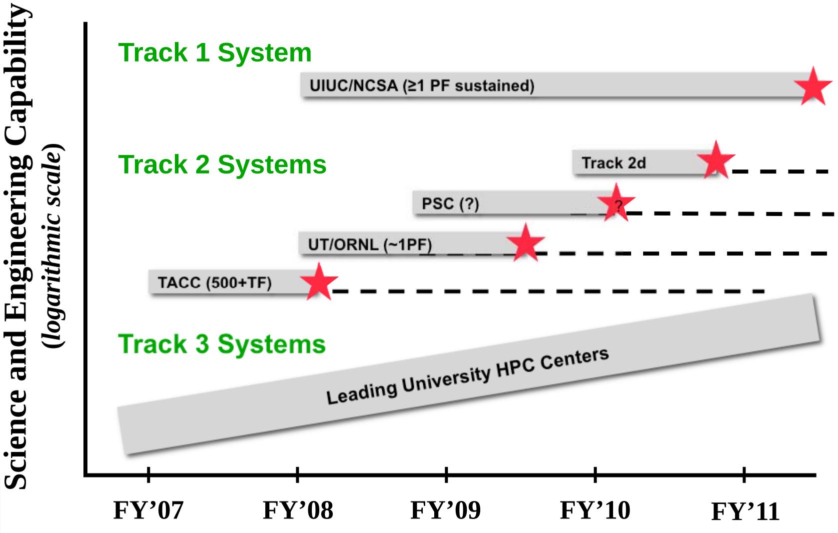
Computing in the Trans-PetaFLOP Era

William Gropp
www.cs.uiuc.edu/homes/wgropp

PetaFLOPS are Here

NSF’s Strategy for High-end Computing



Blue Waters Computing System

System Attribute	Abe	Blue Waters
Vendor	Dell	IBM
Processor	Intel Xeon 5300	IBM Power7
Peak Performance (TF)	0.090	
Sustained Performance (TF)	0.005	≥1PF Full System
Number Cores/Chip	4	multicore
Number Processor Cores	9,600	>200,000
Amount Memory (TB)	14.4	>800
Amount Disk Storage (TB)	100	>10,000
Amount of Archival Storage (PB)	5	>500
External Bandwidth (Gbps)	40	>100

Petascale Computing Facility

Partners

- EYP
- MCF/ Gensler
- IBM
- Yahoo!

- **Modern Data Center**
  - 90,000+ ft^2 total
  - 20,000 ft^2 machine room
- **Energy Efficiency**
  - LEED Silver certified (maybe gold)
  - Efficient cooling system

Could there be applications at 1,000,000 cores?

- The answer is clearly yes - a sequence of reports, including SciDAC, DOE Exascale, and others have shown that there is a need of computing at the scale that will require (with our current understanding of the technology) 1,000,000 cores.
- But how many applications are really ready?

www.appsmatrix.info/ for some application data

- But only “top-level” details

Will they work?

- We don't know – we don't have enough information
  - More precisely, we know *some* will work
- There is lots of anecdotal evidence that we can develop scalable codes
  - Qbox, NAMD, Nek, …
- Consider this debate challenge:
  - Defend the statement:
    - These codes will scale
  - Defend the statement:
    - These codes will not scale
  - Which side would you rather take today?

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"
- In a list of enabling technologies for mathematical software, "Parallel prefix for arbitrary user-defined associative operations should be supported. Conflicts between system and library (e.g., in message types) should be automatically avoided."
  - Note that MPI-1 provided both
- 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"

Programming For Petascale Systems

- Lets look at where we are and where we could be
  - MPI
    - Reasons for its success and how to replace MPI
  - Petsc
    - Abstraction as a key tool
  - Single node performance
    - The elephant in the living room
  - Hybrid programming models
    - A first step toward a more productive software model

Why Was MPI Successful?

- It address all of the following issues:
  - Portability
  - Performance
  - Simplicity and Symmetry
  - Modularity
  - Composability
  - Completeness

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.
  - ♦ More on “Greatest” versus “Least” later ...

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

Symmetry

- Exceptions are hard on users
    - ♦ But easy on implementers — less to implement and test
- Example: MPI_Issend
    - ♦ MPI provides several send modes:
        - Regular, Synchronous, Receiver Ready, Buffered
    - ♦ Each send can be blocking or non-blocking
    - ♦ MPI provides all combinations (symmetry), including the “Nonblocking Synchronous Send”
        - Removing this would slightly simplify implementations
        - Now users need to remember which routines are provided, rather than only the concepts
    - ♦ It turns out he MPI_Issend is useful in building performance and correctness debugging tools for MPI programs
- Some symmetries may not be worth the cost
    - ♦ MPI cancel of send
        - Not just a complexity for the user - real cost to implementation

Modularity

- Many modern algorithms are hierarchical
    - ♦ Do not assume that all operations involve all or only one process
    - ♦ Software tools must not limit the user
- Modern software is built from components
    - ♦ MPI designed to support libraries
    - ♦ Communication contexts in MPI are an example

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
  - OpenMP is an example
    - MPI (the standard) required no changes to work with OpenMP
    - MPI Thread modes provided for performance reasons
- MPI was designed from the beginning to work within a larger collection of software tools
  - What’s needed to make MPI better? More good tools!

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

Conclusions: Lessons From MPI

- A successful parallel programming model must enable more than the simple problems
  - It is nice that those are easy, but those weren’t that hard to begin with
- Scalability is essential
  - Why bother with limited parallelism?
  - Just wait a few months for the next generation of hardware
- Performance is equally important
  - But not at the cost of the other items
- It must also fit into the Software Ecosystem
  - MPI did not replace the languages
  - MPI did not dictate particular process or resource management
  - MPI defined a way to build tools by replacing MPI calls
  - (later) Other interfaces, such as debugging interface, also let MPI interoperate with other tools

Issues that are not Issues

- Latency
  - Users often confuse Memory access times and CPU times; expect to see remote memory access times on the order of register access
  - Without overlapped access, a single memory reference is 100’s to 1000’s of cycles
  - A load-store model for reasoning about program performance isn’t enough
    - Don’t forget memory consistency issues
- MPI “Buffers” as a scalability limit
  - This is an implementation issue that existing MPI implementations for large scale systems already address
    - Buffers do not need to be preallocated

Fault Tolerance (As an MPI Problem)

- Fault Tolerance is a property of the application; there is no magic solution
- MPI implementations can support fault tolerance
- MPI intended implementations to continue through faults when possible
  - ♦ That's why there is a sophisticated error reporting mechanism
  - ♦ What is needed is a higher standard of MPI implementation, not a change to the MPI standard
- But - Some algorithms do need a more convenient way to manage a collection of processes that may change dynamically

---

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 only require small changes in the code
    - Rarely a property of "high productivity" languages
      - Abstractions that make easy programs easier don't solve the problem
  - ♦ Latency hiding is not the same as low latency
    - Need "Support for aggregate operations on large collections"

Even Harder Challenges

- Make it hard to write incorrect programs.
  - ♦ 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 efforts for shared memory/multicore programming are also addressing this issue

---

What's the Real Issue with MPI?

- MPI does not address the management of distributed data structures
  - ♦ Not that it does badly; it is an orthogonal issue
- Languages that provide support for distributed data structures have productivity advantages for those data structures
  - ♦ What if you don't have that sort of data structure?
- This does not mean that we can't significantly improve on MPI, but we must not reduce the space of algorithms and programs by reducing the available data structures
  - ♦ Alternatives include building tools to support domain-specific (distributed) data structures, exploiting advances in compiler and source-to-source transformation infrastructure, extending existing languages
  - ♦ Languages are also including more general support, but the general distribution/decomposition problem is extremely difficult

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
  - 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

Is MPI the Least Common Denominator Approach?

- "Least common denominator"
  - Not the correct term
  - It is "Greatest Common Denominator"! (Ask any Mathematician)
  - This is critical, because it changes the way you make improvements
- If it is "Least" then improvements can be made by picking a better approach. I.e., anything better than "the least".
- If it is "Greatest" then improvements require changing the rules: either the available architectural support ("Denominator"), the scope ("Common"), or the goals (how "Greatest" is evaluated)
- Where can we change the rules for MPI?

Changing the Common

- Give up on ubiquity/portability and aim for a subset of architectures
  - Vector computing was an example (and a cautionary tale)
  - Possible niches include
    - SMT for latency hiding
    - Reconfigurable computing; FPGA
    - Stream processors
    - GPUs
    - Etc.
- Not necessarily a bad thing (if you are willing to accept being on the fringe)
  - Risk: Keeping up with the commodity curve (remember vectors)
  - Is GPGPU the fringe or the emerging commodity processor?
    - And GPGPUs might only change the node programming model

Changing the Denominator

- This means changing the features that are assumed present in every system on which the programming model must run
- Some changes since MPI was designed:
  - RDMA Networks
    - Best for bulk transfers
    - Evolution of these may provide useful signaling for shorter transfers
  - Cache-coherent SMPs (more precisely, lack of many non-cache-coherent SMP nodes)
  - Exponentially increasing gap between memory and CPU performance
  - Better support for source-to-source transformation
    - Enables practical language solutions
- If DARPA HPCS is successful at changing the “base” HPC systems, we may also see
  - Remote load/store, remote simple ops
  - Hardware support for hiding memory latency

Changing the Goals

- Change the space of features
  - ♦ That is, change the problem definition so that there is room to expand (or contract) the meaning of “greatest”
- Some possibilities
  - ♦ Integrated support for concurrent activities
    - Not threads:
      - – “Night of the Living Threads”, http://weblogs.mozillazine.org/roc/archives/2005/12/night_of_the_living_threads.html, 2005
      - – “Why Threads Are A Bad Idea (for most purposes)” John Ousterhout (~2004)
  - ♦ Support for (specialized or general) distributed data structures

Issues for MPI in the Trans-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 detection and 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)

Abstraction in Programming

- Must get away from requiring the management of each detail
- More software will / should be built based on capabilities
  - ♦ Virtualization – abstracts processor resources
    - Provides a powerful tool for load balancing, fault handling
  - ♦ Routines organized by function rather than data structure and/or algorithm provide greater flexibility
    - A different solution to the “multicore”/parallel programming problem
    - An example is another project I’ve had the pleasure to start …

What is PETSc?

- PETSc is a numerical library
  - ♦ Organized around mathematical concepts needed to solve PDEs
- PETSc began as a tool to aid in research into domain decomposition methods for PDEs.
  - ♦ A new library was needed because
    - Numerical libraries organized around particular algorithms, rather than mathematical problems, making experimentation with different algorithms difficult
    - Most libraries were not re-entrant, making recursive use impossible
- PETSc is now used by both applications scientists and researchers (100’s of users including DOE and NSF leadership computing platforms)

What Advantage Does This Approach Give You?

- Example: A Poisson Solver in PETSc
  - ♦ The following slides show the core of a <u>complete </u>2-d Poisson solver in PETSc. Features of this solver:
    - Fully parallel
    - 2-d decomposition of the 2-d mesh
    - Linear system described as a sparse matrix; user can select many different sparse data structures
    - Linear system solved with any user-selected Krylov iterative method and preconditioner provided by PETSc, including GMRES with ILU, BiCGstab with Additive Schwarz, etc.
    - Complete performance analysis built-in
  - ♦ The full example is only 7 slides of code!

Solve a Poisson Problem with Preconditioned GMRES

```
#include <math.h>
#include "petscsles.h"
#include "petscda.h"
int main( int argc, char *argv[] )
{
   SLES      sles;   Mat       A;   Vec      b, x;   DA       grid;
   int       its, n, px, py, worldSize;
   PetscInitialize( &argc, &argv, 0, 0 );
…
   DACreate2d( PETSC_COMM_WORLD, DA_NONPERIODIC, DA_STENCIL_STAR,
                              n, n, px, py, 1, 1, 0, 0, &grid );
   A = FormLaplacianDA2d( grid, n );
   b = FormVecFromFunctionDA2d( grid, n, func );
   VecDuplicate( b, &x );
   SLESCreate( PETSC_COMM_WORLD, &sles );
   SLESSetOperators( sles, A, A, DIFFERENT_NONZERO_PATTERN );
   SLESSetFromOptions( sles );
   SLESSolve( sles, b, x, &its );
   PetscPrintf( PETSC_COMM_WORLD, "Solution is:\n" );
   VecView( x, PETSC_VIEWER_STDOUT_WORLD );
   PetscPrintf( PETSC_COMM_WORLD, "Required %d iterations\n", its );
…
   PetscFinalize( );
   }
```

Define a distributed data structure

PETSc provides routines that solve systems of sparse linear (and nonlinear) equations

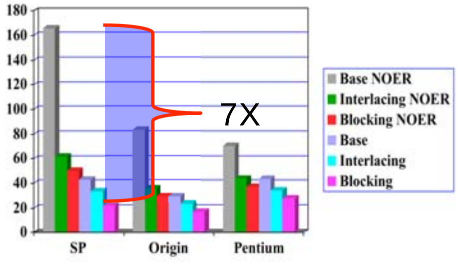
PETSc provides coordinated I/O (behavior is as-if a single process), including the output of the distributed “vec” object

Why Was PETSc a Success?

- The success of PETSc is due to:
  - ♦ Performance and Scalability
    - Performance is only weakly correlated with FLOPS
  - ♦ Consistent interface based on the mathematical problems
  - ♦ Completeness
    - Can overcome "ease of use"
  - ♦ Attention to portability and configuration issues
    - Often the critical factor
    - Portability requires care but isn't hard.
- A key advantage to the PETSc approach
  - ♦ Algorithm Independence
    - Until we know the best way, don't make the choice
    - Users can try new algorithms without giving up the ones with which they are comfortable
- Note that PETSc succeeded for many of the same reasons as MPI!

Addressing Single Node Performance

- Single node and single thread performance remains a major challenge
- The "low fraction of peak performance of parallel computers is really just the poor single core performance
- We need to extend "compilation" to involve 3rd-party, specialized software
  - ♦ Autotuners
  - ♦ Domain-specific languages
  - ♦ Composable Language extensions through annotations



- Common theme: Build *interoperable* components
- Which brings us to *hybrid programming models*

Myths About the Hybrid Model

1. Never works
    - ♦ Examples from FEM assembly, others show benefit
2. Always works
    - ♦ Examples from NAS, EarthSim, others show MPI everywhere often as fast as hybrid models
3. Requires special MPI
    - ♦ In many cases does not; in others, requires a level defined in MPI-2
4. Harder to program
    - ♦ Harder than what?
    - ♦ Really the classic solution to complexity - divide problem into separate problems
        - 10000-fold coarse-grain parallelism + 100-fold fine-grain parallelism gives 1,000,000-fold total parallelism

Where Do OpenMP + MPI Work Well?

- Compute-Bound Loops
    - ♦ This can happen in some kinds of matrix assembly, for example.
- Fine-grain parallelism
    - ♦ E.g., in blocked preconditioners, where fewer, larger blocks, each managed with OpenMP, as opposed to more, smaller, single-threaded blocks in the all-MPI version, gives you an algorithmic advantage (e.g., fewer iterations).
- Load Balancing
    - ♦ Where the computational load isn't exactly the same in all threads/processes; this can be viewed as a variation on fine-grained access.
- Memory bound loops
    - ♦ Where read data is shared, so that cache memory can be used more efficiently.

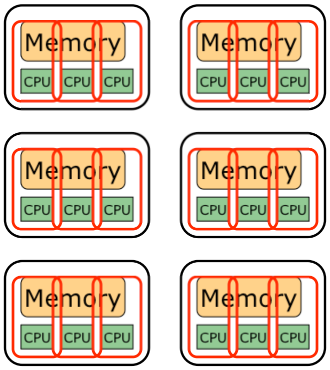
New Programming Models

- We can look at more than just MPI + OpenMP
- PGAS languages offer another tool for building parallel components
- UPC/CAF/MPI interoperability
  - Provides a way to incrementally exploit new programming models
  - Using “local” data items
- Why PGAS?
  - Load-store model may permit more efficient communication of small data items
  - Using many smaller tasks can improve scalability
    - Adaptive load balancing (move tasks around as necessary)
  - May be able to overlap communication and computation more effectively

More General MPI Hybrid Programming Models

- Why consider the Hybrid Model with PGAS or other programming models?
  - Load balancing
  - Shared data (reduce memory pressure, particularly for processor-rich (and hence memory poor) nodes)
  - Component software (use the best programming model to implement a component)
  - OpenMP and MPI understood
  - What about others: MPI/UPC (or PGAS) interoperability
- Possible combinations for MPI and UPC (or other PGAS) languages include:
  - MPI processes are UPC programs
  - MPI processes are UPC threads
  - UPC Programs are combined into MPI programs

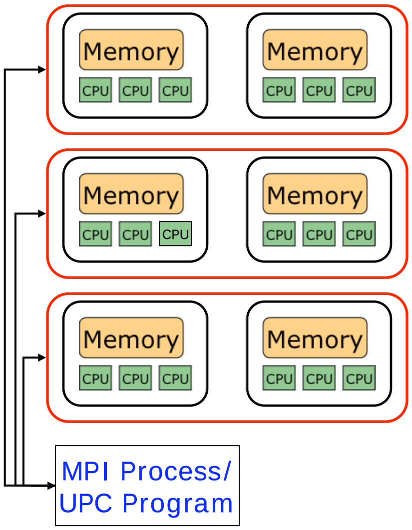
MPI Processes are UPC Threads



- The program starts as a single UPC program. Each UPC thread calls MPI_Init (or MPI_Init_thread). The process management system must permit UPC programs to use MPI_Init to also become MPI programs.
- The program starts as a single MPI program (started with mpiexec). UPC is initialized somehow
  - UPC initialized explicitly with a routine call
  - UPC initialized implicitly because UPC compiler knew this was an MPI + UPC program

MPI Processes are UPC Programs

- MPI Processes are UPC programs (not threads), spanning multiple nodes. This model is the closest counterpart to the MPI+OpenMP model, using PGAS to extend the "process" beyond a single node. (An MPI process need not be an OS process).



MPI Process/
UPC Program

Component-Oriented Software Solutions

- Hybrid programming models exploit complementary strengths
- Evolutionary Path to Hybrid Models
  - Short term - better support for resource sharing
    - We need to experiment with specifying additional information, e.g., through mpiexec
  - Medium term - better support for interoperating components
    - We need to ensure that communication infrastructures can cooperate
    - Consider extensions to make implementations aware that they are in a hybrid model program
  - Long term - Generalized model, efficient sharing of communication and computation infrastructure
- Other approaches also build on software components

We are in this Together

- Support community activities
  - MPI Forum
  - Software consortiums, BOFs, …
  - Come to SC09 in Portland!
- Build collaborations
  - At Illinois, we have many parallel computing activities
    - NSCA (Blue Waters), UPCRC (Multicore Programming), Cloud Computing, IACAT, Research in CS and ECE departments, …
  - Parallel@Illinois (www.parallel.illinois.edu Booth 2040)
    - Serves as an umbrella for Illinois efforts
- Many other efforts around the world
    - Great Lakes Consortium for Petascale Computation
- By developing approaches and tools that can interoperate, we can address the daunting problem of programming trans-petaflop and exeflop systems

Conclusions

- We have strategies that have and will serve us well
  - ♦ Proper use of abstraction
  - ♦ Better use of components
    - Specialized compilation and tuning tools
    - Domain-specific languages
    - New(er) languages that can interoperate with existing codes
- Community efforts are critical
  - ♦ MPI Forum (meetings.mpi-forum.org)
  - ♦ Open Software Consortiums (stay tuned)

Thanks!

- Thanks to *you* for your attention
- Thanks to my many co-workers and collaborators
- Thanks to the Department of Energy, the National Science Foundation, and the HDF Group for their support