# The Distributed and Unified Numerics Environment (DUNE*)

Peter Bastian[†] Markus Blatt[†] Christian Engwer[†]
Andreas Dedner[‡] Robert Klökorn[‡] Sreejith P. Kuttanikkad[†]
Mario Ohlberger[‡] Oliver Sander[§]

August 31, 2006

**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

Finite element or finite volume software packages differ mainly 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.

A solution to this problem is to separate data structures and algorithms by an abstract interface. Realizing interface and implementation using generic programming and static polymorphism one is able to write algorithms based on an abstract interface. At compile-time

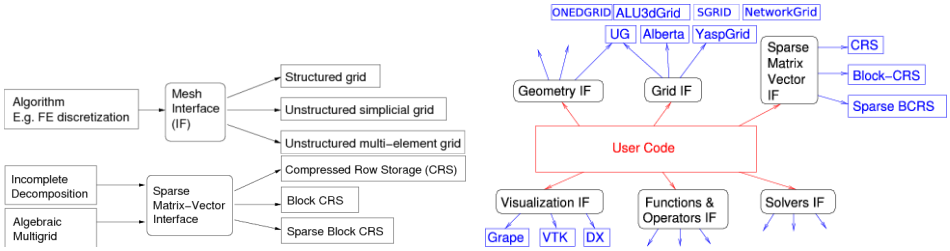
*http://www.dune-project.org

[†]Interdisziplinäres Zentrum für wissenschaftliches Rechnen, Universität Heidelberg, Im Neuenheimer Feld 368, D-69120 Heidelberg

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

[§]Fachbereich Mathematik und Informatik, Freie Universität Berlin, Arnimalle 2-6, D-14195 Berlin-Dahlem

exactly the data structure that fits best to the problem is chosen allowing the application of all compiler optimizations.
Figure 1(a) shows how this concept is a applied in the case of a discretization scheme accessing the mesh data structure and an algebraic multigrid method accessing a sparse matrix data structure through an abstract interface. The interfaces can be implemented in different ways, each offering a different set of features efficiently.




(a) Encapsulation of data structures with abstract interfaces.

(b) DUNE module structure

Figure 1: DUNE modules and 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.

# 2 The DUNE Library

Writing algorithms based on abstract interfaces is not a new concept. In object oriented languages abstract base classes and inheritence are often used to implement polymorphism. E. g. C++ offers virtual functions to implement dynamic polymorphism. The function call itself poses a serious performance penalty in the case where the function in the interface itself consists only of a few instructions. Therefore, function calls and virtual methods can only be used efficiently for interfaces with sufficiently coarse granularity.
However, to utilize the concept of abstract interfaces to it's full extend one needs interfaces with fine granularity. E. g., in the mesh one needs to access the coordinates of nodes, normals of faces or evaluate element transformations at individual quadrature points. Generic programming, implemented in the C++ programming language through templates, offers the possiblity to implement interfaces whithout performance penalty. The abstract algorithm is parameterized by a concrete implementation of the interface at compile-time allowing the compiler to inline small functions and employ all code optimzations. Basically the interface is removed completely at compile time. This technique is called static polymorphism and is extensively used in the standard template library STL, see [MDS01]. Many C++ programming techniques we use are described in [BN94] and [Vel00].

DUNE is supposed to be a template library for all software components required for the numerical solution of partial differential equations. Figure 1(b) shows the high level design. User code will access geometries, grids, sparse linear solvers, etc. through the abstract interfaces. Many specialized implementations of one interface are possible and particular implementations are selected at compile time. This concept allows easy incorporation of existing codes aa well as coupling of different codes.

## 2.1 The Grid Interface

The abstract mesh interface supports finite element grids with the following properties:

- Grids with elements of arbitrary dimension embedded in a space with the same or higher dimension can be described.
- Grids may have elements of arbitrary shape (there is a way to define reference elements) and arbitrary transformation from the reference element to the actual element.
- Grids may be nonconforming, i. e. the intersection of two elements need not be a common vertex, edge or face.
- Local refinement is always nested. I. e. every fine grid element results from subdivision of exactly one coarse grid element.
- Grids may be partitioned into overlapping subgrids for parallel processing.

A grid in the DUNE interface is viewed as a readonly container of entities (i. e. vertices, faces, elements, etc.). The only way to modify a mesh once it is created is through mesh refinement. Access to the entities is only possible via iterators allowing on-the-fly implementations of simple (e. g. structured) meshes.
Several mesh objects of different type can be instantiated in one executable in order to couple problems on different grids.
Each grid provides mappings of classes of entities onto consecutive indices. This allows the storage of user data ouside of the mesh in contiguous memory location (e. g. arrays or ISTL vectors), e. g. for efficient usage of the cache in the fast linear algebra.
See Table 1 for a list of available implementations of the DUNE grid interface.

## 2.2 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), the linear algebra and solver interface of DUNE, allows the definition of recursively block-structured vectors and matrices at compile-time through the use of templates.

| | |
|---|---|
| OneDGrid | A 1d grid with local mesh refinement. |
| NetworkGrid | A grid representing networks of 1d elements in an arbitrary dimensional world capable of local mesh refinement. |
| SGrid | Equidistant structured grid, on-the-fly generation, parallel with arbitrary overlap, $n$-dimensional with only codimension 0 and $n$. |
| YaspGrid | A structured parallel grid in $n$ space dimensions. |
| AlbertaGrid | Unstructured simplicial mesh in 1, 2 and 3 space dimensions, local refinement using bisection, adaptation of the finite element toolbox Alberta [SS05]. |
| 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 [BBL$^+$97]. |
| ALU3DGrid | Unstructured tetrahedral and hexahedral meshes, local mesh refinement with hanging nodes, non-overlapping parallel data decomposition, adaptation of the finite element toolbox ALU3d [DRSW04]. |

Table 1: Availabe Grid implementations

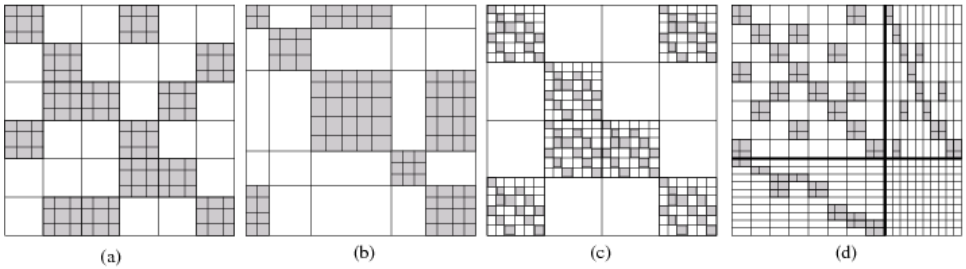



Figure 2: Block structure of matrices arising in the finite element method.

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 supported in a generic way.
The MFLOP rates achieved with ISTL compiled with GNU C++ 4.0 on a Pentium 4 Mobile 2.4 GHz (see Tables 2 and 3) show nearly optimal performance. The stream benchmark achieved 1024 MB/s transfer rate for large numbers of unknowns $N$ on this machine. As expected one can see in Table 3 that for matrices with nonscalar dense blocks (blocksize $b > 1$) better MFLOP rates are achieved due to the better usage of the cache. Here the structure is like in Figure 2a.

| $N$ | 500 | 5000 | 50000 | 500000 | 5000000 |
|---|---|---|---|---|---|
| MFLOPS | 936 | 910 | 108 | 103 | 107 |

Table 2: MFLOPS for daxpy operation: $y = y + \alpha x$, 1200 MB/s transfer rate for large $N$

| $N, b$ | 100,1 | 10000,1 | 1000000,1 | 1000000,2 | 1000000,3 |
|---|---|---|---|---|---|
| MFLOPS | 388 | 140 | 136 | 230 | 260 |

Table 3: MFLOPS for matrix vector product (BCRSMatrix, 5-point stencil, $b$: block size of dense small matrix blocks)

# 3 Sample application

We consider the second order elliptic model problem

$$-\Delta u = f \qquad \text{in } \Omega = (-1/2, 1/2) \times (0,1) \times (0,1), \tag{1}$$

$$-\nabla u \cdot n = 0 \qquad \text{on } \Gamma_N = \{(x,0,z) | -1/2 < x < 0, 0 < z < 1\}, \tag{2}$$

$$u = g \qquad \text{on } \delta\Omega \setminus \Gamma_N \tag{3}$$

where the right hand side $f$ and Dirichlet boundary conditions $g$ have been chosen such that the solution $u$ is $u(r,\varphi,z) = r^{\frac{1}{2}} \sin\left(\frac{\varphi}{2}\right) 4z(1-z)$ in cylindrical coordinates. This solution $u$ has a singularity along the line $(1/2, 0, z)$. The solution is depicted graphically in Figure 3.

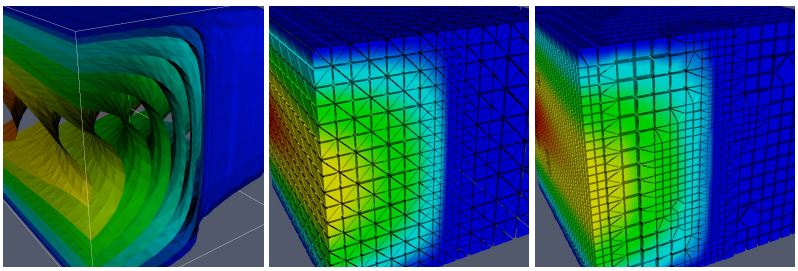


Figure 3: Grid function of the exact solution for the elliptic model problem and adaptivly refined grids generated with AlbertaGrid and UGGrid

Eqs. (1)-(3) are solved numerically using standard conforming $P_1$ finite elements on adaptively refined grids using a residual based error estimator $\|u - u_h\|_1 \leq C\sqrt{\sum_{e \in L^0} \eta_e}$ with the local estimators

$$\eta_e = h_e^2 \int_{\omega_e} f^2 \, dx + \frac{1}{2} \sum_{\substack{\lambda=(e,e',\ldots)\in\mathcal{I}, \\ \omega_\lambda \not\subset \partial\Omega}} h_\lambda \int_{\omega_\lambda} [\nabla u \cdot n]^2 \, ds + \sum_{\substack{\lambda=(e,e,\ldots)\in\mathcal{I}, \\ \omega_\lambda \subset \Gamma_N}} h_\lambda \int_{\omega_\lambda} |\nabla u \cdot n|^2 \, ds.$$

The generic implementation of the adaptive finite element method works on grids of all element types, space dimension, as well as with conforming and nonconforming refinement (hanging nodes).

| Grid | $N$ | MAT | ASS | SLV | EST | ADP | REF | RS | ERR[1] |
|---|---|---|---|---|---|---|---|---|---|
| s, Alberta | 496304 | 12.7 | 16.2 | 5.6 | 46.2 | 37.1 | 9.4 | 1.00 | 7.7 |
| s, Alu3d | 537515 | 31.7 | 32.8 | 8.3 | 37.9 | 24.1 | 5.3 | 1.06 | 12.7 |
| c, UG | 365891 | 8.5 | 15.0 | 5.0 | 15.6 | 15.5 | 13.2 | 0.69 | 13.3 |
| c, ALU3d | 360118 | 11.0 | 12.6 | 4.2 | 9.2 | 5.2 | 1.0 | 0.49 | 14.7 |

Table 4: Timing for various components of the adaptive algorithm

Table 4 shows timings for different parts of the adaptive algorithm on the different grids. All times are given in seconds and have been measured on a Laptop-PC with an Intel T2500 Core Duo processor with 2.0 GHz, 667 MHz FSB and 2 MB L2 cache using the GNU C++ compiler in version 4.0 and -O3 optimization.

# References


[BBL$^+$97] 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.

[BN94] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley, 1994.

[DRSW04] 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.

[MDS01] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 2001.

[SS05] K. Siebert and A. Schmidt. *Design of adaptive finite element software: The finite element toolbox ALBERTA*. Number 42 in LNCSE. Springer, 2005.

[Vel00] T. Veldhuizen. Techniques for scientific c++. Technical Report 542, Indiana University Computer Science, 2000. http://osl.iu.edu/ tveldhui/papers/techniques/.


[1] In the first column (Grid) s and c identify grids using simplices and cubes, respectively. The other columns give the number of degrees of freedom ($N$), the time for setting up the non-zero structure of the sparse matrix in block compressed row storage from the grid (MAT), the time for assembling the matrix entries (ASS), the time for solving the linear system with conjugate gradients preconditioned with symmetric Gauß-Seidel (residual norm reduction $10^{-3}$) (SLV), the time to evaluate the error estimator (EST), the total time for grid adaptation including reorganization of the vector of unknowns (ADP), the time for grid refinement only (this is part of ADP) (REF), the relative speed computed as the sum of columns MAT through ADP divided by $N$ and normalizing that number relative to ALBERTA (RS), the $L_2$ error multiplied with $10^5$ on the given mesh (ERR).