Open MPI from a user perspective

George Bosilca
ICL – University of Tennessee
bosilca@cs.utk.edu
From Scratch?

- Merger of ideas from:
  - FT-MPI (U. of Tennessee)
  - LA-MPI (Los Alamos, Sandia)
  - LAM/MPI (Indiana U.)
  - PACX-MPI (HLRS, U. Stuttgart)

Open MPI
From Scratch?

- Each prior project had different strong points
  - Could not easily combine into one code base
- New concepts could not easily be accommodated in old code bases
- Easier to start over
  - Start with a blank sheet of paper
  - Many years of collective implementation experience
Open MPI Project Goals

- All of MPI (i.e., MPI-1 and MPI-2)
- Open source
  - Vendor-friendly license (BSD)
- Prevent “forking” problem
  - Community / 3rd party involvement
  - Production-quality research platform (targeted)
  - Rapid deployment for new platforms
- Shared development effort
Design Goals

• Extend / enhance previous ideas
  ▪ Message fragmentation / reassembly
  ▪ Design for heterogeneous environments
    • Multiple networks
    • Node architecture (data type representation)
  ▪ Automatic error detection / retransmission
  ▪ Process fault tolerance
Design Goals

• Design for a changing environment
  ▪ Hardware failure
  ▪ Resource changes
  ▪ Application demand (dynamic processes)

• Portable efficiency on any parallel resource
  ▪ Small cluster
  ▪ “Big iron” hardware
  ▪ Grid
  ▪ …
Implementation Goals

- All of MPI
- Low latency
  - E.g., minimize memory management traffic
- High bandwidth
  - E.g., stripe messages across multiple networks
- Production quality
- Thread safety and concurrency (MPI_THREAD_MULTIPLE)
Overall Architecture
Three Main Code Sections

• Open MPI layer (OMPI)
  - Top-level MPI API and supporting logic
• Open Run-Time Environment (ORTE)
  - Interface to back-end run-time system
• Open Portability Access Layer (OPAL)
  - Utility code (lists, reference counting, etc.)
• Dependencies - not layers
  - OMPI & ORTE & OPAL
Three Main Code Sections

- OMPI
- ORTE
- OPAL

Operating system
Open Portability Access Layer

OPAL
• Provide a common level of functionality on all operating systems
  ▪ Unified thread interface
  ▪ Unified mutexes, conditions interface
  ▪ Optimized atomic operations: increment, decrement, compare-and-swap
  ▪ Path control functions (Unix /, Windows \\)
  ▪ Linked Lists, double linked lists, trees, hash tables, arrays both with lock-free mechanism or with mutexes.
- High performance timers
- Processor and memory affinity
- Common interface for memory hooks
- Windows 32/64 abstraction layer
- The base of the Modular Component Architecture (MCA).
Components

• Formalized interfaces
  ▪ Specifies “black box” implementation
  ▪ Different implementations available at run-time
  ▪ Can compose different systems on the fly
Components

- Formalized interfaces
  - Specifies “black box” implementation
  - Different implementations available at run-time
  - Can compose different systems on the fly
Components

- Formalized interfaces
  - Specifies “black box” implementation
  - Different implementations available at run-time
  - Can compose different systems on the fly
Components in HPC

- Components traditionally associated with heavy-weight systems such as:
  - CORBA
  - COM
  - Java beans
- HPC needs much smaller / simpler / faster
- Components therefore only slowly being accepted by the HPC community
Advantages of Components

• Easily customize Open MPI
  - Clear interface for providing a new point-to-point communication channel
  - Don’t like our topology support? Write your own

• Maintenance
  - #if problem goes away
  - Code forks (hopefully) reduced

• Encourage 3rd party research
  - Release independently of Open MPI
  - Combine different 3rd party components
Disadvantages(?) of Components

• Traditionally, high overhead associated with component architectures
  - Multi-language support
  - Distributed object support
• Open MPI minimizes component overheads
  - Interface is C
  - No interprocess support
• Quantification of overhead necessary…
Component Overhead Analysis

• Sources of overhead
  ▪ Indirect function calls
  ▪ Extra memory references
  ▪ Abstraction “penalty”
• Can directly measure function call overhead
• Measuring memory reference and abstraction penalty directly difficult
• Really concerned about effect on application performance
Indirect Function Call Overhead
Open MPI and Components

- Modular Component Architecture (MCA)
- Logical progression of prior MPI component architecture research (LAM)
  - More component types
  - More services provided to components
  - Decentralized management
- End result is a “highly pluggable” MPI
Open MPI and Components

- Components are shared libraries
  - Central set of components in Open MPI installation tree
  - Users can also have components under $HOME
- Can add / remove components after install
  - No need to recompile / re-link MPI apps
  - Download / install new components
  - Develop new components safely
Component Benefits

- Stable, production quality environment for 3rd party researchers
  - Can experiment inside the MPI implementation
  - Small learning curve (learn a few components, not the entire implementation)
- Vendors can quickly roll out support for new platforms
  - Write a few components
3rd Party Components

• Independent development and distribution
  ▪ No need to be part of main Open MPI distribution
  ▪ No need to “fork” Open MPI code base
• Compose to create unique combinations
  ▪ A p2p-based collective component can utilize new ABC network p2p component
• Can distribute as open or closed source
Plugins for HPC (!)

Networks
- Shmem
- TCP
- OpenIB
- mVAPI
- GM
- MX

Run-time environments
- rsh/ssh
- SLURM
- PBS
- BProc
- Xgrid

Your MPI application
Plugins for HPC (!)

Networks
- Shmem
- TCP
- OpenIB
- mVAPI
- GM
- MX

Run-time environments
- rsh/ssh
- SLURM
- PBS
- BProc
- Xgrid

Your MPI application
- Shmem
- TCP
- rsh/ssh
Plugins for HPC (!)

Networks
- Shmem
- TCP
- OpenIB
- mVAPI
- GM
- MX

Run-time environments
- rsh/ssh
- SLURM
- PBS
- BProc
- Xgrid

Your MPI application
- Shmem
- TCP
- GM
- rsh/ssh
Plugins for HPC (!)

Networks
- Shmem
- TCP
- OpenIB
- mVAPI
- GM
- MX

Run-time environments
- rsh/ssh
- SLURM
- PBS
- BProc
- Xgrid

Your MPI application
- Shmem
- TCP
- GM
- rsh/ssh
Plugins for HPC (!)

Networks
- Shmem
- TCP
- OpenIB
- mVAPI
- GM
- MX

Run-time environments
- rsh/ssh
- SLURM
- PBS
- BProc
- Xgrid

Your MPI application
- Shmem
- TCP
- GM
- SLURM
Plugins for HPC (!)

Networks
- Shmem
- TCP
- OpenIB
- mVAPI
- GM
- MX

Run-time environments
- rsh/ssh
- SLURM
- PBS
- BProc
- Xgrid

Your MPI application
- Shmem
- TCP
- GM
- SLURM
Plugins for HPC (!)

Networks
- Shmem
- TCP
- OpenIB
- mVAPI
- GM
- MX

Your MPI application
- Shmem
- TCP
- GM

Run-time environments
- rsh/ssh
- SLURM
- PBS
- BProc
- Xgrid
Plugins for HPC (!)

Networks
- Shmem
- TCP
- OpenIB
- mVAPI
- GM
- MX

Run-time environments
- rsh/ssh
- SLURM
- PBS
- BProc
- Xgrid

Your MPI application
- Shmem
- TCP
- GM
- PBS
Plugins for HPC (!)

Networks
- Shmem
- TCP
- OpenIB
- mVAPI
- GM
- MX

Run-time environments
- rsh/ssh
- SLURM
- PBS
- BProc
- Xgrid

Your MPI application

- Shmem
- TCP
- GM
Open Run-Time Environment

ORTE
Seamless, transparent environment for high-performance applications

- Inter-process communications within and across cells
- Distributed publish/subscribe registry
  - Supports event-driven logic across applications, cells
  - Persistent, fault tolerant
- Dynamic “spawn” of processes, applications both within and across cells
Open Message Passing Interface

OMPI
OMPI layer manage:

- All MPI related interfaces:
  - Languages: C, C++, F77, F90
  - And their profiling interfaces
  - Requests, groups, data types, file
  - Generalized requests
  - MPI thread issues
Point-To-Point Communications

PML/PTL
PML/BTL
Point-to-Point Architecture

User application

MPI API

Architecture Services

Data types

Engine

PML

BTL

TCP/IP

Shared Mem

IB

Memory Pooling

Memory Management

Proc Private

Shared

Pinned

Pow 2 binning

Best fit
Data types
Data Representation

• Different across different machines
  ▪ Length: 32 vs. 64 bits (vs. …?)
  ▪ Endian: big vs. little

• Problems
  ▪ No standard about the data length in the programming languages (C/C++)
  ▪ No standard floating point data representation
    ▪ IEEE Standard 754 Floating Point Numbers
      ▪ Subnormals, infinities, NANs …
    ▪ Same representation but different lengths
MPI Datatypes

• MPI uses “datatypes” to:
  ▪ Efficiently represent and transfer data
  ▪ Minimize memory usage
• Even between heterogeneous systems
  ▪ Used in most communication functions (MPI_SEND, MPI_RECV, etc.)
  ▪ And file operations
• MPI contains a large number of pre-defined datatypes
Some of MPI’s Pre-Defined Datatypes

<table>
<thead>
<tr>
<th>MPI_Datatype</th>
<th>C datatype</th>
<th>Fortran datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short</td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
<td></td>
</tr>
<tr>
<td>MPI_FLOAT_LONG</td>
<td>float</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
<td>DOUBLE PRECISION*8</td>
</tr>
</tbody>
</table>
User-Defined Datatypes

• Applications can define unique datatypes
  ▪ Composition of other datatypes
  ▪ MPI functions provided for common patterns
    • Contiguous
    • Vector
    • Indexed
    • Structures

⇒ Always reduces to a type map of pre-defined datatypes
What About Performance?

- **Bad (old) way**
  - User manually copies data to a pre-allocated buffer, or
  - User calls MPI_PACK and MPI_UNPACK

- **Good (new) way**
  - Trust the [modern] MPI library
  - Uses high performance MPI “datatypes”
Performances

• **Worst case**: the most scattered data representation in memory (i.e., one byte per line of cache) leads to 80-85% of the optimal bandwidth starting from message of size 256 bytes.

• Usually, for HPL like data-types, Open MPI run at between 90 and 100% of the maximal bandwidth (depending on the size of the message).

• Up to 3 times faster than other MPI implementations, depending on the memory layout.
PML/BTL Architecture

- **Component Architecture:**
  - “Plug-ins” for different capabilities (e.g. different networks)
  - Tuneable run-time parameters

- **Three component frameworks:**
  - Point-to-point messaging layer (PML) implements MPI semantics
  - Byte Transfer Layer (BTL) abstracts network interfaces
  - Memory Pool (mpool) provides for memory management/registration
<table>
<thead>
<tr>
<th>System Configuration</th>
<th>Average Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open MPI - OpenIB - *optimized</td>
<td>5.13 usec</td>
</tr>
<tr>
<td>Open MPI - OpenIB - *defaults</td>
<td>5.43 usec</td>
</tr>
<tr>
<td>Open MPI - Mvapi - *optimized</td>
<td>5.64 usec</td>
</tr>
<tr>
<td>Open MPI - Mvapi - *defaults</td>
<td>5.94 usec</td>
</tr>
<tr>
<td>Mvapich - Mvapi (rdma/mem poll)</td>
<td>4.19 usec</td>
</tr>
<tr>
<td>Mvapich - Mvapi (send/recv)</td>
<td>6.51 usec</td>
</tr>
<tr>
<td>Open MPI - GM</td>
<td>6.86 usec</td>
</tr>
<tr>
<td>MPICH-GM</td>
<td>7.10 usec</td>
</tr>
<tr>
<td>Open MPI - Shared Memory</td>
<td>0.94 usec</td>
</tr>
<tr>
<td>MPICH2 - Shared Memory</td>
<td>1.07 usec</td>
</tr>
</tbody>
</table>

* Send/Recv based protocol
Bandwidth – TCP

OpenMPI over TCP (GigE)

Bandwidth in MBps vs Message Size in Bytes

NetPIPE - TCP
Open MPI
MPICH2
Bandwidth - Shared Memory

OpenMPI over Shared Memory

Bandwidth in MBps vs Message Size in Bytes

Open MPI
MPICH2
Bandwidth - GM

![Graph showing Bandwidth - GM](image)
Multi-Network Features

- High bandwidth
  - Stripe single message across different network types
- Low latency
  - Choose the best network for minimizing latency
- Dynamic environments
  - MPI-2 dynamic processes
  - Network failover / retransmission
Striped Message Example

Send Side

PML : TEG
First Fragment

IB - 1
IB - 2
GM

PML : TEG
Last Frags

IB - 1
IB - 2
GM
TCP-Gige

Receive Side

PML : TEG
First Fragment

IB-1

PML : TEG
Last Fragment

TCP-Gige
Collective Communication
Operations

• MPI defines several collective operations
  ▪ Some are rooted (e.g., broadcast)
  ▪ Others are rootless (e.g., barrier)

• “Collectives” generally refers to data-passing collective operations
  ▪ Although technically also refers to any action in MPI where all processes in a communicator must participate
  ▪ Example: communicator maintenance
Collective Operations

- Frequently used
- Numerous (14 MPI 1.2)
- Can significantly affect overall application performance

Collective operations should be optimized!
Collective Operation Optimization

- Numerous algorithms
- Different system architecture
  - Performance portability issue
- Different parameters for the same operation
  - Communicator size, data size, root node, operation to be applied computation...

=> Not trivial.
Performance of several reduce algorithms

Reduce on 24 nodes

Duration [μsec]

Message size [byte]

Linear
Binomial
Binary
Pipeline
2-Pipeline
4-Pipeline
8-Pipeline
How to detect the best ...

- Exhaustive testing
  - Measuring the performance of different algorithms with different parameters to determine optimal implementation for specified parameter range.
- Modelling collective operations
  - Using Parallel Communication Models to predict performance of different algorithms and select optimal one.
- Hybrid approaches
“Optimal” Implementation ...
OpenMPI collective modules

• Basic
  ▪ Linear and Logarithmic (Binomial/Binary) algorithms
  ▪ All collectives

• Tuned
  ▪ Bcast, Reduce, Allreduce, Barrier, Alltoall
  ▪ Numerous algorithms
  ▪ Decision function per communicator
Broadcast Performance

IMB 2.3 benchmark
Reduce Performance

IMB 2.3 benchmark
• High level profiling interface
• Allow detection of starting and ending events for all the MPI functions.

• Most of the time this information is not enough to understand the behaviour of the MPI application

MPI_Send

MPI_Recv
Peruse

- An MPI extension for revealing unexposed implementation information
- Similar to PAPI (who expose processor counters)
- A set of events tracing all the lifetime of an MPI request
- Extension for file access and collective communications
Dynamic Processes
Dynamic Processes

• So far, have discussed fixed sets of processes
  - Communicators
  - MPI_COMM_WORLD
• MPI-2 introduced dynamic processes
  - Create new MPI processes (spawn)
  - Join existing MPI processes
Optimizations Difficult

- Processes can come and go
- Connecting disjoint process sets across high-speed networks may be difficult
  - Potentially changes topologies
  - Requires resource [re-]allocation
  - Some implementations fall back to TCP
MPI 2 Dynamic processes

- Different techniques for increasing the number of processes in an MPI application:
  - spawn:
  - Connect/accept:
  - Join:

- Resources discovery and diffusion in order to allow the new universe to use the best available network.
• Discover the common interfaces
  • Ethernet and Myrinet switches
• Publish this information with the public registry
• Retrieve information about the remote universe
• Create the new universe
MPI and Threads
Other Complications: Threads

- Multiple threads concurrently performing message passing
  - Underlying MPI must serialize
  - Logically similar to a single thread posting non-blocking operations (but not just p2p)
- Challenging to maximize performance (e.g., minimize locks)
  - More details later on threading
Other Complications: Progress

• Many MPI implementations only provide synchronous progress
  ▪ Poll for message passing progress
  ▪ Occurs only when in MPI library
• Does not allow for much communication / computation overlap
• Difficult to implement one-sided communications
Asynchronous Progress

- Allow true message passing progress
  - Regardless of what user application is doing
- Three common approaches
  - Communication co-processors
  - OS / message passing system support
  - Progression threads
Myrinet, Infiniband, ToE, etc.
- MPI gives a buffer to the hardware
- Tells co-processor: “send”
- Can make progress even after MPI has returned (and user application is executing)

Does not help for MPI-level protocols
- RTS / CTS, fragmenting / re-assembling, etc.
- Can only make progress when in MPI library
- Limited overlap
Software Support

• Asynchronous progress provided by some other agent
  ▪ Operating system
  ▪ Hidden threads / external process
  ▪ May use hardware support as well
Progression Threads

- Hidden progression thread in MPI
  - Wakes up on demand for MPI-level protocols
  - Can do RTS / CTS, fragmenting / reassembling, etc.
  - Combined with other methods, can provide true asynchronous progress
- Some added cost of additional thread(s)
## Asynchronous Progress

<table>
<thead>
<tr>
<th>App</th>
<th>MPI implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>One thread</td>
<td>One thread</td>
</tr>
<tr>
<td>Multiple threads</td>
<td>Multiple threads</td>
</tr>
</tbody>
</table>
Threading

• Multi-threading can improve performance
  ▪ Better CPU utilization
  ▪ IO latency hiding
  ▪ Simplified logic (letting threads block)
• Most useful on SMPs
  ▪ Each thread can have its own CPU/core
• Overloading CPU’s can be ok
  ▪ Depends on application (e.g., latency hiding)
  ▪ Even on uniprocessors
Threads and MPI

• Extend the threaded model to multi-level parallelism
  ▪ Threads within an MPI process
  ▪ Possibly spanning multiple processors
  ▪ Allowing threads to block in communication

• Two kinds:
  ▪ Application-level threading
  ▪ Implementation-level threading
Application Level Threading

- Freedom to use blocking MPI functions
  - Allow threads to block in MPI_SEND / MPI_RECV
  - Simplify application logic
- Separate and overlap communication and computation
Implementation Threading

- Asynchronous communication progress
  - Allow communication “in the background”
  - Even while no application threads in MPI
- Can help single-threaded user applications
  - Non-blocking communications can progress independent of application
What About “One Big Lock”?  

- Put a mutex around MPI calls  
  - Only allow one application thread in MPI at any given time  
  - Allows a multi-threaded application to use MPI  
- Problem: can easily lead to deadlock  
  - Example  
    - Thread 1 calls MPI_RECV  
    - Thread 2 later calls matching MPI_SEND
Threads and MPI

- MPI does not define if a MPI process is a thread or an OS process
  - Threads are not addressable
  - MPI_SEND(…thread_id...) is not possible
- MPI-2 specification
  - Does not mandate thread support
  - Does define “Thread Compliant MPI”
  - Specifies four levels of thread support