Introduction to Parallel Computing (CMSC416)

Message Passing and MPI

Alan Sussman, Department of Computer Science

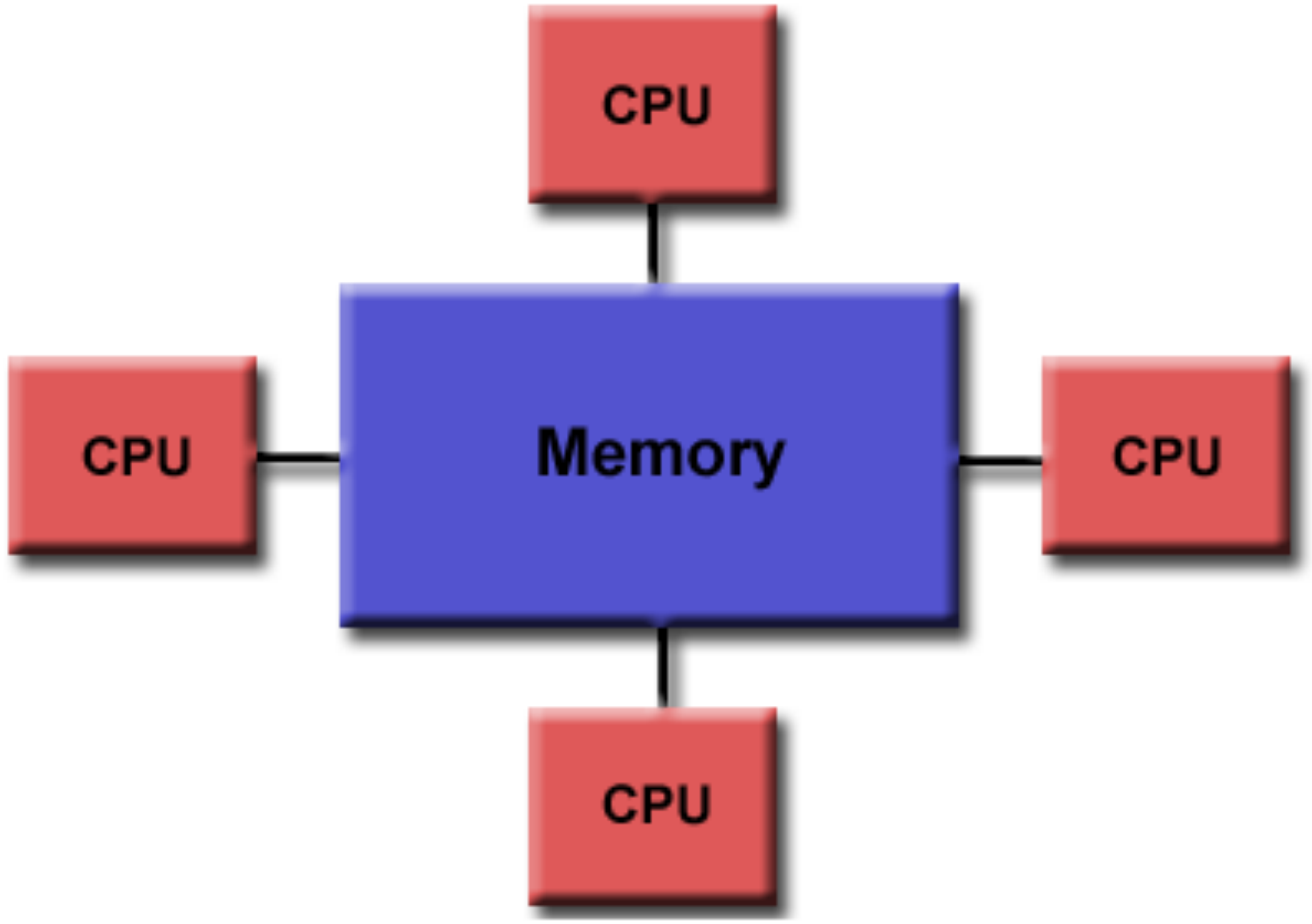
UNIVERSITY OF
MARYLAND

Announcements

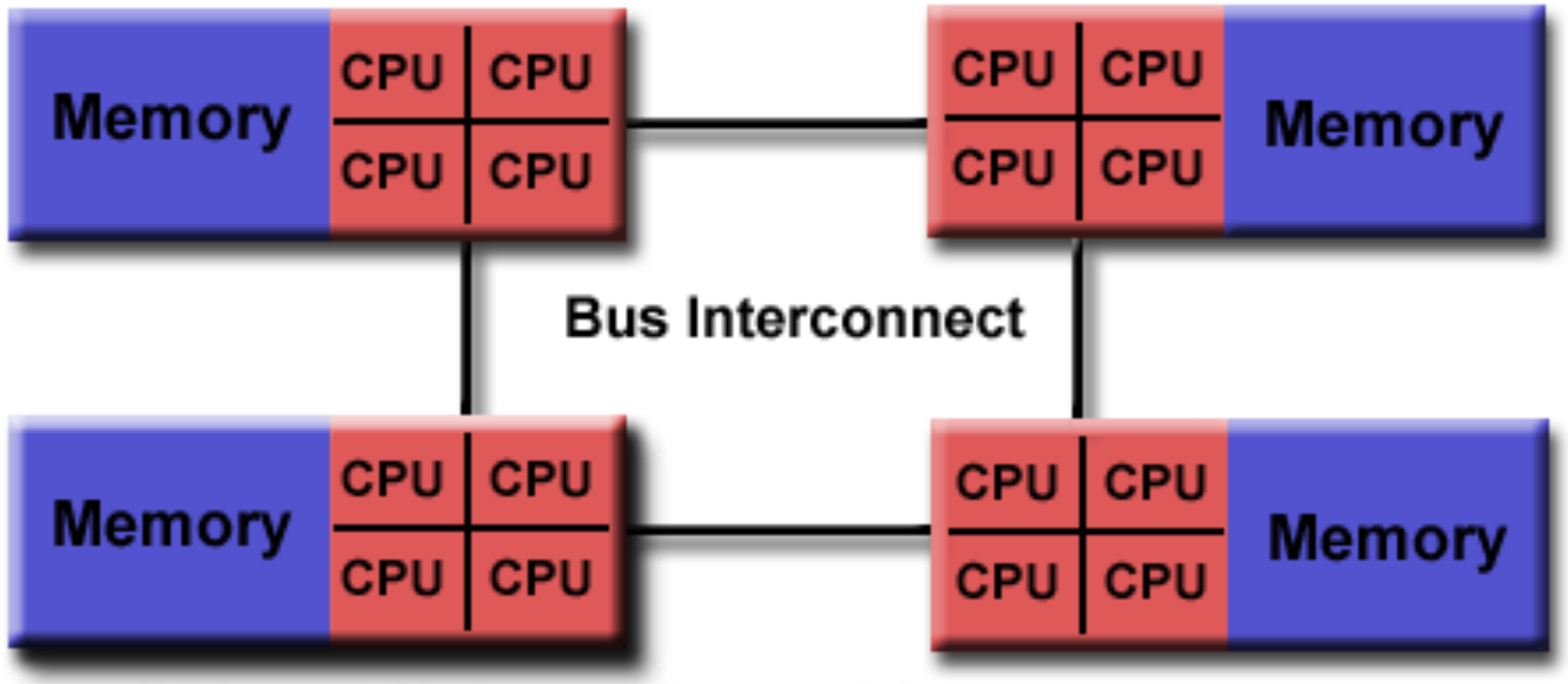
- If you registered for the course recently, please email me for a zaratan account
- Office hours start this week – see web page for times for TA and me
  - Links for Zoom office hours are in an ELMS announcement
- Sign up for Piazza if you have not done so already
  - Link is in ELMS
- Assignment 0 will be posted Thursday, and due a week later
  - Not graded, but you have to submit through gradescope

Shared memory architecture

- All processors/cores can access all memory as a single address space



Uniform Memory Access

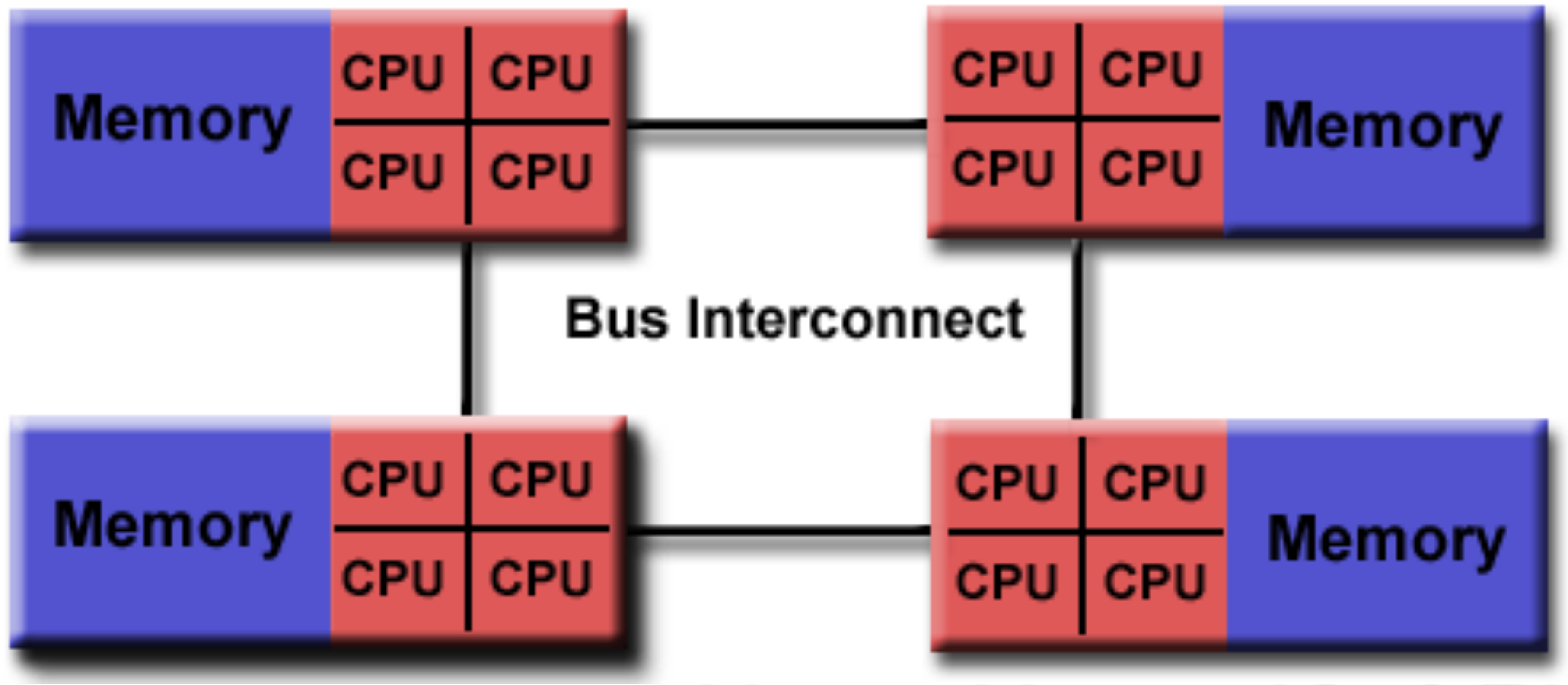


Non-uniform Memory Access (NUMA)

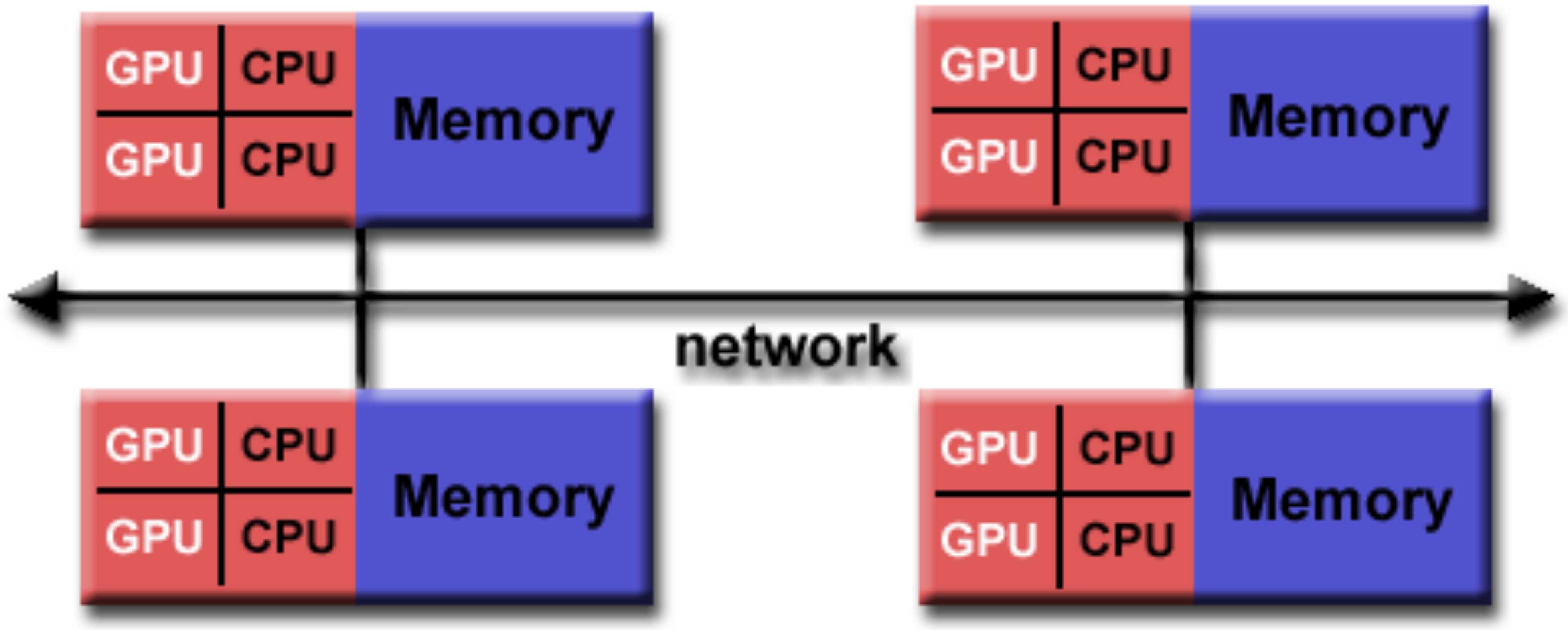
https://computing.llnl.gov/tutorials/parallel_comp/#SharedMemory

Distributed memory architecture

- Each processor/core only has access to its local memory (e.g., on a node, typically)
- Writes in one processor's memory have no effect on another processor's memory



Non-uniform Memory Access (NUMA)



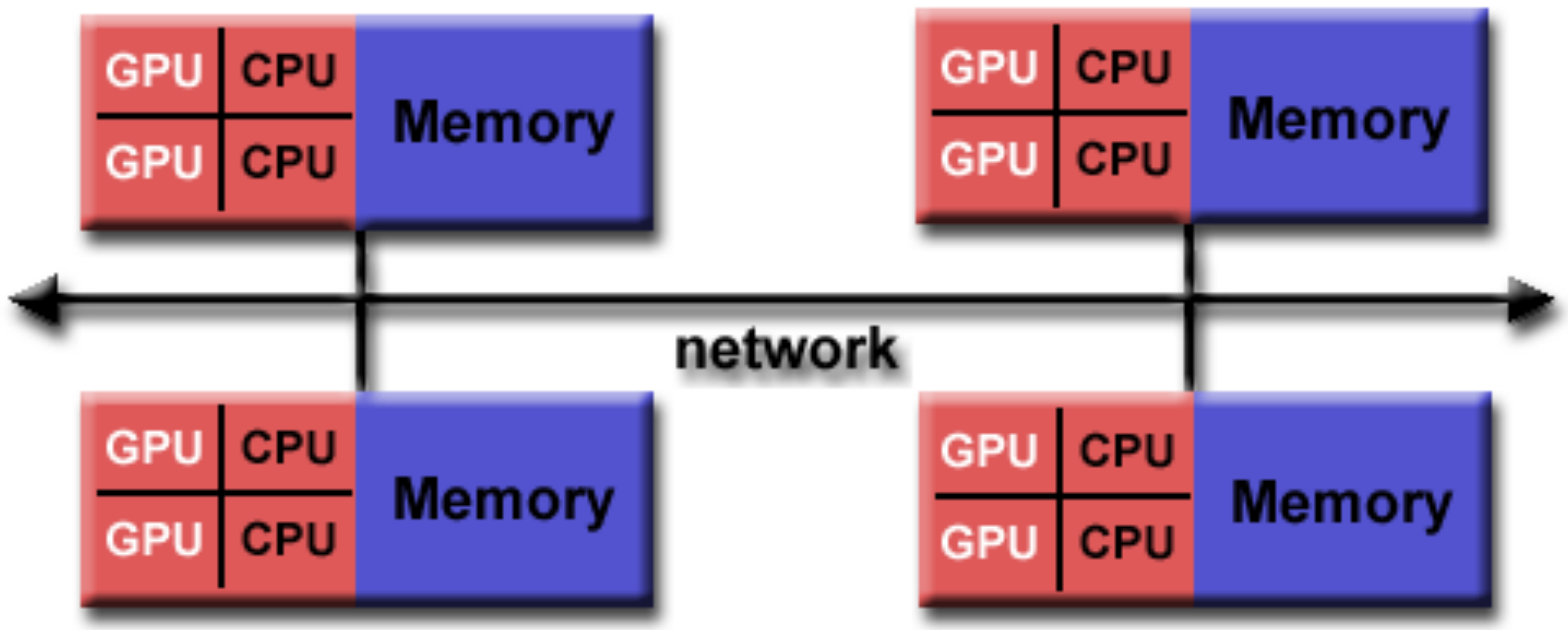
Distributed memory

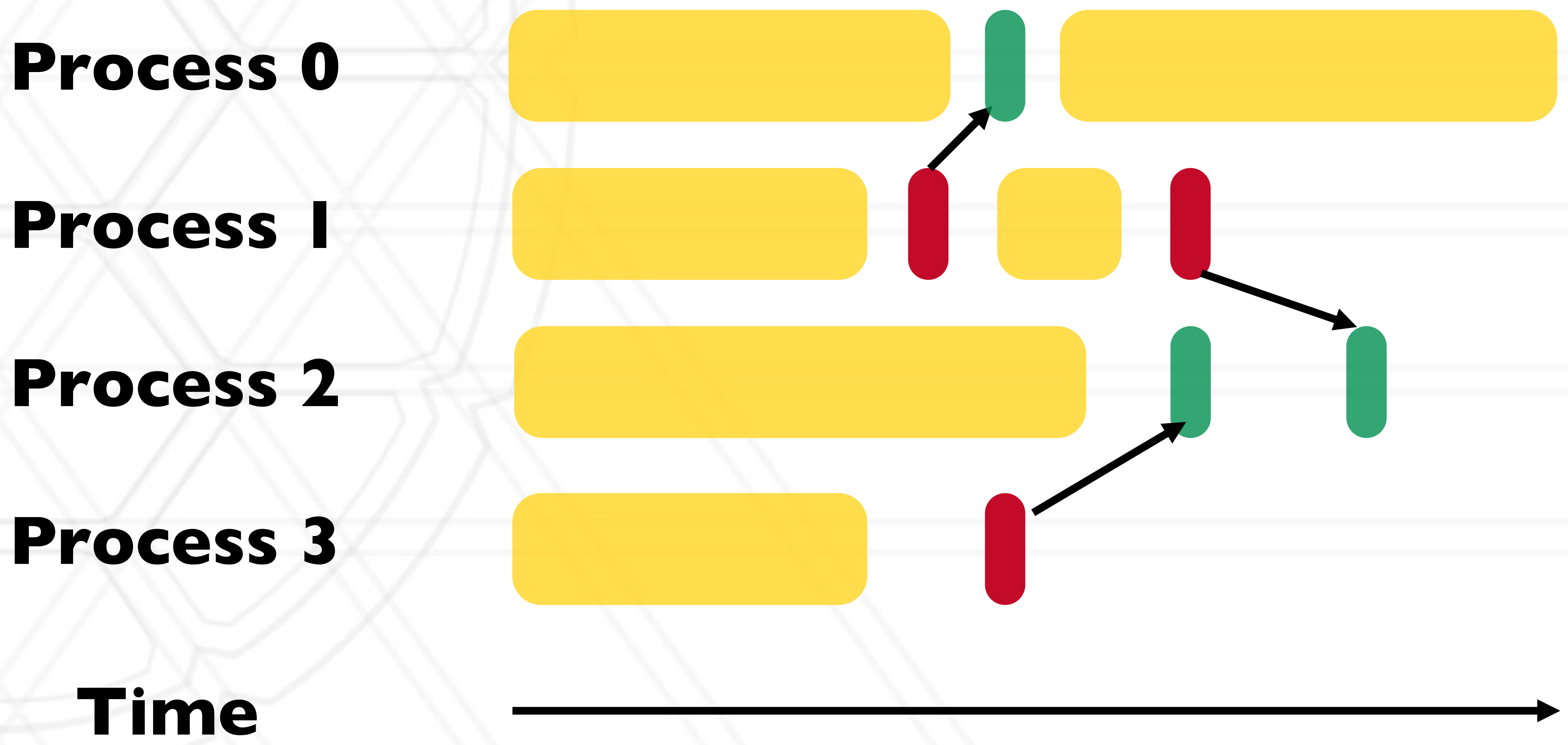
Programming models

- Shared memory model: All threads have access to all of the memory
  - pthreads, OpenMP
- Distributed memory model: Each process has access to its own local memory
  - Also sometimes referred to as message passing
  - MPI, Charm++
- Hybrid models: Use both shared and distributed memory models together
  - MPI+OpenMP, Charm++ (SMP mode)

Distributed memory programming models

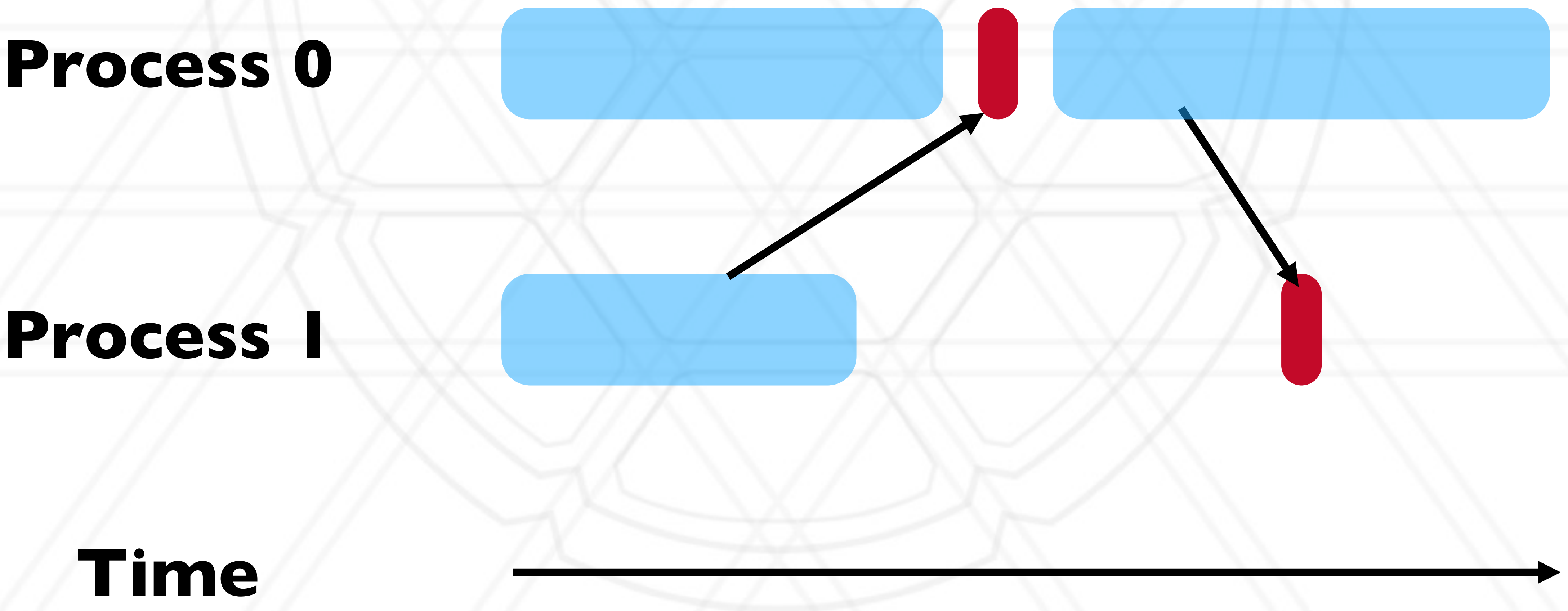
- Each process only has access to its own local memory / address space
- When a process needs data from remote processes, it has to send/receive messages





Message passing

- Each process runs in its own address space
  - Access to only its own memory (no shared data)
- Use special functions to exchange data



Message passing programs

- A parallel message passing program consists of independent processes
  - Processes created by a launch/run script
- (Usually) Each process runs the same executable, but potentially different parts of the program, and on different data
  - Since control flow usually depends on data values
- Often uses *SPMD* style of programming

Message passing history

- PVM (Parallel Virtual Machine) was developed in 1989-1993
- Many vendor libraries in 1980's to mid-90's, all with somewhat different APIs and function semantics
- MPI forum was formed in 1992 to standardize message passing models and MPI 1.0 was released in 1994
    - v2.0 - 1997
    - v3.0 – 2012
    - v4.0 – 2021
    - v5.0 under development

Message Passing Interface (MPI)

- It is an interface standard — defines the operations / functions needed for message passing
- Implemented by vendors and academics/labs for different platforms
  - Meant to be "portable": ability to run the same code on different platforms without modifications
- Some popular (open source) implementations are MPICH, MVAPICH, OpenMPI
  - Several vendors also provide implementations optimized for their products – e.g., Cray/HPE, Intel

Hello world in MPI

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
  int rank, size;
  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Hello world! I'm %d of %d\n", rank, size);

  MPI_Finalize();
  return 0;
}
```

Compiling and running an MPI program with OpenMPI

- Compiling:

```
mpicc -o hello hello.c
```

- Running:

```
mpirun -n 2 ./hello
```

Process creation / destruction

- `int MPI_Init( int argc, char **argv )`
  - Initializes the MPI execution environment
- `int MPI_Finalize( void )`
  - Terminates MPI execution environment

Process identification

- `int MPI_Comm_size( MPI_Comm comm, int *size )`
  - Determines the size of the group associated with a communicator
- `int MPI_Comm_rank( MPI_Comm comm, int *rank )`
  - Determines the rank (ID) of the calling process in the communicator
- Communicator — a set of processes and a system-defined unique tag
  - Default communicator: `MPI_COMM_WORLD`

Send a message

```
int MPI_Send( const void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm )
```

`buf`: address of send buffer

`count`: number of elements in send buffer

`datatype`: datatype of each send buffer element

`dest`: rank of destination process

`tag`: message tag

`comm`: communicator

Receive a message

```
int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int
source, int tag, MPI_Comm comm, MPI_Status *status )
```

`buf`: address of receive buffer

`status`: status object

`count`: maximum number of elements in receive buffer

`datatype`: datatype of each receive buffer element

`source`: rank of source process

`tag`: message tag

`comm`: communicator

Introduction to Parallel Computing (CMSC416)

Message Passing and MPI

Alan Sussman, Department of Computer Science

UNIVERSITY OF
MARYLAND

Announcements

- Again, If you registered for the course recently, please email me for a zaratan account
- Sign up for Piazza if you have not done so already
  - Link is in ELMS
- Assignment 0 is posted, and due a week from today
  - Not graded, but you have to submit through gradescope
- Quiz 0 is posted in ELMS
  - Available to take starting after class today
  - Not graded, but due Monday at 5PM

Semantics of point-to-point communication

- A receive *matches* a send if the arguments to the calls are compatible
    - Same communicator, same tag, datatypes should be the same (otherwise data won’t be interpreted correctly in the receiver)
- If a sender sends two messages to a destination, and both match the same receive, the second message cannot be received if the first is still pending
    - “No-overtaking” messages
    - Always true when processes are single-threaded
    - In other words, two sends from same process to same destination process will arrive in order
- No guarantee of fairness between processes on receive
- Tags (the *tag* field in a send or receive call) can be used to disambiguate between messages in case of non-determinism

Simple send/receive in MPI

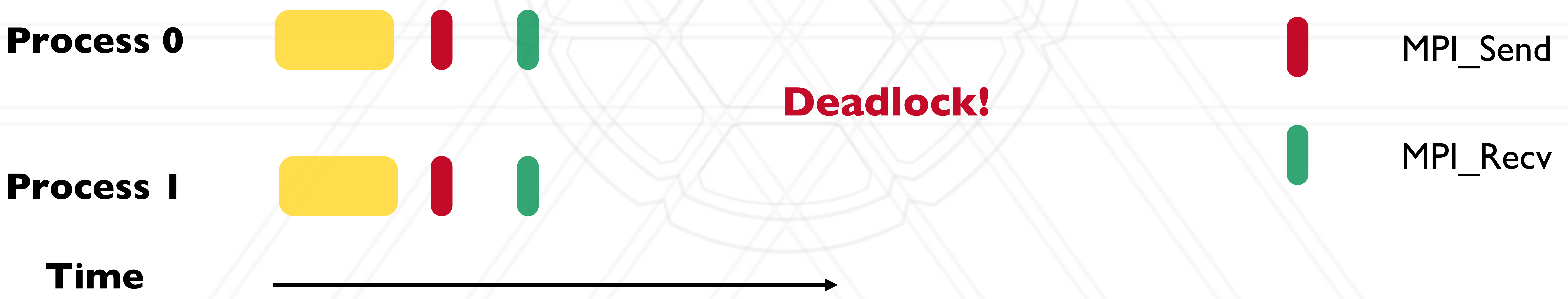
```
int main(int  argc, char *argv) {
  ...
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int data;
  if (rank == 0) {
    data = 7;
    MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  } else if (rank == 1) {
    MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process 1 received data %d from process 0\n", data);
  }

  ...
}
```

Basic MPI_Send and MPI_Recv

- MPI_Send and MPI_Recv routines are blocking
  - Only return when the buffer specified in the call can be (re)used
  - Send: Returns once sender can reuse the buffer
  - Recv: Returns once data from Recv is available in the buffer



Process 0

Process 1

Time

Deadlock!

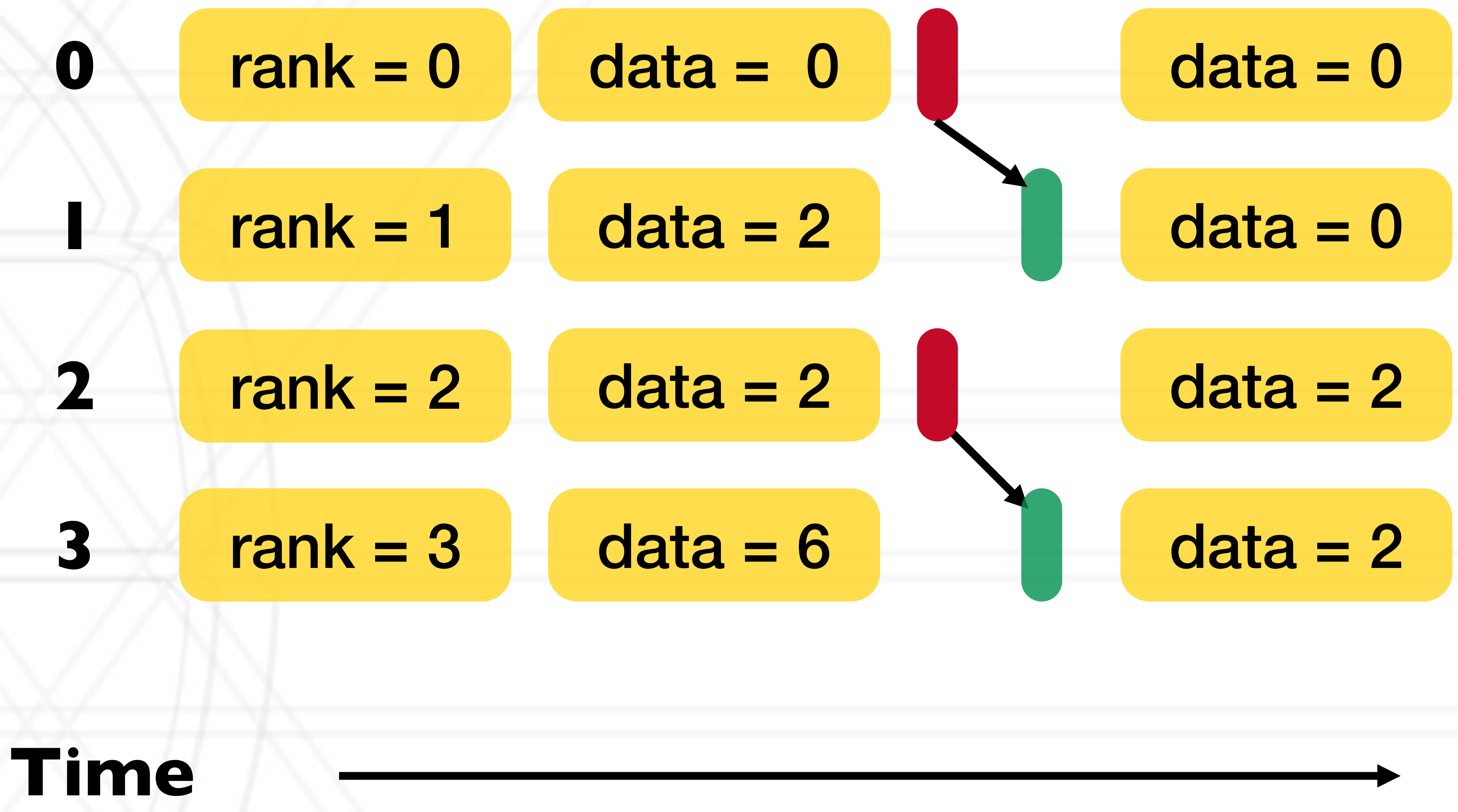
🟥 MPI_Send

🟩 MPI_Recv

Example program

```
int main(int  argc, char *argv) {
  ...
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ...
  if (rank % 2 == 0) {
    data = rank;
    MPI_Send(&data, 1, MPI_INT, rank+1, 0, ...);
  } else {
    data = rank * 2;
    MPI_Recv(&data, 1, MPI_INT, rank-1, 0, ...);

    ...
    printf("Process %d received data %d\n”, data);
  }
  ...
}
```



0 rank = 0 data = 0 data = 0

1 rank = 1 data = 2 data = 0

2 rank = 2 data = 2 data = 2

3 rank = 3 data = 6 data = 2

Time

MPI communicators

- Communicator represents a group or set of processes numbered 0, … , n-1, along with a unique system-defined tag
- Every program starts with MPI_COMM_WORLD (default communicator)
  - Defined by the MPI runtime, this group includes all processes
- Several MPI routines to create new communicators
  - MPI_Comm_split
  - MPI_Cart_create
  - MPI_Group_incl

MPI datatypes

- Can be a pre-defined one: MPI_INT, MPI_CHAR, MPI_DOUBLE, …
- Derived or user-defined datatypes:
  - Array of elements of another datatype
  - struct data type to accomodate sending multiple datatypes

UNIVERSITY OF
MARYLAND