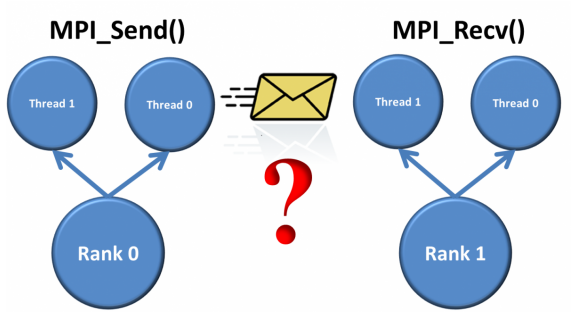
# Parallel programming and a basic introduction to MPI

Cecilia Jarne

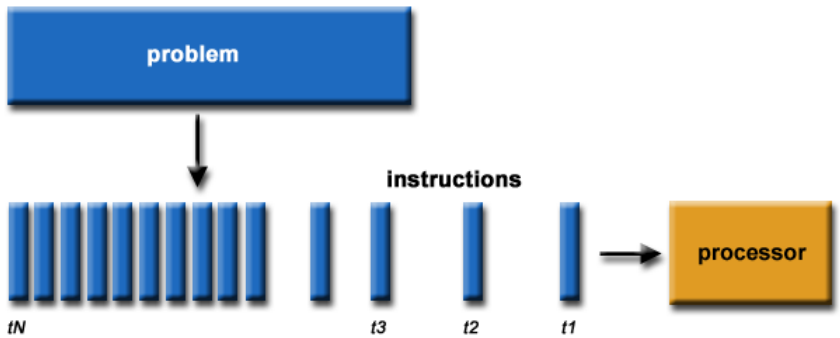
cecilia.jarne@unq.edu.ar



# Serial model

Traditionally, software has been written for serial computation:

- A problem is broken into a discrete series of instructions
- Instructions are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time

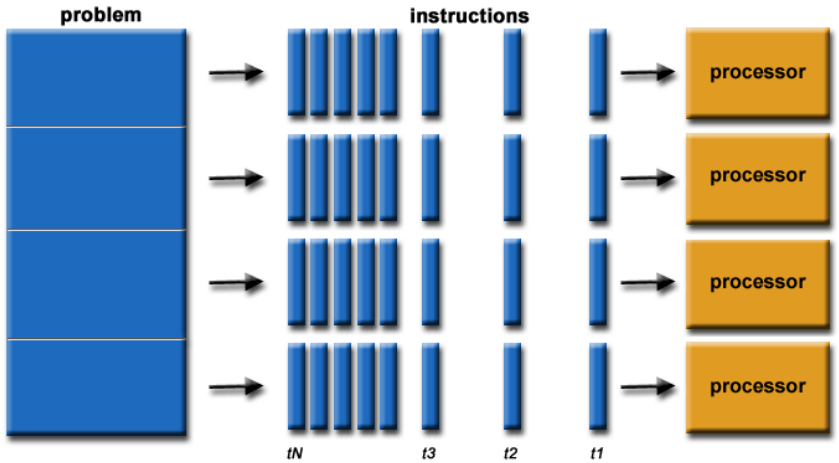



WTPC workshop en técnicas de programación científica

# Parallel Model

**Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem**

- A problem is broken into discrete parts that can be solved concurrently.
- Each part is further broken down to a series of instructions.
- Instructions from each part execute simultaneously on different processors.
- An overall control/coordination mechanism is employed.



# Parallel model

**The computational problem should be able to:**

- Be broken apart into discrete pieces of work that can be solved simultaneously.
- Execute multiple program instructions at any moment in time.
- Be solved in less time with multiple compute resources than with a single compute resource.

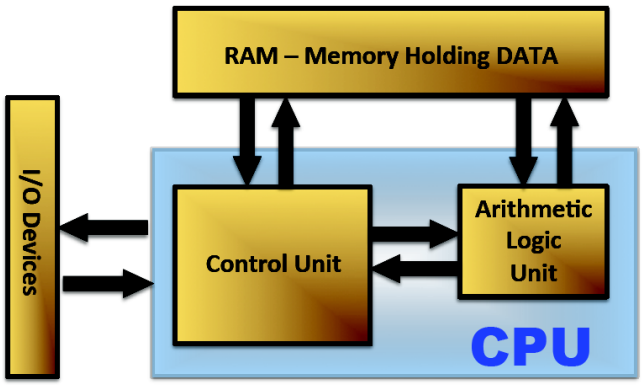
**The compute resources are typically:**

- A single computer with multiple processors/cores.
- An arbitrary number of such computers connected by a network.

WTPC workshop en técnicas de programación científica

# Von Neumann

Four main components:

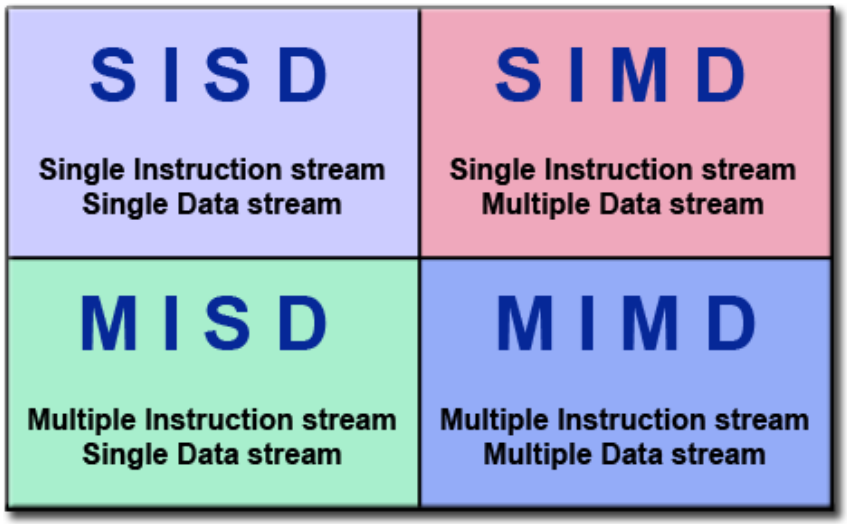



- Memory
- Control Unit
- Arithmetic Logic Unit
- Input/Output

- Read/write, random access memory is used to store both program instructions and data.
- Program instructions are coded data which tell the computer to do something.
- Data is simply information to be used by the program.
- Control unit fetches instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task.
- Arithmetic Unit performs basic arithmetic operations.
- Input/Output is the interface to the human operator.

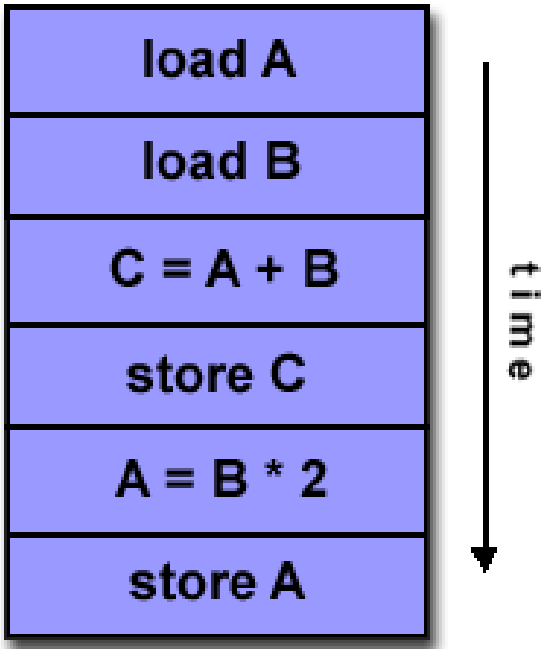
# Parallel model: Concepts and Terminology

Flynn's Classical Taxonomy



# Serial model

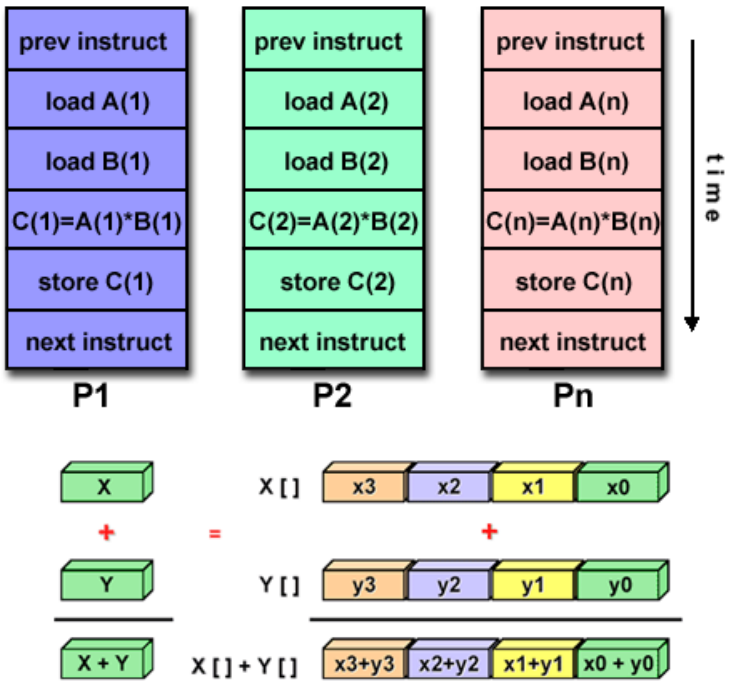
## Single Instruction, Single Data (SISD):




- A serial (non-parallel) computer.
- Single Instruction: Only one instruction by the CPU during any one clock cycle.
- Single Data: Only one data stream is being used during any one clock cycle.
- This is the oldest type of computer.
- Examples: older generation mainframes, minicomputers, workstations and single processor/core PCs.

# Parallel model: SIMD

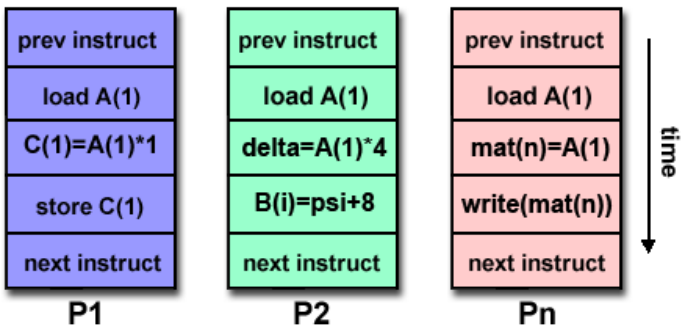
## Single Instruction, Multiple Data (SIMD): A type of parallel computer




- Single Instruction: All processing units execute the same instruction at any given clock cycle.
- Multiple Data: Each processing unit can operate on a different data element.
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- Synchronous (lockstep) and deterministic execution.
- Two varieties: Processor Arrays and Vector Pipelines.

# Parallel model: MISD

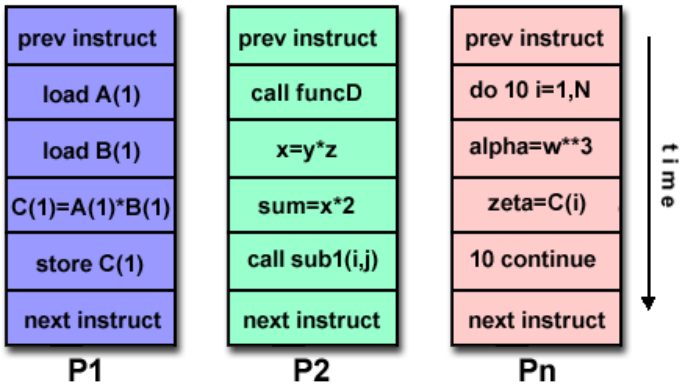
Multiple Instruction, Single Data (MISD) other type of parallel computer:




- Multiple Instruction: Each processing unit operates on the data independently via separate instruction streams.
- Single Data: A single data stream is fed into multiple processing units.
- Few (if any) actual examples of this class of parallel computer have ever existed.

# Parallel model

## Multiple Instruction, Multiple Data (MIMD):




- Multiple Instruction: Every processor may be executing a different instruction stream.
- Multiple Data: Every processor may be working with a different data stream.
- Execution can be synchronous or asynchronous, deterministic or non-deterministic.
- Currently, the most common type of parallel computer - most modern supercomputers fall into this category.
- Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.

WTPC Workshop en técnicas de programación científica

# Parallel model: Concepts and Terminology

- **Node:**
  A standalone çomputer in a box". Usually comprised of multiple CPUs/processors/cores, memory, network interfaces, etc.
- **CPU / Socket / Processor / Core:**
  This varies, depending upon who you talk to. In the past, a CPU (Central Processing Unit) was a singular execution component for a computer. Then, multiple CPUs were incorporated into a node. Then, individual CPUs were subdivided into multiple çores", each being a unique execution unit.
- **Task:**
  Typically a program or program-like set of instructions executed by a processor. A parallel program consists of multiple tasks running on multiple processors.
- **Pipelining:**
  Breaking a task into steps performed by different processor units, with inputs streaming through.

WTPC
workshop en técnicas de programación científica

# Parallel model:Concepts and Terminology

- **Shared Memory:**
  Computer architecture where all processors have direct access to common physical memory. In a programming sense a model where parallel tasks all have the same "picture.of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.
- **Distributed Memory:**
  In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see"local machine memory and must use communications to access memory on other machines where other tasks are executing.
- **Communications:**
  Parallel tasks typically need to exchange data. There are several ways this can be accomplished.
- **Granularity:**

  In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

WTPC workshop en técnicas de programación científica

# Parallel model

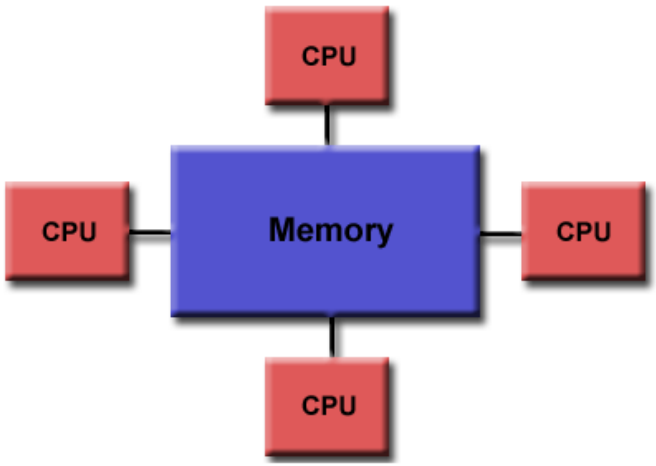
- **Embarrassingly Parallel:**
  Solving many similar, but independent tasks simultaneously; little to no need for coordination between the tasks.
- **Scalability:**
  Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources.
  Factors that contribute to scalability include:
  - Hardware - particularly memory-cpu bandwidths and network communication properties.
  - Application algorithm.
  - Parallel overhead related.
  - Characteristics of your specific application.

WTPC workshop en técnicas de programación científica

# Parallel Computer Memory Architectures

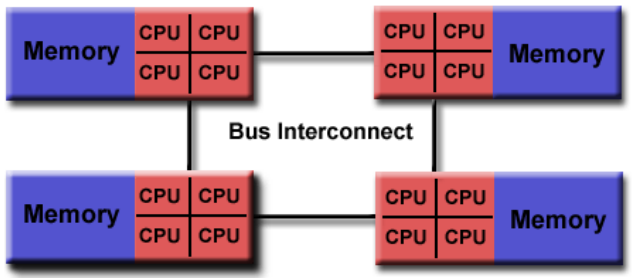
## Shared Memory

Uniform Memory Access (UMA):




- Identical processors.
- Equal access and access times to memory.

Non-Uniform Memory Access (NUMA):




- Not all processors have equal access time to all memories.
- Memory access across link is slower.

# Parallel Programming Models

There are several parallel programming models in common use:

- Shared Memory (without threads)
- Threads
- Distributed Memory / Message Passing
- Data Parallel
- Hybrid
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

WTPC workshop en técnicas de programación científica

# Parallel model: Threads Model

This programming model is a type of shared memory programming. From a programming perspective, threads implementations commonly comprise:

- A library of subroutines that are called from within parallel source code.
- A set of compiler directives embedded in either serial or parallel source code. In both cases, the programmer is responsible for determining the parallelism (although compilers can sometimes help).

# Parallel model: OpenMP


Open spec. for Multi Processing (for shared memory)

- Is not a computer language
- Rather it works in conjunction with existing languages such as standard Fortran or C/C++
- Portable / multi-platform, including Unix and Windows platforms
- Three main components:
  - Compiler directives
  - Runtime library routines
  - Environment variables

WTPC workshop en técnicas de programación científica

# Parallel model:OpenMP

- A directive is a special line of source code with meaning only to certain compilers.
- A directive is distinguished by a sentinel at the start of the line.
- OpenMP sentinels are:

Fortran:

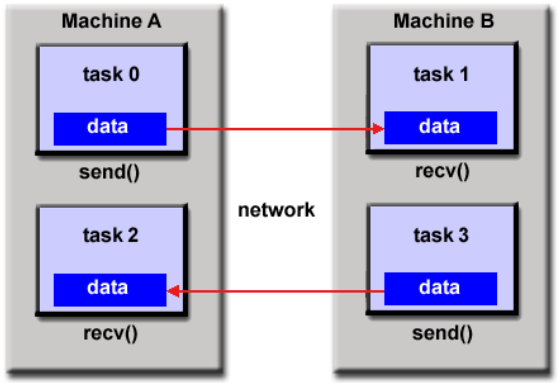
```
1 !$OMP (or C$OMP or *$OMP)
2
```

C/C++:

```
1 #pragma omp
```

# Parallel model: Distributed Memory / Message Passing Model

This model demonstrates the following characteristics:




- A set of tasks that use their own local memory during computation. (Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.)
- Tasks exchange data through communications by sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process.
  (For example, a send operation must have a matching receive operation.)
- **Implementations: Message Passing Interface (MPI)**

WTPC workshop en técnicas de programación científica

# Message Passing Interface (MPI)

MPI's prime goals are:

- To provide source-code portability.
- To allow efficient implementation.

It also offers:

- A great deal of functionality.
- Support for heterogeneous parallel architectures.

# Message Passing Interface (MPI)

Header files
C:

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

Fortran:

```
include 'mpif.h'
```

Python:

```python
from mpi4py import MPI
```

# Initialising MPI:

- MPI controls its own internal data structures. An example with C:

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

- MPI releases 'handles' to allow programmers to refer to these.

# Parallel model(MPI)

Communicator *MPI_COMM_WORLD*


In python:

```
1 comm                  = MPI.COMM_WORLD
```

# MPI: Size

How many processes are contained within a communicator?

```
1 size              = comm.Get_size()
```

# MPI: Rank

How do you identify different processes in a communicator?

```
1 rank              = comm.Get_rank()
```

Indicates the rank of the process that calls it in the range from size-1.

# MPI: Exiting MPI

Must be the last MPI procedure called.

```
1 MPI.Finalize()
```

# MPI: Messages

A message contains a number of elements of some particular datatype.
MPI datatypes:

- Basic types.
- Derived types.
- Derived types can be built up from basic types.
- C types are different from Fortran types.
- Python MPI no need to specify

# MPI Basic Datatypes

For C:

| MPI Datatype | C datatype |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed 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 | |

For FORTRAN:

| MPI Datatype | Fortran Datatype |
|---|---|
| MPI_INTEGER | INTEGER |
| MPI_REAL | REAL |
| MPI_DOUBLE_PRECISION | DOUBLE PRECISION |
| MPI_COMPLEX | COMPLEX |
| MPI_LOGICAL | LOGICAL |
| MPI_CHARACTER | CHARACTER(1) |
| MPI_BYTE | |
| MPI_PACKED | |

WTPC workshop en técnicas de programación científica

# MPI: Point-to-Point Communication

- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator.
- Destination process is identified by its rank in the communicator

# MPI: Sending a message

Python:

```
1 comm.send(data, dest=comm)
```

C:

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

Fortran:

```
1
2 MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG,COMM, IERROR)
3 <type> BUF(*)
4 INTEGER COUNT, DATATYPE, DEST, TAG
5 INTEGER COMM, IERROR
```

WTPC workshop en técnicas de programación científica

# MPI: Receiving a message

Python:

```
1 comm.recv(source=rank_i)
```

C:

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

Fortran:

```
1 MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
2 <type> BUF(*)
3 INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
4 STATUS(MPI_STATUS_SIZE),IERROR
```

WTPC Workshop en técnicas de programación científica

# MPI: Synchronous Blocking Message-Passing

- Processes synchronise.
- Sender process specifies the synchronous mode.
- Blocking both processes wait until the transaction has completed

# MPI: For a communication to succeed...

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message types must match.
- Receiver's buffer must be large enough.

# MPI: Message Order Preservation

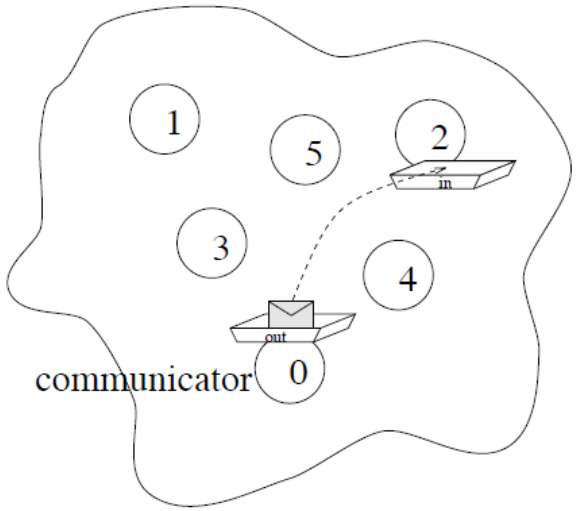
- Messages do not overtake each other.
- This is true even for non-synchronous sends.

# MPI: Non-Blocking Communications

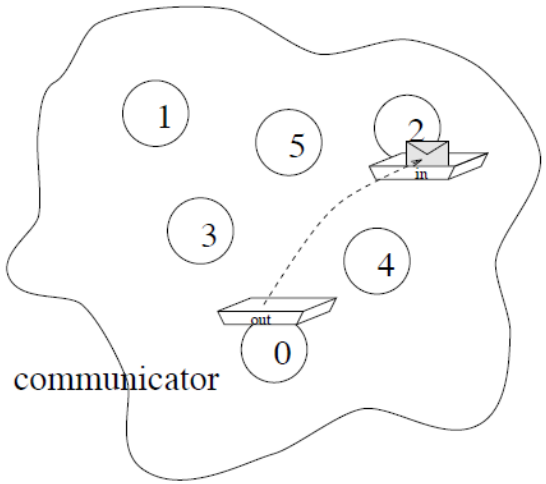
Separate communication into three phases:

- Initiate non-blocking communication.
- Do some work (perhaps involving other communications?)
- Wait for non-blocking communication to complete.

# MPI: Non-Blocking Send

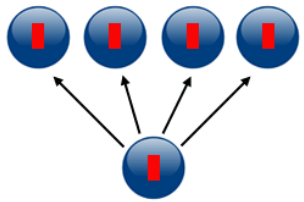


# MPI: Non-Blocking Receive

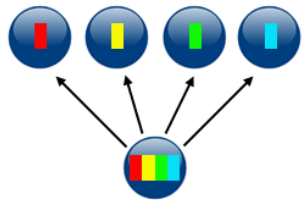


# MPI: Collective Communication

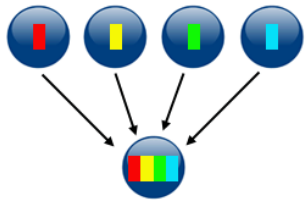
- Communications involving a group of processes.
- Called by all processes in a communicator.



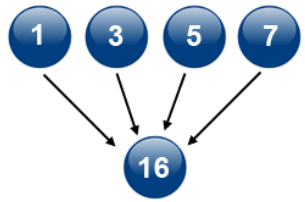
broadcast



scatter



gather




reduction

# MPI: Characteristics of Collective Comms

- Collective action over a communicator.
- All processes must communicate.
- Synchronization may or may not occur.
- All collective operations are blocking.
- No tags.
- Receive buffers must be exactly the right size.

# MPI: Barrier Synchronisation

Python:

```
MPI_Barrier(MPI_Comm)
```

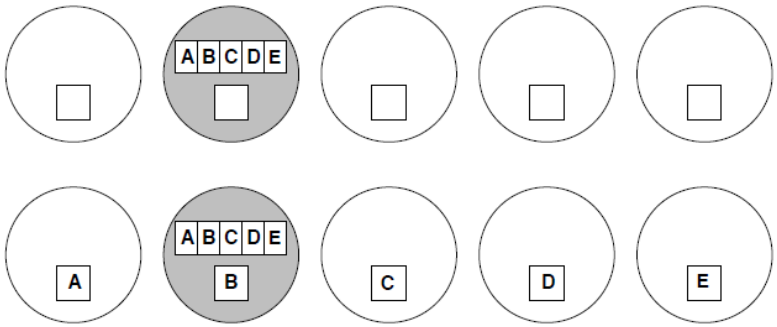
C:

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

Fortran:

```
MPI_BARRIER (COMM, IERROR)
INTEGER COMM, IERROR
```

# MPI:Scatter



# MPI:Scatter

C:

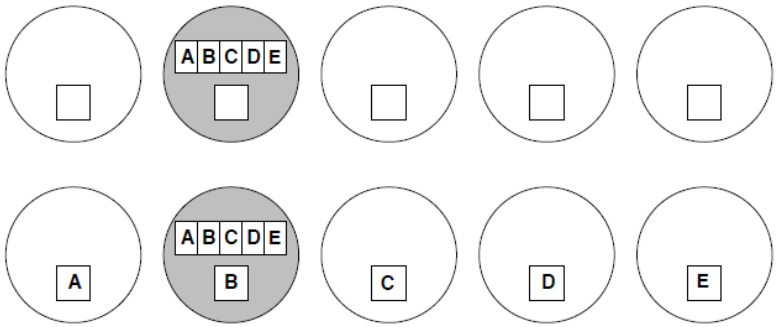
```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
    MPI_Comm comm)
```

Fortran:

```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE,RECVBUF, RECVCOUNT, RECVTYPE,
    ROOT, COMM, IERROR)
<type> SENDBUF, RECVBUF
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT
INTEGER RECVTYPE, ROOT, COMM, IERROR
```

WTPC Workshop en técnicas de programación científica

# MPI:Gather



# MPI:Scatter

C:

```
int MPI_Gather(void *sendbuf, int sendcount,MPI_Datatype sendtype, void
    *recvbuf, int recvcount, MPI_Datatype recvtype,int root, MPI_Comm
    comm)
```

Fortran:

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE,RECVBUF, RECVCOUNT, RECVTYPE,
    ROOT, COMM, IERROR)
<type> SENDBUF, RECVBUF
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT
INTEGER RECVTYPE, ROOT, COMM, IERROR
```

# MPI:Global Reduction Operations

- Used to compute a result involving data distributed over a group of processes.
- Examples:
  global sum or product
  global maximum or minimum
  global user-defined operation

# MPI: Some Predefined Reduction Operations

| MPI Name | Function |
|---|---|
| MPI_MAX | Maximum |
| MPI_MIN | Minimum |
| MPI_SUM | Sum |
| MPI_PROD | Product |
| MPI_LAND | Logical AND |
| MPI_BAND | Bitwise AND |
| MPI_LOR | Logical OR |
| MPI_BOR | Bitwise OR |
| MPI_LXOR | Logical exclusive OR |
| MPI_BXOR | Bitwise exclusive OR |
| MPI_MAXLOC | Maximum and location |
| MPI_MINLOC | Minimum and location |

WTPC workshop en técnicas de programación científica

# MPI: MPI_REDUCE

C:

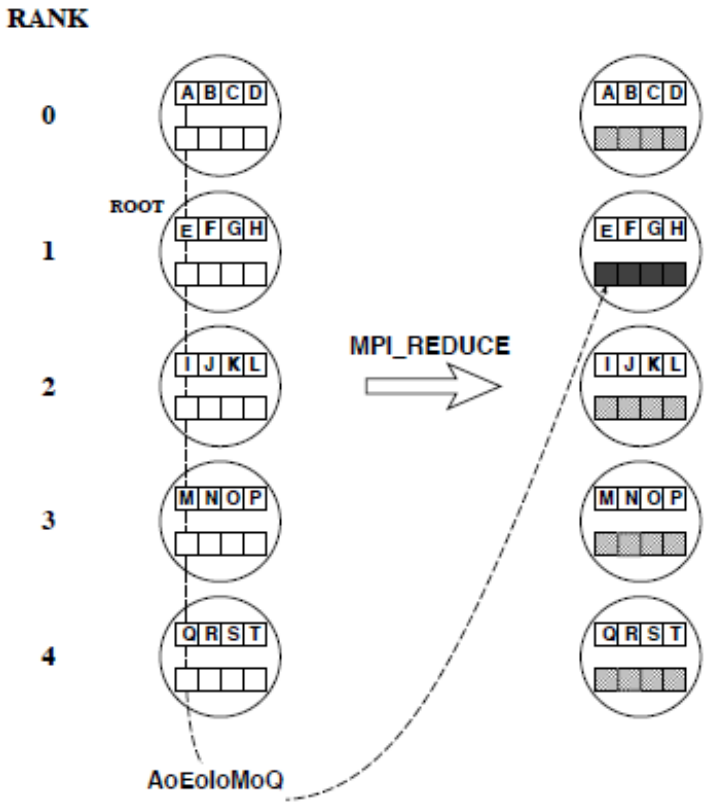
```c
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype
    datatype, MPI_Op op,int root, MPI_Comm comm)
```

Fortran:

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT,DATATYPE, OP, ROOT, COMM, IERROR)
<type> SENDBUF, RECVBUF
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT
INTEGER RECVTYPE, ROOT, COMM, IERROR
```

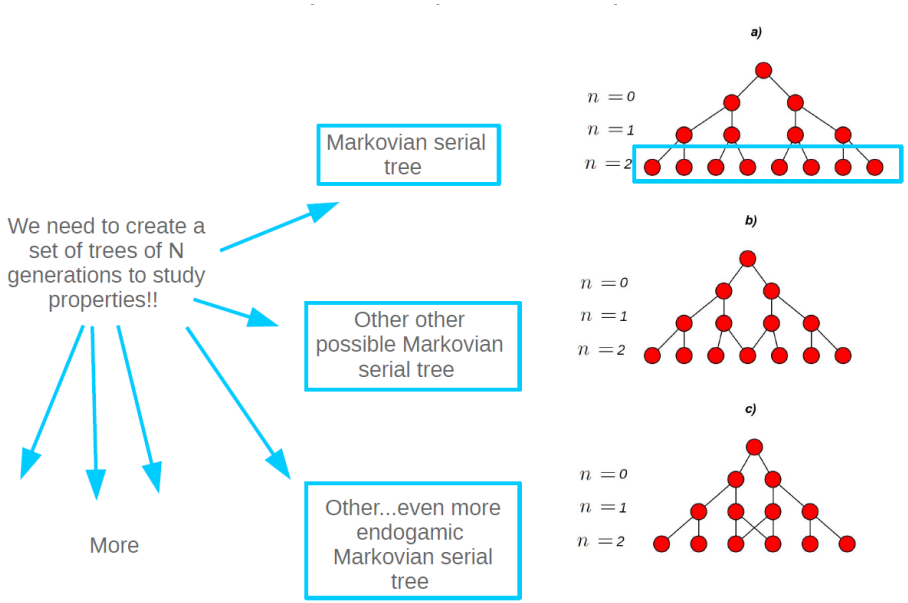
WTPC Workshop en técnicas de programación científica

# MPI: MPI_REDUCE



# One example of code optimization: Tree of ancestors

The problem:




Some steps:

- I installed OpenBLAS library and compile numpy with OpenBLAS integration (not as easy as I expected).
- Then I used MPI for python (I installed the package MPI4py)
- Now the hardest part! I adapted my serial code to used MPI4py.

# One example of code optimization

```
1
2  import numpy as np
3  from get_tree import *
4  import scipy
5  import time
6  from itertools import permutations
7  from random import sample
8  from mpi4py import MPI
9
10 start_time = time.time()
11
12 Nlines    = 200
13 color_lvl = 8
14 rgb = np.array(list(permutations(range(0,256,
       color_lvl),3)))/255.0
15 colors = sample(rgb,Nlines)
```

```
1
2  N               = 20   # track x gen back
3  generation_set  = []
4  ances_solo      = []
5  ances_par_impar = []
6  ances_mariano   = []
7  trees_set       = []
8
9  generation = np.arange(0,N,1)
10 trees      = 50 #number of trees per core
11 generation.tolist()
12 bins       = np.arange(0, N, 1)
13 print 'bins', bins
14 width      = 1.0
```

```
comm           = MPI.COMM_WORLD
rank           = comm.Get_rank()
size           = comm.Get_size()

if rank == 0:
  #print 'rank',rank
  ances_solo_0     = []
  ances_par_0      = []
  ances_mariano_0  = []
  trees_set_0      = []
  generation_set_0 = []

  for kk in np.arange(0,trees,1):
    j = get_tree(N,kk)
    generation_set_0.append(j[0])
    ances_solo_0.append(j[1])
    ances_par_impar.append(j[2])
    ances_mariano_0.append(j[3])
    trees_set_0.append(np.c_[j[1],j[0]])
    print"g:\n", j[0],"an:\n",j[1]
```

```
  for j, pepe in enumerate(generation_set_0):
    print 'j',j
    if len(generation_set_0[j])<N:
      agrego= N-len(generation_set_0[j])
      for i in np.arange(agrego):
        generation_set_0[j].append(len(
      generation_set_0[j])+i)
        ances_solo_0[j].append(0)
        ances_par_impar[j].append(0)
        ances_mariano_0[j].append(0)
    else:
        print 'not add'
  ances_solo.extend(ances_solo_0)
  for rank_i in np.arange(1,size,1):
      pepe=comm.recv(source=rank_i)
      ances_solo.extend(pepe)
      print 'recived from:1'
```

```
if rank != 0: #no for needed in each core!
   ances_solo_1     = []
   ances_par_1      = []
   ances_mariano_1  = []
   trees_set_1      = []
   generation_set_1 = []
   print 'rank',rank
   for kk in np.arange(rank*trees,(rank+1)*trees,1):
     j = get_tree(N,kk)
     generation_set_1.append(j[0])
     ances_solo_1.append(j[1])
     ances_par_impar.append(j[2])
     ances_mariano_1.append(j[3])
     trees_set_1.append( np.c_[j[1],j[0]])
     print "gen:\n", j[0],"an:\n", j[1]
    comm.send(ances_solo_1, dest=0)
```

```

 for j, pepe in enumerate(generation_set_1):
  if len(generation_set_1[j])<N:
   agrego= N-len(generation_set_1[j])
   for i in np.arange(agrego):
    generation_set_1[j].append(len(generation_set_1[
      j])+i)
    ances_solo_1[j].append(0)
    ances_par_impar[j].append(0)
    ances_mariano_1[j].append(0)
  else:
   print 'no agrego'
MPI.Finalize()
tam =len(ances_solo)
print'size:',tam,'from rank:',rank
```

```
1  if rank == 0:
2   media      = np.average(ances_solo, axis=0)
3   uncertanty = np.std(ances_solo, axis=0)*float(1)/
        float(trees)
4   print 'mean:',media,'uncer: ',uncertanty
5   valor_pedido_mariano = []
6   valor_pedido_mariano.append(2)
7   valor_pedido_mariano.append(4)
8   f_out = open('ances.txt', 'w')
9   xxx   = np.c_[bins,media,uncertanty]
10  np.savetxt(f_out,xxx,fmt='%f  %f  %f',delimiter='\t
        ',header="f  #gen  #ances #error")
11  print'gen: ',bins, 'anc: ',media
12  print("- %s seconds -" % (time.time() - start_time
        ))
```

```
mpiexec -n N python
    set_of_tree_par_v4.0.py
```

Where N is core number:

Standar result:

Set size: 100

20.6414310932 seconds

Parallel result: (with 2 cores)

Set size: 100

— 13.1909019947 seconds —

WTPC Workshop en técnicas de programación científica

# Credits:

- Blaise Barney, Lawrence Livermore National Laboratory
  https://computing.llnl.gov/tutorials/parallel_comp/
- ICTP Introductory School on Parallel Programming and Parallel Architecture for High-Performance Computing:
  http://indico.ictp.it/event/7659/overview
- WTPC 2017, Graciela Molina (FACET -UNT).
  https://wtpc.github.io/clases/2017/11_MPI.pdf
- MPI4PY:
  https://www.howtoforge.com/tutorial/distributed-parallel-programming-python-mpi4py/
- http://mpi-forum.org/docs/