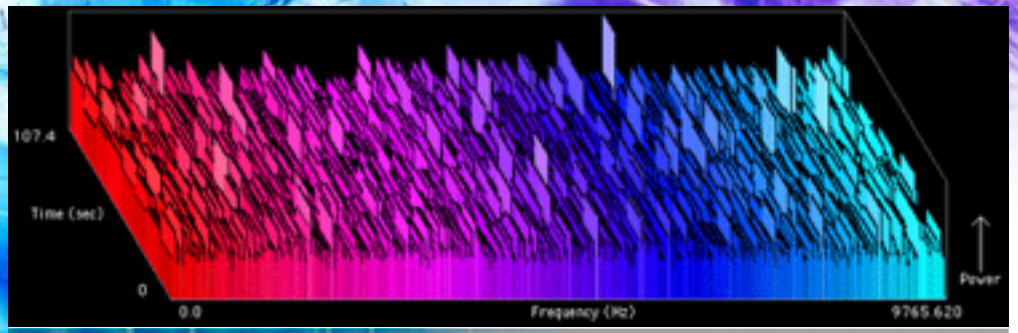
Programming with MPI

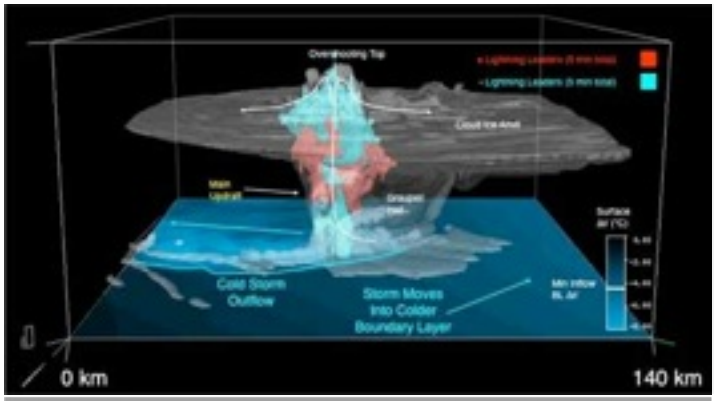
Pedro Velho

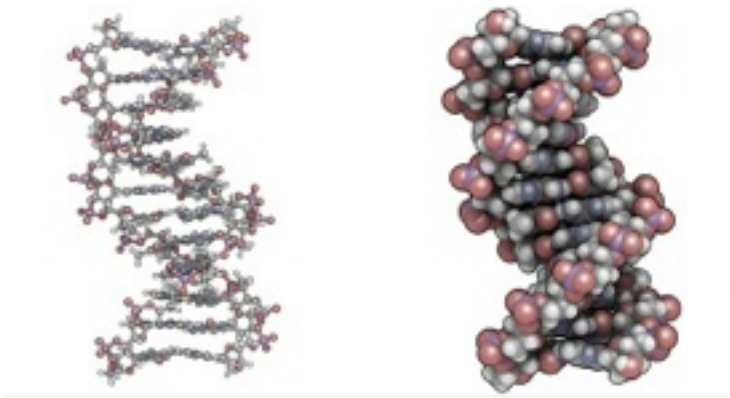


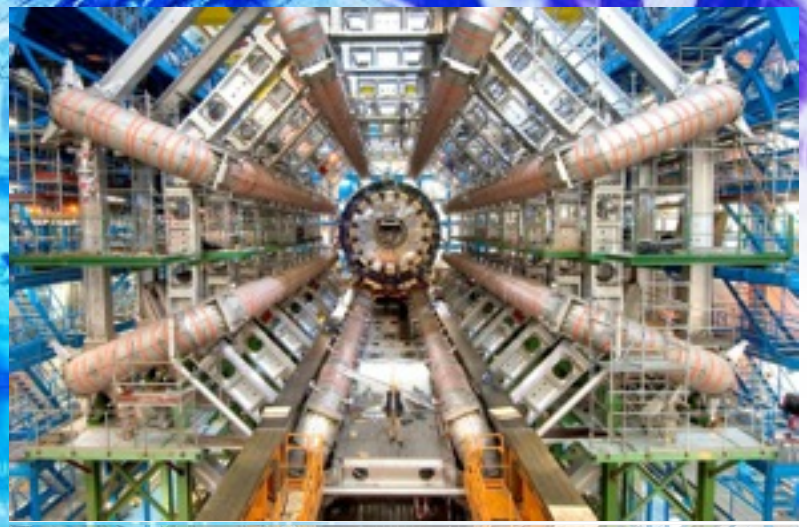
Science Research Challenges

Some applications require tremendous computing power

- Stress the limits of computing power and storage
- Who might be interested in those applications?
- Simulation and analysis in modern science or e-Science







Example LHC Large Hadron Collider (CERN)

LHC Computing Grid

Worldwide collaboration with more than **170 computing** centers in **34 countries**

A lot of **data to store**: **1 GByte/sec**

Need high computing power to obtain results in feasible time

How can we achieve this goal?

Old school high performance computing

- **Sit and wait for a new processor**
  - **Wait** until CPU speed doubles
  - Speed **for free**
    - Don’t need to recompile or rethink the code
    - Don’t need to pay a computer scientist to do the job

Unfortunately this is not true anymore...

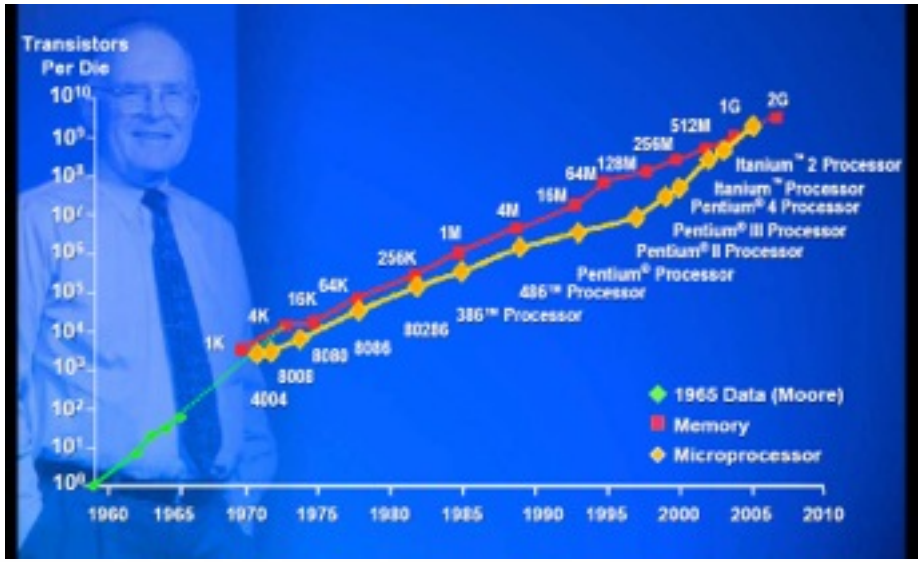
The need for HPC

Moore's Law (is it true or not?)

- Moore said in 1965 that **the number of transistors will double approximately every 18 months**

- **But,** it's wrong to think that if the number of transistors doubles then **processors run two times faster every 18 months**

- **Clock speed and transistor density are not co-related!**



Can I buy a 20GHz processor today?

Single processor is not enough

- **Curiously Moore's law is still true**
  - Number of transistor still doubles every 18 months
- However **there are other factors that limit CPU clock speed:**
  - Heating
  - Power consumption
- Super computers are too expensive for medium size problems

The solution is to use distributed computing!

Distributed Computing

- Cluster of workstations

- Commodity PCs (nodes) interconnected through a local network
- Affordable
- Medium scale
- Very popular among researchers

- Grid computing

- Several cluster interconnected through a wide area network
- Allows resource sharing (less expensive for universities)
- Huge scale

This is the case of GridRS!

Distributed Programming

- **Applications must be rewritten**
  - No shared memory between nodes (processes need to communicate)
  - Data exchange through network
- **Possible solutions**
  - **Ad Hoc**: Work only for the platform it was designed for
  - **Programming models:**

    Ease data exchange and process identification

    Portable solution

    Examples: MPI, PVM, Charm++

Message Passing Interface (MPI)

- **It is a programming model**
  - Not a specific implementation or product
  - Describes the interface and basic functionalities
- **Scalable**
  - Must handle multiple machines
- **Portable**
  - Socket API may change from one OS to another
- **Efficient**
  - Optimized communication algorithms

MPI Programming

- **MPI implementations (all free):**
  - OpenMPI (GridRS)

    http://www.open-mpi.org/

  - MPICH

    http://www.mpich.org

  - LAM/MPI

    http://www.lam-mpi.org/

MPI Programming

- **MPI References**

  - Books

    MPI: The Complete Reference, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996.

    Parallel Programming with MPI, by Peter Pacheco, Morgan Kaufmann, 1997.

  - The standard:

    http://www.mpi-forum.org

MPI Programming

- **SPMD** model: **S**ingle **P**rogram **M**ultiple **D**ata
  - All processes execute the “same source code”
  - But, we can define specific blocks of code to be executed by specific processes (if-else statements)
- MPI offers:
  - A way of identifying processes
  - Abstraction of low-level communication
  - Optimized communication
- MPI doesn't offer:
  - Performance gains for free
  - Automatic transformation of a sequential to a parallel code

MPI Programming

Possible way of parallelizing an application with MPI:

Start from **sequential version** → Split the application in **tasks** → Choose a **parallel strategy** → Implement with **MPI**

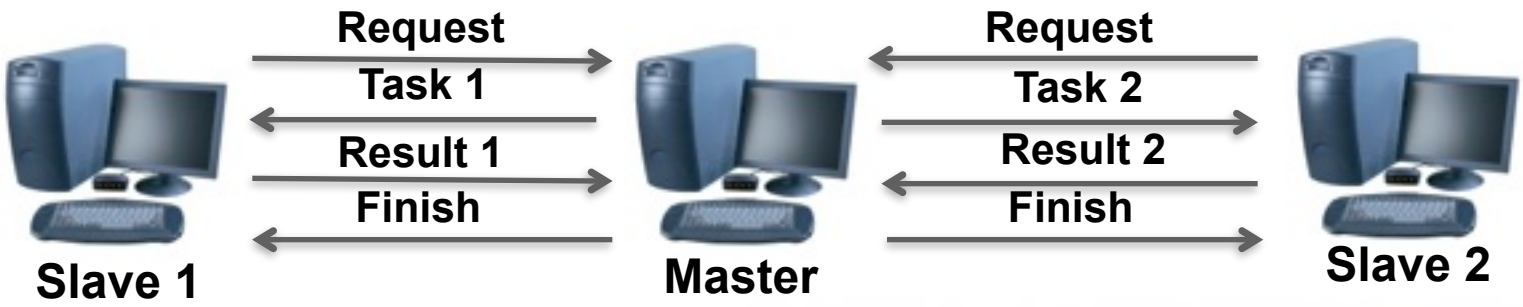
Some parallel strategies

- **Master/Slave**
- **Pipeline**
- **Divide and Conquer**

Parallel Strategies

Master/Slave

- Master is one process that centralizes all tasks
- Slaves request tasks when needed
- Master sends tasks on demand



Parallel Strategies

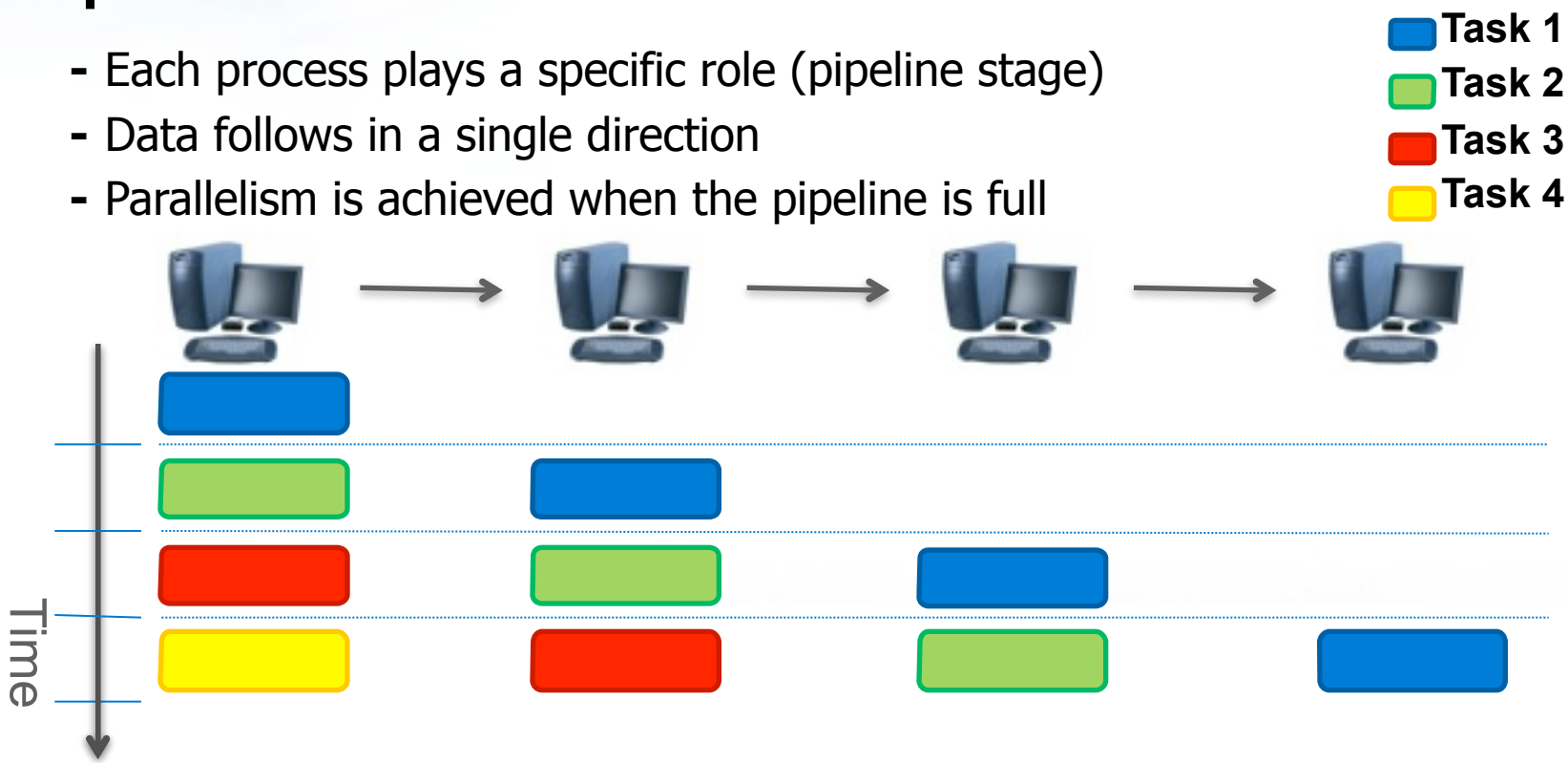
Master/Slave

- Master is often the bottleneck
- Scalability is limited due to centralization
- Possible to use replication to improve performance
- Good for heterogenous platforms

Parallel Strategies

Pipeline

- Each process plays a specific role (pipeline stage)
- Data follows in a single direction
- Parallelism is achieved when the pipeline is full



Parallel Strategies

Pipeline

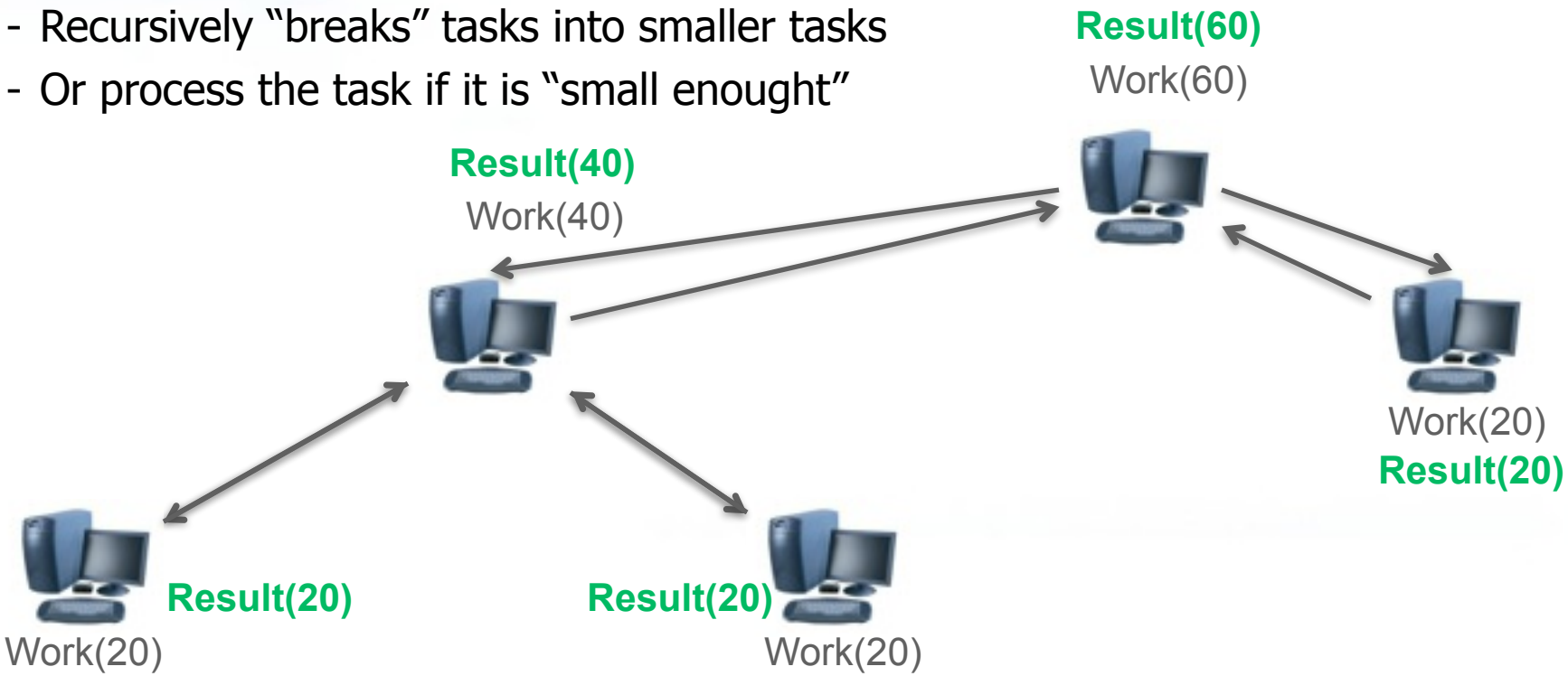
- Scalability is limited by the number of stages
- Synchronization may lead to “bubbles”

  Example: **slow sender** and **fast receiver**
- Difficult to use on heterogenous platforms

Parallel Strategies

Divide and Conquer

- Recursively "breaks" tasks into smaller tasks
- Or process the task if it is "small enought"



Parallel Strategies

Divide and Conquer

- More scalable
- Possible to use replicated branches
- In practice it may be difficult to “break” tasks
- Suitable for **branch and bound** algorithms

Hands-on GridRS

1) Log in for the first time on GridRS

- $ ssh user@gridrs.lad.pucrs.br

2) Configure the SSH and OpenMPI environment on GridRS

- https://bitbucket.org/schnorr/gridrs/wiki

MPI Programming

Exercise 0: Hello World

Write the following hello world program in your home directory.

Compile the source code on the frontend:

$ mpicc my_source.c -o my_binary

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

int main(int argc, char **argv){
 /* A local variable to store the hostname */
 char hostname[1024];

 /* Initialize MPI */
 MPI_Init(&argc, &argv);

 /* Discover the hostname */
 gethostname(hostname, 1023);

 printf("Hello World from %s\n", hostname);

 /* Finalize MPI */
 return MPI_Finalize();
}
```

MPI Programming

Configure GridRS environment, compile and run your app:

- https://bitbucket.org/schnorr/gridrs/wiki/Run_a_Local_experiment

Use timesharing while allocating resources:

- $ oarsub -l nodes=3 **-t timesharing** -I

Running with "X" processes (you can choose the nb. of processes)

- `$ mpirun --mca btl tcp,self -np X --machinefile $OAR_FILE_NODES ./my_binary`

MPI Programming

How many processing units are available?

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

- **comm**
  - **Communicator**: used to group processes
  - For grouping all processes together use **MPI_COMM_WORLD**
- **psize**
  - Returns the total amount of processes in this communicator

Example:

```
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

MPI Programming

Exercise 1

- Create a program that prints hello world **and the total number of available process** on the screen
- Try several processes configurations with –np to see if your program is working

MPI Programming

Assigning Process Roles

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

- **comm**
  - **Communicator**: specifies the process that can communicate
  - For grouping all processes together use **MPI_COMM_WORLD**
- **rank**
  - Returns the unique ID of the calling process in this communicator

Example:

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

MPI Programming

Exercise 2

- Create a program that prints hello world, the total number of available process **and the process rank** on the screen
- Try several processes configurations with –np to see if your program is working

MPI Programming

Exercise 3 – Master/Slave

if "I am process 0" then

Print: "I'm the master!"

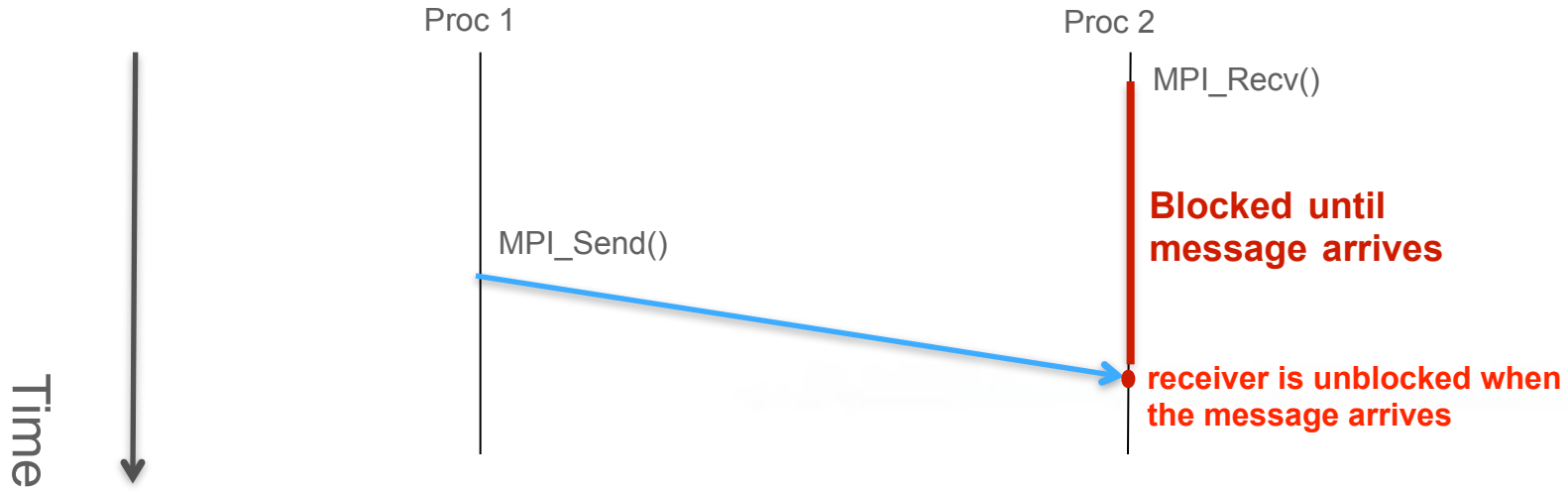
else

Print: "I'm slave <ID> of <SIZE>!", replacing "ID" by the process rank and SIZE by the number of processes.

MPI Programming

Sending messages (synchronous)

- Receiver waits for message to arrive (blocking)
- Sender unblocks receiver when the message arrives



MPI Programming

Sending messages (synchronous)

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

- **buf**: pointer to the data to be sent
- **count**: number of data elements in buf
- **dtype**: type of elements in buf
- **dest**: rank of the destination process
- **tag**: a number to "classify" the message (optional)
- **comm**: communicator

MPI Programming

Receiving messages (synchronous)

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

- **buf**: pointer to where data will be stored
- **count**: maximum number of elements that **buf** can handle
- **dtype**:type of elements in **buf**
- **src**: rank of sender process (use MPI_ANY_SOURCE to receive from any source)
- **tag**: message tag (use MPI_ANY_TAG to receive any message)
- **comm**: communicator
- **status**: information about the received message, if desired can be ignored using **MPI_STATUS_IGNORE**

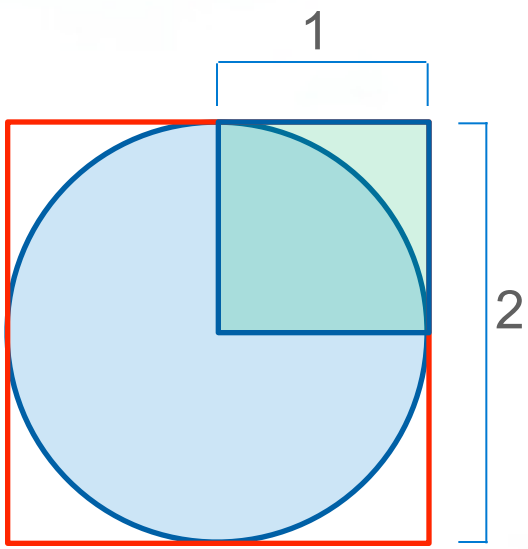
MPI Programming

Exercise 4 – Master/Slave with messages

- Master **receives** one message per slave
- Slaves **send** a single message to the master with their rank
- When the master receives a message, it **prints** the received rank

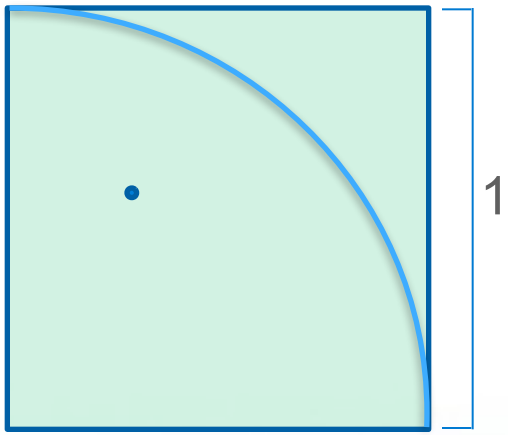
MPI Programming

- **Exercise 5 – Computing π by Monte Carlo Method**



Area of Circle =
$\pi r^2 = \pi (1)^2 = \pi$

$P(I) = A = \frac{\pi}{4}$

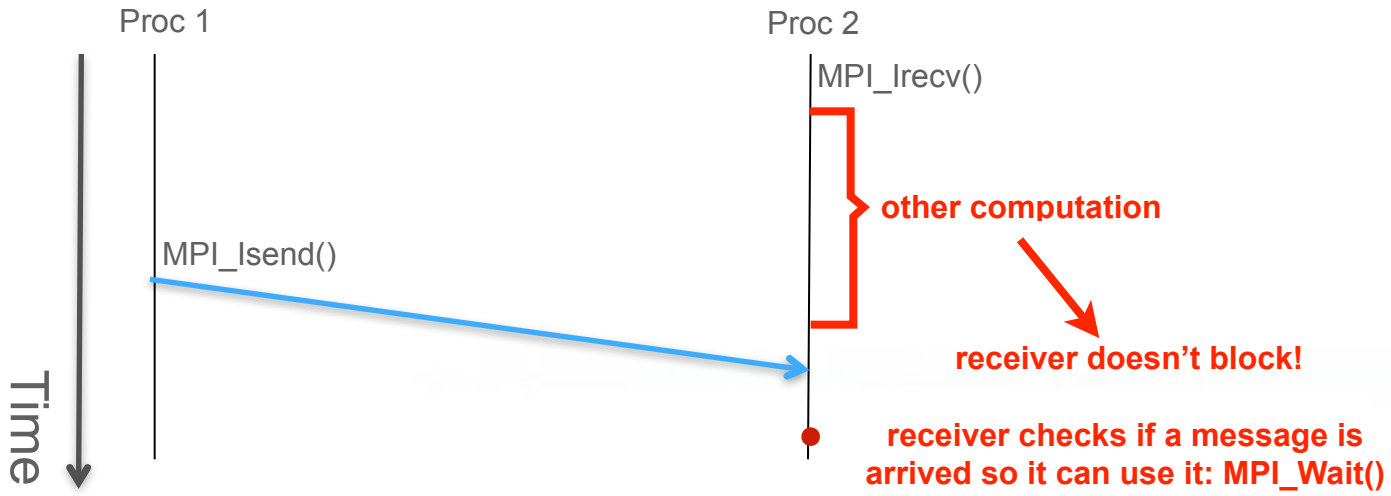


$A = \frac{\pi}{4}$

MPI Programming

Asynchronous/Non-blocking messages

- Process signs it is waiting for a message
- Continue working meanwhile
- Receiver can check any time if the message is arrived



MPI Programming

Master wants to send a message to everybody

- First solution, master **sends N-1 messages**
- Optimized collective communication: send is done in parallel