COMP 322: Fundamentals of Parallel Programming

Lecture 33: Message Passing Interface (contd)

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Acknowledgments for Today's Lecture

- "Principles of Parallel Programming", Calvin Lin & Lawrence Snyder
  - Includes resources available at http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html
- "Parallel Architectures", Calvin Lin
  - Lectures 5 & 6, CS380P, Spring 2009, UT Austin
  - http://www.cs.utexas.edu/users/lin/cs380p/schedule.html
- Slides accompanying Chapter 6 of "Introduction to Parallel Computing", 2nd Edition, Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, Addison-Wesley, 2003
  - http://www-users.cs.umn.edu/~karypis/parbook/Lectures/AG/chap6_slides.pdf
- MPI slides from "High Performance Computing: Models, Methods and Means", Thomas Sterling, CSC 7600, Spring 2009, LSU
  - http://www.cct.lsu.edu/csc7600/coursemat/index.html
- mpiJava home page: http://www.hpjava.org/mpiJava.html
- MPI lectures given at Rice HPC Summer Institute 2009, Tim Warburton, May 2009

Outline of today's lecture

- <u>Blocking communications (Recap)</u>
- Non-blocking communications
- Collective communications

Worksheet #32: MPI send and receive

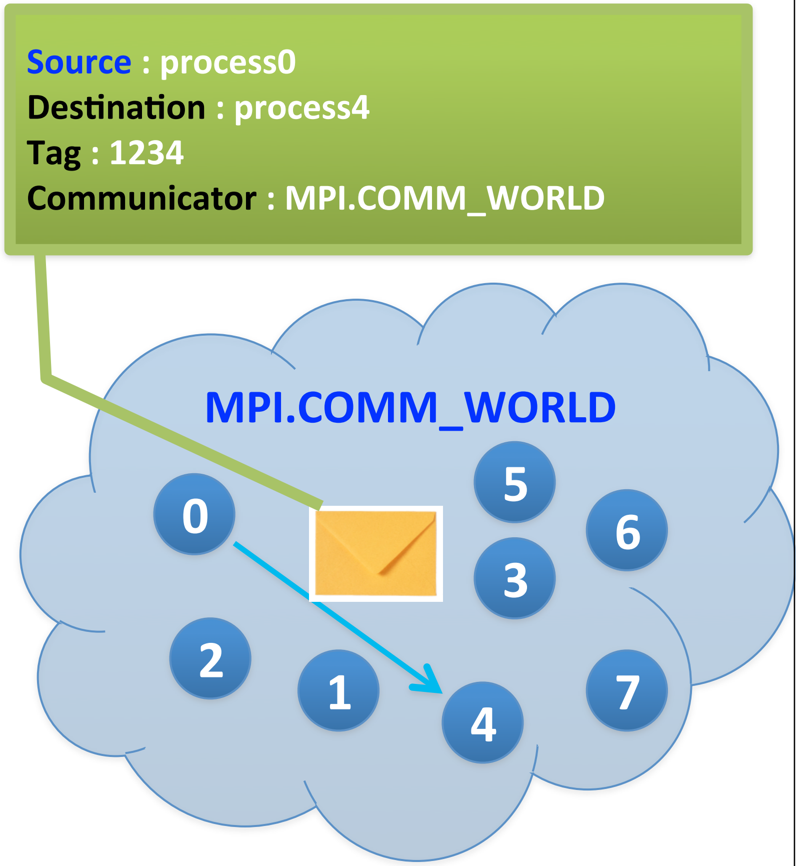
```
1. int a[], b[];
2. ...
3. if (MPI.COMM_WORLD.rank() == 0) {
4.     MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, 1, 1);
5.     MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
6. }
7. else {
8.     Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0, 2);
9.     Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI_INT,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

Question: In the space below, indicate what values you expect the print statement in line 10 to output (assuming the program is invoked with 2 processes).

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

Message Envelope

- Communication across process is performed using messages.
- Each message consists of a fixed number of fields that is used to distinguish them, called the Message Envelope :
  - —Envelope comprises source, destination, tag, communicator
  - —Message comprises Envelope + data
- Communicator refers to the namespace associated with the group of related processes



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).**
  - **The reason declaring buf as an Object rather than an array was that one would then need to overload with about 9 versions of most methods for arrays, e.g.**

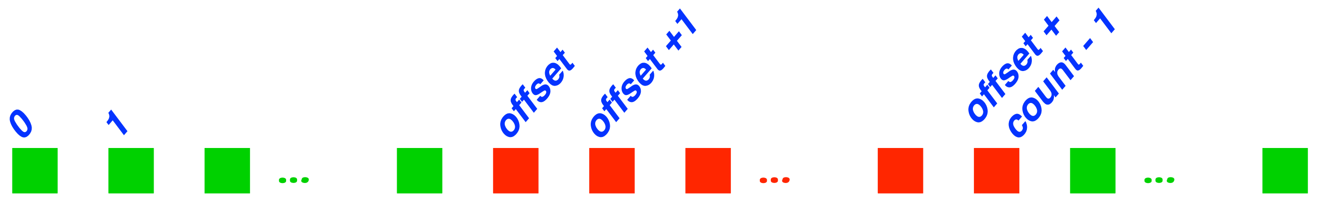
```
    void Send(int [] buf, …)
    void Send(long [] buf, …)
    …
```

  and about 81 versions of operations that involve two buffers, possibly of different type. Declaring Object buf allows any kind of array in one signature.

- **offset is the element in the buf array where message starts. count is the number of items to send. type describes the type of these items.**

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.

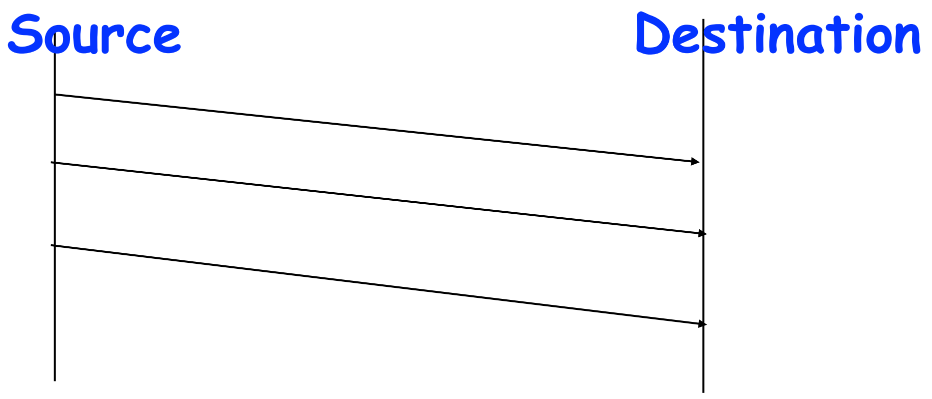
  – In the case of a receive buffer, the red boxes represent elements where the incoming data may be written (other elements will be unaffected). In this case count defines the maximum message size that can be accepted. Shorter incoming messages are also acceptable.

Basic Datatypes

- mpiJava defines 9 basic datatypes: these correspond to the 8 primitive types in the Java language, plus a basic datatype that stands for an Object (or, more formally, a Java reference type).
- The basic datatypes are available as static fields of the MPI class. They are:

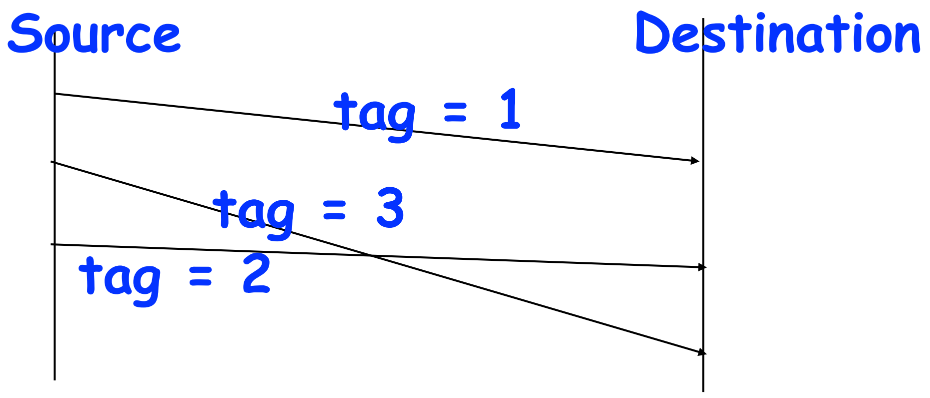
mpiJava datatype	Java type
MPI.BYTE	byte
MPI.CHAR	char
MPI.SHORT	short
MPI.BOOLEAN	boolean
MPI.INT	int
MPI.LONG	long
MPI.FLOAT	float
MPI.DOUBLE	double
MPI.OBJECT	Object

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)



Status values

- The recv() method returns an instance of the Status class.
- This object (referred to as “retval” below) provides access to several useful pieces about the message that arrived:
  - —int field retval.source holds the rank of the process that sent the message (particularly useful if the message was received with MPI.ANY_SOURCE).
  - —int field retval.tag holds the message tag specified by the sender of the message (particularly useful if the message was received with MPI.ANY_TAG).
  - —int method retval.Get_count(type) returns number of items received in the message.
  - —int method retval.Get_elements(type) returns number of basic elements received in the message.
  - —int field retval.index is set by methods like Request.Waitany(), described later.

Deadlock 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.

Deadlock 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.siz();
int myrank = MPI.COMM_WORLD.rank()
MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, (myrank+1)%npes, 1);
MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, (myrank-1+npes)%npes, 1);
```

Once again, we have a deadlock if Send() and Recv() are blocking

Approach #1 to Deadlock Avoidance --- Reorder Send and Recv calls

We can break the circular wait 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);
}
...
```

Approach #2 to Deadlock Avoidance --- a combined Sendrecv() call

- **Since it is fairly common to want to simultaneously send one message while receiving another (as illustrated in Scenario #2), MPI provides a more specialized operation for this.**
- **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) ;
```

—This can be more efficient than doing separate sends and receives, and it can be used to avoid deadlock conditions in certain situations

  – **Analogous to phaser “next” operation, where programmer does not have access to individual signal/wait operations**

—There is also a variant called Sendrecv_replace() which only specifies a single buffer: the original data is sent from this buffer, then overwritten with incoming data.

Using Sendrecv 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-1+npes)%npes,
1);

...
```

A combined Sendrecv() call avoids deadlock in this case

Sources of nondeterminism: ANY_SOURCE and ANY_TAG

- A recv() operation can explicitly specify which process within the communicator group it wants to accept a message from, through the src parameter.
- It can also explicitly specify what message tag the message should have been sent with, through the tag parameter.
- The recv() operation will block until a message meeting both these criteria arrives.
  - —If other messages arrive at this node in the meantime, this call to recv() ignores them (which may or may not cause the senders of those other messages to wait, until they are accepted).
- If you want the recv() operation to accept a message from any source, or with any tag, you may specify the values MPI.ANY_SOURCE or MPI.ANY_TAG for the respective arguments.

Outline of today's lecture

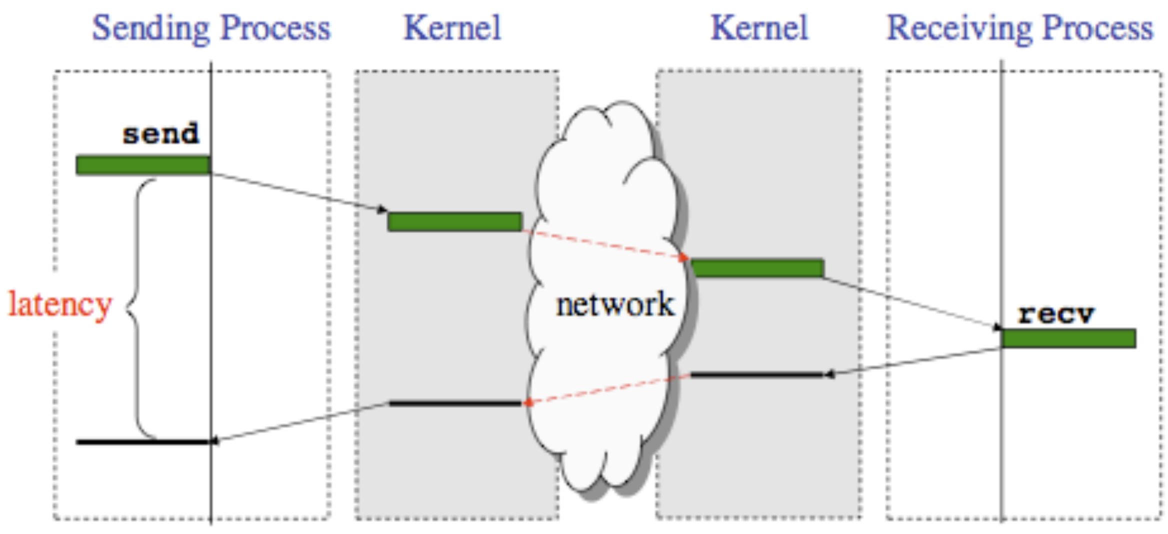
- Blocking communications (Recap)

- <u>Non-blocking communications</u>

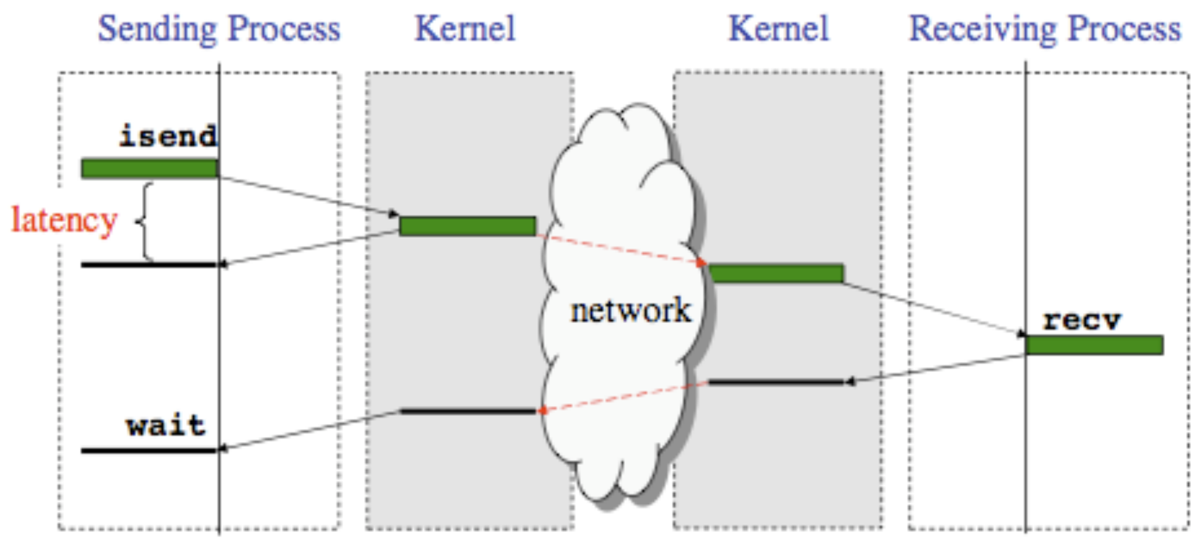
- Collective communications

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")
- The method signatures for Isend() and Irecv() are similar to those for Send() and Recv(), except that Isend() and Irecv() return objects of type Request:

  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) ;
- Function Test() tests whether or not the non-blocking send or receive operation identified by its request has finished.

  Status Test(Request request)
- Wait waits() for the operation to complete.

  Status Wait(Request request)

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 operation
Request request = Irecv(intBuf, 0, n, MPI.INT,
                        MPI.ANY_SOURCE,  0) ;
// Do some work while the receive is in progress
…
// Finished that work, now make sure the message has
  arrived
Status status = request.Wait() ;
// Do something with data received in intBuf
…
```

- **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.**

Non-blocking Example

Example pseudo-code on process 0:

```
if(procid==0){

  Isend outgoing  to 1
  Irecv  incoming from 1

  .. compute ..

  Wait until Irecv  has received incoming

  .. compute ..

  Wait until Isend does not need outgoing

}
```

Example pseudo-code on process 1:

```
if(procid==1){

  Isend outgoing  to 1
  Irecv  incoming from 1

  .. compute ..

  Wait until Irecv  has received incoming

  .. compute ..

  Wait until Isend does not need outgoing

}
```

Using the "*non-blocked*" send and receives allows us to overlap the latency and buffering overheads with useful computation.

Non-blocking Code Snippets (C version)

- Post Irecv first
- Set up outgoing data
- Post Isend
- Do local work
- Wait for incoming data
- Do local work
- Make sure data left

```
/* process 0 does its thing */
if(procid==0){ dest = 1; source = 1; }
if(procid==1){ dest = 0; source = 0; }

/* this process requests unblocked receive from other process */
MPI_Irecv(indata, N, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &inrequest);

/* fill up outgoing data */
if(procid==0) for(n=0;n<N;++n) outdata[n] = 1./(   n+1.);
if(procid==1) for(n=0;n<N;++n) outdata[n] = 1./(.1*n+2.);

/* process 0 requests send to process 1 */
MPI_Isend(outdata, N, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD, &outrequest);

/* compute (each process does its own thing) */
if(procid==0) for(d=1,n=0;n<Nits;++n) d += 1./(n+d);
if(procid==1) for(d=1,n=0;n<Nits;++n) d += 1./(n+2*d);

/* now MPI_Wait to make sure incoming data arrived */
MPI_Wait(&inrequest, &status);

/* now can use inbound data */
for(n=0;n<N;++n) d += indata[n];

/* print out result */
printf("proc: %d result = %f\n", procid, d);

/* now MPI_Wait to make sure outgoing data has gone */
MPI_Wait(&outrequest, &status);
```

Waitall() vs. Waitany()

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

- **Waitall() blocks until all of the operations associated with the active requests in the array 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 in the array has completed.**

Outline of today's lecture

- Blocking communications (Recap)
- Non-blocking communications
- <u>Collective communications</u>

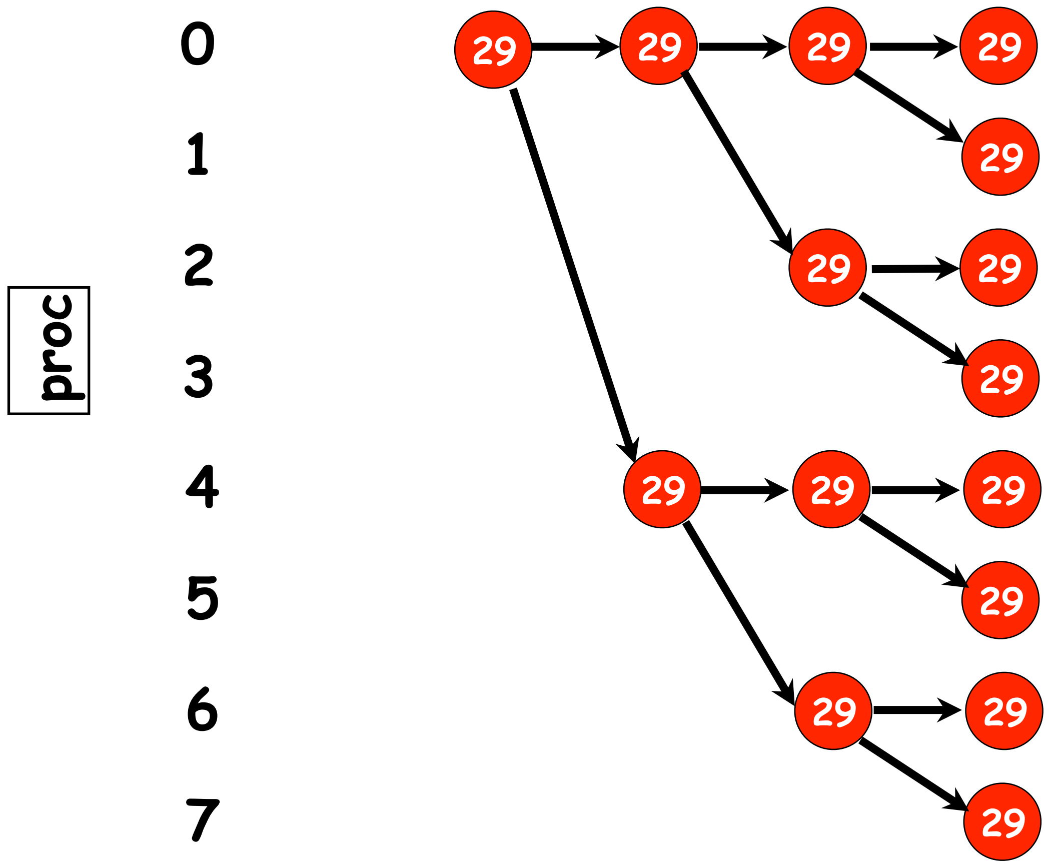
Collective Communications

- A popular feature of MPI is its family of collective communication operations.
- Each of these operations is defined over a communicator.
    - All processes in a communicator must perform the same operation
    - Implicit barrier (next)
- The simplest example is the broadcast operation: all processes invoke the operation, all agreeing on one root process. Data is broadcast from that root.

void Bcast(Object buf, int offset, int count, Datatype type, int root)

- Broadcast a message from the process with rank root to all processes of the group.

MPI_Bcast



Simple tree broadcast

A root process sends same message to all

29 represents an array of values

More Examples of Collective Operations

```
void Barrier()
```

– **Blocks the caller until all processes in the group have called it.**

```
void Gather(Object sendbuf, int sendoffset, int sendcount,
  Datatype sendtype, Object recvbuf, int recvoffset,
  int recvcount, Datatype recvtype, int root)
```

– **Each process sends the contents of its send buffer to the root process.**

```
void Scatter(Object sendbuf, int sendoffset, int sendcount,
  Datatype sendtype, Object recvbuf, int recvoffset,
  int recvcount, Datatype recvtype, int root)
```

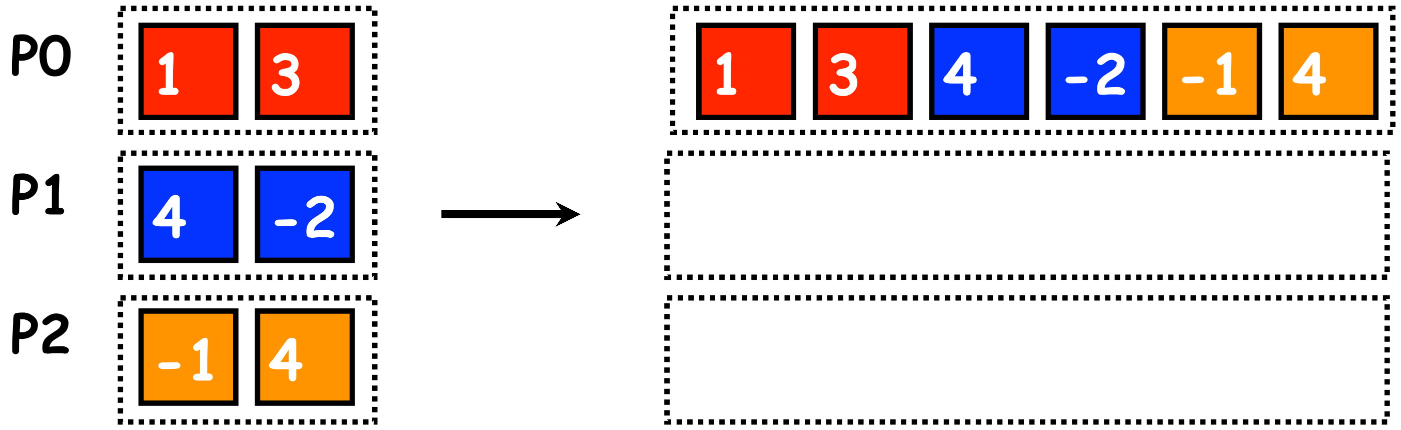
– **Inverse of the operation Gather.**

```
void Reduce(Object sendbuf, int sendoffset, Object recvbuf,
  int recvoffset, int count, Datatype datatype, Op op,
  int root)
```

– **Combine elements in send buffer of each process using the reduce operation, and return the combined value in the receive buffer of the root process.**

MPI_Gather

- On occasion it is necessary to copy an array of data from each process into a single array on a single process.
- Graphically:



- Note: only process 0 (P0) needs to supply storage for the output

Worksheet #33: MPI Gather

Name 1: ______________________ Name 2: ______________________

```
1.    MPI.Init(args) ;
2.    int myrank = MPI.COMM_WORLD.Rank() ;
3.    int numProcs = MPI.COMM_WORLD.Size() ;
4.    int size = ...;
5.    int[] sendbuf = new int[size];
6.    int[] recvbuf = new int[???];
7.    . . . // Each process initializes sendbuf
8.    MPI.COMM_WORLD.Gather(sendbuf, 0, size, MPI.INT,
9.                          recvbuf, 0, size, MPI.INT,
10.                         0/*root*/);
11.   . . .
12.   MPI.Finalize();
13.
```

In the space below, indicate what values should be provided instead of ??? in line 6, and why.