



COSC 6365
Lecture 17
2008-03-11



Introduction to HPC

Lecture 17

Lennart Johnsson
Dept of Computer Science
Director TLC²



COSC 6365
Lecture 17
2008-03-11

CS@UH

Message Passing Model

- Assumptions (very general)
 - A parallel computation consists of a number of processes, each working on some local data.
 - Each process has purely local variables, and there is no mechanism for any process to directly access the memory of another.
 - Sharing of data between processes takes place by message passing, that is, by explicitly sending and receiving data between processes.
- The programmer needs to:
 - Explicitly divide data and work across the processes
 - Design and manage the communications among them.



COSC 6365
Lecture 17
2008-03-11

CS@UH

Message Passing Model (cont.)

- This approach is very flexible and extremely general.
 - Almost any type of parallel computation can be cast in the message passing form.
 - It can be implemented on a wide variety of platforms, from shared-memory multiprocessors to networks of workstations and even single-processor machines.
 - It allows more control over data location and flow within a parallel application than in the shared memory model.
 - Programs can often achieve higher performance
- It is also relatively low level and nontrivial to implement.
- It needs a library of communication subroutines/functions.



UNIVERSITY of HOUSTON

<http://www.lam-mpi.org/tutorials/nd/>

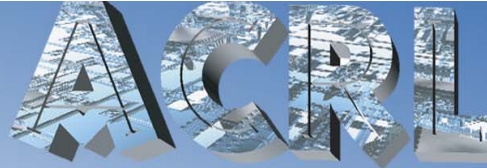


COSC 6365
Lecture 17
2008-03-11



What is MPI?

- MPI stands for "Message Passing Interface". It is a library of functions (in C) or subroutines (in Fortran) that you insert into source code to perform data communication between processes.
- MPI was developed over two years of discussions led by the MPI Forum, a group of roughly sixty people representing some forty organizations.
- It is not a single implementation of the library.
- It is a definition of an interface for the communication library with C and FORTRAN bindings.



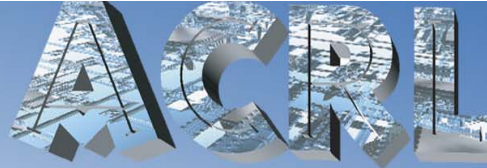
ADVANCED COMPUTING RESEARCH LABORATORY

COSC 6365
Lecture 17
2008-03-11

CS@UH

MPI Standard(s)

- The MPI-1 standard was defined in Spring of 1994.
 - it specifies the names, calling sequences, and results of subroutines and functions to be called from Fortran 77 and C, respectively.
 - All implementations of MPI must conform to these rules, thus ensuring portability.
 - The detailed implementation of the library is left to individual vendors, who are thus free to produce optimized versions for their machines.
 - Implementations of the MPI-1 are available for a wide variety of platforms.
- Corrections and clarifications in MPI-1.1 (June 1995) and MPI-1.2
- Extensions in MPI-2
 - tools for parallel I/O,
 - C++ and Fortran 90 bindings,
 - dynamic process management.



ADVANCED COMPUTING RESEARCH LABORATORY

COSC 6365
Lecture 17
2008-03-11

CS@UH

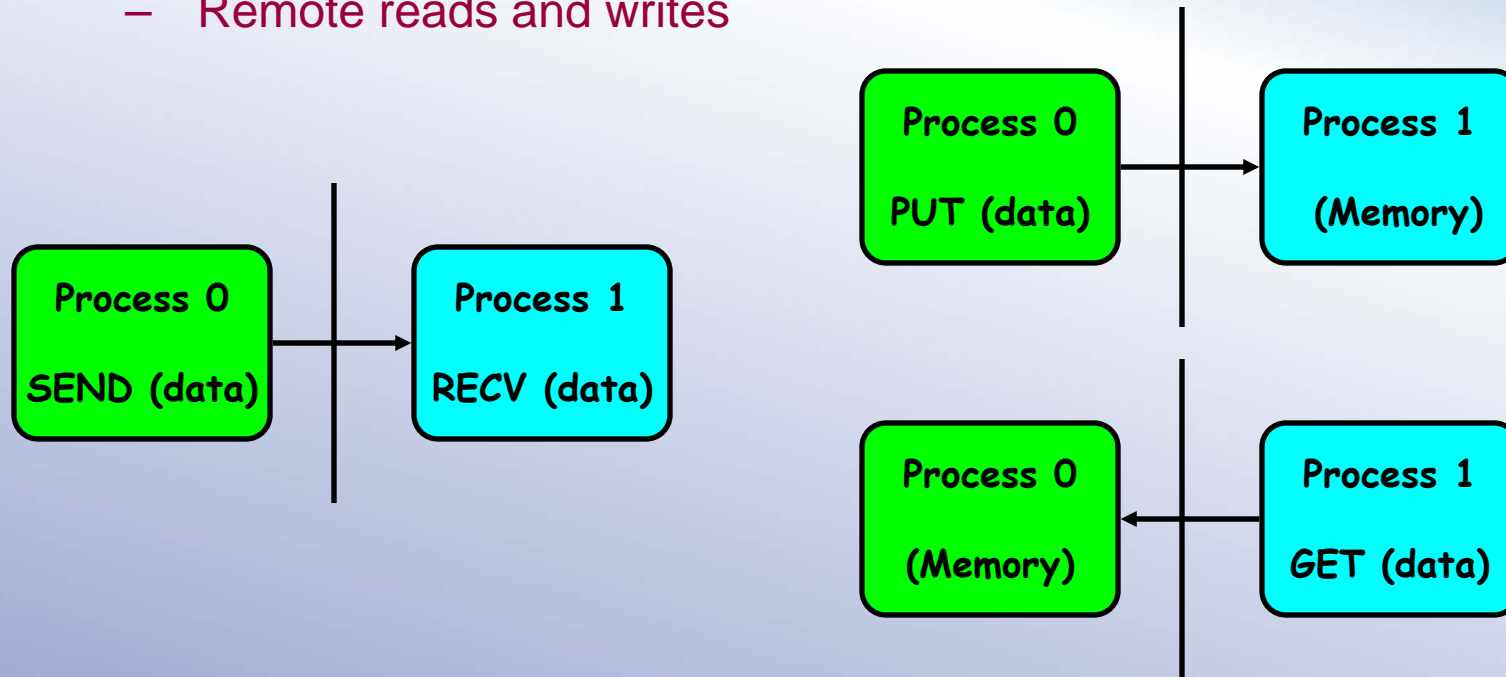
Goals of MPI

- The primary goals:
 - Provide source code portability.
 - Early vendor message passing libraries were not portable.
 - MPI programs should compile and run as-is on any platform.
 - Allow efficient implementations across a range of architectures.
- MPI also offers a great deal of functionality
 - Modularity,
 - Safe communication,
 - Number of different types of communication,
 - Special routines for common "collective" operations,
 - Application oriented process topologies
 - Built-in support for grids and graphs
 - Ability to handle user-defined data types and topologies,
 - Support for heterogeneous parallel architectures,
 - Support for performance measurements.



Communications between processes

- Type of data exchange between processes:
 - Cooperative: all parties agree to transfer data
 - One sided: one worker performs transfer of data
 - Remote reads and writes





Things not in MPI-1

- Some things that are explicitly outside the scope of MPI-1 are
 - The precise mechanism for launching an MPI program.
 - This is platform-dependent and you will need to consult your local documentation to find out how to do this.
 - Dynamic process management
 - changing the number of processes while the code is running.
 - Debugging
 - Parallel I/O
- Several of these issues are addressed in MPI-2.



COSC 6365
Lecture 17
2008-03-11

CS@UH

How big is MPI?

- MPI is large
 - MPI-1 has 128 functions
 - MPI-2 has 152 functions
- MPI is small (6 functions)
 - Many parallel programs can be written with just 6 basic functions
- MPI is just right
 - One can access flexibility when needed.
 - One need not master all parts of MPI to use it.



COSC 6365
Lecture 17
2008-03-11

CS@UH

When to use MPI?

- You should use MPI when you need to
 - Write portable parallel code.
 - Achieve high performance in parallel programming, e.g. when writing parallel libraries,
 - Handle a problem that involves irregular or dynamic data relationships that do not fit well into the "data-parallel" model,
 - Don't like (or know) the other approaches!
- You should not use MPI when you
 - Can achieve sufficient performance and portability using a data-parallel (e.g., High-Performance Fortran) or shared-memory approach (e.g., OpenMP, or proprietary directive-based paradigms).
 - Can use a pre-existing library of parallel routines (which may themselves be written using MPI).
 - Don't need parallelism at all!



COSC 6365
Lecture 17
2008-03-11



Basic Features of Message Passing Programs

- Message passing programs consist of multiple instances of a serial program that communicate by library calls. These calls may be roughly divided into four classes:
 1. Calls used to initialize, manage, and finally terminate communications.
 2. Calls used to communicate between pairs of processors.
 3. Calls that perform communications operations among groups of processors.
 4. Calls used to create arbitrary data types.

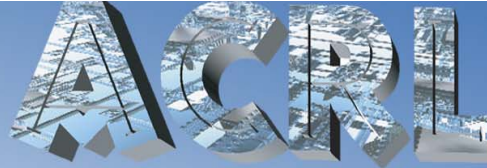


COSC 6365
Lecture 17
2008-03-11

CS@UH

Basic Features (cont.)

- The first class of calls consists of calls for
 - starting communications,
 - identifying the number of processors being used,
 - creating subgroups of processors,
 - identifying which processor is running a particular instance of a program.
- The second class of calls: **point-to-point communications** consists of different types of send and receive operations.
- The third class of calls: **collective operations**
 - synchronization among groups of processes
 - communications operations among groups of processes
 - communication/calculation operations among groups of processes
- The final class of calls provides flexibility in dealing with complicated data structures.



ADVANCED COMPUTING RESEARCH LABORATORY

COSC 6365
Lecture 17
2008-03-11

CS@UH

Getting Started

- Six-function MPI
- Overview
 - Write MPI program
 - Compiling and linking
 - Running MPI programs
- A First Program: Hello World!

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

```
PROGRAM hello
    INCLUDE 'mpif.h'
    INTEGER err
    CALL MPI_INIT(err)
    PRINT *, "Hello world!"
    CALL MPI_FINALIZE(err)
END
```



Commentary

- MPI functions/subroutines have names that begin with MPI_.
- There is an MPI header file (mpi.h or mpif.h) containing definitions and function prototypes that is imported via an "include" statement.
- MPI routines return an error code indicating whether or not the routine ran successfully.
- Each process executes a copy of the entire code, so the output of this program depends on the number of processors:

2 Processors:

Hello world!
Hello world!

4 Processors:

Hello world!
Hello world!
Hello world!
Hello world!

8 Processors:

Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!



COSC 6365
Lecture 17
2008-03-11

CS@UH

Commentary

- Starting MPI:

```
int MPI_Init(int *argc, char *argv[]);  
C
```

```
MPI_INIT(IERR)
```

```
INTEGER IERR
```

- Exiting MPI:

```
int MPI_Finalize(void)
```

```
MPI_FINALIZE(IERR)
```

```
INTEGER IERR
```



COSC 6365
Lecture 17
2008-03-11



MPI Naming Conventions

- The names of all MPI entities (routines, constants, types, etc.) begin with MPI_ to avoid conflicts.

- Fortran routine names are conventionally all upper case:

`MPI_XXXXX (parameter, ... , IERR)`

Example: `MPI_INIT (IERR) .`

- C function names have a mixed case:

`MPI_Xxxxx (parameter, ...)`

Example: `MPI_Init (&argc, &argv) .`

- The names of MPI constants are all upper case in both C and Fortran, for example,

`MPI_COMM_WORLD, MPI_REAL, ...`



COSC 6365
Lecture 17
2008-03-11

CS@UH

MPI Handles

- MPI defines and maintains its own internal data structures.
 - accessible through structure handles.
- Handles are returned by various MPI calls and may be used as arguments in other MPI calls.
- In C, handles are pointers to specially defined datatypes
 - Created via the C `typedef` mechanism
 - Arrays are indexed starting at 0.
- In Fortran, handles are integers or arrays of integers
 - Arrays are indexed starting at 1.
- Examples:
 - MPI_SUCCESS - An integer in both C and Fortran. Used to test error codes.
 - MPI_COMM_WORLD - A pre-defined communicator consisting of all processors.
 - In C, an object of type `MPI_Comm` (a "communicator");
 - In Fortran, an integer.
- Handles may be copied using the standard assignment operation in both C and Fortran.



COSC 6365
Lecture 17
2008-03-11

CS@UH

MPI Datatypes

- MPI provides its own reference datatypes corresponding to the various elementary datatypes in C and Fortran.
 - Variables are normally declared as C/Fortran types.
 - MPI type names are used as arguments in MPI routines when a type is needed.
- MPI hides the details of, e.g., the floating-point representation, which is an issue for the implementor.



Basic MPI Data Types - C

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	(none)
MPI_PACKED	(none)



Basic MPI Datatypes - Fortran

MPI Datatype	Fortran Datatype
MPI_CHARACTER	character(1)
MPI_REAL	real
MPI_LOGICAL	logical
MPI_DOUBLE_PRECISION	double precision
MPI_COMPLEX	complex
MPI_INTEGER	integer
MPI_BYTE	(none)
MPI_PACKED	(none)



COSC 6365
Lecture 17
2008-03-11



Special MPI Datatypes (C)

- In C, MPI provides several special datatypes (structures). Examples include

`MPI_Comm` - a communicator

`MPI_Status` - a structure containing several pieces of status information for MPI calls

`MPI_Datatype`

- These are used in variable declarations, for example,

```
MPI_Comm my_comm;
```

declares a variable called `my_comm`, which is of type `MPI_Comm` (i.e. a communicator).

- In Fortran, the corresponding types are all INTEGERS.



COSC 6365
Lecture 17
2008-03-11



Communicators

- A communicator is a handle representing a group of processors that can communicate with one another.
- The communicator name is required as an argument to all point-to-point and collective operations.
 - The communicator specified in the send and receive calls must agree for communication to take place.
 - Processors can communicate only if they share a communicator.
- There can be many communicators, and a given processor can be a member of a number of different communicators.
- Within each communicator, processors are numbered consecutively (starting at 0). This identifying number is known as the rank of the processor in that communicator.
- The rank is also used to specify the source and destination in send and receive calls.



Communicators (cont.)

- If a processor belongs to more than one communicator, its rank in each can (and usually will) be different!
- MPI automatically provides a basic communicator called `MPI_COMM_WORLD`. It is the communicator consisting of all processors.
- You can define additional communicators consisting of subsets of the available processors.
- A processor can determine its rank in a communicator with
 - Fortran: `MPI_COMM_RANK(COMM, RANK, IERR)`
 - C: `int MPI_Comm_rank(MPI_Comm comm, int *rank);`
- A processor the size of the communicator with
 - Fortran: `MPI_COMM_SIZE(COMM, SIZE, IERR)`
 - C: `int MPI_Comm_size(MPI_Comm comm, int *size);`



Sample Program: Hello World! #2

- In hello2.c we write:

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char
           *argv[]) {
    int myrank, size;
    /*Initialize MPI */
    MPI_Init(&argc, &argv);
    /* Get my rank */
    MPI_Comm_rank(MPI_COMM_WORLD,
                  &myrank);
    /* Get the number of processors*/
    MPI_Comm_size(MPI_COMM_WORLD,
                  &size);
    printf("Processor %d of %d:
           Hello World!\n", myrank,
           size);
    MPI_Finalize(); /*Terminate MPI
    */
}
```

- In hello2.f we write:

```
PROGRAM hello
INCLUDE 'mpif.h'
INTEGER myrank, size, ierr
C Initialize MPI:
  call MPI_INIT(ierr)
C Get my rank:
  call
  MPI_COMM_RANK(MPI_COMM_WORLD,
                myrank, ierr)
C Get the total number of
processors:
  call
  MPI_COMM_SIZE(MPI_COMM_WORLD,
                size, ierr)
  PRINT *, "Processor", myrank,
  "of", size, ": Hello World!"
C Terminate MPI:
  call MPI_FINALIZE(ierr)
```



COSC 6365
Lecture 17
2008-03-11

CS@UH

Compiling and Running MPI Programs

- The MPI standard **does not** specify how MPI programs are to be started. Thus, implementations vary from machine to machine.
- When compiling an MPI program, it may be necessary to link against the MPI library. Typically, to do this, you would include a loader option
-lmpi
- To run an MPI code, you commonly use a "wrapper" called **mpirun** or **mpprun**. The following command would run the executable a.out on four processors:

```
$ mpirun -np 4 a.out
```
- For further details on using MPI on different platforms, consult the local information.



COSC 6365
Lecture 17
2008-03-11

CS@UH

Point-to-Point Communications

- The elementary communication operation in MPI
 - direct communication between two processors, one of which sends and the other receives.
- Point-to-point communication in MPI is "two-sided"
 - both an explicit send and an explicit receive are required. Data are not transferred without the participation of both processors.
- A message consists of
 - an envelope, indicating the source and destination processors, and
 - a body, containing the actual data to be sent.



COSC 6365
Lecture 17
2008-03-11

CS@UH

Point-to-Point Communications

- MPI uses three pieces of information to characterize the message body in a flexible way:
 1. **Buffer *** - the starting location in memory where outgoing data is to be found (for a send) or incoming data is to be stored (for a receive).
 2. **Datatype** - the type of data to be sent.
 - elementary type such as float/REAL, int/INTEGER, etc.
 - a user-defined type (analogous to C structures, and can contain data located anywhere, i.e., not necessarily in contiguous memory locations).
 3. **Count** - the number of items of type datatype to be sent.



COSC 6365
Lecture 17
2008-03-11



Point-to-Point Communications

- Note:
 - MPI standardizes the designation of the elementary types. This means that you don't have to explicitly worry about differences in how machines in heterogeneous environments represent them, e.g., differences in representation of floating-point numbers.
 - In C, `buffer` is the actual address of the array element where the data transfer begins.
 - In Fortran, it is just the name of the array element where the data transfer begins. (Fortran actually gets the address behind the scenes.)



COSC 6365
Lecture 17
2008-03-11

CS@UH

Communication Modes and Completion Criteria

- MPI provides a great deal of flexibility in specifying how messages are to be sent.
- There are four communication modes available for sends:
 - Standard
 - Synchronous
 - Buffered
 - Ready
- For receives there is only a single communication mode.
 - A receive is complete when the incoming data has actually arrived and is available for use.



COSC 6365
Lecture 17
2008-03-11



Blocking and Nonblocking Communication

- In addition to the communication mode used, a send or receive may be blocking or nonblocking.
- A blocking send or receive does not return from the subroutine call until the operation has actually completed.
 - With a blocking send, for example, you are sure that the variables sent can safely be overwritten on the sending processor.
 - With a blocking receive, you are sure that the data has actually arrived and is ready for use.
- A nonblocking send or receive returns immediately, with no information about whether the completion criteria have been satisfied. You can test later to see whether the operation has actually completed.
 - For example, a nonblocking synchronous send returns immediately.
 - The sending processor can then do other useful work, testing later to see if the send is complete.



COSC 6365
Lecture 17
2008-03-11

CS@UH

Sending and Receiving Messages

- Messages consist of 2 parts:
 - the envelope and
 - the message body.
- The envelope of an MPI message has 4 parts:
 1. source - the sending process;
 2. destination - the receiving process;
 3. communicator - specifies a group of processes to which both source and destination belong
 4. tag - used to classify messages.
- The message body has 3 parts:
 1. buffer - the message data;
 2. datatype - the type of the message data;
 3. count - the number of items of type datatype in buffer.



Blocking Send and Receive

- MPI_SEND and MPI_RECV, are the basic point-to-point communication routines in MPI
- Both functions block the calling process until the communication operation is completed.
- Blocking creates the possibility of deadlock.
- MPI_SEND C and Fortran bindings:

```
int MPI_Send(void *buf, int count, MPI_Datatype dtype,  
            int dest, int tag, MPI_Comm comm);
```

—All arguments are input arguments.

—An error code is returned by the function.

```
MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, IERR)
```

—The input argument BUF is an array; its type should match the type given in DTYPE.

—The input arguments COUNT, DTYPE, DEST, TAG, COMM are of type INTEGER.

—The output argument IERR is of type INTEGER;



Receiving a Message: MPI_RECV

- C bindings:

```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int  
    source, int tag, MPI_Comm comm, MPI_Status *status);
```

—The output arguments are `buf` and `status`; the rest are inputs.

—An error code is returned by the function.

- Fortran bindings:

```
MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM, STATUS,  
    IERR)
```

—The output argument `BUF` is an array; its type should match the type in `DTYPE`.

—The input arguments `COUNT`, `DTYPE`, `SOURCE`, `TAG`, `COMM` are of type `INTEGER`.

—The output argument `STATUS` is an `INTEGER` array with `MPI_STATUS_SIZE` elements.

—The output argument `IERR` is of type `INTEGER`; it contains an error code when `MPI_RECV` returns.



COSC 6365
Lecture 17
2008-03-11

CS@UH

MPI_RECV (Comments)

- Notes:
 - A maximum of `COUNT` items of type `DTYPE` are accepted; if the message contains more, it is an error.
 - The sending and receiving processes must agree on the datatype; if they disagree, results are undefined (MPI does not check).
 - When this routine returns, the received message data have been copied into the buffer; and the tag, source, and actual count of data received are available via the status argument.



COSC 6365
Lecture 17
2008-03-11

CS@UH

C Example: Send and Receive

```
/* simple send and receive */
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv) {
    int i, myrank;
    MPI_Status status;
    double a[100];
    MPI_Init(&argc, &argv); /* Initialize MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
    for( i=0; i<100; i++) a[i] = (double)myrank;
    if( myrank == 0 ) /* Send a message */
        MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
    else if( myrank == 1 ) /* Receive a message */
        MPI_Recv( a, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status
    );
    MPI_Finalize(); /* Terminate MPI */
}
```



COSC 6365
Lecture 17
2008-03-11



Fortran Example: Send and Receive

```
PROGRAM simple_send_and_receive
INCLUDE 'mpif.h'
INTEGER myrank, ierr, status(MPI_STATUS_SIZE)
REAL a(100)
C Initialize MPI:
  call MPI_INIT(ierr)
C Get my rank:
  call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
C Process 0 sends, process 1 receives:
  if( myrank.eq.0 )then
    call MPI_SEND( a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)
  else if ( myrank.eq.1 )then
    call MPI_RECV( a, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, status, ierr
  )
  endif
C Terminate MPI:
  call MPI_FINALIZE(ierr)
END
```



Blocking and Completion

- when a message is sent using MPI_SEND, the message is either
 - Copied to a buffer and delivered later asynchronously, or
 - The sending and receiving processes synchronize.
- MPI_SEND and MPI_RECV block the calling process
- Completion
 - MPI_RECV:
 - a matching message has arrived, and the message's data have been copied into the output arguments of the call.
 - the variables passed to MPI_RECV contain a message and are ready to be used.
 - MPI_SEND:
 - the message specified in the call has been handed off to MPI (either buffered or delivered)
 - the variables passed to MPI_SEND can be overwritten and reused.



Deadlock

- occurs when 2 (or more) processes are blocked and each is waiting for the other to make progress.

- Deadlock example:

```
if( myrank ==0 ){  
    MPI_Recv(b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );  
    MPI_Send(a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );  
}else if( myrank ==1 ){  
    MPI_Recv(b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );  
    MPI_Send(a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );  
}
```

- Avoiding deadlock (safe):

```
else if( myrank ==1 ){  
    MPI_Send(a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );  
    MPI_Recv(b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );  
}
```



Avoiding Deadlock

- Avoiding Deadlock (unsafe):

```
if( myrank ==0 ){
    MPI_Send(a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
    MPI_Recv(b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
}
else if( myrank ==1 ){
    MPI_Send(a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
    MPI_Recv(b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
}
```

- Success depends on the availability of buffering in MPI.
- Under most MPI implementations, the program will run to completion.
- However, if the message sizes are increased, sooner or later the program will deadlock.
- The best practice is to write programs that run to completion regardless of the availability of MPI internal buffer.