How To Replace MPI As The Scalable Programming System For Computational Science

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

Why This Talk Here?

- The history of MPI and Beowulf are closely connected
  - ♦ MPI 1 Released May 5 1994 (Forum starts 1992)
  - ♦ Beowulf late 93/early 94 (beowulf.org)
- Beowulf relied on existing, portable, high performance software for parallel programming:
  - ♦ MPI and PVM
- Large, diverse system base supported software for MPI: tools, libraries, applications

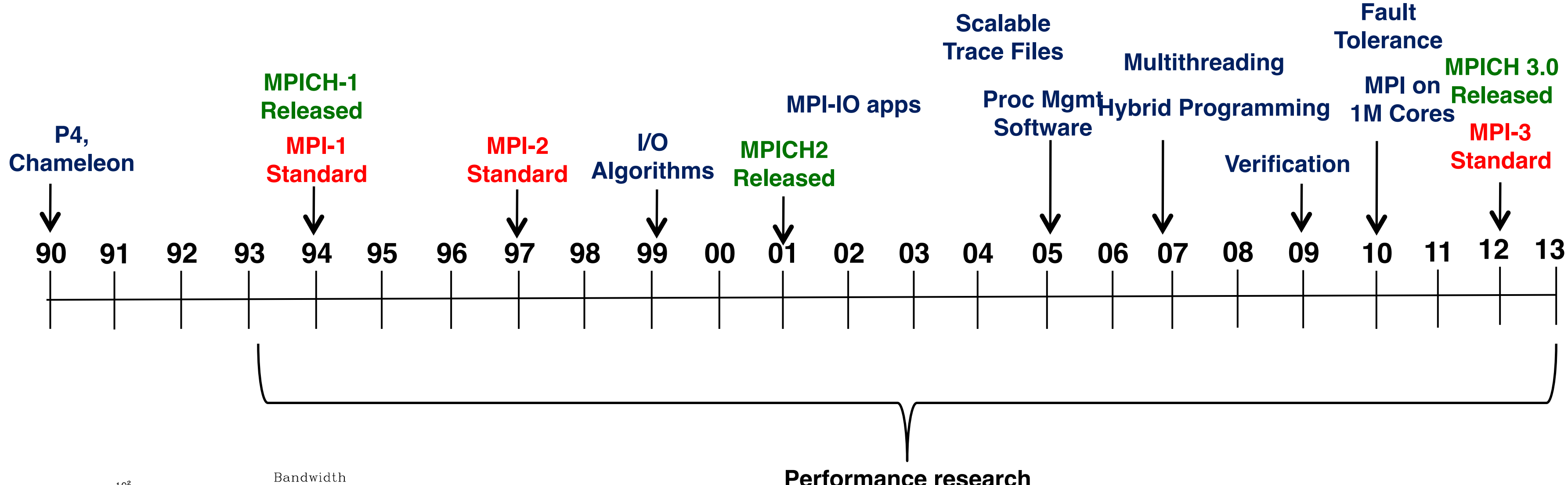
Shared History

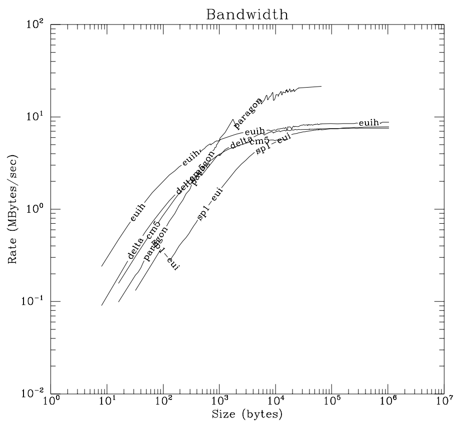
- MPI is older, but not by much
- Neither is a "least common denominator"
  - ♦ Which is a silly term; in math only GCD makes any sense
  - ♦ In fact, Beowulf and MPI are GCDs – they succeeded because they were enough to get the job done and, through "common", created a viable ecosystem for parallel apps
- Many common strengths and weaknesses (I'll get back to that)

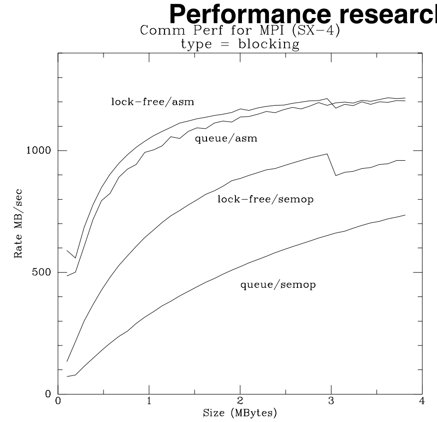
Some Definitions

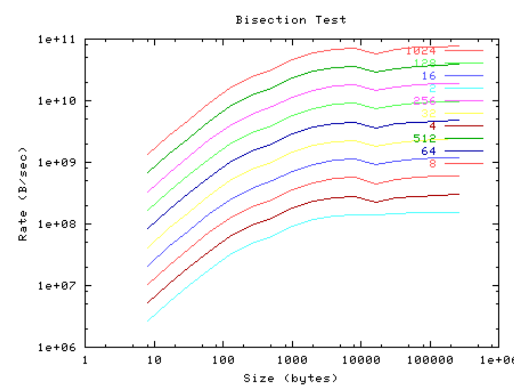
- Programming Model – Abstract approach to programming. Usually a single approach.
  - Message passing is a programming *model*
- Programming System – A realization of (parts of) one or more programming models
  - MPI is a programming *system*
- Execution Model – Abstraction of what the computer hardware (and system software) can *do*
  - Vector processing or a generic GPU are execution models
- Least Common Denominator – No such thing
  - Its *greatest common denominator*. Calling something an LCD is a tacky way of saying you don't like it
  - The distinction is important, as we'll see

MPI and MPICH Timeline



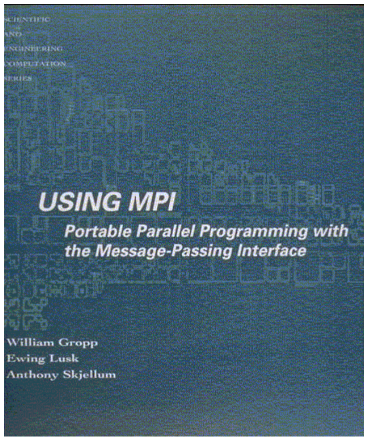


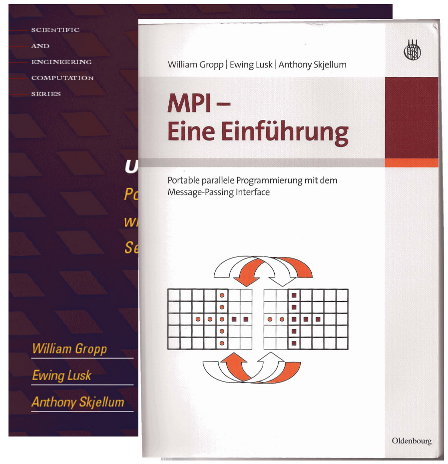


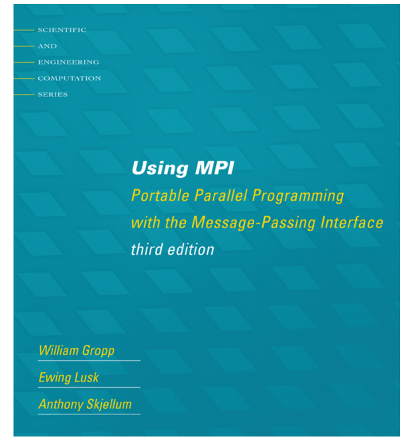


PARALLEL@ILLINOIS

Another Look at the History of MPI

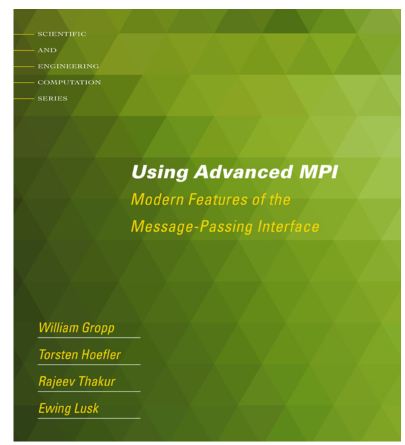






NEW!

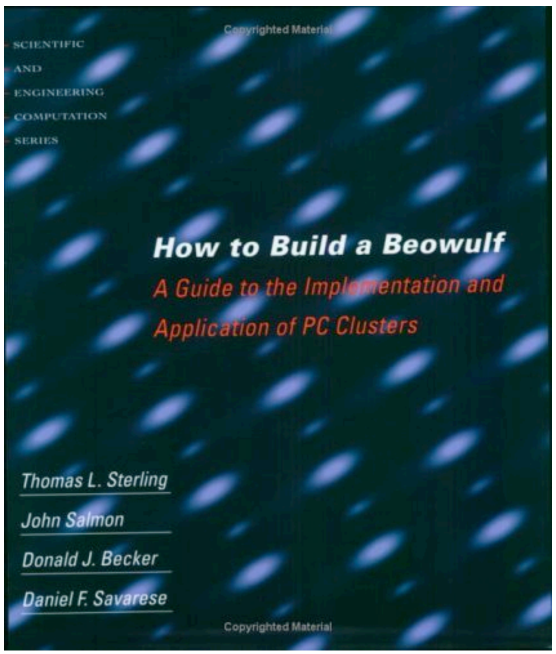
Books are important!

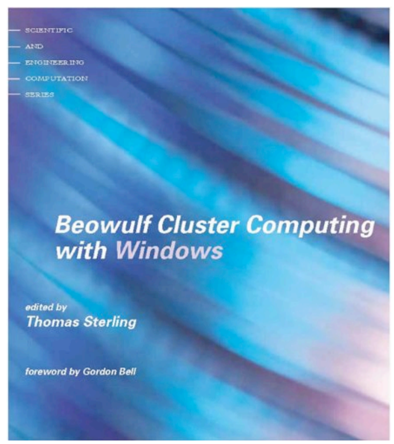


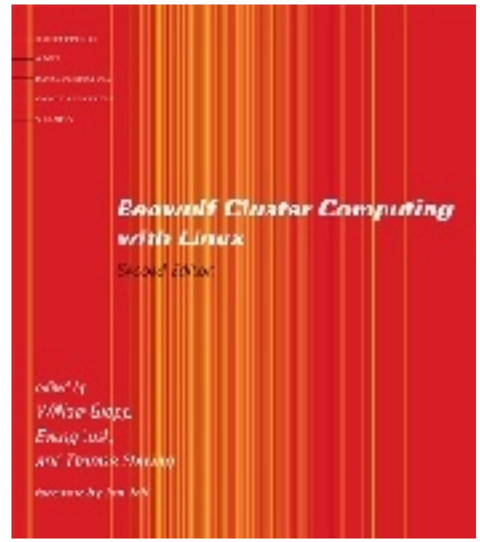
1994 1999 2014

PARALLEL@ILLINOIS

A Early Beowulf Timeline







1999 2001 2003

Why Was MPI Successful?

- It addresses 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/pdata/2001/mpi-lessons.pdf

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!

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/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
  - Example: communication contexts

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
  - ♦ OpenACC, OpenCL newer examples

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
- 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

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
  - Latency hiding is not the same as low latency
    - Need "Support for aggregate operations on large collections"

Challenges

- An even harder challenge: make it hard to write incorrect programs.
  - ♦ OpenMP is not a step in the (entirely) right direction
  - ♦ In general, most legacy 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
    - "Data race free" programming, the approach taken in Java and C++, is in this direction, and a response to the dangers in ad hoc shared memory programming

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
  - 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 …

"Domain-specific" languages

- (First – think abstract data-structure specific, not science domain)
- A possible solution, particularly when mixed with adaptable runtimes
- Exploit composition of software (e.g., work with existing compilers, don't try to duplicate/replace them)
- Example: mesh handling
  - Standard rules can define mesh
    - Including "new" meshes, such as C-grids
  - 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/…)
- Provides a single "user abstraction" whose implementation may use the composition of hierarchical models
  - Also provides a good way to integrate performance engineering into the application

Enhancing Existing Languages

- Embedded DSLs are one way to extend languages
- Annotations, coupled with code transformations is another
  - ♦ Follows the Beowulf philosophy – exploit commodity components to provide new capabilities
  - ♦ Approach taken by the Center for Exascale Simulation of Plasma-Coupled Combustion xpacc.illinois.edu

Replacing MPI/Beowulf

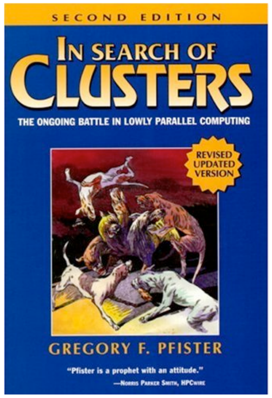
- Really? Are you sure you can do better?
  - Challenge: What needs to be *replaced* (with costs of developing new ecosystem) and what needs only be *improved* (better implemented in the context of existing systems)?
  - Many "alternatives" are working around limitations in current *implementations*, and by doing so, dilute efforts better spent on *fixing* real issues in implementations
- Lets look at the strengths and weaknesses of both

Weaknesses

- Beowulf
  - Distributed Memory. Forces decomposition of work
    - DSM notwithstanding
  - I/O. Harder to use as distributed; POSIX make performance hard to achieve (alternative it to ignore POSIX requirements, as NFS 3 did)
  - Performance code of interfaces (commodity); esp. latency
- MPI
  - Distributed Memory. No built-in support for user-distributions
  - No built-in support for dynamic execution
  - Performance cost of interfaces; overhead of calls; rigidity of choice of functionality
  - I/O is capable but hard to use
    - Way better than POSIX, but rarely implemented well

Strengths

- Beowulf
  - ♦ Commodity, ubiquity (runs everywhere)
  - ♦ Distributed memory provides scalability, reliability, bounds complexity (of hw)
  - ♦ Leverages other technologies, developed independently



- MPI
  - ♦ Ubiquity
  - ♦ Distributed memory provides scalability, reliability, bounds complexity (that MPI implementation must manage)
    - Does not stand in the way of user distributions, dynamic execution
  - ♦ Leverages other technologies (HW, compilers, incl OpenMP/ OpenACC)

If you insist: For MPI

- Add what is missing:
  - Distributed data structures (that the user needs)
    - This is what most "DSL"s really provide
  - Low overhead (node)remote operations
    - MPI-3 RMA a start, but could be lower overhead if compiled in, handled in hardware, consistent with other data transports
  - Dynamic load balancing
    - MPI-3 shared memory; MPI+X; AMPI all workable solutions but could be improved
    - Biggest change still needs to be made by applications – must abandon the part of the *execution model* that guarantees predictiable performance
  - Resource coordination with other programming systems
    - See strength – leverage is also a weakness if the parts don't work well together
  - Lower latency implementation
    - Essential to producitivity – reduces the "grain size" or degree of aggregation that the programmer must provide
    - We need to bring back $n_{1/2}$

For Beowulf…

- Tighter integration of hardware, especially CPU, Memory, and Interconnect
  - See "leverage" issues for MPI
- Better (parallel) I/O
  - POSIX is a terrible, counter-productive model
  - Need I/O that reflects DSM, consistency model required by applications
    - This is where the innovation has been in non-HPC I/O systems
- Better self-awareness
  - Fault prediction/recovery
  - Faults include performance, not just correctness
- OS better supports parallel programming models
  - E.g., thread scheduling, memory management
- *Standardized* support for collective actions
  - Many attempts: Scalable Unix Tools (1994), GLUnix (1997), etc.

Conclusions

- MPI and Beowulf have given computational science 20 years of success
- Both remain successful and relevant today and into the future
- No one feature led to their success
  - Any replacement can't just be better in one way
- Both have evolved and can continue to evolve to support science in the 21^{st} Century