

---

# COMP 322: Principles of Parallel Programming

## Lecture 19: Message Passing Interface, MPI (Chapter 6)

Fall 2009

<http://www.cs.rice.edu/~vsarkar/comp322>

Vivek Sarkar  
Department of Computer Science  
Rice University  
vsarkar@rice.edu



# Acknowledgments for today's lecture

---

- Course text: "Principles of Parallel Programming", Calvin Lin & Lawrence Snyder
  - Includes resources available at <http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html>
- "Parallel Architectures", Calvin Lin
  - Lectures 5 & 6, CS380P, Spring 2009, UT Austin
  - <http://www.cs.utexas.edu/users/lin/cs380p/schedule.html>
- Slides accompanying Chapter 6 of "Introduction to Parallel Computing", 2nd Edition, Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, Addison-Wesley, 2003
  - [http://www-users.cs.umn.edu/~karypis/parbook/Lectures/AG/chap6\\_slides.pdf](http://www-users.cs.umn.edu/~karypis/parbook/Lectures/AG/chap6_slides.pdf)
- MPI slides from "High Performance Computing: Models, Methods and Means", Thomas Sterling, CSC 7600, Spring 2009, LSU
  - <http://www.cct.lsu.edu/csc7600/coursemat/index.html>
- MPI lectures given at Rice HPC Summer Institute 2009, Tim Warburton, May 2009

# Summary of Last Two Lectures: A Look at Six Parallel Computers (Chapter 2)

---

- **Chip Multiprocessors**
    - Intel Core Duo (previous lecture)
    - AMD Dual Core Opteron (previous lecture)
  - **Symmetric Multiprocessors**
    - Sun Fire E25K (this lecture)
  - **Heterogeneous Processors**
    - Cell processor (this lecture)
    - GPUs (this lecture, only brief mention in Chapter 2)
  - **Clusters**
  - **Supercomputers**
    - Blue Gene/L (this lecture)
- } Motivation for MPI

# Principles of Message-Passing Programming

---

- The logical view of a machine supporting the message-passing paradigm consists of  $p$  processes, each with its own exclusive address space.
  1. Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
  2. All interactions (read-only or read/write) require cooperation of two processes - the process that has the data and the process that wants to access the data.
- These two constraints, while onerous, make underlying costs very explicit to the programmer.
- In the loosely synchronous model, tasks or subsets of tasks synchronize to perform interactions. Between these interactions, tasks execute completely asynchronously.
- Most message-passing programs are written using the *single program multiple data* (SPMD) model.

# MPI History

---

- **From 1992-1994, a community representing both vendors and users decided to create a standard interface to message passing calls**
  - In the context of distributed memory parallel computers (MPPs, there weren't really clusters yet)
- **MPI-1 was the result**
  - “Just” an API
  - FORTRAN77 and C bindings
  - Reference implementation (mpich) also developed
  - Vendors also kept their own internals (behind the API)
  - Vendor interfaces faded away over about 2 years

# MPI History (contd.)

---

- **Since then**
  - MPI-1.1
    - **Fixed bugs, clarified issues**
  - MPI-2
    - **Included MPI-1.2**
      - Fixed more bugs, clarified more issues
    - **Extended MPI**
      - New datatype constructors, language interoperability
    - **New functionality**
      - One-sided communication
      - MPI I/O
      - Dynamic processes
    - **FORTRAN90 and C++ bindings**
- **Primary MPI reference**
  - MPI Standard - on-line at: <http://www.mpi-forum.org/>

# MPI Resources

---

- **MPI:** <http://www.mcs.anl.gov/research/projects/mpi/>
- **MPICH2:** <http://www.mcs.anl.gov/research/projects/mpich2/>
- **Wiki:**  
[http://en.wikipedia.org/wiki/Message\\_Passing\\_Interface](http://en.wikipedia.org/wiki/Message_Passing_Interface)
- **Web tutorials:**
  - <https://computing.llnl.gov/tutorials/mpi/>
  - [http://www.ecmwf.int/services/computing/training/material/hpcf/Intro\\_MPI\\_Programming.pdf](http://www.ecmwf.int/services/computing/training/material/hpcf/Intro_MPI_Programming.pdf) (F77)
- **Books:**
  - <http://www.cs.usfca.edu/mpi/>
  - <http://www.netlib.org/utk/papers/mpi-book/mpi-book.html>
- **If** you know the MPI function name then google (tm) for man pages. Also on \*nix try: man MPI\_Foo

# The Minimal Set of MPI Routines

---

---

<code>MPI_Init</code>	Initializes MPI.
<code>MPI_Finalize</code>	Terminates MPI.
<code>MPI_Comm_size</code>	Determines the number of processes.
<code>MPI_Comm_rank</code>	Determines the label of calling process.
<code>MPI_Send</code>	Sends a message.
<code>MPI_Recv</code>	Receives a message.

---

---

# Basic MPI

## (Communication-Free subset)

# Our First MPI Program (procnumber.c)

---

```
#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("My process number is %d \n", myrank);
    MPI_Finalize();
}
```

main() is enclosed in an implicit "forall" --- each process runs a separate instance of main() with "index variable" = myrank

# Starting and Terminating the MPI Library

---

- `MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.
- `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.
- The prototypes of these two functions are:

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

```
int MPI_Finalize()
```

- `MPI_Init` also strips off any MPI related command-line arguments.
- All MPI routines, data-types, and constants are prefixed by "MPI\_". The return code for successful completion is `MPI_SUCCESS`.

# Code Spec 7.1 MPI\_Init().

---

```
MPI_Init()  
int MPI_Init(  
    int *argc,           // Initialize MPI  
    char ***argv,       // Number of command line arguments  
);                       // Command line arguments
```

## Arguments:

- Number of command line arguments.
- Command line arguments.

## Notes:

This routine must be called in every MPI process before any other MPI routine is called. It is an error to call this routine more than once in a process unless a subsequent `MPI_Finalize()` is called.

## Return value:

An MPI error code.

# Code Spec 7.2 `MPI_Finalize()`.

---

```
MPI_Finalize()  
int MPI_Finalize(  
);
```

**Notes:**

This routine should be the last MPI routine called in each process, and it should only be invoked after all other MPI routines have completed. In particular, any pending communication operations should complete before this routine is called.

**Return value:**

An MPI error code.

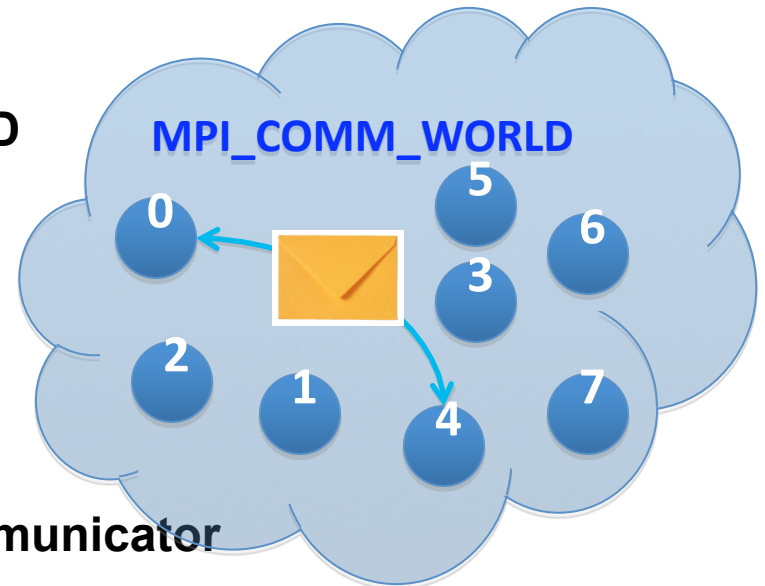
# Communicators

---

- A communicator defines a *communication domain* - a set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type `MPI_Comm`.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different (possibly overlapping) communication domains.
- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes.

# MPI Communicators

- Communicator is an internal object
- MPI Programs are made up of communicating processes
- Each process has its own address space containing its own attributes such as rank, size (and argc, argv, etc.)
- MPI provides functions to interact with it
- Default communicator is `MPI_COMM_WORLD`
  - All processes are its members
  - It has a size (the number of processes)
  - Each process has a rank within it
  - Can think of it as an ordered list of processes
- Additional communicator(s) can co-exist
- A process can belong to more than one communicator
- Within a communicator, each process has a unique rank



# Querying Information

---

- The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively.
- The calling sequences of these routines are as follows:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```
- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

## Code Spec 7.3 `MPI_Comm_Size()`.

---

```
MPI_Comm_Size()  
int MPI_Comm_Size(  
    MPI_Comm comm,           // Retrieve the number of tasks in  
    int *size,              // the specified communicator  
                           // The number of tasks  
);
```

### Arguments:

- The communicator of interest.
- A pointer to the size, whose target will contain the number of tasks in the specified communicator.

### Notes:

This routine obtains the number of processes in a communicator.

### Return value:

An MPI error code.

**Description:** Determines the size of the group associated with a communicator (*comm*). Returns an integer number of processes in the group underlying *comm* executing the program. The *comm* in the argument list refers to the communicator-group to be queried, the result of the query (size of the *comm* group) is stored in the variable *size*.

## Code Spec 7.4 MPI\_Comm\_Rank().

---

```
MPI_Comm_Rank()  
int MPI_Comm_Rank(           // Retrieve rank of a communicator  
    MPI_Comm comm,         // Communicator  
    int *rank,             // Rank  
);
```

### Notes:

This routine obtains a process' rank within a communicator.

### Arguments:

- The communicator of interest.
- A pointer to the rank, whose target will contain the rank of the specified communicator.

### Return value:

An MPI error code

**Description:** Returns the rank of the calling process in the group underlying the comm. The first (input) parameter *comm* in the argument list is the communicator to be queried, and the second (output) parameter *rank* is the integer number rank of the process in the group of *comm*.

# Our First MPI Program (procnumber.c)

---

```
#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("My process number is %d \n", myrank);
    MPI_Finalize();
}
```

# Running *procnumber*

---

- First we compile the *procnumber.c* code:
  - ▶ *mpicc -o procnumber procnumber.c*
- Second we run with *mpiexec* with 4 processes:
  - ▶ *mpiexec -n 4 ./procnumber*  
*My process number is: 1*  
*My process number is: 3*  
*My process number is: 0*  
*My process number is: 2*
- Notice that the processes do not print out in their numerical order. The processes execute in an asynchronous fashion !!!
- Running again:
  - ▶ *mpiexec -n 4 ./procnumber*  
*My process number is: 0*  
*My process number is: 1*  
*My process number is: 2*  
*My process number is: 3*

# The Minimal Set of MPI Routines

---

---

<code>MPI_Init</code>	Initializes MPI.
<code>MPI_Finalize</code>	Terminates MPI.
<code>MPI_Comm_size</code>	Determines the number of processes.
<code>MPI_Comm_rank</code>	Determines the label of calling process.
<code>MPI_Send</code>	Sends a message.
<code>MPI_Recv</code>	Receives a message.

---

- **Note:**
  - the processes have so far acted independently & no information has passed between the processes.
  - “embarrassingly parallel”, Cleve Moler.

---

# MPI Blocking Point-to-point Communication

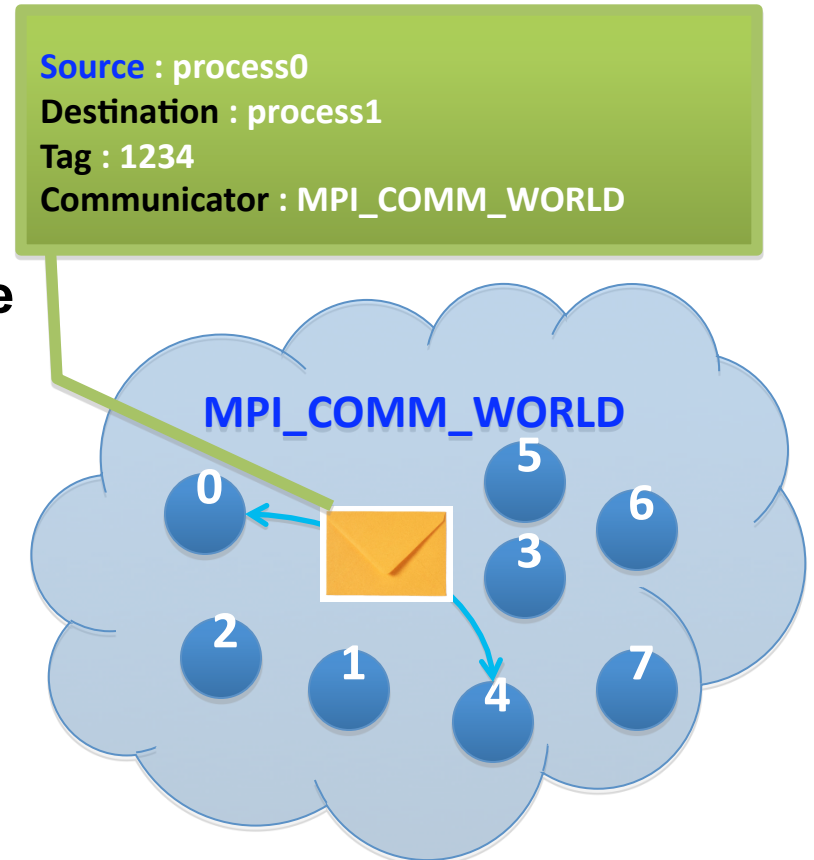
# MPI Point to Point Communication: Basic Idea

---

- A very simple communication between two processes is:
  - process zero sends ten doubles to process one
- In MPI this is a little more complicated than you might expect.
- Process zero has to tell MPI:
  - to send a message to process one
  - that the message contains ten entries
  - the entries of the message are of type double
  - the message has to be tagged with a label (integer number)
- Process one has to tell MPI:
  - to receive a message from process zero
  - that the message contains ten entries
  - the entries of the message are of type double
  - the label that process zero attached to the message

# Message Envelope

- Communication across process is performed using messages.
- Each message consists of a fixed number of fields that is used to distinguish them, called the Message Envelope :
  - Envelope comprises source, destination, tag, communicator
  - Message comprises Envelope + data
- Communicator refers to the namespace associated with the group of related processes



# MPI 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	

---

**You can also define your own (derived datatypes), such as an array of ints of size 100, or more complex examples, such as a struct or an array of structs**

# Figure 7.9 Creating a derived data type.

---

```
1 struct Person
2 {
3     int age;
4     char[6] name;
5 }
6
7 void buildPersonType(Person* p, MPI_Datatype* newType)
8 {
9     MPI_Datatype types[2];           /* The types of each field */
10    MPI_Aint offsets[2];             /* Offsets of each field */
11    MPI_Aint addresses[3];          /* Addresses of each field */
12    int block_lengths[2];           /* The lengths of each field */
13
14    types[0]=MPI_INT;
15    types[1]=MPI_CHAR;
16    block_lengths[0]=1;
17    block_lengths[1]=6;
18
19    /* Get the address of each field in a portable manner */
20    MPI_Address(p, &addresses[0]);
21    MPI_Address(&(p->age), &addresses[1]);
22    MPI_Address(&(p->name), &addresses[2]);
23
24    /* Compute offsets of each field */
25    offsets[0]=addresses[1]-addresses[0];
26    offsets[1]=addresses[2]-addresses[0];
27
28    /* Define and register the new type */
29    MPI_Type_struct(2, block_lengths, offsets, types, newType);
30    MPI_Type_commit(newType);
31 }
```

# MPI\_Status object

Object: **MPI\_Status**

Example usage :

```
MPI_Status status;
```

Description:

The MPI\_Status object is used by the receive functions to return data about the message, specifically the object contains the id of the process sending the message (MPI\_SOURCE), the message tag (MPI\_TAG), and error status (MPI\_ERROR) .

```
#include "mpi.h"
...
MPI_Status status; /* return status for */
...
MPI_Init(&argc, &argv);
...
if (my_rank != 0) {
...
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
else { /* my rank == 0 */
    for (source = 1; source < p; source++) {
        MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
    }
}
...
MPI_Finalize();
...
```

# Code Spec 7.5 MPI\_Send().

---

```
MPI_Send()  
int MPI_Send(                                // Blocking Send routine  
    void      *buffer,                       // Address of the data to send  
    int       count,                         // Number of data elements to send  
    MPI_Datatype type,                      // Type of data elements to send  
    int       dest,                         // ID of destination process  
    int       tag,                          // Tag to distinguish this message  
    MPI_Comm  *comm                         // An MPI communicator  
);
```

## Arguments:

- The address of the data to send.
- The number of data elements to send.
- The type of data elements to send.
- The ID of the process that should receive this message.
- A message tag that distinguishes this message from others that may be sent to the same process.
- The MPI communicator to use.

## Notes:

This routine sends data to another process. This routine has blocking semantics, which means that the routine does not return until the message has been sent. `MPI_Isend()` is a non-blocking version of the send operation; it takes a seventh parameter of type `MPI_Request` that is used to differentiate this send from other invocations of `MPI_Isend()` when waiting for completion.

## Return value:

An MPI error code.

# Code Spec 7.6 MPI\_Recv().

---

```
MPI_Recv()  
int MPI_Recv( // Blocking Receive routine  
    void *buffer, // Address at which to receive data  
    int count, // Number of elements to receive  
    MPI_Datatype type, // Type of each element  
    int source, // ID of sending process  
    int tag, // Identifier to distinguish message  
    MPI_Comm comm, // MPI communicator  
    MPI_Status *status // Status of this receive operation  
);
```

## Arguments:

- The first six arguments correspond to MPI\_Send()
- To receive a message from any other process, use MPI\_ANY\_SOURCE as the source.
- To match on any tag, use MPI\_ANY\_TAG as the fifth parameter.

## Notes:

This routine receives data from another process. This routine has blocking semantics—it does not return until the message is received. MPI\_Irecv() is a non-blocking version of the receive operation; it takes a seventh parameter of type MPI\_Request that is used to differentiate this receive from other invocations of MPI\_Irecv() when waiting for completion.

## Return value:

An MPI error code.

# Notes

---

- Because each process is autonomous it is important to note that:
  - A pointer for one process is not a valid pointer for another process.
  - The hardware running each process may not be of the same type. e.g. a mix of 32 bit and 64 bit processors may be used together.
  - Each process has its own:
    - Variables
    - Memory space
    - Sequence of instructions
    - Pacing [ i.e. the processes do not perform operations in lockstep ]
- In point-to-point communication it is not possible to simply copy between the two data arrays because the arrays for any one process are invisible to all other processes except through MPI communication.

# Global View vs. Local View

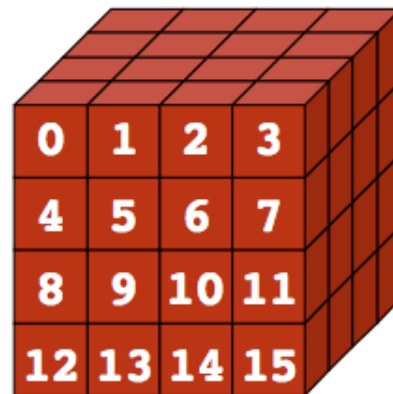
---

## Distributed memory

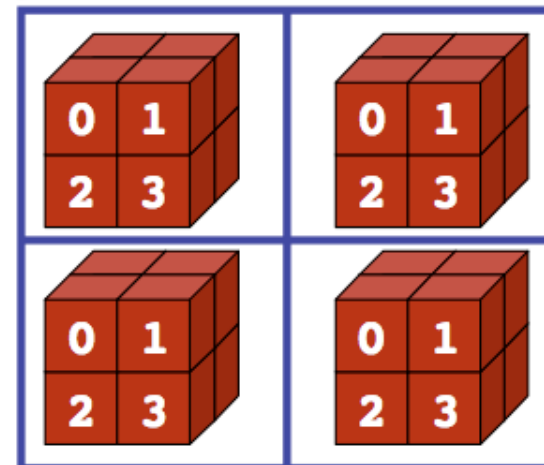
- Each process sees a local address space
- Processes send messages to communicate with other processes

## Data structures

- Presents a Local View instead of Global View
- Programmer must make the mapping



Global View



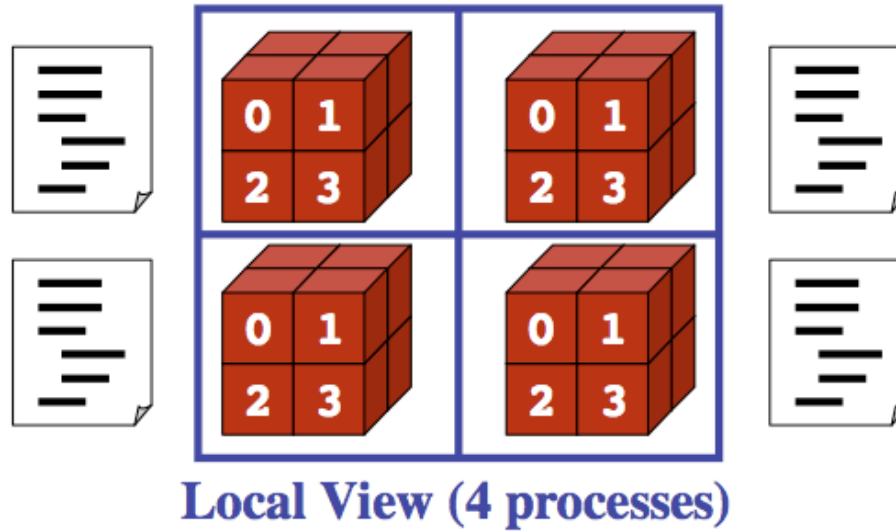
Local View (4 processes)

# Single Program Multiple Data model (SPMD)

---

## SPMD code

- Write one piece of code that executes on each processor



## SPMD vs. SIMD?

- SIMD is a hardware execution model
- Each instruction executes in lock step
- SPMD is a software execution model– each process executes independently

# Process Branching in SPMD programs

---

- In `procnumber.c` each process executed the same instructions.
- We can use conditional statements so that different processes perform operations unique to their number.
- In `procbranch.c` we first find the process number.
  1. Each process checks to see if it is process 0. Only process 0 prints out "First"
  2. Each process checks to see if it is process 1. Only process 1 prints out "Second"
  3. Each process checks to see if it is process > 1. Only these processes print out "No medal"

```
/* procbranch.c */  
/* find the number attached to this process */  
MPI_Comm_rank(MPI_COMM_WORLD, &procid);  
/* this process will print a message based on procid */  
  
if(procid==0) printf("First\n");  
if(procid==1) printf("Second\n");  
if(procid>1) printf("No medal\n");
```

# Running *procbranch*

---

- We compile and run as usual:

- ▶ *mpicc -o procbranch procbranch.c*
- ▶ *mpiexec -n 7 ./procbranch*

*Second*  
*First*  
*No medal*  
*No medal*  
*No medal*  
*No medal*  
*No medal*

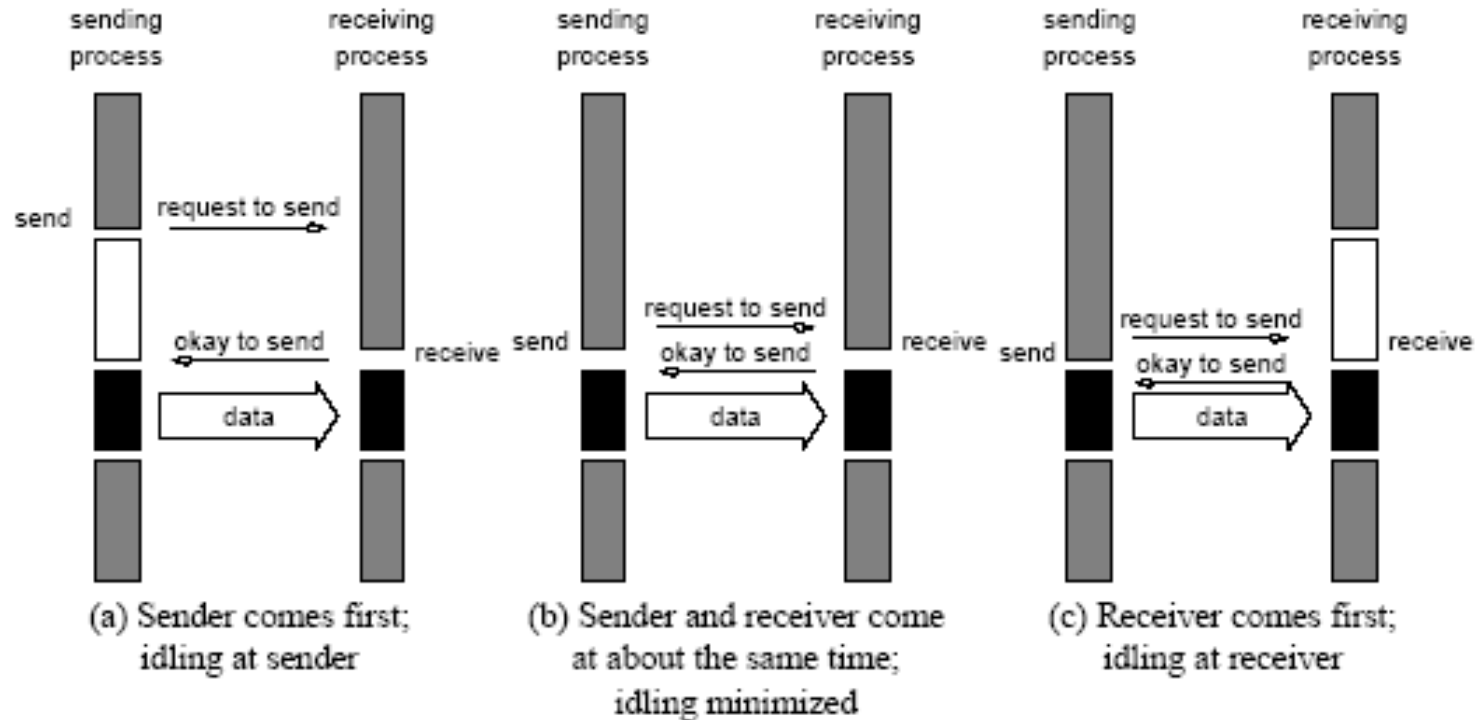
- Notice:
  - Again the processes do not report in numerical order
  - The text string output depends on the process number.

# Non-Buffered Blocking Message Passing Operations

---

- A simple method for forcing send/receive semantics is for the send operation to return only when it is safe to do so.
- In the non-buffered blocking send, the operation does not return until the matching receive has been encountered at the receiving process.
- Idling and deadlocks are major issues with non-buffered blocking sends.
- In buffered blocking sends, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The data is copied at a buffer at the receiving end as well.
- Buffering alleviates idling at the expense of copying overheads.

# Non-Buffered Blocking Message Passing Operations



Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

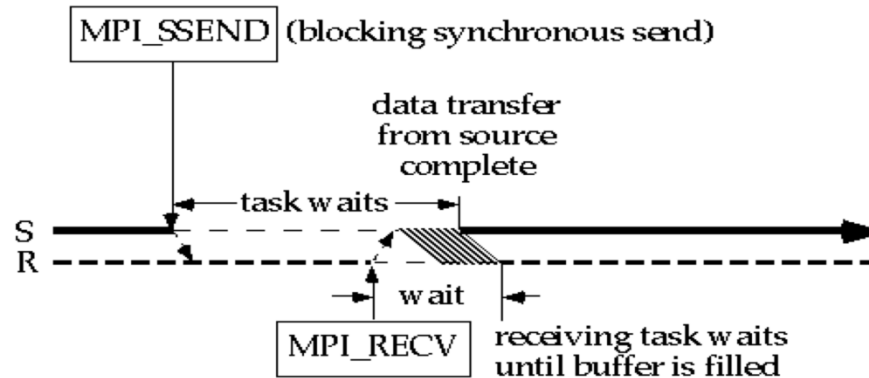
# Modes of Send

---

- The MPI standard defines four modes of send:
  - Standard
  - Synchronous
  - Buffered
  - Ready

# Point to Point Communication

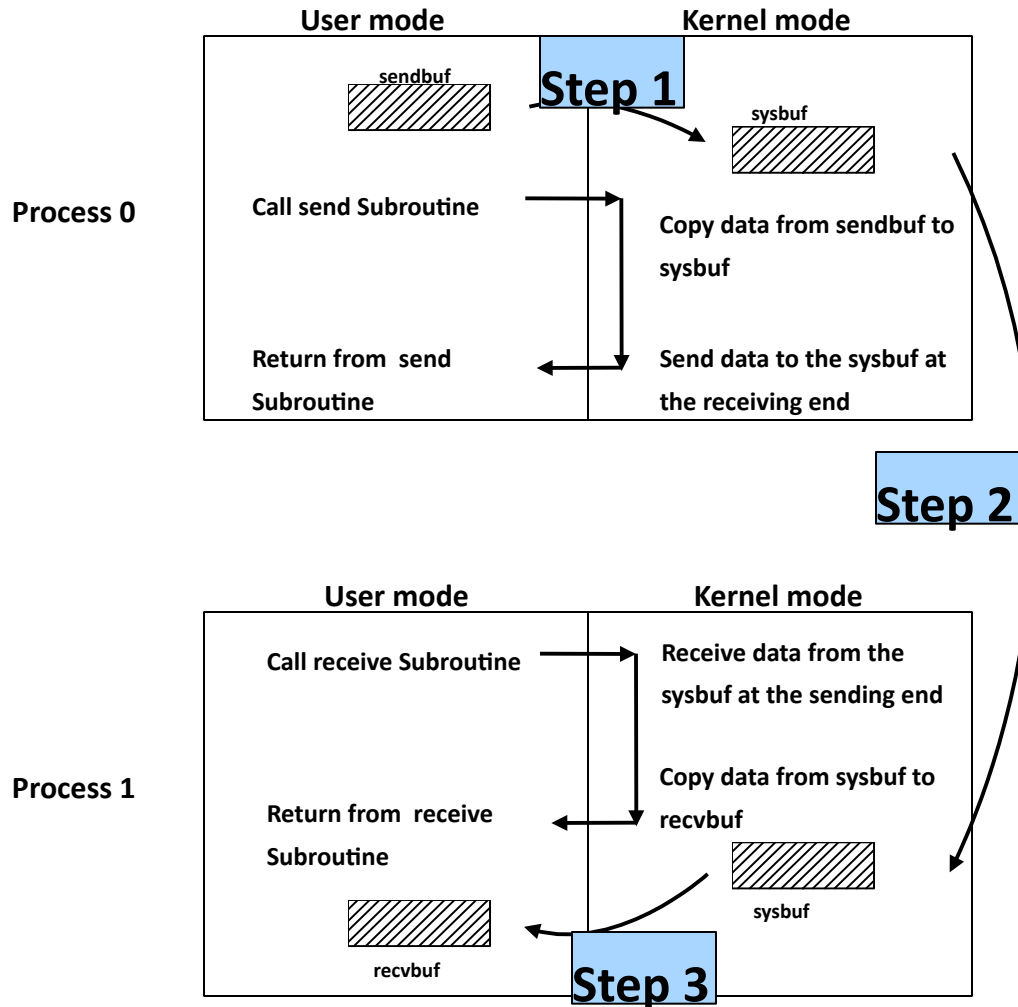
## Blocking Synchronous Send



- The communication mode is selected while invoking the send routine.
- When *blocking synchronous send* is executed (`MPI_Ssend()`), “*ready to send*” message is sent from the sending task to receiving task.
- When the *receive call* is executed (`MPI_Recv()`), “*ready to receive*” message is sent, followed by the transfer of data.
- The sender process must wait for the receive to be executed and for the handshake to arrive before the message can be transferred. (Synchronization Overhead)
- The receiver process also has to wait for the handshake process to complete. (Synchronization Overhead)
- Overhead incurred while copying from sender & receiver buffers to the network.

Source: [http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3\\_pt2pt2.html](http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html)

# Point to Point Communication : Basic concepts (buffered)



1. Data to be sent by the user is copied from the user memory space to the system buffer
2. The data is sent from the system buffer over the network to the system buffer of receiving process
3. The receiving process copies the data from system buffer to local user memory space

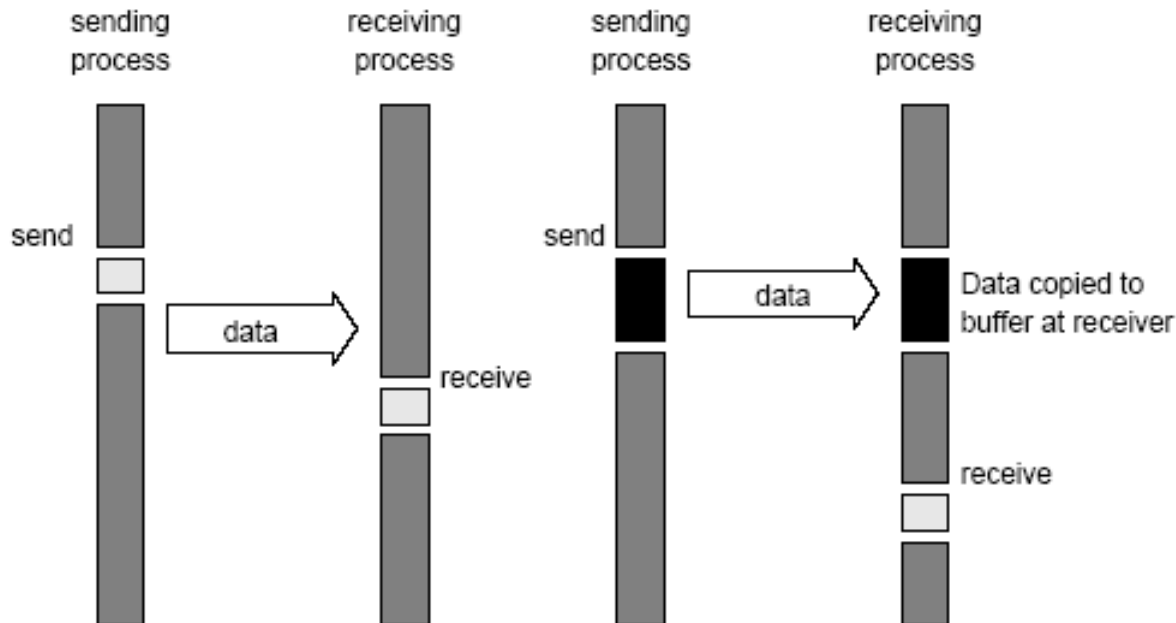
# Buffered Blocking Message Passing Operations

---

- A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends.
- The sender simply copies the data into the designated buffer and returns after the copy operation has been completed.
- The data must be buffered at the receiving end as well.
- Buffering trades off idling overhead for buffer copying overhead.

# Buffered Blocking Message Passing Operations

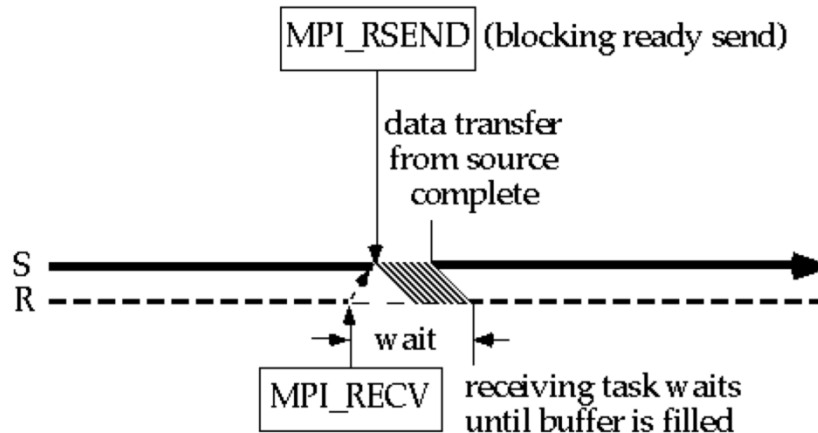
---



Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

# Point to Point Communication

## Blocking Ready Send

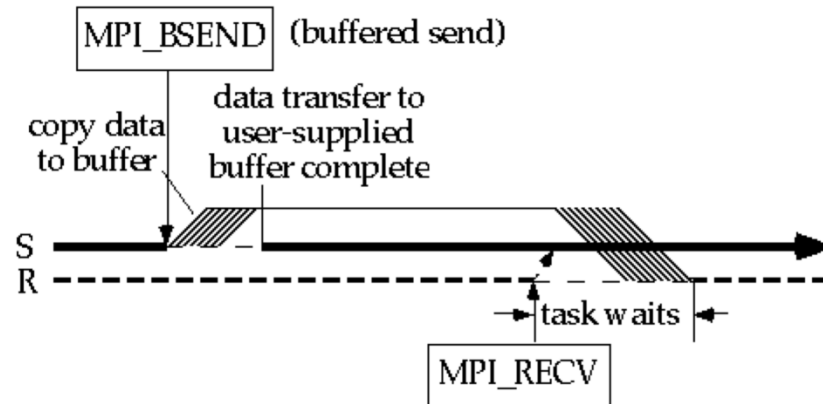


- The ready mode send call (`MPI_Rsend`) sends the message over the network once the “ready to receive” message is received.
- If “ready to receive” message hasn’t arrived, the ready mode send will incur an error and exit. The programmer is responsible to provide for handling errors and overriding the default behavior.
- The ready mode send call minimizes system overhead and synchronization overhead incurred during sending of the task.
- The receive still incurs substantial synchronization overhead depending on how much earlier the receive call is executed.

Source: [http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3\\_pt2pt2.html](http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html)

# Point to Point Communication

## Blocking Buffered Send

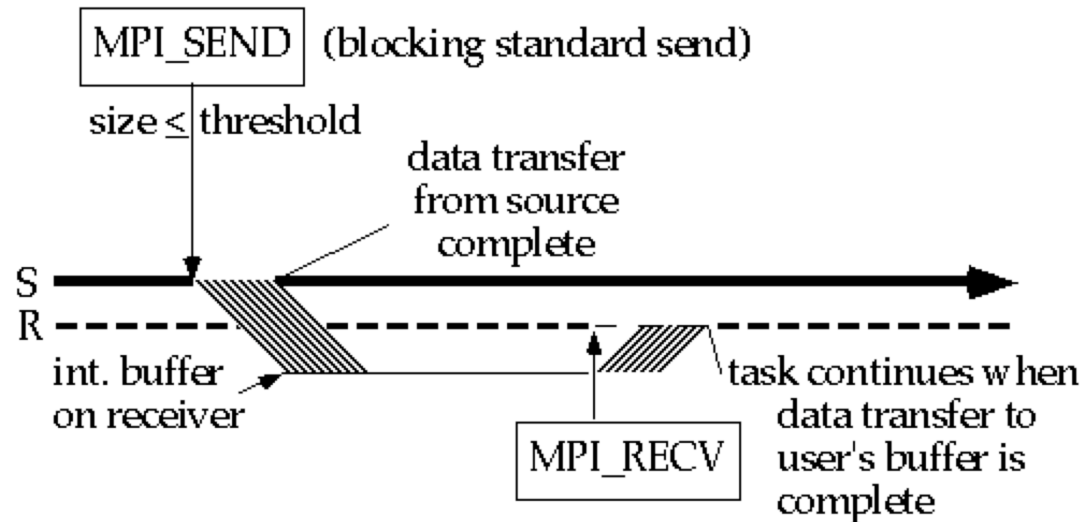


- The blocking buffered send call (`MPI_Bsend()`) copies the data from the message buffer to a user-supplied buffer and then returns.
- The message buffer can then be reclaimed by the sending process without having any effect on any data that is sent.
- When the “ready to receive” notification is received the data from the user-supplied buffer is sent to the receiver.
- Replicated copies of the buffer results in added system overhead.
- Synchronization overhead on the sender process is eliminated as the sending process does not have to wait on the receive call.
- Synchronization overhead on the receiving process can still be incurred, because if the receive is executed before the send, the process must wait before it can return to the execution sequence

Source: [http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3\\_pt2pt2.html](http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html)

# Point to Point Communication

## Blocking Standard Send

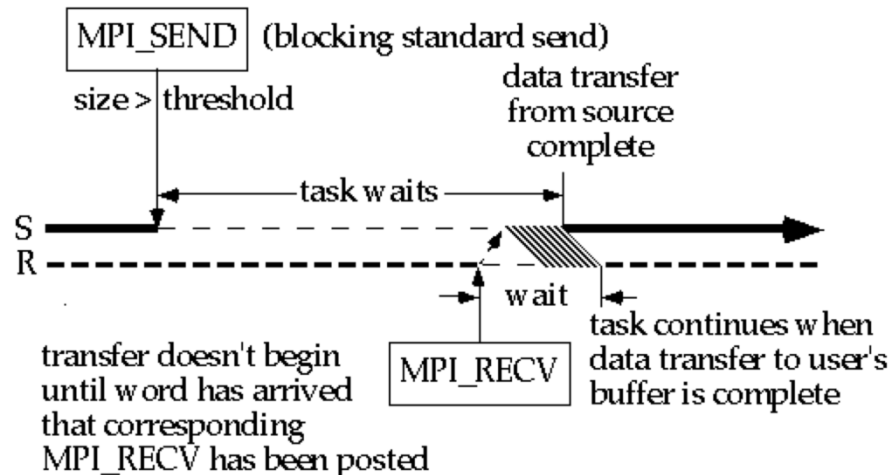


- The `MPI_Send()` operation is implementation dependent
- When the data size is smaller than a threshold value (*varies for each implementation*):
  - The blocking standard send call (`MPI_Send()`) copies the message over the network into the system buffer of the receiving node, after which the sending process continues with the computation
  - When the receive call (`MPI_Recv()`) is executed the message is copied from the system buffer to the receiving task
  - The decreased synchronization overhead is usually at the cost of increased system overhead due to the extra copy of buffers

Source: [http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3\\_pt2pt2.html](http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html)

# Point to Point Communication

## Buffered Standard Send



- **When the message size is greater than a threshold**
  - The behavior is same as for the synchronous mode
  - Small messages benefit from the decreased chance of synchronization overhead
  - Large messages results in increased cost of copying to the buffer and system overhead

Source: [http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3\\_pt2pt2.html](http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html)

# Buffered Blocking Message Passing Operations

---

Bounded buffer sizes can have significant impact on performance.

P0	P1
<pre>for (i=0; i&lt;1000; i++){     produce_data(&amp;a);     send(&amp;a, 1, 1); }</pre>	<pre>for (i=0; i&lt;1000; i++){     receive(&amp;a, 1, 0);     consume_data(&amp;a); }</pre>

What if consumer was much slower than producer?

# Buffered Blocking Message Passing Operations

---

Deadlocks are still possible with buffering since receive operations block.

P0

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

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

P1

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

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

# Sending and Receiving Messages

---

- MPI allows specification of wildcard arguments for both source and tag.
- If source is set to `MPI_ANY_SOURCE`, then any process of the communication domain can be the source of the message.
- If tag is set to `MPI_ANY_TAG`, then messages with any tag are accepted.
- On the receive side, the message must be of length equal to or less than the length field specified.

# Sending and Receiving Messages

---

- On the receiving end, the status variable can be used to get information about the MPI\_Recv operation.

- The corresponding data structure contains:

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

- The MPI\_Get\_count function returns the precise count of data items received.

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype  
                 datatype, int *count)
```

# Avoiding Deadlocks

---

Consider:

```
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);
}
...
```

If `MPI_Send` is blocking, there is a deadlock.

---

# Avoiding Deadlocks

---

Consider the following piece of code, in which process  $i$  sends a message to process  $i + 1$  (modulo the number of processes) and receives a message from process  $i - 1$  (modulo the number of processes).

```
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);
...
```

Once again, we have a deadlock if `MPI_Send` is blocking.

# Avoiding Deadlocks

---

We can break the circular wait to avoid deadlocks as follows:

```
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);
}
...
```

# Sending and Receiving Messages Simultaneously

---

To exchange messages, MPI provides the following function:

```
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)
```

The arguments include arguments to the send and receive functions. If we wish to use the same buffer for both send and receive, we can use:

```
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)
```

# Blocking

---

- The *MPI\_Send* and *MPI\_Recv* point to point communication functions we just saw are referred to as “*blocking*” send and receive operations
- The “blocking” send has the characteristic that the *MPI\_Send* function will not return until the data has been copied out of the user supplied array and into an MPI buffer array
- “blocking” in this context does not mean that the data has to arrive before the function finishes
- The function will end after the user supplied array buffer is free to be used again without changing the outgoing message [ specific details depend on the MPI implementation ]
- The “blocking” receive will only end after the data is safely placed in the user supplied array for the incoming data
- Summary: As soon as either *MPI\_Send* or *MPI\_Recv* finish the user supplied data array is ready to be used again

# Blocking Deadlock V1

---

- The “blocking” *MPI\_Send* and *MPI\_Recv* send and receive calls are quite straight forward.
- However, consider the following communication pattern in pseudo-code:

```
if(procid==0){
  MPI_Recv from 1;
  MPI_Send to 1;
}
if(procid==1){
  MPI_Recv from 0;
  MPI_Send to 0;
}
```

- This will never finish because each process is waiting to receive data from the other that has not been sent yet.

# Blocking Deadlock V2

---

- Next consider the following communication pattern in pseudo-code:

```
if(procid==0){
  MPI_Send to 1;
  MPI_Recv from 1;
}
if(procid==1){
  MPI_Send to 0;
  MPI_Recv from 0;
}
```

- *MPI\_Send* blocks until the data are copied out of the user array
- There will be deadlock if the system has insufficient buffer space for both messages
- If the system can allocate enough buffer for both sends to go, then they will return, and the *MPI\_Recv* will start.
- The order **depends on the system resources** and is unsafe to use.

# No Deadlock Blocking

---

- The following achieves the same goal but is safe:

```
if(procid==0){
  MPI_Send to 1;
  MPI_Recv from 1;
}
if(procid==1){
  MPI_Recv from 0;
  MPI_Send to 0;
}
```

- In this case process zero does not have to compete for outgoing buffer space with process one when it tries to *MPI\_Send*
- And vice-versa on the second *MPI\_Send* from process one.

# More Blocking Badness

---

- Apart from having to be very careful to avoid deadlocking your MPI computations there are important efficiency based reasons to avoid “*blocking*” communications.
- There is a large potential for poor performance when using the “*blocking*” *MPI\_Send* and *MPI\_Recv* functions.
- The time spent inside *MPI\_Send* waiting for outgoing buffer space to free up is wasted time that could be used for computation.
- An opportunity for higher performance is missed if the code could be doing some computation instead of waiting for the *MPI\_Recv* to finalize because the data is in transit.
- Use of “*non-blocked*” communications can avoid some of these pitfalls...

# Another Deadlock Example

---

```
If (rank == 0) {  
    err = MPI_Send(sendbuf, count, datatype, 1, tag, comm);  
    err = MPI_Recv(recvbuf, count, datatype, 1, tag, comm, &status);  
}else {  
    err = MPI_Send(sendbuf, count, datatype, 0, tag, comm);  
    err = MPI_Recv(recvbuf, count, datatype, 0, tag, comm, &status);  
}
```

- If the message sizes are small enough, this should work because of systems buffers
- If the messages are too large, or system buffering is not used, this will hang

# Deadlock Example Solutions

---

```
If (rank == 0) {
    err = MPI_Send(sendbuf, count, datatype, 1, tag, comm);
    err = MPI_Recv(recvbuf, count, datatype, 1, tag, comm, &status);
} else {
    err = MPI_Recv(recvbuf, count, datatype, 0, tag, comm, &status);
    err = MPI_Send(sendbuf, count, datatype, 0, tag, comm);
}
```

or

```
If (rank == 0) {
    err = MPI_Isend(sendbuf, count, datatype, 1, tag, comm, &req);
    err = MPI_Recv(recvbuf, count, datatype, 1, tag, comm);
    err = MPI_Wait(req, &status);
} else {
    err = MPI_Isend(sendbuf, count, datatype, 0, tag, comm, &req);
    err = MPI_Recv(recvbuf, count, datatype, 0, tag, comm);
    err = MPI_Wait(req, &status);
}
```