MPI: The Once and Future King

William Gropp
wgropp.cs.illinois.edu
MPI The King

- MPI remains the dominant programming model for massively parallel computing in the sciences
  - Careful design filled a gap
  - Good and ubiquitous implementations provide reliable performance
  - Applications developers found it (relatively) easy to use
Where Is MPI Today?

• Applications already running at large scale:

<table>
<thead>
<tr>
<th>System</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tianhe-2</td>
<td>3,120,000 (most in Phi)</td>
</tr>
<tr>
<td>Sequoia</td>
<td>1,572,864</td>
</tr>
<tr>
<td>Blue Waters</td>
<td>792,064* + 1/6 acc</td>
</tr>
<tr>
<td>Mira</td>
<td>786,432</td>
</tr>
<tr>
<td>K computer</td>
<td>705,024</td>
</tr>
<tr>
<td>Julich BG/Q</td>
<td>393,216</td>
</tr>
<tr>
<td>Titan</td>
<td>299,008* + acc</td>
</tr>
</tbody>
</table>

* 2 cores share a wide FP unit
Science that can’t be done in any other way

• Plasma simulations – W. Mori (UCLA)
• High sustained floating point performance needed
  ◆ 150 million grid points and 300 million particles
  ◆ (2 cm)$^3$ of plasma
Science that can’t be done in any other way

• Turbulent Stellar Hydrodynamics – P. Woodward (UMN)
  ◆ Sustained 1 PF/s computing for weeks
  ◆ Back to back full system jobs.

• Transistor roadmap projections – G. Klimeck (Purdue)
  ◆ Support for CPU/GPU codes.
Science that can’t be done in any other way

- Earthquake response modeling – T. Jordan (USC)
  - CyberShake workloads using CPU and GPU nodes, sustained, for weeks.
  - Seismic hazard maps (NSHMP) and building codes.

- Severe storm modeling – B. Wilhelmson (Illinois)
  - First-of-its-kind, 3-D simulation of a long-track EF5 tornado.
Science that can’t be done in any other way

- **Nek5000 – P. Fischer (Illinois)**
  - Computational fluid dynamics, heat transfer, and combustion.
  - Strong scales to over a million MPI ranks.

*IC engine intake stroke; G. Giannakopoulos, ETHZ*

*Nek5000 Strong-Scaling Study on Mira*
MPI is not only for Scientific Computing

Collaborative Filtering (Weak scaling, 250 M edges/node)

- MPI
- Combblas
- Graphlab
- Socialite
- Giraph

Time per iteration (seconds)

Number of nodes

Factor of 100!
Becoming The King

• Like Arthur, MPI benefited from the wisdom of (more than one) Wizard

• And like Arthur, there are many lessons for all of us in how MPI became King
  ♦ Especially for those that aspire to rule...
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”,
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

• 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 that 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
The Pretenders

• Many have tried to claim the mantel of MPI
• Why have they failed?
  ♦ They failed to respect one or more of the requirements for success
• What are the real issues in improving parallel programming?
  ♦ I.e., what *should* the challengers try to accomplish?
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
- ICE (Illinois Computing Environment) under development as a way to provide a framework for integrating other performance tools
Let The Compiler Do It

• This is the right answer ...
  ♦ If only the compiler *could* do it

• Lets look at one of the simplest operations for a single core, dense matrix transpose
  ♦ Transpose involves only data motion; no floating point order to respect
  ♦ Only a double loop (fewer options to consider)
Transpose Example Review

- do \( j=1,n \)
  - do \( i=1,n \)
    - \( b(i,j) = a(j,i) \)
  - enddo
- enddo

- No temporal locality (data used once)
- Spatial locality only if \((\text{words/cacheline}) \times n\) fits in cache

- Performance plummets when matrices no longer fit in cache
Blocking for cache helps

- do $jj=1,n,stridej$
  - do $ii=1,n,stridei$
    - do $j=jj,\min(n,jj+stridej-1)$
      - do $i=ii,\min(n,ii+stridei-1)$
        - $b(i,j) = a(j,i)$

- Good choices of $stridei$ and $stridej$ can improve performance by a factor of 5 or more

- But what are the choices of $stridei$ and $stridej$?
Results: Macbook O1
Results: Macbook O3
Results: Blue Waters O1
Results: Blue Waters O3

Simple, unblocked code compiled with O3 – 709MB/s
Compilers Can’t Do It All

• Even for very simple operations, the number of choices that a compiler faces for generating good code can overwhelm the optimizer

• Guidance by a human *expert* is required
  ♦ The programming system must not get in the way of the expert
  ♦ The programming system should make it easy to automate tasks under direction of an expert

• Also note that single code performance portability still not possible
  ♦ Just because it is desirable doesn’t make it a reasonable goal
The Challenges

• Times are changing; MPI is old (for a programming system)
• Can MPI remain relevant?
  ♦ For its core constituency?
  ♦ For new (to MPI) and emerging applications?
Weaknesses of MPI

• MPI
  ♦ Distributed Memory. No built-in support for user-distributions
    • Darray and Subarray don’t count
  ♦ No built-in support for dynamic execution
    • But note dynamic execution easily implemented in MPI
  ♦ 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, in part because HPC systems make the mistake of insisting on POSIX
Strengths of MPI

- **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
  - Process-oriented memory model encourages and provides mechanisms for performance
To Improve on MPI

- **Add what is missing:**
  - Distributed data structures (that the user needs)
    - This is what most parallel programming “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 predictable 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 productivity – reduces the “grain size” or degree of aggregation that the programmer must provide
    - We need to bring back $n^{1/2}$
The Future King

• MPI remains effective as an internode programming system
  ♦ Productivity gains come from writing libraries and frameworks on top of MPI
    • This was the original intention of the MPI Forum

• The real challenge will be in intranode programming...
Likely Exascale Architectures

Figure 2.1: Abstract Machine Model of an exascale Node Architecture

- **From “Abstract Machine Models and Proxy Architectures for Exascale Computing Rev 1.1,” J Ang et al**
Another Pre-Exascale Architecture

Sunway TaihuLight
• Heterogeneous processors (MPE, CPE)
• No data cache
Most Predict Heterogeneous Systems for both Ops and Memory

Table 1. Estimated Performance for Leadership-class Systems

<table>
<thead>
<tr>
<th>Year</th>
<th>Feature size</th>
<th>Derived parallelism</th>
<th>Stream parallelism</th>
<th>PIM parallelism</th>
<th>Clock rate GHz</th>
<th>FMAs</th>
<th>GFLOPS (Scalar)</th>
<th>GFLOPS (Stream)</th>
<th>GFLOPS (PIM)</th>
<th>Processor per node</th>
<th>Node (TFLOP)</th>
<th>Nodes per system</th>
<th>Total (PFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012</td>
<td>22</td>
<td>16</td>
<td>512</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>128</td>
<td>1,024</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>10,000</td>
<td>23</td>
</tr>
<tr>
<td>2020</td>
<td>12</td>
<td>54</td>
<td>1,721</td>
<td>0</td>
<td>2.8</td>
<td>4</td>
<td>1,210</td>
<td>4,819</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>20,000</td>
<td>241</td>
</tr>
<tr>
<td>2023</td>
<td>8</td>
<td>122</td>
<td>3,873</td>
<td>512</td>
<td>3.1</td>
<td>4</td>
<td>3,026</td>
<td>12,006</td>
<td>1,587</td>
<td>4</td>
<td>17</td>
<td>20,000</td>
<td>1,330</td>
</tr>
<tr>
<td>2030</td>
<td>4</td>
<td>486</td>
<td>15,489</td>
<td>1,024</td>
<td>4</td>
<td>8</td>
<td>31,104</td>
<td>61,956</td>
<td>8,192</td>
<td>16</td>
<td>101</td>
<td>20,000</td>
<td>32,401</td>
</tr>
</tbody>
</table>

Feature size is the size of a logic gate in a semiconductor, in nanometers. Derived parallelism is the amount of concurrency, given processor cores with a constant number of components, on a semiconductor chip of fixed size. Stream and PIM parallelism are the number of specialized processor cores for stream and processor-in-memory processing, respectively. FMA is the number of floating-point multiply-add units available to each processor core. From these values, the performance in GigaFLOPS is computed for each processor and node, as well as the total peak performance of a leadership-scale system.

Another estimate, from “CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences,” Slotnick et al, 2013
What This (might) Mean for MPI

- Lots of innovation in the processor and the node
- More complex memory hierarchy; no chip-wide cache coherence
- Tightly integrated NIC
- Execution model becoming more complex
  - Achieving performance, reliability targets requires exploiting new features
What This (might) Mean for Applications

• Weak scaling limits the range of problems
  ♦ Latency may be critical (also, some applications nearing limits of spatial parallelism)

• Rich execution model makes performance portability unrealistic
  ♦ Applications will need to be flexible with both their use of abstractions and their implementation of those abstractions

• Programmers will need help with performance issues, whatever parallel programming system is used
MPI is not a BSP system

- BSP = Bulk Synchronous Programming
  - Programmers *like* the BSP model, adopting it even when not necessary (see FIB)
  - Unlike most programming models, *designed* with a performance model to encourage *quantitative* design in programs
- MPI makes it easy to emulate a BSP system
  - Rich set of collectives, barriers, blocking operations
- MPI (even MPI-1) sufficient for dynamic adaptive programming
  - The main issues are performance and “progress”
  - Improving implementations and better HW support for integrated CPU/NIC coordination the answer
Many reasons to consider MPI+X

♦ Major: We always have:
  • MPI+C, MPI+Fortran

♦ Both C11 and Fortran include support of parallelism (shared and distributed memory)

Abstract execution models becoming more complex

♦ Experience has shown that the programmer must be given some access to performance features

♦ Options are (a) add support to MPI and (b) let X support some aspects
X = MPI (or X = \phi)

- MPI 3.0 features esp. important for Exascale
  - Generalize collectives to encourage post BSP programming:
    - Nonblocking collectives
    - Neighbor - including nonblocking - collectives
  - Enhanced one-sided (recall AMM targets)
    - Precisely specified (see “Remote Memory Access Programming in MPI=3,” Hoefler et al, in ACM TOPC)
    - Many more operations including RMW
  - Enhanced thread safety
X = Programming with Threads

• Many choices, different user targets and performance goals
  ♦ Libraries: Pthreads, TBB
  ♦ Languages: OpenMP 4, C11/C++11

• C11 provides an adequate (and thus complex) memory model to write portable thread code
  ♦ Also needed for MPI-3 shared memory
What are the Issues?

- Isn’t the beauty of MPI + X that MPI and X can be learned (by users) and implemented (by developers) independently?
  - Yes (sort of) for users
  - No for developers
- MPI and X must either partition or share resources
  - User must not blindly oversubscribe
  - Developers must negotiate
More Effort needed on the “+”

• MPI+X won’t be enough for Exascale if the work for “+” is not done very well
  ♦ Some of this may be language specification:
    • User-provided guidance on resource allocation, e.g., MPI_Info hints; thread-based endpoints
  ♦ Some is developer-level standardization
    • A simple example is the MPI ABI specification – users should ignore but benefit from developers supporting
Some Resources to Negotiate

- **CPU resources**
  - Threads and contexts
  - Cores (incl. placement)
  - Cache

- **Memory resources**
  - Prefetch, outstanding load/stores
  - Pinned pages or equivalent NIC needs
  - Transactional memory regions
  - Memory use (buffers)

- **NIC resources**
  - Collective groups
  - Routes
  - Power

- **OS resources**
  - Synchronization hardware
  - Scheduling
  - Virtual memory

MPI has already led the way in defining interlanguage compatibility, application binary interfaces, and resource manager/program interfaces.
Summary

• MPI remains the dominant system for massively parallel HPC because of its greatest common denominator approach and precisely defined programming models
• And because it doesn’t pretend to solve the really hard problem – general locality management and general intranode programming
• MPI is currently the internode programming system planned for the next two generations of US supercomputers
  ♦ And some argue for making it key to the intranode programming, leaving single core to the language/compiler
Thanks!