COMP 322: Fundamentals of Parallel Programming

Lecture 30: Introduction to Message Passing Interface (MPI)

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

Worksheet #29: Characterizing Solutions to the Dining Philosophers Problem

For the five solutions studied in Lecture #29, indicate in the table below which of the following conditions are possible and why:

1. Deadlock: when all philosopher tasks are blocked
2. Livelock: when all philosopher tasks are executing (i.e., no philosopher is blocked) but ALL philosophers are starved (never get to eat)
3. Starvation: when one or more philosophers are starved (never get to eat)
4. Non-Concurrency: when more than one philosopher cannot eat at the same time, even when resources are available i.e., not being used

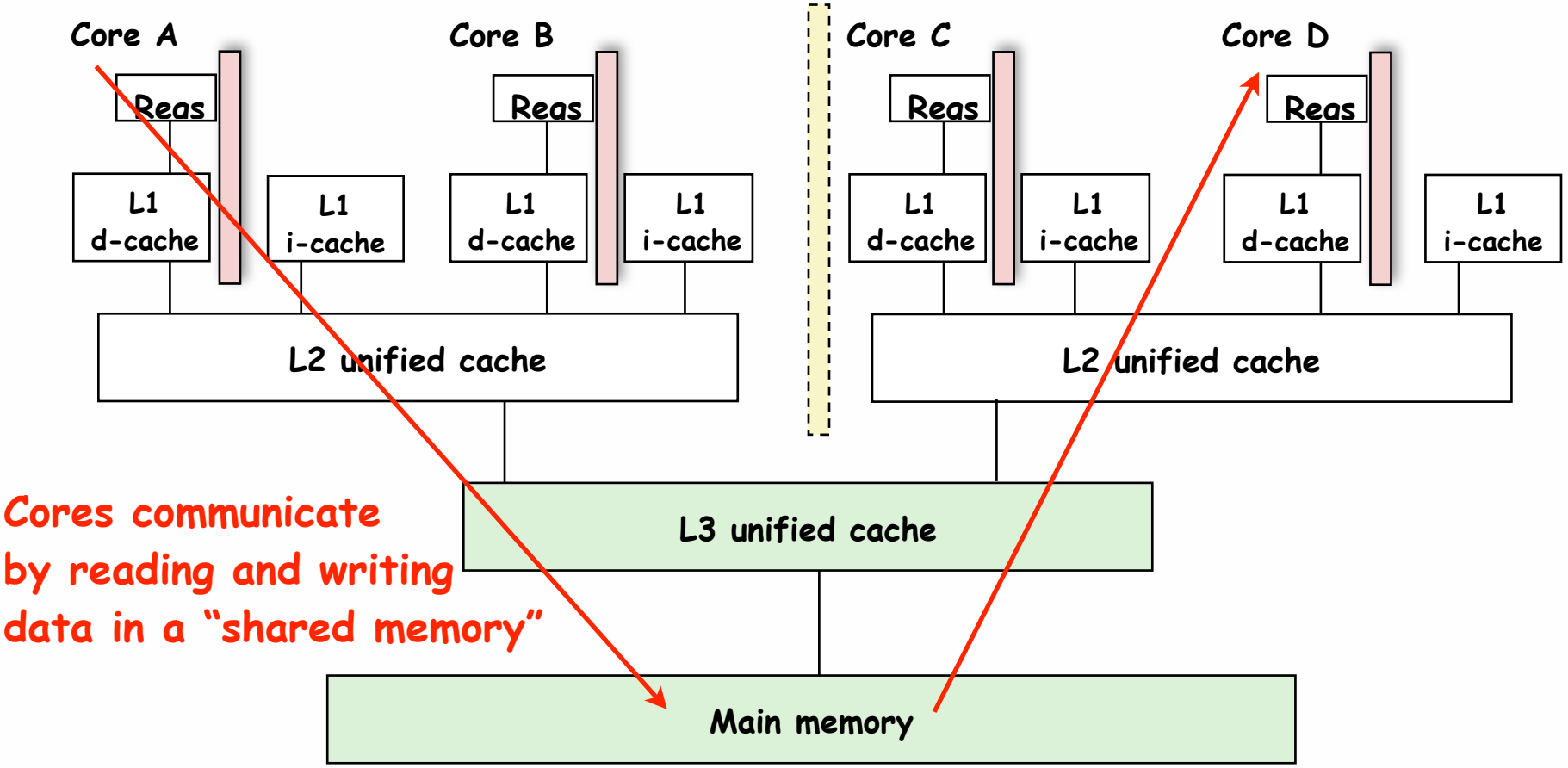
NOTES:

- Deadlock implies Starvation and Non-Concurrency
- Livelock implies Starvation and Non-Concurrency

	Deadlock	Livelock	Starvation	Non-concurrency
Solution 1: synchronized	Yes (100%)	No (78%)	Yes (78%)	Yes (35%)
Solution 2: tryLock/ unLock	No (93%)	Yes (98%)	Yes (89%)	Yes (24%)
Solution 3: isolated	No (100%)	No (96%)	Yes (87%)	Yes (94%)
Solution 4: object-based isolation	No (85%)	No (91%)	Yes (91%)	No (87%)
Solution 5: semaphores	No (93%)	No (93%)	No (67%)	No (63%)

Percentages show fractions of correct responses

Organization of a Shared-Memory Multicore Symmetric Multiprocessor (SMP)



- 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

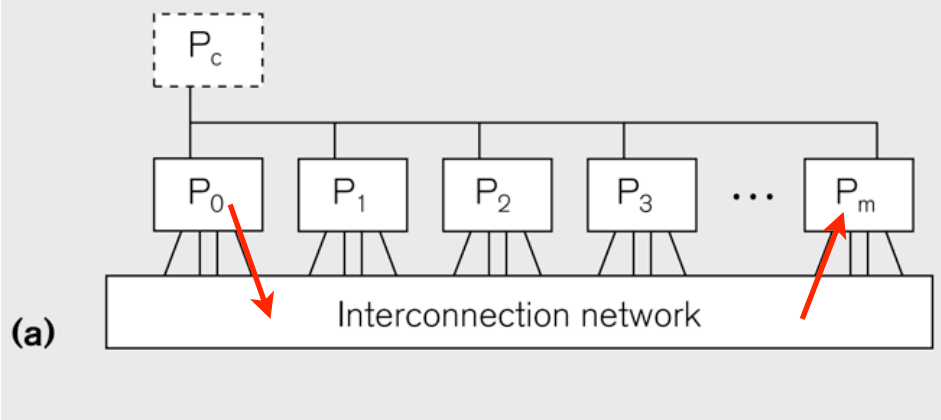
Figure (a)

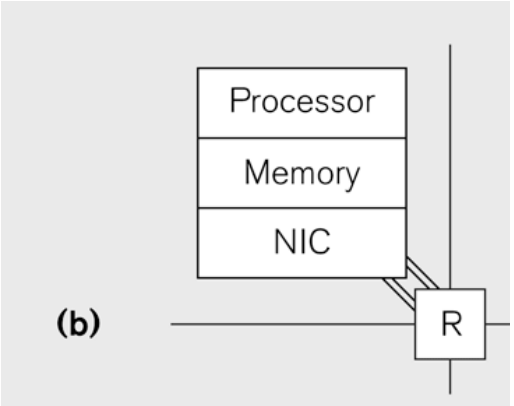
- **Host node (Pc) connected to a cluster of processor nodes (P_0 … P_m)**
- **Processors P_0 … P_m communicate via a dedicated high-performance interconnection network (e.g., Infiniband)**
  - **—Supports much lower latencies and higher bandwidth than standard TCP/IP networks**

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**

Processors communicate by sending messages via an 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

- 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?

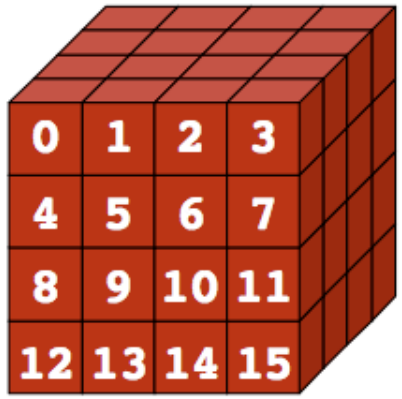
Data Distribution: Local View in Distributed-Memory Systems

Distributed memory

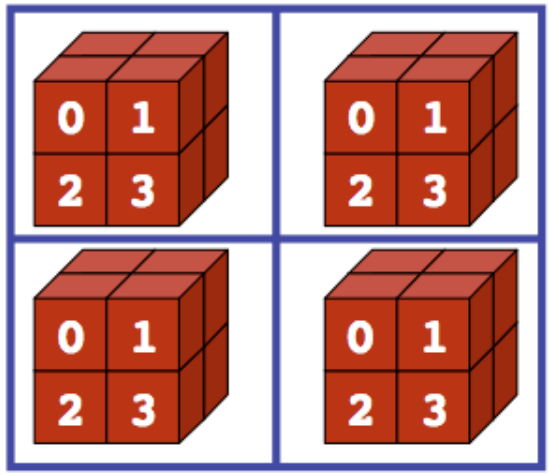
- Each process sees a local address space
- Processes send messages to communicate with other processes

Data structures

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



Global View

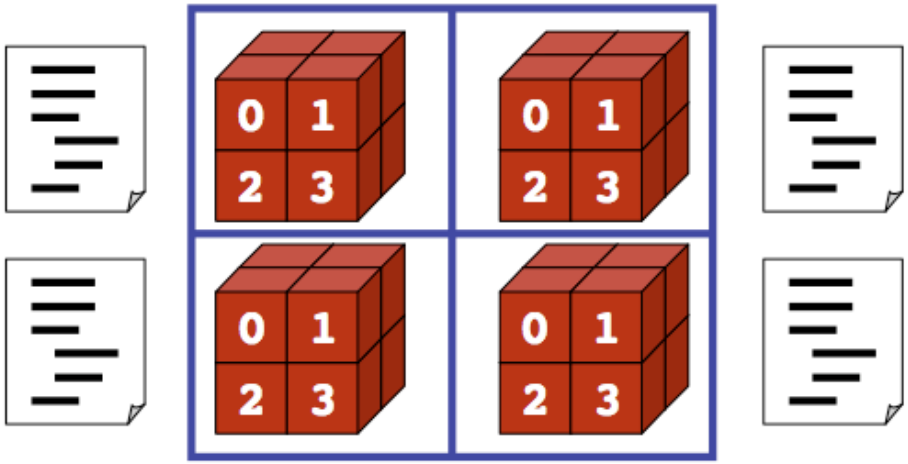


Local View (4 processes)

Using the Single Program Multiple Data (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 allow hybrid combinations of task parallelism and 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.

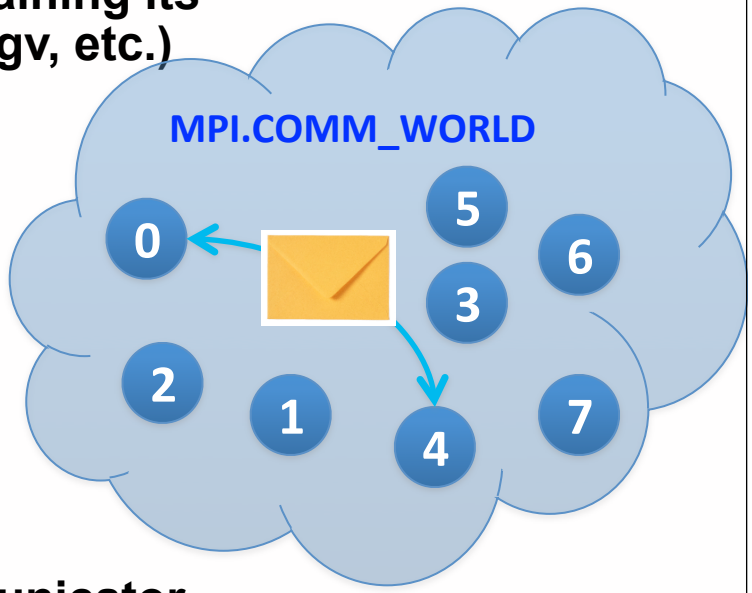
Our First MPI Program (mpiJava version)

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

```
1.import mpi.*;
2.class Hello {
3.    static public void main(String[] args) {
4.       // Init() be called before other MPI calls
5.       MPI.Init(args); /
6.       int npes = MPI.COMM_WORLD.Size()
7.       int myrank = MPI.COMM_WORLD.Rank() ;
8.       System.out.println("My process number is " + myrank);
9.       MPI.Finalize(); // Shutdown and clean-up
10.    }
11.}
```


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)

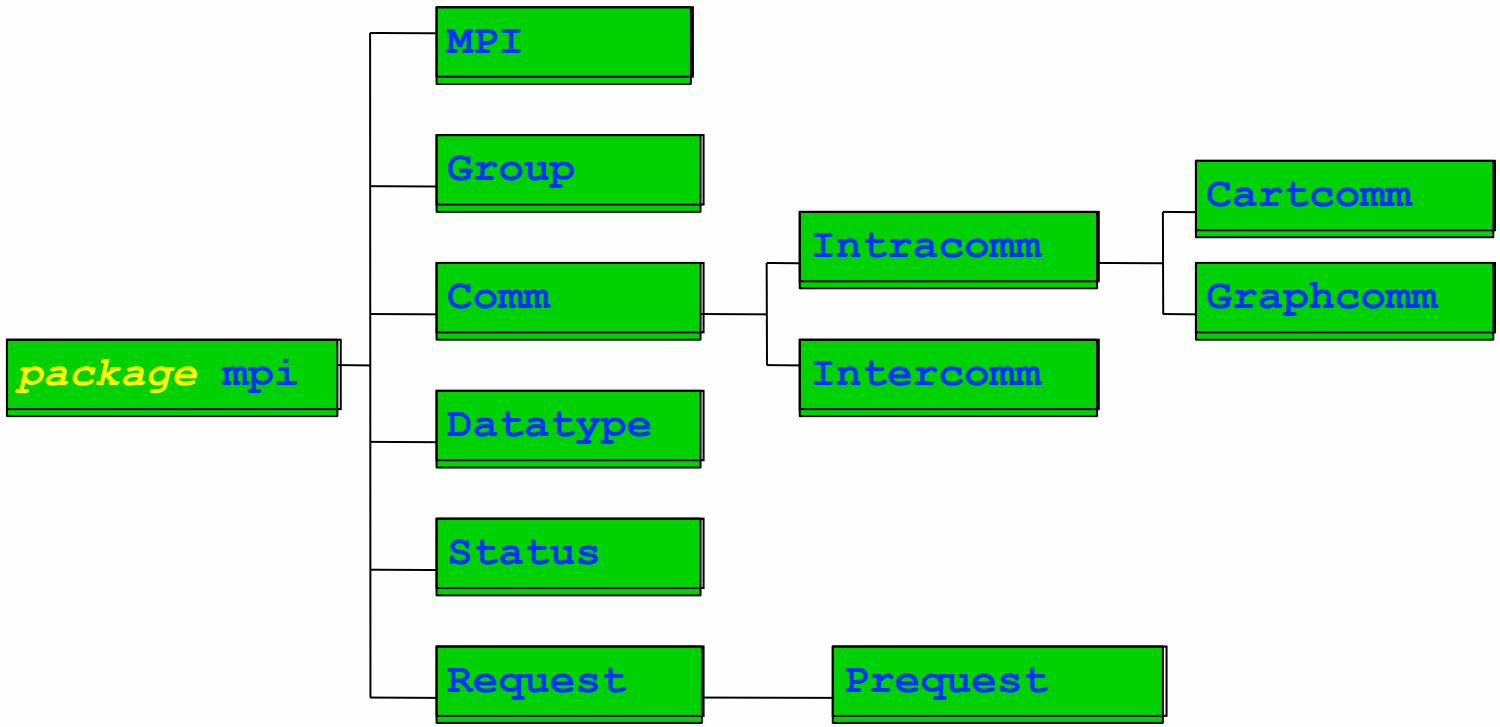
- MPI.Init(args)
  - —initialize MPI in each process
- MPI.Finalize()
  - —terminate MPI
- MPI.COMM_WORLD.Size()
  - —number of processes in COMM_WORLD communicator
- MPI.COMM_WORLD.Rank()
  - —rank of this process in COMM_WORLD communicator
- MPI.COMM_WORLD.Send()
  - —send message using COMM_WORLD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

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

```
1.import mpi.*;
2. class myProg {
3.  public static void main( String[] args ) {
4.    int tag0 = 0; int tag1 = 1;
5.    MPI.Init( args );                    // Start MPI computation
6.    if ( MPI.COMM_WORLD.rank() == 0 ) { // rank 0 = sender
7.      int loop[] = new int[1]; loop[0] = 3;
8.      MPI.COMM_WORLD.Send( "Hello World!", 0, 12, MPI.CHAR, 1, tag0 );
9.      MPI.COMM_WORLD.Send( loop, 0, 1, MPI.INT, 1, tag1 );
10.    } else {                              // rank 1 = receiver
11.      int loop[] = new int[1]; char msg[] = new char[12];
12.      MPI.COMM_WORLD.Recv( msg, 0, 12, MPI.CHAR, 0, tag0 );
13.      MPI.COMM_WORLD.Recv( loop, 0, 1, MPI.INT, 0, tag1 );
14.      for ( int i = 0; i < loop[0]; i++ ) System.out.println( msg );
15.    }
16.    MPI.Finalize( );                     // Finish MPI computation
17.  }
18.}
```

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

Worksheet #30: MPI send and receive

Name: _____ Netid: _____

```
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. ...
```

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.