The dune-subgrid Module and Some Applications

Carsten Gräser and Oliver Sander

March 2, 2009

Abstract

We present an extension module for the DUNE system. This module, called `dune-subgrid`, allows to mark elements of another DUNE hierarchical grid. The set of marked elements can then be accessed as a DUNE grid in its own right. `dune-subgrid` is free software and is available for download [15]. We describe the functionality and use of `dune-subgrid`, comment on its implementation, and give two example applications. First, we show how `dune-subgrid` can be used for micro-FE simulations of trabecular bone. Then we present an algorithm that allows to use exact residuals for the adaptive solution of the spatial problems of time-discretized evolution equations.

1 Introduction

The DUNE system is a set of C++ libraries for solving partial differential equations using grid-based numerical methods [12]. It is split up in several *modules*, with different modules containing different aspects, such as grids, linear algebra, or finite element discretizations. The modules are tied together by a powerful build system, which tracks and resolves inter-module dependencies.

The set of modules is not fixed. A main rationale for the modular design was to allow third-party users to extend DUNE by implementing and providing further modules. These can either provide additional functionality or alternative implementations of existing one. One example is the `dune-fem` module, which is maintained separately from the core DUNE system, and which provides finite element discretizations [13].

The central feature of DUNE is a set of abstract interfaces to its components. This is best exemplified by the `dune-grid` module. Based on a precise mathematical definition of a grid [5], this module contains a set of C++ classes which form a well-defined gateway to grid functionality. Using grids by means of this interface gives the application writer an unseen amount of flexibility, as the grid implementation can be changed with no effort at any point in the development process. While `dune-grid` provides several implementations of the grid interface itself, several legacy grid managers such as UG [3] and ALBERTA [20] have also been adapted to be usable through the interface. The use of generic programming techniques ensures a low run-time overhead of the interface.

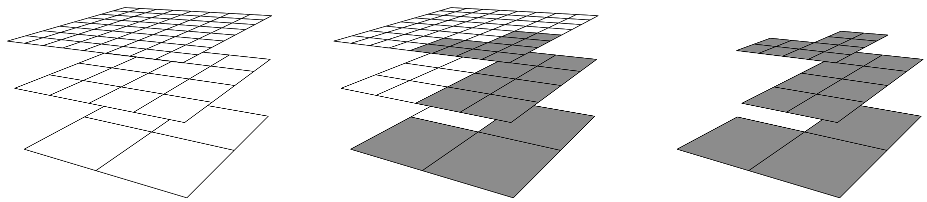

Figure 1: Functionality of `SubGrid`. Left: Host grid; center: host grid with some elements marked. Right: marked elements as a separate grid.

The Dune grid interface was designed to support geometric multigrid and locally adaptive algorithms, and hence its notion of a 'grid' directly contains a hierarchical structure. A Dune grid consists of a finite set of *level grids*, which are connected by a *father relation*. Each level grid in turn consists of an *entity complex* together with a *geometric realization*. While the former represents the combinatorial aspects of a grid, the latter specifies its shape by describing an embedding into a Euclidean space.

Together with the father relation, the grid entities form a forest structure. Under certain conditions on the geometric realizations, the leafs of this forest form the *leaf grid*. This is the natural grid for nonhierarchical methods on a locally adaptive grid. We refer the reader to [5] for a precise definition.

The abstract definition carries over directly into C++ classes [4]. The implementation uses wrapper classes which delegate method calls to engine classes provided as template parameters which do the actual work [22]. Access to the entities is provided by STL-style iterators.

Actual grids that implement this interface can be provided in several ways. The simplest way is to write a grid manager from scratch and make sure that it is accessible via the grid interface methods. Such implementations are, e.g., the structured grid `YaspGrid` and the one-dimensional adaptive grid `OneDGrid` provided in `dune-grid`. A second possibility is to write adapter code which wraps existing legacy grid managers behind the interface. The well-known FE codes UG [3] and Alberta [20] are made available this way [4]. A third way are *meta grids*. These are implementations of the grid interface that are statically parametrized with another Dune grid. They provide extensions and modifications of their parameter grid, or *host grid*. Meta grids are an extremely powerful concept, and several such grids are already available [15].

The `dune-subgrid` module provides the meta grid `SubGrid`. Given a Dune grid object, `SubGrid` allows to mark a subset of elements of this host grid and treat the subset as a Dune grid in its own right (Fig. 1). Consequently, existing algorithms can be made to run on subsets of grids with minimal effort. If the host grid is hierarchical, so is the subgrid, and ancestor elements are set automatically to ensure a consistent hierarchical grid. The subgrid can be adaptively refined and then takes up more elements of the host grid. If the host grid does not contain the necessary elements for this it is refined in turn. Hence the subgrid can be used completely transparently.

Depending on the application several ways to implement such functionality are conceivable. Therefore, `SubGrid` allows to exchange part of the internal memory management using policy classes.

Computing on a subset of grid elements is not a new concept. We mention the existing work on μFE simulations [21] and narrow band methods [11]. To our knowledge, however, special purpose implementations were used there which closely tied the host grid and subgrid implementations together. This is in contrast to our approach, which allows to change the host grid implementation at any time during development. The added flexibility opens the way to new applications.

This paper is organized in four main chapters. Chapter 2 explains the use and functionality of `SubGrid`, whereas Chapter 3 describes its implementation. The last two chapters give two example applications. Chapter 4 shows how `SubGrid` can be used to solve a large scale linear elasticity problem on segmented image data of human trabecular bone. Finally, in Chapter 5, `SubGrid` is used to keep an exact previous time step on an evolution problem on grids that change with time.

2 Using the `dune-subgrid` Module

We begin by describing the use of `SubGrid`. Since descriptions of the Dune grid interface are available elsewhere [4, 5, 12], we concentrate on the aspects that are specific to `SubGrid`.

2.1 Subgrid Construction

Let `HostGridType` be a C++ type that implements the Dune grid interface and `hostgrid` an object of type `HostGridType`. A subgrid object is created by calling the constructor

```
SubGrid<dim,HostGridType,MapIndexStorage> subgrid(hostgrid);
```

The first template parameter specifies the dimension of the subgrid. Currently, only subgrids with the same dimension as the host grid are supported. The third template parameter selects one of two possible implementations of the subgrid index sets (see Sec. 3.2).

The new subgrid object does not contain any elements of the host grid. To start adding elements first call

```
subgrid.createBegin();
```

This initializes the internal data structures. Previous element marks are deleted. To actually add elements there are the following methods:[1]

[1]We abbreviate the method signatures for increased legibility. The precise definitions can be found in the html class documentation provided with the module.

`void insert (const HostGrid::Codim<0>::Entity &e)`
: Adds the host grid element `e` to the subgrid. If the element is not on the coarsest grid level, all its ancestor elements and their direct sons are added as well. That way, consistency of the grid data structure is ensured.

`void insertPartial (const HostGrid::Codim<0>::Entity &e)`
: Adds the given element and all its ancestors to the subgrid. However, unlike `insert`, it does not add the direct sons of the ancestors. While this violates the DUNE grid specification (element children have to form a logical partition of their father [5, Def. 11.1]), it leads to consistent subgrid level grids and can be useful in certain applications (see Sec. 4).

`void insertRaw (const HostGrid::Codim<0>::Entity &e)`
: This method adds only the given element to the subgrid. Ancestors are not inserted automatically. This allows to speed up the construction by inserting ancestors once by hand instead of multiple times automatically if they are known in advance. Omitting ancestors leads to an inconsistent subgrid state and undefined behavior.

`template<class Container>`
`void insertSet (const Container &idContainer)`
: This adds all host grid elements to the subgrid whose ids [4, Sec. 3.2] are contained in `idContainer`. This is a lot faster than entering them one by one. For each element, all ancestors and their direct children are entered as well.

`void insertLevel(int level)`
: Inserts all level grids whose levels are less than or equal to the given level.

`void insertLeaf()`
: Inserts the entire host grid hierarchy into the subgrid.

After you have inserted the elements you call

```
subgrid.createEnd();
```

This finishes off the subgrid creation. You are now ready to use `SubGrid` as you would any other DUNE grid.

2.2 Grid Adaptation

`SubGrid` provides the same interface for adaptation as other DUNE grids. A `SubGrid` is refined by adding more elements from the host grid. If necessary, the host grid is refined as well to make this possible. Three things happen when the method `adapt()` is called for a subgrid:

1. If a subgrid element is marked for refinement and the corresponding host grid element is not a leaf element, then the host grid children are added to the subgrid.

2. If a subgrid element is marked for refinement and the corresponding host grid element is a leaf element, then this host grid element is refined and its newly created children are added to the subgrid.

3. If a subgrid element is marked for coarsening it is simply marked as not part of the subgrid. The corresponding host grid element is not touched.

In order to allow preserving grid functions while adapting grids the DUNE grid interface has the methods `preAdapt()` and `postAdapt()`. These set certain element marks that signal where data needs to be projected to coarser or finer levels when the grid changes. The two methods on the subgrid call the corresponding methods of the host grid if the latter is refined as a consequence of subgrid refinement. This case can be detected using the method

`bool hostGridAdapted()`
: Returns `true` if `preAdapt()/adapt()/postAdapt()` was called for the host grid during the last call to the corresponding subgrid method.

A subgrid's leaf grid will in general be nonconforming and contain hanging nodes even if the host grid itself is conforming. Control over the nonconformity is possible by

`void setMaxLevelDifference(int maxLevelDifference)`
: Sets the maximal level difference of leaf elements sharing a node. The default value is 1 ensuring that only first-order hanging nodes appear. This is implemented by refining additional elements if necessary.

No coarsening is applied to the host grid automatically to prevent loss of data for functions attached to it. However, if there are no functions attached to the host grid it can be useful to have a minimal host grid for the given subgrid. This is achieved by a call to

`void shrinkHostGrid(int maxAdaptations, bool recreateSubgrid)`
: Coarsens the host grid as much as possible without influencing the subgrid, and without removing more than `maxAdaptations` levels. This change corrupts the subgrid data structure, which the method has to recreate from scratch after having modified the host grid. If the subgrid is to be discarded anyways this behavior can be switched off by setting `recreateSubgrid` to `false`. Note that subgrid indices may have changed after a call to this method.

2.3 Data Transfer Between Subgrid and Host Grid

Applications that only involve computations on the `SubGrid` can be implemented using the DUNE interface and the methods described above. However, sometimes you may also want to do computations both on the subgrid and on the host grid (see Sec. 5). Then you need to be able to transfer data from the host grid to the subgrid and back. `SubGrid` provides several methods for this:

```
template <int codim>
HostGrid::Codim<codim>::Entity&
getHostEntity (const SubGrid::Codim<codim>::Entity &e)
```

Given a `SubGrid` entity this method provides you with the corresponding host grid entity. You can then use this entity, e.g., to access host grid index sets.

```
template <int codim>
SubGrid::Codim<codim>::Entity&
getSubGridEntity(const HostGrid::Codim<codim>::Entity &e)
```

This is the other way around. For a given host grid entity you get the corresponding subgrid entity. If the entity is not present in the subgrid a `Dune::GridError` exception is thrown.

To check beforehand whether a given host grid entity exists in the subgrid there is the following method:

```
bool contains(const HostGrid::Codim<codim>::Entity &e)
```

Returns `true` if the given host grid entity exists in the subgrid.

The transfer of entire grid functions is facilitated by the method

```
template<class ElementTransfer>
void transfer(ElementTransfer &elementTransfer)
```

Provides the possibility to transfer data between the subgrid leaf view and the host grid leaf view. The class `ElementTransfer` has to be written by the user and is expected to provide the actual transfer functionality for pairs of elements. When `transfer` is called, the following three things happen:

1. First `elementTransfer.pre()` is called.
2. Then `elementTransfer.transfer(se,he)` is called for each subgrid leaf element `se` and host grid leaf element `he` such that `he` is a descendant of the host grid element corresponding to `se` or the host grid element corresponding to `se` itself.
3. Finally `elementTransfer.post()` is called.

This mechanism allows, e.g., to interpolate discrete functions on the subgrid onto the host grid or to restrict functionals on host grid function spaces to subgrid function spaces. The helper classes `SubGridP1Interpolator` and `SubGridP1Restrictor` provide this functionality for vectors representing piecewise linear finite element functions and functionals on the space of these functions.

3 Implementation

The implementation of the `SubGrid` class is based on a set of `std::vector<bool>` containers. There is a container for each codimension and grid level. An index created from the geometry type and level index of each host grid entity is used to index these containers. The i-th entry of such a vector is `true` if the corresponding host grid entity is contained in the subgrid. Only elements, i.e., entities of codimension 0 are explicitly marked during subgrid creation. The subentities of the marked elements are added automatically by the `createEnd` method.

3.1 Iterators

Many calls to subgrid methods can simply be forwarded to the host grid. The `SubGridLevelIterator` traversing the entities of a fixed codimension in a subgrid level uses a host grid level iterator to iterate over the entire host grid level, stopping only at those entities contained in the subgrid. Similarly the `SubGridHierarchicIterator` uses a `HierarchicIterator` of the host grid stopping only at the descendant elements contained in the subgrid.

The `SubGridLeafIterator` traversing the entities of a fixed codimension on the subgrid leaf also uses the `LevelIterator` of the host grid. It loops over all host grid levels from 0 to the maximal level number of the subgrid and stops at the entities that are leafs in the subgrid. Information whether a given subgrid entity is a leaf of the subgrid and whether the `SubGridLeafIterator` should stop there is provided by the `SubGridIndexStorage` class described in the next section.

3.2 Index Sets

Implementing `Leaf-` and `LevelIndexSets` is nontrivial for `SubGrid`. Remember that sequences of indices have to be consecutive and start at 0 [4, Sec. 3.2]. The subgrid entities do not form any simple pattern within the host grid, and hence the subgrid indices cannot be computed from the host grid index and local information alone. Subgrid indices are therefore computed all at once by the `createEnd` method and stored in a dedicated data structure. There are two different implementations for this structure, which can be selected by a template parameter.

The `SubGridMapIndexStorage` stores a `std::map` for each codimension mapping global ids of entities in the subgrid to objects of type `SubGridMultilevel-Index`. Since copies of entities on different levels have the same global id these objects store the minimal and maximal level an entity appears on. Furthermore they store the leaf index if an entity with the respective global id is contained in the leaf grid. If this is not the case the leaf index is set to -1. Depending on whether minimal and maximal level coincide either the index for a single level or a pointer to a list of indices for a range of levels is stored. The method `isLeaf` returns `true` if a leaf index is stored in the `SubGridMapIndexStorage` object. A leaf iterator stops at an entity if it has a leaf index and if its level is the maximal level it appears on. This is because copies of an entity with positive codimension may all be leaf, but the iterator should stop only at the one with the highest level.

The SubGridVectorIndexStorage stores a vector of level indices for each host grid level and geometry type. These indices are not used if the corresponding entity is not contained in the subgrid. To store leaf indices in a memory efficient way the subgrid leaf entities are divided in two categories:

1. Leaf entities that are copies of subgrid leaf entities on lower grid levels,
2. Leaf entities that do not have copies on lower grid levels.

The level indices are distributed such that first the leaf entities of Category 1 are counted, then the leaf entities of Category 2, and finally the non-leaf entities. In order to determine the leaf index of a given subgrid entity it is first decided whether the entity belongs to Category 1 by checking if its level index is smaller than the number of Category 1 leaf entities. If this is the case the leaf index is looked up in a vector storing the leaf indices indexed with the subgrid level index explicitly for each level and geometry type. Otherwise, the leaf index is just the level index plus a fixed offset which is the difference of the number of leaf entities on lower levels and the number of Category 1 leaf entities on the entity's level. While leaf indices are needed for all copies of leaf entities the leaf iterator should only traverse the copies of leaf entities with the highest level. To store this information an additional `std::vector<bool>` for each subgrid level and geometry type indexed with the subgrid level index is used.

3.3 Leaf Intersection Iterators

A correct implementation of leaf intersection iterators is another challenge in SubGrid. Remember that a subgrid leaf grid may be nonconforming even though the underlying host grid is not. Therefore, while each subgrid level intersection is also an intersection within the host grid, a subgrid leaf intersection may not be, and SubGrid cannot simply forward calls to host grid intersections but has to compute and maintain data structures for subgrid leaf intersections including all information on geometry.

For efficiency reasons leaf intersections are not computed one-by-one as the iterator advances. Instead, each time the iterator reaches a new element face, all intersections of this face are precomputed and stored in a list. The iterator then traverses this list before moving to the next element face. Intersections of a given face are precomputed by traversing the refinement hierarchy to find the second elements of the intersections.

We explain this in some more detail. Internally, the SubGridLeafIntersectionIterator keeps a HostGridLevelIntersectionIterator and iterates over all host grid level intersections of an element. Let `is` be the current host grid level intersection, `inside` the element we constructed the HostGridLevelIntersectionIterator from, and `outside` the second element of the HostGridLevelIntersectionIterator. At each such intersection `is` one of the following cases happens:

- If `is` is a level boundary intersection of the host grid, then `is` is a leaf boundary intersection of the subgrid.

- If `outside` is contained in the subgrid and is leaf there, then `is` is a subgrid leaf intersection.
- If `outside` is contained in the subgrid but is not leaf there, then `inside` has one or more subgrid leaf intersections with descendants of `outside`. Hence we traverse these leaf descendants and collect all intersections with `inside` in a list.
- If `outside` exists but is not contained in the subgrid, then either `is` is a boundary intersection or there is a single intersection of `inside` with the first ancestor `anc` of `outside` that is contained in the subgrid. We find it by following ancestor faces of the face of `is` until we arrive at the level of `anc`.

Each step produces a list of intersections, possibly containing only one or zero elements. The `SubGridLeafIntersectionIterator` then steps through this list. Once the list is exhausted the `HostGridLevelIterator` is incremented and the process is repeated.

This algorithm for the construction of subgrid leaf intersections is problematic because the Dune grid interface does not directly support hierarchical iteration over faces. We simulate a hierarchical face iterator using a combination of a hierarchical element iterator, a level intersection iterator, and a primitive that checks whether a given face is father of a given other face. However, such a simulation is slow. Also, with only the Dune grid interface methods available, the is-father primitive has, to a certain extent, to depend on geometrical information, and may in theory fail for host grids with certain exotic refinement rules.

3.4 Grid Adaptation

The concept of adaptive refinement for `SubGrid` is simple. If a subgrid leaf element is not leaf within the host grid, then refinement means adding the host grid children to the subgrid. If, on the other hand, it is leaf, the host grid has to be refined appropriately to allow subgrid refinement.

The main difficulty of the `SubGrid` adaptivity mechanism is to comply with an additional restriction. It has been pointed out that an adaptively refined subgrid may be nonconforming even though the host grid is not. Since some finite element codes can only handle simple cases of grid nonconformity, the `SubGrid` implementation allows the user to limit the degree of nonconformity with the `setMaxLevelDifference` method (see Sec. 2.2).

If subgrid leaf elements are marked for refinement or coarsening this information is stored in `std::vector<bool>` containers for each subgrid level. This raw information is used by the `preAdapt` method to determine which host grid elements need to be refined to allow the requested subgrid refinement. Let $L_\Delta^{\max}$ be the parameter set by the `setMaxLevelDifference` method. Calling `preAdapt` on `SubGrid` performs the following steps:

1. For each subgrid node p the maximal level l_p^{raw} of the adjacent elements after the (hypothetic) requested refinement and coarsening is determined.

2. For each subgrid leaf element e the maximal level

$$l_e^{\text{raw}} = \max\{l_p^{\text{raw}} : p \text{ is adjacent to } e\}$$

   of adjacent nodes after the (hypothetic) requested adaptation is determined. If $l_e^{\text{raw}} - l_e \geq L_\Delta^{\max}$ where l_e is the level of e and e is marked for coarsening, the coarsening mark for e is unset. If even $l_e^{\text{raw}} - l_e > L_\Delta^{\max}$ the refinement mark for e is set. This loop is organized such that elements with higher levels are processed first. If refinement marks have been set automatically, then adjacent coarsening marks are reevaluated to see if $l_e^{\text{raw}} - l_e \geq L_\Delta^{\max}$ still holds.

3. Remove coarsening marks for all subgrid elements e having siblings that are not marked for coarsening. If subgrid elements are marked for refinement and the corresponding host grid elements are leaf elements mark these host grid elements for refinement.

4. Call `preAdapt` for the host grid if necessary.

Step 2 ensures that the levels of leaf elements sharing a node will at most differ by $L_\Delta^{\max}$. For the default value $L_\Delta^{\max} = 1$ this guarantees that only first order hanging nodes appear.

The `adapt` method refines the subgrid, and also the host grid if necessary. This refinement of the host grid changes the host grid's level indices, which in turn invalidates the subgrid data structures. Hence before calling `adapt` on the host grid all subgrid elements not scheduled for coarsening are stored in a `std::map` together with their refinement marks. Then the `adapt` method of the host grid is called. Finally the subgrid is recreated containing all host grid elements whose ids were stored in the `std::map` and the children of those marked for refinement. The vector for refinement marks is then used to store the information if an element was refined.

The `postAdapt` method resets the refinement marks and calls the `postAdapt` method of the host grid if necessary.

3.5 Implementation Efficiency

Implementing a subgrid using arrays of boolean values needs an amount of storage that is linear in the number of host grid elements. This is not optimal if only a small fraction of the host grid elements is included in the subgrid. Also, iteration over all subgrid elements is linear in the number of host grid elements. On the other hand, the constant is very low. An alternative implementation would use arrays of `EntityPointer`s to the host grid entities in the subgrid. Here, both space and iteration complexity would be linear in only the number of subgrid elements. However, this way the number of bits used per subgrid entity is much higher. Both implementations are reasonable and the best choice depends on the application.

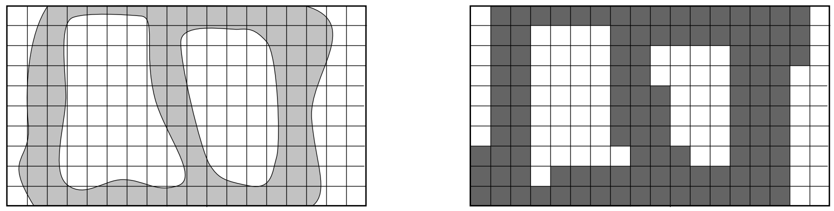

Figure 2: The domain Ω and its discretization Ω_h.

4 Micro-FE Models of Trabecular Bone

It is an important problem in biomechanics to determine macroscopical material properties of bone from its microscopical properties and structure. In the interior of long bones there is a sponge-like structure of trabeculae on a scale of about 0.1 mm. While numerous averaged macroscopical material laws exist for trabecular bone tissue, the availability of large computer power and of micro-computer-tomograph (μCT) scanners has made it possible to perform finite element computations directly on the trabecular structure. Due to the enormous geometrical complexity, this is a very memory-intensive problem. In particular, it has proven infeasible to approximate larger specimen of trabecular bone by unstructured tetrahedral grids. Instead, μFE models use the voxel structure of the segmented image CT data as the grid [21]. The uniform structure of these grids can be captured by a memory-efficient special-purpose grid implementation. Still, at the time of writing larger problems of this type fill supercomputers [6].

The `dune-subgrid` module provides a way to implement μFE computations memory-efficiently without having to implement a special purpose grid manager. Indeed, the voxel structure used for these computations is a subset of elements of a uniform grid. As such, it can be implemented using a `SubGrid` parametrized with a DUNE implementation of a uniform grid. Then, since `SubGrid` implements the DUNE grid interface, existing code for the assembly and solution of mechanics problems can be applied.

Let Ω be a domain in $\mathbb{R}^d$. The boundary $\partial\Omega$ is supposed to consist of two disjoint subsets Γ_D and Γ_N such that $\overline{\Gamma}_D \cup \overline{\Gamma}_N = \partial\Omega$. We do not assume that Ω is connected but that the intersections of Γ_D with each connected component of Ω have positive $(d-1)$-dimensional measure. Let $\mathbf{H}^1_D(\Omega)$ be the set of all d-valued L^2-functions with weak first derivatives in L^2 and that are zero in the sense of traces on Γ_D. We consider the weak boundary value problem of linear elasticity

$$\mathbf{u} \in \mathbf{H}^1_D(\Omega) \quad : \quad a(\mathbf{u}, \mathbf{v}) = l(\mathbf{v}), \qquad \text{for all } \mathbf{v} \in \mathbf{H}^1_D(\Omega), \tag{1}$$

with

$$a(\mathbf{v}, \mathbf{w}) = \int_\Omega \boldsymbol{\varepsilon}(\mathbf{v}) : \mathbf{C} : \boldsymbol{\varepsilon}(\mathbf{w}) \, dx \qquad \text{and} \qquad l(\mathbf{v}) = \int_\Omega \mathbf{f}\mathbf{v} \, dx,$$

where $\boldsymbol{\varepsilon}$ is the linearized strain tensor, $\mathbf{C}$ the forth-order Hooke tensor, and $\mathbf{f} : \Omega \to \mathbb{R}^d$ a prescribed volume force.

Let

$$\mathcal{B} = \prod_{i=1}^{d} [a_i, b_i]$$

be a bounded rectangular domain such that $\overline{\Omega} \subset \mathcal{B}$. Pick $n = (n_1, \ldots, n_d) \in \mathbb{N}^d$ and let α be a d-dimensional multiindex. We introduce a uniform grid on $\mathcal{B}$ consisting of $\prod_i n_i$ elements $\mathcal{B}^h_\alpha$ of equal size. The domain Ω will be discretized by the subset of all elements $\mathcal{B}^h_\alpha$ that intersect Ω (Fig. 2). This defines our discrete domain

$$\Omega_h = \bigcup_{\mathcal{B}^h_\alpha \cap \Omega \neq \emptyset} \mathcal{B}^h_\alpha,$$

and a corresponding grid G. Assuming for simplicity that Γ_D is resolved by $\partial\Omega_h$, we introduce a discretization of Problem (1). Let $\mathbf{V}_{h,D}(\Omega_h)$ be the space of d-valued first-order Lagrangian finite elements on G that are zero on Γ_D. The weak formulation of the discrete problem is

$$\mathbf{u}_h \in \mathbf{V}_{h,D}(\Omega_h) \quad : \quad a(\mathbf{u}_h, \mathbf{v}_h) = l(\mathbf{v}_h), \qquad \text{for all } \mathbf{v} \in \mathbf{V}_{h,D}(\Omega_h), \tag{2}$$

where, in an abuse of notation, we have used $a(\cdot, \cdot)$, $l(\cdot)$ to denote the natural extensions of the forms $a(\cdot, \cdot)$ and $l(\cdot)$ to $\mathbf{H}^1_D(\Omega_h)$. It is well known that Problem (2) has a unique solution [10]. The effect of the approximation of domains by sets of voxels has been studied by Babuška and Chleboun [1].

4.1 Implementation using `dune-subgrid`

Implementing Problem (2) is easy using the `dune-subgrid` module. Given the numbers $(n_1, \ldots, n_d)$ in an array `n` and the bounding box bounds $a_i, b_i, 1 \leq i \leq d$, in an array `bbox`, the C++ line

```
YaspGrid<d> hostgrid(bbox, n, [...]);
```

constructs a uniform grid on $\mathcal{B}$ with a single grid level. Note that this is a dedicated implementation of a structured grid and hence it is space-optimal. The subgrid Ω_h is implemented by constructing

```
SubGrid<d, YaspGrid<d> > subgrid(hostgrid);
```

and marking the elements $\mathcal{B}^h_\alpha$ using the `insert()`-method described in Sec. 2.1. With the bone geometry available as a segmented image data field this amounts to a simple loop over all host grid elements. Then, stiffness matrix and right hand side vector for the discrete problem (2) can be assembled using any linear elasticity assembler for DUNE, as for example the one in the `dune-disc` module. The resulting system can be solved, e.g., with an algebraic multigrid solver.

We demonstrate this with a numerical example. Our data set is a section of the left human tibia obtained by a micro-CT scan with a voxel size of 82 μm in

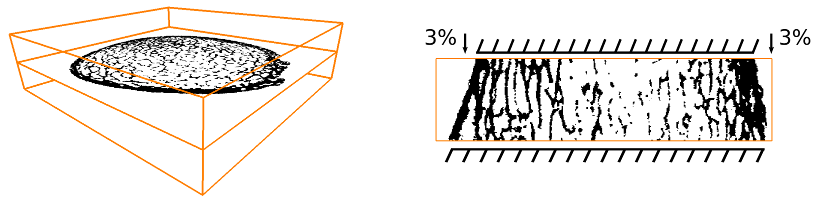

Figure 3: Horizontal cut through the radius data set, and a vertical cut with an illustration of the boundary conditions.

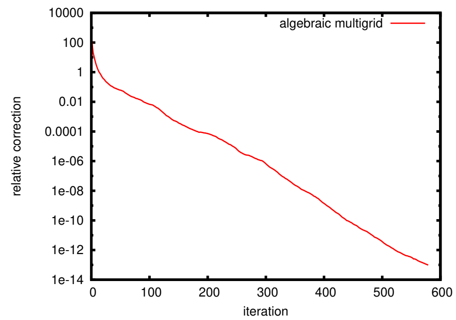

Figure 4: Algebraic multigrid: Relative correction as an estimate for the error.

each direction and a resolution of $449 \times 422 \times 110$ voxels. In this image, 4 563 234 voxels are marked as 'bone' (22 %) (Fig. 3).

We model the bone as a linear elastic material of St. Venant–Kirchhoff type with $E = 17\,\text{GPa}$ and $\nu = 0.3$. The specimen is clamped at the lower cut and subjected to a uniform displacement in the negative z-direction of 0.27 mm on the upper cut. This corresponds to a compression of 3% (Fig. 3, right). Starting from zero, we solve the problem using a conjugate-gradient algorithm preconditioned by a $V(4,4)$-cycle of the algebraic multigrid algorithm supplied with the `dune-istl` module [7]. This AMG is of agglomeration type and we have set the average size of the aggregates to 10 and the target coarse problem size to 100.

Fig. 4 shows the energy norm of the relative correction for each iteration. Note that the relative correction is an estimate of the error. The convergence rate is approximately 0.94, which makes the algorithm usable in practice. The implementation consumed about 14.5 GB of memory and about 100 seconds per iteration on an Intel Xeon processor with 2.33 GHz clock speed. Much time is saved in the setup phase by modifying the linear elasticity assembler to compute the element stiffness matrix only once, as it is identical for all elements.

Note that our goal was not to obtain the best convergence rates but to have a solver that is simple to implement and memory-efficient. Even more memory could be saved by providing a special purpose implementation for the fine-grid stiffness matrix that would make use of the voxel structure of the problem. Provided this special matrix implemented the `dune-istl` matrix interface it could be used as a direct replacement for the block compressed-row-storage implementation used here.

4.2 Geometric Multigrid

We have seen in the previous section how the discrete problem (2) can be solved with an out-of-the-box AMG solver. However, it is also possible to solve the problem using a geometric multigrid method. While this may not be quite as evident, the implementation is nevertheless very simple. In fact, it appears naturally when a standard geometric multigrid implementation is used together with a hierarchical subgrid. We will see that for our example, the geometric multigrid does not converge faster than the algebraic one. However, grid-independent convergence rates can actually be proven for geometric multigrid [19]. Also, there are less parameters that need tuning.

To obtain suitable coarse grid spaces for a geometric multigrid method we first construct a sequence of coarser grids. Assume that $n_1 = \ldots = n_d = 2^J$ for some $J \in \mathbb{N}$, let $h^j = 2^{J-j}h$ for $j \in \{0, \ldots, J\}$ and set

$$\mathcal{B}_\alpha^{h^j} = \prod_{i=1}^{d} [a_i + \alpha_i h_i^j, a_i + (\alpha_i + 1)h_i^j].$$

We define the coarse grid domains by setting

$$\Omega_{h^j} = \bigcup_{\mathcal{B}_\alpha^{h^j} \cap \Omega \neq \emptyset} \mathcal{B}_\alpha^{h^j}.$$

The individual $\mathcal{B}_\alpha^{h^j}$ induce a natural grid on Ω_{h^j}. Note that $\Omega_{h^j} \subset \Omega_{h^k}$ if $j > k$, but $\Omega_{h^j} \neq \Omega_{h^k}$ in general. Due to this fact the canonical finite element spaces $\mathbf{V}_{h^j} = \mathbf{V}_{h^j,D}(\Omega_{h^j})$ are not nested.

To define a nested hierarchy of spaces let $\{\boldsymbol{\lambda}^j\}$ be the nodal basis of $\mathbf{V}_{h_j}$. It consists of all functions $\boldsymbol{\lambda}_{p,i} = \lambda_p \mathbf{e}_i$, with λ_p the scalar hat function of vertex p and $\mathbf{e}_i$ the i-th canonical basis vector of $\mathbb{R}^d$. We now introduce the truncated nodal basis $\{\tilde{\boldsymbol{\lambda}}^j\}$ by setting

$$\tilde{\boldsymbol{\lambda}}_{p,i}^j = \boldsymbol{\lambda}_{p,i}^j|_{\Omega_{h^J}} \in \mathbf{V}_{h^J},$$

for all $0 \leq j \leq J$, and the truncated coarse grid spaces $\widetilde{\mathbf{V}}_{h^j} = \operatorname{span}\{\tilde{\boldsymbol{\lambda}}^j\}$. Note that $\widetilde{\mathbf{V}}_{h^J} = \mathbf{V}_{h^J}$ and $\widetilde{\mathbf{V}}_{h^j} = \mathbf{V}_{h^j}|_{\Omega_h^J}$. Hence these spaces form a conforming hierarchy in the sense that

$$\widetilde{\mathbf{V}}_{h^j} \subset \widetilde{\mathbf{V}}_{h^k}, \qquad \forall j < k.$$

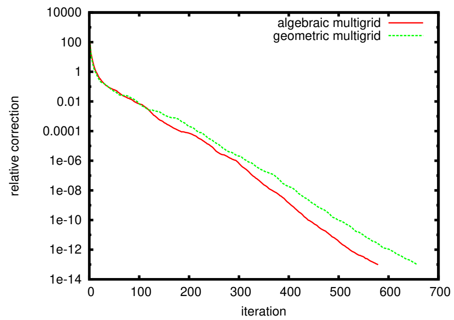

Figure 5: Relative corrections per iteration for the geometric multigrid. The red line gives the relative corrections of the AMG for comparison.

To obtain the algebraic formulation of the problem we introduce the fine grid stiffness matrix

$$\left(A^J_{pq}\right)_{ik} = a(\boldsymbol{\lambda}^J_{p,i}, \boldsymbol{\lambda}^J_{q,k}) = \int_{\Omega_{h^J}} \boldsymbol{\varepsilon}(\boldsymbol{\lambda}^J_{p,i}) : \mathbf{C} : \boldsymbol{\varepsilon}(\boldsymbol{\lambda}^J_{q,k})\, dx$$

and corresponding right hand side

$$(b_p)_i = \int_{\Omega_{h^J}} \mathbf{f} \boldsymbol{\lambda}^J_{p,i}\, dx.$$

Let $\mathcal{N}_{h^j}$ be the set of vertices of the grid covering Ω_{h^j} and $m^j = |\mathcal{N}_{h^j}|$. Then A^J is a $dm^J \times dm^J$ matrix. The algebraic coarse grid problems are constructed by defining the prolongation operators $P^{j\to j+1} \in \mathbb{R}^{dm^{j+1}\times dm^j}$

$$P^{j\to j+1}_{pq} = \mathrm{Id}_d\, \lambda^j_q(x_p) \qquad \forall p \in \mathcal{N}_{h^{j+1}}, q \in \mathcal{N}_{h^j} \tag{3}$$

and setting

$$A^j = (P^{j\to j+1})^T A^{j+1} P^{j\to j+1}.$$

The reader may note the close relationship to truncated multigrid methods [18]. Multigrid convergence can be established using the framework laid out in [19].

To implement this algorithm using the `dune-subgrid` module we note that a standard geometric multigrid algorithm degenerates gracefully to the algorithm described above provided the subgrid is setup correctly. Indeed, a standard assembler for a prolongation operator may loop over the vertices of all children of an element and collect the entries $\lambda^j_q(x_p)$ used in (3). If an element is not entirely covered by its children, the truncated prolongation operator $P^{j\to j+1}$ appears naturally.

We demonstrate the applicability of the geometric multigrid method with the same tibia data set used above. To enable the construction of a suitable grid hierarchy we enlarged the data set to $448 \times 448 \times 128$ voxels by cropping and padding with zeros. We construct a `YaspGrid` consisting of $7 \times 7 \times 2$ elements and refine it six times. This yields a grid hierarchy of seven levels and a fine grid of $448 \times 448 \times 128$ elements. We instantiate a `SubGrid` of this `YaspGrid` and mark all elements which intersect Ω using the method `insertPartial()` (see Sec. 2.1). This leads precisely to a hierarchy of grids and corresponding domains Ω_{h^j} as described in the previous section. Note, however, that the resulting hierarchical grid is strictly speaking not a grid in the DUNE sense, because child elements may not form logical partitions of their fathers. However, the multigrid algorithm only uses the level grids, which are intact.

We solve the Problem (2) using a conjugate gradient algorithm preconditioned by a $V(4, 4)$ cycle of the geometric multigrid. A plot of the estimated error history can be seen in Fig. 5. The convergence rate is slightly worse than the rate of the algebraic multigrid, but we point out again that, unlike for the AMG, for a geometric multigrid method grid independent convergence rates can be established theoretically [19]. The computation consumed roughly 16 GB of memory and 105 seconds per iteration. The increase in memory requirements is due to the coarse subgrid levels which are not needed by the AMG.

5 Implicit Time Integration with a Correct Residual

The numerical solution of evolution problems in function spaces is often tackled by Rothe's method: First discretize the continuous evolution problem in time, then discretize each arising stationary problem in space. The advantage of this is that the grid can be chosen adaptively at each time step [8].

We now consider a (possibly degenerate) parabolic or hyperbolic partial differential equation on a domain Ω, which is discretized in time with an implicit Euler method. At each time step k, a weak spatial problem

$$a(u^k, v) = l(u^{k-1}, v) \qquad \forall v \in H^1(\Omega) \tag{4}$$

has to be solved for the new solution u^k. In (4), the forms $a(\cdot, \cdot)$ and $l(\cdot, \cdot)$ depend linearly on their second arguments, but possibly nonlinearly on their first ones. Note in particular that l depends on the solution u^{k-1} at the previous time step.

The evaluation of $l(\cdot, \cdot)$ typically involves integrals over Ω of expressions depending both on u^{k-1} and v. The function u^{k-1} is a finite element function with respect to G_{k-1}, the grid at time step $k-1$. The test function v, however, is a finite element function with respect to the grid G_k at time step k. In an adaptive method $G_k \neq G_{k-1}$, and hence, conceptually, we have to work with two different grids at the same time.

There are several ways to deal with this problem in an implementation:

1. Use two separate grid objects. This appears straightforward, however, it is actually quite difficult to relate functions on two unrelated grids of the

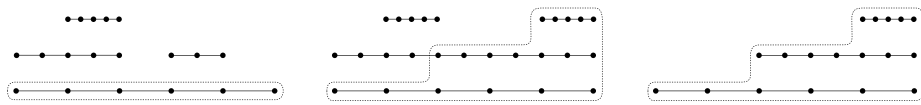

Figure 6: The hierarchical host grid (solid lines) and the subgrid (dashed) during one time step using Rothe's method. Computation is done on the subgrid while the host grid stays fine enough to include both the current grid and the grid of the last time step. Left: the host grid is the grid G_{k-1} of the last time step, computation starts using a subgrid G_k^0 containing the coarsest level of G_{k-1}. Center: the subgrid has been refined adaptively j_k times to a $G_k^{j_k}$, and the host grid along with it where necessary. Right: after the time step is completed the host grid is shrunk as to comprise only the new grid $G_k = G_k^{j_k}$.

same domain Ω. Also, having two grid objects at the same time may use a lot of memory.

2. Use a grid object only for G_k and store nodal function values and coordinates of u^{k-1} only. This avoids the need for two full grid objects. However, a certain amount of error is introduced because it is not possible to reconstruct u^{k-1} from its nodal values alone.

3. Start the adaptivity loop for G_k from the old grid $G_k^0 = G_{k-1}$, instead of from a very coarse grid. After the i-th cycle of the adaptivity loop project u^{k-1} from G_k^i onto the new grid G_k^{i+1}. This approach is also not satisfactory for the following reasons. First, the projections of u^{k-1} again introduce some error. Also, since we start the adaptive cycle at a partially refined grid $G_k^0 = G_{k-1}$, we need a coarsening indicator in addition to the refinement indicator. Finally, since we do not expect the number of degrees of freedom to vary much from one time step to the next, the adaptive loop will solve a sequence of large problems. This is more expensive than starting the adaptive loop on a very coarse grid at each new time step and refining successively.

This last approach appears to be the most frequently used in the literature.

5.1 Rothe's Method using `dune-subgrid`

We now present a new algorithm that solves (4) on adaptively refined grids with an exact l. The algorithm uses `dune-subgrid` and does not have the drawbacks of the methods described above.

Suppose that u^{k-1} is a function represented on a grid G_{k-1}. Then the solution u^k of the next time step on the grid $G_k \neq G_{k-1}$ can be found in the following way:

1. Start with a coarse subgrid G_k^0 contained in the host grid G_{k-1} (Fig. 6, left). Set $i = 0$.

2. Solve (4) on the subgrid G_k^i. When u^{k-1} needs to be evaluated do this on the host grid and transfer it to G_k^i. This is easy since data transfer methods from the host grid to the subgrid are provided (Sec. 2.3).

3. Estimate the discretization error. If it is below the tolerance go to Step 5.

4. Adaptively refine the subgrid G_k^i to obtain G_k^{i+1}. Adapt u^{k-1} if the host grid is refined during subgrid adaptation (Fig. 6, center). Since this is a projection onto a finer grid no information is lost. Increment i and go back to Step 2.

5. Shrink the host grid to the current subgrid and transfer the solution u^k to the host grid (Fig. 6, right).

Throughout the refinement loop, the host grid is fine enough to include the grids from the last and the current time step. The shrinking of the host grid in Step 5 keeps the host grid from growing further and further during the evolution. Since the host grid is alway finer than G_{k-1} the exact representation of u^{k-1} on the host grid is possible. The right hand side $l(\cdot,\cdot)$ of (4) can therefore be evaluated exactly.

Since a `SubGrid` uses less memory than a comparable regular adaptive grid implementation, our approach is less memory-intensive than using two grid objects together. Run-time overhead comes from having to iterate over a finer grid than G_k to assemble $l(\cdot,\cdot)$ and from the `SubGrid` interface itself. Measurements quantifying these effects are given at the end of this chapter.

5.2 Adaptive Solution of the Heat Equation

To demonstrate the advantages of the presented algorithm we now consider the heat equation as the simplest parabolic equation.[2] On the domain $\Omega = [-1,1]^2$ consider

$$\begin{aligned} \frac{\partial}{\partial t}u - \Delta u &= 0 && \text{in } [0,T]\times\Omega, \\ u(0,\cdot) &= u_0 && \text{in } \Omega, \\ \frac{\partial}{\partial \nu}u &= 0 && \text{in } [0,T]\times\partial\Omega, \end{aligned} \tag{5}$$

with a piecewise constant initial value u_0 as depicted in Fig. 7.

After deriving a weak formulation and discretizing in time with the implicit Euler method we obtain a sequence of stationary problems

$$u^k \in H^1(\Omega) \quad : \quad (u^k, v) + \tau(\nabla u^k, \nabla v) = (u^{k-1}, v) \qquad \forall v \in H^1(\Omega), \tag{6}$$

with $u^0 = u_0$ and τ the time step size. We use $(\cdot,\cdot)$ to denote the L^2 scalar product. These problems are discretized with piecewise linear finite elements,

[2] The presented technique has also been successfully applied to degenerate nonsmooth nonlinear problems like the Allen–Cahn and the Cahn–Hilliard equations [16, 17].

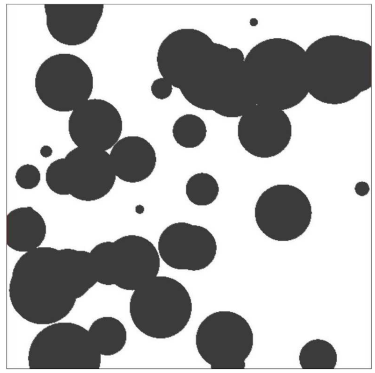

Figure 7: Initial value u_0 for the test evolution (5).

and the first L^2 scalar product is replaced by a lumped scalar product. We use an `ALUSimplexGrid` [9] as the host grid. This grid manager implements simplex grids with nonconforming red refinement. The coarse grid is a uniform simplicial grid on 17×17 vertices. The algebraic problems on each grid are solved using a multigrid algorithm with Gauß–Seidel smoothing.

Discretization errors are measured with a hierarchic error estimator [2]. For the larger space we choose the space of first-order finite element functions on a grid that has been refined once uniformly. The local contributions η_e of the estimator are used as refinement indicators for a marking strategy selecting all contributions larger than σ times the average of the η_e. For the presented computations we set $\sigma = 0.7$.

By testing the continuous variational equation (6) with a constant function it is easy to see that the global energy

$$E(k) = \int_\Omega u^k \, dx$$

is conserved if the right hand side $l(u^{k-1}, v) = (u^{k-1}, v)$ is integrated exactly. This is only done by the expensive Algorithm 1 (p. 16) and the subgrid-based algorithm. On the other hand, it is interesting to see how the computational effort of the subgrid-based algorithm compares to the commonly used method based on grid coarsening (Algorithm 3, p. 17). Since this is not the place to discuss efficient coarsening strategies we only use a rough estimation of the computational effort of Algorithm 3.

Assume that $G_k \neq G_{k-1}$, as will frequently be the case. Then the coarsening-based algorithm will solve at least two large spatial problems at time step k. Indeed, starting on $G_k^0 = G_{k-1}$, Algorithm 3 will at least solve Problem (6) there, estimate the error, and adapt the grid to obtain a grid G_k^1 of similar size. Then it will solve (6) on G_k^1, and estimate the error again. In the best case the solution is accepted now. If not, further refinement/coarsening cycles have to follow. Assuming that the problems at consecutive time steps have a comparable size the computational effort will at least be twice the effort of solving the problem on G_{k-1}. The actual effort may be even higher, because frequently more than one grid adaptation step will be necessary, and for nonlinear problems nested iteration is often used to compute reasonable initial iterates for a nonlinear iterative solver. Hence additional problems on coarser grid are solved.

The subgrid-based algorithm solves only a single problem of about the size of G_{k-1} and a sequence of coarser problems on G_k^i, $0 \leq i < j_k$. Hence the subgrid-based algorithm is more efficient than the coarsening-based one if the time needed for the entire refinement loop is less than the time needed to solve two problems of about the size of G_{k-1}. We measured this for a test evolution of the heat equation with the initial value u_0 as in Fig. 7 and a time step of $\tau = 10^{-3}$. The first time step was discretized with a grid hierarchy consisting of 6 levels and 217 841 vertices. The left picture in Fig. 8 plots the CPU time needed to solve Problem (6) by the subgrid-based algorithm for each time step. While the dashed line represents the time for the entire refinement loop the solid line represents only the time spend for assembling and solving the problem on the final grid G_k in each time step k. The computational amount rapidly decreases with time due to the smoothing property of the heat equation. Also, the computations on the final grid consume a significant amount of the overall time in each time step. To examine this further the right picture shows the CPU time for the entire refinement loop divided by the time needed to assemble and solve Problem (6) on the final grid G_k. We stated above that the subgrid-based algorithm is more efficient than the coarsening-based one if this value stays below two. Except for a single outlier this is indeed the case. Thus the presented approach outperforms Algorithm 3 even in the most optimistic case described above. The flat line beginning in time step 25 marks the regime where the solution is smooth enough to be represented on the coarsest mesh. Since there $G_k = G_{k-1}$ the part has to be omitted from the comparison. For nonlinear problems the fraction of the time spent on the final grid will in general be even larger increasing the performance benefit of the presented approach.

To investigate the run-time overhead of the `SubGrid` interface we also compared timings of an `ALUSimplexGrid` with timings of the `SubGrid` set to be identical to the host grid. We computed the stiffness and mass matrix as well as the multigrid transfer operators for a grid hierarchy of six levels with 217 841 degrees of freedom on the final level, resulting as the final adaptive grid G_1 of the first time step. The measurements of these computations showed that the overhead of the subgrid interface is about 37% for these computations.

To see the cost of the exact integration of (u^{k-1}, v) we also measured the CPU time for the computation of the residual on the hostgrid and its restriction to the subgrid. We found that this took about 7.7% of the time for the entire refinement loop in the second time step. The assembly time is dominated by having to iterate over the host grid leaf level. Hence it remains large regardless of how coarse the subgrid is.

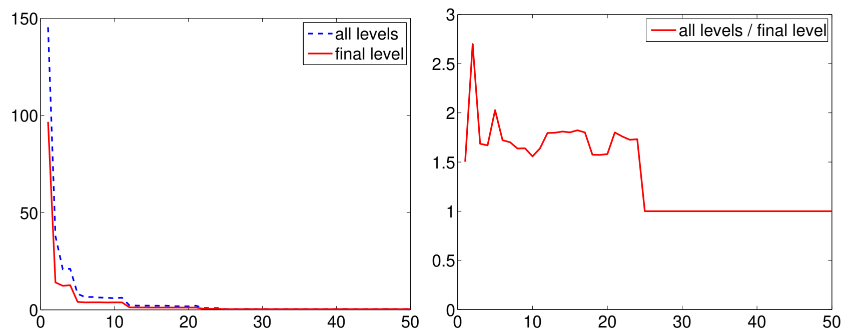

Figure 8: CPU time for assembling, solution and error estimation over time step. a) All levels and last level; b) All levels relative to last level.

6 Conclusions and Outlook

We have presented a DUNE meta grid that allows to do computations on subsets of elements of other DUNE grids. This gives new flexibility to users of the DUNE grid interface.

Two examples applications were described to show the benefits of this approach. They demonstrated that the use of `dune-subgrid` makes treating certain problems easy, which otherwise would have needed involved special-purpose coding. This exemplifies the DUNE spirit of having code consist of modular, reusable components. However, the list of possible applications does not end here. Together with a structured grid implementation, `SubGrid` can facilitate the implementation of narrow-band methods as presented in [11]. Being able to have two refinement states of a single coarse grid (such as in Sec. 5) has also been used in implementations of unfitted discontinuous galerkin methods such as [14]. We suspect that this short list of possible applications is not complete.

Several extensions to our implementation are conceivable. While support for parallel grids has not been implemented yet, there are no structural obstacles that prevent doing so. Very interesting, but unfortunately more difficult, is the possibility to select subgrids of lower dimension than the grid itself. This would allow to handle mortar methods, problems with an additional PDE on the domain boundary, and the simulation of material cracks or blood vessels, which are sometimes modelled as ensembles of lower-dimensional entities of a given grid. However, the DUNE interface distinguishes elements from entities of lower dimension in that they have more methods. Therefore, `SubGrid` would have to provide all methods of an element without being able to use the full element interface of the host grid.

Some further variants of subgrids are also interesting. As discussed in Sec. 3.5, it would be possible to implement a subgrid as an array of pointers to grid entities instead of a bit field. This would make subgrids that only comprise a small fraction of the host grid more efficient. Also, in some applications one is interested not only in working on a subset of elements but in partitioning the entire host grid into a set of disjoint subdomains. While this can be done using a set of subgrids, a dedicated grid implementation would be more efficient.

While these extensions could in principle be realized by generalizing and modularizing the existing code of `SubGrid`, we propose to realize them as separate meta grid implementations in order to keep the code complexity at a reasonable level. The list of extension modules for DUNE is growing [15] and we hope to see such grids there some day.

Acknowledgments

The authors would like to thank Christian Engwer for some help with the implementation and Zully Ritter for providing the tibia data set.

References

[1] I. Babuška and J. Chleboun. Effects of uncertainties in the domain on the solution of Dirichlet boundary value problems. *Numerische Mathematik*, 93(4):583–610, 2003.

[2] R. E. Bank and R. K. Smith. A posteriori error estimates based on hierarchical bases. *SIAM J. Num. Anal.*, 30(4):921–935, 1993.

[3] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuß, H. Rentz–Reichert, and C. Wieners. UG – a flexible software toolbox for solving partial differential equations. *Comp. Vis. Sci.*, 1:27–40, 1997.

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

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

[6] C. Bekas, A. Curioni, P. Arbenz, C. Flaig, G. H. van Lenthe, R. Müller, and A. J. Wirth. Extreme scalability challenges in micro-finite element simulations of human bone. In *Proc. International Supercomputing Conference ISC'08*, Dresden, Germany, 2008.

[7] M. Blatt and P. Bastian. The iterative solver template library. In *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Scientific Computing*, pages 666–675. Springer Verlag, 2007.

[8] F. Bornemann. *An Adaptive Multilevel Approach for Parabolic Equations in Two Space Dimensions*. PhD thesis, Freie Universität Berlin, 1991.

[9] A. Burri, A. Dedner, R. Klöfkorn, and M. Ohlberger. An efficient implementation of an adaptive and parallel grid in DUNE. In *Proc. of the 2nd Russian–German Advanced Research Workshop on Computational Science and High Performance Computing*, 2005.

[10] P. G. Ciarlet. *Mathematical Elasticity*. North-Holland, 1988.

[11] K. Deckelnick, G. Dziuk, C. M. Elliot, and C.-J. Heine. An h-narrow band finite element method for elliptic equations on implicit surfaces. *IMA Journal of Numerical Analysis*, to appear.

[12] DUNE. Distributed and Unified Numerics Environment. URL `http://dune-project.org/`.

[13] DUNE-FEM. URL `http://dune.mathematik.uni-freiburg.de/`.

[14] C. Engwer and P. Bastian. An unfitted finite element method using discontinuous galerkin. *Int. J. Numer. Meth. Engng*, 2008. accepted.

[15] External DUNE Modules. `www.dune-project.org/downloadext.html`.

[16] C. Gräser and R. Kornhuber. Adaptive multigrid methods for Cahn–Hilliard equations with logarithmic potential. 2009. in preparation.

[17] C. Gräser, R. Kornhuber, and U. Sack. Adaptive multigrid methods for anisotropic Allen–Cahn equations with logarithmic potential. 2009. in preparation.

[18] R. Kornhuber. *Adaptive Monotone Multigrid Methods for Nonlinear Variational Problems*. Teubner, Stuttgart, 1. edition, 1997.

[19] R. Kornhuber and H. Yserentant. Multilevel methods for elliptic problems on domains not resolved by the coarse grid. *Contemp. Math.*, 180:49–60, 1994.

[20] A. Schmidt and K. G. Siebert. *Design of Adaptive Finite Element Software: The Finite Element Toolbox ALBERTA*. LNCSE. Springer Verlag, 2005.

[21] B. van Rietbergen, H. Weinans, R. Huiskes, and A. Odgaard. A new method to determine trabecular bone elastic properties and loading using micromechanical finite-element models. *J. Biomech.*, 28:69–81, 1995.

[22] T. Veldhuizen. Techniques for scientific C++. Technical Report 542, Indiana University Computer Science, 2000.