COMP 322: Parallel and Concurrent Programming

Lecture 37: Concurrent and Parallel Languages and Frameworks

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Acknowledgements

- "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

What have we learned in this course?

- Functional programming for parallelism
- Lazy computation, streams
- Futures and promises
- Data-driven programming approach
- Computation graphs and their properties
- Map/Reduce programming model
- Data-parallel programming model
- Loop parallelism
- Locality control
- Handling concurrency while avoiding deadlock/livelock/starvation
- Barrier and point-to-point synchronization
- Actor programming model

Habanero

- Habanero-Java and Habanero-C
- Async/finish, futures/promises, loop parallelism, phasers, locality control, actors, isolation
- HJlib is a library implementation of these features
- Still developed and improved
- Python, Kotlin, Go, X10, MPI, Chapel, Java, C/C++
- There's also PCDP-Java
  - Coursera equivalent of COMP 322
- No streams

https://habanero.cc.gatech.edu/

X10

- Designed and developed at IBM
- One of the original "Next-generation" Asynchronous Partitioned Global Address Space projects
- Ancestor of Habanero Java
- Async, finish, loop parallelism, clocks (phasers), locality control
- No abstract metrics, data-driven execution, actors, streams

http://x10-lang.org/

Chapel

- Designed, implemented and maintained by Cray
- Partitioned Global Address Space
- Loop parallelism, task parallelism
- Locality control
- Distributed system execution
- Tasks, futures, promises
- No phasers, actors, abstract metrics, data-driven execution

https://chapel-lang.org/

Kotlin

- From the creators of IntelliJ
- Based on Java
- Multi-paradigm programming language
  - Functional, object-oriented
- Lots of support for functional programing
- More compact than Java
- Fully interoperable with Java
- Support for coroutines: very similar to asyncs and future tasks
- Low-level synchronization between tasks
- Channels
- No loop parallelism, phasers, abstract metrics, streams, locality control, actors

https://kotlinlang.org/

Go

- Multi-paradigm, object-oriented, concurrent language
- Goroutines (asyncs)
- Channels
- Concurrency control structures
  - Sending messages between coroutines
- No phasers, loop parallelism, futures/promises, abstract metrics, actors, locality control

https://go.dev/

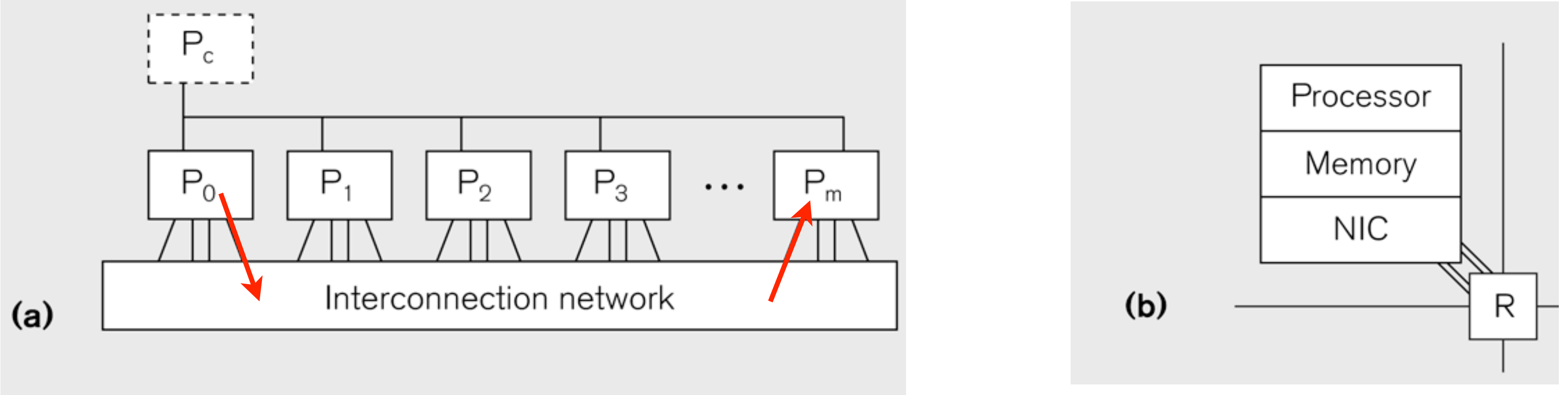
Python/Ray

- Library based approach
- Aimed at data science, machine learning, data processing
- Futures and actors
- No task-level parallelism on shared memory
- No abstract metrics, phasers, loop parallelism

https://www.ray.io/

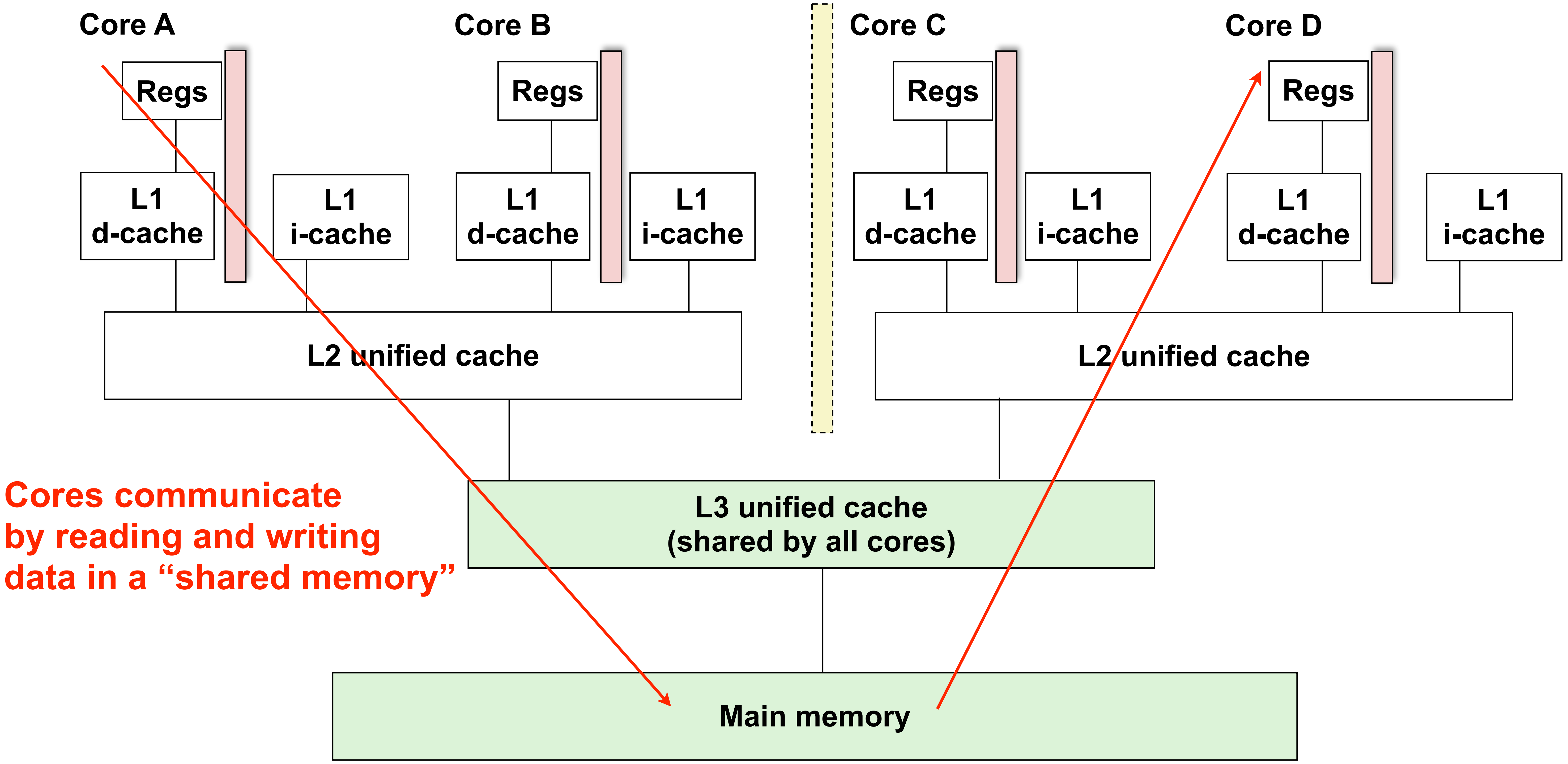
MPI

- Library framework
- Message-passing programming model
- Designed for distributed systems
- Implementations on top of several programming languages
  - C/C++
  - Java
  - Fortran
  - Julia, MATLAB, OCaml, Python, R
- Implementations for most modern supercomputers
- No tasking, futures/promises, abstract metrics, streams, phasers



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

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



Organization of a Distributed-Memory Multiprocessor

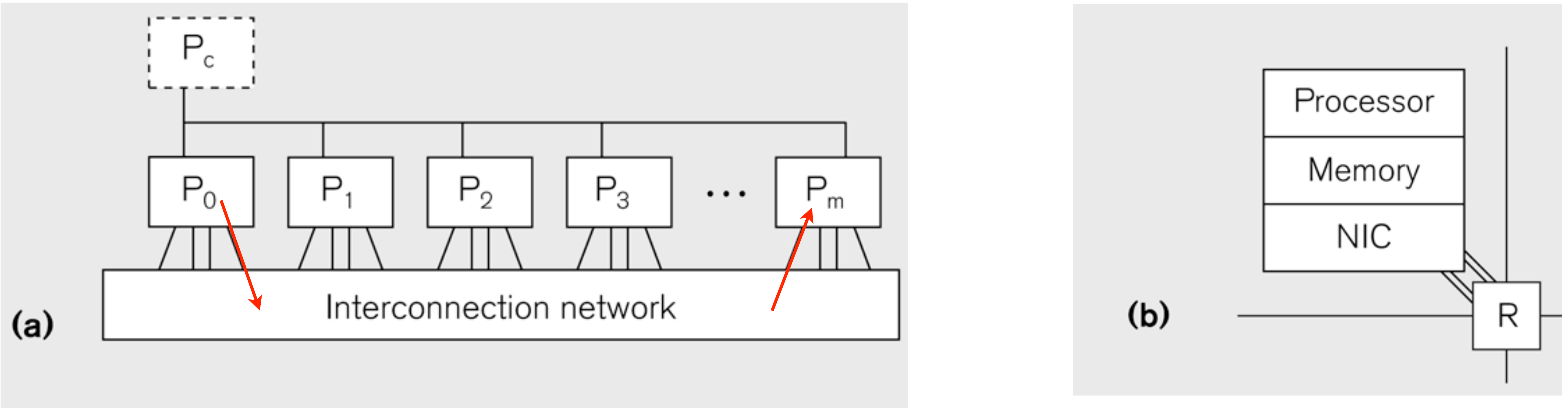
Figure (a)

- 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

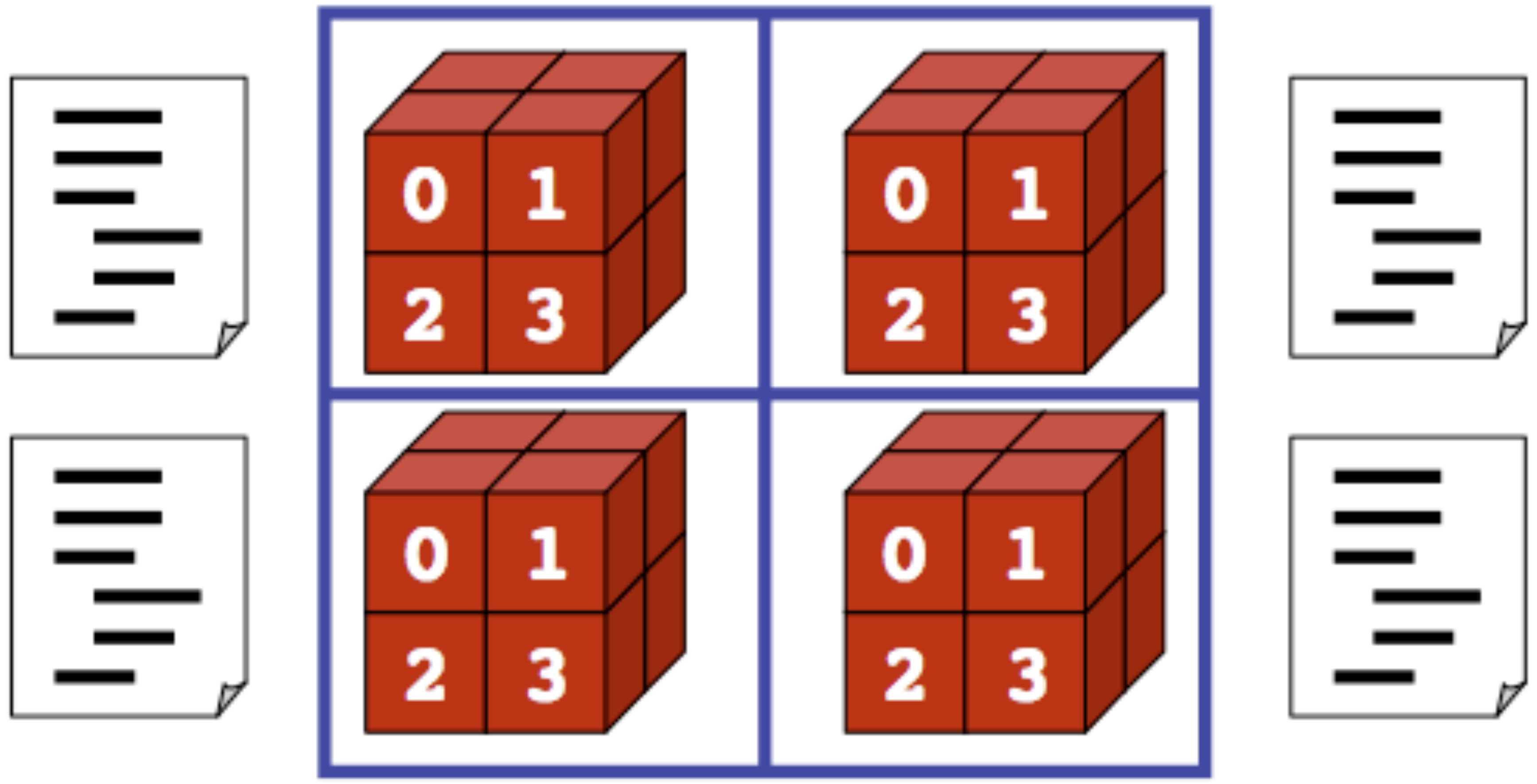
Processors communicate by sending messages via an interconnect



Using Single Program Multiple Data 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

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

```
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
    }
}
```

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 blocking 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

Summary

- Concurrent and parallel programming is becoming pervasive
- Many languages and frameworks support some aspects
- Most of them do not support all aspects of concurrent and parallel programming
- It's possible to build additional features on top of a few basic ones
- You have learned most of the basic concepts in COMP 322