The Distributed and Unified Numerics Environment

Christian Engwer

Institute of Parallel and Distributed Systems, University of Stuttgart

February 29, 2008

christian.engwer@ipvs.uni-stuttgart.de
http://www.dune-project.org/

The Problem with Finite Element Software

Problem:

- There are many PDE software packages, each with a particular set of features:
  - IPARS: block structured, parallel, multiphysics.
  - Alberta: simplicial, unstructured, bisection refinement.
  - UG: unstructured, multi-element, red-green refinement, parallel.
  - QuocMesh: Fast, on-the-fly structured grids.
- Using one framework, it might be
  - either impossible to have a particular feature,
  - or very inefficient in certain applications.
- Extension of the feature set is usually hard.

Reason:

Algorithms must be implemented on the basis of a particular data structure

The Problem with Finite Element Software

Problem:

- There are many PDE software packages, each with a particular set of features:
  - IPARS: block structured, parallel, multiphysics.
  - Alberta: simplicial, unstructured, bisection refinement.
  - UG: unstructured, multi-element, red-green refinement, parallel.
  - QuocMesh: Fast, on-the-fly structured grids.
- Using one framework, it might be
  - either impossible to have a particular feature,
  - or very inefficient in certain applications.
- Extension of the feature set is usually hard.

Reason:

Algorithms must be implemented on the basis of a particular data structure

Outline

Design Principles

Flexibility: Seperation of data structures and algorithms.

Efficiency: Generic programming techniques.

Legacy Code: Reuse existing finite element software.

Design Principles

Flexibility: Seperation of data structures and algorithms.

Efficiency: Generic programming techniques.

Legacy Code: Reuse existing finite element software.

Design Principles

Flexibility: Seperation of data structures and algorithms.

Efficiency: Generic programming techniques.

Legacy Code: Reuse existing finite element software.

Design Principles

Flexibility: Seperation of data structures and algorithms.

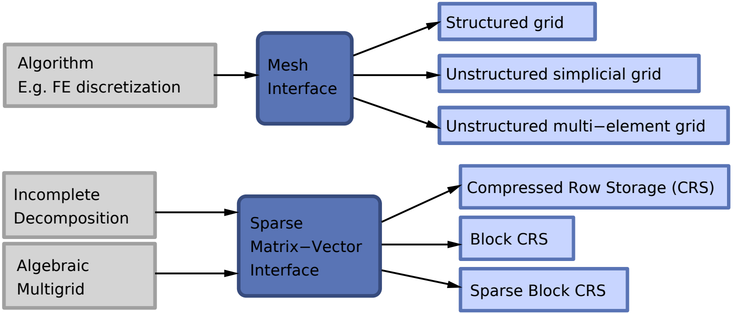
Efficiency: Generic programming techniques.

Legacy Code: Reuse existing finite element software.

Flexibility

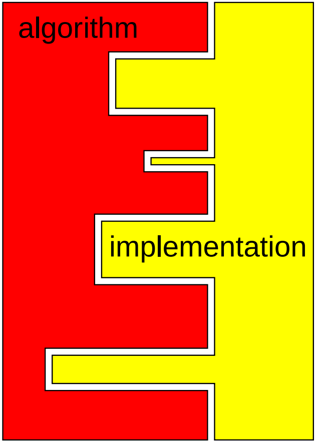
Seperate data structures and algorithms.

- The algorithm determins the data structure to operate on.
- Data structures are hidden under a common interface.
- Algorithms work only on that interface.
- Different implementations of the interface.



Efficiency

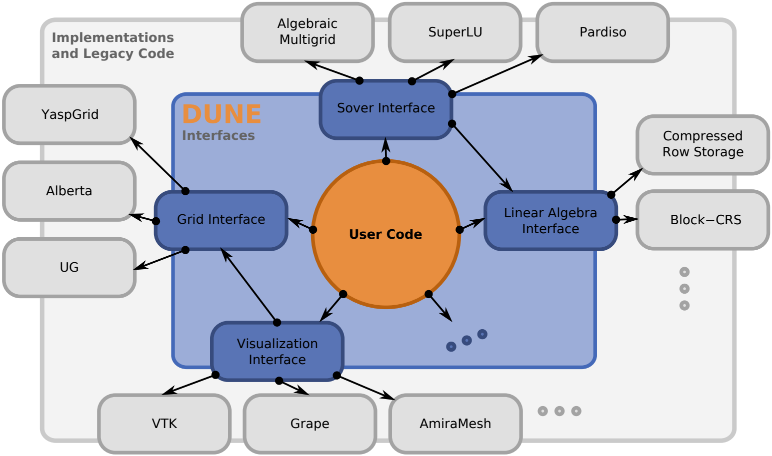
Implementation with generic programming techniques.



- Static Polymorphism $\rightarrow$ Compile-time selection of data structures.
- Compiler generates code for each (algorithm,data structure) combination.
- Allows interfaces with fine granularity.
- All optimizations apply, in particular function inlining.
- see i.e. STL, Blitz++, MTL,. . .
- and Thesis of Gundram Berti (2000): Concepts for grid based algorithms.

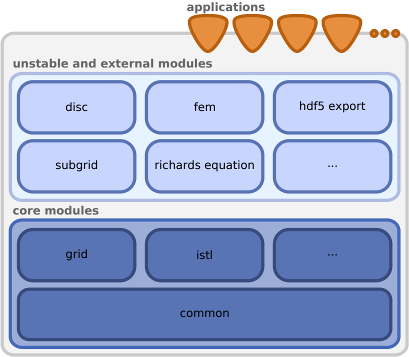
Reuse existing finite element software.

Efficient integration of existing FE software, using interfaces and generic programming.



The Development of DUNE

- Modules
  - Code is split into differnt modules.
  - Applications use only the modules they need.
  - Modules are sorted according to level of maturaty.
  - Everybody can provide his own modules.
- Portability
- Open Development Process
- Free Software Licence



Central contact point is **`http://www.dune-project.org/`**

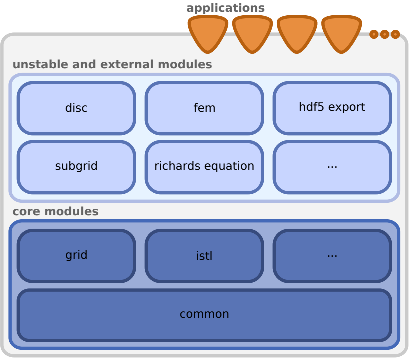
DUNE Core 1.0

Current stable version is 1.0, available since 20th december 2007.

dune-common: foundation classes, infrastructure

dune-grid: grid interface, quadrature rules, visualization

dune-istl: *(Iterative Solver Template Library)* generic sparse matrix/vector classes, solvers (Krylov methods, AMG, etc.)



DUNE Developers

Thanks to my fellow developers

A project like this could not be possible without . . .

- the core developers
  - Peter Bastian
  - Markus Blatt
  - Andreas Dedner
  - Christian Engwer
  - Robert Klöfkorn
  - Mario Ohlberger
  - Oliver Sander
- all the users and testers
- and many other contributers.

Generic Programming Techniques

1. Static Polymorphism
   - Engine Concept (see STL)
   - Curiously Recurring Template Pattern (Barton and Nackman)
2. Iterators
   - Generic access to different data structures.
3. View Concept
   - Access to different partitions of one data set.

Static Polymorphism

vs. Dynamic Polymorphism

Dynamic Polymorphism

- the “usual” polymorphism
- allows exchangeability at run time
- impedes a variaty of optimizations, e.g.
  - inlining
  - loop unrolling
- additional overhead

Static Polymorphism

- allows exchangeability only at compile time
- allows all optimizations
- longer compile time

$\Rightarrow$ especially for fine grained interfaces with short functions ($\leq$ 25 FLOPS), static polymorphism is to be prefered.

Static Polymorphism

vs. Dynamic Polymorphism

Dynamic Polymorphism

- the “usual” polymorphism
- allows exchangeability at run time
- impedes a variaty of optimizations, e.g.
  - inlining
  - loop unrolling
- additional overhead

Static Polymorphism

- allows exchangeability only at compile time
- allows all optimizations
- longer compile time

$\Rightarrow$ especially for fine grained interfaces with short functions ($\leq$ 25 FLOPS), static polymorphism is to be prefered.

Static Polymorphism

vs. Dynamic Polymorphism

Dynamic Polymorphism

- the “usual” polymorphism
- allows exchangeability at run time
- impedes a variaty of optimizations, e.g.
  - inlining
  - loop unrolling
- additional overhead

Static Polymorphism

- allows exchangeability only at compile time
- allows all optimizations
- longer compile time

$\Rightarrow$ especially for fine grained interfaces with short functions ($\leq$ 25 FLOPS), static polymorphism is to be prefered.

Static Polymorphism

Engine Concept

- Used in the STL
- A certain interface is assumed
- Now language features to ensure a certain interface
- Weired errors if this interface is not fulfilled

Barton Nackman Trick

- Recursive template patterns shall ensure a given interface
- Only a trick to work around missing language features

Static Polymorphism

Engine Concept

- Used in the STL
- A certain interface is assumed
- Now language features to ensure a certain interface
- Weired errors if this interface is not fulfilled

Barton Nackman Trick

- Recursive template patterns shall ensure a given interface
- Only a trick to work around missing language features

Iterators

Iterators are a common concept in the STL.

- A container object owns the data.
- Iterators give access to this data.
- A 1-D ordering of the container is required.
- Iterators are a generalization of pointers.
- They allow algorithms to operate on very different containers.

see: Todd Veldhuizen, *Techniques for scientific C++*, 1999.

View Concept

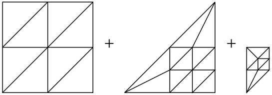
- The container contents ("Data") exist outside the container objects.
- The containers are lightweight handles ("Views") of the Data.
- Multiple containers can refer to the same data, but provide different views.
- "View-Only" containers allow a clear seperation of resposibilities:
  - wide use of **const** allows better optimizations.
  - data modification only in very destinct places

    $\Rightarrow$ allows an even wider range of data structures.

see: Todd Veldhuizen, *Techniques for scientific C++*, 1999.

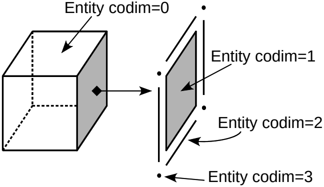
Grid

A formal specification of grids is required to enable an accurate description of the grid interface.



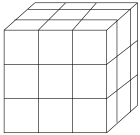
Hierarchic Grid

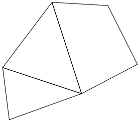
- A (hierarchic) grid has a dimension d, a world dimension w and maximum level J.

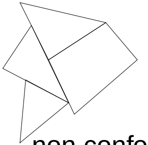


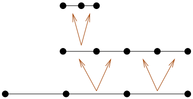
- A grid is a Container of entities (geometrical/topological objects) of different codimensions.

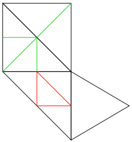
Supports a wide range of Grids

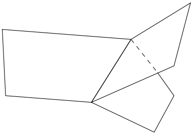

structured

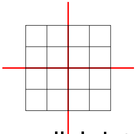

conforming

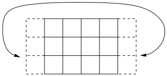

non conforming

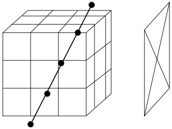

nested, 1D


red-green, bisektion


manifolds


parallel data decomposition


periodic


mixed dimensions

Grid Interface

Barton-Nackman Trick:
Used for all classes associated with a Grid.

View Model:
Read-only access to grid entities, consequent use of **`const`**.

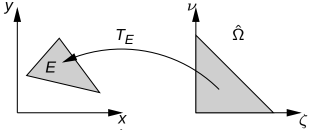
- level view
- leaf view

Iterators:
Access to entities is only through iterators for a certain view.

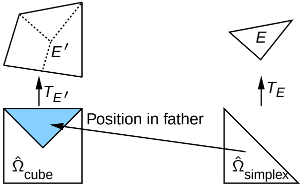
$\Rightarrow$ *Allows on-the-fly implementations.*

Several instances of a grid with different dimension and implementation can coexist in a single program.

Entities



Mapping from $\hat{\Omega}$ into global coordinates.



Entity E is defined by...

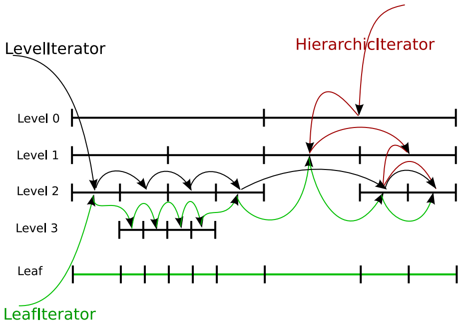
- Reference Element $\hat{\Omega}$
  - Describes all topological information.
  - Can be recursively constructed over dimension.
- Transformation T_E
  - Maps from the reference element into global coordinates.
  - Provides Jacobian, its inverse and tangential vectors.

Entity of Codimension 0 provides...

- subentity and father relations.
- intersections with neighbours and boundary.

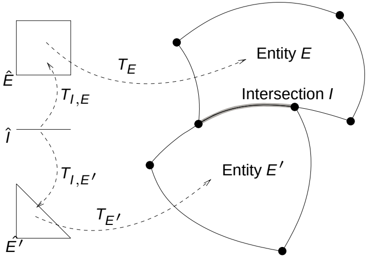
Iterators

Access different views of the grid



- `LeafIterator<d>` iterates over codimension 0 leaf entities.
- `LevelIterator<c,d>` iterates over codimension *c* entities on a given level.
- `HierarchicIterator<d>` iterate over all childs of a codimension 0 entity.

Intersections



- Grids may be non conforming.
- Entities can intersect with neighbors and boundary.
- IntersectionsIterators give access to intersections of an Entity in a given view.
- IntersectionsIterators hold topological and geometrical information.
- Two types, corresponding the two major views:
  - `LeafIntersectionIterator`
  - `LevelIntersectionIterator`
- **Note:** Intersections are always of codimension 1!

Indices and Ids

- Allow association of FE computations data with subsets of entities.
- Subsets could be "vertices of level *l*", "faces of leaf elements", . . .
- Data should be stored in arrays for efficiency.
- Associate index/id with each entity.

  Three types are used:

  Leaf index: zero-starting, consecutive, non-persistent, accessible on copies.
  Used to store solution and stiffness matrix.

  Level index: zero-starting, consecutive, non-persistent.
  Used for geometric multigrid.

  Globally unique id: persistent across grid modifications.
  Used to transfer solution from one grid to another.

Indices and Ids

- Allow association of FE computations data with subsets of entities.
- Subsets could be “vertices of level l”, “faces of leaf elements”, ...
- Data should be stored in arrays for efficiency.
- Associate index/id with each entity.

Three types are used:

Leaf index: zero-starting, consecutive, non-persistent, accessible on copies.
Used to store solution and stiffness matrix.

Level index: zero-starting, consecutive, non-persistent.
Used for geometric multigrid.

Globally unique id: persistent across grid modifications.
Used to transfer solution from one grid to another.

Grid Modification

Modification Methods:

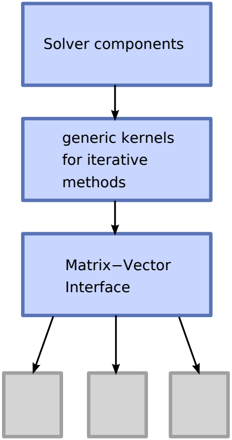
- Global Refinement
- Local Refinement & Adaption
- Load Balancing

$\Rightarrow$ View-Only Concept

- Views offer access to data
- Data can only be modified in the primal container *(the Grid)*

Linear Algebra Interface

Iterative Solver Template Library



Situtation:

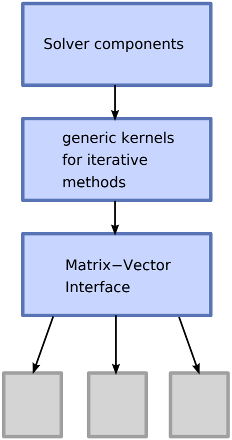
- There are already template libraries for linear algebra: MTL/ITL
- Existing libraries cannot efficiently use (small) structure of FE-Matrices

Interface:

- Solver components: Based on operator concept, Krylov methods, (A)MG preconditioners
- Generic kernels: Triangular solves, Gauss-Seidel step, ILU decomposition
- Matrix-Vector Interface: Support recursively block structured matrices

Linear Algebra Interface

Iterative Solver Template Library



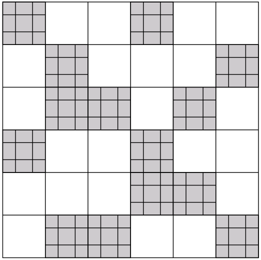
Situtation:

- There are already template libraries for linear algebra: MTL/ITL
- Existing libraries cannot efficiently use (small) structure of FE-Matrices

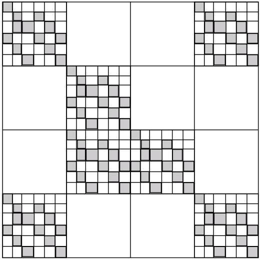
Interface:

- Solver components: Based on operator concept, Krylov methods, (A)MG preconditioners
- Generic kernels: Triangular solves, Gauss-Seidel step, ILU decomposition
- Matrix-Vector Interface: Support recursively block structured matrices

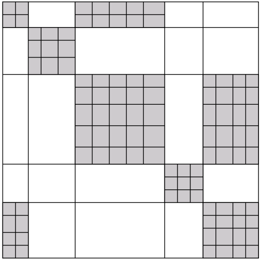
Block Structure in FE Matrices



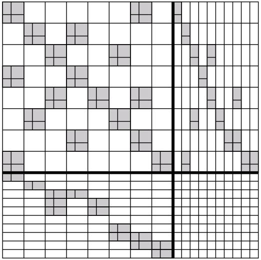
- sparse block matrix
- blocks are dense
- blocks have fixed size
- DG fixed p



- blocks are sparse
- diffusion-reaction systems



- blocks are dense
- blocks have variable size
- DG hp version



- 2x2 block matrix
- each block is sparse
- Taylor-Hood elements

Vector-Matrix Interface

Vector

- Is a one-dimensional container
- Sequential access
- Random access
- Vector space operations: Addition, scaling
- Scalar product
- Various norms
- Sizes

Matrix

- Is a two-dimensional container
- Sequential access using iterators
- Random access
- Organization is row-wise
- Mappings
  $y = y + Ax; y = y + A^T x; y = y + A^H x;$
- Solve, inverse, left multiplication
- Various norms
- Sizes

Engine Concept:

Solver use Kernels via Engine Concept.

Iterators:

Kernels operator on Iterators. This allows very different Matrix/Vector Implementations.

Vector-Matrix Interface

Vector

- Is a one-dimensional container
- Sequential access
- Random access
- Vector space operations: Addition, scaling
- Scalar product
- Various norms
- Sizes

Matrix

- Is a two-dimensional container
- Sequential access using iterators
- Random access
- Organization is row-wise
- Mappings
  $y = y + Ax; y = y + A^T x; y = y + A^H x;$
- Solve, inverse, left multiplication
- Various norms
- Sizes

Engine Concept:

Solver use Kernels via Engine Concept.

Iterators:

Kernels operator on Iterators. This allows very different Matrix/Vector Implementations.

Example Definitions

- A vector containing 20 blocks where each block contains two complex numbers using **double** for each component:

```
typedef FieldVector<complex<double>,2> MyBlock;
BlockVector<MyBlock> x(20);
x[3][1] = complex<double>(1,-1);
```

- A sparse matrix consisting of sparse matrices having scalar entries:

```
typedef FieldMatrix<double,1,1> DenseBlock;
typedef BCRSMatrix<DenseBlock> SparseBlock;
typedef BCRSMatrix<SparseBlock> Matrix;
Matrix A(10,10,40,Matrix::row_wise);
... // fill matrix
A[1][1][3][4][0][0] = 3.14;
```

Conclusions

Publication: P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, O. Sander. **A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework**. Submitted to *Computing*.

P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, O. Sander. **A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part II: Implementation and Tests in DUNE**. Submitted to *Computing*.

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

Distributed and Unified Numerics Environment

Conclusions

- DUNE is based on the following principles:
  - Flexibility through seperation of data structures and algorithms.
  - Efficiency through Generic Programming Techniques.
  - Reuse of existing codes.
- Free and Open Software.
- Offers flexibility with hardly any performance penalty.
- Current plans:
  - Constant improvements of the core modules.
  - New (unified) discretization module.

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

Distributed and Unified Numerics Environment

Performance of the Grid Interface

- Consider Run-time for computing FE interpolation error for polynomial degree 1 and quadrature order 2.
- Same algorithm runs on `YaspGrid` and `UGGrid`

Grid	d	Type	Elements	Time [s]
`UGGrid`	2	simplex	131072	0.49
`UGGrid`	2	cube	65536	0.19
`YaspGrid`	2	cube	65536	0.09
`UGGrid`	3	cube	32768	0.19
`YaspGrid`	3	cube	32768	0.12

- `YaspGrid` is on-the-fly compared to `UGGrid`.
- Basis functions are not cached.

Performance Linear Algebra

- Matrix-Vector performance
  - Pentium 4 Mobile 2.4 GHz, Compiler: GNU C++ 4.0
  - Stream benchmark for $x = y + \alpha z$ is 1084 MB/s
  - Scalar product of two vectors

N	500	5000	50000	500000	5000000
MFLOPS	896	775	167	160	164

  - daxpy operation $y = y + \alpha x$, 1200 MB/s transfer rate for large N

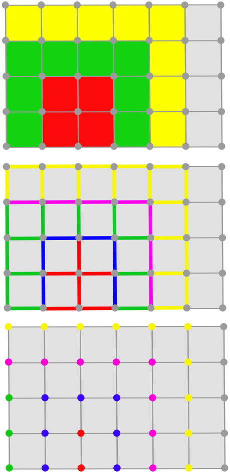
N	500	5000	50000	500000	5000000
MFLOPS	936	910	108	103	107

- Damped Gauß-Seidel solver
  - 5-point stencil on 1000×1000 grid
  - Comparison generic implementation in ISTL with specialized C implementation in AMGLIB

	AMGLIB	ISTL
Time per iteration [s]	0.17	0.18

  - Corresponds to about 150 MFLOPS

Parallel Data Decomposition



- Grid is mapped to $\mathcal{P} = \{0, \dots, P-1\}$.
- Each Entities is present on one or more processors.
- Each Entities is associated to **one** “partition type”.
- partition types:

interior	Nonoverlapping decomposition.	
overlap	Arbitrary size.	
ghost	Rest.	
border	Boundary of interior.	*(codimension$>$0)*
front	Boundary of interior+overlap.	*(codimension$>$0)*

- Allows implementation of overlapping and nonoverlapping Domain Decomposition methods.