CSE 6220
Introduction to High Performance Computing
Introduction to MPI

Ümit V. Çatalyürek

School of Computational Science and Engineering

Georgia Institute of Technology

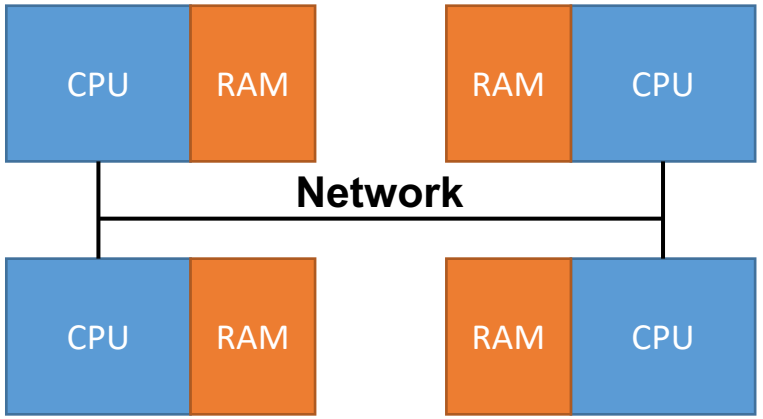
Thanks to Patrick Flick and Jaroslaw Zola for the slides.

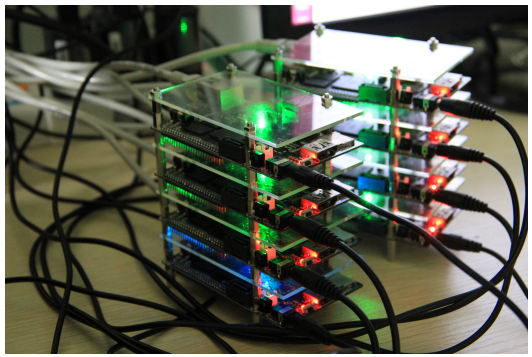
What is MPI?

Georgia Institute of Technology

Message **P**assing **I**nterface

- **Standard** for parallel programming via message passing
- For parallel computers and distributed memory systems
  - Computing clusters
  - Supercomputers
  - Heterogeneous systems
  - Multi-core/processor systems
- Designed to be portable, flexible, and efficient

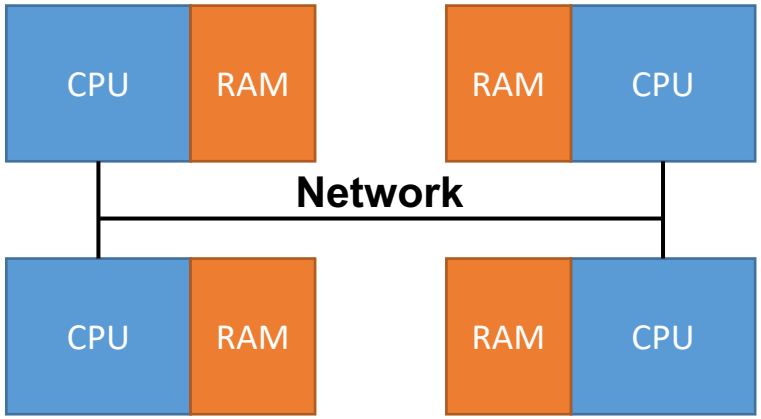




What is MPI?

Message **P**assing **I**nterface

- **Standard** for parallel programming via message passing
- Not a library or software package
- Specifies API for C/C++ and Fortran
- Many Implementations:
  - **OpenMPI**
  - MPICH
  - MVAPICH (MPI via Infiniband)
  - Proprietary vendor specific implementations
    - Intel, Cray, IBM, Microsoft, Oracle
  - E.g. On the IBM Blue Gene, many MPI calls are implemented in Hardware



MPI Implementation

Every MPI implementation must provide three components (we consider only C/C++):

1. MPI library with `mpi.h` header
2. Compiler wrapper: `mpicc` or `mpicxx`
3. Runtime system with `mpirun` tool (sometimes `mpiexec`)

All three are essential:

- MPI library to write parallel programs
- Compiler wrapper (e.g. `mpicc`) to compile MPI programs
- `mpirun` to run parallel MPI programs

Structure of an MPI program

p1.cpp:

```
#include <mpi.h>

int main(int argc, char * argv[])
{
    // initialize MPI
    MPI_Init(&argc, &argv);

    // ...

    // finished with MPI
    MPI_Finalize();
    return 0;
}
```

Every MPI program **must** include `mpi.h`

Every MPI program **must call** `MPI_Init` once, before any other MPI calls

All MPI calls must be within an initialized MPI program

Every MPI program **must call** `MPI_Finalize` once, before exiting the application

Structure of an MPI program

p1.cpp:

```
#include <mpi.h>

int main(int argc, char * argv[])
{
    // initialize MPI
    MPI_Init(&argc, &argv);

    // ...

    // finished with MPI
    MPI_Finalize();
    return 0;
}
```

Compiling and running:

```
$ mpicxx p1.cpp -o p1

$ mpiexec -np 4 ./p1
```

Communicators

Key concept in MPI: **Communicators**

- ***Communicator:*** group of MPI processes
- In a communicator, each process has a unique rank
- Ranks are integers from *0* to *(p − 1)*, where *p* is the total number of processes in the communicator
- Special Communicators:
  - `MPI_COMM_WORLD` (all processes)
  - `MPI_COMM_SELF`
- Operations:
  - **`MPI_Comm_size`**
  - **`MPI_Comm_rank`**
  - `MPI_Comm_split`
  - `MPI_Comm_create`
  - `MPI_Comm_free`
  - …

Communicators: Example

p2.cpp:

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

int main(int argc, char * argv[]) {
    // initialize MPI
    MPI_Init(&argc, &argv);

    // get communicator size and ranks
    MPI_Comm comm = MPI_COMM_WORLD;
    int rank, size;
    MPI_Comm_size(comm, &size);
    MPI_Comm_rank(comm, &rank);

    // print rank and size
    printf("Hello from rank %d/%d.\n", rank, size);

    // finished with MPI
    MPI_Finalize();
    return 0;
}
```

```
int MPI_Comm_size(MPI_Comm comm, int* size);
int MPI_Comm_rank(MPI_Comm comm, int* rank);
```

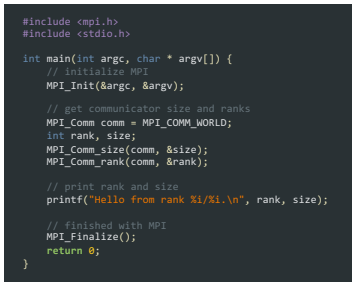
MPI_Comm object initialized to contain all processes (WORLD)

Get total number of processes in communicator, and the current process' rank

Communicators: Example

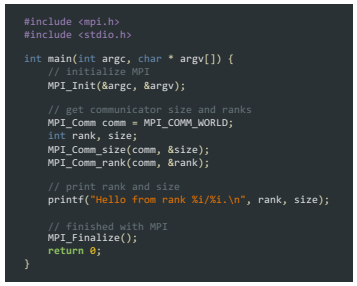
```
mpiexec -np 4 ./p2
```

Process 0



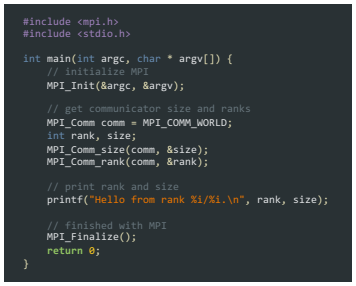
```
size = 4
rank = 0
```

Process 1



```
size = 4
rank = 1
```

Process 2



```
size = 4
rank = 2
```

Process 3

```
size = 4
rank = 3
```

```
Hello from rank 1/4.
```

```
Hello from rank 0/4.
```

```
Hello from rank 3/4.
```

```
Hello from rank 2/4.
```

Point-to-Point Communication

Point to point communication

- Sending messages from one process to another
- Always involves both processes (***source*** and ***destination***)


```
MPI_Send(Message, dest=Y, comm)
```

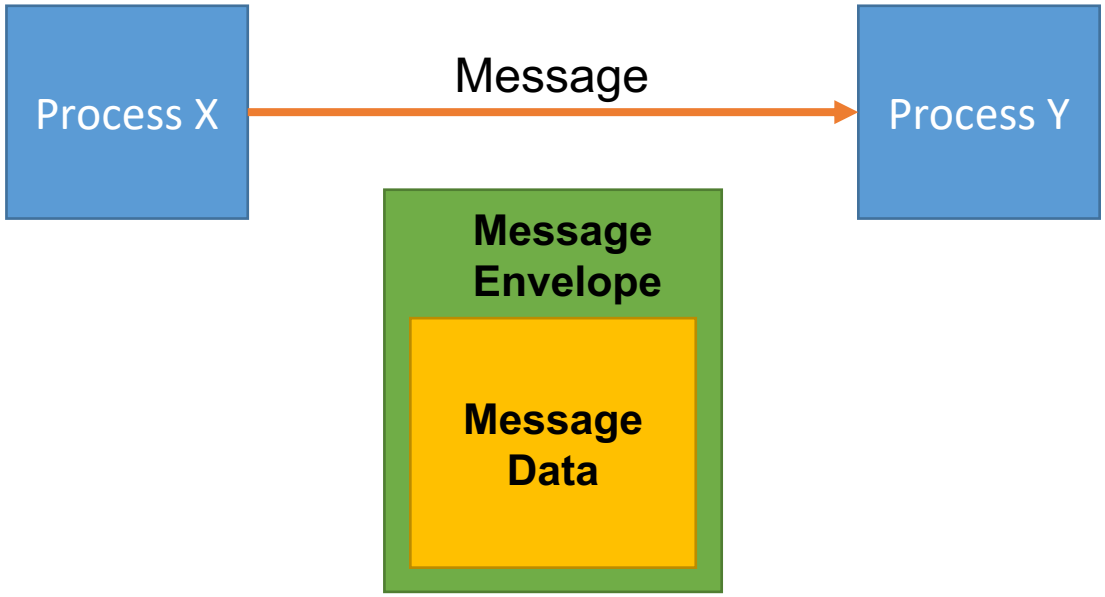
```
MPI_Receive(Message, src=X, comm)
```

Source and **destination** ranks are always with respect to the given communicator ***comm***

(NOTE: These are not the actual APIs of MPI_Send/MPI_Receive)

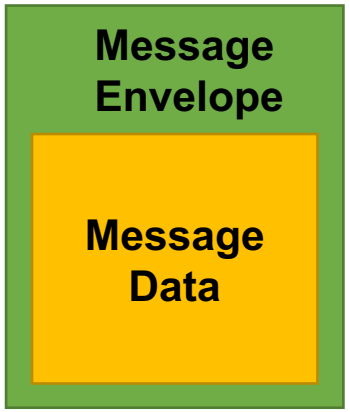
Point to point communication

- Sending messages from one process to another
- Always involves both processes (***source*** and ***destination***)



- Every message consists of a *Message Envelope* and the *Message Data*

Message Data

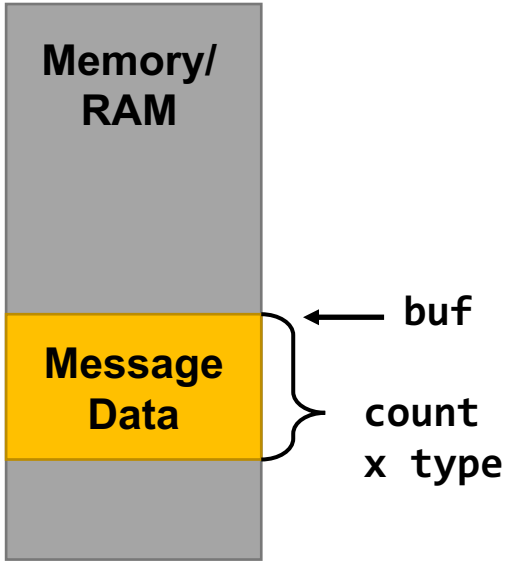


The ***Message Data*** for each message is described by:

1. Memory buffer storing data: `void* buf,`
2. Number of objects in the buffer: `int count,`
3. Data type of the data in the buffer: `MPI_Datatype type`

- MPI defines **`MPI_Datatype`** for all built-in C/C++ types
  - `MPI_CHAR, MPI_INT, MPI_UNSIGNED, MPI_FLOAT, MPI_DOUBLE,…`
  - `See the standard for complete list`

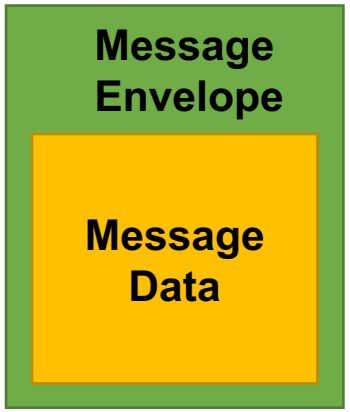
- Effectively, the Message Data is described as a block of memory:
  - **`buf`** is a pointer to where the data lies in memory
  - **`count`** and **`type`** define how large the memory block is (e.g. 8 x MPI_INT = 32 bytes)



Message Envelope

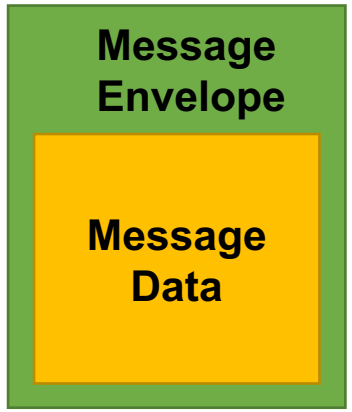
The ***Message Envelope*** contains:

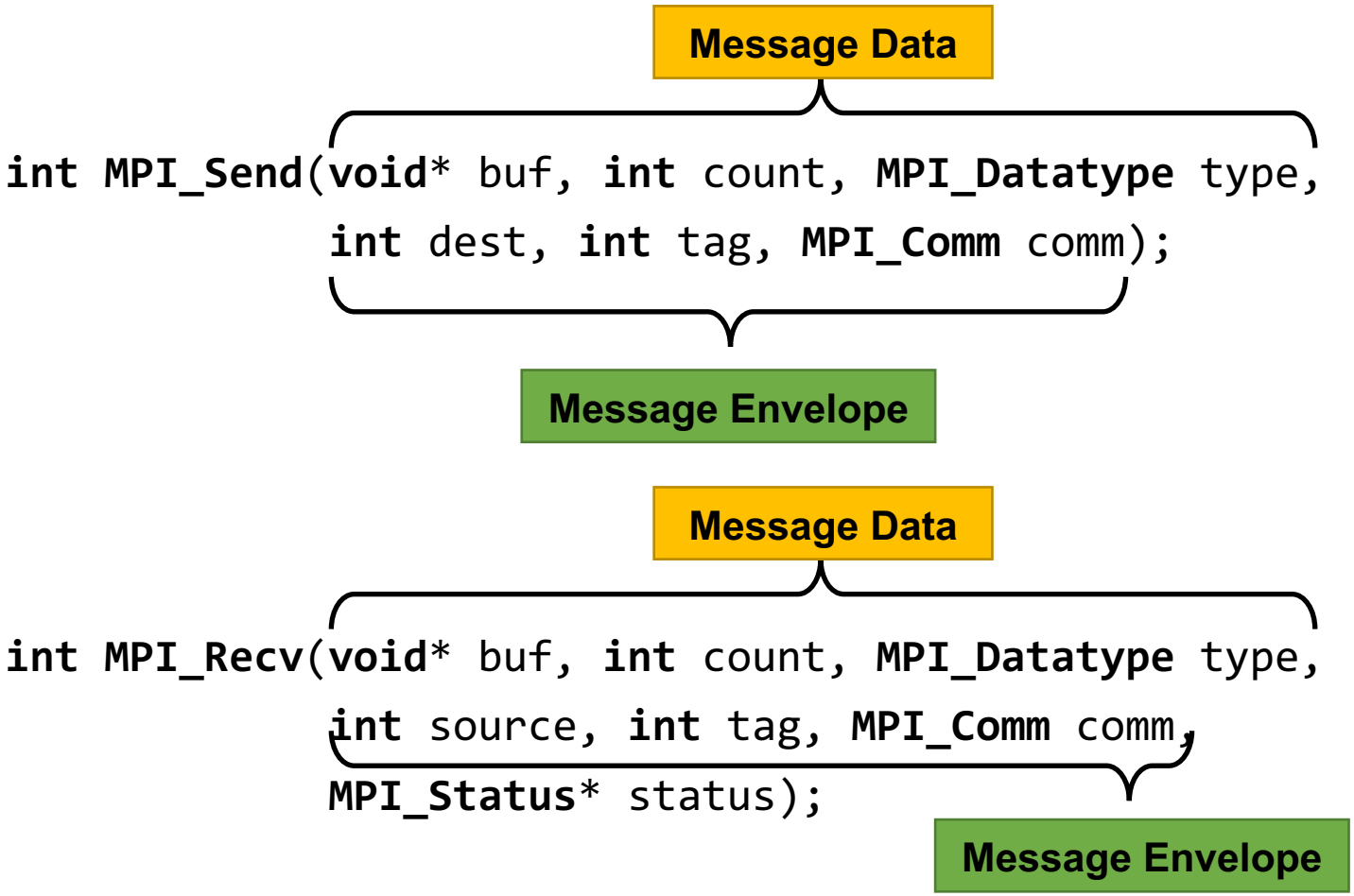
- **source rank:** `int src,`
- **destination rank:** `int dest,`
- **tag**: integer used to distinguish messages `int tag,`
- **Communicator:** universe of communication `MPI_Comm comm`



MPI_Send/MPI_Recv

Function signatures for `MPI_Send` and `MPI_Recv`:





Example Send/Recv

p3.cpp:

```
#include <mpi.h>
#include <iostream>

int main(int argc , char* argv []) {
    int rank;
    MPI_Init (&argc , &argv);
    MPI_Comm_rank (MPI_COMM_WORLD , &rank );
    const int N = 32;
    int tab[N];
    if (rank == 0) {
        tab[N - 1] = 13;
        MPI_Send(tab, N, MPI_INT,
                 1, 111, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Status stat;
        MPI_Recv(tab, N, MPI_INT,
                 0, 111, MPI_COMM_WORLD, &stat);
        std::cout << tab[N - 1] << std::endl;
    }
    return MPI_Finalize();
}
```

Send N integers from rank 0 to rank 1.
Blocks till everything is sent.

Receive N integers.
Blocks till everything is received.

```
Output: 13
```

Message Matching

- Each send operation **must be matched** by a corresponding receive operation
- Messages are matched via their **envelope:**
  - communicator
  - source rank
  - tag
  - **NOT** by type or size
- Thus: for a receive to succeed, a message needs to match the communicator, tag, and rank

Blocking:

- An `MPI_Recv` operation **blocks** until a matching message is received
- An `MPI_Send` operation **may block** until the message has been received by the destination process

Examples: Message Matching

p4.cpp

```
#include <mpi.h>

int main(int argc , char* argv []) {
    int rank;
    MPI_Init(&argc , &argv);
    MPI_Comm_rank(MPI_COMM_WORLD , &rank);
    int x;
    if (rank == 0) {
        x = 13;
        MPI_Send(&x, 1, MPI_INT,
                 1, 111, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Status stat;
        MPI_Recv(&x, 1, MPI_INT,
                 0, 222, MPI_COMM_WORLD, &stat);
    }
    return MPI_Finalize();
}
```

What's wrong?

Tags are not matching
111 != 222

MPI program will deadlock!

Examples: Message Matching

Georgia Institute of Technology

```
#include <mpi.h>
int main(int argc, char* argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int x[16];
    float y[16];
    if (rank == 0) {
        x[13] = 13;
        y[13] = 0.13f;
        MPI_Send(x, 16, MPI_INT,
                 1, 111, MPI_COMM_WORLD);
        MPI_Send(y, 16, MPI_FLOAT,
                 1, 222, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Status stat;
        MPI_Recv(y, 16, MPI_FLOAT,
                 0, 222, MPI_COMM_WORLD, &stat);
        MPI_Recv(x, 16, MPI_INT,
                 0, 111, MPI_COMM_WORLD, &stat);
    }
    return MPI_Finalize();
}
```

What's wrong?

Blocks until message with tag 111 is received.

Blocks until message with tag 222 is sent.

MPI program will deadlock!

Examples: Message Matching

Georgia Institute of Technology

Correct!

```
#include <mpi.h>
int main(int argc , char* argv []) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int x, y;
    if (rank == 0) {
        x = 13;
        MPI_Send (&x, 1, MPI_INT,
                  1, 111, MPI_COMM_WORLD);
        x = 113;
        MPI_Send (&x, 1, MPI_INT,
                  1, 111, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Status stat;
        MPI_Recv (&x, 1, MPI_INT,
                  0, 111, MPI_COMM_WORLD, &stat);
        MPI_Recv (&y, 1, MPI_INT,
                  0, 111, MPI_COMM_WORLD, &stat);
    }
    return MPI_Finalize();
}
```

x = 13
y = 113

ANY

- Useful trick:
  - We can use any combination of **`tag = MPI_ANY_TAG`** and/or **`source = MPI_ANY_SOURCE`** in **`MPI_Recv`** to receive any message from any processor
  - We can use the **`MPI_Status`** structure to check envelope of the received message

- Example:

```
int x;
MPI_Status stat;
MPI_Recv(&x, 1, MPI_INT,
         MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
int source = stat.MPI_SOURCE;
int tag = stat.MPI_TAG;
```

Example: any tag, any src

```
#include <mpi.h>
#include <iostream>

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 );
    int x;
    if (rank == 0) {
        MPI_Status stat;
        for (int i = 1; i < size; ++i) {
            MPI_Recv(&x, 1, MPI_INT,
                     MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
                     &stat);
            std::cout << "From: " << stat.MPI_SOURCE << " "
            << "Tag: " << stat.MPI_TAG << std::endl;
        }
    } else {
        x = rank;
        MPI_Send(&x, 1, MPI_INT,
                 0, size - rank, MPI_COMM_WORLD);
    }
    return MPI_Finalize();
}
```

Receives any one of the messages. No order guaranteed.

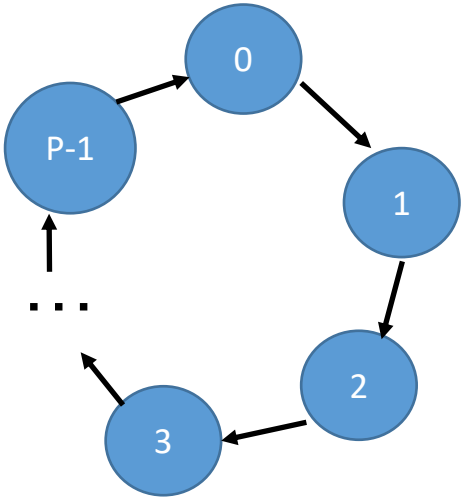
Message Ordering

The **MPI** standard defines the order in which messages will be received, when ANY_TAG or ANY_SRC is used:

- If messages originate from **different processors**, the order in which they are received is **arbitrary**
- If messages have **different tags**, the order in which they are received is **arbitrary**
- Only if two or more messages have the **same source AND** the **same tag**, their **order is preserved**
  - This means: they are received in the same order in which they were sent

Cyclic dependencies

- Let's say we want to send messages in a cyclic dependency:
  - Processor `i` sends a message to `(i+1) mod p`



Example:

```
MPI_Send(&x, 1, MPI_INT,
         (rank+1) % size, 13, MPI_COMM_WORLD);
MPI_Recv(&y, 1, MPI_INT,
         MPI_ANY_SOURCE, 13, MPI_COMM_WORLD, &stat);
```

What's wrong?

May block till message is received, MPI_Recv never called

Reverse?

```
MPI_Recv(&y, 1, MPI_INT,
         MPI_ANY_SOURCE, 13, MPI_COMM_WORLD, &stat);
MPI_Send(&x, 1, MPI_INT,
         (rank+1) % size, 13, MPI_COMM_WORLD);
```

Same problem.

Non-blocking Communication

- Enter: Non-blocking Communication
  - **`MPI_Isend`** and **`MPI_Irecv`** will initiate transfers but **not block** until they succeed
  - Enables:
    - Dead-lock avoidance
    - Hiding latency by overlapping computation and communication
  - Same parameters as `MPI_Send`, `MPI_Recv` but additional output parameter:

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

int MPI_Irecv(void* buf, int count, MPI_Datatype type,
              int source, int tag, MPI_Comm comm,
              MPI_Request* req);
```

Non-blocking Communication

Georgia Institute of Technology

Communication Completion

- Two functions can be used to check for communication status:

```
int MPI_Wait(MPI_Request* req, MPI_Status* status);
int MPI_Test(MPI_Request* req, int* flag, MPI_Status* status);
```

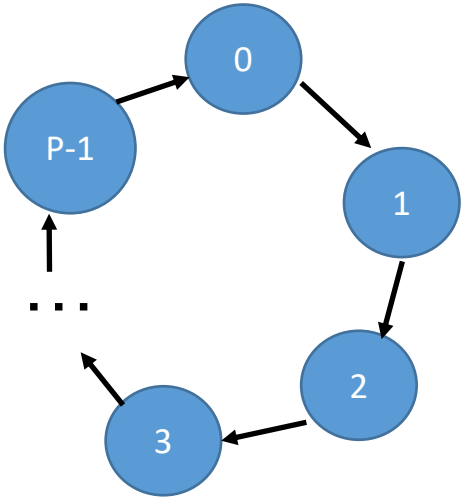
- `MPI_Wait` will block till the send/receive operation is completed.
- `MPI_Test` returns a flag of either 0 or 1 depending on whether the send/receive operation is completed

```
MPI_Status stat;
MPI_Request req;
MPI_Isend(..., &req);
MPI_Wait(&req, &stat);
```

```
MPI_Send(... );
```

Cyclic dependencies

- Let's say we want to send messages in a cyclic dependency:
  - Processor `i` sends a message to `(i+1) mod p`



Example:

```
MPI_Send(&x, 1, MPI_INT,
         (rank+1) % size, 13, MPI_COMM_WORLD);
MPI_Recv(&y, 1, MPI_INT,
         MPI_ANY_SOURCE, 13, MPI_COMM_WORLD, &stat);
```

What's wrong?

May block till message is received, MPI_Recv never called

Using Non-blocking receive:

```
MPI_Request req;
MPI_Irecv(&y, 1, MPI_INT,
         MPI_ANY_SOURCE, 13, MPI_COMM_WORLD, &req);
MPI_Send(&x, 1, MPI_INT,
         (rank+1) % size, 13, MPI_COMM_WORLD);
MPI_Wait(&req, &stat);
```

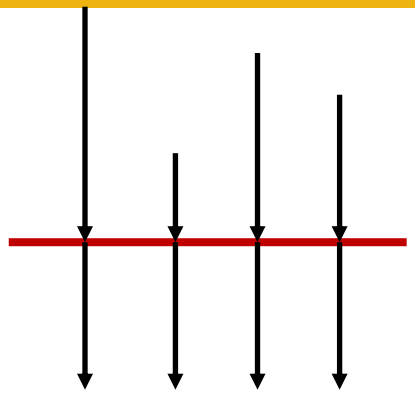
Correct!

Collective Communication

Collective Communication

- All processes within a communicator must participate
- All collective operations are **blocking**
- Except of `MPI_Barrier`, no synchronization can be assumed
- All collective operations are guaranteed to not interfere with point-to-point messages
- Collective operations can be implemented using only point-to-point calls
- In practice, they are optimized to use various hardware (interconnect) properties

Barrier



- Block until all processes called the barrier:

```
int MPI_Barrier(MPI_Comm comm);
```

- Establishes full synchronization

Side Note: Measuring Time

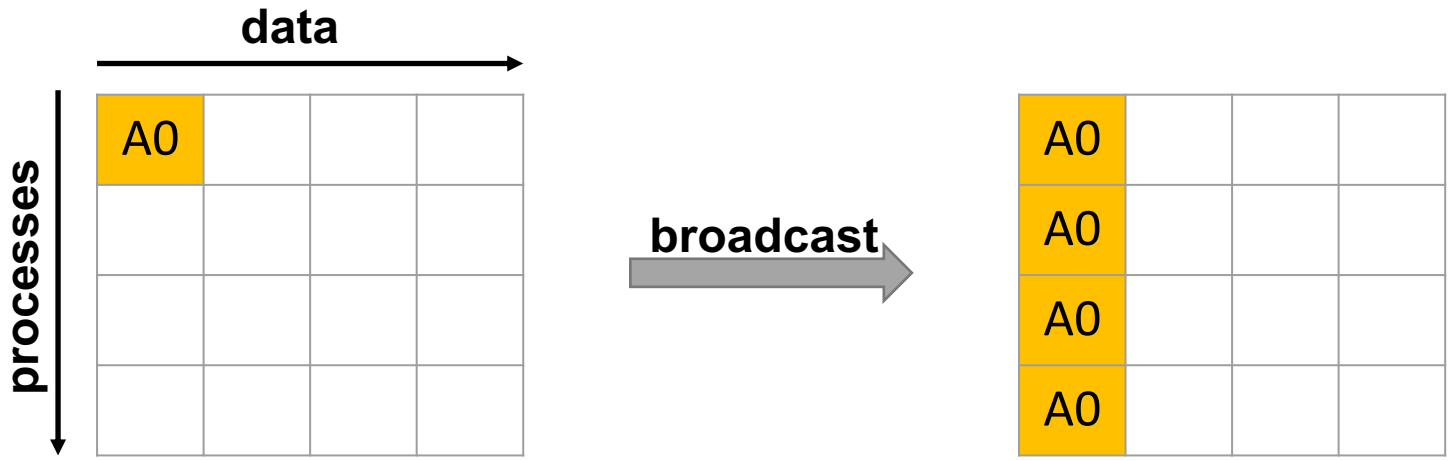
MPI provides a function to check execution time:

```
double MPI_Wtime();
```

It returns time elapsed since “some point” in the past.

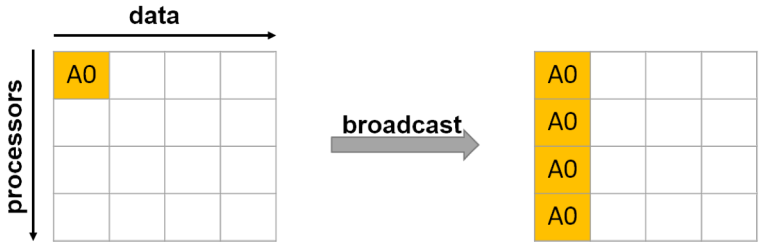
```
#include <mpi.h>
#include <iostream>

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    double t0 = MPI_Wtime();
    // ...
    MPI_Barrier(MPI_COMM_WORLD);
    double t1 = MPI_Wtime();
    std::cout << (t1 - t0) << std::endl;
    return MPI_Finalize();
}
```

Broadcast



Broadcast

Broadcast a message from one process(=root) to all other processes




```
int MPI_Bcast(void* buf, int count, MPI_Datatype type,
              int root, MPI_Comm comm);
```

Example:

```
#include <mpi.h>
#include <iostream>
int main(int argc , char* argv []) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    double x = 0.0;
    if (rank == 0) x = -13.13;
    MPI_Bcast(&x, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    std::cout << x << std::endl;
    return MPI_Finalize ();
}
```

Prints -13.13 on every process.

MPI Programming Assignments

Programming Assignments

- During the semester, there will be multiple programming assignments
- Each one will test different aspects of MPI programming
- C/C++ programming is required

- You will run your programs on instructional HPC resources
  - Information regarding how to use the cluster and few others can be found in Canvas under “Files/Additional Material/MPI”

Getting Set Up

- Before running your programs on one of the Georgia Tech clusters, you should make sure it works locally (on your desktop/laptop).
- We are going to use **OpenMPI (or mvapich)** in this class
  - Can easily be installed on Debian/Ubuntu via *`apt-get`* or on Mac via *`homebrew`*
  - You could install Ubuntu (or your favorite Linux distribution) on either your machine or inside a virtual machine/virtual box
  - All clusters are running Linux
  - MPI support is best on Linux based systems

- **Preparation** for the programming assignments:
  - (Install Ubuntu/Linux)
  - Install OpenMPI
  - Compile and run some of the small sample programs from before
  - Play around with MPI and C/C++ a little

Additional Documentation

Additional Documentation

- A good MPI tutorial: https://computing.llnl.gov/tutorials/mpi/
- Official MPI Standard Documents: http://www.mpi-forum.org/docs/
- MPICH web-based MPI reference: http://www.mpich.org/documentation/guides/
- PACE Training: http://www.pace.gatech.edu/training

C++ and MPI

- MPI used to include C++ bindings, but these are **deprecated** as of MPI 2.2
- Don't use the C++ bindings!

- Caveats when using C++:
  - The buffer argument in MPI calls expects a void pointer to the raw, contiguous data.
  - Only contiguous containers can be sent via MPI: **std::vector** (can't easily send std::map, std::list, std::deque, etc).
  - Sending a `std::vector`:

```
MPI_Send(&vec[0], vec.size(), …)
```

  - Space has to be allocated prior to receiving, i.e., you have to know how many elements you will receive

Georgia Institute of Technology

Sending and Receiving std::vector

- Assuming the receiving end does not yet know the total size of the vector
- First send the size, then the data:

```
// sending a vector (send size first)
std::vector<int> vec(some_size);
int send_size = vec.size();
MPI_Send(&send_size, 1, MPI_INT, dest, 1234, MPI_COMM_WORLD);
MPI_Send(&vec[0], send_size, MPI_INT, dest, 1234, MPI_COMM_WORLD);
```

- On the receiving side:

```
// receiving a vector (receive size first)
MPI_Status stat;
std::vector<int> recv_buf;
int recv_size;
MPI_Recv(&recv_size, 1, MPI_INT, src, 1234, MPI_COMM_WORLD, &stat);
recv_buf.resize(recv_size);
MPI_Recv(&recv_buf[0], recv_size, MPI_INT, src, 1234, MPI_COMM_WORLD, &stat);
```

Questions?