Ateneo de Manila University

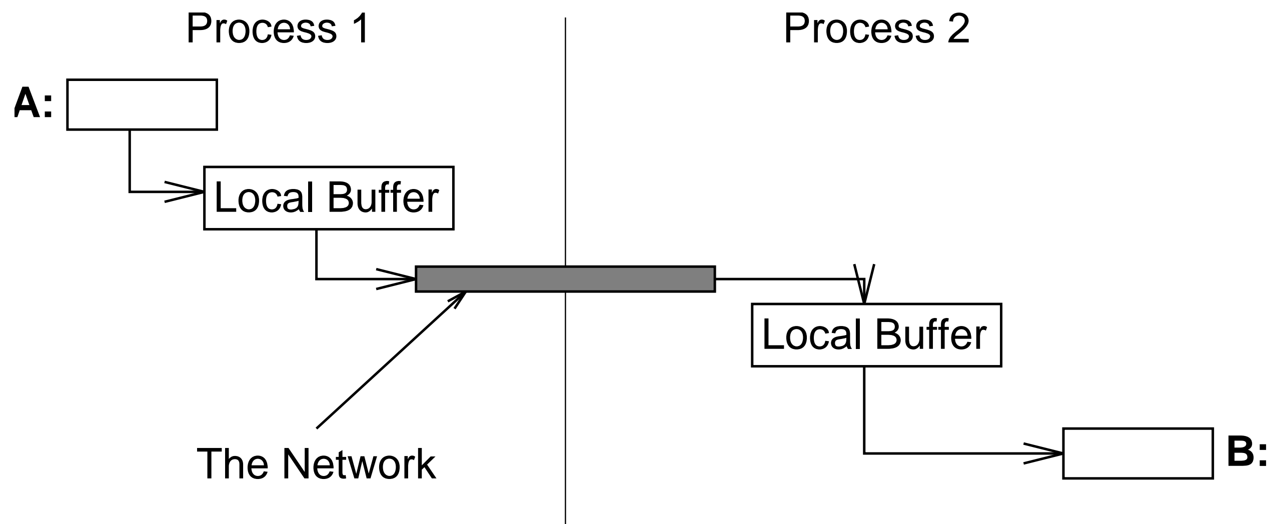
Parallel Programming with MPI: Non-blocking Communication

Ateneo High Performance Computing Group
1st Semester
http://www.math.admu.edu.ph/ahpc/
william.s.yu@ieee.org

Non-blocking Communications

Buffering Issues

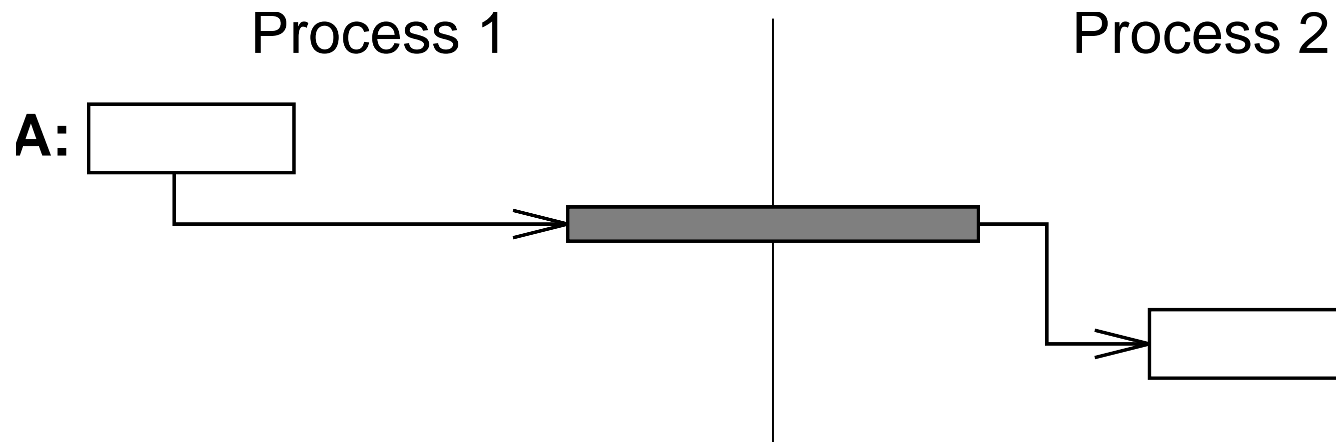
- Where does data go when you send it?
- One possibility is:



- This is not very efficient:
  - Three copies in addition to the exchange of data between processes.
  - Copies are "bad."

Better Buffering

- We prefer



- But this requires that either that `MPI_SEND` not return until the data has been delivered *or* that we allow a send operation to return before completing the transfer.
- In the latter case, we need to test for completion later.

Blocking and Non-Blocking Communication

- So far we have used *blocking* communication:
  - `MPI_SEND` does not complete until buffer is empty (available for reuse).
  - `MPI_RECV` does not complete until buffer is full (available for use).
- Simple, but can be prone to deadlocks:

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

Completion depends in general on size of message and amount of system buffering.

The semantics of blocking/non-blocking has nothing to do with when messages are sent or recieved. The difference is when the buffer is free to re-use.

Some Solutions to the Deadlock 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, with `MPI_SENDRECV`:

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

More Solutions to the Deadlock Problem

- Use non-blocking operations:

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

- Use `MPI_BSEND`
  - Copies message into a user buffer (previously supplied) and returns control to user program
  - Sends message sometime later

MPI's Non-Blocking Operations

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

```
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)
```

Multiple Completions

- It is often desirable to wait on multiple requests.
- An example is a worker/manager program, where the manager waits for one or more workers to send it a message.

```
MPI_WAITALL(count, array_of_requests,
            array_of_statuses)
MPI_WAITANY(count, array_of_requests, index, status)
MPI_WAITSOME(incount, array_of_requests, outcount,
  array_of_indices, array_of_statuses)
```

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

Probing the Network for Messages

- MPI_PROBE and MPI_IPROBE allow the user to check for incoming messages without actually receiving them
- MPI_IPROBE returns "flag == TRUE" if there is a matching message available. MPI_PROBE will not return until there is a matching receive available:

```
MPI_IPROBE(source, tag, communicator, flag, status)
MPI_PROBE(source, tag, communicator, status)
```

It is typically not good practice to use these functions.

MPI Send-Receive

- The send-receive operation combines the send and receive operations in one call.
- The send-receive operation performs a blocking send and receive operation using distinct tags but the same communicator.
- A send-receive operation can be used with regular send and receive operations.

```
MPI_SENDRECV(sendbuf, sendcount, sendtype, dest,
             sendtag, recvbuf, recvcount, recvtype,
             source, recvtag, comm, status)
```

- Avoids user having to order send/receive to avoid deadlock

Non-Blocking Example: 1 Worker

```
/* ... Only a portion of the code */
int flag = 0;
MPI_Status status;
double buffer[BIG_SIZE];
MPI_Request request;

/* Send some data */
MPI_Isend(buffer, BIG_SIZE, MPI_DOUBLE, dest, tag,
          MPI_COMM_WORLD, &request);

/* While the send is progressing, do some useful work */
while (!flag && have_more_work_to_do) {
  /* ...do some work... */
  MPI_Test(&request, &flag, &status);
}
/* If we finished work but the send is still pending, wait */
if (!flag)
  MPI_Wait(&request, &status);
/* ... */
```

Non-Blocking Example: 4 Worker

```
/* ... Only a portion of the code */
MPI_Status status[4];
double buffer[BIG_SIZE];
MPI_Request requests[4];
int i, flag, index, each_size = BIG_SIZE / 4;

/* Send out the data to the 4 workers */
for (i = 0; i < 4; i++)
  MPI_Isend(buffer + (i * each_size), each_size, MPI_DOUBLE, i + 1,
            tag, MPI_COMM_WORLD, &requests[i]);

/* While the sends are progressing, do some useful work */
for (i = 0; i < 4 && have_more_work_to_do; i++) {
  /* ...do some work... */
  MPI_Testany(4, requests, &flag, &index, &status[0]);
  if (!flag)
    i--;
}

/* If we finished work but still have sends pending, wait for the rest*/
if (i < 4)
  MPI_Waitall(4, requests, status);
/* ... */
```

The 5 Sends

MPI_SEND Normal send. Returns after the message has been copied to a buffer OR after the message “on its way”.

MPI_BSEND Buffered send. Returns after the message has been copied to an internal MPI buffer (previously supplied by the user).

MPI_SSEND Synchronous send. Returns after the message reaches the receiver.

MPI_RSEND Ready Send. The matching receive must be posted before the send executes. Returns once the message has left the send buffer.

MPI_ISEND Immediate send. Returns immediately. You may not modify contents of the message buffer until the send has completed (MPI_WAIT, MPI_TEST).

Copyright © 2000-2001 by William Emmanuel S. Yu and Jeff Squires. This material may be distributed only subject to the terms and conditions set forth in the Open Content License, v1.0 or later (the latest version is presently available at http://opencontent.org/opl.shtml).