Approaches to Send/Receive

Blocking

Non-Blocking

Blocking (Non-Buffered) Send/Receive

- Follow some form of "handshaking" protocol
  - Request to Send → Clear to Send → Send Data → Acknowledgement
  - Problem 1: Idling Overhead (both sender/receiver side)
  - Problem 2: Deadlock (sending at same time)

Blocking (Buffered) Send/Receive

- Copy send-data to designated buffer, and returns after "copy" operation is completed
- Problem 1: Buffer Size

```
for (i = 0; i < 1000; i++) {
    produce_data(&a);
    send(&a, P1);
}
```

```
for (i = 0; i < 1000; i++) {
    receive(&a, P0);
    consume_data(&a);
}
```

- Problem 2: Deadlock (sending at same time)

Blocking — Synchronous
Comms

Non-Blocking — Asynchronous.
comms.

Approaches to Send/Receive (cont.)

t_1
1. send was successful.
2. " " not successful.

A

t_0 send()

ACK

t_1

t_2

Non-Blocking Send/Receive

- Return from Send/Receive operation before it is "safe" to return.
- Programmer responsibility to ensure that "sending data" is not altered immediately

- Blocking Operations: Safe and Easy Programming (at cost of overhead and risk of deadlocks)
- Non-Blocking Operations: Useful for Performance optimization, and breaking deadlocks (but brings in plenty of race-conditions if programmer not careful)

t_2
1. send was successful.
2. " " not successful.

Point to Point Communication

MPI_Send(—)   MPI-Recv(—)

I ← non-blocking ——→ I

Types of Point-to-Point Send/Receive Calls

- **Synchronous Transfer**: Send/Receive routines return only when the message transfer is completed. Not only does this transfer data, but it also synchronizes processes

```
MPI_Send()        // Blocking Send
MPI_Recv()        // Blocking Receive
```

- **Asynchronous Transfers**: Send/Receive do not wait for transfer data and proceeds with execution next line of instruction. (Precaution: Do not modify the send/receive buffers)

```
MPI_Isend()       // Non-Blocking Send
MPI_Irecv()       // Non-Blocking Receive
```

Point to Point Communication (cont.)

Sending

```
int MPI_Send(void *buffer,    int count,   MPI_DATATYPE datatype,
             int destination, int tag,     MPI_Comm comm);
```

- Send the data stored in **buffer**
- **Count** is the number of entries in the buffer
- What is the **datatype** of the buffer (MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_CHAR, etc.)
- **Destination** is the rank of process, to whom buffer is to be sent to, residing in communication universe **comm**
- The **tag** of the message (to distinguish between different types of messages)

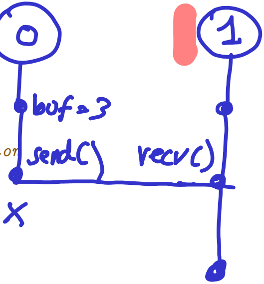
Point to Point Communication (cont.)

Receiving

```
int MPI_Recv(void *buffer,    int count,    MPI_DATATYPE datatype,
             int source,      int tag,      MPI_Comm comm,
             MPI_Status *status);
```

- Store the received message in **buffer**
- **Count** is the number of entries to be received in the buffer. If number of entries is larger than the capacity of buffer, an overflow error MPI_ERR_TRUNCATE is returned.
- **Datatype** is the type of data that has been received
- **Source** is the rank of process, residing in communication domain **comm**, from whom buffer is received. Source can be hard-set, or a wild-card MPI_ANY_SOURCE.
- To retrieve message of certain type, set the **tag** argument. If there are many messages of same tag from same process, any one of them may be retrieved. If message of any tag is to be retrieved, use the wild-card MPI_ANY_TAG.
- Store status of received message in **status** (next slide). If not needed, use MPI_STATUS_IGNORE

Example 1: Blocking Send/Receive between Two Processes




```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

if (myRank == 0) {                          // Sending Process
    int buffer = 3;
    MPI_Send(&buffer,              // Reference to the storage location
             1,                    // 1 item is being sent
             MPI_INT,              // integer data type
             1,                    // Destination rank
             123,                  // tag of message
             MPI_COMM_WORLD);      // communicator
}
else {                             // receiving process
    int buffer;
    MPI_Recv(&buffer,              // reference to storage location
             1,                    // receiving 1 item
             MPI_INT,              // of type integer
             0,                    // from process of rank 0
             123,                  // tag of message
             MPI_COMM_WORLD,       // communicator
             MPI_STATUS_IGNORE);   // status of received message

    printf("%d\n", buffer);
}
```

Example 2: Sending an Array from one process to another

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

int x, array_size = 10, *buffer1, *buffer2;

if (myRank == 0) {                                              // Sending Process
    buffer1 = malloc(sizeof(int)*array_size);     // allocate memory

    for (x = 0; x < array_size; x++)              // put something in it
            buffer1[x] = x+1;

    MPI_Send(buffer1, array_size, MPI_INT, 1, 123, MPI_COMM_WORLD);
}
else {                              // receiving process
    buffer2 = malloc(sizeof(int)*array_size);     // allocate memory
    MPI_Recv(buffer2, array_size, MPI_INT, 0, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    for (x = 0; x < array_size; x++)              // print it
            printf("%d\n", buffer2[x]2[x]);
}
```

int ~~myrandom~~ = rand()

0

a[rand()]

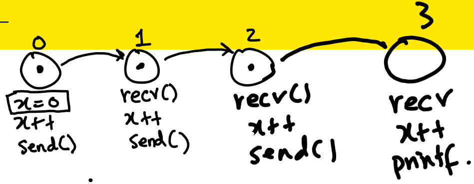
1

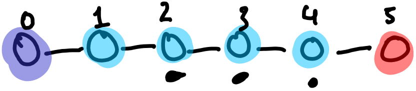
probe(status)

get count(status)

b[count]

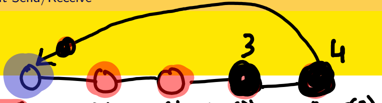
Example 3: Doing distributed $x++$






```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &mySize);

if (myRank == 0) { // Originating Process
    int x = 0;
    x++;

    MPI_Send(&x, 1, MPI_INT, myRank+1, 123, MPI_COMM_WORLD);
}
else if (myRank > 0 && myRank < n-1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, myRank-1, 123, MPI_COMM_WORLD,        MPI_STATUS_IGNORE);
    x++;
    MPI_Send(&x, 1, MPI_INT, myRank+1, 123, MPI_COMM_WORLD);
}
else if (myRank == n-1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, myRank-1, 123, MPI_COMM_WORLD,        MPI_STATUS_IGNORE);
    printf("Incremented to %d\n", x++);
}
```

Example 3: Doing distributed $x++$



n = 5

x=0	recv(0)	recv(1)	recv(2)	recv(3)
x++	x++	x++	x++	x++
send(1)	send(2)	send(3)	send(4)	send(0)
recv(4)	✓	✓	✓	✓
printf				

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &mySize);

if (myRank == 0) { // Originating Process
    int x = 0;
    x++;

    MPI_Send(&x, 1, MPI_INT, myRank+1, 123, MPI_COMM_WORLD);
}
else if (myRank > 0 && myRank < n-1) {
    int x;
                             (myRank-1)%n
    MPI_Recv(&x, 1, MPI_INT, myRank-1, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    x++;
    MPI_Send(&x, 1, MPI_INT, myRank+1, 123, MPI_COMM_WORLD);
                             (myRank+1)%n
}
else if (myRank == n-1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, myRank-2, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Incremented to %d\n", x++);
}
```

MPI-Recv(&x, 1 ....n-1 ....)
printf.

(myRank - 1) % n

(myRank + 1) % n

(4+1) % 5 = 0

(3+1) % 5 = 4

MPI_Probe() and MPI_Get_count()

"x" "y"

Tx $\xrightarrow{\text{tag}}$ Rx

MPI_Status Structure

```
typedef struct _MPI_Status {
  int MPI_SOURCE;        // Rank of Sender
  int MPI_TAG;           // Tag of Message
  int MPI_ERROR;         // Any Error if occurred
} MPI_Status;
```

MPI_SOURCE — Rx

MPI_Source — Tx

- **MPI_Probe()**: Check for incoming messages (without actually receiving them).

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

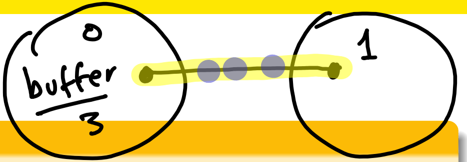
- **MPI_Get_count()**: Returns number of messages received

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

error?

Success — Not success

MPI_Probe() and MPI_Get_count()



MPI -Probe ( )

MPI_Status Structure

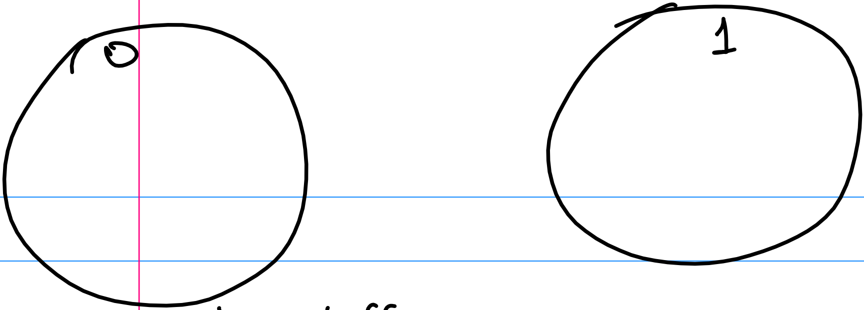
```
typedef struct _MPI_Status {
  int MPI_SOURCE;       // Rank of Sender
  int MPI_TAG;          // Tag of Message
  int MPI_ERROR;        // Any Error if occurred
} MPI_Status;
```

- **MPI_Probe()**: Check for incoming messages (without actually receiving them).

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

- **MPI_Get_count()**: Returns number of messages received

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




```
int *buffer
if ( myrank == 0 )
    buffer = malloc( rand() )
else
    ?
```

Non-Blocking Send/Receive

- For sending, **MPI_Isend()** is used with the syntax:

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

- For receiving, **MPI_Irecv()** is used with the syntax:

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int src,   int tag,   MPI_Comm comm,
              MPI_Request *request)
```

- To check whether a send/receive has completed or not, use **MPI_Test()** with syntax:

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

- To force a wait on a non-blocking send/receive, use **MPI_Wait()**:

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

- To wait for x requests, use **MPI_Waitall** as:

```
int MPI_Waitall(int x, MPI_Request *req, MPI_Status *stat);
```

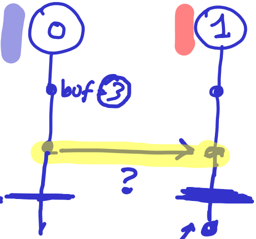
Non-Blocking Send/Receive (cont.)

structs.

```
MPI_Request request;    // handle to the non-blocking operation
MPI_Status  status;     // contains information about the message
int flag;               // 1: sent/received, 0: not-sent/not-received
int buffer;             // the data buffer

if (myRank == 0) {                                    // Sending Side
    buffer = 3;
    MPI_Isend(&buffer, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, &request);
}
else {                                                // Receiving Side
    MPI_Irecv(&buffer, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &request);

    MPI_Test(&request, &flag, &status);               // check status of recv
    if (flag == 0) {
        MPI_Wait(&request, &status);                  // force wait
    }

    printf("%d\n", buffer);
}
```

MPI_Test

MPI-Wait.



Check.

Collective Communications

- Point-to-Point: It is programmer's responsibility to ensure that all processes participate correctly in a given communication (Programmer's burden)
- MPI simplifies this using Collective Communication. Types are:
  - **Synchronization**:
    - Barriers: MPI_Barrier()
  - **Moving Data**:
    - Broadcasting: MPI_Bcast()
    - Scattering: MPI_Scatter()
    - Gathering: MPI_Gather()
  - **Collective Computation**:
    - Reduction: MPI_Reduce()
- Difference to point-to-point communications
  - No message tags
  - Most calls/versions support blocking communication only