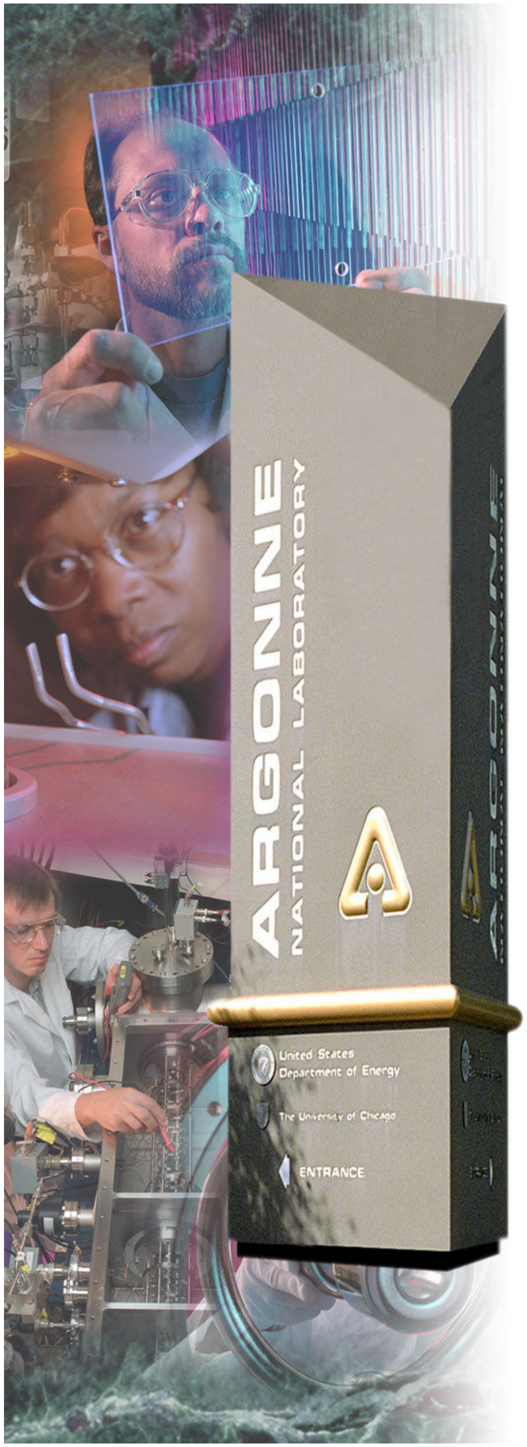


How to Replace MPI as the Programming Model of the Future

William Gropp

www.mcs.anl.gov/~gropp

Argonne National Laboratory

Office of Science
U.S. Department of Energy

A U.S. Department of Energy
Office of Science Laboratory
Operated by The University of Chicago

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)

Some History

- **The Cray 1 was a very successful machine. Why?**
  - Not the first or fastest vector machine (CDC Star 100 and later Cyber 203/205 had much higher peaks)
  - Early users were thrilled that Cray 1 was 1.76 x faster than CDC 7600
    - *1.76 = 22/12.5 = ratio of (scalar) clock speed*
    - *Applications were not vectorizing!*
    - *These were legacy applications*
  - Balanced (by comparison with Star100 etc.) performance
  - Hardware supported programming model; predictable performance
  - Software (compiler) trained users
    - *Worked with the user to performance-tune code*
  - Of course, this was before the memory wall

The Curse of Latency

- **Current vector systems (often) do not perform well on sparse matrix-vector code**
  - Despite the fact that an excellent predictor of performance for these algorithms is STREAM bandwidth on cache-based systems
  - And that there is a great deal of concurrency
- **Lesson: Balance and Amdahl's law**
  - Many thought Amdahls's law would limit the success of massively parallel systems
  - Reality — It is often easier to find concurrent scalar threads in modern algorithms than scalar-free vector code
  - Don’t confuse necessary with sufficient

How Expensive Is MPI?

- **Measured numbers from still untuned MPICH2 shared memory implementation:**
- **MPI_Recv : Number of Instructions 323**
MPI_Recv : Number of Cycles 1240
  - MPI_Recv : Number of L1 Cache Data misses 029
MPI_Recv : Number of L1 Cache Inst misses 002
- **MPI_Send : Number of Instructions 278**
- **MPI_Send : Number of Cycles 995**
  - MPI_Send : Number of L1 Cache Data misses 007
  - MPI_Send : Number of L1 Inst misses 003
- **This may seem like a lot, but much of the cycle cost is latency to remote memory, not overhead.**
- **On “conventional” systems, eliminating MPI overhead gives less than a factor of 2 improvement**
  - Need latency hiding, adequate concurrent, independent threads to make effective cost of an operation the overhead cost instead of the latency cost

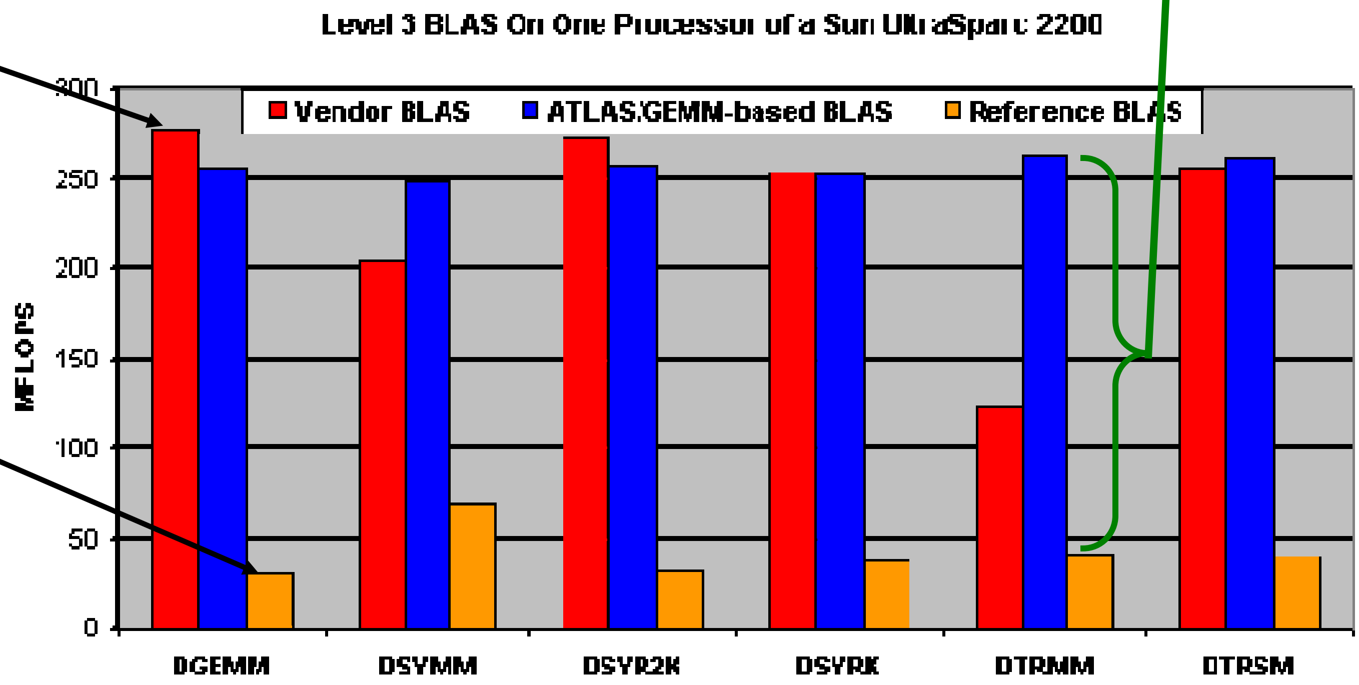
What Performance Can Compilers Deliver?

- **Here are a few questions for a computer vendor**
  1. Do you have a optimized DGEMM?
     - *Did you do it by hand?*
     - *Did you use ATLAS?*
     - *Should users choose it over the reference implementation from netlib?*
  2. Do you have an optimizing Fortran compiler
     - *Is it effective?*
- **Aren't the answers to 1 and 2 *incompatible*?**
- **Conclusion: Writing optimizing compilers, even for the easiest, most studied case, is *hard***

Parallelizing Compilers Are Not the Answer

Large gap between natural code and specialized code

Hand-tuned

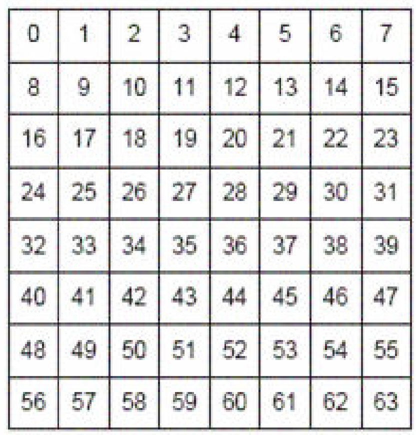


Compiler

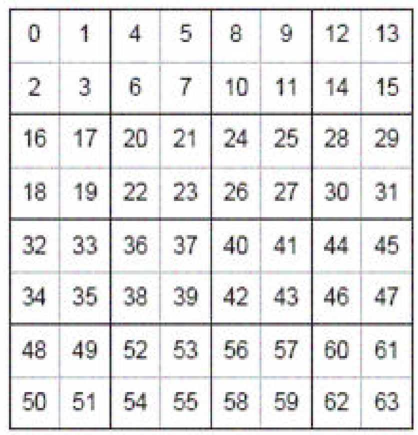
From Atlas

Enormous effort required to get good performance

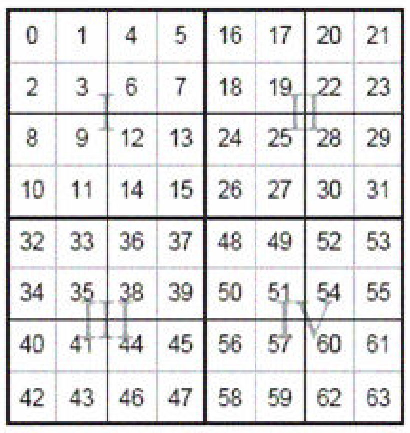
Manual Decomposition of Data Structures



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

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 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 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*

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

Conclusions:
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

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"*
- **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**
  - 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

- **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

PGAS Languages

- **A good first step**
- **Do not address many important issues**
  - Still based on action-at-a-distance
  - No high-level support for data decomposition
  - No middle ground between local and global
  - Lower overhead access to remote data but little direct support for latency hiding
  - I/O not well integrated (e.g., the easiest way to write a CoArray in canonical form (as part of a distributed array) is to use MPI)
  - Not performance transparent; no direct support for performance debugging

How to Replace 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*

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 current thinking on possible directions,**
  - "*Report of the Workshop on High-Productivity Programming Languages and Models*", edited by Hans Zima, May 2004.