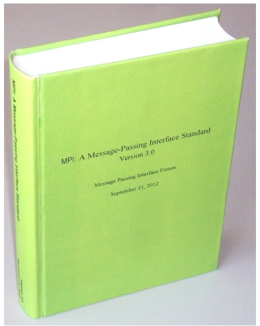
MPI 3 and Beyond: Why MPI is Successful and What Challenges it Faces

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

What? MPI-3 Already?!

- MPI Forum passed MPI-3.0 last Friday, September 21, here in Vienna
  - MPI-2.2 released September, 2009
- Standard available www.mpi-forum.org/docs
- Bound version available
- Significant enhancement from MPI-2.2
- Mostly backward compatible
  - Some previously deprecated functions removed
- Major step positioning MPI-3 for multicore, extreme scale systems



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, U Delaware, and others on formal analysis of MPI programs

MPI Built on a Strong Base

- Standard practice, sensibly extended
  - Datatypes
  - Communicator and context
- Forward looking
  - Where parallel computing was going, *not* where it had been
    - Measurements are about *past* systems
- Precise description
  - Semantics well defined
  - Not all parallel programming models so precise
- Strengths
  - Portability, Performance, Modularity and Composibility, Completeness
- Weaknesses
  - Specification as library prevents close integration with language
  - Lack of support for distributed data structures

Myths about MPI

- Some common myths:
  - MPI requires p^2 buffers
  - MPI is not fault tolerant
  - MPI does not have scalable startup
  - MPI RMA has complex rules
  - MPI requires ordering of messages in the network
- Why discuss these?
  - They still confuse discussions about MPI
  - They reveal a error in thinking about MPI

MPI requires p^2 buffers

- MPI allows any process to communicate with any other. Seems to require p (or p-1) buffers at each process to handle receipt of envelopes, eager data
- But this is an *implementation* decision
- Any *scalable* application will communicate with a fixed number (or log p if a weak scalability is used) of processes
- An *implementation* can trade (buffer) space for implementation complexity and (perhaps) time

MPI is not Fault-Tolerant

- Means "The standard (like virtually all other standards" does not mandate a specific behavior when certain kinds of faults occur
- Most who make this claim make it based on (a) the default error handler (a *very* good idea) and (b) the behavior of some implementations
- **Challenge: Should the *standard* require a fault tolerant system or should an *implementation* be tolerant of certain classes of faults?**
  - Which faults are important to you?
  - See "Fault Tolerance in MPI Programs", G & Lusk, IJHPCA 18, #3, 363-372.

MPI does not have Scalable Startup

- Startup is not part of the MPI standard, so this statement makes no sense
- Typically based on examining how some MPI implementations start
    - ♦ No need to establish all possible connections at initialization time – MPICH never did, even in 1992
    - ♦ No need to start processes sequentially
    - ♦ No need to even use OS processes for MPI
- It *is* more difficult to build a scalable startup system
    - ♦ And you have to design it to be highly scalable, not just scalable-on-the-systems-that-are-available-now

MPI RMA has Complex Rules

- True but misleading
- MPI RMA is precisely defined – much more so that many other one-sided specifications
  - The result is, unfortunately, complexity
  - Also defined to allow and encourage hardware acceleration
- However, sufficient (but not necessary) rules exist
  - These are much simpler and adequate for most uses
- A standard should never be punished for getting hard things *right*
- One-sided memory update rules are more complicated than you think (-> see "MPI and Shared Memory" later in talk)

MPI Requires Ordering of Data in the Network

- Absolutely false.
- MPI requires *apparent* ordering of *certain* operations
  - ♦ Message "envelopes" within the same communicator
  - ♦ In MPI-3, certain RMA accumulate operations (by default, can be relaxed)
- Actual delivery, particularly of data, need not be ordered
  - ♦ Only need to know for certain when all data is available
- Advantageous for high-performance networks
- An example of specifying only as much as necessary
  - ♦ Order of delivery of data up to implementation – both hardware and software

A Common Mistake

- Measurements of an implementation used to compare programming models or ideas
  - Wrong to compare C and Fortran by using measurements with a mature, highly optimizing compilers (e.g., icc) and a less mature, less capable compiler (e.g., gfortran) on a machine or even many compilers on many machines
  - Equally wrong to compare MPI and X by using implementations of MPI and X
- You can gain some insight into what *may be* (not is) hard to implement well, but that's *not a comparison*
- **Action: As reviewers, require precision in titles and descriptions.**
- **Challenge: Balance quantitative thinking about the future with experiments that can be run today.**

Challenges Facing MPI

- Why is now special?
  - End of Denard (frequency) scaling, related challenges of power consumption, heat dissipation, and reliability creating great architectural diversity
  - System scale exceeding that at which many current algorithms are effective, requiring new ideas and the programming models and ideas to support them

- Programming models are changing
  - Most popular parallel programming language in recent years…
    - CUDA
  - New HPC languages, including OpenACC, Chapel, Habanero, Python, Liszt, many DSL proposals,…
  - Perhaps biggest change since vectorization over 30 years ago

Changes in Processor Architecture

- MPI defined when a single processor often required multiple chips (including an attached floating point unit!)
- Many different architectural directions today, including
  - Multicore, Manycore, GPU, FPGA, EMP/PIM
  - Intrachip interconnects, on chip interconnects, smart NICs

MPI Processes and Processors

- MPI remains a single process programming approach
  - Relies on Fortran and C (and until MPI-3, C++) as the base languages
    - All very old, designed as single threaded; only now trying to retrofit thread safety and other forms of chip and node parallelism
  - MPI has relied on *composition of programming models*
    - Strength – can exploit advances in compiler and language abilities
    - Weakness – Unable to enlist help by compiler to optimize and detect user errors
  - Examples: Nonblocking operations and threads

Nonblocking Operations

- *Necessary* for correctness for complex communication patterns (because of difficulty in ordering sends and receives so that buffering limits cannot be exceeded)
  - Easy to order for regular grid communication
  - Hard for adaptive, irregular grid communication
- *Express Communication/Computation Overlap*
  - Both for overall time and moving operations to communication engines
- But *dangerous* for programmer – no clear correspondence in code to when a buffer is available
  - Very difficult for Fortran compiler to optimize code safely

Threads

- MPI-1 designed expecting threads to complement MPI
    - SMPs common (but multi-chip processors)
    - All nonblocking operations can be performed as a blocking operation in a separate thread
        - As long as MPI blocking operations only block thread, not process; clarified in MPI-2.0
        - Semantics inherited from thread model
    - Core communication operations considered too performance-critical
        - MPI_Isend, MPI_Send_init/MPI_Start, etc.
    - Overhead of threads became clearer as thread-safe implementations of MPI, other applications, appeared
        - Thread levels in MPI-2.0, e.g., MPI_THREAD_MULTIPLE

Best Laid Plans

- However, situation worse than appeared
  - Cost of providing threads encouraged at least one HPC vendor to restrict processes to one thread per core (for some definition of "core")
  - Makes threads *useless* as a portable method to implement nonblocking communication and computation
- Led to large and inconsistent increase in the number of nonblocking routines in MPI-3
  - E.g., many algorithms can benefit from nonblocking collective routines
  - MPI-3 added nonblocking versions of many *but not all* collective routines
    - So many that the concern was that too many were being added

Challenges

- **Handle threads consistently**
  - ♦ E.g., Assume threads (> number of cores) are present and efficient. Can be used to implement general nonblocking operations. Only core MPI 1 and 2 nonblocking routines are needed
    - MPI-3 decision: These sort of threads are not widespread enough, *and will not be in the future*, for MPI to depend upon
  - ♦ But some MPI operations, particularly RMA, *require* an "agent" to perform the operation
    - Many appear to assume that these can be done with a thread, but this is *inconsistent* with the design of MPI-3

MPI and Hybrid Models

- **Challenge: How do runtimes of different programming model implementations negotiate shared resources?**
  - E.g., how do MPI and OpenMP implementations agree to share cores, memory, interchip communication, and even threads?
- **Challenge: Is the programmer's help needed, or can this be solved without any explicit program interface?**
- These must be solved for MPI to successfully exploit composition of programming models

RDMA

- Remote Direct Memory Access
  - Networks optimized for one-sided data transfers
  - "Easy" part is the put and get for large transfers
  - Hard part (for *all* one-sided models) includes local and remote completion of transfers
    - Even published papers sometimes fail to properly ensure completion, depend on operations being "fast enough"
- Devil is in the details
  - Data delivery and ordering
  - Short operations critical for many algorithms, high productivity models
    - Fine grain models, many algorithms, work with scalars or very short blocks of data. Productivity lost when programmer must introduce artificial aggregation
    - **Challenge: What operations? What atomicity? What lengths?** (e.g., known that CAS too limited – motivation for transactional memory)
    - See compromise's in MPI-3 RMA design
    - **Challenge: Did we get these right? Will hardware be able to exploit them**

The Real RDMA Challenge

- **Match RDMA hardware capabilities now *and in the next 5-10 years* to the MPI programming model *and preserve performance***
- The issues are not the speed of block transfers but the handling of local and remote completion of memory transfers and efficient synchronization within the programming model

MPI and Shared Memory

- Shared memory programming is harder than you think
  - "You don't know Jack about Shared Variables or Memory Models", CACM Vol 55#2, Feb 2012.
- Users want it to "just work" but without sacrificing performance
- **Challenge: Define a programming model that permits exploitation of shared memory but remains safe for users**
  - Data race-free programming one example
  - MPI-3 has made a good attempt by providing shared memory windows and the unified memory model, but whether this will be effective is yet to be seen

Issues at Scale

- The end of frequency scaling has forced a rapid increase in concurrency
- Systems now have 10,000 times as many processor cores as the "extreme scale" machines when MPI was first developed

Describing Collective Operations

- Many collective routines have O(p) arguments
- **Challenge: Replace collective routines with more scalable versions that match algorithm needs**
  - "Neighbor" collectives in MPI-3 one step in this direction
- **Challenge: Should non-scalable routines be deprecated (e.g., MPI_Alltoall)?**
  - Should we force programmers to think more scalably

Physical and Virtual Topologies

- Important when MPI-1 defined
  - Hypercubes, meshes, trees, …
- MPI-1 attempted to define an abstraction for topologies with MPI_Cart_create/MPI_Graph_create. MPI_Dims_create gives a specific decomposition.
  - None of these provide enough control to match needs
  - Attempts to both provide an abstraction *and* a specific behavior
    - Result is not useful to anyone
- Today the situation is more complex
  - Multilevel nodes, more complex networks, routing methods
  - Performance irregularities common (compute resources unequally shared)
- **Challenge: Define an effective means for programs to express their (dynamic) communication pattern and map that onto physical network resources**

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

Faults and MPI

- Almost no standard interface required to survive unspecified problems
- What are the likely faults?
  - Note poor analysis of hardware / software faults in some studies – persistent faults can sometimes be identified, but transient faults (hardware upsets, timing/race issues in software) means source of many faults unknown
- **Challenge: What are the likely faults and how will system software respond to them? How should a programming model interact with the system? How much should the programmer participate in managing different kinds of faults?**

Library and Language

- No Library-based implementation is ever complete
  - You can always add routines, and give good reasons to do so
  - MPI-1, MPI-2 model
    - Try for few concepts, provide all natural routines
    - E.g., MPI_Issend
  - May be starting to lose the model
    - MPI-3 Assumption: Threads are now and will be for the next 5-10 years too inefficient to use in parallel programming
      - Required to justify many new nonblocking routines, all of which could be implemented within a thread
      - If only "now", then not a valid justification to add something to a standard
        - But would be a justification to add something to a research platform
- It is too easy to add routines to a library
  - Costs: completeness, complexity, short term rather than long term

Library Challenges

- "Basic" Langauge datatypes
  - Used to be int, short, long, float, double, char
  - Now int32_t, wchar_t, …. Expected by programmers
    - How can MPI keep up?
  - Nonblocking operations always an issue
    - C's pointers mostly keep compiler from optimizing away; Fortran actively exploits "knowing" about data lifetime
    - There are language ideas to address this – e.g., Futures. **How can these fit with MPI?**
- C++ : **What is the right level** (this is the general "high level language" issue, and why MPI has succeeded by staying at a lower level)
  - Also adds inter-language issues. What happens to a Fortran routine that calls a C++ routine that throws an exception?

Productivity

- Distributed Data Structures (DDS) essential for high productivity
  - Global Arrays one example
  - HPF, ZPL, …
- **Challenge: Extend MPI to include elements of support to DDS**
  - Sort of in datatypes – but too hard to use
- **Challenge: Find better building blocks for distributed data structures**
  - Why didn't the support emerge?
  - Can libraries alone provide an effective solution?

Miscellaneous Comments

- Fatal mistake: Define semantics and then (tell someone to) make it fast
  - ♦ Performance *requires* choosing semantics that *can* be efficiently implemented
  - ♦ Good design matches performance requirements with usability (typically with compromises)
- Comparison:
  - ♦ Reference implementation: specification is precise enough to be implemented (with functionality but not necessarily with performance); identify inconsistencies between routines;
    - Really good implementation can identify performance issues on the platform on which it is implemented
  - ♦ Paper implementation WRT expected future hardware capabilities: specification will permit access to performance features into the future
    - Suitably careful paper implementation will ensure that the specification is precise enough to be implemented

Conclusions

- Careful design and consideration of the long term made MPI 1 and MPI 2 extraordinarily successful
- MPI started at about the same time as "attack of the killer micros"; enjoyed two decades of relative architectural stability
- The approaching end of CMOS has introduce great uncertainty and opportunity
- MPI can continue to evolve as parallel computing evolves, but only by being careful to take the long view and continue to exploit composition of programming models (hybrid programming)