A very basic Introduction to MPI parallelization

ISYA 41
Socorro
2018

Juan Carlos Muñoz Cuartas
Instituto de Física
Universidad de Antioquia

Lets assume you have done all the algorithmic and code optimizations common sense indicates, but still your problem is too large, or is to slow…

- How do you solve an extensive problem in computers with limited amount of memory?

- How do we solve an intensive problem in a reasonable amount of time?

Divide the work...! Parallel computing!

There are different ways to do parallel computing

- OpenMP
- NVIDIA-CUDA parallelization
- Vectorization for especial purpose hardware
- Message passing (MPI)

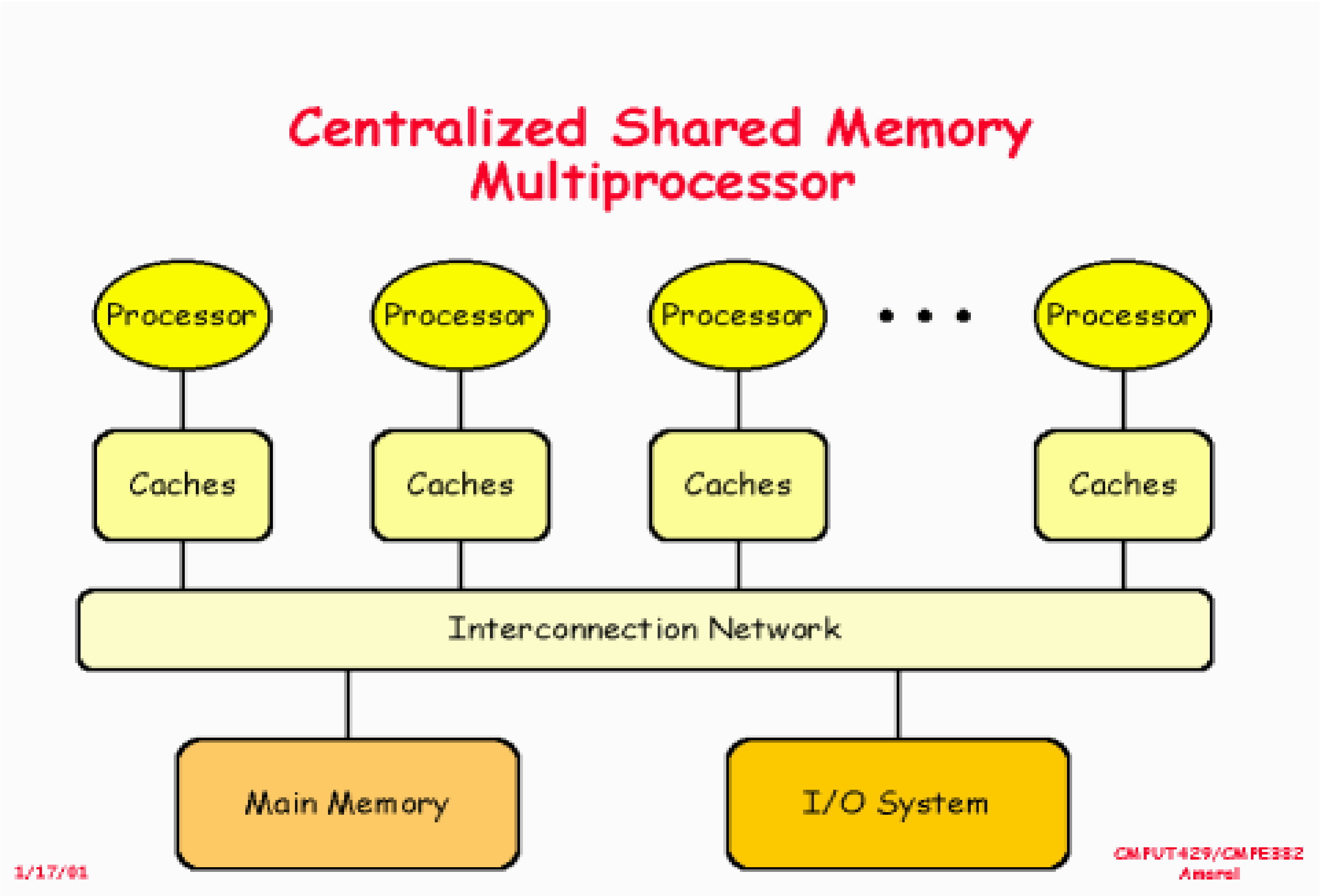
What will we do?

- Because of portability
- Because of time constraints
- Because if you get this strategy, you are ready to go *on your own* with any other methods...
- Because of the standard

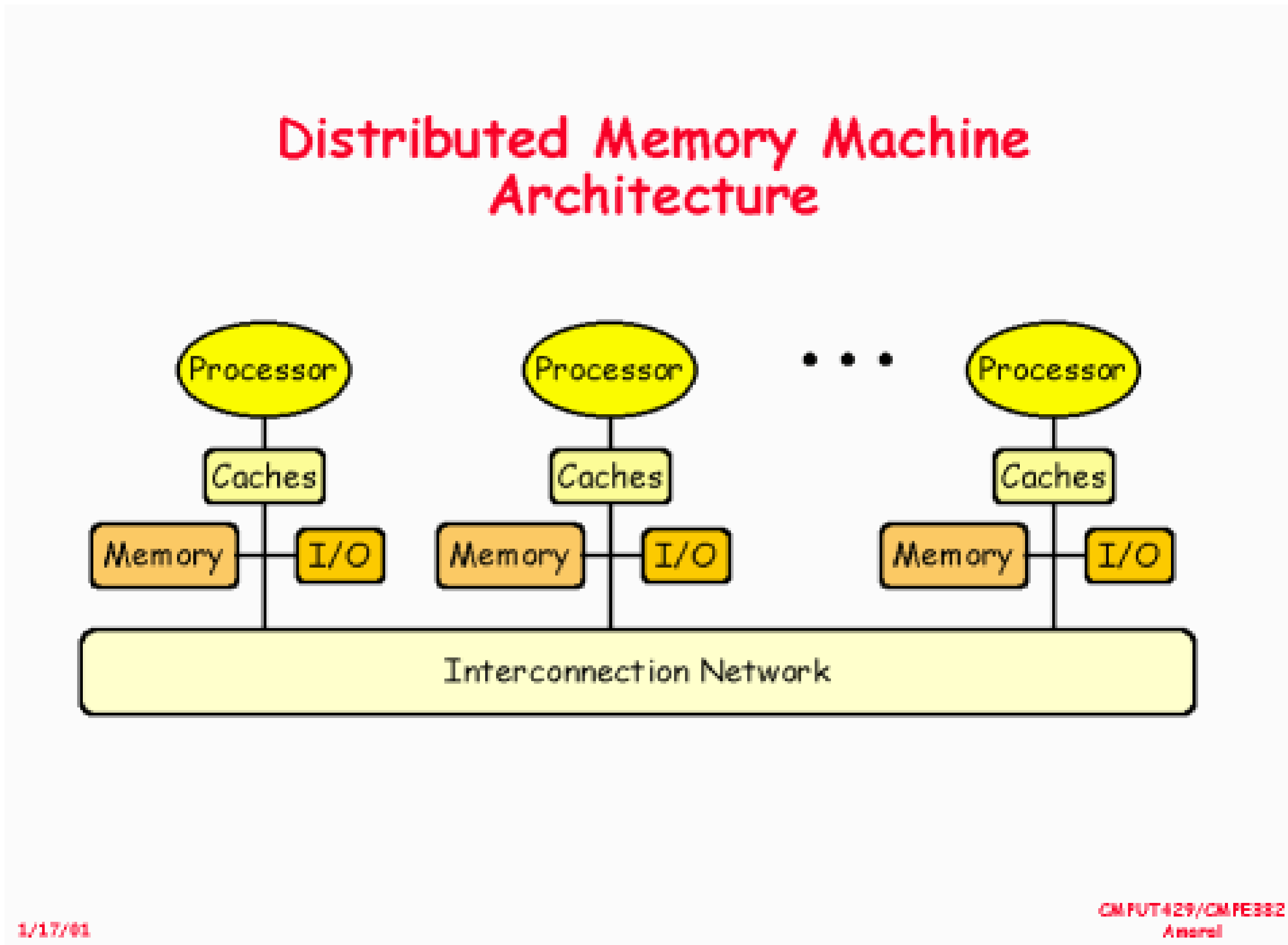
...

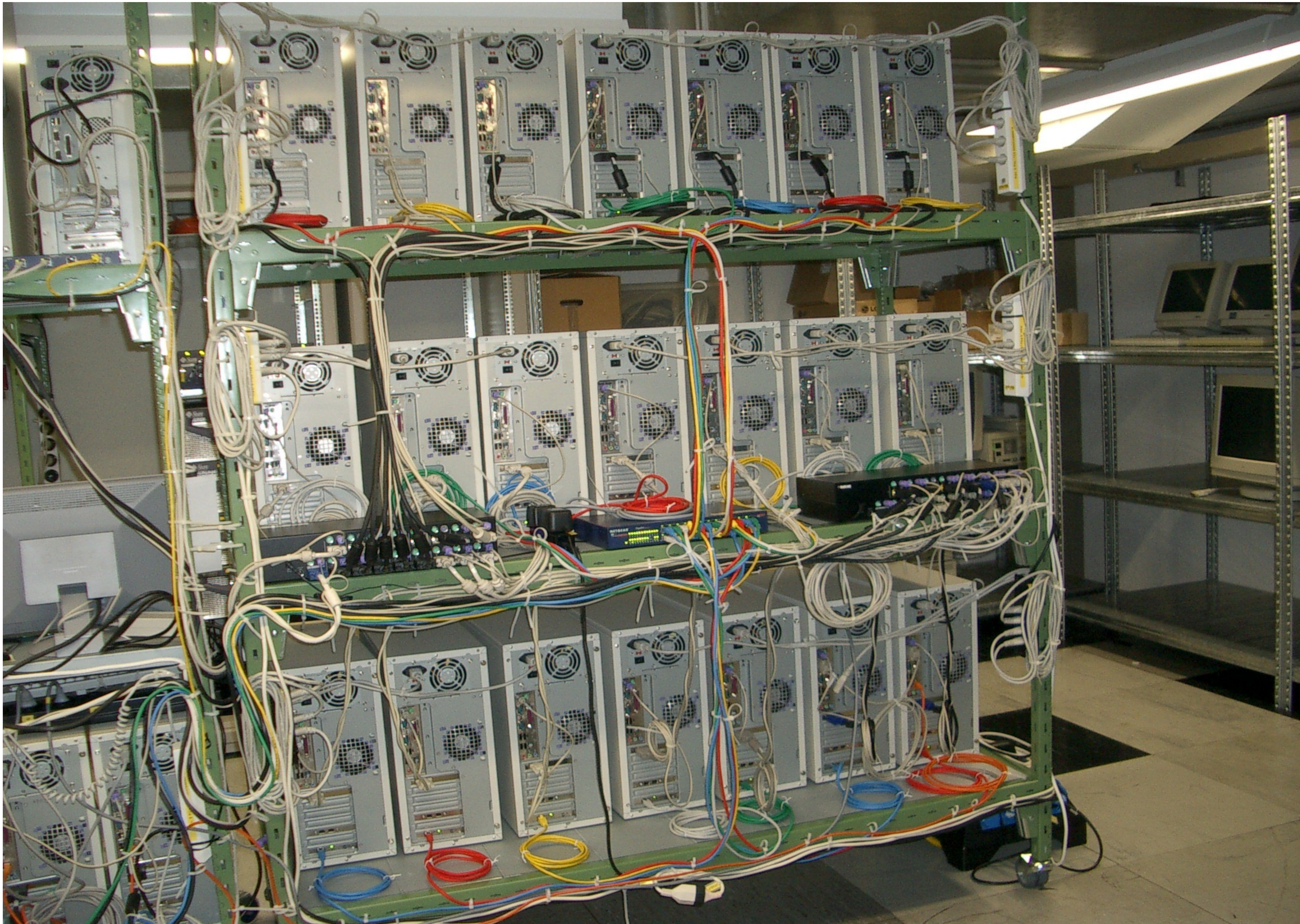
We will keep our attention on learning MPI parallelization

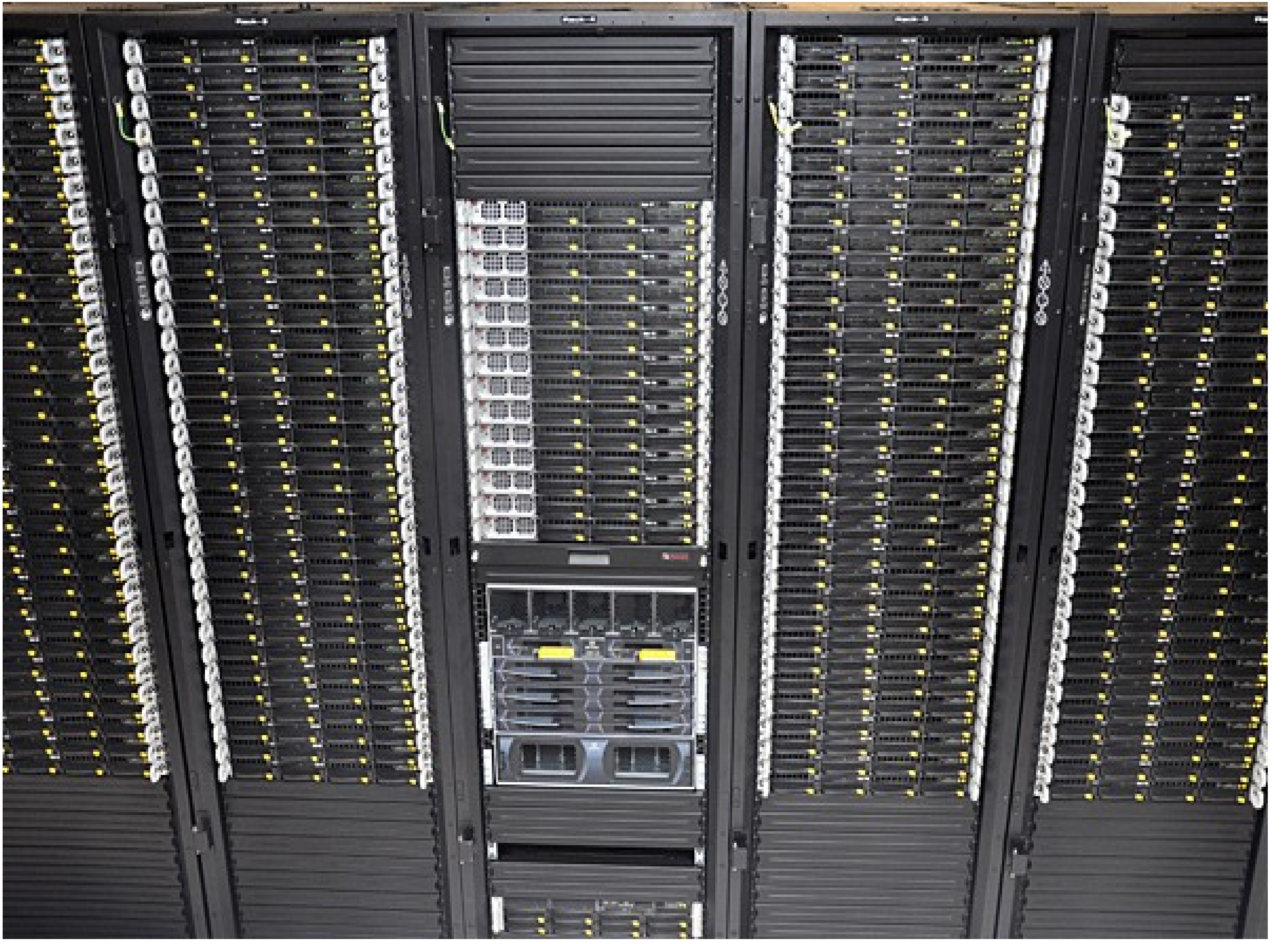
Architecture: **Shared memory**



Architecture: **Distributed memory**







How do we decompose the problem?

The disadvantage of programming for parallel architectures with MPI is that the developer is the one that has to design the program structure and parallelization strategies :

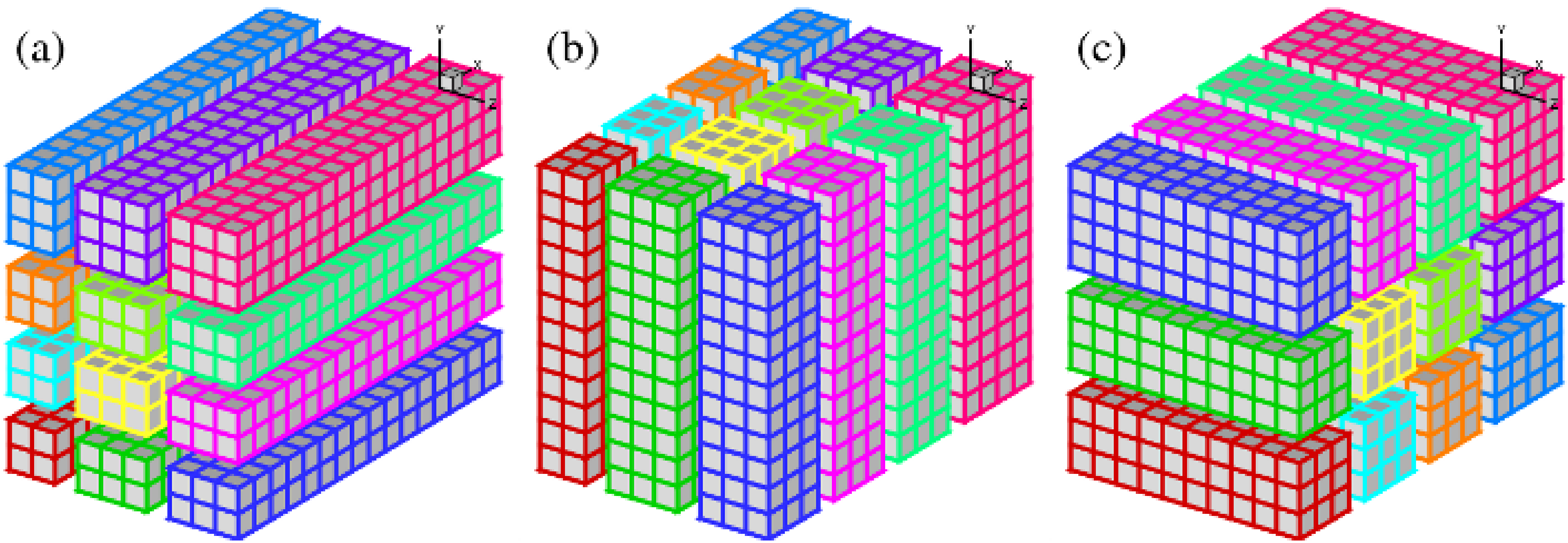
Now, you not only have to think on the computational issues of your scientific problem, you now have to deal with the problem of making all machines work together.

Domain Decomposition

Functional Decomposition

Domain decomposition

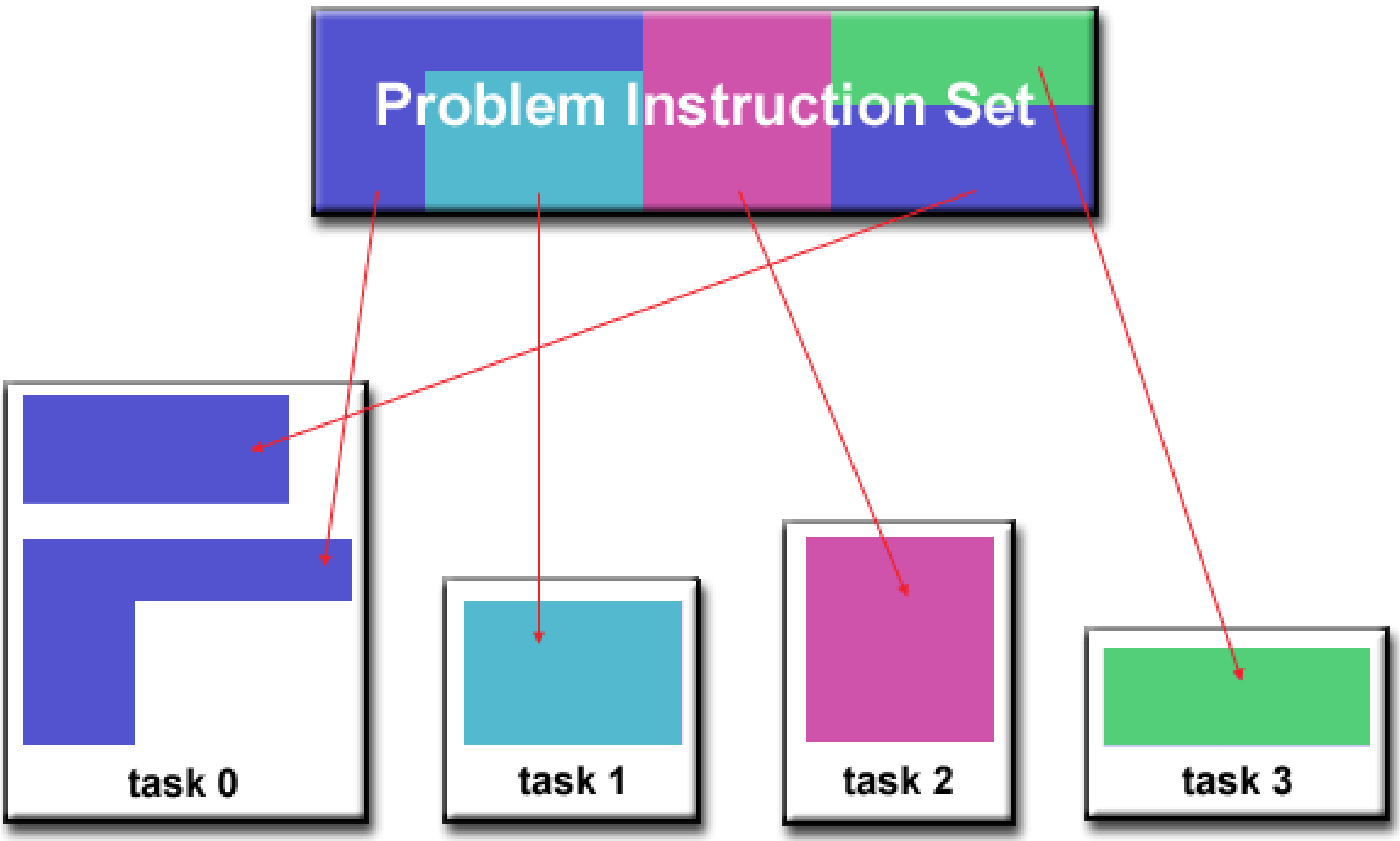
One way to parallelize a problem is based on the idea that you can take the data of your problem and let different instances work on that data



Functional decomposition

Functional decomposition is associated to the decomposition of the program in a set of jobs to be executed by every MPI instance.

This is more easily used when the different instructions are independent, or when the instructions can be executed in different datasets simultaneously.


Problem Instruction Set
task 0
task 1
task 2
task 3

Basic things to be considered

- Try to keep the load balance

Work has to be balanced across processes

Data has to be balanced across processes

- Minimize communication

If moving data from local memory to CPU is time consuming, what do you think about moving data from another CU far away?

- Maximize the simultaneous execution of CPU execution and communication.

Minimizes “lazy” time in the CPU (IDLE). Very difficult to achieve in practice

MPI Message Passing Interface

- Message passing is a model of parallel computing where libraries are used to setup communication across a set of CPU units working together in a job.
- MPI is a standard defining protocols for the implementation of message passing for parallel computing
- There are several implementations of MPI

(LANL, MVAPICH, INTEL, CRAY, ….) all of them follow the same standard.

- MPI **is not a language**, in the end, it is just a library you use on top of your code to implement communication.

Language bindings for MPI 3.0 standard are:
C
FORTRAN

Other bindings may be found in the maket, but not being part of the standard you may not find them available in HPC facilities

Interesting **MPI4Python**
Not in the standard, but still there for use

MPI for Python 3.0.0 documentation » next | modules | index

Table Of Contents
MPI for Python Contents
Next topic
Introduction
This Page
Show Source
Quick search
Go

MPI for Python

Author: Lisandro Dalcin
Contact: dalcinl@gmail.com
Web Site: https://bitbucket.org/mpi4py/mpi4py
Date: Nov 08, 2017

Abstract

This document describes the *MPI for Python* package. *MPI for Python* provides bindings of the *Message Passing Interface* (MPI) standard for the Python programming language, allowing any Python program to exploit multiple processors.

This package is constructed on top of the MPI-1/2/3 specifications and provides an object oriented interface which resembles the MPI-2 C++ bindings. It supports point-to-point (sends, receives) and collective (broadcasts, scatters, gathers) communications of any *picklable* Python object, as well as optimized communications of Python object exposing the single-segment buffer interface (NumPy arrays, builtin bytes/string/array objects)

How do the MPI thing works?

Lets go through by examples!

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

int main(int argc, char** argv) {
  int rank, size;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf("Hello World, I am %d of %d\n", rank, size);

  MPI_Finalize();
  return 0;
}
```

Load the library

New variables for parallelization

Finalize the environment

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

int main(int argc, char** argv) {
  int rank, size;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf("Hello World, I am %d of %d\n", rank, size);

  MPI_Finalize();
  return 0;
}
```

Just for parallelization initialization environment

What is this MPI_COMM_WORLD?

Fortran version

```
program mpitest

use MPI

integer :: rank, size, ierror

call MPI_Init(ierror)
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

write(*,*) 'Hello World, I am ',rank,' of ',size

call MPI_Finalize(ierror)
end
```

The structure is pretty much the same!

- There are as many **independent** copies of the code running as MPI instances

Same code is running in all process!

- The number of MPI instances can be > than number of available CPUs

Performance?

- Is the data the same in all processes?

NO! (see example 1)

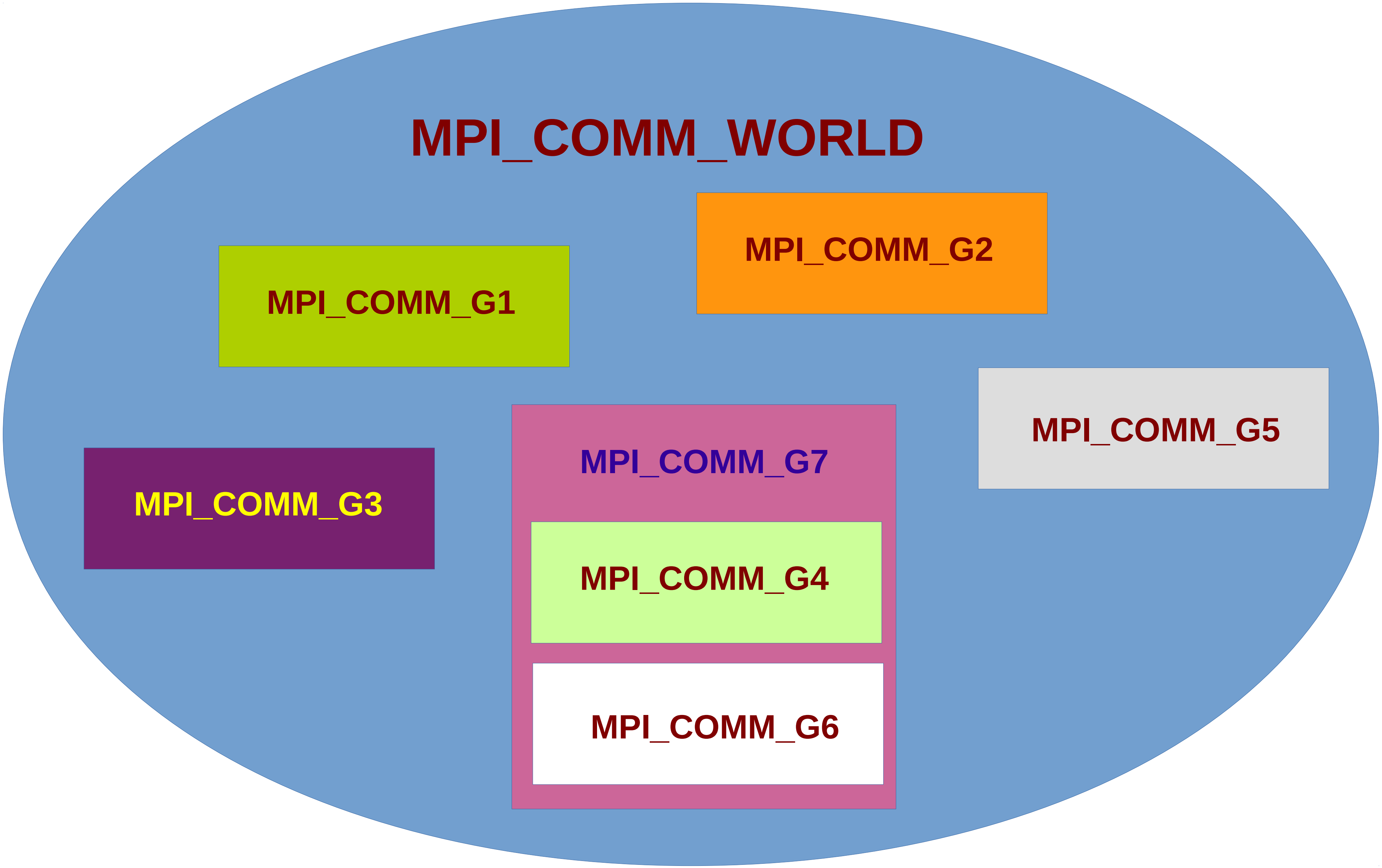
- No shared memory at all!

Even if are actually in a shared memory architecture

That's why you need the communication!!

What is this MPI_COMM_WOLD?

Communication groups



- New variables, new lines for code execution are included

MPI parallelization costs: creation of new variables, execution of more lines of code, ***development!***

- By default synchrony is not expected

(run example 0 several times)

- You have to make sure the code runs with the synchrony you like!

See example 2

- Synchronization issues: one of the challenges during the implementation of MPI parallelization.

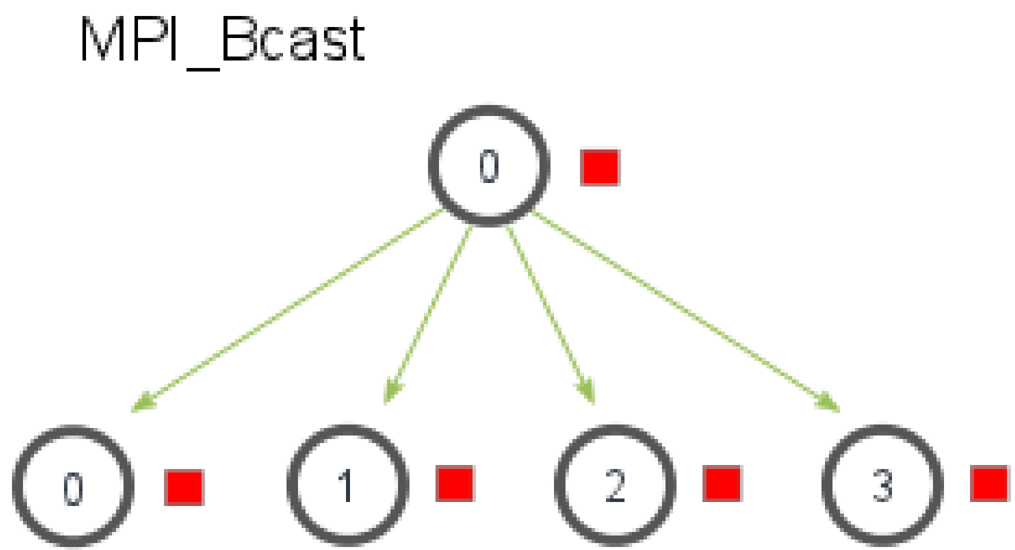
Sync implies waiting…. Reduced efficiency!

Collective communications

Collective communications are associated with the data transfer that involves the full group of processes in a communication group.

- Broadcasting: Sends messages to all processes in a group
- Gather: Gathers data from all processes in the group
- Scatter: Scatter data across all processes in the group
- Reduce: Collects and reduce local data to a “global variable” in the group.

Broadcasting: Sends messages to all processes in a group



MPI_Bcast

Broadcasts a message from the process with rank "root" to all other processes of the communicator

```
int MPI_Bcast(
  void *buffer,
  int count,
  MPI_Datatype datatype,
  int root,
  MPI_Comm comm
);
```

- buffer → Data to be broadcasted
- count → Number of data elements
- datatype → Data type (bits!)
- root → Who is “screaming” the data
- comm → Group

C basic MPI data types

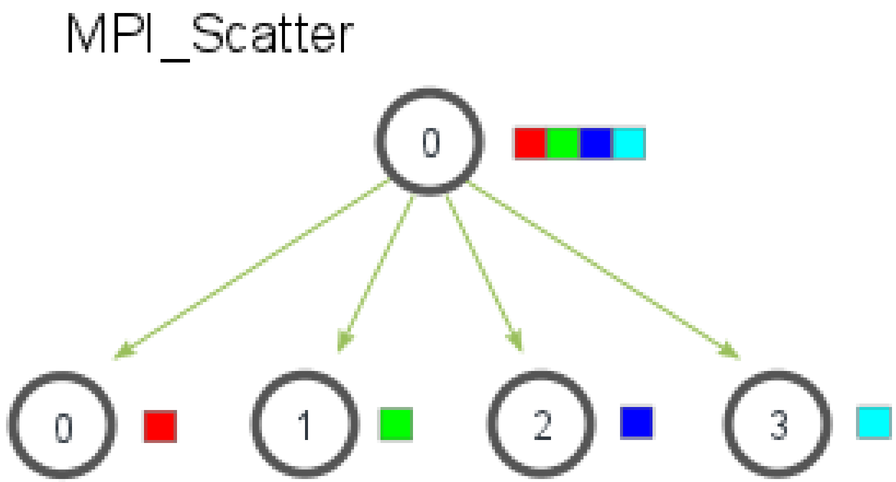
MPI type	C type
MPI_CHAR	signed char
MPI_INT	signed int
MPI_LONG	signed long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	N/A

FORTRAN basic MPI data types

MPI type	Fortran type
MPI_CHAR	CHARACTER(1)
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_BYTE	N/A

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

int main (int argc, char *argv[])
{
  int err;
  int Number_of_Processes, task, i;

  err = MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &Number_of_Processes);
  MPI_Comm_rank(MPI_COMM_WORLD, &task);

  i=0;
  if(task == 0)
    {
      i=10000;
    }

  printf(" >> Hola! before bcast I'm task %d of %d i=%d\n", task, Number_of_Processes,i);
  MPI_Barrier(MPI_COMM_WORLD);

  MPI_Bcast(&i, 1, MPI_INT, 0, MPI_COMM_WORLD);

  printf("Hola! after bcast I'm task %d of %d i=%d\n", task, Number_of_Processes,i);

  err = MPI_Finalize();

  return 0;
}
```

Scatter: Scatter data across all processes in the group



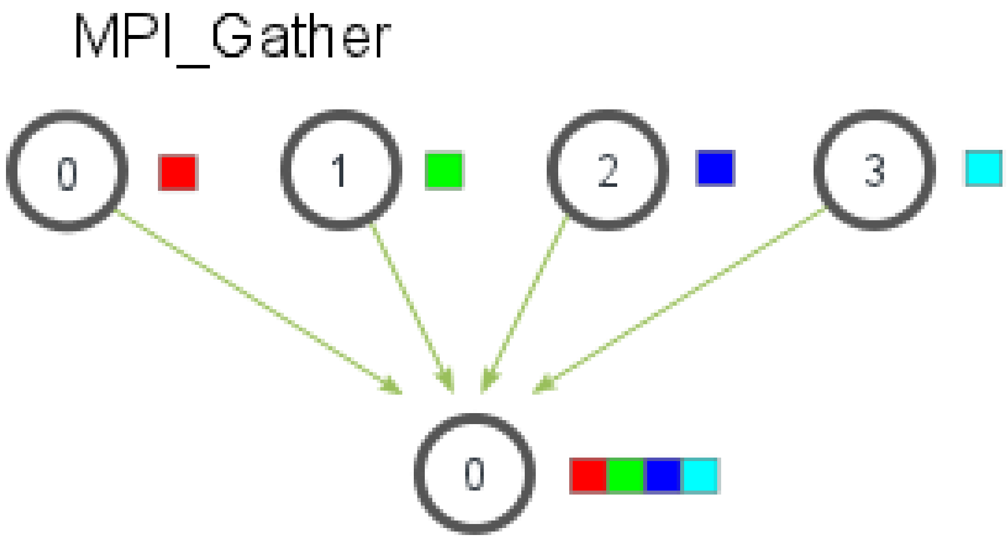
MPI_Scatter

Sends data from one process to all other processes in a communicator

```
int MPI_Scatter(
  void *sendbuf,
  int sendcnt,
  MPI_Datatype sendtype,
  void *recvbuf,
  int recvcnt,
  MPI_Datatype recvtype,
  int root,
  MPI_Comm comm
);
```

- sendbuf → Data to be sent
- sendcnt → # of data elements to send
- recvbuf → Receive buffer
- recvcnt → # of data elements you get
- root → Who was sending the data

Gather: Gathers (collects) data from all processes in a group



MPI_Gather

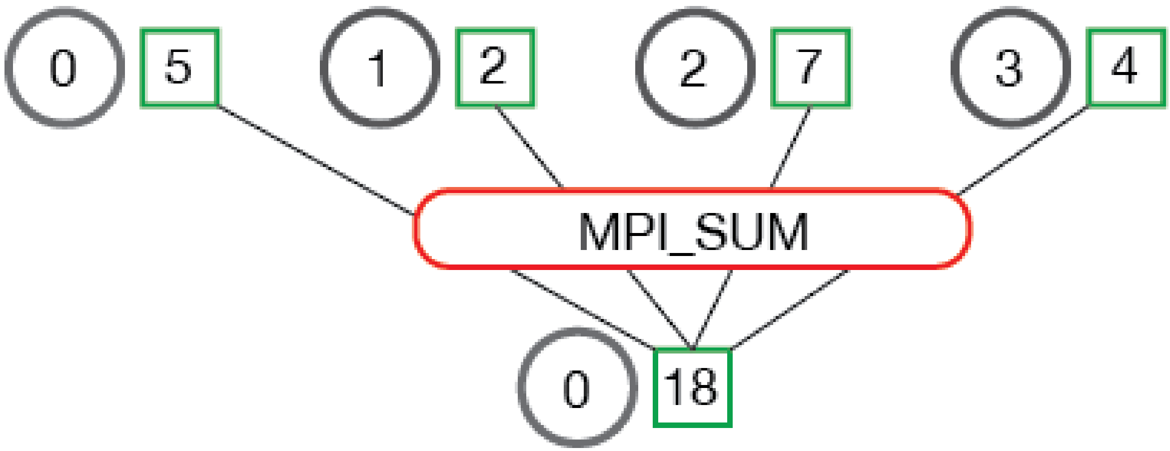
Gathers together values from a group of processes

```
int MPI_Gather(
  void *sendbuf,
  int sendcnt,
  MPI_Datatype sendtype,
  void *recvbuf,
  int recvcnt,
  MPI_Datatype recvtype,
  int root,
  MPI_Comm comm
);
```

- sendbuf → Data to be collected
- sendcnt → # of data elements
- recvbuf → Receive buffer
- recvcnt → # of collected data elements
- root → Who is collecting the data?

```c
k=0;
for(i=task*8; i<(task*8 + 8); i++)
  {
    local_numbers[k] = i;
    k++;
  }

MPI_Barrier(MPI_COMM_WORLD);

for(i=0; i<Number_of_Processes; i++)
  {

    if(i == task)
      {
        for(k=0; k<8; k++)
          printf("task %d input %d\n",task, local_numbers[k]);
      }

    MPI_Barrier(MPI_COMM_WORLD);
  }


MPI_Gather(&local_numbers[0], 8, MPI_INT, &global_numbers[0], 8, MPI_INT, 0, MPI_COMM_WORLD);

if(task == 0)
  {
    for(i=0; i<32; i++)
      printf("Now in task 0 we have %d\n",global_numbers[i]);
  }
```

Reduce: Collects and reduce local data to a “global variable” in the group.



MPI_Reduce

Reduces values on all processes to a single value

```
int MPI_Reduce(
  void *sendbuf,
  void *recvbuf,
  int count,
  MPI_Datatype datatype,
  MPI_Op op,
  int root,
  MPI_Comm comm
);
```

Data to be reduced

Result of the reduction

# of elements to reduce

Reduction operation

MPI_Reduce Operations

MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

```
k=0;
for(i=task*8; i<(task*8 + 8); i++)
  {
    local_numbers[k] = i;
    k++;
  }
MPI_Barrier(MPI_COMM_WORLD);

for(i=0; i<Number_of_Processes; i++)
  {

    if(i == task)
      {
        for(k=0; k<8; k++)
          printf("task %d input %d\n",task, local_numbers[k]);
      }

    MPI_Barrier(MPI_COMM_WORLD);
  }

for(i=0; i<8; i++)
  results[i] = 0;

MPI_Reduce(&local_numbers[0], &results[0], 8, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if(task == 0)
  {
    for(i=0; i<8; i++)
      printf("%d\n",results[i]);
  }
```

Point to Point communication

Point to point communication in MPI is the way we use to make processes in a job talk to each other privately.

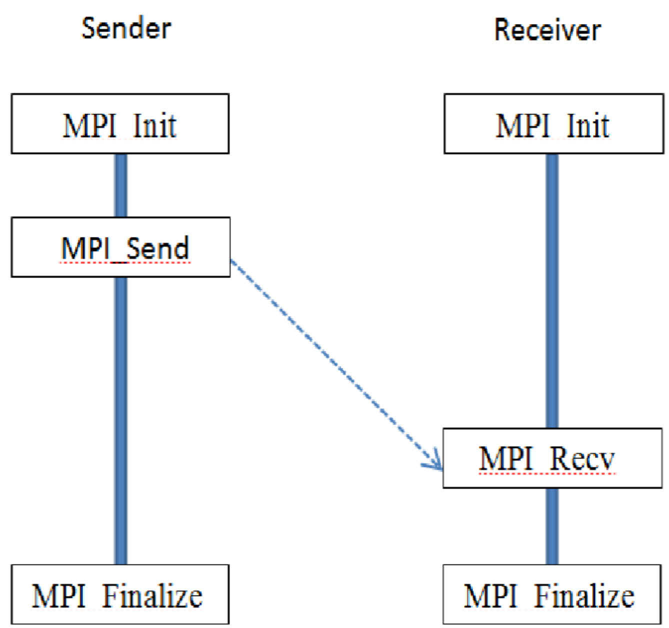
In point communications only two processes are involved.

We will consider first the basic (ans more often used) instructions for point communication

MPI_Send: Sends data to other process in a given communication group

MPI_Recv: Receives a message sent by other process in the communication group

MPI_Send: Sends data to other process in a given communication group



What do you need to send a document?

1) *The document fully documented*
2) *Full address of the receiver*
3) *A sentence of "subject" to indicate the content of the message*

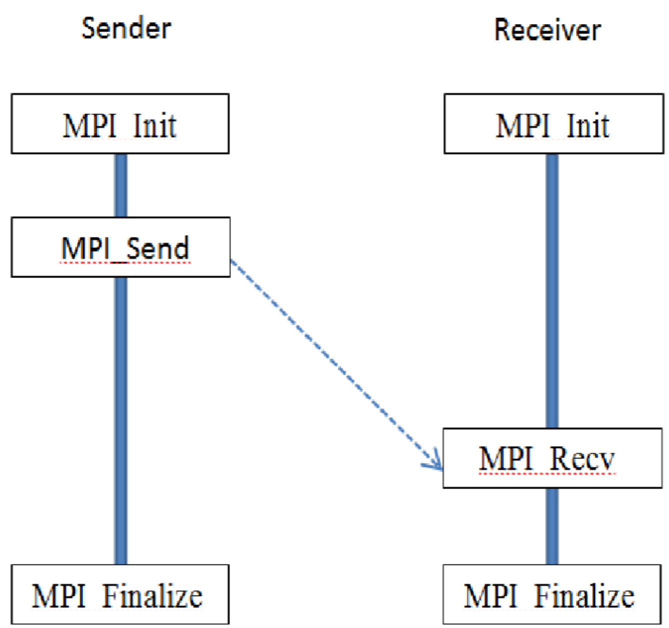
MPI_Send

Performs a blocking send

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

- *buf* → Data to be sent
- *count* → # of data elements
- *datatype* → Data type
- *dest* → ID of receiving task (COMM...)
- *tag* → "Subject" tag

MPI_Recv: Receives a message sent by other process in the communication group



What do you need to receive a document?

1) *A mail box with predefined size*
2) *Do you accept a box from anyone? Sender ID*
3) *A sentence of "subject" to indicate the content of the message*

MPI_Recv

Blocking receive for a message

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

- buf → Memory space to receive the data
- count → # of data elements
- datatype → Data type
- source → ID of sender (COMM...)
- tag → "Subject" tag
- status → Status flag. Received?

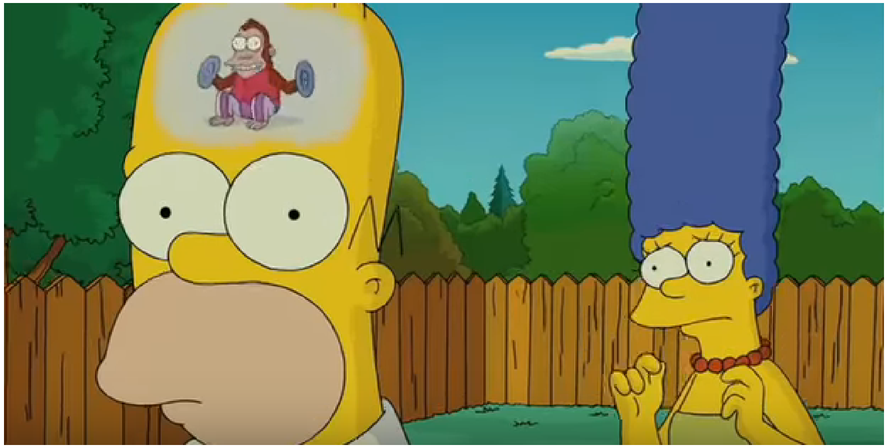
```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main (int argc, char *argv[])
{
  int err, number=0;
  int Number_of_Processes, task, i;
  MPI_Status status;

  err = MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &Number_of_Processes);
  MPI_Comm_rank(MPI_COMM_WORLD, &task);

  if(task == 0)
    {
      number = -1;
      printf("Process 0 sends number %d to process 1\n",number);
      MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
  else if (task == 1)
    {
      MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
      printf("Process 1 received number %d from process 0\n", number);
    }

  err = MPI_Finalize();

}
```

Very simple modification, a more fluent talk!

```
MPI_Comm_size(MPI_COMM_WORLD, &Number_of_Processes);
MPI_Comm_rank(MPI_COMM_WORLD, &task);

if(task == 0)
  {
    number = -1;
    printf("Process 0 sends number %d to process 1\n",number); fflush(stdout);
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

    MPI_Recv(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
    printf("Process 0 received number %d from process 1. That's all!\n", number); fflush(stdout);

  }
else if (task == 1)
  {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    printf("Process 1 received number %d from process 0\n", number); fflush(stdout);

    number=-1000;
    MPI_Send(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  }
```

There are always dysfunctional couples…!

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

int main (int argc, char *argv[])
{
  int err, number=0;
  int Number_of_Processes, task, i;
  MPI_Status status;

  err = MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &Number_of_Processes);
  MPI_Comm_rank(MPI_COMM_WORLD, &task);

  if(task == 0)
    {
      number = -1;
      printf("Process 0 sends number %d to process 1\n",number);
      MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
  else if (task == 1)
    {
      //MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
      printf("Process 1 received number %d from process 0\n", number);
    }

  err = MPI_Finalize();

}
```



If not taken into account, this may have severe consequences!

Lets make two processes to talk fluently...

How do we do to make two processes to play ping-pong?

- Things to be considered

  1) *There are two players*

- Communication:

  task 0 sends to 1. Receives from 1. Just 1 buddy

  Task 1 sends to 0. Receives from 0. Just one buddy

- Every player sends every second move
- Every player receives every second move

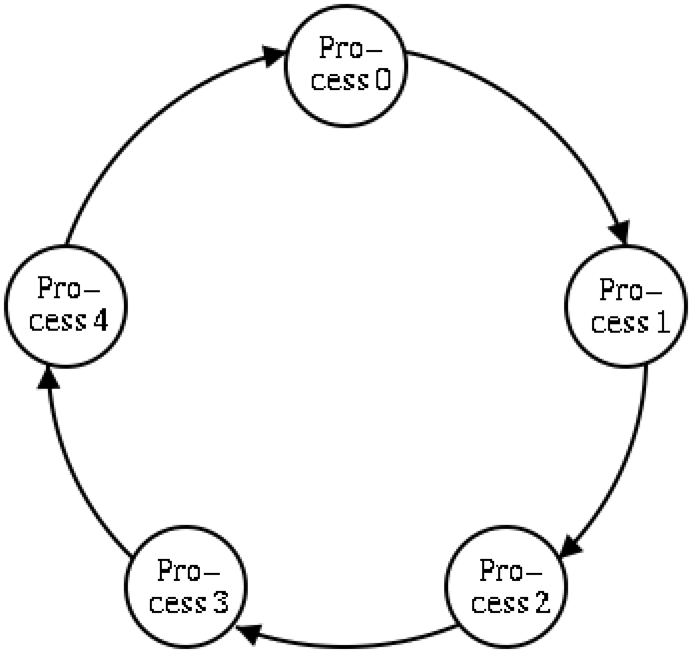
```c
err = MPI_Init(&argc, &argv);

MPI_Comm_size(MPI_COMM_WORLD, &Number_of_Processes);
MPI_Comm_rank(MPI_COMM_WORLD, &task);

partner_task = (task + 1) % 2;
ping_pong_count = 0;

while(ping_pong_count < PING_PONG_LIMIT)
  {

    if(task == (ping_pong_count % 2))
      {
        // Increment the ping pong count before you send it
        ping_pong_count++;
        MPI_Send(&ping_pong_count, 1, MPI_INT, partner_task, 0, MPI_COMM_WORLD);

        printf("%d sent and incremented ping_pong_count " "%d to %d\n",
               task, ping_pong_count, partner_task);
      }
    else
      {

        MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_task, 0, MPI_COMM_WORLD, &status);
        printf("%d received ping_pong_count %d from %d\n",
               task, ping_pong_count, partner_task);
      }

  }
```

0 for task 1

1 for task 0

Every second move there is a change of roles

Now, lets see how do we move data in a ring (topology)



Now, we have a problem that implies the communication of data in a way that it passes through all processes involved in computation.

A ring communication ring

- There are N processes (arbitrary number… as has to be for portability and scalability!)
- Process *i* receives from process *i-1*
- Process *i* sends to process *i+1*
- *Be aware about the “boundaries” of the ring. Processes 0 and N-1*

```c
towhom = (task+1) % Number_of_Processes;
fromwhom = task-1;

if(task == 0)
  fromwhom = Number_of_Processes-1;

if(task != 0)
  {
    MPI_Recv(&value, 1, MPI_INT, task-1, 0, MPI_COMM_WORLD, &status);
    printf("Process %d received dummy %d from process %d\n",
           task, value, task - 1);
  }
else
  {
    // Set the dummy's value if you are process 0
    value = -1;
  }

MPI_Send(&value, 1, MPI_INT, towhom, 0, MPI_COMM_WORLD);

// Now process 0 can receive from the last process.
if(task == 0)
  {
    MPI_Recv(&value, 1, MPI_INT, fromwhom, 0, MPI_COMM_WORLD, &status);
    printf("Process %d received dummy %d from process %d\n",
           task, value, Number_of_Processes-1);
  }
```

(0+1) % 4 = 1
(1+1) % 4 = 2
(2+1) % 4 = 3
(3+1) % 4 = 0

The boundary effect