The Grass is Always Greener: Reflections on the Success of MPI and What May Come After

William Gropp
wgropp.cs.illinois.edu
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 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
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](https://link.springer.com/chapter/10.1007/3-540-45307-5_8)
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? Let’s 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++, Java, and Python runtimes are large
    • Development Frameworks
      • Hundreds to thousands of methods
  • This doesn’t bother millions of programmers

• Exceptions are hard on users
  • But easy on implementers — less to implement and test

• Example: MPI_Issend
  • MPI provides several send modes
  • 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
Modularity and Composability

- Many 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
    - “Programming in the large”
  - Example: communication contexts
- 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
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
I can do “Better”

• “I don’t need x, and can make MPI faster/smaller/more elegant without it”
  • Perhaps, for you
  • Who will support you? Is the subset of interest to enough users to form an ecosystem?
• My hardware has feature x and MPI *must* make it available to me
  • Go ahead and use your non-portable HW
  • Don’t pretend that adding x to MPI will make codes (performance) portable
• Major fallacy – measurements of performance problems with an MPI implementation do *not* prove that MPI (the standard) has a problem
  • All too common to see papers claiming to compare MPI to x when they do no such thing
    • Instead, the compare an implementation of MPI to an implementation of x.
  • Why this is bad (beyond being bad science and an indictment of the peer review system that allows these) – focus on niche, nonviable systems rather than improving MPI implementations
Maybe you *Can* do Better

- There is a gap between the functional definition and the delivered performance
  - Not just an MPI problem – common in compiler optimization
    - Many (irresponsible) comments that the compiler can optimize better than the programmer
    - A true lie – true for simple codes, but often false once nested loops or more complex code; often false if vectorization expected
    - “If I actually had a polyhedral optimizer that did what it claimed…” – comment at PPAM17
  - In MPI:
    - Datatypes
    - Process topologies
    - Collectives
    - Asynchronous progress of nonblocking communication
    - RMA latency
    - Intra-node MPI_Barrier (I did 2x better with naïve code)
    - Parallel I/O performance
    - …

- Challenge for MPI developers:
  - Which is most important? Optimize for latency (hard) or asymptotic bandwidth?
Why Ease of Use isn’t the Goal

- Yes, of course I want ease-of-use
  - I want matter transmitters too – it would make my travel much easier
- Performance is the reason for parallelism
  - Data locality often important for performance
  - MPI forces the user to pay attention to locality
    - That “forces” is often the reason MPI is considered hard to use
- It is easy to define systems where locality is someone else’s problem
  - “Too hard for the user – so the compiler/library/framework will do it automatically for the user!”
  - HPC compilers can’t even do this for dense transpose (!) – why do you think they can handle harder problems?
  - Real solution is to work with the system – don’t expect either user or system to solve the problem
- Making them useful is an unsolved problem

Simple, unblocked code compiled with O3 – 709MB/s
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
The Grass is Always Greener...

- You can either work to improve existing systems like MPI (or OpenMP or UPC or CAF) or create a new thing that shows off your new thing.
- One challenge to fixing MPI implementations:
  - Researchers receive more academic credit for creating a new thing (system y that is “better” than MPI) rather than improving someone else’s thing (here’s the right algorithm/technique for MPI feature y).
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
  • Overheads of ordering
  • Reliability (who is best positioned to recover from an error)

• MPI is both high and low level (See Marc Snir’s talk today) – 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 …
    • Low latency means paying close attention to the implementation
    • RMA latencies sometimes 10-100x point-to-point (!)
  • MPI performance in MPI_THREAD_MULTIPLE mode
  • Integration with code re-writing and JIT systems as an alternative to a full language
Summary

• MPI was successful because
  • It focused on performance, the reason that most users go parallel
  • It focused on completeness, so that there would be a large enough user community to support it
  • It focused on clear and precise semantics, so it was clear what the operations *did*
  • It was pragmatic about not being a language, despite the benefits
  • It supports backwards compatibility, something no longer a goal for modern software 😞
  • It was developed in a truly open process by a diverse group of great people

• MPI should and can be augmented and/or replaced
  • But by something more, not less, capable
  • And as part of an ecosystem that provides both higher and lower level APIs