Outline

1 Motivation

2 Expressing Structure in FE Matrices

3 Performance

4 Solver Components

5 Agglomeration Algebraic Multigrid

6 Parallel Agglomeration Algebraic Multigrid

7 Conclusions
**DUNE**

- **Distributed Unified Numerics Environment (http://dune.uni-hd.de)**
  - Separate data structures and algorithm
  - Formulate algorithms based on interfaces
  - Provide different implementations of the interface
  - No lack performance due to generic programming
  - Parts of DUNE
    - Grid interface (not covered)

- **Iterative Solvers Library (ISTL)**

```plaintext
Algorithm
E.g. FE discretization
```

```plaintext
Mesh Interface (IF)
```

```plaintext
Structured grid
```

```plaintext
Unstructured simplicial grid
```

```plaintext
Unstructured multi–element grid
```

```plaintext
Incomplete Decomposition
```

```plaintext
Algebraic Multigrid
```

```plaintext
Sparse Matrix–Vector Interface
```

```plaintext
Compressed Row Storage (CRS)
```

```plaintext
Block CRS
```

```plaintext
Sparse Block CRS
```
Structure of ISTL

- There are already template libraries for linear algebra: MTL/ITL
- Existing libraries cannot efficiently use (small) structure of FE-Matrices
- 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
- Various implementations of the interface are available
Example Definitions

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

  ```cpp
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:

  ```cpp
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;
```
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
Performance

- Pentium 4 Mobile 2.4 GHz: Stream for $x = y + \alpha z$ is 1084 MB/s
- Compiler: GNU C++ compiler version 4.0
- Scalar product of two vectors (block size 1)

<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>

- Matrix-vector product, BCRSMatrix, 5-point stencil, $b$: block size

<table>
<thead>
<tr>
<th>$N, b$</th>
<th>100,1</th>
<th>10000,1</th>
<th>1000000,1</th>
<th>1000000,2</th>
<th>1000000,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFLOPS</td>
<td>388</td>
<td>140</td>
<td>136</td>
<td>230</td>
<td>260</td>
</tr>
</tbody>
</table>
template<class M, class X, class Y, class K>
static void dbgs (const M& A, X& x, const Y& b, const K& w) {

typedef typename M::ConstRowIterator rowiterator;
typedef typename M::ConstCollIterator coliterator;
typedef typename Y::block_type bblock;
typedef typename X::block_type xblock;

bblock rhs; X xold(x); rowiterator endi=A.end();
for (rowiterator i=A.begin(); i!=endi; ++i) {  // loop over rows
    rhs = b[i.index()];                    // initialize rhs
    coliterator endj=(*i).end();          // end of row i
    coliterator j=(*i).begin();          // start of row i
    for (; j.index()<i.index(); ++j)     // lower triangle
        (*j).mmv(x[j.index()],rhs);
    coliterator diag=j;
    for (; j!=endj; ++j)                 // upper triangle
        (*j).mmv(x[j.index()],rhs);
    algmeta_itsteps<l-1>::dbgs(*diag,x[i.index()],rhs,w); //''solve''
}

x *= w; x.axpy(1-w,xold);            // update with damping
}
Performance II

- Damped Gauss-Seidel solver
- 5-point stencil on 1000 by 1000 grid
- Comparison of generic implementation in ISTL with specialized C implementation in AMGLIB

<table>
<thead>
<tr>
<th></th>
<th>AMGLIB</th>
<th>ISTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time per iteration [s]</td>
<td>0.17</td>
<td>0.18</td>
</tr>
</tbody>
</table>

- Corresponds to about 150 MFLOPS
Operator and Solver Concept

Operator Concept

- Let $A : X \rightarrow Y$, $x \mapsto A(x)$ be a linear Operator with $X$, $Y$ vector spaces.
- Class LinearOperator
  - apply(const X& x, Y& y) : $y = A(x)$
  - applyscaleadd(field_type alpha, const X& x, Y& y): $y = y + \alpha A(x)$
- Problem: Find $x \in X$ such that $A(x) = b$ for $b \in Y$.

Solver Concept

- Preconditioned iterative solvers, e.g. LoopSolver, CGSolver, BiCGStabSolver
- Inherit from abstract base class InverseOperator
- Use only abstract Operator interface functions, provided scalarproduct and preconditioner
Operator and Solver Concept

Operator Concept

• Let \( A : X \rightarrow Y, \ x \mapsto A(\mathbf{x}) \) be a linear Operator with \( X, \ Y \) vector spaces.

• Class LinearOperator
  
  • apply(const X& x, Y& y) : \( y = A(\mathbf{x}) \)
  
  • appliescaleadd(field_type alpha, const X& x, Y& y): \( y = y + \alpha A(\mathbf{x}) \)

• Problem: Find \( \mathbf{x} \in X \) such that \( A(\mathbf{x}) = \mathbf{b} \) for \( \mathbf{b} \in Y \).

Solver Concept

• Preconditioned iterative solvers, e. g. LoopSolver, CGSolver, BiCGStabSolver

• Inherit from abstract base class InverseOperator

• Use only abstract Operator interface functions, provided scalarproduct and preconditioner
Parallelism

- Solvers programmed to the interface of preconditioner, scalarproduct and operator.
- Parallelism is hidden in Operator, Preconditioner and Scalarproduct.
- E.g. OverlappingSchwarzScalarProduct, BlockPreconditioner, parallel agglomeration algebraic multigrid.

```cpp
typedef Dune::OverlappingSchwarzScalarProduct<
    Vector, Communication>
ScalarProduct;
typedef Dune::SeqJac<BCRSMat, Vector, Vector> SeqPrec;
typedef Dune::BlockPreconditioner<
    Vector, Vector, Communication, SeqPrec>
ParPrec;
ScalarProduct sp(comm);
SeqPrec sprec(fop.getmat(), 1, 1);
ParSmoothing pprec(spren, comm);
Dune::CGSolver<Vector> cg(fop, sp, pprec, 10e−08, 10, 0);
cg.apply(x, b, r);
```
Characteristics of Agglomeration AMG

Simple multigrid algorithm

- $P_l$: piecewise constant
- $R_l = P_l^T$
- $A_{l-1} = R_l A_l P_l$
- Proposed by Raw, Vanek et al., Braess

Clustering controlled by

- Strong coupling
- desired size (4, 8)
- minimize fill-in

Observations

- Preserves FV discretization
- Preserves sign of M-matrix
- $O(J)$ iterations for model problem in $d = 2, 3$
- Quite robust for variable coefficient elliptic problems
- $O(J)$ optimal anyway for 2d variable coefficient problems
- Reasonable coarse grid operator for systems
- Allows efficient data-parallel implementation
Characteristics of Agglomeration AMG

**Simple multigrid algorithm**

- $P_l$: piecewise constant
- $R_l = P_l^T$
- $A_{l-1} = R_l A_l P_l$
- Proposed by Raw, Vanek et al., Braess

**Clustering controlled by**

- Strong coupling
- desired size (4, 8)
- minimize fill-in

**Observations**

- Preserves FV discretization
- Preserves sign of M-matrix
- $O(J)$ iterations for model problem in $d = 2, 3$
- Quite robust for variable coefficient elliptic problems
- $O(J)$ optimal anyway for 2d variable coefficient problems
- Reasonable coarse grid operator for systems
- Allows efficient data-parallel implementation
Characteristics of Agglomeration AMG

Simple multigrid algorithm

- $P_i$: piecewise constant
- $R_i = P_i^T$
- $A_{i-1} = R_i A_i P_i$
- Proposed by Raw, Vanek et al., Braess

Clustering controlled by

- Strong coupling
- desired size (4, 8)
- minimize fill-in

Observations

- Preserves FV discretization
- Preserves sign of M-matrix
- $O(J)$ iterations for model problem in $d = 2, 3$
- Quite robust for variable coefficient elliptic problems
- $O(J)$ optimal anyway for 2d variable coefficient problems
- Reasonable coarse grid operator for systems
- Allows efficient data-parallel implementation
Scalar Elliptic Problem

- Solve $\nabla \cdot \{k(x, y)\nabla u\} = f$ in $(0, 1)^2$; $u = g$ on $\partial \Omega$
- AMG(2,2,1) SSOR used as preconditioner in CG
- Iteration numbers for $10^{-8}$ reduction

<table>
<thead>
<tr>
<th>$N$</th>
<th>$k(x, y) = 1$</th>
<th>$k(x, y) = \cdots$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$64^2$</td>
<td>7</td>
<td>$10^{-1}$</td>
</tr>
<tr>
<td>$128^2$</td>
<td>9</td>
<td>$10^{-3}$</td>
</tr>
<tr>
<td>$256^2$</td>
<td>10</td>
<td>$10^3$</td>
</tr>
<tr>
<td>$512^2$</td>
<td>12</td>
<td>$10^1$</td>
</tr>
<tr>
<td>$1024^2$</td>
<td>14</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$2048^2$</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

- Coarsening costs about 3 iterations
Illustration of Agglomeration

Agglomeration is matrix dependent
Follows “strong” connections

homogeneous  checker board  anisotropic
Illustration of Algorithm
Illustration of Algorithm
Illustration of Algorithm
Illustration of Algorithm
Illustration of Algorithm
Illustration of Algorithm
Illustration of Algorithm
Illustration of Algorithm
Scalability Results

CG + AMG prec. + Jacobi(2) smoother, $10^{-8}$ residual reduction

2D heterogeneous problem: $P \cdot 2000^2$, max. $1.6 \cdot 10^9$ unknowns.

<table>
<thead>
<tr>
<th>$P$</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>400</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{build}[s]$</td>
<td>96</td>
<td>103</td>
<td>110</td>
<td>118</td>
<td>123</td>
<td>128</td>
</tr>
<tr>
<td>$T_{solve}[s]$</td>
<td>209</td>
<td>298</td>
<td>331</td>
<td>366</td>
<td>410</td>
<td>407</td>
</tr>
<tr>
<td>$T_{it}[s]$</td>
<td>8.0</td>
<td>9.9</td>
<td>10.0</td>
<td>10.2</td>
<td>10.3</td>
<td>10.4</td>
</tr>
<tr>
<td>#IT</td>
<td>26</td>
<td>30</td>
<td>33</td>
<td>36</td>
<td>40</td>
<td>39</td>
</tr>
</tbody>
</table>

3D heterogeneous problem $P \cdot 150^3$, max. $7.3 \cdot 10^8$ unknowns.

<table>
<thead>
<tr>
<th>$P$</th>
<th>1</th>
<th>8</th>
<th>27</th>
<th>64</th>
<th>125</th>
<th>216</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{build}[s]$</td>
<td>216</td>
<td>228</td>
<td>242</td>
<td>245</td>
<td>251</td>
<td>276</td>
</tr>
<tr>
<td>$T_{solve}[s]$</td>
<td>213</td>
<td>352</td>
<td>294</td>
<td>467</td>
<td>519</td>
<td>443</td>
</tr>
<tr>
<td>$T_{it}[s]$</td>
<td>7.6</td>
<td>9.8</td>
<td>7.7</td>
<td>10.2</td>
<td>9.1</td>
<td>10.3</td>
</tr>
<tr>
<td>#IT($10^{-8}$)</td>
<td>28</td>
<td>36</td>
<td>38</td>
<td>46</td>
<td>57</td>
<td>43</td>
</tr>
</tbody>
</table>
Conclusions

- ISTL is based on the following principles
  - Matrix and vector interface recursive block structure.
  - Algorithms use structure of the finite element methods.
  - No performance lack.
  - Same solver algorithms and code for all implementations due to generic programming.
  - Solver algorithms support sequential and parallel usage.
  - Robust preconditioners for heterogeneous problems

- Current plans
  - Release 1.0 of Dune http://dune.uni-hd.de
  - Apply AMG to DG discretizations (next talk!)