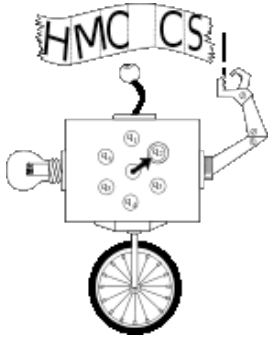


CS 181E: Fundamentals of Parallel Programming

Instructor: Vivek Sarkar

Co-Instructor: Ran Libeskind-Hadas

http://www.cs.hmc.edu/courses/2012/fall/cs181e/

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

Recap of Lecture 11

- Task Affinity with Places
  - hj -places p:w foo
    - Runs program foo with p places and w worker threads per place
  - async at(P) S
    - Creates new task to execute statement S at place P
    - async S is equivalent to async at(here) S
    - Main program task starts at place.factory.place(0)
- Introduction to the Message Passing Interface (MPI)

Example with Send() and Recv() calls

```
1.import mpi.*;

3.class myProg {
4.  public static void main( String[] args ) {
5.    int tag0 = 0;
6.    MPI.Init( args );                    // Start MPI computation
7.    if ( MPI.COMM_WORLD.rank() == 0 ) { // rank 0 = sender
8.      int loop[] = new int[1]; loop[0] = 3;
9.      MPI.COMM_WORLD.Send( "Hello World!", 0, 12, MPI.CHAR, 1, tag0 );
10.     MPI.COMM_WORLD.Send( loop, 0, 1, MPI.INT, 1, tag0 );
11.   } else {                              // rank 1 = receiver
12.     int loop[] = new int[1]; char msg[] = new char[12];
13.     MPI.COMM_WORLD.Recv( msg, 0, 12, MPI.CHAR, 0, tag0 );
14.     MPI.COMM_WORLD.Recv( loop, 0, 1, MPI.INT, 0, tag0 );
15.     for ( int i = 0; i < loop[0]; i++ ) System.out.println( msg );
16.   }
17.   MPI.Finalize( );                     // Finish MPI computation
18. }
19.}
```

Send() and Recv() calls are blocking operations by default

Outline

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

mpiJava send and receive

- Send and receive members of Comm:

```
void Send(Object buf, int offset, int count, Datatype type,  int dst, int tag) ;

Status Recv(Object buf, int offset, int count, Datatype type,  int src, int tag) ;
```

- The arguments buf, offset, count, type describe the data buffer—the storage of the data that is sent or received. They will be discussed on the next slide.
- dst is the rank of the destination process relative to this communicator. Similarly in Recv(), src is the rank of the source process.
- An arbitrarily chosen tag value can be used in Recv() to select between several incoming messages: the call will wait until a message sent with a matching tag value arrives.
- The Recv() method returns a Status value, discussed later.
- Both Send() and Recv() are <u>blocking operations</u>
  - —Analogous to phaser next operations

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 <u>must</u> be an array (or a run-time exception will occur).
  - The reason for declaring buf as an Object rather than an array was that one would then need to overload with about 9 versions of most methods fopr arrays, e.g.

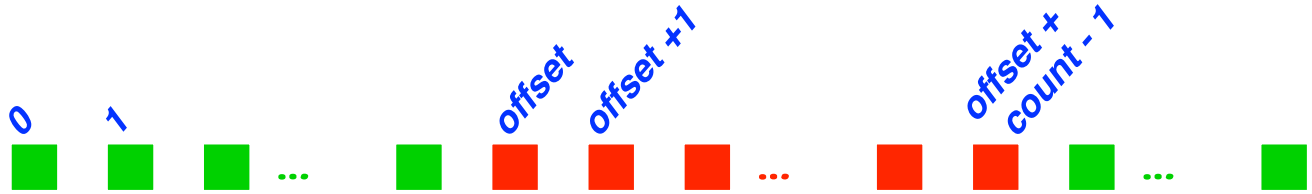
    ```
    void Send(int [] buf, ...)
    void Send(long [] buf, ...)
    ...
    ```

    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:



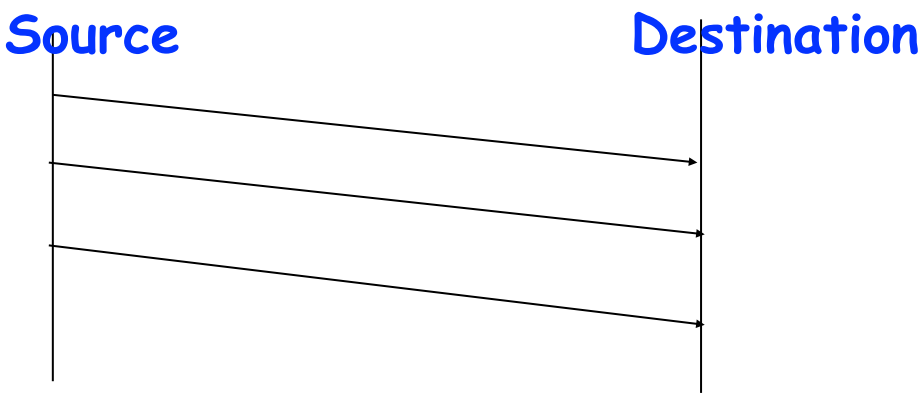
  - In the case of a send buffer, the red boxes represent elements of the buf array that are actually sent.
  - In the case of a receive buffer, the red boxes represent elements where the incoming data may be written (other elements will be unaffected). In this case count defines the maximum message size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

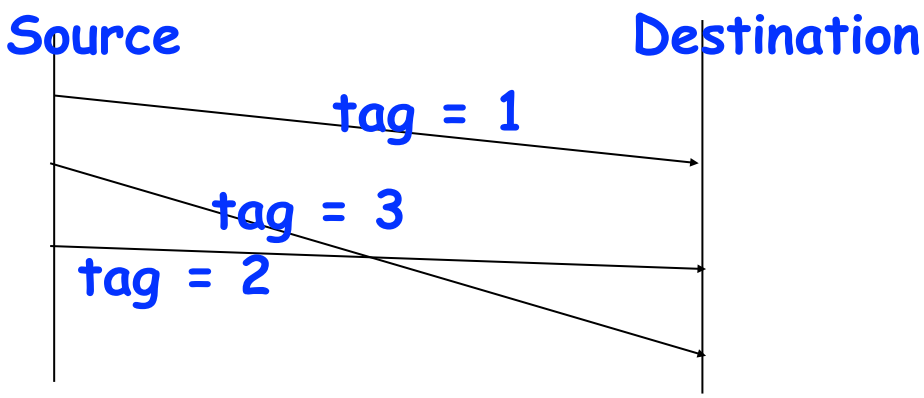
Basic Datatypes

- 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).
  - —MPI.Object uses standard Java serialization, which is SLOW

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)

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, since Send() and Recv() are blocking operations

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.

Status values

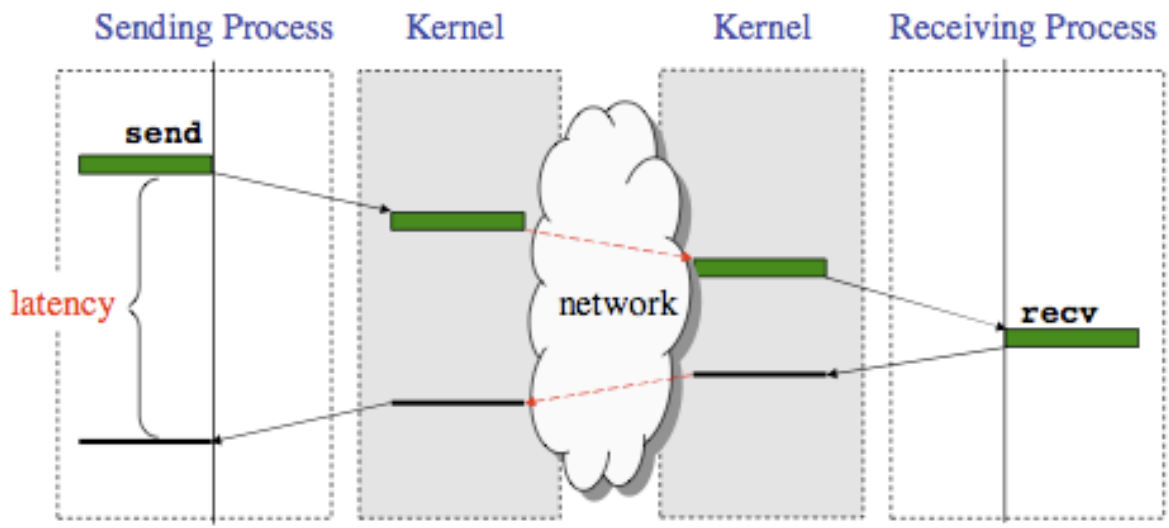
- The recv() method returns an instance of the Status class.
- This object (referred to as "retval" below) provides access to several useful pieces about the message that arrived:
  - int field retval.source holds the rank of the process that sent the message (particularly useful if the message was received 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.

Outline

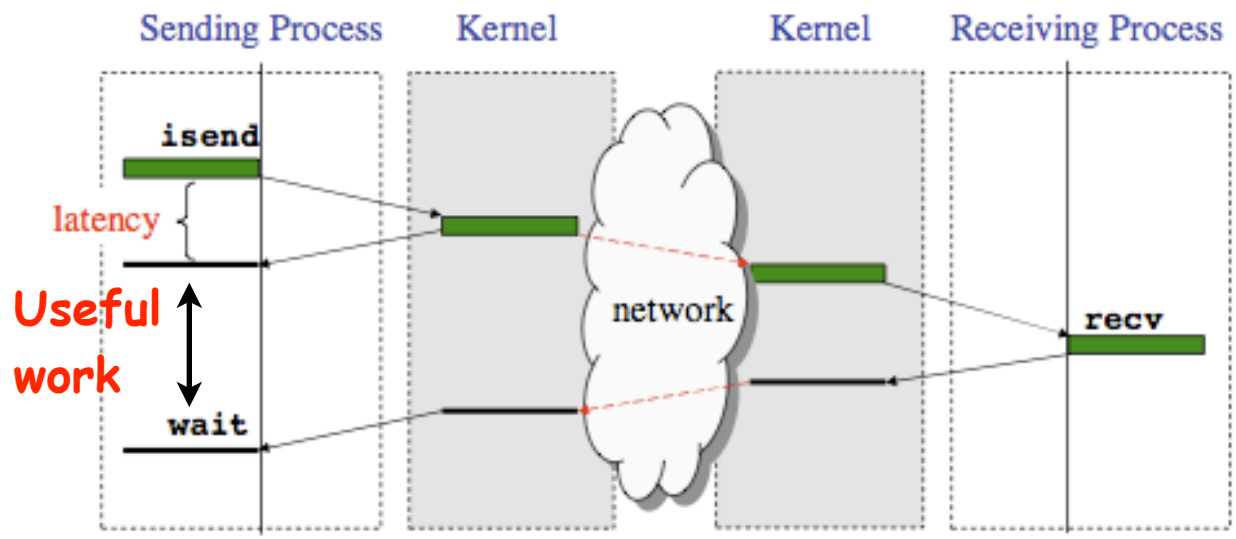
- Blocking communications
- <u>Non-blocking communications</u>
- Collective communications

Latency in Blocking vs. Nonblocking Communication

Blocking communication



Nonblocking communication (like an async or future task)



Non-blocking 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
  .. overwrite 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
  .. overwrite outgoing ..
}
```

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

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 (like a future get() operation)

  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
...
```

It's time for Worksheet 12 --- the last worksheet in this course!

Extensions to Wait(): Waitall() and 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. It returns an array of statuses for each of the requests.

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.

Example with Irecv() and Waitany()

```
. . . int me,tasks,i,index;
MPI.Init(args);
me = MPI.COMM_WORLD.Rank();
tasks = MPI.COMM_WORLD.Size();
int data[] = new int[tasks];
Request req[] = new Request[tasks];
Status status;
if(me > 0)
  MPI.COMM_WORLD.Send(a,0,1,MPI.OBJECT,0,1); // Send object in a[0]
else { // me == 0
  req[0] = MPI.REQUEST_NULL;
  for(i=1;i<tasks;i++)
     // Receive object in b[i]
     req[i] = MPI.COMM_WORLD.Irecv(b,i,1,MPI.OBJECT,i,1);
  for(i=1;i<tasks;i++) {
    status = Request.Waitany(req);
    // status object contains data on Irecv() operation that completed
    System.out.println("Received message from " + status.source);
  }
} . . .
```

Outline

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

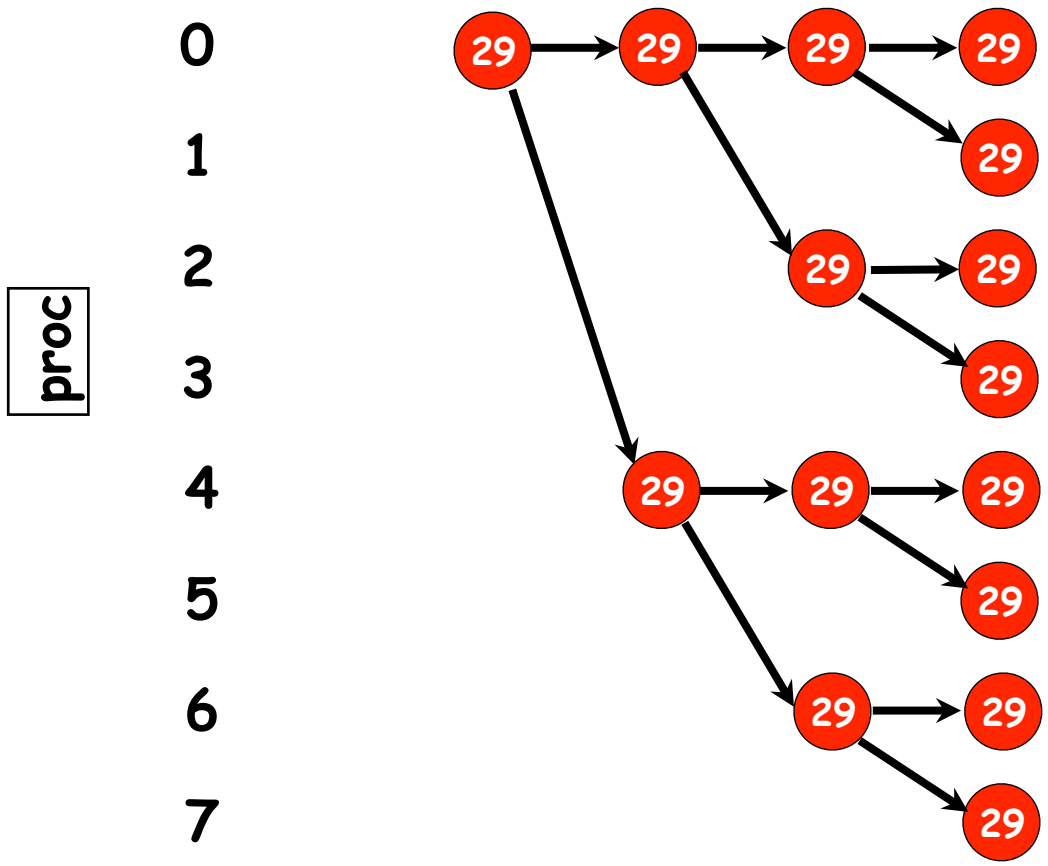
Collective Communications

- 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

- All the following are instance methods of Intracom:

  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)

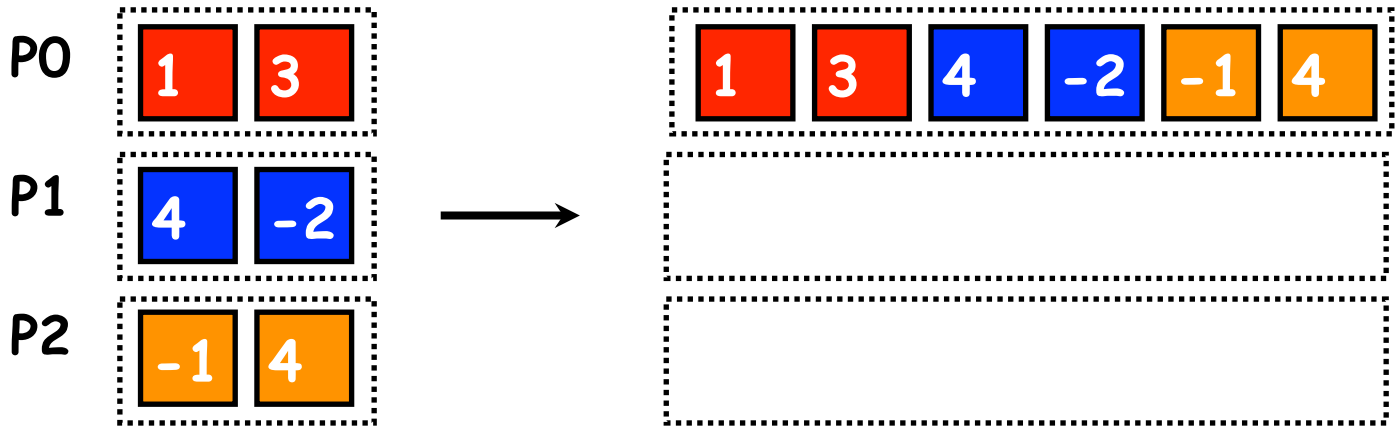
  - Inverse of the operation Gather.

  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

MPI_Reduce

```
void MPI.COMM_WORLD.Reduce(
                    Object[]        sendbuf        /* in */,
                    int             sendoffset     /* in */,
                    Object[]        recvbuf        /* out */,
                    int             recvoffset     /* in */,
                    int             count          /* in */,
                    MPI.Datatype    datatype       /* in */,
                    MPI.Op          operator       /* in */,
                    int             root           /* in */ )
```


```
MPI.COMM_WORLD.Reduce( msg, 0, result, 0, 1, MPI.INT, MPI.SUM, 2);
```

Predefined Reduction Operations

Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

MPI_MAXLOC and MPI_MINLOC

- The operation `MPI_MAXLOC` combines pairs of values (v_i, l_i) and returns the pair (v, l) such that v is the maximum among all v_i 's and l is the corresponding l_i (if there are more than one, it is the smallest among all these l_i 's).
- `MPI_MINLOC` does the same, except for minimum value of v_i.

Value	15	17	11	12	17	11
Process	0	1	2	3	4	5

```
MinLoc(Value, Process) = (11, 2)
MaxLoc(Value, Process) = (17, 1)
```

An example use of the `MPI_MINLOC` and `MPI_MAXLOC` operators.

Datatypes for MPI_MAXLOC and MPI_MINLOC

MPI datatypes for data-pairs used with the `MPI_MAXLOC` and `MPI_MINLOC` reduction operations.

MPI Datatype	C Datatype
`MPI_2INT`	**pair of** int**s**
`MPI_SHORT_INT`	short **and** int
`MPI_LONG_INT`	long **and** int
`MPI_LONG_DOUBLE_INT`	long double **and** int
`MPI_FLOAT_INT`	float **and** int
`MPI_DOUBLE_INT`	double **and** int

More Collective Communication Operations

- If the result of the reduction operation is needed by all processes, MPI provides:

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
                  int count, MPI_Datatype datatype, MPI_Op
op,               MPI_Comm comm)
```

- MPI also provides the MPI_Allgather function in which the data are gathered at all the processes.

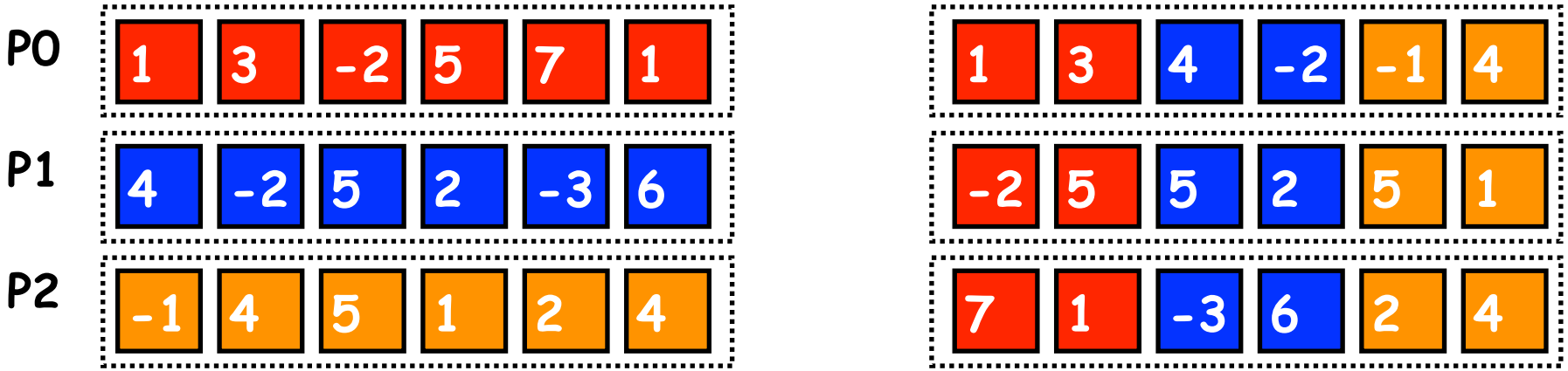
```
int MPI_Allgather(void *sendbuf, int sendcount,
                        MPI_Datatype senddatatype, void
*recvbuf,
                  int recvcount, MPI_Datatype recvdatatype,
                  MPI_Comm comm)
```

- To compute prefix-sums, MPI provides:

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op,
                  MPI_Comm comm)
```

MPI_Alltoall

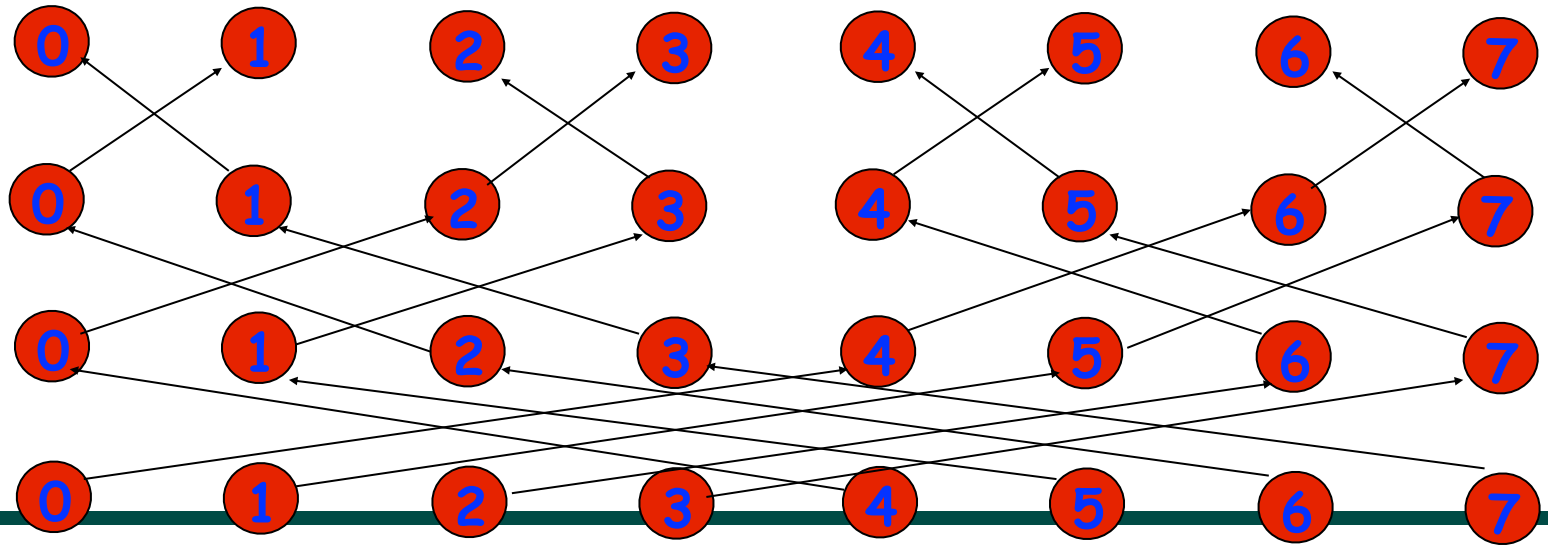
```
int MPI_Alltoall(void *sendbuf, int sendcount,
                 MPI_Datatype senddatatype, void
*recvbuf,
                 int recvcount, MPI_Datatype
recvdatatype,    MPI_Comm comm)
```



- **Each process submits an array to MPI_Alltoall.**
- **The array on each process is split into *nprocs* sub-arrays**
- **Sub-array *n* from process *m* is sent to process *n* placed in the *m*'th block in the result array.**

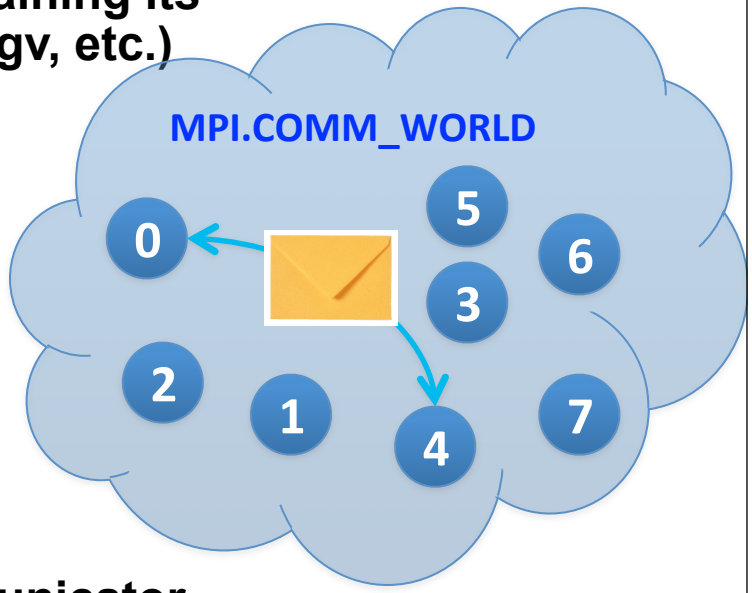
MPI_Allreduce

```
void MPI.COMM_WORLD.Allreduce(
        Object[]        sendbuf         /* in */,
        int             sendoffset      /* in */,
        Object[]        recvbuf         /* out */,
                        int             recvoffset      /* in */,
                        int             count   /* in */,
                        MPI.Datatype    datatype        /* in */,
                        MPI.Op                  operator        /* in */)
```



MPI.COMM_WORLD Communicator

- Communicator is an internal object
  - *Communicator registration is like phaser registration, except that MPI does not support dynamic parallelism*
- MPI programs are made up of communicating processes
- Each process has its own address space containing its own attributes such as rank, size (and argc, argv, etc.)
- MPI provides functions to interact with it
- Default communicator is MPI.COMM_WORLD
  - All processes are its members
  - It has a size (the number of processes)
  - Each process has a rank within it
  - Can think of it as an ordered list of processes
- Additional communicator(s) can co-exist
- A process can belong to more than one communicator
- Within a communicator, each process has a unique rank



Groups and Communicators

- In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.
- MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator.
- The simplest such mechanism is:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
```

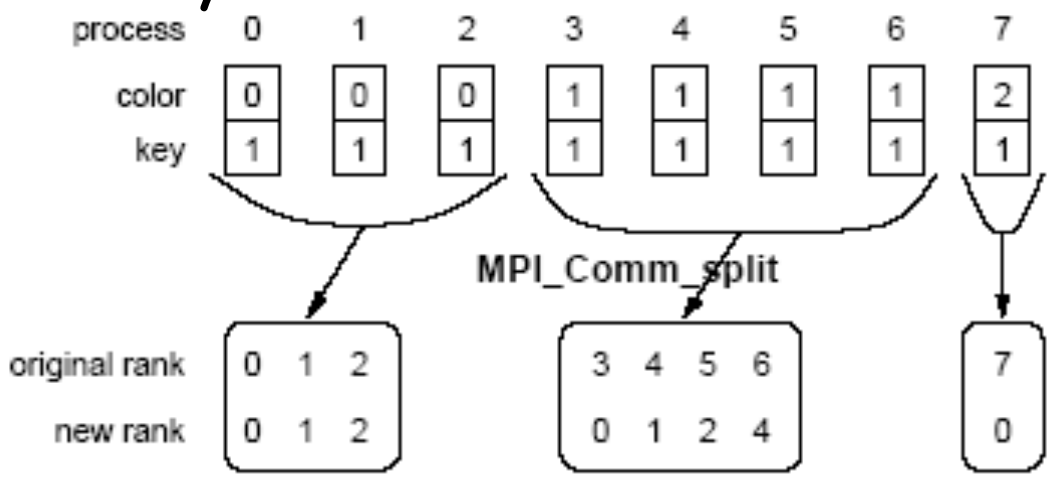
- This operation groups processors by color and sorts resulting groups on the key.

Groups and Communicators

- In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.
- MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups
- The simplest such mechanism is:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
```

- This operation groups processors by color and sorts resulting groups on the key.



Summary of MPI

- MPI is a platform for Single Program Multiple Data (SPMD) parallel computing on distributed memory architectures, with an API for sending and receiving messages
- It includes the abstraction of a "communicator", which is like an N-way communication channel that connects a set of N cooperating processes (analogous to a phaser)
- It also includes explicit datatypes in the API, that are used to describe the contents of communication buffers.
- The logical view of a machine in MPI consists of p processes, each with its own exclusive address space.
  1. Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
  2. All interactions (read-only or read/write) require cooperation of two processes - the process that has the data and the process that wants to access the data.

Worksheet #12 (to be done individually or in pairs): Deadlock avoidance using ISend() and IRecv()

Name 1: _____ Name 2: _____

Consider the code example below from slide 12 which exhibits deadlock. How can you rewrite it using ISend(), IRecv(), and Wait() calls to perform the desired communication while avoiding deadlock? You can use the space below for your solution. Don't worry about minor syntactic errors (e.g., missing semicolons, incorrect spelling of keywords, incorrect API, etc.) so long as the meaning of the program is unambiguous.

```
1. int a[], b[];
2. . . .
3. int npes = MPI.COMM_WORLD.siz();
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-1+npes)%npes, 1);
```

MPI Resources for Further Reading

- MPI: http://www.mcs.anl.gov/research/projects/mpi/
- MPICH2: http://www.mcs.anl.gov/research/projects/mpich2/
- Wiki: http://en.wikipedia.org/wiki/Message_Passing_Interface
- mpiJava home page: http://www.hpjava.org/mpiJava.html
  - Download: http://sourceforge.net/projects/mpijava/
- Web tutorials:
  - https://computing.llnl.gov/tutorials/mpi/
  - http://www.ecmwf.int/services/computing/training/material/hpcf/Intro_MPI_Programming.pdf (F77)
- Books:
  - http://www.cs.usfca.edu/mpi/
  - http://www.netlib.org/utk/papers/mpi-book/mpi-book.html