



Message Passing and MPI

Abhinav Bhatele, Alan Sussman



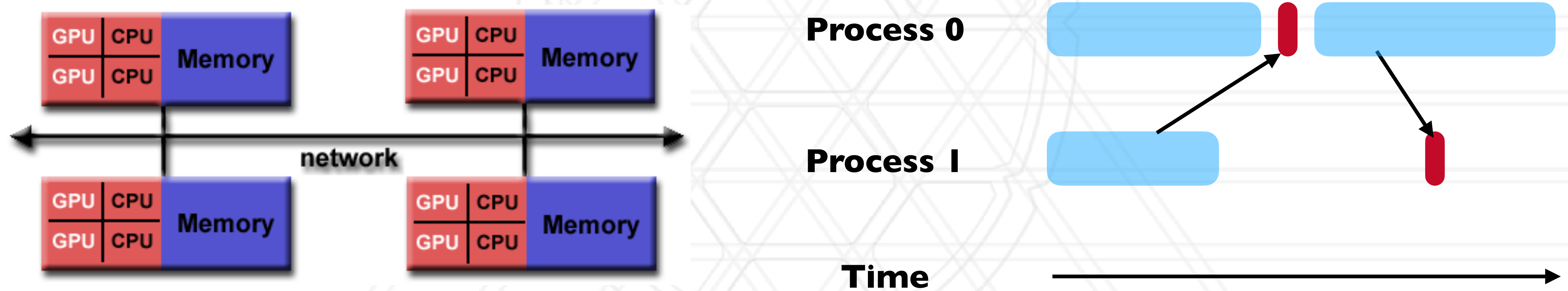
UNIVERSITY OF
MARYLAND

Announcements

- Assignment 1 is posted, due on Sep 18 11:59 pm
 - Resource for OpenMP: <https://computing.llnl.gov/tutorials/openMP>
- Assignment 0.2 is also posted but not due until Sep 24 11:59 pm
 - If you have questions about this assignment, hold off working on it until the topic is covered in class
- Resources for learning MPI:
 - <https://mpitutorial.com>
 - <https://rookiehpc.org>

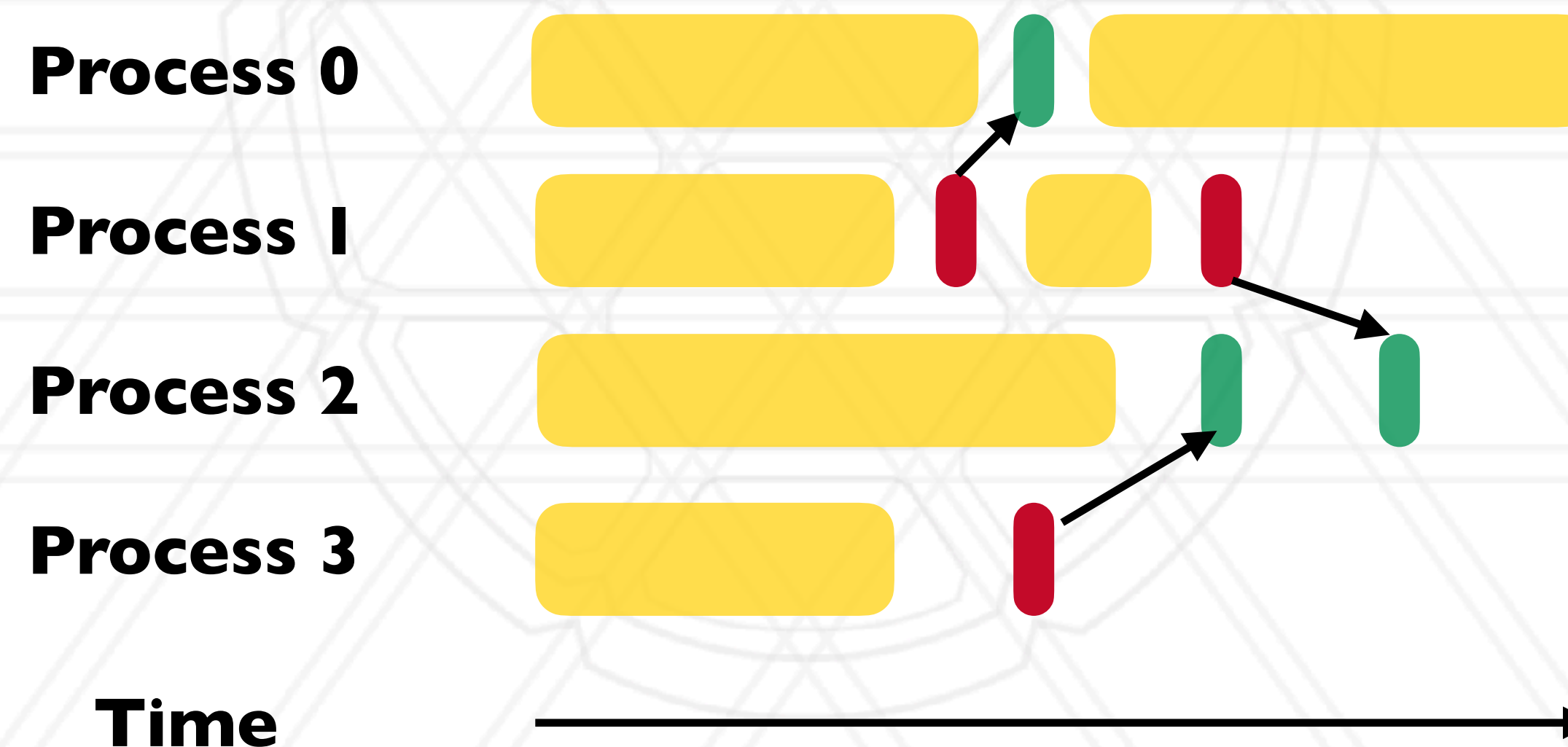
Distributed memory programming models

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



Message passing

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



Message passing programs

- A parallel message passing program consists of independent processes
 - Processes created by a launch/run script
- Each process runs the same executable, but potentially different parts of the program, and on different data
- Often used for SPMD style of programming

Message passing history

- PVM (Parallel Virtual Machine) was developed in 1989-1993
- 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

Message Passing Interface (MPI)

- It is an interface standard — defines the operations / routines needed for message passing
- Implemented by vendors and academics 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
 - Vendors often implement their own versions optimized for their hardware: Cray/HPE, Intel

Hello world in MPI

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

int main(int argc, char *argv[]) {
    int myrank, numpes;
    MPI_Init(&argc, &argv);

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

    MPI_Finalize();
    return 0;
}
```


Compiling and running an MPI program

- 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 the 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 identified by a unique tag
 - Default communicator: `MPI_COMM_WORLD`

Announcements

- Assignment 0.2 is posted and due on Sep 24 11:59 pm
- Assignment 1 autograder changes:
 - We tweaked the autograder a bit so that it does not report scores out of 90
 - Reminder: your solutions will be run by us on zaratan to verify correctness
- Final exam date and time: Dec 11 6:30-8:30 pm
 - In the respective classrooms: IRB 0318 and 1116

Send a blocking pt2pt 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

Send a blocking pt2pt 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



Between a pair
of processes

Receive a blocking pt2pt 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

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

status: status object

MPI_Status object

- Represents the status of the received message
- count: number of received entries
- MPI_SOURCE: source of the message
- MPI_TAG: tag value of the message
- MPI_ERROR: error associated with the message

```
typedef struct _MPI_Status {  
    int count;  
    int cancelled;  
    int MPI_SOURCE;  
    int MPI_TAG;  
    int MPI_ERROR;  
} MPI_Status, *PMPI_Status;
```


Semantics of point-to-point communication

- A receive *matches* a send if certain arguments to the calls match
 - What is matched: source, tag, communicator
 - If the datatypes and count don't match, this could lead to memory errors and correctness issues
- 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
- Tags can be used to disambiguate between messages in case of non-determinism

Semantics of point-to-point communication

- A receive *matches* a send if certain arguments to the calls match
 - What is matched: source, tag, communicator
 - If the datatypes and count don't match, this could lead to memory errors and correctness issues
- 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
- Tags can be used to disambiguate between messages in case of non-determinism

Between a pair of processes

Simple send/receive in MPI

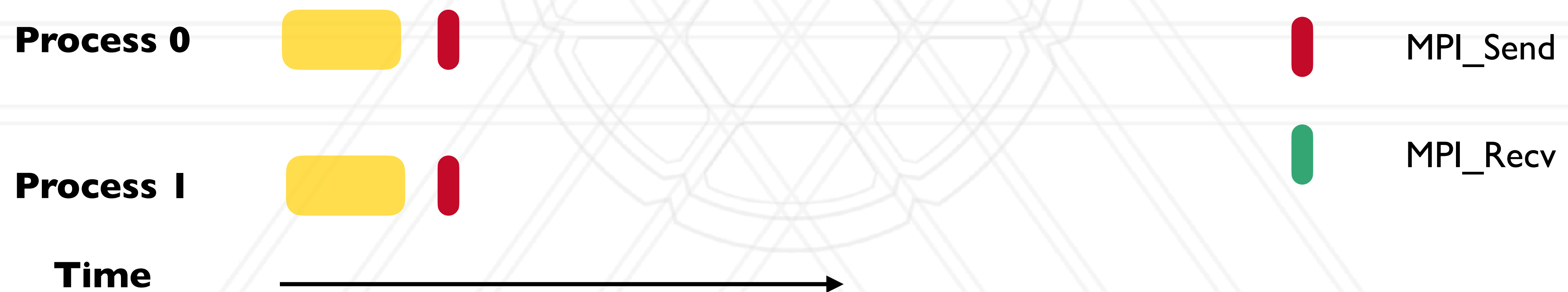
```
int main(int argc, char *argv[]) {
    ...
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    int data;
    if (myrank == 0) {
        data = 7;
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (myrank == 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);
    }

    ...
}
```

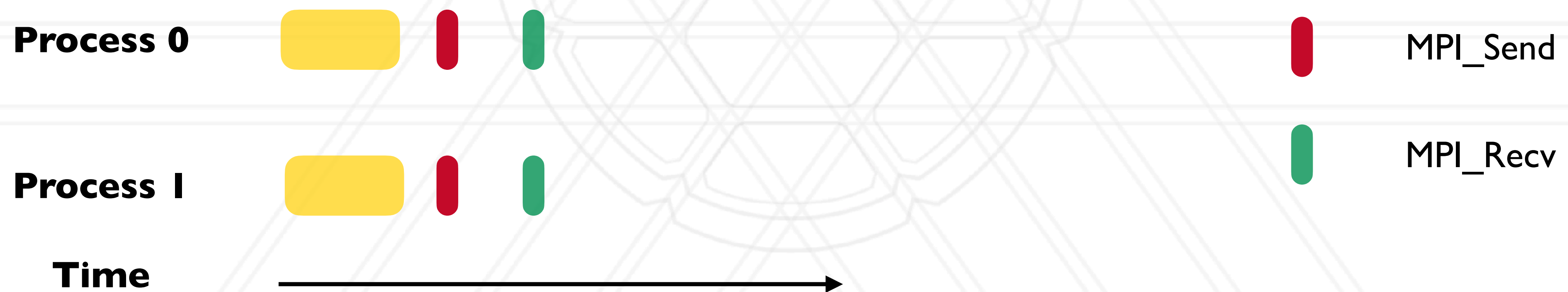
Basic MPI_Send and MPI_Recv

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



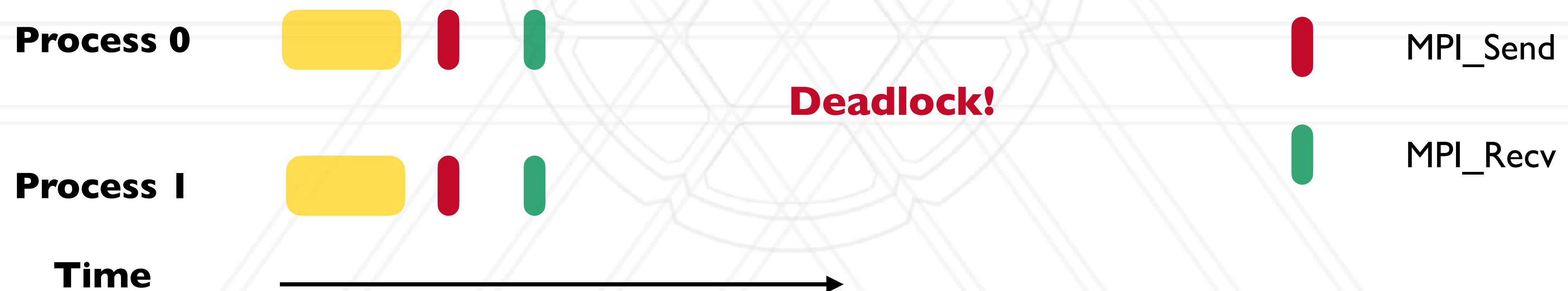
Basic MPI_Send and MPI_Recv

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



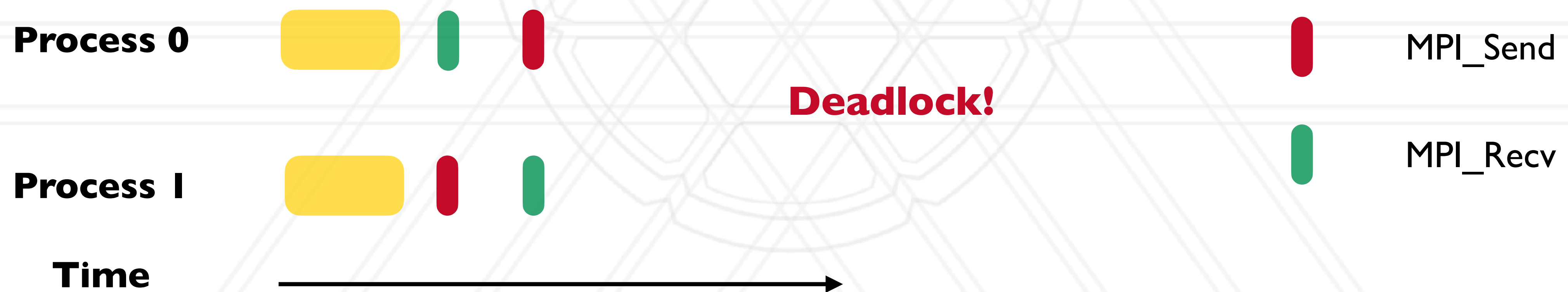
Basic MPI_Send and MPI_Recv

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



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 used again
 - Send: Returns once sender can reuse the buffer
 - Recv: Returns once data from Recv is available in the buffer



Example program

```
int main(int argc, char *argv[]) {  
    ...  
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
    ...  
    if (myrank % 2 == 0) {  
        data = myrank;  
        MPI_Send(&data, 1, MPI_INT, myrank+1, 0, ...);  
    } else {  
        data = myrank * 2;  
        MPI_Recv(&data, 1, MPI_INT, myrank-1, 0, ...);  
    }  
    ...  
    printf("Process %d received data %d\n", myrank, data);  
}  
    ...  
}
```

0 rank = 0

1 rank = 1

2 rank = 2

3 rank = 3

Time



Example program

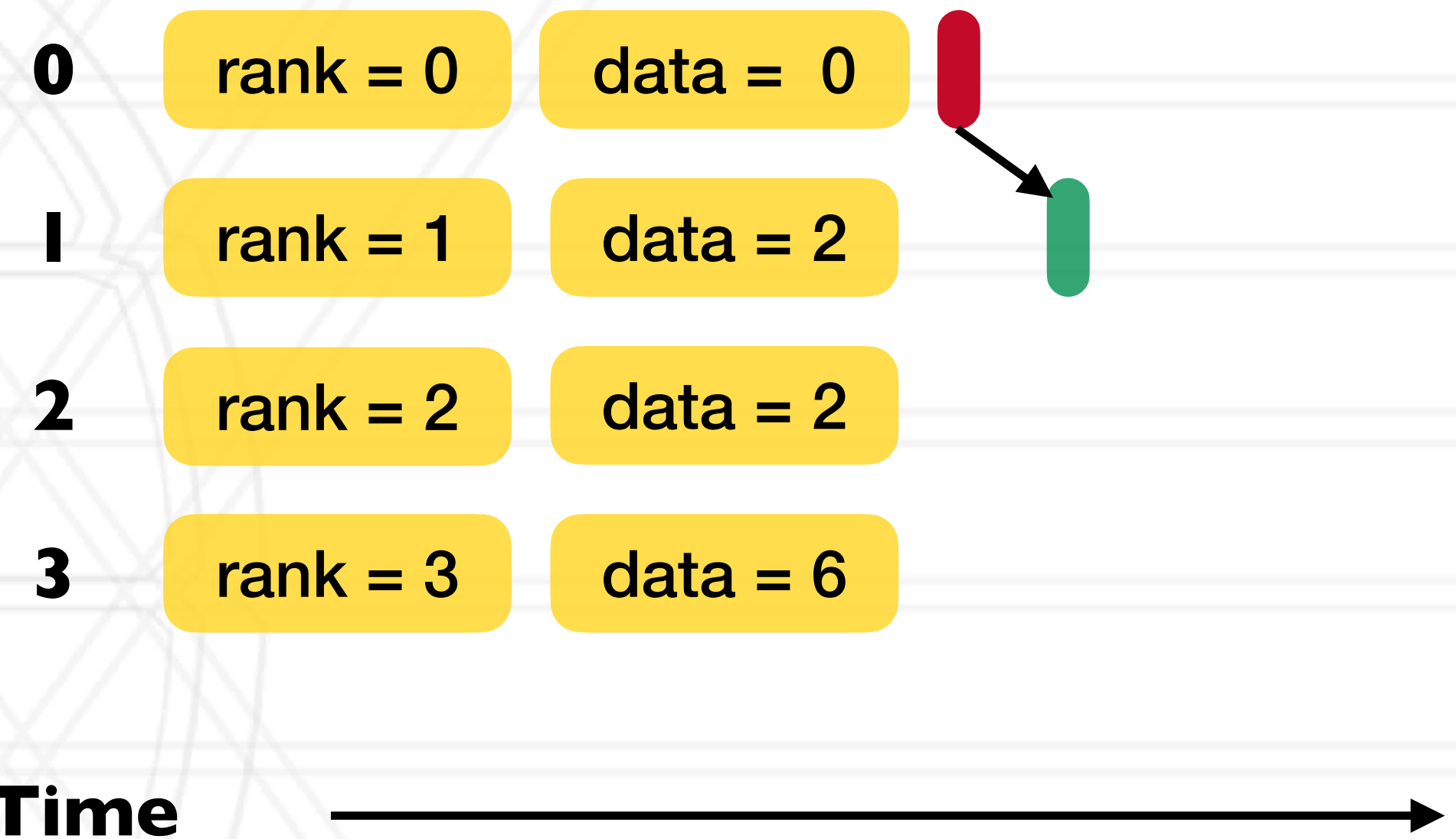
```
int main(int argc, char *argv[]) {  
    ...  
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
    ...  
    if (myrank % 2 == 0) {  
        data = myrank;  
        MPI_Send(&data, 1, MPI_INT, myrank+1, 0, ...);  
    } else {  
        data = myrank * 2;  
        MPI_Recv(&data, 1, MPI_INT, myrank-1, 0, ...);  
  
        ...  
        printf("Process %d received data %d\n", myrank, data);  
    }  
    ...  
}
```

| | | |
|----------|----------|----------|
| 0 | rank = 0 | data = 0 |
| 1 | rank = 1 | data = 2 |
| 2 | rank = 2 | data = 2 |
| 3 | rank = 3 | data = 6 |

Time →

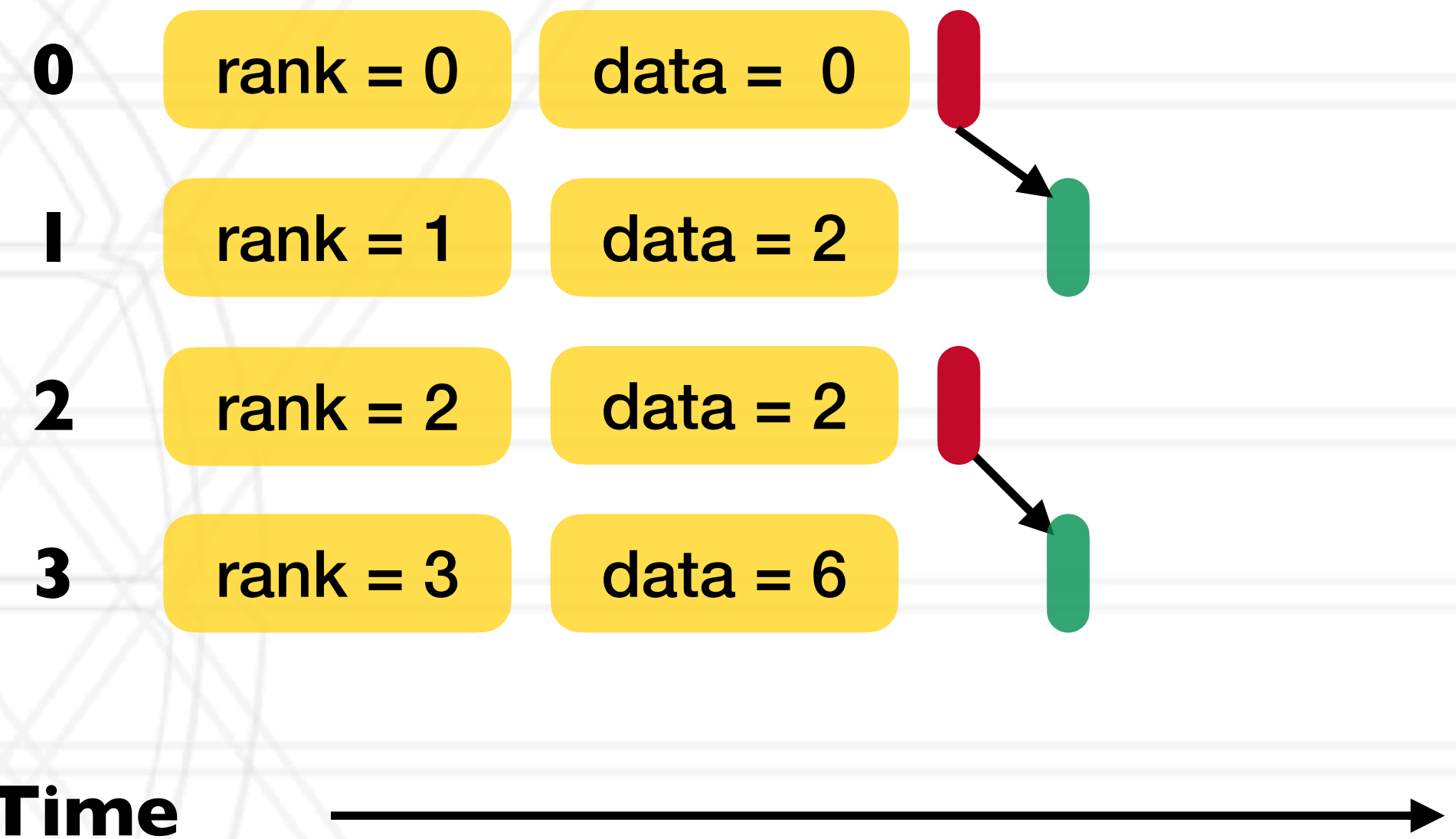
Example program

```
int main(int argc, char *argv[]) {  
    ...  
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
    ...  
    if (myrank % 2 == 0) {  
        data = myrank;  
        MPI_Send(&data, 1, MPI_INT, myrank+1, 0, ...);  
    } else {  
        data = myrank * 2;  
        MPI_Recv(&data, 1, MPI_INT, myrank-1, 0, ...);  
    }  
    ...  
    printf("Process %d received data %d\n", myrank, data);  
}  
    ...  
}
```



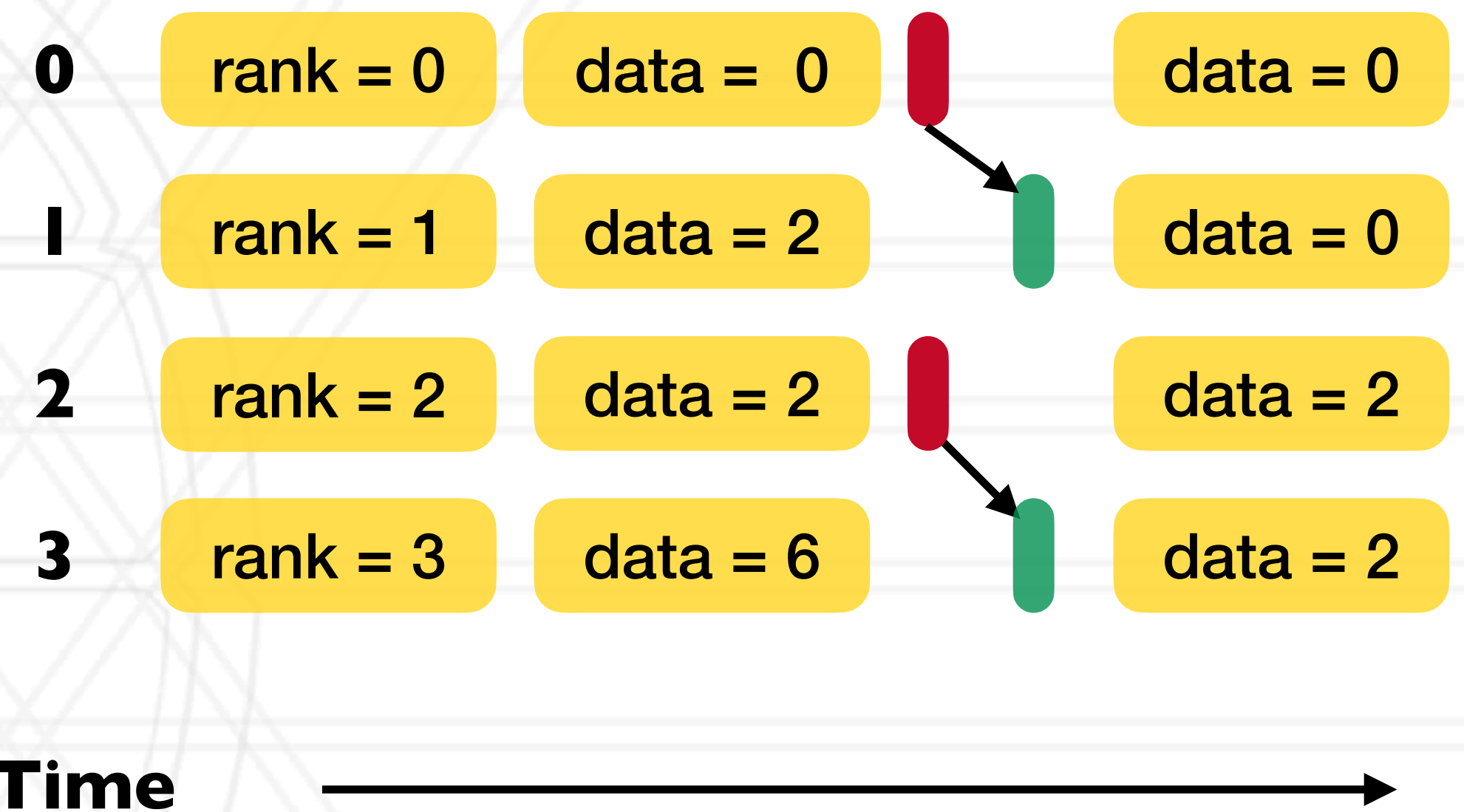
Example program

```
int main(int argc, char *argv[]) {  
    ...  
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
    ...  
    if (myrank % 2 == 0) {  
        data = myrank;  
        MPI_Send(&data, 1, MPI_INT, myrank+1, 0, ...);  
    } else {  
        data = myrank * 2;  
        MPI_Recv(&data, 1, MPI_INT, myrank-1, 0, ...);  
        ...  
        printf("Process %d received data %d\n", myrank, data);  
    }  
    ...  
}
```



Example program

```
int main(int argc, char *argv[]) {  
    ...  
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
    ...  
    if (myrank % 2 == 0) {  
        data = myrank;  
        MPI_Send(&data, 1, MPI_INT, myrank+1, 0, ...);  
    } else {  
        data = myrank * 2;  
        MPI_Recv(&data, 1, MPI_INT, myrank-1, 0, ...);  
    }  
    ...  
    printf("Process %d received data %d\n", myrank, data);  
}  
    ...  
}
```



MPI communicators

- Communicator represents a group or set of processes numbered 0, ... , n-1
 - Identified by a unique “tag” assigned by the runtime
- Every program starts with `MPI_COMM_WORLD` (default communicator)
 - Defined by the MPI runtime, this group includes all processes
- Several MPI routines to create sub-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 datatype to accommodate sending multiple datatypes together



UNIVERSITY OF
MARYLAND