Algorithms and Software in the Post-Petascale Era

William Gropp

www.cs.illinois.edu/~wgropp
Extrapolation is Risky

• 1989 – T – 23 years
  ♦ Intel introduces 486DX
  ♦ Eugene Brooks writes “Attack of the Killer Micros”
  ♦ 4 years before TOP500
  ♦ Top systems at about 2 GF Peak

• 1999 – T – 13 years
  ♦ NVIDIA introduces its GPU (GeForce 256)
    • Programming GPUs still a challenge 13 years later
  ♦ Top system – ASCI Red, 9632 cores, 3.2 TF Peak (about 3 GPUs in 2012)
  ♦ MPI is 7 years old
HPC Today

• High(est)-End systems
  ♦ 1 PF ($10^{15}$ Ops/s) achieved on a few “peak friendly” applications
  ♦ Much worry about scalability, how we’re going to get to an ExaFLOPS
  ♦ Systems are all oversubscribed
    • DOE INCITE awarded almost 900M processor hours in 2009; 1600M-1700M hours in 2010-2012; (big jump planned in 2013 – over 5B hours)
    • NSF PRAC awards for Blue Waters similarly competitive

• Widespread use of clusters, many with accelerators; cloud computing services
  ♦ These are transforming the low and midrange

• Laptops (far) more powerful than the supercomputers I used as a graduate student
HPC in 2012

- **Sustained PF systems**
  - K Computer (Fujitsu) at RIKEN, Kobe, Japan (2011)
  - “Sequoia” Blue Gene/Q at LLNL
  - NSF Track 1 “Blue Waters” at Illinois
  - Undoubtedly others (China, ... )

- Still programmed with MPI and MPI+other (e.g., MPI +OpenMP or MPI+OpenCL/CUDA or MPI+OpenACC)
  - But in many cases using toolkits, libraries, and other approaches
    - And not so bad – applications will be able to run when the system is turned on
  - Replacing MPI will require some compromise – e.g., domain specific (higher-level but less general)
    - Lots of evidence that fully automatic solutions won’t work
End of an Era

• IN THE LONG TERM (~2017 THROUGH 2024)  
  “While power consumption is an urgent challenge, its leakage or static component will become a major industry crisis in the long term, threatening the survival of CMOS technology itself, just as bipolar technology was threatened and eventually disposed of decades ago.”  [ITRS 2009]

• Unlike the situation at the end of the bipolar era, no technology (i.e., CMOS) is waiting in the wings.
The Post-Moore Era

- **Scaling is ending**
  - Voltage scaling ended in 2004 (leakage current)
  - Feature scaling will end in 202x (not enough atoms)
  - Scaling rate will slow down in the next few years
  - Continued scaling in the next decade will need a sequence of (small) miracles (new materials, new structures, new manufacturing technologies)

**Compute Efficiency becomes a paramount concern**
- More computations per joule
- More computations per transistor
HPC in 2020-2023

- Exascale systems are likely to have
  - Extreme power constraints, leading to
    - Clock Rates similar to today’s systems
    - A wide-diversity of simple computing elements (simple for hardware but complex for software)
    - Memory per core and per FLOP will be much smaller
    - Moving data anywhere will be expensive (time and power)
  - Faults that will need to be detected and managed
    - Some detection may be the job of the programmer, as hardware detection takes power
  - Extreme scalability and performance irregularity
    - Performance will require enormous concurrency
    - Performance is likely to be variable
      - Simple, static decompositions will not scale
  - A need for latency tolerant algorithms and programming
    - Memory, processors will be 100s to 10000s of cycles away. Waiting for operations to complete will cripple performance
Algorithms and Applications Will Change

- Applications need to become more dynamic, more integrated
- System software must work with application:
  - Code complexity (Autotuning)
  - Dynamic resources (no simple PGAS)
  - Latency hiding (Nonblocking algorithms, interfaces (including futures))
  - Resource sharing (more performance information, performance asserts, runtime coordination)
How Do We Make Effective Use of These Systems?

• Better use of our existing systems
  ♦ Blue Waters will provide a sustained PF, but that typically requires ~10PF peak

• Improve node performance
  ♦ Make the compiler better
  ♦ Give better code to the compiler
  ♦ Get realistic with algorithms/data structures

• Improve parallel performance/scalability

• Improve productivity of applications
  ♦ Better tools and interoperable languages, not a (single) new programming language

• Improve algorithms
  ♦ Optimize for the real issues – data movement, power, resilience, ...
Make the Compiler Better

• It remains the case that most compilers cannot compete with hand-tuned or autotuned code on simple code
  ♦ Just look at dense matrix-matrix multiplication or matrix transpose
  ♦ Try it yourself!
    • Matrix multiply on my laptop:
      • N=100 (in cache): 1818 MF (1.1ms)
      • N=1000 (not): 335 MF (6s)
How Good are Compilers at Vectorizing Codes?

Media Bench II Applications

<table>
<thead>
<tr>
<th>Appl</th>
<th>XLC</th>
<th>ICC</th>
<th>GCC</th>
<th>XLC</th>
<th>ICC</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPEG Enc</td>
<td>-</td>
<td>1.33</td>
<td>-</td>
<td>1.39</td>
<td>2.13</td>
<td>1.57</td>
</tr>
<tr>
<td>JPEG Dec</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.14</td>
<td>1.13</td>
</tr>
<tr>
<td>H263 Enc</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.25</td>
<td>2.28</td>
<td>2.06</td>
</tr>
<tr>
<td>H263 Dec</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.31</td>
<td>1.45</td>
<td>-</td>
</tr>
<tr>
<td>MPEG2 Enc</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.06</td>
<td>1.96</td>
<td>2.43</td>
</tr>
<tr>
<td>MPEG2 Dec</td>
<td>-</td>
<td>-</td>
<td>1.15</td>
<td>1.37</td>
<td>1.45</td>
<td>1.55</td>
</tr>
<tr>
<td>MPEG4 Enc</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.44</td>
<td>1.81</td>
<td>1.74</td>
</tr>
<tr>
<td>MPEG4 Dec</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.12</td>
<td>-</td>
<td>1.18</td>
</tr>
</tbody>
</table>

Table shows **whole program speedups** measured against unvectorized application.

**How Do We Change This?**

- Test compiler against “equivalent” code (e.g., best hand-tuned or autotuned code that performs the same computation, under some interpretation or “same”)
  - In a perfect world, the compiler would provide the same, excellent performance for all equivalent versions
- As part of the Blue Waters project, Padua, Garzaran, Maleki have developed a test suite that evaluates how the compiler does with such equivalent code
  - Working with vendors to improve the compiler
  - Identify necessary transformations
  - Identify opportunities for better interaction with the programmer to facilitate manual intervention.
  - Main focus has been on code generation for vector extensions
  - Result is a compiler whose realized performance is less sensitive to different expression of code and therefore closer to that of the best hand-tuned code.
    - Just by improving automatic vectorization, loop speedups of more than 5 have been observed on the Power 7.
- But this is a long-term project
  - What can we do in the meantime?
Give “Better” Code to the Compiler

• Augmenting current programming models and languages to exploit advanced techniques for performance optimization (i.e., autotuning)

• Not a new idea, and some tools already do this.

• But how can these approaches become part of the mainstream development?
How Can Autotuning Tools Fit Into Application Development?

• In the short run, just need effective mechanisms to replace user code with tuned code
  ♦ Manual extraction of code, specification of specific collections of code transformations
• But this produces at least two versions of the code (tuned (for a particular architecture and problem choice) and untuned). And there are other issues.
• What does an application want (what is the Dream)?
Application Needs Include

- Code must be portable
- Code must be persistent
- Code must permit (and encourage) experimentation
- Code must be maintainable
- Code must be correct
- Code must be faster
Implications of These Requirements

- **Portable** - augment existing language. Either use pragmas/comments or extremely portable precompiler
  - Best if the tool that performs all of these steps looks like just like the compiler, for integration with build process
- **Persistent**
  - Keep original and transformed code around: *Golden Copy*
- **Maintainable**
  - Let user work with original code *and* ensure changes automatically update tuned code
- **Correct**
  - Do whatever the application developer needs to believe that the tuned code is correct
    - In the end, this *will* require running some comparison tests
- **Faster**
  - Must be able to interchange tuning tools - pick the best tool for *each* part of the code
  - No captive interfaces
  - Extensibility - a clean way to add new tools, transformations, properties, ...
Application-Relevant Abstractions

- Language for interfacing with autotuning must convey concepts that are meaningful to the application programmer
- Wrong: unroll by 5
  - Though could be ok for performance expert, and some compilers already provide pragmas for specific transformations
- Right (maybe): Performance precious, typical loop count between 100 and 10000, even, not power of 2
- Middle ground: Apply unroll, align, SIMD transformations and tune
- We need work at developing higher-level, performance-oriented languages or language extensions
  - This would be the “good” future
  - Early steps include TCE, Orio, Spiral, ...
Better Algorithms and Data Structures

• Autotuning only offers the best performance with the given data structure and algorithm
  ♦ That’s a big constraint

• Processors include hardware to address performance challenges
  ♦ “Vector” function units
  ♦ Memory latency hiding/prefetch
  ♦ Atomic update features for shared memory
  ♦ Etc.
Sparse Matrix-Vector Multiply

Barriers to faster code

- “Standard” formats such as CSR do not meet requirements for prefetch or vectorization
- Modest changes to data structure enable both vectorization, prefetch, for 20-80% improvement on P7

Prefetch results in Optimizing Sparse Data Structures for Matrix Vector Multiply http://hpc.sagepub.com/content/25/1/115
What Does This Mean For You?

• It is time to rethink data structures and algorithms to match the realities of memory architecture
  ♦ We have results for x86 where the benefit is smaller but still significant
  ♦ Better match of algorithms to prefetch hardware is necessary to overcome memory performance barriers

• Similar issues come up with heterogeneous processing elements (someone needs to design for memory motion and concurrent and nonblocking data motion)
Is It Communication Avoiding Or Minimum Solution Time?

- **Example:** non minimum collective algorithms
- **Work of Paul Sack:** see “Faster topology-aware collective algorithms through non-minimal communication”, PPoPP 2012
- **Lesson:** minimum communication need not be optimal
Allgather
Problem: Recursive-doubling

- No congestion model:
  \[ T = (\lg P)\alpha + n(P-1)\beta \]

- Congestion on torus:
  \[ T \approx (\lg P)\alpha + \frac{5}{24}nP^{4/3}\beta \]

- Congestion on Clos network:
  \[ T \approx (\lg P)\alpha + \frac{nP}{\mu}\beta \]

- Solution approach: move smallest amounts of data the longest distance
Allgather: recursive halving
Allgather: recursive halving
Allgather: recursive halving
Allgather: recursive halving
Allgather: recursive halving

\[ T = (\lg P) \alpha + (7/6)nP\beta \]
New Problem: Data Misordered

• Solution: shuffle input data
  ♦ Could shuffle at end (redundant work; all processes shuffle)
  ♦ Could use non-contiguous data moves
  ♦ Shuffle data on network...
Solution: Input shuffle
Solution: Input shuffle
Solution: Input shuffle

a → e → b → f

i → m → j → n

c → g → d → h

k → o → l → p
Solution: Input shuffle

Diagram showing the shuffle process with arrows connecting pairs of letters in a specific pattern.
Solution: Input shuffle

abcd  efg
ijkl  mnop

abcd  efg
ijkl  mnop

abcd  efg
ijkl  mnop

abcd  efg
ijkl  mnop
Solution: Input shuffle

\[ T = (1 + \log P) \alpha + \frac{7}{6} nP \beta \]

\[ T \approx (\log P) \alpha + \frac{7}{6} nP \beta \]
Evaluation: Intrepid BlueGene/P at ANL

- 40k-node system
  - Each is 4 x 850 MHz PowerPC 450
- 512+ nodes is 3d torus; fewer is 3d mesh
- xlc -O4
- 375 MB/s delivered per link
  - 7% penalty using all 6 links both ways
Allgather performance
Notes on Allgather

- Bucket algorithm (not described here) exploits multiple communication engines on BG
- *Analysis shows performance near optimal*
- Alternative to reorder data step is in memory move; analysis shows similar performance and measurements show reorder step faster on tested systems
Performance on a Node

• Nodes are SMPs
  ♦ You have this problem on anything (even laptops)

• Tuning issues include the usual
  ♦ Getting good performance out of the compiler (often means adapting to the memory hierarchy)

• New (SMP) issues include
  ♦ Sharing the SMP with other processes
  ♦ Sharing the memory system
New (?) Wrinkle – Avoiding Jitter

• Jitter here means the variation in time measured when running identical computations
  ♦ Caused by other computations, e.g., an OS interrupt to handle a network event or runtime library servicing a communication or I/O request

• This problem is in some ways less serious on HPC platform, as the OS and runtime services are tuned to minimize impact
  ♦ However, cannot be eliminated entirely
Sharing an SMP

- Having many cores available makes everyone think that they can use them to solve other problems ("no one would use all of them all of the time")
- However, compute-bound scientific calculations are often written as if all compute resources are owned by the application
- Such static scheduling leads to performance loss
- Pure dynamic scheduling adds overhead, but is better
- Careful mixed strategies are even better
- Thanks to Vivek Kale
Happy Medium Scheduling

Performance irregularities introduce load-imbalance. Pure dynamic has significant overhead; pure static too much imbalance. Solution: combined static and dynamic scheduling

Communication Avoiding LU factorization (CALU) algorithm, S. Donfack, L. Grigori, V. Kale, WG, IPDPS ‘12

Scary Consequence: Static data decompositions will not work at scale.

Corollary: programming models with static task models will not work at scale
Synchronization and OS Noise

- “Characterizing the Influence of System Noise on Large-Scale Applications by Simulation,” Torsten Hoefler, Timo Schneider, Andrew Lumsdaine
  - Best Paper, SC10
- Next 3 slides based on this talk...
A Noisy Example – Dissemination Barrier

• Process 4 is delayed
  ♦ Noise propagates “wildly” (of course deterministic)
Single Collective Operations and Noise

- 1 Byte, Dissemination, regular noise, 1000 Hz, 100 $\mu$s
The problem is *blocking* operations

- Simple, data-parallel algorithms easy to reason about but inefficient
  - True for decades, but ignored (memory)
- One solution: fully asynchronous methods
  - Very attractive, yet efficiency is low and there are good reasons for that
  - Blocking can be due to fully collective (e.g., Allreduce) or neighbor communications (halo exchange)
  - Can we save methods that involve global, synchronizing operations?
Saving Allreduce

- One common suggestion is to avoid using Allreduce
  - But algorithms with dot products are among the best known
  - Can sometimes aggregate the data to reduce the number of separate Allreduce operations
  - But better is to reduce the impact of the synchronization by hiding the Allreduce behind other operations (in MPI, using MPI_Iallreduce)
- We can adapt CG to nonblocking Allreduce with some added floating point (but perhaps little time cost)
The Conjugate Gradient Algorithm

- While (not converged)
  niter += 1;
  s = A * p;
  t = p' * s;
  alpha = gmma / t;
  x = x + alpha * p;
  r = r - alpha * s;
  if rnorm2 < tol2 ; break ; end
  z = M * r;
  gmmaNew = r' * z;
  beta = gmmaNew / gmma;
  gmma = gmmaNew;
  p = z + beta * p;
end
The Conjugate Gradient Algorithm

- While (not converged)
  - niters += 1;
  - s = A * p;
  - t = p' * s;
  - alpha = gamma / t;
  - x = x + alpha * p;
  - r = r - alpha * s;
  - if rnorm2 < tol2 ; break ; end
  - z = M * r;
  - gammaNew = r' * z;
  - beta = gammaNew / gamma;
  - gamma = gammaNew;
  - gammaNew;
CG Reconsidered

- By reordering operations, nonblocking dot products (MPI_Iallreduce in MPI-3) can be overlapped with other operations
- Trades extra local work for overlapped communication
  - On a pure floating point basis, the nonblocking version requires 2 more DAXPY operations
  - A closer analysis shows that some operations can be merged
- More work does not imply more time
What’s Different at Peta/Exascale

• Performance Focus
  ♦ Only a little – basically, the resource is expensive, so a premium placed on making good use of resource
  ♦ Quite a bit – node is more complex, has more features that must be exploited

• Scalability
  ♦ Solutions that work at 100-1000 way often inefficient at 100,000-way
  ♦ Some algorithms scale well
    • Explicit time marching in 3D
  ♦ Some don’t
    • Direct implicit methods
  ♦ Some scale well for a while
    • FFTs (communication volume in Alltoall)
  ♦ Load balance, latency are critical issues

• Fault Tolerance becoming important
  ♦ Now: Reduce time spent in checkpoints
  ♦ Soon: Lightweight recovery from transient errors
Preparing for the Next Generation of HPC Systems

• Better use of existing resources
  ♦ Performance-oriented programming
  ♦ Dynamic management of resources at all levels
  ♦ Embrace hybrid programming models (you have already if you use SSE/VSX/OpenMP/…)

• Focus on results
  ♦ Adapt to available network bandwidth and latency
  ♦ Exploit I/O capability (available space grew faster than processor performance!)

• Prepare for the future
  ♦ Fault tolerance
  ♦ Hybrid processor architectures
  ♦ Latency tolerant algorithms
  ♦ Data-driven systems
Recommended Reading

• Bit reversal on uniprocessors (Alan Karp, SIAM Review, 1996)


• Experimental Analysis of Algorithms (Catherine McGeoch, Notices of the American Mathematical Society, March 2001)

• Reflections on the Memory Wall (Sally McKee, ACM Conference on Computing Frontiers, 2004)
Thanks

- Torsten Hoefler
  - Performance modeling lead, Blue Waters; MPI datatype
- David Padua, Maria Garzaran, Saeed Maleki
  - Compiler vectorization
- Dahai Guo
  - Streamed format exploiting prefetch, vectorization, GPU
- Vivek Kale
  - SMP work partitioning
- Hormozd Gahvari
  - AMG application modeling
- Marc Snir and William Kramer
  - Performance model advocates
- Abhinav B hatele
  - Process/node mapping
- Elena Caraba
  - Nonblocking Allreduce in CG
- Van Bui
  - Performance model-based evaluation of programming models
- Funding provided by:
  - Blue Waters project (State of Illinois and the University of Illinois)
  - Department of Energy, Office of Science
  - National Science Foundation