PCSE Lecture 8

MPI Intro to Point-to-Point

Cyrus Proctor
Victor Eijkhout

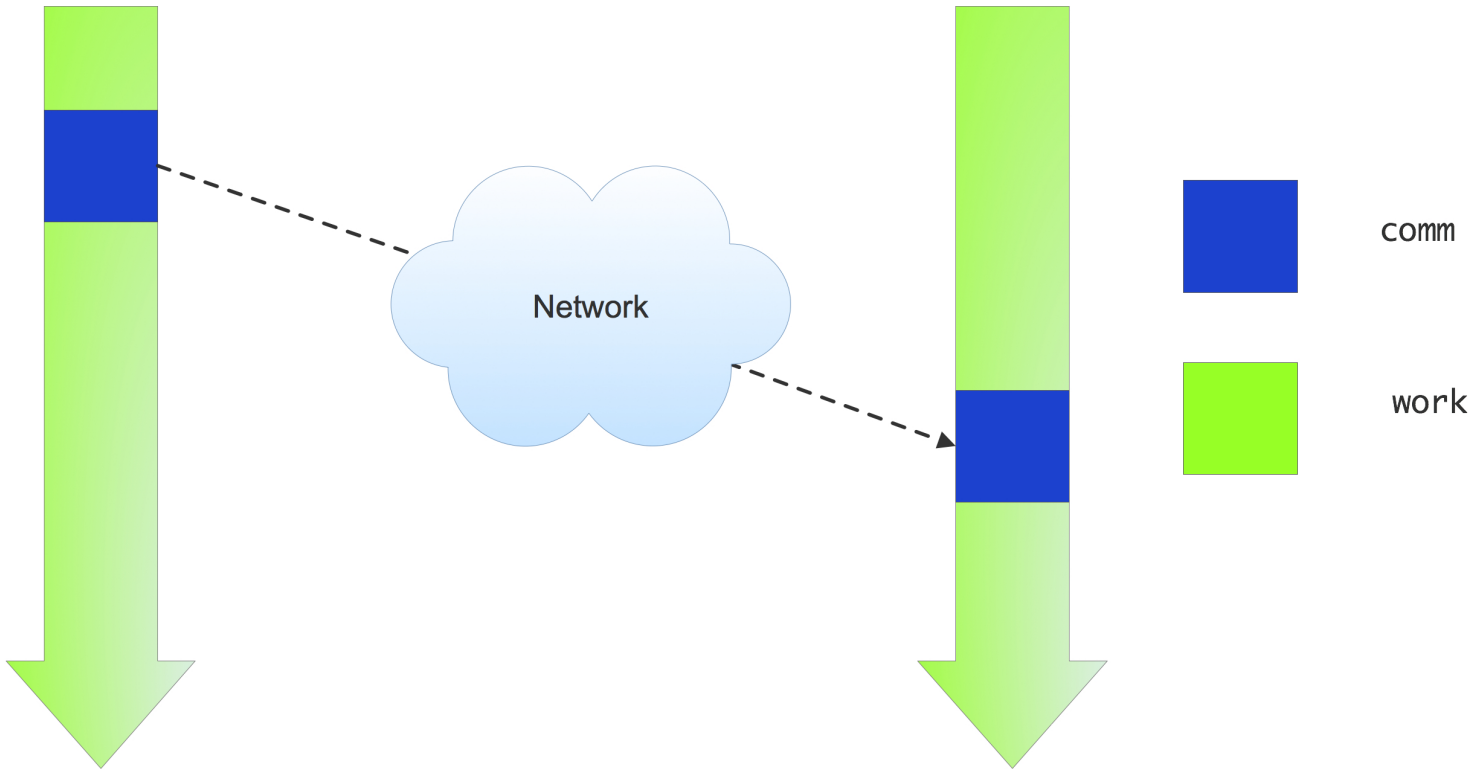
March 24, 2015

Message Passing Paradigm

- A Parallel MPI Program is launched as separate processes (tasks), each with their own address space
  - Requires partitioning data across tasks
- Data is explicitly moved from task to task
  - A task accesses the data of another task through a transaction called "message passing" in which a copy of the data (message) is transferred (passed) from one task to another
- There are two classes of message passing (transfers)
  - Point-to-Point messages – involving only two tasks at a time
  - Collective messages
    - 1-to-many
    - many-to-1
    - many-to-many
- Access to subsets of complex data structures is simplified
- Transfers use synchronous or asynchronous protocols
- Messaging can be arranged into efficient topologies

In an Ideal World...

Processors would just send and receive, and the network would take care of the rest



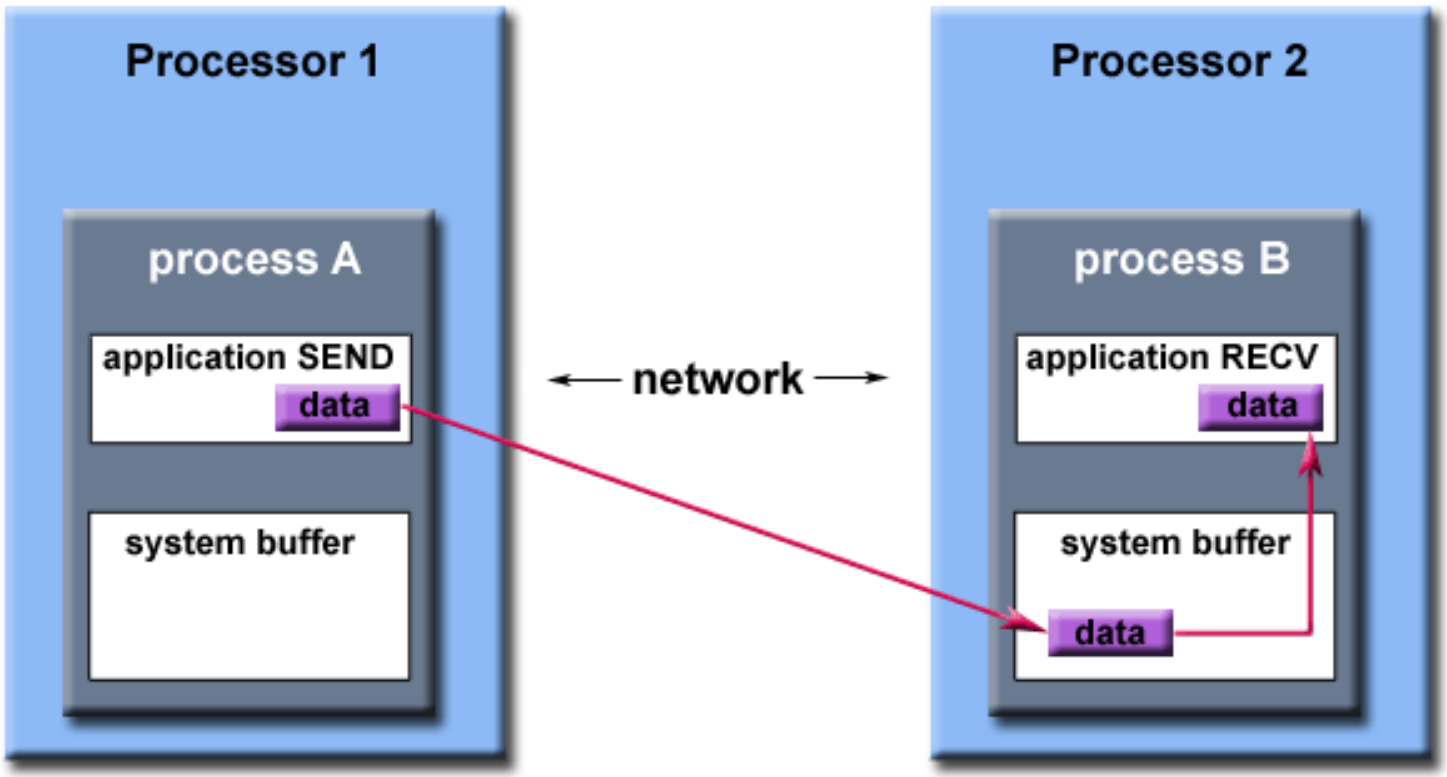
In Reality...

Somehow or other, the MPI implementation must be able to deal with storing data when the two tasks are out of sync
Consider the following two cases:

- A send operation occurs 5 seconds before the receive is ready
  - Where is the message while the receive is pending?

- Multiple sends arrive at the same receiving task which can only accept one send at a time
  - What happens to the messages that are "backing up"?

The MPI implementation (not the MPI standard) decides what happens to data in these types of cases. Typically, a system buffer area is reserved to hold data in transit. For example:



Path of a message buffered at the receiving process

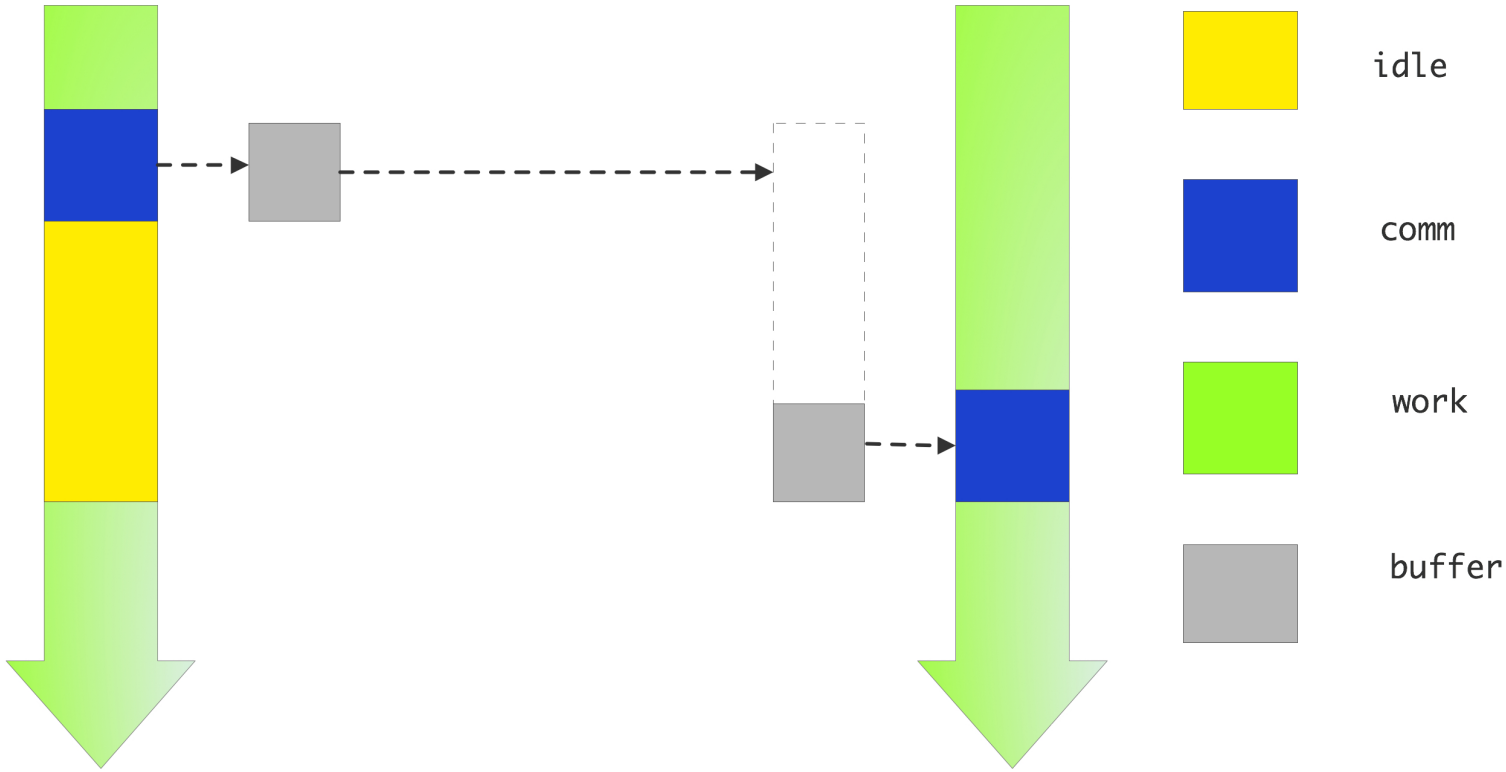
System Buffer Space is:

- Opaque to the programmer and managed entirely by the MPI library
- A finite resource that can be easy to exhaust
- Often mysterious and not well documented
- Able to exist on the sending side, the receiving side, or both
- Something that may improve program performance because it allows send/receive operations to be asynchronous

User managed address space (i.e. your program variables) is called the application buffer. MPI also provides for a user managed send buffer.

Blocking Routines

Data has to be somewhere: on one process or the other

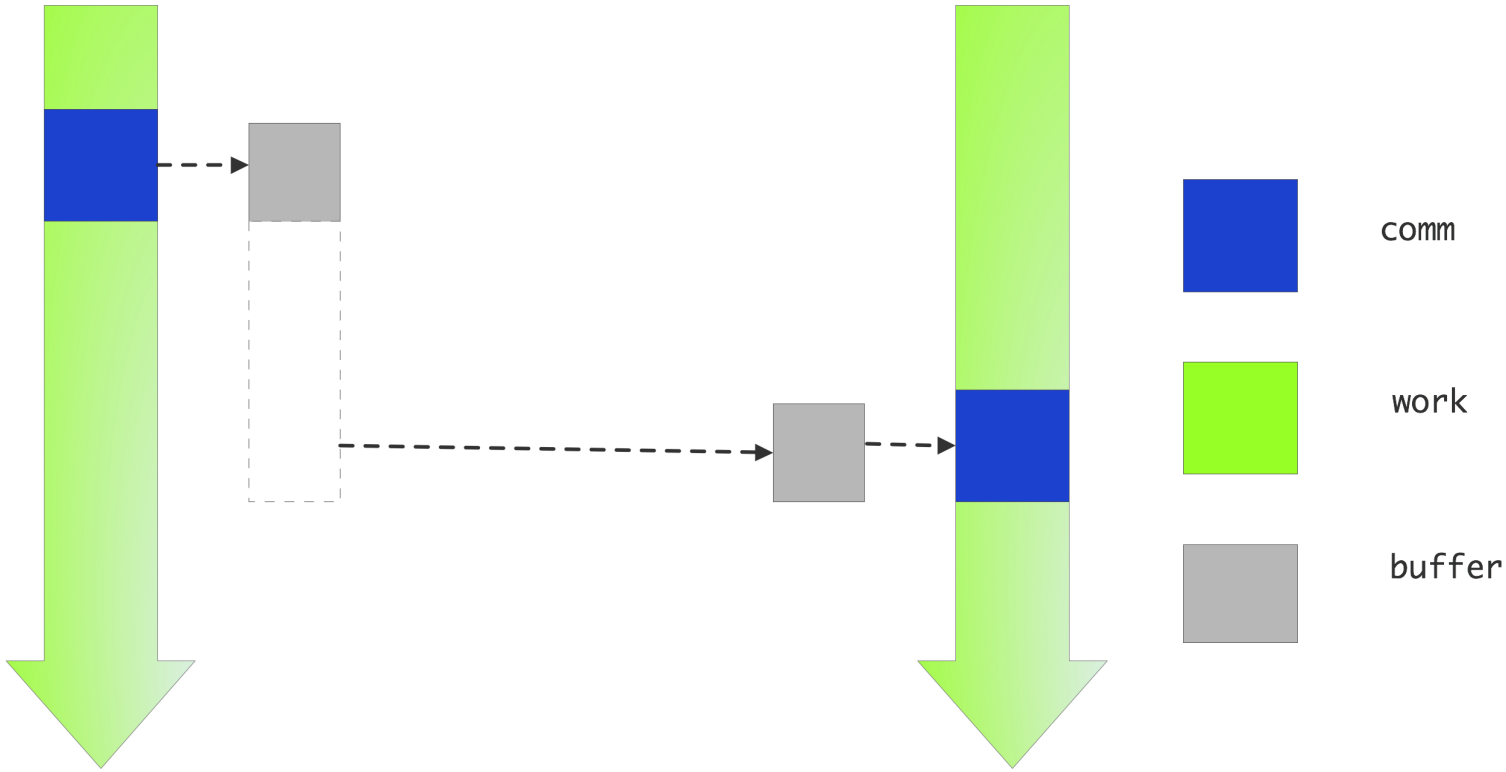


Blocking Routines

- A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received – it may very well be sitting in a system buffer
- A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send
- A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive
- A blocking receive only "returns" after the data has arrived and is ready for use by the program

Non-Blocking Routines

Create a buffer and let the send data sit there until someone picks it up



Non-Blocking Routines

- Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message
- Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
- It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this
- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains

Order and Fairness

Order:

- MPI guarantees that messages will not overtake each other
- If a sender sends two messages (msg 1 and msg 2) in succession to the same destination, and both match the same receive, the receive operation will receive msg 1 before msg 2
- If a receiver posts two receives (rcv 1 and rcv 2), in succession, and both are looking for the same message, rcv 1 will receive the message before rcv 2
- Order rules do not apply if there are multiple threads participating in the communication operations

Fairness:

- MPI does not guarantee fairness – it's up to the programmer to prevent "operation starvation"
- Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete

P-2-P Blocking Arguments

MPI point-to-point communication routines generally have an argument list that takes one of the following formats:

Blocking Sends	MPI_Send(buffer, count, type, dest, tag, comm*)
Blocking Receives	MPI_Recv(buffer, count, type, source, tag, comm, status*)

- When MPI sends a message, it doesn't just send the contents; it also sends an envelope** describing the contents:

buffer	initial address of send/receive buffer	(choice)
count	number of items to send	(non-neg int)
type	MPI data type of items to send/receive	(handle)
dest	MPI rank of task receiving the data	(int)
source	MPI rank of task sending the data	(int)
tag	message ID	(int)
comm	MPI communicator	(handle)
status	returns information on the message received	(status)

Red = Data, Blue = Send to/Receive from, Brown = Message ID
*Fortran Folks: ierr is an addintional optional last argument (int)
**Includes source, dest, tag, comm

MPI Data Types

Some C Data Types

- MPI_CHAR
- MPI_INT
- MPI_LONG
- MPI_FLOAT
- MPI_DOUBLE
- MPI_PACKED
- User Defined

Some Fortran Data Types

- MPI_CHARACTER
- MPI_INTEGER
- MPI_REAL
- MPI_DOUBLE_PRECISION
- MPI_PACKED
- User Defined

Recall Last Week...

Fortran MPI Basics

```
program simple
   use mpi
   implicit none
   integer numtasks, rank, len, ierr, errorcode
   character(MPI_MAX_PROCESSOR_NAME) hostname
   double precision t1,t2,my_sleep_time

   call MPI_INIT(ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
   call MPI_GET_PROCESSOR_NAME(hostname, len, ierr)
   write(*,*)"Number of tasks=",numtasks," My rank=",rank, &
   &         " Running on=",trim(hostname)

  t1 = MPI_WTIME()
  call sleep(2*(rank+1))
  t2 = MPI_WTIME()
  my_sleep_time = t2 - t1
  write(*,*)"My rank=",rank," Running on ",trim(hostname), &
  &         ": Sleep",my_sleep_time

  call MPI_FINALIZE(ierr)
end program simple
```

Recall Last Week...

C MPI Basics

```
#include "mpi.h"
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
  int  numtasks, rank, len, ierr;
  double t1,t2,my_sleep_time;
  char hostname[MPI_MAX_PROCESSOR_NAME];

  ierr = MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Get_processor_name(hostname, &len);
  printf("Number of tasks= %d ", numtasks);
  printf("My rank= %d Running on %s\n",rank,hostname);

  t1 = MPI_Wtime();
  sleep(2*(rank+1));
  t2 = MPI_Wtime();
  my_sleep_time = t2 - t1;
  printf ("My rank= %d Running on %s: Sleep %f\n",
          rank,hostname,my_sleep_time);

  MPI_Finalize();
}
```

Expected Output with Two MPI Tasks

Fortan Output

```
Number of tasks=          2  My rank=          0  Running on=spren
Number of tasks=          2  My rank=          1  Running on=spren
My rank=            0  Running on spren: Sleep    2.0001342296600342
My rank=            1  Running on spren: Sleep    4.0000786781311035
```

C Output

```
Number of tasks= 2 My rank= 0 Running on spren
Number of tasks= 2 My rank= 1 Running on spren
My rank= 0 Running on spren: Sleep 2.000134
My rank= 1 Running on spren: Sleep 4.000064
```

For Fun...

- Still assuming only two MPI tasks, pass to your MPI task neighbor the amount of time you slept and print to stdout using MPI_Send and MPI_Recv

Fortran MPI Partial Send/Recv

```
  ...
  integer dest,src,tag1
  integer stat(MPI_STATUS_SIZE)
  double precision their_sleep_time

  call MPI_INIT(ierr)
  ...
  my_sleep_time = t2 - t1
  write(*,*)"My rank=",rank," Running on ",trim(hostname), &
  &         ":    My Sleep",my_sleep_time
  tag1 = 42
  if (rank == 0) then
    dest = 1
    call MPI_SEND(my_sleep_time,1, MPI_DOUBLE_PRECISION,dest, &
    &             tag1,MPI_COMM_WORLD,ierr)
  else if (rank == 1) then
    src = 0
    call MPI_RECV(their_sleep_time,1,MPI_DOUBLE_PRECISION,src, &
    &             tag1,MPI_COMM_WORLD,stat,ierr)
  end if
  write(*,*)"My rank=",rank," Running on ",trim(hostname), &
  &         ": Their Sleep",their_sleep_time

  call MPI_FINALIZE(ierr)
end program simple
```

C MPI Partial Send/Recv

```
  ...
  int dest,src,tag1;
  MPI_Status Stat;
  double their_sleep_time;

  ierr = MPI_Init(&argc,&argv);
  ...
  my_sleep_time = t2 - t1;
  printf ("My rank= %d Running on %s:    My Sleep %f\n",
          rank,hostname,my_sleep_time);
  tag1 = 42;
  if (rank == 0){
    dest = 1;
    MPI_Send(&my_sleep_time,1,MPI_DOUBLE,dest,
             tag1,MPI_COMM_WORLD);
  } else if (rank == 1) {
    src = 0;
    MPI_Recv(&their_sleep_time,1,MPI_DOUBLE,src,
             tag1,MPI_COMM_WORLD,&Stat);
  }
  printf ("My rank= %d Running on %s: Their Sleep %f\n",
          rank,hostname,their_sleep_time);
  MPI_Finalize();
}
```

Expected Output with Two MPI Tasks

Fortan Partial Send/Recv Output

```
Number of tasks=          2  My rank=          0  Running on=spren
Number of tasks=          2  My rank=          1  Running on=spren
My rank=      0  Running on spren:     My Sleep   2.0001397132873535
My rank=      0  Running on spren: Their Sleep   0.0000000000000000
My rank=      1  Running on spren:     My Sleep   4.0000689029693604
My rank=      1  Running on spren: Their Sleep   2.0001397132873535
```

C Partial Send/Recv Output

```
Number of tasks= 2 My rank= 0 Running on spren
Number of tasks= 2 My rank= 1 Running on spren
My rank= 0 Running on spren:     My Sleep 2.000148
My rank= 0 Running on spren: Their Sleep 0.000000
My rank= 1 Running on spren:     My Sleep 4.000077
My rank= 1 Running on spren: Their Sleep 2.000148
```

Fortran MPI Complete Send/Recv

```
  ...
  integer dest,src,tag1,tag2
  integer stat(MPI_STATUS_SIZE)
  double precision their_sleep_time
  ...
  write(*,*)"My rank=",rank," Running on ",trim(hostname), &
  &         ":    My Sleep",my_sleep_time
  tag1 = 42
  tag2 = 67
  if (rank == 0) then
    dest = 1; src = 1
    call MPI_SEND(my_sleep_time,1, MPI_DOUBLE_PRECISION,dest, &
    &             tag1,MPI_COMM_WORLD,ierr)
    call MPI_RECV(their_sleep_time,1,MPI_DOUBLE_PRECISION,src, &
    &             tag2,MPI_COMM_WORLD,stat,ierr)
  else if (rank == 1) then
    src = 0; dest = 0
    call MPI_RECV(their_sleep_time,1,MPI_DOUBLE_PRECISION,src, &
    &             tag1,MPI_COMM_WORLD,stat,ierr)
    call MPI_SEND(my_sleep_time,1, MPI_DOUBLE_PRECISION,dest, &
    &             tag2,MPI_COMM_WORLD,ierr)
  end if
  write(*,*)"My rank=",rank," Running on ",trim(hostname), &
  &         ": Their Sleep",their_sleep_time
  call MPI_FINALIZE(ierr)
end program simple
```

C MPI Complete Send/Recv

```
  ...
  int dest,src,tag1,tag2;
  MPI_Status Stat;
  double their_sleep_time;
  ...
  printf ("My rank= %d Running on %s:    My Sleep %f\n",
          rank,hostname,my_sleep_time);
  tag1 = 42;
  tag2 = 67;
  if (rank == 0){
    dest = 1; src = 1;
    MPI_Send(&my_sleep_time,1,MPI_DOUBLE,dest,
             tag1,MPI_COMM_WORLD);
    MPI_Recv(&their_sleep_time,1,MPI_DOUBLE,src,
             tag2,MPI_COMM_WORLD,&Stat);
  } else if (rank == 1) {
    src = 0; dest = 0;
    MPI_Recv(&their_sleep_time,1,MPI_DOUBLE,src,
             tag1,MPI_COMM_WORLD,&Stat);
    MPI_Send(&my_sleep_time,1,MPI_DOUBLE,dest,
             tag2,MPI_COMM_WORLD);
  }
  printf ("My rank= %d Running on %s: Their Sleep %f\n",
          rank,hostname,their_sleep_time);
  MPI_Finalize();
}
```

Expected Output with Two MPI Tasks

Fortan Complete Send/Recv Output

```
Number of tasks=          2  My rank=          0  Running on=spren
Number of tasks=          2  My rank=          1  Running on=spren
My rank=      0  Running on spren:     My Sleep    2.0001358985900879
My rank=      1  Running on spren:     My Sleep    4.0000832080841064
My rank=      1  Running on spren: Their Sleep    2.0001358985900879
My rank=      0  Running on spren: Their Sleep    4.0000832080841064
```

C Complete Send/Recv Output

```
Number of tasks= 2 My rank= 0 Running on spren
Number of tasks= 2 My rank= 1 Running on spren
My rank= 0 Running on spren:     My Sleep 2.000151
My rank= 1 Running on spren:     My Sleep 4.000080
My rank= 1 Running on spren: Their Sleep 2.000151
My rank= 0 Running on spren: Their Sleep 4.000080
```

For Fun...

- Reverse the order of only Rank 0's Send and Recv. What happens?
- Reverse the order of only Rank 1's Send and Recv. What happens?
- Reverse the order of both Rank's Send and Recv. What happens?

Rank 0 R/S; Rank 1 R/S

Deadlock: The Code Hangs!

```
...
tag1 = 42
tag2 = 67
if (rank == 0) then
  dest = 1; src = 1
  call MPI_RECV(their_sleep_time,1,MPI_DOUBLE_PRECISION,src, &
  &             tag2,MPI_COMM_WORLD,stat,ierr)
  call MPI_SEND(my_sleep_time,1, MPI_DOUBLE_PRECISION,dest, &
  &             tag1,MPI_COMM_WORLD,ierr)
else if (rank == 1) then
  src = 0; dest = 0
  call MPI_RECV(their_sleep_time,1,MPI_DOUBLE_PRECISION,src, &
  &             tag1,MPI_COMM_WORLD,stat,ierr)
  call MPI_SEND(my_sleep_time,1, MPI_DOUBLE_PRECISION,dest, &
  &             tag2,MPI_COMM_WORLD,ierr)
end if
...
```

- Both Rank 0 and Rank 1 are waiting for the other one to send a message!

Rank 0 S/R; Rank 1 S/R

It Works!?* Why?

```
...
tag1 = 42
tag2 = 67
if (rank == 0) then
  dest = 1; src = 1
  call MPI_SEND(my_sleep_time,1, MPI_DOUBLE_PRECISION,dest, &
  &             tag1,MPI_COMM_WORLD,ierr)
  call MPI_RECV(their_sleep_time,1,MPI_DOUBLE_PRECISION,src, &
  &             tag2,MPI_COMM_WORLD,stat,ierr)
else if (rank == 1) then
  src = 0; dest = 0
  call MPI_SEND(my_sleep_time,1, MPI_DOUBLE_PRECISION,dest, &
  &             tag2,MPI_COMM_WORLD,ierr)
  call MPI_RECV(their_sleep_time,1,MPI_DOUBLE_PRECISION,src, &
  &             tag1,MPI_COMM_WORLD,stat,ierr)
end if
...
```

- Both Rank 0 and Rank 1 first send a message
- The Send operations complete when it is safe to modify buffer – not when when received on the other end!
- This will work as long as the system buffer is not full

*Can depend on MPI implementation, system buffer size and fullness, message length

Rank 0 R/S; Rank 1 S/R

It Works!

```
...
integer dest,src,tag1,tag2
integer stat(MPI_STATUS_SIZE)
double precision their_sleep_time
...
tag1 = 42
tag2 = 67
if (rank == 0) then
  dest = 1; src = 1
  call MPI_RECV(their_sleep_time,1,MPI_DOUBLE_PRECISION,src, &
  &             tag2,MPI_COMM_WORLD,stat,ierr)
  call MPI_SEND(my_sleep_time,1, MPI_DOUBLE_PRECISION,dest, &
  &             tag1,MPI_COMM_WORLD,ierr)
else if (rank == 1) then
  src = 0; dest = 0
  call MPI_SEND(my_sleep_time,1, MPI_DOUBLE_PRECISION,dest, &
  &             tag2,MPI_COMM_WORLD,ierr)
  call MPI_RECV(their_sleep_time,1,MPI_DOUBLE_PRECISION,src, &
  &             tag1,MPI_COMM_WORLD,stat,ierr)
end if
write(*,*)"My rank=",rank," Running on ",trim(hostname), &
&         ": Their Sleep",their_sleep_time
...
```

- Both Rank 0 is waiting for a message while Rank 1 is sending

Blocking Send/Recv Summary

Outcome	Rank 0	Rank 1
Okay	Send/Recv	Recv/Send
Deadlock	Recv/Send	Recv/Send
Possible Deadlock*	Send/Recv	Send/Recv
Okay	Recv/Send	Send/Recv

*Can create non-portable / non-reproducible code

P-2-P Non-Blocking Arguments

MPI point-to-point communication routines generally have an argument list that takes one of the following formats:

Non-blocking Sends	MPI_Isend(buffer, count, type, dest, tag, comm, request*)
Non-blocking Receives	MPI_Irecv(buffer, count, type, source, tag, comm, request*)

- When MPI sends a message, it doesn't just send the contents; it also sends an envelope** describing the contents:

buffer	initial address of send/receive buffer
count	number of items to send
type	MPI data type of items to send/receive
dest	MPI rank of task receiving the data
source	MPI rank of task sending the data
tag	message ID
comm	MPI communicator
request	returns a handle to determine completion of routine

Red = Data, Blue = Send to/Receive from, Brown = Message ID
*Fortran Folks: ierr is an addintional optional last argument (int)
**Includes source, dest, tag, comm

For More Fun...

- Still assuming only two MPI tasks, pass to your MPI task neighbor the amount of time you slept and print to stdout using MPI_Isend and MPI_Irecv

Fortan Complete Isend/Irecv Output

```
...
integer dest,src,tag1,tag2
integer stats(MPI_STATUS_SIZE,4), reqs0(2), reqs1(2)
double precision their_sleep_time
...
tag1 = 42
tag2 = 67
if (rank == 0) then
  src = 1; dest = 1
  call MPI_ISEND(my_sleep_time,1,MPI_DOUBLE_PRECISION,dest,tag1, &
  &              MPI_COMM_WORLD,reqs0(1),ierr)
  call MPI_IRECV(their_sleep_time,1,MPI_DOUBLE_PRECISION,src,tag2, &
  &              MPI_COMM_WORLD,reqs0(2),ierr)
  call MPI_WAITALL(2,reqs0,stats,ierr)
else if (rank == 1) then
  call MPI_IRECV(their_sleep_time,1,MPI_DOUBLE_PRECISION,src,tag1, &
  &              MPI_COMM_WORLD,reqs1(1),ierr)
  call MPI_ISEND(my_sleep_time,1,MPI_DOUBLE_PRECISION,dest,tag2, &
  &              MPI_COMM_WORLD,reqs1(2),ierr)
  call MPI_WAITALL(2,reqs1,stats,ierr)
end if
...
```

Another Alternative: MPI_Sendrecv

- Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.

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

C MPI Complete Send/Recv

```
...
int dest,src,tag1,tag2;
MPI_Status Stat;
double their_sleep_time;
...
tag1 = 42;
tag2 = 67;
if (rank == 0){
  dest = 1; src = 1;
  MPI_Sendrecv (&my_sleep_time,1,MPI_DOUBLE,dest,tag1,
                &their_sleep_time,1,MPI_DOUBLE,src,tag2,
                MPI_COMM_WORLD,&Stat);
} else if (rank == 1) {
  src = 0; dest = 0;
  MPI_Sendrecv (&my_sleep_time,1,MPI_DOUBLE,dest,tag2,
                &their_sleep_time,1,MPI_DOUBLE,src,tag1,
                MPI_COMM_WORLD,&Stat);
}
...
```

References

- Victor Eijkhout, "Introduction to High Performance Scientific Computing"
- Victor Eijkhout, "Parallel Computing for Science and Engineering"
- Blaise Barney, "MPI Tutorial"
- Mark Lubin, "Introduction into new features of MPI-3.0 Standard"
- "MPI: A Message-Passing Interface Standard Version-3.0"