The Distributed and Unified Numerics Environment (DUNE)

Peter Bastian[1], Markus Blatt[1], Andreas Dedner[2], Christian Engwer[1], Robert Klöfkorn[2], Mario Ohlberger[2], and Oliver Sander[3]

[1] Interdisziplinäres Zentrum für Wissenschaftliches Rechnen, Universität Heidelberg, Im Neuenheimer Feld 368, D-69120 Heidelberg, Germany
`Markus.Blatt@iwr.uni-heidelberg.de`,
WWW home page: `http://hal.iwr.uni-heidelberg.de/dune/index.html`

[2] Abteilung für Angewandte Mathematik, Universität Freiburg, Hermann-Herder-Str. 10, D-79104 Freiburg, Germany

[3] Institut für Mathematik II, Freie Universität Berlin, Arnimallee 2-6, D-14195 Berlin, Germany

Abstract. 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 finite element discretization scheme, can work efficiently on different mesh implementations. These ideas have also been extended to vectors and sparse matrices where iterative solvers can be written in a generic way using the interface. These components are available within the "Distributed Unified Numerics Environment" (DUNE).

1 Introduction

There exist many simulation packages for the numerical solution of partial differential equations (PDEs) 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 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.

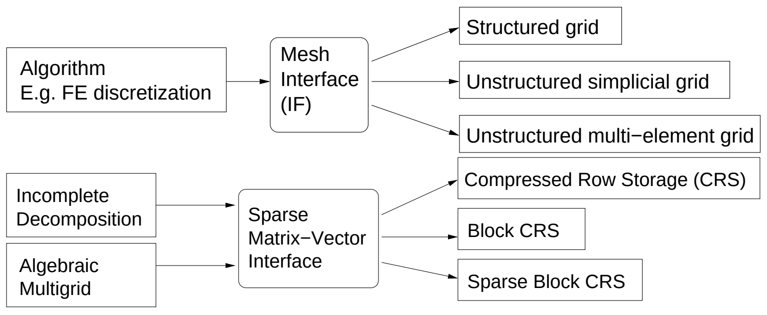

Fig. 1. Encapsulation of data structures with abstract interfaces.

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
– choses at compile-time 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.

Another important aspect is that the interface and its implementations are realized using generic programming techniques. Using static polymorphism instead of dynamic polymorphism allows one to have very small functions in the interface without introducing a severe performance penalty. We have choosen the C++ programming language since it is widely available and highly optimizing compilers exist. Finally, it is important that the interface is designed for code reuse. Several major finite element packages have been implemented under the new interface.

This concept of abstract interfaces has so far been realized for two different components of the PDE solution process: 1) the mesh interface and 2) the matrix-vector interface. Both components will be discussed and numerical results confirming the efficiency of the approach will be presented.

2 Mesh Interface

The abstract mesh interface supports a wide range of finite element meshes which are in common use:

- Fully unstructured grids.
- Elements with arbitrary shape (there is a way to define reference elements) and arbitrary transformation from the reference element to the actual element.
- Full dimension independence. Types of mesh entities are parametrized with dimension and codimension.
- Meshes on manifolds.
- Conforming and non-conforming meshes.
- Nested local mesh refinement with arbitrary refinement rules (e. g. bisection, red-green type, hanging nodes)
- Distributed meshes for data parallel computations supporting overlapping and non-overlapping decompositions.

The following important aspects have been taken into account in the design of the interface:

- User data associated with mesh entities (e. g. degrees of freedom in the finite element method) is stored outside the mesh in arrays (contiguous memory locations). Flexible and effective ways of accessing data from a mesh entity are provided.
- A grid is viewed as a container of entities. Access to entities is only possible via iterators. This allows on-the-fly implementation for simple (e. g. structured) meshes.
- The interface provides only a view on an existing mesh which is expressed in the code through a consequent use of the `const` keyword. The only way to modify a mesh is through nested local mesh refinement.
- Several mesh objects of different type can be instantiated in one executable in order to couple problems on different meshes.

Currently the following implementations are available:

- `SGrid`: equidistant structured grid, on-the-fly generation, n-dimensional, all codimensions are supported.
- `YaspGrid`: equidistant structured grid, on-the-fly generation, parallel with arbitrary overlap, n-dimensional with only codimension 0 and n.
- `AlbertaGrid`: unstructured simplicial mesh in 1, 2 and 3 space dimensions, local refinement using bisection, adaptation of the finite element toolbox Alberta [1].
- `UGGrid`: unstructured multi-element meshes in 2 and 3 space dimensions, red-green type local mesh refinement, non-overlapping decomposition for parallel processing, adaptation of the finite element toolbox UG [2].
- `ALU3DGrid`: unstructured tetrahedral and hexahedral meshes, local mesh refinement with hanging nodes, non-overlapping parallel data decomposition, adaptation of the finite element toolbox ALU3d [3].

3 Iterative Solver Template Library

Sparse matrices obtained from finite element discretizations exhibit a lot of structure that is usually not exploited in available sparse matrix packages. In Fig. 2 several examples are shown: (a) discretization of three-component system with linear finite elements and point-wise ordering, (b) p-adaptive discontinuous Galerkin method, (c) system of reaction-diffusion equations, (d) discretization of Stokes' equation with equation-wise ordering. The Iterative Solver Template Library (ISTL), which is the linear algebra and solver part of DUNE, allows the definition recursively block-structured vectors and matrices at comile-time through the use of templates.

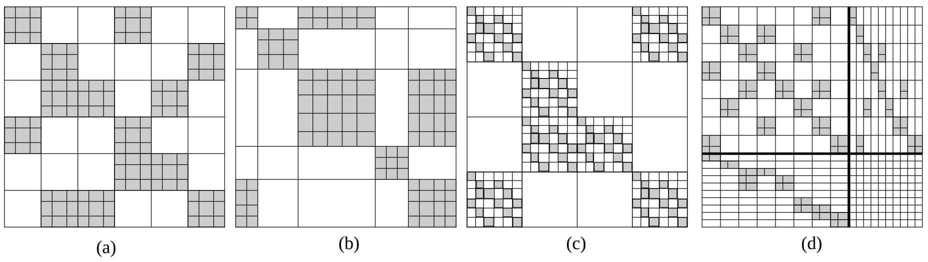

Fig. 2. Block structure of matrices arising in the finite element method.

Vectors and matrices are viewed as one- and two-dimensional containers and provide the same functionality as the sparse BLAS standard. On top of this interface a variety of Krylov methods (Gradient method, CG, BiCGStab) and preconditioners ranging from simple Jacobi, Gauß-Seidel and incomplete decompositions to overlapping Schwarz and algebraic multigrid methods have been implemented. For parallel computations arbitrary data decompositions are possible through the concept of a distributed and possibly overlapping index set.

Numerical results confirm that for a standard model problem this very flexible C++ implementation is as fast as a hand-coded C version for this special problem.

References

1. K. Siebert and A. Schmidt. Design of adaptive finite element software: The finite element toolbox ALBERTA, Springer LNCSE Series 42, 2005.
2. P. Bastian, K. Birken, S. Lang, K. Johannsen, N. Neuß, H. Rentz-Reichert, and C. Wieners. UG: A flexible software toolbox for solving partial differential equations. *Computing and Visualization in Science*, 1:27–40, 1997.
3. A. Dedner, C. Rohde, B. Schupp and M. Wesenberg. A Parallel, Load Balanced MHD Code on Locally Adapted, Unstructured Grids in 3D. *Computing and Visualization in Science*, 7:79–96, 2004.