From Peta-Scale to Exa-Scale Computing with Multilevel Methods

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

joint work with

J. Götz, M. Stürmer, K. Iglberger, S. Donath, C. Feichtinger, T. Preclik,
T. Gradl, C. Freundl, H. Köstler, T. Pohl, D. Ritter, D. Bartuschat, P. Neumann,
G. Wellein, G. Hager, T. Zeiser, J. Habich (RRZE)
N. Thürey (ETH Zürich)

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

May 7, 2010
LSEC, Beijing
Overview

- From PetaScale to Exa-Scale
  - Trends in Supercomputer Architecture
- Optimizing Memory Access and Cache-Aware Programming
- Massively Parallel Performance Results
- MultiCore Architectures
- Multigrid on Cell Processor and Graphics Cards
- Conclusions
Part I

Supercomputer Architecture
Motivation 1: A key experiment 10 years ago
Structured vs. Unstructured Grids

Extinct Dinosaur HLRB-I: Hitachi SR 8000
No. 5 in TOP-500 in 2000
2 TFlops

- MFlops rates for matrix-vector multiplication on one node
- Structured versus sparse matrix
- Many emerging architectures have similar properties (Cell, GPU)
Motivation II: Theory versus Practice

**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
Motivation III: Power Considerations
How much is a PetaFlops?

- $10^6 = 1$ MegaFlops: Intel 486 33MHz PC (~1989)
- $10^9 = 1$ GigaFlops: Intel Pentium III 1GHz (~2000)
  - If every person on earth computes one operation every 6 seconds, all humans together have ~1 GigaFlops performance (less than a current laptop)
- $10^{12} = 1$ TeraFlops: HLRB-I 1344 Proc., ~2000
- $10^{15} = 1$ PetaFlops
  - 294,912 Cores (Jugene, 2009)
  - If every person on earth runs a 486 PC, we all together have an aggregate Performance of 6 PetaFlops.
- $10^{18} = 1$ ExaFlops (around 2020)?
IBM Blue Gene

0.825 petaflop/s performance speed running the Linpack benchmark.

Theoretical peak capability 1.0027 Petaflop/s

294,912 cores

#4 on TOP 500 List

Nov 2009

Björn Gmeiner (LSS) visiting Jülich Supercomputing Center

Picture taken March 2010
What’s the problem?

replacing 4 strong jet engines

Would you want to propel a Super Jumbo

with 300,000 blow dryer fans?
Evolution of Semiconductor Technology

- Collects trends in semiconductor technology
- See [http://www.itrs.net/reports.html](http://www.itrs.net/reports.html)

<table>
<thead>
<tr>
<th></th>
<th>2008</th>
<th>2011</th>
<th>2014</th>
<th>2020</th>
</tr>
</thead>
<tbody>
<tr>
<td>nm</td>
<td>57</td>
<td>40</td>
<td>28</td>
<td>14</td>
</tr>
<tr>
<td>M Tr/ Chip</td>
<td>1106</td>
<td>2212</td>
<td>4424</td>
<td>17696</td>
</tr>
<tr>
<td>On Chip/local GHz</td>
<td>10</td>
<td>17</td>
<td>28</td>
<td>73</td>
</tr>
<tr>
<td>Chip-Board GHz</td>
<td>6</td>
<td>11</td>
<td>23</td>
<td>72</td>
</tr>
<tr>
<td>Number pins/chip</td>
<td>4400</td>
<td>5094</td>
<td>5896</td>
<td>7902</td>
</tr>
</tbody>
</table>
Exponential growth in parallelism for the foreseeable future.
Where does Computer Architecture Go?

- Computer architects have capitulated: It may not be possible anymore to exploit progress in semiconductor technology for automatic performance improvements.
  - Even today a single core CPU is a highly parallel system:
  - superscalar execution, complex pipeline, ... and additional tricks
  - Internal parallelism is a major reason for the performance increases until now, but ...
  - There is a limited amount of parallelism that can be exploited automatically

- Multi-core systems concede the architects’ defeat:
  - Architects fail to build faster single core CPUs given more transistors
  - Clock rate increases only slowly (due to power considerations)
  - Therefore architects have started to put several cores on a chip:
    - programmers must use them directly
What are the consequences?

- For the application developers “the free lunch is over”
  - Without explicitly parallel algorithms, the performance potential cannot be used any more
- For HPC
  - CPUs will have 2, 4, 8, 16, ..., 128, ..., ??? cores - maybe sooner than we are ready for this
  - We will have to deal with systems with millions of cores
What are the problems?

- Unprecedented levels of parallelism
  - maybe billions of cores/threads needed

- Hybrid architectures
  - standard CPU
  - vector units (SSE)
  - accelerators (GPU)

- Memory wall
  - memory response slow: latency
  - memory transfer limited: bandwidth

- Power considerations dictate
  - limits to clock speed => multi core
  - limits to memory size (byte/flop)
  - limits to address references per operation
  - limits to fault safety
Memory Wall
Cache Aware Programming
Computer Memory Hierarchy
(from wikipedia)
Memory Organization

- Registers, typically 32-128 double precision
- Load-Store architecture
- Explicit use of registers: assembly language (?) or register attribute in C, but often ignored by compiler.
- Register allocation algorithms are quite good in state of the art compilers (but not perfect)
- More registers in future CPUs (128 in Itanium)
- Latency for decoding
- Instruction set (register windows)
- Register allocation more complex, compile times get longer
Caches

- Fast (but small) extra memory
  - holding identical copies of main memory
  - lower latency
  - higher bandwidth
  - usually several levels (2 or 3)
  - same principle as virtual memory

- Memory request satisfied from
  - fast cache, if the data is stored there: cache hit
  - or from slow main memory, if the data is not stored in the cache: cache miss

- Issues:
  - Uniqueness and transparency of addressing
  - Finding a working set
  - Data consistency with main memory
1. direct mapped (associativity 1): Each main memory word can be stored in one (and only) one location in the cache.

2. (fully) associative: A main memory word can be stored in any location in the cache.

3. set associative (associativity k, typically k=2,4,8,...): Each main memory word can be stored in one of k places in the cache.

\[\text{1 and 3 give rises to conflict misses.}\]

\[\text{Direct mapped caches are faster, fully associative caches too expensive and slow (if reasonably large). Set-associative caches are a compromise.}\]
Part II

Optimizing Memory Access and Cache-Aware Programming
More Motivation: DiMe - Project

Data Local Iterative Methods (1996-2007) for the Efficient Solution of Partial Differential Equations

www10.informatik.uni-erlangen.de/de/Research/Projects/DiME/

- Started jointly with Linda Stals in 1996 in Ausgburg!
- Cache-optimizations for sparse matrix/stencil codes (1996-2007)
- Efficient hardware optimized
- Multigrid Solvers
- Lattice-Boltzmann CFD
  - with free surface flow
  - fluid structure interaction
Increasing single-CPU performance by optimizing data locality

Caches work due to the locality of memory accesses (instructions + data)

(Numerically intensive) codes should exhibit:

- **Spatial locality:**
  Data items accessed within a short time period are located close to each other in memory

- **Temporal locality:**
  Data that has been accessed recently is likely to be accessed again in the near future

**Goal:** Increase spatial and temporal locality in order to enhance cache utilization (cache-aware progr.)
Cache performance optimizations

- Data layout optimizations:
  Change the data layout in memory to enhance spatial locality

- Data access optimizations:
  Change the order of data accesses to enhance spatial and temporal locality

These transformations preserve numerical results and their introduction can (theoretically) be automated!
Data access optimizations:
Loop fusion

Example: red/black Gauss-Seidel iteration in 2D
Data access optimizations:
Loop fusion (cont’d)

Code **before** applying loop fusion technique
(standard implementation w/ efficient loop ordering,
Fortran semantics: row major order):

```plaintext
for it= 1 to numIter do
    // Red nodes
    for i= 1 to n-1 do
        for j= 1+(i+1)%2 to n-1 by 2 do
            relax(u(j,i))
        end for
    end for
end for
```
Data access optimizations: Loop fusion (cont’d)

// Black nodes
for i = 1 to n-1 do
    for j = 1+i%2 to n-1 by 2 do
        relax(u(j,i))
    end for
end for

This requires **two sweeps** through the whole data set per single GS iteration!
Data access optimizations: Loop fusion (cont’d)
Data access optimizations: Loop fusion (cont’d)

Code **after** applying loop fusion technique:

```plaintext
for it= 1 to numIter do
    // Update red nodes in first grid row
    for j= 1 to n-1 by 2 do
        relax(u(j,1))
    end for
```
Data access optimizations: Loop fusion (cont’d)

// Update red and black nodes in pairs
for i = 1 to n-1 do
    for j = 1+(i+1)%2 to n-1 by 2 do
        relax(u(j,i))
        relax(u(j,i-1))
    end for
end for
// Update black nodes in last grid row
for j= 2 to n-1 by 2 do
    relax(u(j,n-1))
end for

Solution vector $u$ passes through the cache only once instead of twice per GS iteration!
Data access optimizations: Loop split

- The inverse transformation of loop fusion
- Divide work of one loop into two to make body less complicated
  - Leverage compiler optimizations
  - Enhance instruction cache utilization
Data access optimizations: Loop blocking

- Loop blocking = loop tiling
- Divide the data set into subsets (blocks) which are small enough to fit in cache
- Perform as much work as possible on the data in cache before moving to the next block
- This is not always easy to accomplish because of data dependencies
Data access optimizations: Loop blocking

Example: 1D blocking for red/black GS, respect the data dependencies!

```
Update direction

i-2
i-1
i
t
i+1

> >
```

---

LEHRSTUHL FÜR INFORMATIK 10 (SYSTEMSIMULATION)
Data access optimizations: Loop blocking

- Code after applying 1D blocking technique
- \( B = \) number of GS iterations to be blocked/combined

```plaintext
for it = 1 to numIter/B do
    // Special handling: rows 1, ..., 2B-1
    // Not shown here ...
```
Data access optimizations: Loop blocking

// Inner part of the 2D grid
for k = 2*B to n-1 do
    for i = k to k-2*B+1 by -2 do
        for j = 1+(k+1)%2 to n-1 by 2 do
            relax(u(j,i))
            relax(u(j,i-1))
        end for
    end for
end for
Data access optimizations: Loop blocking

// Special handling: rows n-2B+1, ..., n-1
// Not shown here ...
end for

- Result: Data is loaded once into the cache per B Gauss-Seidel iterations, provided that 2*B+2 grid rows fit in the cache simultaneously
- If grid rows are too large, 2D blocking can be applied
Cache Blocking: leaping over the memory wall

- **Idea:** Change the order of operations to increase cache locality
- **matrix-matrix-multiplication**
  - **blocking:** multiplications of sub-matrices that can be performed completely in-cache each
- **stencil-based kernels**

**Example:** 5-point-stencil

without blocking
Cache Blocking: leaping over the memory wall

- Idea: Change the order of operations to increase cache locality
  - matrix-matrix-multiplication
  - add multiplications of sub-matrices that can be performed completely in-cache each
- stencil-based kernels

Example: 5-point-stencil

without blocking
Cache Blocking: leaping over the memory wall

- Idea: Change the order of operations to increase cache locality
- matrix-matrix-multiplication
  - add multiplications of sub-matrices that can be performed completely in-cache each
- stencil-based kernels

Example: 5-point-stencil

spatial blocking
Cache Blocking: leaping over the memory wall

- Idea: Change the order of operations to increase cache locality
- matrix-matrix-multiplication
  - add multiplications of sub-matrices that can be performed completely in-cache each
- stencil-based kernels

Example: 5-point-stencil

spatial blocking
Cache Blocking: leaping over the memory wall

Idea: Change the order of operations to increase cache locality

- matrix-matrix-multiplication
  - add multiplications of sub-matrices that can be performed completely in-cache each

- stencil-based kernels

Example: 5-point-stencil

temporal blocking
Cache Blocking: leaping over the memory wall

- Idea: Change the order of operations to increase cache locality
  - matrix-matrix-multiplication
    - add multiplications of sub-matrices that can be performed completely in-cache each
  - stencil-based kernels

Example: 5-point-stencil
temporal blocking
Temporal LS-blocking of 3D Stencil Code?

- options for temporal cache blocking in parallel
  - synchronize data accesses at boundaries ✗
  - compute required intermediate results locally ✔
- waste on block-boundaries
  (high surface/volume ratio for small LS)
- alignment and size of DMA and SIMD
  constraint block size
- low potential (estimate: max. 150% for 2× temporal blocking)

»Not even Markus would like to program that!«
Data access optimizations
Loop blocking

- More complicated blocking schemes exist
- Illustration: 2D square blocking
Part III

Towards Scalable FE Software

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

Relax on $A^h u^h = f^h$

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

Restrict $r^H = I_H^h r^h$

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

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

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

by recursion
Cache-optimized multigrid: DiMEPACK library

- DFG project DiME: Data-local iterative methods
- Fast algorithm + fast implementation
- Correction scheme: V-cycles, FMG
- Rectangular domains
- Constant 5-/9-point stencils
- Dirichlet/Neumann boundary conditions
- http://www10.informatik.uni-erlangen.de/dime
## V(2,2) cycle (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>
Parallel High Performance FE Multigrid

- Parallelize „plain vanilla“ multigrid
  - 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?

- Multigrid vs. Domain Decomposition
  - DD without coarse grid does not scale (algorithmically) and is inefficient for large problems/many processors
  - DD with coarse grids is like multigrid and is as difficult to parallelize
  - We get good results for parallel multigrid ...
Hierarchical Hybrid Grids (HHG)

- Unstructured input grid
  - Resolves geometry of problem domain
- Patch-wise regular refinement
  - Generates nested grid hierarchies naturally suitable for geometric multigrid algorithms
- New:
  - Modify storage formats and operations on the grid to exploit the regular substructures
- Does an unstructured grid with 1000 000 000 000 elements make sense?

HHG - Best possible parallel FE performance!
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
Parallel HHG - Framework
Design Goals

To realize good parallel scalability:

- Minimize latency by reducing the number of messages that must be sent
- Optimize for high bandwidth interconnects ⇒ large messages
- Avoid local copying into MPI buffers
HHG for Parallelization

Use regular HHG patches for partitioning the domain
HHG Parallel Update Algorithm

for each vertex do
    apply operation to vertex
end for

update vertex primary dependencies

for each edge do
    copy from vertex interior
    apply operation to edge
    copy to vertex halo
end for

update edge primary dependencies

for each element do
    copy from edge/vertex interiors
    apply operation to element
    copy to edge/vertex halos
end for

update secondary dependencies
Towards Scalable FE Software

Performance Results
Node Performance is Difficult! (B. Gropp)

DiMe project: Cache-aware Multigrid (1996-...)

<table>
<thead>
<tr>
<th>grid size</th>
<th>173</th>
<th>333</th>
<th>653</th>
<th>1293</th>
<th>2573</th>
<th>5133</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>1072</td>
<td>1344</td>
<td>715</td>
<td>677</td>
<td>490</td>
<td>579</td>
</tr>
<tr>
<td>no blocking</td>
<td>2445</td>
<td>1417</td>
<td>995</td>
<td>1065</td>
<td>849</td>
<td>819</td>
</tr>
<tr>
<td>2x blocking</td>
<td>2400</td>
<td>1913</td>
<td>1312</td>
<td>1319</td>
<td>1284</td>
<td>1282</td>
</tr>
<tr>
<td>3x blocking</td>
<td>2420</td>
<td>2389</td>
<td>2167</td>
<td>2140</td>
<td>2134</td>
<td>2049</td>
</tr>
</tbody>
</table>

# Performance of 3D-MG-Smoothie for 7-pt stencil in Mflops on Itanium 1.4 GHz

- Array Padding
- Temporal blocking - in EPIC assembly language
- Software pipelineing in the extreme (M. Stürmer - J. Treibig)

Node Performance is Possible!
<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>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>

Parallel scalability of scalar elliptic problem in 3D discretized by tetrahedral finite elements.

Times to solution on SGI Altix: Itanium-2 1.6 GHz.

Largest problem solved to date: 3.07 x 10^{11} DOFS (1.8 trillion tetrahedra) on 9170 Procs in roughly 90 secs.

B. Bergen, F. Hülsemann, U. Rüde, G. Wellein: ISC Award 2006, also: „Is 1.7× 10^{10} unknowns the largest finite element system that can be solved today?“, SuperComputing, Nov’ 2005.
HLRB-II

- Shared memory architecture
  - 9728 CPU Cores (Itanium2 Montecito Dual Core)
  - 39 TBytes RAM
  - NUMAlink network
  - 62.3 TFLOP/s Peak Performance

Our testing ground for scalability experiments

HLRB-II: SGI Altix 4700 at the Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften No. 10 in TOP-500 of June 2007
Goal: solve \( A^h u^h = f^h \) using a hierarchy of grids

\[
\begin{align*}
A^h u^h & = f^h \\
r^h & = f^h - \frac{4}{3} A^h u^h + \frac{1}{3} A^H I_h^H u^h \\
r^H & = I_h^H r^h \\
e^H & = I_H^H e^H \\
A^H e^H & = r^H
\end{align*}
\]
τ-Extrapolation

- τ-Extrapolation is a multigrid specific technique and works for both CS and FAS
- For CS the defects of two different grid levels are combined
- Higher accuracy is achieved by modifying the coarse grid correction at the finest grid level only

\[ \hat{\tilde{u}}_{h}^{m+1} = u_{h}^{m} + I_{H}^{h} A_{H}^{-1} \left( \frac{4}{3} I_{h}^{H} (f_{h} - A_{h} u_{h}^{m}) - \frac{1}{3} (I_{h}^{H} f_{h} - A_{H} \tilde{I}_{h}^{H} u_{h}^{m}) \right) \]

- special care needed
  - when choosing the restriction operator and
  - the smoothing procedure in order not to destroy the higher accuracy

- In 2-D, for (unstructured) triangular meshes τ-Extrapolation is equivalent to using higher order (quadratic) elements. For 3D this is unknown. See:
Model Problem

\[-\Delta u = f \text{ in } \Omega\]

Boundary conditions (Dirichlet) and solution:

\[u = e^{-20((2x-1+0.5 \sin(2z\pi))^2+(2y-1+0.5 \cos(2z\pi))^2)} + e^{-20((2x-1+0.5 \sin(2z\pi+\pi))^2+(2y-1+0.5 \cos(2z\pi+\pi))^2)}\]
## Consistency Error

<table>
<thead>
<tr>
<th>Levels</th>
<th>Unknowns</th>
<th>Correction Scheme Discr. Error</th>
<th>Consistency</th>
<th>(\tau)-extrapolation Discr. Error</th>
<th>Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>55</td>
<td>(6.75 \cdot 10^{-1})</td>
<td>-</td>
<td>(6.75 \cdot 10^{-1})</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>623</td>
<td>(6.53 \cdot 10^{-2})</td>
<td>3.37</td>
<td>(6.91 \cdot 10^{-2})</td>
<td>3.29</td>
</tr>
<tr>
<td>5</td>
<td>5.855</td>
<td>(1.90 \cdot 10^{-2})</td>
<td>1.78</td>
<td>(1.56 \cdot 10^{-2})</td>
<td>2.14</td>
</tr>
<tr>
<td>6</td>
<td>50.623</td>
<td>(4.95 \cdot 10^{-3})</td>
<td>1.94</td>
<td>(2.44 \cdot 10^{-3})</td>
<td>2.68</td>
</tr>
<tr>
<td>7</td>
<td>420.735</td>
<td>(1.25 \cdot 10^{-3})</td>
<td>1.99</td>
<td>(1.68 \cdot 10^{-4})</td>
<td>3.86</td>
</tr>
<tr>
<td>8</td>
<td>3.43 \cdot 10^6</td>
<td>(3.12 \cdot 10^{-4})</td>
<td>2.00</td>
<td>(1.06 \cdot 10^{-5})</td>
<td>3.98</td>
</tr>
<tr>
<td>9</td>
<td>2.80 \cdot 10^7</td>
<td>(7.77 \cdot 10^{-5})</td>
<td>2.00</td>
<td>(1.95 \cdot 10^{-6})</td>
<td>2.45</td>
</tr>
</tbody>
</table>

![Consistency Error Diagram](image)
Adaptive Refinement with HHG

- Hanging nodes $\rightarrow$ non-conforming grids

- Red-green $\rightarrow$ conforming grids
Adaptivity in HHG (with conforming meshes)
Refinement with Hanging Nodes in HHG

Coarse grid
Fine Grid
Adaptive Grid with Hanging Nodes
Smoothing operation
Residual computation
Restriction Computation

Treating Hanging Nodes as in FAC: see e.g.
- UR: Mathematical and Computational Techniques for Multilevel Adaptive Methods, SIAM, 1993
Part IV

Multicore Architectures
### Evolution of processors: Improvements

<table>
<thead>
<tr>
<th></th>
<th>Std.-CPU</th>
<th>CBEA</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipelining</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>superscalar execution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>out-of-order</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>wider buses</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SIMD</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>multithreading</td>
<td>✓</td>
<td>✓/✓</td>
<td>✓</td>
</tr>
<tr>
<td>multiprocessing</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>caches</td>
<td>✓</td>
<td>✓/✓</td>
<td>✓</td>
</tr>
<tr>
<td>hardware prefetcher</td>
<td>✓</td>
<td>?/✓</td>
<td>✓</td>
</tr>
<tr>
<td>local storage</td>
<td>✓</td>
<td>✓/✓</td>
<td>✓</td>
</tr>
<tr>
<td>resource virtualization</td>
<td>✓</td>
<td>✓/✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- Green checkmark: Yes
- Red cross: No
- Orange question mark: Unknown
IBM Cell Processor

- Available Cell systems:
  - Blades
  - Playstation 3
  - unfortunately not developed further
The STI Cell Processor

- hybrid multicore processor based on IBM Power architecture
- (simplified) PowerPC core
  - runs operating system
  - controls execution of programs
- multiple co-processors (8, on Sony PS3 only 6 available)
  - operate on fast, private on-chip memory
  - optimized for computation
  - DMA controller copies data from/to main memory
    - multi-buffering can hide main memory latencies completely for streaming-like applications
    - loading local copies has low and known latencies
- memory with multiple channels and links can be exploited if many memory transactions are in-flight
Cell/B.E. and PowerXCell 8i

Diagram of Cell/B.E. and PowerXCell 8i showing interconnections between components such as SPE, PPUs, PPEs, MIC, EIB, BEI, MFC, LS, SXU, and SPU.
Synergistic Processor Unit

- “very small computer of its own”
  - 128 128-bit all-purpose registers
  - operates on 256 kB of Local Store (LS)
- nearly all operations are SIMD only
  - one scalar operation is more expensive than a SIMD
  - only load and store of 16 naturally aligned bytes from/to LS
- 25.6 GFlops (single precision fused-multiply-add)
  - only truncation, fast double precision will be available soon
- no dynamic branch prediction, only hints in software
  - but around 20 cycles branch miss penalty
- no system calls or privileged operations
Memory Flow Controller

- communication interface (to PPE and other SPEs)
  - mailboxes and signal notification
  - memory mapping of Local Store and register file
  - utilized by PPU to upload programs and control SPU

- asynchronous data transfers (DMA)
  - LS <-> main memory, other LSes or devices
  - 16 DMAs in-flight
  - list transfers possible (scatter / gather)
  - only naturally aligned transfers of 1, 2, 4, 8, n·16 bytes
  - usually multiple transfers on multiple MFCs are necessary to saturate main memory bandwidth

- all interaction with SPU through channel interface
Programming the Cell-BE

- the hard way
  - control SPEs using management libraries
  - issue DMAs by language extensions
  - do address calculations manually
  - exchange main memory addresses, array sizes etc.
  - synchronization using mailboxes, signals or libraries

- frameworks
  - Accelerated Library Framework (ALF) and Data, Communication, and Synchronization (DaCS) by IBM
  - Rapidmind SDK

- accelerated libraries

- single-source-compiler
  - IBM’s xlc-cbe-sse is in alpha stage, uses OpenMP
Naive SPU implementation: \( A[] = A[] \times c \)

```c
volatile vector float ls_buffer[8] __attribute__((aligned(128)));

void scale(   unsigned long long gs_buffer, // main memory address of vector
             int number_of_chunks,       // number of chunks of 32 floats
             float factor ) {            // scaling factor
    vector float v_fac = spu_splats(factor); // create SIMD vector with all
                                            // four elements being factor

    for ( int i = 0 ; i < number_of_chunks ; ++i ) {
        mfc_get( ls_buffer , gs_buffer , 128 , 0 ,0,0);  // DMA reading i-th chunk
        mfc_write_tag_mask( 1 << 0 );                    // wait for DMA...
        mfc_read_tag_status_all();                      // ...to complete

        for ( int j = 0 ; j < 8 ; ++j )
            ls_buffer[j] = spu_mul( ls_buffer[j] , v_fac ); // scale local copy using SIMD

        mfc_put( ls_buffer ,gs_buffer , 128 , 0 ,0,0);    // DMA writing i-th chunk
        mfc_write_tag_mask( 1 << 0 );                     // wait for DMA...
        mfc_read_tag_status_all();                       // ...to complete
        gs_buffer += 128;                               // incr. global store pointer
    }
}
```
Remove latencies using multi-buffering

volatile vector float ls_buffer[3][8] __attribute__((aligned(128)));  
...

mfc_get( ls_buffer[0] , gs_buffer , 128 , 0 ,0,0);  // request first chunk
for (int i = 0; i < number_of_chunks; ++i) {
    int cur = ( i ) % 3;  // buffer no. and DMA tag for i-th chunk
    int next = (i+1) % 3;  // " for (i-2)-th and (i+1)-th chunk
    if (i < number_of_chunks-1) {
        mfc_write_tag_mask( 1 << next );  // make sure the (i-2)-th chunk...
        mfc_read_tag_status_all();  // ...has been stored
        mfc_get( ls_buffer[next] , gs_buffer+128 , 128 , next ,0,0);  // request (i+1)-th chunk
    }
    mfc_write_tag_mask( 1 << cur );  // wait until i-th chunk...
    mfc_read_tag_status_all();  // ...is available
    for (int j = 0; j < 8; ++j) ls_buffer[cur][j] = spu_mul(ls_buffer[cur][j],v_fac);
}

mfc_put( ls_buffer[cur] , gs_buffer , 128 , cur ,0,0);  // store i-th chunk
    gs_buffer += 128;
}

mfc_write_tag_mask( 1 | 2 | 4 );  // wait for any...

mfc_read_tag_status_all();  // outstanding DMA
Leaping the memory wall:

cache blocking techniques
Leaping the memory wall:

local storage blocking techniques
Temporal LS-blocking of 3D Stencil Code?

- options for temporal cache blocking in parallel
  - synchronize data accesses at boundaries ❌
  - compute required intermediate results locally ✔
- waste on block-boundaries
  (high surface/volume ratio for small LS)
- alignment and size of DMA and SIMD constraint block size
- low potential (estimate: max. 150% for 2× temporal blocking)

»Not even Markus would like to program that!«
An approach for local storage blocking
Buffered blocking

buffer structures
(local storage)

source grid
(memory)

target grid
(memory)
Buffered blocking

- Can hold one tile at a time.
- 1 stripe per SPE at a time.
Buffered blocking
Buffered blocking
Buffered blocking
Buffered blocking

- can also be used with caches
- supporting (hybrid) framework feasible
Framework architecture

**Framework**
- Thread management: creation, synchronization, affinity
- Data structures: alignment, padding (optional)
- Control traversal of grid: distribution of work, calling of library kernels

**Application Code**
- General configuration: threads and their properties, type of per-thread buffers
- Setup of shared data
- Transfer of control: grid size, constraints, number of sweeps
- Kernels:
  - storage2buffer( tiledesc&,buf& )
  - compute( bufferstack& )
  - buffer2storage( tiledesc&,buf& )
Buffered blocking in action: Multigrid Method for „Complex Diffusion“
Assumption (white additive Gaussian noise):
Relation between an original, unknown image $u : \Omega \subseteq R^d \rightarrow R$ and an observed image $u^0$ can be expressed by

$$ u^0 = u + \eta $$

where $\eta$ stands for the noise.
Denoising by Diffusion

Idea:

Use nonlinear anisotropic diffusion process to denoise the image $u^0$ in the domain $\Omega$, i.e. solve the time-dependent PDE

$$\frac{\partial u}{\partial t} - div(g \nabla u) = 0 \quad \text{in} \ \Omega \times T$$

$$\langle g \nabla u, n \rangle = 0 \quad \text{on} \ \partial \Omega \times T$$

$$u(x,0) = u^0(x) \quad \text{in} \ \Omega$$
Complex Diffusion

- Extend the nonlinear diffusion model by using a complex diffusivity function

\[ g(u) = \frac{e^{i\theta}}{1 + \left(\frac{\text{Im}(u)}{k\theta}\right)^2} \]

- Parameter \( \theta \) denotes a small angle, \( k \) is a soft threshold
- Solution of the anisotropic diffusion PDE becomes complex
- Real part of solution is denoised image, its imaginary part acts as an edge detector
Time-dependent, complex, nonlinear diffusion PDE:

- Spatial discretization by finite volumes
- Semi-implicit time discretization
- Nonlinear diffusion is handled by inexact lagged diffusivity
- Cell-based FAS (full approximation scheme) multigrid
  - Damped Jacobi smoother
  - Standard transfers
Image Denoising Example

Denoising of a test image with added Gaussian noise.

Noisy Image  Denoised Image  Imaginary Part
**Image smoothing by complex diffusion**

- smoothing image by solving a non-linear, complex diffusion equation
- imaginary part works as an edge detector
- full approximation scheme multigrid solver
- simple variant of Schrödinger equation

\[
\begin{align*}
\text{div}(g \nabla u) &= \frac{\partial u}{\partial t} & \text{in } \Omega \times \mathbb{T} \\
\langle g \nabla u, n \rangle &= 0 & \text{on } \partial \Omega \times \mathbb{T} \\
u(x, 0) &= u^0(x) & \text{in } \Omega \\

\end{align*}
\]

\[
g(u) = \frac{e^{i\theta}}{1 + \left(\frac{\text{Im}(u)}{k\theta}\right)^2}
\]
FAS for Complex Diffusion

~400/450 flop/unknown
on each grid level

- compute RHS (except finest level)
- two $\omega$-Jacobi iterations
- compute residual
- restrict current solution and residual

- two $\omega$-Jacobi iterations
- interpolate and apply correction

~400/450 flop/unknown
on each grid level
Framework performance results

V(2,2)-cycle for 4096×4096 image

straight-forward C++ implementation with OpenMP on Core architectures >1s

* Core2 Xeon Penryn @ 2.8 GHz
** Core i7 @ 2.93 GHz
Why GPUs?

- Graphics Processing Units (GPUs) are a massively parallel computer architecture
- GPUs offer high computational performance at low costs
- GPGPU
  - General-Purpose computing on a GPU
  - Using graphics hardware for non-graphic computations
  - Programming Tools: CUDA, Brook+ or OpenCL
Nvidia GeForce GTX 295

- **Costs**: 450 €
- **Interface**: PCI-E 2.0 x16
- **Shader Clock**: 1242 MHz
- **Memory Clock**: 999 MHz
- **Memory Bandwidth**: 2x112 GB/s
- **FLOPS**: 2x894 GFLOPS
- **Max Power Draw**: 289 W
- **Framebuffer**: 2x896 MB
- **Memory Bus**: 2x448 bit
- **Shader Processors**: 2x240
ATI Radeon HD 4870

- **Costs**: 150 €
- **Interface**: PCI-E 2.0 x16
- **Shader Clock**: 750 MHz
- **Memory Clock**: 900 MHz
- **Memory Bandwidth**: 115 GB/s
- **FLOPS**: 1200 GFLOPS
- **Max Power Draw**: 160 W
- **Framebuffer**: 1024 MB
- **Memory Bus**: 256 bit
- **Shader Processors**: 800
Memory Architecture

- Constant Memory
- Shared Memory
- Texture Memory
- Device Memory
Grid size: The size and shape of the data that the program will be working on

Block size: The block size indicates the sub-area of the original grid that will be assigned to a multiprocessor
CPU vs. GPU

- CPUs are great for task parallelism
  - Fast caches
  - Program Logic and Branching

- GPUs are great for data parallelism
  - Multiple ALUs
  - Fast onboard memory
  - High throughput on parallel tasks
    - Executes program on each fragment

- Think of the GPU (device) as a massively-threaded co-processor
Performance of Multigrid for image processing on GPUs

(partly sponsored by Siemens Medical Solutions)
The Multigrid Idea

- iterative solvers only
  remove local components of the error fast
- instead:
  - smooth
    remove local features of error
  - restrict
c    coarsen error
  - compute correction of error on coarse grid
  - prolongation
    interpolate and apply correction
  - smooth again
Parallel Smoothers I

Red-black Gauß-Seidel: Divide domain into black and red points

2D:

3D:
Interpolation and Restriction

- Linear interpolation
- Full weighting restriction

In case of red-black splitting:
- Linear interpolation (black only)
- Half weighting restriction (results in injection)
Full Multigrid Cycle

- **h**
- **2h**
  - Interpolation of solution \( u \)
- **4h**
  - Exact solution

- **\( r = f - Au \)**

- **Smoothing**

- **\( \mu_{2h} \)**

- **V-cycle**
  - Interpolation of error and correction of solution \( u \)
Interpolation and Restriction

![Diagram showing interpolation and restriction with grid and arrows indicating flow.]
Multigrid on Nvidia GeForce GTX 295

Image size

Runtime V(2,2) in ms

- 1024x1024
- 1024x2048
- 2048x2048
- 2048x4096
- 4096x4096
Multigrid on ATI Radeon HD 4870

![Bar Chart]

- Runtime V(2,2) in ms
- Image size
- 1024x1024
- 2048x2048
- 4096x4096

Harald Köstler (harald.koestler@informatik.uni-erlangen.de)
Runtime Comparison

- Nvidia GeForce GTX 295
  - Half of the GPU
  - Memory bandwidth 112 GB/s
  - Runtime 41.5 ms (4096 x 4096)

- ATI Radeon HD 4870
  - Memory bandwidth 115 GB/s
  - Runtime 40.4 ms (4096 x 4096)

→ Both cards show very similar performance
Red-black Splitting

- Store red and black values in two different arrays
- Doubles the performance
Multigrid on GTX 295 with red-black splitting

![Runtime V(2,2) in ms vs Image size](image)
Memory Bandwidth

In percent from maximum measured (rounded) streaming bandwidth (100 GB / s)
Runtime Distribution for GPU Kernels

- RBGS: 52%
- memcpy: 29%
- 2ndDerivative: 3%
- Gradients: 3%
- Interpolate_co: 6%
- Residual_Restrict: 7%
- 2ndDerivative: 3%
- Gradients: 3%
- Interpolate_co: 6%
- Residual_Restrict: 7%
Frames per second for Image Stitching

CPU: Intel Core2 Quad Q9550@2.83GHz with OpenMP (4 cores)
Conclusions and Outlook
Acknowledgements

**Collaborators**
- In Erlangen: WTM, LSE, LSTM, LGDV, RRZE, LME, Neurozentrum, Radiologie, Appl. Mathematics, Theoretical Physics, etc.
- Especially for foams: C. Körner (WTM)
- International: Utah, Technion, Constanta, Ghent, Boulder, München, Zürich, Delhi, ...

**Dissertationen Projects**
- N. Thürey, T. Pohl, S. Donath, S. Bogner (LBM, free surfaces, 2-phase flows)
- M. Kowarschik, J. Treibig, M. Stürmer, J. Habich (architecture aware algorithms)
- K. Iglberger, T. Preclik, K. Pickel (rigid body dynamics)
- J. Götz, C. Feichtinger (Massively parallel LBM software, suspensions)
- C. Mihoubi, D. Bartuschat (Complex geometries, parallel LBM)

**(Long Term) Guests in summer/fall 2009/10:**
- Dr. S. Ganguly, IIT Kharagpur (Humboldt) - Electroosmotic Flows
- Prof. V. Buwa, IIT Delhi (Humboldt) - Gas-Fluid-Solid flows
- Felipe Aristizabal, McGill Univ., Canada (LBM with Brownian Motion)
- Prof. Popa, Constanta, Romania (DAAD) Numerical Linear Algebra
- Prof. N. Zakaria, Universiti Petronas, Malaysia
- Prof. Hanke, KTH Stockholm (DAAD), Mathematical Modelling

- ~25 Diplom- /Master- Thesis, ~30 Bachelor Thesis
- Funding by KONWIHR, DFG, BMBF, EU, Elitenetzwerk Bayern
Next Week:
Lattice Boltzmann Method
Flow Simulation
Free Surface Flows
Multibody Dynamics
Granular Flows
Particule Ladden Flows
Thanks for your attention!

Questions?

Slides, reports, thesis, animations available for download at: www10.informatik.uni-erlangen.de