Introduction to Parallel Programming

Giovanni Erbacci - g.erbacci@cineca.it
CINECA - Supercomputing, Applications & Innovation Department

CINECA

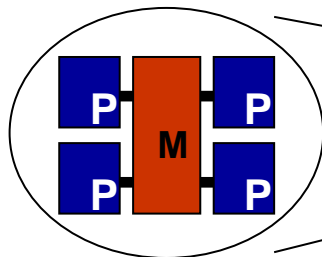# Outline
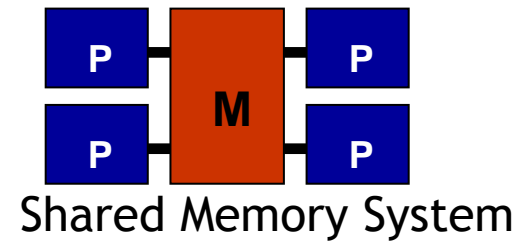
Parallel programming

Shared memory paradigm

Distributed memory paradigm

Processes and Threads

Local or Global Addressing

SPMD Philosophy

Models of parallelism

    Data parallelism

    Control parallelism

Load balancing

Critical sections and Mutual exclusion
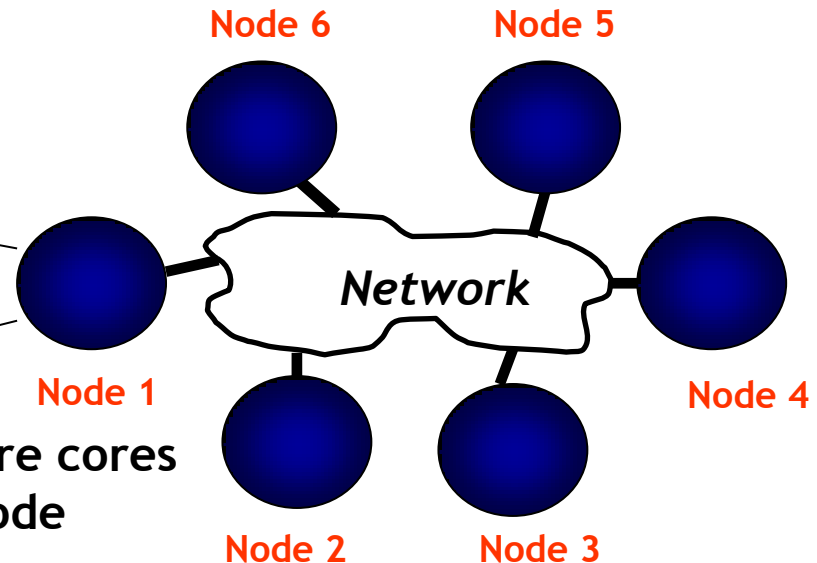
Deadlock

Parallel computers

A **parallel computer** is a system consisting of a collection of processors able to communicate and cooperate to solve large computational problems quickly.

Distribuited Memory System

Shared Memory System
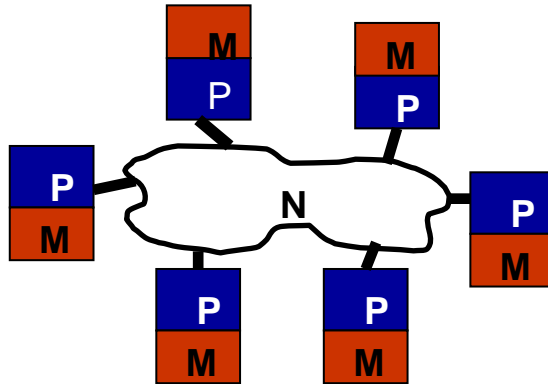
Distributed Memory system with more cores sharing the memory of the single node

Parallel Programming

Parallel programming is a programming technique that involves the use of multiple processors working together on a single problem

The global problem is split in different sub-problems, each of which is performed by a different processor in parallel.

Parallel Program
program composed from different tasks that communicate with each other to achieve an overall computational target.

To realize and execute a parallel program is requires:

- A programming language that allows to formally describe non-sequential algorithms
- A non-sequential computer able to perform any number of tasks simultaneously.

Parallel Programming paradigms

A **programming model** is a collection of program abstractions that provides a simplified and transparent vision of the hardware and software system in its entirety.

Communication in a parallel computer is possible according to these patterns:
- **Shared memory**: by accessing shared variables
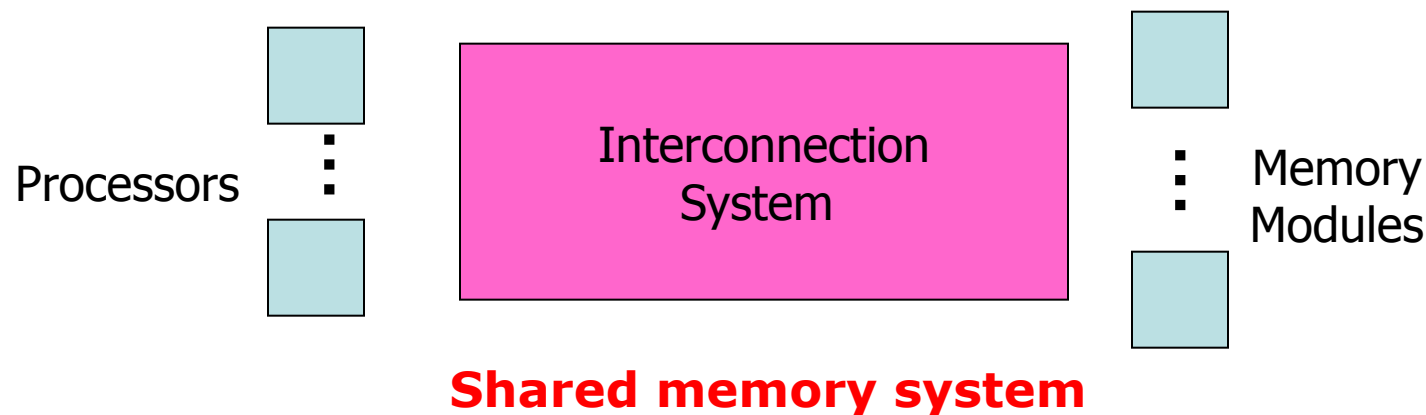- **Message-passing**: exchanging messages

These patterns identify two parallel programming paradigms:
- **Shared memory** or global environment paradigm
  where processes interact exclusively working on common resources

- **Message passing** or local environment paradigml
  where there are no shared resources, processes handle only local information and the only way to interact is by exchange of messages (message passing)

Shared Memory Paradigm

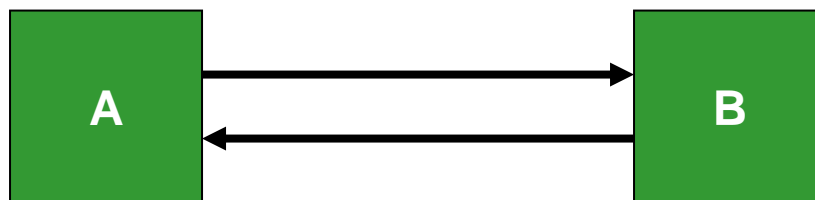Processes communicate by accessing shared variables and shared data structures.

Processors

Interconnection System

Memory Modules

Shared memory system

Basic shared memory primitives:

– Read from a shared variable

– Write on a shared variable

Message Passing Paradigm

Tasks communicate by exchanging messages

Basic message passing Primitives:

–Send (*parameter list*)

–Receive (*parameter list*)

What is a Process

Algorithm

identify the sequence of logical steps that must be followed to solve a given problem.

Program

implementation of the algorithm, by means of a suitable formalism (programming language) so that it can be executed on a specific computer.

Sequential process

sequence of events (execution of operations) which gives place the computer when operates under the control of a particular program.
Abstract entity which identifies the activity of the computer on the program execution.

Process

A process is created by the operating system, and requires a fair amount of "overhead".

Processes contain information about program resources and program execution state, including:

- Process ID, process group ID, user ID, and group ID
- Environment
- Working directory.
- Program instructions
- Registers
- Stack
- Heap
- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools
  (such as message queues, pipes, semaphores, or shared memory).

Thread

A thread is defined as an **independent stream of instructions** that can be scheduled to run as such by the operating system.

Threads
- exist within the process and use the process resources
- are able to be scheduled by the operating system
- run as independent entities
- they duplicate only the bare essential resources that enable them to exist as executable code.

This independent flow of control is accomplished because a thread maintains its own:
- Stack pointer
- Registers
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Thread specific data.

Thread /1

Threads may share the process resources with other threads that act equally independently (and dependently)

Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

Thread die if the parent process dies

Thread Iis "lightweight" because most of the overhead has already been accomplished through the creation of its process.
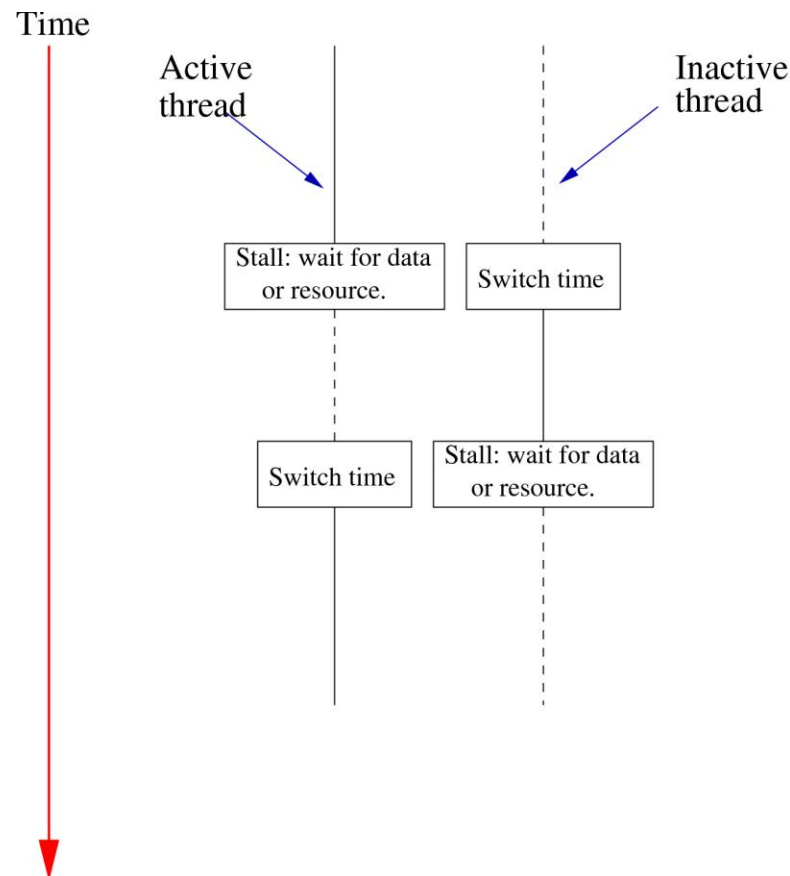
Available in almost all main processor families.

Specific hardware support on some processors

However, care must be taken in using **automatic multi-threading**: can, in some case, slow down applications.

Time

Active thread

Inactive thread

Stall: wait for data or resource.

Switch time

Switch time

Stall: wait for data or resource.

Toward a parallel algorithm

Bubble Sort

Input: a sequence of n numbers $<a_1, a_2, a_3, \ldots a_n>$

Output: a permutation $<a'_1, a'_2, a'_3, \ldots a'_n>$ of the elements such that $a'_1 \leq a'_2, \leq a'_3 \leq \ldots \leq a'_n$

```
/* Bubble sort for integers */
#define SWAP(a,b) { int t; t=a; a=b; b=t; }
void SORT( int a[], int n )
/* Pre-condition: a contains n items to be sorted */
    {
    int i, j;
    /* Make n passes through the array */
    for(i=0;i<n;i++)
        {
        /* From the first element to the end of the unsorted section */
        for(j=1;j<(n-i);j++)
            {
            /* If adjacent items are out of order, swap them */
            if( a[j-1]>a[j] ) SWAP(a[j-1],a[j]);
            }
        }
    }
```

Sort of n numbers

Idea: - Split the array to sort into two array of n / 2 elements each,
        - Order the two array separately
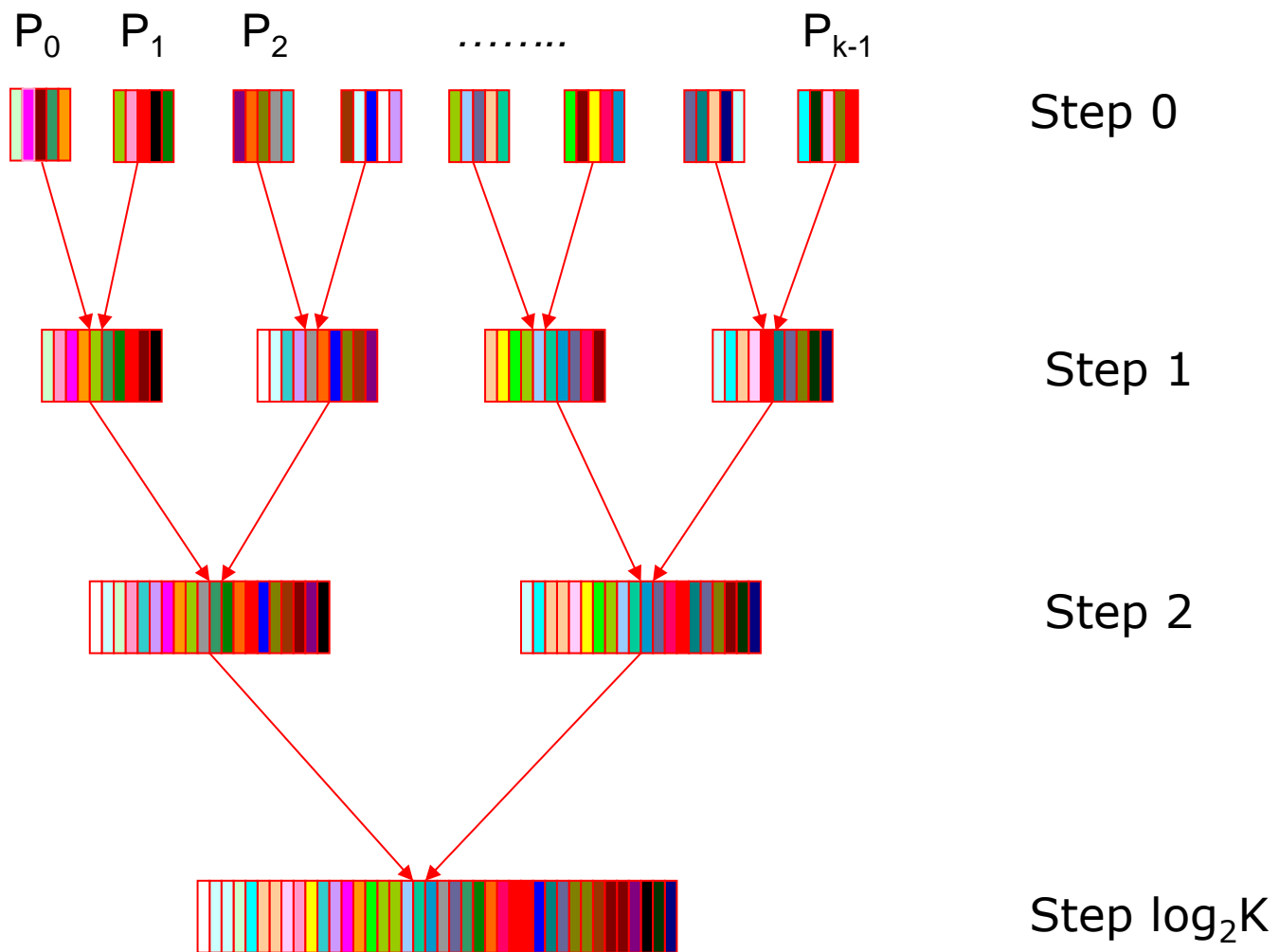   - Merge the two ordered arrays to reconstruct the whole array

```
SORT(a[0 : n/2-1])

SORT(a[n/2 : n-1])

MERGE(a[0 : n/2-1], a[n/2 : n-1])
```

Sort on k Processors

P_0 P_1 P_2 …….. P_{k-1}

Step 0

Step 1

Step 2

Step $\log_2 K$

Problem

```
Read the data to sort

    SORT(a[0 : n/2-1])

    SORT(a[n/2 : n-1])

    MERGE(a[0 : n/2-1], a[n/2 : n-1])

Print the ordered array
```

Parallel programming requires to address problems that do not occur with sequential programming.
We need to decide:
  · what are the parts of the code which form the parallel sections
  · when to start the execution of different parallel sections
  · when to end the execution of parallel sections
  · when and how to make the communication between the parallel entities
  . when make the synchronization between the parallel entities

Then we need the right tools to implement all this

Local or Global addressing

With the shared memory paradigm we rely on the global memory addressing

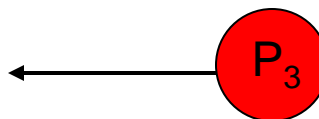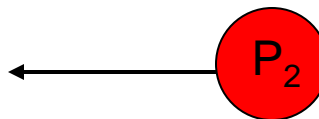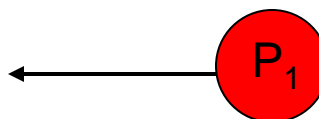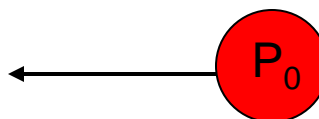With the distributed memory paradigm we rely only on local memories and so we can only handle a local address space.
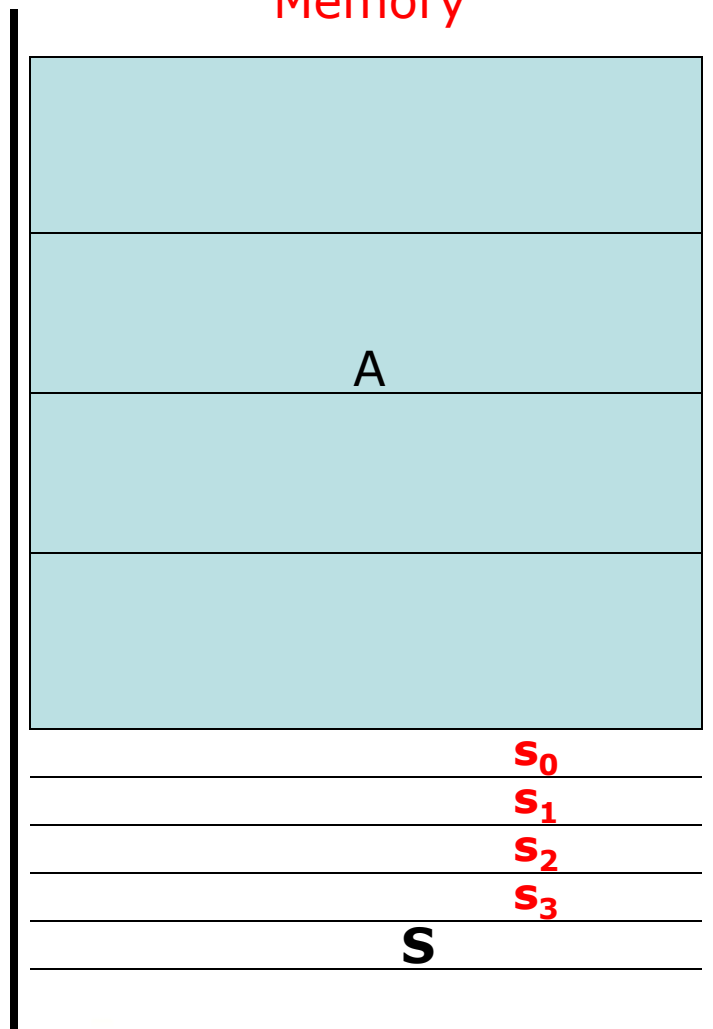
Example: Compute the sum of the elements of
array **A[n, n]**

$$s = \sum_i \sum_j a_{ij}$$

Global Addressing

Memory

A

s_0
s_1
s_2
s_3
S

P_0

P_1
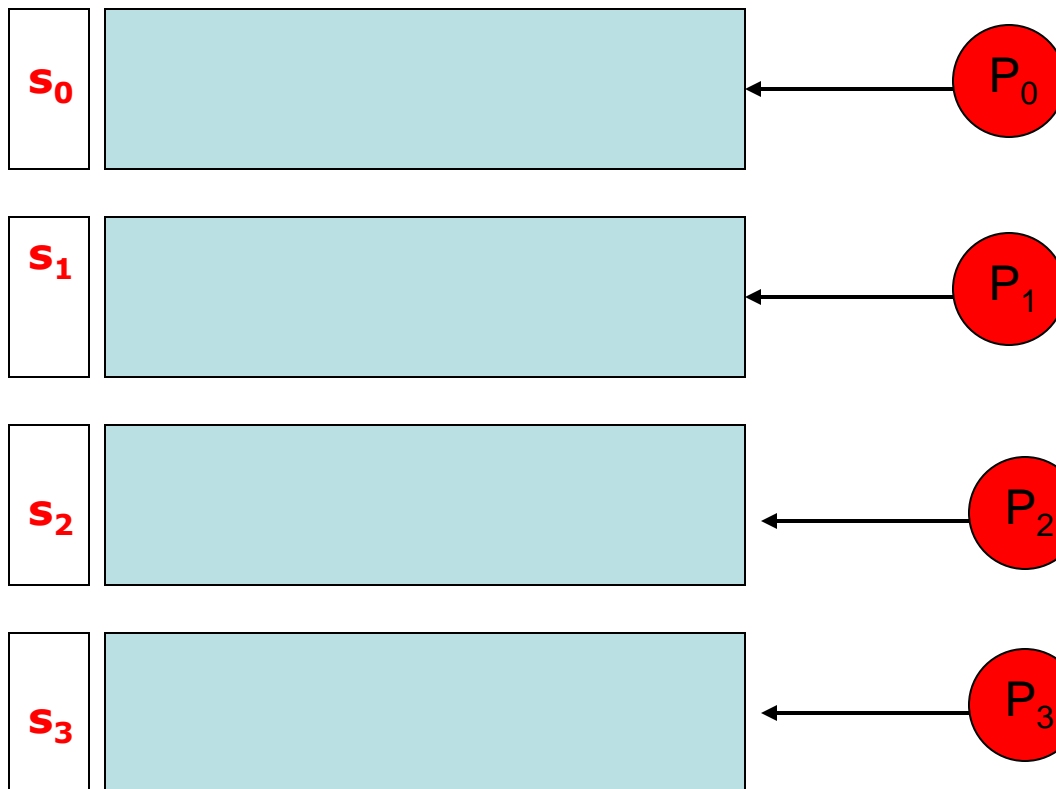
P_2

P_3

Array **A[n, n]** is allocated in the shared memory.

All the processors involved in the computation can reference any element of **A**.

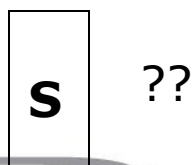A(i,j)   i = 1, n, j = 1,m

Temp = A(857,760) + A(321, 251)

Local Addressing

s_0

P_0

Each processor can reference only his own local memory.

s_1

P_1

A slice of **A[n, n]** is allocate in each local memory

s_2

P_2

$A(i,j)$ $i = 1, n/k,$ $j = 1,m$

s_3

P_3

S   ??

Temp = A(857,760) + A(321, 251) $\rightarrow$

$\rightarrow$ Temp = A(107,760)P_3 + A(71, 251) P_1

(N = 1000, 4 proc)

Master Slave and SPMD philosophy

Master / Slave

A single process (the master) controls the work done by other processes (slaves, workers). These can run the same program or different programs

Single Program Multiple Data

Each process runs the same copy of the program
The execution flow of each proces varies as a function of the local environment (data, number of process, etc..)
We can emulate the master / slave philosophy

```c
C
main (int argc, char **argv)
{
    if (process is to become a controller process)
    {
        Controller (/* Arguments /*);
    }
    else
    {
        Worker (/* Arguments /*);
    }
}
```

```fortran
Fortran
PROGRAM
IF (process is to become a controller process) THEN
        CALL Controller (/* Arguments /*)
ELSE
        CALL Worker (/* Arguments /*)
ENDIF
END
```

Implementing Parallel Programming Paradigms

- Shared Memory Paradigm (Open-MP)
- Message Passing Paradigm (PVM, MPI)

Sequential procedural languages (Fortran 90,C,C++) + API (Compiler Directives)
It tends to favor an implicit parallelism

      - Parallelism is not visible to the programmer

      - Compiler responsible for parallelism

      - Easy to do

      - Small improvements in performance

Sequential procedural languages (Fortran 90, C, C++) + API (Library routines)
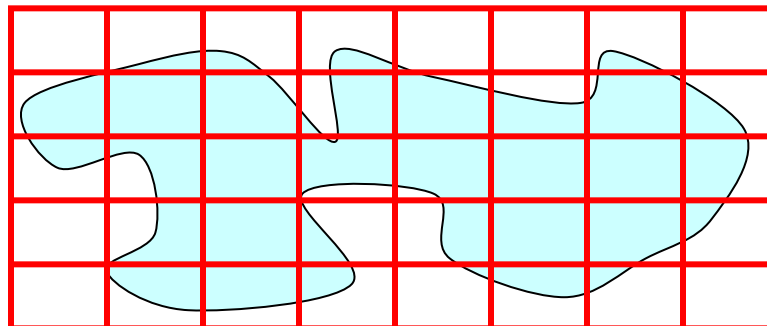   Expilcit Parallelism

      - Parallelism is visible to the programmer

      - Difficult to do (right)

      - Large improvements in performance

Models of parallelism

Data Parallelism (domain decomposition)

Data structures partitioned (data parallelism)

- Each process execute the same work on a sub-set of the data structure
- Data placement is critical
- More scalable than functional parallelism

Problem for the boundary management

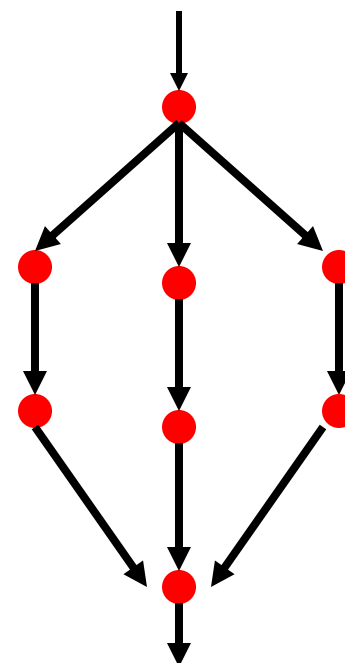**Load balancing (in some cases)**

Models of parallelism / 1

Control Parallelism (Functional Parallelism)

- the different functions are distributed

Partitioning by task:

each process executes a different
"function": Identify the functions, and
then the data requirements

Load balancing

Functional or data Parallelism

Functional or Data Parallelism?

Partition by task (functional parallelism)
- o   each process performs a different "function"
- o   identify functions, then data requirements
- o   commonly programmed with message-passing
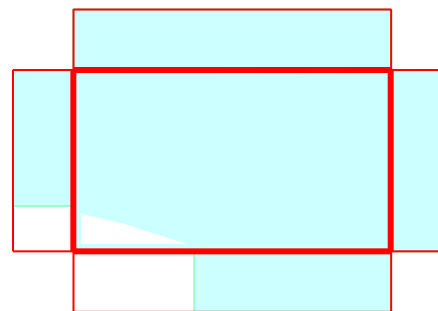
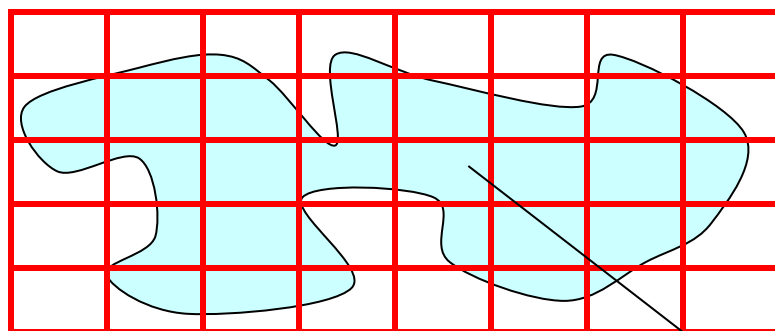Partition by data (data parallelism)
- o   each process does the same work on a unique piece of data
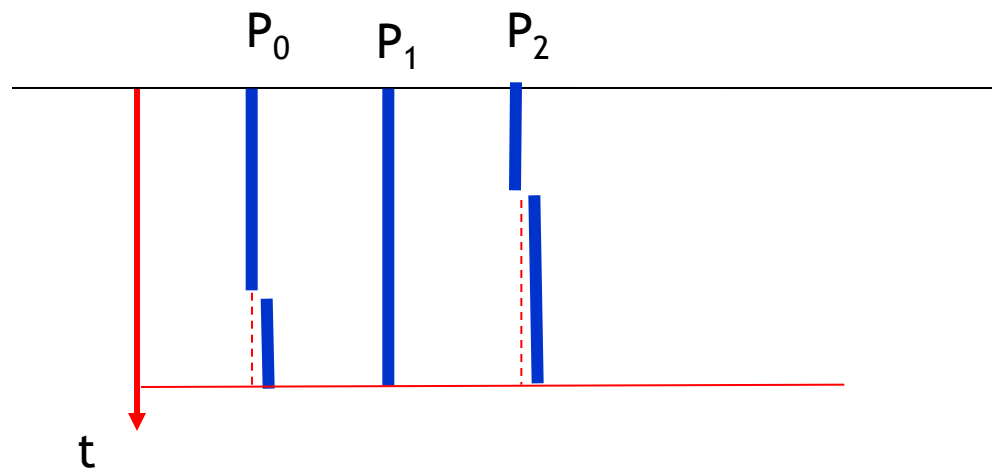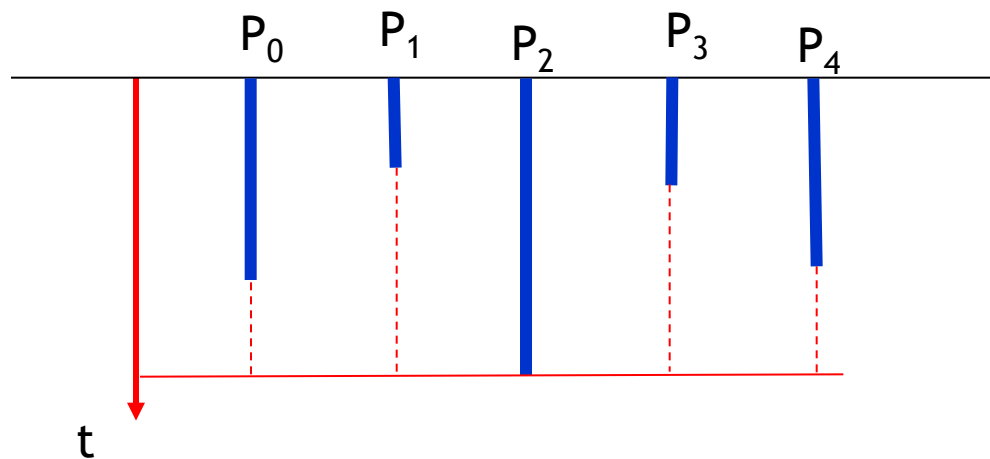- o   data placement is critical
- o   more scalable than functional parallelism
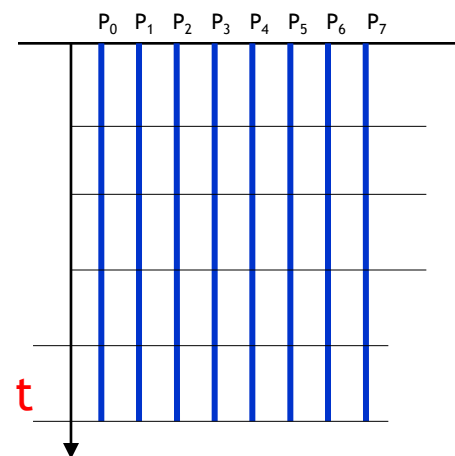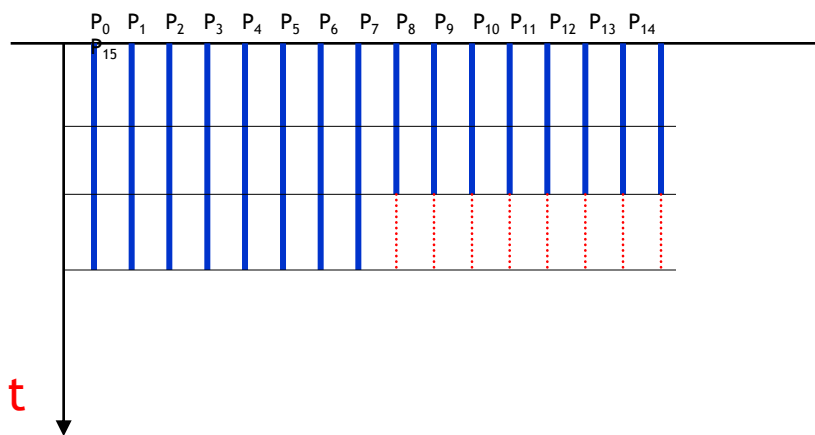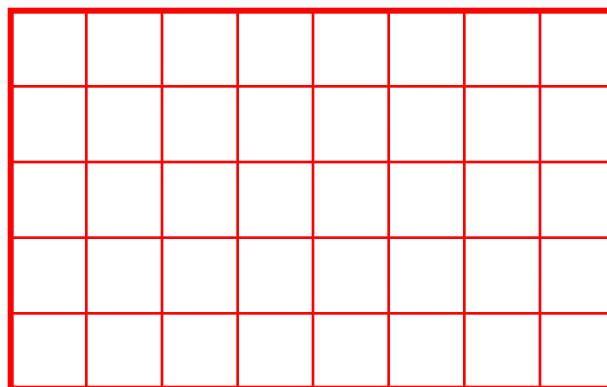
Boundary management

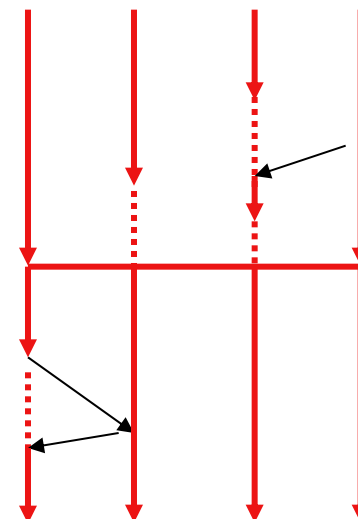Data Parallelism

Load Balancing

Load balancing /1

MPI Execution Model

- Single Program Multiple Data

- A copy of the code is executed by each process

- The execution flow is differnt dipending from the context (process id, local data, etc)

OpenMP Execution Model

- A single thread starts execute sequentially

- When a parallel region is reached, several slave threads are forked to run in parallel

- At the end of the parallel region, all the slave threads die

- Only the master thread continues the sequential execution
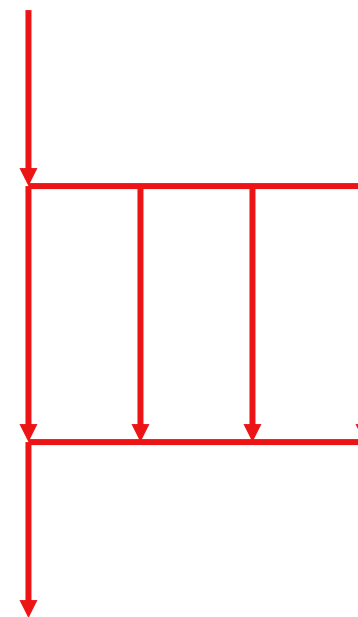
Notes on Shared Memory model: Access to Shared variables

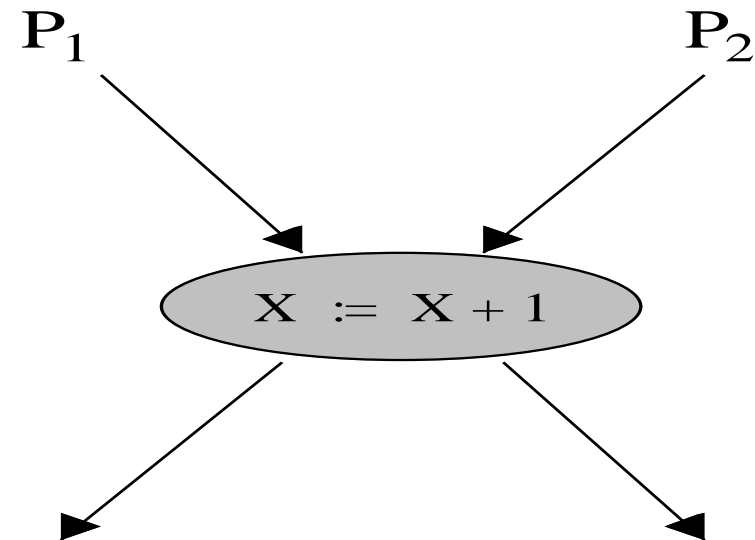Two processes P1 and P2 share a variable x that both must increment

More process can read (**load**) concurrently the same memory location without any problems.

This operation is well-defined conceptually each process makes a copy of the contents of the memory location and stores it in its own register.

Problems can occur when there is a concurrent access in writing (**store**) that is when multiple processes simultaneously write to the same memory location.

$$P_1 \qquad\qquad P_2$$

$$X := X + 1$$

The programmer, the programming language and the architecture should provide tools to solve the conflicts

What is the final value of x?

P1 loads x in a Register(P1)

    P2 loads x in a Register(P2)

P1 increments the value loaded

    P2 increments the value loaded

P1 stores the new value

    P2 stores the new value

Notes on Shared Memory model: non determinism

Non-determinism is caused by **race conditions**.
A race condition occurs when two different concurrent tasks access the **same memory location**, at least one of them in writing.

There is not a guaranteed execution order between the accesses.
The access must be **mutually exclusive**
The problem of non-determinism can be solved by synchronizing the use of shared data.

The portions of a parallel program that require synchronization to avoid non-determinism are called **critical sections**. These sections must be executed in a mutual exclusive way

Notes on Shared Memory model: Locks

In shared-memory programming specific constructs are needed to guarantee the execution of critical sections in a mutually exclusive way.

i.e lock (), or higher level constructs, with hardware support

Thread 1:
    LOCK (X)
        X = X + 1
    UNLOCK (X)

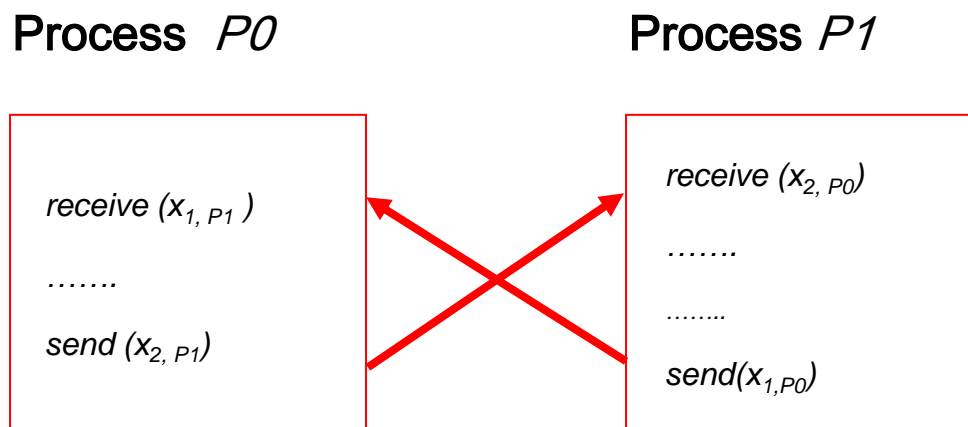Thread 2:
    LOCK (X)
        X = X + 2
    UNLOCK (X)

Notes on message passing model: Deadlock

Situation in which one or more processes remain indefinitely blocked because do not happen the necessary conditions for their continuation

A group of processes are in deadlock when all the processes of the group are waiting for an event (acquisition or release of resources) that can be caused only by one of the waiting processes.

Process *P0*　　　　　　　　**Process** *P1*

receive ($x_{1, P1}$)

…….

send ($x_{2, P1}$)

receive ($x_{2, P0}$)

…….

……..

send($x_{1,P0}$)

Parallelization: Goals and decisions

Goals (ideals): Ensure the speed-up and scalability:
- · Assign each process a unique amount of work
- · Assign each process the data required for the job to do
- · Minimize the replication of data and computation
- · Minimizie the communication between processes
- · Balance the work load

Keep in mind that:
- - For a problem there are several parallel solutions
- - The best parallel solution not always comes from the best scalar solution