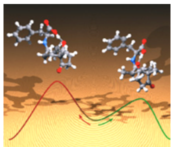
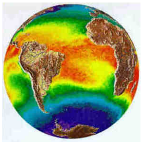
Performance, Correctness, and Programmability: Challenges for Parallel Programming at Exascale

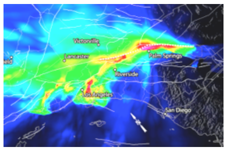
William Gropp
www.cs.illinois.edu/~wgropp

Why Exascale?

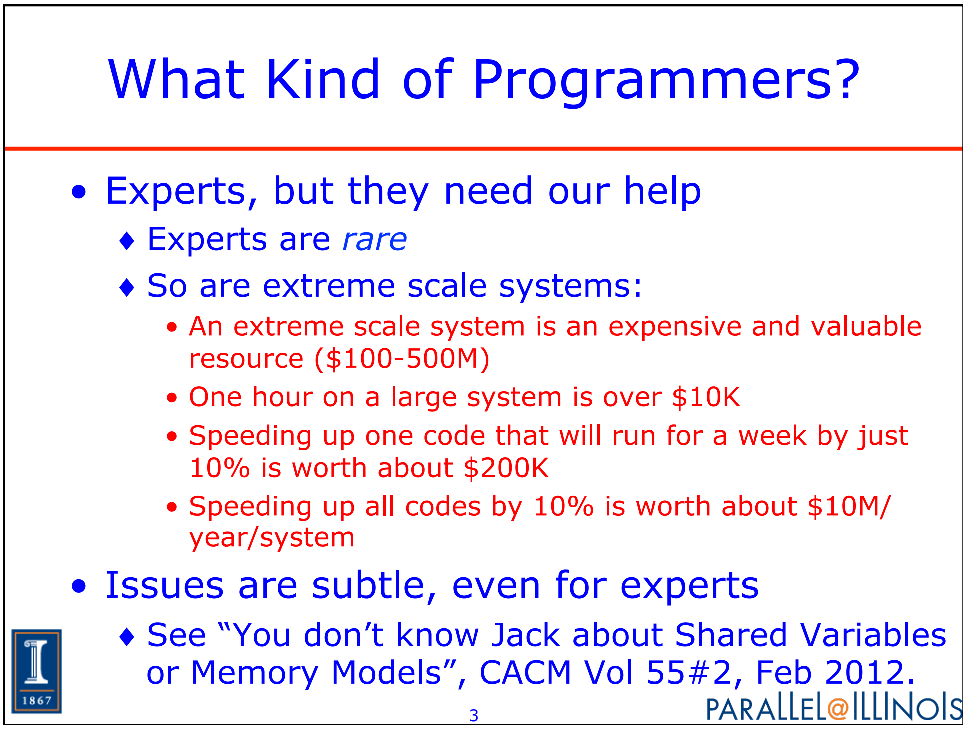
- Many applications require more accuracy/ features that currently available
  - ♦ More computation (more complex models, higher resolution)
  - ♦ More data (which in turn requires computation)
  - ♦ The challenge is to do ***science*** with these





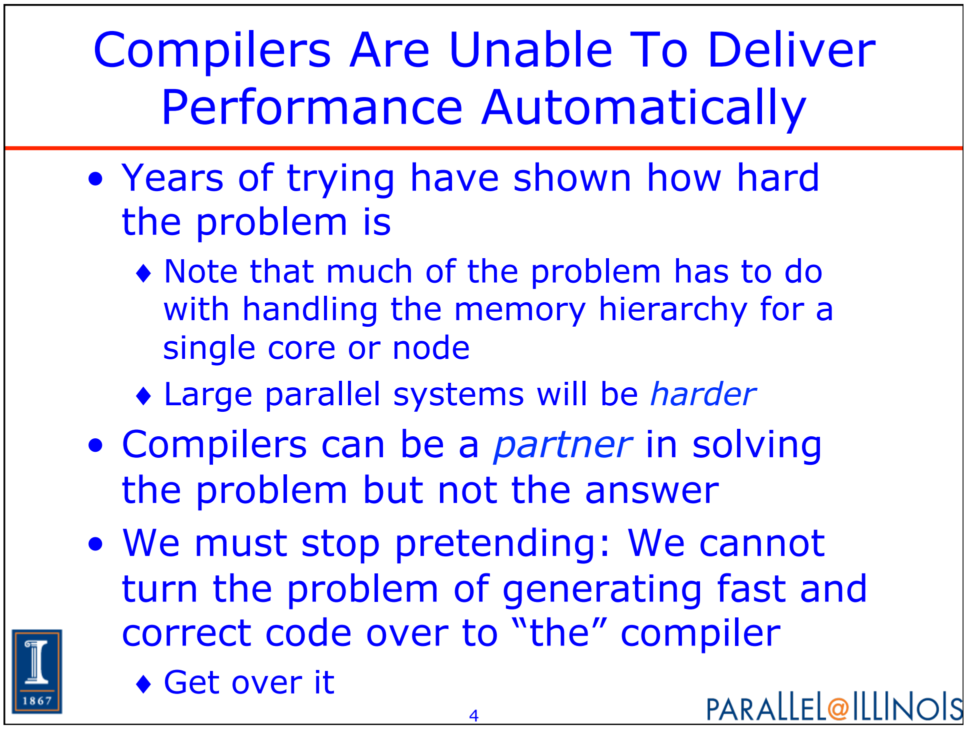


- Common theme is *performance*.



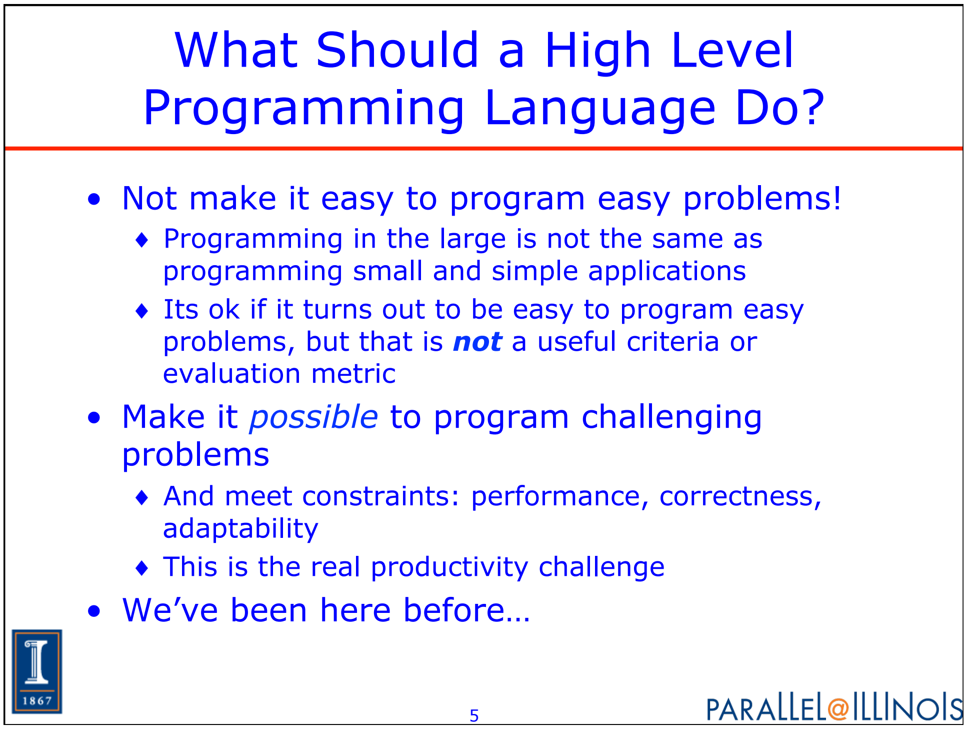
What Kind of Programmers?

- Experts, but they need our help
  - ♦ Experts are *rare*
  - ♦ So are extreme scale systems:
    - An extreme scale system is an expensive and valuable resource ($100-500M)
    - One hour on a large system is over $10K
    - Speeding up one code that will run for a week by just 10% is worth about $200K
    - Speeding up all codes by 10% is worth about $10M/year/system
- Issues are subtle, even for experts
  - ♦ See "You don't know Jack about Shared Variables or Memory Models", CACM Vol 55#2, Feb 2012.



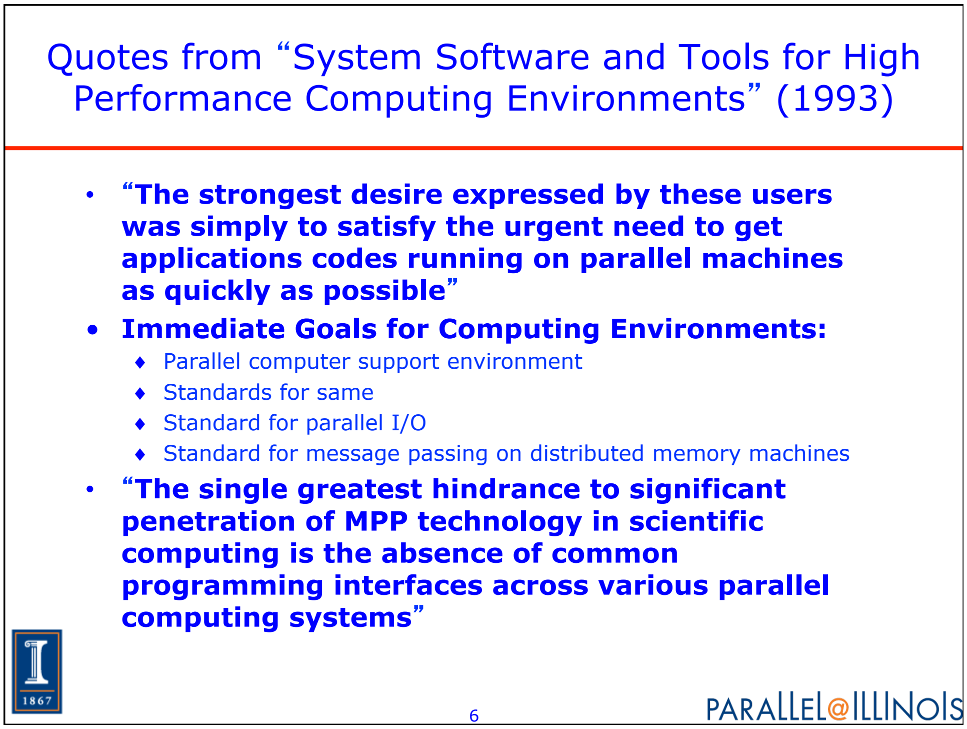
Compilers Are Unable To Deliver Performance Automatically

- Years of trying have shown how hard the problem is
  - ♦ Note that much of the problem has to do with handling the memory hierarchy for a single core or node
  - ♦ Large parallel systems will be *harder*
- Compilers can be a *partner* in solving the problem but not the answer
- We must stop pretending: We cannot turn the problem of generating fast and correct code over to "the" compiler
  - ♦ Get over it



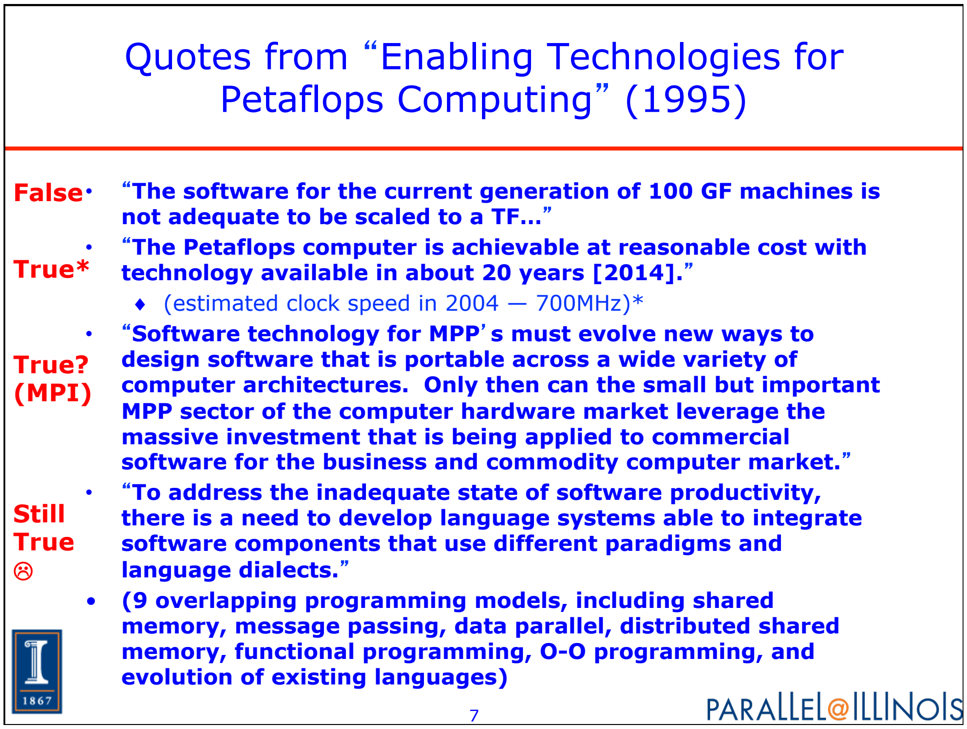
What Should a High Level Programming Language Do?

- Not make it easy to program easy problems!
  - Programming in the large is not the same as programming small and simple applications
  - Its ok if it turns out to be easy to program easy problems, but that is ***not*** a useful criteria or evaluation metric
- Make it *possible* to program challenging problems
  - And meet constraints: performance, correctness, adaptability
  - This is the real productivity challenge
- We've been here before…



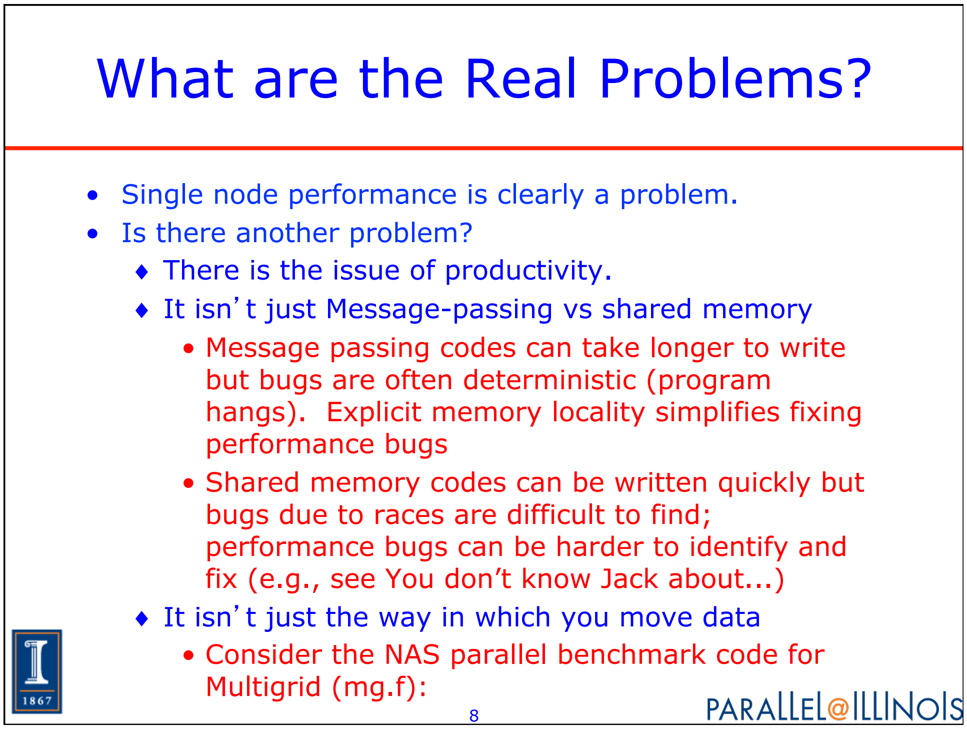
Quotes from "System Software and Tools for High Performance Computing Environments" (1993)

- **"The strongest desire expressed by these users was simply to satisfy the urgent need to get applications codes running on parallel machines as quickly as possible"**
- **Immediate Goals for Computing Environments:**
  - Parallel computer support environment
  - Standards for same
  - Standard for parallel I/O
  - Standard for message passing on distributed memory machines
- **"The single greatest hindrance to significant penetration of MPP technology in scientific computing is the absence of common programming interfaces across various parallel computing systems"**



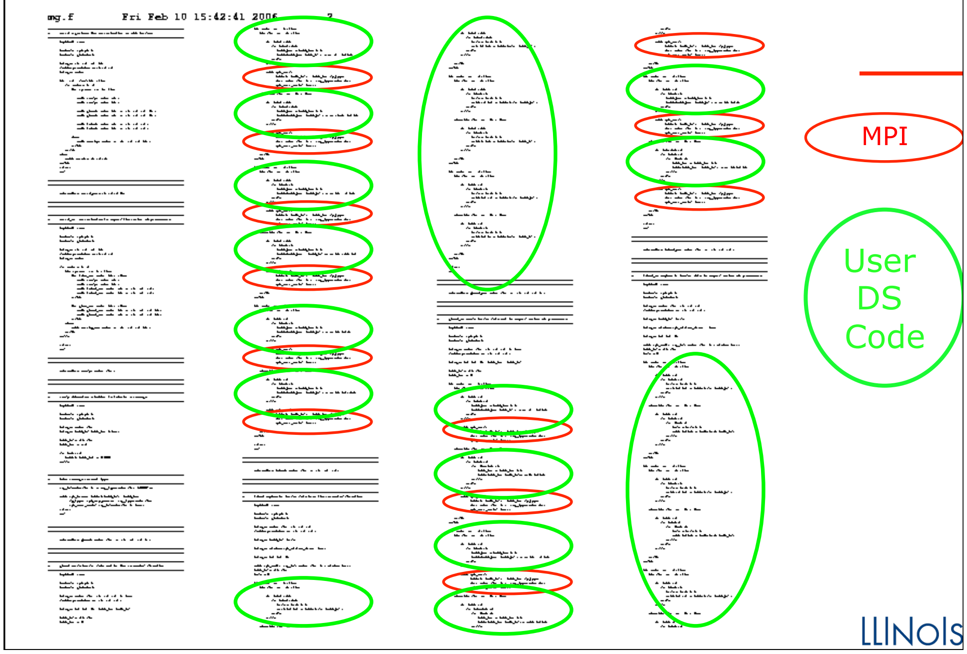
Quotes from “Enabling Technologies for Petaflops Computing” (1995)

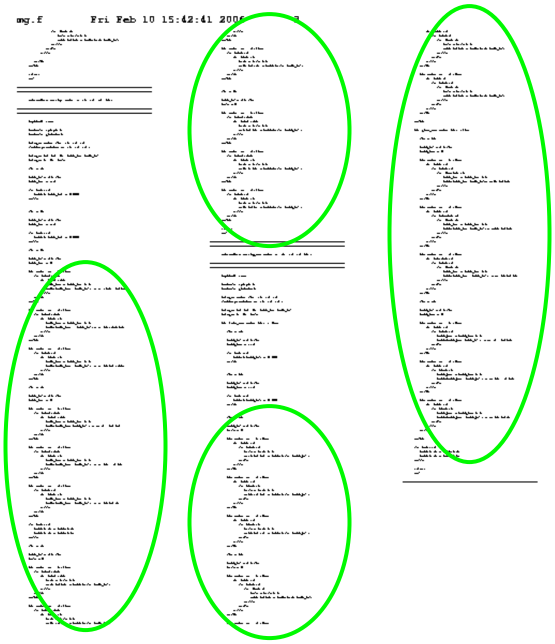
- **False** **“The software for the current generation of 100 GF machines is not adequate to be scaled to a TF…”**
- **True*** **“The Petaflops computer is achievable at reasonable cost with technology available in about 20 years [2014].”**
  - (estimated clock speed in 2004 — 700MHz)*
- **True? (MPI)** **“Software technology for MPP’s must evolve new ways to design software that is portable across a wide variety of computer architectures. Only then can the small but important MPP sector of the computer hardware market leverage the massive investment that is being applied to commercial software for the business and commodity computer market.”**
- **Still True ☹** **“To address the inadequate state of software productivity, there is a need to develop language systems able to integrate software components that use different paradigms and language dialects.”**
- **(9 overlapping programming models, including shared memory, message passing, data parallel, distributed shared memory, functional programming, O-O programming, and evolution of existing languages)**



What are the Real Problems?

- Single node performance is clearly a problem.
- Is there another problem?
  - There is the issue of productivity.
  - It isn’t just Message-passing vs shared memory
    - Message passing codes can take longer to write but bugs are often deterministic (program hangs). Explicit memory locality simplifies fixing performance bugs
    - Shared memory codes can be written quickly but bugs due to races are difficult to find; performance bugs can be harder to identify and fix (e.g., see You don’t know Jack about...)
  - It isn’t just the way in which you move data
    - Consider the NAS parallel benchmark code for Multigrid (mg.f):

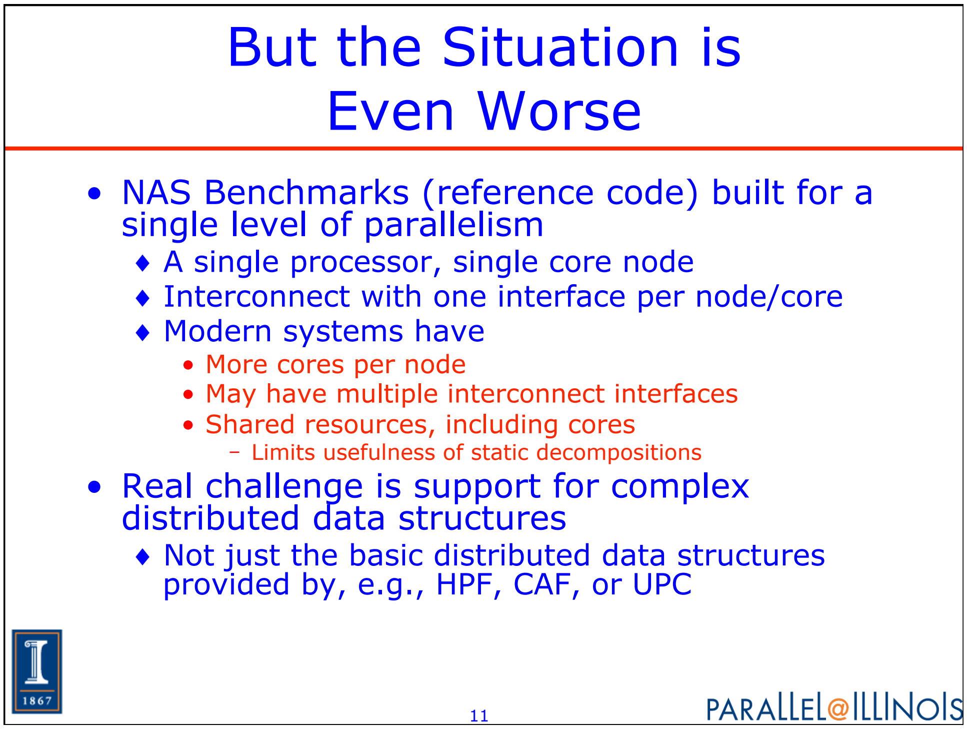




What is the problem? The user is responsible for all steps in the decomposition of the data structures across the processors

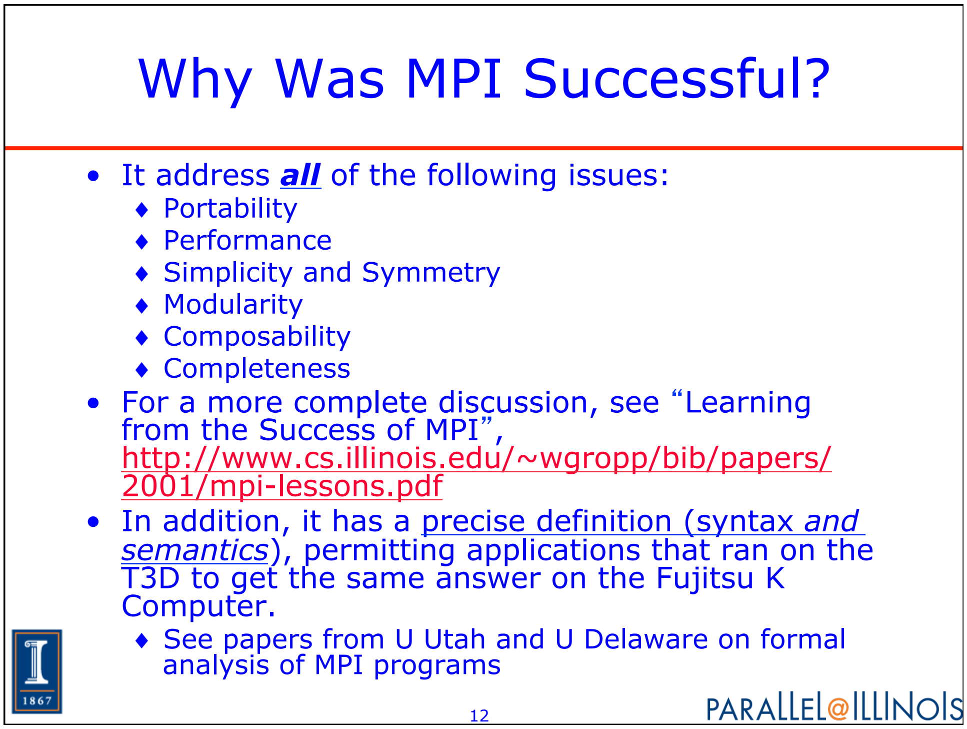
Note that this does give the user (or someone) a great deal of flexibility, as the data structure can be distributed in arbitrary ways across arbitrary sets of processors

*Users want **their** data structures*



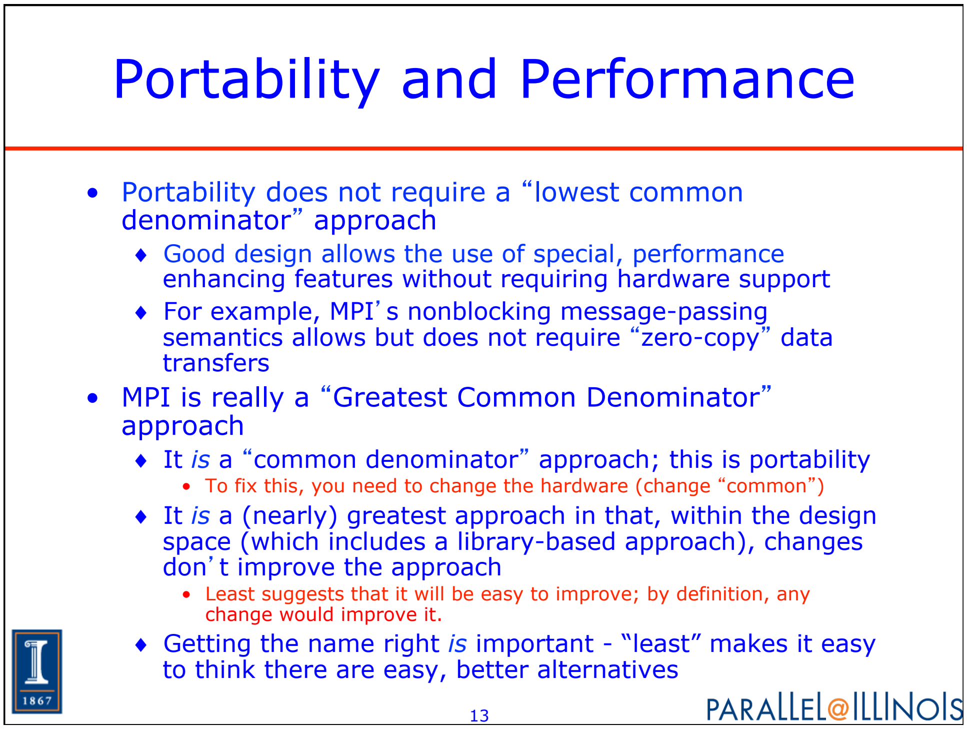
But the Situation is Even Worse

- NAS Benchmarks (reference code) built for a single level of parallelism
  - ♦ A single processor, single core node
  - ♦ Interconnect with one interface per node/core
  - ♦ Modern systems have
    - More cores per node
    - May have multiple interconnect interfaces
    - Shared resources, including cores
      - – Limits usefulness of static decompositions
- Real challenge is support for complex distributed data structures
  - ♦ Not just the basic distributed data structures provided by, e.g., HPF, CAF, or UPC



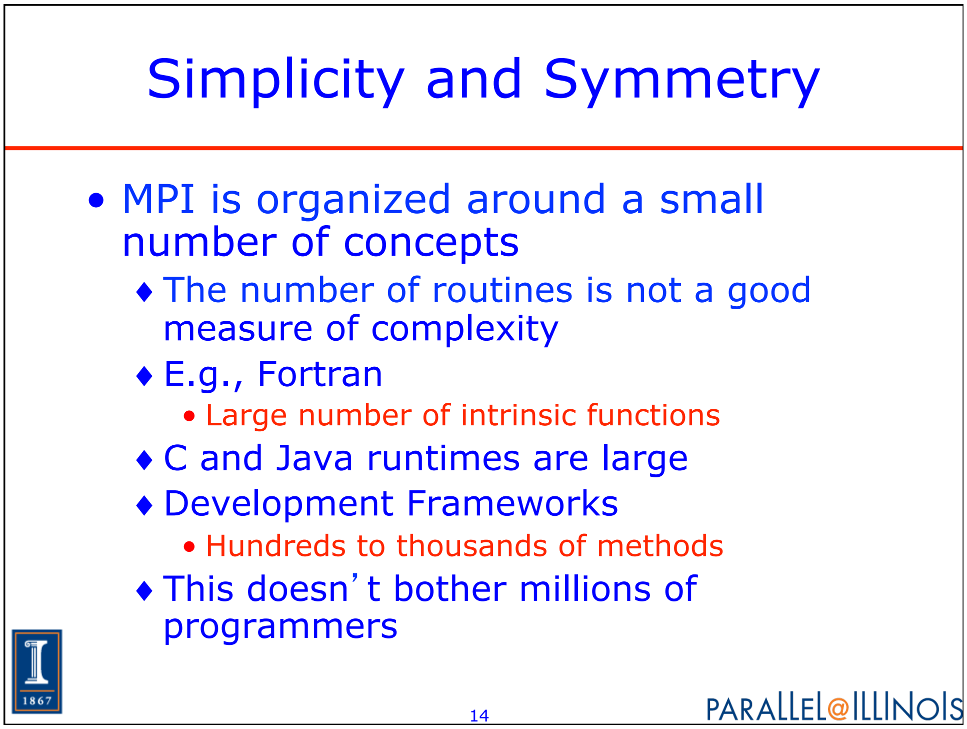
Why Was MPI Successful?

- It address ***all*** of the following issues:
  - ♦ Portability
  - ♦ Performance
  - ♦ Simplicity and Symmetry
  - ♦ Modularity
  - ♦ Composability
  - ♦ Completeness
- For a more complete discussion, see “Learning from the Success of MPI”, http://www.cs.illinois.edu/~wgropp/bib/papers/2001/mpi-lessons.pdf
- In addition, it has a precise definition (syntax *and semantics*), permitting applications that ran on the T3D to get the same answer on the Fujitsu K Computer.
  - ♦ See papers from U Utah and U Delaware on formal analysis of MPI programs



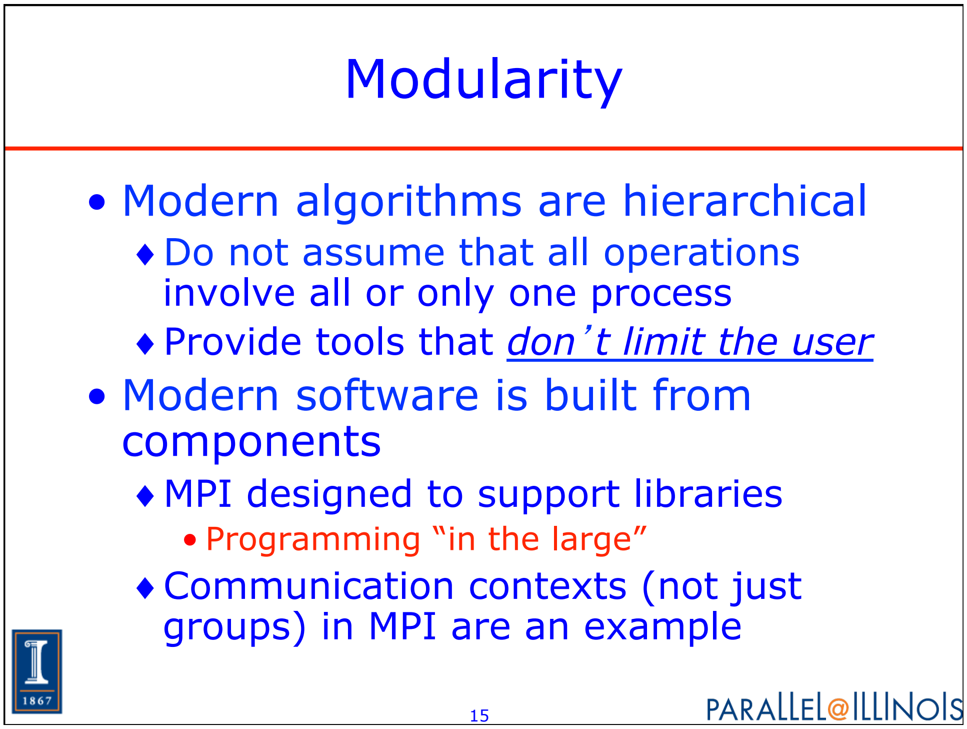
Portability and Performance

- Portability does not require a “lowest common denominator” approach
  - ♦ Good design allows the use of special, performance enhancing features without requiring hardware support
  - ♦ For example, MPI’s nonblocking message-passing semantics allows but does not require “zero-copy” data transfers
- MPI is really a “Greatest Common Denominator” approach
  - ♦ It *is* a “common denominator” approach; this is portability
    - To fix this, you need to change the hardware (change “common”)
  - ♦ It *is* a (nearly) greatest approach in that, within the design space (which includes a library-based approach), changes don’t improve the approach
    - Least suggests that it will be easy to improve; by definition, any change would improve it.
  - ♦ Getting the name right *is* important - “least” makes it easy to think there are easy, better alternatives



Simplicity and Symmetry

- MPI is organized around a small number of concepts
  - ♦ The number of routines is not a good measure of complexity
  - ♦ E.g., Fortran
    - Large number of intrinsic functions
  - ♦ C and Java runtimes are large
  - ♦ Development Frameworks
    - Hundreds to thousands of methods
  - ♦ This doesn’t bother millions of programmers

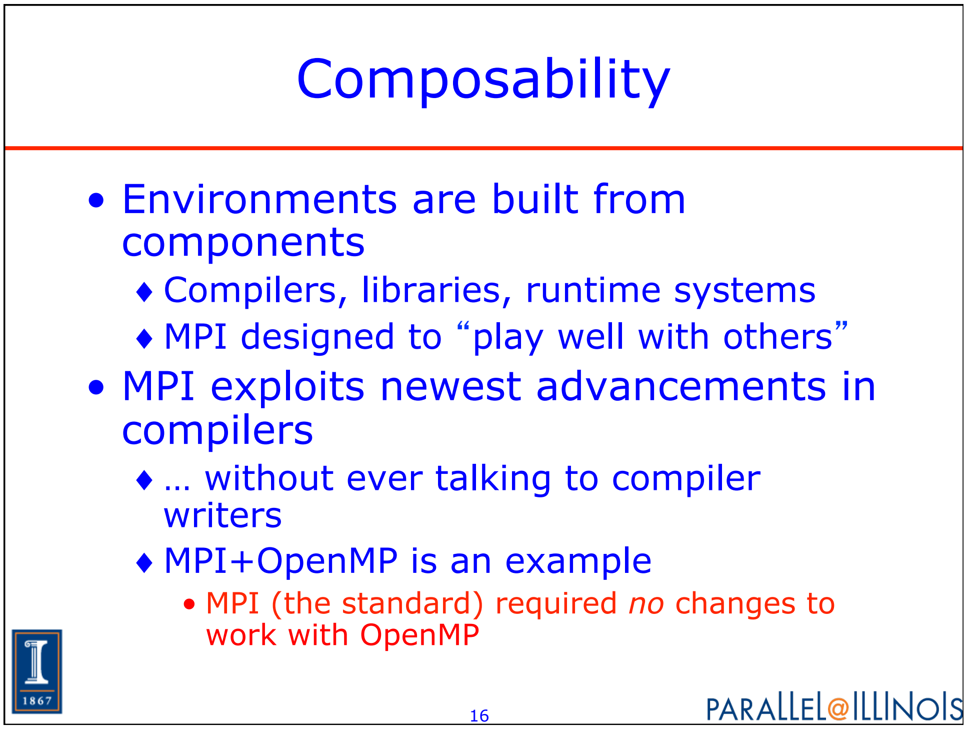


Modularity

- Modern algorithms are hierarchical
  - ♦ Do not assume that all operations involve all or only one process
  - ♦ Provide tools that *don't limit the user*
- Modern software is built from components
  - ♦ MPI designed to support libraries
    - Programming "in the large"
  - ♦ Communication contexts (not just groups) in MPI are an example

15

PARALLEL@ILLINOIS

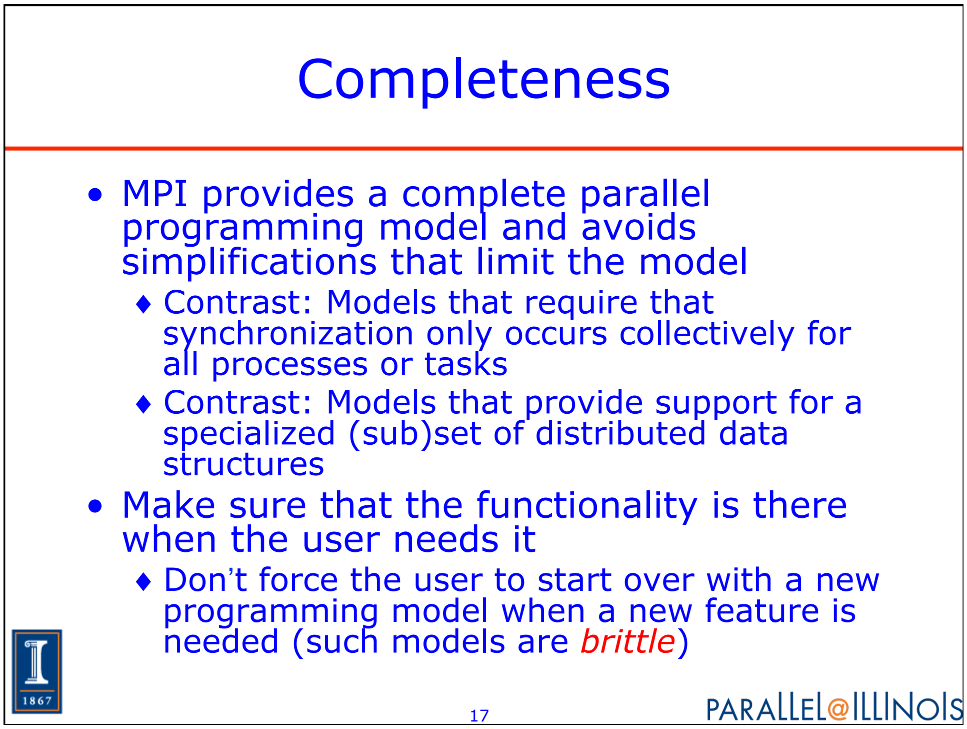


Composability

- Environments are built from components
  - ♦ Compilers, libraries, runtime systems
  - ♦ MPI designed to "play well with others"
- MPI exploits newest advancements in compilers
  - ♦ … without ever talking to compiler writers
  - ♦ MPI+OpenMP is an example
    - MPI (the standard) required *no* changes to work with OpenMP

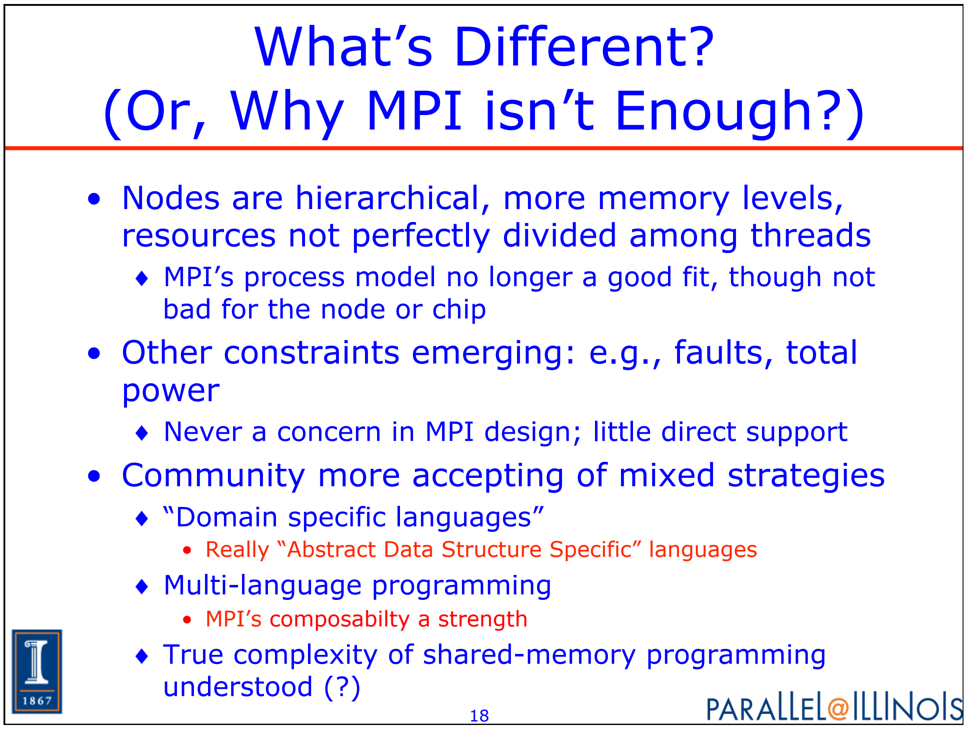
16

PARALLEL@ILLINOIS



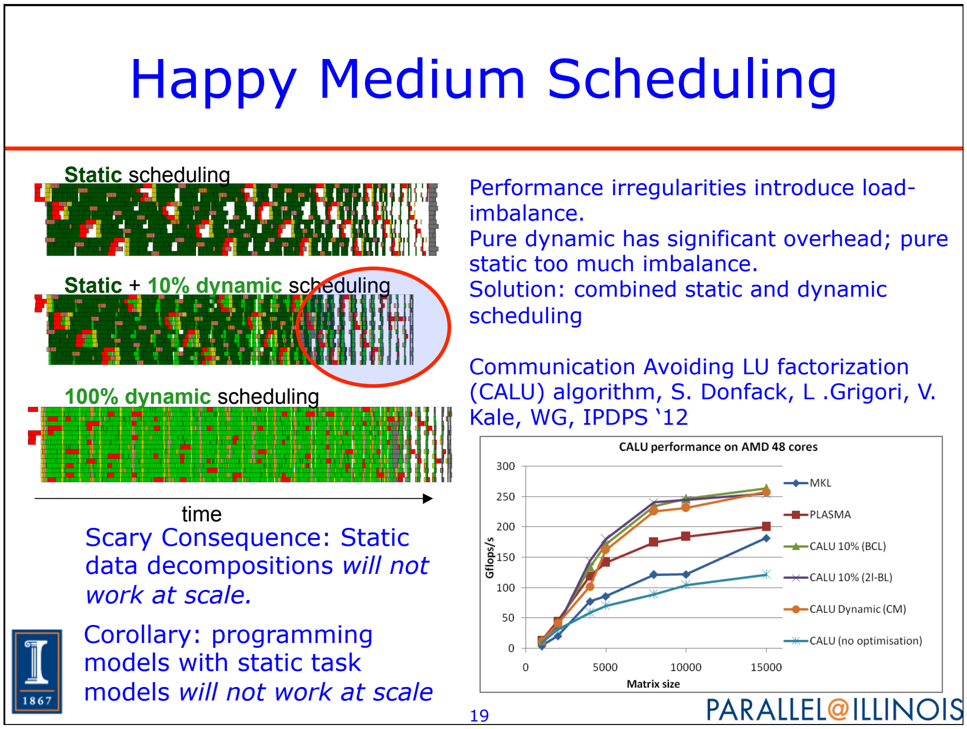
Completeness

- MPI provides a complete parallel programming model and avoids simplifications that limit the model
  - ♦ Contrast: Models that require that synchronization only occurs collectively for all processes or tasks
  - ♦ Contrast: Models that provide support for a specialized (sub)set of distributed data structures
- Make sure that the functionality is there when the user needs it
  - ♦ Don't force the user to start over with a new programming model when a new feature is needed (such models are *brittle*)



What's Different? (Or, Why MPI isn't Enough?)

- Nodes are hierarchical, more memory levels, resources not perfectly divided among threads
  - ♦ MPI's process model no longer a good fit, though not bad for the node or chip
- Other constraints emerging: e.g., faults, total power
  - ♦ Never a concern in MPI design; little direct support
- Community more accepting of mixed strategies
  - ♦ "Domain specific languages"
    - Really "Abstract Data Structure Specific" languages
  - ♦ Multi-language programming
    - MPI's composabilty a strength
  - ♦ True complexity of shared-memory programming understood (?)



Happy Medium Scheduling

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

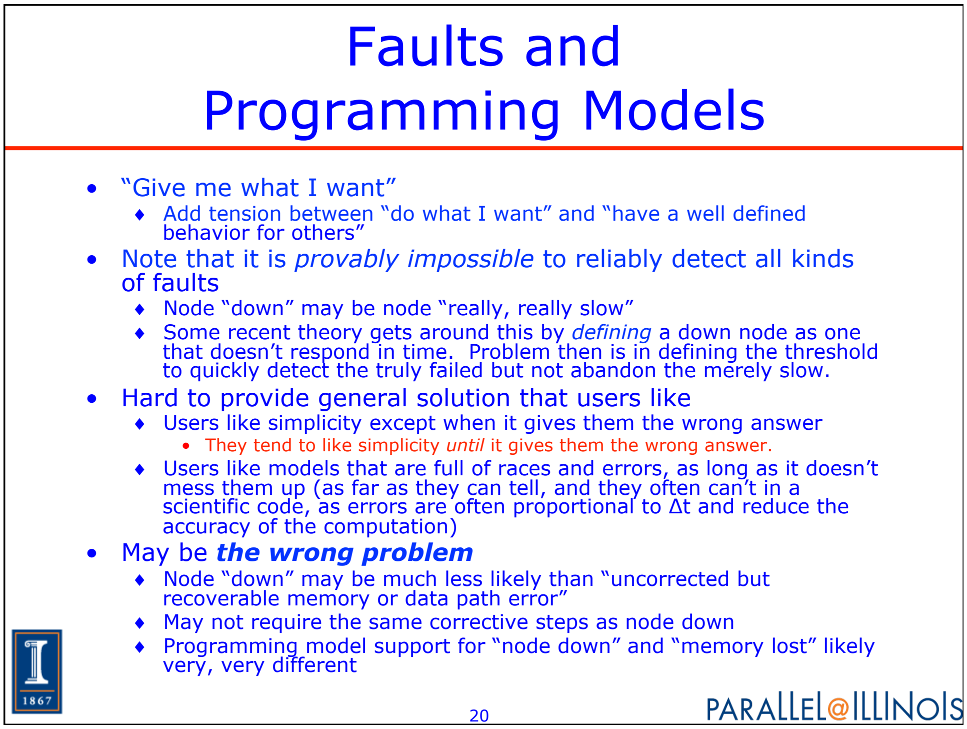
Corollary: programming models with static task models *will not work at scale*

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

19

PARALLEL@ILLINOIS

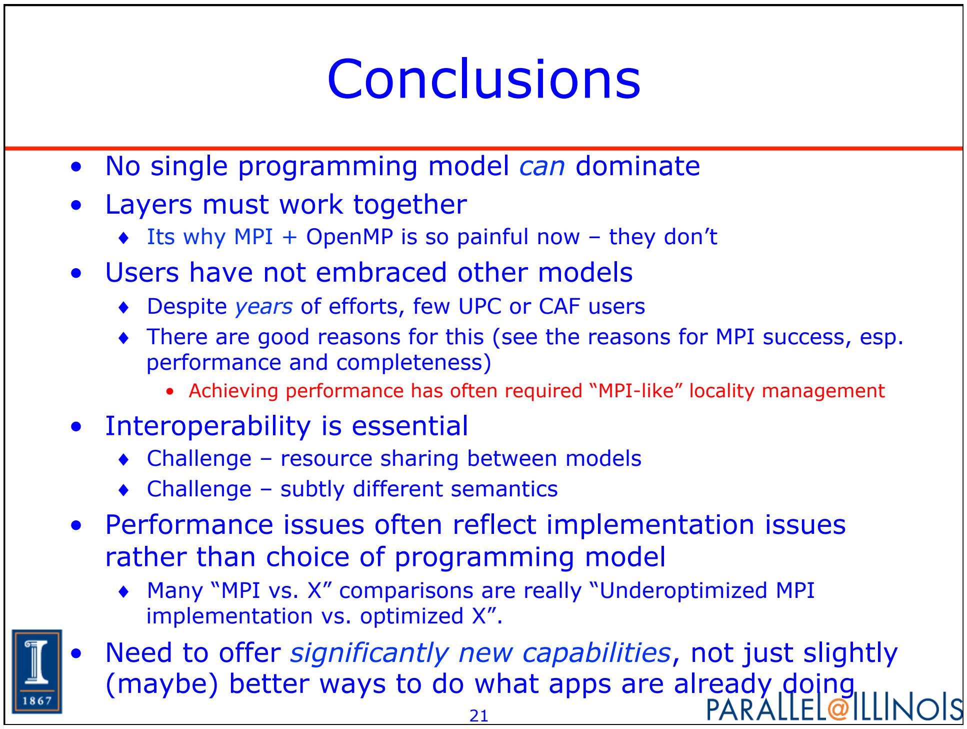


Faults and Programming Models

- "Give me what I want"
  - Add tension between "do what I want" and "have a well defined behavior for others"
- Note that it is *provably impossible* to reliably detect all kinds of faults
  - Node "down" may be node "really, really slow"
  - Some recent theory gets around this by *defining* a down node as one that doesn't respond in time. Problem then is in defining the threshold to quickly detect the truly failed but not abandon the merely slow.
- Hard to provide general solution that users like
  - Users like simplicity except when it gives them the wrong answer
    - They tend to like simplicity *until* it gives them the wrong answer.
  - Users like models that are full of races and errors, as long as it doesn't mess them up (as far as they can tell, and they often can't in a scientific code, as errors are often proportional to Δt and reduce the accuracy of the computation)
- May be ***the wrong problem***
  - Node "down" may be much less likely than "uncorrected but recoverable memory or data path error"
  - May not require the same corrective steps as node down
  - Programming model support for "node down" and "memory lost" likely very, very different

20

PARALLEL@ILLINOIS



Conclusions

- No single programming model *can* dominate
- Layers must work together
  - Its why MPI + OpenMP is so painful now – they don't
- Users have not embraced other models
  - Despite *years* of efforts, few UPC or CAF users
  - There are good reasons for this (see the reasons for MPI success, esp. performance and completeness)
    - Achieving performance has often required "MPI-like" locality management
- Interoperability is essential
  - Challenge – resource sharing between models
  - Challenge – subtly different semantics
- Performance issues often reflect implementation issues rather than choice of programming model
  - Many "MPI vs. X" comparisons are really "Underoptimized MPI implementation vs. optimized X".
- Need to offer *significantly new capabilities*, not just slightly (maybe) better ways to do what apps are already doing