Towards a Unified Framework for Scientific Computing

Peter Bastian[1], Mark Droske[3], Christian Engwer[1], Robert Klöfkorn[2], Thimo Neubauer[1], Mario Ohlberger[2], Martin Rumpf[3]

[1] Interdisziplinäres Zentrum für Wissenschaftliches Rechnen, Universität Heidelberg, Im Neuenheimer Feld 368, D-69120 Heidelberg,
[2] Abteilung für Angewandte Mathematik, Universität Freiburg, Hermann-Herder-Str. 10, D-79104 Freiburg,
[3] Fachbereich Mathematik, Universität Duisburg Lotharstr. 63/65, D-47048 Duisburg; DUNE website: `http://hal.iwr.uni-heidelberg.de/dune/`

Summary. Most finite element, or finite volume software is built around a fixed mesh data structure. Therefore, each software package can only be used efficiently for a relatively narrow class of applications. For example, implementations supporting unstructured meshes allow the approximation of complex geometries but are in general much slower and require more memory than implementations using structured meshes. In this paper we show how a generic mesh interface can be defined such that one algorithm, e. g. a discretization scheme, works on different mesh implementations. For a cell centered finite volume scheme we show that the same algorithm runs thirty times faster on a structured mesh implementation than on an unstructured mesh and is only four times slower than a non-generic version for a structured mesh. The generic mesh interface is realized within the *Distributed Unified Numerics Environment* DUNE.

1 Introduction

There exist many simulation packages for the numerical solution of partial differential equations ranging from small codes for particular applications or teaching purposes up to large ones developed over many years which can solve a variety of problems. Each of these packages has a set of features which the designers decided to need to solve their problems. In particular, the codes differ in the kind of meshes they support: (block) structured meshes, unstructured meshes, simplicial meshes, multi-element type meshes, hierarchical meshes, bisection and red-green type refinement, conforming or non-conforming meshes, sequential or parallel mesh data structures are possible.

Using one particular code it may be impossible to have a particular feature (e. g. local mesh refinement in a structured mesh code) or a feature may be very inefficient to use (e. g. structured mesh in unstructured mesh code). If efficiency matters, there will never be one optimal code because the goals

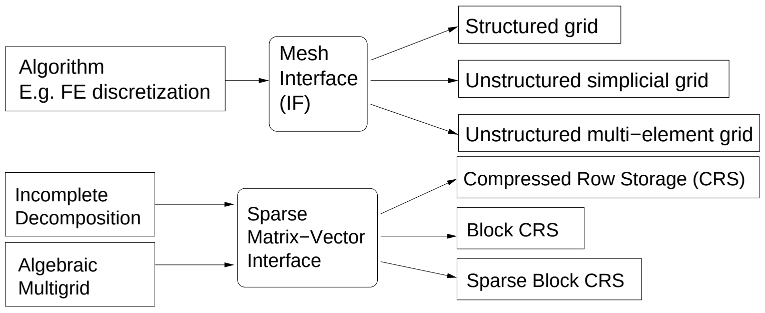


Fig. 1. Encapsulation of data structures with abstract interfaces.

are conflicting. Extension of the set of features of a code is often very hard. The reason for this is that most codes are built upon a particular mesh data structure. This fact is well known in computer science (Brooks [1975]).

A solution to this problem is to separate data structures and algorithms by an abstract interface, i. e.

- one writes algorithms based on an abstract interface and
- uses exactly the data structure that fits best to the problem.

Figure 1 shows the application of this concept to two different places in a finite element code: A discretization scheme accesses the mesh data structure through an abstract interface. The interface can be implemented in different ways, each offering a different set of features efficiently. In the second example an algebraic multigrid method accesses a sparse matrix data structure through an abstract interface.

Of course, this principle also has its implications: The set of supported features is built into the abstract interface. Again, it is in general very difficult to change the interface. However, not all implementations need to support the whole interface (efficiently). Therefore, the interface can be made very general. At run-time the user pays only for functionality needed in the particular application.

The paper is organized as follows: The next section describes the *Distributed Unified Numerics Environment* (DUNE) which is based on abstract interfaces and shows how these interfaces can be implemented very efficiently using generic programming in C++. Then, in Section 3, we describe in more detail the abstract interface for a general finite element or finite volume mesh and in Section 4 we evaluate the concept on the basis of a cell centered finite volume scheme for various implementations of the mesh interface.

2 The DUNE Library

Writing algorithms based on abstract interfaces is not a new concept. Classical implementations of this concept in procedural languages use function calls. As an example consider the basic linear algebra subroutines BLAST [2001]. In

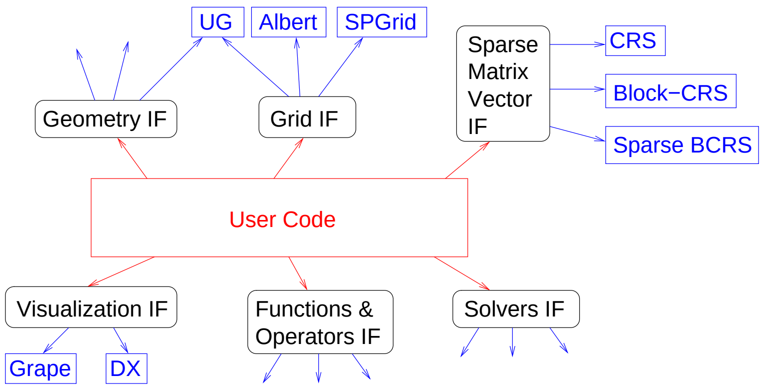


Fig. 2. DUNE module structure.

object oriented languages one uses abstract base classes and inheritance to implement polymorphism. E. g., C++ offers virtual functions to implement dynamic polymorphism. The function call itself poses a serious performance penalty in case a function/method in the interface consists only of a few instructions. Therefore, function calls and virtual method invocation can only be used efficiently for interfaces with sufficiently coarse granularity.

However, to utilize the concept of abstract interfaces to full extent one needs interfaces with fine granularity. E. g., in the case of a mesh interface one needs to access coordinates of nodes, normals of faces or evaluate element transformations at individual quadrature points. Generic programming, implemented in the C++ language through templates, offers a possibility to implement interfaces without performance penalty. The abstract algorithm is parameterized by an implementation of the interface (a concrete class) at compile-time. The compiler will then be able to inline small functions and to employ all code optimizations. Basically, the interface is removed completely at compile-time. This technique is also called static (or compile-time) polymorphism and is used extensively in the well-known standard template library STL, see Musser et al. [2001]. Many C++ programming techniques we use are described in Barton and Nackman [1994] and Veldhuizen [2000].

DUNE is a template library for all software components required for the numerical solution of partial differential equations. Figure 2 shows the high level design. User code written in C++ will access geometries, grids, sparse linear algebra, visualization and the finite element functionality through abstract interfaces. Many implementations of one interface are possible and particular implementations are selected at compile-time. It is very important that incorporation of existing codes is very natural within this concept. Moreover, the design can also be used to couple different existing codes in one application.

In the rest of this paper we concentrate on the design of the abstract interface for finite element and finite volume meshes.

3 Design of the Sequential Grid Interface

There are many different types of finite element or finite volume grids. We have selected the features of our grid interface according to the needs of our applications. In particular, we wanted to support grids that

- discretize unions of manifolds (e. g. fracture networks, shell elements),
- consist of elements of different geometric shapes (e. g tetrahedra, prisms, pyramids and hexahedra),
- support local, hierarchical mesh refinement.

In the following we define a grid $\mathcal{T}$ in mathematical terms. It is supposed to discretize a domain $\Omega \subset \mathbb{R}^n$, $n \in \mathbb{N}$, $n > 0$, with piecewise smooth boundary $\partial\Omega$. A grid $\mathcal{T}$ consists of $L + 1$ grid levels

$$\mathcal{T} = \{\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_L\}\,.$$

Each grid level $\mathcal{T}_l$ consists of sets of grid entities $\mathcal{E}_l^c$ of codimension $c \in \{0, 1, \ldots, d\}$ where $d \leq n$ is the dimensionality of the grid:

$$\mathcal{T}_l = \{\mathcal{E}_l^0, \ldots, \mathcal{E}_l^d\}\,.$$

Each entity set consists of individual grid entities which are denoted by $\Omega_{l,i}^c$:

$$\mathcal{E}_l^c = \left\{\Omega_{l,0}^c, \Omega_{l,1}^c, \ldots, \Omega_{l,N(l,c)-1}^c\right\}.$$

The number of entities of codimension c on level l is $N(l,c)$ and we define a corresponding index set

$$I_l^c = \{0, 1, \ldots, N(l,c) - 1\}.$$

Definition 1. *$\mathcal{T}$ is called a grid if the following conditions hold:*

1. *(Partitioning). The entities of codimension 0 on level 0 define a partitioning of the whole domain:*

$$\bigcup_{i \in I_0^0} \overline{\Omega_{0,i}^0} = \overline{\Omega}, \qquad \forall i \neq j : \Omega_{0,i}^0 \cap \Omega_{0,j}^0 = \emptyset.$$

2. *(Nestedness). Entities of codimension 0 on different levels form a tree structure. We require:*

$$\forall l > 0, i \in I_l^0 : \exists! j \in I_{l-1}^0 : \Omega_{l,i}^0 \subset \Omega_{l-1,j}^0.$$

This $\Omega_{l-1,j}^0$ is called father of $\Omega_{l,i}^0$. For entities with at least one side on the boundary this condition can be relaxed. We define the set of all descendant entities of codimension 0 and level $l \leq L$ of an entity $\Omega_{k,i}^0$ as

$$\mathcal{C}_L(\Omega_{k,i}^0) = \{\Omega_{l,j}^0 |\; \Omega_{l,j}^0 \subset \Omega_{k,i}^0, l \leq L\}.$$

3. *(Recursion over codimension). The boundary of a grid entity is composed of grid entities of the next higher codimension, i. e. for $c < d$ we have*

$$\partial \Omega_{l,i}^{c} = \bigcup_{j \in I_{l,i}^{c+1} \subset I_{l}^{c+1}} \overline{\Omega_{l,j}^{c+1}}.$$

Grid entities $\Omega_{l,j}^{d}$ of codimension d are points in $\mathbb{R}^n$.

4. *(Reference elements and dimension). For each grid entity $\Omega_{l,i}^{c}$ there is a reference element $\omega_{l,i}^{c} \subset \mathbb{R}^{d-c}$ and a sufficiently smooth map*

$$m_{l,i}^{c} : \overline{\omega_{l,i}^{c}} \to \overline{\Omega_{l,i}^{c}}$$

from the reference element to the actual element. Reference elements are convex polyhedrons in $\mathbb{R}^{d-c}$. The dimension of the grid d is the dimension of the reference elements corresponding to grid entities of codimension 0. For $c = d$ the map $m_{l,i}^{d}$ simply returns the corresponding point in $\mathbb{R}^n$.

5. *(Nonconformity). Note that we do not require the mesh to be conforming in the sense that the intersection of the closure of two grid entities of codimension c is either zero or a grid entity with codimension greater than c. However, we require that all grid entities in $\mathcal{E}_l^c$ are distinct, i. e. :*

$$\forall i, j, c, l : \quad \Omega_{l,i}^{c} = \Omega_{l,j}^{c} \Rightarrow i = j.$$

The set of all neighbors of an entity $\Omega_{l,i}^{0}$ is represented by the set of all non empty intersections with that entity:

$$\mathcal{I}(\Omega_{l,i}^{0}) = \{\overline{\Omega_{l,i}^{0}} \cap \overline{\Omega_{l,j}^{0}} | \ \overline{\Omega_{l,i}^{0}} \cap \overline{\Omega_{l,j}^{0}} \neq \emptyset, i \neq j\}.$$

Classes in the DUNE grid interface

According to the description in Definition 1, the grid interface consists of the following abstract classes:

1. `Grid⟨dim, dimworld, ...⟩`
   This class corresponds to the whole grid $\mathcal{T}$. It is parametrized by the grid dimension $d = dim$ and the space dimension $n = dimworld$. The grid class provides iterators for the access to its entities.
2. `Entity⟨codim, dim, dimworld, ...⟩`, `Element⟨dim, dimworld, ...⟩`
   Grid entities $\Omega_{l,i}^{c}$ of codimension $c = codim$ are realized by the classes Entity and Element. The Entity class contains all topological information, while geometrical specifications are provided by the Element class.
3. `LevelIterator⟨codim, dim, dimworld, ...⟩`
   The level iterator gives access to all grid entities on a specified level l. This allows a traversal of the set $\mathcal{E}_l^c$.

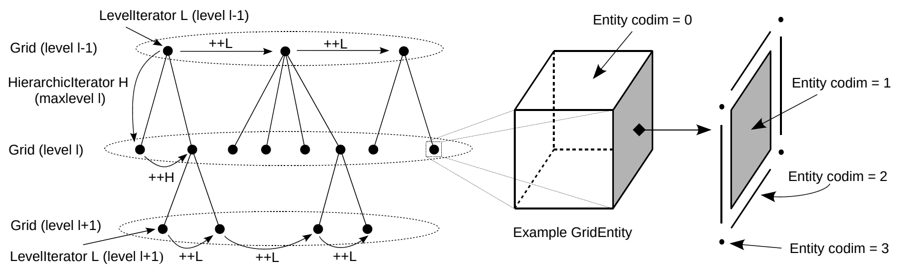


Fig. 3. DUNE Grid walk-trough of an 3 dimensional Grid.

4. `HierarchicIterator`$\langle$`dim, dimworld, ...`$\rangle$
   Another possibility to access grid entities is provided by the hierarchic iterator. This iterator runs over all descendant entities with level $l \leq L$ of a given entity $\Omega^0_{k,i}$. Therefore, it traverses the set $\mathcal{C}_L(\Omega^0_{k,i})$.
5. `IntersectionIterator`$\langle$`dim, dimworld, ...`$\rangle$
   Part of the topological information provided by the Entity class of codimension 0 is realized by the intersection iterator. For a given entity $\Omega^0_{l,i}$ the iterator traverses the set $\mathcal{I}(\Omega^0_{l,i})$.

A specific grid is realized by an implementation of derived classes of these abstract interface classes. The efficiency of the specific implementations is guaranteed by using static polymorphism (see Barton and Nackman [1994]). Figure 3 gives a sketch of the functionality of the grid interface. It shows the access of the grid entities via level, or hierarchic iterators and displays the recursive definition of entities via codimension.

4 Example and Performance Evaluation

In order to assess the performance of the proposed concept we compare different implementations for the numerical solution of the following linear hyperbolic equation:

$$\frac{\partial u}{\partial t} + \nabla \cdot (\mathbf{v}u) = 0 \text{ in } \Omega, \quad u = g \text{ on } \Gamma_{in} = \{\mathbf{x} \in \partial\Omega \mid \mathbf{v}(\mathbf{x}) \cdot \mathbf{n} \leq 0\}.$$

We discretize this equation with a cell-centered finite volume scheme using full upwind and an implicit Euler scheme in time. We note that this is not a particularly good scheme. However, it is very well suited to compare run-times since it is simple but contains all essential features of more complicated schemes as far as the mesh interface is concerned.

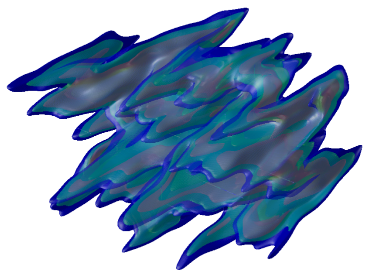
Table 1 shows the run-time for assembling the system matrix using various implementations. Implementation A is for a structured mesh. In implementations B and C the discretization is based on the DUNE mesh interface, thus it can be used for any dimension and element type. B uses simplegrid, an implementation of the mesh interface supporting structured meshes of variable dimension with entities of codimension d and 0. C uses an implementation of the

Table 1. Run-time for matrix assembly in a cell-centered finite volume scheme. We used a Pentium IV computer (2.4 GHz) with the Intel C++ compiler `icc` 7.0.

Key	Implementation	Grid	Run-time $[s]$
A	structured grid	128^3 hex	0.37
B	DUNE/simplegrid	128^3 hex	1.59
C	DUNE/albertgrid	$6 \cdot 64^3$ tet	4.20
D	Albert	$6 \cdot 64^3$ tet	3.13
E	UG	$6 \cdot 64^3$ tet	2.64
F	UG	128^3 hex	45.60

mesh interface based on the PDE toolbox ALBERT, see Schmidt and Siebert [2000], D is the same test using ALBERT directly without going through the interface. ALBERT supports simplicial elements in two and three space dimensions with bisection refinement. Finally, in E and F, we implemented the discretization scheme within the PDE framework UG (Bastian et al. [1997]) using hexahedral and tetrahedral meshes. Increase of run-time per element from E to F is due to the more costly element transformation for hexahedra. We are currently implementing the DUNE mesh interface based on UG. Run-times of cases E and F can be considered as preliminary results for a DUNE/UG mesh module.

From Table 1 we conclude that performance can be increased by a factor of 30 when we replace the unstructured hexahedral mesh (F) by a structured mesh (B). Memory requirements are reduced by a factor 10. These improvements are achieved *without* changes in the application code (here the discretization). Additional savings by a factor 4 in run-time (A) are only possible at the cost of reduced functionality of the user code. Memory requirements of the DUNE/simplegrid and structured mesh variants are the same.



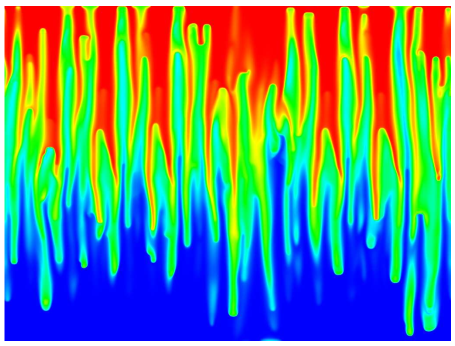


Fig. 4. Large scale parallel three-dimensional simulations. Contaminant transport in a heterogeneous medium (higher order Godunov scheme, $5 \cdot 10^8$ cells, left), density driven flow in a porous medium ($8 \cdot 10^8$ cells in three dimensions, right).

5 Conclusion and Future Prospects

In this paper we presented a new framework for the numerical solution of partial differential equations. The concept consequently separates data structures and algorithms. Algorithms are written in terms of abstract interfaces, the interfaces are implemented efficiently using static polymorphism in C++. We evaluated the performance of this approach for a simple discretization scheme. The run-time can be improved by up to a factor 30 by replacing an unstructured mesh implementation with a structured mesh implementation. The improvement is possible without any changes in the application code.

Currently we are extending the mesh interface by a general parallel data distribution model which will allow the formulation of overlapping and non overlapping domain decomposition methods as well as parallel multigrid methods on the same interface. First results of the parallel implementation are shown in Figure 4. Large scale computations with up to 10^{10} grid cells are possible on a 500 processor Linux cluster with a structured mesh implementation.

For data visualization, DUNE will be linked to the graphics packages GRAPE, Geßner et al. [1999], and AMIRA, Amira [2002].

References

Amira. *Amira 3.0 Visualization Software.* http://www.amiravis.com/, 2002.

J. Barton and L. Nackman. *Scientific and Engineering C++.* Addison-Wesley, 1994.

P. Bastian et al. UG - A flexible software toolbox for solving partial differential equations. *Computing and Visualization in Science*, 1:27–40, 1997.

BLAST. *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard.* http://www.netlib.org/blas/blast-forum/, 2001.

F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1975.

T. Geßner et al. A procedural interface for multiresolutional visualization of general numerical data. Report 28, SFB 256, Bonn, 1999.

D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 2001.

A. Schmidt and K. Siebert. ALBERT – An adaptive hierarchical finite element toolbox. Preprint 06/2000 Freiburg, 2000.

T. Veldhuizen. Techniques for scientific C++. Technical Report 542, Indiana University Computer Science, 2000. http://osl.iu.edu/˜tveldhui/papers/techniques/.