COMP 322: Fundamentals of Parallel Programming

Lecture 30: Introduction to the Message Passing Interface (MPI) cont.

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Worksheet #29: MPI send and receive

In the space below, indicate what values you expect the print statement in line 10 to output, assuming that the program is executed with two MPI processes.

```
1. int a[], b[];
2. ...
3. if (MPI.COMM_WORLD.rank() == 0) {
4.     MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, 1, 1);
5.     MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
6. }
7. else {
8.     Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0, 2);
9.     Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI.INT, 0, 1);
10.     System.out.println("a = " + a + " ; b = " + b);
11.}
12. …
```

Answer: Nothing! The program will deadlock due to mismatched tags, with process 0 blocked at line 4, and process 1 blocked at line 8.

Basic Datatypes

- mpiJava defines 9 basic datatypes: these correspond to the 8 primitive types in the Java language, plus the MPI.OBJECT datatype that stands for an Object (or, more formally, a Java reference type).
  —MPI.OBJECT value can only be dereferenced on process where it was created

- The basic datatypes are available as static fields of the MPI class. They are:

mpiJava datatype	Java type
MPI.BYTE	byte
MPI.CHAR	char
MPI.SHORT	short
MPI.BOOLEAN	boolean
MPI.INT	int
MPI.LONG	long
MPI.FLOAT	float
MPI.DOUBLE	double
MPI.OBJECT	Object

Outline of today's lecture

- Blocking communications (contd)

- Non-blocking communication

Communication Buffers

- Most of the communication operations take a sequence of parameters like

  Object buf, int offset, int count, Datatype type

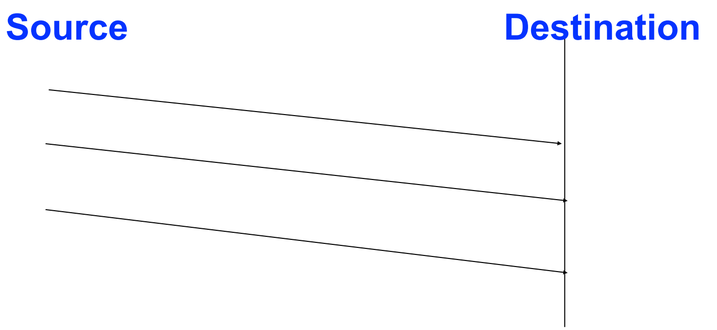
- In the actual arguments passed to these methods, buf must be an array (or a run-time exception will occur)

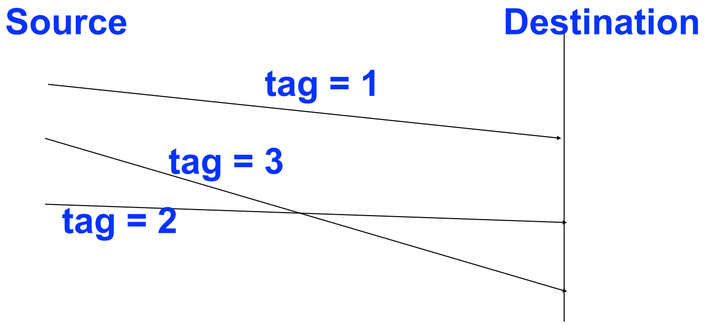
Message Ordering in MPI

- FIFO ordering only guaranteed for same source, destination, data type, and tag

- In HJ actors, FIFO ordering was guaranteed for same source and destination

  —Actor send is also “one-sided” and “non-blocking” (unlike send/recv in MPI)





Layout of Buffer

- If type is a basic datatype (corresponding to a Java type), the message corresponds to a subset of the array buf, defined as follows:

  - In the case of a send buffer, the red boxes represent elements of the buf array that are actually sent.

  - In the case of a receive buffer, the red boxes represent elements where the incoming data may be written.

Scenario #1

Consider:

```
int a[], b[];
...
if (MPI.COMM_WORLD.rank() == 0) {
    MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, 1, 1);
    MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
}
else {
    Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0, 2);
    Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI.INT, 0, 1);
}
…
```

Blocking semantics for Send() and Recv() can lead to a deadlock.

Approach #1 to Deadlock Avoidance: Reorder Send/Recv calls

We can break the circular wait in the worksheet by reordering Recv() calls to avoid deadlocks as follows:

```
int a[], b[];
...
if (MPI.COMM_WORLD.rank() == 0) {
    MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, 1, 1);
    MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
}
else {
    Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI.INT, 0, 1);
    Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0, 2);
}
...
```


Scenario #2

Consider the following piece of code, in which process i sends a message to process *i* + 1 (modulo the number of processes) and receives a message from process *i* - 1 (modulo the number of processes)

```
1.int a[], b[];
2.. . .
3.int npes = MPI.COMM_WORLD.size();
4.int myrank = MPI.COMM_WORLD.rank()
5.MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, (myrank+1)%npes, 1);
6.MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, (myrank+npes-1)%npes, 1);
```

Question: Does this MPI code deadlock?

Approach #2 to Deadlock Avoidance: A combined Sendrecv call

- Since it is fairly common to want to simultaneously send one message while receiving another.
- In mpiJava, the Sendrecv() method has the following signature:

```
Status Sendrecv(Object sendBuf, int sendOffset, int sendCount, Datatype sendType, int dst, int sendTag,
                Object recvBuf,  int recvOffset,  int recvCount,  Datatype recvType,  int src, int recvTag) ;
```

—More efficient than separate sends and receives

—Can avoid deadlock

—There is also a variant called Sendrecv_replace() which only specifies a single buffer

Using Sendrevc for Deadlock Avoidance in Scenario #2

Consider the following piece of code, in which process i sends a message to process *i* + 1 (modulo the number of processes) and receives a message from process *i* - 1 (modulo the number of processes)

```
int a[], b[];
. . .
int npes = MPI.COMM_WORLD.size();
int myrank = MPI.COMM_WORLD.rank()
MPI.COMM_WORLD.Sendrecv(a, 0, 10, MPI.INT, (myrank+1)%npes, 1,
                        b, 0, 10, MPI.INT, (myrank+npes-1)%npes, 1);

...
```

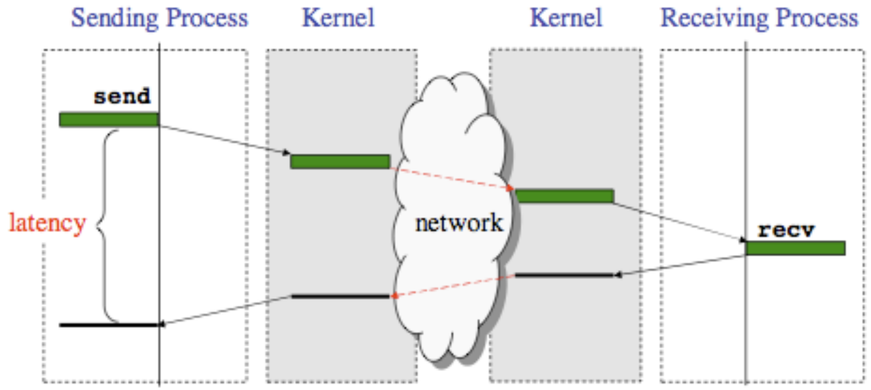
A combined Sendrecv() call avoids deadlock in this case

Outline of today's lecture

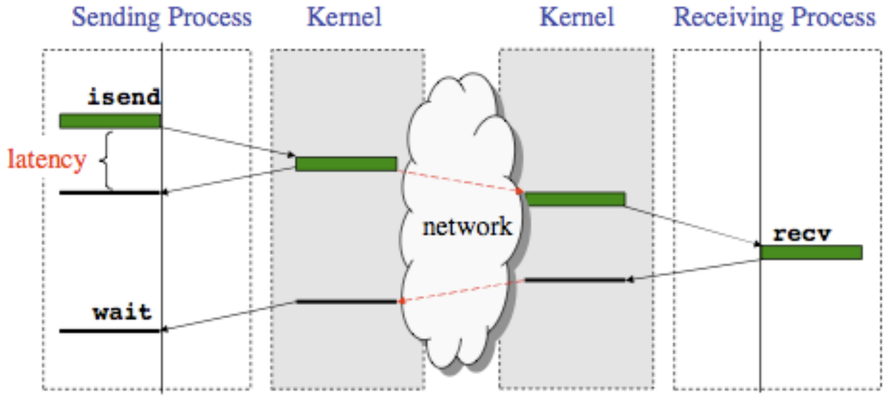
- Blocking communications (contd)
- Non-blocking communication

Latency in Blocking vs Nonblocking Communication

Blocking communication



Nonblocking communication (like an async or future task)



Non-Blocking Send and Receive Operations

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations ("I" stands for "Immediate")

```
Request Isend(Object buf, int offset, int count, Datatype type, int dst, int tag) ;
Request Irecv(Object buf, int offset, int count, Datatype type, int src, int tag) ;
```

- Use Wait() to wait for operation to complete (like future get).

```
Status Wait(Request request)
```

- The Wait() operation is declared to return a Status object. In the case of a non-blocking receive operation, this object has the same interpretation as the Status object returned by a blocking Recv() operation.

Simple Irecv() Example

The simplest way of waiting for completion of a single non-blocking operation is to use the instance method Wait() in the Request class, e.g:

```
// Post a receive (like a “communication async”)
Request request = Irecv(intBuf, 0, n, MPI.INT,
                        MPI.ANY_SOURCE,  0) ;
// Do some work while the receive is in progress
…
// Wait for message to arrive (like a future get)
Status status = request.Wait() ;
// Do something with data received in intBuf
…
```


Waitall() vs Waitany()

public static Status[] Waitall (Request [] array_of_request)

- Waitall() blocks until all operations associated with the active requests have completed.
- Returns an array of statuses for each of the requests.
  — Waitall() is a like a finish scope for all requests in the array

public static Status Waitany(Request [] array_of_request)

- Waitany() blocks until one of the operations associated with the active requests has completed.
  —Source of nondeterminism

Announcements & Reminders

- Quiz for Unit 7 is due Friday, April 16th at 11:59pm
- Hw #4 (Checkpoint #1) is due Monday, April 19th at 11:59pm

Worksheet #30: MPI send and receive

In the space below, use the minimum amount of non-blocking communication to reach the print statement in line 10 (assume that the program is executed with two MPI processes).

```
1. int a[], b[];
2. ...
3. if (MPI.COMM_WORLD.rank() == 0) {
4.     MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, 1, 1);
5.     MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
6. }
7. else {
8.     Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0, 2);
9.     Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI.INT, 0, 1);
10.     System.out.println("a = " + a + " ; b = " + b);
11.}
12. …
```