## Message Passing Interface (MPI)

- A standard message-passing library
  - p4, NX, PVM, Express, PARMACS are precursors
- An MPI program defines a set of processes
  (usually one process per node)
- ... that communicate by calling MPI functions
  (point-to-point and collective)
- ... and can be constructed in a modular fashion
  (communicators are the key)

## A First MPI Program

```
#include "mpi.h"
#include <stdio.h>
int main( argc, argv )
int argc;char **argv;
{
    MPI_Init( &argc, &argv );
    printf( "Hello world\n" );
    MPI_Finalize();
    return 0;
}
```

Basic definitions and types

Starts MPI

Exits MPI

*Non-MPI routines are local:* printf *runs on each process*

## What is MPI?

- MPI defines a language-independent interface
  - Not an implementation
- Bindings are defined for different languages
  - So far, C and Fortran 77
  - C++ and Fortran 90 under development
- Multiple implementations
  - MPICH is a widely-used portable implementation
  - See http://www.mcs.anl.gov/mpi/

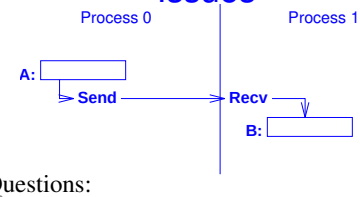
## Compiling and Linking (in MPICH)

- Special compiler commands for simple programs
  mpicc -o first first.c  mpif77 -o firstf firstf.f
- Options enable MPI profiling features
  - -mpilog: Generate log files of MPI calls
  - -mpitrace: Trace execution of MPI calls
  - -mpianim: Real-time animation (on some systems)
- Standard makefiles for larger programs
  - See /usr/local/mpi/examples/Makefile.in

## Running MPI Programs (in MPICH)

- To run a program on two processors
  mpirun -np 2 hello
- To list available command-line argument
  mpirun -h
- To list commands mpirun would execute
  mpirun -t -np 2 hello
- All these mechanisms are MPICH specific
  - Standard does not specify startup mechanism

## Sending/Receiving Messages: Issues




- Questions:
  - What is sent?
  - To whom is the data sent?
  - How does the receiver identify it?

## First-Generation Message-Passing Systems

A typical blocking send looked like

```
send(dest, type, address, length)
```

where

- `dest` = integer identifier for receiving process
- `type` = nonnegative integer that the destination can use to selectively screen messages
- (`address`, `length`) describes a contiguous area in memory containing the message to be sent

## First Generation Message-Passing, contd.

- A typical global operation looks like:

```
broadcast(type, address, length)
```

- These specifications are
  - a good match to hardware
  - easy to understand
- However, they are too inflexible

## The Buffer

- Sending and receiving only a contiguous array of bytes
  - Hides the real data structure from hardware which might be able to handle it directly
  - Requires pre-packing dispersed data, e.g.:
    - Rows of a matrix stored columnwise
    - General collections of structures
  - Prevents communications between machines with different data representations

## Generalizing the Buffer Description

- Specified in MPI by *starting address*, *datatype*, and *count*, where datatype can be
  - Elementary (all C and Fortran datatypes)
  - Contiguous array of datatypes
  - Strided blocks of datatypes
  - Indexed array of blocks of datatypes
  - General structures
- Datatypes are constructed recursively

## Advantages of Datatypes

- Specifications of elementary datatypes allows heterogeneous communication.
- Elimination of length in favor of count is clearer
- Specifying application-oriented layouts allows maximal use of special hardware

## Generalizing the Type

- A single type field is too constraining
  - Often overloaded to provide needed flexibility
- Problems
  - Under user control
  - Wild cards allowed (MPI_ANY_TAG)
  - Library use conflicts with user and with other libraries

## Sample Program using Library

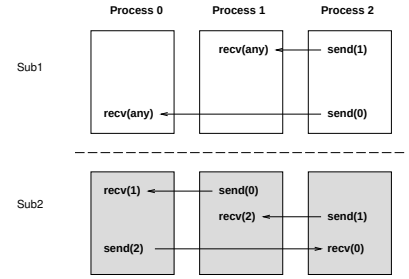
- Calls `Sub1` and `Sub2` are from different libraries

```
Sub1();
Sub2();
```

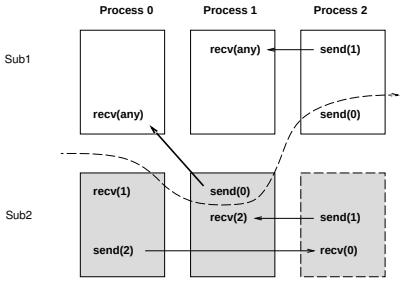
- `Sub1a` and `Sub1b` are from the same library

```
Sub1a();
Sub2();
Sub1b();
```

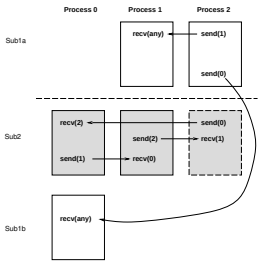
## Correct Execution



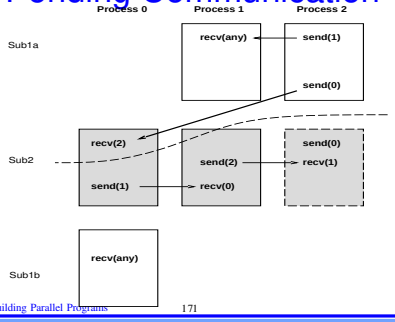
## Incorrect Execution



## Correct Execution with Pending Communcication



## Incorrect Execution with Pending Communication



## Solution to the Type Problem

- A separate communication *context* for each family of messages
  - Used for queueing and matching
  - Often simulated in the past by overloading tags
- No wild cards allowed, for security
- Allocated by the system, for security
- Types (called *tags*) retained for normal use
  - wild cards OK for tags

## Delimiting Scope of Communication: Summary

- Disjoint process groups working on subproblems
  - Merging of process name space interferes with modularity
  - "Local" process identifiers desirable
- Parallel invocation of parallel libraries
  - Need to separate application & library messages
  - Knowledge of library message types interferes with modularity

## Generalizing the Process Identifier

- Collective operations typically operate on all processes (subgroups in some systems)
  - This is too restrictive (e.g., need minimum over a column or a sum across a row, of processes)
- MPI supports process groups
  - Initial "all" group
  - Group management routines (build, delete)
  - Collective operations on groups

## Process Groups

- All communication (not just collective) takes place in groups
- Source/destination in send/receive operations refer to **rank** in group
- **Communicator** = group + context
- MPI_ANY_SOURCE permitted in receive

## MPI Basic Send/Receive

- Thus the basic (blocking) send has become:

```
MPI_Send(start, count, datatype, dest,
         tag, comm)
```

- and the receive:

```
MPI_Recv(start, count, datatype, source,
         tag, comm, status)
```

  - The source, tag, and count of the message actually received can be retrieved from status

## Two Simple Collective Operations

- `MPI_Bcast(start, count, datatype, root, comm)`
  - Broadcast value to members of comm
- `MPI_Reduce(start, result, count, datatype, operation, root, comm)`
  - Perform reduction using "operation"

## Message Information

```
MPI_Status status;
MPI_Recv( ..., &status );
```

`status.MPI_TAG`
`status.MPI_SOURCE`

If `MPI_ANY_TAG` &/or `MPI_ANY_SOURCE` specified in receive

```
MPI_Get_count(&status,
              datatype,
              &count);
```

How much data received (type must be specified)

## Message Information Example

```
while (1) {
   MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
             &status);
   switch (status.MPI_TAG) {
   case IM_DONE:
      ...
      break;
   case RESULT:
      MPI_Recv(buf, maxsize, datatype, status.MPI_SOURCE,
               status.MPI_TAG, MPI_COMM_WORLD, &status );
      ....
      MPI_Send( ... more work ... );
   }
}
```

## Simple Fortran Example

```
 program main
 include 'mpif.h'
 integer rank, size, tag, count, i, ierr
 integer src, dest, st_source, st_tag, st_count
 integer status(MPI_STATUS_SIZE)
 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
 if (rank .eq. 0) then
    data  = 1096
    call MPI_SEND( data, 1, MPI_INTEGER, dest, 99,
+                   MPI_COMM_WORLD, ierr )
 else ...
```

## Simple Fortran Example (contd)

```
   else if (rank .eq. dest) then
     tag    = MPI_ANY_TAG
     from   = MPI_ANY_SOURCE
     call MPI_RECV(data, count, MPI_INTEGER, from,
+                  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)C
     print *, 'Status info: source = ', st_source,
+            ' tag = ', st_tag, ' count = ', st_count
     print *, rank, ' received', data
   endif
   call MPI_FINALIZE( ierr )
   end
```

## Six-Function MPI

- MPI is very simple: 6 functions allow you to write many programs
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Send
  - MPI_Recv

## Collective Communication

- Provides standard interfaces to common global operations
- A collective operation involves a group of processes
  - Specialized functions for defining process groups
  - Same function is called by all processes in group
- Use different communicators instead of message tags

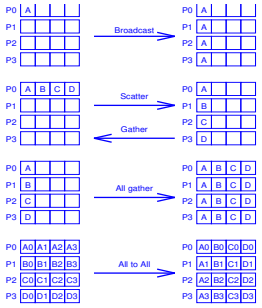
## Collective Communication Functions

- All collective operations are blocking
  - No process returns until all processes have called function
- Three classes of collective operations
  - Synchronization
  - Data movement
  - Collective computation

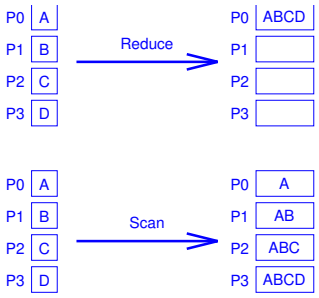
## Synchronization

- MPI_Barrier(comm)
  - Blocks until all processes in communicator comm call it

## Data Movement



## Collective Computation Patterns



## MPI Datatypes

- MPI datatypes have two main purposes
  - Heterogenity: communication between processors of different types
  - Noncontiguous data: structures, vectors with non-unit stride, etc.
- Basic datatypes, corresponding to the underlying language, are predefined
- The user can construct new derived datatypes at run time

## MPI Datatypes

- Elementary: MPI_INT, MPI_REAL, etc.
- Vector: Separated by constant “stride”
- Contiguous: Vector with stride of one
- Hvector: Vector, with stride in bytes
- Indexed: Array of indices (for scatter/gather)
- Hindexed:Indexed, with indices in bytes
- Struct: General mixed types (for C structs etc.)

## Tools for Writing Libraries

- MPI is specifically designed to make it easier to write message-passing libraries
- Communicators solve tag/source wild-card problem
- Attributes provide a way to attach information to a communicator

## Other MPI Features

- Specialized communication modes
- Intercommunicators for modular construction of multicomponent programs
- Error handlers
- Attributes on communicators

## Task Decomposition Example: Server

```
while (1) {
    MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
              &status );
    switch (status.MPI_TAG) {
        case IM_DONE: ... break;
        case RESULT:
             MPI_Recv(buf, maxsize, datatype,
                      status.MPI_SOURCE, status.MPI_TAG,
                      MPI_COMM_WORLD, &status );
             ....
             MPI_Send( ... more work ... );
    }
}
```

## Alternative Server

```
#define large 128
MPI_Request requests[large];
MPI_Status  statuses[large];
int         indices[large], buf[large];
for (i=1; i<size; i++)
  MPI_Irecv(buf+i, 1, MPI_INT, i, MPI_ANY_TAG, MPI_COMM_WORLD,
            &requests[i-1] );
while(not done) {
    MPI_Waitsome(size-1, requests, &ndone, indices, statuses );
    for (i=0; i<ndone; i++) {
        j = indices[i];
        ... switch on statuses[i].MPI_TAG ...
        MPI_Irecv(buf+j, 1, MPI_INT, j, MPI_ANY_TAG,
                  MPI_COMM_WORLD, &requests[j] );
    }
}
```

## Regular Grid Example

- MPI contains routines to simplify writing programs for regular grid operations
  - MPI_CART_CREATE etc. for the decomposition
  - MPI_SENDRECV for exchanging data
  - MPI_PROCNULL for boundaries

## Regular Grid Example: Getting the Decomposition

```
int size, dims[2], periods[2];
MPI_Comm_size( MPI_COMM_WORLD, &size );
MPI_Dims_create( size, 2, dims );
periods[0] = periods[1] = 0;
MPI_Cart_create( MPI_COMM_WORLD, 2, dims,
                 periods,  1, &comm2d );
```

## Regular Grid Example: Conclusion

- Getting neighbors

```
MPI_CART_SHIFT( comm2d, 0, 1, &nbrleft, &nbrright );
MPI_CART_SHIFT( comm2d, 1, 1, &nbrbottom, &nbrtop );
```

- Sending data
  - Can use any MPI send and receive routine
  - Neighbors at boundary have rank MPI_PROCNULL
  - Send/receive to MPI_PROCNULL has no effect
  - No explicit support for managing "ghost points"