



Cornell University
Center for Advanced Computing

Introduction to MPI

Susan Mehringer
Cornell Center for Advanced Computing

May 2012

Based on materials developed by CAC and TACC



Overview

Outline

- Overview
- Basics
 - Hello World in MPI
 - Compiling and running MPI programs
- MPI Messages
- MPI Communicators
- Point-to-point communication
- Collective communication
- Releases
- MPI references and documentation



Overview

Introduction

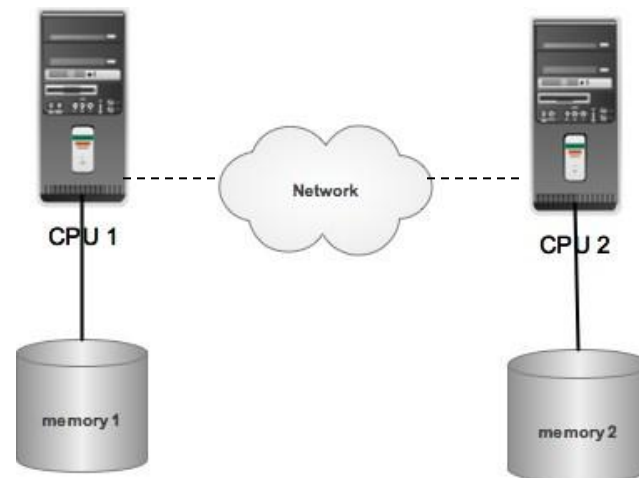
- What is MPI? Message Passing Interface
- What is message passing?
 - Sending and receiving messages between tasks or processes
 - Can include performing operations on data in transit and synchronizing tasks
- Why send messages?
 - Clusters have distributed memory, i.e. each process has its own address space and no way to get at another's
- How do you send messages?
 - Programmer makes use of an Application Programming *Interface* (API) that specifies the functionality of high-level communication routines
 - Functions give access to a low-level *implementation* that takes care of sockets, buffering, data copying, message routing, etc.



Overview

API for distributed memory parallelism

- Assumption: processes do not see each other's memory
- Communication speed is determined by some kind of network
 - Typical network = switch + cables + adapters + software stack...
- Key: the implementation of a message passing API (like MPI) can be optimized for any given network
 - Program gets the benefit
 - No code changes required
 - Works in shared memory, too





Overview

Why use MPI?

- MPI is a de facto standard
 - Public domain versions are easy to install
 - Vendor-optimized version are available on most hardware
- MPI is “tried and true”
 - MPI-1 was released in 1994, MPI-2 in 1996
- MPI applications can be fairly portable
- MPI is a good way to learn parallel programming
- MPI is expressive: it can be used for many different models of computation, therefore can be used with many different applications
- MPI code is efficient (though some think of it as the “assembly language of parallel processing”)



Basics

The basic outline of an MPI program follows these general steps:

- *Include the MPI header file --*
#include <mpi.h> for basic definitions and types, implementation-specific.
- *Initialize communications --*
MPI_INIT initializes the MPI environment
MPI_COMM_SIZE returns the number of processes
MPI_COMM_RANK returns this process's number (rank)
- *Communicate to share data between processes --*
MPI_SEND sends a message
MPI_RECV receives a message
- *Exit from the message-passing system --*
MPI_FINALIZE



Basics

Minimal Code Example

```
• #include <. . . .>
• #include "mpi.h"
• main(int argc, char **argv)
• {
•     char message[20];
•     int i, rank, size, type = 99;
•     MPI_Status status;
•     MPI_Init(&argc, &argv);
•     MPI_Comm_size(MPI_COMM_WORLD, &size);
•     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
•     if (rank == 0) {
•         strcpy(message, "Hello, world");
•         for (i = 1; i < size; i++)
•             MPI_Send(message, 13, MPI_CHAR, i, type, MPI_COMM_WORLD);
•     }
•     else
•         MPI_Recv(message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
•     printf( "Message from process = %d : %.13s\n", rank,message);
•     MPI_Finalize();
• }
```



Basics

Initialize and Close Environment

```
• #include <...>
• #include "mpi.h"
• main(int argc, char**argv)
• {
•     char message[20];
•     int i, rank, size, type = 99;
•     MPI_Status status;
•     MPI_Init(&argc, &argv);
•     MPI_Comm_size(MPI_COMM_WORLD, &size);
•     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
•     if (rank == 0) {
•         strcpy(message, "Hello, world");
•         for (i = 1; i < size; i++)
•             MPI_Send(message, 13, MPI_CHAR, i, type, MPI_COMM_WORLD);
•     }
•     else
•         MPI_Recv(message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
•     printf("Message from process = %d : %.13s\n", rank, message);
•     MPI_Finalize();
• }
```

Initialize MPI environment

An implementation may also use this call as a mechanism for making the usual argc and argv command-line arguments from “main” available to all tasks (C language only).

Close MPI environment



Basics

Query Environment

```
• #include <. . . .>
• #include "mpi.h"
• main(int argc, char **argv)
• {
•     char message[20];
•     int i, rank, size, type = 99;
•     MPI_Status status;
•     MPI_Init(&argc, &argv);
•     MPI_Comm_size(MPI_COMM_WORLD, &size);
•     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
•     if (rank == 0) {
•         strcpy(message, "Hello, world");
•         for (i = 1; i < size; i++)
•             MPI_Send(message, 13, MPI_CHAR, i, type, MPI_COMM_WORLD);
•     }
•     else
•         MPI_Recv(message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
•     printf("Message from process = %d\n", rank);
•     MPI_Finalize();
• }
```

Returns number of Processes

This, like nearly all other MPI functions, must be called after MPI_Init and before MPI_Finalize. Input is the name of a communicator (MPI_COMM_WORLD is the default communicator) and output is the size of that communicator.

Returns this process' number, or rank

Input is again the name of a communicator and the output is the rank of this process in that communicator.



Basics

Pass Messages

```
• #include <...>
• #include "mpi.h"
• main(int argc, char **argv)
• {
•     char message[20];
•     int i, rank, size, type = 99;
•     MPI_Status status;
•     MPI_Init(&argc, &argv);
•     MPI_Comm_size(MPI_COMM_WORLD, &size);
•     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
•     if (rank == 0) {
•         strcpy(message, "Hello, world");
•         for (i = 1; i < size; i++)
•             MPI_Send(message, 13, MPI_CHAR, i, type, MPI_COMM_WORLD);
•     }
•     else
•         MPI_Recv(message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
•     printf("Message from process = %d : %.13s\n", rank, message);
•     MPI_Finalize();
• }
```

Send a message

Blocking send of data in the buffer.

Receive a message

Blocking receive of data into the buffer.



Basics

Compiling MPI programs

- Generally use a special compiler or compiler wrapper script
 - Not defined by the standard
 - Consult your implementation
 - Correctly handles include path, library path, and libraries
- MPICH-style (the most common)
 - `mpicc -o foo foo.c`
 - `mpicxx -o foo foo.cc`
 - `mpif90 -o foo foo.f` (also `mpif77`)
- Some MPI specific compiler options
 - `-mpilog` -- Generate log files of MPI calls
 - `-mpitrace` -- Trace execution of MPI calls
 - `-mpianim` -- Real-time animation of MPI (not available on all systems)
- Note: compiler/linker names are specific to MPICH. On IBM Power systems, they are `mpicc_r` and `mpxlf_r` respectively



Basics

Running MPI programs

- To run a simple MPI program using MPICH

```
mpirun -help
```
- Some MPI specific running options
 - t -- shows the commands that *mpirun* would execute
 - help -- shows all options for *mpirun*
- To run over Ranger's InfiniBand (as part of an SGE script)

```
ibrun ./foo
```

 - The scheduler handles the rest
- Note: *mpirun* and *mpiexec* are not part of MPI, but a similar command can be found in nearly all implementations
 - There are exceptions: on the IBM SP, for example, it is *poe*



Basics

Submitting MPI Programs

```
#!/bin/bash          # Use Bash Shell
#$ -V               # Inherit the submission environment
#$ -cwd            # Start job in submission directory
#$ -N PI           # Job Name
#$ -j y            # combine stderr & stdout into stdout
#$ -o $JOB_NAME.o$JOB_ID # Name of the output file (eg. myMPI.oJobID)
#$ -pe 12way 12     # Lonestar: Requests 12 cores/node, 12 cores total
##$ -pe 16way 16    # Ranger: Requests 16 cores/node, 16 cores total
#$ -q development   # Queue name
#$ -l h_rt=01:00:00 # Run time (hh:mm:ss) - 1 hour

echo 2000 > input
ibrun ./a.out < input # Run the MPI executable named "a.out"
```



Basics

Submitting MPI Programs

```
...  
#$ -pe 16way 16          # Ranger: Requests 16 tasks/node, 16 cores total  
...  
ibrun ./a.out < input    # Run the MPI executable named "a.out"  
  
#$ -pe [tasks/node] [nodes x 16] tasks/node can be 1, 2, 4, 8, 12, 15  
#$ -pe 16way 64         64 tasks, 64 cores, or 4 nodes  
#$ -pe 8way 64         32 tasks, 64 cores, or 4 nodes
```

```
ibrun -n 32 -o 0 ./a.out
```

Why use less than 16 tasks on a 16 core node? Memory or threads.

How does serial differ? **#\$ -pe 1way 16** (**#\$ -q serial & no ibrun**)



Messages

3 Parameters Describe the Data

```
MPI_Send( message, 13, MPI_CHAR, i, type, MPI_COMM_WORLD );
```

```
MPI_Recv( message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
```

Type of data, should be same
for send and receive
MPI_Datatype type

Number of elements (items, not bytes)
Recv number should be greater than or
equal to amount sent
int count

Address where the data start
void data*



Messages

3 Parameters Specify Routing

```
MPI_Send( message, 13, MPI_CHAR, i, type, MPI_COMM_WORLD );
```

```
MPI_Recv( message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
```

Identify process you're communicating with by rank number
int dest/src

Arbitrary tag number, must match up (receiver can specify MPI_ANY_TAG to indicate that any tag is acceptable)
int tag

Communicator specified for send and receive must match, no wildcards
MPI_Comm comm

Returns information on received message
MPI_Status status*



Messages

Fortran Notes

```
mpi_send (data, count, type, dest, tag, comm, ierr)
mpi_recv (data, count, type, src, tag, comm, status, ierr)
```

- A few Fortran particulars
 - All Fortran arguments are passed by reference
 - *INTEGER ierr*: variable to store the error code (in C/C++ this is the return value of the function call)
- Wildcards are allowed
 - *src* can be the wildcard `MPI_ANY_SOURCE`
 - *tag* can be the wildcard `MPI_ANY_TAG`
 - *status* returns information on the source and tag, useful in conjunction with the above wildcards (receiving only)



MPI_COMM

MPI Communicators

- Communicators
 - Collections of processes that can communicate with each other
 - Most MPI routines require a communicator as an argument
 - Predefined communicator MPI_COMM_WORLD encompasses all tasks
 - New communicators can be defined; any number can co-exist
- Each communicator must be able to answer two questions
 - *How many processes exist in this communicator?*
 - MPI_Comm_size returns the answer, say, N_p
 - *Of these processes, which process (numerical rank) am I?*
 - MPI_Comm_rank returns the rank of the current process within the communicator, an integer between 0 and N_p-1 inclusive
 - Typically these functions are called just after MPI_Init



MPI_COMM

C Example

```
#include <mpi.h>
main(int argc, char **argv) {
    int np, mype, ierr;

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &np);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &mype);
        :
    MPI_Finalize();
}
```



MPI_COMM

C++ Example

```
#include "mpif.h"
[other includes]
int main(int argc, char *argv[]){
    int np, mype, ierr;
    [other declarations]
        :
        MPI::Init(argc, argv);
    np    = MPI::COMM_WORLD.Get_size();
    mype  = MPI::COMM_WORLD.Get_rank();
        :
    [actual work goes here]
        :
        MPI::Finalize();
}
```



```
program param
  include 'mpif.h'
  integer ierr, np, mype

  call mpi_init(ierr)
  call mpi_comm_size(MPI_COMM_WORLD, np , ierr)
  call mpi_comm_rank(MPI_COMM_WORLD, mype, ierr)
  :
  call mpi_finalize(ierr)
end program
```



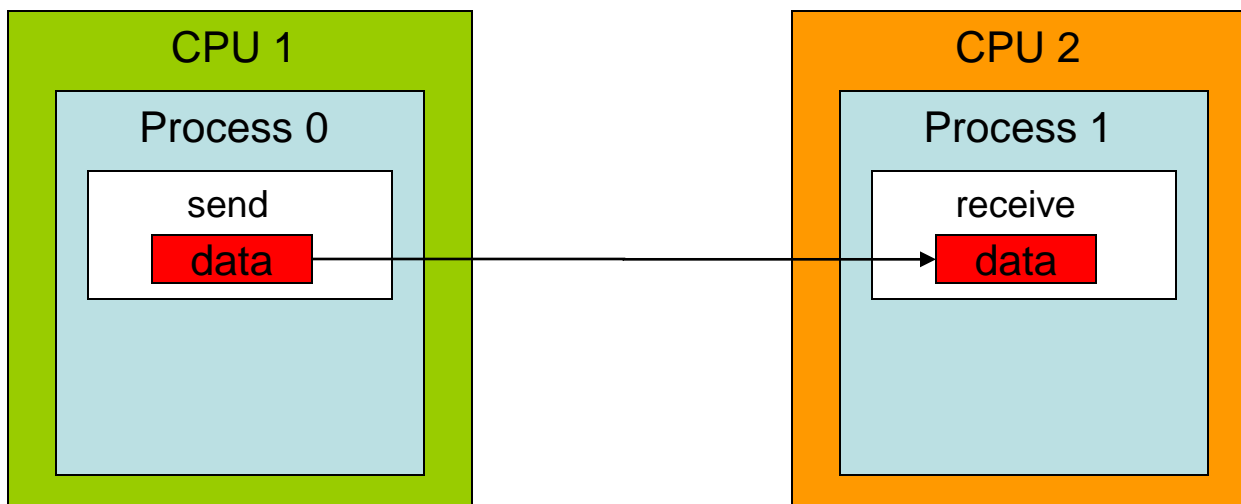
Point to Point | Topics

- MPI_SEND and MPI_RECV
- Synchronous vs. buffered (asynchronous) communication
- Blocking send and receive
- Non-blocking send and receive
- Combined send/receive
- Deadlock, and how to avoid it



Point to Point | Send and Recv: Simple

- Sending data **from** one point (process/task) **to** another point (process/task)
- One task sends while another receives





Point to Point | Send and Recv: So many choices

The communication mode indicates how the message should be sent.

Communication Mode	Blocking Routines	Non-Blocking Routines
Synchronous	MPI_Ssend	MPI_Issend
Ready	MPI_Rsend	MPI_Irsend
Buffered	MPI_Bsend	MPI_Ibsend
Standard	MPI_Send	MPI_Isend
	MPI_Recv	MPI_Irecv
	MPI_Sendrecv	
	MPI_Sendrecv_replace	

Note: the receive routine does not specify the communication mode -- it is simply blocking or non-blocking.



Point to Point | Blocking vs Non-Blocking

A **blocking** send or receive call suspends execution of the process until the message buffer being sent/received is safe to use.

A **non-blocking** call initiates the communication process; the status of data transfer and the success of the communication must be verified independently by the programmer.



Point to Point Communication Modes

Mode	Pros	Cons
Synchronous – sending and receiving tasks must ‘handshake’.	<ul style="list-style-type: none">- Safest, therefore most portable- No need for extra buffer space- SEND/RECV order not critical	Synchronization overhead
Ready- assumes that a ‘ready to receive’ message has already been received.	<ul style="list-style-type: none">- Lowest total overhead- No need for extra buffer space- Handshake not required	RECV <i>must</i> precede SEND
Buffered – move data to a buffer so process does not wait.	<ul style="list-style-type: none">- Decouples SEND from RECV- no sync overhead on SEND- Programmer controls buffer size	Buffer copy overhead
Standard – defined by the implementer; meant to take advantage of the local system.	<ul style="list-style-type: none">- Good for many cases- Compromise position	Your program may not be suitable

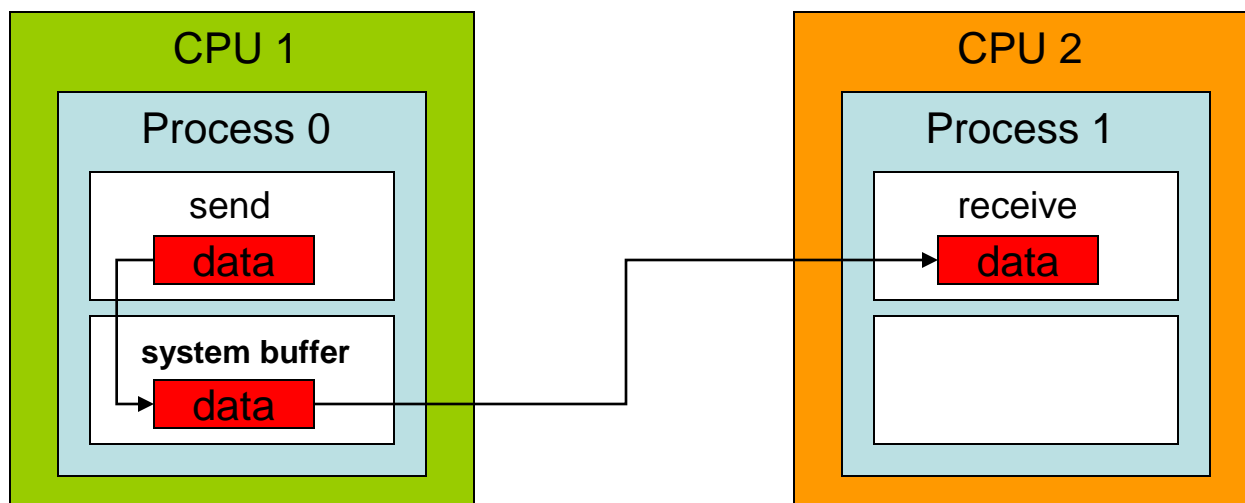


Point to Point Overhead

- **System overhead**
cost of transferring data from the sender's message buffer onto the network, then from the network into the receiver's message buffer.
- **Synchronization overhead**
time spent waiting for an event to occur on another task, e.g. waiting for a receive to be executed and for the handshake to arrive before the message can be transferred.
- **MPI_Send():** A blocking call which returns only when data has been sent from its buffer
- **MPI_Recv():** A blocking receive which returns only when data has been received onto its buffer
- Generally speaking, MPI communications operate in the “rendezvous protocol”, which involves a [handshake procedure](#) in order to establish communication.



Point to Point Buffered send, MPI_Bsend



- Message contents are sent to a system-controlled block of memory
- Process 0 continues executing other tasks; when process 1 is ready to receive, the system simply copies the message from the system buffer into the appropriate memory location controlled by process
- Must be preceded with a call to `MPI_Buffer_attach`



Point to Point | MPI_Sendrecv

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest,  
sendtag, recvbuf, recvcount, recvtype, source, recvtag,  
comm, status)
```

- Useful for communication patterns where each of a pair of nodes both sends and receives a message (two-way communication).
- Executes a blocking send and a blocking receive operation
- Both operations use the same communicator, but have distinct tag arguments



Point to Point | One-way blocking/non-blocking

- Blocking send, non-blocking recv

```
IF (rank==0) THEN
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, MPI_COMM_WORLD, ie)
ELSEIF (rank==1) THEN
  CALL MPI_IRecv(recvbuf, count, MPI_REAL, 0, tag, MPI_COMM_WORLD, req, ie)
  CALL MPI_WAIT(req, status, ie)
ENDIF
```

- Non-blocking send, non-blocking recv

```
IF (rank==0) THEN
  CALL MPI_ISEND(sendbuf, count, MPI_REAL, 1, tag, MPI_COMM_WORLD, req, ie)
ELSEIF (rank==1) THEN
  CALL MPI_IRecv(recvbuf, count, MPI_REAL, 0, tag, MPI_COMM_WORLD, req, ie)
ENDIF
CALL MPI_WAIT(req, status, ie)
```



Point to Point | Two-way communication: deadlock!

- **Deadlock 1**

```
IF (rank==0) THEN
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, MPI_COMM_WORLD, status, ie)
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, MPI_COMM_WORLD, ie)
ELSEIF (rank==1) THEN
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, MPI_COMM_WORLD, status, ie)
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, MPI_COMM_WORLD, ie)
ENDIF
```

- **Deadlock 2**

```
IF (rank==0) THEN
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, MPI_COMM_WORLD, ie)
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, MPI_COMM_WORLD, status, ie)
ELSEIF (rank==1) THEN
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, MPI_COMM_WORLD, ie)
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, MPI_COMM_WORLD, status, ie)
ENDIF
```



Point to Point | Two-way communication: solutions

- Solution 1

```
IF (rank==0) THEN
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, MPI_COMM_WORLD, ie)
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, MPI_COMM_WORLD, status, ie)
ELSEIF (rank==1) THEN
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, MPI_COMM_WORLD, status, ie)
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, MPI_COMM_WORLD, ie)
ENDIF
```

- Solution 2

```
IF (rank==0) THEN
  CALL MPI_SENDRECV(sendbuf, count, MPI_REAL, 1, tag, &
                   recvbuf, count, MPI_REAL, 1, tag, MPI_COMM_WORLD, status, ie)
ELSEIF (rank==1) THEN
  CALL MPI_SENDRECV(sendbuf, count, MPI_REAL, 0, tag, &
                   recvbuf, count, MPI_REAL, 0, tag, MPI_COMM_WORLD, status, ie)
ENDIF
```




Point to Point Solutions (continued)

- Solution 3

```
IF (rank==0) THEN
  CALL MPI_IRecv(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req,ie)
  CALL MPI_Send(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
ELSEIF (rank==1) THEN
  CALL MPI_IRecv(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req,ie)
  CALL MPI_Send(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
ENDIF
CALL MPI_WAIT(req,status)
```

- Solution 4

```
IF (rank==0) THEN
  CALL MPI_BSEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
  CALL MPI_RECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
ELSEIF (rank==1) THEN
  CALL MPI_BSEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
  CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
ENDIF
```



Point to Point Two-way communications: summary

	CPU 0	CPU 1
Deadlock1	Recv/Send	Recv/Send
Deadlock2	Send/Recv	Send/Recv
Solution1	Send/Recv	Recv/Send
Solution2	SendRecv	SendRecv
Solution3	IRecv/Send, Wait	IRecv/Send, Wait
Solution4	BSend/Recv	BSend/Recv



Point to Point C Example

```
#include "mpi.h"
main(int argc, char **argv){
int ipe, ierr; double a[2];
MPI_Status status;
MPI_Comm icomm = MPI_COMM_WORLD;
ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_rank(icom, &ipe);
ierr = MPI_Comm_size(icom, &myworld);
if(ipe == 0){
    a[0] = mype; a[1] = mype+1;
    ierr = MPI_Send(a,2,MPI_DOUBLE, 1,9, icom);
}
else if (ipe == 1){
    ierr = MPI_Recv(a,2,MPI_DOUBLE, 0,9,icom,&status);
    printf("PE %d, A array= %f %f\n",mype,a[0],a[1]);
}
MPI_Finalize();
}
```



Point to Point Fortran Example

```
program sr
  include "mpif.h"
  real*8, dimension(2) :: A
  integer, dimension(MPI_STATUS_SIZE) :: istat
  icomm = MPI_COMM_WORLD
  call mpi_init(ierr)
  call mpi_comm_rank(icom,mype,ierr)
  call mpi_comm_size(icom,np ,ierr);

  if(mype.eq.0) then
    a(1) = real(ipe); a(2) = real(ipe+1)
    call mpi_send(A,2,MPI_REAL8, 1,9,icom, ierr)
  else if (mype.eq.1) then
    call mpi_recv(A,2,MPI_REAL8, 0,9,icom, istat,ierr)
    print*,"PE ",mype,"received A array =",A
  endif

  call mpi_finalize(ierr)
end program
```



Collective

Topics

- Overview
- Barriers
- Data Movement Operations
- Reduction Operations



Collective

Overview

- What if one processor wants to send to everyone else?

```
if (mytid == 0 ) {  
    for (tid=1; tid<ntids; tid++) {  
        MPI_Send( (void*)a, /* target= */ tid, ... );  
    }  
} else {  
    MPI_Recv( (void*)a, 0, ... );  
}
```

- Implements a very naive, serial broadcast
- Too primitive
 - leaves no room for the OS / switch to optimize
 - leaves no room for more efficient algorithms
- Too slow: most receive calls will have a long wait for completion



Collective

Overview

- Involve ALL processes within a communicator
- There are three basic types of collective communications:
 - Synchronization (MPI_Barrier)
 - Data movement (MPI_Bcast/Scatter/Gather/Allgather/AlltoAll)
 - Collective computation (MPI_Reduce/Allreduce/Scan)
- Programming considerations & restrictions
 - **Blocking operation**
 - No use of message tag argument
 - Collective operation within subsets of processes require separate grouping and new communicator
 - Can only be used with MPI predefined datatypes



Collective

Barrier synchronization and broadcast

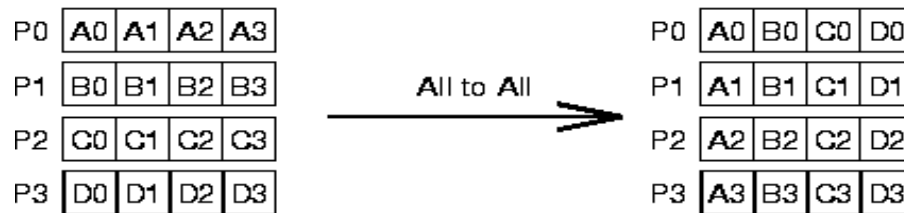
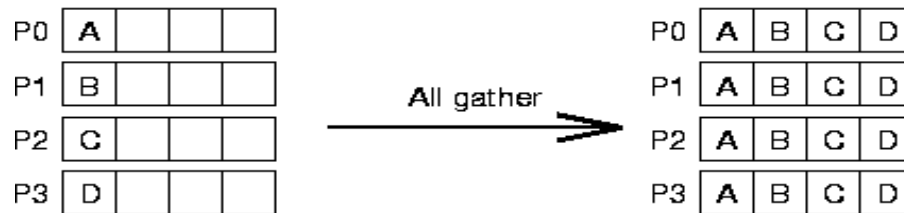
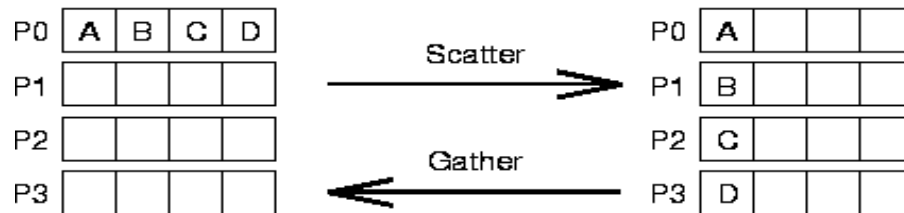
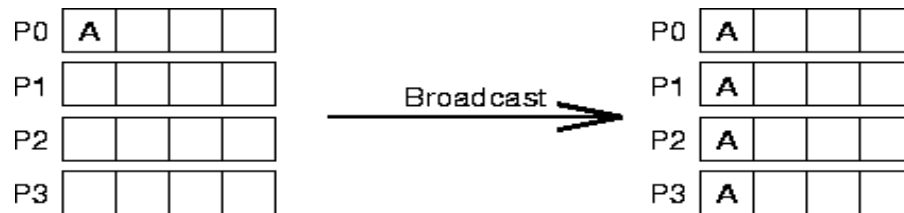
- *Barrier* blocks until all processes in comm have called it
- Useful when measuring communication/computation time
 - `mpi_barrier(comm, ierr)`
 - `MPI_Barrier(comm)`
- *Broadcast* sends data from root to all processes in comm
 - `mpi_bcast(data, count, type, root, comm, ierr)`
 - `MPI_Bcast(data, count, type, root, comm)`



Collective

Data movement

- Broadcast
- Scatter
- Gather
- Allgather
- Alltoall

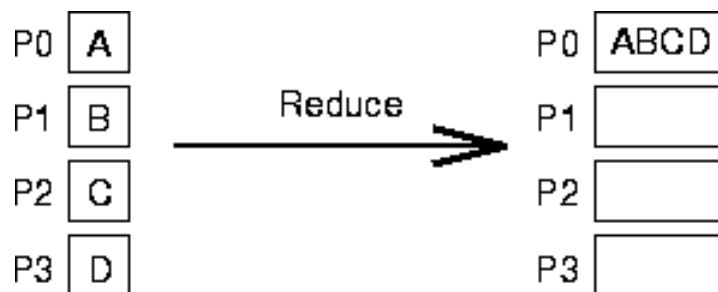




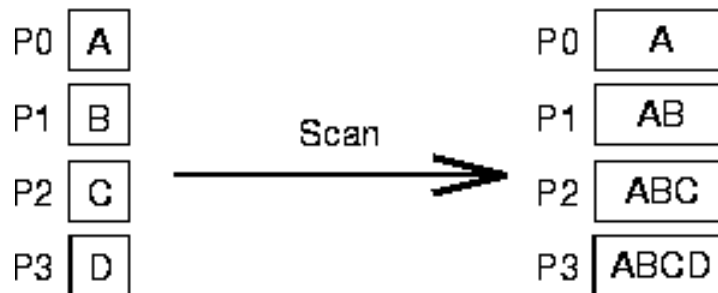
Collective

Reduction Operations

- Reduce



- Scan





Collective

Reduction Operations

Name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bit-wise and
MPI_LOR	Logical or
MPI_BOR	Bit-wise or
MPI_LXOR	Logical xor
MPI_BXOR	Logical xor
MPI_MAXLOC	Max value and location
MPI_MINLOC	Min value and location



Collective

C Example

```
#include <mpi.h>
#define WCOMM MPI_COMM_WORLD
main(int argc, char **argv){
    int npes, mype, ierr;
    double sum, val; int calc, knt=1;
    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_size(WCOMM, &npes);
    ierr = MPI_Comm_rank(WCOMM, &mype);

    val = (double) mype;

    ierr=MPI_Allreduce(&val,&sum,knt,MPI_DOUBLE,MPI_SUM,WCOMM);

    calc=(npes-1 +npes%2)*(npes/2);
    printf(" PE: %d sum=%5.0f calc=%d\n",mype,sum,calc);
    ierr = MPI_Finalize();
}
```



Collective

Fortran Example

```
program sum2all
include 'mpif.h'

    icomm = MPI_COMM_WORLD
    knt = 1
    call mpi_init(ierr)
    call mpi_comm_rank(icomm,mype,ierr)
    call mpi_comm_size(icomm,npes,ierr)
    val = dble(mype)

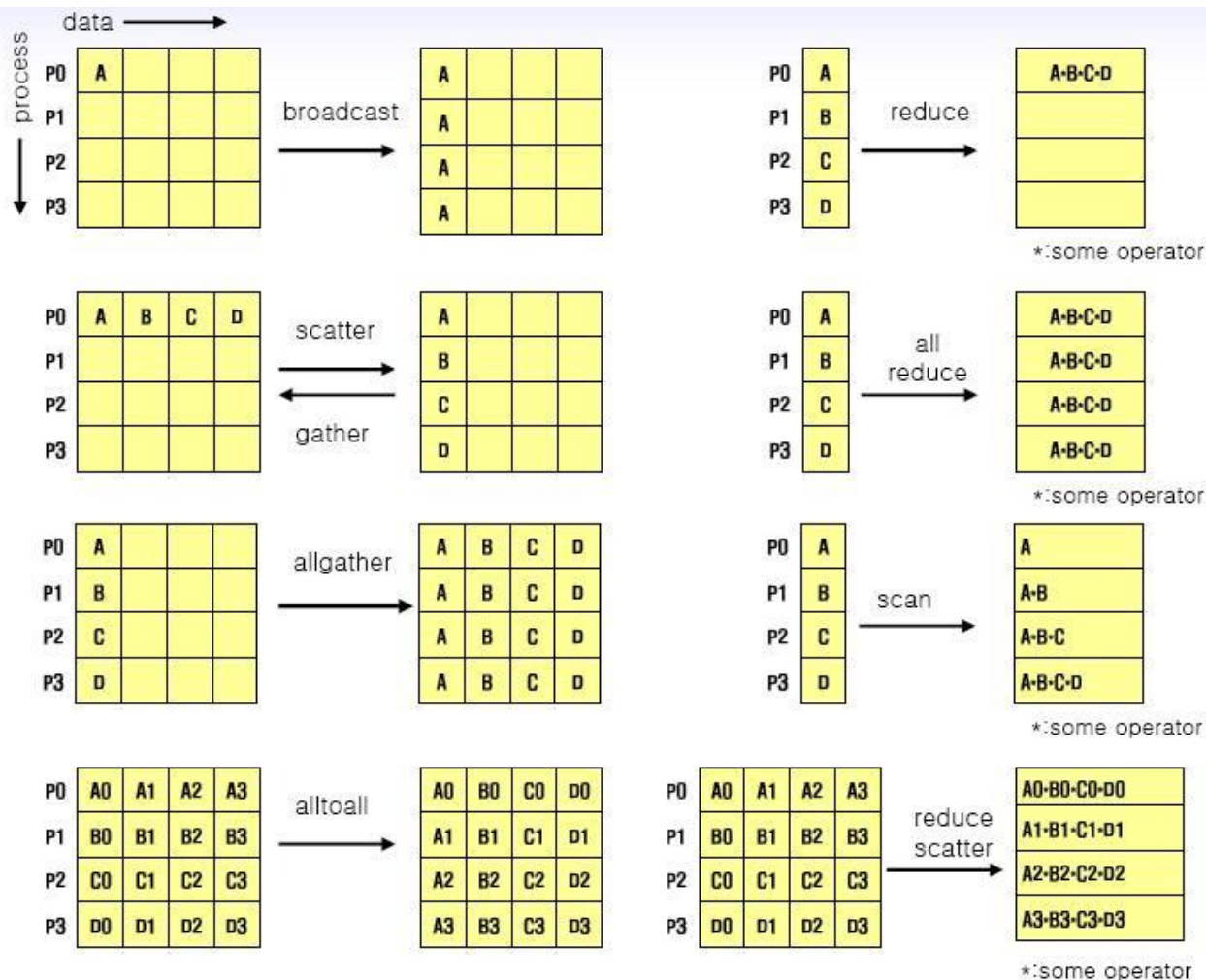
    call mpi_allreduce(val,sum,knt,MPI_REAL8,MPI_SUM,icomm,ierr)

    ncalc=(npes-1 + mod(npes,2))*(npes/2)
    print*,' pe#, sum, calc. sum = ',mype,sum,ncalc
    call mpi_finalize(ierr)

end program
```



Collective





MPI-1

- MPI-1 - Message Passing Interface (v. 1.2)
 - Library standard defined by committee of vendors, implementers, and parallel programmers
 - Used to create parallel SPMD codes based on explicit message passing
- Available on almost all parallel machines with C/C++ and Fortran bindings (and occasionally with other bindings)
- About 125 routines, total
 - 6 basic routines
 - The rest include routines of increasing generality and specificity



MPI-2

- Includes features left out of MPI-1
 - One-sided communications
 - Dynamic process control
 - More complicated collectives
 - Parallel I/O (MPI-IO)
- Implementations came along only gradually
 - Not quickly undertaken after the reference document was released (in 1997)
 - Now OpenMPI, MPICH2 (and its descendants), and the vendor implementations are nearly complete or fully complete
- Most applications still rely on MPI-1, plus maybe MPI-IO



References

- MPI-1 and MPI-2 standards
 - <http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html>
 - <http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.htm>
 - <http://www.mcs.anl.gov/mpi/> (other mirror sites)
- Freely available implementations
 - MPICH, <http://www.mcs.anl.gov/mpi/mpich>
 - LAM-MPI, <http://www.lam-mpi.org/>
- Books
 - *Using MPI*, by Gropp, Lusk, and Skjellum
 - *MPI Annotated Reference Manual*, by Marc Snir, *et al*
 - *Parallel Programming with MPI*, by Peter Pacheco
 - *Using MPI-2*, by Gropp, Lusk and Thakur
- Newsgroup: comp.parallel.mpi