Lecture 20

On the ugrad machine:

use mpich

Use the package from

"/cs/local/lib/pkg"

Use can also download and install it yourself from mpich.org

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **MPI_Init**
  - **MPI_Finalize**
  - **MPI_Comm_size**
  - **MPI_Comm_rank**
  - **MPI_Send**
  - **MPI_Recv**

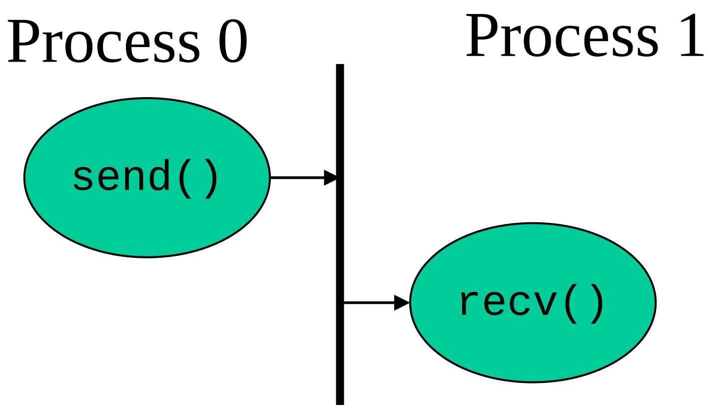
Simple Hello (C)

```
#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;
}
```

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?

Gropp, Lusk

MPI Basic (Blocking) Send

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.

Gropp, Lusk

MPI Basic (Blocking) Receive

MPI_Recv(start, count, datatype, source, tag, comm, status*)

- Waits until a matching (on **`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`**.
- **`status`** structure contains further information
- Receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error.

Gropp, Lusk

Identifying Processes

- MPI Communicator
  - Defines a *group* (set of ordered processes) and a *context* (a virtual network)
- Rank
  - Process number within the group
  - `MPI_ANY_SOURCE` will receive from any process
- Default communicator
  - `MPI_COMM_WORLD` the whole group

Identifying Messages

- An MPI Communicator defines a virtual network, `send/recv` pairs must use the same communicator
- `send/recv` routines have a *tag* (integer variable) argument that can be used to identify a message, or screen for a particular message.
  - `MPI_ANY_TAG` will receive a message with any tag

Identifying Data

- Data is described by a triple (address, type, count)
  - For `send`, this defines the message
  - For `recv`, this defines the size of the receive buffer
- Amount of data received, source, and tag available via *status* data structure
  - Useful if using `MPI_ANY_SOURCE`, `MPI_ANY_TAG`, or unsure of message size (must be smaller than buffer)

MPI Types

- Type may be recursively defined as:
  - An MPI predefined type
  - A contiguous array of types
  - An array of equally spaced blocks
  - An array of arbitrary spaced blocks
  - Arbitrary structure
- Each user-defined type constructed via an MPI routine, e.g. `MPI_TYPE_VECTOR`

MPI Predefined Types

C:	Fortran:
MPI_INT	MPI_INTEGER
MPI_FLOAT	MPI_REAL
MPI_DOUBLE	MPI_DOUBLE_PRECISION
MPI_CHAR	MPI_CHARACTER
MPI_UNSIGNED	MPI_LOGICAL
MPI_LONG	MPI_COMPLEX

Language Independent:

MPI_BYTE

MPI Types

- Explicit data description is useful:
  - Simplifies programming, e.g. send row/column of a matrix with a single call
  - Heterogeneous machines
  - May improve performance
    - Reduce memory-to-memory copies
    - Allow use of scatter/gather hardware
  - May hurt performance
    - User packing of data likely faster

Point-to-point Example

Process 0

```
#define TAG 999
float a[10];
int dest=1;
MPI_Send(a, 10,
MPI_FLOAT, dest, TAG,
MPI_COMM_WORLD);
```

Process 1

```
#define TAG 999
MPI_Status status;
int count;
float b[20];
int sender=0;
MPI_Recv(b, 20,
MPI_FLOAT, sender, TAG,
MPI_COMM_WORLD,
&status);
MPI_Get_count(&status,
MPI_FLOAT, &count);
```

MPI Basic (Standard) Send

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.

Gropp, Lusk

MPI Basic (Standard) Receive

MPI_Recv(start, count, datatype, source, tag, comm, status*)

- Waits until a matching (on **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**.
- **status** structure contains further information
- Receiving fewer than **count** occurrences of **datatype** is OK, but receiving more is an error.

Gropp, Lusk

MPI Status Data Structure

- In C

```
MPI_Status status;
int recvd_tag, recvd_from, recvd_count;
recvd_tag = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, MPI_INT,
  &recvd_count);
```

MPI's Non-blocking Operations

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on. (Posts a send/receive)

```
MPI_Request request;
MPI_Isend(start, count, datatype,
    dest, tag, comm, &request)
MPI_Irecv(start, count, datatype,
    dest, tag, comm, &request)
MPI_Wait(&request, &status)
```

- One can also test without waiting:

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

Example

```
#define MYTAG 123
  #define WORLD MPI_COMM_WORLD
  MPI_Request request;
  MPI_Status status;
Process 0:
  MPI_Irecv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD,
  &request)
  MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD)
  MPI_Wait(&request, &status)
Process 1:
  MPI_Irecv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD,
  &request)
  MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD)
  MPI_Wait(&request, &status)
```

Using Non-Blocking Send

Also possible to use non-blocking send:

- "status" argument to **MPI_Wait** doesn't return useful info here.

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;
p=1-me; /* calculates partner in exchange */
```

Process 0 and 1:

```
MPI_Isend(A, 100, MPI_DOUBLE, p, MYTAG, WORLD,
          &request)
MPI_Recv(B, 100, MPI_DOUBLE, p, MYTAG, WORLD,
                &status)
MPI_Wait(&request, &status)
```

Non-Blocking Gotchas

- Obvious caveats:
  - 1. You may not modify the buffer between Isend() and the corresponding Wait(). Results are undefined.
  - 2. You may not look at or modify the buffer between Irecv() and the corresponding Wait(). Results are undefined.
  - 3. You may not have two pending Irecv()s for the same buffer.
- Less obvious:
  - 4. You may not *look* at the buffer between Isend() and the corresponding Wait().
  - 5. You may not have two pending Isend()s for the same buffer.
- **Why the isend() restrictions?**
  - Restrictions give implementations more freedom, e.g.,
    - Heterogeneous computer with differing byte orders
    - Implementation swap bytes in the original buffer

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.

Multiple completion

- Source of non-determinism (new issues fairness?), process what is ready first
- Latency hiding, parallel slack
- Still need to poll for completion, {do some work; check for comm}*
- Alternative: multiple threads or co-routine like support

MPI point to point routines

MPI_Send	Standard send
MPI_Recv	Standard receive
MPI_Bsend	Buffered send
MPI_Rsend	Ready send
MPI_Ssend	Synchronous send
MPI_Ibsend	Nonblocking, buffered send
MPI_Irecv	Nonblocking receive
MPI_Irsend	Nonblocking, ready send
MPI_Isend	Nonblocking send
MPI_Issend	Nonblocking synchronous send
MPI_Sendrecv	Exchange
MPI_Sendrecv_replace	Exchange, same buffer
MPI_Start	Persistent communication

Communication Modes

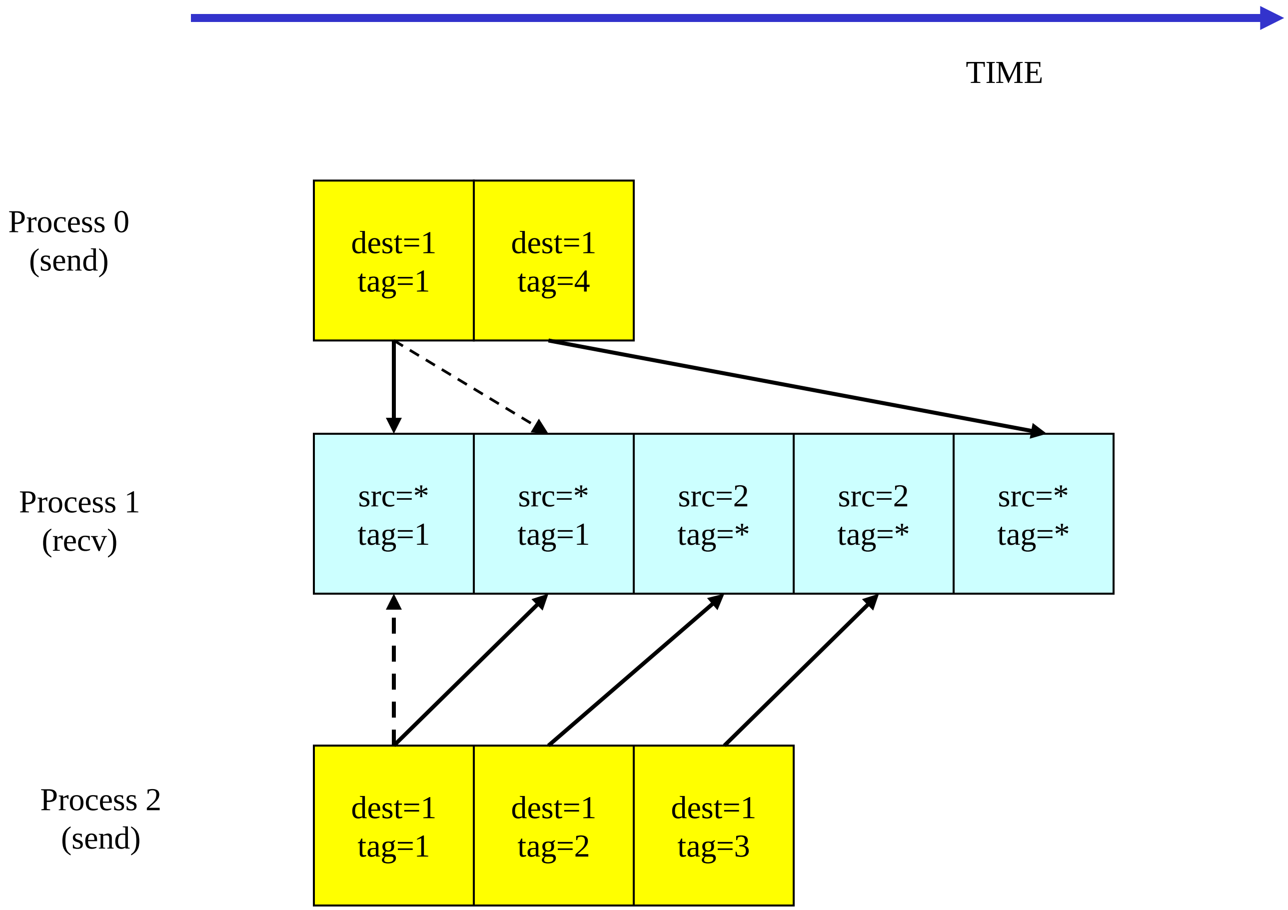
- Standard
  - Usual case (system decides)
  - **MPI_Send**, **MPI_Isend**
- Synchronous
  - The operation does not complete until a matching receive has started copying data into its receive buffer. (no buffers)
  - **MPI_Ssend**, **MPI_Issend**
- Ready
  - Matching receive already posted. (0-copy)
  - **MPI_Rsend**, **MPI_Irsend**
- Buffered
  - Completes after being copied into user provided buffers (Buffer_attach, Buffer_detach calls)
  - **MPI_Bsend**, **MPI_Ibsend**

Point to point with modes

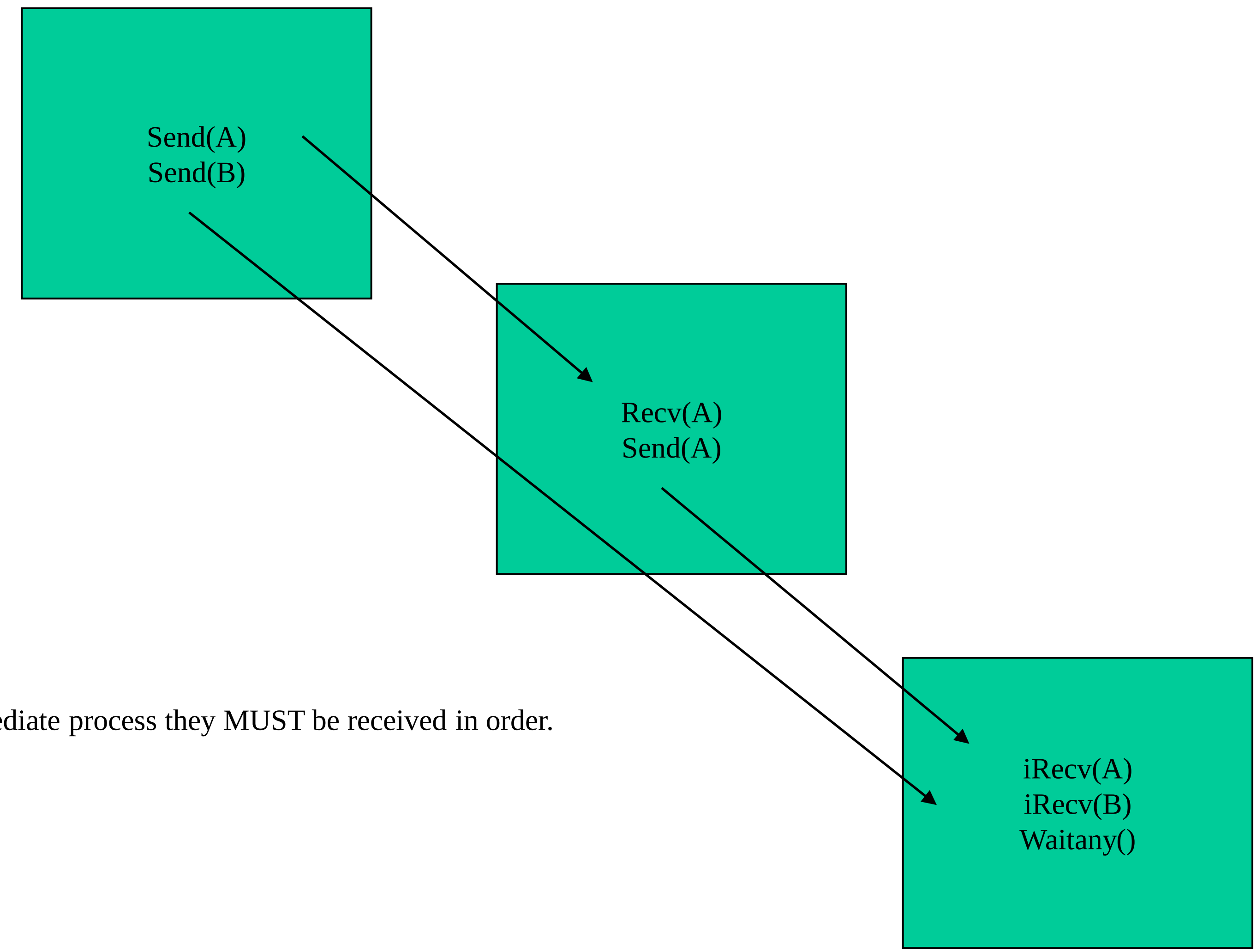
MPI_[SBR]send(start, count, datatype, dest, tag, comm)

There is only one mode for receive!

Messages matched in order



Message ordering



Without the intermediate process they MUST be received in order.

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

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