HPC in 2020
How Will We Get There?

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

Extrapolation is Risky

- 1989 – T – 21 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 – 11 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
    - 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

- Sustained PF systems
  - NSF Track 1 “Blue Waters” at Illinois
  - “Sequoia” Blue Gene/Q at LLNL
  - Undoubtedly others (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
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

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](http://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)
  - 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

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 been very successful
  - Not an accident
  - Replacing MPI requires understanding the strengths of MPI, not just the (sometimes alleged) weaknesses

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 (proposed 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 (MPIT discussions)
    • 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

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

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

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
  ♦ Add better atomic remote op support
• Parallel I/O model needs more support
  ♦ For optimal productivity of the computational scientist, data files should be processor-count independent (canonical form)
Breaking the Fortran/C/C++ Stranglehold

• Issue:
  ♦ Ad hoc concurrency model
  ♦ Mismatch to user needs
  ♦ Mismatch to hardware
  ♦ Lack of support for correctness

• 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)

Compilers Versus Libraries in DFT


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
  ♦ Identify necessary transformations and 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 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 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
• We need work at developing higher-level, performance-oriented languages or language extensions
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 froze in legacy programming systems; modified for idiosyncrasies of this year’s system
- Solution
  - Use of abstraction, autotuning, tools
  - Interoperable programming models and frameworks

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
  - “Tomorrow’s hardware, with today’s software, running yesterday’s algorithms”
  - Particularly essential for fault tolerance, latency hiding

Research Directions

- Integrated, interoperable, component oriented languages
  - Generalization of so-called domain-specific language
    - Really data-structure-specific languages
- 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)