KYBERNETIKA — VOLUME *46* (2010), NUMBER 2, PAGES 294–315

GENERIC IMPLEMENTATION OF FINITE ELEMENT METHODS IN THE DISTRIBUTED AND UNIFIED NUMERICS ENVIRONMENT (DUNE)

Peter Bastian, Felix Heimann and Sven Marnach

In this paper we describe PDELab, an extensible C++ template library for finite element methods based on the Distributed and Unified Numerics Environment (Dune). PDELab considerably simplifies the implementation of discretization schemes for systems of partial differential equations by setting up global functions and operators from a simple element-local description. A general concept for incorporation of constraints eases the implementation of essential boundary conditions, hanging nodes and varying polynomial degree. The underlying Dune software framework provides parallelization and dimension-independence.

Keywords: finite elements, generic programming

Classification: 65M02, 65N02, 65Y02

1. INTRODUCTION

Dune [2, 3, 9] is a set of C++ libraries for the grid-based numerical solution of partial differential equations (PDEs). Its main design principles are: (i) separation of data structures and algorithms through abstract interfaces, (ii) use of generic programming techniques for achieving performance and (iii) enabling reuse of existing finite element software through appropriate interface design. Dune provides support for many different kinds of grids, a flexible linear solver package [4, 1], is parallel as well as dimension-independent and offers a full simulation workflow using free software. Salome [16] can be used to work interactively with CAD models through the OpenCascade [13] CAD kernel. Meshes are generated with Gmsh [10] which incorporates several mesh generators and can access CAD models through the OpenCascade kernel. Gmsh meshes are then read by Dune into the various mesh implementations and subsequently simulation results are visualized using Paraview/VTK [14] supporting parallel visualization.

The implementation of real world application problems using the Dune grid and solver modules only is a tedious task. Several attempts have been undertaken to support the implementation of discretization schemes in Dune, most notably the Dune-Fem [8] module. In this paper we describe PDELab, a C++ template library

based on the DUNE framework that considerably simplifies the implementation of discretization schemes. It has been designed with the following goals in mind:

- Substantially reduce time to implement discretization schemes for systems of PDEs based on DUNE (rapid prototyping).
- Simple things should be simple – suitable for teaching. By this we mean that we mean that a simple discretization scheme for a simple equation should be implementable with a few pages of code in a few hours (provided the student has the required background in numerical methods and programming).
- Support of general finite element spaces including non-conforming spaces, *hp*-spaces and vector-valued spaces.
- General approach to constraints such as essential boundary conditions and hanging nodes.
- Generic approach to systems of PDEs.
- Exchangeable linear algebra backend that allows use of external solver libraries such as PetSc [15] or Trilinos [17] in addition to the DUNE solvers.

This paper is structured as follows. In Section 2 we give an abstract formulation of PDE discretization methods based on a weighted residual formulation. In Section 3 we describe how finite element spaces are realized in PDELAB while Section 4 is devoted to a description of the generic assembly process. Then in Section 5 some numerical results are presented to evaluate the flexibility and performance of the approach. Finally, Section 6 draws some conclusions.

2. WEIGHTED RESIDUAL FORMULATION

2.1. Stationary problems

Let us first consider stationary, possibly nonlinear systems of partial differential equations (PDEs). For a general PDE discretization framework we need an abstract problem formulation.

Definition 2.1 (Weighted residual formulation)**.** We claim that a large class of discretization schemes for partial differential equations can formally be written as

$$\text{Find } u_h \in w_h + \tilde{U}_h : \qquad r_h(u_h, v) = 0 \qquad \forall\, v \in \tilde{V}_h. \tag{1}$$

Here $\tilde{U}_h \subseteq U_h$ and $\tilde{V}_h \subseteq V_h$ are subspaces of finite dimensional function spaces U_h (trial space) and V_h (test space). $w_h \in U_h$ is a function that incorporates the essential boundary conditions and the solution u_h is sought in the affine subspace $w_h + \tilde{U}_h = \{y_h = w_h + \tilde{u}_h : \tilde{u}_h \in \tilde{U}_h\}$. $r_h : U_h \times V_h \to \mathbb{R}$ is the residual form which may be nonlinear in its first argument and is always linear in its second argument. Finally, we assume that problem (1) has a unique solution.

This abstract formulation encompasses many well-known discretization methods such as conforming and non-conforming finite element methods, finite volume methods or finite difference methods. In contrast to many text book presentations of these methods, we want to emphasize the importance of how to treat essential boundary conditions as this is often a complication in the implementation of these methods.

In order to proceed we introduce basis representations of the spaces involved

$$U_h = \operatorname{span} \Phi_h, \qquad \Phi_h = \{\phi_i \,:\, i \in \mathcal{I}_{U_h}\}, \qquad \mathrm{FE}_{\Phi_h}(\mathbf{u}) = \sum_{i \in \mathcal{I}_{U_h}} \mathbf{u}_i \phi_i,$$

$$V_h = \operatorname{span} \Psi_h, \qquad \Psi_h = \{\psi_i \,:\, i \in \mathcal{I}_{V_h}\}, \qquad \mathrm{FE}_{\Psi_h}(\mathbf{v}) = \sum_{i \in \mathcal{I}_{V_h}} \mathbf{v}_i \psi_i.$$

$\mathcal{I}_{U_h}$, $\mathcal{I}_{V_h}$ are index sets and $\mathrm{FE}_{\Phi_h} : \mathbf{U} = \mathbb{R}^{\mathcal{I}_{U_h}} \to U_h$, $\mathrm{FE}_{\Psi_h} : \mathbf{V} = \mathbb{R}^{\mathcal{I}_{V_h}} \to U_h$ are finite element isomorphisms. Inserting the basis representation into (1) yields a nonlinear algebraic system which reads in the unconstrained case $\tilde{U}_h = U_h$, $\tilde{V}_h = V_h$ (the constrained case will be treated below):

$$\mathbf{u} \in \mathbf{U} : \quad \mathcal{R}(\mathbf{u}) = \mathbf{0} \tag{2}$$

where we introduced the nonlinear residual map $\mathcal{R}$ given by

$$\big(\mathcal{R}(\mathbf{u})\big)_i = r_h(\mathrm{FE}_{U_h}(\mathbf{u}), \psi_i). \tag{3}$$

2.2. Some examples

In order to illustrate the generality of the weighted residual formulation we formulate several different schemes for a linear second order elliptic PDE. Let Ω be a domain in $\mathbb{R}^d$ with boundary $\partial\Omega$. We consider

$$\nabla \cdot \sigma = f \qquad \text{in } \Omega, \tag{4a}$$

$$\sigma = -K(x)\nabla u \qquad \text{in } \Omega, \tag{4b}$$

$$u = g \qquad \text{on } \Gamma_D \subseteq \partial\Omega, \tag{4c}$$

$$\sigma \cdot \nu = j \qquad \text{on } \Gamma_N = \partial\Omega \setminus \Gamma_D. \tag{4d}$$

This equation describes e. g. stationary groundwater flow in a fully saturated porous medium. Then u is the hydraulic head and K is the tensor-valued hydraulic conductivity.

Let $\mathcal{T}_h$ be a shape regular family of triangulations of Ω (assumed to be polyhedral) with a generic element denoted by T. Depending on the type of scheme non-conforming triangulations (hanging nodes) may be allowed.

Example 2.2 (Conforming finite elements)**.** The conforming spaces are

$$U_h^k = \{u \in C^0(\Omega) \,:\, u|_T \in \mathbb{P}_k\}, \qquad \tilde{U}_h^k = \{u \in U_h^k \,:\, u(x) = 0 \text{ for } x \in \Gamma_D\},$$

with $\mathbb{P}_k$ the space of polynomials of total degree less than or equal to k if $\mathcal{T}_h$ consists of simplicial elements. The residual form reads

$$r_h^{\mathrm{FE}}(u_h, v) = \sum_{T\in\mathcal{T}_h} \int_T (K\nabla u_h)\cdot\nabla v \,\mathrm{d}x + \sum_{F\in\mathcal{F}_h^N} \int_F jv \,\mathrm{d}s - \sum_{T\in\mathcal{T}_h} \int_T fv \,\mathrm{d}x.$$

Since this is a Galerkin method we have $V_h = U_h^k$ and $\tilde{V}_h = \tilde{U}_h^k$. $\mathcal{F}_h^{\partial\Omega} = \mathcal{F}_h^D \cup \mathcal{F}_h^N$ is the set of element faces covering the domain boundary which is partitioned into Dirichlet and Neumann boundary faces. The affine shift w_h is any function with $w_h = g$ at vertices on Γ_D.

Example 2.3 (Cell centered finite volumes)**.** For this method we require that $\mathcal{T}_h$ is an axi-parallel, structured mesh. Moreover the conductivity coefficient is assumed to be scalar, i. e. $K(x) = k(x)I$. The cell-centered method is based on the space of piecewise constant functions

$$W_h^0 = \{u \in L_2(\Omega) \,:\, u|_T = \mathrm{const}\} = U_h = V_h.$$

A face F is an interior face if we can find two elements $T^-(F), T^+(F) \in \mathcal{T}_h$ such that $F = T^-(F) \cap T^+(F)$. By $x^-(F)$ and $x^+(F)$ we denote the centers of the elements $T^-(F)$ and $T^+(F)$ and by x_F we denote the center of the face F. The unit normal vector ν_F is chosen to point from element $T^-(F)$ to element $T^+(F)$ and by $\mathcal{F}_h^i$ we denote the set of all interior faces F. For boundary faces $F \in \mathcal{F}_h^{\partial\Omega}$ the normal direction ν_F is always the exterior normal and the element T_F^- denotes the element which has F as its face. For a point x on an interior face F, we define the jump of a function $u \in W_h^0$ as:

$$[u](x) = \lim_{\epsilon\to 0+} u(x - \epsilon\nu_F) - \lim_{\epsilon\to 0+} u(x + \epsilon\nu_F).$$

Using the two-point flux approximation the residual form for this method reads

$$\begin{aligned} r_h^{\mathrm{CC}}(u_h, v) = &\sum_{F\in\mathcal{F}_h^i} \int_F k(F) \frac{u_h(x_F^-) - u_h(x_F^+)}{\|x_F^+ - x_F^-\|} [v] \,\mathrm{d}s - \sum_{T\in\mathcal{T}_h} \int_T fv \,\mathrm{d}x \\ &+ \sum_{F\in\mathcal{F}_h^D} \int_F k(F) \frac{u_h(x_F^-) - g(x_F)}{\|x_F^+ - x_F^-\|} v \,\mathrm{d}s + \sum_{F\in\mathcal{F}_h^N} \int_F jv \,\mathrm{d}s. \end{aligned} \tag{5}$$

The conductivity $k(F)$ is typically computed as a harmonic average of the conductivity of the adjacent elements. Note that here we use W_h^0 as trial and test space as the constraints are already built into the residual form.

Example 2.4 (Discontinuous Galerkin method)**.** For this method the discrete function space is

$$W_h^k = \{u \in L_2(\Omega) \,:\, u|_T \in \mathbb{P}_{k(T)}\} = U_h = V_h,$$

where $k : \mathcal{T}_h \to \mathbb{N}$, $k(T) \geq 2$ assigns a polynomial degree to each element. For a point x on an interior face F, we define the average of a function $u \in W_h^k$ as:

$$\langle u\rangle(x) = \frac{1}{2}\left(\lim_{\epsilon\to 0+} u(x - \epsilon\nu_F) + \lim_{\epsilon\to 0+} u(x + \epsilon\nu_F)\right).$$

The residual form defining the Oden–Babuška–Baumann method [12] is given by

$$\begin{aligned} r_h^{\mathrm{OBB}}(u_h, v) = &\sum_{T\in\mathcal{T}_h} \int_T ((K\nabla u_h)\cdot\nabla v - fv)\,\mathrm{d}x + \sum_{F\in\mathcal{F}_h^N}\int_F jv\,\mathrm{d}s \\ &+ \sum_{F\in\mathcal{F}_h^i}\int_F (\langle\langle(K\nabla v)\cdot\nu_F\rangle[u] - [v]\langle(K\nabla u)\cdot\nu_F\rangle)\,\mathrm{d}s \\ &+ \sum_{F\in\mathcal{F}_h^D}\int_F (((K\nabla v)\cdot\nu_F)(u-g) - v((K\nabla u)\cdot\nu_F))\,\mathrm{d}s. \end{aligned}$$

Example 2.5 (Mixed finite element method)**.** In this method we use (4) directly in its first order (mixed) form. For the velocity σ we might use the Raviart–Thomas space of lowest order on triangles:

$$S_h = \left\{\sigma \in (L_2(\Omega))^2 \,:\, \sigma|_T = \begin{pmatrix} a_T \\ b_T \end{pmatrix} + c_T \begin{pmatrix} x \\ y \end{pmatrix} \quad \forall\, T\in\mathcal{T}_h\right\}.$$

and its subspace

$$\tilde{S}_h = \{\sigma\in S_h \,:\, \text{“}\sigma\cdot\nu = 0\text{” on } \Gamma_N\}.$$

Then the discrete problem in residual form reads:

$$\text{Find } (\sigma_h, u_h)\in(w_h+\tilde{S}_h)\times W_h^0 : \quad r_h^{\mathrm{MFE}}((\sigma_h,u_h),(v,q)) = 0 \quad \forall\,(v,q)\in\tilde{S}_h\times W_h^0$$

with

$$\begin{aligned} r_h^{\mathrm{MFE}}((\sigma_h,u_h),(v,q)) = &\sum_{T\in\mathcal{T}_h}\int_T \left((K^{-1}\sigma_h)\cdot v - u_h\,\nabla\cdot v - \nabla\cdot\sigma_h\, q + fq\right)\mathrm{d}x \\ &+ \sum_{F\in\mathcal{F}_h^D}\int_F gv\cdot\nu\,\mathrm{d}s \end{aligned}$$

and w_h some function with $w_h\cdot\nu = j$ on Γ_N.

Observation 2.6. From the examples given in this section we make the following observations that hold also for more complicated problems:

- The residual form can be split into contributions coming from integrals over elements $T\in\mathcal{T}_h$, integrals over interior faces $F\in\mathcal{F}_h^i$ and integrals over boundary faces $F\in\mathcal{F}_h^{\partial\Omega}$.
- Even for nonlinear problems, the residual form is always linear in its second argument.
- In case of systems of PDEs trial and test spaces are products of function spaces.

2.3. Instationary problems

In case of instationary problems we assume that discretization in space and time reduces the problem to a sequence of steps having the form postulated in Definition 2.1.

Example 2.7 (Heat equation)**.** Consider the time-dependent PDE

$$\partial_t u - \nabla \cdot \{K \nabla u\} = f \qquad \text{in } \Omega \times \Sigma$$

with $\Sigma = (a, b)$ a time interval and initial condition $u(x, a) = u^a(x)$ (boundary conditions are the same as in (4)). Using one of the spatial discretizations from above, e. g. conforming finite elements, we can write this in a method of lines approach as: Find $u_h(t) \in \tilde{U}_h^k$ such that

$$\frac{\mathrm{d}}{\mathrm{d}t} \int_\Omega u_h(t) v \,\mathrm{d}x + r_h^{\mathrm{FE}}(u_h(t), v; t) = 0 \qquad \forall\, v \in V_h,\ t \in \Sigma$$

and $u_h(a) = u_h^a$. Discretizing the time interval $a = t^0 < t^1 < \ldots < t^N = b$, $\Delta t^k = t^{k+1} - t^k$, and using the implicit Euler method we arrive at

$$r_h^{\mathrm{IEFE}}(u_h, v) = \int_\Omega \left(u_h^{k+1} - u_h^k\right) v \,\mathrm{d}x + \Delta t^k r_h^{\mathrm{FE}}(u_h^{k+1}, v; t) = 0 \quad \forall\, v \in V_h,\ 0 < k \leq N.$$

Other discretizations such as Runge–Kutta methods, multi-step methods or space-time Galerkin methods can be treated in the same way. Formally, explicit and implicit methods can be treated with the only difference that explicit methods lead to a linear system that is trivial to solve.

3. GRID FUNCTIONS

In order to construct finite-dimensional function spaces in a generic way we rely on affine families of finite elements [7].

Definition 3.1. A finite dimensional function space U_h defined on a triangulation $\mathcal{T}$ is defined by

$$\begin{aligned} U_h(\mathcal{T}_h) = \Bigg\{ & u_h(x) : \bigcup_{T \in \mathcal{T}_h} T \to \mathbb{R}^m \ : \\ & u_h(x) = \sum_{T \in \mathcal{T}_h} \sum_{i=0}^{n(T)-1} (\mathbf{u})_{g(T,i)}\, \pi_T(\hat{\phi}_{T,i}(\hat{x}), \hat{x})\, \chi_T(x);\ \hat{x} = \mu_T^{-1}(x) \Bigg\}, \end{aligned} \tag{6}$$

where $\hat{\phi}_{T,i}$ is a local basis function defined on the reference element of element T, $n(T)$ is the number of local basis functions on element T, the local-to-global map $g(T, i)$ associates this local basis function with a global degree of freedom, χ_T is the characteristic function of element T and $\mu : \hat{T} \to T$ maps the reference element of T to T. Finally, π_T transforms the local basis function values to global coordinates (an example would be the Piola transformation for $H(\mathrm{div})$-conforming finite elements).

3.1. Local Finite Elements

The global finite element space is build up from a local description element by element. In this subsection we give an overview of the classes giving this local description. By giving some code fragments we want to give an impression of the use of these classes. They are collected in the separate module `dune-local functions` as they might be reused by other finite element implementations.

First ingredient is the basis on the reference element which is given by a class implementing `C0LocalBasisInterface`. E.g. getting the Lagrange basis of order 4 on the tetrahedron is done by

```
typedef Pk3DLocalBasis<float,double,4>
LocalBasis; LocalBasis localbasis;
```

The class is parametrized by the type to represent coordinates (`float` here), the type to represent basis function values (`double` here) and the polynomial order (4 here). Evaluating *all* basis functions at a given position is done by the following code fragment:

```
LocalBasis::Traits::DomainType xlocal(1.0/3.0);
std::vector<LocalBasis::Traits::RangeType>
  phi(localbasis.size());
localbasis.evaluateFunction(x,phi);
```

The `Traits` class within `LocalBasis` holds all relevant types. The vector `phi` contains the values of all basis function at the point `xlocal` in coordinates of the reference element. Extensions of that class allow also the evaluation of first and higher order derivatives.

The second ingredient is a class implementing `LocalInterpolationInterface` which allows to get the basis representation of a given function `f`. If `f` cannot be represented exactly then this function returns an approximation (pointwise evaluation, L_2-projection). To this end we first define a class with an `evaluate` method that defines the function ($f(x) = x_0^2$ in this example) to interpolate:

```
template<class Traits> class F { public:
  void evaluate (typename Traits::DomainType x,
                 typename Traits::RangeType\& rv) const {
    rv = x[0]*x[0];
  }
};
```

Evaluating the coefficients with respect to the basis is then done by

```
typedef Pk3DLocalInterpolation<LocalBasis>
  LocalInterpolation;
LocalInterpolation localinterpolation;
F<LocalBasis::Traits> f;
std::vector<LocalBasis::Traits::RangeType>
  u(localbasis.size());
localinterpolation.interpolate(f,u);
```

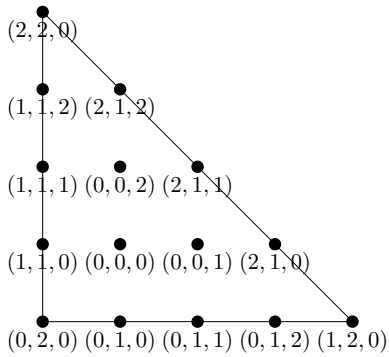

Fig. 1. Assignment of degrees of freedom to entities in the Lagrange basis of $\mathbb{P}_4$.

Finally, the third ingredient of the local description of finite element spaces is a class implementing `LocalCoefficientsInterface` which assigns degrees of freedom to geometrical entities on the reference element. This allows a generic construction of the local-to-global map $g(T, i)$ in Definition 3.1. Consider as an example the Lagrange basis of $\mathbb{P}_4$ on a triangle as shown in Figure 1 (we consider the two-dimensional case for ease of presentation). It has 15 degrees of freedom, three located in the element (codimension 0), three located in each edge (codimension 1) and one in each vertex (codimension 2). The assignment of each degree of freedom is encoded as a triple (s, c, i) where s is the number of the geometrical entity (the numbering is defined in the DUNE grid interface) with codimension c and i is an index within the entity. Listing all the triples for the $\mathbb{P}_4$ Lagrange basis is performed by the following code:

```
typedef Pk3DLocalCoefficients<4> LocalCoefficients;
LocalCoefficients localcoefficients; for (int i=0;
i<localcoefficients.size(); i++)
  std::cout << "degree of freedom " << i << " in "
  << localcoefficients.localKey(i).subEntity() << ","
  << localcoefficients.localKey(i).codim() << ","
  << localcoefficients.localKey(i).index() << ","
  << std::endl;
```

`LocalBasis`, `LocalInterpolation` and `LocalCoefficients` together make up the local description of a finite element and are accessible from a class implementing `LocalFiniteElementInterface`. In Definition 3.1 every element $T \in \mathcal{T}_h$ can have a different local basis. All local descriptions are collected in a container providing access to the local description for a given element T. These containers are called local finite element maps and implement `LocalFiniteElementMapInterface`. Generic setup of the local finite element map for our $\mathbb{P}_4$ example for a given `gridview` object of type `GridView` (a part of a DUNE grid) is shown here:

```
typedef GridView::ctype D; // coordinate type typedef double R;
```

```
// value type typedef Pk3DLocalFiniteElementMap<GridView,D,R,4>
LFEM; LFEM lfem(gridview);
```

In an *hp*-version of the finite element method the local finite element map would store the polynomial degree per element and deliver the appropriate local description.

Table 1. Finite element spaces currently implemented in PDELAB
Source lines of code for their implementation gives at least
a relative measure of the amount of work involved
(Note: the edge element code is documented in more detail than the others).

Local finite element	Source lines of code
Lagrange, order 1, simplex, $d = 1, 2, 3$	708
Lagrange, order 1, cube, $d = 1, 2, 3$	495
Lagrange, order 2, cube, $d = 2$	262
Lagrange, order k, simplex, $d = 2, 3$	1075
Monomial, order k, any d	520
Rannacher–Turek, quadrilateral	209
Raviart–Thomas, order 1, simplex	323
Nedelec (edge) elements, simplex	1688

The finite element spaces currently available in PDELAB are listed in Table 1. We also list the number of lines of code needed for their implementation.

3.2. Grid function space

From the local description of the finite element spaces introduced in the previous subsection, the global finite element space introduced in Definition 3.1 is constructed generically with the class template `GridFunctionSpace`. It is parametrized by a `GridView` and the local description of the finite element space:

```
typedef GridFunctionSpace<GridView,LFEM> GFS;
GFS gfs(gridview,lfem);
std::cout << "NDOFS=" << gfs.globalSize() << std::endl;
```

Moreover, a grid function space provides a means to set up a container holding the degrees of freedom (here of type `double`) of the global finite element space:

```
typedef GFS::VectorContainer<double>::Type U;
U u(gfs,0.0);
```

In the default implementation degrees of freedom are stored in a `std::vector`. Other implementations can be used by providing alternative backends as in:

```
typedef GridFunctionSpace<GridView,LFEM,
  NoConstraints, ISTLVectorBackend<1> > GFS;
GFS gfs(gridview,lfem);
typedef GFS::VectorContainer<double>::Type U;
U u(gfs,0.0);
```

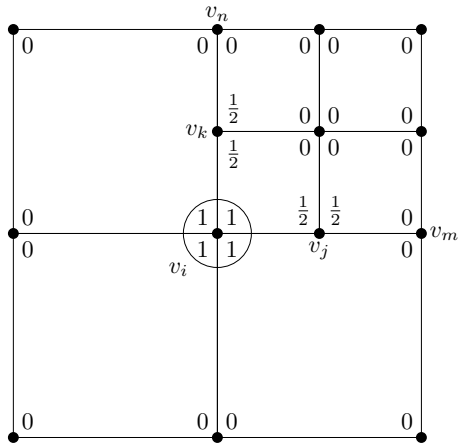

Fig. 2. Conforming Lagrange basis function for v_i on a mesh with hanging nodes.

Here, the backend using vectors from the iterative solver template library (ISTL) [4], which is part of DUNE, is used. ISTL supports recursively block-structured vectors and matrices with block sizes known at compile time. The integer template parameter indicates the first level block size. Through the backends other solver libraries such as PetSc [15] or Trilinos [17] could be integrated as well. The additional class parameter `NoConstraints` is introduced in the next subsection.

3.3. Constraints

Solving in a subspace $\tilde{U}_h \subseteq U_h$ is a fundamental complication in the implementation of finite element methods. Here we will call $\tilde{U}_h$ the *constrained space*. Constraints occur in several circumstances in a finite element code:

- Essential boundary conditions.
- "Hanging" node degrees of freedom due to non-conforming refinement.
- Periodic boundary conditions.
- Artificial essential boundary conditions due to parallelization (e. g. in overlapping Schwarz methods).
- Constants or rigid body modes preventing uniqueness of the solution.
- Combinations of the above.

In PDELAB all these different types of constraints are handled in a uniform way. The subspace $\tilde{U}_h$ has a corresponding index set $\tilde{\mathcal{I}}_{U_h} \subseteq \mathcal{I}_{U_h}$ and we define $\bar{\mathcal{I}}_{U_h} := \mathcal{I}_{U_h} \setminus \tilde{\mathcal{I}}_{U_h}$. As the next step we introduce a basis for the subspace $\tilde{U}_h$

$$\tilde{U}_h = \operatorname{span} \tilde{\Phi}_{U_h} \subseteq U_h, \qquad \tilde{\Phi}_{U_h} = \{\tilde{\phi}_i \,:\, i \in \tilde{\mathcal{I}}_{U_h}\}$$

and assume that it is related to the original basis Φ_{U_h} of U_h in the following way:

$$\tilde{\phi}_i = \phi_i + \sum_{j \in \bar{\mathcal{I}}_{U_h}} \left(\mathbf{T}_{\tilde{U}_h}\right)_{i,j} \phi_j, \quad i \in \tilde{\mathcal{I}}_{U_h}. \tag{7}$$

The matrix $\mathbf{T}_{\tilde{U}_h}$ is sparse and can be assembled locally element by element (with rigid body modes and periodic boundary conditions as exceptions). For a particular function space a class providing the local constraints needs to be given as a template parameter to the `GridFunctionSpace`. If no constraints are to be put on the function space then `NoConstraints` is given as a parameter.

Example 3.2 (Hanging nodes)**.** Consider the case of nonconforming refinement and bilinear conforming Lagrange basis functions on the mesh shown in Figure 2. Assume that v_m is the only vertex where a Dirichlet boundary condition is applied. Then $\bar{\mathcal{I}}_{U_h} = \{k, j, m\}$, $\tilde{\mathcal{I}}_{U_h} = \mathcal{I}_{U_h} \setminus \bar{\mathcal{I}}_{U_h}$ and

$$\tilde{\phi}_i = \phi_i + \frac{1}{2}\phi_k + \frac{1}{2}\phi_j, \qquad \tilde{\phi}_n = \phi_n + \frac{1}{2}\phi_k, \qquad \tilde{\phi}_l = \phi_l \quad l \neq i, n.$$

Using the basis we can now represent functions in the subspace as $\mathrm{FE}_{\tilde{\Phi}_h}(\tilde{\mathbf{u}}) = \sum_{i \in \tilde{\mathcal{I}}_{U_h}} \tilde{\mathbf{u}}_i \tilde{\phi}_i$ with $\tilde{\mathbf{u}} \in \tilde{\mathbf{U}} = \mathbb{R}^{\tilde{\mathcal{I}}_{U_h}}$. Due to the basis transformation (7) we have the relation

$$\mathrm{FE}_{\tilde{\Phi}_h}(\tilde{\mathbf{u}}) = \mathrm{FE}_{\Phi_h}(\mathbf{S}_{U_h}^T \tilde{\mathbf{u}}) \tag{8}$$

with the rectangular block matrix

$$\mathbf{S}_{U_h} = \begin{pmatrix} \mathbf{I} & \mathbf{T}_{\tilde{U}_h} \end{pmatrix}. \tag{9}$$

Here we have assumed that the indices in $\mathcal{I}_{U_h}$ are ordered such that all indices in $\tilde{\mathcal{I}}_{U_h}$ are smaller than those in $\bar{\mathcal{I}}_{U_h}$. Note that the test space V_h is decomposed in a similar way in case it is different from the trial space.

3.4. Systems of PDEs

For systems of PDEs we need products of function spaces, as has been illustrated in Example 2.5. Formally, given $m > 1$ function spaces $U_h^{(0)}, \ldots, U_h^{(m-1)}$ we define the composite function space

$$U_h = U_h^{(0)} \times U_h^{(1)} \times \ldots \times U_h^{(m-1)}. \tag{10}$$

If all component spaces are the same, we can also write

$$U_h = V_h^m. \tag{11}$$

This can be done recursively leading to a tree structure of discrete function spaces.

As an example consider the Taylor-Hood element for solving the Stokes equations which we can write in d space dimensions as

$$U_h^{\mathrm{TH}} = \left(U_h^2\right)^d \times U_h^1,$$

meaning that each velocity component is piecewise quadratic and the pressure is piecewise linear. To construct such a function space in PDELAB we first make two grid function spaces for velocity components and pressure:

```
typedef GridView::ctype D; // coordinate type
typedef double R;          // value type
typedef Pk3DLocalFiniteElementMap<GridView,D,R,1> P1LFEM;
typedef Pk3DLocalFiniteElementMap<GridView,D,R,2> P2LFEM;
P1LFEM p1lfem(gridview);
P2LFEM p2lfem(gridview);
typedef GridFunctionSpace<GridView,P1LFEM> P1GFS;
typedef GridFunctionSpace<GridView,P2LFEM> P2GFS;
P1GFS p1gfs(gridview,p1lfem);
P2GFS p2gfs(gridview,p2lfem);
```

Now the class template `PowerGridFunctionSpace` produces a new grid function space out of a compile-time given number of grid function spaces of the same type:

```
const int dim=GridView::Grid::dimension;
typedef PowerGridFunctionSpace<P2GFS,dim> VGFS;
VGFS vgfs(p2gfs);
```

Using class template `CompositeGridFunctionSpace` one can make a new grid function space from given grid function spaces of different types:

```
typedef CompositeGridFunctionSpace<
  GridFunctionSpaceLexicographicMapper,
  VGFS,P1GFS> THGFS;
THGFS thgfs(vgfs,p1gfs); // the Taylor--Hood space
```

The template parameter `GridFunctionSpaceLexicographicMapper` indicates that the degrees of freedom are concatenated in the order given. This is also the default in the power version. If all component spaces have the same size, a cyclic numbering can be selected as well. The new grid function space can be used as before. E.g. the following code instantiates a random access container holding all degrees of freedom:

```
typedef THGFS::VectorContainer<R>::Type X;
X x(thgfs,0.0);
```

The tree structure of the grid function space is encoded in its recursive type. Using template metaprogramming [18] one can iterate over the constituents of the grid function space.

3.5. Parallelization support

The DUNE grid interface provides support for two parallelization models: nonoverlapping and overlapping. The grid function space provides means for the exchange of degrees of freedom stored in more than one process. The exchange mechanism

can be parametrized in a flexible way with respect to which data is exchanged, what data is sent and what is done when data is received.

When the iterative solver template library is used as a solver, certain types of solvers such as overlapping Schwarz methods or Krylov subspace methods (CG, BiCGStab, GMRES) can be used without any additional programming effort. Preconditioners on nonoverlapping grids are not yet available in a generic way.

4. GRID OPERATORS

4.1. Algebraic form of the constrained problem

We now come back to the solution of the weighted residual problem introduced in (1) which reads:

$$\text{find } u_h \in w_h + \tilde{U}_h: \qquad r_h(u_h, v) = 0 \qquad \forall\, v \in \tilde{V}_h.$$

To solve it, we introduce the basis representation of the subspaces from Section 3.3 and the equivalent problem in terms of coefficients:

$$\text{find } \tilde{\mathbf{u}} \in \tilde{\mathbf{U}}: \qquad r_h(\mathrm{FE}_{\Phi_h}(\mathbf{w}) + \mathrm{FE}_{\tilde{\Phi}_h}(\tilde{\mathbf{u}}), \tilde{\psi}_i) = 0 \qquad \forall\, i \in \tilde{\mathcal{I}}_{V_h}. \tag{12}$$

Employing the residual map introduced in (3) and the matrices $\mathbf{S}_{U_h}$, $\mathbf{S}_{V_h}$ from (9) this is then equivalent to the solution of the nonlinear algebraic system

$$\text{find } \tilde{\mathbf{u}} \in \tilde{\mathbf{U}}: \qquad \mathbf{S}_{V_h}\mathcal{R}(\mathbf{w} + \mathbf{S}_{U_h}^T\tilde{\mathbf{u}}) = \mathbf{0}. \tag{13}$$

It is important to note that the residual map $\mathcal{R}$, in part to be provided by the user, is evaluated with respect to the original basis Φ_h, Ψ_h which *does not* involve the constraints!

4.2. Newton's method

The nonlinear algebraic system (13) is now solved with Newton's method. To that end assume that some approximate solution $\tilde{\mathbf{u}}^k$ is available. We seek an update $\tilde{\mathbf{z}}^k$ such that the next iterate $\tilde{\mathbf{u}}^{k+1} = \tilde{\mathbf{u}}^k + \tilde{\mathbf{z}}^k$ is an improved approximation. Linearization of the equation $\mathbf{S}_{V_h}\mathcal{R}(\mathbf{w} + \mathbf{S}_{U_h}^T\tilde{\mathbf{u}}^{k+1}) = \mathbf{0}$ yields an equation for the update:

$$\mathbf{S}_{V_h}\mathcal{R}(\mathbf{w} + \mathbf{S}_{U_h}^T\tilde{\mathbf{u}}^k) + \mathbf{S}_{V_h}\nabla\mathcal{R}(\mathbf{w} + \mathbf{S}_{U_h}^T\tilde{\mathbf{u}}^k)\mathbf{S}_{U_h}^T\tilde{\mathbf{z}}^k = 0$$

where we introduced the Jacobian $\nabla\mathcal{R}$ of the nonlinear map $\mathcal{R}$. Together this results in the iteration

$$\tilde{\mathbf{u}}^{k+1} = \tilde{\mathbf{u}}^k - \left(\mathbf{S}_{V_h}\nabla\mathcal{R}(\mathbf{w} + \mathbf{S}_{U_h}^T\tilde{\mathbf{u}}^k)\mathbf{S}_{U_h}^T\right)^{-1}\mathbf{S}_{V_h}\mathcal{R}(\mathbf{w} + \mathbf{S}_{U_h}^T\tilde{\mathbf{u}}^k). \tag{14}$$

In this iteration scheme the solution vector $\tilde{\mathbf{u}}$ contains coefficients with respect to the transformed basis $\tilde{\Phi}_h$. Multiplication of (14) with $\mathbf{S}_{U_h}^T$ from the left and addition of $\mathbf{w}$ on both sides yields

$$\mathbf{w} + \mathbf{S}_{U_h}^T\tilde{\mathbf{u}}^{k+1} = \mathbf{w} + \mathbf{S}_{U_h}^T\tilde{\mathbf{u}}^k - \mathbf{S}_{U_h}^T\left(\mathbf{S}_{V_h}\nabla\mathcal{R}(\mathbf{w} + \mathbf{S}_{U_h}^T\tilde{\mathbf{u}}^k)\mathbf{S}_{U_h}^T\right)^{-1}\mathbf{S}_{V_h}\mathcal{R}(\mathbf{w} + \mathbf{S}_{U_h}^T\tilde{\mathbf{u}}^k)$$

where we can now set $\mathbf{u}^k := \mathbf{w} + \mathbf{S}_{U_h}^T \tilde{\mathbf{u}}^k$ to get the final form of the nonlinear iteration:

$$\mathbf{u}^{k+1} = \mathbf{u}^k - \mathbf{S}_{U_h}^T \left(\mathbf{S}_{V_h} \nabla \mathcal{R}(\mathbf{u}^k) \mathbf{S}_{U_h}^T \right)^{-1} \mathbf{S}_{V_h} \mathcal{R}(\mathbf{u}^k). \tag{15}$$

Note that (15) is now an iteration where the solution $\mathbf{u}$ is computed with respect to the original basis Φ_h. The residual evaluation $\mathcal{R}(\mathbf{u}^k)$ and the evaluation of the Jacobian $\nabla \mathcal{R}(\mathbf{u}^k)$ are also with respect to the original basis and are identical to the unconstrained case. Thus the incorporation of the constraints can be done in a completely generic way by the PDELab framework. The algorithmic formulation of the Newton scheme is given as follows:

Algorithm 4.1 (Newton's method for constrained problem)**.** Let the initial guess $\mathbf{u}^0$ with $\mathrm{FE}_{\Phi_{U_h}}(\mathbf{u}^0) \in w_h + \tilde{U}_h$ be given. Iterate until convergence

1. Compute residual: $\mathbf{r}^k = \mathcal{R}\left(\mathbf{u}^k\right)$.
2. Transform residual: $\tilde{\mathbf{r}}^k = \mathbf{S}_{V_h} \mathbf{r}^k$.
3. Solve the update equation in the transformed basis:
$$\left(\mathbf{S}_{V_h} \nabla \mathcal{R}(\mathbf{u}^k) \mathbf{S}_{U_h}^T \right) \tilde{\mathbf{z}}^k = \tilde{\mathbf{r}}^k.$$
4. Transform update to original basis: $\mathbf{z}^k = \mathbf{S}_{U_h}^T \tilde{\mathbf{z}}^k$. (This is where e. g. interpolation to hanging nodes is done).
5. Update solution: $\mathbf{u}^{k+1} = \mathbf{u}^k - \mathbf{z}^k$.

4.3. Generic assembler

As has been stated in Observation 2.6 the residual form can be split into contributions from elements, interior faces and boundary faces. Moreover, we can split the residual form into $r_h(u, v) = \alpha(u, v) + \lambda(v)$ where λ is only allowed to depend on the test function. The splitting of the residual form r_h results into a corresponding splitting of the nonlinear residual map $\mathcal{R}$ and its Jacobian $\nabla \mathcal{R}$. The splitting of the nonlinear residual map e. g. reads

$$\begin{aligned}
\mathcal{R}(\mathbf{u}) = &\sum_{e \in E_h^0} \mathbf{R}_e^T \boldsymbol{\alpha}_{h,e}^{\mathrm{vol}}(\mathbf{R}_e \mathbf{u}) &&+ \sum_{e \in E_h^0} \mathbf{R}_e^T \boldsymbol{\lambda}_{h,e}^{\mathrm{vol}} \\
&+ \sum_{f \in E_h^1} \mathbf{R}_{l(f),r(f)}^T \boldsymbol{\alpha}_{h,f}^{\mathrm{skel}}(\mathbf{R}_{l(f),r(f)} \mathbf{u}) &&+ \sum_{f \in E_h^1} \mathbf{R}_{l(f),r(f)}^T \boldsymbol{\lambda}_{h,f}^{\mathrm{skel}} \\
&+ \sum_{b \in B_h^1} \mathbf{R}_{l(b)}^T \boldsymbol{\alpha}_{h,b}^{\mathrm{bnd}}(\mathbf{R}_{l(b)} \mathbf{u}) &&+ \sum_{b \in B_h^1} \mathbf{R}_{l(b)}^T \boldsymbol{\lambda}_{h,b}^{\mathrm{bnd}},
\end{aligned}$$

where the restriction matrices $\mathbf{R}_x$ extract all degrees of freedom related to an element, interior face or boundary face which can be easily determined from the local-to-global map computed in the grid function space.

The implementor of a discretization scheme only has to provide at most the six methods $\boldsymbol{\alpha}^{\mathrm{vol}}$, $\boldsymbol{\alpha}^{\mathrm{skel}}$, $\boldsymbol{\alpha}^{\mathrm{bnd}}$, $\boldsymbol{\lambda}^{\mathrm{vol}}$, $\boldsymbol{\lambda}^{\mathrm{skel}}$ and $\boldsymbol{\lambda}^{\mathrm{bnd}}$ providing element, interior face and

boundary face contributions depending on solution and test functions ($\boldsymbol{\alpha}$-terms) or only on the test functions ($\boldsymbol{\lambda}$-terms). Contributions to the Jacobian can be generated through numerical differentiation for rapid prototyping. If an analytical Jacobian is required or a modified Newton method is desired additional methods have to be provided.

From these user given methods assembling of the residual and the Jacobian (stiffness matrix) as well as the complete Newton iteration with consideration of all types of constraints is provided in a completely generic way including parallel computations, with the exception of the preconditioner.

In order to give a code example for the implementation of a discretization scheme we consider the cell-centered finite volume method from example 2.3. The following class `Laplace` implements this method for the pure Neumann problem on axiparallel cube meshes in any dimension:

```
template<typename Real> class Laplace : public
NumericalJacobianApplySkeleton<Laplace<Real> >,
        public NumericalJacobianSkeleton<Laplace<Real> >,
        public FullSkeletonPattern, public FullVolumePattern,
        public LocalOperatorDefaultFlags
{ public:
  // pattern assembly flags
  enum { doPatternVolume = true }; enum { doPatternSkeleton = true };

  // residual assembly flags
  enum { doAlphaSkeleton  = true };

  template<typename IG, typename LFSU, typename X, typename LFSV, typename R>
  void alpha_skeleton (const IG& ig,
                const LFSU& lfsu_s, const X& x_s, const LFSV& lfsv_s,
                const LFSU& lfsu_n, const X& x_n, const LFSV& lfsv_n,
                R& r_s, R& r_n) const
  {
    // face volume for integration
    Real face_volume = ig.geometry().volume();

    // cell centers in global coordinates
    Dune::FieldVector<Real,IG::dimension>
      inside_global = ig.inside()->geometry().center();
    Dune::FieldVector<Real,IG::dimension>
      outside_global = ig.outside()->geometry().center();

    // distance between the two cell centers
    inside_global -= outside_global;
    Real distance = inside_global.two_norm();

    // contribution to residual on inside element
    r_s[0] += (x_s[0]-x_n[0])*face_volume/distance;
    r_n[0] -= (x_s[0]-x_n[0])*face_volume/distance;
  }
};
```

Table 2. Coding effort required to implement various discretization schemes for an elliptic model problem.

Method	Source lines of code
Conforming finite elements	298
Discontinuous Galerkin	914
Cell-centered finite volumes	222
Mixed finite elements	288
Mimetic finite differences	395

In that case only one term in (5), the interior skeleton term $\boldsymbol{\alpha}^{\text{skel}}$ is needed which is implemented in the method `alpha_skeleton`. This also illustrates the fact that simple methods can be implemented with very few lines of code which makes the system usable for teaching.

5. NUMERICAL RESULTS

In this Section we present some applications of the PDELab framework in order to demonstrate its flexibility and performance.

5.1. Six easy pieces

The elliptic problem (4) with tensor diffusion coefficient and Dirichlet as well as Neumann boundary conditions can be solved with the following methods:

- Conforming finite elements (arbitrary order for simplices and $d = 2, 3$, order 1 and 2 for hexahedral elements in $d = 2, 3$, hanging nodes for order 1).
- Discontinuous Galerkin finite elements (Oden–Babuška–Baumann, symmetric interior penalty method, nonsymmetric interior penalty method).
- Nonconforming Rannacher–Turek element in $d = 2$.
- Lowest order Raviart–Thomas (mixed) elements on triangles (non-hybrid version).
- Cell-centered finite volumes with harmonic permeability averaging (scalar diffusion coefficient only) in any dimension on axi-parallel hexahedral meshes.
- Mimetic finite difference method [6] on any mesh in any dimension.

Table 2 lists the number of lines of code required to implement the various discretization schemes in PDELab. It should be noted that all schemes, except the cell-centered finite volumes and lowest-order mixed method, implement also the analytical Jacobian. The table shows that all schemes can be implemented with a very low effort. Note also that this table does not include the source lines required to implement the function spaces. These are given in Table 1.

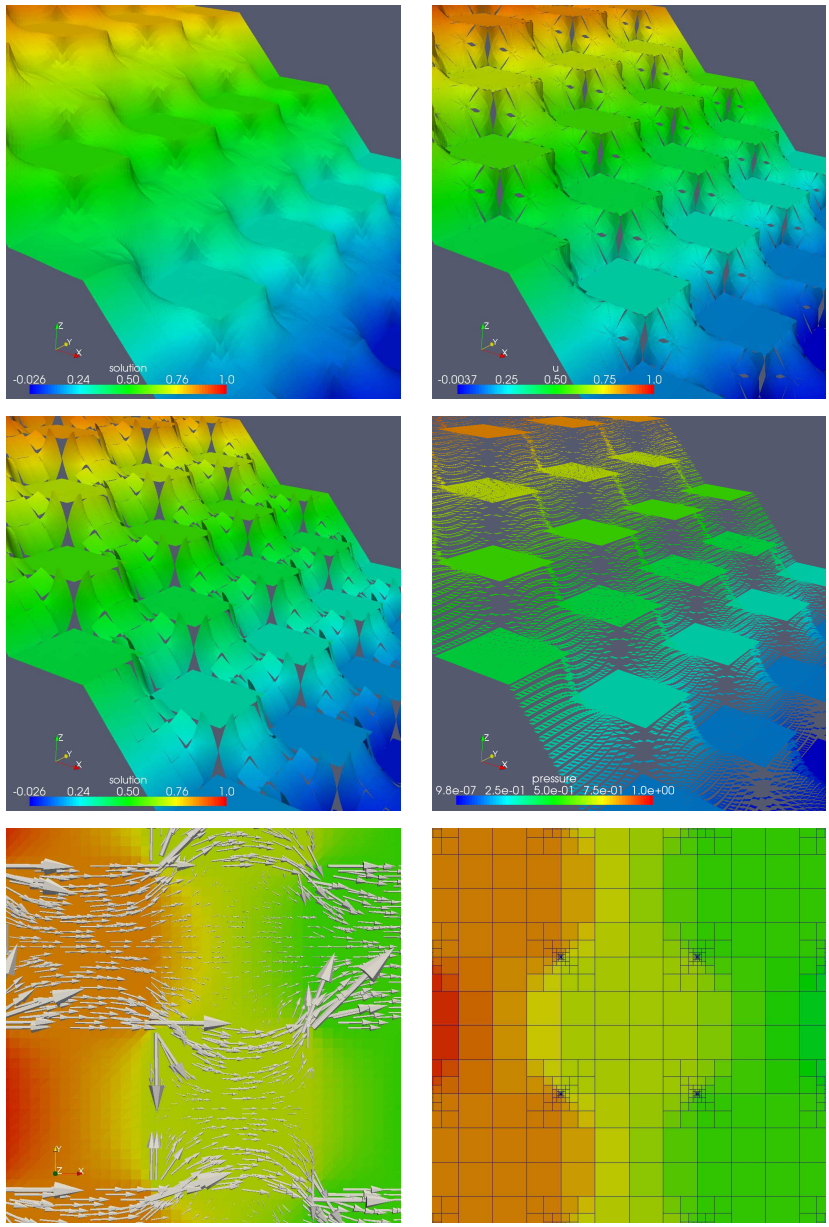

Fig. 3. Solution of problem (4) with checkerboard diffusion coefficient using various methods implemented in PDELab. From top left to bottom right: CG order 4, DG OBB order 4, Rannacher Turek, lowest order Raviart–Thomas (RT0) mixed elements, RT0 velocity field detail, mimetic finite difference method on mesh with hanging node refinement.

Figure 3 shows visualizations of the results obtained with the different schemes applied to a model problem with a checker-board coefficient distribution in two space dimensions.

5.2. Multiphase flow in porous media

In order to demonstrate the suitability of PDELAB for a more complicated problem, we solve the problem of two-phase immiscible flow in porous media, see e. g. [11] for an introduction. The phases considered are liquid (l) and gas (g) and the model consists of conservation of mass for each phase $\alpha \in \{l, g\}$:

$$\partial_t(\phi s_\alpha \nu_\alpha) + \nabla \cdot \{\nu_\alpha u_\alpha\} = q_\alpha,$$

where s_α is the phase saturation, ν_α is the molar density, u_α is the phase velocity and q_α is the source/sink term. The phase velocity depends on the pressure via the extended Darcy law

$$u_\alpha = -\frac{k_{r\alpha}(s_\alpha)}{\mu_\alpha} K \left(\nabla p_\alpha - \rho_\alpha g\right),$$

where the relative permeability $k_{r\alpha}$ depends nonlinearly on saturation, K is the absolute permeability, μ_α is the dynamic viscosity of the fluid, ρ_α is the mass density of the fluid and g is the gravity vector. In addition there are the following algebraic constraints for saturations and pressures:

$$s_l + s_g = 1 \qquad\qquad p_l + p_g = p_c(s_l),$$

where $p_c(s_l)$ is the strongly nonlinear capillary pressure saturation relation. Finally, for the gas phase we have the ideal gas law $\nu_g = p_g/RT$ and for the liquid phase $\nu_l = \text{const}$.

This system is solved in a so-called pressure-pressure formulation which amounts to a time-dependent, strongly nonlinear system of two coupled PDEs which is discretized using a cell-centered finite-volume scheme with harmonic averaging of permeabilities and upwinding of mobilities. In time, a fully-implicit Euler scheme is used. The resulting nonlinear algebraic system per time step is solved using Newton's method with line search globalization and the arising linear systems are solved with a BiCGStab Krylov subspace method with an overlapping inexact Schwarz preconditioner using a few steps of Block-SSOR as subdomain solver.

The code is fully parallelized and dimension-independent. The implementation of the model in PDELAB (i. e. the cell-centered finite volume discretization for the above-mentioned set of equations including an interface for the parameter functions) required 587 lines of code (including comments). The driver code defining the parameter functions, setting up the grid, selecting the solvers and making the time loop and nonlinear iteration took another 755 lines of code. In total, such a model can be implemented in a few days by an experienced programmer. Figure 4 shows a result of a simulation where an initially dry porous medium was put in contact with a liquid at the lower boundary and subsequently the water infiltrates from below.

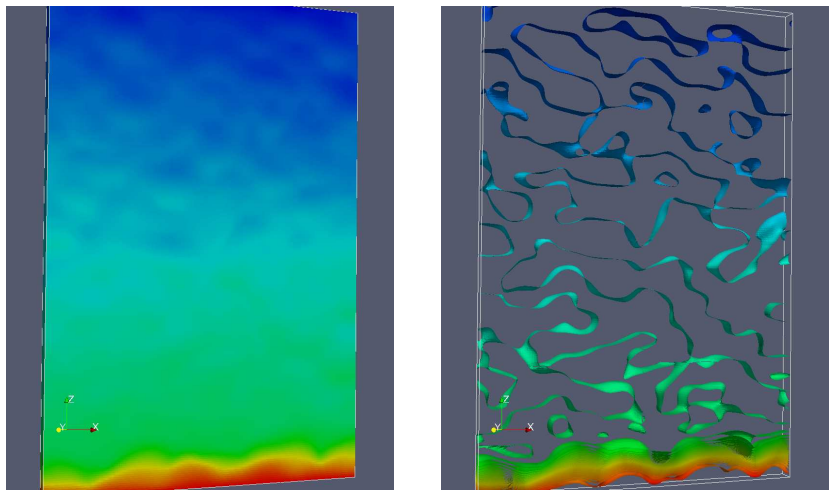

Fig. 4. Capillary rise of a fluid in a hele-shaw cell containing a coarse sand. Left image shows liquid phase pressure, right image shows contours of liquid phase saturation.

Nowadays multi-core CPUs are standard for various technical reasons. A major problem in using such CPUs for scientific computing is the limited memory bandwidth when all cores access main memory. Table 3 gives some performance numbers of DUNE and PDELAB on such architectures. We use a system with four quad-core AMD Opteron 8380 processors (2.5 GHz). Each core has 512KB L2 cache and all four cores in a CPU share a 6MB L3 cache. We solve one time step of the two-phase flow problem on a $160 \times 160 \times 96$ grid resulting in roughly 5 million degrees of freedom. The first two columns give the maximum number of iterations and the corresponding computation time of the overlapping Schwarz preconditioner (3 cells overlap) in any Newton step. Iteration numbers are robust with respect to number of processors and the speedup for *one* iteration (time in fourth column) on 16 processors is 9. The time for assembly of the Jacobian (next column) scales slightly better. Finally the speedup in total computation time is 8 on 16 processors.

5.3. Parallel solver example

Figure 5 shows results for the scalability of an additive geometric multigrid preconditioner, the so-called BPX method [5], implemented in DUNE. All multigrid components are implemented with sparse linear algebra classes from the iterative solver template library [1, 4] such that no access to the grid is necessary during multigrid cycles. The YaspGrid (a structured, parallel grid) implementation from `dune-grid` is used in two space dimensions. Strong scaling means that a 2048×2048 problem is solved on one up to 256 processors. The speedup attained on 256 processors is 124, i. e. almost 50% efficiency. In the weak scaling test each processor has a 1024×1024 grid and the problem size is increased proportional with the number of

Table 3. Strong scaling for two-phase flow problem of size $160 \times 160 \times 96$ on a 4×quad-core AMD Opteron 8380 system (2.5 GHz).

P	#IT(max)	T_{lin}(max)	T_{it}	T_{ass}	T_{total}	Speedup
1	40.5	264.5	6.5	39.2	1347.3	-
2	44.5	138.5	3.2	31.1	810.5	1.7
4	44.5	71.2	1.6	16.1	407.8	3.3
8	42.5	53.3	1.3	7.7	279.5	4.8
16	50.0	34.4	0.7	3.8	163.1	8.2

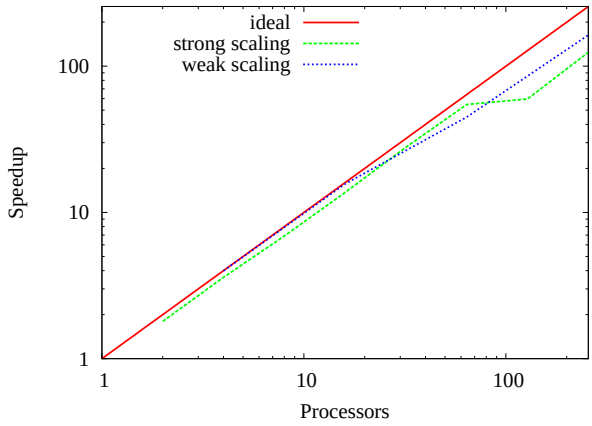

Fig. 5. Strong and weak scalability of an additive geometric multigrid preconditioner using YaspGrid on a Linux cluster (dual processor dual core AMD Opteron 2.8 GHz, Myrinet 10GBit interconnect).

processors. In this mode a speedup of 163 is achieved on 256 processors.

6. CONCLUSIONS

In this paper we presented software abstractions for the generic implementation of finite element methods. The system allows a very general definition of finite element spaces including higher order, continuous and discontinuous as well as scalar and vector-valued. Moreover, a general way for the incorporation of constraints is provided. In the examples it is shown that many schemes can be implemented with a low programming effort with the underlying DUNE framework providing dimension independence and parallelization. Learning how to use a system like PDELab takes a certain amount of time which strongly depends on the background and experience of the user. However, as soon as features like higher-order or parallelism are required this initial investment should pay off.

The current state of implementation of PDELAB encompasses the global finite

element spaces described in Section 3, including constraints and generic parallelism. The generic assembler for stationary problems and the Newton scheme are implemented as well. As the next steps generic support for adaptivity and time dependent problems will be integrated into PDELab.

ACKNOWLEDGEMENT

We wish to thank all Dune developers for their effort and support, Christoph Grüninger for the provision of the DG code, Jö Fahlke for the implementation of several finite element spaces. The support of StatoilHydro for the Dune project is also greatly acknowledged.

(Received March 3, 2010)

REFERENCES

[1] P. Bastian and M. Blatt: On the generic parallelisation of iterative solvers for the finite element method. Internat. J. Comput. Sci. Engrg. *4* (2008), 1, 56–69.

[2] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander: A generic grid interface for parallel and adaptive scientific computing. Part I: Abstract framework. Computing *82* (2008), 2-3, 103–119.

[3] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander: A generic grid interface for parallel and adaptive scientific computing. Part II: Implementation and tests in DUNE. Computing *82* (2008), 2-3, 121–138.

[4] M. Blatt and P. Bastian: The iterative solver template library. In: Applied Parallel Computing. State of the Art in Scientific Computing (B. Kagström, E. Elmroth, J. Dongarra, and J. Wasniewski, eds.) (Lecture Notes in Sci. Comput. *4699*.) Spinger, Berlin 2007, pp. 666–675.

[5] J. H. Bramble, J. E. Pasciak, and J. Xu: Parallel multilevel preconditioners. Math. Comput. *55* (1990), 1–22.

[6] F. Brezzi, K. Lipnikov, and V. Simoncini: A family of mimetic finite difference methods on polygonal and polyhedral meshes. Math. Models and Methods in Applied Sciences *15* (2005), 10, 1533–1551.

[7] P. G. Ciarlet: The Finite Element Method for Elliptic Problems. SIAM, Philadelphia 2002.

[8] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger: A generic interface for parallel and adaptive scientific computing: Abstraction principles and the Dune-Fem module. Preprint No. 3, Mathematisches Institut, Universität Freiburg, 2009. Submitted to Transactions on Mathematical Software.

[9] http://www.dune-project.org/, Dune Homepage, link visited August 3, 2009.

[10] C. Geuzaine and J.-F. Remacle: Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. Internat. J. Num. Methods in Eng., 2009. http://www.geuz.org/gmsh/, link visited August 3, 2009.

[11] R. Helmig: Multiphase Flow and Transport Processes in the Subsurface – A Contribution to the Modeling of Hydrosystems. Springer–Verlag, 1997.

[12] J. T. Oden, I. Babuška, and C. E. Baumann: A discontinuous *hp* finite element method for diffusion problems. J. Comput. Phys. *146* (1998), 491–519.

[13] http://www.opencascade.com/, link visited August 3, 2009.

[14] http://www.paraview.org/, link visited August 3, 2009.

[15] http://www.mcs.anl.gov/petsc/petsc-as/, link visited August 5, 2009.

[16] http://www.salome-platform.org/, link visited August 3, 2009.

[17] http://trilinos.sandia.gov/, link visited August 5, 2009.

[18] D. Vandevoorde and N. M. Josuttis: C++ Templates – The Complete Guide. Addison-Wesley, 2003.

Peter Bastian, IWR, Im Neuenheimer Feld 368, D-69120 Heidelberg. Germany.
e-mail: peter.bastian@iwr.uni-heidelberg.de

Felix Heimann, IWR, Im Neuenheimer Feld 368, D-69120 Heidelberg. Germany.
e-mail: felix.heimann@iwr.uni-heidelberg.de

Sven Marnach, IWR, Im Neuenheimer Feld 368, D-69120 Heidelberg. Germany.
e-mail: sven.marnach@iwr.uni-heidelberg.de