Lecture 31: Introduction to the Message Passing Interface (MPI)
(Start of Module 3 on Distribution and Locality)

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Worksheet 30 Solution: Characterizing Solutions to the Dining Philosophers Problem

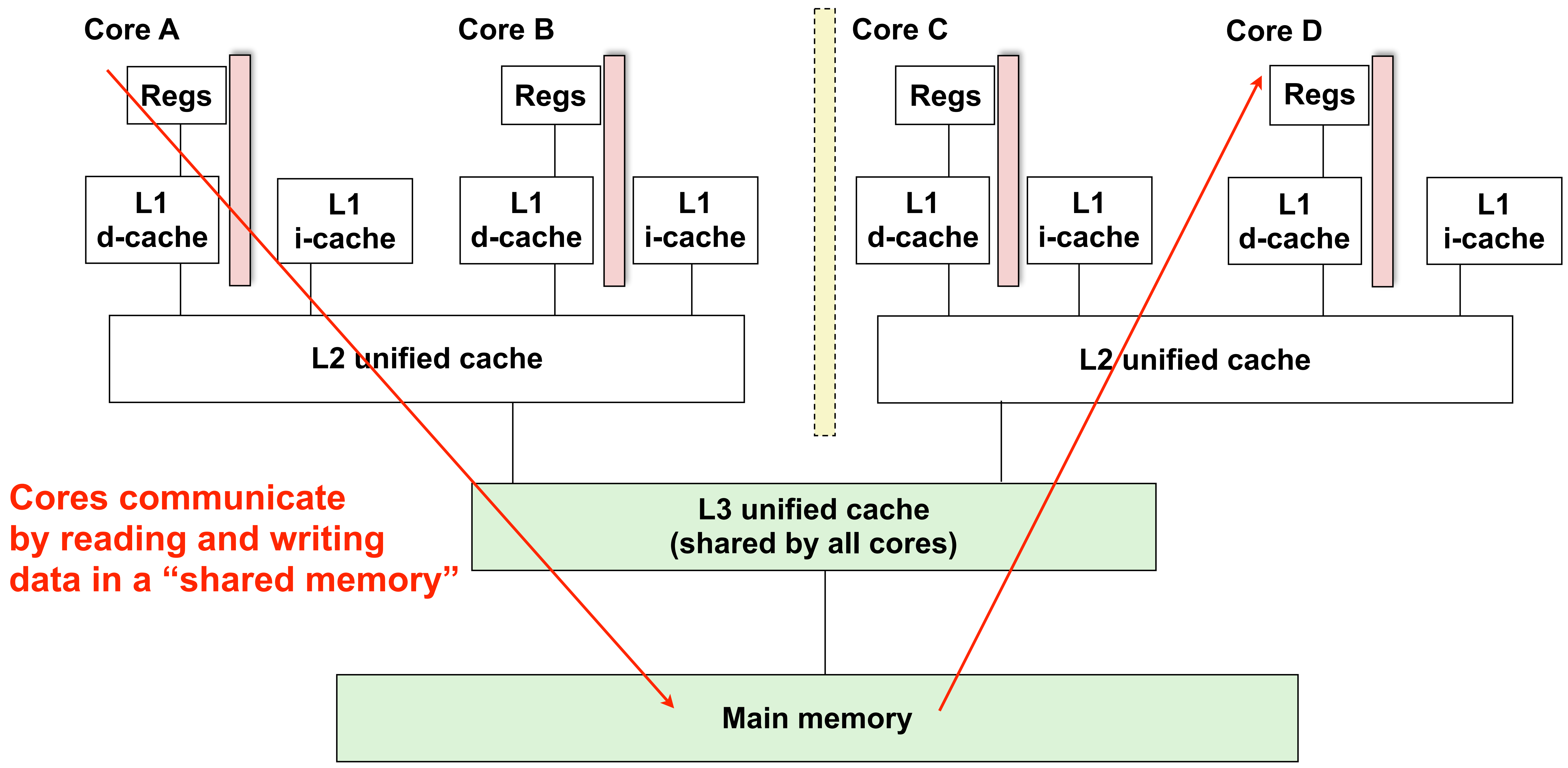
	Deadlock	Livelock	Starvation	Non-concurrency
Solution 1: synchronized	Yes	No	Yes	Yes
Solution 2: tryLock/unLock	No	Yes	Yes	Yes
Solution 3: isolated	No	No	Yes	Yes
Solution 4: object-based isolation	No	No	Yes	No
Solution 5: semaphores	No	No	No	No

Acknowledgements for Today's Lecture

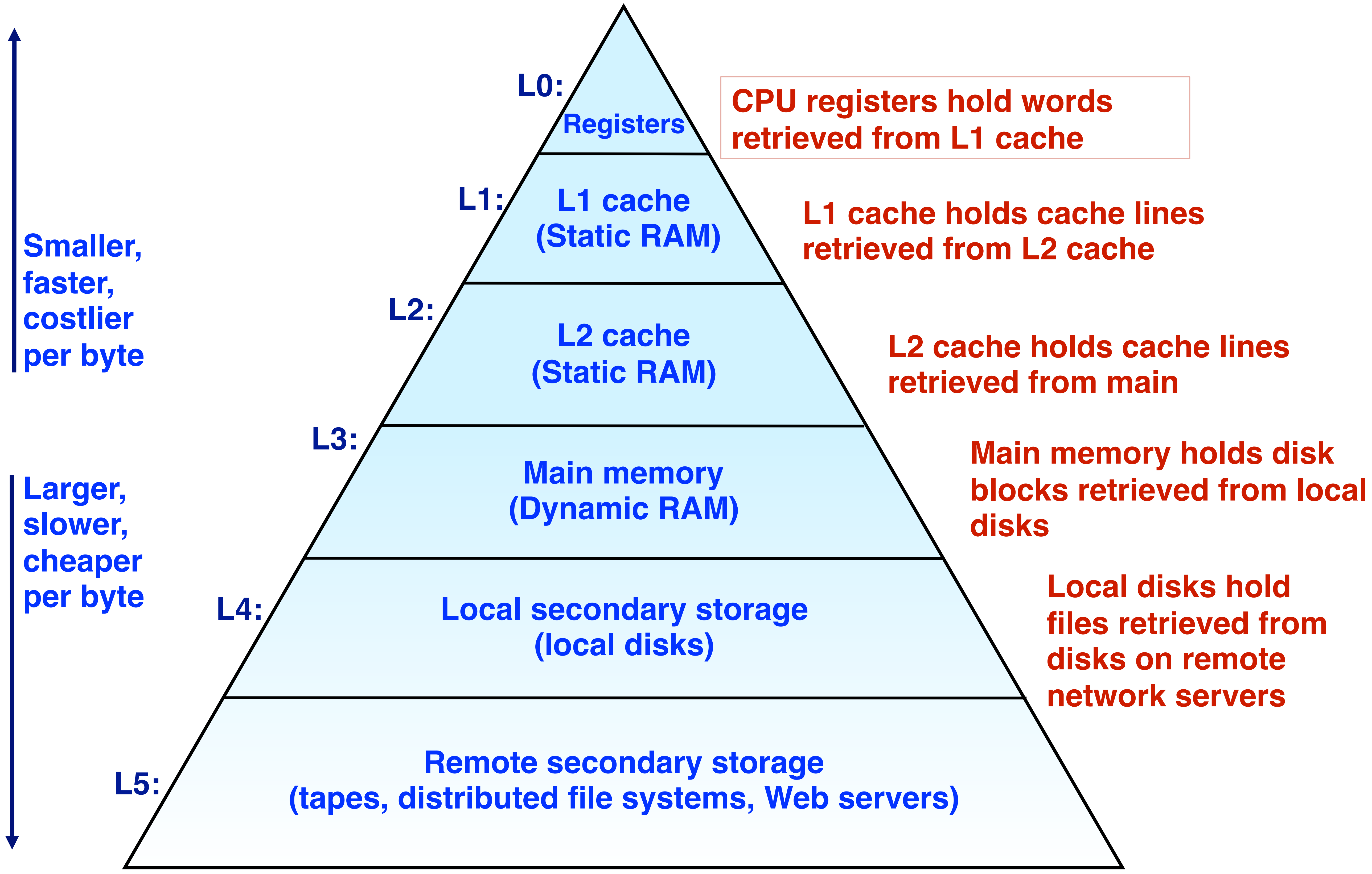
- "Principles of Parallel Programming", Calvin Lin & Lawrence Snyder
  - —Includes resources available at http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html
- "Parallel Architectures", Calvin Lin
  - —Lectures 5 & 6, CS380P, Spring 2009, UT Austin
  - —http://www.cs.utexas.edu/users/lin/cs380p/schedule.html
- 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 Symmetric Multiprocessor (SMP)

Memory hierarchy for a single Intel Xeon (Nehalem) Quad-core processor chip



What is the cost of a Memory Access? An example Memory Hierarchy



Organization of a Distributed-Memory Multiprocessor

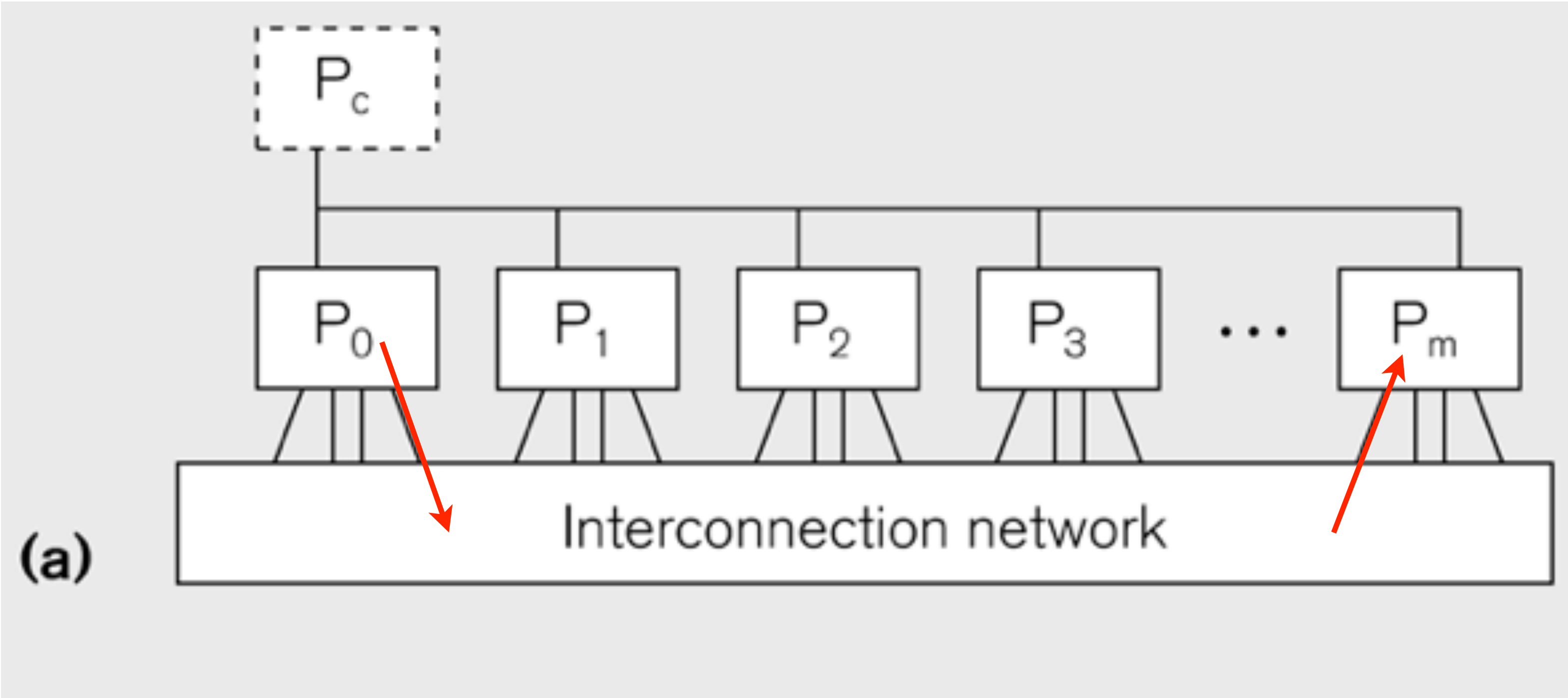
Figure (a)

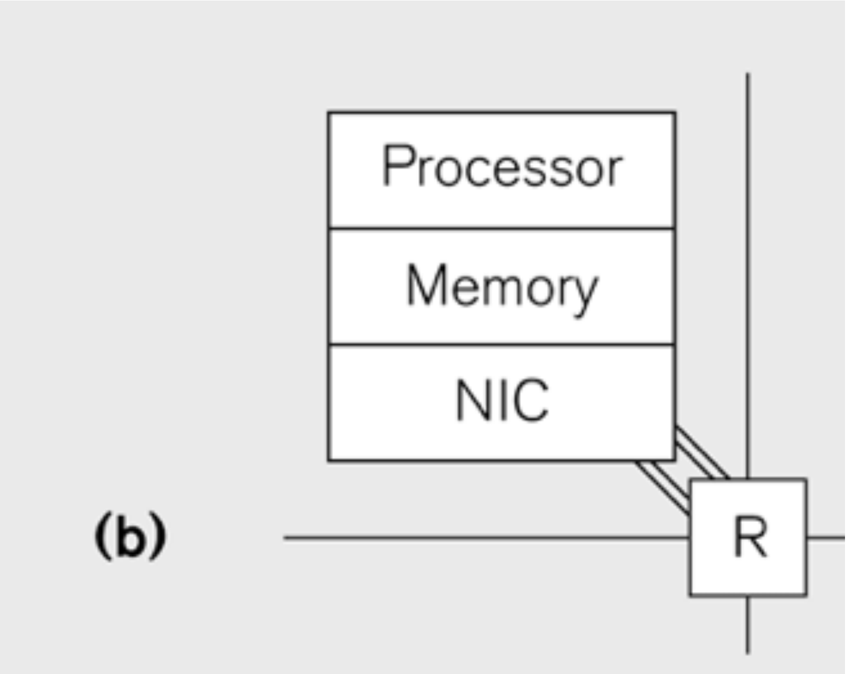
- Host node (P_c) connected to a cluster of processor nodes (P_0 … P_m)
- Processors P_0 … P_m communicate via an interconnection network which could be standard TCP/IP (e.g., for Map-Reduce) or specialized for high performance communication (e.g., for scientific computing)

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

Processors communicate by sending messages via an interconnect





Message Passing for Distributed-Memory Multiprocessors

- The logical view of a machine supporting the message-passing paradigm consists of *p processes*, each with its own exclusive address space, that are capable of executing on different nodes in a distributed-memory multiprocessor
  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 ("bulk synchronous") model, processes synchronize infrequently to perform interactions. Between these interactions, they execute completely asynchronously.

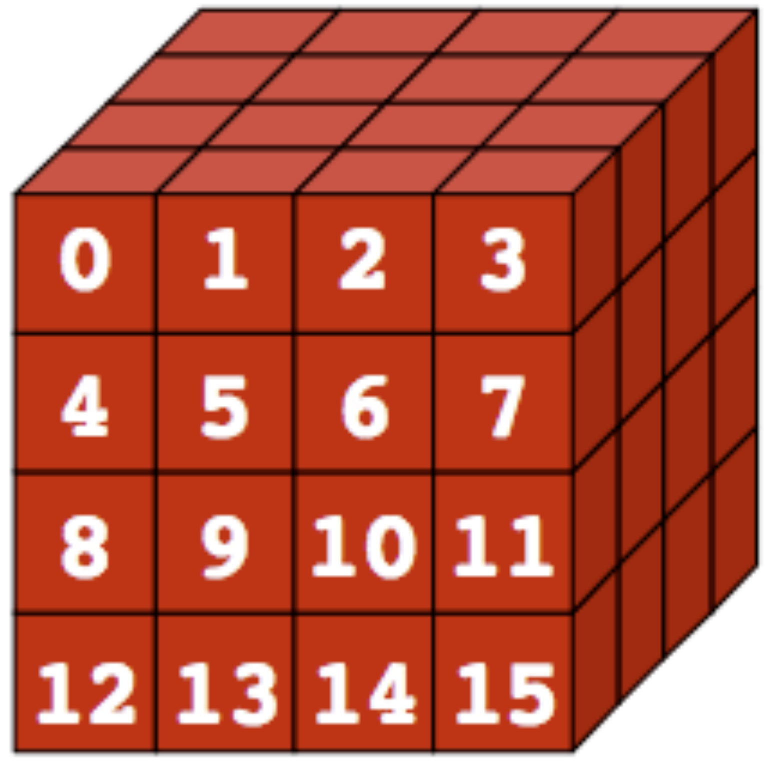
Data Distribution: Local View in Distributed-Memory Systems

Distributed memory

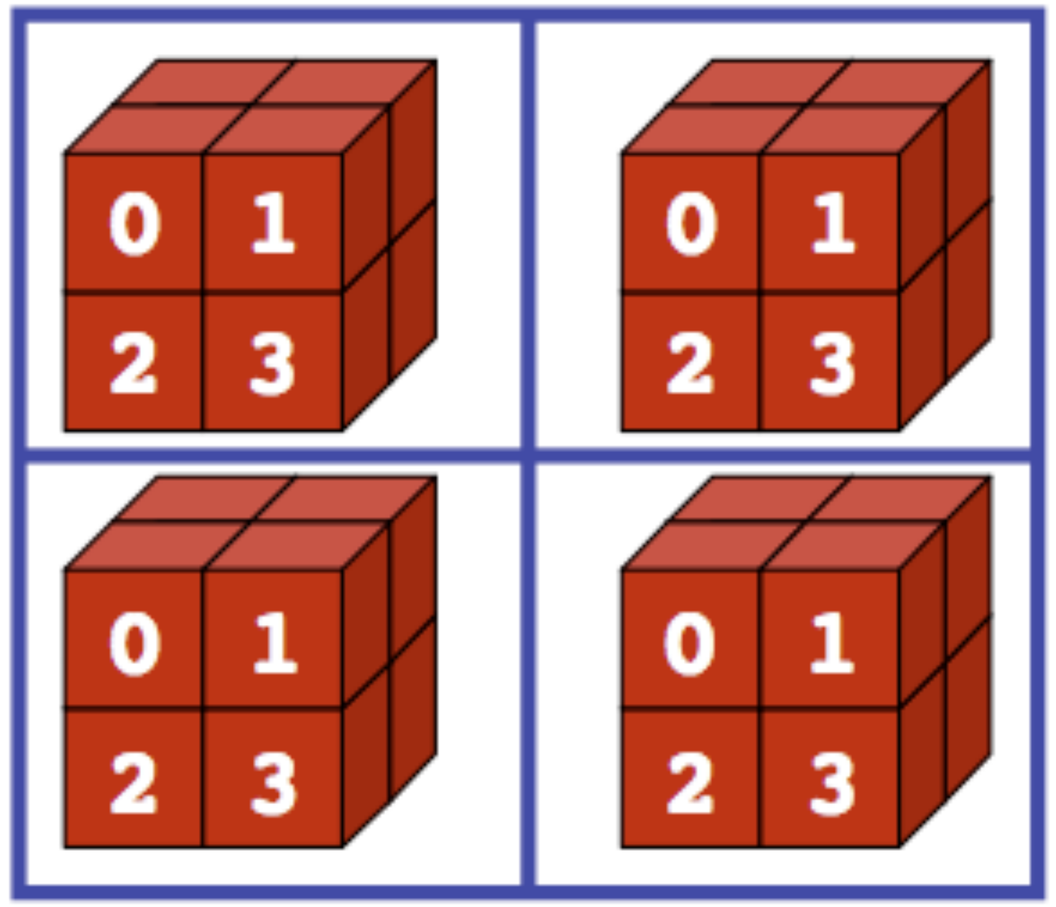
- Each process sees a local address space
- Processes send messages to communicate with other processes

Data structures

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



Global View



Local View (4 processes)

MPI: The Message Passing Interface

- 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

- Most MPI programs are written using the single program multiple data (SPMD) model

SPMD Pattern

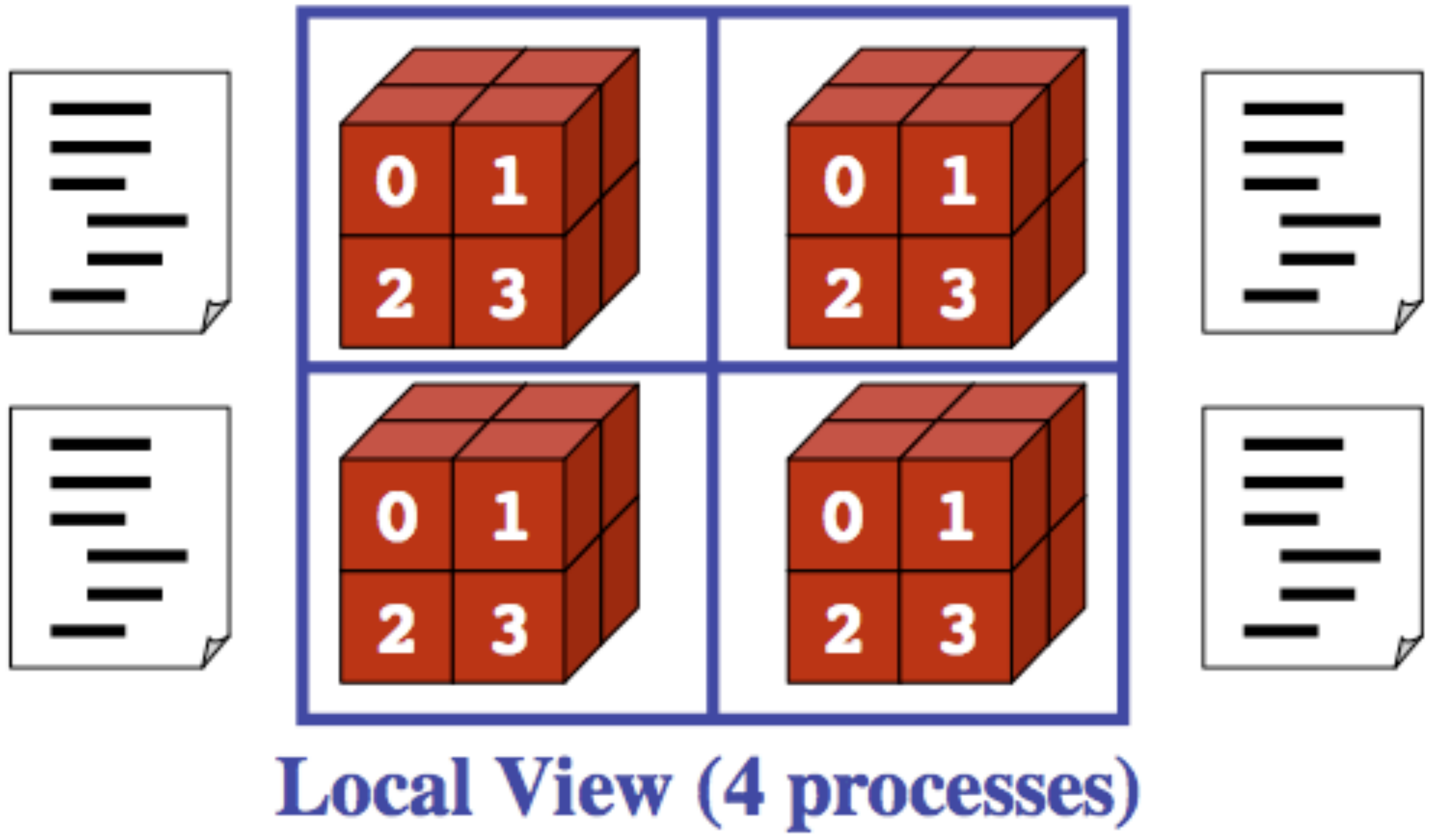
- SPMD: Single Program Multiple Data
- Run the same program on P processing elements (PEs)
- Use the "rank" … an ID ranging from 0 to (P-1) … to determine what computation is performed on what data by a given PE

  ⇒ you can think of the rank as the index of an implicit foralls across all PEs
- 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 Single Program Multiple Data model with a Local View

SPMD code

– Write one piece of code that executes on each processor



- Processors must communicate via messages for non-local data accesses
- Similar to communication constraint for actors — except that we allow hybrid combinations of task parallelism and actor parallelism in HJlib, but there is no integration (as yet) of HJlib or Java ForkJoin with mpiJava

The Minimal Set of MPI Routines

- `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:
  - — COMM_WORLD is the default communicator that includes all N processes, and numbers them with ranks from 0 to N-1

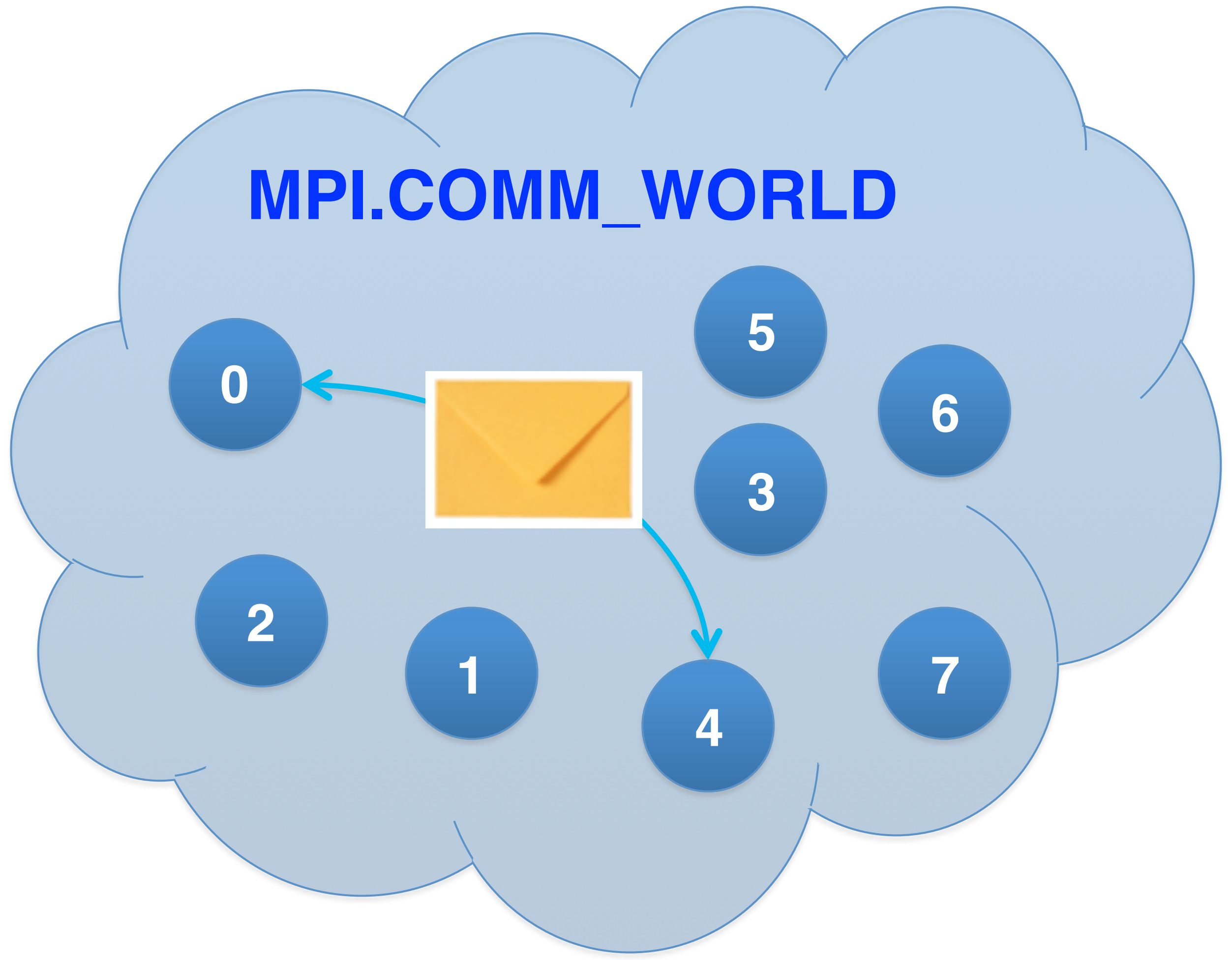
Our First MPI Program (mpiJava)

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

```java
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 creation of new processes in a communicator*
- 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

- `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

Point-to-point communication

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 Send and Receive

- Send and Recv methods in Comm object:

```
void Send(Object buf, int offset, int count,
          Datatype type, int dest, 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 to be sent and received.

- Both `Send()` and `Recv()` are <u>blocking</u> operations ==> potential for deadlock!
  — Send() waits for a matching Recv() from its dest rank with matching type and tag
  — Recv() waits for a matching Send() from its src rank with matching type and tag
  — Analogous to a phaser-specific `next` operation between two tasks registered in SIG_WAIT mode
  — The `Recv()` method also returns a `Status` value, discussed later.

Example of Send and Recv

```
import mpi.*;
class myProg {
  public static void main( String[] args ) {
    int tag0 = 0; int tag1 = 1;
    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, tag1 );
    } 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, tag1 );
      for ( int i = 0; i < loop[0]; i++ )
        System.out.println( msg );
    }
    MPI.Finalize( );                     // Finish MPI computation
  }
}
```

Send() and Recv() calls are blocking operations

Announcements & Reminders

- Hw #4 (Checkpoint #1) is due by Monday, April 13th at 11:59pm

- Quiz for Unit 7 is due Friday, April 17th at 11:59pm

Worksheet #31: 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. …
```