Parallel Computing for Science & Engineering Introduction to MPI Spring 2018

Instructors:
Charlie Dey, TACC
Lars Koesterke, TACC

Outline

computing.llnl.gov/tutorials/mpi
mpi-forum.org/docs/ --pdfs
www.mpich.org/static/docs/v3.2/www3/

- Tricks
- Data Types
  - Predefined
  - Status and Wildcards
- Pt-2-Pt Communication Modes
  - Blocking
    - Standard, Buffered, Synchronous, Ready
  - Non-blocking
    - Standard, Buffered, Synchronous, Ready
- 1- and 2-way Communication & Deadlocking

Tricks

- mpicc/mpif90 -show myprog.c/f90
  - Shows details of compiling and loading
  - Shows where include files are located.
- watch –n 8 squeue -u <username>
  - Shows results of squeue every 8 seconds
  - “squeue” can be any command.

MPI Data Types

- MPI data types are used in data communication operation.
- MPI has many different predefined data types
  - Defined to match C/Fortran data types
- MPI handles endianness conversion (though a mixed architecture system is rare)
- Packed/opaque types– User Defined Types can be made to handle C/F90 structures

MPI Predefined Data Types in C

C MPI Types	
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 i
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long ir
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	-
MPI_PACKED	-

MPI Predefined Data Types in F90

MPI Parameter	F90 type
MPI_INTEGER	Integer
MPI_REAL	Real
MPI_DOUBLE_PRECISION	Double Precision
MPI_COMPLEX	Complex
MPI_LOGICAL	Logical
MPI_CHARACTER	Character
MPI_BYTE	Raw Byte (no conversion)
MPI_PACKED	MPI calls pack/unpack

Experiment #1

Using your Ping-Pong program from Tuesday:

Try sending a data buffer of one type and receive a data buffer of a different type.

Try sending and receiving a different count size

Try sending and receiving messages with different tags.

Experiment #2

Using your Ping-Pong program from Tuesday:

Have each task send a message back and forth to all the other tasks

Example, we have 8 tasks acquired,

task 1 will send and receive a message to tasks 2, 3, 4, 5, 6, 7, 8
task 2 will send and receive a message to tasks 3, 4, 5, 6, 7, 8, 1
task 3 will send and receive a message to tasks 4, 5, 6, 7, 8, 1, 2
…
…
…
….
(hopefully you see the pattern)

Wildcards

- Enables programmer to avoid having to specify a tag and/or source.
- Example:

```
MPI_Status status;
int        data[5];
int        ierr;
ierr = MPI_Recv(&data[0], 5, MPI_INT,
                MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD,&status);
```

- **`MPI_ANY_SOURCE`** and **`MPI_ANY_TAG`** are wild cards
- **`status`** structure is used to get wildcard values

MPI_STATUS_IGNORE: not practical here

Wildcards

- Enables programmer to avoid having to specify a tag and/or source.
- Example:

```
integer :: status;
integer :: data(5);
integer :: ierr;
call   MPI_Recv( data,    5, MPI_INTEGER,
                 MPI_ANY_SOURCE, MPI_ANY_TAG,
                 MPI_COMM_WORLD, status, ierr);
```

- **`MPI_ANY_SOURCE`** and **`MPI_ANY_TAG`** are wild cards
- **`status`** structure is used to get wildcard values

Wildcards

- `MPI_PROC_NULL`
  - can be used for destination or source in send or receive calls
  - operation completes immediately
  - no communications involved
- Great for handling edges of partitioned data
- Useful with Generic Send/Recv & `MPI_Sendrecv`

Blocking Pt-2-Pt communications

	Standard	Buffered	Synchronous	Ready	← Mode
call MPI_				Recv Posted	
Send Start	MPI_Send	MPI_Bsend	MPI_Ssend	MPI_Rsend	
Blocked until data area free i.e. Send Complete	In System buffer or Recv posted and data copied	data moved to user buffer	Receive posted & data copied	data copied	
Return	local non-local	local	non-local	non-local	Call Interaction

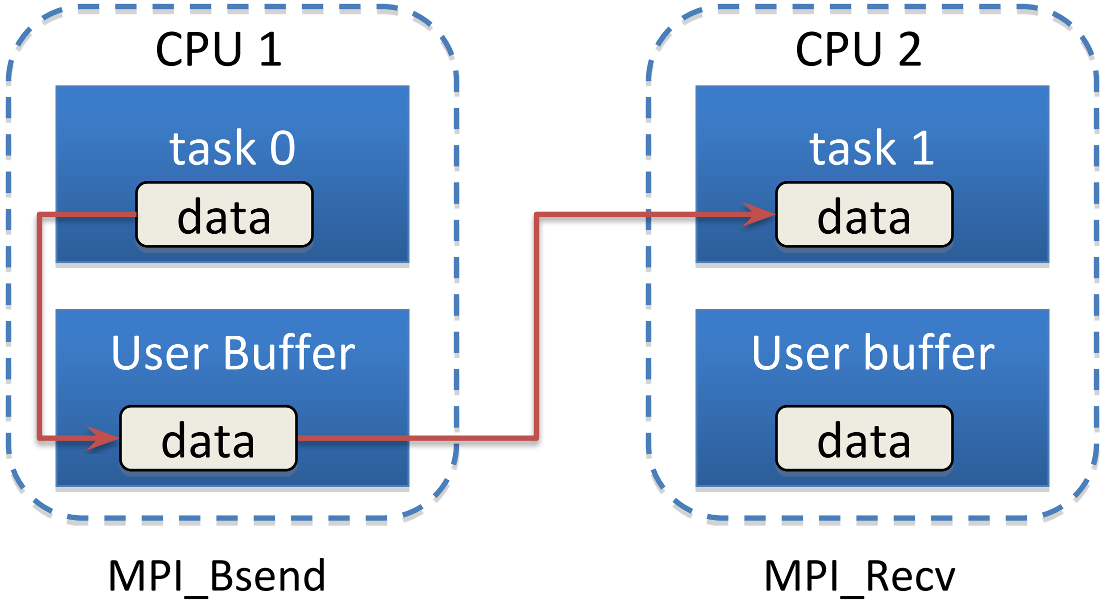
MPI_Recv is used with MPI_Send, MPI_Bsend & MPI_Ssend and MPI_Rsend.

MPI Receive Modes

IMPORTANT: From the MPI-2 Standard:

There is only one receive operation, but it matches any of the send modes. The receive operation described in the last section is blocking: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed -- an ACK in sync must be seen by send before it can continue -- (of course, it can complete only after the matching send has started).

Buffered Communication



- The content of the message is copied into a system-controlled block of memory (User Buffer).
- MPI_Bsend returns when the copy to User buffer is complete.
- There is no MPI_Brecv.
- Use MPI_BSend_OVERHEAD to provide room for message headers
- Fails if there isn't enough space for buffering
- Buffer area must contain (MPI_BSEND_OVERHEAD) room for each message.

User-Buffer Communication

- **BSend**

```
...
char* cbuffer;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &irank);

i = 1;
isize_bytes = sizeof(i) + MPI_BSend_OVERHEAD ;
cbuffer = malloc((size_t)isize_bytes );
MPI_Buffer_attach(cbuffer , isize_bytes );

if(irank == 0) {
   MPI_Bsend(&i, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
} else {
   MPI_Recv( &j, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, &status);
}
```

User-Buffer Communication

- **BSend**

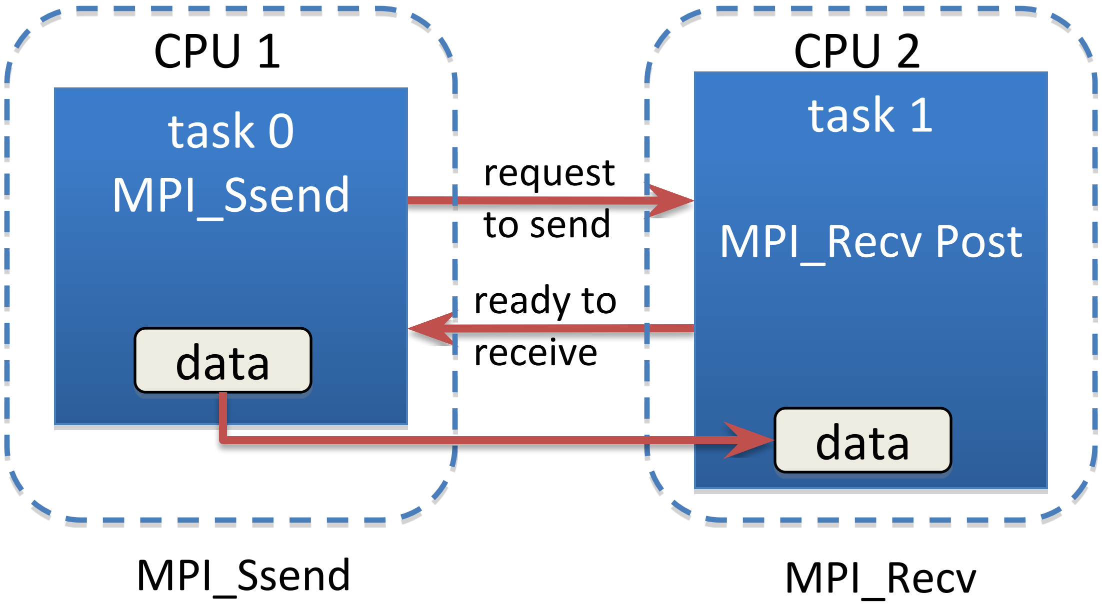
```
...
character,allocatable,dimension(:) :: cbuffer

call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, irank, ierr)

i=1
isize_bytes = sizeof(i) + MPI_BSend_OVERHEAD
allocate( cbuffer(isize_bytes) )
call MPI_Buffer_attach (cbuffer , isize_bytes , ierr )

if(irank == 0) then
   call MPI_Bsend(i, 1, MPI_INTEGER, 1, 9, MPI_COMM_WORLD, ierr)
else
   call MPI_Recv( j, 1, MPI_INTEGER, 0, 9, MPI_COMM_WORLD, status, ierr)
endif
```

Synchronous Communication



- Data isn't sent until Receive has been posted.
- Synchronous send returns when data area is safe for re-use.
- There is no MPI_Srecv

synchronous Communication

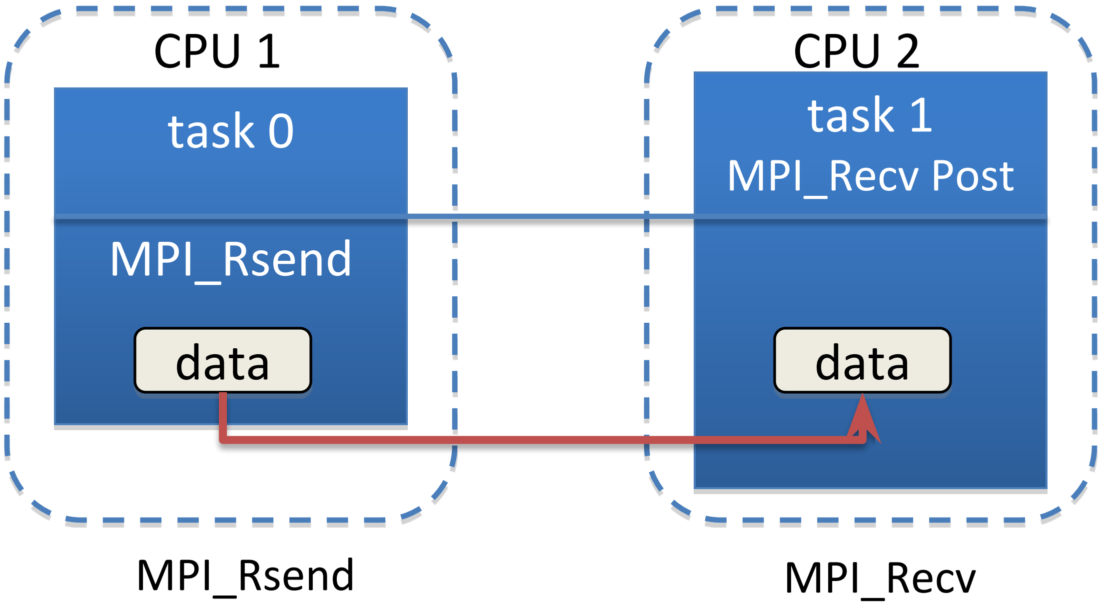
- **Ssend C**

```
     ...
 i=1;
if(irank == 0){
         MPI_Ssend(&i, 1, MPI_INT,      1, 9, MPI_COMM_WORLD);
}else {
         MPI_Recv( &j, 1, MPI_INT,      0, 9, MPI_COMM_WORLD, &status);
}
```

- **Ssend F90**

```
  ...
 i=1
if(irank == 0) then
   call MPI_Ssend( i, 1, MPI_INTEGER, 1, 9, MPI_COMM_WORLD, ierr)
else
   call MPI_Recv(  j, 1, MPI_INTEGER, 0, 9, MPI_COMM_WORLD, status,ierr)
endif
```

Ready Communication



- Receive is guaranteed to be posted.
- Ready returns when data area is safe for re-use.
- Not often used. Behavior is not defined if receive has not been posted first.
- There is no MPI_Rrecv. You might find it in some MPI implementations but it is NOT part of the MPI-2 standard

Ready Communication

- **Rsend F90**

```
    ...
i=1;
if(irank == 0)then

   call MPI_Rsend(i, 1, MPI_INTEGER, 1, 9, MPI_COMM_WORLD, ierr);
else
   call MPI_Recv(…);

    ...
endif
```

- **Rsend C**

```
  ...
i=1;
if(irank == 0){

   MPI_Rsend(&i, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
}else {
   MPI_Recv(…);

 ...}
```

Need MPI_Recv to be posted 1st! then execute MPI_Rsend. Synchronize through barrier.

Need Barrier before MPI_Rsend, but after MPI_Recv– problem!

Ready Communication

- **Rsend F90**

```
   ...
i=1;
if(irank == 0)then
   call MPI_Barrier(MPI_COMM_WORLD, ierr);
   call MPI_Rsend(i, 1, MPI_INTEGER, 1, 9, MPI_COMM_WORLD, ierr);
else
   call MPI_Irecv(…);
   call MPI_Barrier(MPI_COMM_WORLD, ierr);
    ...
endif
```

- **Rsend C**

```
  ...
i=1;
if(irank == 0){
   MPI_Barrier(MPI_COMM_WORLD);
   MPI_Rsend(&i, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
}else {
   MPI_Irecv(…);
   MPI_Barrier(MPI_COMM_WORLD);
 ...}
```

Barrier is a collective communication:
All task must encounter a barrier.

More about this later.
This allows Recv to be posted
without blocking--
the call returns immediately.

MPI_SendRecv

```
MPI_SendRecv(senddat,sendcount,sendtype, dest,sendtag,
           recvdat,recvcount,recvtype, src, recvtag,
comm, status)
```

- Initiates send and receive at the same time. (Why is that important?)
- Completes when both send and receive buffers are safe to use
- Useful for communication patterns where each task (rank) sends and receives messages (neighbor (stencil) updates --two-way communication). Good for avoiding deadlock, implementing shifts/rings.
- Executes a **standard mode** send & receive operation for dest and src, respectively.
- The send and receive operations use the same communicator, but have distinct tags.

Bidirectional Communication with MPI_Sendrecv

- C

```
ierr=MPI_Sendrecv(&sb[0],scnt,stype,dest,stag,
                  &rb[0],rcnt,rtype, src,rtag,
                  MPI_COMM_WORLD,&status);
```

- Fortran

```
call MPI_Sendrecv( sb,  scnt,stype,dest,stag,
  rb,  rcnt,rtype, src,rtag,
                  MPI_COMM_WORLD, status,ierr)
```

Blocking vs Non-blocking

Blocking

- A blocking send routine will only return after it is *safe* to modify the data area.
- *Safe* means that modifications in the data area will not affect the data to be sent.
- *A Safe send* does not imply that the data was actually received.
- A "standard" send can be either synchronous or asynchronous.

Non-blocking

- Send/receive routines return immediately.
- Non-blocking operations request the MPI library to perform the operation when possible.
- It is **unsafe** to modify the data area until the requested operation has been performed. There are *wait* routines used to do this (MPI_Wait)
- Both--Send/Receives can be Synchronous. Primarily used to overlap computation with communication

Blocking vs non-Blocking Routines

Description	Syntax for C bindings
Blocking send	MPI_Send(buf,count, datatype, dest, tag, comm)
Non-blocking send	MPI_Isend(buf,count, datatype, dest, tag, comm, request)
Blocking receive	MPI_Recv(buf,count, datatype, source, tag, comm, status)
Non-blocking receive	MPI_Irecv(buf,count, datatype, source, tag, comm, request)
Wait for completion	MPI_Wait(request, status)

request: used by non-blocking send/receive operation.

Non-blocking Communication

- Non-blocking send
  - send call returns Immediately-- send actually occurs later
  - When send data area is needed-- call MPI_wait
- Non-blocking receive
  - receive call returns Immediately-- receive actually occurs later
  - when received data is needed-- call MPI_wait
- Non-blocking communication is used to overlap communication with computation
  (and communication with communication!).
- Can be used to prevent deadlock.

Non-Blocking Pt-2-Pt communications

call MPI_	Standard	Buffered	Synchronous	Ready	← Mode
				Recv Posted	
Send Start	MPI_Isend	MPI_Ibsend	MPI_Issend	MPI_Irsend	
Return data area not free					
Complete When	In System buffer or Recv posted and data copied	data moved to user buffer	Receive posted & data copied	data copied	
	local non-local	local	non-local	non-local	Call Interaction

Completeness

Test⟶ MPI_Tes

Guarantee⟶ MPI_tWait

MPI_Irecv or MPI_Recv is used with: MPI_Isend, MPI_Ibsend, MPI_Issend, MPI_Irsend & blocking versions.

Non-blocking Send with MPI_Isend

- C

```
MPI_Request request;
ierr = MPI_Isend(&data, count, datatype,
                  dest, tag,   comm, &request);
```

- Fortran

```
integer request
call MPI_Isend(   data, count, datatype,
                  dest, tag,   comm,  request, ierr)
```

- `request` is the id for the message call
- Don't use `data` area until communication is complete

Non-blocking Receive with MPI_Irecv

- C

```
MPI_Request request;
ierr = MPI_Irecv(&data, count, datatype,
                 source, tag,   comm, &request);
```

- Fortran

```
integer request
call MPI_Irecv(   data, count, datatype,
                 source, tag,   comm,  request, ierr)
```

- `request` is an id for communication
- Note: There is **no status parameter**.
- Don't use **`data`** area until communication is complete

MPI_Wait Used to Complete Communication

- `request` from `MPI_Isend` or `MPI_Irecv`
  - completion of a send operation indicates that the sender is now free to update the data in the send buffer
  - completion of a receive operation indicates that the receive buffer contains the received message
- `MPI_Wait` blocks until message specified by `request` is complete

MPI_Wait Usage

- C

```
MPI_Request request;
MPI_Status status;
...
ierr = MPI_Wait(&request, &status)
```

- Fortran

```
integer request
integer status(MPI_STATUS_SIZE)
...
call MPI_Wait(   request,  status, ierr)
```

MPI_Test

- Flag value indicates message completeness
- Call is similar to **`MPI_Wait`**, but does not block
- C

```
int flag;
ierr= MPI_Test(&request, &flag, &status);
```

- Fortran

```
logical flag
call MPI_Test( request,  flag,  status, ierr)
```

MPI_Cancel

- Cancel a pending non-blocking send or receive
- C

```
MPI_Request request;
ierr= MPI_Cancel(&request);
```

- Fortran

```
integer request
call MPI_Cancel(  request, ierr)
```

Order Semantics

- Messages with the same tag are ordered
  - the first receive always matches the first send in the following

```
tag=123456
if (rank.EQ.0) then
  call MPI_BSend(b1,cnt,MPI_REAL,1,tag,comm,err)
  call MPI_BSend(b2,cnt,MPI_REAL,1,tag,comm,err)
ELSE ! rank.EQ.1
  call MPI_Recv(b1,cnt,MPI_REAL,0,tag,comm,
  status,ierr)
  call MPI_Recv(b2,cnt,MPI_REAL,0,tag,comm,
       status, ierr)
END if
```

One-way Send/Recv Communication

- **Non-blocking Send & Blocking Recv**

```
if (rank==0) then
     call MPI_Isend( sendbuf, count, MPI_REAL, 1, tag, MPI_COMM_WORLD, req,     ierr)
elseif (rank==1) then
     call MPI_Recv(  recvbuf, count, MPI_REAL, 0, tag, MPI_COMM_WORLD, status0, ierr)
endif
...
if(rank==0)  call MPI_Wait(req,status1,ierr)
```

- **Non-blocking Send & Non-blocking Recv**

```
if (rank==0) then
     call MPI_Isend(  sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req0,ierr)
elseif (rank==1) then
     call MPI_Irecv(  recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req1,ierr)
endif
...
if(rank==0)   call MPI_Wait(req0,status0,ierr)
if(rank==1)   call MPI_Wait(req1,status1,ierr)
```

Experiment #3

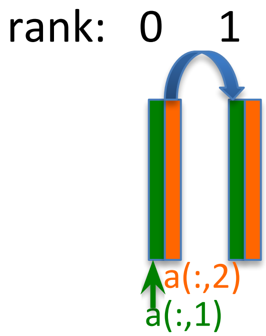
Using your Ping-Pong program from Experiment #2:

Have each task send a message back and forth to all the other tasks, using Non-Blocking

Example, we have 8 tasks acquired,

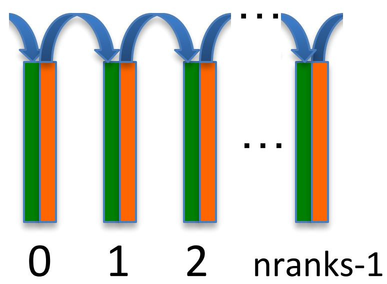
task 1 will send and receive a message to tasks 2, 3, 4, 5, 6, 7, 8
task 2 will send and receive a message to tasks 3, 4, 5, 6, 7, 8, 1
task 3 will send and receive a message to tasks 4, 5, 6, 7, 8, 1, 2
…
…
…
….
(hopefully you see the pattern)

One-way -- Generic

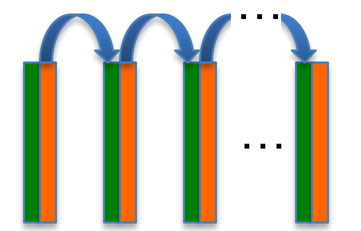



```
if (rank==0) then
     call MPI_Isend( a(1,2),N,MPI_REAL, 1, 9, comm, req,    ierr)
elseif (rank==1) then
     call MPI_Recv(  a(1,1),N,MPI_REAL, 0, 9, comm, status, ierr)
endif
```

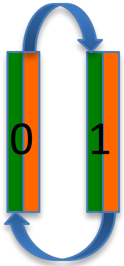
We can make a generic "send right" algorithm out of this:




```
dest=rank +1; src = rank -1
     call MPI_Isend( a(1,2),N,MPI_REAL, dest, 9, comm, req,    ierr)
     call MPI_Recv(  a(1,1),N,MPI_REAL,  src, 9, comm, status, ierr)
```




```
dest=rank +1; src = rank -1
if(dest == nranks) dest=MPI_PROC_NULL
if( src ==     -1)  src=MPI_PROC_NULL
     call MPI_Isend( a(1,2),N,MPI_REAL, dest, 9, comm, req,    ierr)
     call MPI_Recv(  a(1,1),N,MPI_REAL,  src, 9, comm, status, ierr)
```



Two-way Communication: Deadlock

Deadlock 1 (always deadlocks)

```
if (rank==0) then
     call MPI_Recv(    recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ierr)
     call MPI_Send(    sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
elseif (rank==1) then
     call MPI_Recv(    recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
     call MPI_Send(    sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ierr)
endif
```

Deadlock 2 (deadlocks when system buffer is too small)

```
if (rank==0) then
     call MPI_Send(    sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
     call MPI_Recv(    recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ierr)
elseif (rank==1) then
     call MPI_Send(    sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ierr)
     call MPI_Recv(    recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
endif
```

Two-way Communication: Solutions

Solution 1 (single direction) **Sends: 0→1, then 1→0**

```
if (rank==0) then
     call MPI_Send(  sendbuf,count,MPI_REAL, 1,tag,MPI_COMM_WORLD,ierr)
     call MPI_Recv(  recvbuf,count,MPI_REAL, 1,tag,MPI_COMM_WORLD,status,ierr)
elseif (rank==1) then
     call MPI_Recv(  recvbuf,count,MPI_REAL, 0,tag,MPI_COMM_WORLD,status,ierr)
     call MPI_Send(  sendbuf,count,MPI_REAL, 0,tag,MPI_COMM_WORLD,ierr)
endif
```

Solution 2 (allows bidirectional communication) **Sends: 0→1, while 1→0**

```
if (rank==0) then
     call MPI_SendRecv(sendbuf,sendcount,sendtype,dest,  sendtag,
                       recvbuf,recvcount,recvtype,source,recvtag,
                                                         MPI_COMM_WORLD,status,ierr)
elseif (rank==1) then
     call MPI_SendRecv(sendbuf,sendcount,sendtype,dest,  sendtag,
                       recvbuf,recvcount,recvtype,source,recvtag,
                                                         MPI_COMM_WORLD,status,ierr)
endif
```

Two-way Communication: Solutions

Solution 3

```
if (rank==0) then
     call MPI_ISend(    sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req1,ierr)
     call MPI_IRecv(    recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req2,ierr)
elseif (rank==1) then
     call MPI_ISend(    sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req1,ierr)
     call MPI_IRecv(    recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req2,ierr)
endif
         call MPI_Wait(    req1,status,ierr)
         call MPI_Wait(    req2,status,ierr)
```

Solution 4

```
if (rank==0) then
     call MPI_BSend(    sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
     call MPI_Recv(     recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
elseif (rank==1) then
     call MPI_BSend(    sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
     call MPI_Recv(     recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
endif
```

Two-way Communications Summary

	CPU 1	CPU 2
Deadlock 1	Recv/Send	Recv/Send
Deadlock 2	Send/Recv	Send/Recv
Solution 1	Send/Recv	Recv/Send
Solution 2	SendRecv	SendRecv
Solution 3	Isend/Irecv/Wait	Isend/Irecv/Wait
Solution 4	Bsend/Recv	Bsend/Recv

MPI_Probe

- **`MPI_Probe`**

  allows incoming messages to be checked without actually receiving them

  - the user can then decide how to receive the data
  - Used when different actions need to be taken, depending on the "who, what, and how much" information of the message.

MPI_Probe

- C

```
ierr=MPI_Probe(source, tag, comm, &status);
```

- Fortran

```
        MPI_Probe(source, tag, comm,  status,
  ierr)
```

- Parameters
  - source: source rank or **MPI_ANY_SOURCE**
  - tag: tag value or **MPI_ANY_TAG**
  - comm: communicator
  - status: status object

Status doesn't have Length of Message

call MPI_Get_count(status, datatype, **count**, ierr)

ierr=MPI_Get_count(&status, datatype, &**count**)