PDEs with a Trillion Elements
When Theory Fails to Predict Performance

U. Rüde (LSS Erlangen, ruede@cs.fau.de)

joint work with

and many more students

Lehrstuhl für Informatik 10 (Systemsimulation)
Universität Erlangen-Nürnberg
www10.informatik.uni-erlangen.de

Dagstuhl, February 2, 2009
Overview

Motivation:
- Towards PetaScale and Beyond

Efficient Parallel Multigrid Software
- **Data Local Iterative Methods** (DiMe)
- MultiCore MultiGrid on the IBM-Cell Processor
- Hierarchical Hybrid Grids (HHG)
- *Parallel Expression Templates for PDE* (ParExPDE)

Conclusions
How fast are our algorithms (multigrid) on current CPUs

**Assumptions:**
- Multigrid requires 27.5 Ops/unknown to solve an elliptic PDE (Griebel ´89 for Poisson)
- A modern laptop CPU delivers >10 GFlops peak

**Consequence:**
- We should solve one million unknowns in 0.00275 seconds
- ~ 3 ns per unknown

**Revised Assumptions:**
- Multigrid takes 500 Ops/unknown to solve your favorite PDE
- you can get 5% of 10 Gflops performance

**Consequence:** On your laptop you should
- solve one million unknowns in 1.0 second
- ~ 1 microsecond per unknown

Consider Banded Gaussian Elimination on the Play Station (Cell Processor), single Prec. 250 GFlops, for 1000 x 1000 grid unknowns
- ~2 Tera-Operations for factorization - will need about 10 seconds to factor the system
- requires 8 GB Mem.
- Forward-backward substitution should run in about 0.01 second, except for bandwidth limitations
Trends in Computer Architecture

- On Chip Parallelism
  - instruction level
  - multicore
- Off Chip parallelism
- Limits to clock rate
- Limits to memory bandwidth and latency
Part II

DiMe

Cache-Aware Multigrid
Multigrid: V-Cycle

Goal: solve $A^h u^h = f^h$ using a hierarchy of grids

Relax on

$A^h u^h = f^h$

Correct

$u^h \leftarrow u^h + e^h$

Residual

$r^h = f^h - A^h u^h$

Restrict

$r^H = I_h^H r^h$

Interpolate

$e^h = I_H^h e^H$

Solve

$A^H e^H = r^H$

by recursion

Goal: solve $A^h u^h = f^h$ using a hierarchy of grids
**V(2,2) cycle - bottom line**
(old results)

<table>
<thead>
<tr>
<th>Mflops</th>
<th>For what</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Standard 5-pt. Operator</td>
</tr>
<tr>
<td>56</td>
<td>Cache optimized (loop orderings, data merging, simple blocking)</td>
</tr>
<tr>
<td>150</td>
<td>Constant coeff. + skewed blocking + padding</td>
</tr>
<tr>
<td>220</td>
<td>Eliminating rhs if 0 everywhere but boundary</td>
</tr>
</tbody>
</table>
Part III
Multicore Architectures
Multigrid on STI Cell
Part IV

Hierarchical Hybrid Grids
Parallel High Performance FE Multigrid

- Parallelize "plain vanilla" multigrid
  - tune single core performance first
  - partition domain
  - parallelize all operations on all grids
  - use clever data structures

- Do not worry (so much) about Coarse Grids
  - idle processors?
  - short messages?
  - sequential dependency in grid hierarchy?

- Why we do not use conventional domain decomposition
  - DD without coarse grid does not scale (algorithmically) and is suboptimal for large problems/many processors
  - DD with coarse grids may be as efficient as multigrid but is as difficult to parallelize (the difficulty is in parallelizing the coarse grid)
Hierarchical Hybrid Grids (HHG)

- Joint work with
- Frank Hülsemann (now EDF), Ben Bergen (now Los Alamos), T. Gradl (still Erlangen)

**HHG Goal: Ultimate Parallel FE Performance!**

- unstructured adaptive refinement grids with
  - regular substructures for
  - efficiency
  - superconvergence effects
HHG refinement example

Input Grid
HHG Refinement example

Refinement Level one
HHG Refinement example

Refinement Level Two
HHG Refinement example

Structured Interior
HHG Refinement example

Structured Interior
HHG Refinement example

Edge Interior
HHG Refinement example

Edge Interior
Adaptivity in HHG (with conforming meshes)
Parallel scalability of scalar elliptic problem discretized by tetrahedral finite elements.

Times for 12 $V(2,2)$ cycles on SGI Altix: Itanium-2 1.6 GHz.

Largest problem solved to date: $3.07 \times 10^{11}$ DOFS on 9170 Procs: 7.8 s per $V(2,2)$ cycle

<table>
<thead>
<tr>
<th>#Proc</th>
<th>#unkn. x $10^6$</th>
<th>Ph.1: sec</th>
<th>Ph. 2: sec</th>
<th>Time to sol.</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>134.2</td>
<td>3.16</td>
<td>6.38*</td>
<td>37.9</td>
</tr>
<tr>
<td>8</td>
<td>268.4</td>
<td>3.27</td>
<td>6.67*</td>
<td>39.3</td>
</tr>
<tr>
<td>16</td>
<td>536.9</td>
<td>3.35</td>
<td>6.75*</td>
<td>40.3</td>
</tr>
<tr>
<td>32</td>
<td>1,073.7</td>
<td>3.38</td>
<td>6.80*</td>
<td>40.6</td>
</tr>
<tr>
<td>64</td>
<td>2,147.5</td>
<td>3.53</td>
<td>4.92</td>
<td>42.3</td>
</tr>
<tr>
<td>128</td>
<td>4,295.0</td>
<td>3.60</td>
<td>7.06*</td>
<td>43.2</td>
</tr>
<tr>
<td>252</td>
<td>8,455.7</td>
<td>3.87</td>
<td>7.39*</td>
<td>46.4</td>
</tr>
<tr>
<td>504</td>
<td>16,911.4</td>
<td>3.96</td>
<td>5.44</td>
<td>47.6</td>
</tr>
<tr>
<td>2040</td>
<td>68,451.0</td>
<td>4.92</td>
<td>5.60</td>
<td>59.0</td>
</tr>
<tr>
<td>3825</td>
<td>128,345.7</td>
<td>6.90</td>
<td></td>
<td>82.8</td>
</tr>
<tr>
<td>4080</td>
<td>136,902.0</td>
<td></td>
<td>5.68</td>
<td></td>
</tr>
<tr>
<td>6102</td>
<td>205,353.1</td>
<td></td>
<td>6.33</td>
<td></td>
</tr>
<tr>
<td>8152</td>
<td>273,535.7</td>
<td></td>
<td>7.43*</td>
<td></td>
</tr>
<tr>
<td>9170</td>
<td>307,694.1</td>
<td></td>
<td>7.75*</td>
<td></td>
</tr>
</tbody>
</table>

Part V

Towards User Friendly Scalable FE Software
ParExPDE
(Parallel Expression Templates for PDE)

- work done by C. Freundl
- A library for the user friendly, rapid development of numerical PDE solvers on parallel (super-)computers
- Provides a high level and intuitive user interface without compromising on efficiency
- Regularly refined hexahedral grids
- Support for multigrid hierarchies
C++ Expression Templates

- Encapsulation of arithmetic expressions
  - tree-like structure
  - C++ template constructs
- Evaluation of expression at compile time
- Avoid unnecessary copying and temporary objects

Elegance & Performance
C++ Expression Templates

```
Evaluation of an expression:
```
```
template <class T>
void Vector::operator=(Expr<T>& expr) {
    for (int i = 0; i < _size; i++)
        _values[i] = expr.valueAt(i);
}
```
C++ Expression Templates

\[ z = a \times x + b \]

C++ Compiler
(Template instantiation, Inlining)

for (int i = 0; i < z._size; i++) {
  z._values[i] =
    a \times x._values[i] + b._values[i];
}

Subsequent compiler optimisations can be applied
Program code of a V-cycle:

```java
for (int l = 0; l < nlevels - 1; l++) {
    for (int s = 0; s < npre; s++) {
        u = u + (f - laplace(u)) / Diag(laplace) | interior_points;
    }
    r = f - laplace(u) | interior_points;
    r.doRestrict();
    f.levelDown();
    f = r;
    u.levelDown();
    u = 0.0;
}
```
ParExPDE: Serial Performance

- AMD Opteron 848:
  - Rpeak = 4.4 GFLOPS
  - Memory bandwidth: 5.3 GB/s
  - Machine balance: 0.1506

- Jacobi smoother (constant coefficients):
  - 28 loads, 1 store
  - 56 floating point operations
  - Loop balance: 0.5179

→ maximum achievable performance:
  \[(0.1506 / 0.5179) \times 4.4 \text{ GFLOPS} = 1280 \text{ MFLOPS}\]
Implementation of Jacobi smoother with ParExPDE

- Intel C++ compiler 9.1
- Carefully chosen optimisation flags

→ Code performs with up to 990 MFLOPS

- Excellent performance for pure C++ code
ParExPDE: Parallel Performance

Strong scaleup of Jacobi smoother on LSS cluster (210 hexahedrons of size $100^3$)
ParExPDE: Parallel Performance

Weak scaleup of MG V(2,2) solver on HLRB 2
\(\approx 1.7 \cdot 10^7\) unknowns per processor

Av. Time per V-cycle

Time per V-cycle

Number processors

\(\approx 1.7 \cdot 10^7\) unknowns per processor
Part VI

Conclusions