The Challenges of Exascale

William D Gropp
www.cs.illinois.edu/~wgropp

Extrapolation is Risky

- 1989 – T – 22 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 – 12 years

  - NVIDIA introduces the GPU (GeForce 256)
    - Programming GPUs still a challenge
  - Top system – ASCI Red, 9632 cores, 3.2 TF Peak
  - 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, many turned away; almost 1.7B in 2011
    - 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 2011

- <u>Sustained</u> PF systems
  - ~~NSF Track 1 "Blue Waters" at Illinois~~ **2012**
  - ~~"Sequoia" Blue Gene/Q at LLNL~~ **2012**
  - K Computer (Japan) , China?, …)
- Still programmed with MPI and MPI+other (e.g., MPI+OpenMP)
  - 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)
    - Still can't compile single-threaded code to reliably get good performance – see the work in autotuners. Lesson – there's a limit to what can be automated. Pretending that there's an automatic solution will stand in the way of a real solution

Blue Waters: A Sustained Petascale System

- Cray XE/XK system
- > 235 Cabinets XE (2 AMD CPU/node)
- >30 Cabinets XK (1 AMD CPU/1 NVIDEA GPU/node)
- >1.5 PB memory
- >25 PB disk
- Upto 500 PB tape storage
- Able to sustain > 1PF on a range of applications (not just dense matrix-matrix multiply)

HPC in 2018-2020

- Exascale (10^{18}) systems arrive
  - Issues include power, concurrency, fault resilience, memory capacity
- Likely features
  - Memory per core (or functional unit) smaller than today's systems
  - 10^8-10^9 threads
  - Heterogeneous processing elements
- Software *will* be different
  - You *can* use MPI, but constraints will get in your way
  - Likely a combination of tools, with domain-specific solutions and some automated code generation
  - New languages possible but not certain
- Algorithms need to change/evolve
  - Extreme scalability, reduced memory
  - Managed locality
  - Participate in fault tolerance

ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems

Peter Kogge, Editor & Study Lead
Keren Bergman
Shekhar Borkar
Dan Campbell
William Carlson
William Dally
Monty Denneau
Paul Franzon
William Harrod
Kerry Hill
Jon Hiller
Sherman Karp
Stephen Keckler
Dean Klein
Robert Lucas
Mark Richards
Al Scarpelli
Steven Scott
Allan Snavely
Thomas Sterling
R. Stanley Williams
Katherine Yelick

September 28, 2008

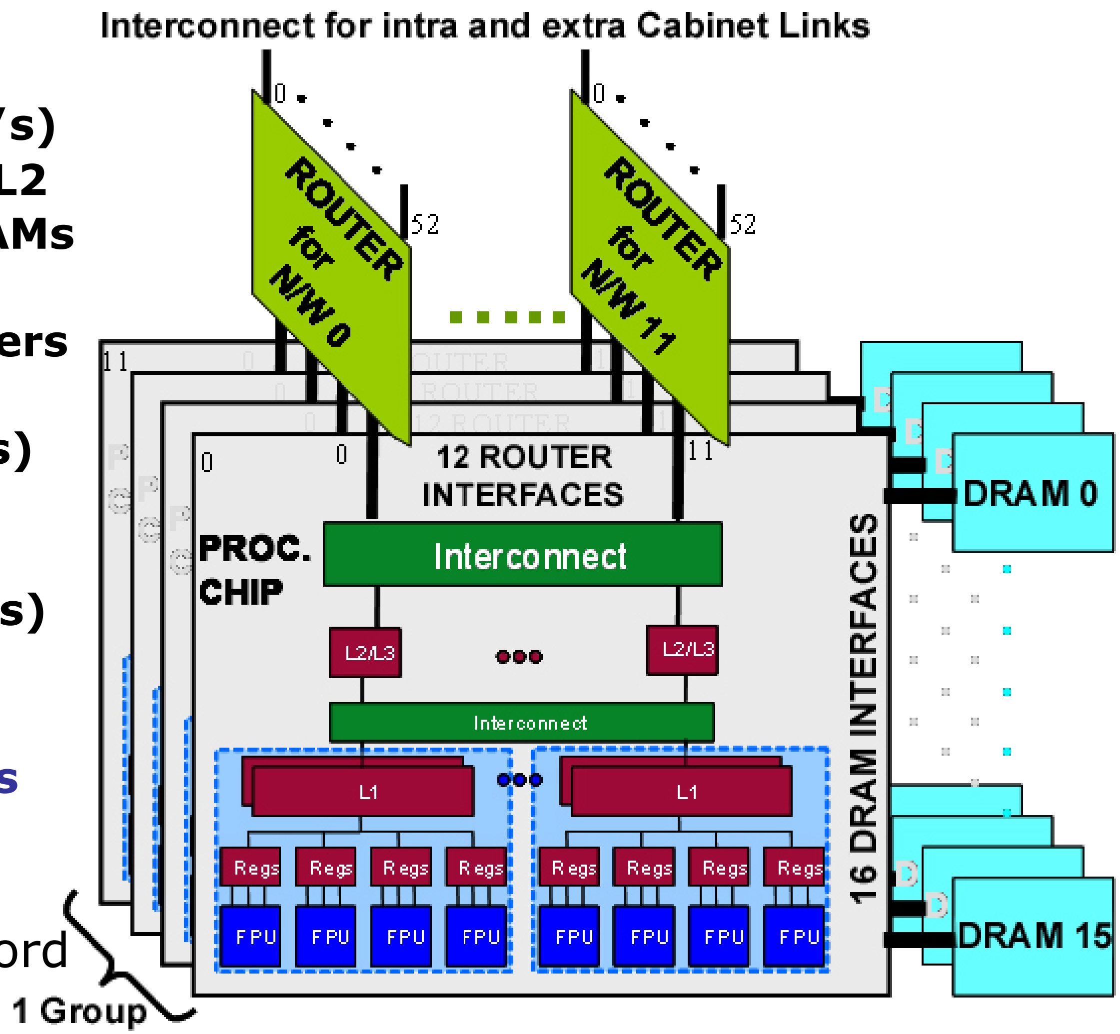
This work was sponsored by DARPA IPTO in the ExaScale Computing Study with Dr. William Harrod as Program Manager; AFRL contract number **FA8650-07-C-7724.** This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings

1 EFlop/s "Clean Sheet of Paper" Strawman

Sizing done by "balancing" power budgets with achievable capabilities

- **4 FPUs+RegFiles/Core (=6 GF @1.5GHz)**
- **1 Chip = *742* Cores (=4.5TF/s)**
  - **213MB of L1I&D; 93MB of L2**
- **1 Node = 1 Proc Chip + 16 DRAMs (16GB)**
- **1 Group = 12 Nodes + 12 Routers (=54TF/s)**
- **1 Rack = 32 Groups (=1.7 PF/s)**
  - **384 nodes / rack**
- **3.6EB of Disk Storage included**
- **1 System = 583 Racks (=1 EF/s)**
  - **166 MILLION cores**
  - **680 MILLION FPUs**
  - **3.6PB = 0.0036 bytes/flops**
  - **68 MW w'aggressive assumptions**

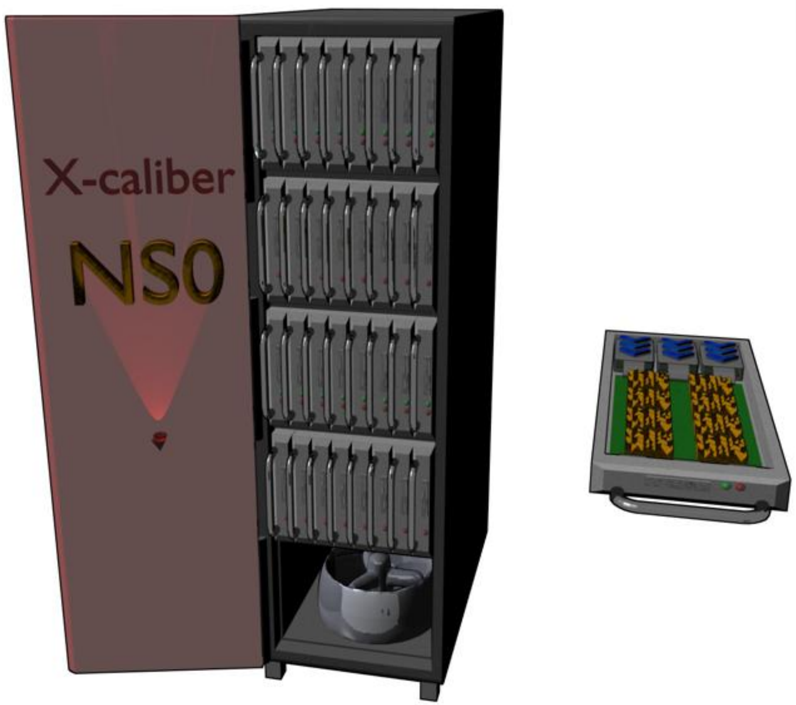
Largely due to Bill Dally, Stanford



UNIVERSITY OF NOTRE DAME

Thanks to Peter Kogge for this slide, based on the DARPA report

An Even More Radical System



- Rack Scale
  - Processing:128 Nodes, 1 (+) PF/s
  - Memory:
    - 128 TB DRAM
    - 0.4 PB/s Aggregate Bandwidth
  - NonVolatile Memory
    - 1 PB Phase Change Memory (addressable)
    - Additional 128 for Redundancy/RAID
  - Network
    - 0.13 PB/sec Injection, 0.06 PB/s Bisection

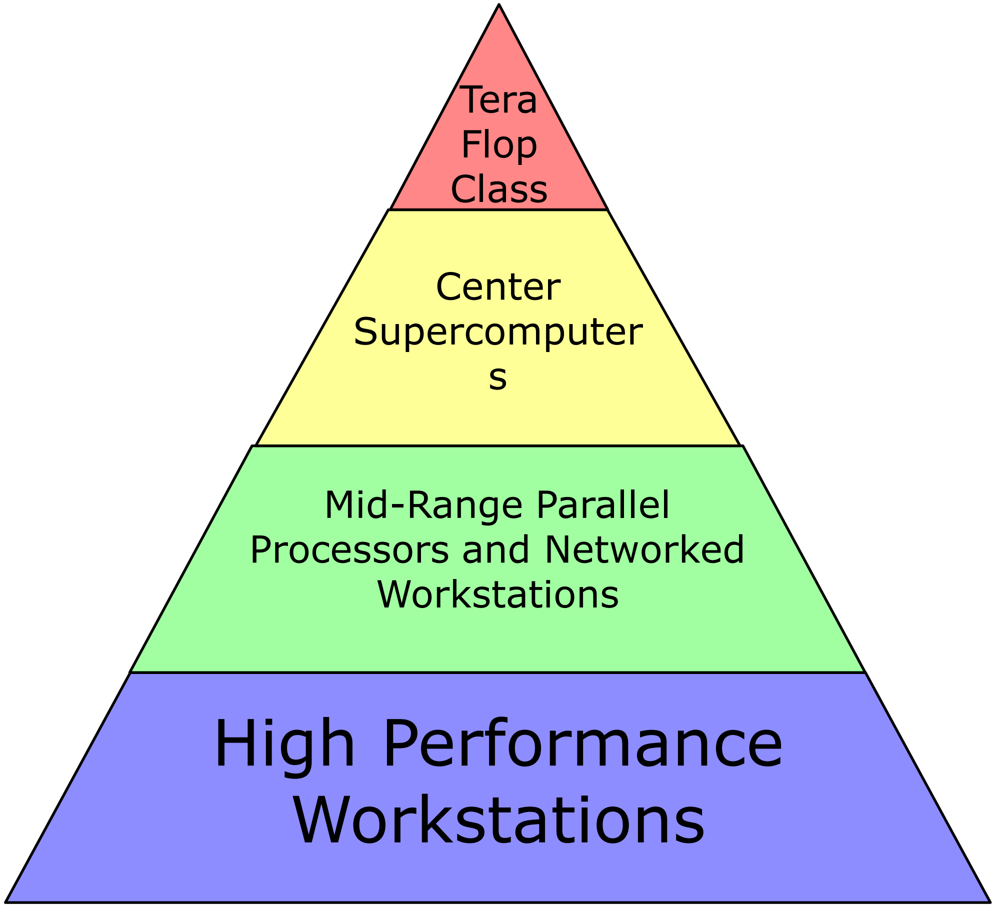
Deployment	Nodes	Topology	Compute	Mem BW	Injection BW	Bisection BW
Module	1	N/A	8 TF/s	3 TB/s	1 TB/s	N/A
Deployable Cage	22	All-to-All	176 TF/s	67.5 TB/s	22.5 TB/s	31 TB/s
Rack	128	Flat. Butterfly	1 PF/s	.4 PB/s	0.13 PB/s	0.066 PB/s
Group Cluster	512	Flat. Butterfly	4.1 PF/s	1.6 PB/s	0.52 PB/s	0.26 PB/s
National Resource	128k	Hier. All-to-All	1 EF/s	0.4 EB/s	0.13 EB/s	16.8 PB/s
Max Configuration	2048k	Hier. All-to-All	16 EF/s	6.4 EB/s	2.1 EB/s	0.26 EB/s

Thanks to Richard Murphy for this slide

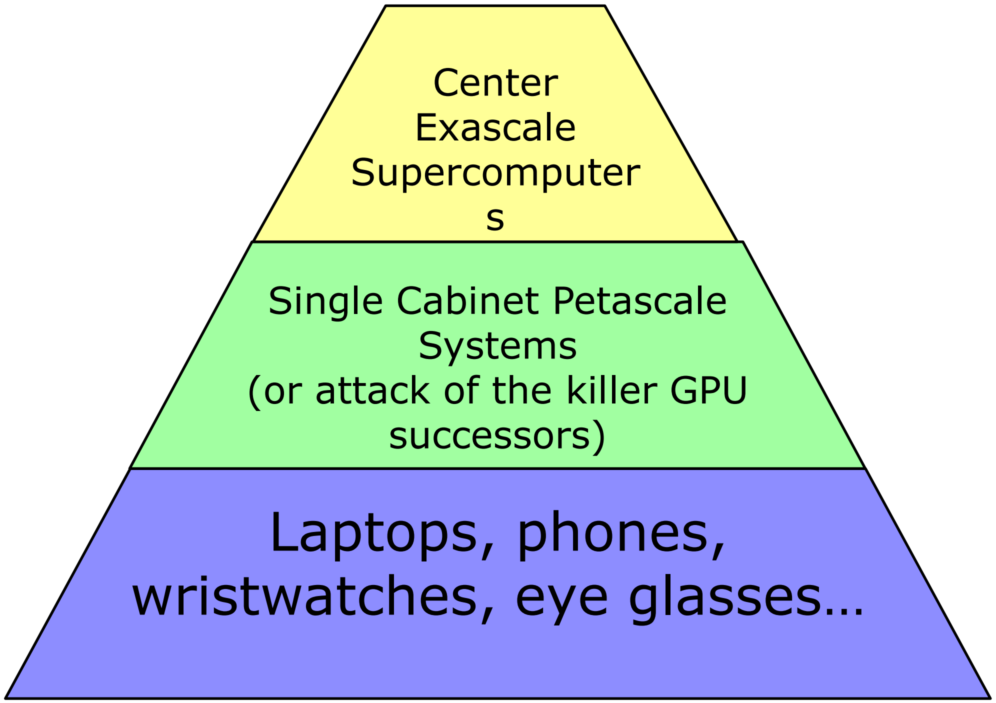
HPC in 2030

- Will we even have Zettaflops (10^{21} Ops/s)?
  - Unlikely (but not impossible) in a single (even highly parallel) system
    - Power (again) – you need an extra 1000-fold improvement in results/Joule over Exascale
    - Concurrency
      - 10^{11}-10^{12} threads (!)
- See the Zettaflops workshops – www.zettaflops.org
  - Will require new device technology
- Will the high-end have reached a limit after Exascale systems?

The HPC Pyramid in 1993



The HPC Pyramid in 2029 (?)



Exascale Challenges

- Exascale will be hard (see the DARPA Report [Kogge])
  - Conventional designs plateau at 100 PF (peak
    - all energy is used to move data
  - Aggressive design is at 70 MW and is very hard to use
    - 600M instruction/cycle - Concurrency
    - 0.0036 Byte moved/flop – All operations local
    - No ECC, no redundancy – Must detect/fix errors
    - No cache memory – Manual management of memory
    - HW failure every 35 minutes – Eeek!
- Waiting doesn’t help
  - At the limits of CMOS technology

Exascale Directions

- 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

Performance, then Productivity

- Note the “then” – not “instead of”
  - For “easier” problems, it is correct to invert these
- For the very hardest problems, we must focus on getting the best performance possible
  - Rely on other approaches to manage the complexity of the codes
  - Performance can be understood and engineered (note I did not say predicted)
- We need to start now, to get practice
  - “Vector” instructions, GPUs, extreme scale networks
  - Because Exascale platforms will be even more complex and harder to use effectively

Going Forward

- What needs to change?
  - Everything!
  - Are we in a local minima (no painless path to improvements)?
- MPI (and parallel languages/frameworks)
- Fortran/C/C++ and "node" language
- Operating System
- Application
- Architecture

Breaking the MPI Stranglehold

- MPI has be very successful
  - Not an accident
  - Replacing MPI requires understanding the strengths of MPI, not just the (sometimes alleged) weaknesses
  - See “Learning from the Success of MPI”, Springer LNCS 2228.

Where Does MPI Need to Change?

- Nowhere
  - There are many MPI legacy applications
  - MPI has added routines to address problems rather than changing them
  - For example, to address problems with the Fortran binding and 64-bit machines, MPI-2 added MPI_Get_address and MPI_Type_create_xxx and deprecated (but did not change or remove) MPI_Address and MPI_Type_xxx.
- Where does MPI need to add routines and deprecate others?
  - For example, the MPI One Sided (RMA) does not match some popular one-sided programming models
  - Nonblocking collectives (approved for MPI-3) needed to provide efficient, scalable performance

Extensions

- What does MPI need that it doesn't have?
- Don't start with that question. Instead ask
  - ♦ What tool do I need? Is there something that MPI needs to work well with that tool (that it doesn't already have)?
- Example: Debugging
  - ♦ Rather than define an MPI debugger, develop a thin and simple interface to allow any MPI implementation to interact with any debugger
- Candidates for this kind of extension
  - ♦ Interactions with process managers
    - Thread co-existence
    - Choice of resources (e.g., placement of processes with Spawn) Interactions with Integrated Development Environments (IDE)
  - ♦ Tools to create and manage MPI datatypes
  - ♦ Tools to create and manage distributed data structures
    - A feature of the HPCS languages

Challenges

- Must avoid the traps:
  - The challenge is **not** to make easy programs easier. The challenge is to make hard programs **possible**.
  - We need a "well-posedness" concept for programming tasks
    - Small changes in the requirements should require small changes in the code
    - Rarely a property of "high productivity" languages
  - Latency hiding is not the same as low latency
    - Need "Support for aggregate operations on large collections"
- An even harder challenge: make it hard to write incorrect programs.
  - OpenMP is not a step in the (entirely) right direction
  - In general, current shared memory programming models are very dangerous.
    - They also perform action at a distance
    - They require a kind of user-managed data decomposition to preserve performance without the cost of locks/memory atomic operations
  - **Deterministic** algorithms should have **provably deterministic implementations**
    - Some steps in this direction, such as deterministic parallel Java

How to Replace MPI

- Retain MPI's strengths
  - Performance from matching programming model to the realities of underlying hardware
  - Ability to compose with other software (libraries, compilers, debuggers)
  - Determinism (without MPI_ANY_{TAG,SOURCE})
  - Run-everywhere portability
- Add to what MPI is missing, such as
  - Distributed data structures (not just a few popular ones)
  - Low overhead remote operations; better latency hiding/management; overlap with computation (not just latency; MPI can be implemented in a few hundred instructions, so overhead is roughly the same as remote memory reference (memory wall))
  - Dynamic load balancing for dynamic, distributed data structures
  - Unified method for treating multicores, remote processors, other resources
- Enable the transition from MPI programs
  - Build component-friendly solutions
    - There is no one, true language

Issues for MPI in the Petascale Era

- Complement MPI with support for
  - Distributed (possibly dynamic) data structures
  - Improved node performance (including multicore)
    - May include tighter integration, such as MPI+OpenMP with compiler and runtime awareness of both
    - Must be coupled with latency tolerant and memory hierarchy sensitive algorithms
  - Fault tolerance
  - Load balancing
- Address the real memory wall - latency
  - Likely to need hardware support + programming models to handle memory consistency model
- MPI RMA model needs updating
  - To match locally cache-coherent hardware designs, atomic remote op
  - All part of current MPI 3 RMA proposal; likely to pass
- Parallel I/O model needs more support
  - For optimal productivity of the computational scientist, data files should be processor-count independent (canonical form)

MPI-3 For Petascale and Beyond

- MPI Forum active and defining new features for MPI
  - New collectives, including non-blocking and neighbor
  - New remove memory access (RMA), including optimization for cache coherent systems and remote atomic ops
  - Better support for hybrid programming models and threads
  - Improved language bindings for Fortran, C++
  - Fault tolerance
  - Enhanced tool interface ("performance debugging")
- See meetings.mpi-forum.org and https://svn.mpi-forum.org/trac/mpi-forum-web/wiki

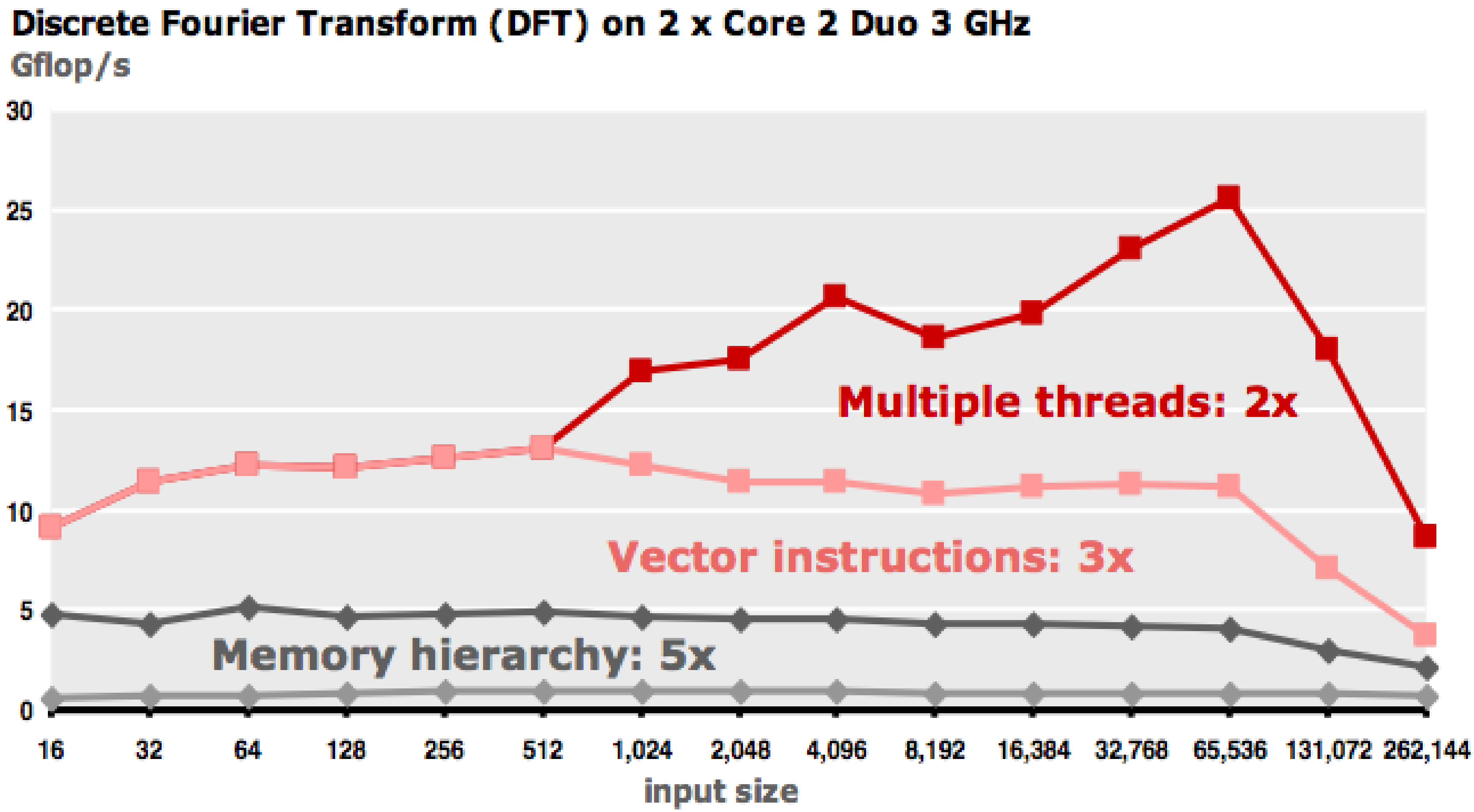
Breaking the Fortran/C/C++ Stranglehold

- Issue:
  - Ad hoc concurrency model
  - Mismatch to user needs
  - Mismatch to hardware
  - Lack of support for correctness or performance
- Summed up: Support for what is really hard in writing effective programs
- Improve node performance
  - Make the compiler better
  - Give better code to the compiler
  - Get realistic with algorithms/data structures

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)
  - Possibly most studied numerical kernel for compilation *and* a key part of major benchmarks, yet good performance requires use of specialized code ….

Compilers Versus Libraries in DFT



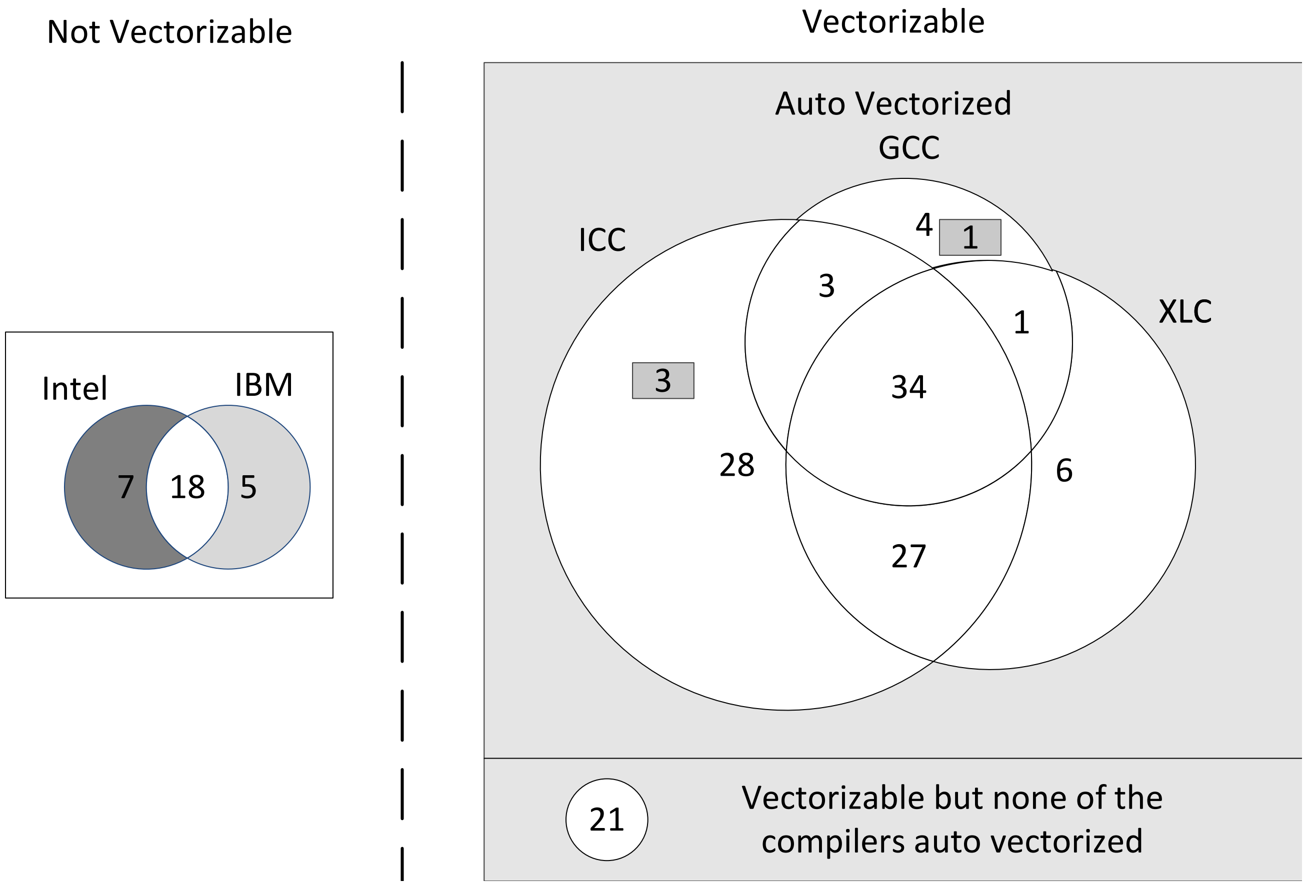
Source: Markus Püschel. Spring 2008.

PARALLEL@ILLINOIS

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 are developing a test suite that evaluates how the compiler does with such equivalent code
  - 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.

How Good are Compilers at Vectorizing Codes?



S. Maleki, Y. Gao, T. Wong, M. Garzarán, and D. Padua. An Evaluation of Vectorizing Compilers. PACT 2011.

Give "Better" Code to the Compiler

- Fixing the compilers is a long term (at best) project. What can we do in the meantime?
- 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 Requirements and Implications

- Portable - augment existing language.
  - 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
- Maintainable
  - Let user work with original code *and* ensure changes automatically update tuned code
- Correct
  - Do whatever the app developer needs to believe that the tuned code is correct
- 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 may be ok for performance expert; some compilers already provide pragmas for some transformations
- Right (maybe): Performance precious, typical loop count between 100 and 10000, even, not power of 2
- We need work at developing higher-level, performance-oriented languages or language extensions
  - DOE "X-stack" may (or may not) help with this

Breaking the OS Stranglehold

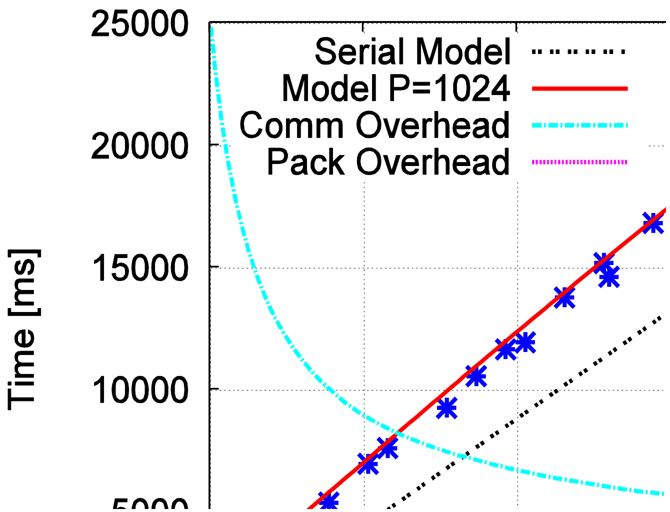
- Middle ground between single system image and single node OS everywhere
- Single system image
  - Hard to fully distribute
  - Not clear that it is needed
  - But *some* features require coordination
  - Examples include collective I/O (for file open/close and coordinated read/write), scheduling (for services that must not interfere with loosely synchronized applications), and memory allocation for PGAS languages

Breaking the Application Stranglehold

- Problem
  - Applications often frozen in legacy programming systems; modified for idiosyncrasies of this year's system
- Solution
  - Use of abstraction, autotuning, tools
  - Interoperable programming models and frameworks
  - Make "performance correctness" equally important; guide decisions in selection of system, algorithms, implementation

Model-guided Optimization

- Application is MILC, a lattice QCG code
- Analytic model showed possible improvement of 12% by eliminating the pack before communicating
- Torsten Hoefler implemented and analyzed for EuroMPI’10
  - Up to **18%** faster!
- Next bottleneck: CG phase
  - Investigating use of nonblocking collectives in a modified CG
  - Also model-driven (because involves more floating point but same or less data motion)



PARALLEL@ILLINOIS

Hardest: Breaking the Architecture Stranglehold

- Greater power efficiency implies less speculation in operation, memory
- Must still be able to reason about what is happening (can’t just have ad hoc memory consistency, e.g.)
- Need coordinated advances in software, algorithms, and architecture
  - Danger is special purpose hardware, constrained by today’s software, old algorithms
  - “Tomorrows hardware, with today’s software, running yesterday’s algorithms”
  - Particularly essential for fault tolerance, latency hiding

Research Directions Towards Exascale

- Integrated, interoperable, component oriented languages
  - Generalization of so-called domain-specific language
    - Really (abstract) data-structure-specific languages
    - Example: matlab is not a D(omain)SL but is a D(atastructure)SL
- Performance modeling and tuning
  - Performance info in language; performance considered as part of correctness
- Fault tolerance at the high end
  - Fault tolerance features in the language, working with hardware and algorithms
- Correctness
  - Correctness features for testing in the language
  - Support for special cases (e.g., provably deterministic expression of deterministic algorithms)

Conclusions

- Planning for extreme scale systems requires rethinking both algorithms and programming approaches
- Key requirements include
  - Minimizing memory motion at all levels
  - Avoiding unnecessary synchronization at all levels
- Decisions must be informed by performance modeling / understanding
  - Not necessarily performance estimates – the goal is to guide the decisions

Of Interest

- Special Interest Group in HPC
  - sighpc.org
- Annual Supercomputing conference
  - SC12.supercomputing.org in Salt Lake City, Utah
  - SC13 in Denver Colorado
- New Parallel Computing Institute at Illinois
  - www.parallel.illinois.edu

Conclusions

- Practical issues require separating algorithm, data structure, and implementation
  - ♦ Libraries will need to be supplemented by generated code
  - ♦ They may be data-structure-specific languages or annotations
    - Most proposals are not for domain specific, as they make assumptions about data structure and algorithm
    - Matlab is, after all, not domain specific – it is primarily data structure specific

Possible Solution Directions

- Use mathematics as the organizing principle
  - Continuous representations, possibly adaptive, memory-optimizing representation, lossy (within accuracy limits) but preserves essential properties (e.g., conservation)
- Manage code by using data-structure-specific languages to handle operations and vertical integration across components
  - So-called "domain specific languages" are really data-structure specific languages – they support more applications but fewer algorithms.
  - Difference is important because a "domain" almost certainly require flexibility with data structures and algorithms

Possible Solution Directions

- Adaptive program models with a multi-level approach
  - Lightweight, locality-optimized for fine grain
  - Within node/locality domain for medium grain
  - Regional/global for coarse grain
  - May be different programming models (hierarchies are ok!) but they must work well together
- Performance annotations to support a complex compilation environment
- Asynchronous algorithms
- Integrated Development Environment (IDE) to ease vertical code development, maintenance, and refactoring