A^{\text{conv}}(\theta^{\text{old}}, \vec{v}^{\text{old}})$ . The resulting Oseen equations

$$\begin{pmatrix} A^{\text{diff}}(\theta^{\text{old}}, \vec{v}^{\text{old}}) + A^{\text{conv}}(\theta^{\text{old}}, \vec{v}^{\text{old}}) & B \\ B^T & 0 \end{pmatrix} \cdot \begin{pmatrix} \vec{v} \\ p \end{pmatrix} = \begin{pmatrix} -C \cdot \theta^{\text{old}} \\ 0 \end{pmatrix} \quad (15)$$

can be solved similarly as the Stokes problem

$$\begin{pmatrix} A^{\text{diff}}(\theta^{\text{old}}, \vec{v}^{\text{old}}) & B \\ B^T & 0 \end{pmatrix} \cdot \begin{pmatrix} \vec{v} \\ p \end{pmatrix} = \begin{pmatrix} -A^{\text{conv}}(\theta^{\text{old}}, \vec{v}^{\text{old}}) \cdot \vec{v}^{\text{old}} - C \cdot \theta^{\text{old}} \\ 0 \end{pmatrix} \quad (16)$$



that considers only the symmetric part (the diffusion) in the system matrix. In any case we have to solve the saddle point problem

$$\begin{pmatrix} A & B \\ B^T & 0 \end{pmatrix} \cdot \begin{pmatrix} \vec{v} \\ p \end{pmatrix} = \begin{pmatrix} g \\ 0 \end{pmatrix} \quad (17)$$

with  $A$  and  $g$  according to (15) or (16).

### 5.2. Pressure correction scheme

A widely used approach to solve (17) takes into account the time dependent origin of the saddle point problem and assumes a known pressure approximation  $p^*$  such that the unknown pressure can be expressed as  $p = p^* + p'$ . This reformulates (17) into

$$\begin{pmatrix} A & B \\ B^T & 0 \end{pmatrix} \cdot \begin{pmatrix} \vec{v} \\ p' \end{pmatrix} = \begin{pmatrix} g - B \cdot p^* \\ 0 \end{pmatrix} \quad (18)$$

wherein the system matrix can be factorized

$$\begin{pmatrix} A & 0 \\ B^T & I \end{pmatrix} \cdot \begin{pmatrix} A^{-1} & 0 \\ 0 & C \end{pmatrix} \cdot \begin{pmatrix} A & B \\ 0 & I \end{pmatrix} \cdot \begin{pmatrix} \vec{v} \\ p' \end{pmatrix} = \begin{pmatrix} g - B \cdot p^* \\ 0 \end{pmatrix} \quad (19)$$

with  $C = -B^T A^{-1} B$ . This results formally in a sequence of three equations we have to solve.

$$\text{Solve} \quad A\vec{v}^* = r \quad [ := g - B \cdot p^* ] \quad (20)$$

$$\text{Solve} \quad Cp' = -B^T \cdot \vec{v}^* \quad (21)$$

$$A\vec{v} = r - B \cdot p' \quad [ \stackrel{(20)}{=} A \cdot \vec{v}^* - B \cdot p' ]$$

$$\text{Solve} \quad A \underbrace{(\vec{v} - \vec{v}^*)}_{\vec{v}'} = -B \cdot p' \quad (22)$$

With a virtual time step  $\Delta t$  we approximate the operator in (21) by  $C \approx -\Delta t \nabla^T \nabla$  (scaled discrete Poisson operator for the pressure) and in (22) the operator is approximated by its diagonal  $D = \text{diag}(A)$ . Finally we end up with the *SIMPLE algorithm* [3] for (18) containing the *pressure correction step* (24):

$$\text{Solve} \quad A\vec{v}^* = g - B \cdot p^* \quad (23)$$

$$\text{Solve} \quad -\Delta t \nabla^T \nabla p' = -B^T \cdot \vec{v}^* \quad (24)$$

$$\text{Solve} \quad \vec{v}' = -D^{-1} B \cdot p' \quad (25)$$

$$\text{Update} \quad \begin{pmatrix} \vec{v} \\ p \end{pmatrix} = \begin{pmatrix} \vec{v}^* \\ p^* \end{pmatrix} + \begin{pmatrix} \vec{v}' \\ p' \end{pmatrix} \quad (26)$$

## 6. EXASTENCILS EXTENSIONS

As a reference for the application described in section 4, we consider the work of Vasco et al. [36] and the source code described therein. It implements a solver for the simulation of non-isothermal and non-Newtonian fluids based on the SIMPLE algorithm in FORTRAN 90. The power-law differencing scheme of Patankar [4] is used to calculate the convective terms at the finite volume surfaces while the diffusion terms are determined by using linear interpolation functions for the dependent variables between the nodes of the staggered grids. The arising linear systems of equations (LSEs) for the single components are solved using TDMA solvers. The classical Thomas or tri-diagonal matrix algorithm (TDMA) is a direct method for one-dimensional problems. Moreover, it can be applied iteratively, in a line by line fashion, to solve, e.g., the problem at hand [37].

For our implementation, we decided to exchange the TDMA routines by geometric multigrid solvers. Furthermore, so far, ExaStencils has been focused mainly on finite difference discretizations on uniform grids. Consequently, suitable DSL and code generator extensions towards the newly introduced concepts have great potential. Please note that the DSL extensions described here are of prototypical nature and subject to change in future versions of our language. This applies especially to the syntax of specialized functions presented in section 6.2 which, in their current state, aim at exposing the required functionality rather than being final implementations.

### 6.1. Converting existing FORTRAN applications

In contrast to internal DSLs, we require specifying every relevant part of the application in our external DSL, that is, mixing with existing source code in C++ or FORTRAN is not possible. Consequently, rewriting the complete application from scratch is a highly non-trivial task. Thus, we decided to first give our generator the ability to automatically set up FORTRAN compliant interfaces to existing legacy code. For this, we automate established approaches for standard FORTRAN/C interfacing. This includes the adaptation of functions to make them callable from FORTRAN code and the generation of external field functions honoring the different data layout used by the FORTRAN counterparts. In case of the external fields, as described in section 3, this amounts to supporting multi-dimensional arrays as well as index order inversion. For the function adaption part, specialized transformations are applied. They target any function declaration that is marked as a possible interface function, usually all functions coming directly from the DSL and some additional special functions used for, e.g., initialization and deinitialization. The signature of those functions is automatically adapted such that function names follow a given specification. Depending on the compiler, this can, e.g., include converting names to lower case and ensuring that names end with an underscore. Additionally, all function parameters that are not of pointer or array data types are converted to follow call-by-reference semantics. Next, the function body is adapted accordingly by either resolving the newly introduced pointers at the beginning or by directly changing accesses to any altered function parameters. Lastly, all function calls targeting functions modified by this transformation are changed in a similar fashion. Here, the only special case is that of constant parameters which have to be assigned to actual (temporary) variables beforehand to allow for pointer arguments.

**Converting simple kernels** Using the described interfaces, the exchange of isolated parts of legacy applications with generated counterparts becomes possible. For simple kernels, setting up a representation in our DSL is straightforward, as illustrated in listing 7 and listing 8. Here, looping over certain fields automatically determines loop bounds depending on the underlying field layout. Furthermore, distributed and/or shared memory parallelization is added automatically. For convenient usage, the `rho` field has multiple slots that can be accessed through, e.g., the usage of the `active` and `next` keywords. Advancing to the next algorithmic step, in this case time step, is done via the `advance` statement.

**Boundary conditions** The main challenge is usually to convert parts of the code that do not optimally fit the concepts of a given DSL. This can occur when an optimization of code parts has already been attempted, making the code hard to understand, or when code was written for a specific use case such as serial or shared memory parallel execution. In particular, the implementation of boundary conditions is a frequent issue. To mitigate this problem, at least to some extent, we provide multiple options in our DSL. Firstly, as already described in section 3, simple and uniform boundary conditions can be specified intuitively at the field declaration stage. Here, Dirichlet boundary conditions can be specified by providing a constant value or an expression to be evaluated at the boundary. Additionally, Neumann boundary conditions are possible by setting `Neumann`. While this interface is straightforward and enables the user to easily specify the desired behavior, it is not sufficient for more complex problems. Considering, for instance, the typical test case of a lid driven cavity, specialized boundary conditions for each interface of the computational domain have to be taken into account. For this broader class of problems, we equip our DSL with specialized loops iterating over parts of the computational domain. As illustrated in listing 9, we allow targeting points

```

1 subroutine advance_fields ()
2 ! ...
3
4 !$omp parallel do &
5 !$omp private(i,j,k) firstprivate(l1,m1,n1) &
6 !$omp shared(rho,rho0) schedule(static) default(none)
7 do k=1,n1
8   do j=1,m1
9     do i=1,l1
10      rho0(i,j,k)=rho(i,j,k)
11    end do
12  end do
13 end do
14 !$omp end parallel do

```

Listing 7: Simplified OpenMP-parallel kernel for advancing the *rho* field to the next time step in FORTRAN.

```

1 Function AdvanceFields@finest ( ) : Unit {
2   loop over rho@current {
3     rho[next]@current = rho[active]@current
4   }
5   advance rho@current
6 }

```

Listing 8: Simplified kernel for advancing the *rho* field to the next time step in ExaSlang 4, corresponding to the FORTRAN snippet in listing 7. Possible parallelization through MPI and/or OpenMP is added implicitly.

inside specific regions of a field such as `duplicate` and `ghost`. Optionally, a direction can be specified to allow selecting only distinct interfaces and the modifier `on boundary` can be added to omit inner boundaries, that is boundaries between different fragments of the domain partition. Further information about the domain partitioning and regions can be found in [22]. To facilitate taking care of executing boundary handling kernels, it is also possible to summarize them in one function. As evident from listing 9, this function can then be registered with the field and executed through the common interface of `apply bc`. While these techniques already cover a pretty large set of possibilities, further simplifications and an even more intuitive interface are crucial in our opinion. In particular, a careful tuning towards currently developed concepts for ExaSlang Layer 1 through 3 is vital.

## 6.2. Special extensions for fluid solvers

Another specialty of the application at hand is the usage of specialized grids, similar to the one illustrated in figure 2. The main challenges lie in the varying grid widths and the use of staggered grids. In combination with the finite volume discretization, further extensions of our DSL and code generation framework are appropriate.

**Data layouts for staggered grids** With respect to the DSL, the specification of field layouts for staggered grids is rather straightforward. In addition to the `cell` and `node` options, as described in section 3, `face_x`, `face_y` and `face_z` are now allowed as well when defining the localization of values. Inside the code generation framework, an easy extension with regard to data handling and communication routines is possible due to the fact that layouts are treated separately for each dimension. Thus, face-centered layouts can be mapped to cell- and node-centered layouts for the non-staggered and staggered dimensions respectively. Other functions, such as e.g., boundary handling, may require a more in-depth adaption.

```

1 Function ApplyBC_u@finest ( ) : Unit {
2   loop over u@current only duplicate [ 1, 0] on boundary {
3     u@current = 0.0
4   }
5   loop over u@current only duplicate [-1, 0] on boundary {
6     u@current = 0.0
7   }
8   loop over u@current only duplicate [ 0, 1] on boundary {
9     u@current = wall_velocity
10  }
11  loop over u@current only duplicate [ 0, -1] on boundary {
12    u@current = -1.0 * wall_velocity
13  }
14 }
15
16 Field Solution < global, DefNodeLayout, ApplyBC_u >@finest

```

Listing 9: Kernel for handling user-defined boundary conditions. Specialized loops iterate over the edges of the computational boundary and set according values to implement a shear flow problem. The wrapping function is registered with the field and can be used by calling `apply bc` to `u@current`.

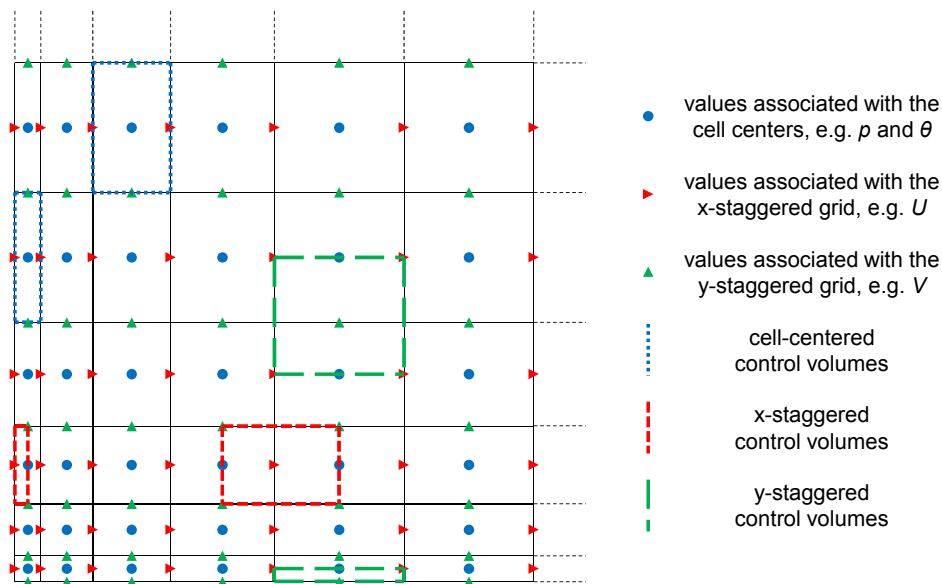


Figure 2. 2D illustration of the lower left part of a non-equidistant, staggered grid. Velocity components are associated with the centers of edges (resp. faces in 3D). Staggered control volumes get halved at the boundary.

**Additional geometric information** In the context of finite difference and finite volume discretizations, usually some geometric information is required to set up the corresponding stencils. In the most basic case of finite differences on uniform grids, this boils down to the grid widths per dimension and multigrid level. However, often more diverse parameters, such as cell widths, distance from the (staggered) cell center to a given (staggered) cell face or stagger distances, are required. In the case of uniform grids, according extensions are trivial.

**Geometric parameters for non-uniform grids** In addition to the increased number of geometric variables, the extension to varying grid widths and staggered grids requires additional features such as evaluation at the current index or at the position of neighboring indices. Previously, these parameters were regarded as constant per dimension and multigrid level in our framework and DSL. Access

```

1 // previous access through functions doesn't allow offset access
2 rhs@current = sin ( nodePosition_x@current ( ) )
3
4 // newly introduced virtual fields allow virtually identical behavior
5 rhs@current = sin ( vf_nodePosition_x@current )
6
7 // ... and in addition allow offset accesses
8 Val dif : Real = (
9     vf_nodePosition_x@current@[ 1, 0, 0]
10    - vf_nodePosition_x@current@[ 0, 0, 0] )

```

Listing 10: Example for using geometric information. Previously, specialized functions have been used. The current switch to virtual fields allows using offset semantics to access geometric properties of neighboring points

to them was enabled through specialized functions on Layer 4, as depicted in listing 10. Since this function mechanism does not intuitively support offset accesses, that is accesses to data of neighboring grid points, we decided to switch to specialized field accesses instead. Internally, these fields are handled as so-called virtual fields. In contrast to ‘real’ fields, the user is not responsible for specifying data layouts or for allocating the fields themselves. Instead, our framework is able to match corresponding accesses and resolve them accordingly. In the case of uniform grids, this means simply replacing them with constants such as the grid width of the current multigrid level or expressions to calculate, e.g., positions of grid nodes. For non-uniform grids, such as the one used here, the corresponding DSL code remains identical and all novel functionality is encapsulated in the code generator. Since target grids are axis aligned, it is sufficient to store geometric parameters per dimension, that is in edge-like structures represented by 1D arrays, for each fragment. However, many of these values can be expressed as calculations involving one or more other virtual fields. For instance, the width of a cell can be expressed as the difference of its adjacent nodal positions. This gives rise to possible optimizations by not storing all information explicitly but instead recalculating it on the fly. In our generator, both variants can be implemented, but which is to be applied for which virtual field is still an open question and most probably going to be an optimization parameter. For now, we decided to only store nodal positions of the original grid and calculate all other geometric information as necessary. One reason for this approach is a possible extension towards reading a grid from file for cases where a suitable setup routine for the geometric information cannot be generated. In this case, we want to keep the amount of information to be provided and accessed minimal. For finite volume discretizations on staggered grids we additionally store the respective widths of the staggered control volumes since an adaptation at the boundaries is required and we want to avoid branching during evaluation.

**Specialized evaluation functions** Due to the increased complexity of the computational domain, and the chosen discretization, the evaluation of expressions occurring in the setup of stencil coefficients also becomes increasingly complex. As a countermeasure, our DSL is equipped with specialized functions to encapsulate typical operations occurring in the context of finite volume discretizations and in particular those on staggered grids. One example are evaluation functions for simple expressions on distinct interfaces of control volumes. These features are illustrated in listing 11: a simple function call is used to specify what kind of evaluation should take place. For simple expressions, resolving is straightforward: our code generator checks the discretization type of the accessed field, i.e., cell-centered, node-centered or staggered in any dimension, and the specified point of evaluation. If the two can be aligned, the function call is replaced by a standard field access. Otherwise, an interpolation taking the geometric information of the grid into account is generated. Various interpolation schemes can be chosen through arguments to the original function call in the DSL, where the default is linear.

```

1 // evaluate with respect to cells of the grid
2 evalAtSouthFace ( rho[active]@current )
3
4 // integrate expressions across faces of grid cells
5 integrateOverEastFace (
6     u[active]@current * rho[active]@current )
7
8 // integrate expressions across faces of cells of the staggered grid
9 integrateOverXStaggeredEastFace (
10    u[active]@current * rho[active]@current )
11
12 // integrate expressions using specialized interpolation schemes
13 integrateOverZStaggeredEastFace (
14    evalAtZStaggeredEastFace ( vis@current, "harmonicMean" ) )

```

Listing 11: Examples of specialized functions newly added to our DSL. Evaluation at specific (relative) locations and integration of expressions over interfaces facilitates writing code for finite volume based discretizations.

When resolving integration functions, such as the ones illustrated in listing 11, more complex expressions are allowed as arguments. Conceptually, we require that the integration interval  $I$  is given by one interface of an arbitrary (staggered or non-staggered) control volume. Furthermore, we assume that values are constant with respect to matching control volume interfaces, that is, e.g., evaluation of non-staggered values yields a constant across interfaces of non-staggered control volumes. Consequently, integration can be expressed as a simple evaluation combined with a multiplication of the size of  $I$  in the case of non-staggered grids. In our case however, values to be integrated may change across  $I$  due to the staggered grid structure. When this happens, it is preferable to split  $I$  such that all occurring expressions are constant with respect to each resulting sub interval  $I_s$ .

In our generator, we map this concept following a three-step scheme. Firstly, all field accesses in the expression to be integrated are wrapped by evaluation functions. In case of field accesses already wrapped suitably, e.g., due to the user requiring a specific interpolation strategy for one of the values, we do nothing. Secondly, we compare the discretizations of involved fields and split  $I$  into  $I_s$  as required. In the simplest case, there will be only one sub interval, that is no actual split is performed. Thirdly, we duplicate the expression to be integrated and add a multiplication with the respective interval size for each  $I_s$ . Of course, this also requires adapting indices of field accesses and evaluation functions inside the duplicated expressions depending on the respective field's discretization and integration interval. Eventually, we replace the original function call with the compiled integration expression.

One possible application of these functions is given in listing 12. Here, evaluate and integrate functions are used to compute one component of the stencil used to solve for  $U$ . The calculations are performed for every (staggered) grid point. Afterwards, the stencil field is restricted to coarser levels. For reference, an example of C++ code emitted by our generator is shown in listing 13. It is evident, that the DSL specification is more concise, easier to understand and considerably easier to debug.

## 7. EXPERIENCES AND ASSESSMENT

In this section, we want to summarize the workflow when porting existing codes to our DSL and discuss the experiences gathered during the implementation of the presented extensions. Moreover, a comparison with internal DSLs and the more traditional approach of using large-scale libraries is drawn.

**Porting applications to ExaSlang** In general, the approach for porting an application to our DSL is straightforward. First, data structures are prepared in ExaSlang. This also allows the automatic

```

1 Function calc_difflow ( flow : Real, diff : Real ) : Real {
2   Val tmp : Real = ( diff - 0.1 * fabs ( flow ) ) / diff
3   return max ( 0.0, diff * ( tmp ** 5 ) )
4 }
5
6 Function CompileStencil_u@finest ( ) : Unit {
7   loop over AuStencil@current {
8     Val flowSouth : Real =
9       integrateOverXStaggeredSouthFace (
10        v[active]@current * rho[active]@current
11      )
12
13     Val diffSouth : Real =
14       integrateOverXStaggeredSouthFace (
15        evalAtXStaggeredSouthFace ( vis@current, "harmonicMean" )
16      ) / vf_stagCVWidth_y@current
17
18     AuStencil@current:[ 0, -1, 0] = -1.0 * (
19       calc_difflow ( flowSouth, diffSouth ) + max ( 0.0, flowSouth )
20     )
21
22     /* ... other stencil components ... */
23   }
24
25   // restrict stencil to corser levels
26   StencilRestriction_u@current ( )
27 }

```

Listing 12: Example for using the evaluation and integration functions to calculate the stencil entries for the u component of the velocity. Only the calculation of the south entry is shown.

generation of copy-in and copy-out functions, thus effectively facilitating coupling with existing code. Next, single kernels are replicated in our DSL using the data structures just set up. The original kernels can then be replaced by calling the generated counterparts. This also facilitates early detection of coding errors. After the first steps have been repeated for some iterations, multiple kernels may be linked through more complex control flow in the DSL. This allows replacing whole functions from the original code with generated versions, thus minimizing overhead due to data copy operations. As an optional step, auxiliary functionalities can be added. These may include adding debug output, post-processing data, loading and storing simulation data from file, and eliciting performance data. Ideally, the whole process is repeated up to a point where the whole application can be generated and the original code is used only as reference. In cases where this is not possible, or too expensive in terms of implementation time, we envision replacing only certain blocks such as the numerical solver components. From the user's perspective, the interfacing then works similarly to using an external library.

In the present case, however, we were able to implement the whole application in ExaSlang. The full specification requires around 1,000 lines of code, depending on the actual version, i.e., Newtonian or non-Newtonian, how much profiling code is added, etc. Compared with the original FORTRAN code, this amounts to a reduction of about one order of magnitude.

**DSL and code generator extensions** Considering language extensions, external DSLs have the big advantage that their language specification is not limited by, e.g., restrictions imposed by the host language. This allows for a very fast and easy extension of language features such as the ones presented. For internal DSLs, this may be more complicated. Furthermore, external DSLs allow full control over a program's syntax tree which is, additionally, limited in its complexity by the capabilities of the input language. This facilitates implementation of transformations which perform changes at multiple points in the syntax tree. One example where this is necessary can be found in the context of an extension towards non-uniform grids: Here, buffers for storing the geometric

```

1 void CompileStencil_u_5 () {
2 // enter (parallel) outer loop
3 #pragma omp parallel for schedule(static) num_threads(20)
4 for (int z = 0; z<32; z += 1) {
5   for (int y = 0; y<32; y += 1) {
6     // pre-calculate utilized addresses
7     double* const _fd_AuStencilData_4_p0 = (&fieldData_AuStencilData
8       [4][((1190*z)+(35*y))]);
9     double* const _fd_node_pos_x_4_p0 = (&fieldData_node_pos_x[4][0]);
10    double* const _fd_node_pos_z_4_p0 = (&fieldData_node_pos_z[4][z]);
11    double* const _fd_stag_cv_width_y_4_p0 = (&
12      fieldData_stag_cv_width_y[4][y]);
13    double* const _fd_vis_4_p0 = (&fieldData_vis[4][((1156*z)+(34*y))
14      ]);
15    double* const _slottedFieldData_rho_4_cs = (&slottedFieldData_rho
16      [4][(currentSlot_rho[4]%2)][((1156*z)+(34*y))]);
17    double* const _slottedFieldData_v_4_cs = (&slottedFieldData_v[4][
18      (currentSlot_v[4]%2)][((1190*z)+(34*y))]);
19
20    // enter inner loop
21    for (int x = iterationOffsetBegin[0]; x<(iterationOffsetEnd[0]+33)
22      ; x += 1) {
23
24      // calculate flowSouth
25      double _i02_flow = (0.5*(_fd_node_pos_z_4_p0[3]-
26        _fd_node_pos_z_4_p0[2])*(((_fd_node_pos_x_4_p0[(x+2)]-
27          _fd_node_pos_x_4_p0[(x+1)])*_slottedFieldData_v_4_cs[(x+1224)]*
28          _slottedFieldData_rho_4_cs[(x+1156)])+((_fd_node_pos_x_4_p0[(x
29            +3)]-_fd_node_pos_x_4_p0[(x+2)])*_slottedFieldData_v_4_cs[(x
30              +1225)]*_slottedFieldData_rho_4_cs[(x+1157)]))));
31
32      // calculate diffSouth
33      double _i02_diff = (0.5*(((fd_node_pos_z_4_p0[3]-
34        _fd_node_pos_z_4_p0[2])*((fd_node_pos_x_4_p0[(x+2)]-
35          _fd_node_pos_x_4_p0[(x+1)])*_fd_vis_4_p0[(x+1156)])+((
36            _fd_node_pos_x_4_p0[(x+3)]-_fd_node_pos_x_4_p0[(x+2)])*
37              _fd_vis_4_p0[(x+1157)])))/_fd_stag_cv_width_y_4_p0[2]));
38
39      // apply calc_diflow ...
40      double _i02_tmp = ((_i02_diff-(0.1*fabs(_i02_flow)))/_i02_diff);
41
42      // ... and store result in stencil field
43      _fd_AuStencilData_4_p0[(x+122606)] = (std::min({(-(0.5*(
44        _fd_node_pos_z_4_p0[3]-_fd_node_pos_z_4_p0[2])*(((
45          _fd_node_pos_x_4_p0[(x+2)]-_fd_node_pos_x_4_p0[(x+1)])*
46            _slottedFieldData_v_4_cs[(x+1224)]*_slottedFieldData_rho_4_cs[(
47              x+1156)])+((_fd_node_pos_x_4_p0[(x+3)]-_fd_node_pos_x_4_p0[(x
48                +2)])*_slottedFieldData_v_4_cs[(x+1225)]*
49                  _slottedFieldData_rho_4_cs[(x+1157)]))}),0.0))-std::max({0.0,(
50                    _i02_diff*_i02_tmp*_i02_tmp*_i02_tmp*_i02_tmp*_i02_tmp)}));
51
52      /* ... other stencil components ... */
53    }
54  }
55 }
56 // restrict stencil to corser levels
57 StencilRestriction_u_5();
58 }

```

Listing 13: Possible output of our code generator for listing 12. The code has been annotated with comments for improved comprehensibility.



parameters are required. They need to be allocated, initialized and, ultimately, freed. Of course, this must happen only once, usually in different parts of the program than where the buffers are accessed. Additionally, buffer accesses may need to be adapted. For axis-aligned grids, this includes projecting the index coordinates appropriately. Moreover, if a user chooses to switch to, e.g., a uniform grid, no changes to the DSL specification should be necessary. In this case, an implementation also needs to be flexible enough to not generate unnecessary buffers and buffer accesses. In our opinion, a realization in a modular, transformation based code generation framework fed through an external DSL is substantially easier.

ExaStencils provides such an infrastructure. This makes, e.g., adding a module to handle and represent different grids quite easy. Here, one of the big benefits is that the newly introduced functions still map to existing syntax tree types. This, in turn, allows keeping the compiler work-flow largely unchanged. Even automatic optimizations usually work without further intervention, although introducing specializations may be beneficial in some cases. Furthermore, generator extensions for other types of grids that are not strictly axis-aligned are straightforward since the infrastructure is already present and most of the already implemented code can be reused. Compared with libraries, we deem the implementation effort for the core module to be about equal. However, integration with other parts of the library and code generator, respectively, is in our opinion easier when using a code transformation based approach as in ExaStencils.

## 8. RESULTS

**Evaluation setup** For our numerical tests we regard two configurations, both based on the method described in section 5. That is, the 5 PDEs for velocity components, pressure correction and temperature are solved separately and coupled through the SIMPLE algorithm. In the first variant, Newtonian behavior is studied, while in the second, non-Newtonian properties are incorporated by using the Bingham model as described in section 4. In both cases, the corresponding application and the contained solvers are fully generated from ExaSlang 4 code and automatically parallelized using OpenMP. That is none of the original code is coupled. All obtained results have been checked against the output of the original FORTRAN code as well as experimental data where available. Comparing to the original code, results are usually not fully identical since the utilized solvers have been switched from TDMA to geometric multigrid. Nevertheless, all tested configurations yield satisfactory accordance with reference data.

As a concrete test case for the subsequent performance evaluation, we choose the simulation of natural convection. Here, the left and right walls are heated and cooled respectively while all other walls are assumed to be adiabatic, i.e., follow Neumann boundary conditions. Velocities are initialized to zero and no-slip (i.e., Dirichlet) boundary conditions are imposed. Due to temperature and density differences, buoyancy and gravitational forces, a flow is induced. The necessity of incorporating the non-Newtonian behavior is illustrated in figure 3. Here, a single slice through the domain is shown, where  $z$  is chosen to equal half of the computational domain's depth. Using this slice, the temperature distribution after 10,000 time steps is depicted for the Newtonian and the non-Newtonian cases. In the Newtonian case, the isotherms show the typical behavior of a strong natural convective flow, that is almost horizontal isotherms along all the domain but close to the vertical walls, where the ascendant and descendant flows are strong. When the fluid acquires a Bingham non-Newtonian behavior with  $\tau_y = 0.1$  Pa, the heat transfer in the fluid becomes almost purely conductive, which is demonstrated by the vertical shape of the isotherms.

To assess the performance of our generated code, we examine grid sizes of  $16^3$ ,  $32^3$  and  $64^3$ . These numbers correspond to the number of cells in the original grid, i.e., the number of unknowns for the pressure  $p$  and temperature  $\theta$ . For values associated with the staggered grids, i.e.,  $U$ ,  $V$  and  $W$ , the number of unknowns is increased by 1 in the respective staggered dimension.

The target hardware platform is one socket of our local compute cluster featuring an Intel Xeon E5-2660v2. To match the available resources of 10 cores with two hardware threads each, we use 20 OpenMP threads. For the simulation, we perform 10,000 time steps, usually enough to reach steady-state, and average the observed execution time.

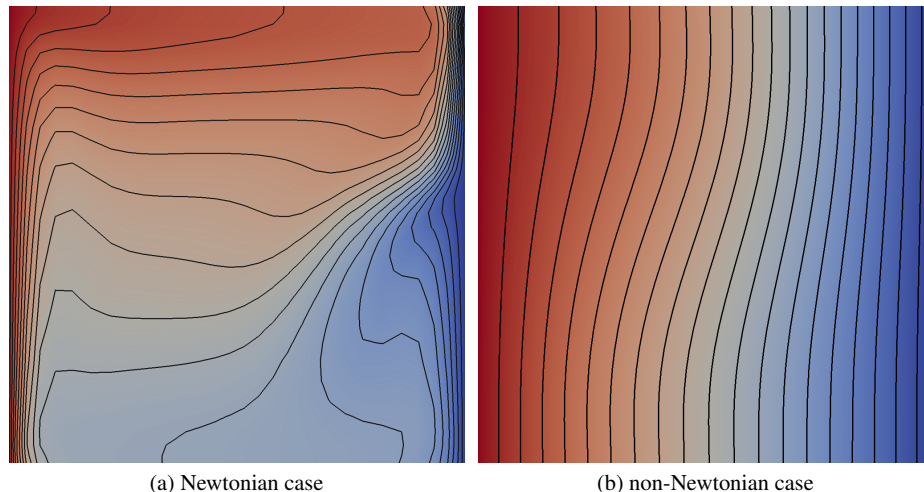


Figure 3. Temperature distribution along a slice with  $z$  at 50% of the box depth for the (a) Newtonian and (b) non-Newtonian case.

Please note that convergence and execution time characteristics can be highly dependent on tuning parameters of the method such as the relaxation factors for the linearization and the exit criteria used in the SIMPLE algorithm and the solvers therein. In our configuration, relaxation factors are set to 0.5 for each of the components. The convergence criteria used in our generated solvers and the reference code match a common specification: For solving the single components, the number of iterations  $n_{it}$  has to be larger than 0, that is we always perform at least one step, and

$$\|r_c^n\| \leq \alpha(1 + \beta\|f_c\|) \quad (27)$$

has to be fulfilled, where  $r_c^n$  is the residual in iteration  $n$  and  $f_c$  is the right-hand side of the LSE, for component  $c$  respectively. In our case,  $c$  may be either of the velocity components, the pressure correction or the temperature. Additionally, we improve the performance of non-converging configurations by exiting as well if

$$\|r_c^{n-1}\| - \|r_c^n\| < \tilde{\alpha}. \quad (28)$$

This does not harm overall accuracy since SIMPLE iterations are performed until convergence is reached. We assume convergence for a given time step if each component achieved convergence (i.e., (27) is fulfilled) *after* the respective LSE has been updated but *before* the respective component solver has been started. For this check, (28) is not taken into account. For our numerical experiments we choose  $\alpha = 10^{-6}$ ,  $\beta = 1$  and  $\tilde{\alpha} = 10^{-8}$ .

Our generator is able to automatically apply a wide range of low-level optimizations. For the results presented in this section, we enable function inlining, address pre-calculation and polyhedral optimizations as described in [38]. We deliberately postpone the evaluation of our automatic vectorization and loop fusion strategies. The main reason for this is the planned extension of the optimizations to handle common subexpressions, also across loop iterations. Moreover, the main focus of this paper is demonstrating that efficient source code can be generated for complex problems.

**Distribution of execution time** Figure 4 shows separate timings for the different components of our generated code. This variant already includes the aforementioned optimizations. As evident, both variants behave similarly with respect to performance. Adding the non-Newtonian behavior model is noticeable, but does not impact overall performance by a large amount. Interesting for us, and probably the community as a whole, is the finding that only roughly half the time is spent in actually solving for components while the other half is mostly spent in re-computing stencil

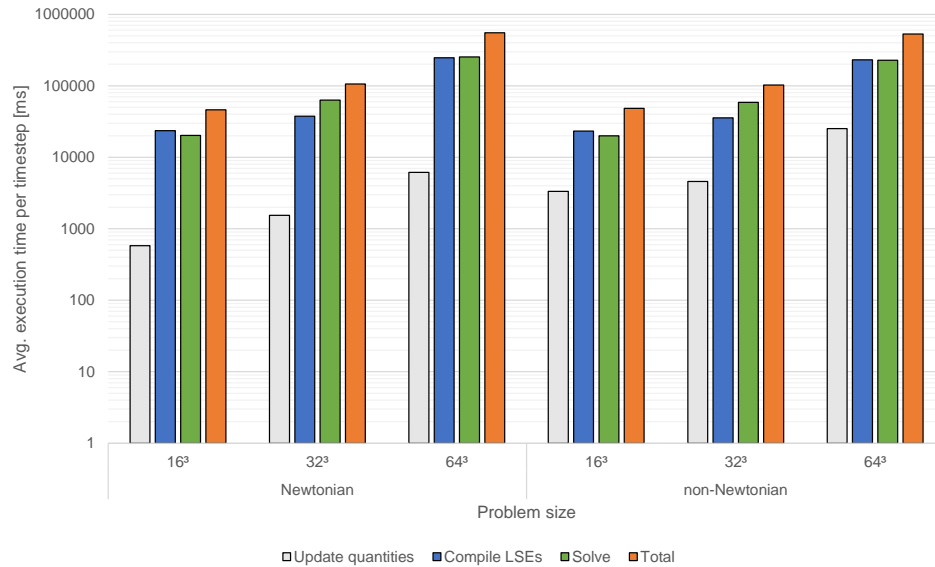


Figure 4. Execution times per time step for varying problem sizes in log scale. 10,000 timesteps were performed. Separate timings are given for the portions of the program spent in updating physical properties such as viscosity, compiling the LSEs, i.e., updating the stencil coefficients, and solving. Total time includes other factors such as convergence checks.

coefficients. These frequent updates are necessary due to the non-linear nature of the problem and the chosen SIMPLE algorithm. Computing and storing the coefficients before the solvers are executed is necessary since a coarsening of the system has to be performed, and due to the chosen convergence criteria. Furthermore, the LSEs remain constant during one multigrid solve, i.e., the coefficients are re-used multiple times. While the solver parts can be optimized by applying established techniques, the update phase provides a tougher challenge. For instance, a popular way of increasing the performance of bandwidth-bound codes is using temporal blocking, which is not possible here. Consequently, adaptation of traditional and development of new optimization techniques for these types of stencil codes is necessary in our opinion. Moreover, a holistic approach would be preferable over an isolated optimization of single kernels.

**Performance evaluation** In figure 5, we compare the performance of our generated solvers with respect to the applied optimizations. Furthermore, we apply a simple roofline model [39] to assess the attainable performance of our program. This is done in multiple steps. First, we apply the model to all executed kernels. Here, we assume the theoretical bandwidth taken from the processor’s manual and that each hardware thread is able to perform one vectorized fused multiply-add per cycle. Applying the model reveals that the (multigrid) solver parts are strongly memory bound, which is to be expected. The kernels setting up the stencil coefficients and right-hand sides are more or less at the break-even point between being memory and compute bound. On the chosen architecture, however, there is a slight tendency towards being memory bound. After evaluating theoretical performance of all kernels we aggregate the performance for key components of the program, such as single v-cycles and setup of single LSEs. These values are then multiplied with the measured number of calls since it is not possible to estimate these beforehand. As usual, roofline estimates can only give a limited prediction. On the one hand, effects such as reuse of data residing in caches across different kernels can lead to performance beyond the theoretically optimal roofline numbers. On the other hand, the roofline model assumes the code to be optimized perfectly, that is no unnecessary data is loaded or stored, additions and multiplications can always be fused, and the code is vectorized perfectly.

As figure 5 shows, our performance optimizations are currently not able to increase performance by a lot. However, at least for larger problem sizes, performance is already close to the roofline

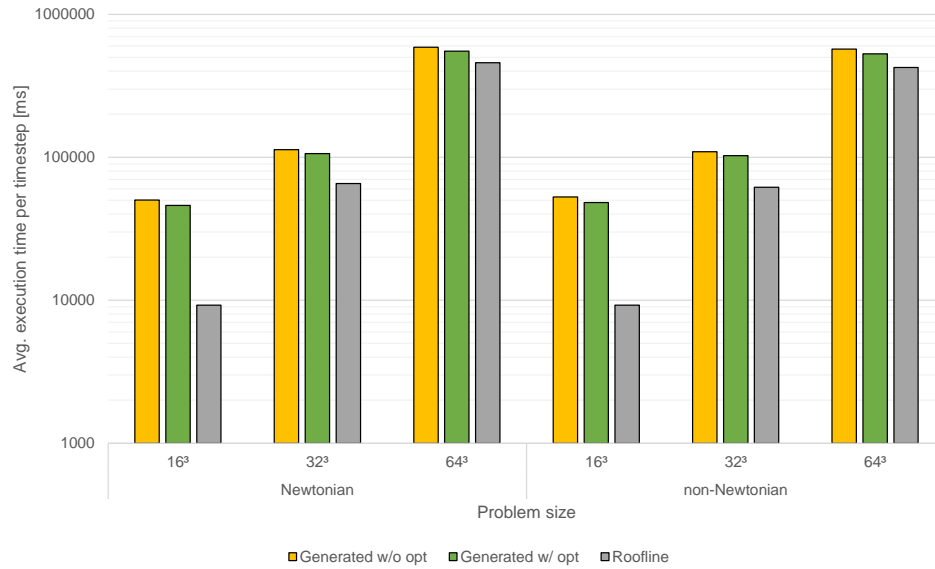


Figure 5. Zoom-in on execution times per time step for varying problem sizes in log scale. 10,000 timesteps were performed. Generated version with and without applied optimizations are compared with a theoretical time estimated using the roofline model.

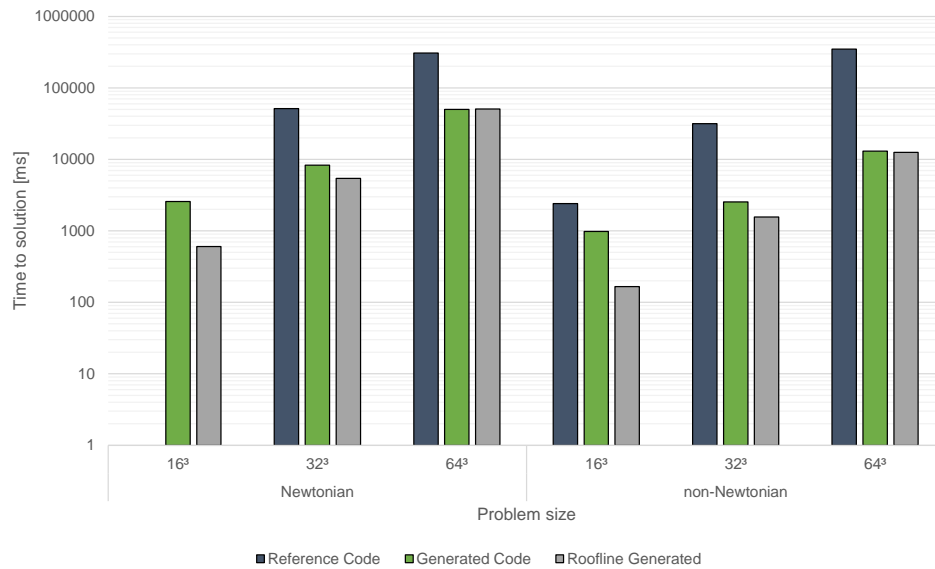


Figure 6. Total time to solution for varying problem sizes in log scale. Instead of performing timesteps we solve directly for the steady state. The original FORTRAN reference code is compared with our fully generated solver and a theoretical time estimated using the roofline model. The latter is estimated with respect to the generated version. The reference code was not able to converge for the 16<sup>3</sup> Newtonian test case.

estimate. One issue with smaller configurations is the increased overhead introduced by the ghost layers. These lead to a reduced effective memory bandwidth as full cache line utilization is not possible anymore. Furthermore, our code is not able to achieve good resource utilization and load balancing due to the implemented OpenMP parallelization. Currently, we only target the outermost loop which leads to 16/17, 32/33 and 64/65 parallel loop iterations distributed among 20 threads. A more fine-grained parallelization might help in this case.

**Comparison with the original code** Lastly, we perform a comparison of our generated applications with the original FORTRAN code. Just as the generated solver, no manual vectorization has been performed. Unfortunately, there have been some minor inconsistencies in execution time when using the original code for some of the chosen configurations. Thus, we switch from computing 10,000 timesteps to solving directly for the steady state. In this case, one of the configurations fails to converge in a suitable number of steps ( $16^3$ , Newtonian), but the other ones yield stable performance characteristics. The results are visualized in figure 6. Additionally, the performance of the generated solvers is again compared with the theoretical roofline estimate computed as described above.

For the converging configurations, we observe speedups between 2 and 27 where the average is around 10. This speedup can partly be explained by the better choice of solvers. Especially in the non-Newtonian case where diffusive processes dominate, the geometric multigrid solvers work quite well. Furthermore, there are still some isolated serial parts in the original code. Nevertheless, we can show that our generated simulation code matches, and in this case even outperforms, hand-written code. For larger problem sizes, our generated solvers are also able to attain performance close to the roofline estimates.

**General assessment** The presented results show that attained performance is already quite good. Some optimizations, however, are still necessary. More specifically, automatic vectorization needs to be added and we need to verify that it is able to fully handle the complicated kernels arising in this application. In conjunction, a sophisticated common subexpression elimination has the potential to simplify the kernels to be optimized. Nevertheless, the performance results already show a considerable improvement over the hand-coded reference code with observed speedups being around one order of magnitude on average. Furthermore, the implementation of new models, discretizations, data layouts and parallelization concepts is facilitated substantially. This also shows in the fact that the number of lines of code has been reduced by approximately an order of magnitude when comparing the new DSL code with the original FORTRAN counterpart.

## 9. CONCLUSION AND OUTLOOK

**Conclusion** In conclusion, our goal of fully generating a simulation code for researching non-Newtonian fluids was achieved. This represents an important first step towards supporting a whole class of relevant applications in the field of computational fluid dynamics. Our code generator and DSL are now equipped with many features required in this scope. These extensions include support for non-equidistant meshes and staggered grids as well as various boundary conditions. Furthermore, operations inherent to the application domain, e.g., evaluation and integration of expressions with respect to specific geometric locations, are available. Using these enhancements, which could easily be employed in similar projects, we are able to fully generate OpenMP parallel solvers for coupled Navier-Stokes and temperature equations based on the SIMPLE algorithm. Additionally, they are already coupled with models for the incorporation of non-Newtonian behavior. Further motivation is given by the presented performance results.

**Outlook** After demonstrating the general applicability of our code generation approach, we plan a closer investigation of newly arising optimization opportunities. Furthermore, introducing distributed memory parallelism through, e.g., MPI, and supporting a broader range of hardware, such as accelerators, is highly desirable. For this, the setup code for geometric information for non-uniform staggered grids has to be extended. Additionally, when moving to large scale clusters featuring hundreds of thousands cores directly solving on the coarsest level is not feasible any more. In these cases, a dedicated coarse grid solver, e.g., a BiCGSTAB variant, is crucial for maintaining good scalability. As our code generation framework is already capable of emitting OpenMP parallel code, an extension for OpenMP 4.0 and OpenACC is easily conceivable. Similarly, we plan an extension for accelerators based on CUDA and/or OpenCL. Moreover, we currently examine further extensions for ExaSlang 4, such as ways to write stencils in a more generic and dimension independent way.

The current solver approach solves the five PDEs ( $v_x, v_y, v_z, p, \theta$ ) separately using multigrid and combines them via the SIMPLE algorithm. A better approach is given by using one single multigrid solver for the linearized coupled system (17) with a Vanka-type smoother [40, 41] for that saddle point problem. In the smoothing step, local  $7 \times 7$  systems taking the staggered grid into account have to be extracted and solved. Since manual specification of the arising stencils is quite complicated, especially also considering the incorporation of boundary conditions, we aim at a more abstract specification. More specifically, we currently examine possibilities to simply specify a set of unknowns to be solved for in a given neighborhood and a set of equations to be fulfilled at these points. The code generator will then be able to distinguish values to be solved for and fixed ones as well as their connections. This, in turn, allows automatically setting up a local system to be solved for directly. As a last step, the unknowns can be updated accordingly. An extension of the multigrid solver for the whole linearized coupled system including the temperature  $\theta$  (12) can be done easily as the temperature values are stored in the same place as the pressure  $p$ . Once this multigrid algorithm can be generated automatically for the linear coupled problem, we can additionally tackle the original non-linear problem (12) by means of the full approximation scheme (FAS).

#### ACKNOWLEDGEMENTS

This work is supported by the German Research Foundation (DFG), as part of the Priority Programme 1648 “Software for Exascale Computing” in project under contract RU 422/15-1 and RU 422/15-2.

This has been cooperative work that started at the Dagstuhl Seminar *Advanced Stencil-Code Engineering* in April 2015. SPPEXA also partially supported the attendance of Gundolf Haase at this seminar.

D.Vasco acknowledges CONICYT-CHILE for the support received in the FONDECYT project 11130168.

#### REFERENCES

1. Hackbusch W. *Multi-Grid Methods and Applications*. Springer-Verlag, 1985.
2. Trottenberg U, Oosterlee CW, Schüller A. *Multigrid*. Academic Press, 2001.
3. Patankar SV, Spalding DB. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *International Journal of Heat and Mass Transfer* 1972; **15**(10):1787–1806.
4. Patankar SV. *Numerical Heat Transfer and Fluid Flow*. Series in Computational Methods in Mechanics and Thermal Sciences, McGraw-Hill: New York, 1980.
5. Baker AH, Falgout RD, Kolev TV, Meier Yang U. Scaling hypre’s multigrid solvers to 100,000 cores. *High-Performance Scientific Computing*. Springer, 2012; 261–279.
6. Meier Yang U, Henson VE. Boomer AMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics* 2002; **41**(1):155–177.
7. Bungartz HJ, Mehl M, Neckel T, Weinzierl T. The PDE framework Peano applied to fluid dynamics: An efficient implementation of a parallel multiscale fluid dynamics solver on octree-like adaptive Cartesian grids. *Computational Mechanics* 2010; **46**(1):103–114.
8. Bastian P, Blatt M, Dedner A, Engwer C, Klöfkorner R, Kornhuber R, Ohlberger M, Sander O. A generic grid interface for parallel and adaptive scientific computing. Part II: Implementation and tests in DUNE. *Computing* 2008; **82**(2):121–138.
9. Heroux MA, Bartlett RA, Howle VE, Hoekstra RJ, Hu JJ, Kolda TG, Lehoucq RB, Long KR, Pawlowski RP, Phipps ET, et al.. An overview of the trilinos project. *ACM Transactions on Mathematical Software* 9 2005; **31**(3):397–423.
10. Heroux MA, Willenbring JM. A new overview of the trilinos project. *Scientific Programming* 4 2012; **20**(2):83–88.
11. Balay S, Gropp WD, McInnes LC, Smith BF. Efficient management of parallelism in object-oriented numerical software libraries. *Modern Software Tools for Scientific Computing*, Birkhäuser Press, 1997; 163–202.
12. Unat D, Cai X, Baden SB. Mint: Realizing CUDA performance in 3d stencil methods with annotated C. *Proceedings of the International Conference on Supercomputing (ISC)*, 2011; 214–224.
13. Gysi T, Osuna C, Fuhrer O, Bianco M, Schulthess TC. STELLA: A domain-specific tool for structured grid methods in weather and climate models. *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, 2015; 41:1–41:12.
14. Christen M, Schenk O, Burkhart H. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. *Parallel & Distributed Processing Symposium (IPDPS)*, IEEE, 2011; 676–687.
15. Tang Y, Chowdhury RA, Kuszmaul BC, Luk CK, Leiserson CE. The Pochoir stencil compiler. *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, ACM, 2011; 117–128.
16. DeVito Z, Joubert N, Palacios F, Oakley S, Medina M, Barrientos M, Elsen E, Ham F, Aiken A, Duraisamy K, et al.. Liszt: A domain specific language for building portable mesh-based PDE solvers. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, 2011. Paper 9, 12 pp.
17. Membarth R, Hannig F, Teich J, Körner M, Eckert W. Generating device-specific GPU code for local operators in medical imaging. *Parallel & Distributed Processing Symposium (IPDPS)*, IEEE: Shanghai, China, 2012; 569–581.

18. Membarth R, Reiche O, Schmitt C, Hannig F, Teich J, Stürmer M, Köstler H. Towards a performance-portable description of geometric multigrid algorithms using a domain-specific language. *Journal of Parallel and Distributed Computing* 2014; **74**(12):3191–3201.
19. Logg A, Mardal KA, Wells GN. *Automated Solution of Differential Equations by the Finite Element Method, Lecture Notes in Computational Science and Engineering*, vol. 84. Springer, 2012.
20. Lengauer C, Apel S, Bolten M, Größlinger A, Hannig F, Köstler H, Rude U, Teich J, Grebhahn A, Kronawitter S, et al.. ExaStencils: Advanced stencil-code engineering. *Euro-Par 2014: Parallel Processing Workshops, Lecture Notes in Computer Science*, vol. 8806, Springer, 2014; 553–564.
21. Schmitt C, Kuckuk S, Hannig F, Köstler H, Teich J. ExaSlang: A domain-specific language for highly scalable multigrid solvers. *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, IEEE Computer Society, 2014; 42–51.
22. Kuckuk S, Köstler H. Automatic generation of massively parallel codes from exaslang. *Computation* 2016; **4**(3):27.
23. Barnes HA. The yield stress—a review—everything flows? *Journal of Non-Newtonian Fluid Mechanics* 1999; **81**:133–178.
24. Zhu H, Kim Y, De Kee D. Non-newtonian fluids with a yield stress. *Journal of Non-Newtonian Fluid Mechanics* 2005; **129**:177–181.
25. Gratao ACA, Silveira Jr V, Telis-Romero J. Laminar flow of soursop juice through concentric annuli: Friction factors and rheology. *Journal of Food Engineering* 2007; **78**:1343–1354.
26. Telis-Romero J, Telis VRN, Yamashita F. Friction factors and rheological properties of orange juice. *Journal of Food Engineering* 1999; **40**:101–106.
27. Yue J, Klein B. Influence of rheology on the performance of horizontal stirred mills. *Minerals Engineering* 2004; **17**:1169–1177.
28. Genc AM, Kilickaplan I, Laskowski JS. Effect of pulp rheology on flotation of nickel sulphide ore with fibrous gangue particles. *Canadian Metallurgical Quarterly* 2012; **51**:368–375.
29. Richmond WR, Jones RL, Fawell PD. The relationship between particle aggregation and rheology in mixed silica–titania suspensions. *Chemical Engineering Journal* 1998; **71**(1):67–75.
30. Katiyar A, Singh AN, Shukla P, Nandi T. Rheological behavior of magnetic nanofluids containing spherical nanoparticles of fe–ni. *Powder Technology* 2012; **224**:86–89.
31. Sharma AK, Tiwari AK, Dixit AR. Rheological behaviour of nanofluids: A review. *Renewable and Sustainable Energy Reviews* 2016; **53**:779–791.
32. Tseng WJ, Li SY. Rheology of colloidal batio3 suspension with ammonium polyacrylate as a dispersant. *Materials Science and Engineering: A* 2002; **333**(1–2):314–319.
33. Tseng WJ, Tzeng F. Effect of ammonium polyacrylate on dispersion and rheology of aqueous {ITO} nanoparticle colloids. *Colloids and Surfaces A: Physicochemical and Engineering Aspects* 2006; **276**(1–3):34–39.
34. Banaszek J, Jaluria Y, Kowalewski TA, Rebow M. Semi-implicit fem analysis of natural convection in freezing water. *Numerical Heat Transfer, Part A: Applications* 1999; **36**(5):449–472.
35. O'Donovan EJ, Tanner RI. Numerical study of the bingham squeeze film problem. *Journal of Non-Newtonian Fluid Mechanics* 1984; **15**(1):75–83.
36. Vasco DA, Moraga NO, Haase G. Parallel finite volume method simulation of three-dimensional fluid flow and convective heat transfer for viscoplastic non-newtonian fluids. *Numerical Heat Transfer, Part A: Applications* 2014; **66**(2):990–1019.
37. Versteeg HK, Malalasekera W. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*. Pearson Education Limited, 2007.
38. Kronawitter S, Lengauer C. Optimizations applied by the ExaStencils code generator. *Technical Report MIP-1502*, Faculty of Informatics and Mathematics, University of Passau 2015.
39. Williams S, Waterman A, Patterson D. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM* 2009; **52**(4):65–76.
40. Vanka SP. Block-implicit multigrid calculation of two-dimensional recirculating flows. *Computer Methods in Applied Mechanics and Engineering* 1986; **59**(1):29–48.
41. Martynenko SI, Martynenko PS. Robust multigrid method for solving the Navier-Stokes equations on structured grids. *Modern Applied Science* 2012; **6**(6):73–91.