Introduction to MPI

https://sites.google.com/lbl.gov/cs267-spr2023/

Slides from the CS267 collection

Outline

- **Programming Distributed Memory Machines using Message Passing**
  - **Overview of MPI**
  - **Basic send/receive use**
  - **Non-blocking communication**
  - **Collectives**

Programming Distributed Memory Machines with Message Passing

Slides by

Aydin Buluc, Jonathan Carter, Jim Demmel,

Bill Gropp, Kathy Yelick

Message Passing Libraries

- **All communication, synchronization require subroutine calls**
  - **No shared variables**
  - **Program run on a single processor just like any uniprocessor program, except for calls to message passing library**
- **Subroutines for**
  - **Communication**
    - **Pairwise or point-to-point: Send and Receive**
    - **Collectives all processor get together to**
      - Move data: Broadcast, Scatter/gather
      - Compute and move: sum, product, max, prefix sum, … of data on many processors
  - **Synchronization**
    - **Barrier**
    - **Initial version: no locks because there are no shared variables to protect**
  - **Enquiries**
    - **How many processes? Which one am I? Any messages waiting?**

Novel Features of MPI

- Communicators encapsulate communication spaces for library safety
- Datatypes reduce copying costs and permit heterogeneity
- Multiple communication modes allow precise buffer management
- Extensive collective operations for scalable global communication
- Process topologies permit efficient process placement, user views of process layout
- Profiling interface encourages portable tools

MPI References

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
  - Latest version MPI 4.0, released June 2021
- Other information on Web:
  - at http://www.mcs.anl.gov/research/projects/mpi/index.htm
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

Finding Out About the Environment

- **Two important questions that arise early in a parallel program are:**
  - **How many processes are participating in this computation?**
  - **Which one am I?**
- **MPI provides functions to answer these questions:**
  - `MPI_Comm_size` **reports the number of processes.**
  - `MPI_Comm_rank` **reports the *rank*, a number between 0 and size-1, identifying the calling process**

Hello (C)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

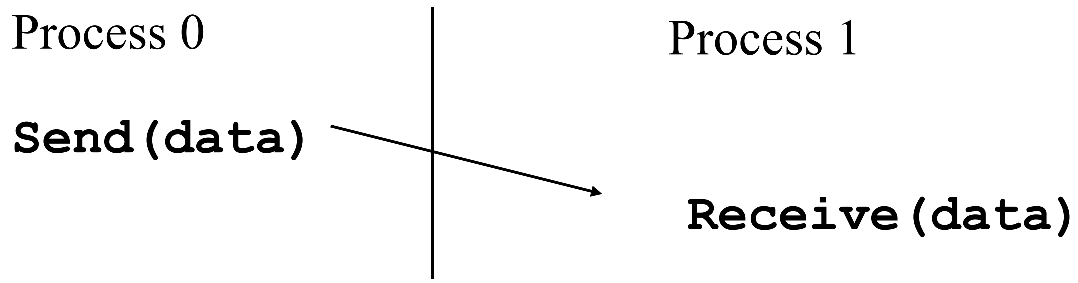
Notes on Hello World

- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI “job”
- Each statement executes independently in each process
  - including the `printf/print` statements
- The MPI-1 Standard does not specify how to run an MPI program, but many implementations provide

```
mpirun -np 4 a.out
```

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

Some Basic Concepts

- Processes can be collected into groups
- Each message is sent in a context, and must be received in the same context
  - Provides necessary support for libraries
- A group and context together form a communicator
- A process is identified by its rank in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called `MPI_COMM_WORLD`

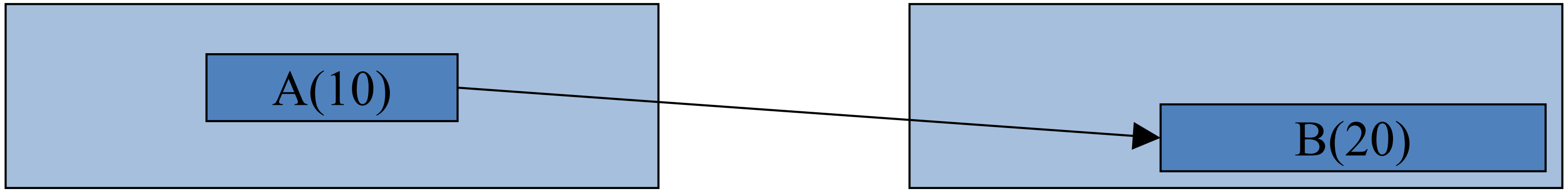
MPI Datatypes

- The data in a message to send or receive is described by a triple (address, count, datatype), where
- An MPI datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, in particular ones for subarrays
- May hurt performance if datatypes are complex

MPI Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying MPI_ANY_TAG as the tag in a receive
- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes

MPI Basic (Blocking) Send



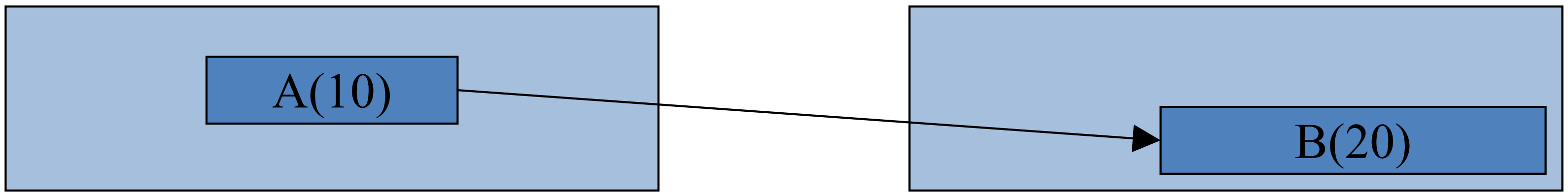
MPI_Send(A, 10, MPI_DOUBLE, 1, …)

MPI_Recv(B, 20, MPI_DOUBLE, 0, …)

`MPI_SEND(start, count, datatype, dest, tag, comm)`

- The message buffer is described by (`start, count, datatype`).
- The target process is specified by `dest`, which is the rank of the target process in the communicator specified by `comm`.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI Basic (Blocking) Receive



MPI_Send(A, 10, MPI_DOUBLE, 1, …)

MPI_Recv(B, 20, MPI_DOUBLE, 0, …)

`MPI_RECV(start, count, datatype, source, tag, comm, status)`

- Waits until a matching (both `source` and `tag`) message is received from the system, and the buffer can be used
- `source` is rank in communicator specified by `comm`, or `MPI_ANY_SOURCE`
- `tag` is a tag to be matched or `MPI_ANY_TAG`
- receiving fewer than `count` occurrences of `datatype` is OK, but receiving more is an error
- `status` contains further information (e.g. size of message)

A Simple MPI Program

```
#include “mpi.h”
#include <stdio.h>
int main( int argc, char *argv[])
{
  int rank, buf;
  MPI_Status status;
  MPI_Init(&argv, &argc);
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  /* Process 0 sends and Process 1 receives */
  if (rank == 0) {
    buf = 123456;
    MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  else if (rank == 1) {
    MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
              &status );
    printf( “Received %d\n”, buf );
  }

  MPI_Finalize();
  return 0;
}
```

Retrieving Further Information

- `Status` is a data structure allocated in the user's program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

MPI can be simple

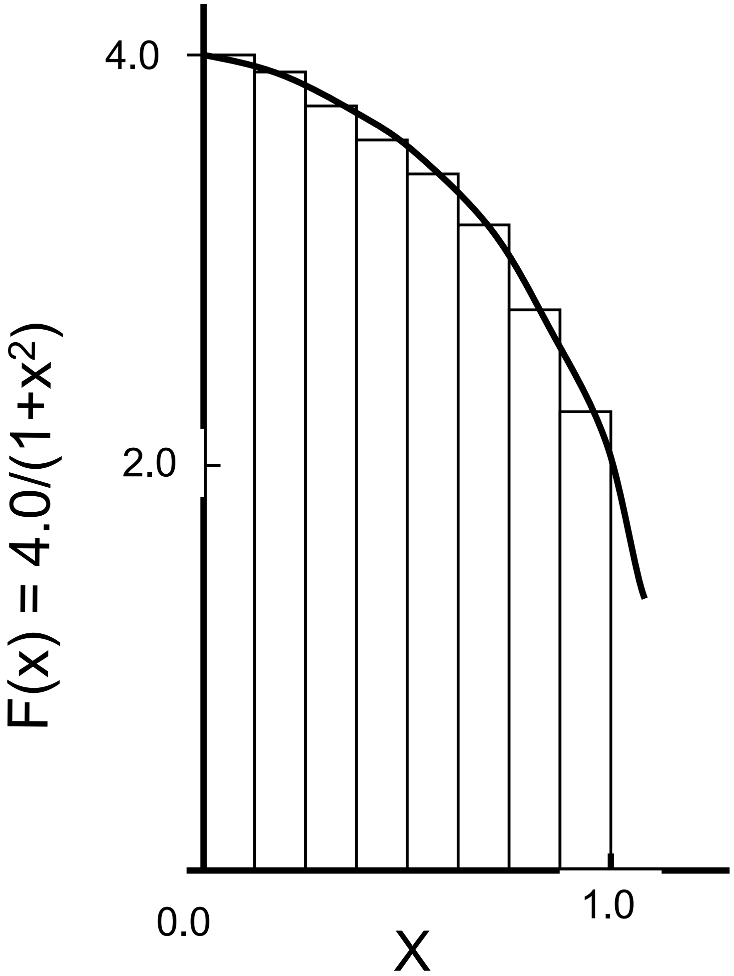
- **Claim: most MPI applications can be written with only 6 functions (although which 6 may differ)**

- Using point-to-point:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_SEND`
  - `MPI_RECEIVE`

- Using collectives:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_BCAST`
  - `MPI_REDUCE`

- **You may use more for convenience or performance**

PI redux: Numerical integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

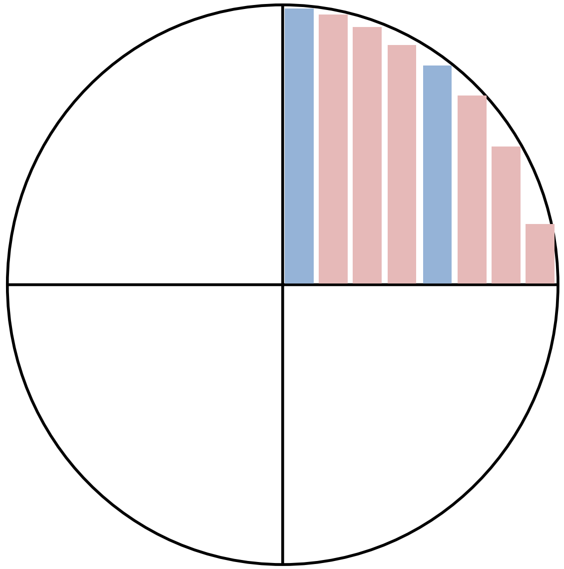
We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Example: Calculating Pi

E.g., in a 4-process run, each process gets every 4th interval. Process 0 slices are in red.



- Simple program written in a data parallel style in MPI
  - E.g., for a reduction (recall "data parallelism" lecture), each process will first reduce (sum) its own values, then call a collective to combine them
- Estimates pi by approximating the area of the quadrant of a unit circle
- Each process gets 1/p of the intervals (mapped round robin, i.e., a cyclic mapping)

Example: PI in C – 1/2

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

int main(int argc, char *argv[])

{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.141592653589793238462643;
   double mypi, pi, h, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   while (!done)  {
     if (myid == 0) {
       printf("Enter the number of intervals: (0 quits) ");
       scanf("%d",&n);
     }
     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
     if (n == 0) break;
```

Example: PI in C – 2/2

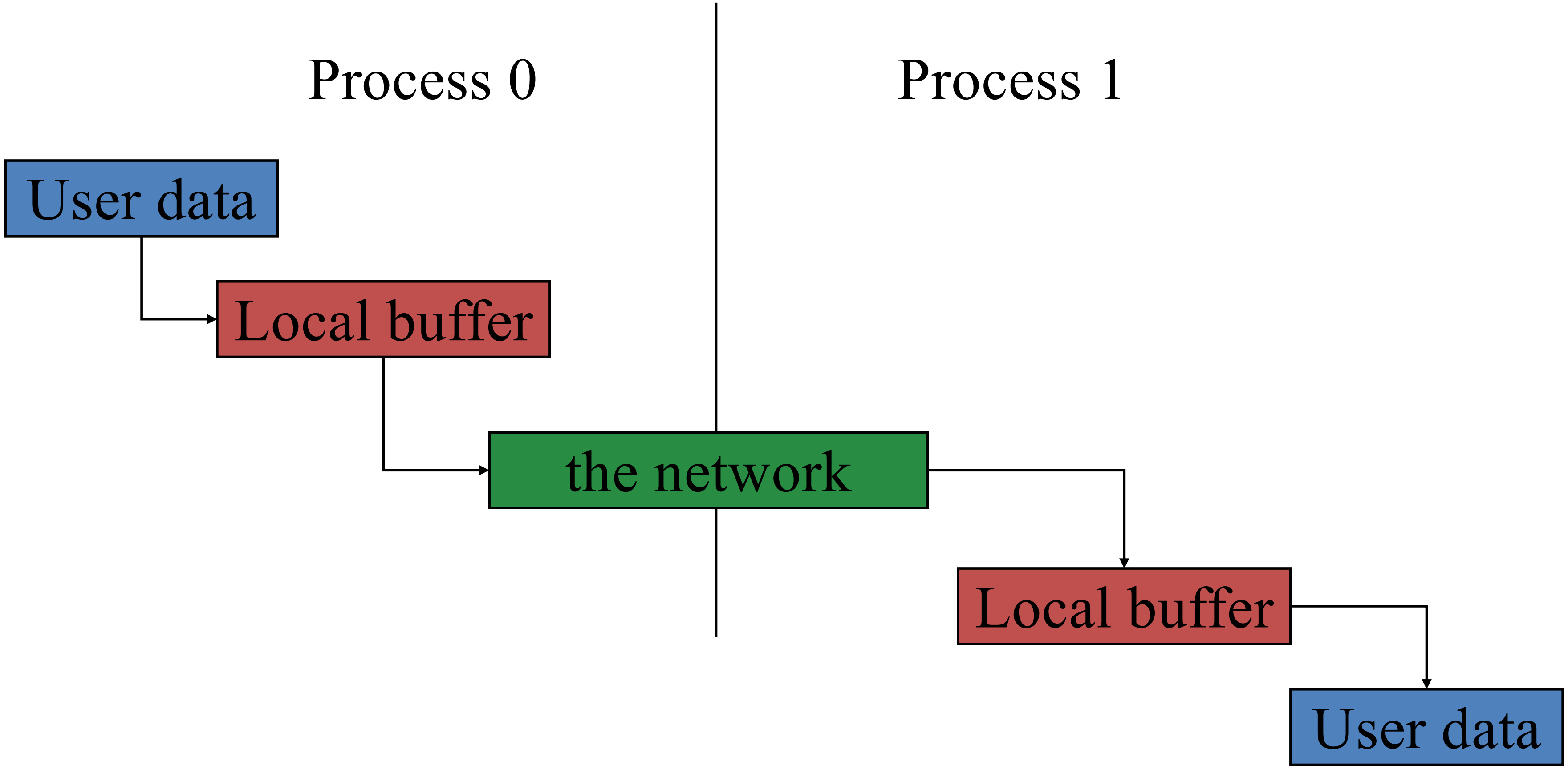
```
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi is approximately %.16f, Error is .16f\n",
             pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();

  return 0;

}
```

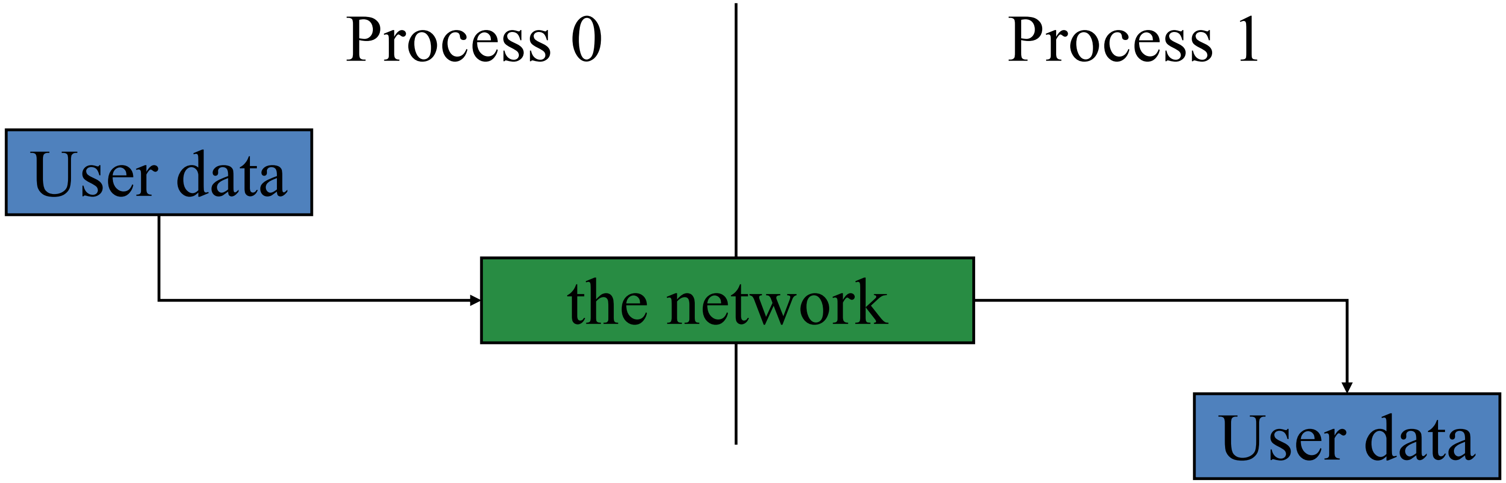
Buffers

- **When you send data, where does it go? One possibility is:**



Avoiding Buffering

- Avoiding copies uses less memory
- May use more or less time



This requires that **`MPI_Send`** wait on delivery, or that **`MPI_Send`** return before transfer is complete, and we wait later.

Slide source: Bill Gropp, ANL

Blocking and Non-blocking Communication

- So far we have been using *blocking* communication:
  - `MPI_Recv` does not complete until the buffer is full (available for use).
  - `MPI_Send` does not complete until the buffer is empty (available for use).
- Completion depends on size of message and amount of system buffering.

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

Process 0	Process 1
`Send(1)`	**`Send(0)`**
`Recv(1)`	**`Recv(0)`**

- This is called "unsafe" because it depends on the availability of system buffers in which to store the data sent until it can be received

Some Solutions to the "unsafe" Problem

- **Order the operations more carefully:**

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

- Supply receive buffer at same time as send:

Process 0	Process 1
`Sendrecv(1)`	`Sendrecv(0)`

More Solutions to the "unsafe" Problem

- **Supply own space as buffer for send**

Process 0	Process 1
`Bsend(1)`	**`Bsend(0)`**
`Recv(1)`	**`Recv(0)`**

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	**`Isend(0)`**
`Irecv(1)`	**`Irecv(0)`**
`Waitall`	**`Waitall`**

MPI's Non-blocking Operations

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on:

```
MPI_Request request;

MPI_Status status;

MPI_Isend(start, count, datatype,
    dest, tag, comm, &request);

MPI_Irecv(start, count, datatype,
    dest, tag, comm, &request);

MPI_Wait(&request, &status);
```

(each request must be Waited on)

- One can also test without waiting:

```
MPI_Test(&request, &flag, &status);
```

- Accessing the data buffer without waiting is undefined

Multiple Completions

- It is sometimes desirable to wait on multiple requests:

```
MPI_Waitall(count, array_of_requests,
  array_of_statuses)

MPI_Waitany(count, array_of_requests,
  &index, &status)

MPI_Waitsome(count, array_of_requests,
  array_of indices, array_of_statuses)
```

- There are corresponding versions of `test` for each of these.

Communication Modes

- **MPI provides multiple *modes* for sending messages:**
  - **Synchronous mode (`MPI_Ssend`): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)**
  - **Buffered mode (`MPI_Bsend`): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.**
  - **Ready mode (`MPI_Rsend`): user guarantees that a matching receive has been posted.**
    - **Allows access to fast protocols**
    - **undefined behavior if matching receive not posted**
- **Non-blocking versions (`MPI_Issend`, etc.)**
- **`MPI_Recv` receives messages sent in any mode.**
- **See www.mpi-forum.org for summary of all flavors of send/receive**