Lecture 2

Message Passing
Using MPI

(Foster Chapter 8)

Outline

- Background
  - The message-passing model
  - Origins of MPI and current status
  - Sources of further MPI information
- Basics of MPI message passing
  - HiLO, World
  - Fundamental concepts
  - Simple examples in Fortran and C
- Extended point-to-point operations
  - non-blocking communication
  - modes
- Advanced MPI topics
  - Collective operations
  - More on MPI data types
  - Application topologies
  - The profiling interface
- Toward a portable MPI environment

The Message-Passing Model

- A process is (traditionally) a program counter and address space.
- Processes may have multiple threads
  - program counters and associated stacks
  - sharing a single address space.
- MPI is for communication among processes
  - separate address spaces
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process's address space to another's.
Types of Parallel Computing Models

- **Data Parallel**
  - the same instructions are carried out simultaneously on multiple data items (SIMD)
- **Task Parallel**
  - different instructions on different data (MIMD)
- **SPMD** (single program, multiple data)
  - not synchronized at individual operation level
- **SPMD** is equivalent to MIMD since each MIMD program can be made SPMD (similarly for SIMD, but not in practical sense)

Message passing (and MPI) is for MIMD/SPMD parallelism. HPF is an example of a SIMD interface.

Message Passing

- **Basic Message Passing:**
  - **Send**: Analogous to mailing a letter
  - **Receive**: Analogous to picking up a letter from the mailbox
  - **Scatter-gather**: Ability to "scatter" data items in a message into multiple memory locations and "gather" data items from multiple memory locations into one message
- **Network performance:**
  - **Latency**: The time from when a Send is initiated until the first byte is received by a Receive.
  - **Bandwidth**: The rate at which a sender is able to send data to a receiver.

Scatter-Gather

- **Scatter (Receive)**
- **Gather (Send)**

Message passing (and MPI) is for MIMD/SPMD parallelism. HPF is an example of a SIMD interface.
Basic Message Passing Issues

- Issues include:
  - **Naming**: How to specify the receiver?
  - **Buffering**: What if the output is not available? What if the receiver is not ready to receive the message?
  - **Reliability**: What if the message is lost in transit? What if the message is corrupted in transit?
  - **Blocking**: What if the receiver is ready to receive before the sender is ready to send?

Cooperative Operations for Communication

- message passing approach ➔ cooperative exchange of data
- data explicitly sent by one process and received by another
- Advantage: any change in receiving process’s memory is made with receiver’s explicit participation
- Communication and synchronization are combined
  - Push model (active data transfer)

One-Sided Operations for Communication

- One-sided operations b/w processes include remote memory reads and writes
- Only one process needs to explicitly participate
- An advantage is that communication and synchronization are decoupled
- One-sided operations are part of MPI-2.
  - Pull model (passive data transfer) for get

Process 0

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(data)</td>
<td>Receive(data)</td>
</tr>
</tbody>
</table>

Process 1

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put(data)</td>
<td>Get(data)</td>
</tr>
<tr>
<td>(memory)</td>
<td>(memory)</td>
</tr>
</tbody>
</table>
Collective Communication

- More than two processes involved in communication
  - Barrier
  - Broadcast (one-to-all), multicast (one-to-many)
  - All-to-all
  - Reduction (all-to-one)

Barrier

Broadcast and Multicast
All-to-All

CSC 591C

Reduction

CSC 591C

What is MPI?

CSC 591C
**MPI Sources**

- **Standard:** [http://www.mpi-forum.org](http://www.mpi-forum.org)
- **Books:**
  - Designing and Building Parallel Programs, by Ian Foster, Addison-Wesley, 1995.
  - Parallel Programming with MPI, by Peter Pacheco, Morgan-Kaufmann, 1997.
- **Other information on Web:** [http://www.mcs.anl.gov/mpi](http://www.mcs.anl.gov/mpi)

---

**MPI History**

- 1990 PVM: Parallel Virtual Machine (Oak Ridge Nat'l Lab)
  - Message-passing routines
  - Execution environment (spawn + control parallel processes)
  - No an industry standard
- 1992 meetings (Workshop, Supercomputing'92)
- 1993 MPI draft
- 1994 MPI Forum (debates)
- 1994 MPI-1.0 release (C & Fortran bindings) + standardization
- 1995 MPI-1.1 release
- 1997 MPI-1.2 release (error) +
  - MPI-2 release (new features, C++ & Fortran 90 bindings)
- ??? MPI-3 release (new: FT, hybrid, p2p, RMA, ...)

---

**Why Use MPI?**

- MPI provides a powerful, efficient, and portable way to express parallel programs
- MPI was explicitly designed to enable libraries...
- ... which may eliminate the need for many users to learn (much of) MPI
- It's the industry standard!
A Minimal MPI Program

In C:
```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    printf("Hello, world\n");
    MPI_Finalize();
    return 0;
}
```

In Fortran:
```
program main
use MPI
integer ierr

call MPI_INIT(ierr)
print *, 'Hello, world'
call MPI_FINALIZE(ierr)
```

Notes on C and Fortran

- C and Fortran bindings correspond closely
- In C:
  - `mpi.h` must be included
  - MPI functions return error codes or MPI_SUCCESS
- In Fortran:
  - `mpi.f` must be included, or use MPI module (MPI-2)
  - All MPI calls are to subroutines, with a place for the return code in the last argument
- C++ bindings, and Fortran-90 issues, are part of MPI-2.

Error Handling

- By default, an error causes all processes to abort.
- The user can cause routines to return (with an error code) instead.
  - In C++, exceptions are thrown (MPI-2)
- A user can also write and install custom error handlers.
- Libraries might want to handle errors differently from applications.
Running MPI Programs

- The MPI-1 Standard does not specify how to run an MPI program (just as the Fortran standard does not specify how to run a Fortran program).
- In general, starting an MPI program is dependent on the implementation of MPI you are using:
  - might require scripts, program arguments, and/or environment variables
- `mpirun <args>` is part of MPI-2, as a recommendation, but not a requirement:
  - You can use `mpirun/mpiexec` for MPI-1.

Finding Out About the Environment

- Two important questions that arise in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - `MPI_Comm_size` reports the number of processes.
  - `MPI_Comm_rank` reports the rank, a number between 0 and size-1, identifying the calling process.

Better Hello (C)

```c
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("I am #d of #d\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

CSC 591C
Better Hello (Fortran)

program main
use MPI
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end

MPI Basic Send/Receive

- We need to fill in the details in
  
  Process 0
  | Send(data) |
  |
  Process 1
  | Receive(data) |

- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

What is message passing?

- Data transfer plus synchronization
  - if it is blocking message passing

- Requires cooperation of sender and receiver
  - Cooperation not always apparent in code
Some Basic Concepts

- Processes can be collected into groups.
- Each message is sent in a context and must be received in the same context
  - Tag relative to context (discussed later)
- A (group, context) form a communicator.
- A process is identified by its rank in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called MPI_COMM_WORLD.

MPI Datatypes

- Data in a message described by a triple
  - (address, count, datatype), where
  - An MPI datatype is recursively defined as:
    - predefined, corresponding to a data type from the language
      (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
    - a contiguous array of MPI datatypes
    - a stride block of datatypes
    - an indexed array of blocks of datatypes
    - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, such as an array of (int, float) pairs, or a row of a matrix stored columnwise.

MPI Tags

- Messages sent with an accompanying user-defined integer tag
  - to assist the receiving process in identifying the message
- Messages can be screened (filtered) at the receiving end
  - by specifying a specific tag,
  - or not screened by specifying MPI_ANY_TAG as the tag
- Note: Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes.
MPI Basic (Blocking) Send

\texttt{MPI\_SEND} (start, count, datatype, dest, tag, comm)

- message buffer is described by (start, count, datatype)
- target process is specified by dest
  - rank of target process in communicator specified by comm
- When this function returns, the data has been delivered
  - buffer can be reused
  - but may not have been received by target process (yet)

CSC 591C

---

MPI Basic (Blocking) Receive

\texttt{MPI\_RECV} (start, count, datatype, source, tag, comm, status)

- waits until a matching (on source and tag) message is received
  - buffer can be used
- source is rank in communicator specified by comm or \texttt{MPI\_ANY\_SOURCE}
- status contains further information
- Receiving fewer than count occurrences of datatype is OK
  - but receiving more is an error

CSC 591C

---

Retrieving Further Information

- Status is a data structure allocated in the user's program.
  - In C
    \begin{verbatim}
    int send_tag, send_from, send_count;
    MPI_Status status;
    MPI_Status(...) MPI_ANY_SOURCE, MPI_ANY_TAG, ..., status );
    send_tag = status.MPI_TAG;
    send_from = status.MPI_SOURCE;
    MPI_Send(...); int status, datatype, sendcount );
\end{verbatim}
  - In Fortran
    \begin{verbatim}
    integer send_tag, send_from, send_count
    integer status(MPI_STATUS_SIZE)
    call MPI\_SEND\_STATUS(...) MPI\_ANY\_SOURCE, MPI\_ANY\_TAG, ..., status, ierr)
    tag\_send = status.MPI\_TAG
    send\_from = status.MPI\_SOURCE
    call MPI\_SEND\_STATUS(status, datatype, sendcount, ierr)
    \end{verbatim}

CSC 591C
Simple Fortran Example

```fortran
program main
  use mpi
  integer rank, size, myx, myy, tag10
  integer source, dest
  integer st_source, st_tag, st_source(MPI_STATUS_IGNORE)
  double precision data(10)
  call mpi_init(ierr)
  call mpi_comm_rank(MPI_COMM_WORLD, rank, ierr)
  call mpi_comm_size(MPI_COMM_WORLD, size, ierr)
  print *, "Source", rank, " of ", size
  st_source = rank
  st_tag = 10
  call mpi_sendrecv(|
```

Why Datatypes?

- Since all data is labeled by type, an MPI implementation can support communication between processes on machines with very different memory representations and lengths of elementary datatypes (heterogeneous communication)
- Specifying application-oriented layout of data in memory
  - reduces memory-to-memory copies in the implementation
  - allows the use of special hardware (scatter/gather) when available

Basic C Datatypes in MPI

<table>
<thead>
<tr>
<th>MPI_Type</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_BYTE</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI醮long</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI醮long_int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI醮long_int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI醮long_int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI醮long_int</td>
<td>signed long int</td>
</tr>
</tbody>
</table>
Tags and Contexts

- Separation of msgs used to be accomplished by use of tags, but
  - requires libraries to be aware of tags used by other libraries
  - can be defeated by use of "wild card" tags
- Contexts are different from tags
  - no wild cards allowed
  - allocated dynamically by the system when a library sets up a communicator for its own use
- User-defined tags still provided in MPI for user convenience in organizing application
- Use MPI_Comm_split to create new communicators

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - MPI_INIT
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Send
  - MPI_Recv
- Point-to-point (send/recv) isn’t the only way...

Introduction to Collective Operations in MPI

- Collective ops are called by all processes in a communicator.
  - No tags
  - Blocking
- MPI_BCAST distributes data from one process (the root) to all others in a communicator.
- MPI_REDUCE/ALLREDUCE combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, SEND/RECEIVE can be replaced by BCAST/REDUCE, improving both simplicity and efficiency.
- Others:
  - MPI_[ALL]SCATTERV[/ALL]GATHERV
Example: PI in C

```c
#include "mp.h"
#include <math.h>
int main(int argc, char *argv[]) {
    int done = 0, n, p, r, m, s, i;
    double PI = 3.14159265358979323846264338;
    char *errstr;
    MP_Frac_t h, m, s, i;
    MP_2exp_t x, y, lim, x2;
    double eps = 1.0E-14;
    MP_Frac_set(&m, 1.0);  // 1/1
    MP_Frac_set(&h, PI);  // PI/1
    MP_Frac_set(&x2, 1.0);  // 1/1
    MP_Frac_set(&x, PI);  // PI/1
    MP_Frac_set(&y, 1.0);  // 1/1
    MP_Frac_set(&lim, 1.0);  // 1/1
    h = 1.0;  // (double) a
    s = 0.0;  // err (mpfr_t, errn; isemaphore)
    while (h > 1.0E-14 && done == 0) {
        double eps = h * 1.0E-14;
        MP_Frac_sub(&m, &h, &m);  // PI - PI/1
        MP_Frac_div(&y, &m, &y);  // PI/1 - PI/1
        MP_Frac_add(&x, &x, &y);  // PI + PI/1
        MP_Frac_abs(&x2, &x2);  // PI/1
        MP_Frac_abs(&y, &y);  // PI
        MP_Frac_sub(&x, &x2, &x);  // PI/1 - PI
        MP_Frac_abs(&x, &x);  // PI
        MP_Frac_add(&x, &x, &x);  // PI/1 + PI/1 + PI/1
        MP_Frac_abs(&x, &x);  // PI
        MP_Frac_div(&lim, &lim, &eps);  // 1/1
        MP_Frac_sub(&x, &x, &lim);  // PI/1 - PI
        MP_Frac_abs(&s, &s);  // err
        MP_Frac_add(&s, &s, &s);  // err + err + err
        MP_Frac_abs(&s, &s);  // err
        MP_Frac_sub(&h, &h, &s);  // PI - err
        MP_Frac_abs(&h, &h);  // err
        done = (h < 1.0E-14);
    }
    printf("Done...
    ");
    return 0;
}
```

Approximation of Pi

Integration to evaluate $\pi$

```
\text{Compute approximation to title using numerical integrations}
\int_0^1 \sqrt{x(1-x)} \, dx
\]
```

Reduction

```
\text{sum} = 0
\text{for } i = 1 \text{ to } p \text{ do}
\text{sum} = \text{sum} + A(i)
```

CSC 591C
Example: PI in Fortran

```
Example: PI in Fortran

```}

Alternative 6 Functions for Simplified MPI

- MPI_INIT
- MPI_FINALIZE
- MPI_COMM_RANK
- MPI_COMM_SIZE
- MPI_BCAST
- MPI_REDUCE

- What else is needed (and why)?

```
Alternative 6 Functions for Simplified MPI

```}

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, send must wait for user to provide memory space (via a receive)
- What happens with

```
Sources of Deadlocks

```
Some Solutions to the “unsafe” Problem

- Order operations more carefully:
  
<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Recv(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Send(0)</td>
</tr>
</tbody>
</table>

- Use non-blocking operations:
  
<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isend(1)</td>
<td>Isend(0)</td>
</tr>
<tr>
<td>Irecv(1)</td>
<td>Irecv(0)</td>
</tr>
<tr>
<td>Waitall</td>
<td>Waitall</td>
</tr>
</tbody>
</table>

- How about races?
  
  - Multiple recv processes w/ wildcard MPI_ANY_SOURCE

Optimization by Non-blocking Communication

- Non-blocking operations work, but:
  
<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isend(1)</td>
<td>Isend(0)</td>
</tr>
<tr>
<td>Irecv(1)</td>
<td>Irecv(0)</td>
</tr>
<tr>
<td>Waitall</td>
<td>Waitall</td>
</tr>
</tbody>
</table>

- May want to reverse send/receive order: (Why?)
  
<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irecv(1)</td>
<td>Irecv(0)</td>
</tr>
<tr>
<td>Isend(1)</td>
<td>Isend(0)</td>
</tr>
<tr>
<td>Waitall</td>
<td>Waitall</td>
</tr>
</tbody>
</table>

Communication and Blocking Modes

- Communication modes:
  
  - Std: int send w/o recv
  - Ready: send iff recv ready
  - Sync: see Std but send only completes if recv OK
  - Buf: see Std but reserves place to put data
  - MPI_Buffer_attach/detach

- Nonblocking completed?
  
  - MPI_Wait/Waitall
  - MPI_Waitany/all/some

- Send/Recv w/ same/diff buffer
  
  - MPI_Ssend
  - MPI_Ssendrecv
  - MPI_Ssendrecv_replace

<table>
<thead>
<tr>
<th></th>
<th>Send</th>
<th>Blocking</th>
<th>Nonblocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
<td></td>
</tr>
<tr>
<td>Ready</td>
<td>MPI_Recv</td>
<td>MPI_Irecv</td>
<td></td>
</tr>
<tr>
<td>Synchronous</td>
<td>MPI_Ssend</td>
<td>MPI_Isend</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Receive</th>
<th>Blocking</th>
<th>Nonblocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>MPI_Recv</td>
<td>MPI_Irecv</td>
<td></td>
</tr>
</tbody>
</table>
Communicators

- Alternative to avoid deadlocks:
  - Use different communicators
  - Often used for different libraries
- Group: MPI_Comm_group, MPI_Comm_incl
- Context: for a group: MPI_Comm_create
- How about multicast?

Toward a Portable MPI Environment

- MPICH: high-performance portable implementation of MPI (1+2)
- runs on MPI’s, clusters, and heterogeneous networks of workstations
- In a wide variety of environments, one can do:
  - configure
  - make
  - mpicc -mpistrace myprog.c
  - mpirun -np 10 myprog
  - cc: mpixxer -O 10 myprog
- to build, compile, run, and analyze performance
- Others: LAM MPI, OpenMPI, vendor X MPI

Extending the Message-Passing Interface

- Dynamic Process Management
  - Dynamic process startup
  - Dynamic establishment of connections
- One-sided communication
  - Put/Get
  - Other operations
- Parallel I/O
- Other MPI-2 features
  - Generalized requests
  - Bindings for C++/Fortran-90; interlanguage issues
Profiling Support: PMPI

- Profiling layer of MPI
- Implemented via additional API in MPI library
  - Different name: PMPI_Init()
  - Same functionality as MPI_Init()
- Allows user to:
  - Define own MPI_Init()
  - Need to call PMPI_Init():
- User may choose subset of MPI routines to be profiled
- Useful for building performance analysis tools
  - Vampir: Timeline of MPI traffic (Emus, Inc.)
  - Paraview: Performance analysis (U. Wisconsin)
  - mpI: J. Vetter (LLNL)
  - Scalate: F. Mueller et al. (NCSU)

When to use MPI

- Portability and Performance
- Irregular Data Structures
- Building Tools for Others
  - Libraries
  - Need to manage memory on a per-processor basis

When not (necessarily) to use MPI

- Regular computation matches HPF
  - But see PETSc/HPF comparison
  - Solution (e.g., library) already exists
  - http://www.ncsa.uiuc.edu/mpi/libraries.html
- Require Fault Tolerance
  - Sockets
    - will see other options (research)
- Distributed Computing
  - CORBA, DCOM, etc.
- Embarassingly parallel data division
  - Google map-reduce
Is MPI Simple?

- We said: Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - MPI_Init
  - MPI_Comm_Size
  - MPI_Comm_Rank
  - MPI_Send
  - MPI_Finalize
  - MPI_Comm_Finalize

- Empirical study for large-scale benchmarks shows (IPDPS'02):

<table>
<thead>
<tr>
<th>Routine</th>
<th>MPICH</th>
<th>SNC3D</th>
<th>SPRINT</th>
<th>Simple/Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>MPI_FINALIZE</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>MPI_Comm_Init</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>MPI_Comm_Finalize</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>MPI_RECV</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>MPI_ALLREDUCE</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>MPI_COMM_TEST</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>MPI_COMM_WAIT</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>MPI_COMM_FINALIZE</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Summary

- Parallel computing community has cooperated on development of
  - Standard for message-passing libraries
- Many implementations, on nearly all platforms
- MPI subsets are easy to learn and use
- Lots of MPI material available
- Trends to adaptive computation (adaptive mesh refinement)
  - Addi MPI routines may be needed (even MPI-2 sometimes)

Before MPI-2

1995 user poll showed:
- Diverse collection of users
- All MPI functions in use, including "obscure" ones.
- Extensions requested:
  - Parallel I/O
  - Process management
  - Connecting to running processes
  - Put/get, active messages
  - Interrupt-driven receive
  - Non-blocking collective
  - C++ bindings
  - Threads, odds and ends
MPI-2 Origins

- Began meeting in March 1995, with
  - veterans of MPI-1
  - new vendor participants (especially Cray and SGI, and Japanese manufacturers)
- Goals:
  - Extend computational model beyond message-passing
  - Add new capabilities
  - Respond to user reaction to MPI-1
- MPI-1 announced in June 1995 with MPI-1 repairs, some bindings changes
- MPI-1.2 and MPI-2 released July 1997
- Implemented in most (all?) MPI libraries today

Contents of MPI-2

- Extensions to the message-passing model
  - Parallel I/O
  - One-sided operations
  - Dynamic process management
- Making MPI more robust and convenient
  - C++ and Fortran 90 bindings
  - Extended collective operations
  - Language interoperability
  - MPI-2 interaction with threads
  - External interfaces

MPI-2 Status Assessment

- All MPI vendors now have MPI-1. Free implementations (MPICH, LAM) support heterogeneous workstation networks.
- MPI-2 implementations are in for most (all?) vendors.
- MPI-2 implementations appearing piecemeal, with I/O first.
  - I/O available in most MPI implementations
  - One-sided available in most (may still depend on interconnect, e.g., InfiniBand has it, Ethernet may have it)
  - Parts of dynamic and one-sided in LAM/OpenMPI/MPICH
Dynamic Process Management in MPI-2

- Allows an MPI job to spawn new processes at run time and communicate with them
- Allows two independently started MPI applications to establish communication

Starting New MPI Processes

- MPI_Comm_spawn
  - Starts new processes
  - Collective over communicator
  - Necessary for scalability
  - Returns an intercommunicator
    - Does not change MPI_COMM_WORLD

Connecting Independently Started Programs

- MPI_Open_port, MPI_Comm_connect, MPI_Comm_accept allow two running MPI programs to connect and communicate
  - Not intended for client/server applications
  - Designed to support HPC applications
- MPI_Sock allows the use of a TCP socket to connect two applications
  - Important for multi-scale simulations
    - Connect multiple independent simulations, combine calculations
**One-Sided Operations: Issues**

- Balancing efficiency and portability across a wide class of architectures
  - shared-memory multiprocessors
  - NUMA architectures
  - distributed-memory MPPs, clusters
  - Workstation networks
- Retaining "look and feel" of MPI-1
- Dealing with subtle memory behavior issues: cache coherence, sequential consistency
- Synchronization is separate from data movement

---

**Remote Memory Access Windows and Window Objects**

- Process 0
  - Get
  - window
- Process 1
  - Put
- Process 2
- Process 3

= address spaces

= window object

---

**One-Sided Communication Calls**

- MPI_Put - stores into remote memory
- MPI_Get - reads from remote memory
- MPI_Accumulate - combined local/remote memory
  - like reduction, need to specify "op", e.g., MPI_SUM
- All are non-blocking: data transfer is described, maybe even initiated, but may continue after call returns
- Subsequent synchronization on window object is needed to ensure operations are complete, e.g., MPI_Win_fence