COMP 322: Fundamentals of Parallel Programming

Lecture 33: Introduction to MPI (Message Passing Interface)

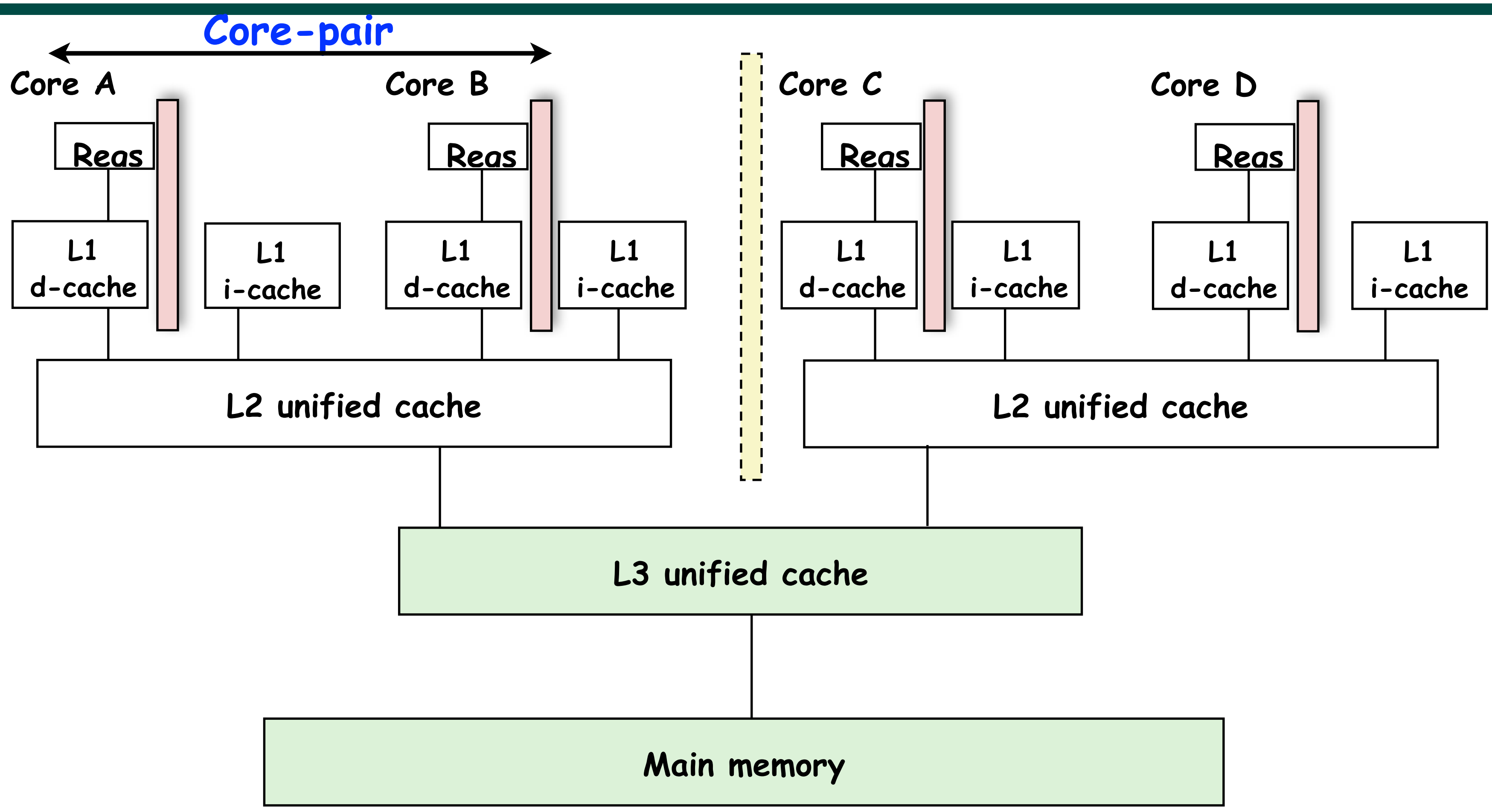
Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

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

Organization of a Shared-Memory Multicore SMP (Lecture 17)



- Memory hierarchy for a single Intel Xeon Quad-core E5440 HarperTown processor chip
  - A SUG@R node contains TWO such chips, for a total of 8 cores

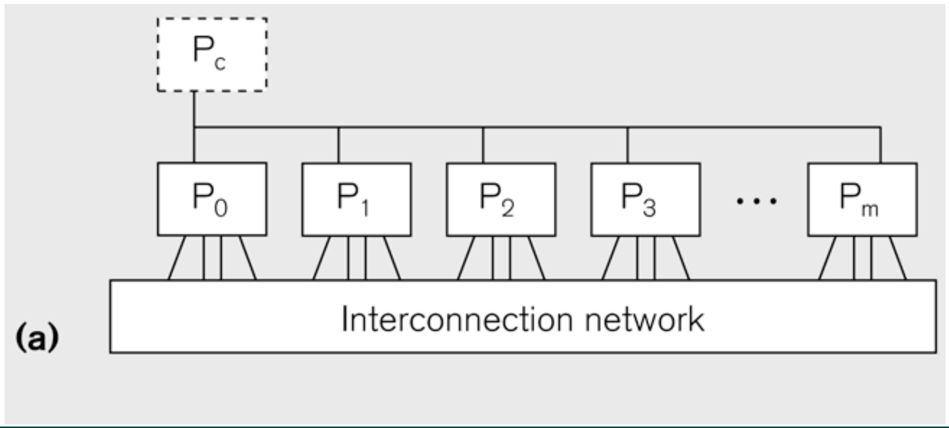
Organization of a Distributed-Memory Multiprocessor

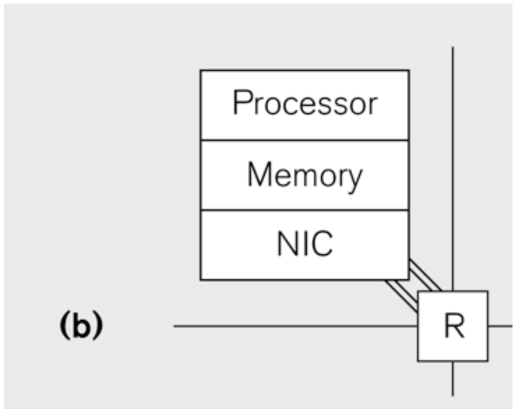
Figure (a)

- Host node (Pc) connected to a cluster of processor nodes (P_0 … P_m)
- Processors P_0 … P_m communicate via a dedicated high-performance interconnection network (e.g., Infiniband)
  - —Supports much lower latencies and higher bandwidth than standard TCP/IP networks

Figure (b)

- Each processor node consists of a processor, memory, and a Network Interface Card (NIC) connected to a router node (R) in the interconnect





Principles of Message-Passing Programming

- The logical view of a machine supporting the message-passing paradigm 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.
- These two constraints, while onerous, make underlying costs very explicit to the programmer.
- In this loosely synchronous model, processes synchronize infrequently to perform interactions. Between these interactions, they execute completely asynchronously.
- Most message-passing programs are written using the single program multiple data (SPMD) model.

SPMD Pattern (Lecture 26)

- SPMD: Single Program Multiple Data
- Run the same program on P processing elements (PEs)
- Use the "rank" … an ID ranging from 0 to (P-1) … to determine what computation is performed on what data by a given PE
- Different PEs can follow different paths through the same code
- Convenient pattern for hardware platforms that are not amenable to efficient forms of dynamic task parallelism
  - General-Purpose Graphics Processing Units (GPGPUs)
  - Distributed-memory parallel machines
- Key design decisions --- how should data and computation be distributed across PEs?

Using the SPMD model with a Global View of Data: Iterative Averaging (Slide 9, Lecture 13)

```
double[] gVal=new double[n+2]; double[] gNew=new double[n+2];
gVal[n+1] = 1; // Boundary condition
int Cj = Runtime.getNumOfWorkers();
forall (point [jj]:[0:Cj-1]) { // SPMD computation with "id" = jj
  double[] myVal = gVal; double[] myNew = gNew; // Local copy
  for (point [iter] : [0:numIters-1]) {
    // Compute MyNew as function of input array MyVal
    for (point [j]:getChunk([1:n],[Cj],[jj]))
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
    next; // Barrier before executing next iteration of iter loop
    // Swap myVal and myNew (replicated computation)
     double[] temp=myVal; myVal=myNew; myNew=temp;
    // myNew becomes input array for next iter
  } // for
} // forall
```

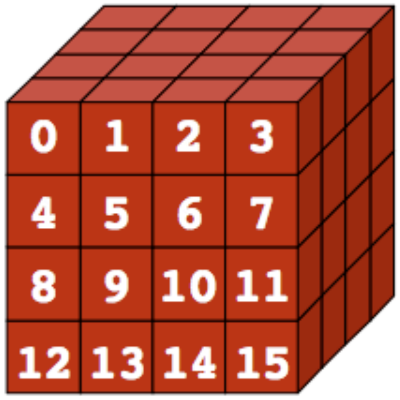
Data Distribution: Local View in Distributed-Memory Systems

Distributed memory

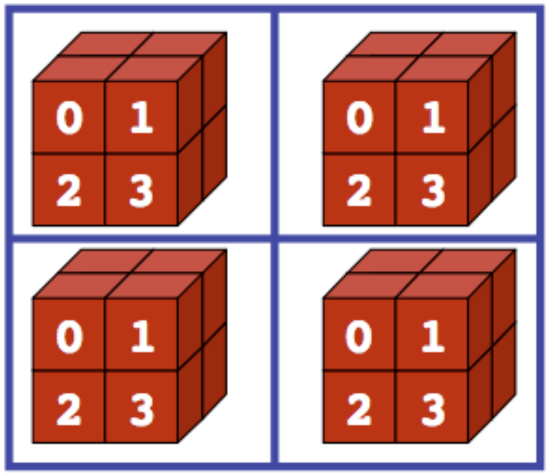
- Each process sees a local address space
- Processes send messages to communicate with other processes

Data structures

- Presents a Local View instead of Global View
- Programmer must make the mapping



Global View

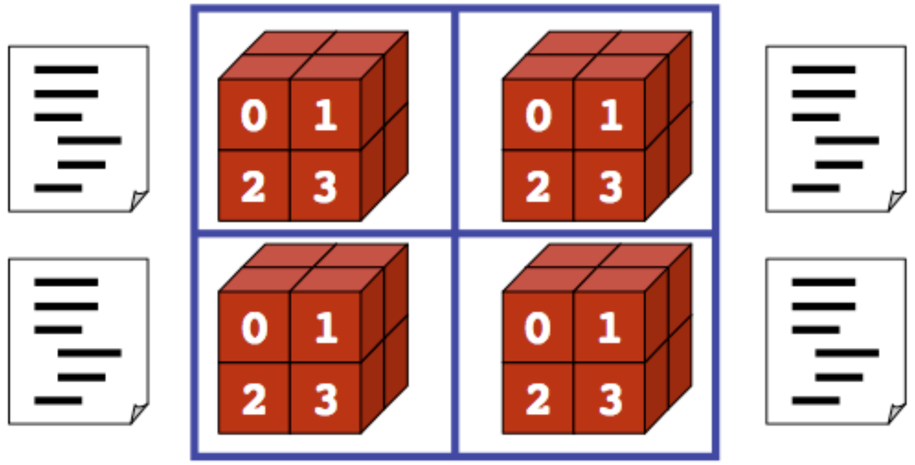


Local View (4 processes)

Using the SPMD model with a Local View

SPMD code

– Write one piece of code that executes on each processor



Local View (4 processes)

Processors must communicate via messages for non-local data accesses

- Similar to communication constraint for actors (except that we allowed hybrid combinations of global task parallelism and local actor parallelism in HJ)

MPI: The Message Passing Interface

- **Sockets and Remote Method Invocation (RMI) are communication primitives used for distributed Java programs.**
  - —Designed for standard TCP/IP networks rather than high-performance interconnects
- **The Message Passing Interface (MPI) standard was designed to exploit high-performance interconnects**
  - —MPI was standardized in the early 1990s by the MPI Forum—a substantial consortium of vendors and researchers
    - http://www-unix.mcs.anl.gov/mpi
  - —It is an API for communication between nodes of a distributed memory parallel computer
  - —The original standard defines bindings to C and Fortran (later C++)
    - Java support is available from a research project, mpiJava, developed at Indiana University 10+ years ago

      http://www.hpjava.org/mpiJava.html

Features 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 Minimal Set of MPI Routines (mpiJava)

- MPI.Init(args)
  - initialize MPI in each process
- MPI.Finalize()
  - terminate MPI
- MPI.COMM_WORLD.Size()
  - number of processes in COMM_WORLD communicator
- MPI.COMM_WORLD.Rank()
  - rank of this process in COMM_WORLD communicator
- Note:
  - In this subset, processes act independently with no information communicated among the processes.
  - "embarrassingly parallel", Cleve Moler.

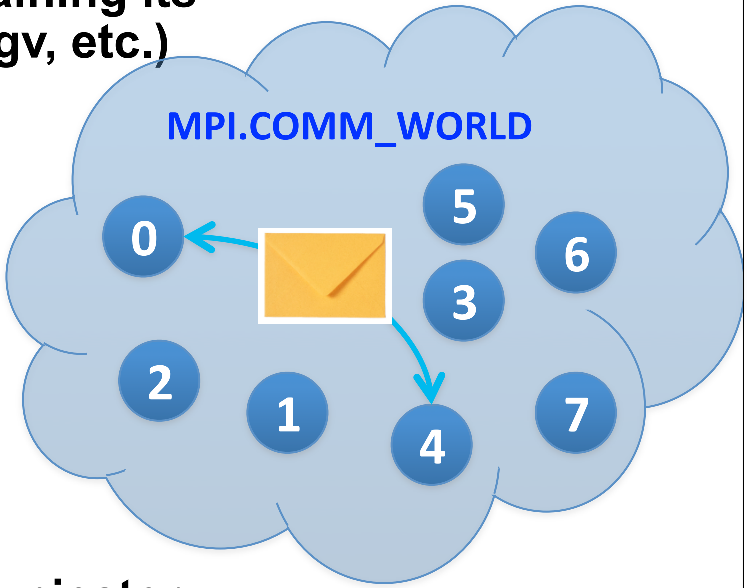
Our First MPI Program (mpiJava version)

main() is enclosed in an implicit "forall" --- each process runs a separate instance of main() with "index variable" = myrank

```
import mpi.*;
class Hello {
    static public void main(String[] args) {
        // Init() be called before other MPI calls
        MPI.Init(args); /
        int npes = MPI.COMM_WORLD.Size()
        int myrank = MPI.COMM_WORLD.Rank() ;
        System.out.println("My process number is " + myrank);
        MPI.Finalize(); // Shutdown and clean-up
    }
}
```

MPI Communicators

- 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



Adding Send() and Recv() to the Minimal Set of MPI Routines (mpiJava)

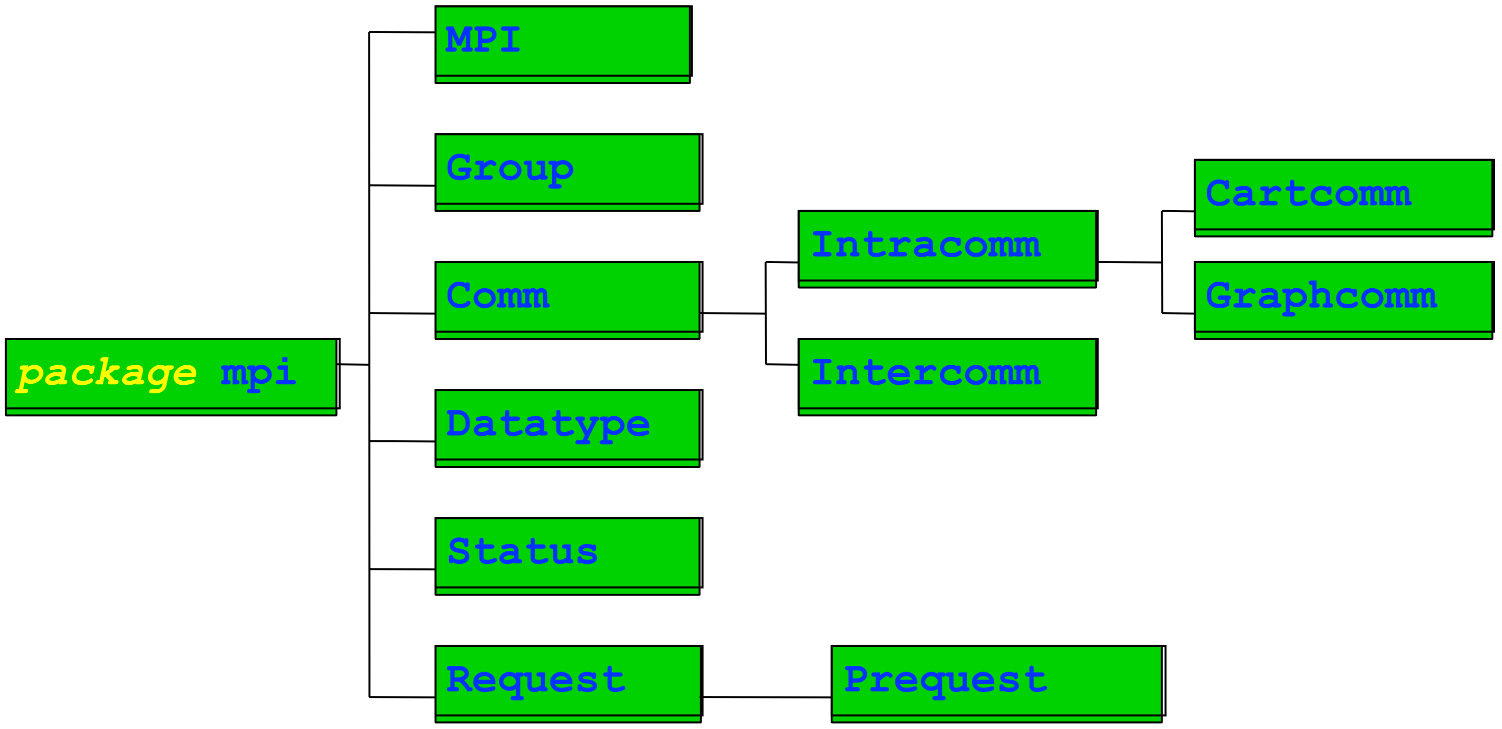
- MPI.Init(args)
  - initialize MPI in each process
- MPI.Finalize()
  - terminate MPI
- MPI.COMM_WORLD.Size()
  - number of processes in COMM_WORLD communicator
- MPI.COMM_WORLD.Rank()
  - rank of this process in COMM_WORLD communicator
- MPI.COMM_WORLD.Send()
  - send message using COMM_WORLD communicator
- MPI.COMM_WORLD.Recv()
  - receive message using COMM_WORLD communicator

$\updownarrow$ Point-to-point commn

MPI Blocking Point to Point Communication: Basic Idea

- A very simple communication between two processes is:
  - process zero sends ten doubles to process one
- In MPI this is a little more complicated than you might expect.
- Process zero has to tell MPI:
  - to send a message to process one
  - that the message contains ten entries
  - the entries of the message are of type double
  - the message has to be tagged with a label (integer number)
- Process one has to tell MPI:
  - to receive a message from process zero
  - that the message contains ten entries
  - the entries of the message are of type double
  - the label that process zero attached to the message

mpiJava Class hierarchy



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 blocking operations by default
  - Analogous to a phaser next operation

Example of Send and Recv

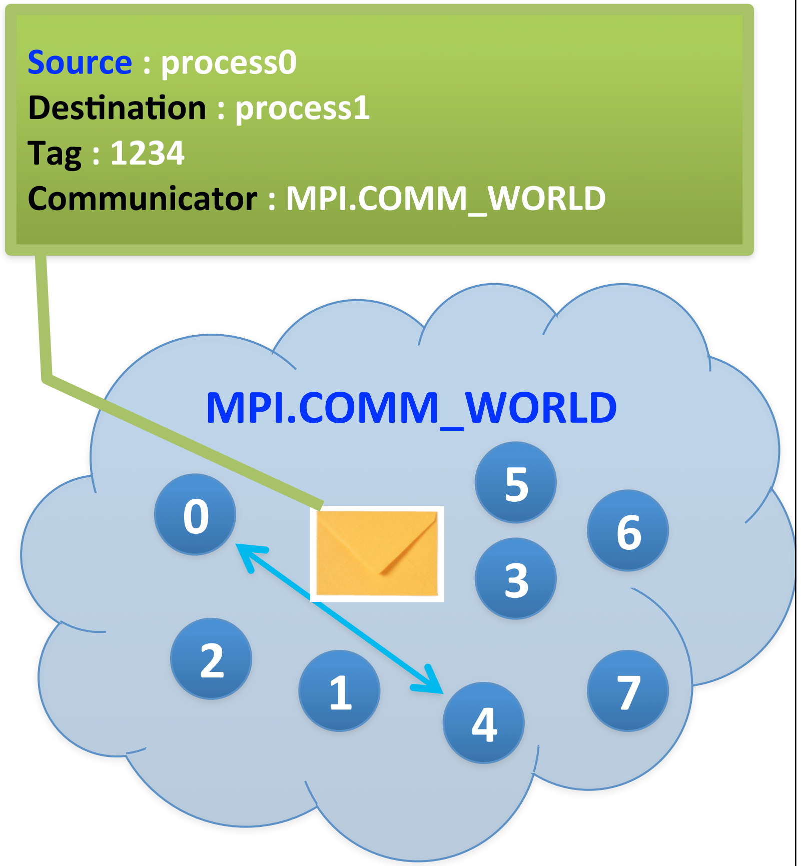
```
import mpi.*;

class myProg {
  public static void main( String[] args ) {
    int tag0 = 0;
    MPI.Init( args );                   // Start MPI computation
    if ( MPI.COMM_WORLD.rank() == 0 ) { // rank 0 = sender
      int loop[] = new int[1]; loop[0] = 3;
      MPI.COMM_WORLD.Send( "Hello World!", 0, 12, MPI.CHAR, 1, tag0 );
      MPI.COMM_WORLD.Send( loop, 0, 1, MPI.INT, 1, tag0 );
    } else {                            // rank 1 = receiver
      int loop[] = new int[1]; char msg[] = new char[12];
      MPI.COMM_WORLD.Recv( msg, 0, 12, MPI.CHAR, 0, tag0 );
      MPI.COMM_WORLD.Recv( loop, 0, 1, MPI.INT, 0, tag0 );
      for ( int i = 0; i < loop[0]; i++ ) System.out.println( msg );
    }
    MPI.Finalize( );                    // Finish MPI computation
  }
}
```

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

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


Source : process0
Destination : process1
Tag : 1234
Communicator : MPI.COMM_WORLD

Communication Buffers

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

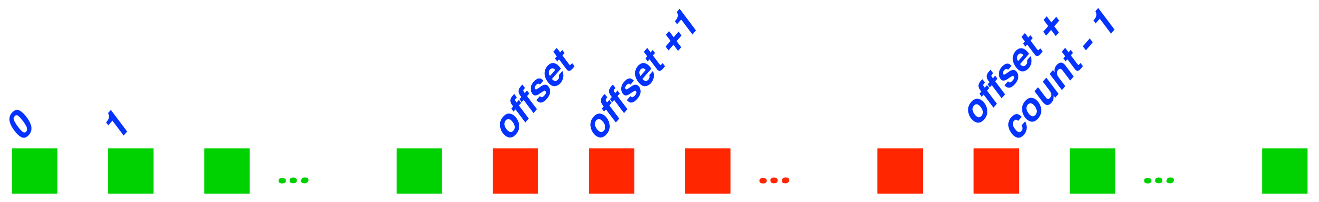
  Object buf, int offset, int count, Datatype type

- 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 fopr arrays, e.g.
    ```
    void Send(int [] buf, …)
    void Send(long [] buf, …)
    …
    ```
    and about 81 versions of some odd 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:



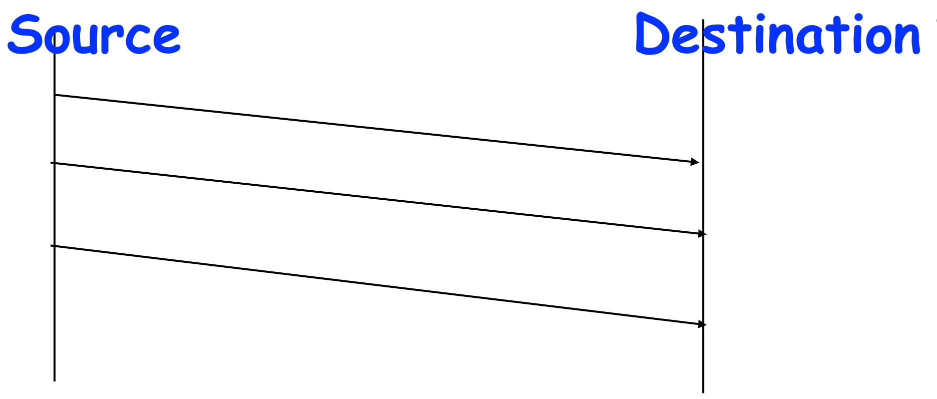
  - In the case of a send buffer, the red boxes represent elements of the buf array that are actually sent.
  - In the case of a receive buffer, the red boxes represent elements where the incoming data may be written (other elements will be unaffected). In this case count defines the maximum message size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

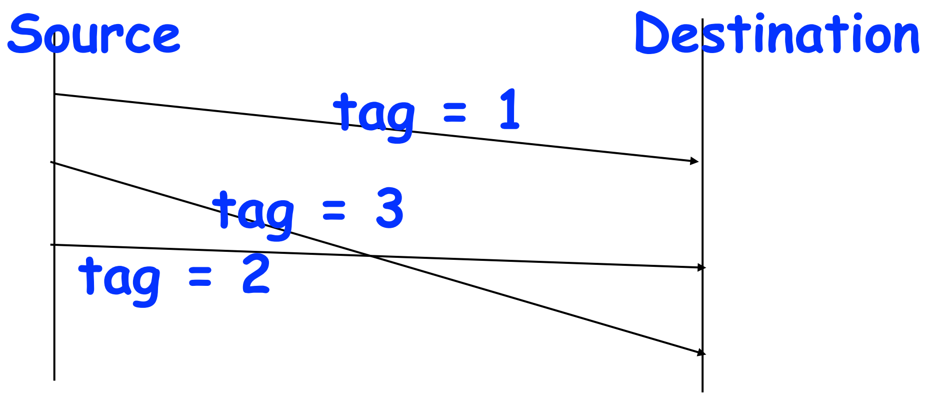
Basic Datatypes

- 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.

Reminders

- Graded midterms can be picked up from Amanda Nokleby in Duncan Hall 3137
- Homework 5 due by 5pm TODAY
- Homework 3 has been graded
  - Email will be sent this weekend
- Homework 6 will be posted this weekend