CSCI 5451: Introduction to Parallel Computing

Lecture 10: Introduction to MPI

computer science
& engineering

University of Minnesota
Driven to Discover®

Announcements (10/06)

- HW1 → Use more recent Program + Unit Tests (Updated on Thursday Evening)

Lecture Overview

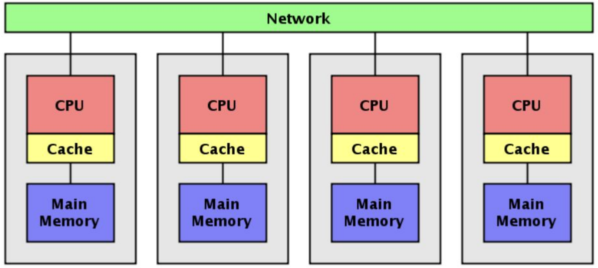
- Basics of Message Passing
    - Overview
    - Buffering vs. Non-Buffering & Blocking vs. Non-Blocking
- MPI
    - Primitive Commands
    - Sending & Receiving

Lecture Overview

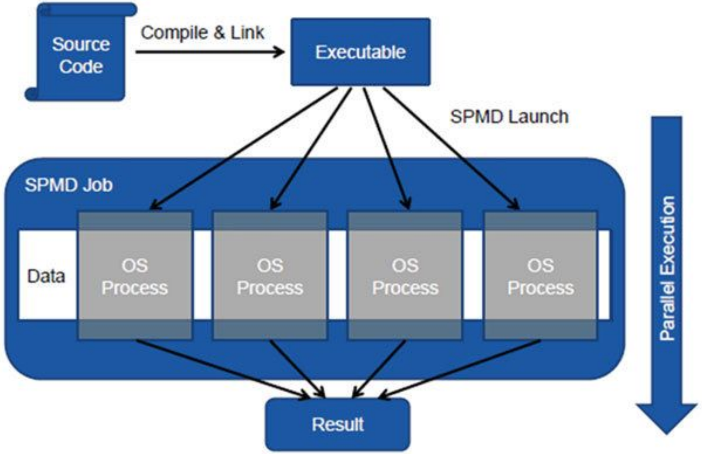
- **Basics of Message Passing**
  - **Overview**
  - Buffering vs. Non-Buffering & Blocking vs. Non-Blocking
- MPI
  - Primitive Commands
  - Sending & Receiving

Basics of Message Passing

- Message Passing programming assumes a distributed address space (sometimes also called a partitioned address space)
- Assumes
  - Data must belong to at least one of the partitions
  - All interactions (communications) require two processes to participate
- Most Message Passing Programs are Single Program Multiple Data (SPMD)



Source: Kaminsky/Parallel Java



Basic Communications

We can decompose complex interactions in Message Passing into two categories

- Send
- Receive

Basic Communications

We can decompose complex interactions in Message Passing into two categories

- Send
- Receive

If we designed an interface for each command, what information would be necessary?

Basic Communications

We can decompose complex interactions in Message Passing into two categories

- ❑ Send
- ❑ Receive

```
send(void *sendbuff, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

Basic Communications

We can decompose complex interactions in Message Passing into two categories

- Send
- Receive

The sending process must fill a buffer with the message it wants to send and the receiving process must prepare a buffer for this message to be written to.

```
send(void *sendbuff, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

Basic Communications

We can decompose complex interactions in Message Passing into two categories

- ❑ Send
- ❑ Receive

Both processes must specify the size of the message. Critically, when two processes communicate, this value must be the same in both the sending and receiving process.

```
send(void *sendbuff, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

Basic Communications

We can decompose complex interactions in Message Passing into two categories

- ❑ Send
- ❑ Receive

The sending process must determine which process to send the message to (*int dest*) and the receiving process must decide where the message comes from (*int source*).

```
send(void *sendbuff, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

Simple Communication Example

```
P0

a = 100;
send(&a, 1, 1);
a=0;
```

```
P1

receive(&a, 1, 0)
printf("%d\n", a);
```

Simple Communication Example

```
P0

a = 100;
send(&a, 1, 1);
a=0;
```

```
P1

receive(&a, 1, 0)
printf("%d\n", a);
```

Both programs are running at the same time.

Simple Communication Example

```
P0

a = 100;
send(&a, 1, 1);
a=0;
```

```
P1

receive(&a, 1, 0)
printf("%d\n", a);
```

Size of message must be the same.

Simple Communication Example

```
P0

a = 100;
send(&a, 1, 1);
a=0;
```

P0 sends to P1.

```
P1

receive(&a, 1, 0)
printf("%d\n", a);
```

P1 receives from P0.

Simple Communication Example

```
P0

a = 100;
send(&a, 1, 1);
a=0;
```

```
P1

receive(&a, 1, 0)
printf("%d\n", a);
```

What prints?

Simple Communication Example

```
P0

a = 100;
send(&a, 1, 1);
a=0;
```

```
P1

receive(&a, 1, 0)
printf("%d\n", a);
```

What prints?

Depends on implementation!! (more on this in next slides)

Lecture Overview

- **Basics of Message Passing**
    - Overview
    - **Buffering vs. Non-Buffering & Blocking vs. Non-Blocking**
- MPI
    - Primitive Commands
    - Sending & Receiving

Blocking Non-Buffering Message Passing

One way to guarantee that ‘100’ is sent from the example on the previous slide is to halt any further execution on P0 until the send operation completes

```
P0

a = 100;
send(&a, 1, 1);
a=0;
```

```
P1

receive(&a, 1, 0)
printf("%d\n", a);
```

Blocking Non-Buffering Message Passing

Pause here until P1 acknowledges receive

```
P0

a = 100;
send(&a, 1, 1);
a=0;
```

This type of pattern is called Blocking Non-Buffering

- Blocking → We *block* further execution on P0 until send completes
- Non-Buffering → We make no use of intermediate buffers and instead wait until CPU acknowledgement from receiver (we will see the Buffering example to contrast this in the next slides)

```
P1

receive(&a, 1, 0)
printf("%d\n", a);
```

Blocking Non-Buffering Problems

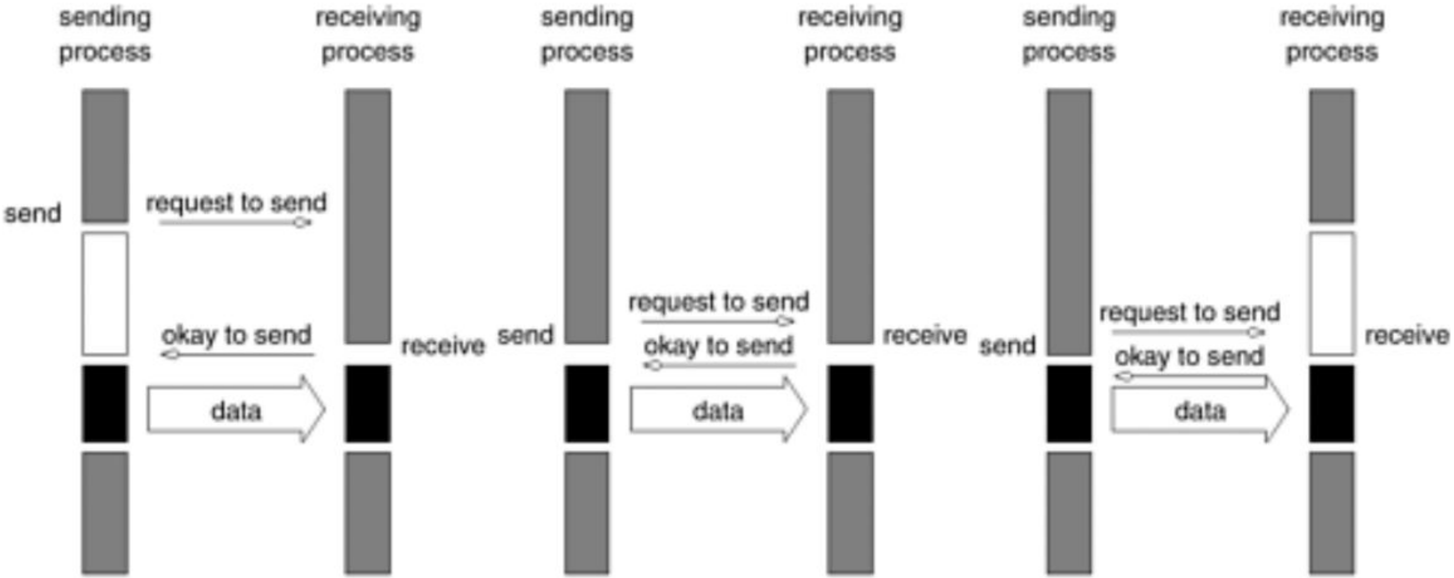
In general, there are two issues with Blocking Non-Buffering

- Idling
- Deadlocks

Blocking Non-Buffering Problems

In general, there are two issues with Blocking Non-Buffering

- ❑ Idling
- ❑ Deadlocks



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

Blocking Non-Buffering Problems

In general, there are two issues with Blocking Non-Buffering

- ❑ Idling
- ❑ Deadlocks

Problem?

```
P0

send(&a, 1, 1);
receive(&b, 1, 1);
```

```
P1

send(&a, 1, 0);
receive(&b, 1, 0);
```

Blocking Non-Buffering Problems

In general, there are two issues with Blocking Non-Buffering

- Idling
- Deadlocks

Both block forever on the *send* operation, waiting for the other to call *receive*.

```
P0

send(&a, 1, 1);
receive(&b, 1, 1);
```

```
P1

send(&a, 1, 0);
receive(&b, 1, 0);
```

Blocking Buffering Message Passing

- A solution to the idling/deadlocking problems is to use *Buffering*
- There are two ways to achieve this kind of operation
  - Asynchronous Communication
  - Synchronous Communication

Blocking Buffering Message Passing

- A solution to the idling/deadlocking problems is to use *Buffering*
- There are two ways to achieve this kind of operation
  - Asynchronous Communication
  - Synchronous Communication

- Asynchronous communication requires a form of hardware support called Direct Memory Access (DMA)
- This means that communications can occur ***independent of the CPU***, allowing program execution to continue
- If there is no hardware support, then this option is not possible

Blocking Buffering Message Passing

- A solution to the idling/deadlocking problems is to use *Buffering*
- There are two ways to achieve this kind of operation
  - Asynchronous Communication
  - $\boxed{\text{Synchronous Communication}}$

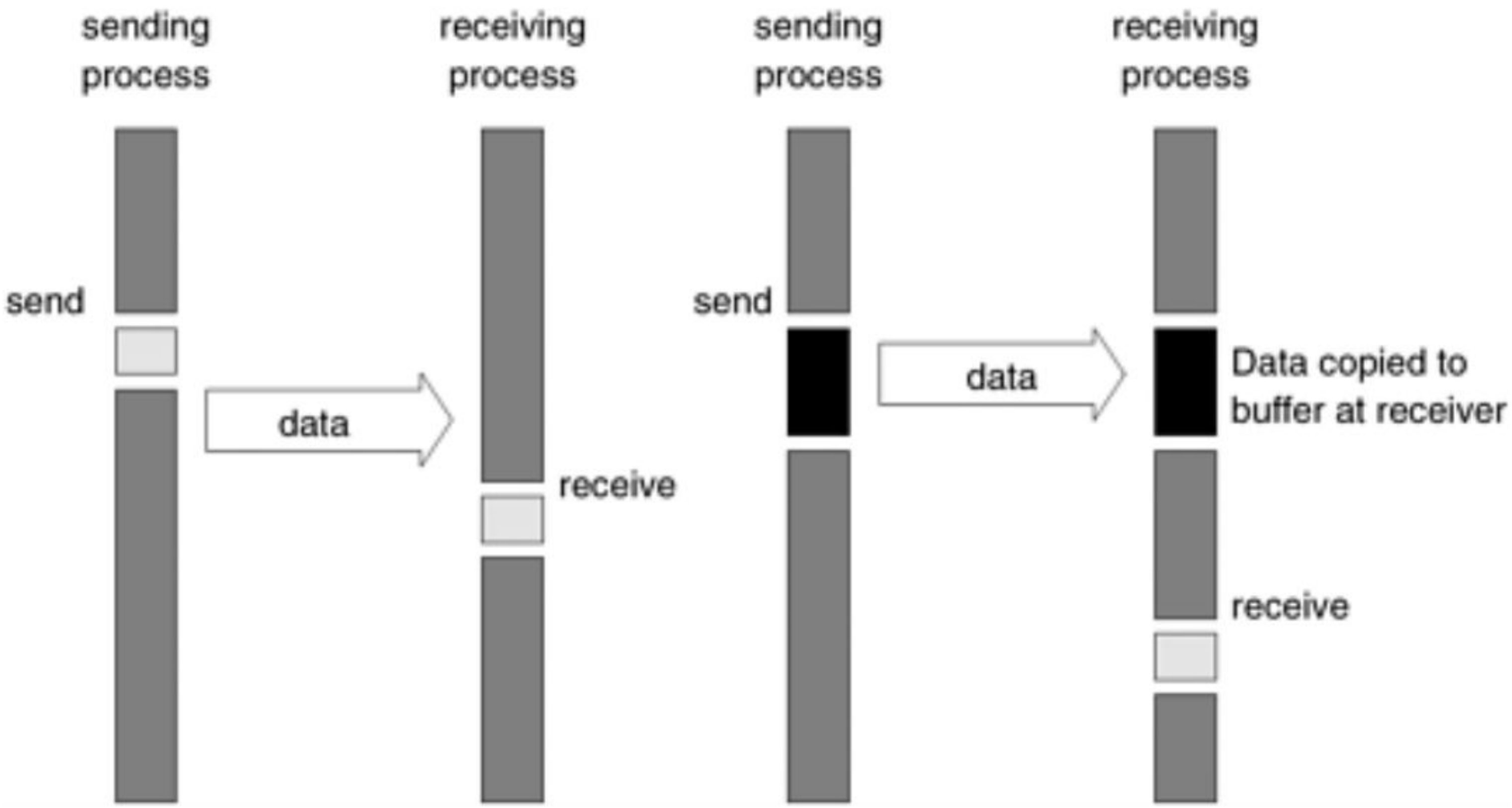
- If there is no hardware support, then you can also use synchronous communication
- The communication still requires CPU support, but the programmer provides a buffer to temporarily copy whatever needs to be sent
- In this case, the CPU cannot continue executing during the communication on either process
- The buffer can be either on the sending or receiving process

Blocking Buffering Message Passing

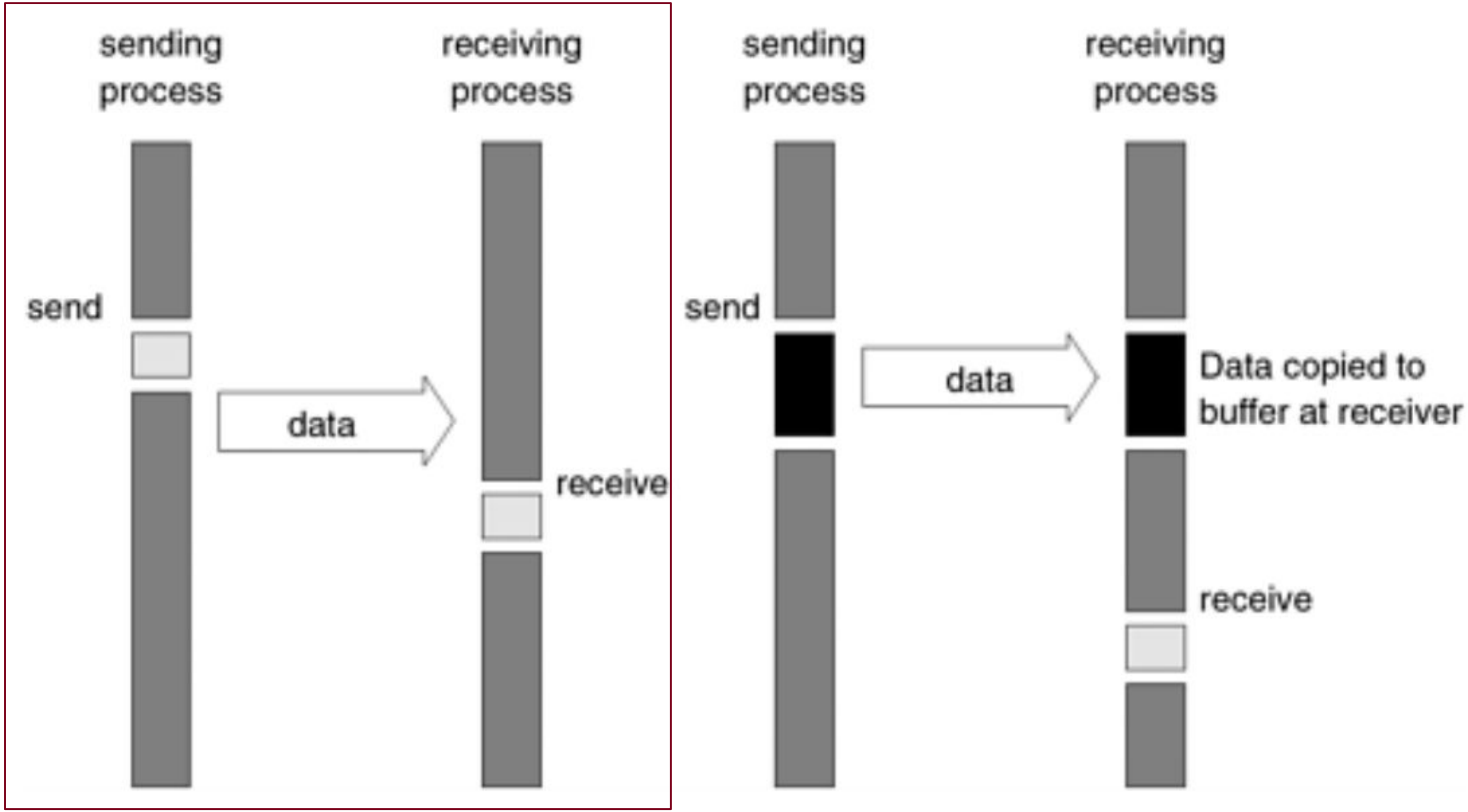
- A solution to the idling/deadlocking problems is to use *Buffering*
- There are two ways to achieve this kind of operation
  - Asynchronous Communication
  - $\boxed{\text{Synchronous Communication}}$

If too many messages are sent at once, this buffer space may overhead. If using this communication, be sure not to overuse the temporary buffer constraints

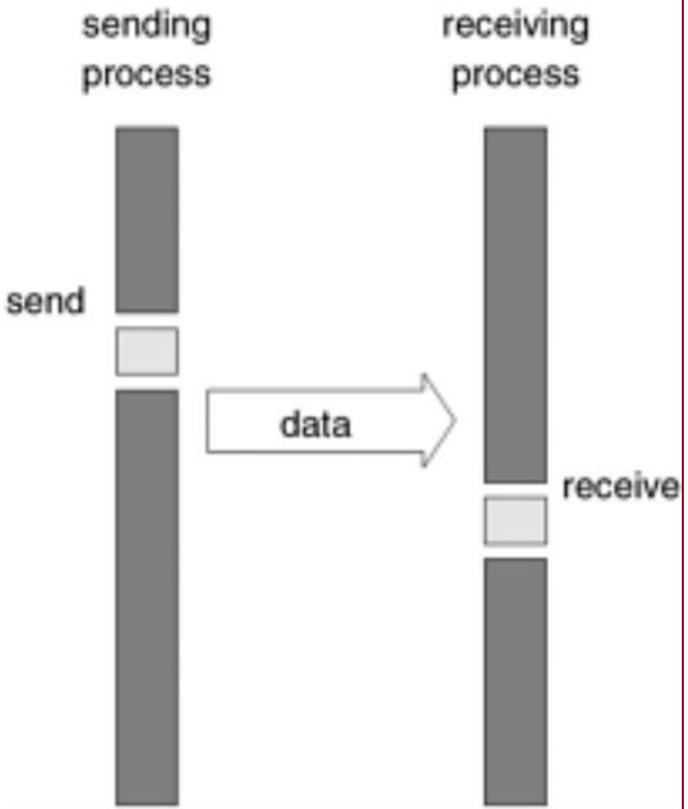
Blocking Buffering Message Passing

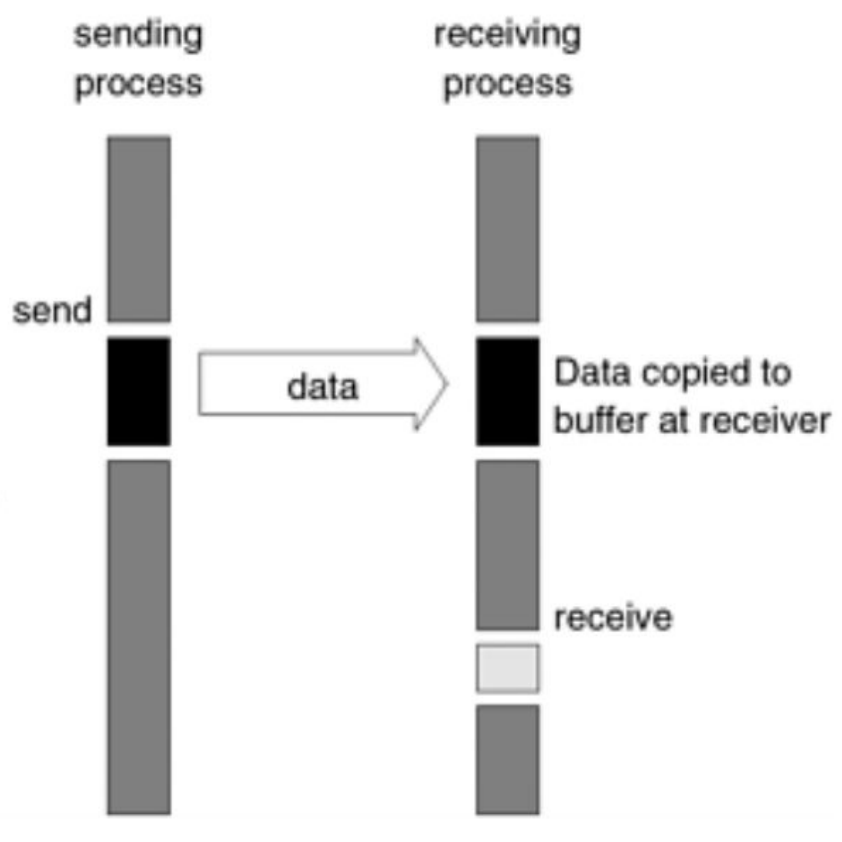


Asynchronous Communication



Synchronous Communication





Deadlocks in Blocking Buffering

- In the buffering case, the ***receive*** calls will still be blocking
- This means we can still get deadlocks if we are not careful when constructing our programs

```
P0

receive(&a, 1, 1);
send(&b, 1, 1);
```

```
P0

receive(&a, 1, 1);
send(&b, 1, 1);
```

Deadlocks in Blocking Buffering

- In the buffering case, the ***receive*** calls will still be blocking
- This means we can still get deadlocks if we are not careful when constructing our programs

```
P0

receive(&a, 1, 1);
send(&b, 1, 1);
```

```
P0

receive(&a, 1, 1);
send(&b, 1, 1);
```

Both block here

Blocking Comparison

Which communication pattern is preferable in which circumstances (buffering vs. non-buffering)?

Blocking Comparison

Which communication pattern is preferable in which circumstances (buffering vs. non-buffering)?

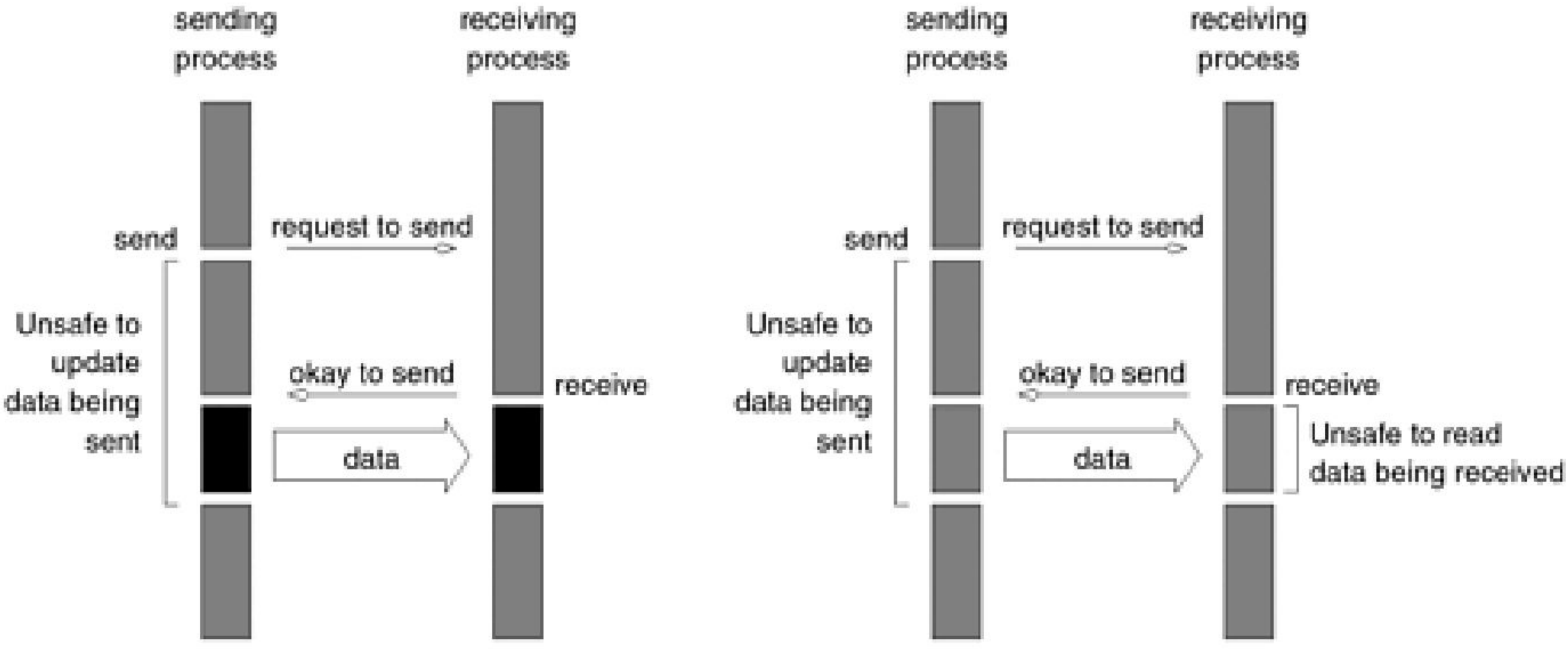
- Non-buffering when additional time spent buffering would take longer than additional time spent idling
- Buffering when idling times will be worse than buffering times.
- Usually - buffering is better for smaller messages & non-buffering is better for larger messages

Non-blocking Message Passing

- Non-blocking operations mean that program execution ***does not pause*** on either send or receive calls
- This means that we cannot guarantee correct program execution without additional logic
- However, this can be helpful for overlapping communication with computation if we structure our programs properly
- We will explore this in later lectures

	Blocking Operations	Non–Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non–Buffered	Sending process blocks until matching receive operation has been encountered	
	Send and Receive semantics assured by corresponding operation	Programmer must explicitly ensure semantics by polling to verify completion

Non-blocking Message Passing



(a) Without hardware support

(b) With hardware support

Lecture Overview

- Basics of Message Passing
  - Overview
  - Buffering vs. Non-Buffering & Blocking vs. Non-Blocking
- **MPI**
  - **Primitive Commands**
  - Sending & Receiving

MPI Primitives

- The ***send*** and ***receive*** of previous slides provides the basis for MPI
- Additional primitives are given in the table at right
- One can write almost all remaining MPI communications we cover using only these primitives

MPI_Init	Initializes MPI
MPI_Finalize	Terminates MPI
MPI_Comm_size	Determines number of communicating processes
MPI_Comm_rank	Determines the label of the communicating process
MPI_Send	Sends a message
MPI_Recv	Receives a message

Starting & Terminating the MPI Library

In order to use MPI we must…

- Initialize MPI
  - Initializes the MPI environment
  - ***Must*** be called before any other MPI routines
  - Can only be called once
  - Dispatches program arguments to all processes
- Finalize MPI
  - Called at the end of computation
  - Cleans up the environment
  - No MPI calls made be made after this point

Starting & Terminating the MPI Library

In order to use MPI we must…

- Initialize MPI
  - Initializes the MPI environment
  - ***Must*** be called before any other MPI routines
  - Can only be called once
  - Dispatches program arguments to all processes
- Finalize MPI
  - Called at the end of computation
  - Cleans up the environment
  - No MPI calls made be made after this point

*int MPI_Init(int *argc, char ***argv)*

MPI requires that the arguments used by the ***main*** function in your c program are passed along to MPI.

Starting & Terminating the MPI Library

In order to use MPI we must...

- Initialize MPI
  - Initializes the MPI environment
  - ***Must*** be called before any other MPI routines
  - Can only be called once
  - Dispatches program arguments to all processes
- Finalize MPI
  - Called at the end of computation
  - Cleans up the environment
  - No MPI calls made be made after this point

*int MPI_Init(int *argc, char ***argv)*

MPI returns ***MPI_SUCCESS*** when ***MPI_Init*** successfully establishes the MPI runtime + environment. It returns an implementation-defined error code otherwise

Starting & Terminating the MPI Library

In order to use MPI we must...

- Initialize MPI
  - Initializes the MPI environment
  - ***Must*** be called before any other MPI routines
  - Can only be called once
  - Dispatches program arguments to all processes
- Finalize MPI
  - Called at the end of computation
  - Cleans up the environment
  - No MPI calls made be made after this point

*int MPI_Init(int *argc, char ***argv)*

- Before entering this function, all processes will already be executing, but will be unable to communicate.
- After this function, they will be able to communicate.

Starting & Terminating the MPI Library

In order to use MPI we must...

- ❑ Initialize MPI
  - Initializes the MPI environment
  - ***Must*** be called before any other MPI routines
  - Can only be called once
  - Dispatches program arguments to all processes
- ❑ Finalize MPI
  - Called at the end of computation
  - Cleans up the environment
  - No MPI calls made be made after this point

*int MPI_Init(int *argc, char ***argv)*

Command line processing should be performed ***after*** you have called this function as MPI will modify the arguments and decrement ***argc*** + remove args from ***argv,*** accordingly

Starting & Terminating the MPI Library

In order to use MPI we must...

- Initialize MPI
  - Initializes the MPI environment
  - ***Must*** be called before any other MPI routines
  - Can only be called once
  - Dispatches program arguments to all processes
- Finalize MPI
  - Called at the end of computation
  - Cleans up the environment
  - No MPI calls made be made after this point

int MPI_Finalize()

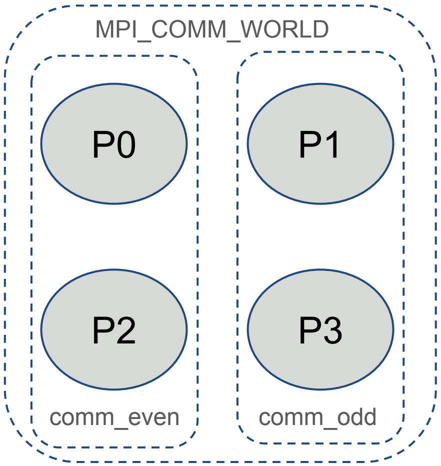
- Call this after using MPI to clean up the runtime + environment
- MPI returns ***MPI_SUCCESS*** if successful & an implementation-defined error code otherwise

Communicators

- ***Communication Domains*** are sets of processes which can communicate with one another
- Information about communication domains are stored in variables of type ***MPI_Comm***
- These variables are called communicators
- Note that processors can belong to more than one communicator
- ***MPI_COMM_WORLD*** is a communicator which includes every process - this is usually the default communicator
- We can define our own communicators to allow for communication operations within specific subsets of processes

Communicators

- ***Communication Domains*** are sets of processes which can communicate with one another
- Information about communication domains are stored in variables of type ***MPI_Comm***
- These variables are called communicators
- Note that processors can belong to more than one communicator
- ***MPI_COMM_WORLD*** is a communicator which includes every process - this is usually the default communicator
- We can define our own communicators to allow for communication operations within specific subsets of processes



- ***MPI_COMM_WORLD*** is default and includes all processes
- We can define ***comm_even*** and ***comm_odd*** separately
- We will see how to both create & use these communicators in later slides

Process + Program Information

Within a communicator we can

- ❑ Determine how many processes are in the communicator
- ❑ Determine identifying information about the calling process (also called the *label* or *rank*)

*int MPI_Comm_size(MPI_Comm comm, int *size)*

Process + Program Information

Within a communicator we can

- ❑ Determine how many processes are in the communicator
- ❑ Determine identifying information about the calling process (also called the *label* or *rank*)

*int MPI_Comm_size(MPI_Comm comm, int *size)*

The communicator we wish to get information for

Process + Program Information

Within a communicator we can

- Determine how many processes are in the communicator
- Determine identifying information about the calling process (also called the *label* or *rank*)

*int MPI_Comm_size(MPI_Comm comm, int *size)*

size will store the number of processes in the ***comm*** communicator after the function has executed

Process + Program Information

Within a communicator we can

- ❑ Determine how many processes are in the communicator
- ❑ Determine identifying information about the calling process (also called the *label* or *rank*)

*int MPI_Comm_rank(MPI_Comm comm, int *rank)*

Process + Program Information

Within a communicator we can

- ❑ Determine how many processes are in the communicator
- ❑ Determine identifying information about the calling process (also called the *label* or *rank*)

*int MPI_Comm_rank(MPI_Comm comm, int *rank)*

The communicator we wish to get information for

Process + Program Information

Within a communicator we can

- ❑ Determine how many processes are in the communicator
- ❑ Determine identifying information about the calling process (also called the *label* or *rank*)

*int MPI_Comm_rank(MPI_Comm comm, int *rank)*

rank will store the label of the calling process in the ***comm*** communicator after the function has executed. The possible values of ***rank*** are [0, ***size*** - 1]

Hello World Example

```
#include <mpi.h>

main(int argc, char *argv[])
{
  int npes, myrank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  printf("From process %d out of %d, Hello World!\n",
          myrank, npes);
  MPI_Finalize();
}
```

Hello World Example

```
#include <mpi.h>

main(int argc, char *argv[])
{
  int npes, myrank;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &npes);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  printf("From process %d out of %d, Hello World!\n",
           myrank, npes);
  MPI_Finalize();
}
```

What will this program print out when run with 4 processes?

Hello World Example

```
#include <mpi.h>

main(int argc, char *argv[])
{
  int npes, myrank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  printf("From process %d out of %d, Hello World!\n",
          myrank, npes);
  MPI_Finalize();
}
```

From process 0 out of 4, Hello World!
From process 1 out of 4, Hello World!
From process 2 out of 4, Hello World!
From process 3 out of 4, Hello World!

Lecture Overview

- Basics of Message Passing
  - Overview
  - Buffering vs. Non-Buffering & Blocking vs. Non-Blocking
- **MPI**
  - Primitive Commands
  - **Sending & Receiving**

Sending + Receiving

*int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

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

Sending + Receiving

Return value from each call

- Returns ***MPI_SUCCESS*** if successful
- Returns an implementation-defined error code otherwise

*int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

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

Sending + Receiving

Buffer storing the message for sending or receiving

- The buffer can be variable length
- Should be the same size at both the sending and receiving process
- Be careful to set the ***count*** and ***datatype*** arguments to be the size of the buffer and the type of each element in the buffer, respectively

*int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

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

Sending + Receiving

The number of elements you wish to send inside of the buffer ***buf***

*int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

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

Sending + Receiving

The datatype of each entry in the the buffer ***buf***

- (See following slides for table of all possible data types)

*int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

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

Datatypes

Use these when you have arrays of the given C datatypes

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	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Datatypes

Use ***MPI_BYTE*** when you want to send raw data. For the previous datatypes, MPI may attempt conversions - ***MPI_BYTE*** will not do so

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	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Datatypes

Useful for bundling multiple non-contiguous messages at once for sending

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	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Datatypes

You can also create MPI structs to send more general data structures

```
struct Particle {
    double x, y, z;
    int id;
};

MPI_Datatype MPI_PARTICLE;

void create_particle_type() {
    int lengths[2] = {3, 1};
    const MPI_Aint disps[2] = {offsetof(struct Particle, x),
                               offsetof(struct Particle, id)};
    MPI_Datatype types[2] = {MPI_DOUBLE, MPI_INT};

    MPI_Type_create_struct(2, lengths, disps, types, &MPI_PARTICLE);
    MPI_Type_commit(&MPI_PARTICLE);
}
```

Sending + Receiving

- ***dest*** indicates which process to send to in the ***comm*** communicator
  - Must always specify an existing process in ***comm***
- ***source*** indicates which process to receive from in the ***comm*** communicator
  - Can use ***MPI_ANY_SOURCE*** to indicate that it will accept a message from any process in ***comm***

*int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

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

Sending + Receiving

- Multiple messages can be sent between a given two processes at a time
- ***tag*** can be used to align the messages
- The receiving process can use ***MPI_ANY_TAG*** to denote that it will accept messages

*int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

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

Sending + Receiving

- The communicator the send and receive processes should occur over
- The ***source*** and ***dest*** variables should be chosen such that they both exist in the ***comm*** communicator (or else the program will deadlock)
- Further, the ***dest*** process must have an ***MPI_Recv*** and ***source*** must have an ***MPI_Send***

*int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

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

Sending + Receiving

- Contains helpful information regarding the received data following this struct
- ***MPI_SOURCE*** and ***MPI_TAG*** are useful when ***MPI_Recv*** is called with ***MPI_ANY_SOURCE*** or ***MPI_ANY_TAG***
- ***MPI_ERROR*** contains error code information for the specific communication

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
}
```

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

Sending + Receiving

- ***MPI_Send*** is blocking. Whether buffering is used is implementation-specific. Typically, it will default to buffering with smaller messages and non-buffering with larger messages
- ***MPI_Recv*** is a blocking operation

*int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

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

Simple Send+Recv

```
int main(int argc, char *argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int data[10];

    if (rank == 0) {
        for (int i = 0; i < 10; i++)
             data[i] = i;
        printf("Rank 0: Before MPI_Send\n");
        fflush(stdout);
        MPI_Send(data, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Rank 0: After MPI_Send\n");
        fflush(stdout);
    } else if (rank == 1) {
        sleep(3);  // delay the receiver deliberately
        printf("Rank 1: About to call MPI_Recv\n");
        fflush(stdout);
        MPI_Recv(data, 10, MPI_INT, 0, 0, MPI_COMM_WORLD,
                         MPI_STATUS_IGNORE);
        printf("Rank 1: Received data successfully\n");
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

Simple Send+Recv

The integers which have been placed in ***data*** by process with ***rank==0*** are sending to process with ***rank=1***, which has an un-initialized array

```
int main(int argc, char *argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int data[10];

    if (rank == 0) {
        for (int i = 0; i < 10; i++)
            data[i] = i;
        printf("Rank 0: Before MPI_Send\n");
        fflush(stdout);
        MPI_Send(data, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Rank 0: After MPI_Send\n");
        fflush(stdout);
    } else if (rank == 1) {
        sleep(3);  // delay the receiver deliberately
        printf("Rank 1: About to call MPI_Recv\n");
        fflush(stdout);
        MPI_Recv(data, 10, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        printf("Rank 1: Received data successfully\n");
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

Simple Send+Recv

Send array of 10 ints

```
int main(int argc, char *argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int data[10];

    if (rank == 0) {
        for (int i = 0; i < 10; i++)
            data[i] = i;
        printf("Rank 0: Before MPI_Send\n");
        fflush(stdout);
        MPI_Send(data, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Rank 0: After MPI_Send\n");
        fflush(stdout);
    } else if (rank == 1) {
        sleep(3);  // delay the receiver deliberately
        printf("Rank 1: About to call MPI_Recv\n");
        fflush(stdout);
        MPI_Recv(data, 10, MPI_INT, 0, 0, MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);
        printf("Rank 1: Received data successfully\n");
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

Simple Send+Recv

- rank==0 sends to process with rank==1
- rank==1 receives from process with rank==0

```
int main(int argc, char *argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int data[10];

    if (rank == 0) {
        for (int i = 0; i < 10; i++)
             data[i] = i;
        printf("Rank 0: Before MPI_Send\n");
        fflush(stdout);
        MPI_Send(data, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Rank 0: After MPI_Send\n");
        fflush(stdout);
    } else if (rank == 1) {
        sleep(3);  // delay the receiver deliberately
        printf("Rank 1: About to call MPI_Recv\n");
        fflush(stdout);
        MPI_Recv(data, 10, MPI_INT, 0, 0, MPI_COMM_WORLD,
                      MPI_STATUS_IGNORE);
        printf("Rank 1: Received data successfully\n");
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

Simple Send+Recv

The ***tag*** is 0 for both to identify this specific message

```
int main(int argc, char *argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int data[10];

    if (rank == 0) {
        for (int i = 0; i < 10; i++)
            data[i] = i;
        printf("Rank 0: Before MPI_Send\n");
        fflush(stdout);
        MPI_Send(data, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Rank 0: After MPI_Send\n");
        fflush(stdout);
    } else if (rank == 1) {
        sleep(3);  // delay the receiver deliberately
        printf("Rank 1: About to call MPI_Recv\n");
        fflush(stdout);
        MPI_Recv(data, 10, MPI_INT, 0, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
        printf("Rank 1: Received data successfully\n");
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

Simple Send+Recv

All processes which are running (***MPI_COMM_WORLD)*** are chosen as the communicator

```
int main(int argc, char *argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int data[10];

    if (rank == 0) {
        for (int i = 0; i < 10; i++)
            data[i] = i;
        printf("Rank 0: Before MPI_Send\n");
        fflush(stdout);
        MPI_Send(data, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Rank 0: After MPI_Send\n");
        fflush(stdout);
    } else if (rank == 1) {
        sleep(3);  // delay the receiver deliberately
        printf("Rank 1: About to call MPI_Recv\n");
        fflush(stdout);
        MPI_Recv(data, 10, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        printf("Rank 1: Received data successfully\n");
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

Simple Send+Recv

We ignore any status information in this program

```
int main(int argc, char *argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int data[10];

    if (rank == 0) {
        for (int i = 0; i < 10; i++)
            data[i] = i;
        printf("Rank 0: Before MPI_Send\n");
        fflush(stdout);
        MPI_Send(data, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Rank 0: After MPI_Send\n");
        fflush(stdout);
    } else if (rank == 1) {
        sleep(3);  // delay the receiver deliberately
        printf("Rank 1: About to call MPI_Recv\n");
        fflush(stdout);
        MPI_Recv(data, 10, MPI_INT, 0, 0, MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);
        printf("Rank 1: Received data successfully\n");
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

Simple Send+Recv

What will the outputs of a buffered vs. non-buffer MPI_Send be?

```
int main(int argc, char *argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int data[10];

    if (rank == 0) {
        for (int i = 0; i < 10; i++)
            data[i] = i;
        printf("Rank 0: Before MPI_Send\n");
        fflush(stdout);
        MPI_Send(data, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Rank 0: After MPI_Send\n");
        fflush(stdout);
    } else if (rank == 1) {
        sleep(3);  // delay the receiver deliberately
        printf("Rank 1: About to call MPI_Recv\n");
        fflush(stdout);
        MPI_Recv(data, 10, MPI_INT, 0, 0, MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);
        printf("Rank 1: Received data successfully\n");
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

Simple Send+Recv

Buffered MPI_Send

Rank 0: Before MPI_Send
Rank 0: After MPI_Send
Rank 1: About to call MPI_Recv
Rank 1: Received data successfully

```
int main(int argc, char *argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int data[10];

    if (rank == 0) {
        for (int i = 0; i < 10; i++)
            data[i] = i;
        printf("Rank 0: Before MPI_Send\n");
        fflush(stdout);
        MPI_Send(data, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Rank 0: After MPI_Send\n");
        fflush(stdout);
    } else if (rank == 1) {
        sleep(3);  // delay the receiver deliberately
        printf("Rank 1: About to call MPI_Recv\n");
        fflush(stdout);
        MPI_Recv(data, 10, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        printf("Rank 1: Received data successfully\n");
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

Simple Send+Recv

Non-Buffered MPI_Send

Rank 0: Before MPI_Send
Rank 1: About to call MPI_Recv
Rank 0: After MPI_Send
Rank 1: Received data successfully

```
int main(int argc, char *argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int data[10];

    if (rank == 0) {
        for (int i = 0; i < 10; i++)
            data[i] = i;
        printf("Rank 0: Before MPI_Send\n");
        fflush(stdout);
        MPI_Send(data, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Rank 0: After MPI_Send\n");
        fflush(stdout);
    } else if (rank == 1) {
        sleep(3);  // delay the receiver deliberately
        printf("Rank 1: About to call MPI_Recv\n");
        fflush(stdout);
        MPI_Recv(data, 10, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        printf("Rank 1: Received data successfully\n");
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

Deadlocks with Send + Receive

Programs (especially those without buffered sends) are more likely to deadlock without great care

```
int a[10], b[10], myrank;
MPI_Status status;
...
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
   MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
   MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
```

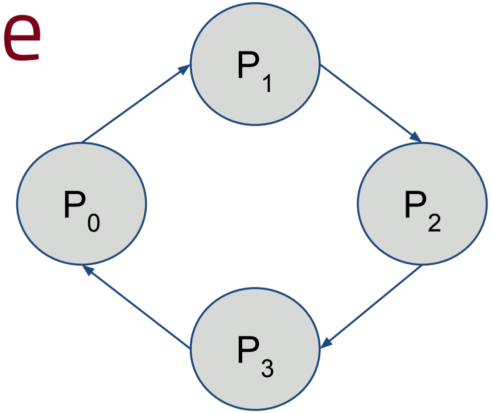
Deadlocks with Send + Receive

Programs (especially those without buffered sends) are more likely to deadlock without great care

Non-Buffered Deadlock

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
```

Deadlocks with Send + Receive



Non-Buffered Deadlock

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
```

```c
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
```

No more deadlocks

```c
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
   MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
   MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
}
else {
   MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
   MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
}
```

Simultaneous Communications

- The example of explicitly pairing sends + receives presented on the previous slide are rather frequent
- MPI provides two functions which are helpful for this kind of communication pattern

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, int dest, int sendtag,
        void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

```
int MPI_Sendrecv_replace(void *buf, int count,
        MPI_Datatype datatype, int dest, int sendtag,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

Simultaneous Communications

Fix deadlock with ***MPI_Sendrecv***

```
int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int send_val = rank;
    int recv_val = -1;

    int dest  = (rank + 1) % size;              // next process in ring
    int source = (rank - 1 + size) % size;      // previous process in ring

    MPI_Sendrecv(&send_val, 1, MPI_INT, dest, 0,
                            &recv_val, 1, MPI_INT, source, 0,
                            MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    printf("Process %d received %d from process %d\n", rank, recv_val, source);

    MPI_Finalize();
    return 0;
}
```

Simultaneous Communications

Fix deadlock with
MPI_Sendrecv_replace

```
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int val = rank;
    int dest   = (rank + 1) % size;          // next process in ring
    int source = (rank - 1 + size) % size; // previous process in ring

    MPI_Sendrecv_replace(&val, 1, MPI_INT,
                         dest,   0,
                         source, 0,
                         MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    printf("Rank %d received %d from rank %d\n", rank, val, source);

    MPI_Finalize();
    return 0;
}
```