Argonne NATIONAL LABORATORY

... for a brighter future

U.S. Department of Energy

UChicago ▶ Argonne LLC

Office of Science
U.S. DEPARTMENT OF ENERGY

A U.S. Department of Energy laboratory managed by UChicago Argonne, LLC

MPI and High Productivity Programming

William Gropp
Argonne National Laboratory
www.mcs.anl.gov/~gropp

Dedicated to Ken Kennedy

A gentleman and a scholar.

His many contributions to computing included starting the process that led to the MPI standard.

In addition to his outstanding intellectual contributions to computing, he was generous, open, and funny. He will be missed.

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"**

Quotes from "Enabling Technologies for Petaflops Computing" (1995):

- **"The software for the current generation of 100 GF machines is not adequate to be scaled to a TF…"**
- **"The Petaflops computer is achievable at reasonable cost with technology available in about 20 years [2014]."**
  - (estimated clock speed in 2004 — 700MHz)*
- **"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."**
- **"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)**

MPI is a *Success*

- Applications
  - Most recent Gordon Bell prize winners use MPI
- Libraries
  - Growing collection of powerful software components
- Tools
  - Performance tracing (Vampir, Jumpshot, etc.)
  - Debugging (Totalview, etc.)
- Results
  - Papers and science: http://www.mcs.anl.gov/mpi/papers
- Implementations
  - Multiple, high-quality implementations
- Beowulf
  - Ubiquitous parallel computing

But "MPI is the Problem"

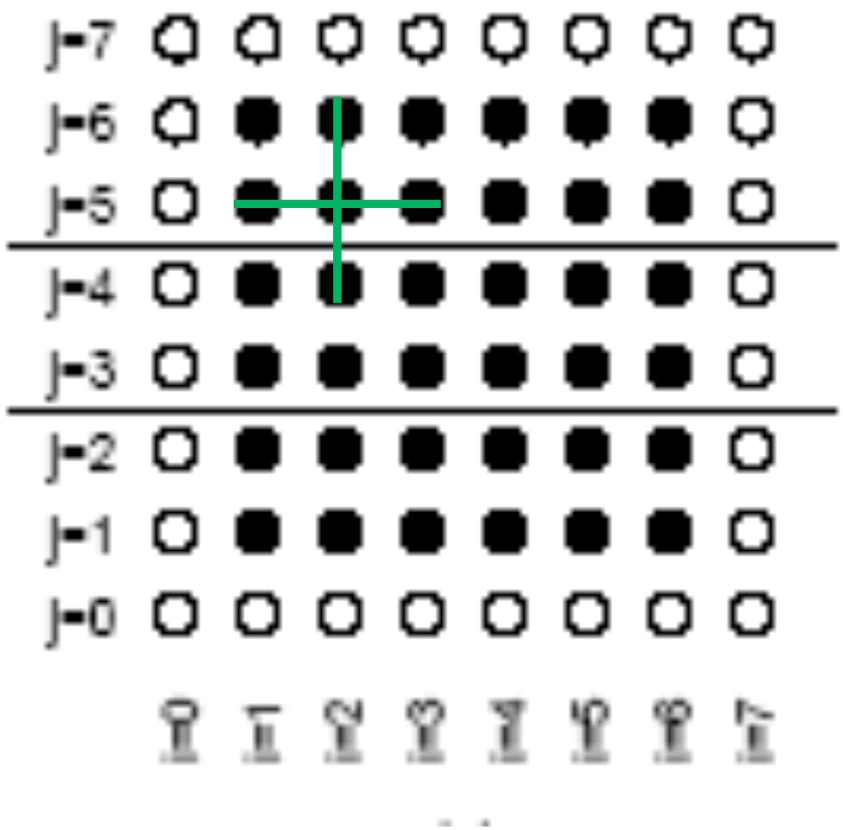
- Many people feel that programming with MPI is too hard
  - And they can prove it
- Others believe that MPI is fine
  - And they can prove it

Consider These Five Examples

- Three Mesh Problems
  - Regular mesh
  - Irregular mesh
  - C-mesh
- Indirect access
- Broadcast of data to all processes

Regular Mesh Codes

- Classic example of what is wrong with MPI
  - Some examples follow, taken from *CRPC Parallel Computing Handbook* and ZPL web site, of mesh sweeps
- The algorithm requires the averaging of nearby values, shown by the green cross (the *stencil* of the computation)



Uniprocessor Sweep

```
do k=1, maxiter
    do j=1, n-1
        do i=1, n-1
            unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                                  u(i,j+1) + u(i,j-1) - &
                                      h * h * f(i,j) )
        enddo
    enddo
    u = unew
enddo
```

MPI Sweep

```
do k=1, maxiter
   ! Send down, recv up
   call MPI_Sendrecv( u(1,js), n-1, MPI_REAL, nbr_down, k &
          u(1,je+1), n-1, MPI_REAL, nbr_up, k, &
          MPI_COMM_WORLD, status, ierr )
   ! Send up, recv down
   call MPI_Sendrecv( u(1,je), n-1, MPI_REAL, nbr_up, k+1, &
          u(1,js-1), n-1, MPI_REAL, nbr_down, k+1,&
          MPI_COMM_WORLD, status, ierr )
   do j=js, je
      do i=1, n-1
         unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + u(i,j+1) + u(i,j-1) - &
                     h * h * f(i,j) )
      enddo
    enddo
   u = unew
enddo
```

And the more scalable 2-d decomposition is even messier

HPF Sweep

```
!HPF$ DISTRIBUTE u(:,BLOCK)
!HPF$ ALIGN unew WITH u
!HPF$ ALIGN f WITH u
do k=1, maxiter
   unew(1:n-1,1:n-1) = 0.25 * &
          ( u(2:n,1:n-1) + u(0:n-2,1:n-1) + &
          u(1:n-1,2:n) + u(1:n-1,0:n-2) - &
          h * h * f(1:n-1,1:n-1) )
   u = unew
enddo
```

OpenMP Sweep

```
!$omp parallel
!$omp do
   do j=1, n-1
         do i=1, n-1
            unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                          u(i,j+1) + u(i,j-1) - &
                          h * h * f(i,j) )
         enddo
   enddo
!$omp enddo
```

ZPL Sweep

```
region
   R = [ 0..n+1,0..n+1];
direction
   N=[-1,0]; S = [1,0]; W=[0,-1];  E=[0,1];
Var
   u : [BigR] real;
[R] repeat
   u:=0.25*(u@n + u@e + u@s + u@w)-h*h*f;
Until (…convergence…);
```

(Roughly, since I'm not a ZPL programmer)

Lessons

- Strengths of non-MPI solutions
  - Data decomposition done for the programmer
  - No “action at a distance”
- So why does anyone use MPI?
  - Performance
  - Completeness
  - Ubiquity
    - *Does your laptop have MPI on it? Why not?*
- But more than that…

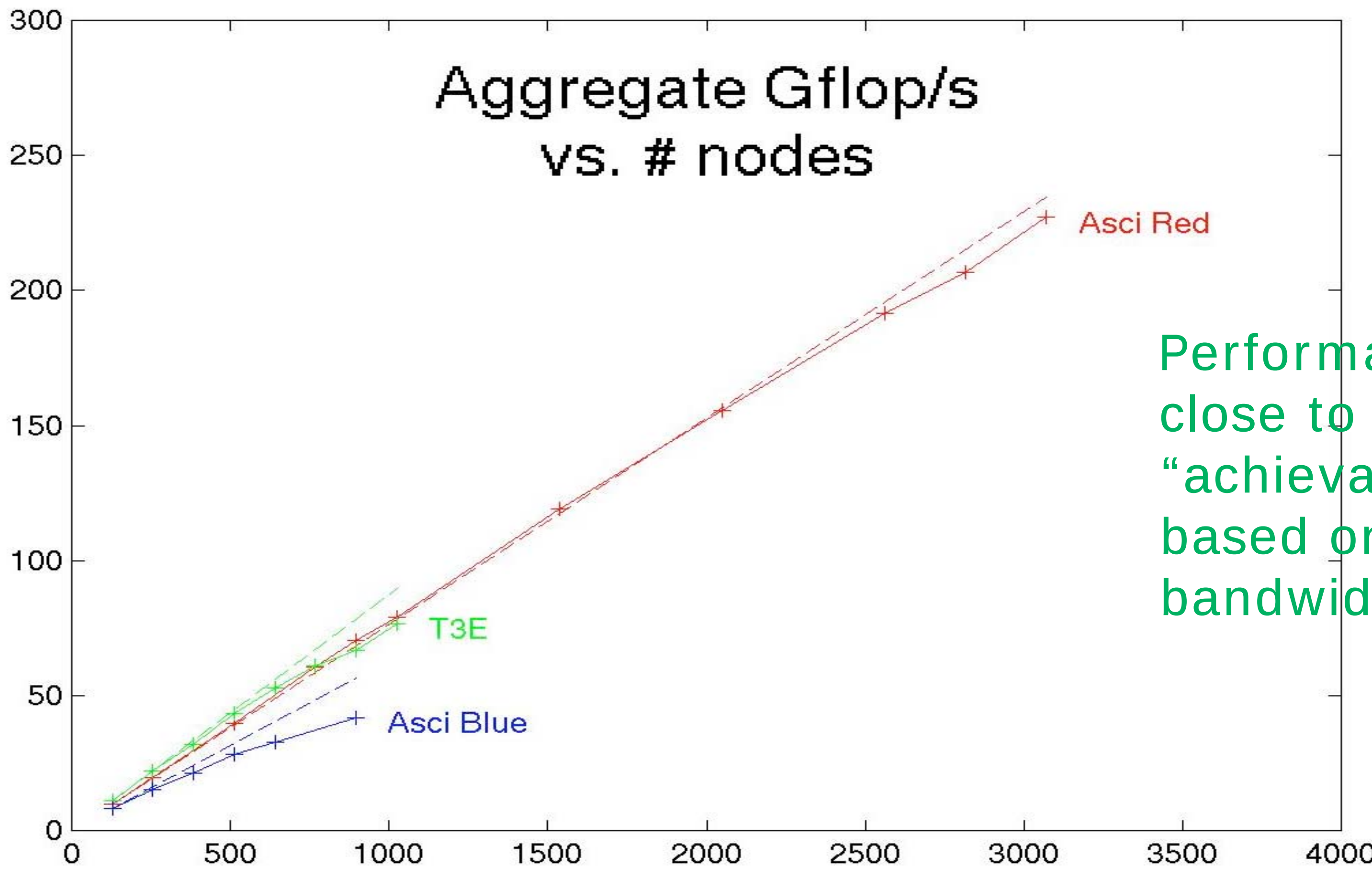
Not All Codes Are Completely Regular

- Examples:
  - Adaptive Mesh refinement
    - *How does one process know what data to access on another process?*
      - Particularly as mesh points are dynamically allocated
    - *(You could argue for fine-grain shared/distributed memory, but performance cost is an unsolved problem in general)*
    - *Libraries exist (in MPI), e.g., Chombo, KeLP (and successors)*
  - Unstructured mesh codes
    - *More challenging to write in any language*
    - *Support for abstractions like index sets can help, but only a little*
    - *MPI codes are successful here …*

FUN3d Characteristics

- Tetrahedral vertex-centered unstructured grid code developed by W. K. Anderson (NASA LaRC) for steady compressible and incompressible Euler and Navier-Stokes equations (with one-equation turbulence modeling)
- Won Gordon Bell Prize in 1999
- Uses MPI for parallelism
- Application contains ***ZERO*** explicit lines of MPI
  - All MPI within the PETSc library

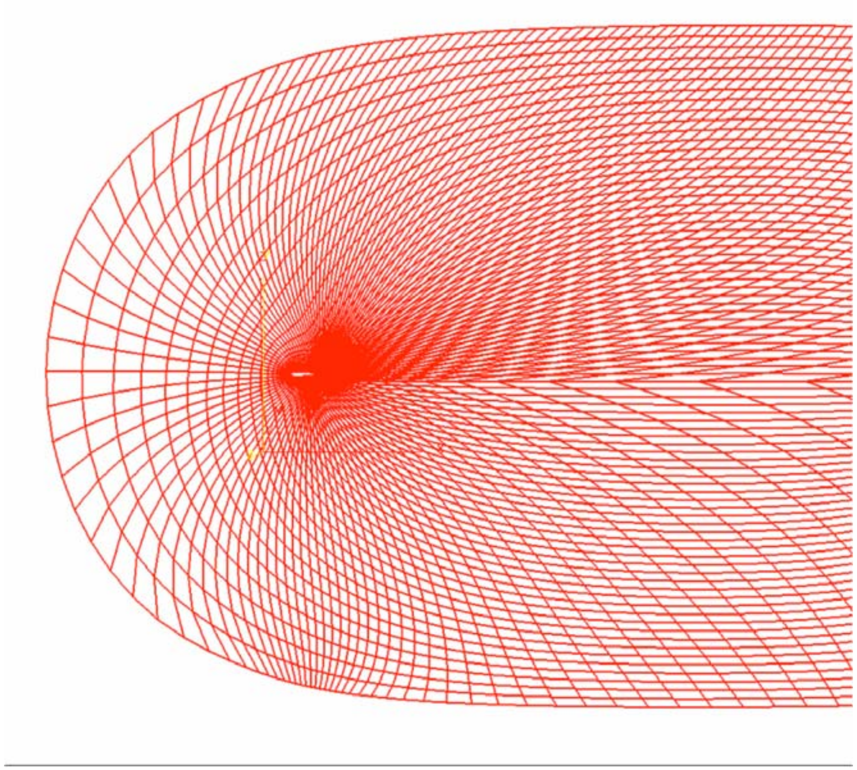
Fun3d Performance



Performance close to "achievable peak" based on memory bandwidth

Another Example: Regular Grids—But With a Twist

- "C Grids" common for certain geometries
- Communication pattern is regular but not part of "mesh" or "matrix" oriented languages
  - |i-n/2|>L, use one rule, otherwise, use a different rule
  - No longer transparent in HPF or ZPL
  - Convenience features are <u>*brittle*</u>
    - *Great when they match what you want*
    - *But frustrating when they don't*



Irregular Access

- For j=1, zillion
  table[f(j)] ^= intable[f(j)]
- Table, intable are “global” arrays (distributed across all processes)
- Seems simple enough
  – ^ is XOR, which is associative and commutative, so order of evaluation is irrelevant
- Core of the GUPS (also called TableToy) example
  – Two version: MPI and shared memory
  – MPI code is much more complicated

But…

- MPI version produces the same answer every time
- Shared/Distributed memory version *does not*
  - Race conditions are present (did you spot them?)
  - Benchmark is from a problem domain where getting the same answer every time is not required
  - Scientific simulation often does not have this luxury
- You *can* make the shared memory version produce the same answer every time, but
  - You either need fine-grain locking
    - *In software, costly in time, may reduce effective parallelism*
    - *In hardware, with sophisticated remote atomic operations (such as a remote compare and swap), but costly in €/£/¥/$/Ft/…*
    - *(Transactional memory may address some of these issues)*
  - Or you can aggregate operations
    - *Code starts looking like MPI version (if you have to do it by hand)…*

Broadcast And Allreduce

- Simple in MPI:
  - MPI_Bcast, MPI_Allreduce
- Simple in shared memory (?)
  - ```
    do i=1,n
       a(i) = b(i)   ! B (shared) broadcast to A
    enddo
    ```
  - ```
    do i=1,n
       sum = sum + A(i)   ! A (shared) reduced to sum
    enddo
    ```
- But wait — how well would those perform?
  - Poorly. Very Poorly (much published work in shared-memory literature)
  - Optimizing these operations is not easy
  - Unrealistic to expect a compiler to come up with these algorithms
  - E.g., OpenMP admits this and contains a special operation for scalar reductions (OpenMP v2 adds vector reductions)
- What can we say about the success of MPI?

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.mcs.anl.gov/~gropp/bib/papers/2001/mpi-lessons.pdf

Portability

- Hardware changes (and improves) frequently
  - Moving from system to system is often the fastest route to higher performance
  - Lifetime of an application (typically 5-20 years) greatly exceeds any hardware (3 years)
- Non-portable solutions trap the application
  - Short-term gain is not worth the long term cost

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.*
    - *Have a suggestion that meets the requirements? Lets talk!*
  - More on "Greatest" versus "Least" at the end of this talk…

Simplicity and Symmetry

- MPI is organized around a small number of concepts
  - The number of routines is not a good measure of complexity
  - 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

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
  - Communication contexts in MPI are an example
    - *Other features, such as communicator attributes, were less successful features*

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) requried no changes to work with OpenMP*

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

Is Ease of Use the Overriding *Goal?*

- MPI often described as “the assembly language of parallel programming”
- C and Fortran have been described as “portable assembly languages”
  - (That’s company MPI is proud to keep)
- Ease of use is important. But *completeness* is more important.
  - Don’t force users to switch to a different approach as their application evolves
    - *Remember the mesh examples*

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

More Lessons

- A general programming model for high-performance technical computing must address many issues to succeed, including:
- Completeness
  - Support the evolution of applications
- Simplicity
  - Focus on users not implementors
  - Symmetry reduces the burden on users
- Portability rides the hardware wave
  - Sacrifice only if the advantage is huge and persistent
  - Competitive performance and elegant design is not enough
- Composability rides the software wave
  - Leverage improvements in compilers, runtimes, algorithms
  - Matches hierarchical nature of systems
- Even that is not enough. Also need:
  - Good design
  - Buy-in by the community
  - Effective implementations
- MPI achieved these through an Open Standards Process

Improving Parallel Programming

- How can we make the programming of real applications easier?
- Problems with the Message-Passing Model
  - User's responsibility for data decomposition
  - "Action at a distance"
    - *Matching sends and receives*
    - *Remote memory access*
  - Performance costs of a library (no compile-time optimizations)
  - Need to choose a particular set of calls to match the hardware
- In summary, the lack of abstractions that match the applications

Challenges

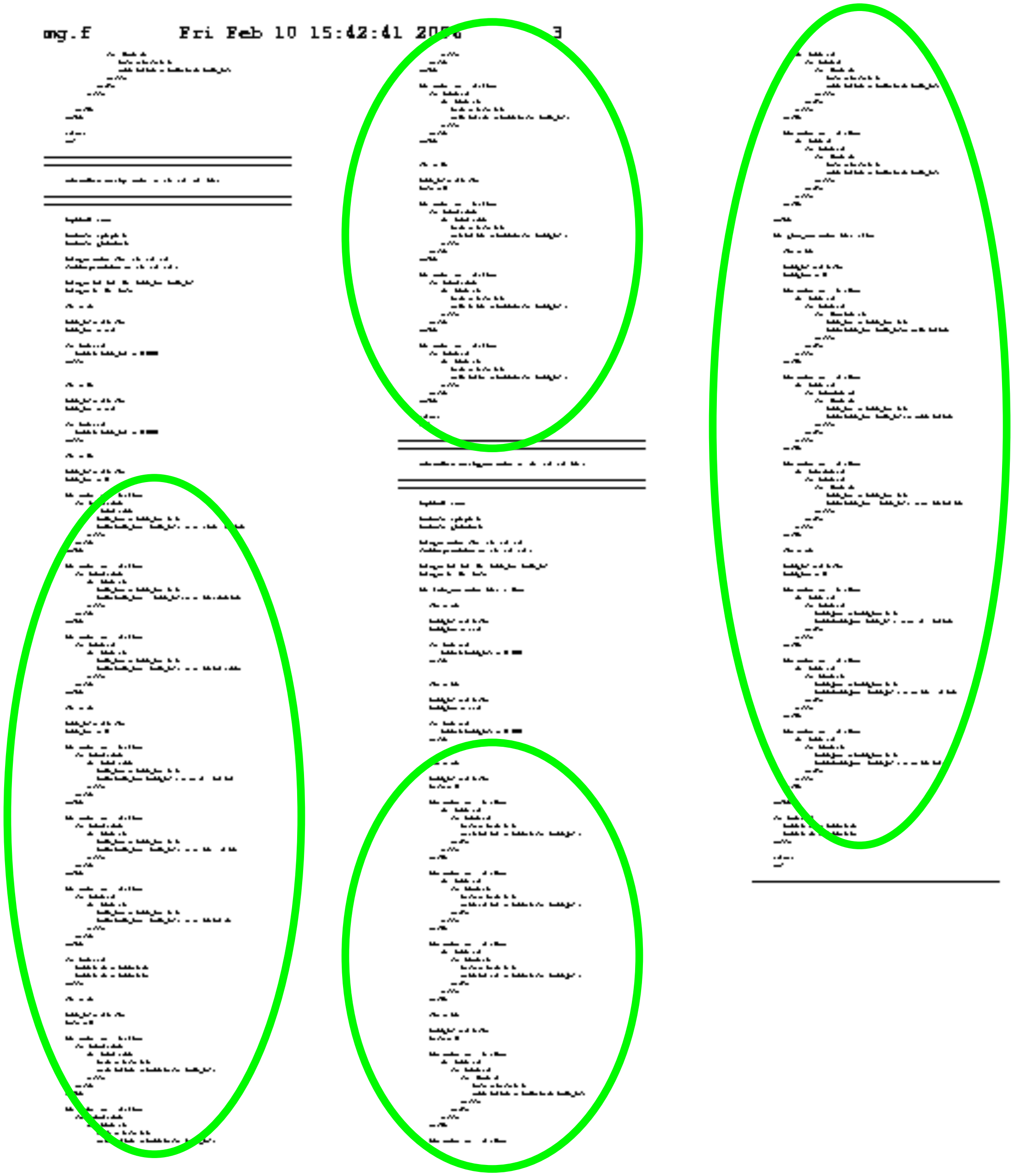
- Must avoid the traps:
  - The challenge is not to make easy programs easier. The challenge is to make hard programs possible.
  - We need a "well-posedness" concept for programming tasks
    - *Small changes in the requirements should only require small changes in the code*
    - *Rarely a property of "high productivity" languages*
      - Abstractions that make easy programs easier don't solve the problem
  - Evaluating a specific MPI implementation is not the same as evaluating MPI the programming model
  - 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

What is Needed To Achieve Real High Productivity Programming

- Simplify the construction of correct, high-performance applications
- Managing Data Decompositions
  - Necessary for both parallel and uniprocessor applications
  - Many levels must be managed
  - Strong dependence on problem domain (e.g., halos, load-balanced decompositions, dynamic vs. static)
- Possible approaches include
  - Language-based
    - *Limited by predefined decompositions*
      - Some are more powerful than others; Divacon provided a built-in divided and conquer
  - Library-based
    - *Overhead of library (incl. lack of compile-time optimizations), tradeoffs between number of routines, performance, and generality*
  - Domain-specific languages
    - *Example: mesh handling*
      - Standard rules can define mesh
      - Alternate mappings easily applied (e.g., Morton orderings)
      - Careful source-to-source methods can preserve human-readable code
      - In the longer term, debuggers could learn to handle programs built with language composition (they already handle 2 languages – assembly and C/Fortran/…)

Distributed Memory code

- Single node performance is clearly a problem.
- What about parallel performance?
  - Many successes at scale (e.g., Gordon Bell Prizes for >200TF on 64K BG/L nodes
  - Some difficulties with load-balancing, designing code and algorithms for latency, but skilled programmers and applications scientists have been remarkably successful
- Is there a 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*
  - 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

Another example…

Manual Decomposition of Data Structures

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

0	1	4	5	8	9	12	13
2	3	6	7	10	11	14	15
16	17	20	21	24	25	28	29
18	19	22	23	26	27	30	31
32	33	36	37	40	41	44	45
34	35	38	39	42	43	46	47
48	49	52	53	56	57	60	61
50	51	54	55	58	59	62	63

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

- Trick!
  - This is from a paper on dense matrix tiling for uniprocessors!
- This suggests that managing data decompositions is a common problem for real machines, whether they are parallel or not
  - *Not just an artifact of MPI-style programming*
  - Aiding programmers in data structure decomposition is an important part of solving the productivity puzzle

How to Replace (or Evolve) 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*

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 "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)

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
  - 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:*
      - See, e.g., Edward A. Lee, "The Problem with Threads," Computer, vol. 39, no. 5, pp. 33-42, May, 2006.
      - “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)
      - “If I were king: A proposal for fixing the Java programming language's threading problems” http://www-128.ibm.com/developerworks/library/j-king.html, 2000
  - Support for (specialized or general) distributed data structures

Conclusions

- MPI is a successful "Greatest Common Denominator" parallel programming model
- Properly used, MPI has enabled high(er) productivity
  - Libraries, design templates, tools
- The next success must
  - Change the rules
  - Be an developed as an open process
  - Have a clear focus on the audience

Interested in working with us?

- DOE has many opportunities for student participation:
  - Undergraduate internships
  - Graduate research aide appointments
  - See http://www.dep.anl.gov/ for details (unfortunately, the Summer 2007 deadline just past. But you there are both fall and winter term programs, and you can put January 2008 on your calendar)

Backup

Further Reading

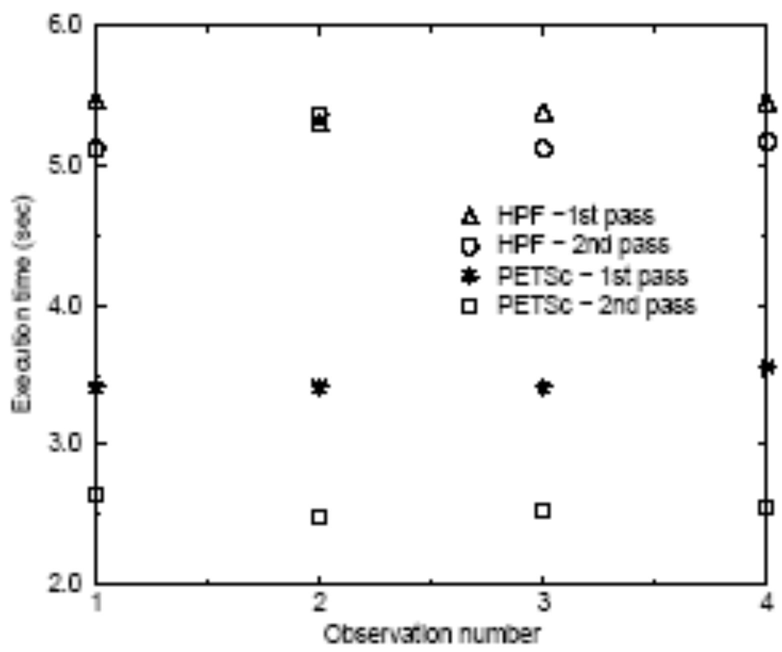
- For a historical perspective (and a reality check),
  - “*Enabling Technologies for Petaflops Computing*”, Thomas Sterling, Paul Messina, and Paul H. Smith, MIT Press, 1995
  - “*System Software and Tools for High Performance Computing Environments*”, edited by Paul Messina and Thomas Sterling, SIAM, 1993
- For recent thinking on possible directions,
  - “*Report of the Workshop on High-Productivity Programming Languages and Models*”, edited by Hans Zima, May 2004.

More Lessons

- Completeness
  - Support the evolution of applications
- Simplicity
  - Focus on *users* not implementors
  - Symmetry reduces users burden
- Portability rides the hardware wave
  - Sacrifice only if the advantage is *huge* and *persistent*
  - Competitive performance and elegant design is not enough

Why Not Always Use HPF?

- Performance!
  - From "A Comparison of PETSC Library and HPF Implementations of an Archetypal PDE Computation" (*M. Ehtesham Hayder, David E. Keyes, and Piyush Mehrotra)*
  - PETSc (Library using MPI) performance *double* HPF
- Maybe there's something to explicit management of the data decomposition…



Performance Portability

- Goal: A programming model that ensures that any program achieves best (or near best) performance on *all* hardware.
  - MPI is sometimes criticized because there are many ways to express the same operation.
- Reality: This is an unsolved problem, even for Fortran on uniprocessors. Expecting a solution for *parallel* systems is unrealistic.
  - Consider dense matrix-matrix multiplications.
  - 6 ways to order the natural loops, discussed in a famous paper
  - *None* of these is optimal (various cache blocking strategies are necessary)
  - Automated search techniques can out-perform hand-code (ATLAS)

Performance

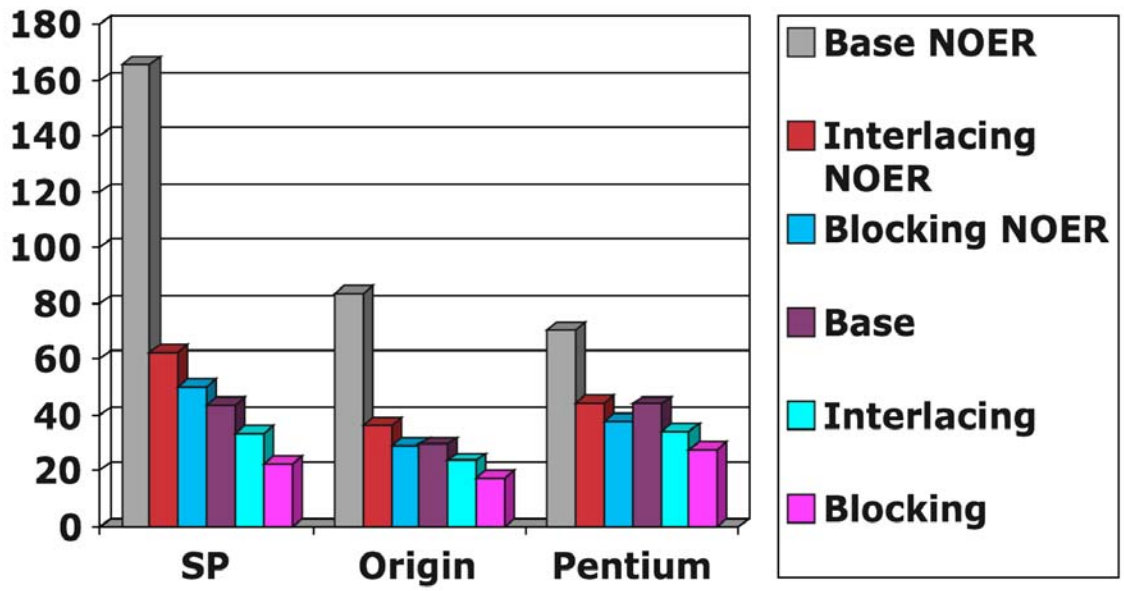
- Performance must be competitive
  - Pay attention to memory motion
  - Leave freedom for implementers to exploit any special features
    - *Standard document requires careful reading*
    - *Not all implementations are perfect*
      - (When you see MPI pingpong asymptotic bandwidths that are much below the expected performance, it is the implementation that is broken, not MPI)

Method	Bandwidth
MPI	793
Shmem	2230

These should be the same

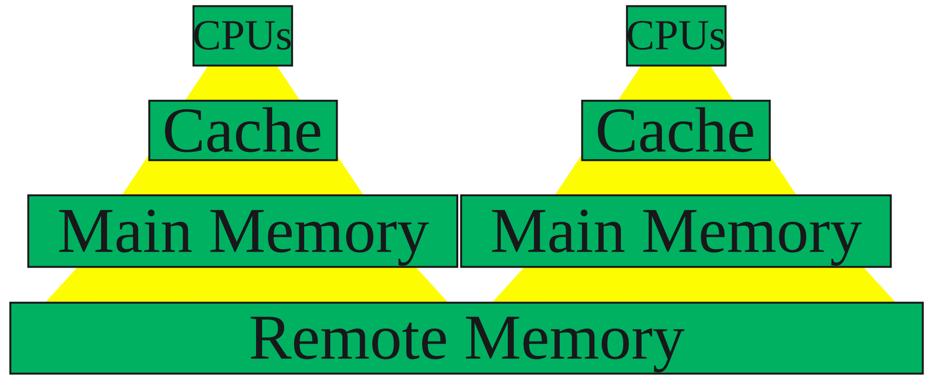
MPI's Memory Model

- Match to OS model
  - OS: Each *process* has memory whose locality is important
  - Locality for threads may not be appropriate, depending on how the thread is used.
- Not a new approach
  - register in C
  - Local and shared data in HPF, UPC, CoArray Fortran



Parallel Computing and Uniprocessor Performance

- Deeper memory hierarchy
- Synchronization/ coordination
- Load balancing



This is the hardest gap

Not this

Memory Layer	Access Time (cycles)	Relative
Register	1	1
Cache	1–10	10
DRAM Memory	1000	100
Remote Memory (with MPI)	10000	10

Measuring Complexity

- Complexity should be measured in the number of *concepts*, not functions or size of the manual
- MPI is organized around a few powerful concepts
  - Point-to-point message passing
  - Datatypes
  - Blocking and nonblocking buffer handling
  - Communication contexts and process groups

Elegance of Design

- MPI often uses one concept to solve multiple problems
- Example: Datatypes
  - Describe noncontiguous data transfers, necessary for performance
  - Describe data formats, necessary for heterogeneous systems
- "Proof" of elegance:
  - Datatypes *exactly* what is needed for high-performance I/O, added in MPI-2.