MPICH2: A High-Performance, Portable Implementation of MPI

William Gropp
Ewing Lusk

And the rest of the MPICH team:
Rajeev Thakur, Rob Ross, Brian Toonen, David Ashton,
Anthony Chan, Darius Buntinas

Mathematics and Computer Science Division
Argonne National Laboratory
Outline

• The Message Passing Interface standard
• The MPICH implementation of MPI
• Challenging aspects of MPI implementation
  • MPI-1
  • MPI-2
• MPICH2
• Related systems
• Development tools
• MPI implementation research
• Status
• Futures
MPI – A Successful Community Standard

• The Message Passing Interface standard
  • A specification, not an implementation
  • A library, not a compiler
  • For the (extended) message-passing model of parallel computation

• The MPI Forum
  • Everyone invited to participate, but lots of work involved
  • Ultimately about 30-50 computer scientists, application scientists (users), best technical people from nearly all vendors

• Two phases
  • MPI-1 – 1992-1994: MPI 1.2
  • MPI-2 – 1995-1997: MPI 2.0
    • Substantial extensions to MPI-1
Some Reasons MPI Has Been Successful

• Complete – many flavors of message passing are provided
  • Multiple send modes
  • Extensive collective operations
  • “One-sided” operations
• Allows maximum performance
  • E.g., does not require buffering, allows but does not require thread safety
• Small subset is easy to learn and use
• Easy to port earlier message-passing codes
• New ideas have proven useful, even beyond MPI itself
  • Communicators encapsulate process groups and contexts
  • Datatypes express non-contiguous, typed data
  • Profiling interface provides generalized access for tools
• Modularity provided by communicators allows encapsulation of MPI calls in libraries
• Not famously convenient to use, but
  • Can express all parallel algorithms
  • Libraries can hide complexity from users
The MPICH Implementation of MPI

- MPICH is a team effort at Argonne National Laboratory, with several collaborators elsewhere
  - Research goal – to conduct research in portable, high-performance parallel message-passing implementation issues, guided by the MPI standard
  - Software goal – to provide the community with a complete, open source, portable MPI (and MPI-2) implementation for use by everyone
    - Community = end users, library writers, vendors building specialized MPI implementations
- Both of these goals apply to a number of tools for parallel programming development, also described here.
- http://www.mcs.anl.gov/mpi/mpich
Message-Passing Implementation
Background

• p4 combined early shared-memory systems with sockets, supported heterogeneous systems
• PVM became a de facto standard for clusters of workstations, first Beowulf
• Parallel computing vendors built incompatible systems supporting similar programming model
  • Intel’s nx, IBM’s EUI, TMC’s CMMD, nCube
  • Some commercial portable systems – ParaSoft’s Express
• Chameleon created portability layer over high-performance vendor systems
• All participated in MPI Forum, and adopted MPI standard
• Early MPICH combined p4 and Chameleon for fast start, evolved along with standard itself
  • Provided early start for vendors
• Eventually multiple commercial, research, and publicly available implementations
  • E.g., CHIMP, BIP, LAM
Challenging Aspects of MPI implementation

- Basic message passing
  - high-bandwidth, low latency for all message lengths
  - Collective performance all sizes, topologies
  - Portability to various message fabrics
- New already in MPI-1
  - Extensive collective operations
  - Datatypes for heterogeneity, portability
  - Communicators for modularity
  - Portability across Unixes, embedded systems, Windows
- New in MPI-2
  - Dynamic process management (spawn, connect, accept, join)
  - Parallel I/O
  - One-sided (especially passive target, NIC-based)
  - Multiploee levels of thread safety
MPICH2

- All new implementation of MPI-1 and MPI-2
- Vehicle for research into MPI implementation issues
- Highly modular
  - Easy to exchange components
  - Easy to experiment with new algorithms and communication interconnects
- Portable
  - Most Unix-based systems + Windows
- High performance
  - Not just ping-pong—Threads, datatypes, collectives, …
- Highly scalable
  - To 100K+ processors (in theory)
- Friendly development environment
  - Extensive internal error checks
  - Clean integration with other tools, such as performance visualization
  - Primitive parallel debugging with MPD process manager
What’s New in MPICH2

• Beta-test version available for groups that expect to perform research on MPI implementations with MPICH2
  • Version 0.97 to be released soon
• Contains
  • All of MPI-1, MPI-I/O, services functions from MPI-2, all active-target RMA, passive target awaiting full thread-safety
  • C, C++, Fortran 77 bindings
  • Example devices for TCP, Infiniband, shared memory, TCP+shared memory, general RDMA
  • Documentation on individual MPI routines
• Passes extensive correctness tests
  • Intel test suite, as corrected; good unit test suite
  • MPICH test suite; adequate system test suite
  • Notre Dame C++ tests, based on IBM C test suite
  • Expanding MPICH2 test suite
MPICH2 Research

• MPICH2 is our vehicle for research in
  • Optimized MPI datatypes
  • New collective communication algorithms
  • Optimized Remote Memory Access (RMA)
  • Thread safety and efficiency (e.g., avoid thread locks)
  • High Scalability (64K MPI processes and more)
  • Exploiting Remote Direct Memory Access (RDMA) capable networks
  • All of MPI-2, including dynamic process management, parallel I/O, RMA
  • Fault tolerance issues
Some Target Platforms

- Clusters (TCP, UDP, Infiniband, Myrinet, Proprietary Interconnects, …)
- Clusters of SMPs
- Grids (UDP, TCP, Globus I/O, …)
- Cray Red Storm
- BlueGene/x
  - 64K processors; 64K address spaces
- QCDoC
- Other systems
Structure of MPICH-2
The Major Components

- **PMI**
  - Process Manager Interface
  - Provides scalable interface to both process creation and communication setup
  - Designed to permit many implementations, including with/without demons and with 3rd party process managers

- **ADIO**
  - I/O interface. No change from current ROMIO (except for error reporting and request management). Extensions to support high-performance I/O system; redesign to exploit RMA planned

- **ADI3**
  - New device interface aimed at higher performance networks and new network capabilities
The Layers

- **ADI3**
  - Full-featured interface, closely matched to MPI point-to-point and RMA operations
  - Most MPI communication routines perform (optional) error checking and then “call” ADI3 routine
  - Modular design allows replacement of parts, e.g.,
    - Datatypes
    - Topologies
    - Collectives

- **New Channel Interface**
  - Much smaller than ADI-3, easily implemented on most platforms
  - Nonblocking design is more robust and efficient than MPICH-1 version

- **MPI can be supported on a new platform (e.g. BG/L) by implementing either**
  - ADI-3 – maximum performance, specialization to hardware
  - Channel – minimum effort
Expose Structures To All Levels of the Implementation

- All MPI opaque objects are defined structs for all levels (ADI, channel, and lower)
  - No copying of data across layer boundaries
- All objects have a handle that includes the type of the object within the handle value
  - Permits runtime type checking of handles
  - Null handles are now distinct
  - Easier detection of misused values
  - Fortran Integer-valued handles simplify the implementation for 64-bit systems
- Consistent mechanism to extend definitions to support needs of particular devices
- Defined fields simplify much code
  - E.g., direct access to rank, size of communicators
Special Case: Predefined Objects

- Many predefined objects contain all information within the handle
  - Predefined MPI datatype handles contain
    - Size in bytes
    - Fact that the handle is a predefined datatype
  - No other data needed by most MPI routines
    - Eliminates extra loads, pointer chasing, and setup at MPI_Init time
  - Predefined attributes handled separately from general attributes
    - Special case anyway, since C and Fortran versions are different for the predefined attributes
- Other predefined objects initialized only on demand
  - Handle always valid
  - Data areas may not be initialized until needed
  - Example: names (MPI_Type_set_name) on datatypes
Built-in MPI Datatypes

- MPI_TYPE_NULL has datatype code in upper bits
- Index is used to implement MPI_Type_set/get_name
Channel “CH3”

- One possible implementation design for ADI3
  - Others possible and underway
- Thread safe by design
  - Requires* atomic operations
- Nonblocking design
  - Requires some completion handle, so
- Delayed allocation
  - Only allocate/initialize if a communication operation did not complete
Needs of an MPI Implementation

• Point-to-point communication
  • Can be implemented using a polling interface

• Cancel of send
  • Requires some agent (interrupt-driven receive, separate thread, guaranteed timer)

• Active target RMA
  • Can work with polling
  • Performance may require “bypass”

• Passive target RMA
  • Requires some agent
    • For some operations, the agent may be special hardware capabilities
An Example: CH3 Implementation over TCP

- Pollable and active-message data paths
- RMA Path
CH3 Summary

- Nonblocking interface for correctness and “0 copy” transfers
- struct iovec routines to provide “0 copy” for headers and data
- Lazy request creation to avoid unnecessary operations when data can be sent immediately (low latency case); routines to reuse requests during incremental transfers
- Thread-safe message queue manipulation routines
- Supports both polling and preemptive progress
CH3 Implementations

- Multiple implementations of the CH3 (also called Channel) device:
  - Sock – simple Sockets
    - Default device, suitable for most clusters
  - SSM – Sockets + Shared memory
    - Optimized for SMP clusters
  - SHM – Shared memory only
    - Primarily intended as an internal implementation to help in tuning performance
  - RDMA – Remote DMA
    - Simple example for devices with remote put/get
    - Does not include remote atomic operations
      - Required to achieve best performance and lowest latency
Early Results on Channel/TCP Device

- Conclusion: little added overhead over low-level communication
  - But will become more critical with high-performance network
MPICH2 Bandwidth
Sock channel over Gig-E

![Graph showing MPICH2 Bandwidth over message size (Bytes)]
MPICH2 Latency
Sock channel over Gig-E

- MPICH2: 0 Byte 30.87μs; 1 Byte 33.99μs
- TCP: 1 Byte 24.96μs
MPICH2 Bandwidth
Sock channel over Fast-E

![Graph showing MPICH2 Bandwidth vs Message Size (Bytes)]
MPICH2 Latency
Sock channel over Fast-E

- MPICH2: 0 Byte 61.12μs; 1 Byte 63.99μs
- TCP: 1 Byte 51.49μs
MPICH2 Bandwidth
GASNet channel over GM

Bandwidth (MBps)

Message Size (Bytes)
MPICH2 Latency
GASNet channel over GM

- MPICH2: 0 Byte 8.48µs; 1 Byte 8.85µs
- GM: 1 Byte 7.68µs
Thread Safety Issues

- **MPI levels**
  - Four levels of thread safety; most users will use either THREAD_FUNNELED or THREAD_MULTIPLE

- **Tradeoffs**
  - Performance concerns
  - Thread overhead is not zero:
    - Cost of ensuring atomic updates to data structures that may be shared among threads
      - Note all MPI objects (e.g., datatypes, requests) may be created in one thread and used in another
    - Cost of sharing external resources (e.g., network connections)
      - Depending on the implementation, may include a context switch to a communication service thread
Problems with Thread Programming Models

• Very low-level; easy to
  • Make a mistake (e.g., return and forget you’re holding a thread lock; enter a deadly embrace)
  • Add significant overhead

• Mismatch with desired semantics
  • Locking a data structure serializes access
  • Atomic updates to the data structure are usually all that is desired
  • Thread_lock/unlock overkill, expensive
Designing for Fast Thread Safety

- MPICH2 uses abstractions for implementing thread safety
  - Monitors used to control access to shared resources
  - Where possible, atomic updates to data structures use lock-free methods, exploiting special features of the processor (e.g., store-conditional/load reservation or compare-and-swap)

- Goals (we’re not there yet):
  - MPICH2, configured with -enable-thread=single, performs as well as the best polling-based implementation
  - Configured with -enable-thread=multiple, but with MPI_INIT_THREAD( requested, MPI_THREAD_FUNNELED), pays a tiny cost (immeasurable on commodity clusters)
  - With MPI_THREAD_MULITPLE, but only one thread per process, also pays a tiny cost, even on a uniprocessor system
Other Features

- mpiexec
  - MPICH2 supports mpiexec, as defined by the MPI-2 specification
  - mpirun supported for backward compatibility

- Profiling tools
  - MPE tools from MPICH1 enhanced and included with MPICH2
    - Multiple profiling libraries

- MPI-IO and ROMIO
  - Updated to use Generalized requests (MPI_Request, not MPIO_Request)
  - Interfaces exploit features of PVFS, PVFS2

- PMI
  - Modular interface to process manager allows the same executable to run with multiple process managers
  - Allows process to find one another in scalable way
The MPD Process Manager

- Provides scalable fast startup via pre-connected daemons
  - Persistent: no need to restart daemons for each job or session
  - Can run as root
- Implements the PMI interface for parallel programs
  - Helps processes find one another to make connections dynamically and lazily
  - Supports fast startup of even many-process programs to support interactivity
- Manages stdio scalably and conveniently
- Serves as back-end for the process manager component in the DOE Scalable Systems Software cluster management software suite
- Supports `mpigdb` as a debugger for parallel programs
Coercing gdb Into Functioning as a Primitive Parallel Debugger

- Key is control of stdin, stdout, stderr by MPD, through mpigdb
  - Replaces mpiexec on command line
- Stdout, stderr collected in tree, labeled by rank, and merged for scalability
  (0-9) (gdb) p x
  (0-2): $1 = 3.4
  (3): $1 = 3.8
  (4-9): $1 = 4.1
- Stdin can be broadcast to all or to a subset of processes
  - z 3 (to send input to process 3 only)
  - Same for interrupts
- Can run under debugger control, interrupt and query hung processes, parallel attach to running parallel job
Scalability in Performance Visualization Tools

- **Challenge:** Performance data may be large (big trace files) and complicated (processes, threads, messages, cpu’s), yet details are needed for *understanding*.

- **Scalability for trace data:** New SLOG uses “bounding box” approach to manage gigabyte-size log files produced by
  - MPI profiling library (any MPI implementation)
  - IBM systems logging (shows thread, individual cpu’s)

- **Scalability for visualization:** New Jumpshot uses “shadow objects” to display amalgamated messages and states.
  - Allows for both high-level understanding and analysis of detail
Viewing at Multiple Scales with Jumpshot

Each line represents 1000’s of messages

Detailed view shows opportunities for optimization
Development Tools

• Extensive automation in development:
  • Nightly tests and web-page summaries
  • Code coverage tests
  • Automatic creation of language binding wrappers
  • Build environment
    • Extensive use of GNU configure for portability
    • Can configure multiple devices, channels, process managers, PMI implementations
Nightly tests

- Tests run every night on a variety of systems
- Data summarized every morning on a web page
- Complete test output available for examination of problems
- 1887 separate test programs from 4 test suites:
  - MPICH1 (MPI1), MPICH2 (MPI1 and 2), Intel (MPI1), C++ (MPI1; derived from the IBM Test suite)
- Tests include
  - Common configuration options
  - Randomly chosen configuration options
Coverage tests

- Coverage tests run every night
- Common problem with coverage tests — too much is marked as uncovered
  - Error handling and reporting code
  - Debugging support (e.g., extra routines to print internal structures)
- MPICH2 code contains structured comments marking blocks that don’t need to be covered
- Coverage reports count only the code that we believe should be covered
- Acid test — Do the coverage tests help?
  - Yes! Adding tests for uncovered code has found bugs
Wrapper creation tools

• MPICH2 provides the C binding to MPI-1 and MPI-2. Other bindings are built using *wrapper generators*

• Why build the Fortran 77 and C++ wrappers automatically?
  • Automatic tool (working directly from mpi.h (for function prototypes) and data extracted directly from the standard document source files ensure correct bindings
  • Allows uniform correction to any problems discovered after the initial implementation (e.g., generation of multiple weak symbols for Fortran routine names)
  • Permits custom binding subsets to limit library/executable sizes (particularly important should we provide an MPI module for Fortran 90)
Other Development Tools

• All Makefile.in’s are created with a special tool, allowing simple descriptions of the necessary file dependencies
  • Provides more control and features than automake
• Documentation on each routine generated from a structured comment at the head of each MPI routine; same tool generates man pages for commands such as mpicc
Recent MPI Implementation Research

- Fast Datatype packing algorithms
- Fault Tolerance
- New collective algorithms
Fast datatype processing

• Create a simplified representation
  • Fewer options to deal with
  • Must maintain expressiveness (no flattening)!

• Avoid recursive processing
  • Eliminates function call overhead
  • Creates additional optimization opportunities (e.g. stack preloading)

• Use expressive leaf nodes in representation
  • Further eliminates function calls
  • Can leverage optimizations (e.g. vector copies)

• Optimize at type creation and just before use
  • MPI_TYPE_COMMIT

• See *Fast (and Reusable) Datatype Processing*, Rob Ross, Neill Miller, Bill Gropp
Datatype Performance

<table>
<thead>
<tr>
<th>Test</th>
<th>Manual (MB/sec)</th>
<th>MPICH2 (%)</th>
<th>MPICH (%)</th>
<th>LAM (%)</th>
<th>Size (MB)</th>
<th>Extent (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contig</td>
<td>1,156.40</td>
<td>97.2</td>
<td>98.3</td>
<td>86.7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Struct Array</td>
<td>1,055.00</td>
<td>107.0</td>
<td>107.0</td>
<td>48.6</td>
<td>5.75</td>
<td>5.75</td>
</tr>
<tr>
<td>Vector</td>
<td>754.37</td>
<td>99.9</td>
<td>98.7</td>
<td>65.1</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Struct Vector</td>
<td>746.04</td>
<td>100.0</td>
<td>4.9</td>
<td>19.0</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Indexed</td>
<td>654.35</td>
<td>61.3</td>
<td>12.7</td>
<td>18.8</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3D Face, XY</td>
<td>1,807.91</td>
<td>99.5</td>
<td>97.0</td>
<td>63.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>3D Face, XZ</td>
<td>1,244.52</td>
<td>99.5</td>
<td>97.3</td>
<td>79.8</td>
<td>0.25</td>
<td>63.75</td>
</tr>
<tr>
<td>3D Face, YZ</td>
<td>111.85</td>
<td>100.0</td>
<td>100.0</td>
<td>57.4</td>
<td>0.25</td>
<td>64</td>
</tr>
</tbody>
</table>

- Struct vector is similar to the struct example
  - Convenient way to describe N element vector
- Indexed test shows necessity of indexed node processing (though we should still do better!)
- Clear need for loop reordering in 3D YZ test
  - Should be able to beat straightforward hand-coded packing
Fault Tolerance in MPI Programs

• Fault tolerance in MPI programs requires cooperation from both the application and the MPI implementation

• MPI provides features for fault-tolerance within the standard
  • Classification of errors
  • Extensive control of errors through user-defined error handlers
  • Isolation of errors through separate communicators

• MPI Applications can be written to be fault-tolerant by isolating communication in separate communicators, so that a communicator may become invalid without causing entire program to fail.

• Dynamic process features of MPI-2 can help

• See “Fault Tolerance in MPI Programs” by Gropp and Lusk
Enhancing Collective Performance

- MPICH-1 collective algorithms are a combination of purely functional and minimum spanning tree (MST)
- Better algorithms, based on scatter/gather operations, exist for large messages
  - E.g., see van de Geijn for 1-D mesh
- And better algorithms, based on MST, exist for small messages
  - Correct implementations must be careful of MPI Datatypes
- Rajeev Thakur and Bill Gropp have developed and implemented algorithms for switched networks that provide much better performance
Broadcast – New Algorithm for Long Messages

- (Van de Geijn) Broadcast implemented as a scatter followed by an allgather

\[ T_{new} = (\lg p + p - 1)\alpha + 2 \frac{p-1}{p} n\beta \]

\[ T_{tree} = (\lg p)\alpha + (\lg p)n\beta \]

Van de Geijn algorithm is better for large messages and when \( \log p > 2 \)
Collective Performance

Broadcast performance

Allgather performance
Summary

• MPICH2 is an all-new implementation of the full MPI-2 standard
• Both a research vehicle and useful open-source software
• Beta version available now (0.96)
  • Saving “1.0” designation for when MPI-2 implementation is complete.
• In use now by users and vendors
  • Cray, IBM, Intel using MPICH2 as basis of MPI for next-generation systems
  • Many individual users
• Available from http://www.mcs.anl.gov/mpi/mpich
• Support from mpich2-maint@mcs.anl.gov
• Most users of MPICH1 should switch