Towards an Integrated Performance Oriented Co-Design Methodology

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

Lehrstuhl für Informatik 10 (Systemsimulation)

www10.informatik.uni-erlangen.de

Dagstuhl Seminar
Connecting Performance Analysis and Visualization to Advance Extreme Scale Computing

January 2014
Overview

- Motivation
- Two HPC Software Projects (app proxies)
  - HHG: A multigrid finite element package
    - ISC Award 2007
  - waLBerla: Complex flows with Lattice Boltzmann Methods
    - Prace Award 2010
- What do we learn about performance engineering?
- Conclusions
## Hardware

<table>
<thead>
<tr>
<th>JUQUEEN</th>
<th>SuperMUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forschungszentrum Jülich, Germany</td>
<td>LRZ, Garching (Munich), Germany</td>
</tr>
<tr>
<td>IBM system</td>
<td>IBM system</td>
</tr>
<tr>
<td>Blue Gene/Q</td>
<td>Intel Sandy Bridge-EP</td>
</tr>
<tr>
<td>28,672 nodes</td>
<td>9,216 nodes</td>
</tr>
<tr>
<td>458,752 cores</td>
<td>147,456 cores</td>
</tr>
<tr>
<td>5.9 Petaflops peak</td>
<td>3.2 Petaflops peak</td>
</tr>
<tr>
<td>448 TB main memory</td>
<td>288 TB main memory</td>
</tr>
<tr>
<td>5D Torus Network</td>
<td>Non-blocking tree / 4:1 pruned tree</td>
</tr>
</tbody>
</table>
First Insight:

**Extreme-Scale Performance can be Easy!**

- If parallel efficiency is bad, choose a slower serial algorithm
  - it is probably easier to parallelize
  - and will make your speedups look much more impressive
- Introduce the “CrunchMe” variable for getting high Flops rates
  - advanced method: disguise CrunchMe by using an inefficient (but compute-intensive!) algorithm from the start
- Introduce the “HitMe” variable to get good cache hit rates
  - advanced version: Implement HitMe in the “Hash-Brown Lookaside Table for the Multi-Threatened Cash-Filling Clouded Tree” data structure
  - ... impressing yourself and others
- Never cite “time-to-solution”
  - who cares whether you solve a real-life problem anyway
  - it is the MachoFlops that interest the people who pay for your research
- Never waste your time by trying to use a complicated algorithm in parallel
  - Use Primitive Algorithm => Easy to Maximize your MachoFlops
  - A few million CPU hours can easily save you days of reading in boring math books
Performance Oriented Algorithm Design (1)

- **predict** a-priori what performance is achievable:
  - white box performance model
- **evaluate** concrete algorithm/implemention against prediction
- **account** for discrepancies
  - **refine** assumptions
  - **improve** performance model
  - **improve** implementation
  - **improve** solver, discretization, model
- With other words:

  **performance driven co-design!**
Performance Oriented Algorithm Design (2)

Example: Multigrid solves 2D Poisson in < 30 N operations \((O(h^2))\) 5-pt stencil

- consequently, we should solve
  - on a 100 Gflops CPU: a 1000x1000 grid in 0.0003 seconds
- If not, we must question
  - the discretization
  - the solver
  - the implementation
  - the performance model
  - ...
- There are good reasons to be slower, but we must use scientific methodology to understand why and how
- When we burn 20 MW and miss the best published result by 4 orders of magnitude then it is not enough to state "our code is not fully optimized yet"
- Where are the math papers publishing concrete complexity constants for other problems (Poisson 3D, Stokes, Lame, ...)?
- New theory must relate algorithms better to hardware performance characteristics (such as mem bandwidth, peak flops, energy consumption, ...)
Two Examples of Large Scale Multi-Scale and Multi-Physics Simulation
Multigrid for structured refinement of an unstructured tetrahedral mesh

Use regular HHG patches for partitioning the domain

Coarse grid with 132k elements, as assigned to supercomputer

communication of ghost layers
TERRA-NEO
Co-Design of an Exascale Earth Mantle Modeling Framework

DFG SPP 1648/1 - Software for Exascale Computing
A uniform discretization of the mantle at for instance 1 km resolution would result in meshes with nearly a trillion elements, which is far beyond the capacity of the largest available supercomputers.

Multigrid Scalability on JuQueen for Stokes

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Threads</th>
<th>Grid points</th>
<th>Resolution</th>
<th>Time: (A)</th>
<th>(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>$2.1 \cdot 10^7$</td>
<td>32 km</td>
<td>30 s</td>
<td>89 s</td>
</tr>
<tr>
<td>4</td>
<td>240</td>
<td>$1.6 \cdot 10^8$</td>
<td>16 km</td>
<td>38 s</td>
<td>114 s</td>
</tr>
<tr>
<td>30</td>
<td>1,920</td>
<td>$1.3 \cdot 10^9$</td>
<td>8 km</td>
<td>40 s</td>
<td>121 s</td>
</tr>
<tr>
<td>240</td>
<td>15,360</td>
<td>$1.1 \cdot 10^{10}$</td>
<td>4 km</td>
<td>44 s</td>
<td>133 s</td>
</tr>
<tr>
<td>1,920</td>
<td>122,880</td>
<td>$8.5 \cdot 10^{10}$</td>
<td>2 km</td>
<td>48 s</td>
<td>153 s</td>
</tr>
<tr>
<td>15,360</td>
<td>983,040</td>
<td>$6.9 \cdot 10^{11}$</td>
<td>1 km</td>
<td>54 s</td>
<td>170 s</td>
</tr>
</tbody>
</table>

Coupled with Energy Equation: Buoyancy Driven Flow

Largest run: fully implicit - coupled
4.1 trillion tetrahedra
3.4 trillion DOF
Analysing Efficiency: RB-GS Smoother

for (int i=1; i < (tsize-j-k-1); i=i+2) {
}

This loop should be executed on each SuperMuc core at
- 720 M updates/sec (in theory - peak performance)
- 176 M updates/sec (in practice - memory access bottleneck; RB-ordering prohibits vector loads)

Thus whole SuperMuc should perform
- 147456*176M ≈ 26T elementary updates/sec
Linking: algorithmic knowledge with performance analysis

- we know that MG as scalable algorithm should solve FE system with cost equivalent to 10 RB-GS updates:
  - *textbook multigrid efficiency*
- we can perform 26T RB-GS updates per sec
- Thus we should be able to solve >1T unknowns per sec
- HHG still fails to reach this by approx a factor 10 but beats many other implementations by 2-3 orders of magnitude
  - communication overhead
  - node efficiency loss on coarser grids
  - still unused algorithmic optimization potential
Pessimizing the Performance

with greetings from D. Bailey's: *Twelve Ways to Fool the Masses*

- Bring loops in wrong order, ignore caches, randomize memory access, use many small MPI messages
  - $10^{12} \rightarrow 10^{11}$ unknowns
- Do not use a matrix-free implementation (keep in mind that a single multiplication with the mass and stiffness matrix can easily cost 50 mem accesses per unknown):
  - $10^{11} \rightarrow 10^{10}$ unknowns
- Gain additional flexibility by using unoptimized unstructured grids (indirect mem access costs!)
  - $10^{10} \rightarrow 10^{9}$ unknowns
- Increase algorithmic overhead, e.g. permanently checking convergence, use an expensive error estimator, etc. etc.
  - $10^{9} \rightarrow 10^{8}$ unknowns ( ... still a large system ... )
waLBerla – an HPC Framework for Complex Flows

- Focus on lattice Boltzmann method
- written in C++
- Hybridly parallelized (MPI + OpenMP)
- manually (painstakingly) machine-specific LBM kernels for max performance
- generic, easily adaptable kernels for experiments
- all data structures exa-scalable
  - from desktop to multi-petascale machines (and beyond)
- portable (Compiler/OS)
- is being set in public domain 1/2014
Simulation of Metal Foams

- Example application:
  - Engineering: metal foam simulations

- Based on LBM:
  - Free surfaces
  - Surface tension
  - Disjoining pressure to stabilize thin liquid films
  - Parallelization with MPI and load balancing

- Collaboration with C. Körner (Dept. of Material Sciences, Erlangen)

- Other applications:
  - Food processing
  - Fuel cells
SuperMUC - LDC - Weak

- SuperMUC – single socket

already quite optimized!
SuperMUC - LDC - Weak

- SuperMUC – single socket

⇒ limited by memory bandwidth

already quite optimized!
• JUQUEEN – single node

![Graph showing MLUP/s vs. number of cores with a trend line limited by memory bandwidth.](image)

- Hybrid version (4 threads per core)
- Limited by memory bandwidth
SuperMUC - LDC - Weak

- SuperMUC – TRT kernel

![Graph showing performance of SuperMUC with different configurations.]

- #processes per node
  - 16P 1T
  - 4P 4T
  - 2P 8T

- #threads per process

2 islands
• JUQUEEN – TRT kernel

1.93 x 10^{12} cells updated per second
(19 values per cell)

⇒ 383 TFlop/s (6.5% peak)
⇒ 800 TB/s (67% peak)
What do we learn?

- Optimize for
  - Core -> Node -> System ... in that order and re-iterate
- Algorithm specific base metric (LUP/s)
  - need link from algorithm to the architecture
- Algorithmic (theory) knowledge necessary to link LUP/s to application cost (solver)
- Identify limiting resource
  - flops vs. bandwidth vs. instr. throughput vs. ...
- The most attractive performance potential is only accessible by optimization across the whole CSE pipeline
  - app -> model -> descr -> data struc -> solver -> parallel impl.
- Tools are essential in the multilevel feedback loop
  - hypothesis -> benchmark -> analyse -> optimize
    - on core -> node -> system
Thank you for your attention!

Video generated at LSS with the massively parallel waLberla Software framework for Lattice Boltzmann based multi-physics applications.

Questions?

Slides, reports, thesis, animations available for download at:

www10.informatik.uni-erlangen.de