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
  - Hello, 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)

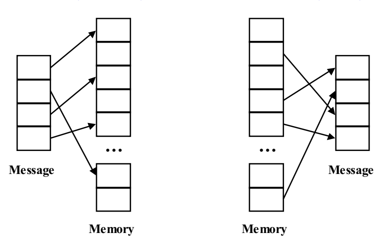
Message passing (and MPI) is for MIMD/SPMD parallelism. HPF is an example of a SIMD interface.

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)**



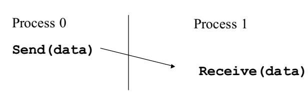
Basic Message Passing Issues

- Issues include:
  - Naming: How to specify the receiver?
  - Buffering: What if the out port 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

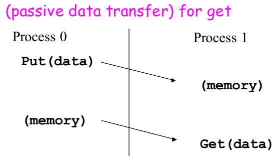
➢ Push model (active data transfer)



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.

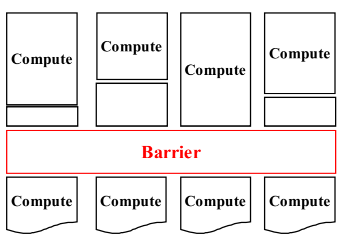
➢ Pull model (passive data transfer) for get



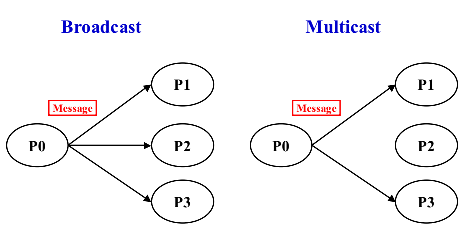
Collective Communication

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

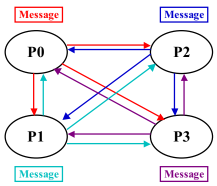
Barrier



Broadcast and Multicast

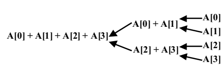


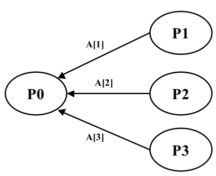
All-to-All

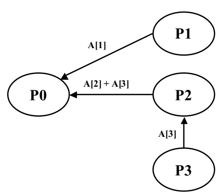


Reduction

sum ← 0
for i ← 1 to p do
  sum ← sum + A[i]







What is MPI?

- A message-passing library specification (an API)
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers
- Portability

MPI Sources

- Standard: http://www.mpi-forum.org
- Books:
  - Using MPI: Portable Parallel Programming with the Message-Passing Interface, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
  - MPI: The Complete Reference, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996.
  - Designing and Building Parallel Programs, by Ian Foster, Addison-Wesley, 1995.
  - Parallel Programming with MPI, by Peter Pacheco, Morgan-Kaufmann, 1997.
  - MPI: The Complete Reference Vol 1 and 2,MIT Press, 1998(Fall).
- Other information on Web 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 (errata) +
  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 )
end
```


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:
  - mpif.h 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 MPICH

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)

```
#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;
}
```

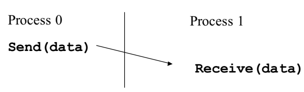
Better Hello (Fortran)

```
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

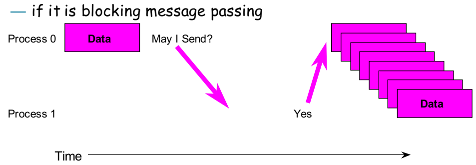
- We need to fill in the details in



- 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**

CSC 591C 28

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 strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored columnwise

CSC 591C 29

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.

CSC 591C 30

MPI Basic (Blocking) Send

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 msg may not have been received by target process (yet)

MPI Basic (Blocking) Receive

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 **`MPI_ANY_SOURCE`**
- **`status`** contains further information
- Receiving fewer than **`count`** occurrences of **`datatype`** is OK
  - but receiving more is an error

Retrieving Further Information

- **`Status`** is a data structure allocated in the user's program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

- In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, .. status, ierr)
tag_recvd  = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```

Simple Fortran Example

```
program main
use MPI

integer rank, size, to, from, tag10
integer count, i, ierr
integer src, dest
integer st_source, st_tag, st_count
integer status(MPI_STATUS_SIZE)
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 *, 'Process ', rank, ' of ',
+        size, ' is alive'
dest = size - 1
src  = 0
```

```
 if (rank .eq. 0) then
 do 10, i=1, 10
   data(i) = i
 continue
 call MPI_SEND( data, 10, MPI_DOUBLE_PRECISION,
+              dest, 2001, MPI_COMM_WORLD, ierr)
else if (rank .eq. dest) then
 tag = MPI_ANY_TAG
 source = MPI_ANY_SOURCE
 call MPI_RECV( data, 10, MPI_DOUBLE_PRECISION,
+              source, tag, MPI_COMM_WORLD,
+              status, ierr)
 call MPI_GET_COUNT( status, MPI_DOUBLE_PRECISION,
                     st_count, ierr )
 st_source = status( MPI_SOURCE )
 st_tag    = status( MPI_TAG )
 print *, 'status info: source = ', st_source,
+        ' tag = ', st_tag, 'count = ', st_count
endif

call MPI_FINALIZE( ierr )
end
```

CSC 591C 34

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

CSC 591C 35

Basic C Datatypes in MPI

MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

CSC 591C 36

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

CSC 591C 37

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...

CSC 591C 38

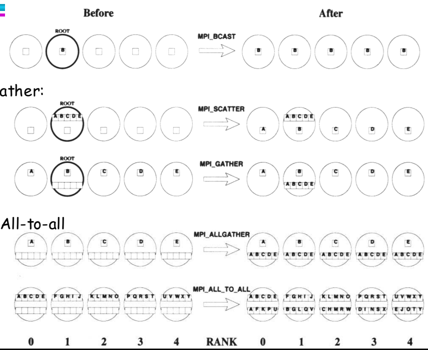
Introduction to Collective Operations in MPI

- 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]SCATTER[V]/[ALL]GATHER[V]**

CSC 591C 39

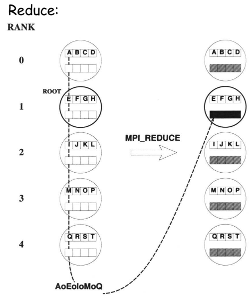
Collectives at Work

- BCAST:
- Scatter/Gather:
- Allgather/All-to-all



Collectives at Work (2)

- Reduce:

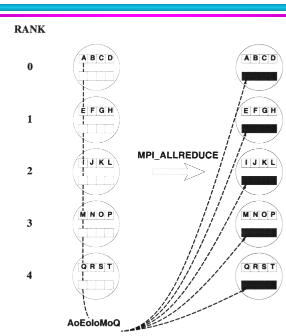


- Predefined Ops (assocociative & commutative) / user ops (assoc.)

MPI Name	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum & location
MPI_MINLOC	Minimum & location

Collectives at Work (3)

- Allreduce:



Example: PI in C

```
#include "mpi.h"

#include <math.h>

int main(int argc, char *argv[])

{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT =
   3.141592653589793238462643;
   double mypi, pi, h, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   while (!done)  {
     if (myid == 0) {
       printf("Enter the number of

               intervals: (0 quits) ");
       scanf("%d",&n);
     }
     MPI_Bcast(&n, 1, MPI_INT, 0,

               MPI_COMM_WORLD);
     if (n == 0) break;
       h = 1.0 / (double) n;
       sum = 0.0;
       for (i=myid+1; i<=n; i+=numprocs) {
         x = h * ((double)i - 0.5);
         sum += 4.0 / (1.0 + x*x);
       }
       mypi = h * sum;
       MPI_Reduce(&mypi, &pi, 1,

                  MPI_DOUBLE, MPI_SUM, 0,

                   MPI_COMM_WORLD);
       if (myid == 0)
         printf("pi is approximately

                  %.16f, Error is %.16f\n",
                  pi, fabs(pi - PI25DT));
      }
      MPI_Finalize();

     return 0;

   }
```


Approximation of Pi

Integration to evaluate π

Computer approximations to π by using numerical integration
Know

$$tan(45^0) = 1;$$

same as

$$tan\frac{\pi}{4} = 1;$$

So that;

$$4 * tan^{-1}1 = \pi$$

From the integral tables we can find

$$tan^{-1}x = \int \frac{1}{1+x^2}dx$$

or

$$tan^{-1}1 = \int_0^1 \frac{1}{1+x^2}dx$$

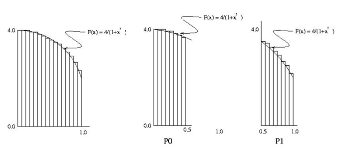
Using the mid-point rule with panels of uniform length $h = 1/n$, for various values of n.
Evaluate the function at the midpoints of each subinterval (x_{i-1}, x_i) $i * h - h/2$ is the midpoint.
Formula for the integral is

$$x = \sum_{i=1}^{n} f(h * (i - 1/2))$$

$$\pi = h * x$$

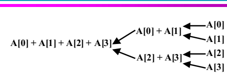
where

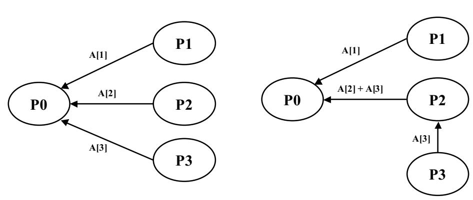
$$f(x) = \frac{4}{1+x^2}$$



Reduction

sum ← 0
for i ← 1 to p do
sum ← sum + A[i]





Example: PI in Fortran

```
      program main
      use MPI
      double precision  PI25DT
      parameter (PI25DT = 3.141592653589793238462643d0)
      double precision  mypi, pi, h, sum, x, f, a
      integer n, myid, numprocs, i, ierr
c                                 function to integrate
      f(a) = 4.d0 / (1.d0 + a*a)
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
 10   if ( myid .eq. 0 ) then
         write(6,98)
 98      format('Enter the number of intervals: (0 quits)')
         read(5,99) n
 99      format(i10)
      endif
      call MPI_BCAST( n, 1, MPI_INTEGER, 0,
     +                MPI_COMM_WORLD, ierr)
c                              check for quit signal
      if ( n .le. 0 ) goto 30
c                         calculate the interval size
      h = 1.0d0/n
      sum  = 0.0d0
      do 20 i = myid+1, n, numprocs
        x   = h * (dble(i) - 0.5d0)
        sum = sum + f(x)
 20   continue
      mypi = h * sum
c                        collect all the partial sums
      call MPI_REDUCE( mypi, pi, 1, MPI_DOUBLE_PRECISION,
     +                MPI_SUM, 0, MPI_COMM_WORLD,ierr)
```

```
c                               node 0 prints
the answer
      if (myid .eq. 0) then
         write(6, 97) pi, abs(pi - PI25DT)
 97      format('  pi is approximately: ',
F18.16,
     +          '  Error is: ', F18.16)
      endif
      goto 10
 30   call MPI_FINALIZE(ierr)
      end
```


Alternative 6 Functions for Simplified MPI

- `MPI_INIT`
- `MPI_FINALIZE`
- `MPI_COMM_SIZE`
- `MPI_COMM_RANK`
- `MPI_BCAST`
- `MPI_REDUCE`
- What else is needed (and why)?

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

Process 0	Process 1
Send(1)	**Send(0)**
Recv(1)	**Recv(0)**

- This is called "unsafe" because it depends on the availability of system buffers

Some Solutions to the "unsafe" Problem

- Order operations more carefully:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`
`Waitall`	`Waitall`

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

Optimization by Non-blocking Communication

- Non-blocking operations work, but:

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`
`Waitall`	`Waitall`

- May want to reverse send/receive order: (Why?)

Process 0	Process 1
`Irecv(1)`	`Irecv(0)`
`Isend(1)`	`Isend(0)`
`Waitall`	`Waitall`

Communication and Blocking Modes

- Communication modes:
  - Std: init 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/Test
  - MPI_Waitall/any/some
- Send+Recv w/ same/diff buffer
  - MPI_Sendrecv
  - MPI_Sendrecv_replace

Send	Blocking	Nonblocking
Standard	MPI_Send	MPI_Isend
Ready	MPI_Rsend	MPI_Irsend
Synchronous	MPI_Ssend	MPI_Issend
Buffered	MPI_Bsend	MPI_Ibsend

Receive	Blocking	Nonblocking
Standard	MPI_Recv	MPI_Irecv

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 MPP's, clusters, and heterogeneous networks of workstations
- In a wide variety of environments, one can do:

```
configure
make
mpicc -mpitrace myprog.c
mpirun -np 10 myprog
or: mpiexec -n 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():

```
MPI_Init(…) {
  collect pre stats;
  PMPI_Init(…);
  collect post stats;
}
```

- User may choose subset of MPI routines to be profiled
- Useful for building performance analysis tools
  - Vampir: Timeline of MPI traffic (Etnus, Inc.)
  - Paradyn: Performance analysis (U. Wisconsin)
  - mpiP: J. Vetter (LLNL)
  - ScalaTrace: 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 (ICASE 97-72)
- Solution (e.g., library) already exists
  - http://www.mcs.anl.gov/mpi/libraries.html
- Require Fault Tolerance
  - Sockets
  - will see other options (research)
- Distributed Computing
  - CORBA, DCOM, etc.
- Embarrassingly 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_SEND`
  - `MPI_FINALIZE` — `MPI_COMM_RANK` — `MPI_RECV`
- Empirical study for large-scale benchmarks shows (IPDPS'02):

Routines	sPPM	SMG2000	SPHOT	Sweep3D	Samrai
MPI_Allreduce	X	X		X	X
MPI_Barrier			X		
MPI_Bcast				X	
MPI_Irecv	X	X	X		X
MPI_Isend	X	X			X
MPI_Recv				X	
MPI_Reduce			X		
MPI_Send			X	X	
MPI_Test					X
MPI_Wait	X	X			X
MPI_Waitall		X	X		

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)
  - — Add'l 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-1.1 released 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 interaction with threads
  - External interfaces

MPI-2 Status Assessment

- All MPP 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 n new processes
  - Collective over communicator
    - Necessary for scalability
  - Returns an intercommunicator
    - Does not change MPI_COMM_WORLD

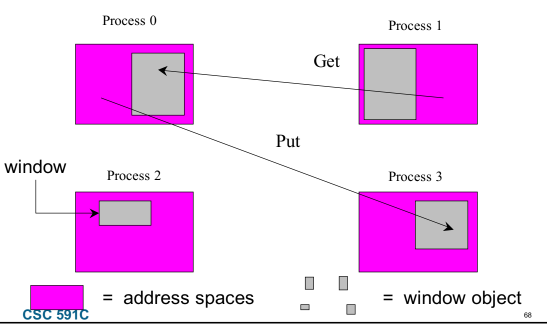
Connecting Independently Started Programs

- **`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_Join`** 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 MPP's, 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



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`