IN COLLABORATION WITH
MIT MANAGEMENT SLOAN SCHOOL

UNIVERSITÀ DEGLI STUDI DI NAPOLI PARTHENOPE

MASTER MEIM 2022-2023

DIGITAL TECH

High Performance Computing

Lesson 3

Prof. Livia Marcellino

Prof. of High Performance Computing, Università degli Studi di Napoli Parthenope

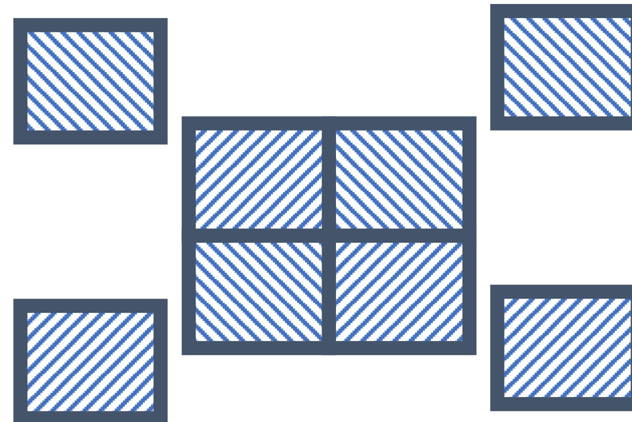
PARALLEL COMPUTING

Decompose a problem

in more subproblems

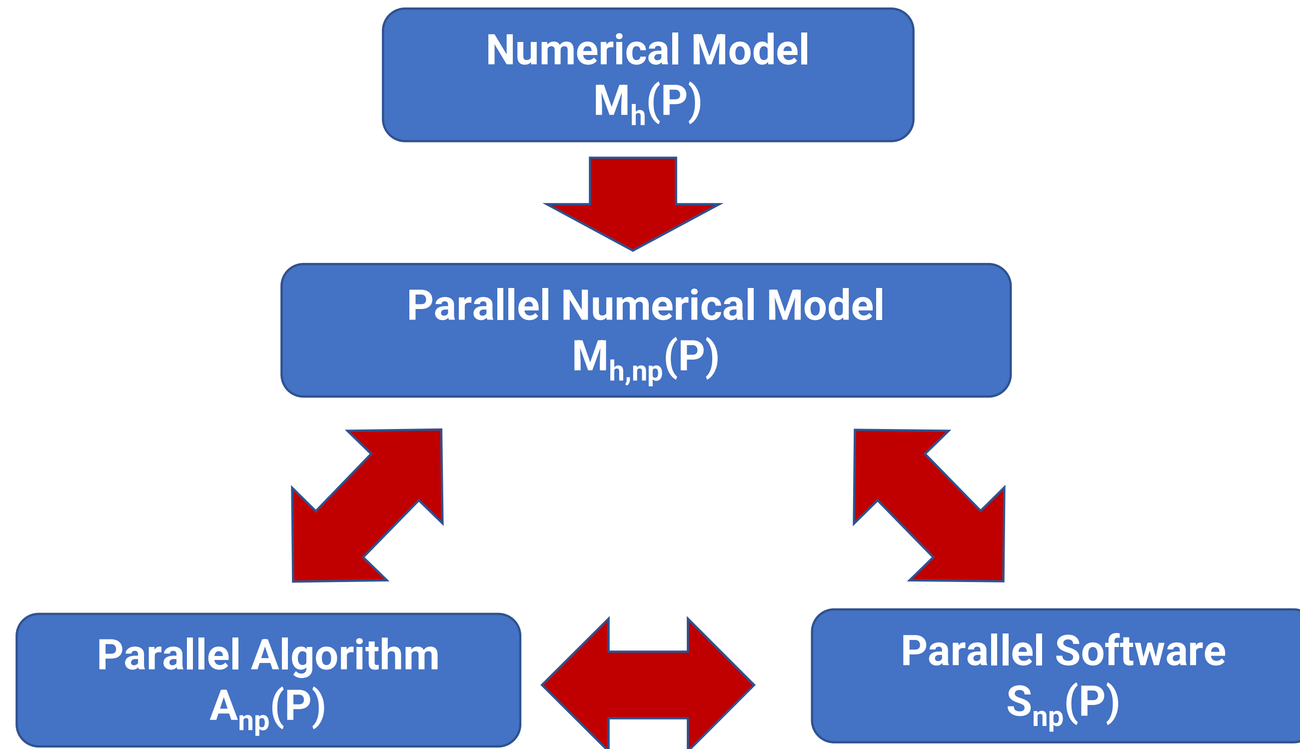
and solve them at the same time

with more processing units!



Need to create machines that can distribute the work among them
hardware development

What does parallel thinking mean?

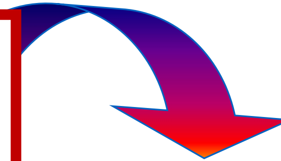


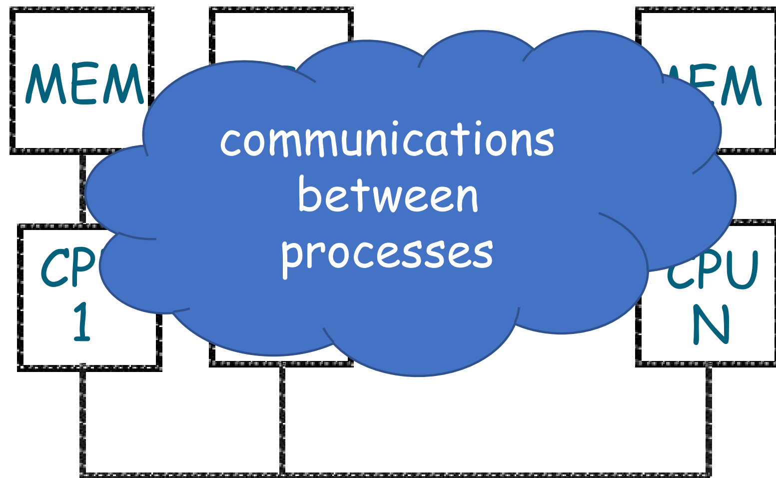
Parallel and distributed software design

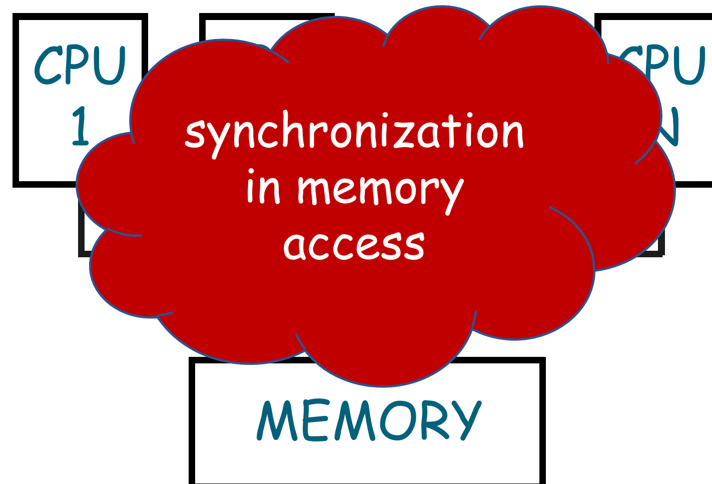
The most important modern parallel architectures

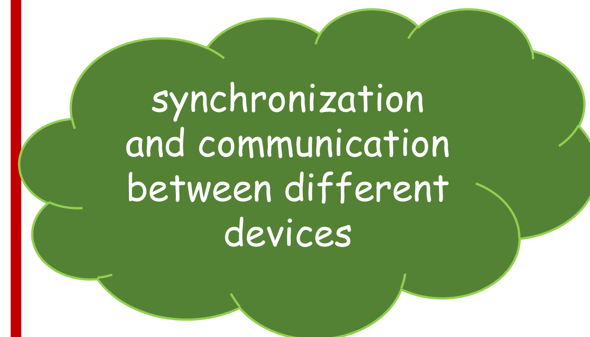
Computer MIMD
DM
(distributed-memory)

Computer MIMD
SM
(shared-memory)









Computational kernels:
data structures

whatever is the problem to be solved

(and you have seen and will see it in the previous and next lessons)

the basic computational kernels to parallelize are based on

only two kind data structures:

vectors and matrices

i.e. 1D-2D arrays

Computational kernels:

domain decomposition

to decompose the problem and solve it in parallel

it is enough to better understand the concept of

domain decomposition

A simple example:
sum of N elements of 1D - array

a

$$a_0 + a_1 + \ldots + a_{N-1}$$

Sum of N numbers

On a single processor computer, the sum is computed by performing N-1 additions

one at a time

$sumtot := a_0$

$sumtot := sumtot + a_1$

$sumtot := sumtot + a_2$

⋮

$sumtot := sumtot + a_{N-1}$

Sum of N numbers

On a single processor computer, the sum is computed by performing N-1 additions

one at a time

```
begin
     sumtot:= a0;
     for i=1 to N-1 do
             sumtot:= sumtot+ai ;
     endfor
end
```

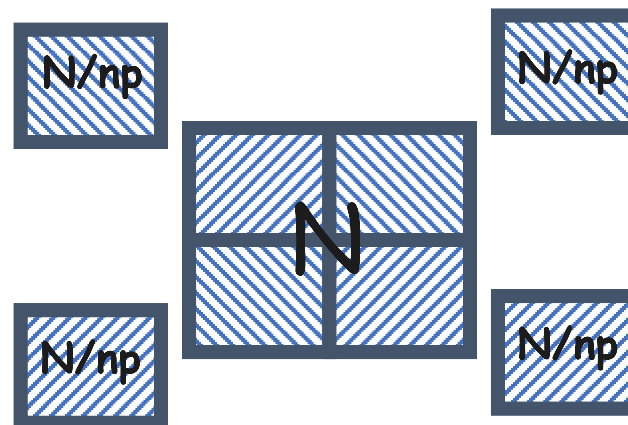
Which is the

PARALLEL ALGORITHM?

The parallel paradigm of MIMD architectures

sum of N numbers

To split a problem of size N in **np** equal sub-problems of size N/**np** e to solve them concurrently by using **np** CPU



A simple example:
sum of N elements

a

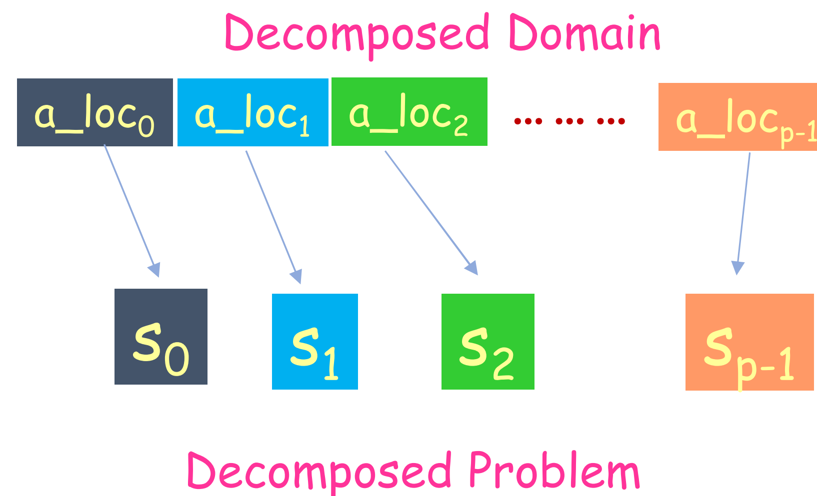
1D – array domain decomposition

a_loc_0 a_loc_1 a_loc_2 … … … a_loc_{p-1}

To decompose the domain in order to decompose the problem

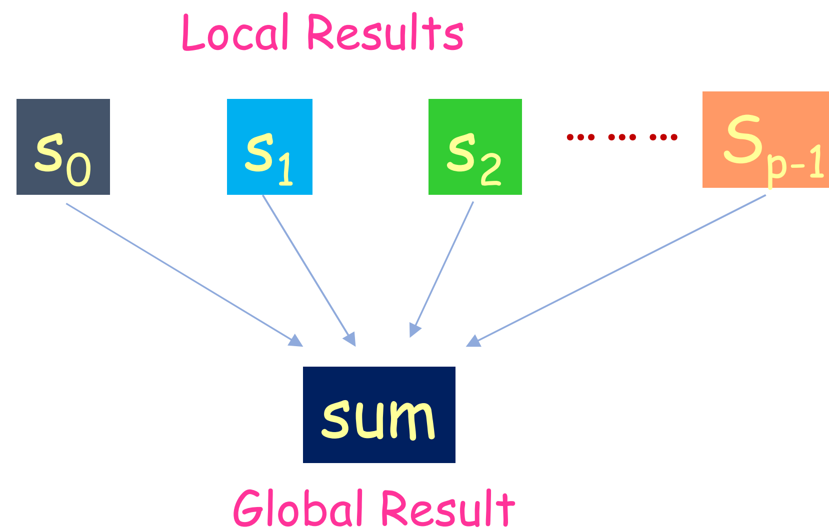
STEP 1: domain and problem decomposition...

...divide the sum into partial sums and assign each partial sum to a processor...



STEP 2: local results collection...

... then the partial sums must be combined *properly* to obtain the total sum



The parallel algorithm

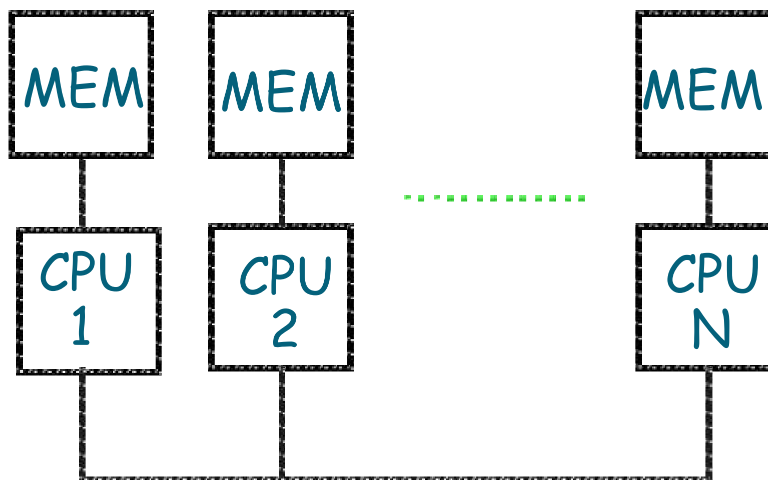
once it is clear the basic concept of

the domain-problem decomposition and the results collection,

to write the algorithm it is essential to have in mind

the hardware characteristics of the computer machine

Computer MIMD
DM
(distributed-memory)

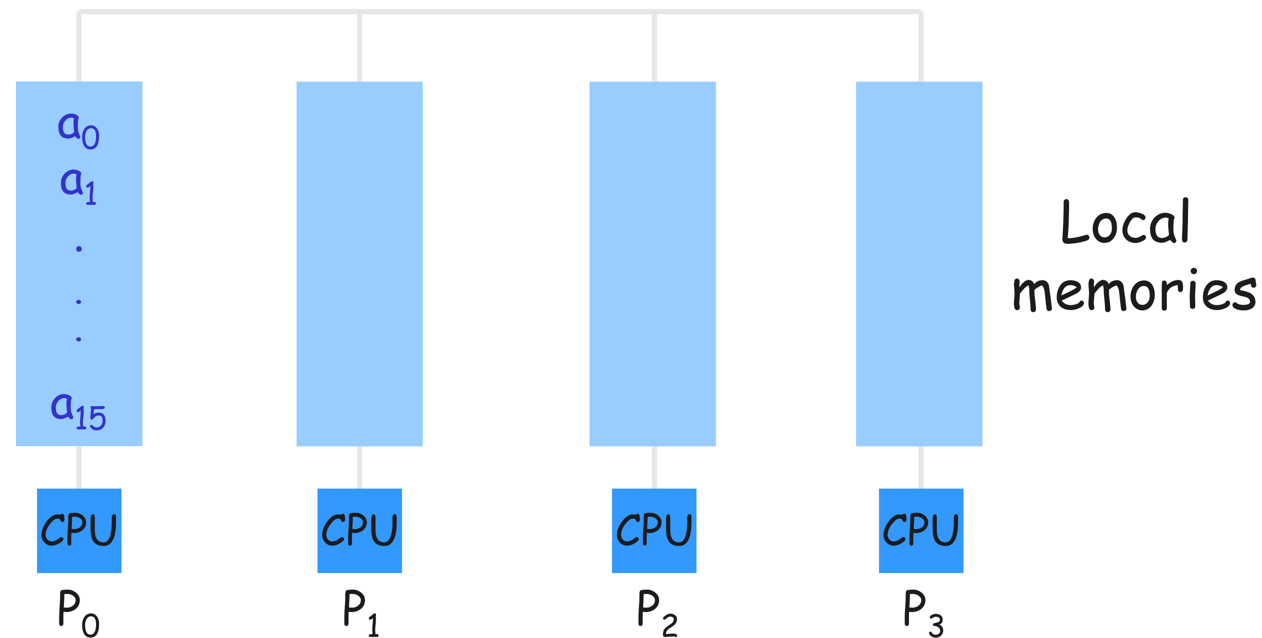


Parallel sum – MIMD-DM

data must be distributed among the processors

Input: only one PC (the master) can be read input data

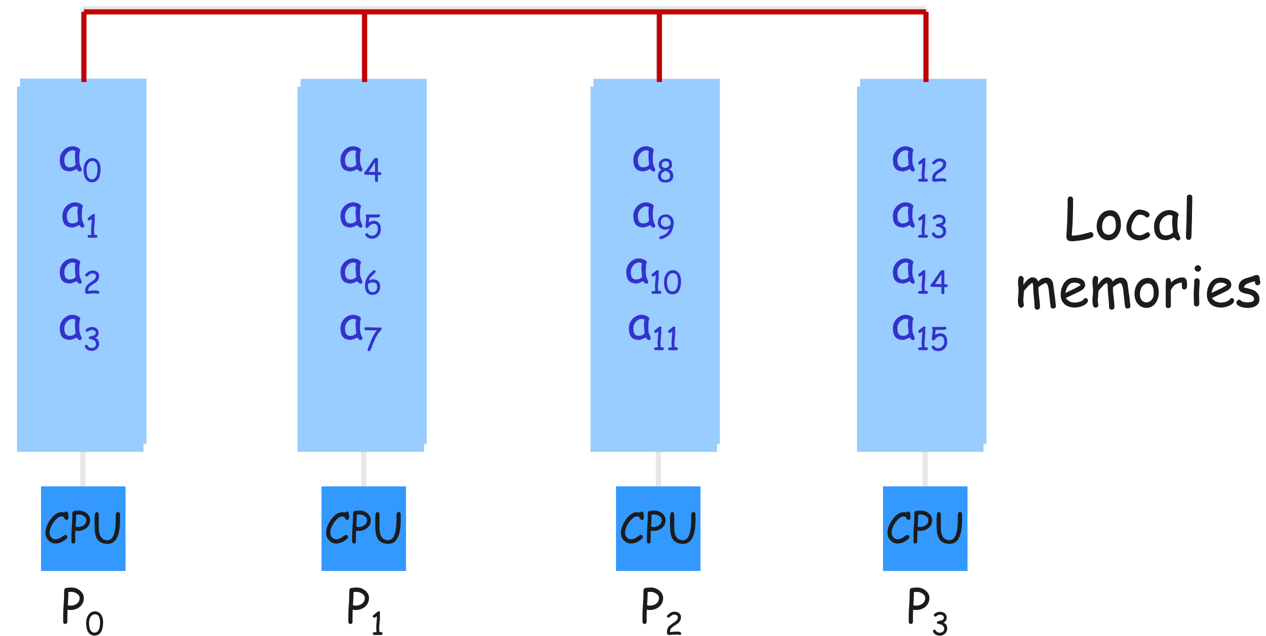
Example: N=16, p=4



Parallel sum – MIMD-DM

Data distribution: **the master PC sends local data to the other PCs in the cluster**

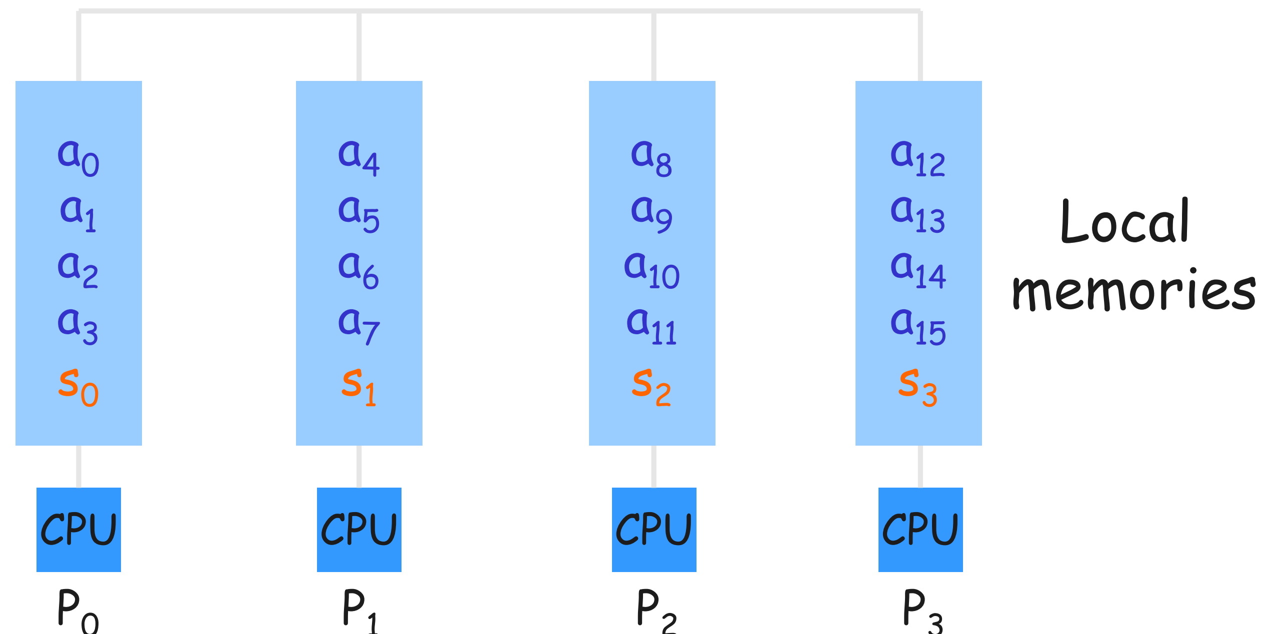
Example: N=16, p=4



Parallel sum – MIMD-DM

Local sum computation: **each PC computes a sum using the data in its memory**

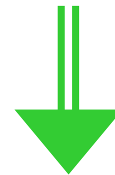
Example: N=16, p=4



Parallel sum – MIMD-DM

Local results collection:

in order **to obtain** the sumtot value,
each PC (processor) must communicate its local result to others PCs



Communication

between the different memories

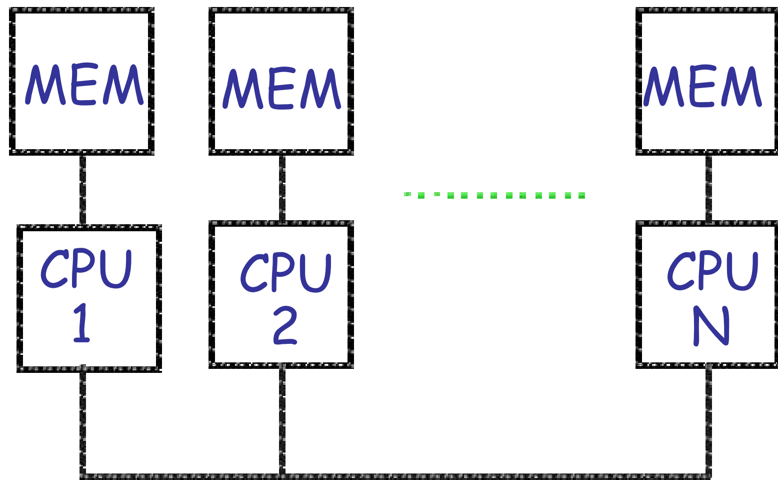
MIMD: distributed memory (DM)

To create a distributed memory MIMD machine that has undemanding production costs, **clusters** are generally used, i.e. sets of autonomous computers connected to each other through the I/O interconnections, and therefore with connectors and cables typical of a standard network.

Each computer has its own **separate copy of the operating system**, which **increases the administration costs**, but this drawback can be easily **overcome by using virtual shared memory machines.**

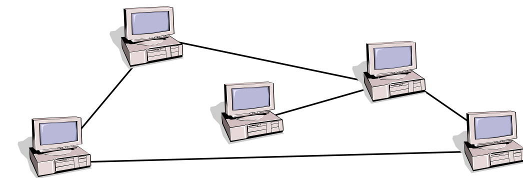
MIMD-DM architectures

cluster of multiprocessors - tools



Message Passing Interface
MPI

Homogeneous cluster

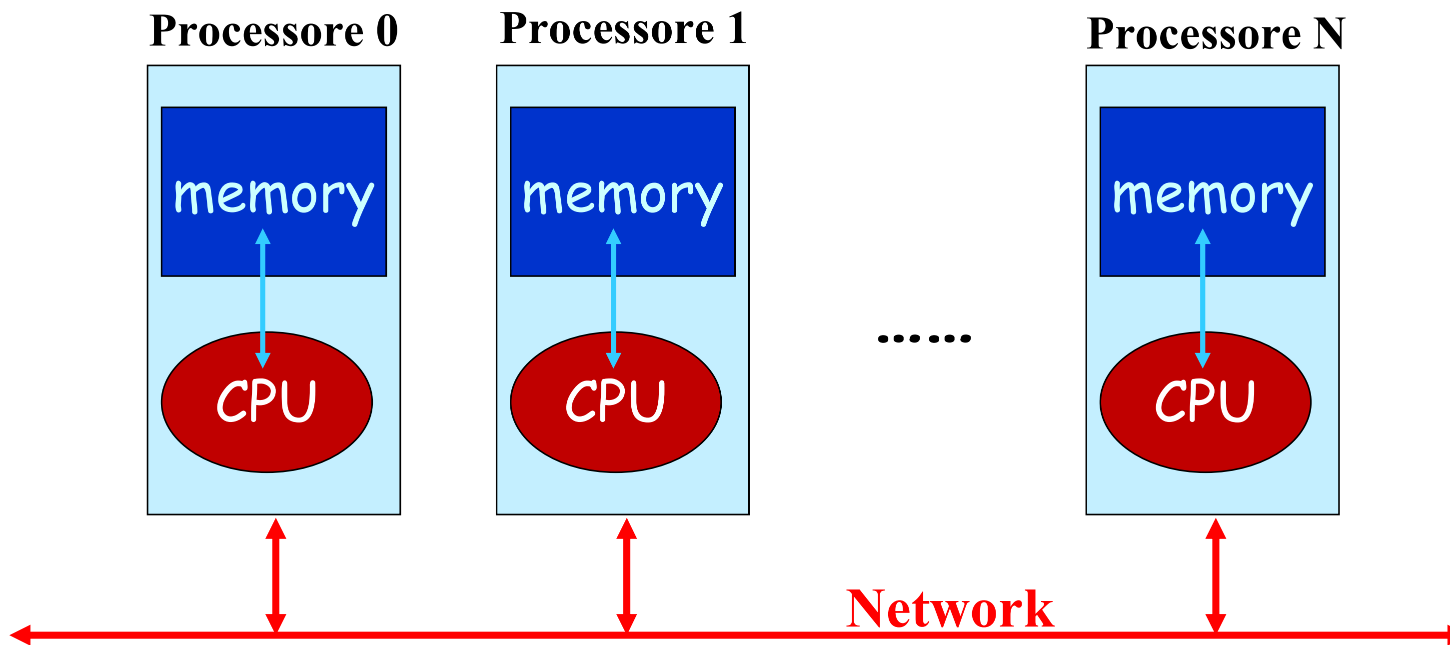


Need to organize communications between processors

MIMD-DM architectures

cluster of multiprocessors - tools

Message Passing Interface MPI



Each processor directly accesses to its own local memory directly
and
can know data stored in memory of other processors, through **data transfer**.

MIMD-DM architectures

cluster of multiprocessors - tools

The **MPI library** born in **1991** and over the years many versions have been proposed to make it more *user friendly*.
To date, any other library for developing parallel code in the MIMD-DM environment is based on MPI and its message passing paradigm:

- **PBLAS** (**P**arallel **B**asic **L**inear **A**lgebra **S**ubprograms), based on **BLAS**
- **ScaLAPACK** (**Sca**lable **L**inear **A**lgebra **PACK**age), based on **LAPACK**
- …
- …
- **PETSc** (**P**ortable, **E**xtensible **T**oolkit for **S**cientific **C**omputation)

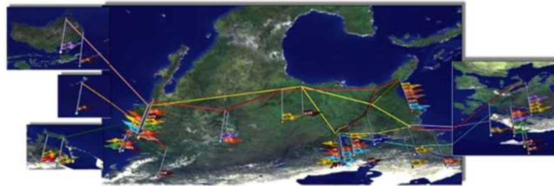
Specific parallel software for numerical solution (in MIMD-DM environment) of problems modeled by differential equations (ODE-PDE)and in particular to deal with systems of linear and non-linear equations which represent the computational kernel in discretizing of ODEs and PDEs.

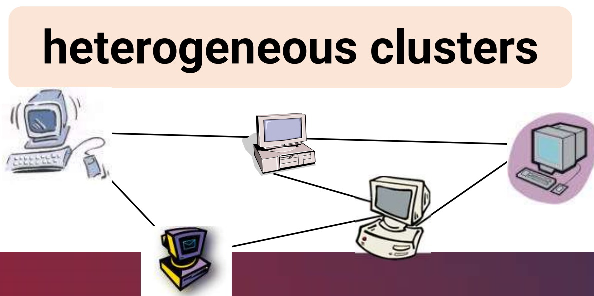
MIMD-DM architectures

cluster of multiprocessors - tools

Today, to do parallel computing in a distributed environment can be considered a bit agè compared to parallel paradigms available on new HPC architectures, especially for time required by data transfer!
But it remains very useful for problems characterized by a large amount of data…

…and above all it has been transformed, in a very useful way, into the new technologies of **GRID computing** and in particular into the well-known **CLOUD computing o storage**

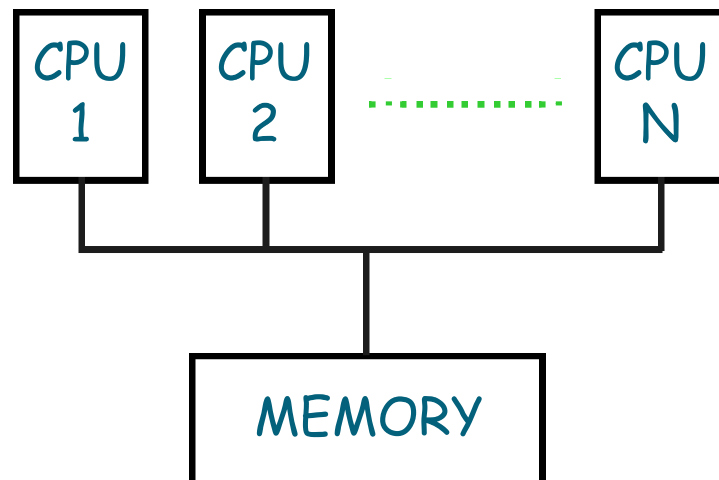


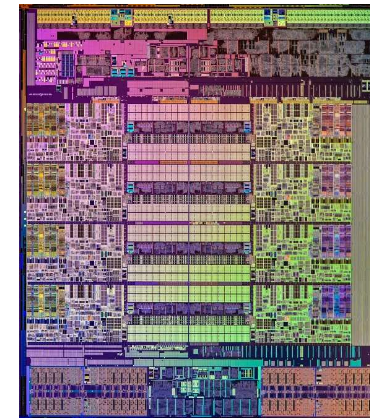


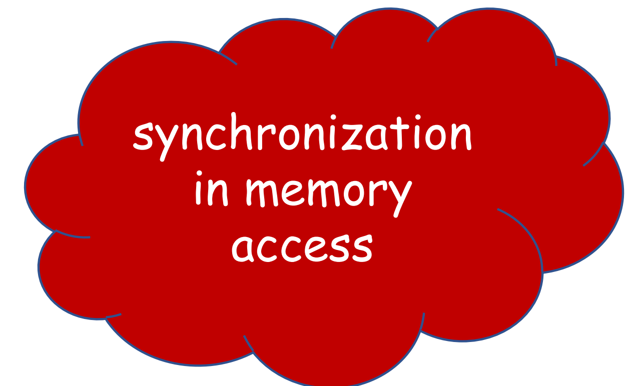
Re-use of existing resources:
users can acquire and release resources dynamically

Let’s take a break

Computer MIMD
SM
(shared-memory)

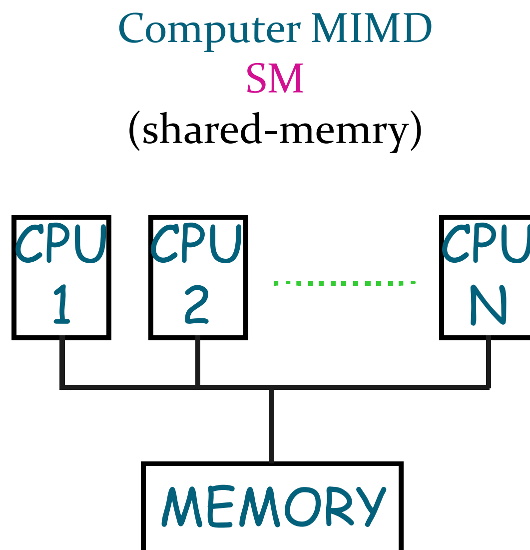


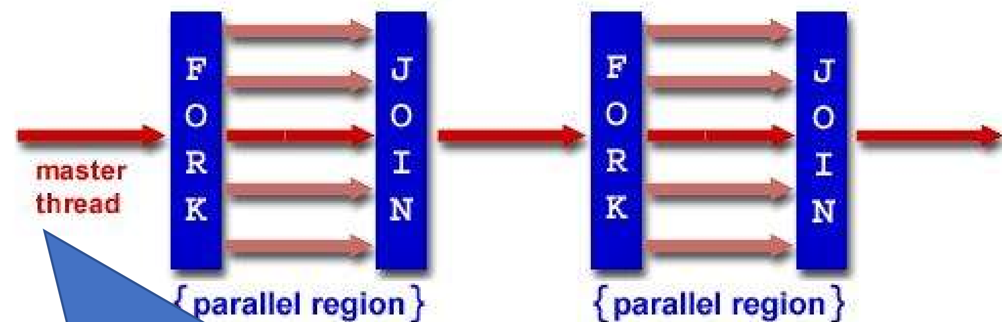




MIMD-SM

this type of hardware works according to the *fork-join model*

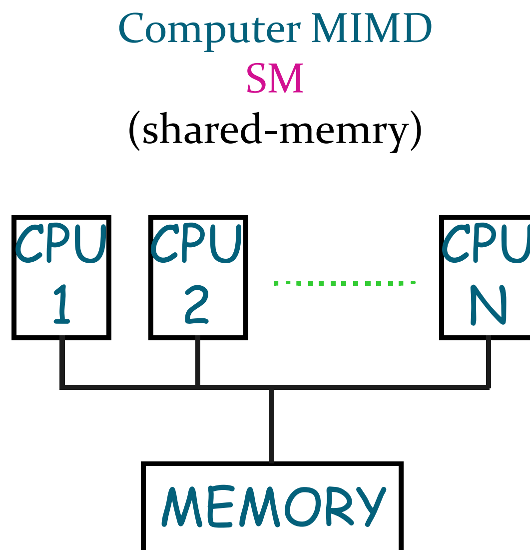


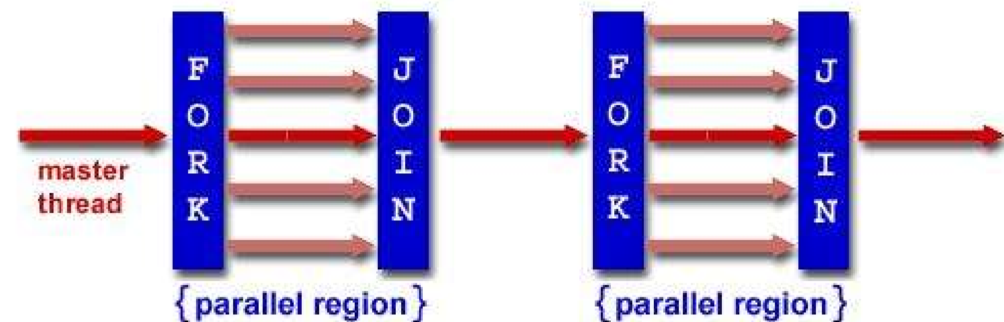


All processes start with a single thread (master thread) which executes sequentially

MIMD-SM

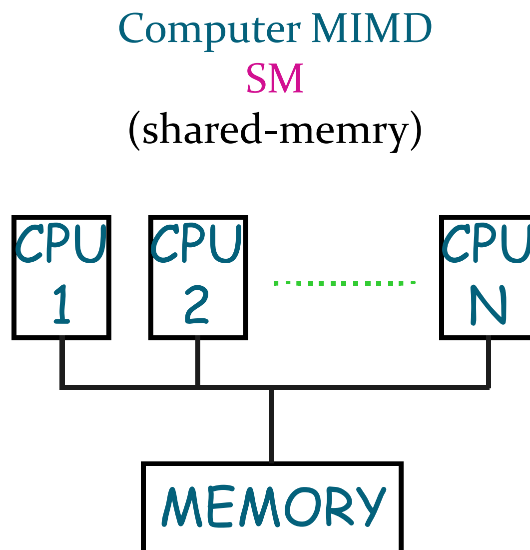
this type of hardware works according to the *fork-join model*

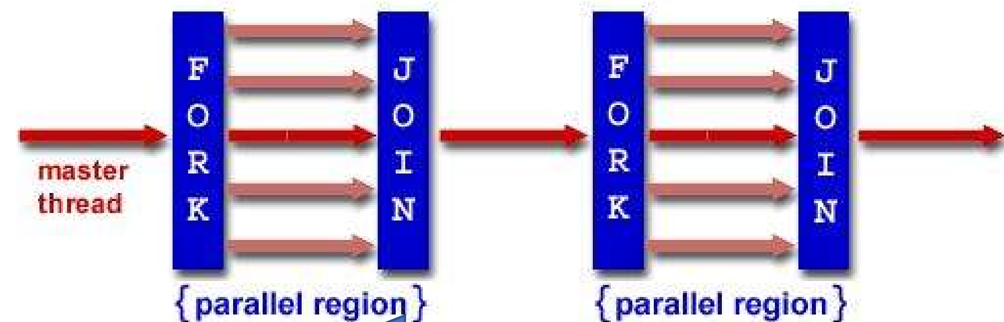




Fork: a parallel region starts when a team of threads is created and it proceeds in parallel

MIMD-SM
this type of hardware works according to the *fork-join model*



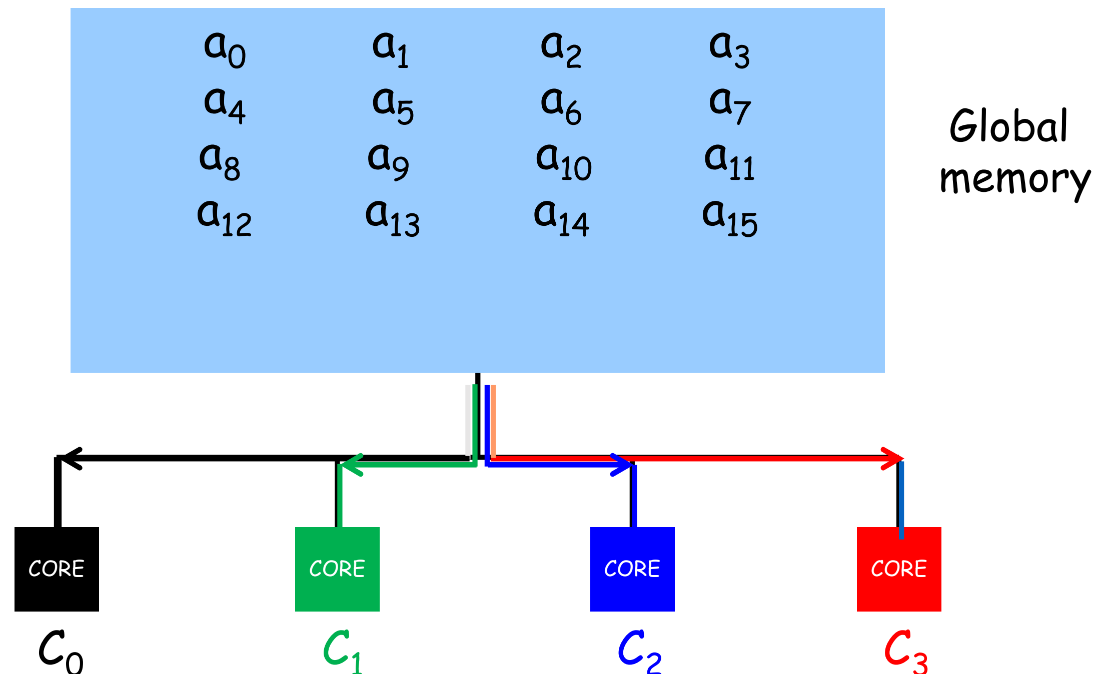


Join: when all the threads of the team have finished the instructions of the parallel region, the master thread works again sequentially, until a new FORK call is made

Parallel sum – MIMD-SM

Input: **the master core (thread) reads input data**

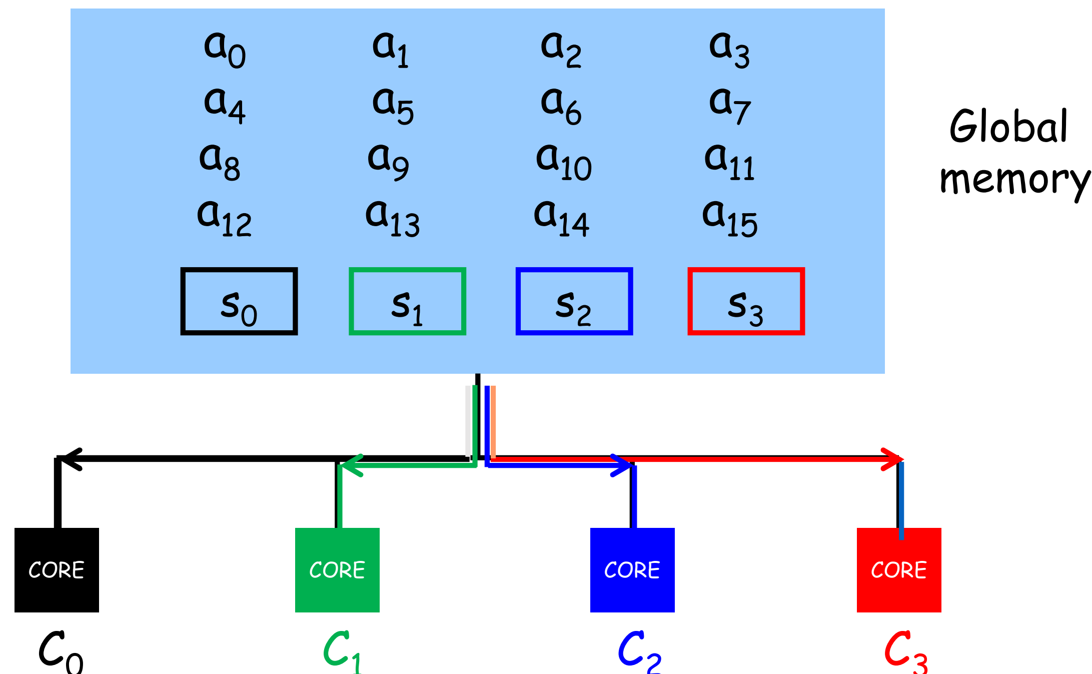
Example: N=16, p=4



Parallel sum – MIMD-SM

Local sum computation: all cores can simultaneously access global memory **on different data**

Example: N=16, p=4

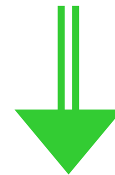


How to compute the global sum?

Parallel sum – MIMD-SM

Local results collection:

in order **to update correctly** the sumtot value,
each core must have **exclusive access** to this variable during the last phase



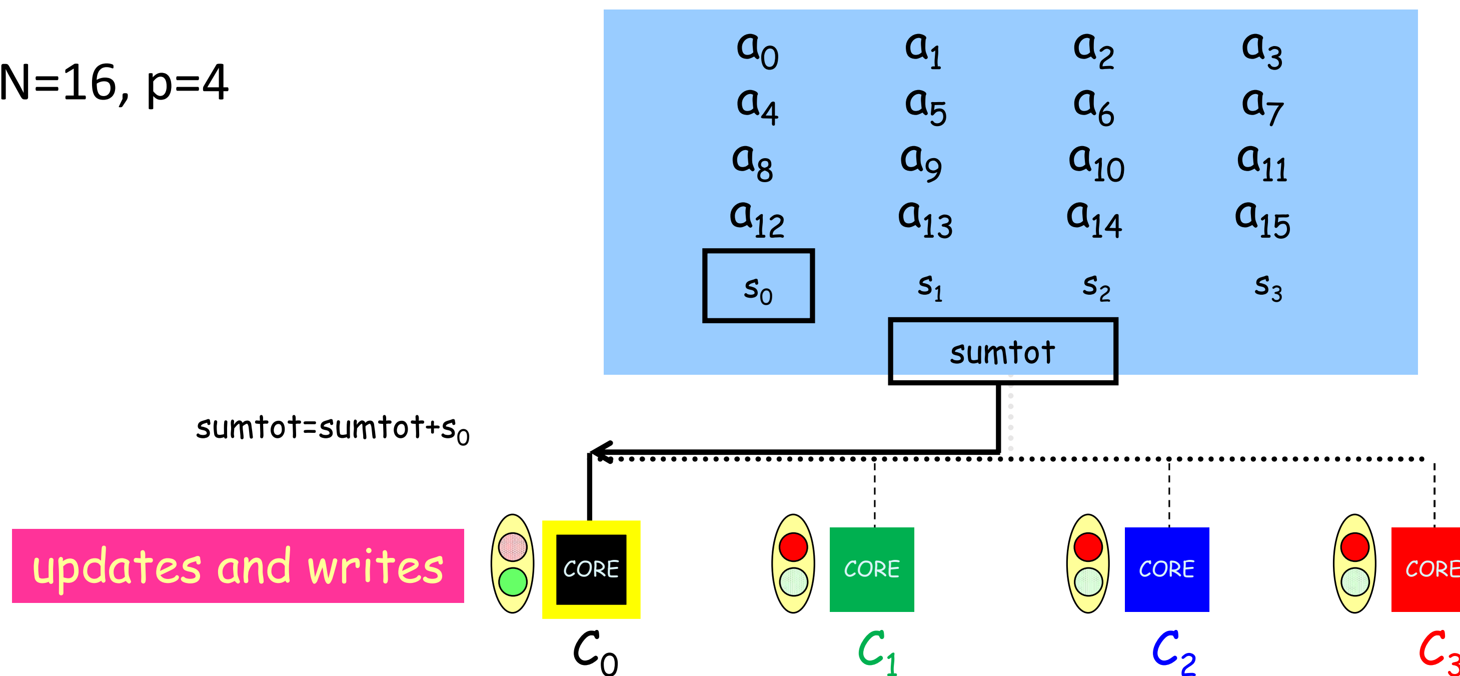
Synchronization

of memory accesses

Parallel sum – MIMD-SM

Local results collection: 1 strategy

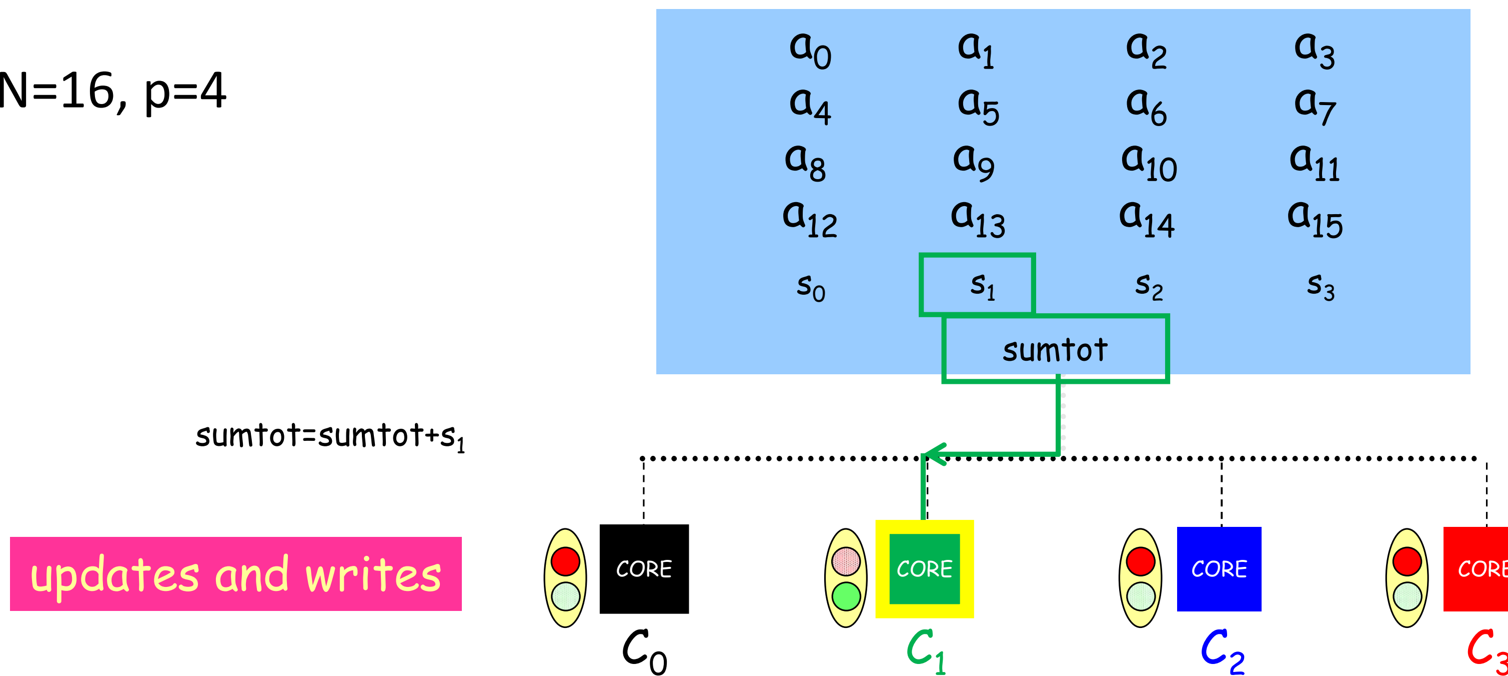
Example: N=16, p=4



Parallel sum – MIMD-SM

Local results collection: 1 strategy

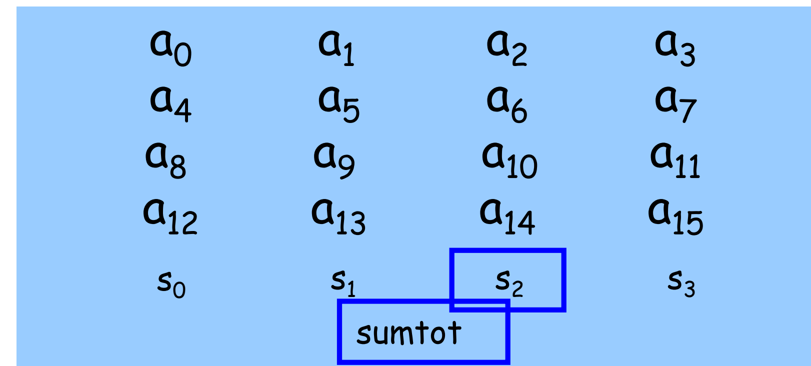
Example: N=16, p=4



Parallel sum – MIMD-SM

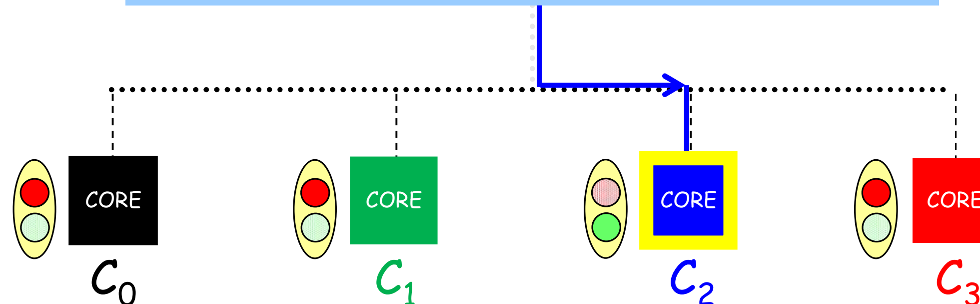
Local results collection: 1 strategy

Example: N=16, p=4



sumtot=sumtot+s_2

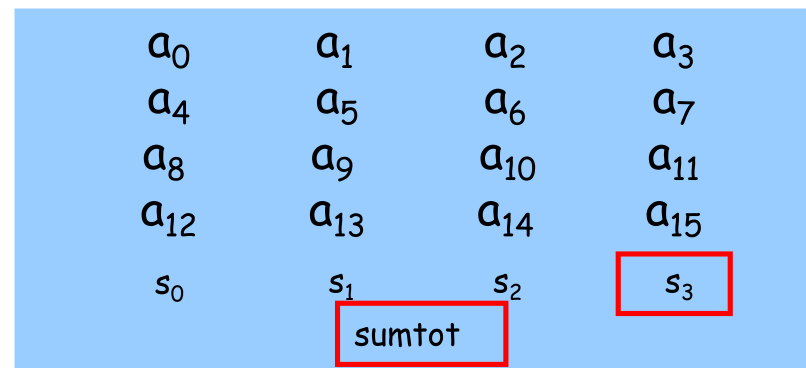
updates and writes



Parallel sum – MIMD-SM

Local results collection: 1 strategy

Example: N=16, p=4



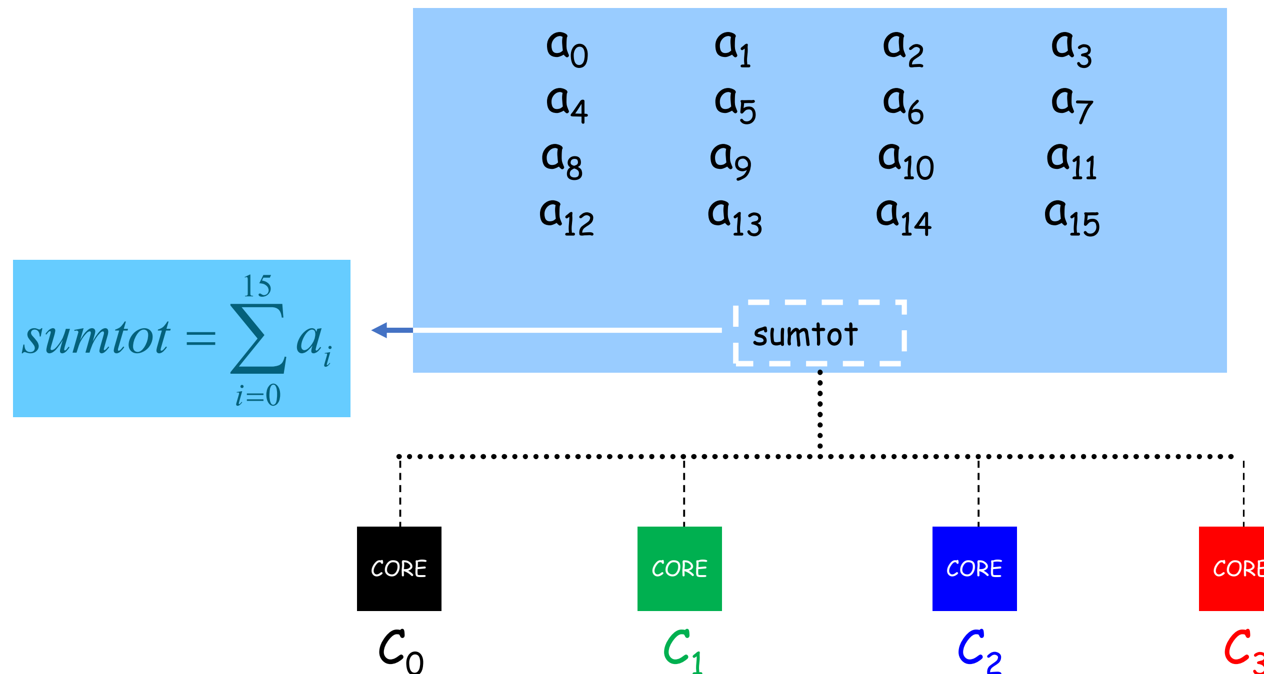
$sumtot=sumtot+s_3$

updates and writes

Parallel sum – MIMD-SM

Local results collection: 1 strategy

Example: N=16, p=4



I strategy (MIMD-SM)

Each core
- compute its own partial sum.

At every step
- each core adds, its own partial sum to a single predetermined values.

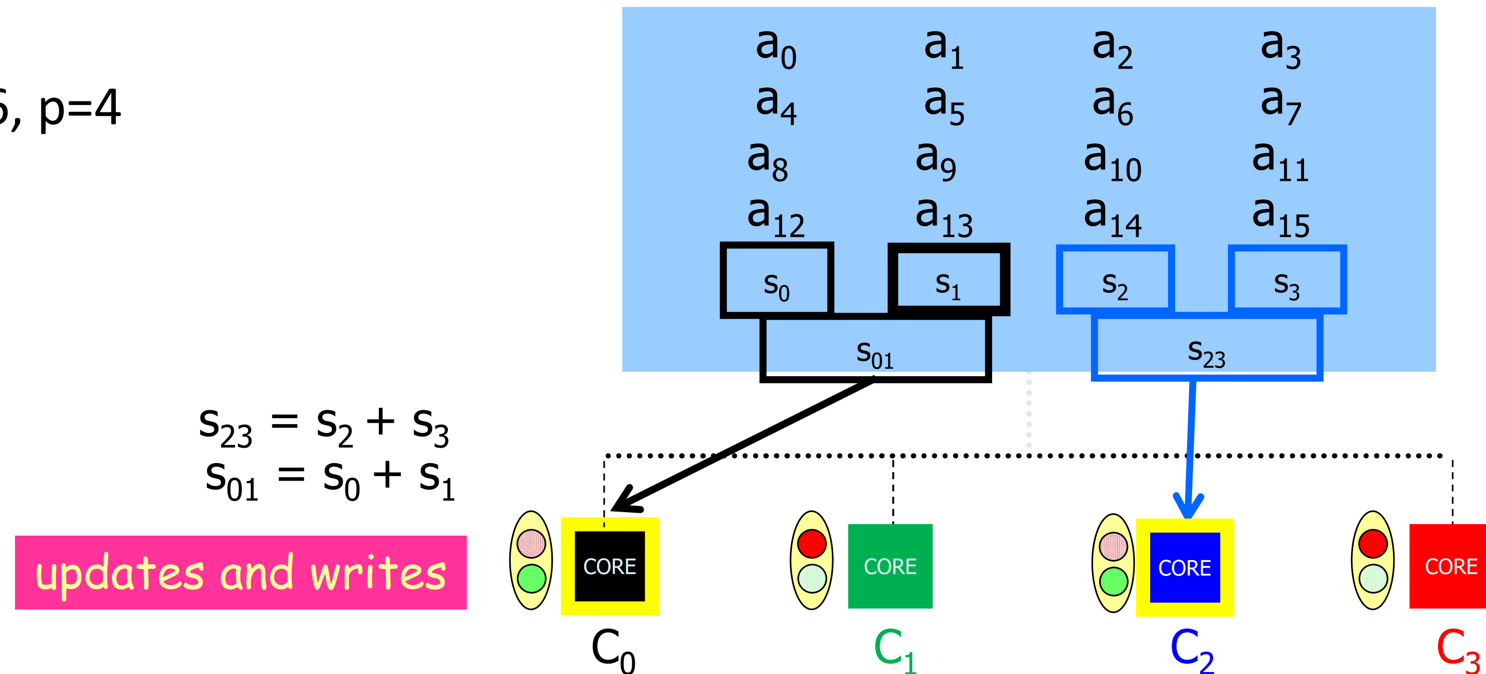
The global sum is stored in the shared memory.

Concurrent operations

Parallel sum – MIMD-SM

Local results collection: 2 strategy

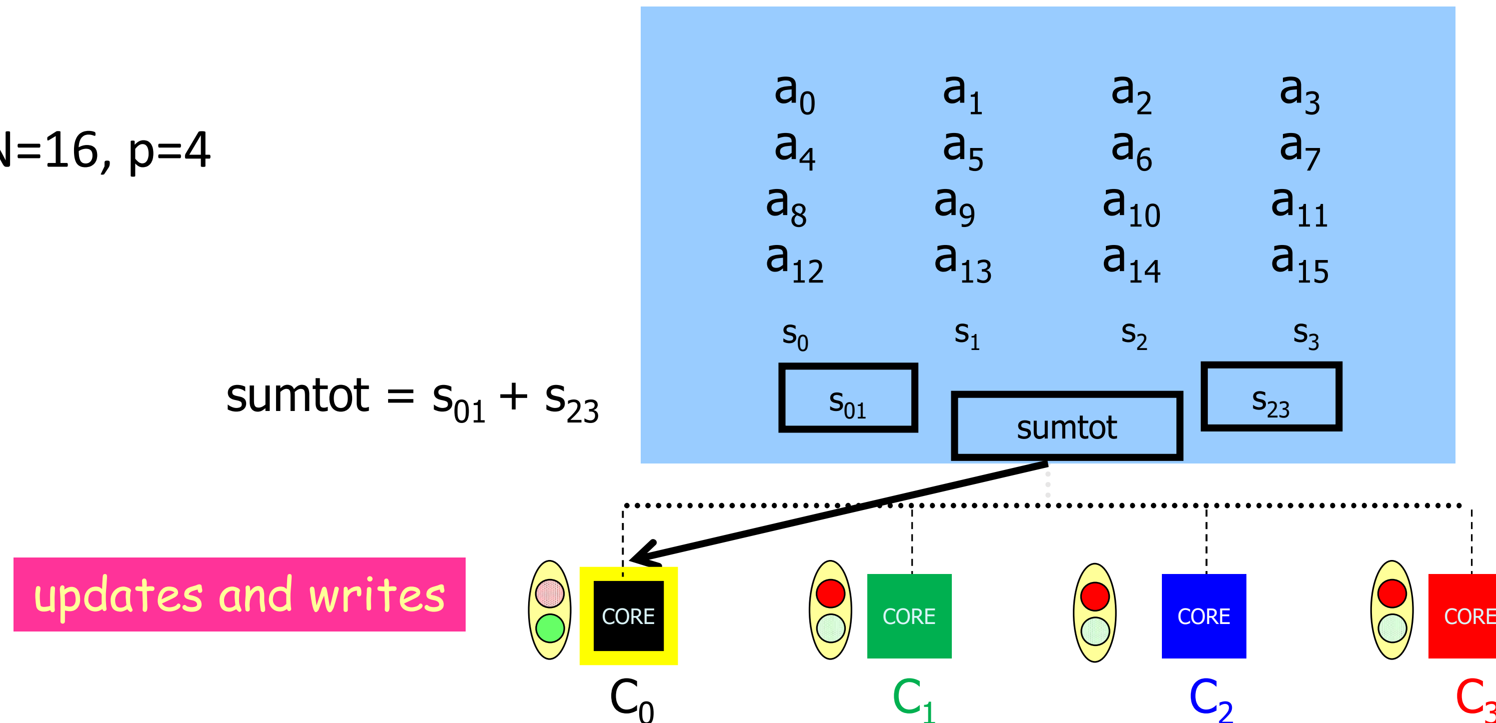
Example: N=16, p=4



Parallel sum – MIMD-SM

Local results collection: 2 strategy

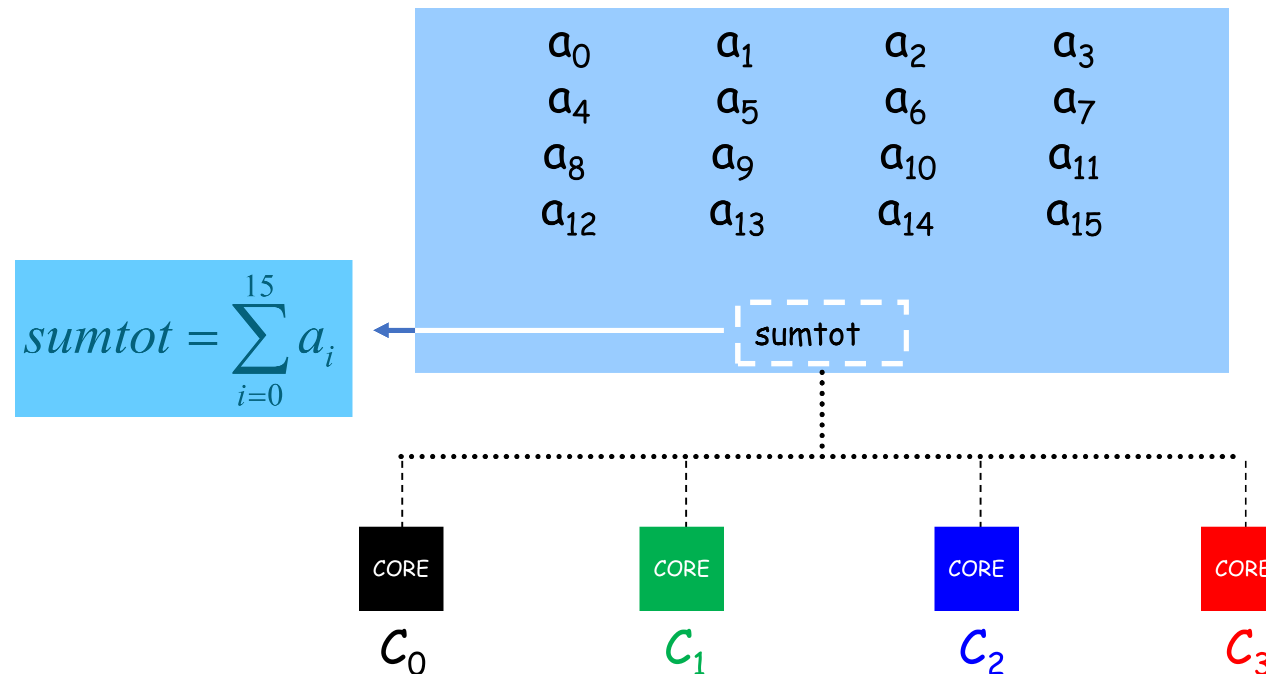
Example: N=16, p=4



Parallel sum – MIMD-SM

Local results collection: 2 strategy

Example: N=16, p=4



II strategy (MIMD-SM)

Each core

- compute its own partial sum.

At every step

- half of cores (with respect to the previous step) computes a contribution of the partial sum.

The global sum is stored in the shared memory.

Concurrent operations

There are several tools for software development in MIMD-Shared Memory computing environment

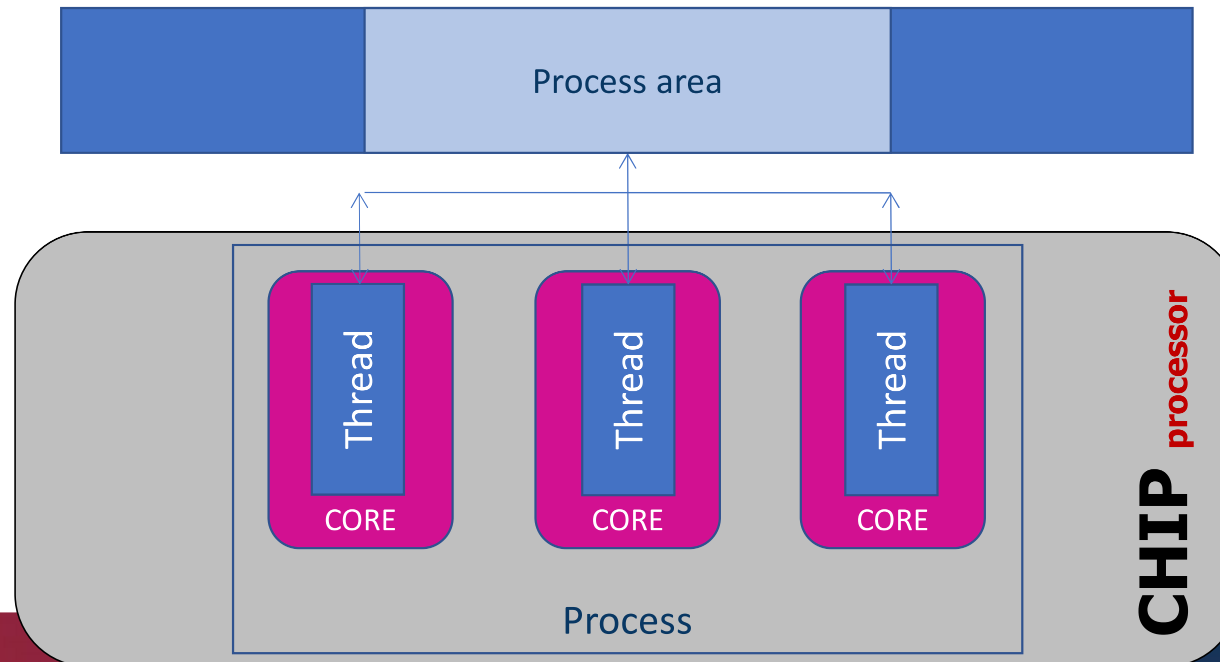
OpenMp - Pthreads

MIMD-SM architectures

multicore processor - tools

Open specifications for Multi Processing
OpenMP

Cores of the same processor share the same memory area, working together, by synchronizing the accesse to shared variables.



MIMD-SM architectures

multicore processor - tools

The **OpenMP library** born in **1997**. It is composed by **directives** for compiler to create teams of threads and to establish which instructions must be executed in parallel and how they must work, by means of some **clauses**. The OpenMP library is very simple to use.

Più complessa è invece la libreria POSIX Threads (**Pthreads**), ma su quest'ultima si basano le versioni per l'ambiente multi-core delle più note librerie numeriche:
The POSIX Threads (**Pthreads**) library is more complex, but on this one are based all numerical libraries for multi-core environment of the:

- **PLASMA** (**Parallel Linear Algebra for Scalable Multi-core Architectures**). Developed in 2006 but still under development, it includes a set of routines for basic linear algebra operations (based on BLAS) and more routines for solving systems of linear equations (based on LAPACK)…. …

It is also possible to install and use the PETSc library in order to run parallel code which uses its routines in multicore environment, avoiding data transfer.

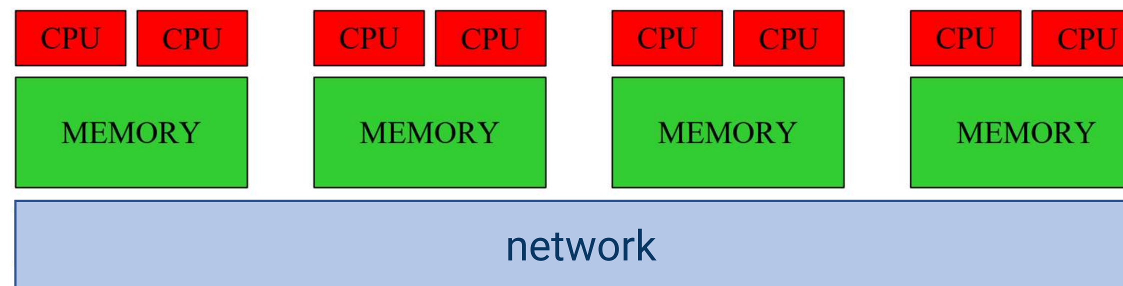
MIMD-SM architectures

cluster of multiprocessors with multicore - tools

Two or more parallel paradigms can be combined together on hybrid architectures.

Example:

A merge of parallel paradigm for Shared Memory environment (multicore) by using the **OpenMP** library (inside each cluster's node) with a parallel paradigm for Distributed Memory environment (cluster of multiprocessors) by using the **MPI** library (outside between nodes).



Computer MIMD

+

GPU

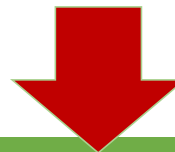
(Graphic Processing Unit)

synchronization and communication between different devices

What are GPUs?
GPU = Graphic Processing Unit

parallel microprocessors of modern video cards
for computer or console

GPUs born in the ***Computer Graphics*** field: rendering and graphics operations, parallel approach for big data



Thanks to their parallel processing power,
the GPUs were also used in
General Purpose applications!

why use GPUs?

Actually, GPUs offer the best performance at a low market cost.

Referring to Moore's Law (CPUs performance doubles every 18 months), GPUs performance double every 6 months.

Remark: triple Moore's Law!

GPU: the programming model

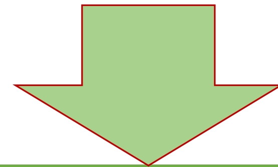
The programming model considers the **CPU** and the **GPU** as two distinct and separate machines, called **host** and **device**.

Each program combines:
sequential parts (demanded to the **host**) and parallel parts (demanded to the **device**).

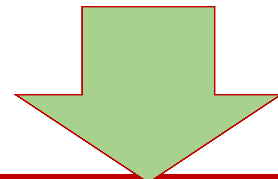
The parts of code which work in parallel are called **kernel**.
The **host** calls the **kernels** by configuring the **device** to run in parallel, passing it some parameters. The device runs only one *kernel* **at a time**.

Basic structure of a CUDA application

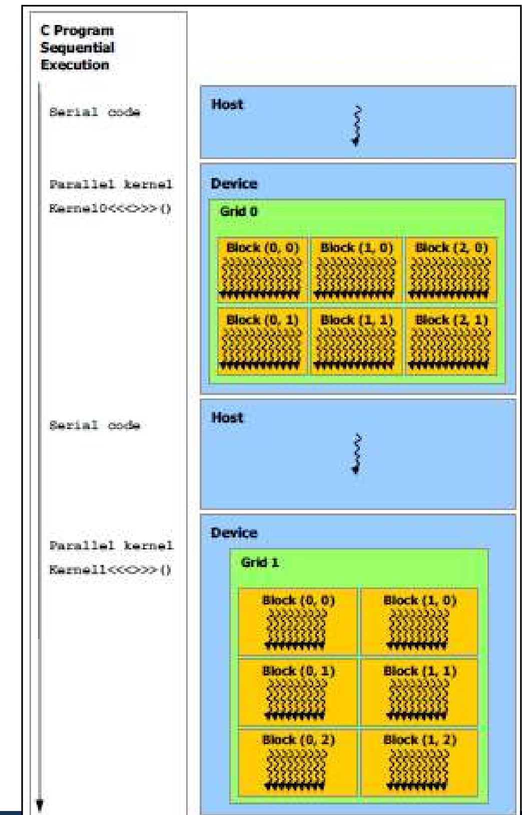
Before calling a kernel, the host transfers the data to be processed from the CPU memory to the GPU memory.



The host calls the kernel, passing it
the parameters and configuring its execution.



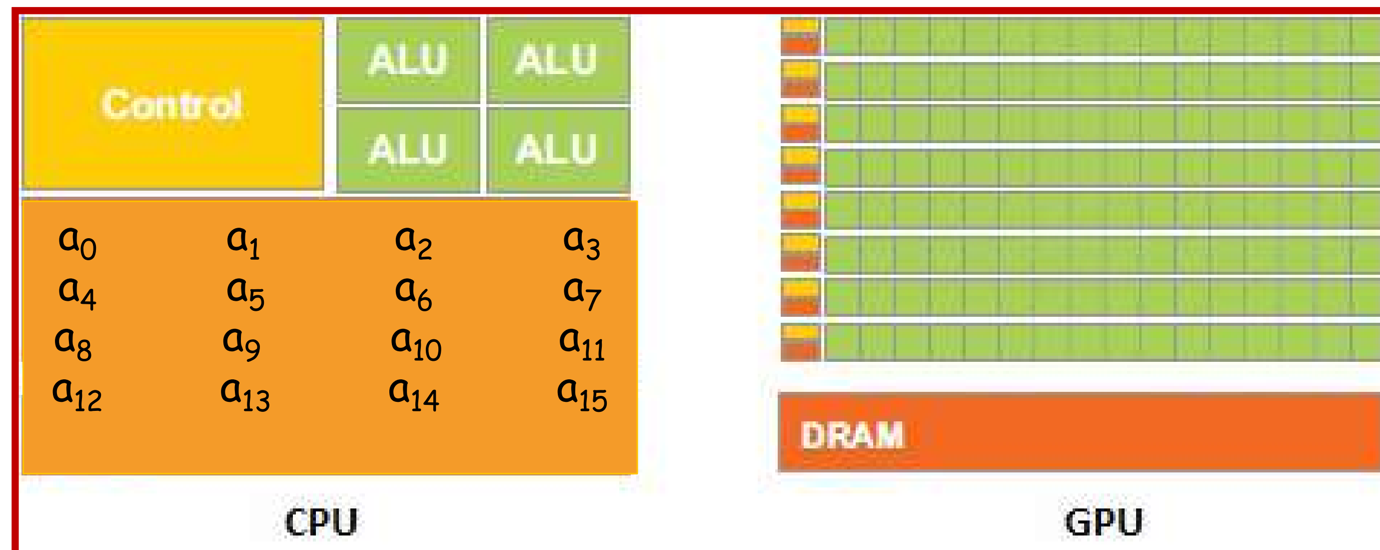
The host get back the results from the
device memory to the host memory



Parallel sum – on GPU

Input: the CPU (hots) reads input data

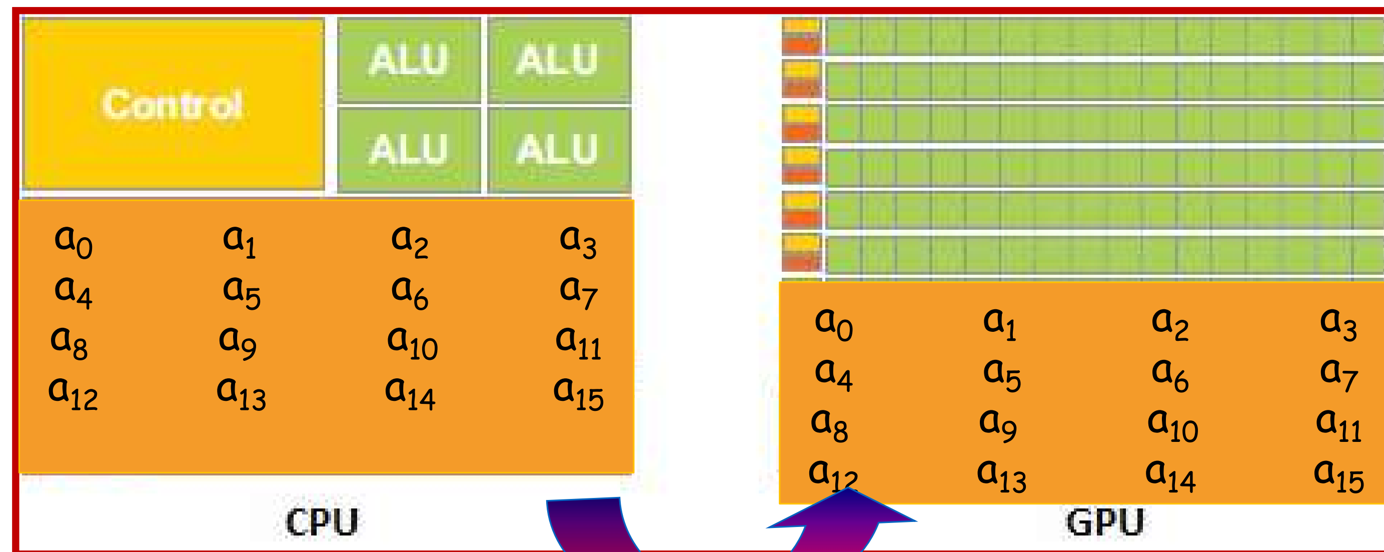
Example: N=16, p = many, many, many… all that I need!



Parallel sum – on GPU

Data transfert: the host transfers data to be processed from the CPU memory to the GPU memory.

Example: N=16, p = many, many, many… all that I need!



GPU memory organization

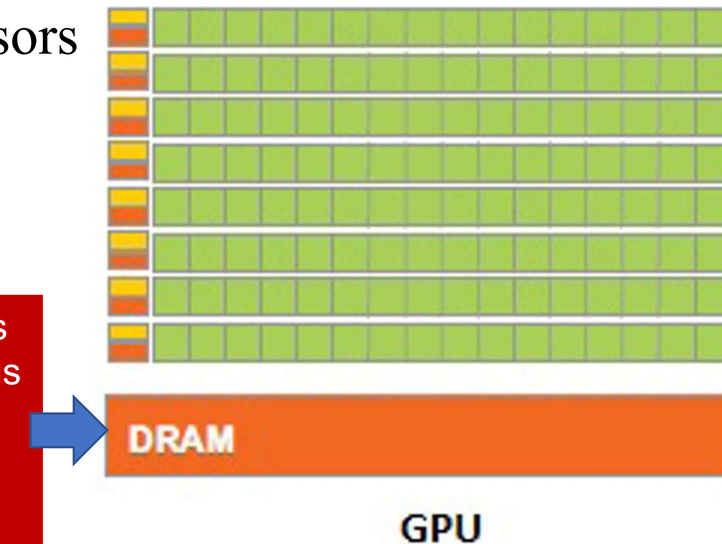
The memory of the GPU device can be divided into different types distinguishable by the latency in access time.
The most important are: GLOBAL memory, LOCAL memory and SHARED memory.

- ***Global memory*** is a **read/write** area, external to multiprocessors streaming and **shared** between all multiprocessor

In this space, controlled by the host, are the variables transferred from the host to the device and vice versa.
Memory interface between HOST-DEVICE

the access time to this memory is very high, but the global memory is the immediate interface with the RAM memory of the CPU.

So it is inevitable to use it



GPU memory organization

The memory of the GPU device can be divided into different types distinguishable by the latency in access time.
The most important are: GLOBAL memory, LOCAL memory and SHARED memory.

- ***Shared memory*: low-latency access area shared between all processors of the same streaming of multiprocessors.**

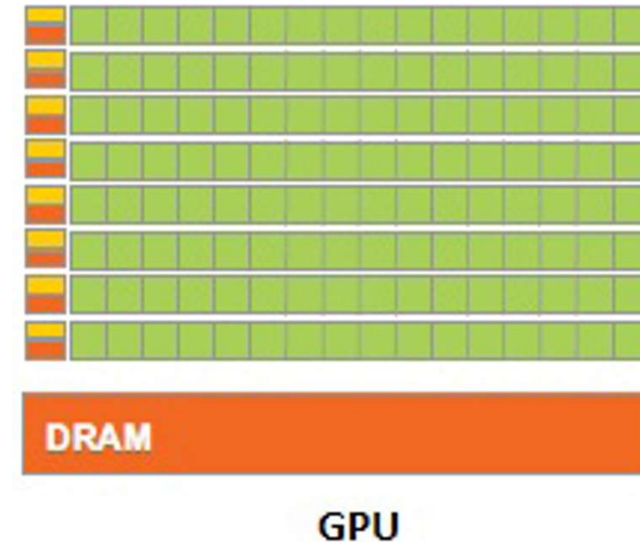
This memory **is very fast**, but **very small**

GPU

GPU memory organization

The memory of the GPU device can be divided into different types distinguishable by the latency in access time.
The most important are: GLOBAL memory, LOCAL memory and SHARED memory.

➤***Local memory***: private space for each individual processors where local variables are stored.



Also, these memories **are very fast**, but **they can be used only for local computation**

Organization of GPU memories

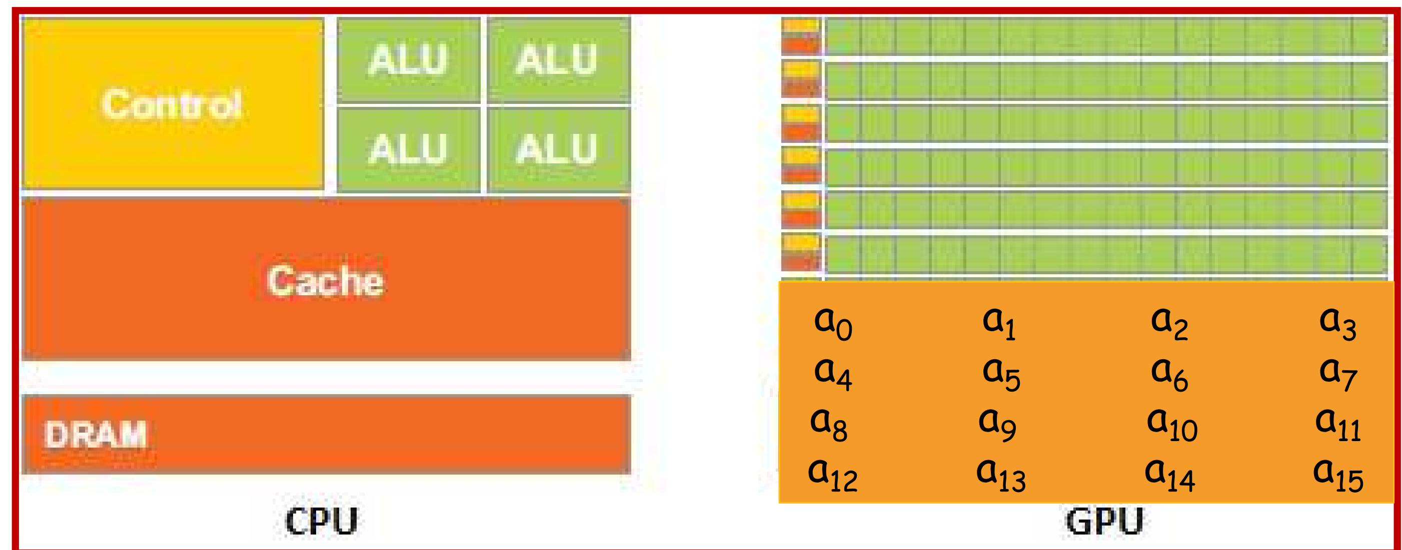
by suitably combining the use of these memories, very high performances can be achieved

Parallel sum – on GPU

Local sum computation: all processors can simultaneously access global memory on different data, in a very similar way to MIMD-SM environment

Example: N=16, p = many, many, many… all that I need!

local computation
is stored in local memory

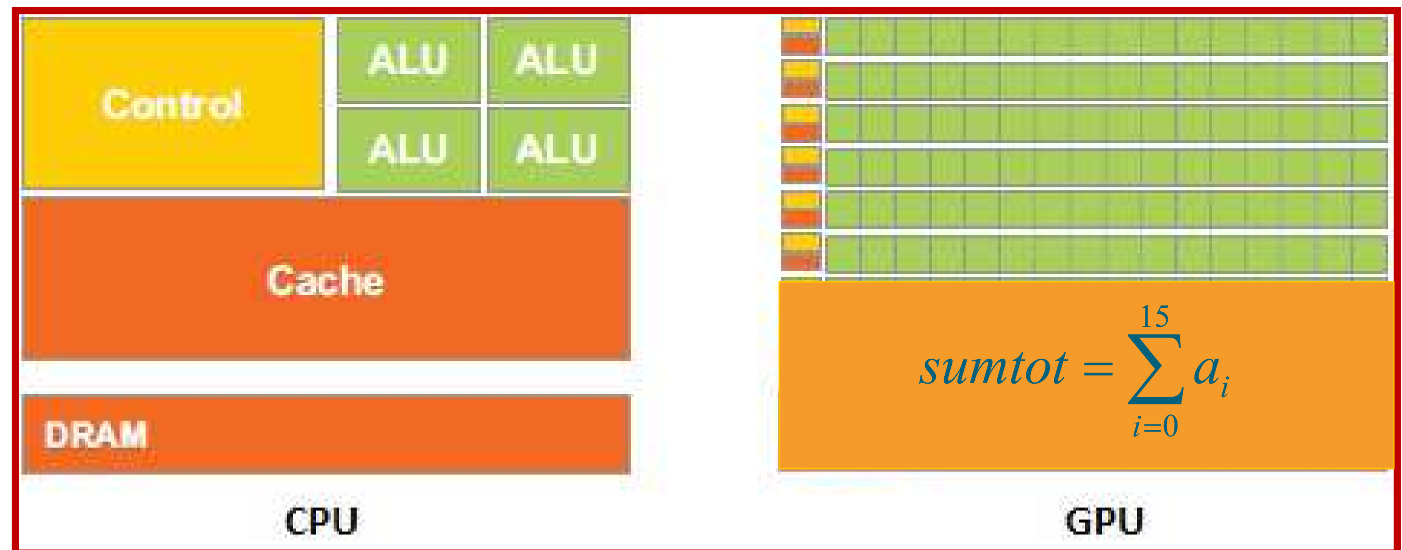


Parallel sum – on GPU

Local results collection: **in a very similar way to MIMD-SM environment, by 1st or 2nd strategy**

Example: N=16, p = many, many, many… all that I need!

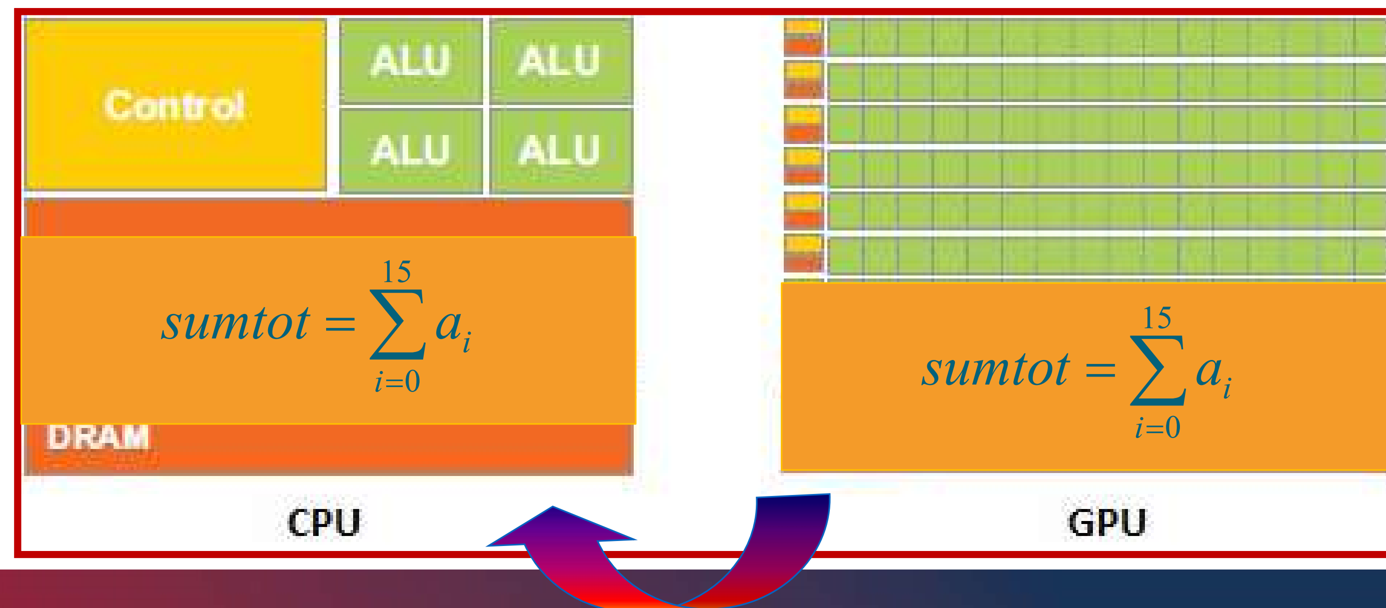
Final sums
is stored in the global memory



Parallel sum – on GPU

Data transfert: **the device transfers results from the GPU global memory to the CPU memory.**

Example: N=16, p = many, many, many… all that I need!



it is clear that using GPUs only makes sense when
data to be processed concurrently is really a lot
and
to use many many processors provides
high performance, in terms of exection time,
despite the price of host-device data transfer

There are several tools for software development for GPU environment

CUDA

MIMD-SM architectures

many-core GPU - tools

The **CUDA library (or environment)** born in **2006**. It combine serial code for the host (CPU) with parallel code, called **kernel** for the device (GPU).

The CUDA environment provides a suite of libraries for high-level programming…

MIMD-SM architectures

many-core GPU - tools

The CUDA environment provides a suite of libraries for high-level programming…

CUFFT

The NVIDIA CUDA Fast Fourier Transform library (cuFFT) provides a simple interface for computing FFTs up to 10x faster. By using hundreds of processor cores inside NVIDIA GPUs, cuFFT delivers the...

CUBLAS

The NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) library is a GPU-accelerated version of the complete standard BLAS library that delivers 6x to 17x faster performance than the latest MKL...

EM PHOTONICS CULA TOOLS

CULA is a GPU-accelerated linear algebra library that utilizes the NVIDIA CUDA parallel computing architecture to dramatically improve the computation speed of sophisticated mathematics. Because it...

MAGMA

MAGMA is a collection of next generation, GPU accelerated ,linear algebra libraries. Designed for heterogeneous GPU-based architectures. It supports interfaces to current LAPACK and BLAS standards...

MIMD-SM architectures

many-core GPU - tools

CUSOLVER LIBRARY

The **cuSolver** library is an high-level toolkit composed by direct solutors in managing problems characterized by matrix dense and sparse. It is based on **CUDA** and in particular on the **cuBLAS** (CUDA **B**asic **L**inear **A**lgebra **S**ubroutines) e **cuSPARSE** (CUDA Sparse Matrix) libraries.

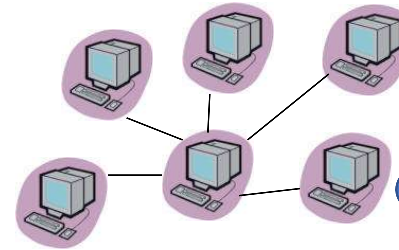
cuSolver provides useful function for:

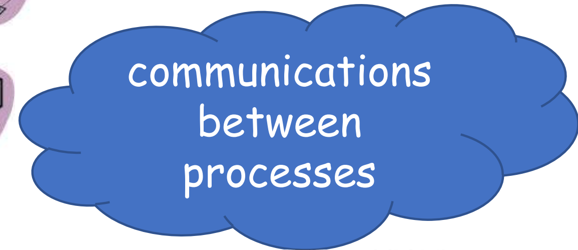
- ✓ matrix factorization,
- ✓ linear and non-linear system solution,
- ✓ least squares problems,
- ✓ eigenvalues comutation,
- ✓ …

MEiM | MASTER IN ENTREPRENEURSHIP INNOVATION MANAGEMENT IN COLLABORATION WITH MIT SLOAN

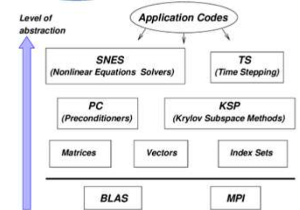
UNIVERSITÀ DEGLI STUDI DI NAPOLI PARTHENOPE

Thinking parallel

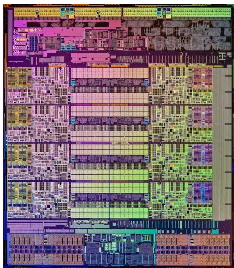


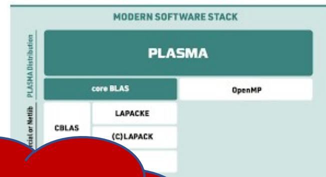


communications between processes



That’s all for today!





synchronization in memory access

synchronization and communication between different devices