ME964
High Performance Computing for Engineering Applications

Parallel Computing with MPI

Building/Debugging MPI Executables

MPI Send/Receive

Collective Communications with MPI

April 10, 2012

“The best things in life aren’t things.”
- Art Buchwald, Pulitzer Prize winner

Before We Get Started…

- Last lecture
  - Message Passing Interface (MPI) standard
  - Learn how to run an MPI executable on Euler
  - MPI Send: looking under the hood

- Today
  - Running and Debugging MPI code from Eclipse
  - MPI Send/Receive, cntd.
  - Collective communications: MPI_Broadcast, MPI_Reduce, MPI_Scatter

- Other issues
  - Two page or shorter Final Project proposal due today, use Mercurial
    - If you want to continue the Midterm Project then generate a doc with one line stating so
    - If Final Project is default choice; i.e., banded solver, I'll provide a doc spelling out what needs to be accomplished
    - I encourage you to do something related to your research
  - Midterm Project due on Thursday, 11:59 pm

Midterm Exam Coming Up…

- One week from today
  - You can bring your class notes, no other textbooks, no internet access

- There will be a review session at 6 pm on Monday, April 16
  - Room TBA
  - I'll try to answer questions you might have in relation to material covered so far

- Midterm Exam covers
  - CUDA/GPU computing
  - MPI material, three lectures

Goals/Philosophy of MPI

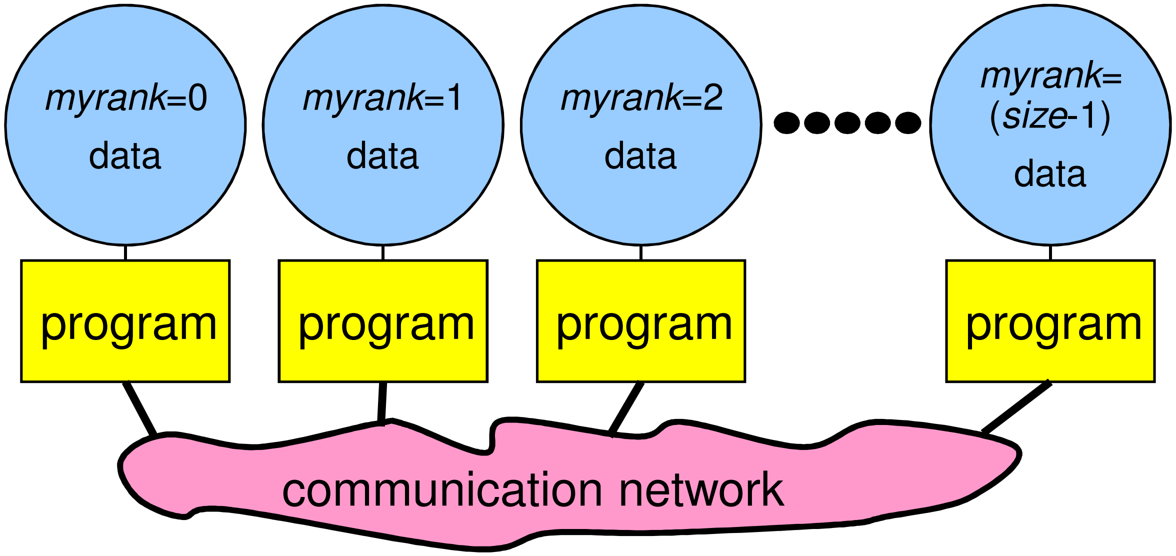
- MPI's prime goals
  - Provide a message-passing interface for parallel computing
  - Make source-code portability a reality
  - Provide a set of services (building blocks) that increase developer's productivity

- The philosophy behind MPI:
  - Specify a standard and give vendors the freedom to go about its implementation
  - Standard should be hardware platform & OS agnostic – key for code portability

Data and Work Distribution

- To communicate together MPI processes need identifiers: **rank = identifying number**

- All work distribution decisions are based on the ***rank***
  - Helps establish which process works on which data
  - Just like we had thread and block IDs in CUDA



myrank=0 data

myrank=1 data

myrank=2 data

myrank= (*size*-1) data

program

program

program

program

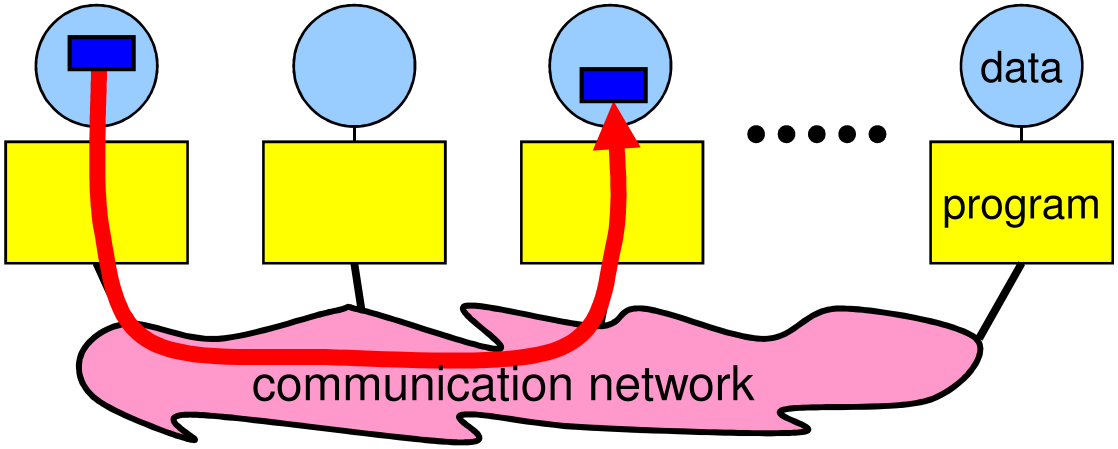
communication network

[ICHEC]→

Message Passing

- Messages are packets of data moving between different processes
- Necessary information for the message passing system:
  - sending process + receiving process } i.e., the two “ranks”

  - source location + destination location
  - source data type + destination data type
  - source data size + destination buffer size } 🟦



MPI: An Example Application

[From previous lecture]

```c
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int         my_rank;       /* rank of process      */
    int         p;             /* number of processes  */
    int         source;        /* rank of sender       */
    int         dest;          /* rank of receiver     */
    int         tag = 0;       /* tag for messages     */
    char        message[100];  /* storage for message  */
    MPI_Status  status;        /* return status for receive  */

    MPI_Init(&argc, &argv); // Start up MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
    MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        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);
            printf("%s\n", message);
        }
    }

    MPI_Finalize(); // Shut down MPI
    return 0;
} /* main */
```

Program Output

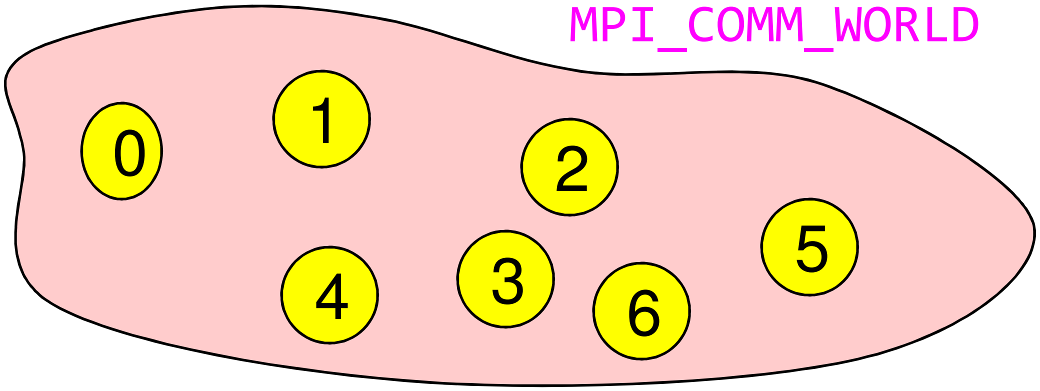
```
[negrut@euler CodeBits]$ mpiexec -np 8 ./greetingsMPI.exe
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
[negrut@euler CodeBits]$
```

Communicator MPI_COMM_WORLD

- All processes of an MPI program are members of the default communicator MPI_COMM_WORLD

- MPI_COMM_WORLD is a predefined **handle** in **mpi.h**

- Each process has its own **rank** in a given communicator:
  - starting with 0
  - ending with (size-1)



- You can define a new communicator in case you find it useful
  - Use MPI_Comm_create call. Example creates the communicator DANS_COMM_WORLD

```
MPI_Comm_create(MPI_COMM_WORLD, new_group, &DANS_COMM_WORLD);
```

[ICHEC]→

MPI_Comm_create

- Synopsis

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm);
```

- Input Parameters
  - `comm` - communicator (handle)
  - `group` - subset of the family of processes making up the `comm` (handle)

- Output Parameter
  - `comm_out` - new communicator (handle)

Point-to-Point Communication

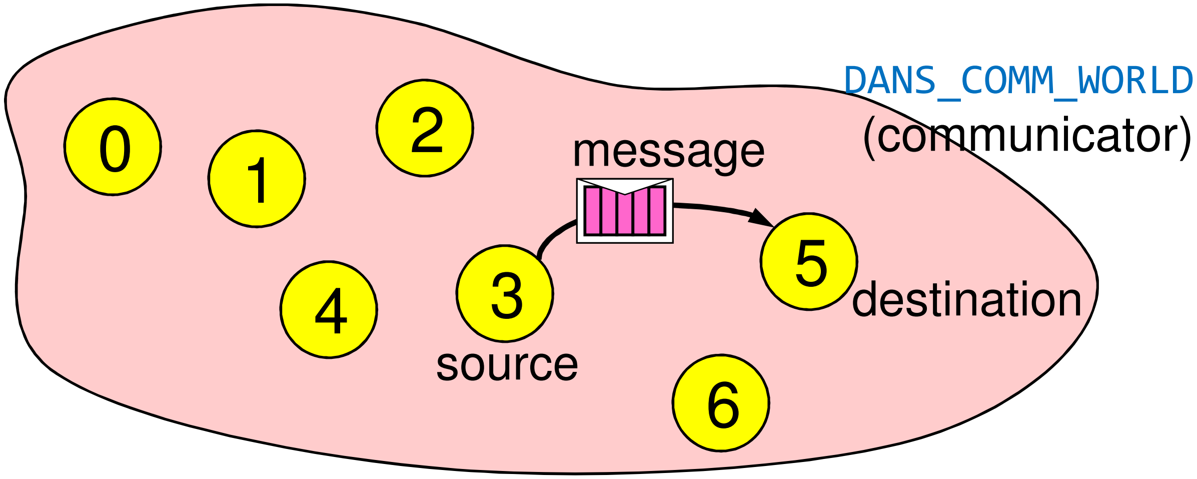
- Simplest form of message passing

- One process sends a message to another process
  - `MPI_Send`
  - `MPI_Receive`

- Sends and receives can be
    - Blocking
    - Non-blocking
  - More on this shortly

Point-to-Point Communication

- Communication between two processes
- Source process sends message to destination process
- Communication takes place within a communicator, e.g., `DANS_COMM_WORLD`
- Processes are identified by their ranks in the communicator



[ICHEC]→

MPI_Send & MPI_Recieve: The Eager and Rendezvous Flavors

- If you send small messages, the content of the buffer is sent to the receiving partner immediately
  - Operation happens in “eager mode”

- If you send a large amount of data, you (the sender) wait for the receiver to post a receive before sending the actual data of the message

- Why this eager-rendezvous dichotomy?
  - Because of the size of the data and the desire to have a safe implementation
  - If you send small amount of data, the MPI implementation can buffer the content and actually carry out the transaction later on when the receiving process asks for data
    - Can’t play this trick if you place a send with 10 GB of data :-)

MPI_Send & MPI_Recieve: Blocking vs. Non-blocking

- NOTE: Each implementation of MPI has a default value (which might change during at run time) beyond which a larger `MPI_Send` stops acting “eager”
    - The MPI standard doesn’t provide specifics
    - You don’t know how large it too large…

- In the message-passing paradigm for parallel programming you’ll always have to deal with the fact that the data that you send needs to “live” somewhere before it is received and the send-receive transaction completes

- This is where the blocking & non-blocking issue comes into play
    - Blocking send: upon return from a send operation, you can modify the content of the buffer in which you stored data to be sent since the data has been sent
        - The data “lives” in your buffer, so problem solved
    - Non-blocking: the send call returns immediately and there is no guarantee that the data has actually been transmitted upon return from send call
        - Take home message: before you modify the content of the buffer you better make sure (through a MPI status call) that the send actually completed

MPI_Send & MPI_Recieve: Blocking vs. Non-blocking

- If non-blocking, the data “lives” in your buffer – that’s why it’s not safe to change it since you don’t know when transaction was closed
  - This typically realized through a `MPI_Isend`
    - “I” stands for “immediate”

- However, there is a second way of you providing a buffer region, where your sent message is buffered so that the fact that it’s non-blocking is still safe
  - This typically realized through `MPI_Bsend`
    - “B” stands for “buffered”
  - The problem here is that *you* need to provide this additional buffer that stages the transfer
    - Interesting question: how large should *that* staging buffer be?
  - Adding another twist to the story: if you keep posting non-blocking sends that are not matched by corresponding “`MPI_Receive`” operations, you are going to overflow this staging buffer

MPI_Send & MPI_Recieve: Blocking vs. Non-blocking

- The plain vanilla MPI_Send & MPI_Recieve pair is blocking
  - It's safe to modify the data buffer upon return

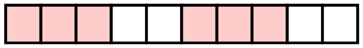
- The problem with plain vanilla:
  - 1: when sending large messages, there is no overlap of compute & data movement
    - This is what we strived for when using "streams" in CUDA
  - 2: if not done properly, the processes executing the MPI code can hang

- There are several other flavors of send/receive operations, to be discussed later, that can help with concerns 1 and 2 above

Messages

- A message contains a number of elements of some particular data type.

- MPI data types:
  - Basic data type
  - Derived data types



- Data type handles are used to describe the type of the data in the memory

Example: message with 5 integers

2345	654	96574	-12	7676

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	

Example:

2345	654	96574	-12	7676

count=5
datatype=MPI_INT

int arr[5]

[ICHEC]→

The Mechanics of P2P Communication: Sending a Message

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

- `buf` is the starting point of the message with `count` elements, each described with `datatype`

- `dest` is the rank of the destination process within the communicator `comm`

- `tag` is an additional nonnegative integer piggyback information, additionally transferred with the message
  - The `tag` can be used to distinguish between different messages
  - Rarely used

The Mechanics of P2P Communication: Receiving a Message

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

- `buf`/`count`/`datatype` describe the receive buffer
- Receiving the message sent by process with rank `source` in `comm`
- Only messages with matching `tag` are received
- Envelope information is returned in the `MPI_Status` object `status`

[ICHEC]→

MPI_Recv:
The Need for an MPI_Status Argument

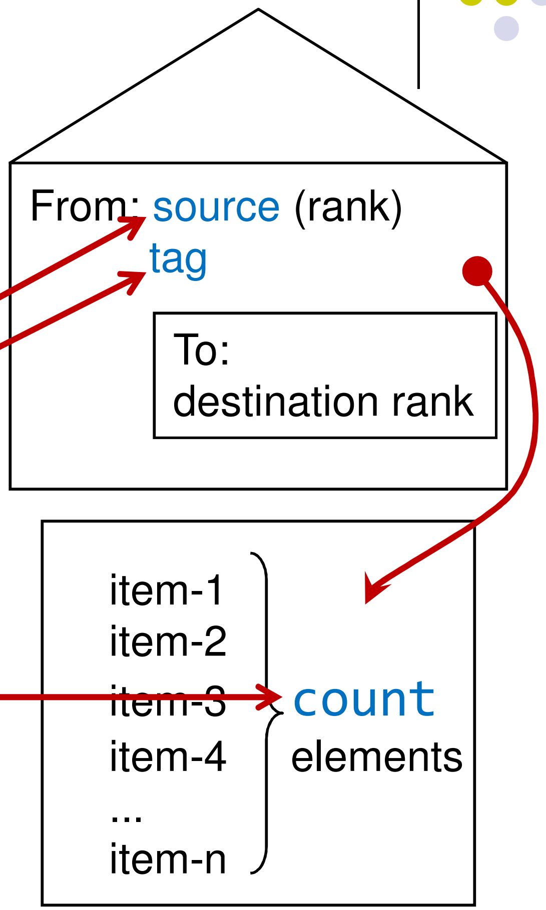
- The MPI_Status object returned by the call settles a series of questions:
  - The receive call does not specify the size of an incoming message, but only an upper bound
  - If multiple requests are completed by a single MPI function, a distinct error code may need to be returned for each request
  - The source or tag of a received message may not be known if wildcard values were used in a receive operation

The Mechanics of P2P Communication: Wildcarding

- Receiver can wildcard
  - To receive from any source – `source` = `MPI_ANY_SOURCE`
  - To receive from any tag – `tag` = `MPI_ANY_TAG`
  - Actual source and tag returned in receiver’s `status` argument

The Mechanics of P2P Communication: Communication Envelope

- Envelope information is returned from MPI_RECV in `status`.
- `status.MPI_SOURCE`
  `status.MPI_TAG`
  `count` via `MPI_Get_count()`



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

The Mechanics of P2P Communication: Some Rules of Engagement

For a communication to succeed:

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Message data types must match
- Receiver's buffer must be large enough

Blocking Type: Communication Modes

- Send communication modes:
  - Synchronous send → `MPI_SSEND`
  - Buffered [asynchronous] send → `MPI_BSEND`
  - Standard send → `MPI_SEND`
  - Ready send → `MPI_RSEND`
- Receiving all modes → `MPI_RECV`

Communication Modes — Summary

Sender modes	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH
Synchronous **MPI_SEND**	Standard send. Either uses an internal buffer or buffered	
Ready send **MPI_RSEND**	May be started **only** if the matching receive is already posted!	avoid, might cause unforeseen problems…
Receive **MPI_RECV**	Completes when a the message (data) has arrived	

[ICHEC]→

Comments, Communication Modes

- Standard send (**MPI_SEND**)
  - minimal transfer time
  - may block due to synchronous mode
  - → risks with synchronous send

- Synchronous send (**MPI_SSEND**)
  - risk of deadlock
  - risk of serialization
  - risk of waiting → idle time
  - high latency / best bandwidth

- Buffered send (**MPI_BSEND**)
  - low latency / bad bandwidth

- Ready send (**MPI_RSEND**)
  - use **never**, except you have a 200% guarantee that Recv is already called in the current version and all future versions of your code

Synchronous/Buffered Sending in MPI

- Synchronous with MPI_Ssend
  - In synchronous mode, a send will not complete until a matching receive is posted.
    - Copy data to the network, wait for an acknowledgement
    - The sender has to wait for a receive to be posted
    - No buffering of data

- Buffered with MPI_Bsend
  - Send completes once message has been buffered internally by MPI
    - Buffering incurs an extra memory copy
    - Does not require a matching receive to be posted
    - May cause buffer overflow if many bsends and no matching receives have been posted yet

[A. Snavely]→

Standard/Ready Send

- Standard with MPI_Send
  - Up to MPI to decide whether to do synchronous or buffered, for performance reasons
  - The rationale is that a correct MPI program should not rely on buffering to ensure correct execution

- Ready with MPI_Rsend
  - May be started *only* if the matching receive has been posted
  - Can be done efficiently on some systems as no hand-shaking is required

[A. Snavely]→

Most Important Issue: Deadlocking

- Deadlock situations: appear when due to a certain sequence of commands the execution hangs

PROCESS 0		PROCESS 1

...
MPI_Ssend()
MPI_Recv()
...
	🚫 Deadlock	...
		MPI_Ssend()
		MPI_Recv()
		...

...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...
	○ No Deadlock	...
		MPI_Buffer_attach()
		MPI_Bsend()
		MPI_Recv()
		...

...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...
	○ No Deadlock	...
		MPI_Ssend()
		MPI_Recv()
		...

[A. Snavely]→

Deadlocking, Another Example

- MPI_Send can respond in eager or rendezvous mode
- Example, on a certain machine running MPICH v1.2.1:

PROCESS 0

```
...
MPI_Send()
MPI_Recv()
...
```

Deadlock

Data size > 127999 bytes

Data size < 128000 bytes

No Deadlock

PROCESS 1

```
...
MPI_Send()
MPI_Recv()
...
```

[A. Snavely]→

Avoiding Deadlocking

- Easy way to eliminate deadlock is to pair MPI_Ssend and MPI_Recv operations the right way:

PROCESS 0

```
...
MPI_Ssend()
MPI_Recv()
...
```

No Deadlock

PROCESS 1

```
...
MPI_Recv()
MPI_Ssend()
...
```

- Conclusion: understand how the implementation works and what its pitfalls/limitations are

[A. Snavely]→

Example

- Always succeeds, even if no buffering is done

```
if(rank==0)
{
        MPI_Send(...);
        MPI_Recv(...);
}
else if(rank==1)
{
        MPI_Recv(...);
        MPI_Send(...);
}
```

Example

- Will always deadlock, no matter the buffering mode

```
if(rank==0)
{
       MPI_Recv(...);
       MPI_Send(...);
}
else if(rank==1)
{
       MPI_Recv(...);
       MPI_Send(...);
}
```

Example

- Only succeeds if sufficient buffering is present -- unsafe!

```
if(rank==0)
{
        MPI_Send(...);
        MPI_Recv(...);
}
else if(rank==1)
{
        MPI_Send(...);
        MPI_Recv(...);
}
```

More on the Buffered Send

- Relies on the existence of a buffer, which is set up through a call

  ```
  int MPI_Buffer_attach(void* buffer, int size);
  ```

- It is a local operation. It does not depend on the occurrence of a matching receive in order to complete

- If a send operation is started and no matching receive is posted, the outgoing message is buffered to allow the send call to complete

- Return from an `MPI_Bsend` does not guarantee the message was sent

- Message may remain in the buffer until a matching receive is posted. `MPI_Buffer_Detach()` will block until all messages are received

The Buffered Send

[Cntd.]

- Make sure you have enough buffer space available. An error occurs if the message must be buffered and there is there is not enough buffer space.

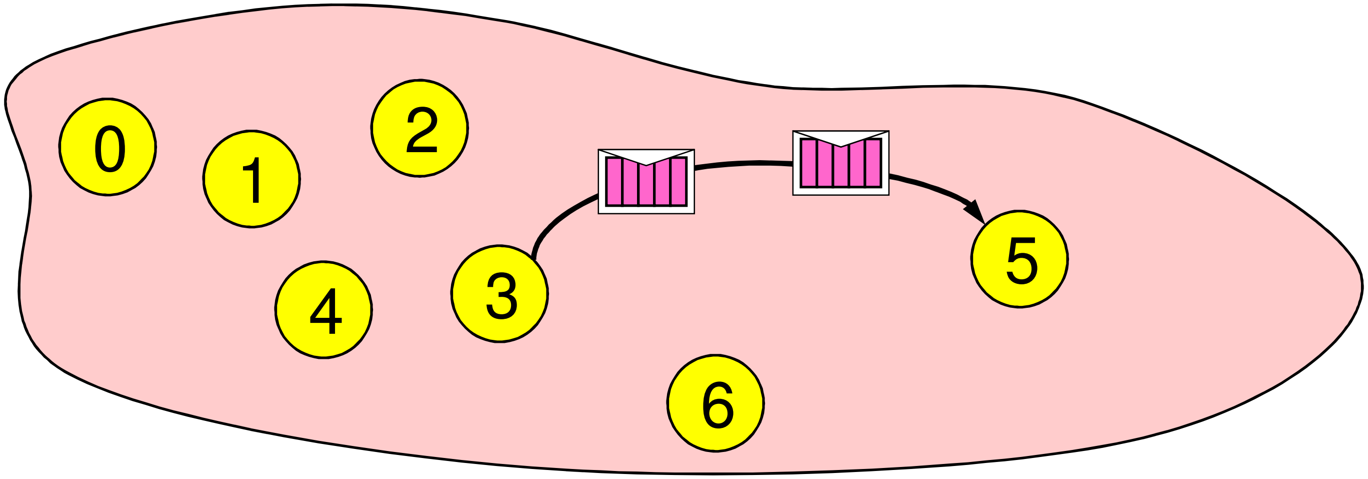
- The amount of buffer space needed to be safe depends on the expected peak of pending messages. The sum of the sizes of all of the pending messages at that point plus (MPI_BSEND_OVERHEAD*number_of_messages) should be sufficient.

- The `MPI_Buffer_attach` subroutine provides MPI a buffer in the user's memory. This buffer is used only by messages sent in buffered mode, and only one buffer is attached to a task at any time.

- `MPI_Bsend` adds overhead because it requires an extra memory-to-memory copy of the outgoing data.

Message Order Preservation

- Rule for messages on the same connection, i.e., same communicator, source, and destination rank:
  - **Messages do not overtake each other**
  - True even for non-synchronous sends



- If both receives match both messages, then the order is preserved

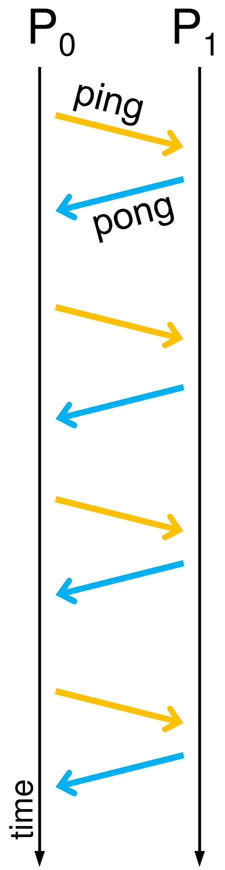
[ICHEC]→

A Word on Assignment 11

- Write a program according to the time-line diagram:
  - process 0 sends a message to process 1 (ping)
  - after receiving this message, process 1 sends a message back to process 0 (pong)
- Repeat this ping-pong with a loop of length 50
- Add timing calls before and after the loop:
- For timing purposes, you might want to use

```
double MPI_Wtime();
```

- MPI_Wtime returns a wall-clock time in seconds
- At process 0, print out the transfer time in seconds
  - Might want to use a log scale



[ICHEC]→

More on Timing

[Useful, assignment 10 and 11]

```
int main()
{
   double starttime, endtime;
   starttime = MPI_Wtime();
    ....  stuff to be timed  ...
   endtime   = MPI_Wtime();
   printf("That took %f seconds\n", endtime - starttime);
   return 0;
}
```

- Resolution is typically 1E-3 seconds
- Time of different processes might actually be synchronized, controlled by the variable `MPI_WTIME_IS_GLOBAL`

More on Timing

[Useful, assignment 10 and 11; Cntd.]

- Latency = transfer time for zero length messages
- Bandwidth = message size (in bytes) / transfer time

- Message transfer time and bandwidth change based on the nature of the MPI send operation
  - Standard send (MPI_Send)
  - Synchronous send (MPI_Ssend)
  - Buffered send
  - Etc.