COMP 322: Fundamentals of Parallel Programming

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

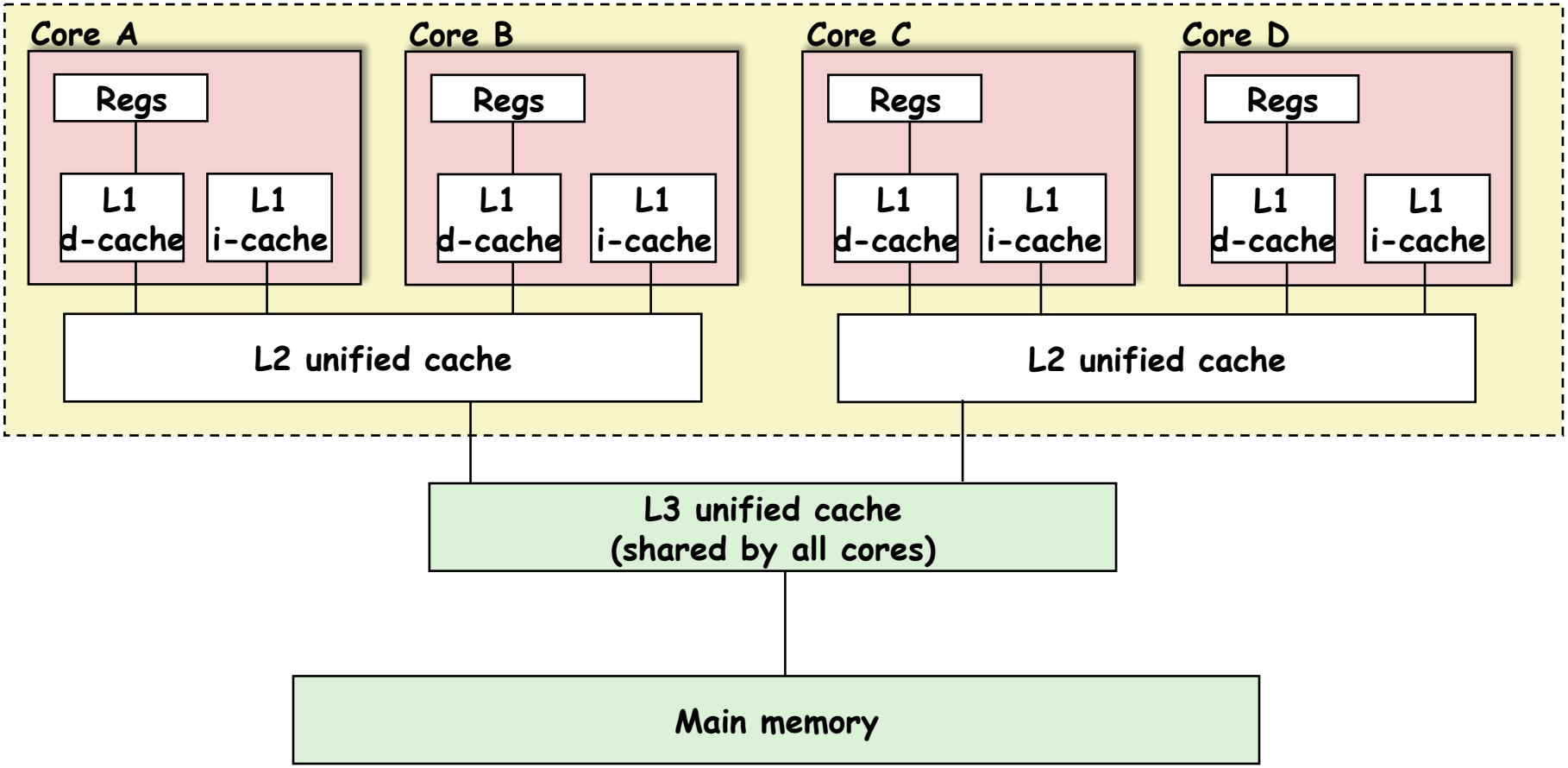
Lecture 36: Introduction to MPI

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

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 22)



- Memory hierarchy for a single Intel Xeon Quad-core E5440 HarperTown processor chip
  - A SUG@R node contains two such chips

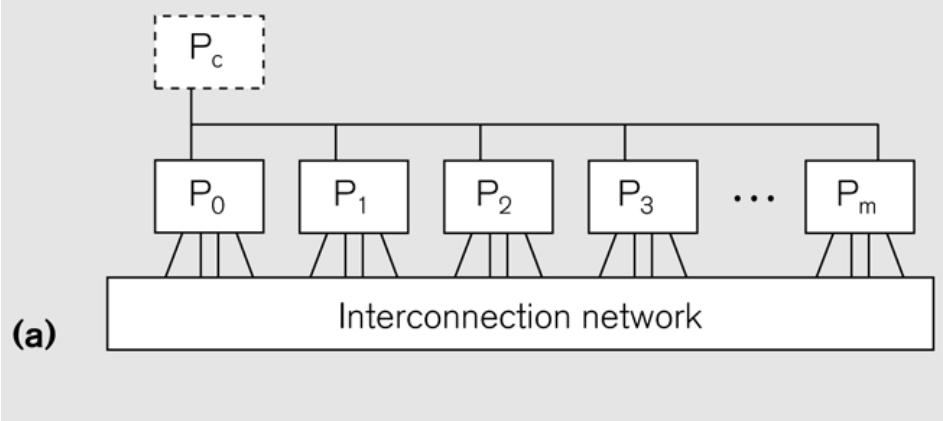
Organization of a Distributed-Memory Multiprocessor

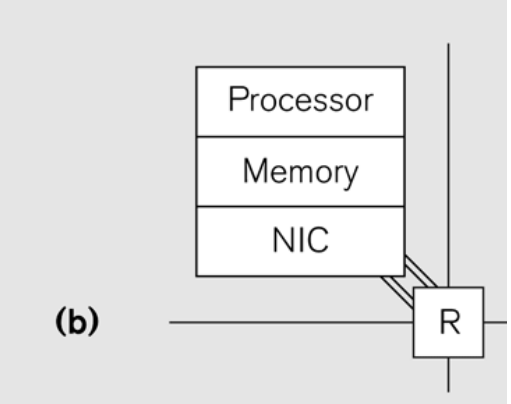
Figure (a)

- Host node (Pc) connected to a *cluster* of processor *nodes* (P_0 … P_m)
- Processors P_0 … P_m communicate via an *interconnection network*
  - 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.

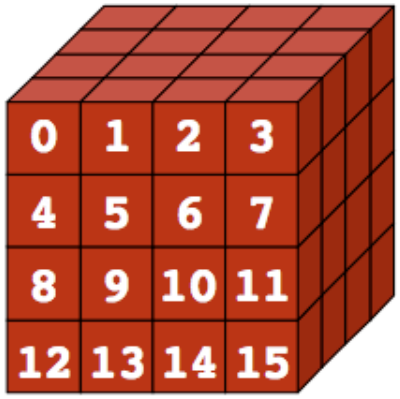
Global View vs. Local View

Distributed memory

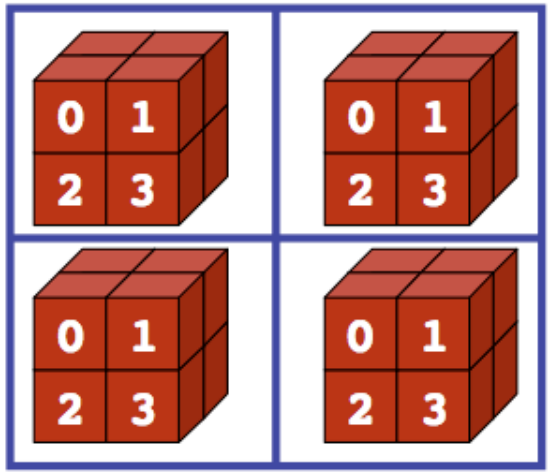
- Each process sees a local address space
- Processes send messages to communicate with other processes

Data structures

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



Global View

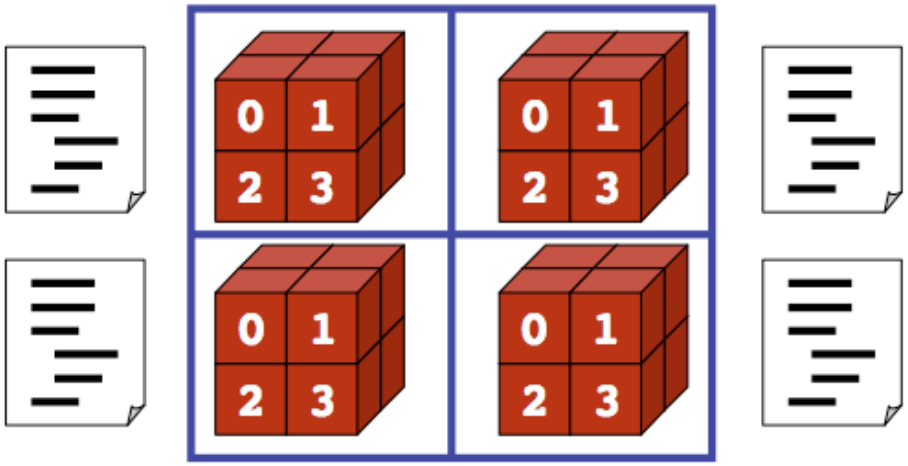


Local View (4 processes)

Single Program Multiple Data model (SPMD)

SPMD code

- Write one piece of code that executes on each processor



Local View (4 processes)

SPMD vs. SIMD?

- SIMD is a hardware execution model
- Each instruction executes in lock step
- SPMD is a software execution model– each process executes independently

MPI: The Message Passing Interface

- RMI originated in the Java world. Efforts like *JavaParty* and *Manta* aimed to bring RMI into the HPC world, by improving its performance.
- MPI is a technology from the HPC world, which various people have worked on importing into Java.
  - MPI is the HPC *Message Passing Interface* standardized in the early 1990s by the *MPI Forum*—a substantial consortium of vendors and researchers.
  - It is an API for communication between nodes of a distributed memory parallel computer (typically, now, a workstation cluster).
  - The original standard defines bindings to C and Fortran (later C++).
  - The low-level parts of API are oriented to: fast transfer of data from user program to network; supporting multiple modes of message synchronization available on HPC platforms; etc.
  - Higher level parts of the API are concerned with organization of process groups and providing the kind of collective communications seen in typical parallel applications.

Features of MPI

- MPI (http://www-unix.mcs.anl.gov/mpi) is an API for sending and receiving messages. But it goes further than this.
  - It is essentially a general platform for *Single Program Multiple Data* (SPMD) parallel computing on distributed memory architectures.
  - In this respect it is directly comparable with the *PVM* (Parallel Virtual Machine) environment that was one of its precursors.
- It introduced the important abstraction of a *communicator*, which is an object something like an N-way communication channel, connecting all members of a group of cooperating processes.
  - This was introduced partly to support using multiple parallel libraries without interference.
- It also introduced a novel concept of *datatypes*, used to describe the contents of communication buffers.
  - Introduced partly to support "zero-copying" message transfer.

The Minimal Set of MPI Routines

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

Our First MPI Program (mpiJava version)

```
import mpi.*;

class Hello {
    static public void main(String[] args) {
        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() ;
    }
}
```

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

Starting and Terminating the MPI Library

- `MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.
- `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.
- The prototypes of these two functions are (C version):

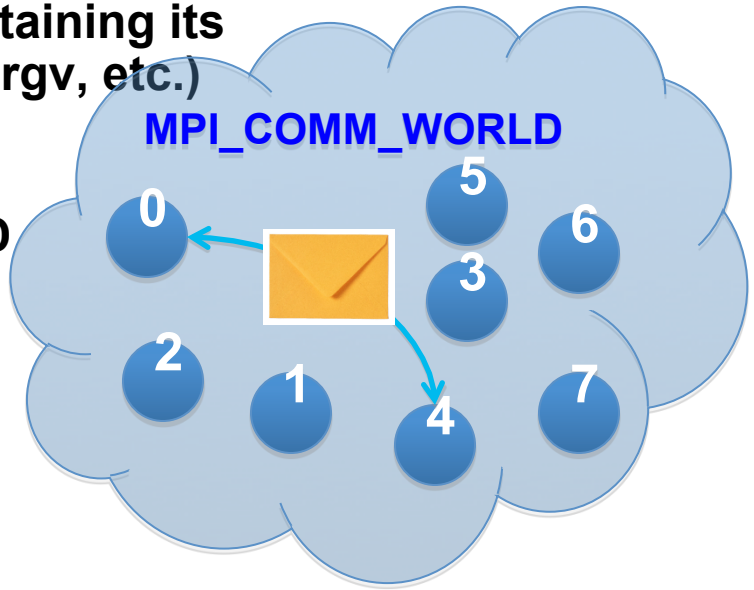
```
int MPI_Init(int *argc, char ***argv)

int MPI_Finalize()
```

- `MPI_Init` also strips off any MPI related command-line arguments.
- All MPI routines, data-types, and constants are prefixed by "`MPI_`". The return code for successful completion is `MPI_SUCCESS`.

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



Process Branching in SPMD programs

- In procnumber.c each process executed the same instructions.
- We can use conditional statements so that different processes perform operations unique to their number.
- In *procbranch.c* we first find the process number.

1. Each process checks to see if it is process 0. Only process 0 prints out "First"
2. Each process checks to see if it is process 1. Only process 1 prints out "Second"
3. Each process checks to see if it is process > 1. Only these processes print out "No medal"

```
/* procbranch.c  (C example) */
/* find the number attached to this process */
MPI_Comm_rank(MPI_COMM_WORLD, &procid);
/* this process will print a message based on procid */

if(procid==0)  printf("First\n");
if(procid==1)  printf("Second\n");
if(procid>1)  printf("No medal\n");
```


Running *procbranch*

- We compile and run as usual:
  - *mpicc -o procbranch procbranch.c*
  - *mpiexec -n 7 ./procbranch*

    Second
    First
    No medal
    No medal
    No medal
    No medal
    No medal
- Notice:
  - Again the processes do not report in numerical order
  - The text string output depends on the process number.

The Minimal Set of MPI Routines

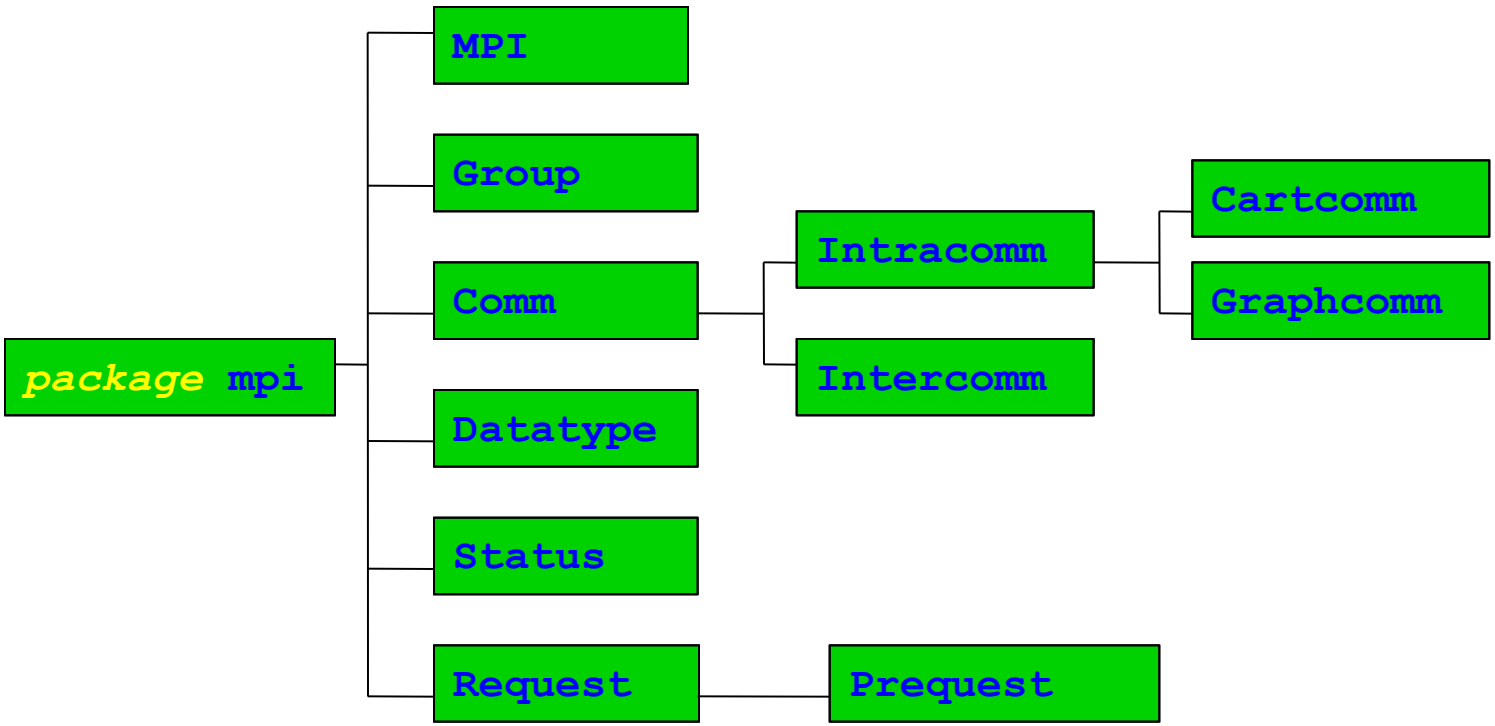
`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

- **Note:**
  - the processes have so far acted independently & no information has passed between the processes.
  - "embarrassingly parallel", Cleve Moler.

MPI 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

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 for not *declaring* it as an array was that one would then need to overload with about 9 versions of most methods, 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.

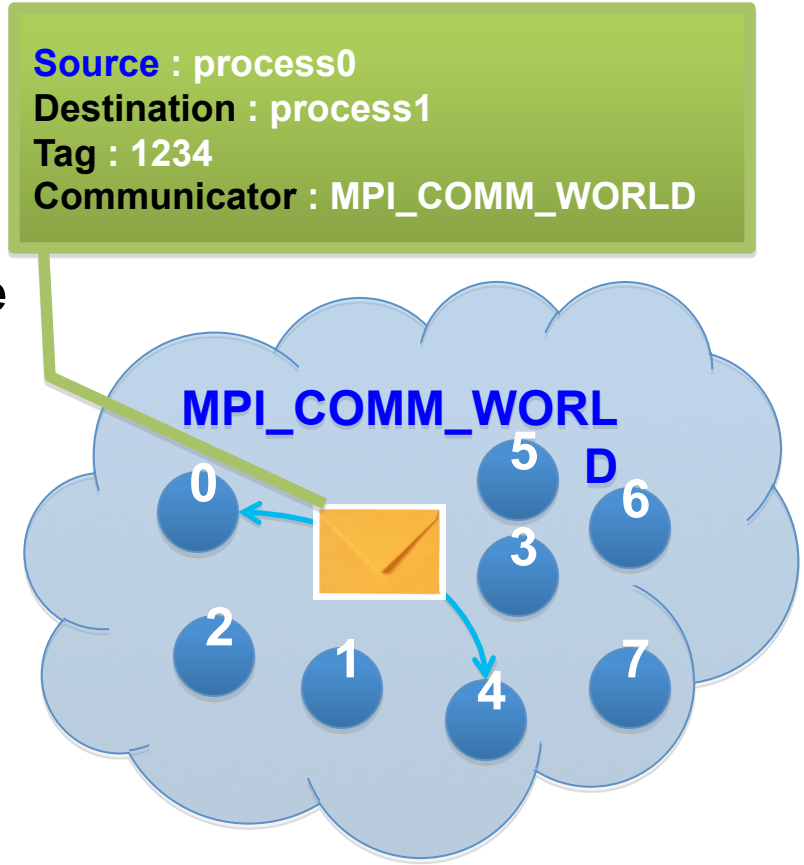
Basic Datatypes

- 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:

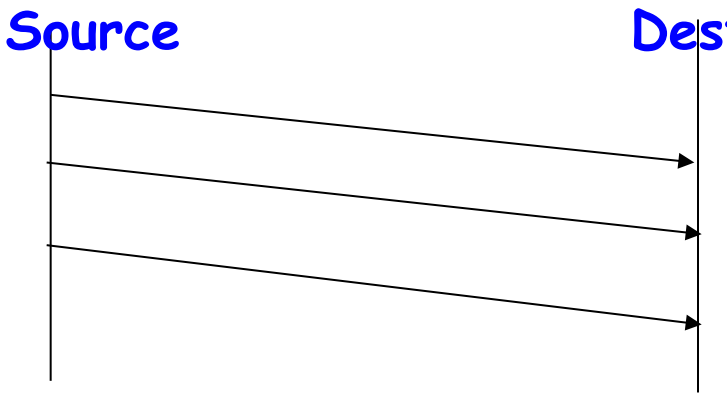
mpiJava datatype	Java type
MPI.BYTE	byte
MPI.CHAR	char
MPI.SHORT	short
MPI.BOOLEAN	boolean
MPI.INT	int
MPI.LONG	long
MPI.FLOAT	float
MPI.DOUBLE	double
MPI.OBJECT	Object

Message Envelope

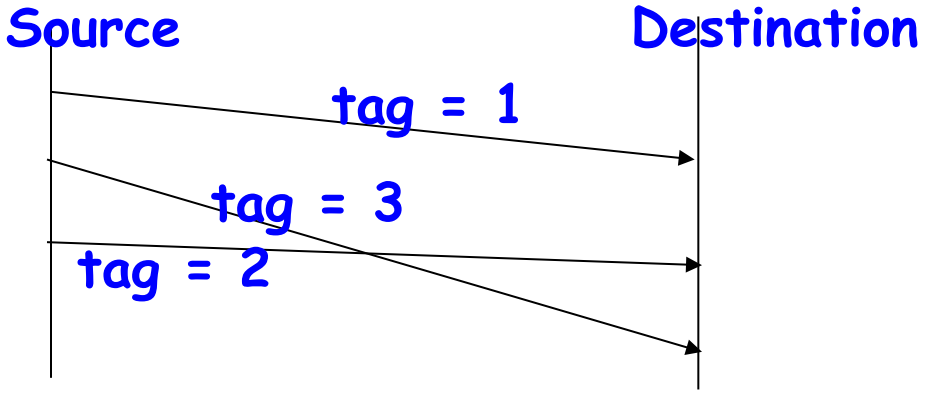
- Communication across process is performed using messages.
- Each message consists of a fixed number of fields that is used to distinguish them, called the Message Envelope :
  - —Envelope comprises source, destination, tag, communicator
  - —Message comprises Envelope + data
- Communicator refers to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



Message Ordering in MPI



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



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 provides access to several useful pieces about the message that arrived. Below we assume the Status object is saved to a variable called status:
  - int field status.source holds the rank of the process that sent the message (particularly useful if the message was received with MPI.ANY_SOURCE).
  - int field status.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 status.Get_count(type) returns number of items received in the message.
  - int method status.Get_elements(type) returns number of basic elements received in the message.
  - int field status.index is set by methods like Request.Waitany(), described later.

Sending and Receiving Messages

- On the receiving end, the status variable can be used to get information about the `MPI_Recv` operation.
- The corresponding data structure contains:

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR; };
```

- The MPI_Get_count function returns the precise count of data items received.

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype
                  datatype, int *count)
```


Communication Modes

- Following MPI, several communication modes are supported through a family of send methods. They differ mostly in their approaches to buffering and synchronization.
  - Send() implements MPI's *standard mode* semantics. The message *may* be buffered by the system, allowing Send() to return before a matching Recv() has been posted, but the implementation does not guarantee this.
  - Bsend() the system will attempt to buffer messages so that Bsend() method can return immediately. But it is the programmer's responsibility to tell the system how much buffer will be needed through MPI.Buffer_attach().
  - Ssend() is guaranteed to block until the matching Recv() is posted.
  - Rsend() is obscure—see the MPI standard.
- It is recommended that you use *standard mode sends*, and program defensively to guard against deadlocks (i.e. assume that the Send() method *may* block if the receiver is not ready).
  - Send() *may* behave like Bsend(), or it *may* behave like Ssend().

Avoiding Deadlocks (C versions)

Consider:

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

If MPI_Send is blocking, there is a deadlock.