

MPI – The Message Passing Interface

Principles of Message-Passing Systems

Intro to MPI

Send-Receive; Blocking-NonBlocking

Deadlocks

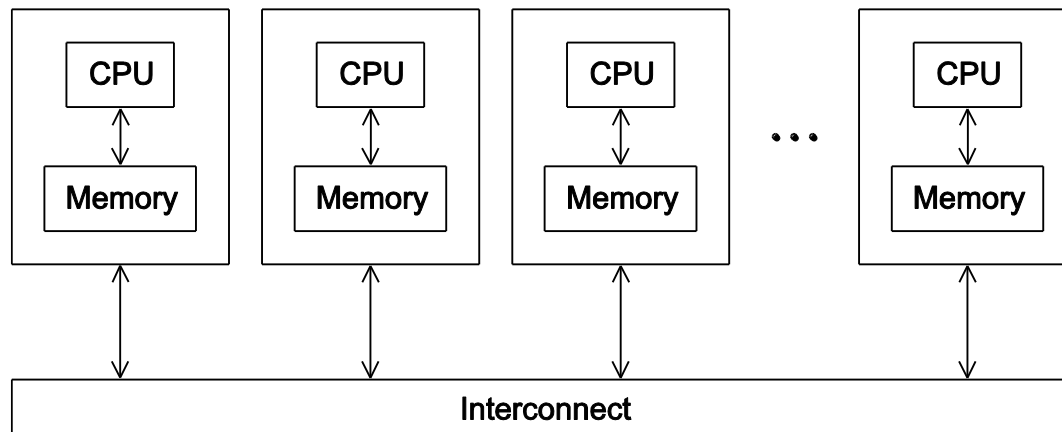
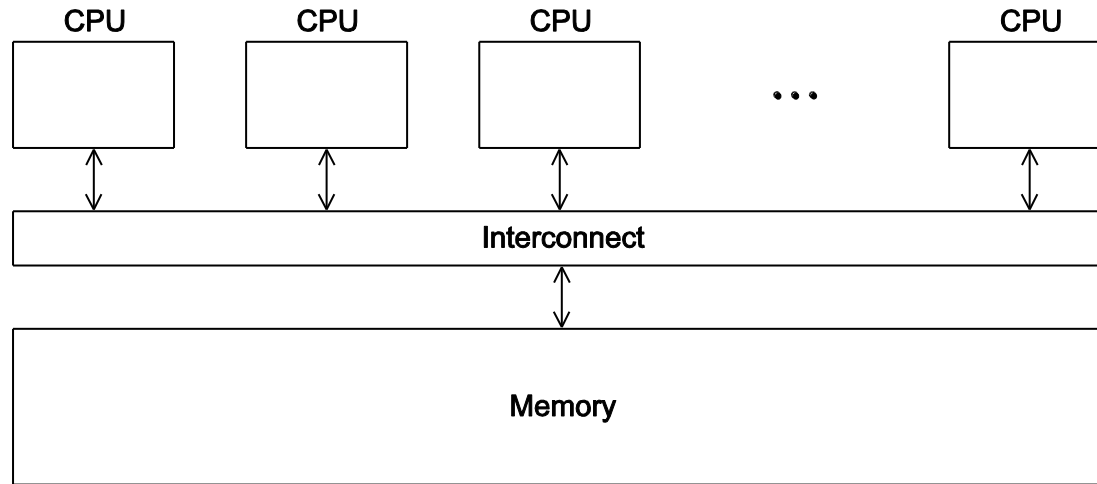
First Parallel Program with MPI

Synchronization Points - Barrier

Bibliography

- [Pacheco]: Peter Pacheco, Matthew Malensek, *Introduction to Parallel Programming*, 2nd Edition, Morgan Kaufmann Publisher, March 2020, Chapter 3
- [GGKK] Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, Chapter 4, Chapter 6
- <https://hpc-tutorials.llnl.gov/mpi/>

Shared mem vs Distributed mem

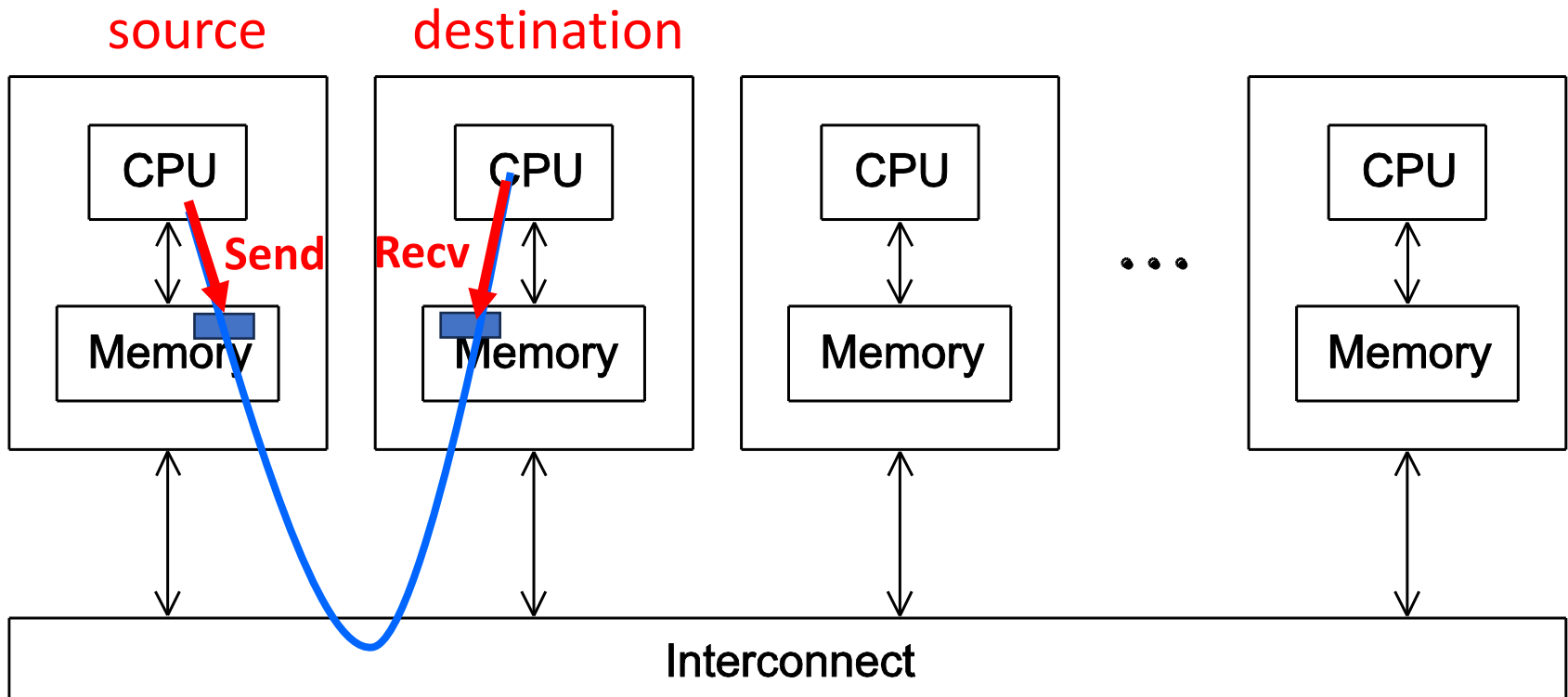


Parallel Programming for Distributed Memory Systems

- Distributed Memory Systems are programmed using the **Message-Passing Paradigm**:
 - The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.
 - Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
 - All interactions to access data (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. *Data access has explicit costs!*

The Building Blocks: Send and Receive Operations

send(sendbuf, nelems, destination)
receive(recvbuf, nelems, source)



Possible Protocols for Send and Receive

- **Blocking Communication:** processes *halt their execution* until the communication operation is complete or in a phase where it is semantically safe for the initiator to continue.
- **Non-blocking (asynchronous) communication:** processes continue their execution *immediately after initiating* communication operations, without waiting for the operations to complete.

Blocking Send and Receive

- **Blocking Communication:** processes halt their execution until the communication operation is complete or ***“it is safe to return”***.
 - The **sender waits** (no return from the Send function) ***until “it is safe”***: *it can guarantee that the semantics will not be violated on return*
 - Message can be kept in internal buffers until it is later delivered (**Buffered Blocking**)
 - Sender waits until receiver is ready (handshake) (**Non Buffered Blocking**)
 - The **receiver waits** (no return from the Receive function) ***until data is received***
 - ***Adv: Operations are easy to be used***
 - ***Disadv: Can cause deadlocks if used wrong***

Non-Blocking Send and Receive

- **Non-blocking communication:** processes continue their execution *immediately after initiating* communication operations, without waiting for the operations to complete.
 - The Send/Recv functions ***return before it is semantically safe to do so***
 - The program must test whether operation is finished
 - After returning from a non-blocking send/recv op., the process can perform any operation that does not depend upon the completion of the communication
 - Later in the program, the process can ***check whether or not the non blocking communication is finished, and, if necessary, wait for its completion***
- ***Adv: Do not cause deadlocks; Can be used to overlap communication with computation***
- ***Disadv: More complicated to use; Can be unsafe if used wrong***

Blocking vs Non-Blocking

- Blocking Send:

```
A=1;
```

```
Send(A, theDest);
```

```
A =7;    // It is safe to modify A!
```

- Non-blocking Send:

```
A=1;
```

```
Send(A, theDest);
```

```
A =7;    // It is NOT yet safe to modify A!
```

```
...
```

```
If (TestSendCompleted)
```

```
    A=7;    // Now it is safe to modify A
```

Blocking vs Non-Blocking

- Blocking Receive:

```
X=0;
```

```
Receive(X, fromSource);
```

```
Print(X); // safe to use NEW received value
```

- Non-blocking Receive:

```
X=0;
```

```
Receive(X, fromSource);
```

```
Print(X); // NOT safe to use value!
```

```
If (TestReceiveCompleted)
```

```
    Print(X); // Now it is safe to use
```

Possible Protocols for Send and Receive

	Blocking Operations	Non-Blocking Operations
Buffered	<p>Sending process returns after data has been copied into communication buffer</p>	<p>Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return</p>
Non-Buffered	<p>Sending process blocks until matching receive operation has been encountered</p> <p>Send and Receive semantics assured by corresponding operation</p>	<p>Programmer must explicitly ensure semantics by polling to verify completion</p>

Intro to MPI- The Message Passing Interface

MPI Overview

- MPI defines a **standard library for message-passing** that can be used to develop **portable** message-passing programs using either C, C++ or Fortran.
- Other (unofficial) language bindings: Java, Ocaml, Python
- The **MPI standard** defines both the **syntax** as well as the **semantics** of a core set of library routines.
- The standardization body for MPI: The MPI Forum:
<https://www.mpi-forum.org/>
- Different Implementations are available:
 - In lab: MS MPI
 - [OpenMPI](#)
 - [MPICH](#)

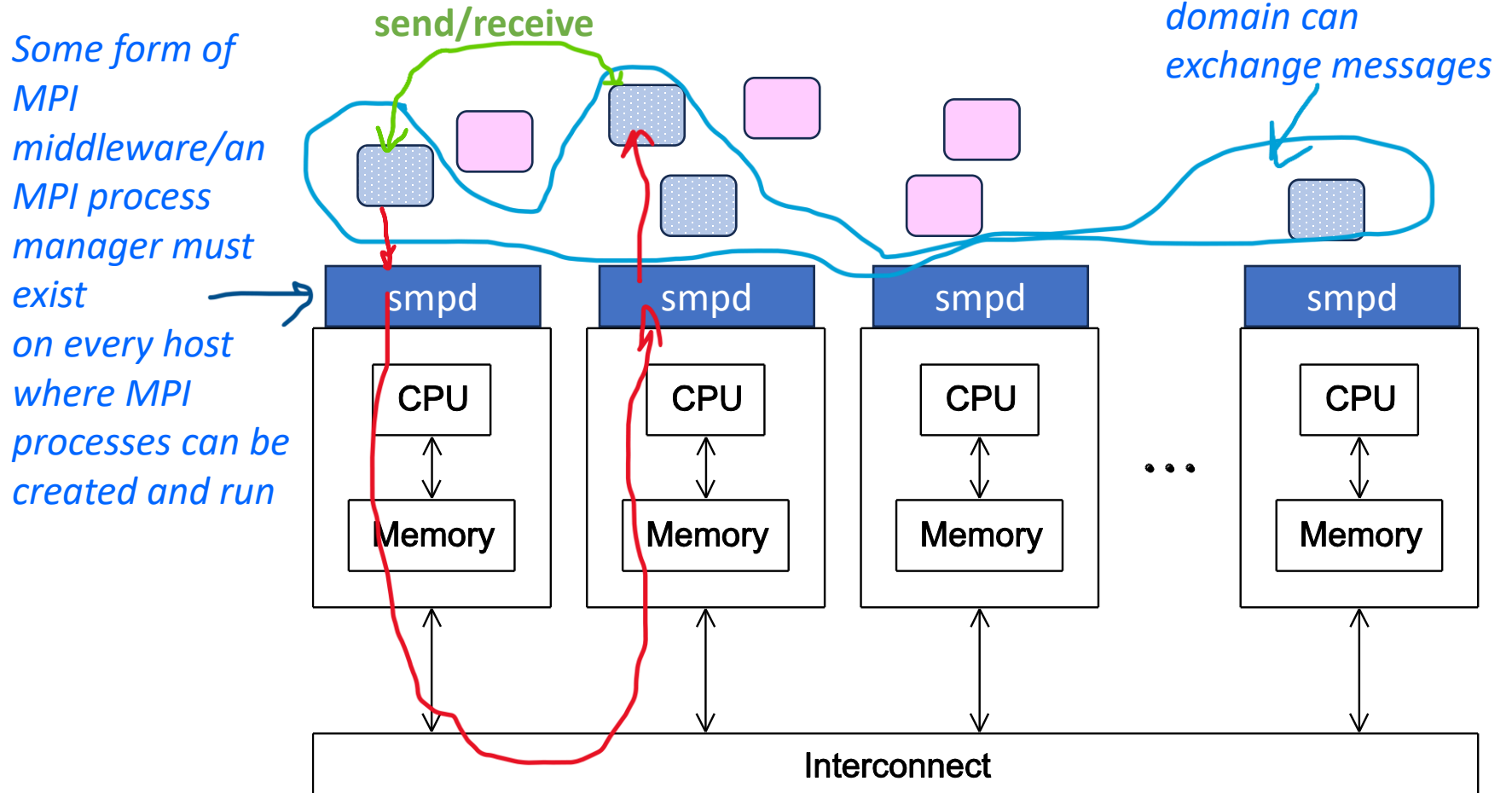
MPI: the Message Passing Interface

- MPI *defines an API*
 - Defines a set of operations(functions) for message passing
 - Standardizes function names
 - Each function takes fixed arguments
 - Each function has fixed semantics
 - Standardizes what the MPI implementation provides and what the application can and cannot expect
 - Each system can implement it differently as long as the semantics match
- MPI **is not:**
 - a language or compiler specification
 - a specific implementation or product

MPI: the Message Passing Interface

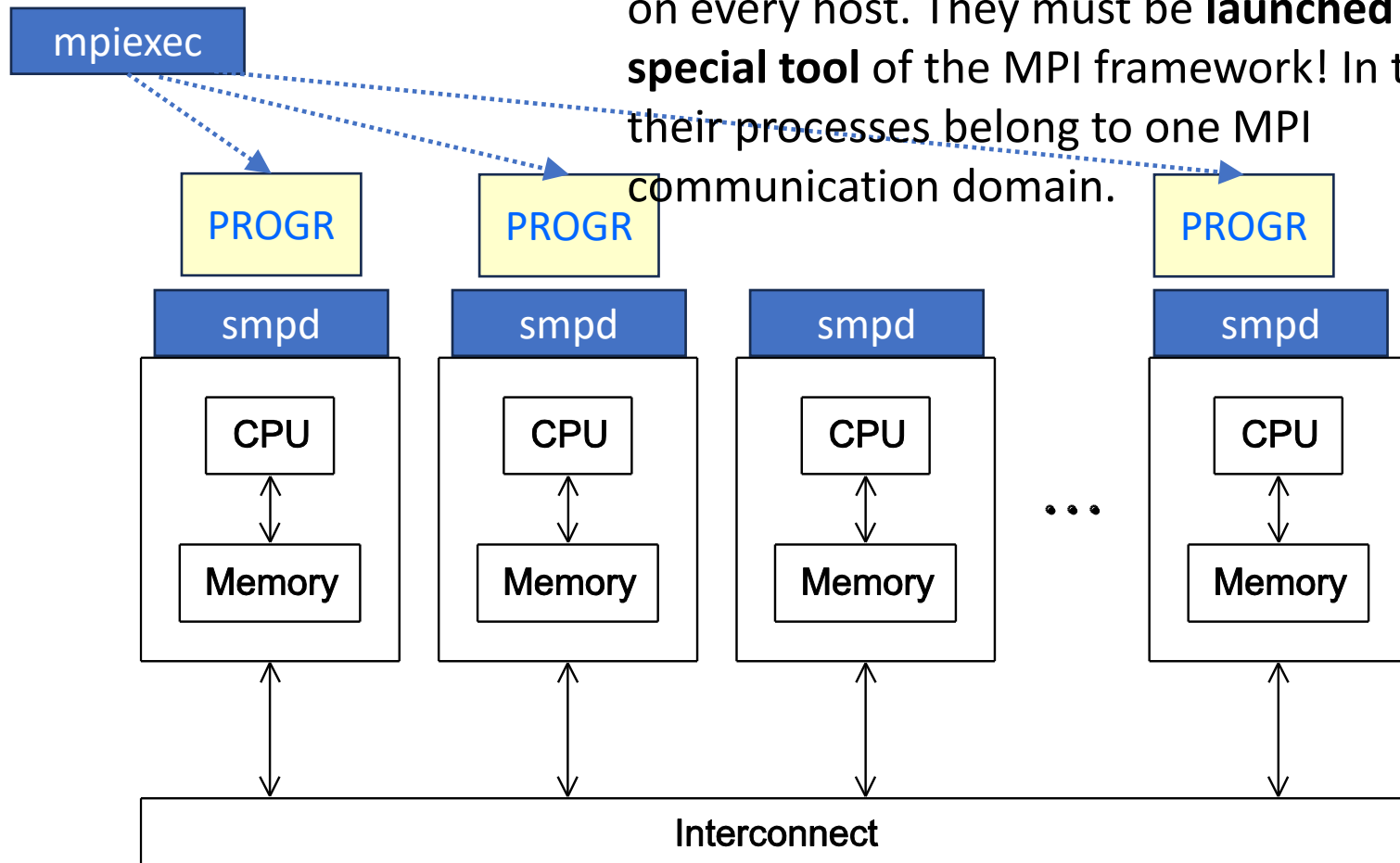
- MPI Implementations however are more than a library: they come also with some runtime support (an MPI process manager) that helps managing the MPI processes
- Important note: in order to use MPI messaging, the processes must belong to a special group of processes that are created and managed by the MPI runtime support (the MPI process manager).
- NOT any process randomly created by the operating system can use MPI messaging, only the processes that belong to the same *MPI communication domain (MPI communicator)* !

MPI Infrastructure



SPMD (Single Program Multiple Data)

Several **copies of a program** are launched on several hosts. A copy of the program must exist on every host. They must be **launched using a special tool** of the MPI framework! In this way their processes belong to one MPI communication domain.



Communicators

- A *communicator* defines a *communication domain* - a set of processes that are allowed to communicate with each other by MPI messages
- Information about communication domains is stored in variables of type **MPI_Comm**.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different (possibly overlapping) communication domains.
- MPI defines a default communicator called **MPI_COMM_WORLD** which includes all the processes launched together

MPI: the Message Passing Interface

The minimal set of MPI routines.

<code>MPI_Init</code>	Initializes MPI.
<code>MPI_Finalize</code>	Terminates MPI.
<code>MPI_Comm_size</code>	Determines the number of processes.
<code>MPI_Comm_rank</code>	Determines the label of calling process.
<code>MPI_Send</code>	Sends a message.
<code>MPI_Recv</code>	Receives a message.

It is possible to write fully-functional message-passing programs by using only six basic MPI routines.

Starting and Terminating the MPI Library

- `MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.
- `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.
- The prototypes of these two functions are:

```
int MPI_Init(int *argc, char ***argv)
```

```
int MPI_Finalize()
```

- `MPI_Init` also strips off any MPI related command-line arguments.
- All MPI routines, data-types, and constants are prefixed by “MPI_”. The return code for successful completion is `MPI_SUCCESS`.

Querying Information

- The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the rank of the calling process, respectively.

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```


- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

Our First MPI Program

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define MASTER 0

int main(int argc, char *argv[])
{
    int numtasks, taskid, len;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Get_processor_name(hostname, &len);
    printf("Hello from task %d on %s!\n", taskid, hostname);
    if (taskid == MASTER)
        printf("MASTER: Number of MPI tasks is: %d\n", numtasks);
    MPI_Finalize();
}
```



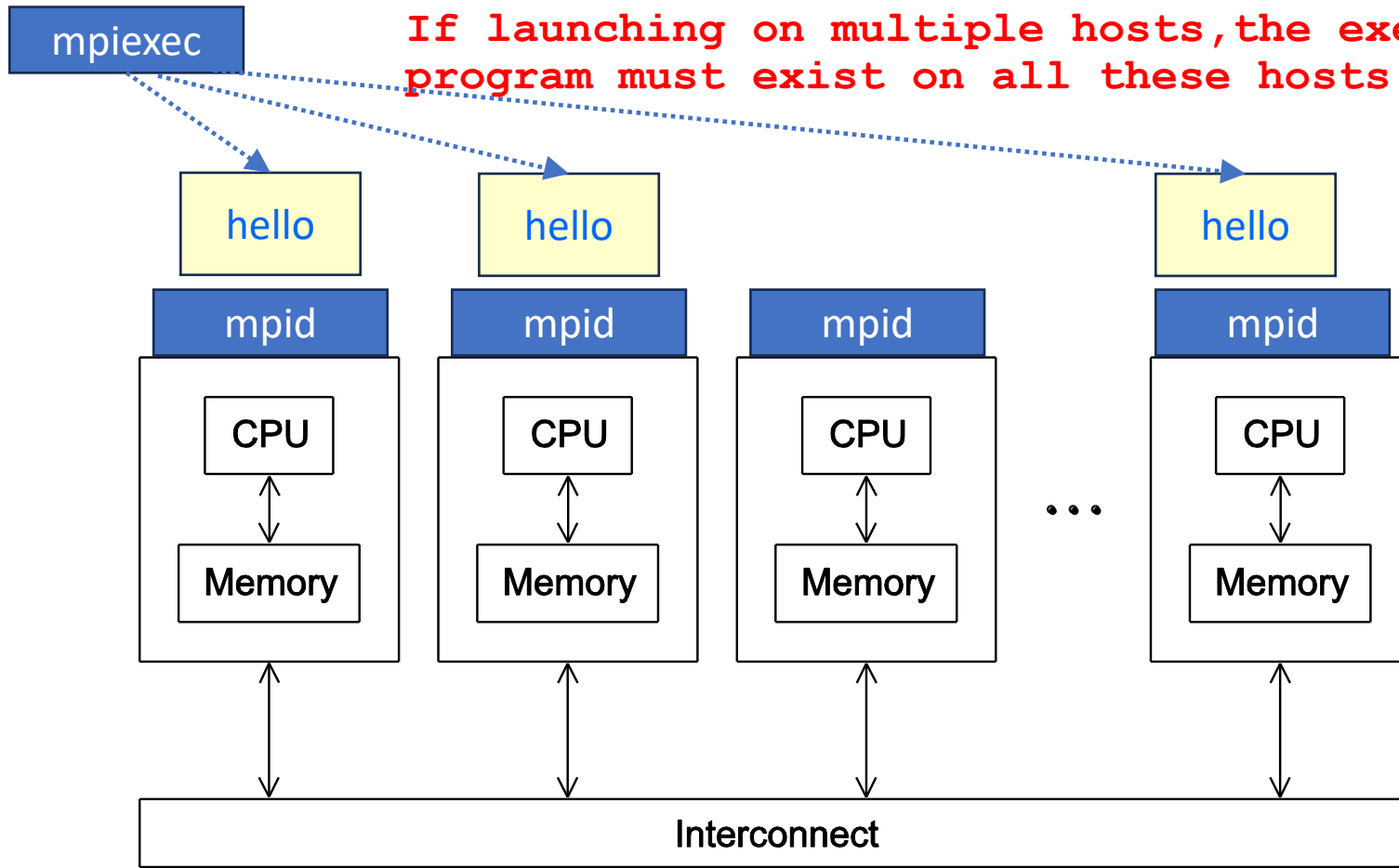
These variables are
NOT shared !
Every process has
its own copies of
these variables!

Running MPI program

➤ `mpiexec -n 4 hello`

➤ `mpiexec -hosts 2 COMP1 COMP2 hello`

If launching on multiple hosts, the executable program must exist on all these hosts!



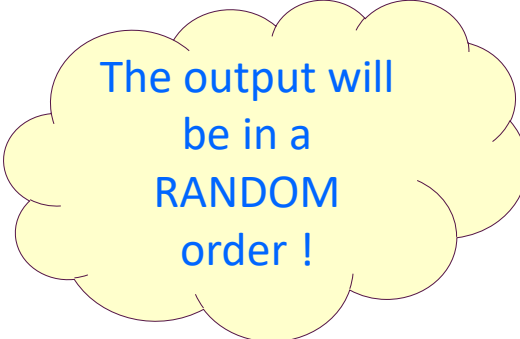
Our First MPI Program

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define MASTER 0

int main(int argc, char *argv)
{
    int numtasks, taskid, len;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Get_processor_name(hostname, &len);
    printf("Hello from task %d on %s!\n", taskid, hostname);
    if (taskid == MASTER)
        printf("MASTER: Number of MPI tasks is: %d\n", numtasks);
    MPI_Finalize();
}
```

```
C:\Users\ioana\Desktop\APD\MPI>
C:\Users\ioana\Desktop\APD\MPI>mpiexec -n 4 hello
Hello from task 0 on LAPTOP-IOANA!
MASTER: Number of MPI tasks is: 4
Hello from task 2 on LAPTOP-IOANA!
Hello from task 1 on LAPTOP-IOANA!
Hello from task 3 on LAPTOP-IOANA!
```



The output will
be in a
RANDOM
order !

Dealing with I/O

- Output from different processes:
 - In the hello example: each process writes to its `stdout`, we see all output in the console where the programs have been launched
 - The processes could be distributed on different hosts! In this case it depends on the MPI implementation what is the outcome
- Input:
 - Most MPI implementations only allow process 0 in `MPI_COMM_WORLD` access to `stdin`
 - If other processes need input data, then program must be written such that process 0 reads the data and send to the other processes.

Send and Receive
Blocking and NonBlocking

Sending and Receiving Messages

- The basic functions for sending and receiving messages in MPI according to the Blocking messaging paradigm are the **MPI_Send** and **MPI_Recv**

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

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

- MPI provides **equivalent datatypes** for all C datatypes. This is done for portability reasons.

Data types

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Message matching

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,  
send_comm);
```

MPI_Send

src = q



MPI_Recv

dest = r

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,  
recv_comm, &status);
```

r

q

Receiving messages

- A receiver can get a message without knowing:
 - the sender of the message,
 - or the tag of the message.
 - the amount of data in the message,
- MPI allows specification of wildcard arguments for both source and tag:
 - If source is set to `MPI_ANY_SOURCE`, then any process of the communication domain can be the source of the message.
 - If tag is set to `MPI_ANY_TAG`, then messages with any tag are accepted.
- The received message must be of length equal to or less than the length field specified.

Receiving Messages

- On the receiving end, the status variable can be used to get information about the `MPI_Recv` operation.

- The corresponding data structure contains:

```
typedef struct MPI_Status {  
    int MPI_SOURCE;  
    int MPI_TAG;  
    int MPI_ERROR; };
```

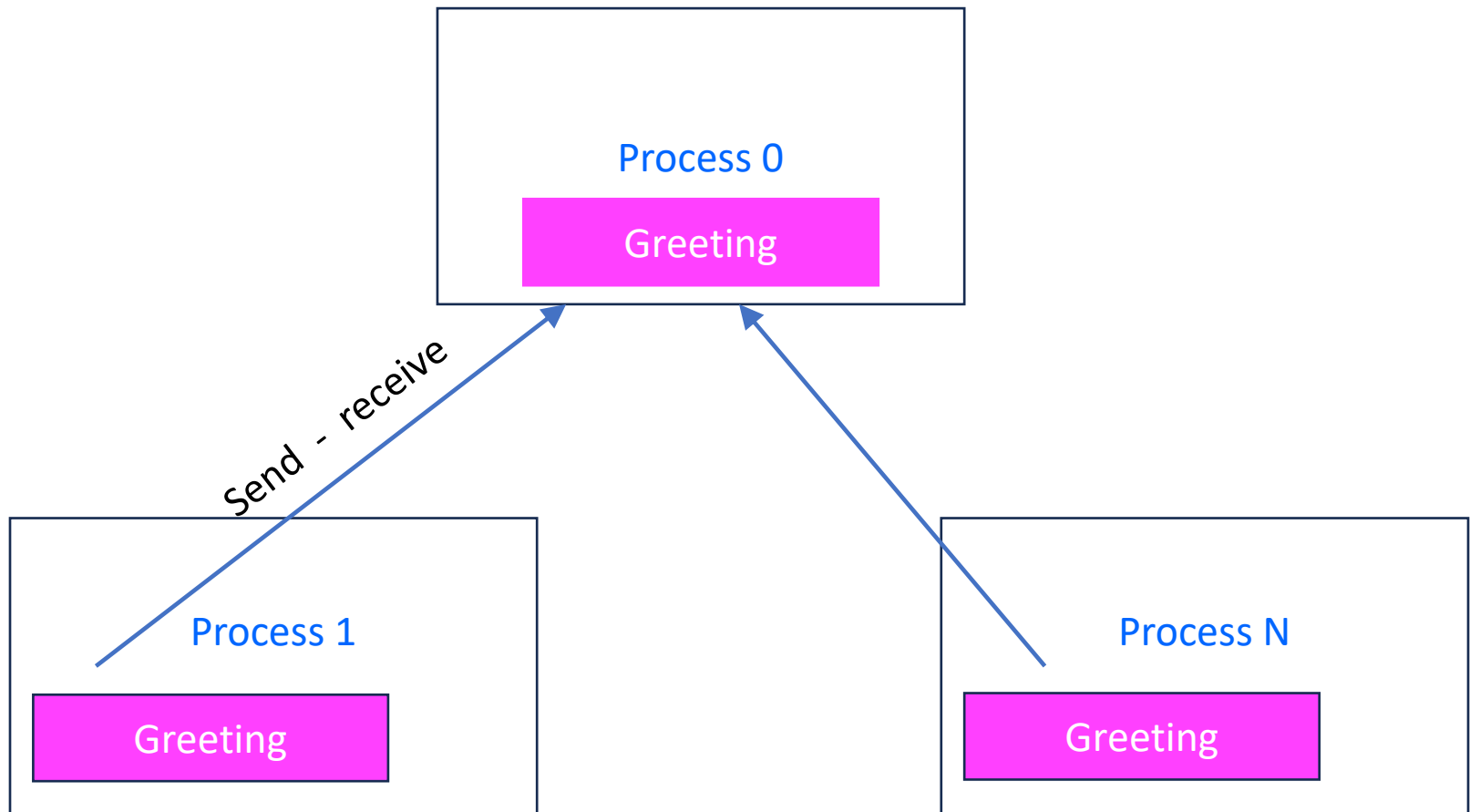
- The `MPI_Get_count` function returns the precise count of data items received.

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype  
                  datatype, int *count)
```

Semantic of MPI_Send and MPI_Recv

- MPI_Send and MPI_Receive belong to the Blocking messaging paradigm:
- **MPI_Recv always blocks** until a matching message is received.
- **MPI_Send behavior depends** on the MPI implementation: it can be **blocking buffered** or **blocking nonbuffered**
- **MPI_Send could, depending on the implementation:**
 - return when the message has been copied in the internal send buffer, but it is not yet arrived at the receiver
 - return only when the message is at the receiver

Example: Send/Receive Messages



Example: Send/Receive Message

```
#include <stdio.h>
#include <string.h> /* For strlen */
#include "mpi.h" /* For MPI functions, etc */

#define MAX_STRING 100

int main(void) {
    char    greeting[MAX_STRING]; /* String storing message */
    int     comm_sz;              /* Number of processes */
    int     my_rank;              /* My process rank */

    /* Start up MPI */
    MPI_Init(NULL, NULL);

    /* Get the number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    /* Get my rank among all the processes */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
```

Example contd.

```
if (my_rank != 0) {
    /* Create message */
    sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
    /* Send message to process 0 */
    MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
             MPI_COMM_WORLD);
} else {
    /* Print my message */
    printf("I am process %d of %d!\n", my_rank, comm_sz);
    for (int q = 1; q < comm_sz; q++) {
        /* Receive message from process q */
        MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
                 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        /* Print message from process q */
        printf("%s\n", greeting);
    }
}
/* Shut down MPI */
MPI_Finalize();
return 0;
}
```

Apparently there is a shared variable, BUT actually they are **different buffers because they are in different processes !**

Example contd.

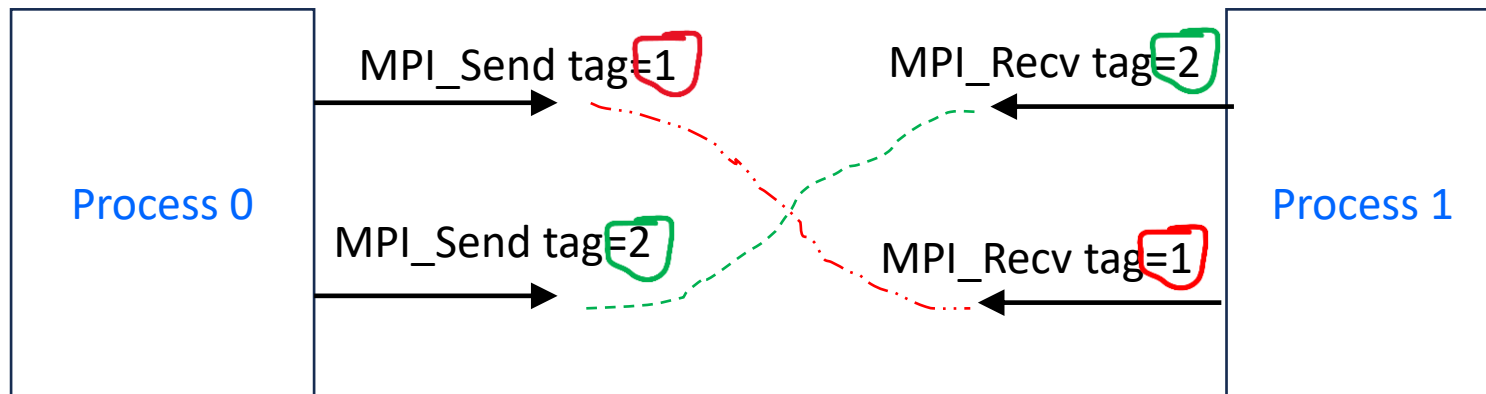
```
C:\Users\ioana\Desktop\APD\MPI>mpiexec -n 4 mpi_hello  
I am process 0 of 4!  
Greetings from process 1 of 4!  
Greetings from process 2 of 4!  
Greetings from process 3 of 4!
```

The output of this run will
ALWAYS be in this order!
Process 0 **waits** to receive
(because it uses BLOCKING
RECEIVES)
in order, messages from
processes 1 to N-1 !

Issues with MPI_Send and MPI_Recv

- MPI_Send and MPI_Receive are Blocking functions:
- **MPI_Recv always blocks** until a matching message is received.
- **MPI_Send behavior depends** on the MPI implementation: it can be **blocking buffered** or **blocking nonbuffered**
- **MPI_Send could, depending on the implementation:**
 - return when the message has been copied in the internal send buffer, but it is not yet sent (blocking buffered)
 - return only when the message is at the receiver (blocking non-buffered)

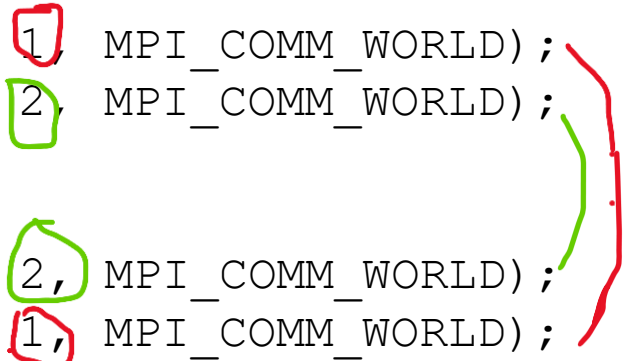
Danger of Deadlock Example



- MPI_Recv is always blocked until message is received.
- **If MPI_Send is unbuffered, it does not return until receiver got the message, thus the first Send causes a deadlock.**
- If MPI_Send is buffered, there is no deadlock.
- In order to be safe, programmers should assume that MPI_Send is unbuffered blocking !

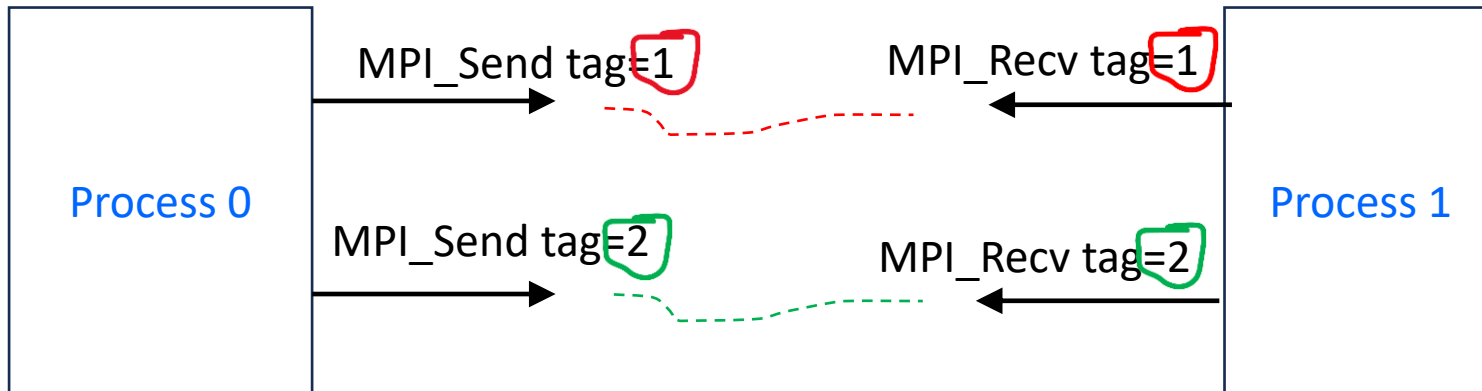
Deadlock Example

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```



If MPI_Send is unbuffered blocking, there is a deadlock.

Deadlock-Free Solution

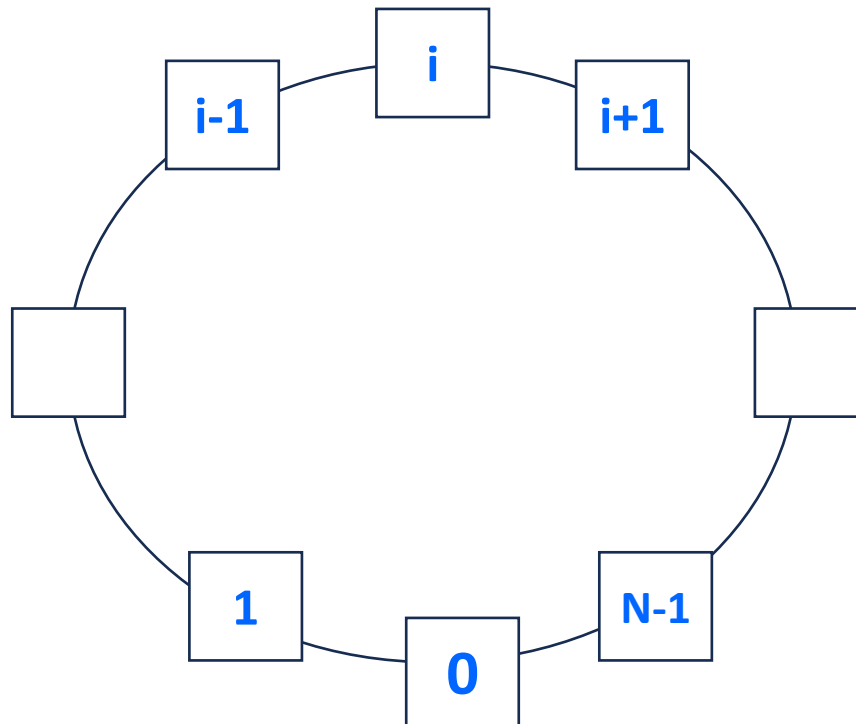


Send and Receive in the same order of tags

Even if MPI_Send is unbuffered blocking, there is NO deadlock

Example: Communicating around a Ring topology

- A frequent pattern in distributed algorithms
- Consider N processes forming a Ring topology.
- We must ensure the following communication pattern: every process i sends a message to process $i + 1$ (modulo the number of processes) and receives a message from process $i - 1$ (modulo the number of processes).



Deadlock

Consider that **every process** in the Ring executes a sequence of code:

```
int a[10], b[10], npes, myrank;  
MPI_Status status;  
...  
MPI_Comm_size(MPI_COMM_WORLD, &npes);  
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
...  
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);  
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);  
...
```

We always have a deadlock because MPI_Recv is blocking!
Every process initiates a Receive operation, but no process can reach the Send operation, thus Receive remains blocked forever!

Deadlock

Consider that **every process** in the Ring executes a sequence of code:

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
...
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
...
```

We have a deadlock if MPI_Send is unbuffered blocking!

Every process initiates a Send operation, but no process can reach the Receive operation, thus Send remains blocked forever!

Avoiding Deadlocks

We can break the circular wait to **be sure that we avoid deadlocks**:

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
. . .
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
}
else {
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
}
...
```

Blocking vs NonBlocking Communication

- **MPI_Send/MPI_Recv are blocking communication operations**
 - Return of the function implies completion
 - When these calls return the memory locations used in the message transfer can be safely accessed for reuse
 - For “send” completion implies variable sent can be reused/modified
 - Modifications will not affect data intended for the receiver
 - For “receive” variable received can be read
- **MPI_Isend/MPI_Irecv are non-blocking (asynchronous) communication operations**
 - Function returns immediately
 - ***completion has to be separately tested for by the programmer!!!!***
 - These are primarily used to overlap computation and communication to improve performance

Non-Blocking Communication

- MPI provides functions for non-blocking send and receive:

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,  
             int dest, int tag, MPI_Comm comm,  
             MPI_Request *request)
```

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,  
             int source, int tag, MPI_Comm comm,  
             MPI_Request *request)
```

- These operations return before the operations have been completed. Function **MPI_Test** tests whether or not the non-blocking send or receive operation identified by its request has finished.

```
int MPI_Test(MPI_Request *request, int *flag,  
            MPI_Status *status)
```

- **MPI_Wait** waits for the operation to complete.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Deadlock with blocking Receive

Consider that **every process** in the Ring executes a sequence of code:

```
int a[10], b[10], npes, myrank;  
MPI_Status status;  
...  
MPI_Comm_size(MPI_COMM_WORLD, &npes);  
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
...  
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);  
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);  
...
```

We always have a deadlock because MPI_Recv is blocking!
Every process initiates a Receive operation, but no process can reach the Send operation, thus Receive remains blocked forever!

Avoiding Deadlocks with unblocking Receive

Use non-blocking receive to avoid deadlocks:

```
int a[10], b[10], npes, myrank;
MPI_Status status;
MPI_Request request;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
. . .

MPI_IRecv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
          MPI_COMM_WORLD, &request);

MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);

MPI_Wait(&request, &status);

...
```


Source Code

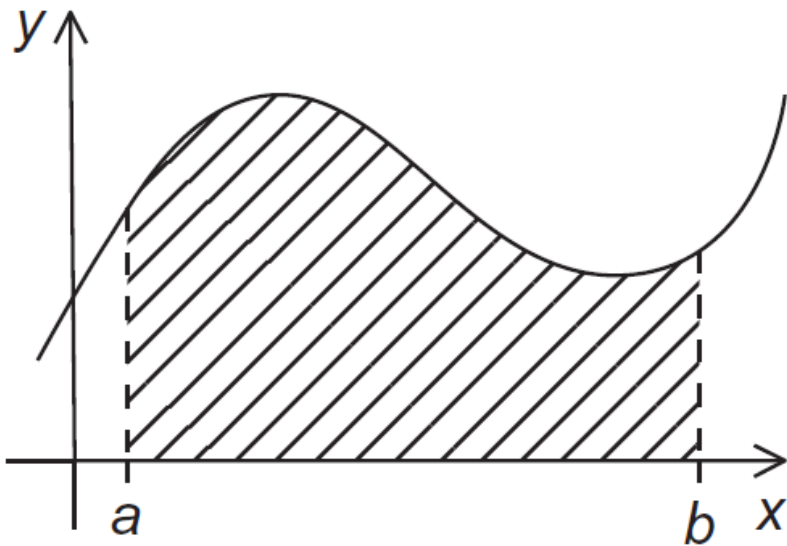
- [mpi_first.c](#)
- [mpi_hello.c](#)

First MPI Parallel Program

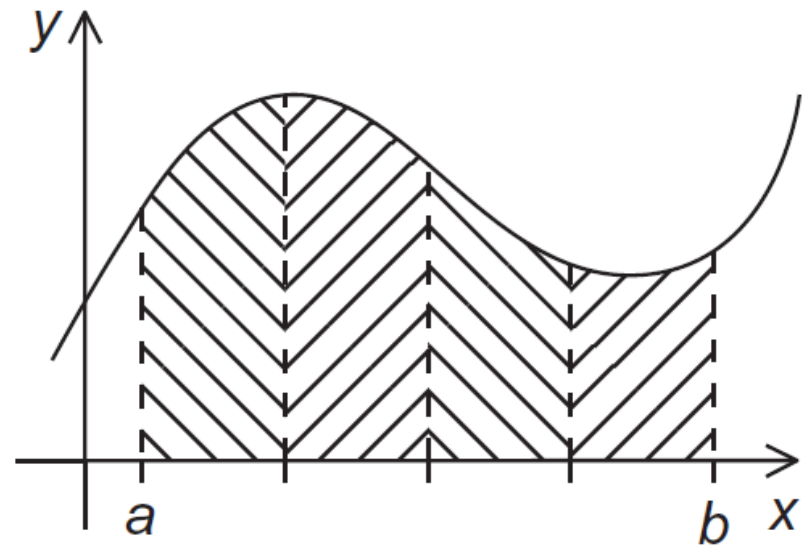
The Trapezoidal Rule with MPI

The Trapezoidal Rule

- Area under curve or Numerical Integration can be computed by the Trapezoidal Rule method

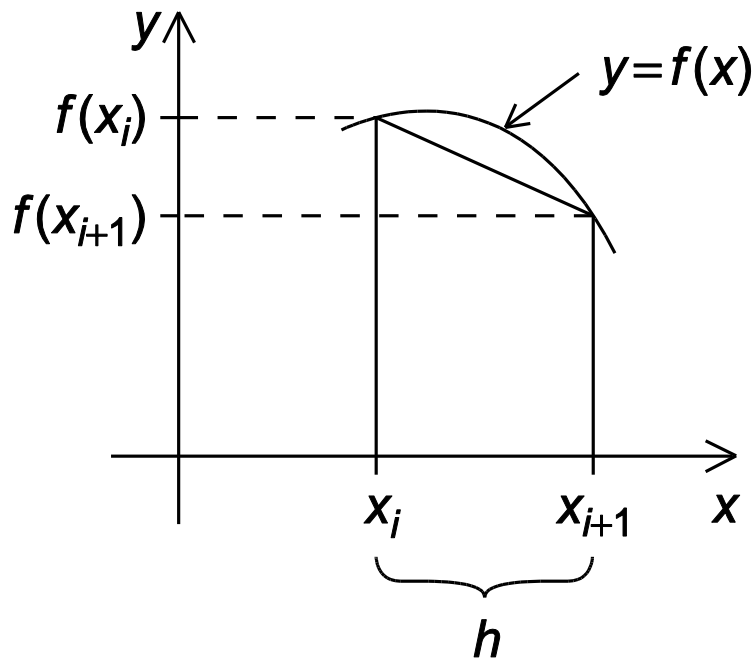


(a)



(b)

The Trapezoidal Rule

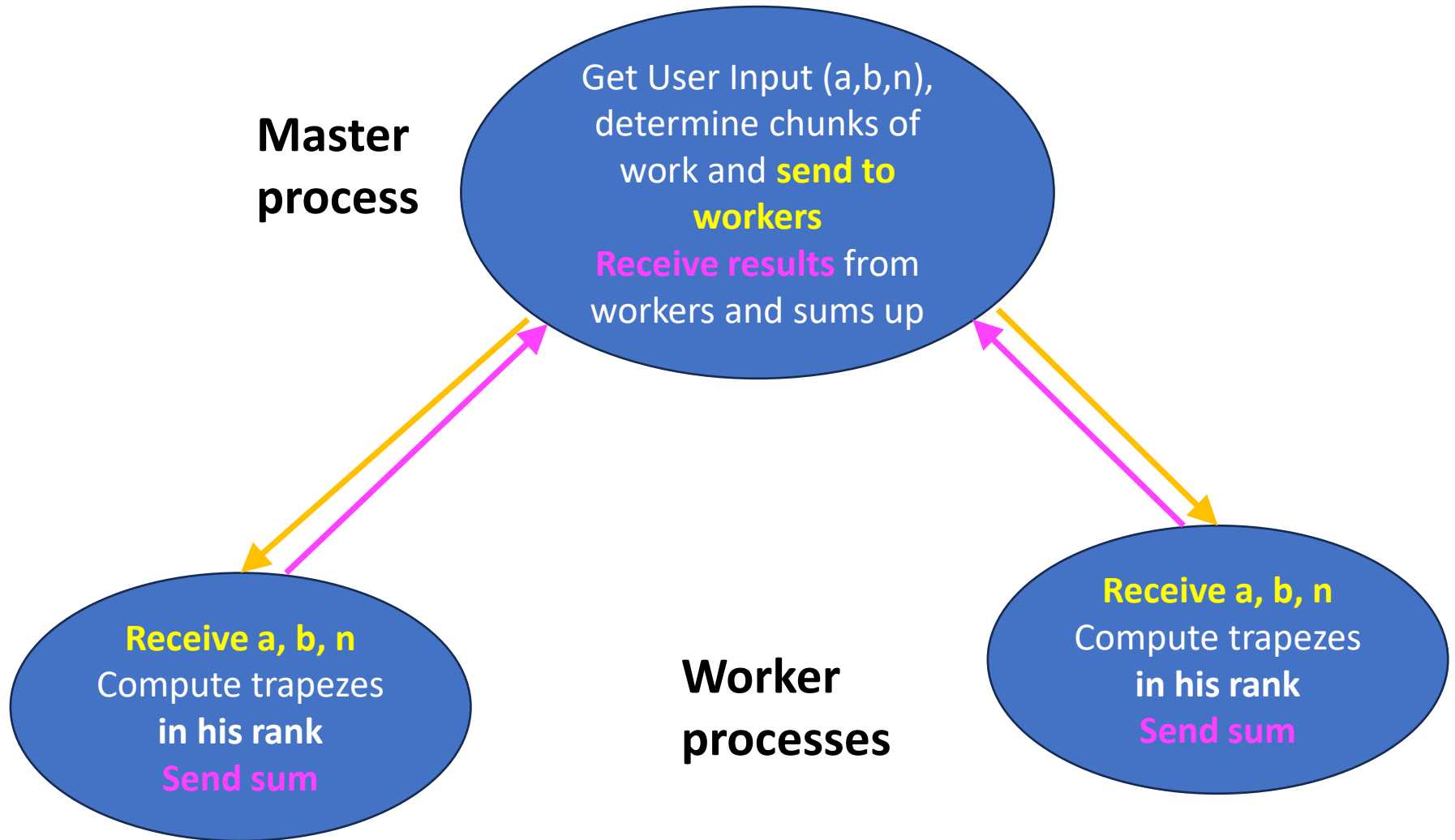


$$\text{Area of one trapezoid} = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$
$$h = \frac{b - a}{n}$$

$$x_0 = a, x_1 = a + h, x_2 = a + 2h, \dots, x_{n-1} = a + (n-1)h, x_n = b$$

$$\text{Sum of trapezoid areas} = h[f(x_0)/2 + f(x_1) + f(x_2) + \dots + f(x_{n-1}) + f(x_n)/2]$$

Tasks and communications for Trapezoidal Rule with MPI



Tasks and communications for Trapezoidal Rule with MPI – Better:

Master Process

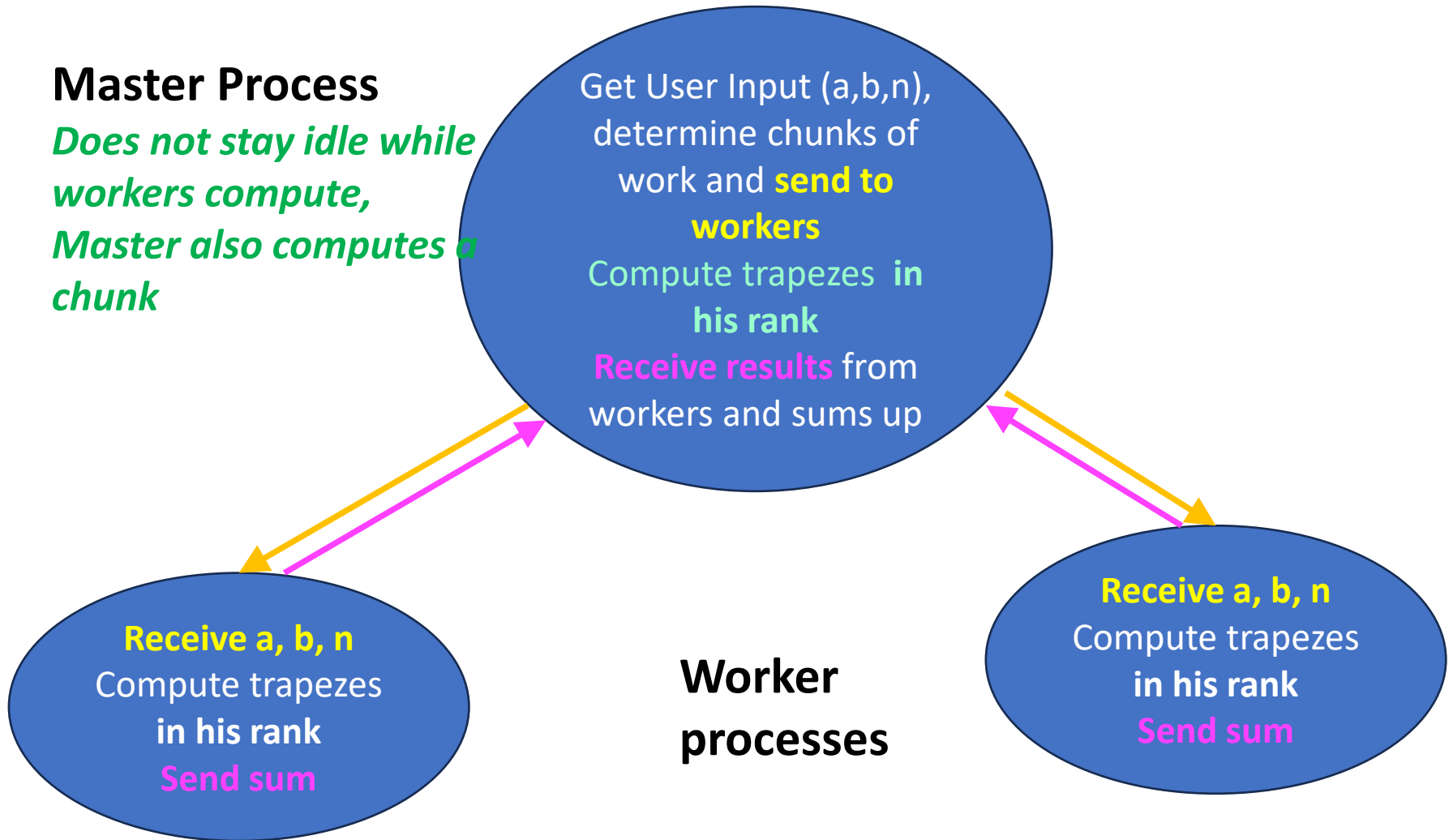
Does not stay idle while workers compute, Master also computes a chunk

Get User Input (a,b,n),
determine chunks of
work and **send to
workers**
Compute trapezes in
his rank
Receive results from
workers and sums up

Worker processes

Receive a, b, n
Compute trapezes
in his rank
Send sum

Receive a, b, n
Compute trapezes
in his rank
Send sum



```

void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
               int* n_p) {

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
        for (int dest = 1; dest < comm_sz; dest++) {
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else { /* my_rank != 0 */
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    }
} /* Get_input */

```

mpi_trap2.c

```
int main(void) {
    /* Every process will have its own copy of these variables */
    int my_rank, comm_sz;
    int n, local_n;
    double a, b, h, local_a, local_b;
    double local_int, total_int;

    /* Let the system do what it needs to start up MPI */
    MPI_Init(NULL, NULL);

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    Get_input(my_rank, comm_sz, &a, &b, &n);

    h = (b-a)/n;          /* h is the same for all processes */
    local_n = n/comm_sz;  /* So is the number of trapezoids */
}
```

Continued next slide ...


```

local_a = a + my_rank * local_n * h;
local_b = local_a + local_n * h;
local_int = Trap(local_a, local_b, local_n, h);

/* Add up the integrals calculated by each process */
if (my_rank != 0) {
    MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
else {
    total_int = local_int;
    for (int source = 1; source < comm_sz; source++) {
        MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total_int += local_int;
    }
}
if (my_rank == 0) {
    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.15e\n",
           a, b, total_int);
}
MPI_Finalize();
} /* main */

```

Source code

- [mpi_trap2.c](#)

Synchronization Points – MPI Barrier

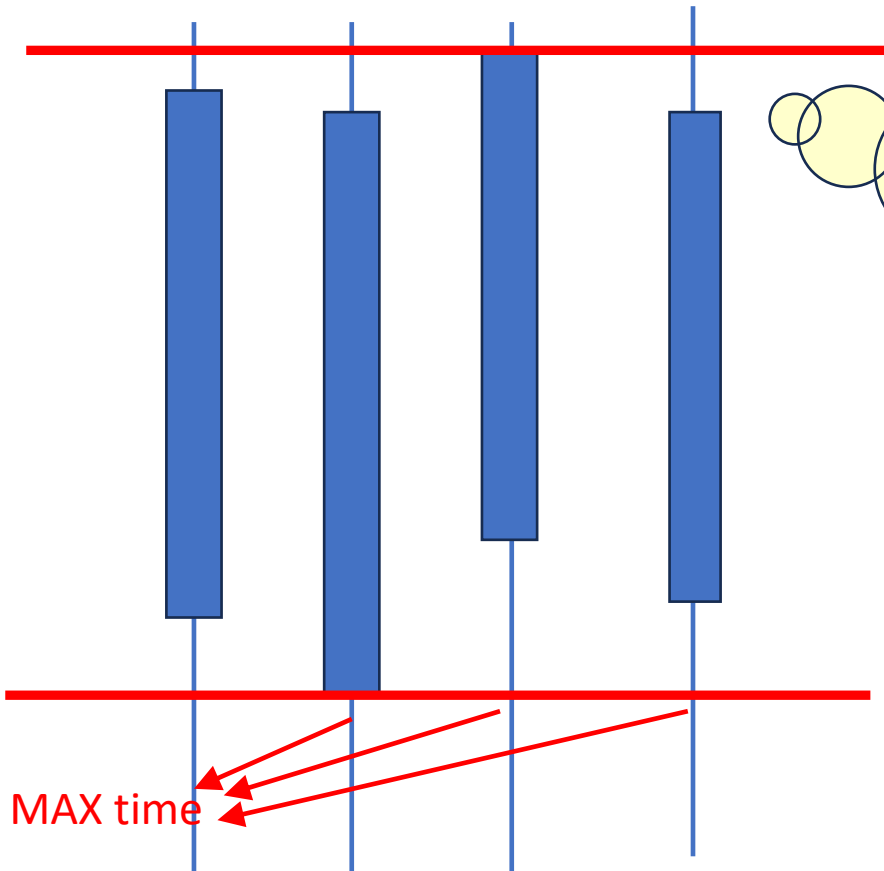
Barrier

- Ensures that no process will return from calling it until every process in the communicator has started calling it.
- The barrier synchronization operation is performed in MPI using:

```
int MPI_Barrier(MPI_Comm comm)
```

Using Barrier when measuring runtime

Several MPI processes running in parallel, doing some work



To measure runtime of a program composed by several processes:

1. Barrier to have all processes start together
2. Barrier to know when all processes have finished
3. Each process measures localtime between barriers
4. Send localtimes to Master
5. Master finds longest localtime

Message Passing – Conclusions

- Basic operations in message passing systems are Send and Receive.
 - They can be Blocking and NonBlocking
 - NonBlocking operations can help to have more efficient programs - can do in parallel communication and computation
 - NonBlocking operations are more difficult to be used in programs
 - Blocking operations can lead to deadlock if not used properly
 - In MPI:
 - blocking: MPI_Send, MPI_Recv
 - Non-blocking: MPI_Isend, MPI_Irecv