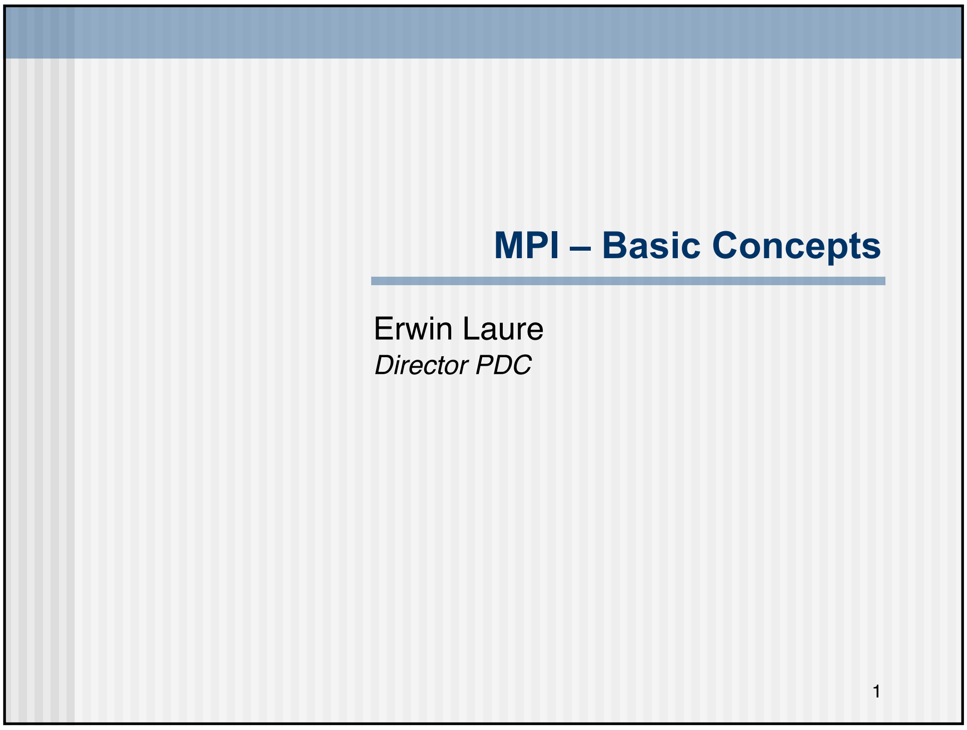
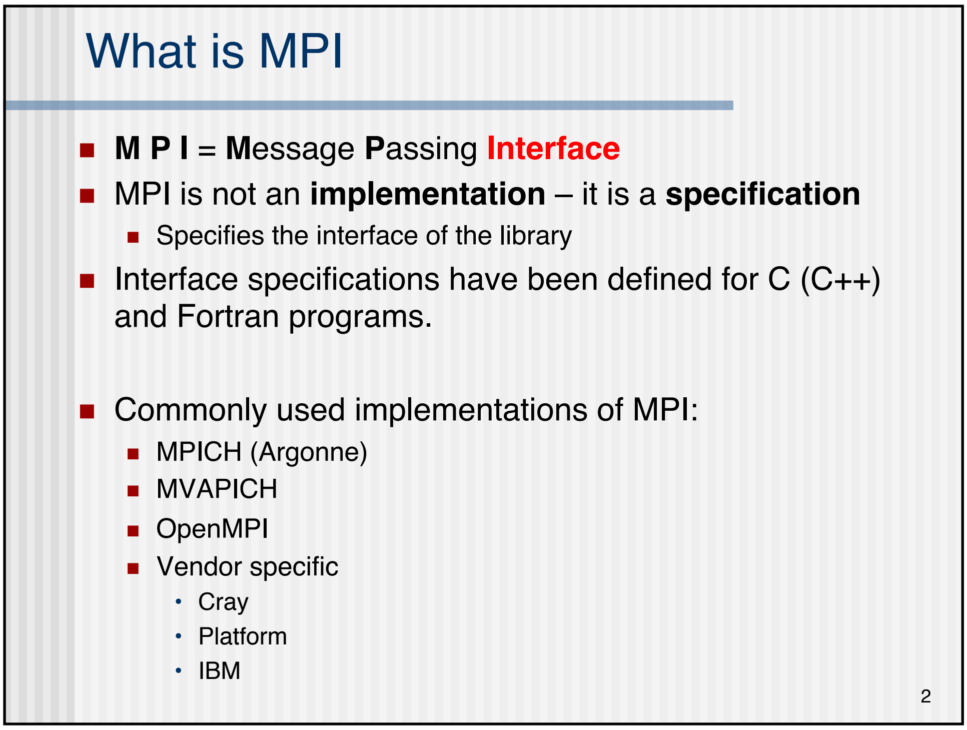


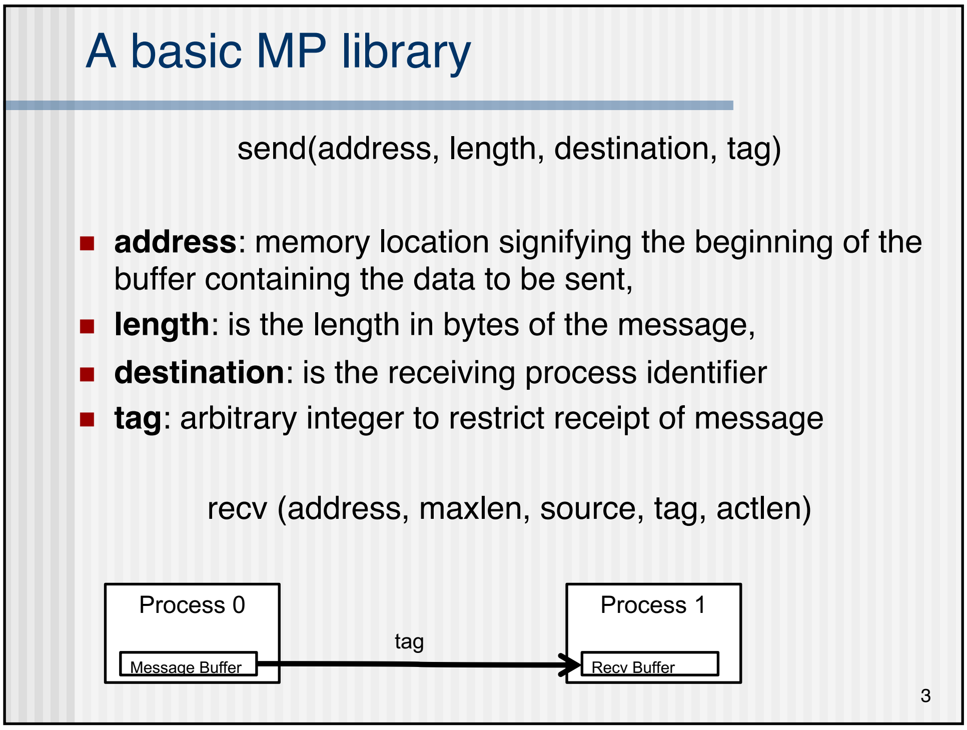
MPI – Basic Concepts

Erwin Laure
Director PDC



What is MPI

- **M P I** = **M**essage **P**assing **Interface**
- MPI is not an **implementation** – it is a **specification**
  - Specifies the interface of the library
- Interface specifications have been defined for C (C++) and Fortran programs.
- Commonly used implementations of MPI:
  - MPICH (Argonne)
  - MVAPICH
  - OpenMPI
  - Vendor specific
    - Cray
    - Platform
    - IBM



A basic MP library

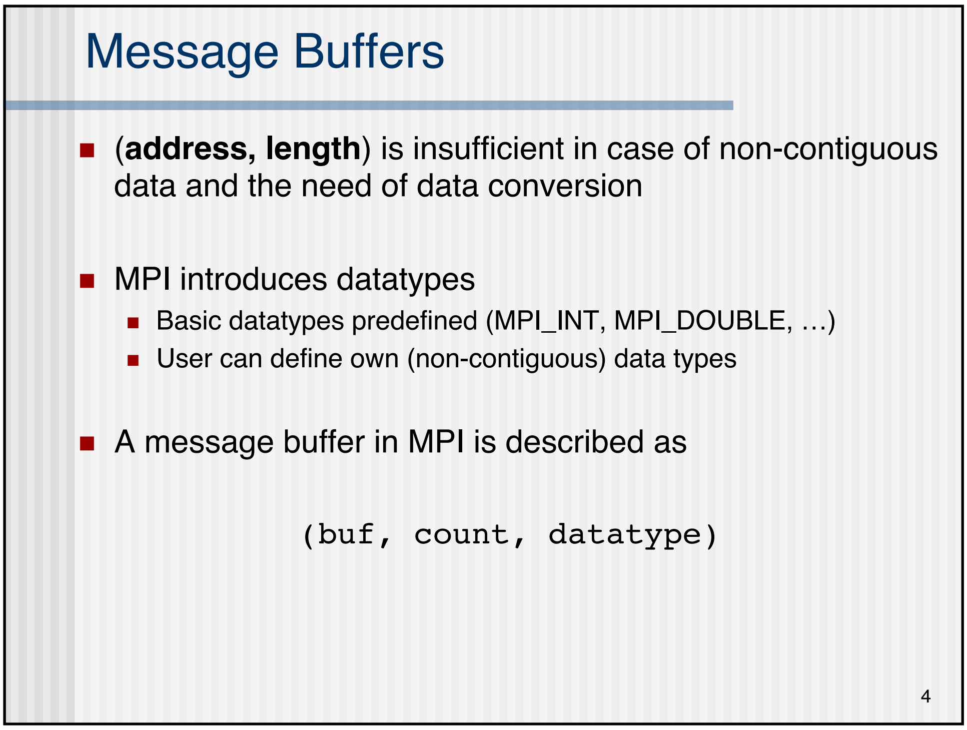
send(address, length, destination, tag)

- **address**: memory location signifying the beginning of the buffer containing the data to be sent,
- **length**: is the length in bytes of the message,
- **destination**: is the receiving process identifier
- **tag**: arbitrary integer to restrict receipt of message

recv (address, maxlen, source, tag, actlen)

Process 0
Message Buffer
tag
Process 1
Recv Buffer

3



Message Buffers

- (**address, length**) is insufficient in case of non-contiguous data and the need of data conversion
- MPI introduces datatypes
  - Basic datatypes predefined (MPI_INT, MPI_DOUBLE, …)
  - User can define own (non-contiguous) data types
- A message buffer in MPI is described as

```
(buf, count, datatype)
```

4

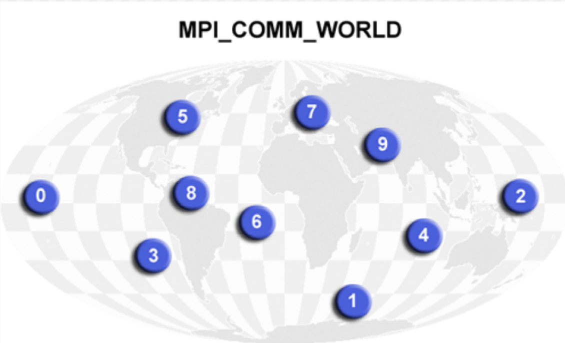
MPI Basic Datatypes (Fortran)

MPI Datatype	Fortran Datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE_PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Note: the names of the MPI C datatypes are slightly different

Processes and Communicators

- Processes belong to **groups**
- Processes within a group are identified with their **rank**
  - A group of n processes has ranks 0 … n-1
- MPI uses objects called **communicators** and groups to define which collection of processes may communicate with each other
  - `MPI_COMM_WORLD` is the default communicator covering all of the original MPI processes

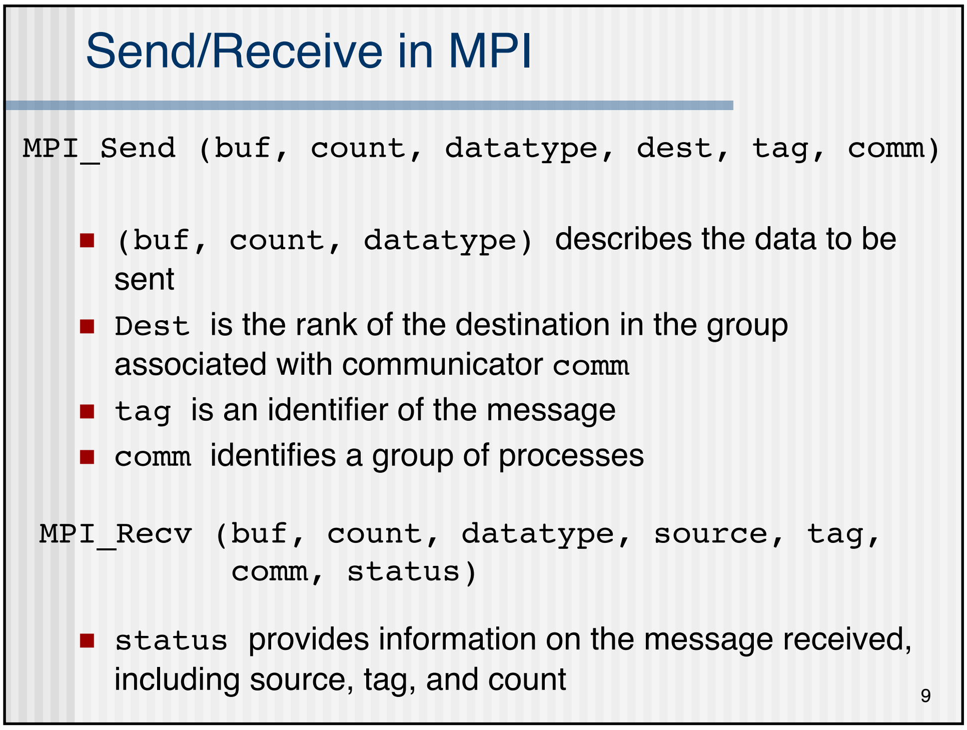


Why Communicators?

- How to chose safe (unique) tags when writing a library? I.e. how to avoid a message being picked up by the wrong receiver?
- Collective operations (broadcast, reductions) can be easily defined over subgroups by using communicators

Note: Processes vs. Processors

- MPI defines **processes**, it does not specify how these processes are mapped to physical **processors/cores**
- The mapping of processes to processors/cores is done at program start and dependent on the startup mechanism available on a certain resource – more about that later on.
- In principle, a MPI process does not necessarily correspond to an OS process – in practice it very often does.



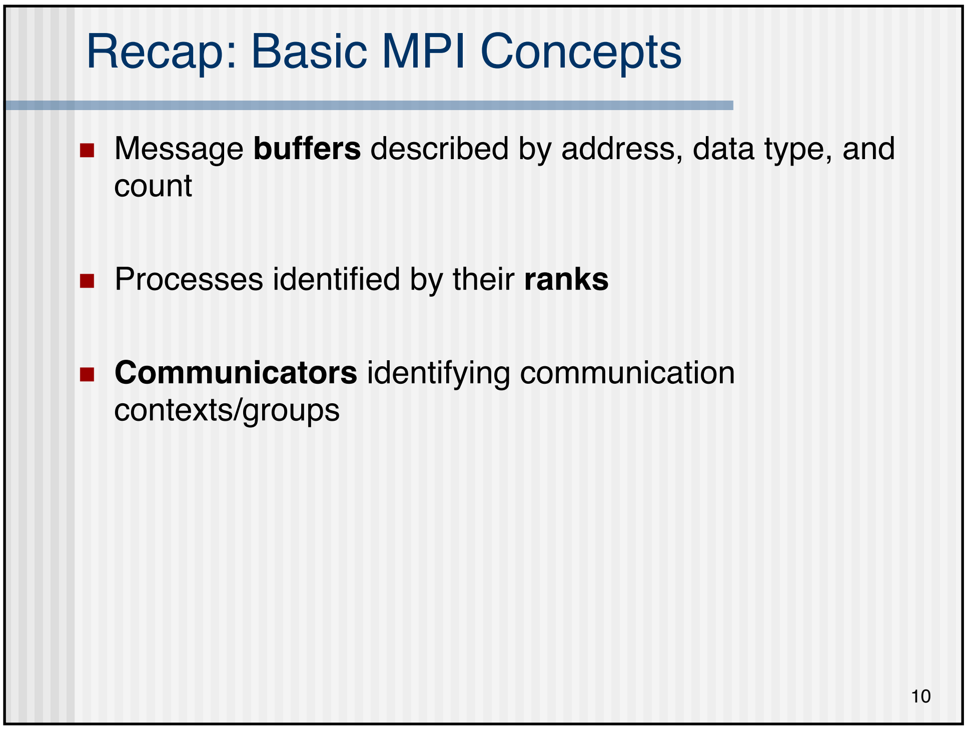
Send/Receive in MPI

```
MPI_Send (buf, count, datatype, dest, tag, comm)
```

- `(buf, count, datatype)` describes the data to be sent
- `Dest` is the rank of the destination in the group associated with communicator `comm`
- `tag` is an identifier of the message
- `comm` identifies a group of processes

```
 MPI_Recv (buf, count, datatype, source, tag,
            comm, status)
```

- `status` provides information on the message received, including source, tag, and count



Recap: Basic MPI Concepts

- Message **buffers** described by address, data type, and count
- Processes identified by their **ranks**
- **Communicators** identifying communication contexts/groups

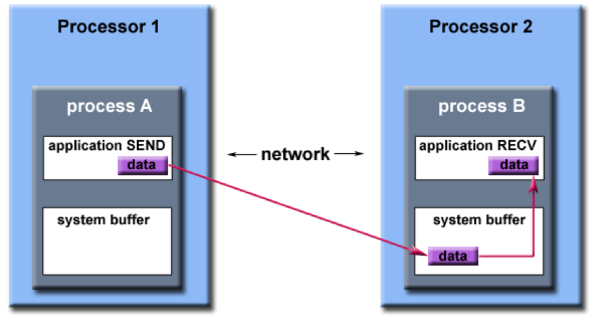
MPI has over 300 functions …

- How many years do I have to study before I can use it?
- In fact, you will hardly ever use most of the MPI functions
- 6 functions are sufficient for simple programs:
  - `MPI_Init` – to initialize the MPI environment
  - `MPI_Comm_Size` – to know the number of processes
  - `MPI_Comm_Rank` – to know the rank of the calling process
  - `MPI_Send` – to send a message
  - `MPI_Recv` – to receive a message
  - `MPI_Finalize` – to exit in a clean way

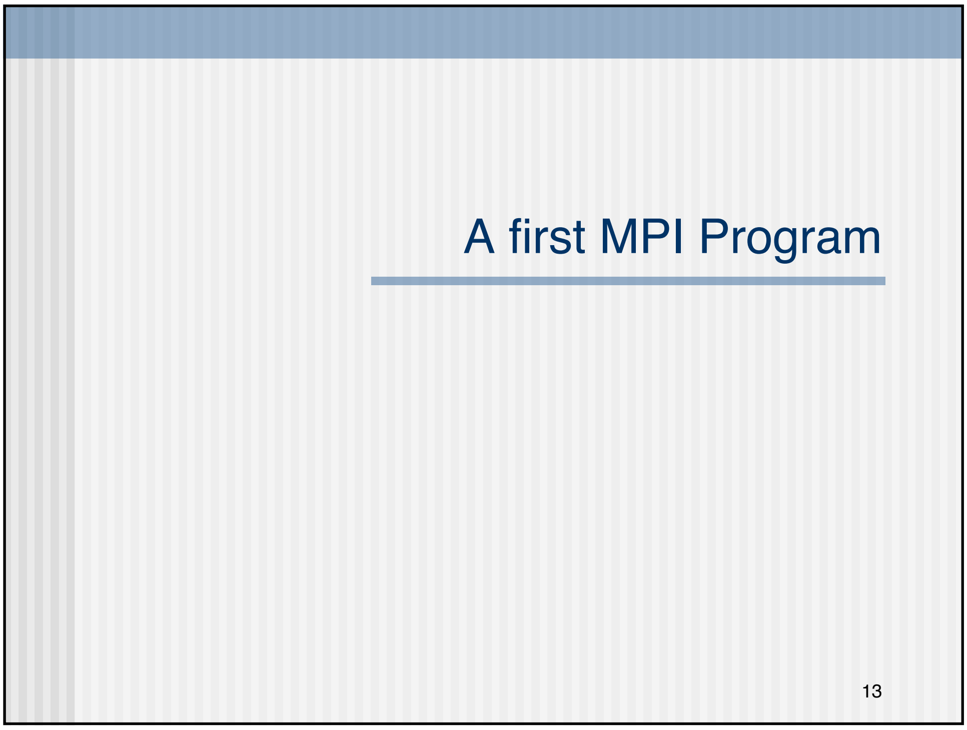
11

What is not specified

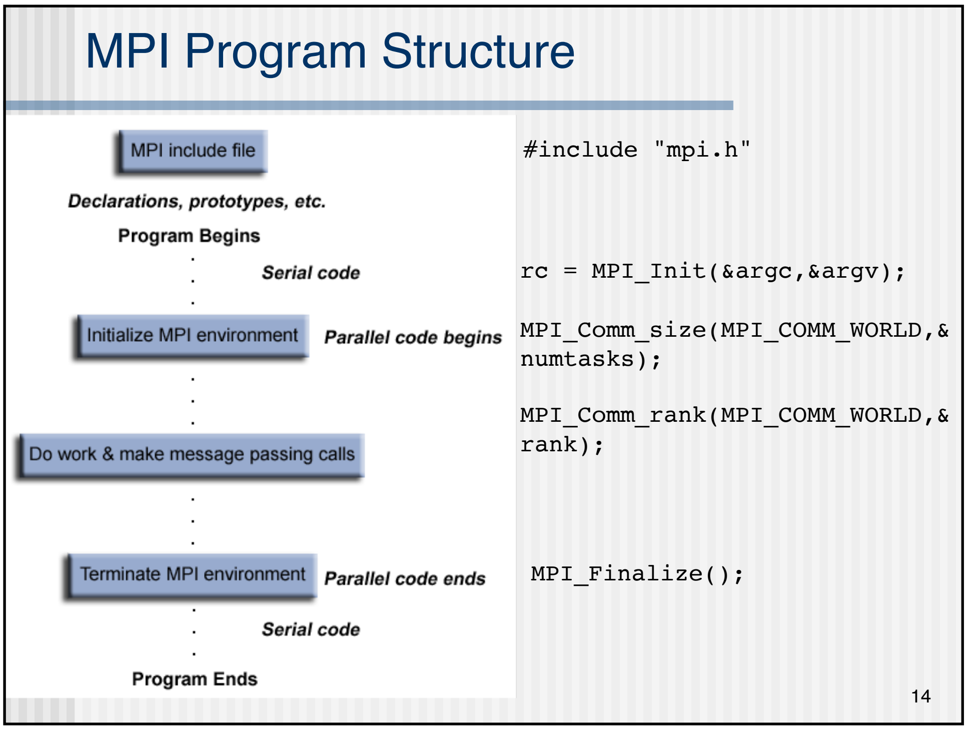
- Certain aspects are not specified in the MPI standard but left as implementation detail:
  - Process startup (how to start an MPI program)
    - All what happens before `MPI_Init` is executed
  - Richer error codes are allowed
  - Message buffering



Path of a message buffered at the receiving process



A first MPI Program



MPI Program Structure

MPI include file

Declarations, prototypes, etc.

Program Begins

Serial code

Initialize MPI environment

Parallel code begins

Do work & make message passing calls

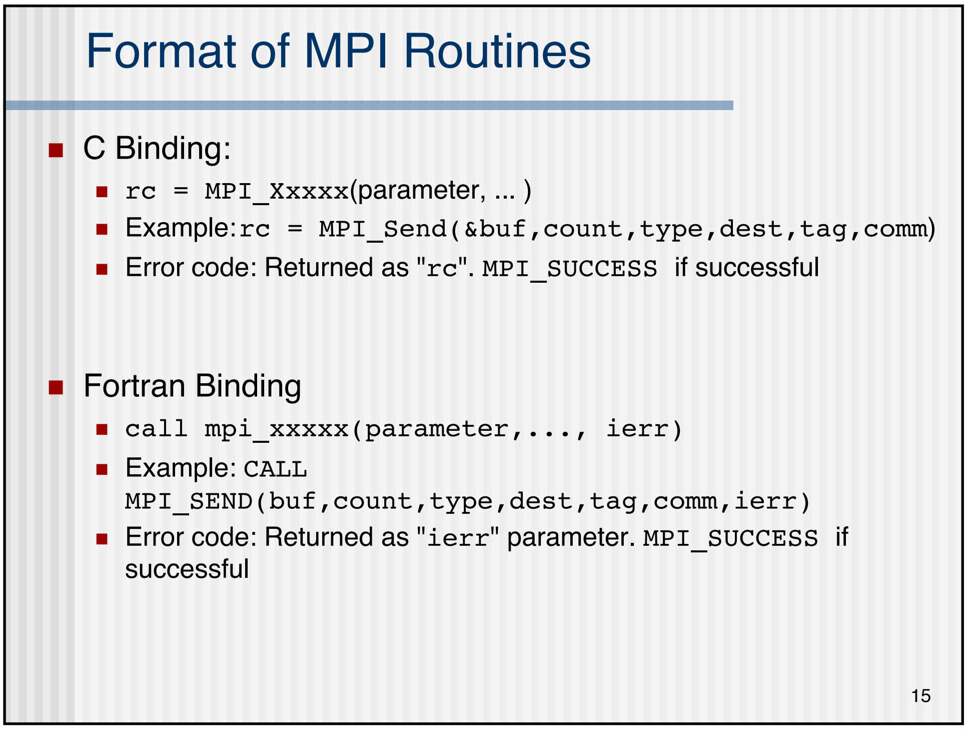
Terminate MPI environment

Parallel code ends

Serial code

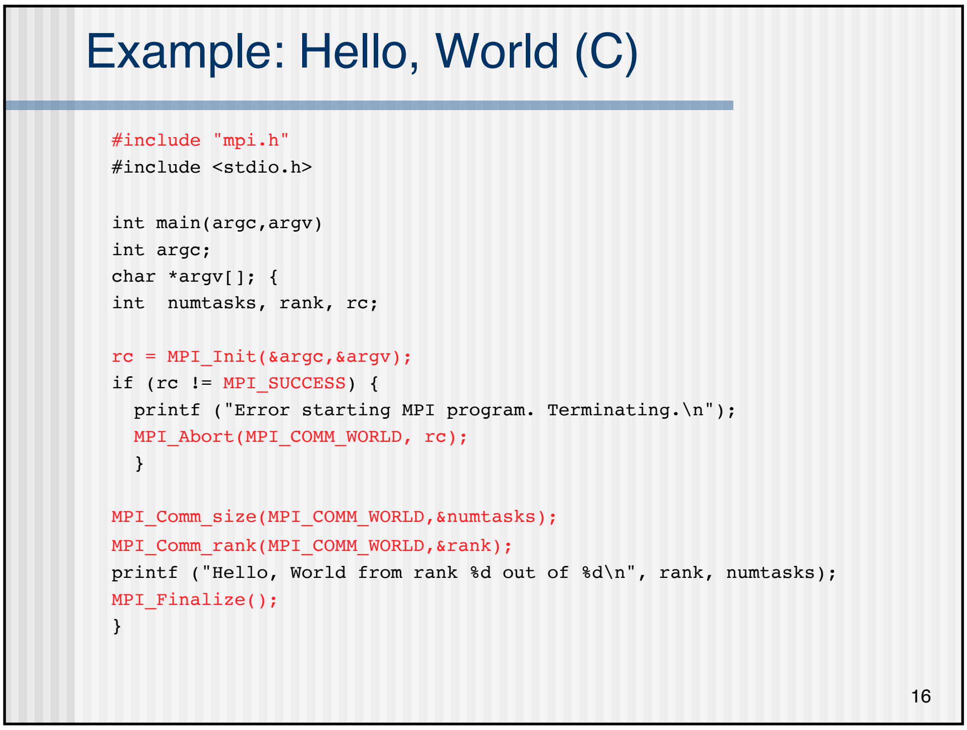
Program Ends

```
#include "mpi.h"

rc = MPI_Init(&argc,&argv);

MPI_Comm_size(MPI_COMM_WORLD,&
numtasks);

MPI_Comm_rank(MPI_COMM_WORLD,&
rank);

 MPI_Finalize();
```



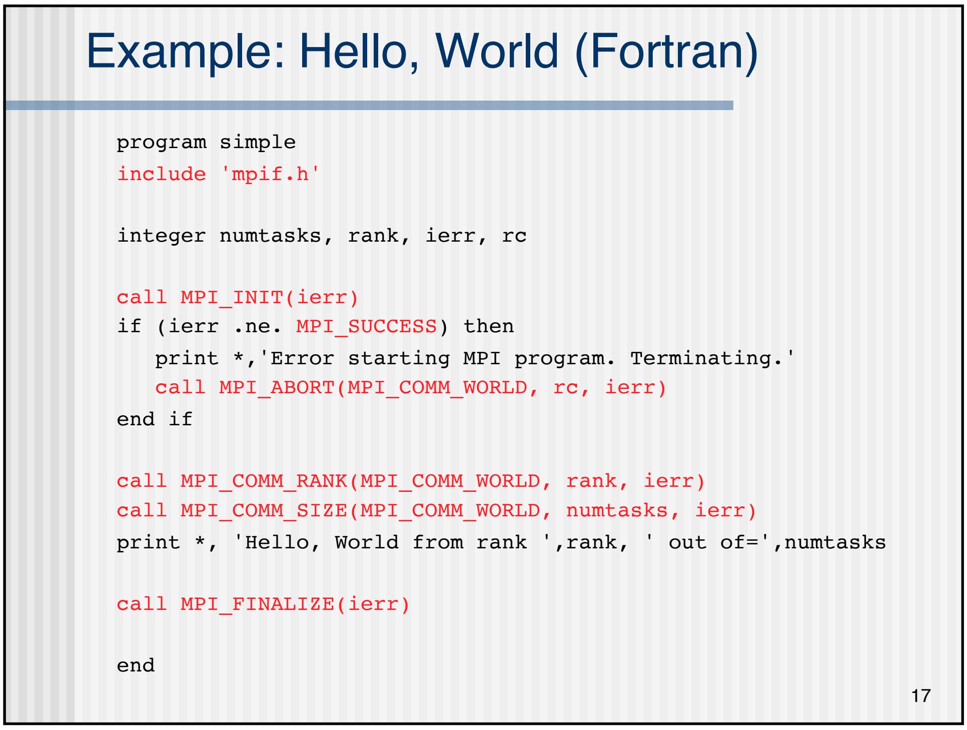
Format of MPI Routines

- C Binding:
  - `rc = MPI_Xxxxx`(parameter, ...)
  - Example:`rc = MPI_Send(&buf,count,type,dest,tag,comm)`
  - Error code: Returned as "`rc`". `MPI_SUCCESS` if successful
- Fortran Binding
  - `call mpi_xxxxx(parameter,..., ierr)`
  - Example: `CALL MPI_SEND(buf,count,type,dest,tag,comm,ierr)`
  - Error code: Returned as "`ierr`" parameter. `MPI_SUCCESS` if successful



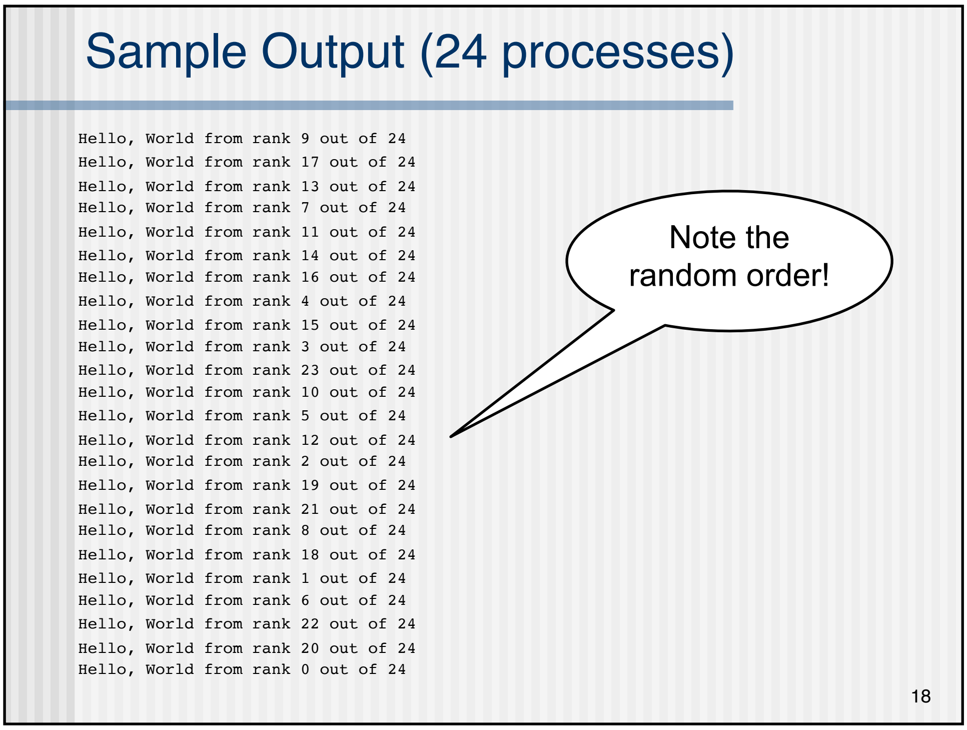
Example: Hello, World (C)

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

int main(argc,argv)
int argc;
char *argv[]; {
int  numtasks, rank, rc;

rc = MPI_Init(&argc,&argv);
if (rc != MPI_SUCCESS) {
  printf ("Error starting MPI program. Terminating.\n");
  MPI_Abort(MPI_COMM_WORLD, rc);
  }

MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
printf ("Hello, World from rank %d out of %d\n", rank, numtasks);
MPI_Finalize();
}
```



Example: Hello, World (Fortran)

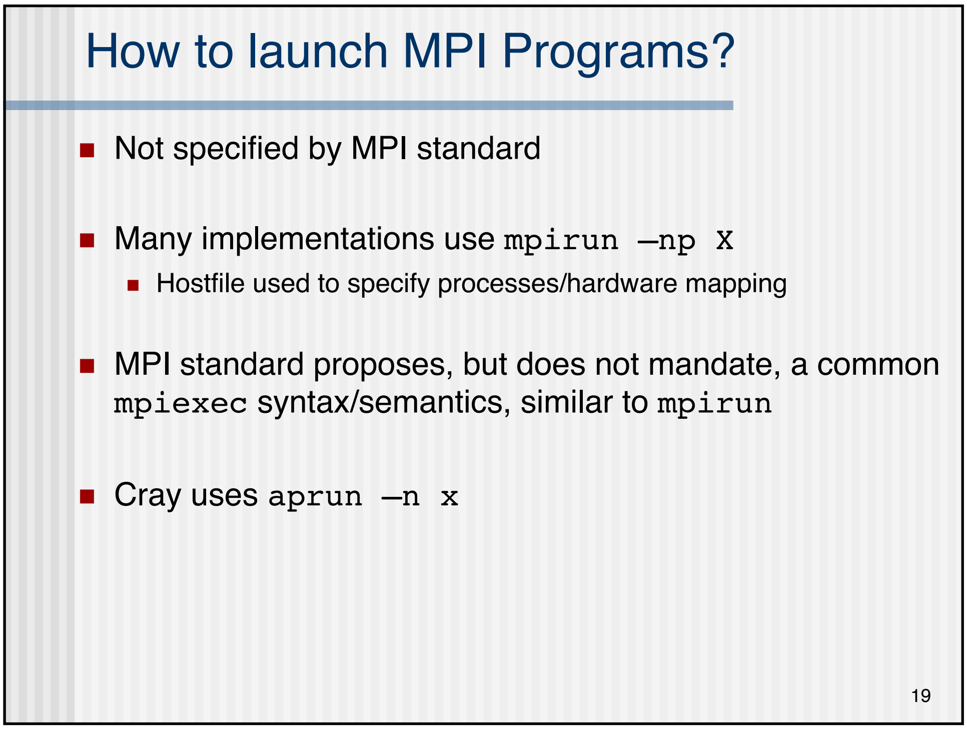
```
program simple
include 'mpif.h'

integer numtasks, rank, ierr, rc

call MPI_INIT(ierr)
if (ierr .ne. MPI_SUCCESS) then
   print *,'Error starting MPI program. Terminating.'
   call MPI_ABORT(MPI_COMM_WORLD, rc, ierr)
end if

call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
print *, 'Hello, World from rank ',rank, ' out of=',numtasks

call MPI_FINALIZE(ierr)

end
```



Sample Output (24 processes)

```
Hello, World from rank 9 out of 24
Hello, World from rank 17 out of 24
Hello, World from rank 13 out of 24
Hello, World from rank 7 out of 24
Hello, World from rank 11 out of 24
Hello, World from rank 14 out of 24
Hello, World from rank 16 out of 24
Hello, World from rank 4 out of 24
Hello, World from rank 15 out of 24
Hello, World from rank 3 out of 24
Hello, World from rank 23 out of 24
Hello, World from rank 10 out of 24
Hello, World from rank 5 out of 24
Hello, World from rank 12 out of 24
Hello, World from rank 2 out of 24
Hello, World from rank 19 out of 24
Hello, World from rank 21 out of 24
Hello, World from rank 8 out of 24
Hello, World from rank 18 out of 24
Hello, World from rank 1 out of 24
Hello, World from rank 6 out of 24
Hello, World from rank 22 out of 24
Hello, World from rank 20 out of 24
Hello, World from rank 0 out of 24
```

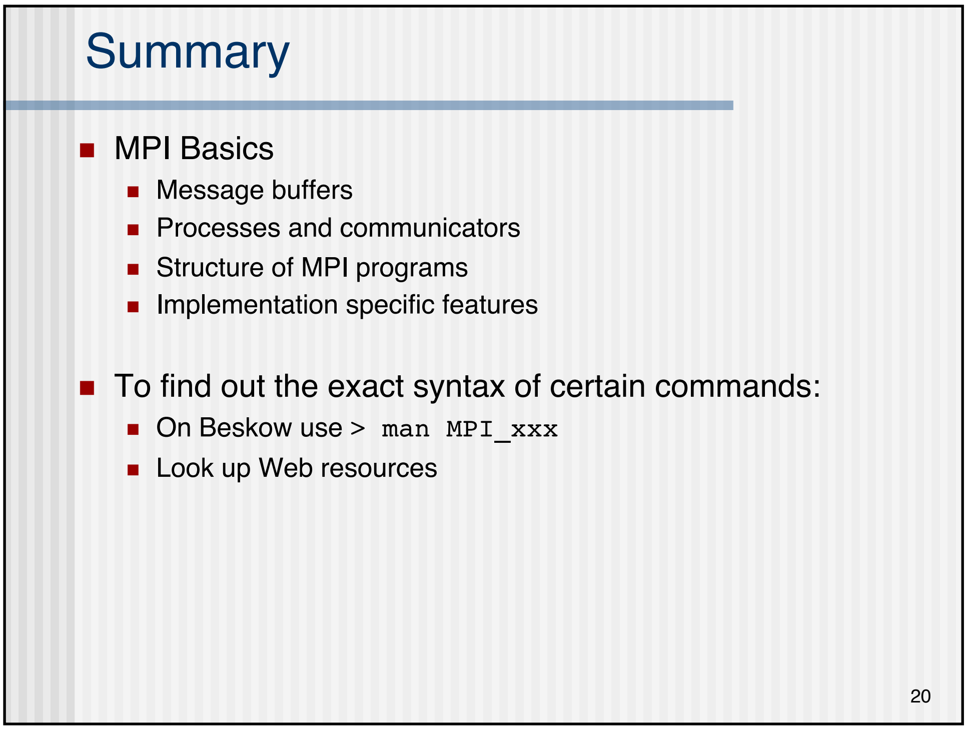
Note the random order!



How to launch MPI Programs?

- Not specified by MPI standard
- Many implementations use `mpirun –np X`
  - Hostfile used to specify processes/hardware mapping
- MPI standard proposes, but does not mandate, a common `mpiexec` syntax/semantics, similar to `mpirun`
- Cray uses `aprun –n x`

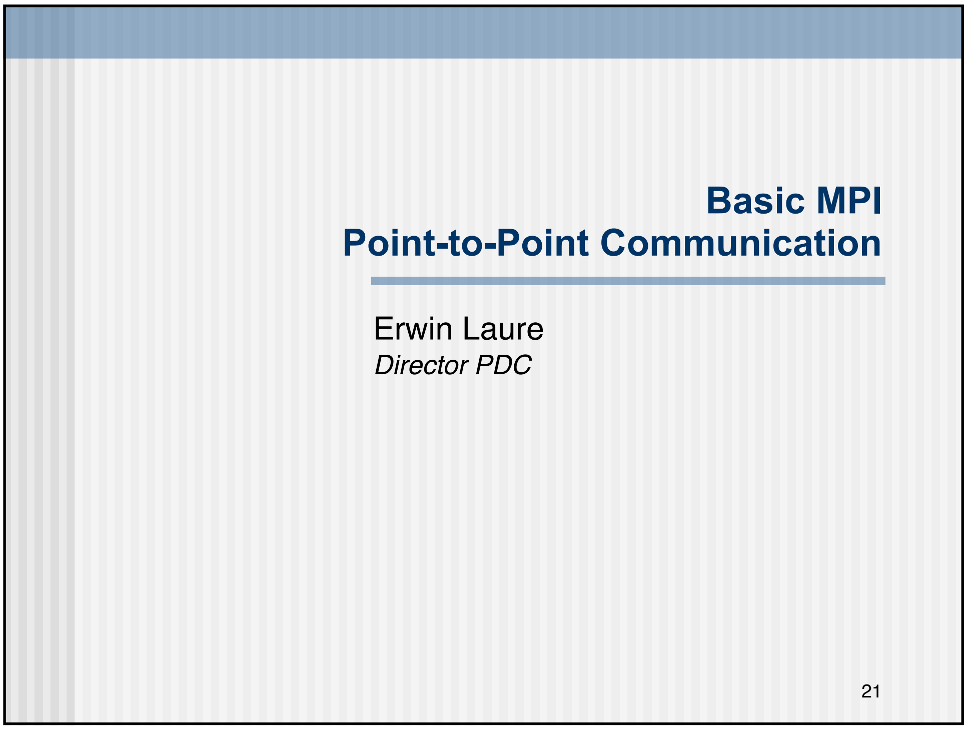
19



Summary

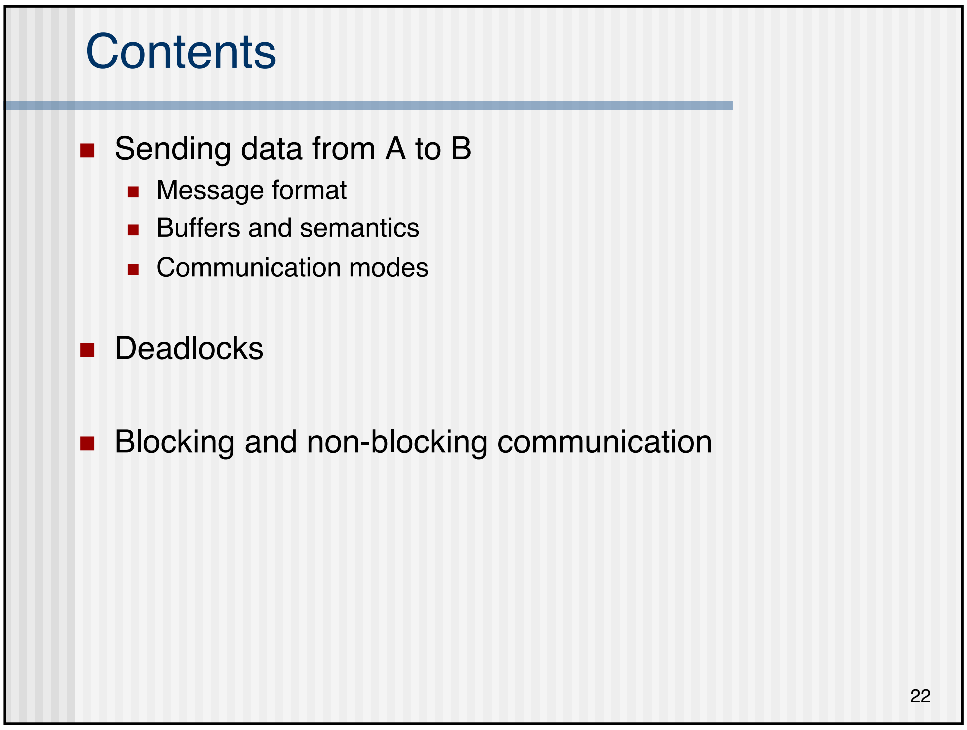
- MPI Basics
  - Message buffers
  - Processes and communicators
  - Structure of MPI programs
  - Implementation specific features
- To find out the exact syntax of certain commands:
  - On Beskow use > `man MPI_xxx`
  - Look up Web resources

20



Basic MPI
Point-to-Point Communication

Erwin Laure
Director PDC

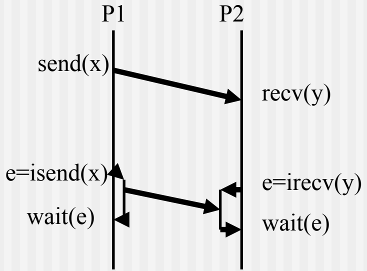


Contents

- Sending data from A to B
  - Message format
  - Buffers and semantics
  - Communication modes
- Deadlocks
- Blocking and non-blocking communication

Sending Data from A to B …

- The basic function of any message passing library
  - Typically a SEND/RECEIVE pair
- Needed when process X needs data from process Y
- Two main incarnations
  - Blocking: stops the program until it is safe to continue
  - Non-blocking: separates communication from computation



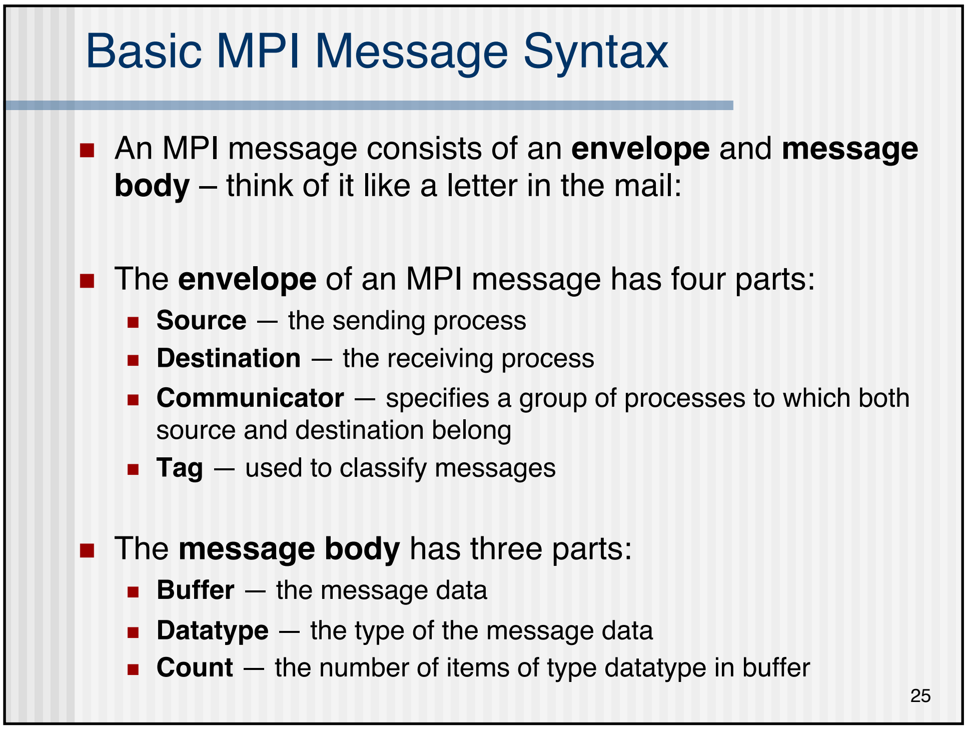
Send/Receive in MPI

```
MPI_Send (buf, count, datatype, dest, tag, comm)
```

- `(buf, count, datatype)` describes the data to be sent
- `Dest` is the rank of the destination in the group associated with communicator `comm`
- `tag` is an identifier of the message
- `comm` identifies a group of processes

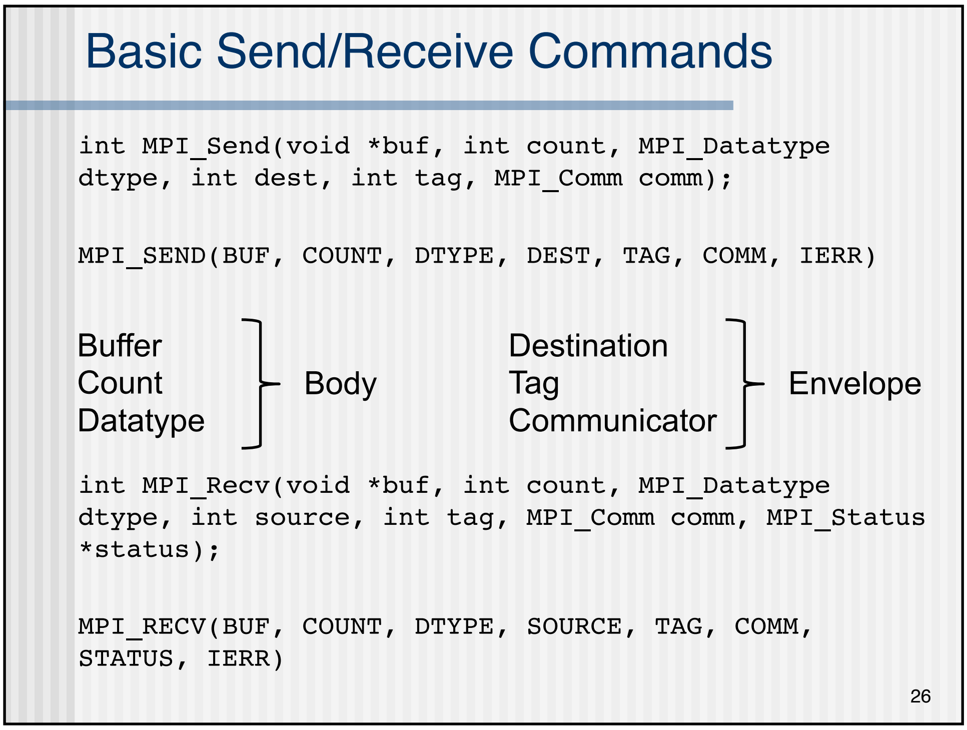
```
MPI_Recv (buf, count, datatype, source, tag,
          comm, status)
```

- `status` provides information on the message received, including source, tag, and count



Basic MPI Message Syntax

- An MPI message consists of an **envelope** and **message body** – think of it like a letter in the mail:
- The **envelope** of an MPI message has four parts:
  - **Source** — the sending process
  - **Destination** — the receiving process
  - **Communicator** — specifies a group of processes to which both source and destination belong
  - **Tag** — used to classify messages
- The **message body** has three parts:
  - **Buffer** — the message data
  - **Datatype** — the type of the message data
  - **Count** — the number of items of type datatype in buffer



Basic Send/Receive Commands

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

MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, IERR)
```

Buffer
Count
Datatype

Body

Destination
Tag
Communicator

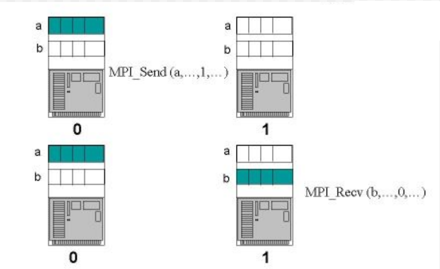
Envelope

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

MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM,
STATUS, IERR)
```

Example

```
double a[100],b[100];

if( myrank == 0 )        /* Send a message */
{
  for (i=0;i<100;++i)
     a[i]=sqrt(i);
  MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
}
else if( myrank == 1 )    /* Receive a message */
  MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
```

What happens if b is replaced with a?



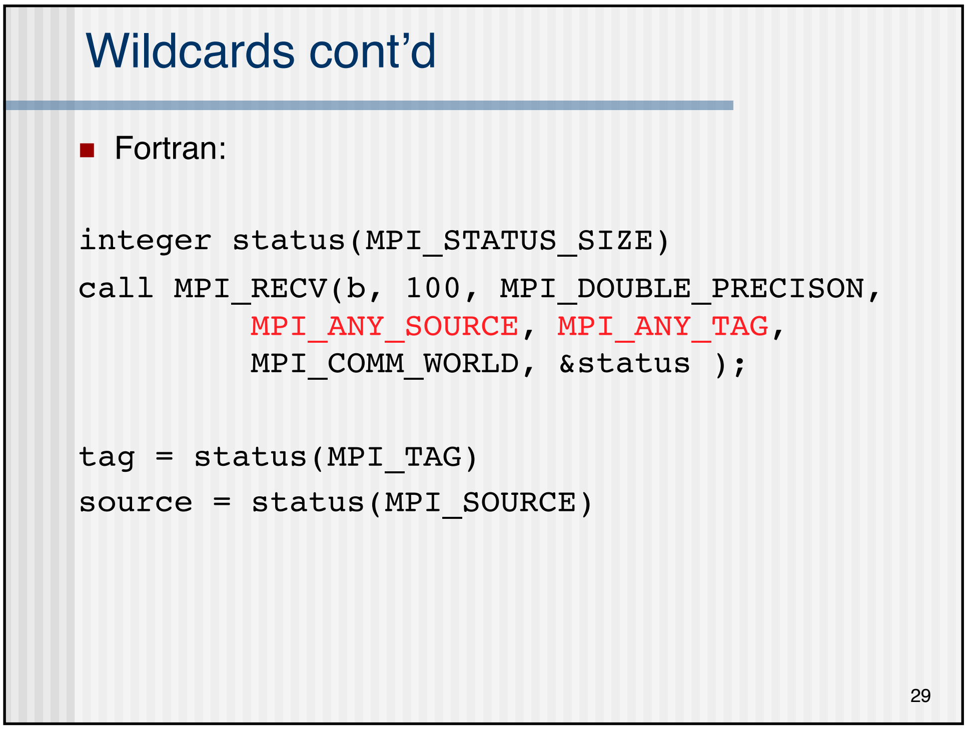
Wildcards

- Instead of specifying everything in the envelope explicitly, wildcards can be used for sender and tag:

  `MPI_ANY_SOURCE` and `MPI_ANY_TAG`
- Actual source and tag are stored in `STATUS` variable

C:

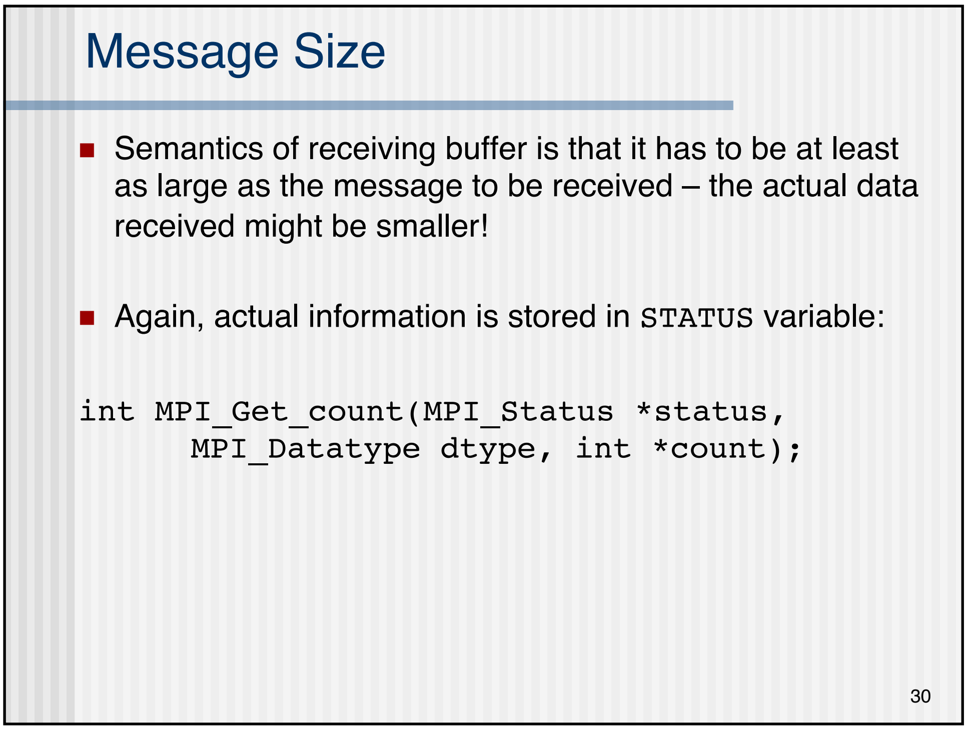
```
MPI_Status status;
MPI_Recv(b, 100, MPI_DOUBLE,
         MPI_ANY_SOURCE, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status );

source = status.MPI_SOURCE;
tag = status.MPI_TAG;
```



Wildcards cont’d

- Fortran:

```
integer status(MPI_STATUS_SIZE)
call MPI_RECV(b, 100, MPI_DOUBLE_PRECISON,
         MPI_ANY_SOURCE, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status );

tag = status(MPI_TAG)
source = status(MPI_SOURCE)
```



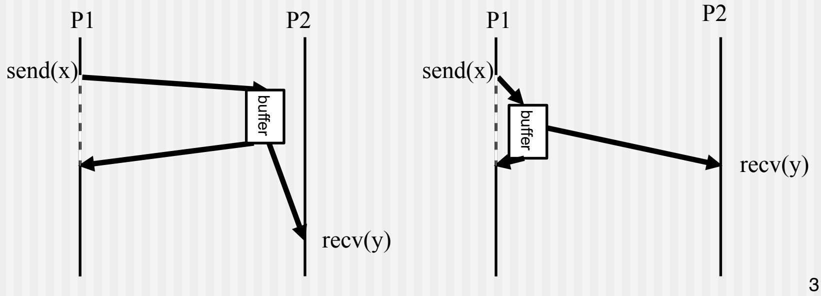
Message Size

- Semantics of receiving buffer is that it has to be at least as large as the message to be received – the actual data received might be smaller!
- Again, actual information is stored in `STATUS` variable:

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

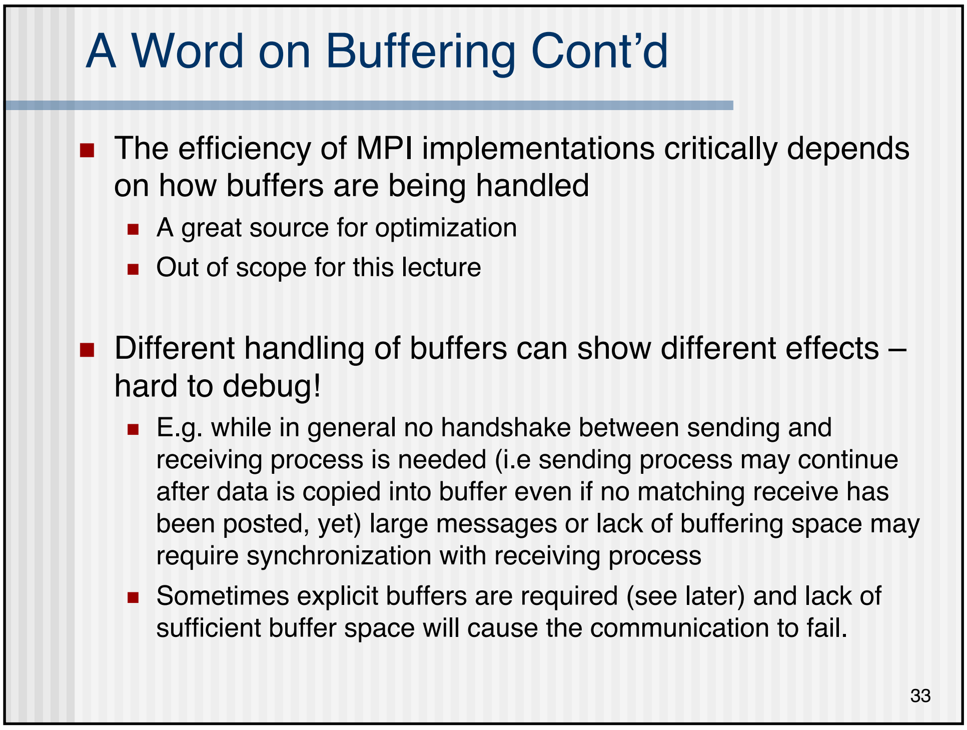
A Word on Buffering

- MPI implementations typically use (**internal**) message buffers
  - Sending process can safely modify the sent data once it is copied into the buffer, irrespectively of status of receiving process
  - Receiving process can buffer incoming messages even if no (user space) receiving buffer is provided, yet
  - Buffers can be on both sides



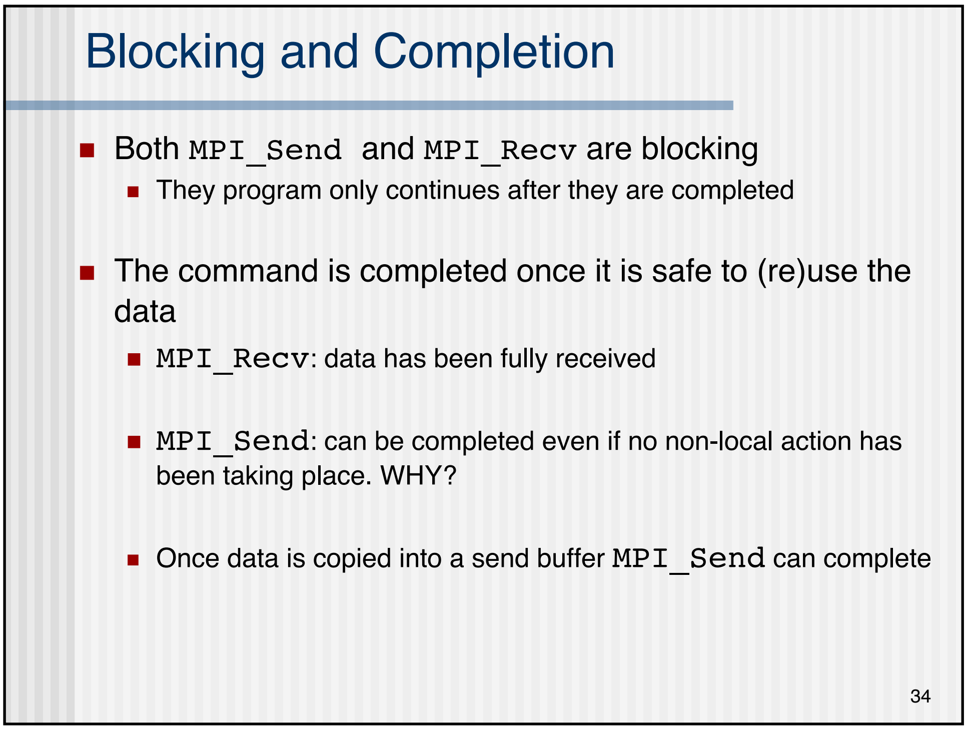
Note

This system buffer is **DIFFERENT** to the message buffer you specify in the `MPI_Send` or `MPI_Recv` calls!



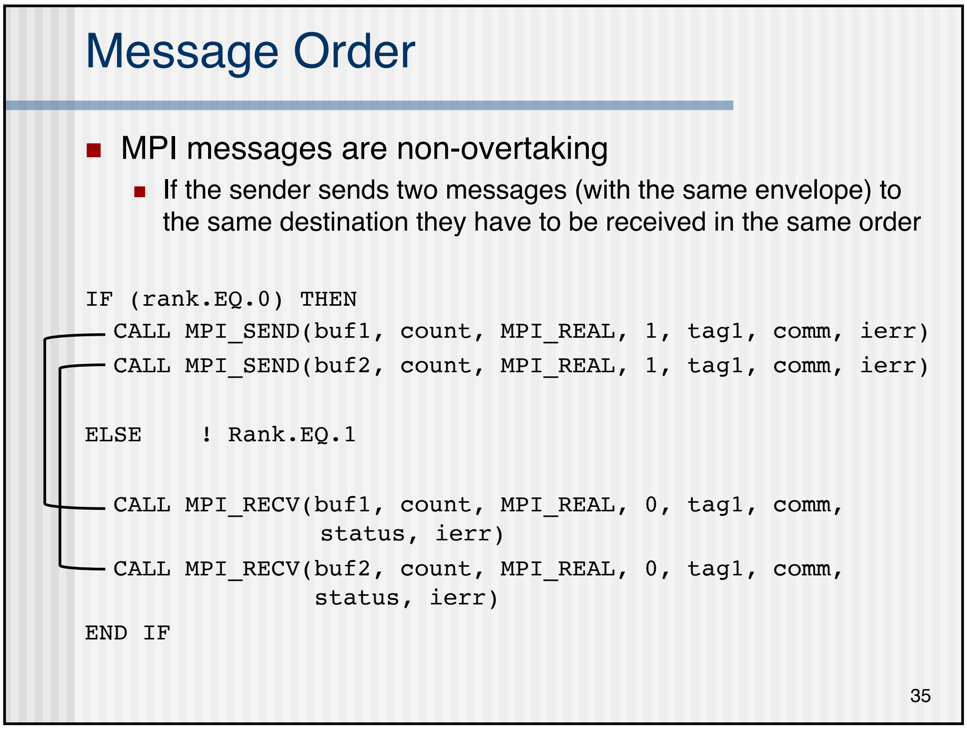
A Word on Buffering Cont'd

- The efficiency of MPI implementations critically depends on how buffers are being handled
  - A great source for optimization
  - Out of scope for this lecture
- Different handling of buffers can show different effects – hard to debug!
  - E.g. while in general no handshake between sending and receiving process is needed (i.e sending process may continue after data is copied into buffer even if no matching receive has been posted, yet) large messages or lack of buffering space may require synchronization with receiving process
  - Sometimes explicit buffers are required (see later) and lack of sufficient buffer space will cause the communication to fail.



Blocking and Completion

- Both `MPI_Send` and `MPI_Recv` are blocking
  - They program only continues after they are completed
- The command is completed once it is safe to (re)use the data
  - `MPI_Recv`: data has been fully received
  - `MPI_Send`: can be completed even if no non-local action has been taking place. WHY?
  - Once data is copied into a send buffer `MPI_Send` can complete

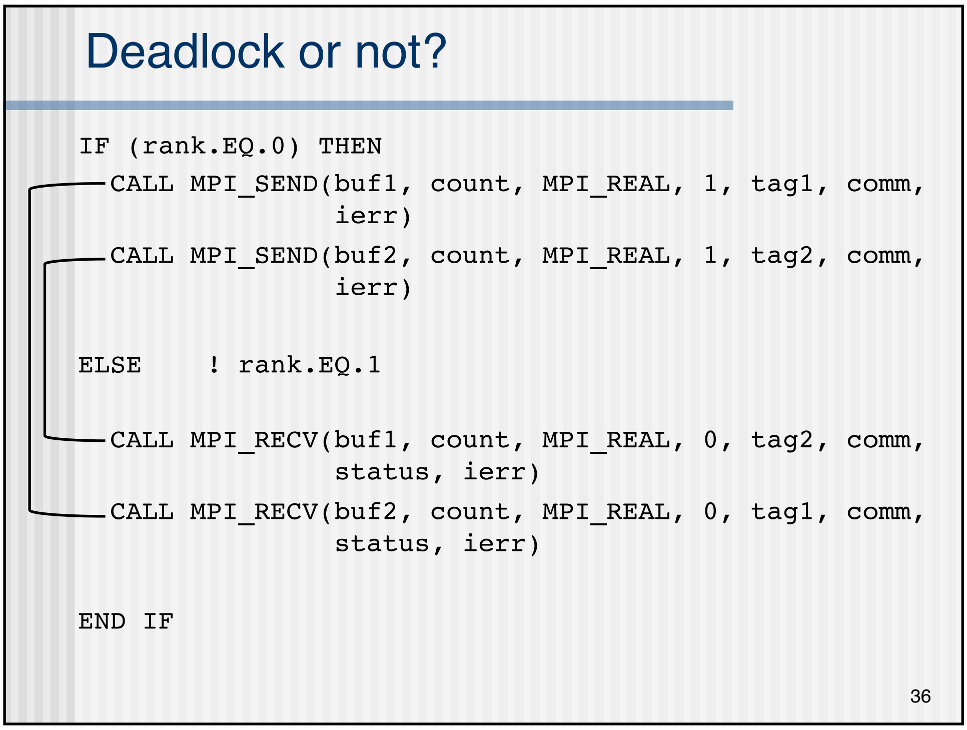


Message Order

- MPI messages are non-overtaking
  - If the sender sends two messages (with the same envelope) to the same destination they have to be received in the same order

```
IF (rank.EQ.0) THEN
  CALL MPI_SEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
  CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag1, comm, ierr)

ELSE    ! Rank.EQ.1

  CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag1, comm,
                status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm,
                status, ierr)
END IF
```

35



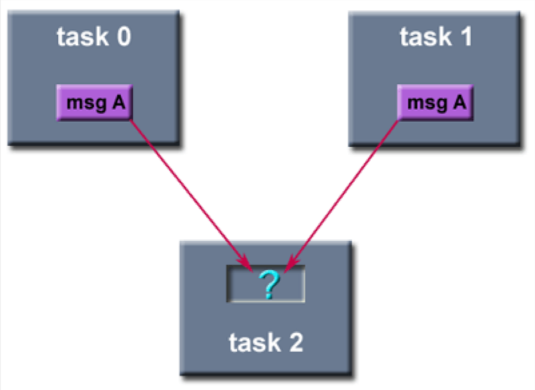
Deadlock or not?

```
IF (rank.EQ.0) THEN
  CALL MPI_SEND(buf1, count, MPI_REAL, 1, tag1, comm,
                ierr)
  CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag2, comm,
                ierr)

ELSE    ! rank.EQ.1

  CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm,
                status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm,
                status, ierr)

END IF
```

36

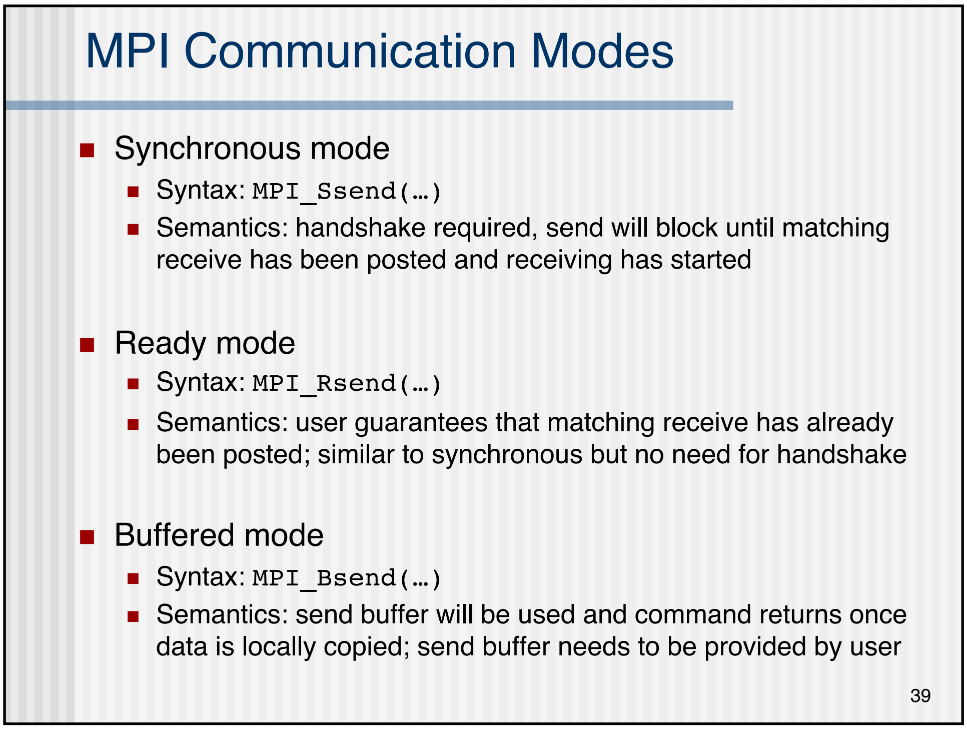
Fairness

- MPI makes no guarantees about fairness
  - If there are two matching sends (from different sources) for a receive any of these can be successful
  - MPI does not prevent operation starvation (e.g. sends that will never be picked up)



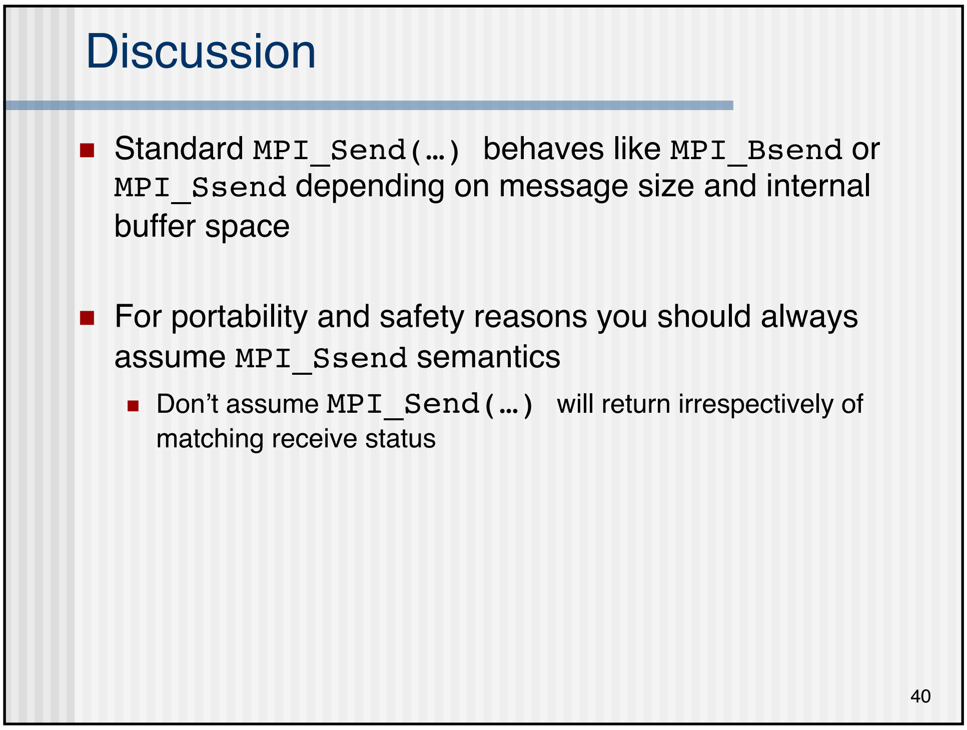
What have we learned?

- The semantics of `MPI_Send/MPI_Recv` are quite implementation dependent
- How can we get more control on what is actually happening?
  - MPI provides different communication modes with different semantics



MPI Communication Modes

- Synchronous mode
  - Syntax: `MPI_Ssend(…)`
  - Semantics: handshake required, send will block until matching receive has been posted and receiving has started
- Ready mode
  - Syntax: `MPI_Rsend(…)`
  - Semantics: user guarantees that matching receive has already been posted; similar to synchronous but no need for handshake
- Buffered mode
  - Syntax: `MPI_Bsend(…)`
  - Semantics: send buffer will be used and command returns once data is locally copied; send buffer needs to be provided by user



Discussion

- Standard `MPI_Send(…)` behaves like `MPI_Bsend` or `MPI_Ssend` depending on message size and internal buffer space
- For portability and safety reasons you should always assume `MPI_Ssend` semantics
  - Don't assume `MPI_Send(…)` will return irrespectively of matching receive status

Discussion Cont'd

- `MPI_Bsend` will fail if not enough buffer space is available
  - You must provide sufficient buffer space in user space to an MPI process:

```
int MPI_Buffer_attach( void* buffer, int size)
MPI_BUFFER_ATTACH( BUFFER, SIZE, IERROR)

int MPI_Buffer_detach( void* buffer_addr, int* size)
MPI_BUFFER_DETACH( BUFFER_ADDR, SIZE, IERROR)
```

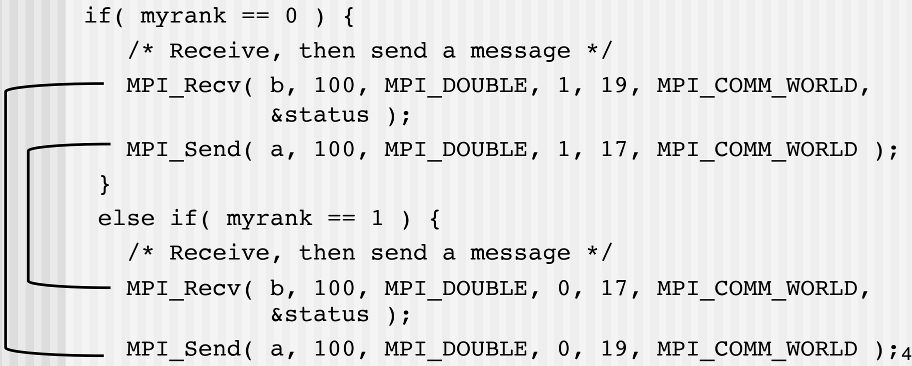
- This buffer is only used for buffered send and detach will block until all data is actually sent.

Pros and Cons of different modes

Advantages	Disadvantages
Synchronous Mode	
Safest, most portable	Can occur substantial synchronization overhead
Ready Mode	
Lowest total overhead	Difficult to guarantee that receive precedes send
Buffered Mode	
Decouples send from receive	Potentially substantial overhead through buffering
Standard Mode	
Most flexible, general purpose	Implementation dependent

Deadlocks

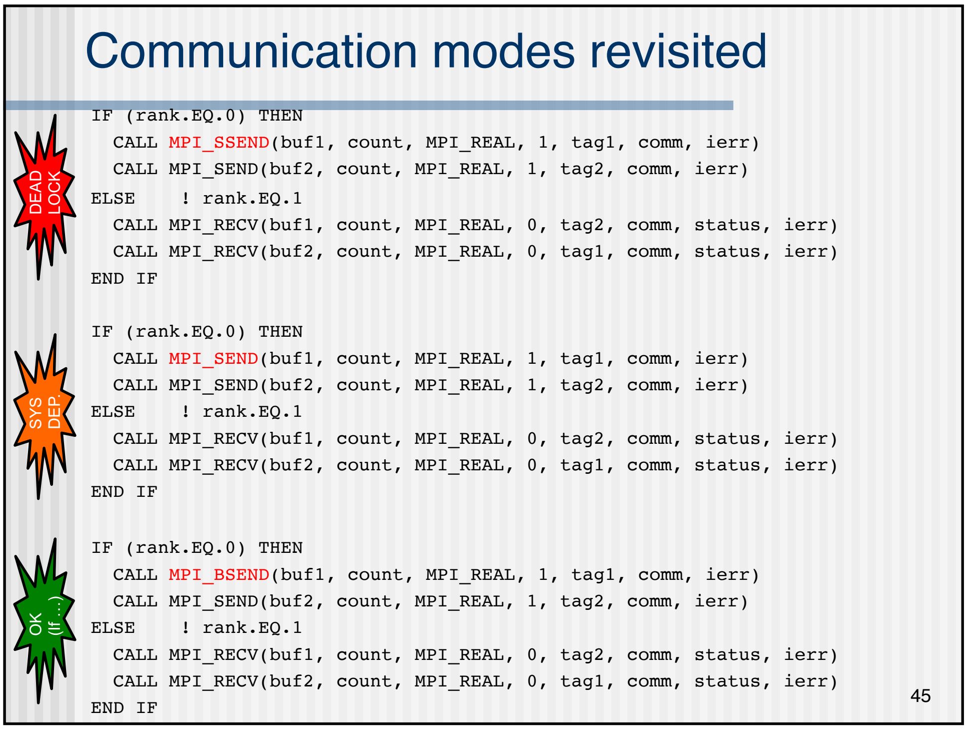
- Deadlocks are common (and hard to debug) errors in message passing programs
- A deadlock occurs when two (or more) processes wait on the progress of each other:



```
if( myrank == 0 ) {
   /* Receive, then send a message */
   MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD,
             &status );
   MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
 }
 else if( myrank == 1 ) {
   /* Receive, then send a message */
   MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD,
             &status );
   MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
```

How to avoid Deadlocks?

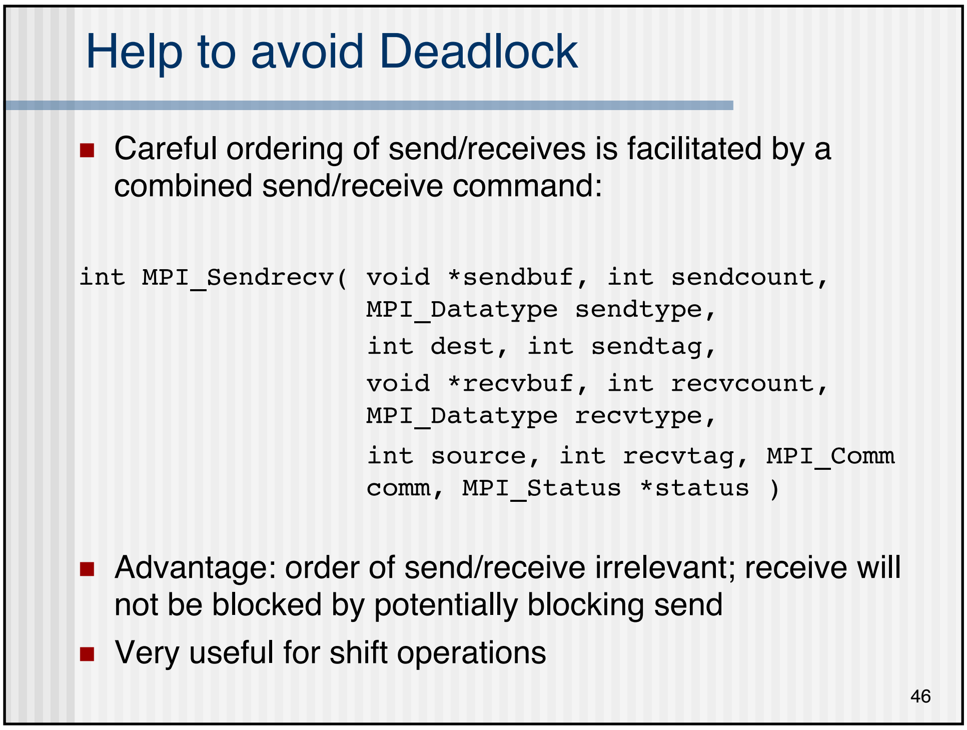
- Careful organize the communication in your program
  - Make sure sends are always paired with receives in the correct order
  - A difficult task in large programs!
- Don’t depend on how specific implementations handle their internal buffers
  - A program may work well with certain problem sizes but deadlock once you increase the problem size or move to a different architecture or MPI implementation because of internal buffer limitations



Communication modes revisited

DEAD
LOCK
SYS
DEP
OK
(if ...)

```
IF (rank.EQ.0) THEN
  CALL MPI_SSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
  CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE    ! rank.EQ.1
  CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF

IF (rank.EQ.0) THEN
  CALL MPI_SEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
  CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE    ! rank.EQ.1
  CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF

IF (rank.EQ.0) THEN
  CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
  CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE    ! rank.EQ.1
  CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

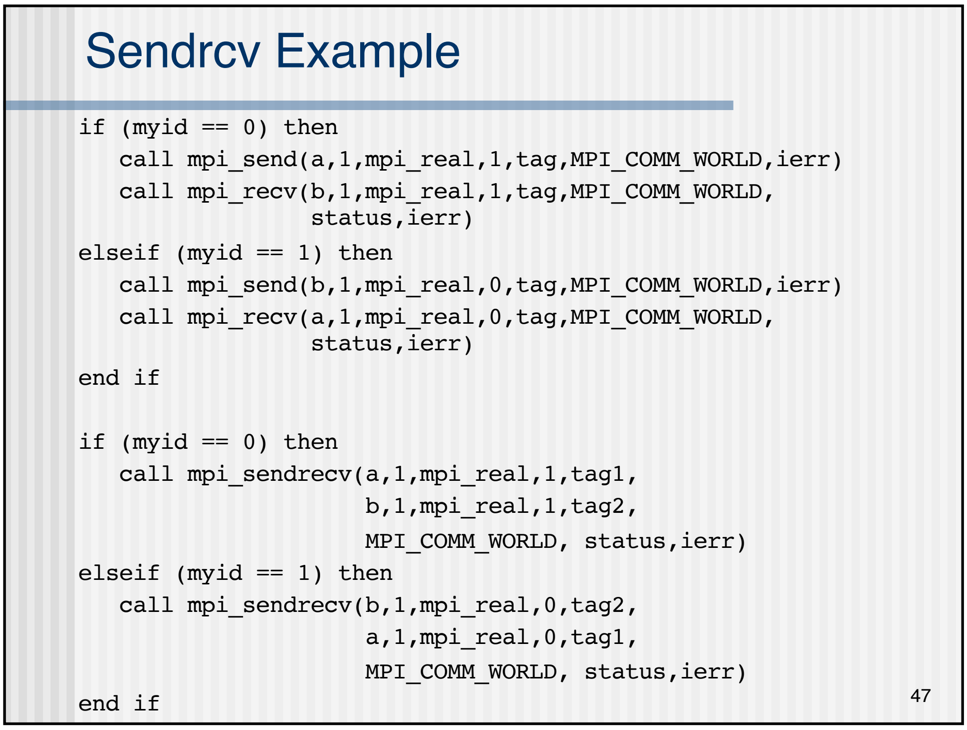


Help to avoid Deadlock

- Careful ordering of send/receives is facilitated by a combined send/receive command:

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

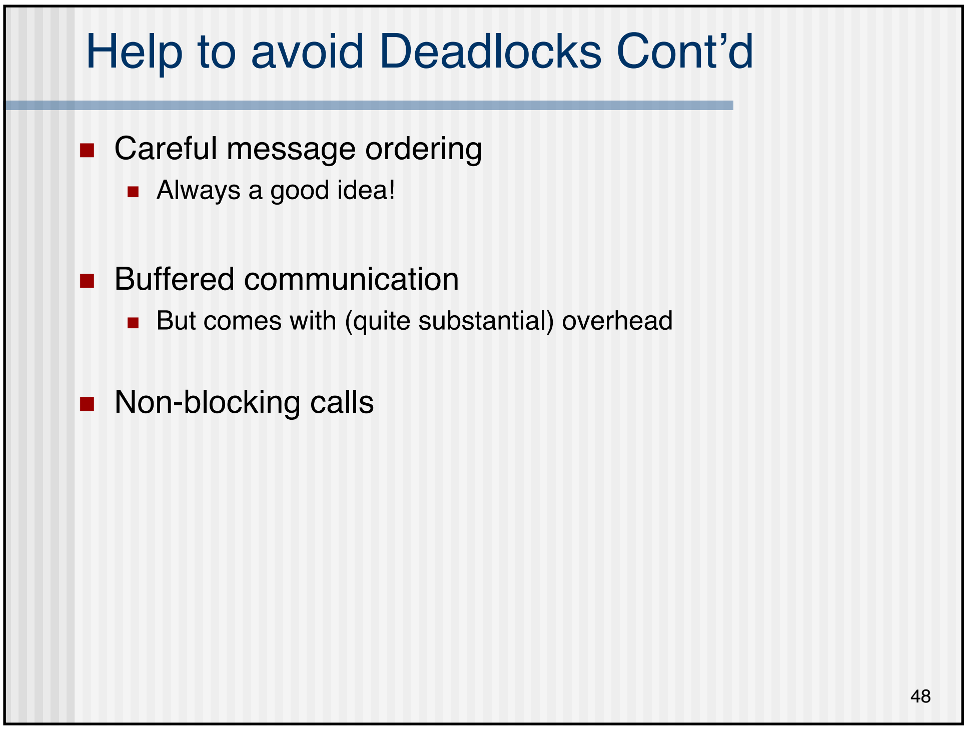
- Advantage: order of send/receive irrelevant; receive will not be blocked by potentially blocking send
- Very useful for shift operations



Sendrcv Example

```
if (myid == 0) then
   call mpi_send(a,1,mpi_real,1,tag,MPI_COMM_WORLD,ierr)
   call mpi_recv(b,1,mpi_real,1,tag,MPI_COMM_WORLD,
                 status,ierr)
elseif (myid == 1) then
   call mpi_send(b,1,mpi_real,0,tag,MPI_COMM_WORLD,ierr)
   call mpi_recv(a,1,mpi_real,0,tag,MPI_COMM_WORLD,
                 status,ierr)
end if

if (myid == 0) then
   call mpi_sendrecv(a,1,mpi_real,1,tag1,
                     b,1,mpi_real,1,tag2,
                     MPI_COMM_WORLD, status,ierr)
elseif (myid == 1) then
   call mpi_sendrecv(b,1,mpi_real,0,tag2,
                     a,1,mpi_real,0,tag1,
                     MPI_COMM_WORLD, status,ierr)
end if
```

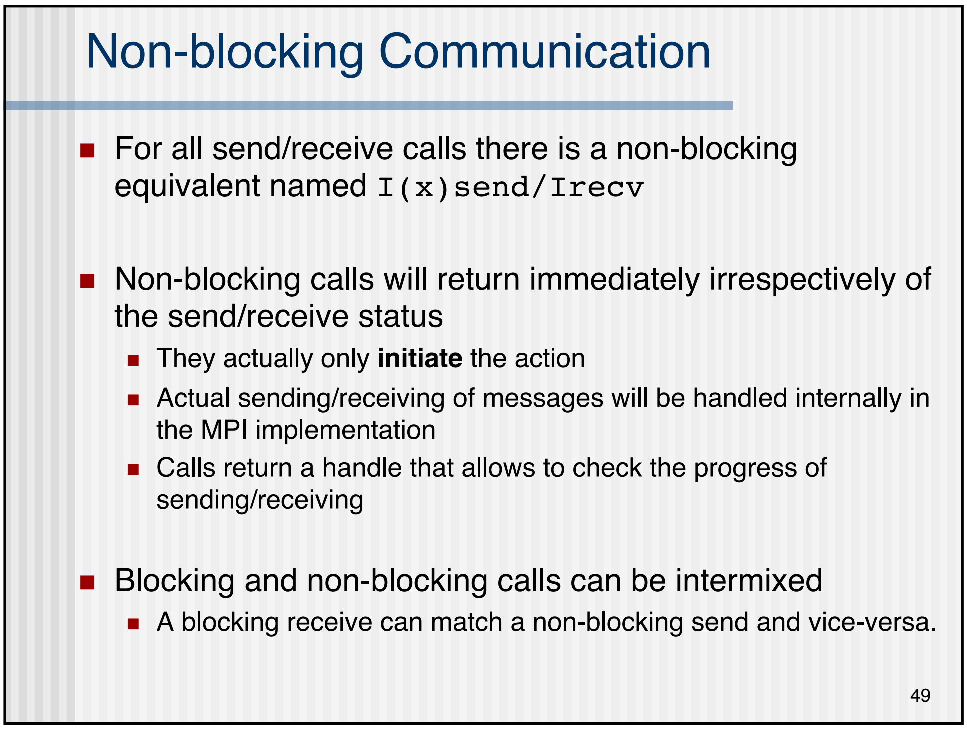
47



Help to avoid Deadlocks Cont’d

- Careful message ordering
  - Always a good idea!
- Buffered communication
  - But comes with (quite substantial) overhead
- Non-blocking calls

48

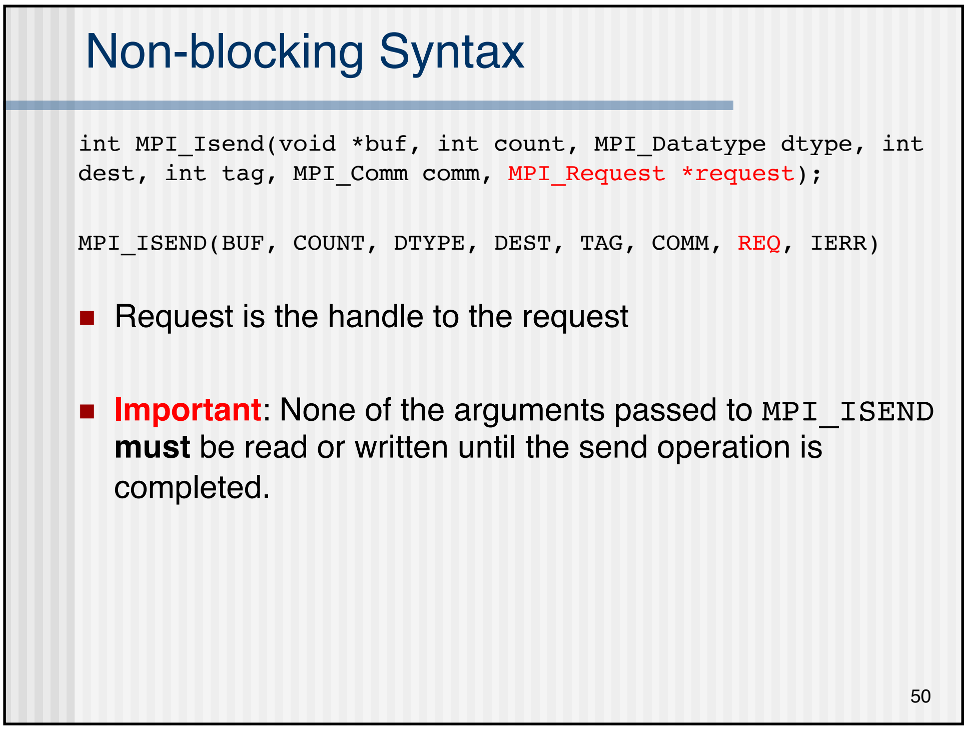


Non-blocking Communication

- For all send/receive calls there is a non-blocking equivalent named `I(x)send/Irecv`

- Non-blocking calls will return immediately irrespectively of the send/receive status
  - They actually only **initiate** the action
  - Actual sending/receiving of messages will be handled internally in the MPI implementation
  - Calls return a handle that allows to check the progress of sending/receiving

- Blocking and non-blocking calls can be intermixed
  - A blocking receive can match a non-blocking send and vice-versa.

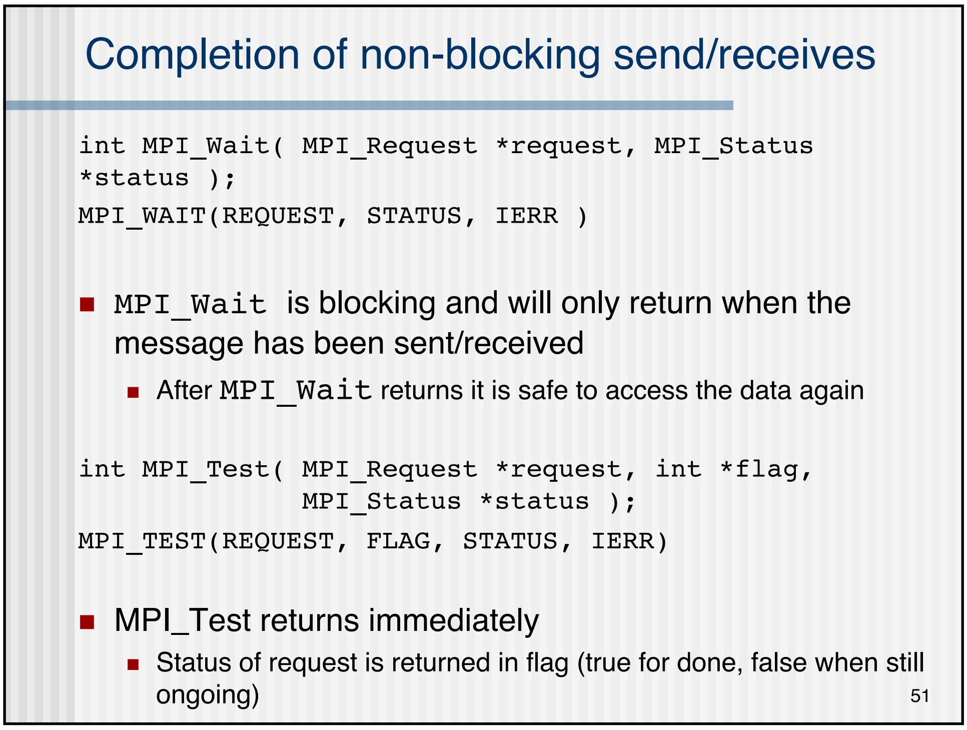


Non-blocking Syntax

```
int MPI_Isend(void *buf, int count, MPI_Datatype dtype, int
dest, int tag, MPI_Comm comm, MPI_Request *request);

MPI_ISEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, REQ, IERR)
```

- Request is the handle to the request

- **Important**: None of the arguments passed to `MPI_ISEND` **must** be read or written until the send operation is completed.



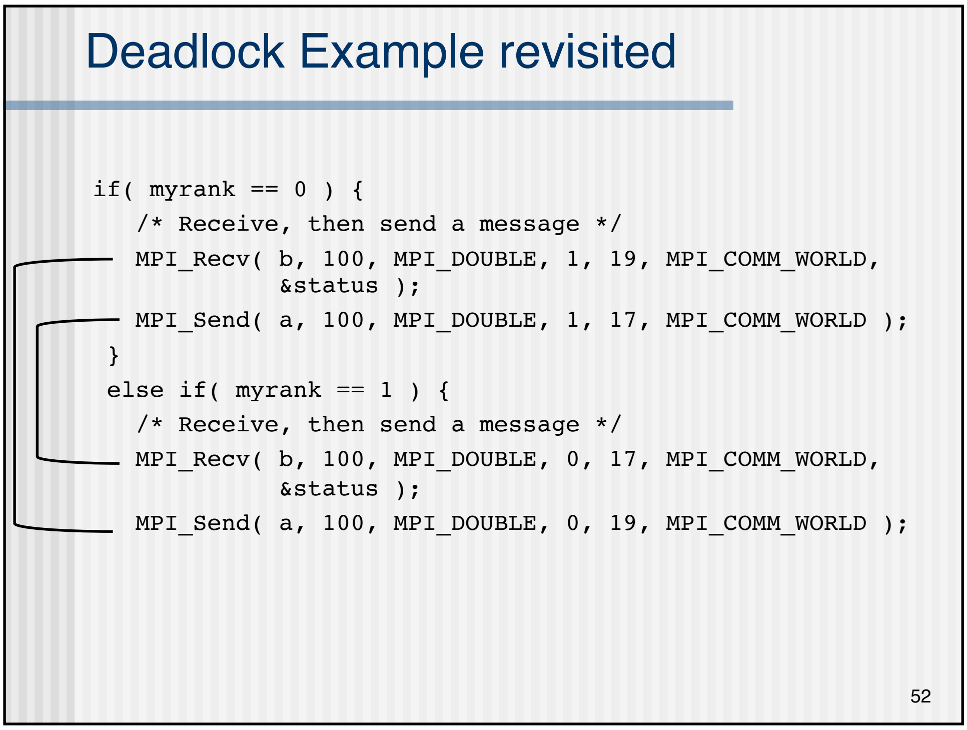
Completion of non-blocking send/receives

```
int MPI_Wait( MPI_Request *request, MPI_Status
*status );
MPI_WAIT(REQUEST, STATUS, IERR )
```

- `MPI_Wait` is blocking and will only return when the message has been sent/received
  - After `MPI_Wait` returns it is safe to access the data again

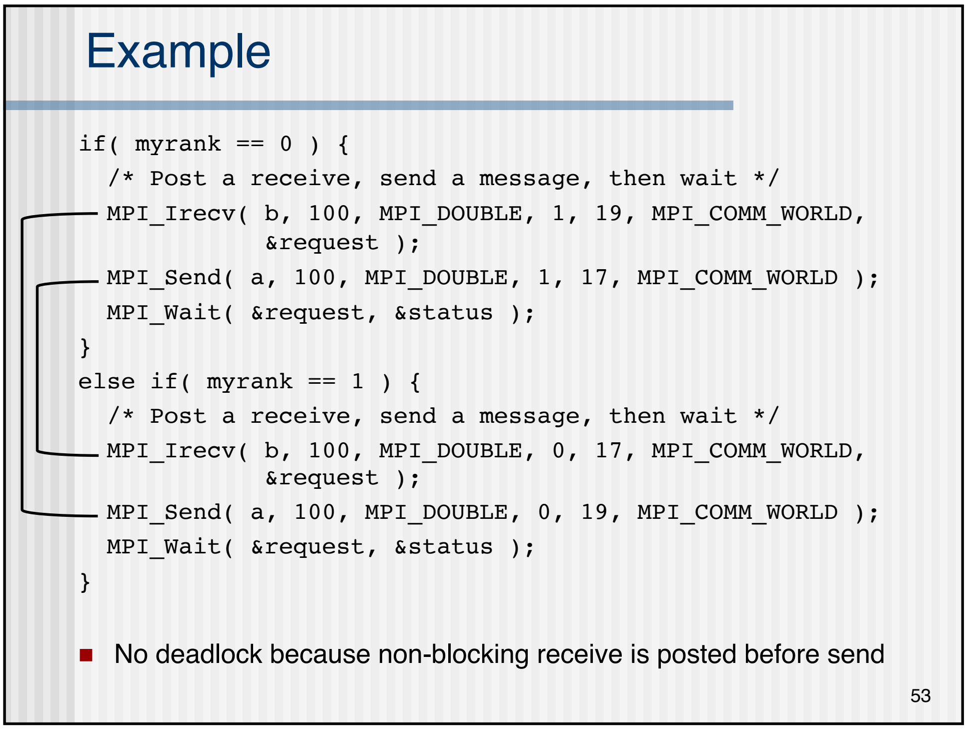
```
int MPI_Test( MPI_Request *request, int *flag,
              MPI_Status *status );
MPI_TEST(REQUEST, FLAG, STATUS, IERR)
```

- MPI_Test returns immediately
  - Status of request is returned in flag (true for done, false when still ongoing)



Deadlock Example revisited

```
 if( myrank == 0 ) {
    /* Receive, then send a message */
    MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD,
              &status );
    MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
  }
  else if( myrank == 1 ) {
    /* Receive, then send a message */
    MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD,
              &status );
    MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
```

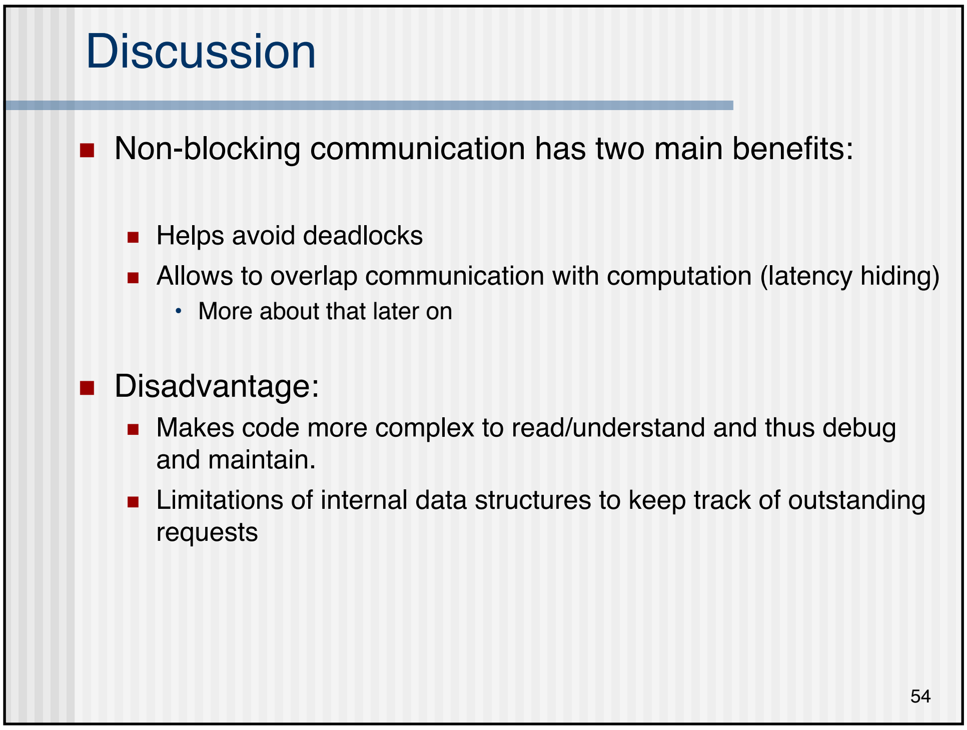


Example

```
if( myrank == 0 ) {
  /* Post a receive, send a message, then wait */
  MPI_Irecv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD,
             &request );
  MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
  MPI_Wait( &request, &status );
}
else if( myrank == 1 ) {
  /* Post a receive, send a message, then wait */
  MPI_Irecv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD,
             &request );
  MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
  MPI_Wait( &request, &status );
}
```

- No deadlock because non-blocking receive is posted before send

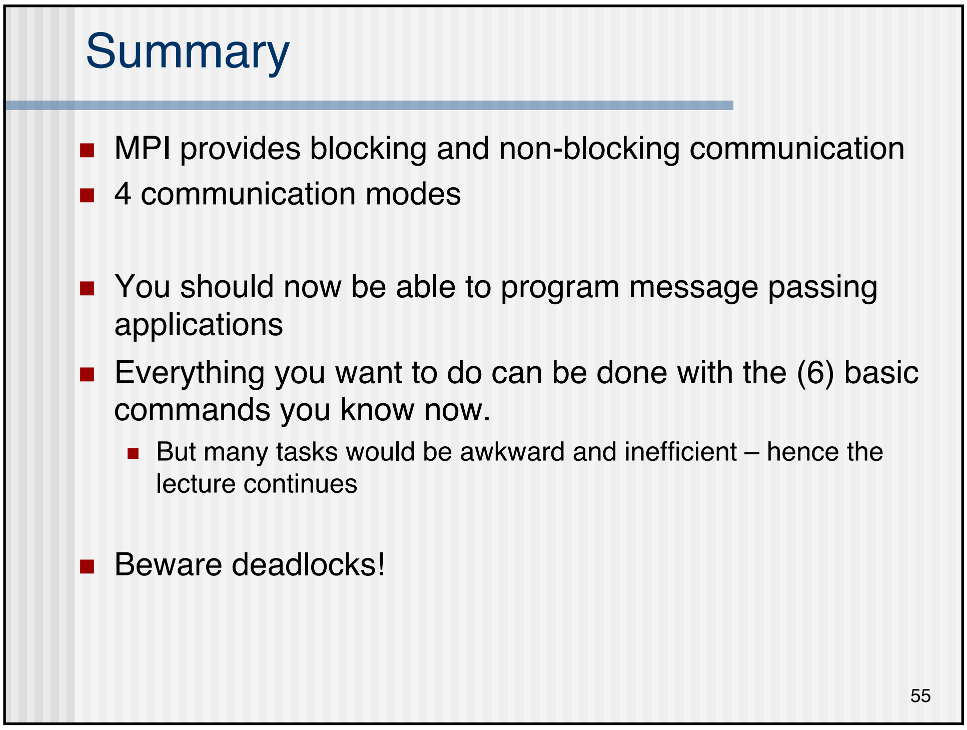
53



Discussion

- Non-blocking communication has two main benefits:
  - Helps avoid deadlocks
  - Allows to overlap communication with computation (latency hiding)
    - More about that later on
- Disadvantage:
  - Makes code more complex to read/understand and thus debug and maintain.
  - Limitations of internal data structures to keep track of outstanding requests

54



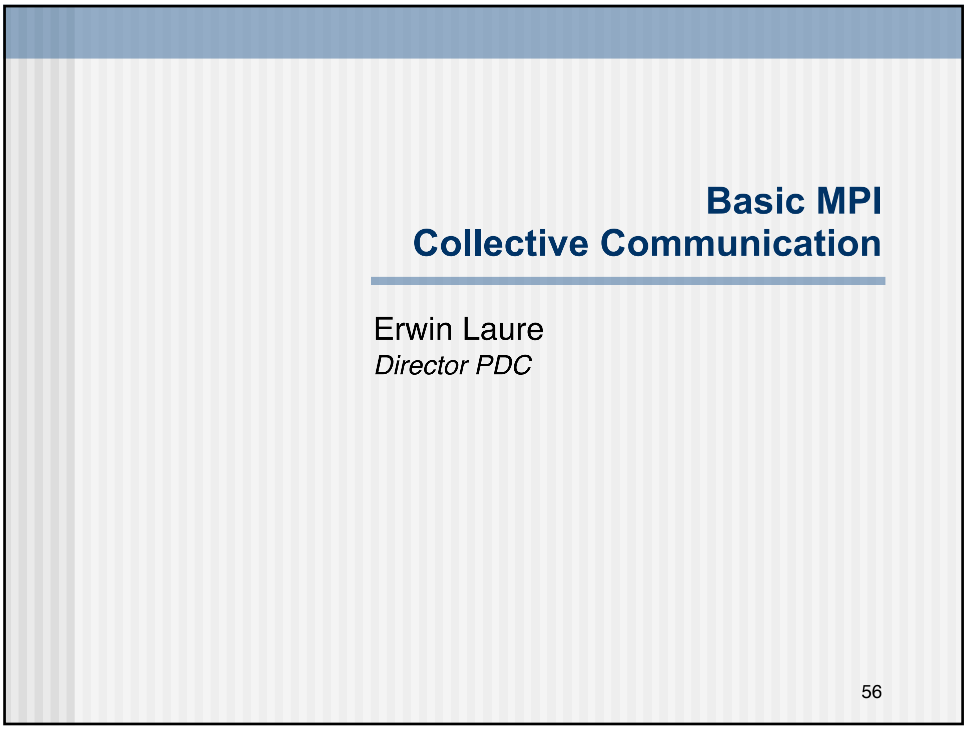
Summary

- MPI provides blocking and non-blocking communication
- 4 communication modes

- You should now be able to program message passing applications
- Everything you want to do can be done with the (6) basic commands you know now.
  - But many tasks would be awkward and inefficient – hence the lecture continues

- Beware deadlocks!

55

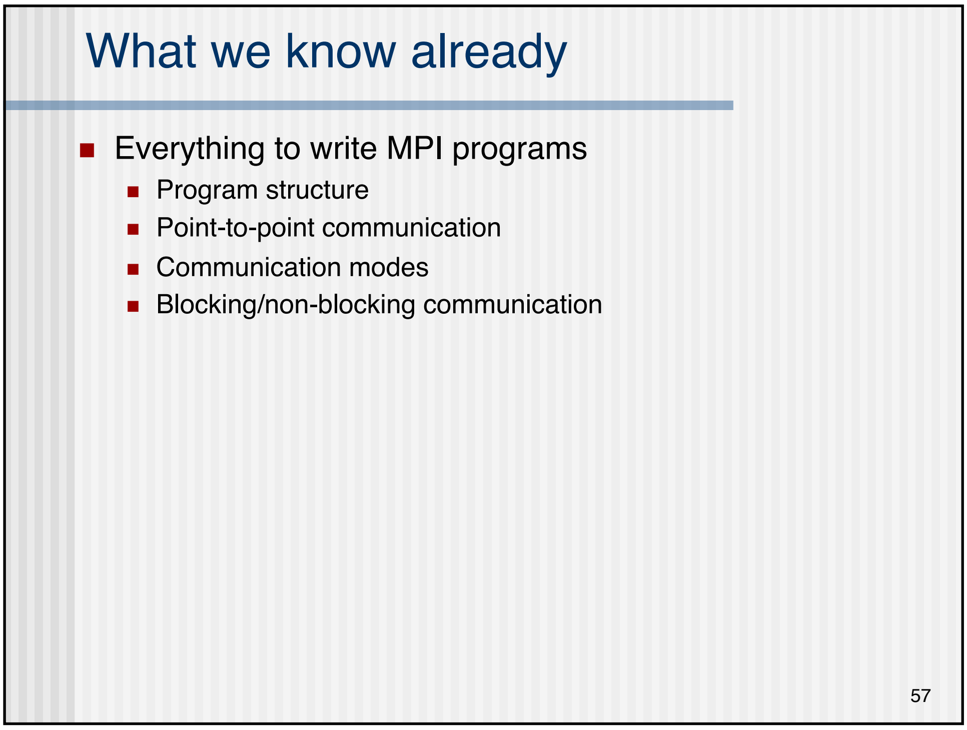


Basic MPI Collective Communication

Erwin Laure

Director PDC

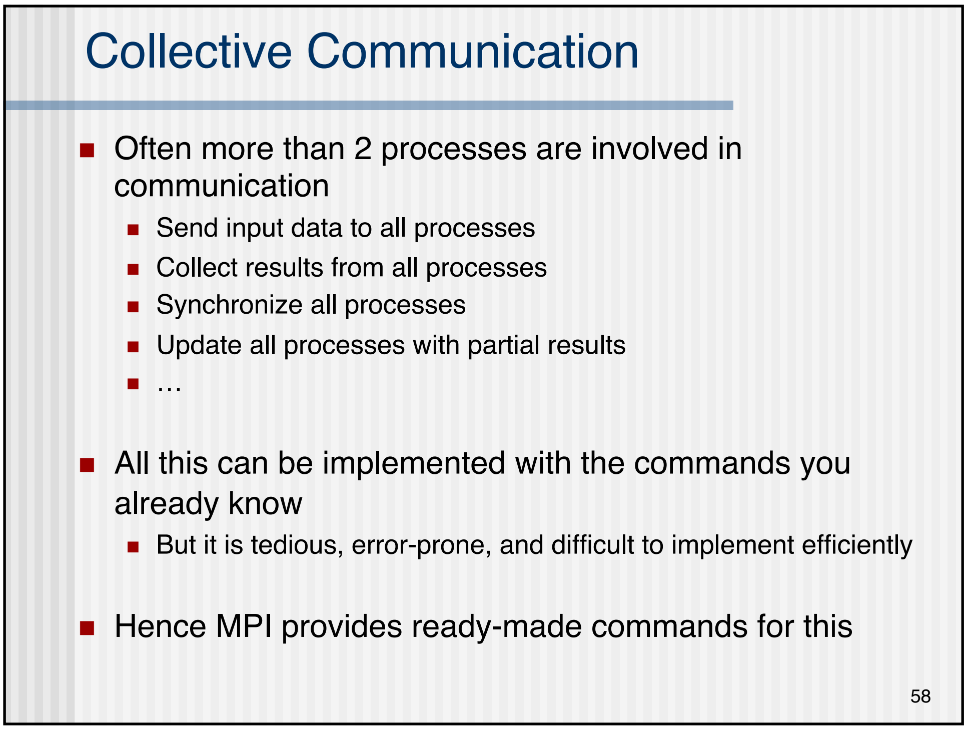
56



What we know already

- Everything to write MPI programs
  - Program structure
  - Point-to-point communication
  - Communication modes
  - Blocking/non-blocking communication

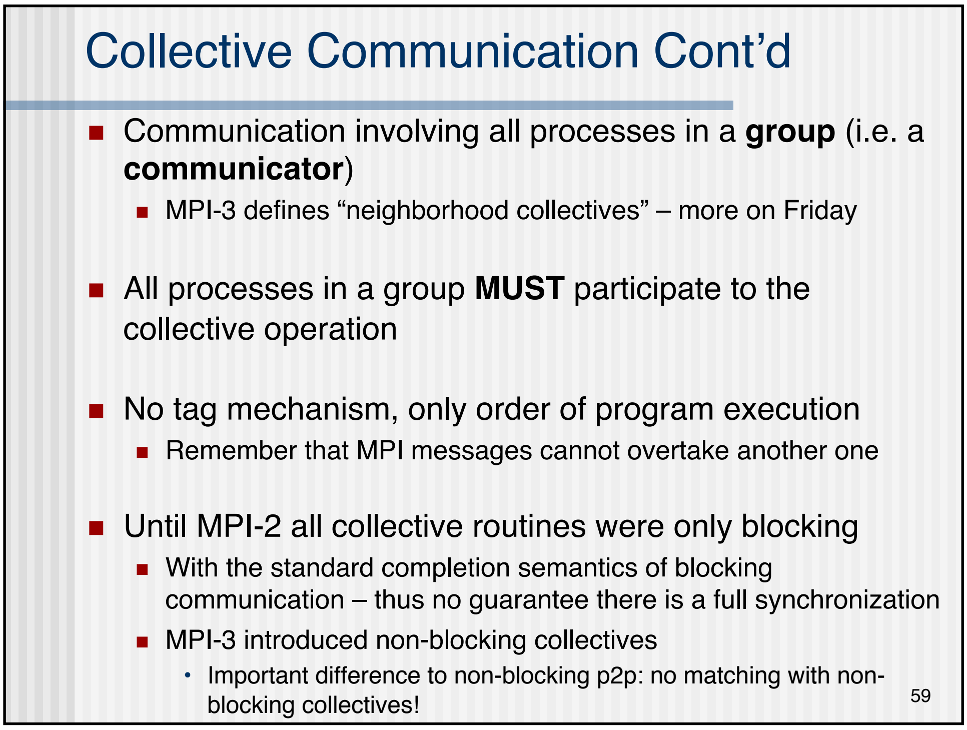
57



Collective Communication

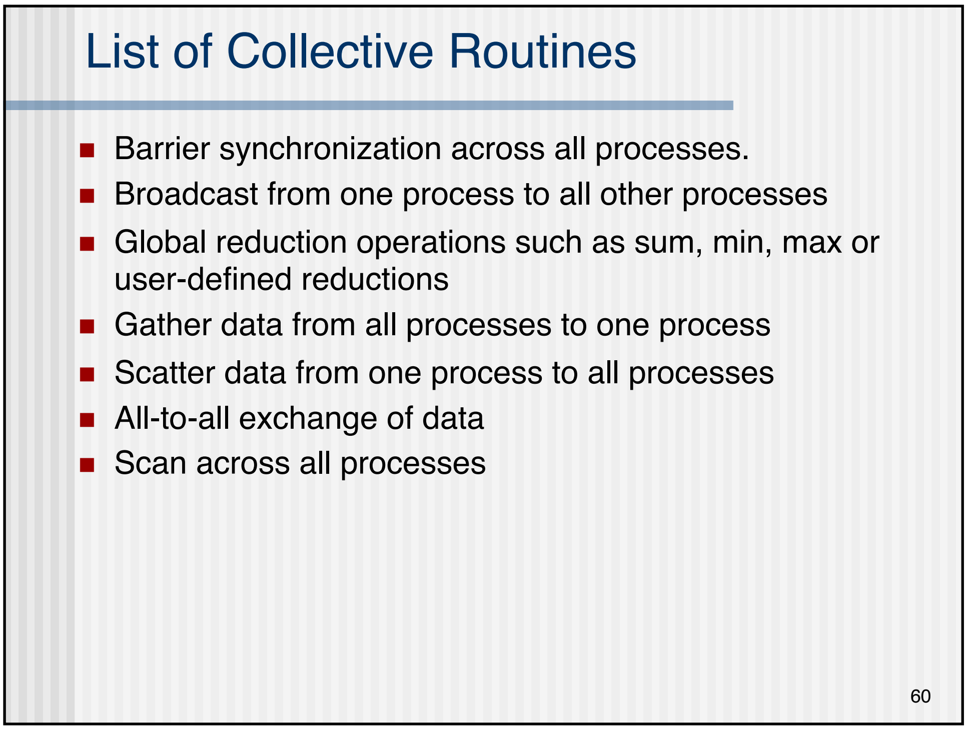
- Often more than 2 processes are involved in communication
  - Send input data to all processes
  - Collect results from all processes
  - Synchronize all processes
  - Update all processes with partial results
  - …
- All this can be implemented with the commands you already know
  - But it is tedious, error-prone, and difficult to implement efficiently
- Hence MPI provides ready-made commands for this

58



Collective Communication Cont'd

- Communication involving all processes in a **group** (i.e. a **communicator**)
  - MPI-3 defines "neighborhood collectives" – more on Friday
- All processes in a group **MUST** participate to the collective operation
- No tag mechanism, only order of program execution
  - Remember that MPI messages cannot overtake another one
- Until MPI-2 all collective routines were only blocking
  - With the standard completion semantics of blocking communication – thus no guarantee there is a full synchronization
  - MPI-3 introduced non-blocking collectives
    - Important difference to non-blocking p2p: no matching with non-blocking collectives!



List of Collective Routines

- Barrier synchronization across all processes.
- Broadcast from one process to all other processes
- Global reduction operations such as sum, min, max or user-defined reductions
- Gather data from all processes to one process
- Scatter data from one process to all processes
- All-to-all exchange of data
- Scan across all processes

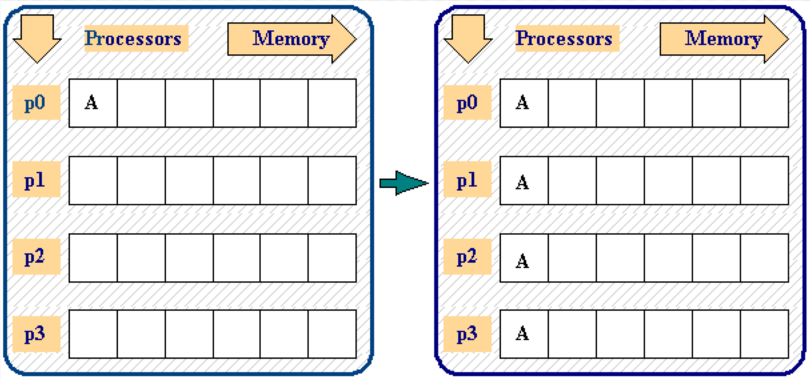
Barrier Synchronization

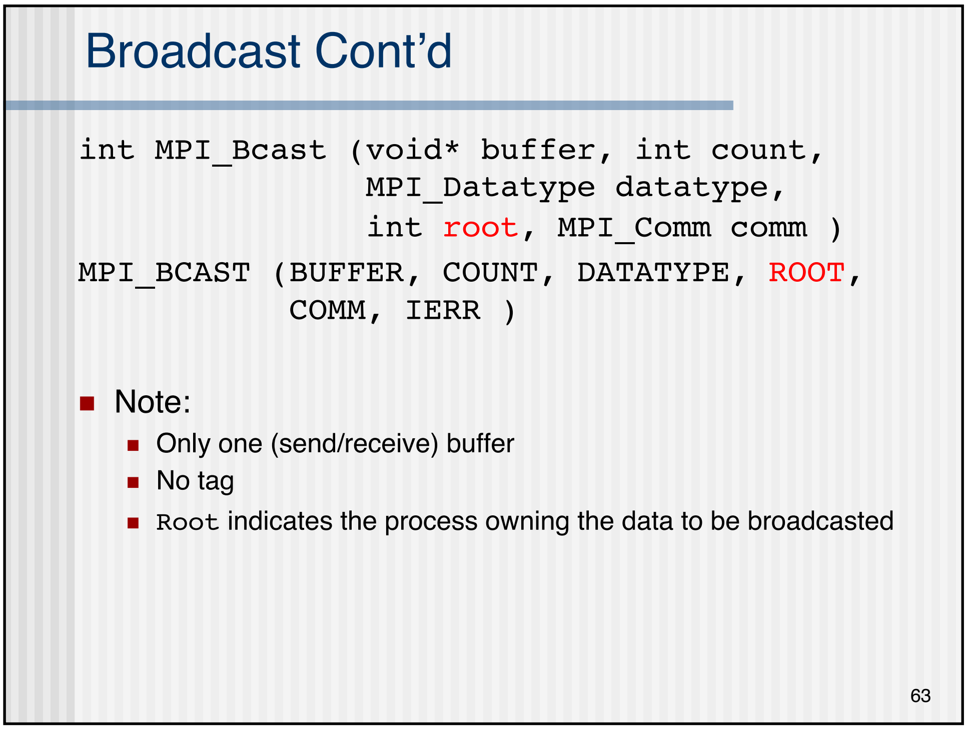
- Sometimes there is a need to synchronize all processes before them continuing independently
  - E.g. read in input data
- `MPI_Barrier` blocks the calling process until all processes in the group have also called `MPI_Barrier`

```
int MPI_Barrier ( MPI_comm comm  )

MPI_BARRIER ( COMM, ERROR )
```

Broadcast

- Broadcast sends data from one process to the same memory location in all other processes
  - send and receive buffer are the same!

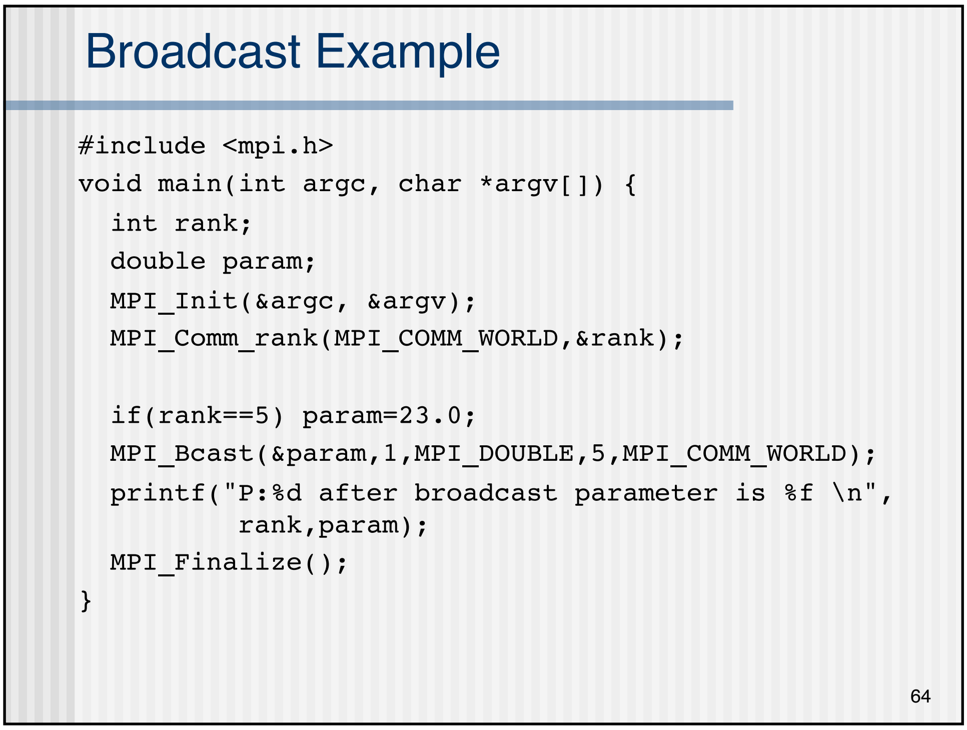




Broadcast Cont’d

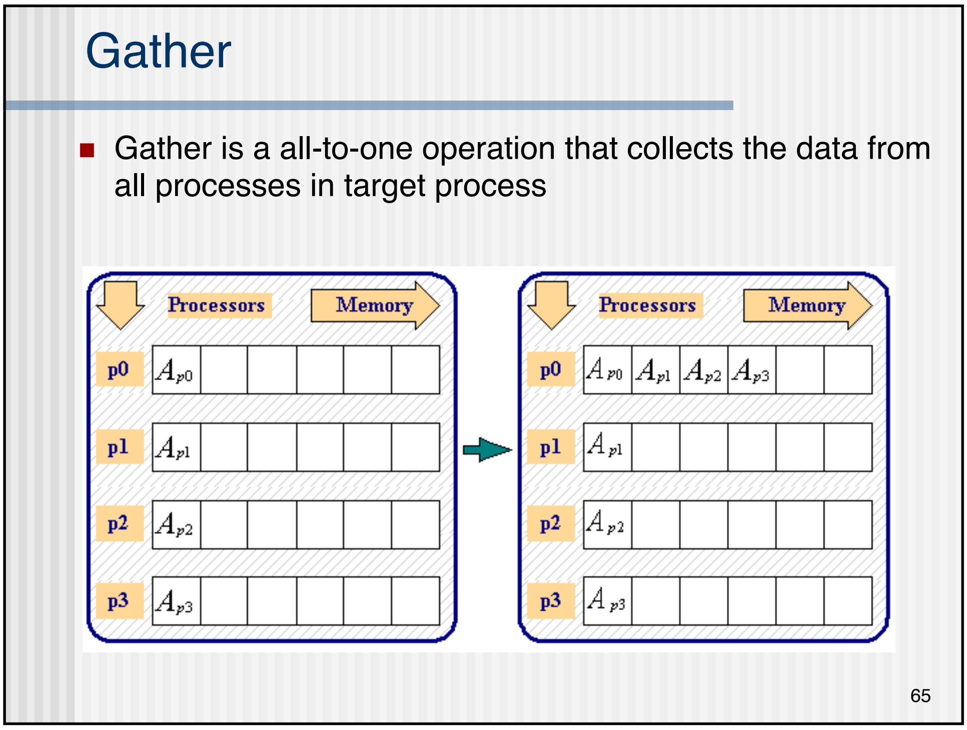
```
int MPI_Bcast (void* buffer, int count,
               MPI_Datatype datatype,
               int root, MPI_Comm comm )
MPI_BCAST (BUFFER, COUNT, DATATYPE, ROOT,
           COMM, IERR )
```

- Note:
  - Only one (send/receive) buffer
  - No tag
  - `Root` indicates the process owning the data to be broadcasted



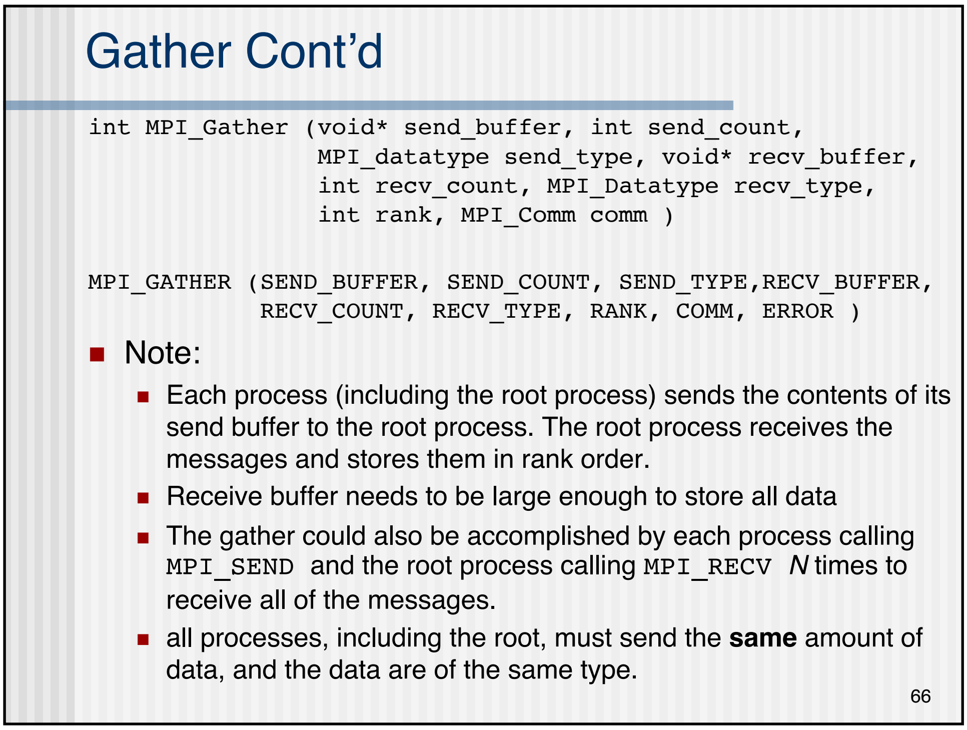
Broadcast Example

```
#include <mpi.h>
void main(int argc, char *argv[]) {
  int rank;
  double param;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if(rank==5) param=23.0;
  MPI_Bcast(&param,1,MPI_DOUBLE,5,MPI_COMM_WORLD);
  printf("P:%d after broadcast parameter is %f \n",
          rank,param);
  MPI_Finalize();
}
```



Gather

- Gather is a all-to-one operation that collects the data from all processes in target process



Gather Cont'd

```
int MPI_Gather (void* send_buffer, int send_count,
                MPI_datatype send_type, void* recv_buffer,
                int recv_count, MPI_Datatype recv_type,
                int rank, MPI_Comm comm )


MPI_GATHER (SEND_BUFFER, SEND_COUNT, SEND_TYPE,RECV_BUFFER,
            RECV_COUNT, RECV_TYPE, RANK, COMM, ERROR )
```

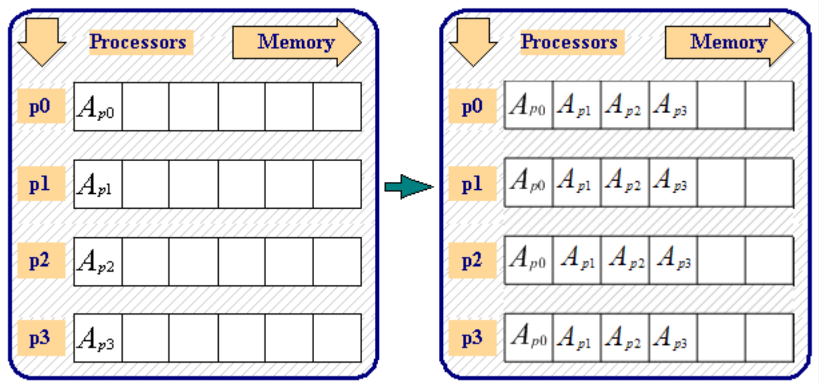
- Note:
  - Each process (including the root process) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order.
  - Receive buffer needs to be large enough to store all data
  - The gather could also be accomplished by each process calling `MPI_SEND` and the root process calling `MPI_RECV` *N* times to receive all of the messages.
  - all processes, including the root, must send the **same** amount of data, and the data are of the same type.

Gather Example

```
int rank,size;
double param[16],mine;
int sndcnt,rcvcnt; I;

sndcnt=1;
mine=23.0+rank;
if(rank==7) rcvcnt=1;

MPI_Gather(&mine,sndcnt,MPI_DOUBLE,param,rcvcnt,
           MPI_DOUBLE,7,MPI_COMM_WORLD);

if(rank==7)
for(i=0;i<size;++i) printf("PE:%d param[%d] is %f \n",
    rank,i,param[i]]);
```

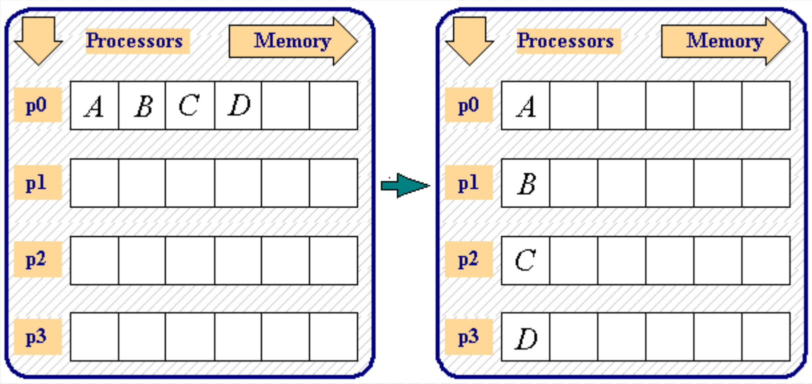
Allgather

- Sometimes it is also useful to gather the data not only into one process but all
- Equivalent to `MPI_Gather` plus `MPI_Bcast`
- `MPI_Allgather` has same syntax as `MPI_Gather`


Processors
Memory
p0
A_{p0}
p1
A_{p1}
p2
A_{p2}
p3
A_{p3}
Processors
Memory
p0
A_{p0} A_{p1} A_{p2} A_{p3}
p1
A_{p0} A_{p1} A_{p2} A_{p3}
p2
A_{p0} A_{p1} A_{p2} A_{p3}
p3
A_{p0} A_{p1} A_{p2} A_{p3}

Scatter

- Distribute data to all processes – one-to-all communication
- Inverse to gather



Scatter Cont'd

```
int MPI_Scatter (void* send_buffer, int send_count,
                 MPI_datatype send_type,
                 void* recv_buffer, int recv_count,
                 MPI_Datatype recv_type,
                 int rank, MPI_Comm comm )


MPI_Scatter (SEND_BUFFER, SEND_COUNT, SEND_TYPE,
             RECV_BUFFER, RECV_COUNT, RECV_TYPE,
             RANK, COMM, ERROR )
```

- root process breaks up the send buffer into equal chunks and sends one chunk to each processor.
  - The outcome is the same as if the root executed *N* MPI_SEND operations and each process executed an MPI_RECV.

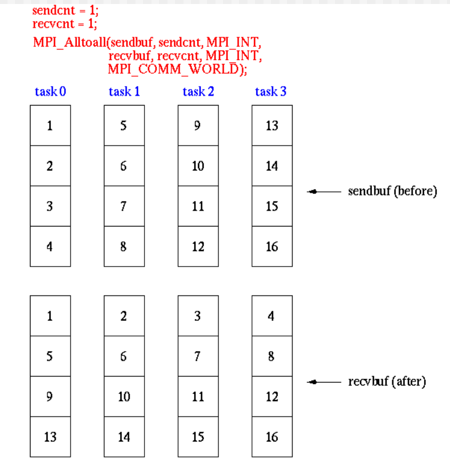
Scatter Example

```
rcvcnt=1;
if(rank==3) {
  for(i=0;i<8;++i) param[i]=23.0+i;
  sndcnt=1;
}
MPI_Scatter(param,sndcnt,MPI_DOUBLE,&mine,rcvcnt,
            MPI_DOUBLE,3,MPI_COMM_WORLD);
for(i=0;i<size;++i)  {
  if(rank==i) printf("P:%d mine is %f \n",rank,mine);
  fflush(stdout);
  MPI_Barrier(MPI_COMM_WORLD);
}
MPI_Finalize();
}
```

What will this barrier result in?

Other Gather/Scatter Variants

- Gather/Scatter is also defined over vectors
  - `MPI_GATHERV` and `MPI_SCATTERV` allow a varying count of data from/to each process.
- `MPI_ALLTOALL`
  - Every process performs a scatter



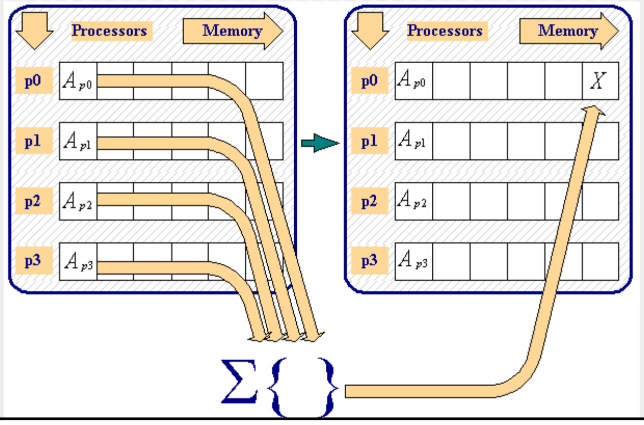
```
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT,
             recvbuf, recvcnt, MPI_INT,
             MPI_COMM_WORLD);
```

task 0	task 1	task 2	task 3	
1	5	9	13	
2	6	10	14	sendbuf (before)
3	7	11	15	
4	8	12	16	

1	2	3	4	
5	6	7	8	recvbuf (after)
9	10	11	12	
13	14	15	16	

Reduction

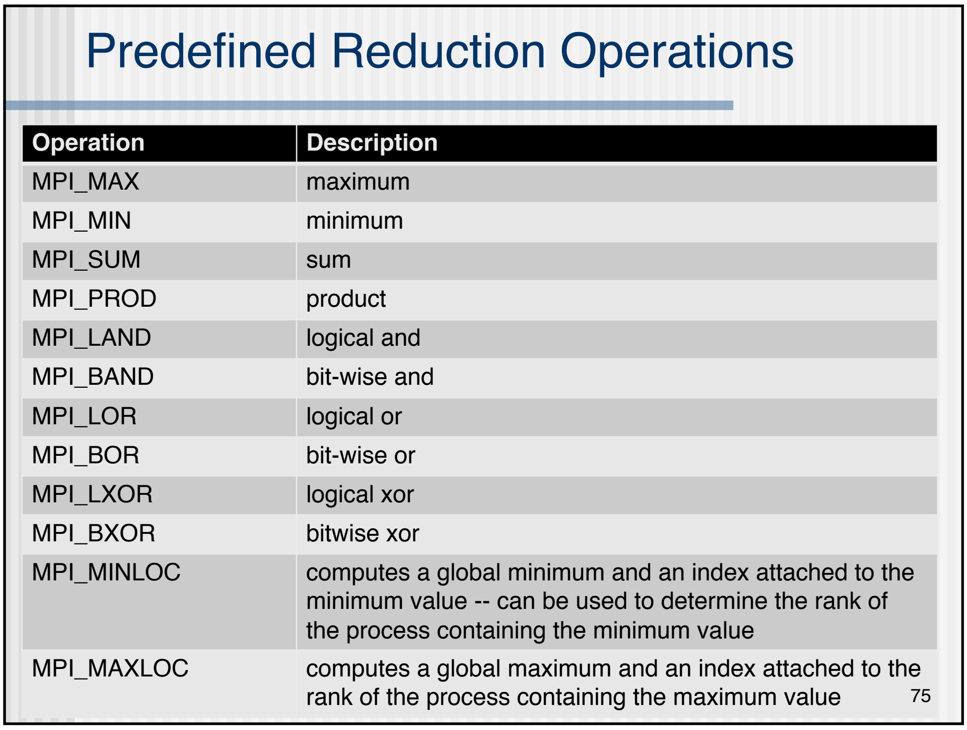
- Collect data from each processor
- Reduce these data to a single value (such as a sum or max)
- Store the reduced result on the root processor



Reduction Cont'd

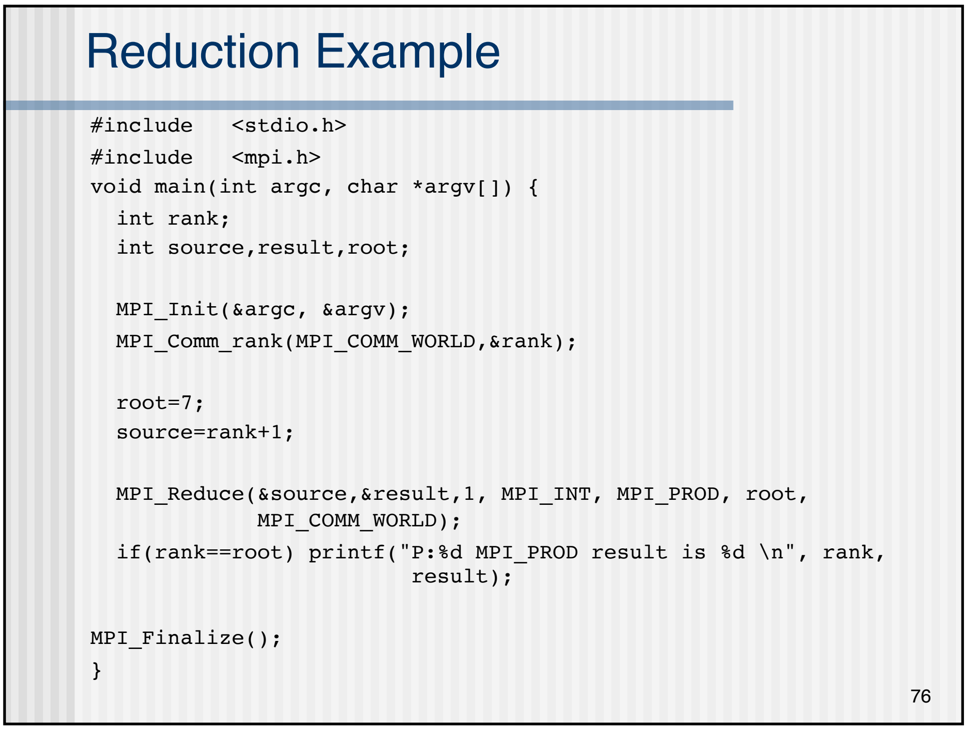
```
int MPI_Reduce (void* send_buffer, void* recv_buffer, int
                count, MPI_Datatype datatype, MPI_Op
                operation, int rank, MPI_Comm comm )


MPI_REDUCE ( SEND_BUFFER, RECV_BUFFER, COUNT, DATATYPE,
             OPERATION, RANK, COMM, ERROR )
```

- Note:
  - `Rank` denotes the process that stores the result in `recv_buffer`
  - `Operation` can be one of 12 pre-defined operations or user-defined
  - Both send and receive buffers must have the same number of elements with the same type.
    - The arguments `count` and `datatype` must have identical values in all processes.
  - The argument `rank` must also be the same in all processes.



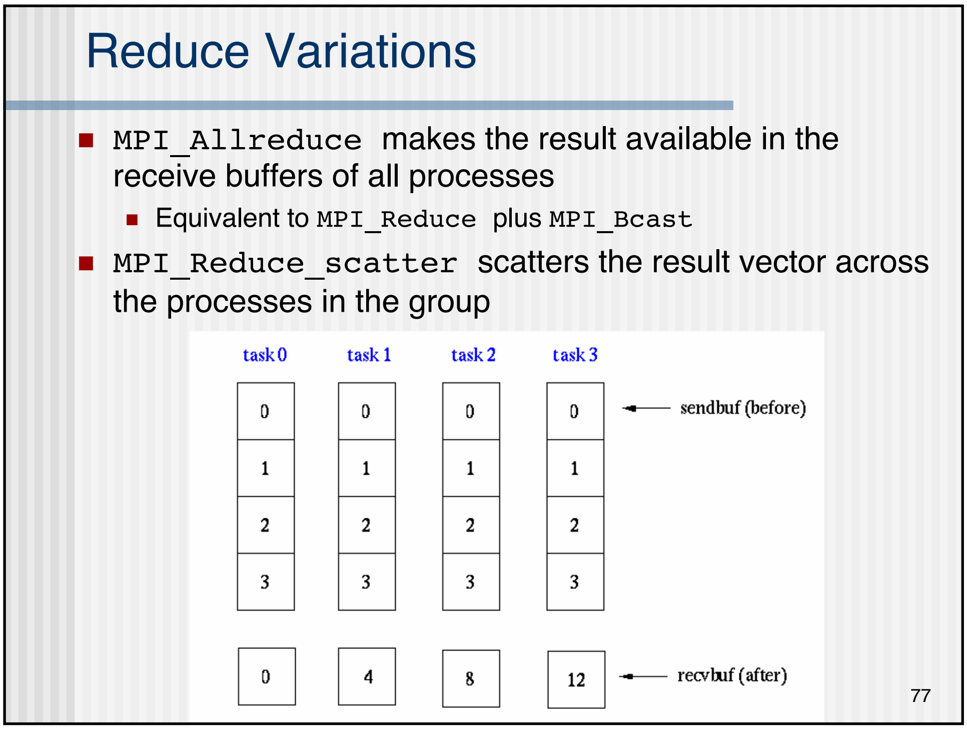
Predefined Reduction Operations

Operation	Description
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bitwise xor
MPI_MINLOC	computes a global minimum and an index attached to the minimum value -- can be used to determine the rank of the process containing the minimum value
MPI_MAXLOC	computes a global maximum and an index attached to the rank of the process containing the maximum value



Reduction Example

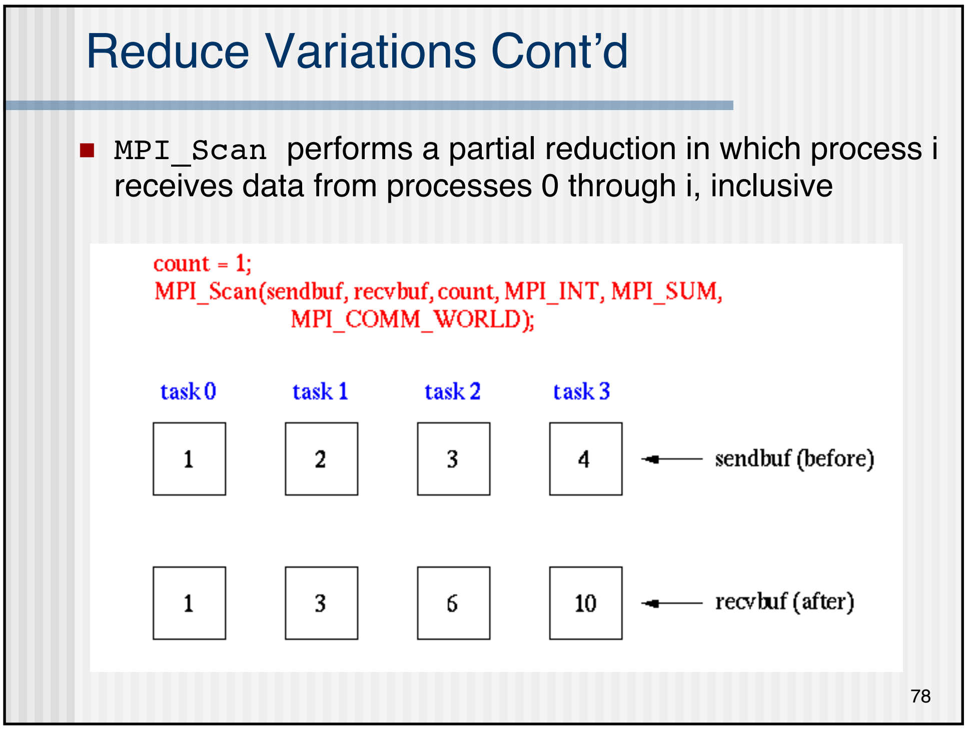
```
#include   <stdio.h>
#include   <mpi.h>
void main(int argc, char *argv[]) {
  int rank;
  int source,result,root;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  root=7;
  source=rank+1;

  MPI_Reduce(&source,&result,1, MPI_INT, MPI_PROD, root,
             MPI_COMM_WORLD);
  if(rank==root) printf("P:%d MPI_PROD result is %d \n", rank,
                        result);

MPI_Finalize();
}
```



Reduce Variations

- `MPI_Allreduce` makes the result available in the receive buffers of all processes
  - Equivalent to `MPI_Reduce` plus `MPI_Bcast`
- `MPI_Reduce_scatter` scatters the result vector across the processes in the group

task 0	task 1	task 2	task 3	
0	0	0	0	← sendbuf (before)
1	1	1	1	
2	2	2	2	
3	3	3	3	
0	4	8	12	← recvbuf (after)

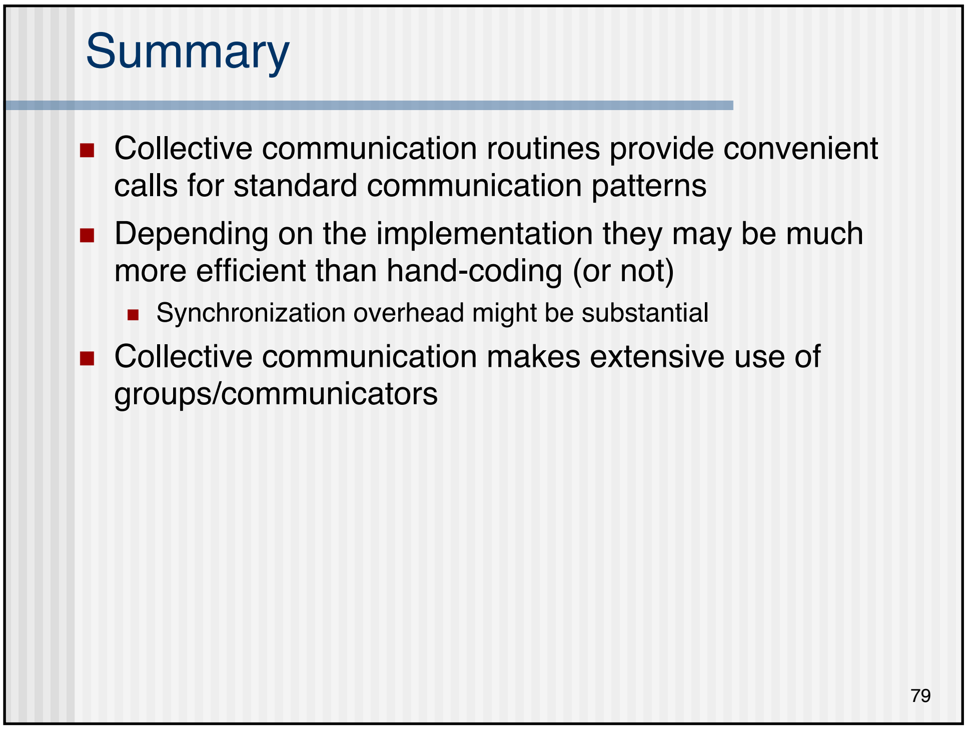


Reduce Variations Cont'd

- `MPI_Scan` performs a partial reduction in which process i receives data from processes 0 through i, inclusive

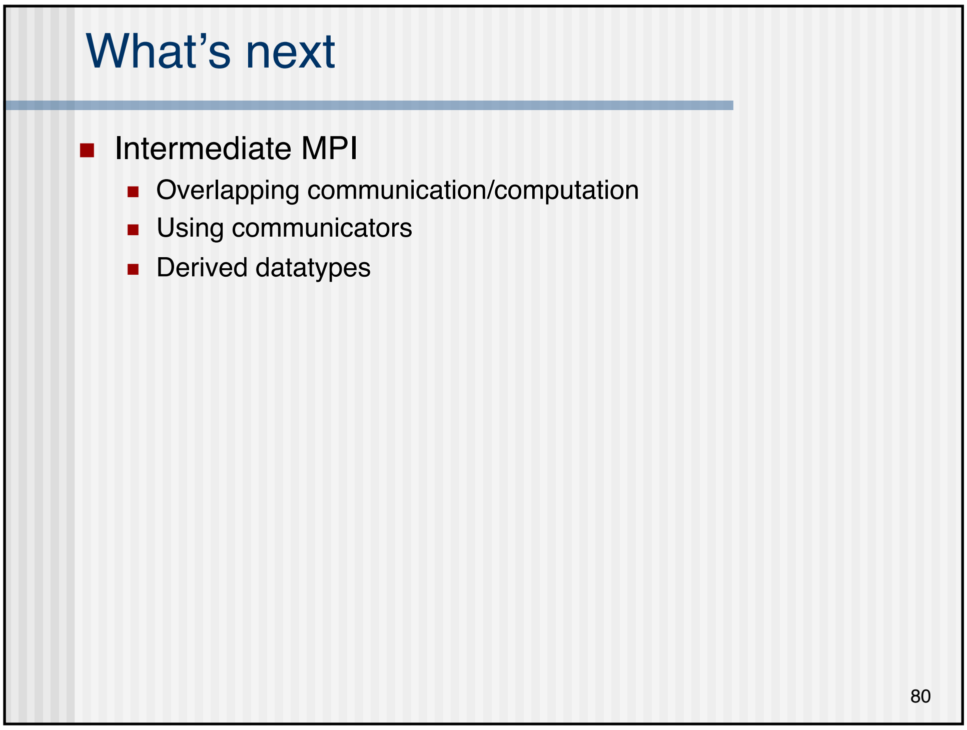
```
count = 1;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
                    MPI_COMM_WORLD);
```

task 0	task 1	task 2	task 3	
1	2	3	4	← sendbuf (before)
1	3	6	10	← recvbuf (after)



Summary

- Collective communication routines provide convenient calls for standard communication patterns
- Depending on the implementation they may be much more efficient than hand-coding (or not)
  - Synchronization overhead might be substantial
- Collective communication makes extensive use of groups/communicators



What's next

- Intermediate MPI
  - Overlapping communication/computation
  - Using communicators
  - Derived datatypes