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

1. Design Principles
2. The Development of DUNE
3. Generic Programming Techniques
4. DUNE Grid Interface
5. Linear Algebra Interface
6. Conclusions
Design Principles

Flexibility: Separation of data structures and algorithms.

Efficiency: Generic programming techniques.

Legacy Code: Reuse existing finite element software.
Design Principles

**Flexibility:** Separation of data structures and algorithms.

**Efficiency:** Generic programming techniques.

**Legacy Code:** Reuse existing finite element software.
Design Principles

Flexibility: Separation of data structures and algorithms.

Efficiency: Generic programming techniques.

Legacy Code: Reuse existing finite element software.
Design Principles

Flexibility: Separation of data structures and algorithms.

Efficiency: Generic programming techniques.

Legacy Code: Reuse existing finite element software.
Flexibility

Separate data structures and algorithms.

- The algorithm determines 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, . . .
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 different modules.
  - Applications use only the modules they need.
  - Modules are sorted according to level of maturity.
  - Everybody can provide his own modules.

- Portability
- Open Development Process
- Free Software Licence

Central contact point is [http://www.dune-project.org/](http://www.dune-project.org/)
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.)
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 contributors.**
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 variety of optimizations, e.g.
  - inlining
  - loop unrolling
- additional overhead

⇒ especially for fine grained interfaces with short functions (≤ 25 FLOPS), static polymorphism is to be preferred.

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

vs. Dynamic Polymorphism

Dynamic Polymorphism

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

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

Static Polymorphism

- allows exchangeability only at compile time
- allows all optimizations
- longer compile time
Static Polymorphism vs. Dynamic Polymorphism

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

⇒ especially for fine grained interfaces with short functions (≤ 25 FLOPS), static polymorphism is to be preferred.

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

Engine Concept

- Used in the STL
- A certain interface is assumed
- Now language features to ensure a certain interface
- Weird 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
- Weird 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 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.

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 separation of responsibilities:
  - wide use of `const` allows better optimizations.
  - data modification only in very distinct places
    ⇒ allows an even wider range of data structures.

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

- 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
- nested, 1D
- parallel data decomposition
- conforming
- red-green, bisektion
- periodic
- non conforming
- manifolds
- 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.
⇒ *Allows on-the-fly implementations.*

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


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!
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

⇒ View-Only Concept

• Views offer access to data
• Data can only be modified in the primal container (the Grid)
Situation:

- 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
Situation:

- 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
  \[ y = y + Ax; \quad y = y + A^T x; \quad y = y + A^H x; \]
- 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
  \[ y = y + Ax; \quad y = y + A^T x; \quad y = y + A^H x; \]
- 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


Conclusions

- DUNE is based on the following principles:
  - Flexibility through separation 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

Christian Engwer (IPVS, Stuttgart)
Consider Run-time for computing FE interpolation error for polynomial degree 1 and quadrature order 2.

Same algorithm runs on YaspGrid and UGGrid

<table>
<thead>
<tr>
<th>Grid</th>
<th>$d$</th>
<th>Type</th>
<th>Elements</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>UGGrid</td>
<td>2</td>
<td>simplex</td>
<td>131072</td>
<td>0.49</td>
</tr>
<tr>
<td>UGGrid</td>
<td>2</td>
<td>cube</td>
<td>65536</td>
<td>0.19</td>
</tr>
<tr>
<td>YaspGrid</td>
<td>2</td>
<td>cube</td>
<td>65536</td>
<td>0.09</td>
</tr>
<tr>
<td>UGGrid</td>
<td>3</td>
<td>cube</td>
<td>32768</td>
<td>0.19</td>
</tr>
<tr>
<td>YaspGrid</td>
<td>3</td>
<td>cube</td>
<td>32768</td>
<td>0.12</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$N$</th>
<th>500</th>
<th>5000</th>
<th>50000</th>
<th>500000</th>
<th>5000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFLOPS</td>
<td>896</td>
<td>775</td>
<td>167</td>
<td>160</td>
<td>164</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$N$</th>
<th>500</th>
<th>5000</th>
<th>50000</th>
<th>500000</th>
<th>5000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFLOPS</td>
<td>936</td>
<td>910</td>
<td>108</td>
<td>103</td>
<td>107</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>AMGLIB</th>
<th>ISTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time per iteration [s]</td>
<td>0.17</td>
</tr>
</tbody>
</table>

- Corresponds to about 150 MFLOPS
Parallel Data Decomposition

- Grid is mapped to $\mathcal{P} = \{0, \ldots, P-1\}$.
- Each Entity is present on one or more processors.
- Each Entity is associated to one “partition type”.
- partition types:
  - interior: Nonoverlapping decomposition.
  - overlap: Arbitrary size.
  - ghost: Rest.
  - border: Boundary of interior. ($\text{codimension} > 0$)
  - front: Boundary of interior + overlap. ($\text{codimension} > 0$)
- Allows implementation of overlapping and nonoverlapping Domain Decomposition methods.