CME 194 Introduction to MPI

Inchoare Commercio & Rebus Iniuriam

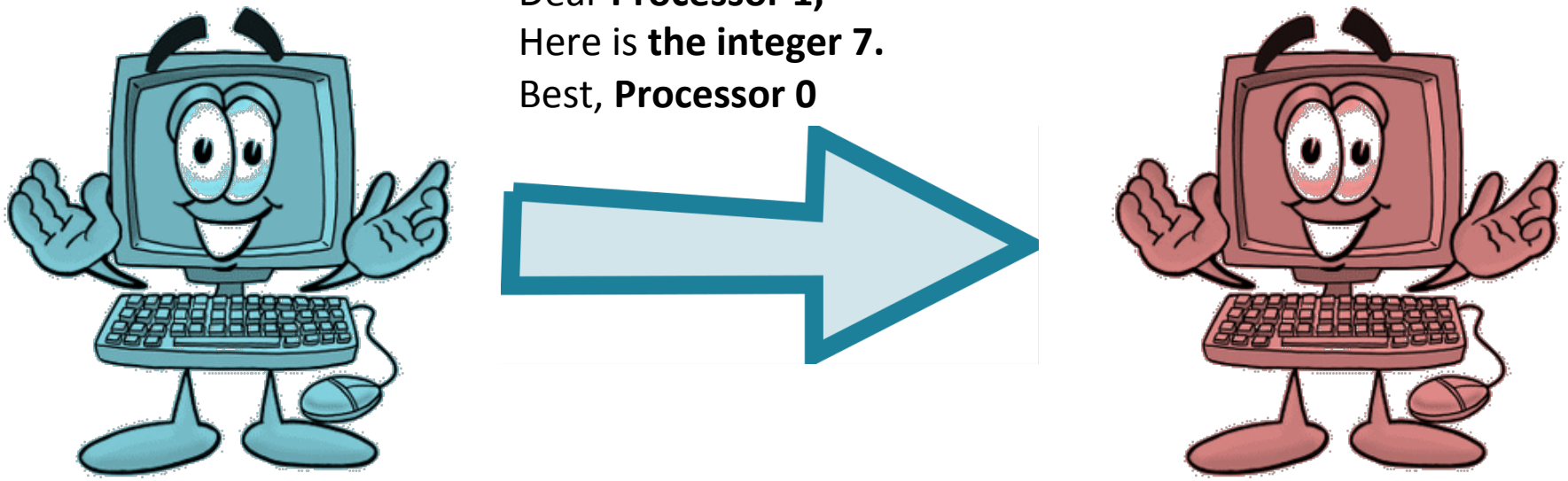
http://cme194.stanford.edu

Point to Point Communication

- **Last class:** Saw the Basics
  - Solve problems with ideal parallelism
    (massively parallel problems)
- **Bad News:** Sadly world isn't perfect
  - Processors need to communicate
- **Good news:** Distributed algorithms are interesting
  - We can attack more interesting problems

Simplest Example

- Two processes
- One process **sends** an integer to the other process.



Send

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

- **buf** Initial address of send buffer.
- **count** Number of elements send (nonnegative integer).
- **datatype** Datatype of each send buffer element (handle).
- **dest** Rank of destination (integer).
- **tag** Message tag (integer).
- **comm** Communicator (handle).

Receive

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

- **buf** Initial address of send buffer.
- **count** Number of elements send (nonnegative integer).
- **datatype** Datatype of each send buffer element (handle).
- **source** Rank of source (integer) (or use **MPI_ANY_SOURCE**)
- **tag** Message tag (integer) (or use **MPI_ANY_TAG**)
- **comm** Communicator (handle)
- **status** Structure containing with message information

(or use **MPI_STATUS_IGNORE**)

Basic MPI Datatypes

MPI datatype	C datatype
MPI_CHAR	char
MPI_SHORT	short int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

Learn to send structs and classes: Lecture 4

Some Code

```
13 std::size_t the_number_seven=1;
14 if( rank == 0){
15           the_number_seven=7;
16           MPI_Send( &the_number_seven, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
17           std::cout << "0: has returned from the send." << std::endl;
18 } else {
19         MPI_Status status;
20         std::cout << rank << ": the_number_seven is at first "
21                   << the_number_seven << std::endl;
22         MPI_Recv( &the_number_seven, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
23         std::cout << "1: has returned from the receive." << std::endl;
24         std::cout << rank << ": the_number_seven is now equal to "
25                   << the_number_seven << std::endl;
26 }
```

Details

- Each **Send** must be **matched** with a **Recv**.
- Messages are **delivered in** the **order sent**.
- Unmatched sends/receives **may** result in **deadlock**

More on MPI_Status

Contains:

- **rank of sender** useful with MPI_ANY_SOURCE
- **tag of message** useful with MPI_ANY_TAG
- **error code**
- **message length** since we can receive larger data than we need

Given a valid Status object we may do:

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

To extract the length.

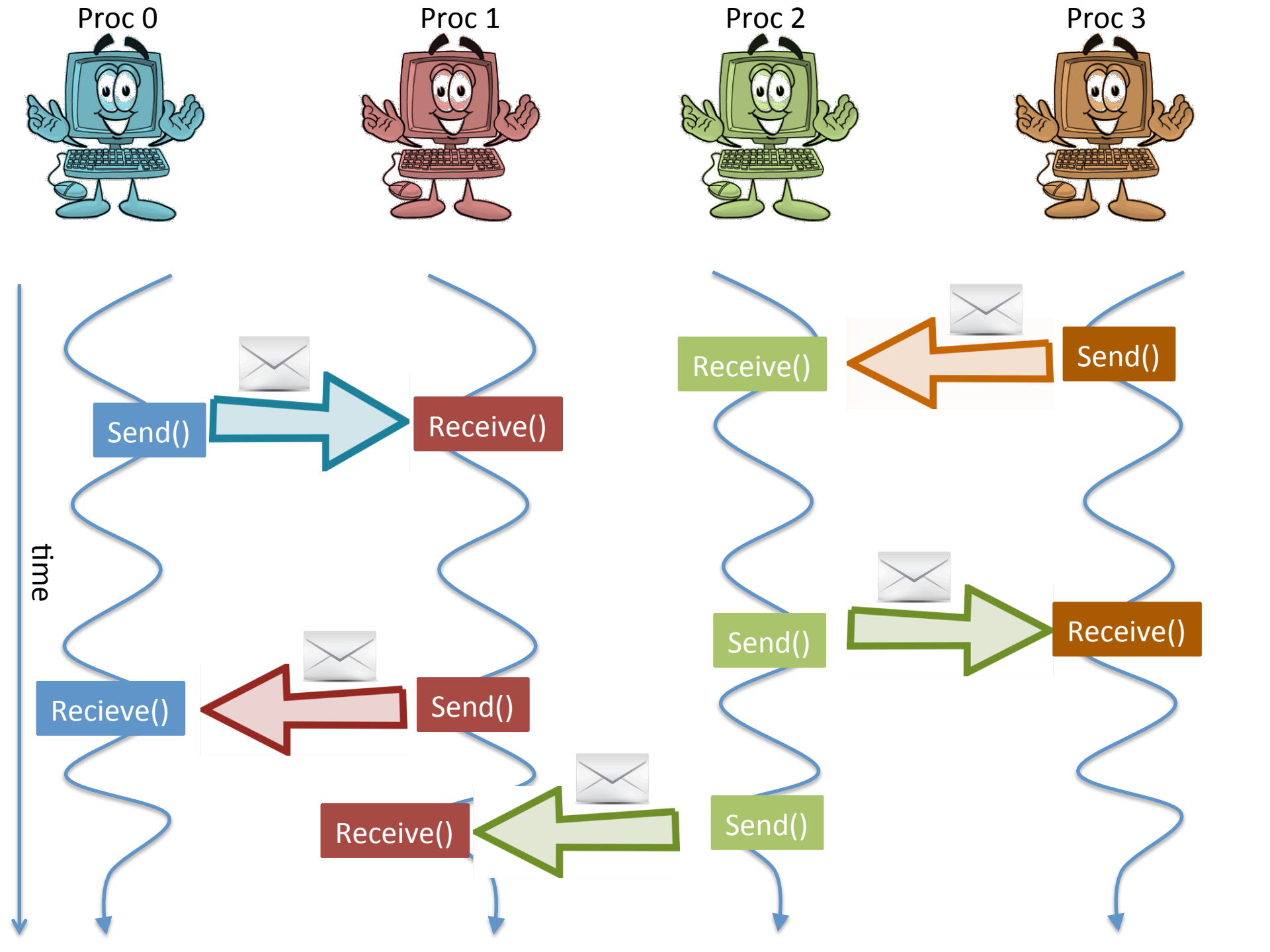
Wait! Better to get this data *before* message?

MPI_Probe

MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status* status)

- Check for incoming messages without actual receipt of them.
- Useful for receiving messages of dynamic **type** and **size**

```
1
2  MPI_Status status;
3  // Probe for an incoming message from process zero
4  MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
5
6  // When probe returns, the status object has the size and other
7  // attributes of the incoming message. Get the size of the message
8  MPI_Get_count(&status, MPI_INT, &number_amount);
9
10 // Allocate a buffer just big enough to hold the incoming numbers
11 int* number_buf = (int*)malloc(sizeof(int) * number_amount);
12
13 // Now receive the message with the allocated buffer
14 MPI_Recv(number_buf, number_amount, MPI_INT, 0, 0, MPI_COMM_WORLD,
15         MPI_STATUS_IGNORE);
~
```


Proc 0
Proc 1
Proc 2
Proc 3
time
Send()
Receive()
Send()
Receive()
Send()
Receive()
Recieve()
Send()
Receive()
Send()

Application: Scalable Sorting

Input: Array of **n** integers divided across **p** machines.
Problem: Sort the total array quickly with constant extra space

- Parallel Merge/quick-sort require too much communication

Solution: Bitonic Sort!

Bitonic sequence:

- Is first increasing and then decreasing

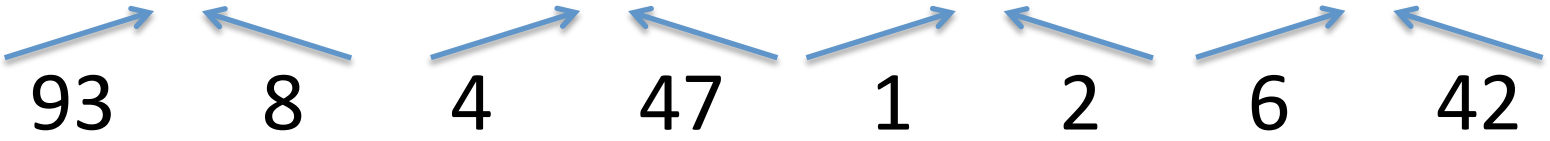
Idea:

First: Turn sequence into a bitonic one

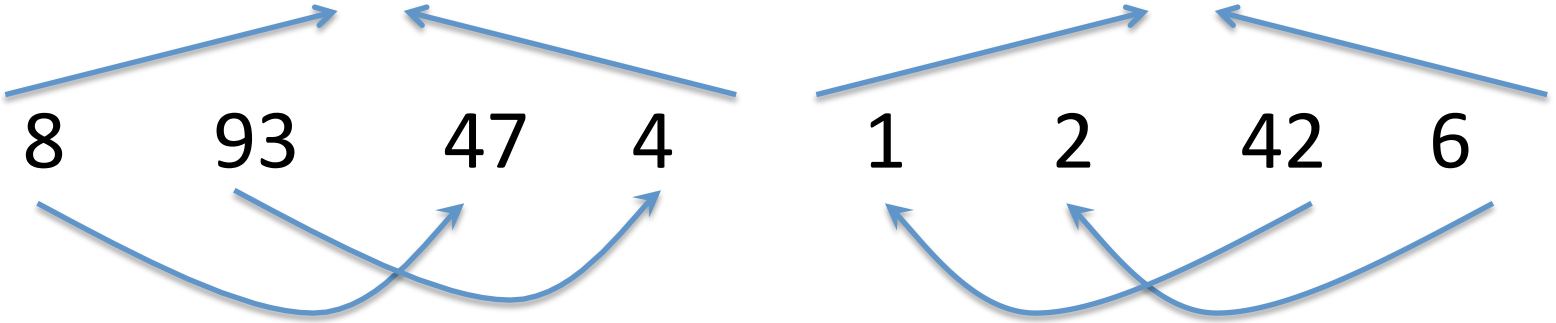
Second: Then do reverse “quicksort”-like recursion

Application: Bitonic Sorting

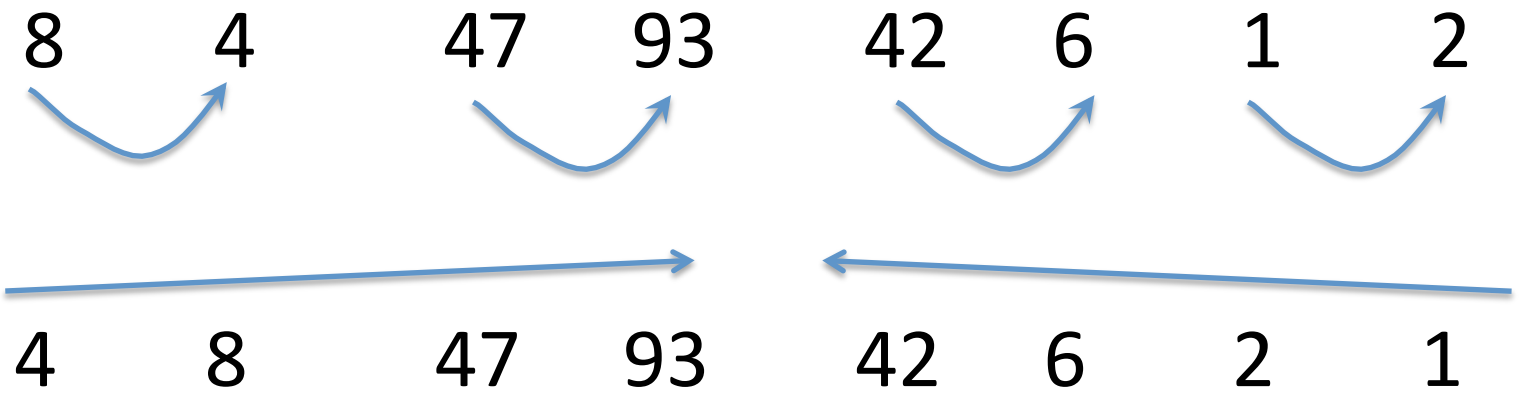
Step 0: Look, lots of small bitonic sequences!



Step 1: Want bigger bitonic sequences like this:

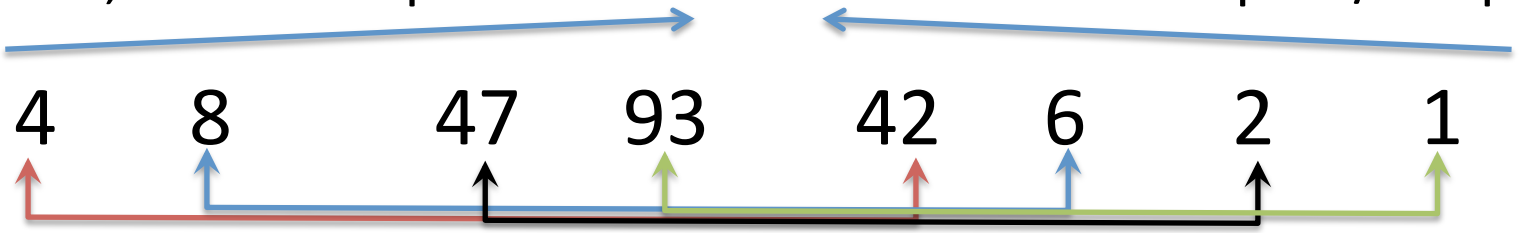


Step 2**: - Repeat -** Compare & Recurse



Application: Bitonic Sorting

Great, bitonic sequence? Now we do shifted compare/swap



Recurse, left and right half, now partially ordered.

1 2 4 6 8 42 47 93

Psuedocode: Bitonic Sorting

Let $s = \langle s_0, s_1, \ldots, s_{n-1} \rangle$ such that

$s_0 \leq s_1 \leq \ldots s_{n/2-1}$ and $s_{n/2} \geq s_{n/2+1} \leq \ldots \leq s_{n-1}$

Notice that these two lists are bitonic and $l_1 \leq l_2$

$l_1 = \langle \min(s_0, s_{n/2}), \min(s_1, s_{n/2+1}), \ldots \min(s_{n/2-1}, s_{n-1}) \rangle$

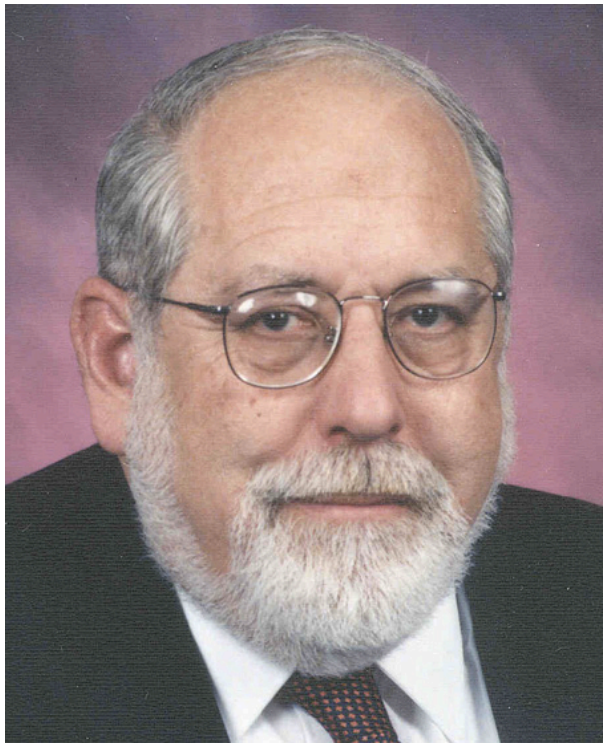
$l_2 = \langle \max(s_0, s_{n/2}), \max(s_1, s_{n/2+1}), \ldots \max(s_{n/2-1}, s_{n-1}) \rangle$

- Recurse on these lists.

$O(n \log^2(n))$ serial time complexity

- When implemented in parallel not much extra space needed

Application: Bitonic Sorting



Ken Batcher

Discovered Bitonic Sort in 1968

Homework #1

Last Problem: Implement Bitonic Sort

Ken Batcher's Hint:

Try to use a special point to point primitive

Ken Batcher's Hint

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

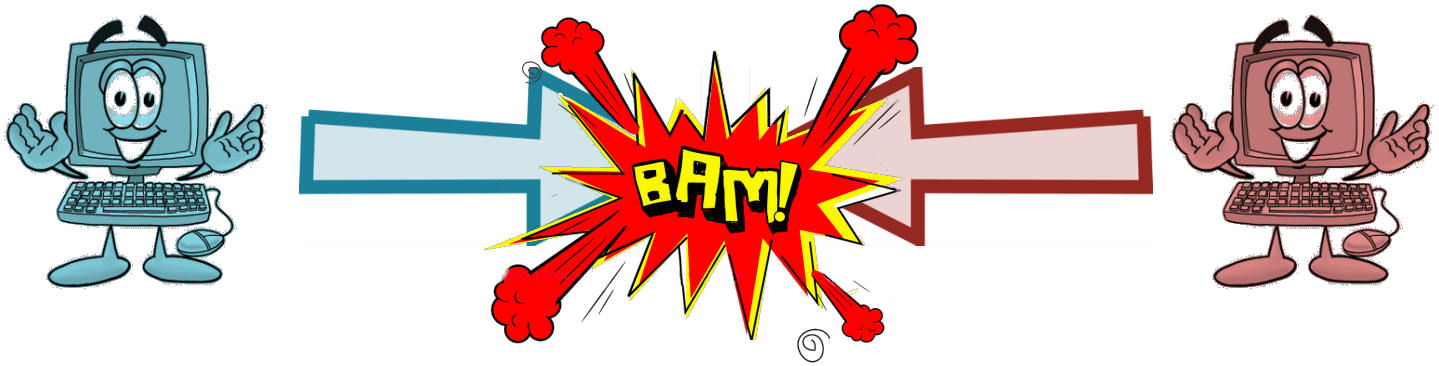
- Simultaneous send and receive
- There is a version called **MPI_sendrecv_replace** which reuses one of the buffers.

Communication Modes

- There are different ways to send messages in MPI.
- They are:
  - Blocking vs. nonblocking,
  - Synchronous vs. asynchronous
  - Buffered vs. unbuffered
- We will see all of this in Lecture 5.

Deadlocks

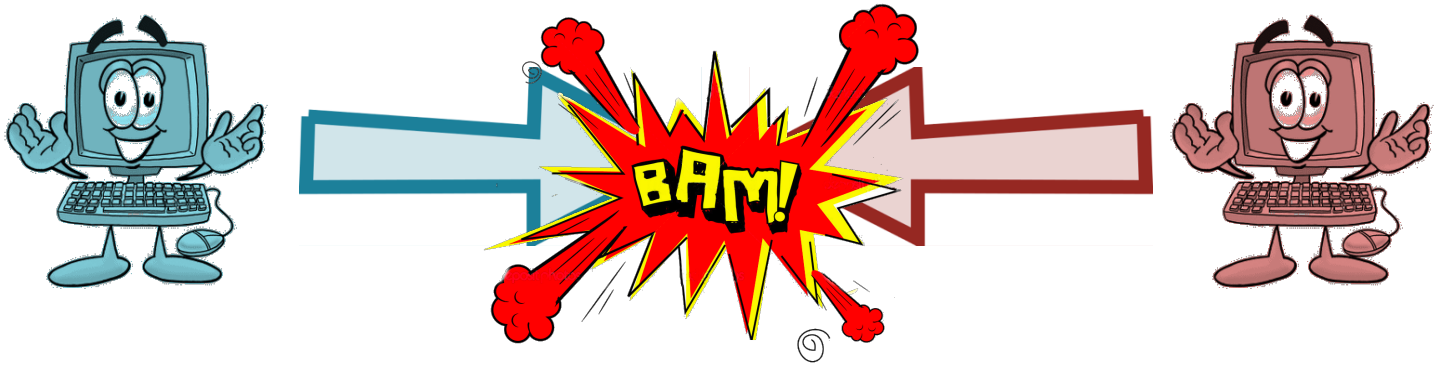
Suppose we had this situation:




```
 6 if (rank == 0) {
 7       MPI_Send(..., 1, tag, MPI_COMM_WORLD);
 8       MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);
 9 } else if (rank == 1) {
10       MPI_Send(..., 0, tag, MPI_COMM_WORLD);
11       MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);
12 }
~
```

Deadlocks

Suppose we had this situation:



"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

~ Kansas Legislature Early 20th century.

Dealing with Bugs

- Quick review of serial debugging
  - Examine your source code for errors
  - Add enough debugging (print) statements to your output
  - Use a symbolic debugger (gdb)
  - Find bugs, fix, and repeat.
  - Testing, early and often.

"It can be virtually impossible to predict the behavior of an erroneous program."

Classic Deadlocks & Common Failures

- Trying to receive data before sending in an exchange
- Trying to receive data with an unmatched send
- Incorrect send/receive parameters
- Code which depends on implementation not standard
  - Different systems means different errors
- Code works with **n** cores but not **n+1**

Words of wisdom

“Many (if not most) parallel program bugs have nothing to do with the fact that the program is parallel.
Infact, most bugs are caused by the same mistakes that Cause serial program bugs.”

Parallel Debugging

- Use all the techniques of serial debugging!
  - Examine your source code for errors
  - Add enough debugging (print) statements to your output
  - Use a symbolic debugger
  - Find bugs, fix, and repeat.

Wait, how does one use a debugger with many processes?
…. On different computers?
Nonfree debuggers exist.

Allinea DDT

- Two popular options are:
  - Totalview by Roguewave
  - and Allinea DDT.
- You all may evaluate the latter on ICME's MPI cluster.
- **http://www.allinea.com/products/downloads**
- Download the remote clients for Mac OS X of Windows.
- I believe Linux users may use the regular DDT as a remote client.

Recommendation: Use SSH Public Key for Cluster Access

MPI Error Handling

MPI_Errhandler_set(MPI_Comm comm MPI_Errhandler handler)

- Error Handlers are functions to which control

is transferred to in the event of an error condition

- MPI Standard Defines two:
  - MPI_ERRORS_ARE_FATAL → program crashes on error
  - MPI_ERRORS_RETURN → error codes returned
- The latter is good to use with

MPI_Error_string(int errorcode, char* string, int* resultlen)

MPI Error Handling

```
2
3  char error_message[ MPI_MAX_ERROR_STRING];
4
5  int message_length;
6
7  error_code = MPI_Send( ... );
8
9  if ( error_code != MPI_SUCCESS){
10         MPI_Error_string( error_code, error_message, &me
   ssage_length);
11         std::cerr << "Error in send: " << error_message
   << std::endl;
12         MPI_Abort( MPI_COMM_WORLD, -1);
13 }
```

Homework 1 & Stuff to debug

- Homework 1 is now available
- I've put some programs which need fixing up as well.