Realizing Programming Models

- CAD
- Database
- Scientific modeling

- Parallel applications
  - Programming models
  - Communication abstraction
  - User/system boundary
  - Hardware/software boundary

- Multi-programming
- Shared address
- Message passing
- Data parallel

- Compilation or library
- Operating systems support

- Communication hardware

- Physical communication medium

Conceptual Picture

P1

Memory

Pn
What is message passing?

- **Data transfer plus synchronization**

  - Requires cooperation of sender and receiver
Quick review of MPI Message Passing

- **Basic terms**
  - **Nonblocking** - Operation does not wait for completion
  - **Synchronous** - Completion of send *requires* initiation (but not completion) of receive
  - **Ready** - *Correct* send requires a matching receive
  - **Asynchronous** - communication and computation take place simultaneously, *not* an MPI concept (implementations *may* use asynchronous methods)
Basic Send/Receive modes

- **MPI_Send**
  - Sends data. May wait for matching receive. Depends on implementation, message size, and possibly history of computation

- **MPI_Recv**
  - Receives data

- **MPI_Ssend**
  - Waits for matching receive

- **MPI_Rsend**
  - Expects matching receive to be posted
Nonblocking Modes

- **MPI_Isend**
  - Does not complete until send buffer available
- **MPI_Irsend**
  - Expects matching receive to be posted when called
- **MPI_Issend**
  - Does not complete until buffer available *and* matching receive posted
- **MPI_Irecv**
  - Does not complete until receive buffer available (e.g., message received)
Completion

- **MPI_Test**
  - Nonblocking test for the completion of a nonblocking operation
- **MPI_Wait**
  - Blocking test
- **MPI_Testall, MPI_Waitall**
  - For all in a collection of requests
- **MPI_Testany, MPI_Waitany**
- **MPI_Testsome, MPI_Waitsome**
Persistent Communications

- **MPI_Send_init**
  - Creates a request (like an MPI_Isend) but doesn’t start it
- **MPI_Start**
  - Actually begin an operation
- **MPI_Startall**
  - Start all in a collection
- **Also MPI_Recv_init, MPI_Rsend_init, MPI_Ssend_init, MPI_Bsend_init**
Testing for Messages

- **MPI_Probe**
  - Blocking test for a message in a specific communicator
- **MPI_Iprobe**
  - Nonblocking test
- **No way to test in all/any communicator**
Buffered Communications

- **MPI_Bsend**
  - May use user-defined buffer
- **MPI_Buffer_attach**
  - Defines buffer for all buffered sends
- **MPI_Buffer_detach**
  - Completes all pending buffered sends and releases buffer
- **MPI_Ibsend**
  - Nonblocking version of MPI_Bsend
Abstract Model of MPI Implementation

- **The MPI Abstraction**
  - Mechanism that implements MPI
  - Handles the coordination with the network
  - Polling, interrupt, shared processor implementations
- **Some mechanism must manage the coordination**
  - Polling - User’s process checks for MPI messages; low overhead
  - Interrupt - Processes respond “immediately” to messages; higher overhead but more responsive
  - Shared processor - Like constant polling
- **Combinations possible**
  - Polling with regular timer interrupts
  - Threads
- **MPI Implementation**
  - The protocols used to deliver messages
Message protocols

- Message consists of “envelope” and data
  - Envelope contains tag, communicator, length, source information
- Short
  - Message data (message for short) sent with envelope
- Eager
  - Message sent assuming destination can store
- Rendezvous
  - Message not sent until destination oks
Special Protocols for DSM

- Message passing is a good way to use distributed shared memory (DSM) machines because it provides a way to express memory locality
- Put
  - Sender puts to destination memory (user or MPI buffer). Like Eager
- Get
  - Receiver gets data from sender or MPI buffer. Like Rendezvous.
- Short, long, rendezvous versions of these
Dedicated Message Processor

The network transaction is processed by dedicated hardware resources consisting of a Communications or Message Processor (MP) and a Network Interface (NI).

- The MP can offload the protocol processing associated with the message passing abstraction. This may include the buffering, matching, copying, and the acknowledgement operations, as well as the remote read operation from a requesting node.
- The MPs communicate across the network and may cooperate to provide a global address space by providing a general capability to move data from one region of the global address space to another.
The Compute Processor (P) operates at the user level and is symmetric on the memory bus with the MP. This configuration is similar to a 2-way SMP with one processor (the MP) focused on communications, such that the two processors communicate via shared memory. The MP may support multiple types of communication (e.g., word-by-word, DMA) and can inspect portions of the message to determine how it should be directed. When a message is received by the MP, it can be passed to P by simply passing a data pointer. This design likely performance bound by the memory bandwidth.
Levels of Network Transaction

- **User Processor stores cmd / msg / data into shared output queue**
  - Must still check for output queue full (or make elastic)
- **Communication assists make transaction happen**
  - Checking, translation, scheduling, transport, interpretation
- **Effect observed on destination address space and/or events**
- **Protocol divided between two layers**
Message Processor Assessment

- **Concurrency Intensive**
  - Need to keep inbound flows moving while outbound flows stalled
  - Large transfers segmented
- Reduces overhead but added functions may increase latency
An alternative approach is for the MP to be integrated directly into the Network Interface. The approach is similar to the RNIC (RDMA NIC) approach to offload RDMA, TCP/IP, etc.
The Message Passing Interface (MPI) is the de facto standard for message passing parallel programming on large scale distributed systems
  - MPI is defined by a large committee of experts from industry and academia

One of the main goals of the MPI standard is to enable portability so that parallel applications run on small development platforms and larger productions systems
  - The MPI design was influenced by decades of best practices in parallel computing

While collectively referred to as the “MPI standard”, there are actually two documents MPI-1 and MPI-2
  - MPI-1 is the “core” set of MPI services for message passing providing abstractions and mechanisms for basic message passing between MPI processes
  - MPI-2 is a set of extensions and functionality beyond what is defined in MPI-1 such as dynamic process control and one-sided message passing

Implementations of the MPI standard provide message passing (and related) services for parallel applications
  - MPI actually defines a lot more services than just message passing, but the focus is passing messages between MPI processes

Many implementations of the MPI standard exist
Open MPI Overview

- A High Performance Message Passing Library
- Open MPI is a project combining technologies and resources from several other projects (e.g., FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI) in order to build the best MPI library available.
- A completely new MPI-2 compliant implementation
- Open MPI offers advantages for system and software vendors, application developers and computer science researchers.
- Open MPI is based on an open component architecture allowing modular replacement of many system components without recompilation.
Open MPI Overview

- The goals driving the effort include:
  - To write a maintainable, open source, production-quality MPI implementation
  - Emphasize high performance on a wide variety of platforms
  - Implement all of MPI-1 and MPI-2
    - including true support for multi-threaded MPI applications and asynchronous progress
  - Pool collective MPI implementation expertise and eliminate replicated effort between multiple MPI projects
  - Take only the best ideas from prior projects
  - Provide a flexible MPI implementation suitable for a wide variety of run-time environments and parallel hardware
  - Create a platform that enable world-class parallel computing research
  - Enable the greater HPC community to contribute
Open MPI Overview

• The organizations contributing to Open MPI are:
  – Indiana University (LAM/MPI)
  – University of Tennessee (FT-MPI)
  – Los Alamos National Laboratory (LA-MPI)

• Additional collaborators include:
  – Sandia National Laboratories
  – High Performance Computing Center at Stuttgart

• These developers bring many years of combined experience to the project
## Open MPI Overview

Features implemented or in short-term development for Open MPI include:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full MPI-2 standards conformance</td>
<td>Many OS’s supported (32 and 64 bit)</td>
</tr>
<tr>
<td>Thread safety and concurrency</td>
<td>Production quality software</td>
</tr>
<tr>
<td>Dynamic process spawning</td>
<td>Portable and maintainable</td>
</tr>
<tr>
<td>High performance on all platforms</td>
<td>Tunable by installers and end-users</td>
</tr>
<tr>
<td>Reliable and fast job management</td>
<td>Extensive user and installer guides</td>
</tr>
<tr>
<td>Network and process fault tolerance</td>
<td>Internationalized error messages</td>
</tr>
<tr>
<td>Support network heterogeneity</td>
<td>Component-based design, documented APIs</td>
</tr>
<tr>
<td>Single library supports all networks</td>
<td>CPAN-like tool for component management</td>
</tr>
<tr>
<td>Run-time instrumentation</td>
<td>Active, responsive mailing list</td>
</tr>
<tr>
<td>Many job schedulers supported</td>
<td>Open source license based on the BSD license</td>
</tr>
</tbody>
</table>

- The majority of clusters use some shared memory, Infiniband, quadrics, Myrinet networking, and/or some form of TCP/IP (e.g., Ethernet)
- Open MPI natively supports all of these network types and can use them simultaneously
- For example, if a message is sent from one MPI process to another on the same node, shared memory will be used. However, if a message is sent to a different node, the best available network will be used
Open MPI is fundamentally based on the MPI Component Architecture (MCA) which is a collection of component frameworks that provide services to Open MPI at run-time.

Each framework provides a single API for its services with different implementations of this API being called components. A component paired with a resource is a module.

For example, a process running on a compute node that contains two Gigabit Ethernet cards may have two modules of the TCP/IP component in the point-to-point transfer framework.
The following is an abbreviated list of the MPI layer component frameworks in Open MPI

- **coll**: MPI collective algorithms. Provide back-end implementations for MPI_BARRIER, MPI_BCAST, etc.
- **io**: MPI-2 I/O functionality. Currently only supports the ROMIO MPI-2 IO implementation from Argonne National Labs
- **one**: MPI-2 one-sided operations. Provide back-end implementations for MPI_GET, MPI_PUT, etc. This framework isn’t included in Open MPI’s first release.
- **op**: Collective reduction operations. Provide optimized implementations of the intrinsic MPI reduction operations, such as MPI_SUM, MPI_PROD, etc. This framework isn’t included in Open MPI’s first release.
- **pml**: MPI point-to-point management layer (PML). This framework is the top layer in point-to-point communications; the PML is responsible for fragmenting messages, scheduling fragments to PTL modules, high level flow control, and reassembling fragments that arrive from PTL modules.
- **ptl**: MPI point-to-point transport layer (PTL). This framework is the bottom layer in point-to-point communications; the PTL is responsible for communicating across a specific device or network protocol (e.g., TCP, shared memory, Elan4, OpenFabric, etc.)
- **topo**: MPI topology support. Provide back-end implementations of all the topology creation and mapping functions
Components are implemented as shared libraries; hence, using components means searching directories and loading dynamic libraries which is transparently handled by the MCA.

Extending Open MPI’s functionality is simply a matter of placing components in the directories that Open MPI searches at run-time.

For example, adding support for a new network type entails writing a new PTL component and placing its resulting shared library in Open MPI’s component directory. Thus MPI applications will instantly have access the component and be able to use it at run-time.

Therefore, the MCA enable:

- Run-time decisions about which components can be used
- Enables third parties to develop and distribute their own components
Open MPI Summary

- Many organizations are interested in a high quality implementation of MPI, since much of today’s parallel computing is done with MPI.
- Open MPI is easily leveraged by writing one or more components as “plug-ins” and is the first project of Open HPC (http://www.OpenHPC.org/) established for the promotion, dissemination, and use of open source software in high-performance computing.
Cray (Octigabay) Blade Architecture (Revisited)

- MPI offloaded in hardware throughput 2900 MB/s and latency 1.6us
- Processor and communication interface is Hyper Transport
- Dedicated link and communication chip per processor
- FPGA Accelerator available for additional offload
Cray Blade Architecture (Revisited)

- Six blades per 3U shelf
- Twelve 4x IB external links for primary switch
- An additional twelve links are available with optional redundant switch
Network Offload Technology (Revisited)

RDMA over TCP/IP Reference Models

RNIC Model

- Remote Direct Memory Access Protocol (RDMA)
- Markers with PDU Alignment (MPA)
- Direct Data Placement (DDP)
- Transmission Control Protocol (TCP)
- Internet Protocol (IP)
- Media Access Control (MAC)

Physical

RNIC Model shown with SCSI application

- Internet SCSI (iSCSI)
  - iSCSI Extensions for RDMA (iSER)
- Remote Direct Memory Access Protocol (RDMA)
- Markers with PDU Alignment (MPA)
- Direct Data Placement (DDP)
- Transmission Control Protocol (TCP)
- Internet Protocol (IP)
- Media Access Control (MAC)

Physical

RNIC Model shown with MPI application

- MPI application
  - MPI Component Architecture
- Remote Direct Memory Access Protocol (RDMA)
- Markers with PDU Alignment (MPA)
- Direct Data Placement (DDP)
- Transmission Control Protocol (TCP)
- Internet Protocol (IP)
- Media Access Control (MAC)

Physical
Example and Background Material
```c
#include <stdio.h>
#include <string.h>     // this allows us to manipulate text strings
#include "mpi.h"        // this adds the MPI header files to the program

int main(int argc, char* argv[]) {
    int my_rank;        // process rank
    int p;             // number of processes
    int source;         // rank of sender
    int dest;           // rank of receiving process
    int tag = 0;        // tag for messages
    char message[100];  // storage for message
    MPI_Status status;  // stores status for MPI_Recv statements

    // starts up MPI
    MPI_Init(&argc, &argv);
    // finds out rank of each process
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    // finds out number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank!=0) {
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0; // sets destination for MPI_Send to process 0
        // sends the string to process 0
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    } else {
        for(source = 1; source < p; source++){
            // receives greeting from each process
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s
", message); // prints out greeting to screen
        }
    }
    MPI_Finalize(); // shuts down MPI
    return 0;
}
```
Compiling and running

- **Head file**
  - Fortran -- mpif.h
  - C -- mpi.h (*we use C in this presentation)

- **Compile:**
  - implementation dependent. Typically requires specification of header file directory and MPI library.
  - SGI: cc source.c -lmpi

- **Run:**
  - mpirun -np <# proc> <executable>
Result

- cc hello.c -lmpi
- mpirun -np 6 a.out
  Greetings from process 1!
  Greetings from process 2!
  Greetings from process 3!
  Greetings from process 4!
  Greetings from process 5!
Startup and endup

- **int MPI_Init(int *argc, char ***argv)**
  - The first MPI call in any MPI process
  - Establishes MPI environment
  - One and only one call to MPI_INIT per process

- **int MPI_Finalize(void)**
  - Exiting from MPI
  - Cleans up state of MPI
  - The last call of an MPI process
Point to point communication

• Basic communication in message passing libraries
  – Send(dest, tag, addr, len), Recv(src, tag, addr, len)
  – Src/dest: integer identifying sending/receiving processes.
  – Tag: integer identifying message
  – (addr, len): communication buffer, contiguous area.

• MPI extensions.
  – Messages are typed: supports heterogeneous computing.
  – Buffers need not be contiguous: supports scatter/gather.
  – Non-interfering communication domains: Used for scoping of communication and process name space.
Point to point communication cont’d...

- MPI_Send(start, count, datatype, dest, tag, comm)
- MPI_Recv(start, count, datatype, source, tag, comm, status)
  - Start: buffer initial address
  - Count: (maximum) number of elements received.
  - Datatype: a descriptor for type of data items received; can describe an arbitrary (noncontiguous) data layout in memory.
  - Source: rank within communication group; can be MPI_ANY_SOURCE
  - Tag: Integer message identifier; can be MPI_ANY_TAG
  - Communicator:
    » specify an ordered group of communicating processes.
    » specify a distinct communication domain. Message sent with one communicator can be received only with “same” communicator.
  - Status: provides information on completed communication.
• Message = data + envelope
• MPI_Send(startbuf, count, datatype, dest, tag, comm)
MPI data

• startbuf (starting location of data)
• count (number of elements)
  – receive count $\geq$ send count
• datatype (basic or derived)
  – receiver datatype = send datatype (unless MPI_PACKED)
  – Specifications of elementary datatypes allows heterogeneous communication.
Datatype

- **MPI Datatype C Datatype**
  - MPI_CHAR
  - MPI_SHORT
  - MPI_INT
  - MPI_LONG
  - MPI_UNSIGNED_CHAR
  - MPI_UNSIGNED_SHORT
  - MPI_UNSIGNED
  - MPI_UNSIGNED_LONG
  - MPI_FLOAT
  - MPI_DOUBLE
  - MPI_LONG_DOUBLE
  - MPI_BYTE
  - MPI_PACKED

- **Derived datatypes**
  - mixed datatypes
  - contiguous arrays of datatypes
  - strided blocks of datatypes
  - indexed array of blocks of datatypes
  - general structure

- **Datatypes are constructed recursively.**
Functions to create new types

- **MPI_Type_contiguous(count, old, new)**
  - define a new MPI type comprising count contiguous values of type old

- **MPI_Type_commit(type)**
  - commit the type - must be called before the type can be used

**Derived types routines**

- `MPI_Type_commit`
- `MPI_Type_contiguous`
- `MPI_Type_count`
- `MPI_Type_extent`
- `MPI_Type_free`
- `MPI_Type_hindexed`
- `MPI_Type_hvector`
- `MPI_Type_lb`
- `MPI_Type_struct`
- `MPI_Type_vector`
- `MPI_Type_indexed`
- `MPI_Type_indexed`
- `MPI_Type_size`
- `MPI_Type_ub`
MPI envelope

- **destination or source**
  - rank in a communicator
  - receive = sender or MPI_ANY_SOURCE
- **tag**
  - integer chosen by programmer
  - receive = sender or MPI_ANY_TAG
- **communicator**
  - defines communication "space"
  - group + context
  - receive = send
• MPI provides groups of processes
  – initial all group
  – group management routines (build, delete groups)
• A context partitions the communication space.
• A message sent in one context cannot be received in another context.
• Contexts are managed by the system.
• A group and a context are combined in a communicator.
• Source/destination in send/receive operations refer to rank in group associated with a given communicator
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Group_size</td>
<td>returns number of processes in group</td>
</tr>
<tr>
<td>MPI_Group_rank</td>
<td>returns rank of calling process in group</td>
</tr>
<tr>
<td>MPI_Group_compare</td>
<td>compares group members and group order</td>
</tr>
<tr>
<td>MPI_Group_translate_ranks</td>
<td>translates ranks of processes in one group to those in another group</td>
</tr>
<tr>
<td>MPI_Comm_group</td>
<td>returns the group associated with a communicator</td>
</tr>
<tr>
<td>MPI_Group_union</td>
<td>creates a group by combining two groups</td>
</tr>
<tr>
<td>MPI_Group_intersection</td>
<td>creates a group from the intersection of two groups</td>
</tr>
</tbody>
</table>
Group routines ...

- MPI_Group_difference creates a group from the difference between two groups
- MPI_Group_incl creates a group from listed members of an existing group
- MPI_Group_excl creates a group excluding listed members of an existing group
- MPI_Group_range_incl creates a group according to first rank, stride, last rank
- MPI_Group_range_excl creates a group by deleting according to first rank, stride, last rank
- MPI_Group_free marks a group for deallocation
Communicator routines

- `MPI_Comm_size` returns number of processes in communicator's group
- `MPI_Comm_rank` returns rank of calling process in communicator's group
- `MPI_Comm_compare` compares two communicators
- `MPI_Comm_dup` duplicates a communicator
- `MPI_Comm_create` creates a new communicator for a group
- `MPI_Comm_split` splits a communicator into multiple, non-overlapping communicators
- `MPI_Comm_free` marks a communicator for deallocation
The End