Challenges for MPI in Its Third Decade

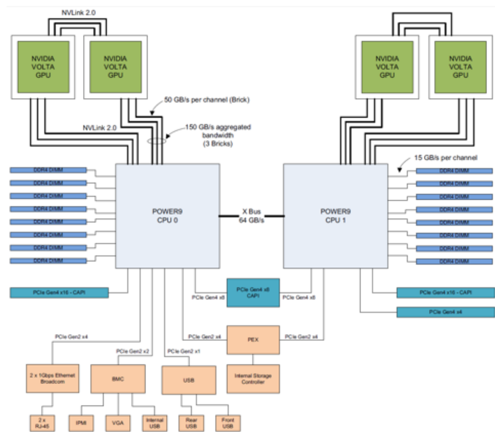
William Gropp
wgropp.cs.Illinois.edu

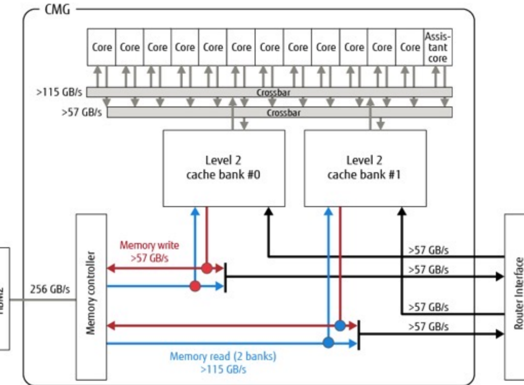
NCSA

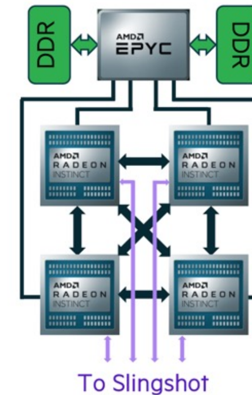
Some Context

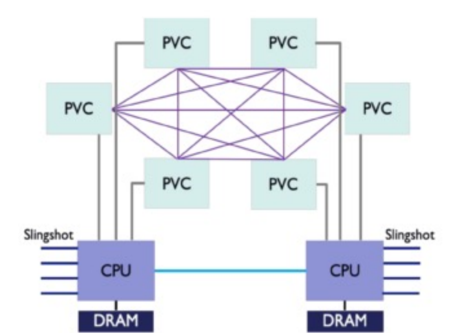
- Before MPI, there was chaos – many systems, but mostly different names for similar functions.
  - Even worse – similar but not identical semantics
- Same time(ish) as attack of the killer micros
  - Single core per node for almost all systems
- Era of rapid performance increases due to Dennard scaling
  - Most users could just wait for their codes to get faster on the next generation hardware
  - MPI benefitted from a stable hardware and thus software environment
    - Node programming changed slowly, mostly due to slow quantitative changes in cache, instruction sets (e.g., new vector instructions)
- The end of Dennard scaling unleashed architectural innovation
  - And imperatives – more performance requires exploiting parallelism or specialized architectures
  - (Finally) innovation in memory – at least for bandwidth

HPC Nodes are Increasingly Complex









DOE Sierra

- Power 9 with 4 NVIDIA Volta GPU
- 4320 nodes

DOE Summit similar, but

- 6 NVIDIA GPUs/node
- 4608 nodes

Fugaku

- Fujitsu A64FX (includes Vector Extensions)
- 158,976 (+) nodes

DOE Frontier

- AMD with 4 AMD GPU
- 100+ racks

NCSA Delta similar but NVIDIA GPUs and fewer racks ☺

DOE Aurora

- Intel SR with 6 Intel Ponte Vecchio GPUs
- Being deployed, >9K nodes

CFD in 2030: NASA Study

Prediction in 2012 (published 2014)

Year	Feature size	Derived parallelism	Stream parallelsim	PIM Parallelism	Clock rate GHz	FMAs	GFLOPS (Scalar)	GFLOPS (Stream)	GFLOPS (PIM)	Processor per node	Processor (TFLOP)	nodes per system	Total (PFLOPS)
2012	22	16	512	0	2	2	128	1024	0	2	1	10000	23
2020	12	54	1721	0	2.8	4	1210	4819	0	2	6	20000	241
2023	8	122	3873	512	3.1	4	3026	12006	1587	4	17	20000	1330
2030	4	486	15489	1024	4	8	31104	61956	8192	16	101	20000	32401

Compare 2023 prediction to actual systems in 2022

- Not bad – if “processor” includes GPU, change to 8/node and 10k/system.
- PIM – stands in for HBM with operations. Maybe a miss (or not?)
  - Note that PIM still not a dominant part
- DP performance of an NVIDIA A100 is 9.7TF (19.5TF tensor core) – not too far from “Processor”
- Missed – Assumed tighter integration – “Stream” flops part of processor, not in a separate socket (GPU)
- Jeffrey P Slotnick, Abdollah Khodadoust, Juan J Alonso, David L Darmofal, William D Gropp, Elizabeth A Lurie, Dimitri J Mavriplis, and Venkat Venkatakrishnan. Enabling the environmentally clean air transportation of the future: a vision of computational fluid dynamics in 2030. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 372(2022), 2014.

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",
- https://link.springer.com/chapter/10.1007/3-540-45307-5_8

Performance vs. Productivity

- MPI gives the tools for achieving performance
  - In large part by not getting in the way of locality management
- But that very feature impacts productivity
  - User has *no choice* but to manage locality, which is both hard and tricky
- In addition, as Marc Snir has noted, MPI is neither high nor low level
- But is that part of MPI's ***success*** – it does both high and low level, and the tradeoff in greater use (mostly) makes up for loss of performance/function
- Any programming system will need to consider the tradeoffs of
  - Latency vs. Bandwidth vs. Convenience vs. Modularity (among others)

But What about the Programming Crisis?

- Use the right tools!
- MPI tries to satisfy everyone, but the real strengths are in
  - Attention to performance and scalability
  - Support for libraries and tools
- Many computational scientists use frameworks and libraries built upon MPI
  - This is the right answer for most people
  - Saying that MPI is the problem is like saying C (or C++) is the problem, and if we just eliminated MPI (or C or C++) in favor of a high productivity framework *everyone's* problems would be solved
  - In some ways, MPI is *too* usable – many people can get their work done with it, which has reduced the market for other tools
    - Particularly when those tools don't satisfy the 6 features in the success of MPI

What Might Be Next

- Intranode considerations
  - SMPs (but with multiple coherence domains); new memory architectures
  - Accelerators, customized processors (custom probably necessary for power efficiency)
  - MPI can be used (MPI+MPI or MPI everywhere), but somewhat tortured
    - No implementation built to support SIMD on SMP, no sharing of data structures or coordinated use of the interconnect
- Internode considerations
  - Networks supporting RDMA, remote atomics, even message matching (partially supported in MPI now – but what's next?)
  - Overheads of ordering
  - Reliability (who is best positioned to recover from an error)

What Might Be Next

- MPI is both high and low level – can we resolve this?
- Challenges and Directions
  - Scaling at fixed (or declining) memory per node
    - How many MPI processes per node is “right”?
  - Realistic fault model that doesn’t guarantee state after a fault
  - Support for complex memory models (MPI_Get_address ☺)
  - Support for applications requiring strong scaling
    - Implies very low latency interface and overheads
    - Low latency means paying close attention to the implementation
      - RMA latencies sometimes 10-100x point-to-point in implementations (!)
  - MPI performance in MPI_THREAD_MULTIPLE mode
  - Integration with code re-writing and JIT systems as an alternative to a full language

Adapt to Innovation in Architecture

- Complex nodes
  - MPI + X, for X such as OpenMP, CUDA, OpenACC, etc. often effective
  - But challenges in the "**+**": sharing of resources such as cores, memory, …
- Implementation of MPI on complex nodes
  - Sharing information between MPI processes on the same node that must share resources, such as memory, network, accelerators, …
  - Optimize data movement
- Some can be hidden from the user (shared memory for intranode message passing)
- Some requires user action – e.g., node-aware algorithms and methods

Adapt to New Language Models – And to Their Rapid Evolution

- Is (long-term) backward compatibility still important?
  - Many newer languages and systems don't think so – 5 years is long for them
- How does the value of backward compatibility change with age?
  - As older codes become less important (or more modern codes become available), what is the tradeoff in making newer codes more capable/flexible/etc. or the environment more productive?
- What is the cost to future applications and usage from providing backward compatibility?
  - Many of us started careers when long-term backward compatibility was expected. Is this still the right thing?
- What does this mean for MPI?

MPI and I/O

- One area where the the technology is changing *despite* HPC is I/O
  - POSIX I/O never designed to support parallel applications
    - Does define behavior of concurrent writers, readers
  - High performance with POSIX I/O challenging
    - Lots of state (e.g., access times, permissions), hierarchical file structure
    - Reflected in concerns about "metadata" impacting performance and stability
  - Apps that say they requires POSIX almost always mean
    - I can't/won't change my code, which uses read/write/open/close
    - Almost never depend on full POSIX semantics
    - That is, they do not need POSIX – and some HPC centers
  - Modern "big data" systems are not based on POSIX
    - "Object store" decouples where data is stored from high-level information about how data relationships are organized

MPI I/O

- Essential features
  - Whole job I/O (communicator scope)
  - Efficient mapping of data from MPI processes to storage object ("file")
  - Relaxed consistency model
  - Very limited metadata requirements
- Implementation Issues
  - POSIX a poor match to MPI I/O – overly strong consistency model, defines behavior of individual processes, not parallel application
  - HPC sites still focus on POSIX
  - Thread interactions – why not have the user put thread-same MPI calls into a thread to obtain non-blocking actions?
- Opportunities
  - Database
  - Coordination with storage APIs to match needs of MPI applications

MPI I/O and POSIX

- MPI I/O designed to be high performance
  - Relaxed consistency encourages data caching, optimization of data transfers – even between MPI processes
  - Use MPI datatypes to describe arbitrary data access patterns
    - Reduces number of routine calls to describe data motion
  - Collective open informs file system that multiple processes are coordinating I/O activity
  - Almost no file metadata requirements
- POSIX I/O supports *none of these performance features*
  - This has been true and recognized since MPI-IO added to MPI (1997)
  - MPI never required POSIX I/O – and in fact POSIX I/O has been detrimental to MPI I/O and scientific application performance
    - POSIX I/O not designed for parallel science applications
- Modern systems have more complex I/O hardware
  - Node and server SSD
  - Non HPC systems do not use POSIX I/O
  - Some (many) HPC systems do not provide POSIX, but claim to
    - "turn off" consistency or other features that are "inconvenient" – but in the process, violate POSIX standard.
    - Applications (including MPI IO implementations targeting POSIX I/O) that depend on correctness may fail in hard-to-find race conditions
- Some work on MPI IO on non-POSIX parallel file systems

Limitations of MPI I/O

- Did not define other file system operations
  - Access, metadata, naming (including directory structure) implementation-dependent
  - MPI 4.0, section 14.5: “In particular, an implementation must specify how familiar operations similar to POSIX cp, rm, and mv can be performed on the file.” (Pretty much unchanged since MPI-2)
- Results in relying on some underlying system that provides those features – for HPC, it was (still is, most places) POSIX
- Asynchronous I/O (progress) not mandatory
  - MPI Forum discussing “strong” and “weak” progress definition
  - Users want “timely” progress, not just functional progress
    - Very hard to define in a standard – but can still be expected
    - Quality of code generated by a compiler is not part of language standard – but users expect and depend upon that

Is the MPI I/O Model Correct for Todays' Systems and Applications?

- Data vs. Files
  - MPI I/O is mostly low level – Focus on moving bytes
  - One high-level feature – mapping of distributed data structure to files with MPI_File_set_view
- Some applications use the file system as a primitive database
  - All too common to use file as a database
    - Each record a separate file
  - Problem: file metadata can be a bottleneck
    - Rarely of interest to the (scientific) programmer
    - For record-is-file, metadata overly fine-grained
- Modern databases can provide performance and efficiencies
  - EMPRESS: Accelerating Scientific Discovery Through Descriptive Metadata Management
  - https://doi.org/10.1145/3523698 ACM Transactions on Storage (2022)
- How would you drive adoption of a new, non-POSIX approach?
  - Sharing attributes across shared data
  - What role should MPI play?

RMA and MPI

- MPI originally designed for two-sided message passing
  - Both point-to-point and collective
- Success of MPI-1 + emergence of alternative one-sided communication approach led MPI Forum to add more distributed memory programming features
  - MPI is really a “Distributed Memory Programming Interface”
- RMA is a good example of the tension between high- and low-level interfaces

Different Perspectives on RMA

- Different communities have different goals for RMA
  - Express algorithms with different synchronization patterns
    - Improves (perhaps significantly) programmability for some algorithms
    - Better match to algorithm; e.g., avoid extra messages/data transfers/synchronization
  - Access higher performance
    - Reduce CPU overhead (e.g., tag matching, message ordering)
    - Minimize memory motion, control messages
    - Exploit hardware support for communication, overlap with computation, especially for large block transfers
    - Low-latency for short data transfers
    - Remote atomic memory operations or other hardware assist
      - But tricky – e.g., atomics not always faster than lock/ops/unlock for sequences of operations
  - Optimize for important special cases
    - E.g., symmetric allocation of memory

Thinking about RMA performance

- Different applications have different needs
  - Some need low overhead/latency on short messages; common for strong scaling case
  - Some need high bandwidth for large messages; overlap with computation; common for weak scaling case
- These can be in tension, with performance tradeoffs
- MPI "look and feel" adds additional tradeoffs
  - Datatypes can provide optimization opportunities (e.g., bundle move, from strided to general scatter/gather, together) but add complexity (increase latency for simple cases)
  - Rich capabilities can force reliance on remote agent, impacting performance

What Can an RMA Design Assume?

- MPI has succeeded by embracing a “Greatest Common Denominator” approach
  - Take advantage of community consensus on hardware features
  - Ensure portability with good performance
- But the “Common” part stifles innovation
  - Features not already adopted are hard to fully exploit from within the standard
  - The tyranny of “Common” forces the standard to sacrifice both performance and programmability for portability and precise semantics
- Q: How can MPI encourage innovation in RMA interfaces?
- Q: Is portability (with performance) a hard or soft constraint?

MPI One Sided/Remote Memory Access History

- MPI-2 added RMA in 1997 (25 years ago!)
  - Some practice, but semantics of other systems often imprecise
  - Matched hardware capabilities of high-end systems of the time (Cray T3D/T3E; NEC Earth Simulator)
  - Expected support in network NIC with local memory (hence memory model)
  - Only collective association of memory with MPI_Win
- Both Fence and PSCW defined to exploit the hardware of the day
  - Fence could be implemented in hardware and was *very* fast on some systems (e.g., T3D/T3E)
  - PSCW described halo exchange well (though hard to exploit)
- Active target and passive target captured different application styles
  - Passive target limitations an attempt to ensure portable performance

MPI One Sided/Remote Memory Access History

- MPI-2 RMA had limited adoption
  - Complex memory model hard to explain
  - Limitations on passive target memory limit usefulness
  - Limitations on operations, memory, etc.
  - Poor performance of implementations – often unnecessarily so
  - Even with that, some apps and implementations did very well
- MPI-3 substantially revised and enhanced RMA in 2012
  - Address overly strong correctness semantics (undefined rather than erroneous) and additional use cases for applications
  - Add "unified" memory model – HW support for coherency now widespread
  - Add additional ways to associate memory, describe data transfers, complete operations, and extend to processes sharing memory
  - But added to MPI-2 RMA – keeping all features from (then) 15 years before

Relevance

- Is MPI RMA too complex, portable, limited, constrained, etc. to be useful?
  - Consider challenges in using MPI RMA for implementing other one-sided programming systems and libraries
- MPI-2 RMA, for all of its limitations, was driven by use examples of the time. MPI-3 also driven by different use examples.
  - What are the right use cases for MPI-5 RMA?
  - Who is the right audience?
- Is MPI RMA a high-level interface, expected to be used freely within user applications, or a low-level interface, used to implement core abstractions in an application framework?
  - Like much of MPI, as Marc Snir points out, it is both – and that is likely a bad choice

Synchronization and Notification

- Moving data is the easy part. Synchronization/notification is the hard part
  - This is the biggest area where RMA has struggled, with many different mechanisms for completing RMA, both locally and remotely
    - Example: Fence – with hardware support, can be incredibly fast – but imposes a "BSP"-like structure. More general semantics (groups != WORLD) may not have same hardware support, but this is difficult for the programmer to determine
- Notification is both more powerful and harder/more demanding to implement while meeting user performance expectations
- Small changes in semantics can have large performance impact if they change what can be done in hardware and what requires CPU assist
  - This applies to *both* the MPI specification and the capabilities of hardware

MPI RMA Synchronization

- MPI RMA Synchronization is complex
- Trying to keep things simple for programmer (all sync methods available at all times) makes implementation complex and adds overhead
- Q: How important is this level of interoperability?
- Q: Are these the right ways to synchronize?

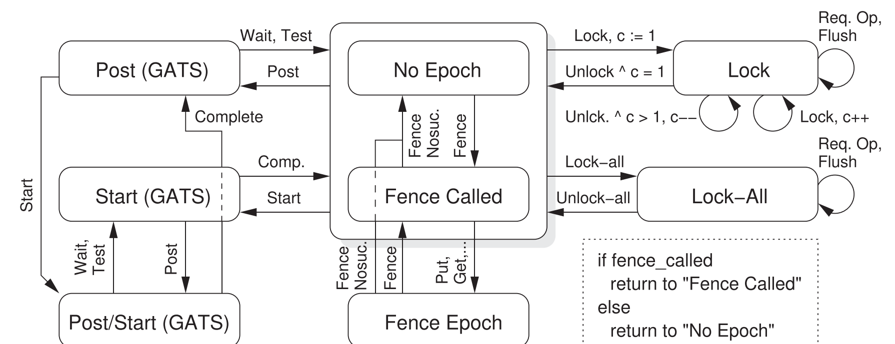


Figure 3. Remote memory access synchronization state tracking diagram. Dashed lines indicate that a particular state is bypassed, depending on the fence state of the process.

An implementation and evaluation of the MPI 3.0 one-sided communication interface, James, Dinan, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur, *Concurrency and Computation: Practice and Experience*, 28, 17, 4385–4404, cpe.3758, 2016.

NCSA

Innovation and Stability

- How can MPI RMA stay current with technology when there isn't consensus?
  - It can't – so we'll need to make some compromises
    - We're currently accepting lower performance and capability to get portability and stability of code. Is that the right choice?
- Q: Should MPI RMA provide access to features that can't be efficiently supported on all platforms?
  - Should implementations be required to support all features, even if they are inefficient, or should the features not be supported?
  - How can users reliably determine what is supported, and what is supported with adequate performance (and how do you define that)?

Audience

- Who is expected to use MPI RMA? End users? Tool developers? Compiler writers?
  - More precisely, *which* parts of RMA are for each of these groups?
  - What is the role of libraries?
  - For end users, how expert are the users? Shared memory issues are very tricky; RMA shares many of these hazards.
- The MPI RMA specification is aimed at MPI implementors, and (correctly) worries about all the edge cases
  - Users (mostly) don't want that – they want a subset that is easy to use and understand, even if that might mean missing some optimizations
- Q: Can we describe RMA for a single audience, or should there be different user profiles?
- Q: Can/Should MPI RMA define easier-to-understand subsets? Or is this the obligation of training materials?

Feature Lifetime

- What is the lifetime required? Do RMA codes need to run without change in 20 years? 10? 5? At what cost in potential performance?
  - This impacts how we approach hardware innovation
  - Many modern software systems expect to break backward compatibility – is it time for MPI to do the same, at least in some places?
- Note we’re leaving an era of over 3 decades of architectural stability – which has been of great benefit to MPI
  - But we’re leaving that era – what made sense while architectures were (mostly) stable may no longer make sense
- Q: Is backward compatibility essential?
- Q: Could the standard offer some features with an explicit limited lifetime?

Progress

- One-sided nature of RMA requires some progress guarantee
- But TANSTAAFL (There Ain't No Such Thing As A Free Lunch)
  - Many tradeoffs – e.g., more frequent/responsive progress *may* increase latency, lower performance. Or increase latency but increase performance. Or increase performance, because you found a good use for an idle core…
- Many changing technical tradeoffs (dark silicon, "extra" cores, …)
  - Tradeoffs that made sense with < 1core/chip may not with >100 cores/chip
- Rather than all-or-nothing progress, is there something in the middle?
  - Note that MPI-2 permitted restricting passive target operations to special memory – something many did not like, but made sense at the time
  - MPI Forum discussions of weak and strong progress an example
- "Functional progress," e.g., a guarantee of eventual progress, does not meet most users' expectation for "timely progress"

Performance and Generality

- MPI is a greatest common denominator approach
  - Often described insultingly as “least common denominator” – which is a nonsense phrase
  - But even “greatest common” is limited to “common”
- Significant performance impact when abstraction is far from what is supported in hardware – but hardware operations esp. for RDMA are still evolving
  - Some systems handle this by giving up on precision in the specification (!!)
- Is high performance low latency or high bandwidth? What if you can’t have both?
- The goal of “performance portability” without code change opposes making “optional” features available
  - But what if a framework (e.g., PETSc) insulates the user from those changes?
  - Q: A what level (if any) do we need performance portability?

Thoughts for RMA in MPI 5.0

- RMA in MPI-2 was driven by the hardware of the day, including limitations
  - Examples: Fence in hardware, limited memory in NIC (separate memory model, passive target memory restrictions)
  - 25+ years is too long to simply tweek the programming model to match the hardware – MPI RMA should be rethought from the ground up to meet current hardware
- One-sided hardware acceleration remains in flux
  - No consensus on what are the right abstractions (though some are clear)
  - Suggests:
    - Don’t require greatest common denominator for RMA.
    - Provide a way to access extensions and query for capabilities.
    - Define a likely subset where portability (in time and across vendors) is important as a trade off in performance
  - Applications are likely to define communication abstractions – and can provide implementations that can exploit optional features without imposing a large burden on the programmer
    - Many do this today

Thoughts for RMA in MPI 5.0

- Progress may be solved, at least to first order
  - Can we assume that there are enough cores/execution contexts to ensure some progress?
    - Or is this the wrong direction? Should we be looking at progress with no CPU involvement, at least with the right hardware?
  - As above, are there intermediate levels of progress, as there are for thread support?
- Evolution should be driven by use cases
  - Where do we want to see MPI RMA used? How do we engage that community?
  - But a warning: an RMA interface that is the Union of all features may satisfy no-one.
    - A strength of MPI is its support for tools and higher-level interfaces
    - Can we ensure that users are served by these tools without requiring MPI to directly support everyone?

Accelerators and MPI

- Major challenge: Separate memory spaces
  - Even if "unified", they have different performance characteristics
- Data Movement
  - MPI allows arbitrary access patterns – not just contiguous or strided
  - Like RMA, generality may be a mismatch for hardware support
- Synchronization, not just data movement
  - MPI assumed general processor
- MPI designed around messages, not streams
  - Streams imply some ordering, which can be a performance problem in large networks
  - But intranode, some sort of stream ordering may be preferable – and supported in hardware

Small Steps

- MPI Partitioned Communication
- “High quality” MPI implementation
  - Recognize different memory regions, adapt internally
- MPI Forum HACC WG
  - Continuations proposal #6 (also see #585)
  - Clarification of thread ordering rules #117 #748
  - Integration with accelerator programming models:
    - Accelerator info keys #3 #714
    - Accelerator Synchronous MPI Operations #11
    - Accelerator bindings for partitioned communication #4
    - Partitioned communication buffer preparation (shared with Persistence WG) #264
  - Asynchronous operations #585
  - https://github.com/mpiwg-hybrid/hybrid-issues/wiki

Summary

- MPI has been remarkably successful
  - Powerful abstractions, avoided being tied too closely to HW at a moment in time
  - Benefitted from stability in architecture
- That era of stability has ended
  - MPI needs to adapt
  - HPC no longer driving all high-performance HW, SW
- Time to identify and rethink assumptions
  - Tradeoffs in portability, performance, programmability
- Rethink building blocks
  - Consider streams, notification, subsets
  - Explicit consideration of latency and bandwidth ***separately***