ABINIT School 2026
Learning electronic structure calculations using ABINIT
Feb. 2 - 6 2026 - Bruyères-le-Châtel, France

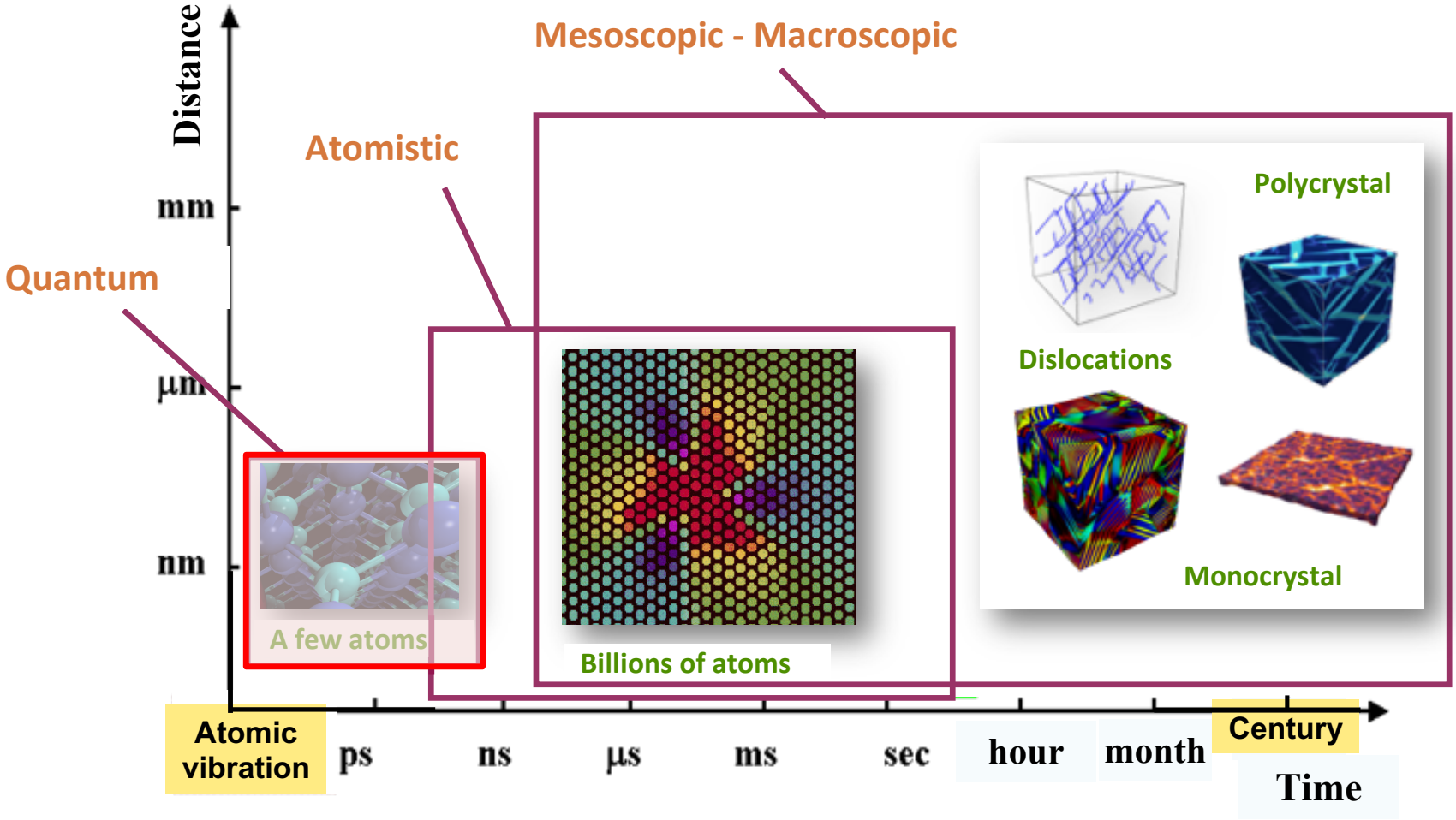
Density-functional Theory and High-performance computing

ABINIT on supercomputers

Clémentine Barat, Marc Torrent

CEA, DAM, DIF, F-91297 Arpajon cedex
Université Paris-Saclay, LMCE, F-91680 Bruyères-le-Châtel

Numerical simulation at quantum scale



DFT – A numerical challenge

Quantum physics

Calculation of electron properties

Schrödinger equation

Eigenvalue problem

$$\mathbf{H}(\rho)\psi_n = \varepsilon_n \psi_n$$

High dimensionality

An “N-body” problem

Diagonalization of a large matrix

Prohibitive computational cost

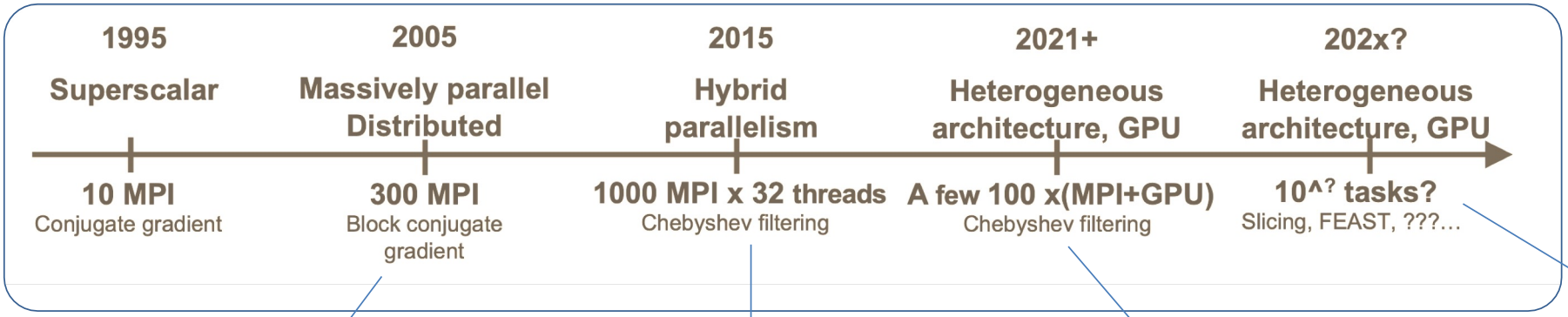
Iterative algorithms

« Matrix-free» algorithms

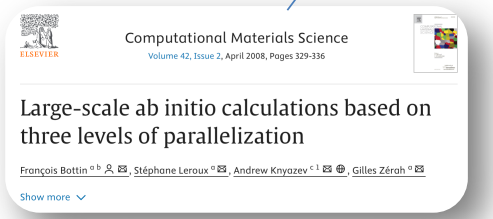
Step 1 : estimation of eigenvectors
Compute-bound

Step 2 : Rayleigh-Ritz method
Memory/Com-bound

ABINIT & High Performance Computing



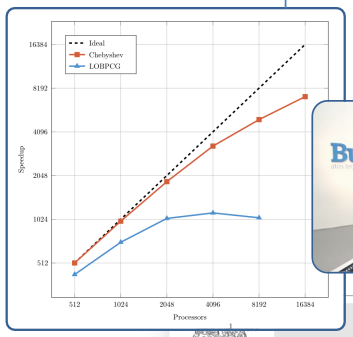
Spectrum slicing algorithm **2026**

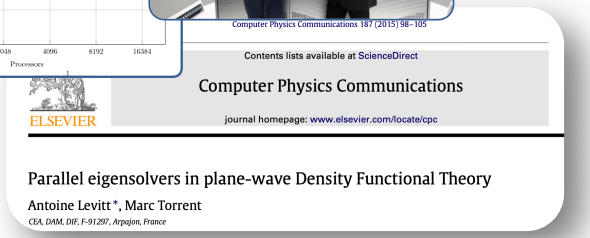

Computational Materials Science
Volume 42, Issue 2, April 2008, Pages 329-336

Large-scale ab initio calculations based on three levels of parallelization

François Bottin, Stéphane Leroux, Andrew Knyazev, Gilles Zérah

Show more

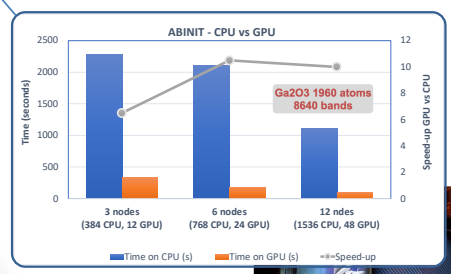



Computer Physics Communications 187 (2015) 98–105
Contents lists available at ScienceDirect
Computer Physics Communications
journal homepage: www.elsevier.com/locate/cpc

Parallel eigensolvers in plane-wave Density Functional Theory

Antoine Levitt *, Marc Torrent

CEA, DAM, DIF, F-91297, Arpajon, France



Performance: faster, better, ...

More or more efficient workers?

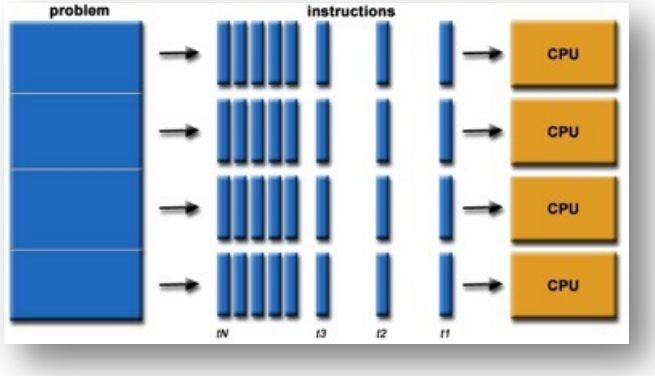
What is parallel computing?

Easy to say …

Simultaneous use of multiple compute resources to solve a computational problem

… not so easy to implement!

The problem has to be splittable in multiple parts which can be solved concurrently



Outline

What are super-computers made of?

- Parallel computer technologies

How to measure the parallel efficiency

- Speedup, efficiency, scalability

DFT parallelization strategy

- Time consuming parts, parallelism levels

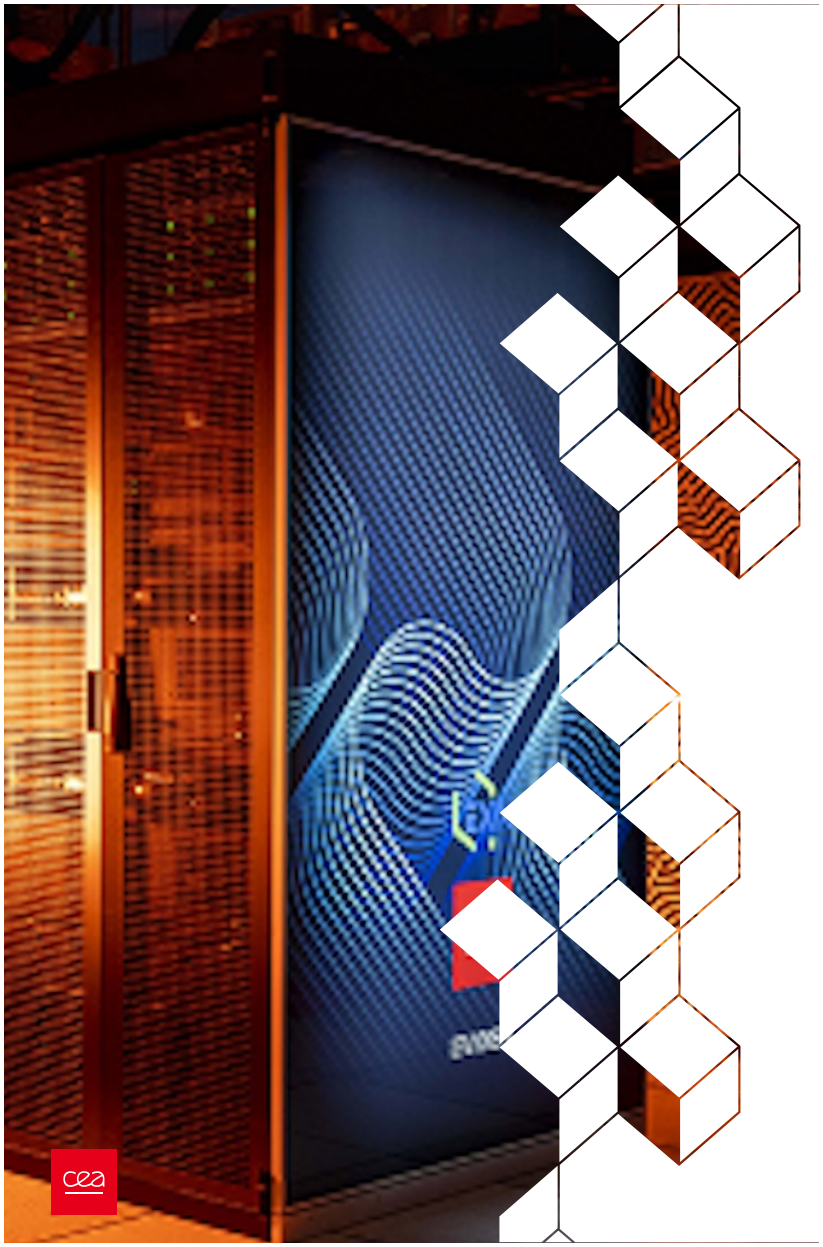
Iterative diagonalization algorithms

- Minimization, filtering

Parallelism with ABINIT

- Howto

Performance (examples)



Supercomputers?

Key concepts

Improving the compute processing

How to measure performance?

Traditional Measure of computing performance: FLOPS

- FLoating point OPerations per Second

How to increase the FLOPS of a computer?

- Do more operations per second
  → Increase the frequency!
- Do several operations simultaneously (overlap)
  → Use more computing units!
  → Use vectorization!

Warning 1: the power cost of frequency

Much is better than faster

- Power increases as **Frequency^3** ($P \propto f.V^2 \propto f^3$)
  → Clock rate is limited!

- Power is a **limited factor** for supercomputers
  → Around 3-5W per CPU nowadays

- **Multiple slower devices** are preferable than one superfast device!
  → Multiples computing units per CPU!

The power cost of frequency

	Cores	Hz	(Flop/s)	W	Flop/s/W
Superscalar	1	1.5 ×	1.5 ×	3.3 ×	0.45
Multicore	2	0.75 ×	1.5 ×	0.8 ×	1.88

Warning 1: the power cost memory transfer

Memory is a limiting factor

CPU cache: 40 GB/s, RAM: 20 GB/s, network: 3.5 GB/s

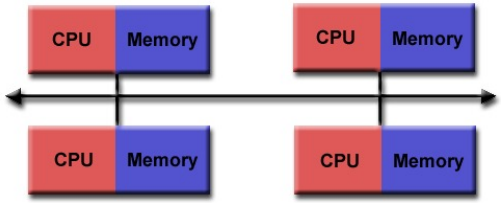
- **Memory evolves less than computational power**
  90's: 0.25 FLOPS/Byte transferred, nowadays: 100/Byte transferred
- **Moving data has a power cost**
  Moving data in RAM costs 2 nJ
  Communicating data (network) costs 3 nJ
- **Random access costs much more than strided access**
  Strided access triggers prefetchers, reduces the latency

Computing a data is cheaper than fetching it in memory

Improving the data processing

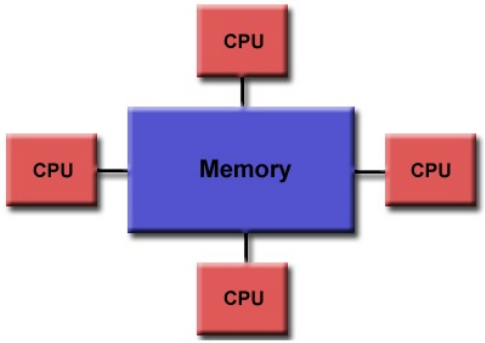
Memory management

Distributed Memory



- Private memory
- Processors operate independently
- Data transfer should be programmed explicitly (MPI)
- Relies on network performances

Shared Memory



- Memory is common to all processors
- Tasks operate concurrently on data
- Relies on bandwidth performances

Parallel computing – What do we need?

What do we want?

⇢ *Solve our problem in multiple concurrent tasks*

For that, we need:
- **A parallel computer** – A task for the computer vendor
- **An adapted software** – A task for the programmer

What do we really want?

⇢ *If we could have **N processing units** (compute+memory), we would like one calculation be be **N times faster**!*

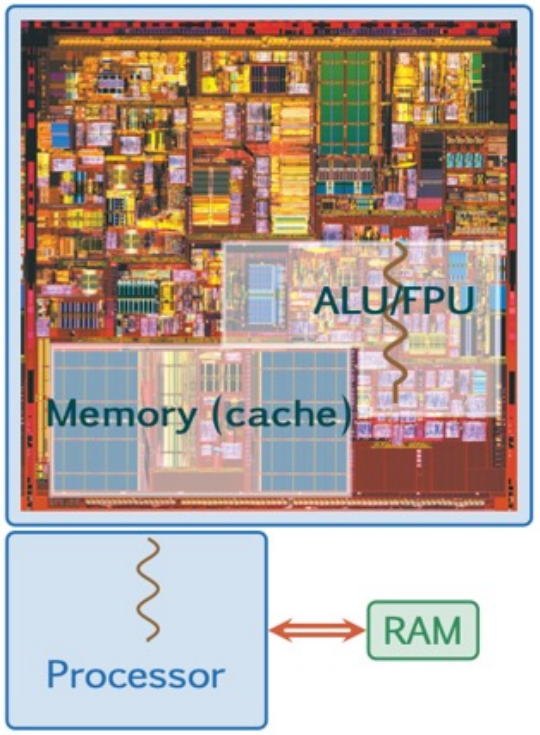
2 processing units are twice faster than one!
1 processing unit working twice faster spend 8 times more power!

But, what is a processing unit on a (super)computer?

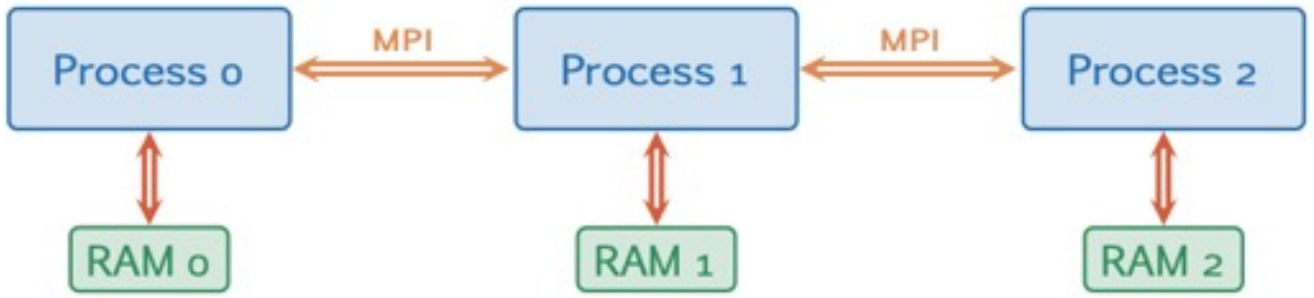
A basic processor design (old fashion)

- Arithmetic and Logic Unit
- Floating-Point Unit
- Memory (small)
- Controllers
- …
- RAM is far from the processor

- 1 processor (CPU) has 1 core!



Building a supercomputer with old CPUs



Message Passing paradigm

- Distributed memory model: process X cannot access RAM of processor Y
- For 100,000 CPUS, need for a very efficient communication network!

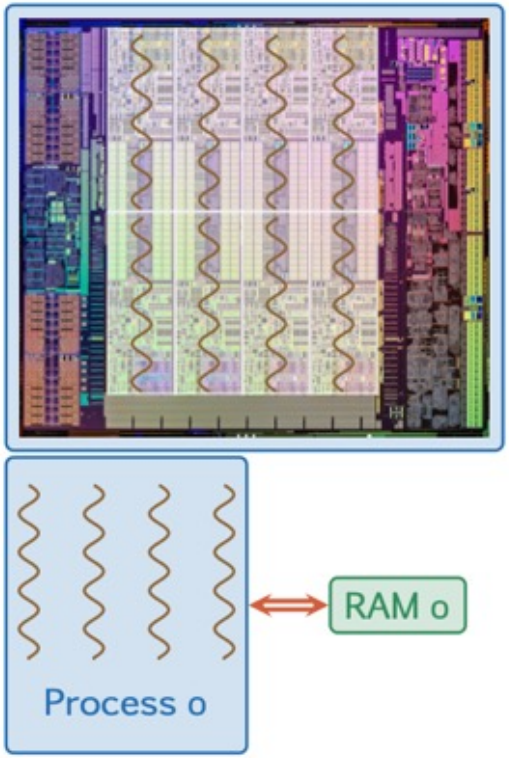
How to use it?

- Install a **MPI** library and compile the code with it.
- Launch:
  mpirun –n N executable

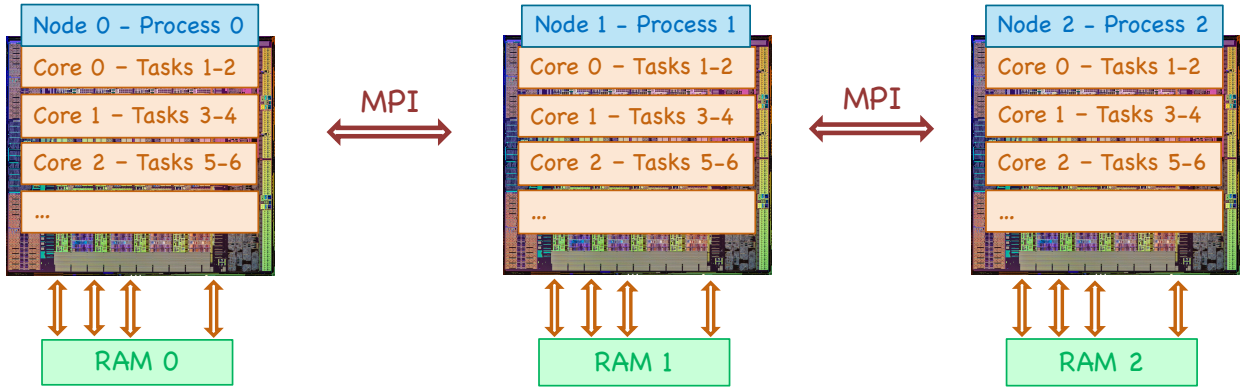
A modern processor design: « manycore »

- 1 processor has several cores
  (nowadays: 8 to 300)
- 1 core = ALU/FPU/cache memory
- Each core may have 2+ threads (concurrent tasks)
- All the cores share the RAM memory
- Core can be slow but highly vectorizable

- Note : the core may be grouped by “sockets”.
  Sharing memory is easy inside a socket;
  it is not from one socket to the other
  ⟶ *Non Uniform Memory Access* (NUMA)



Building a supercomputer with modern CPUs



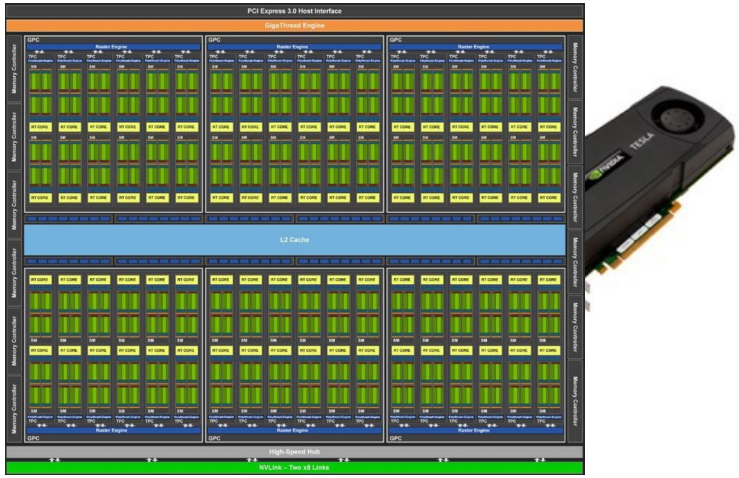
Hybrid parallelism

- Distributed memory between nodes
- Shared memory inside a node (beware to NUMA effect)
- Need to know the computer architecture to run a code!

How to use it?

- Select the number of concurrent tasks on a node (ex.: **openMP**): **export OMP_NUM_THREADS=x**
- Launch the code in hybrid mode: **mpirun –n N –c x executable**

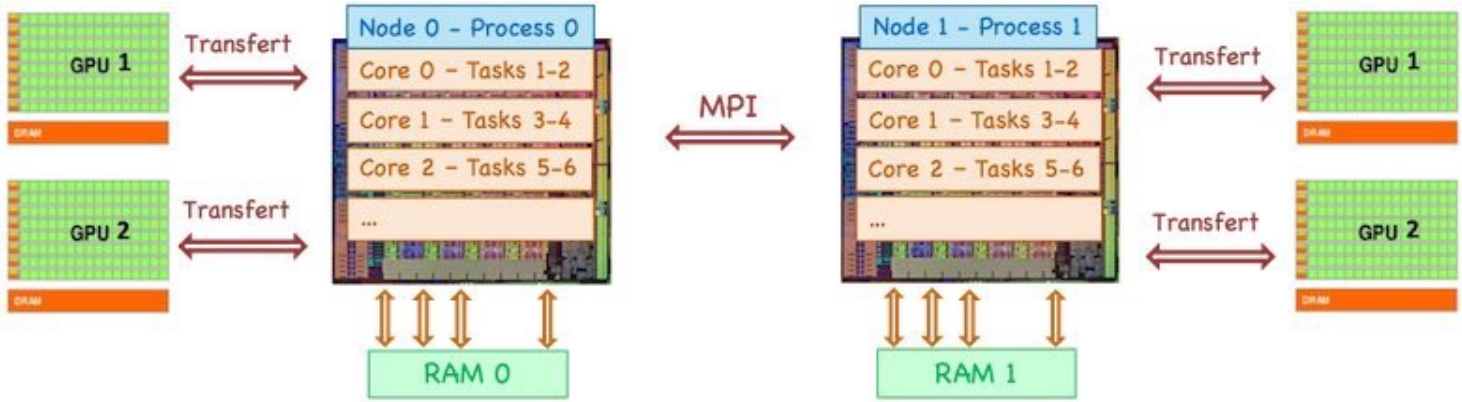
Graphic processing unit (GPU)



- A GPU is a Highly parallel multi-processor
- It has its own memory
- It is defined by the system as an external device
- It is optimized for doing compute-intensive and highly-parallel computation (simple tasks)

- It has less control, more computation units, less compute capabilities, less flexibility than a CPU
- The GPU programming model is SIMD *Single Instruction Multiple Data*

Building a supercomputer with CPUs and GPUs



Hybrid CPU+GPU parallelism

- Distributed memory between nodes
- Shared memory inside a node
- Offloaded use of the GPU computing resources

Let's play

Differents devices – Different parallel paradigms
Speed vs energy consumption

or

or

How to program GPUs?

- **Native kernel-based (low-level) models**
  Write explicit compute kernels for GPU execution
  Strengths: highest performance for vendor-specific features, need strong GPU arch. knowledge
  Examples: CUDA (NVIDIA), HIP (AMD), ROCm (AMD), Metal (Apple), etc
- **Directive-based models**
  Annotate code with special directives to offload parts of the computation to the GPU
  Strengths: ease of use, minimal code changes, incremental porting, portability
  Examples: OpenACC, OpenMP

ABINIT choice

- **Performance library-based models**
  Hardware-agnostic way to support multiple GPU architectures
  Strengths: write code once, balance performance and portability
  Examples: Kokkos, OpenCL, SYCL, Kokkos, RAJA, etc
- **High-level language bindings**
  Friendly APIs and programming abstractions in higher-level languages
  Strengths: speed up prototyping and scientific workflows
  Examples: PyCUDA, CuPy, Numba, CUDA.jl, AMDGPU.jl, etc

ABINIT & GPU

Programming model

`openMP offload`

Directive-based programming

Multi-platform support

Few dependencies

Porting strategy

Favor batch processing

Use vendor libraries
(linear algebra, FFT)

Use a spectral filtering algorithm

Target architectures

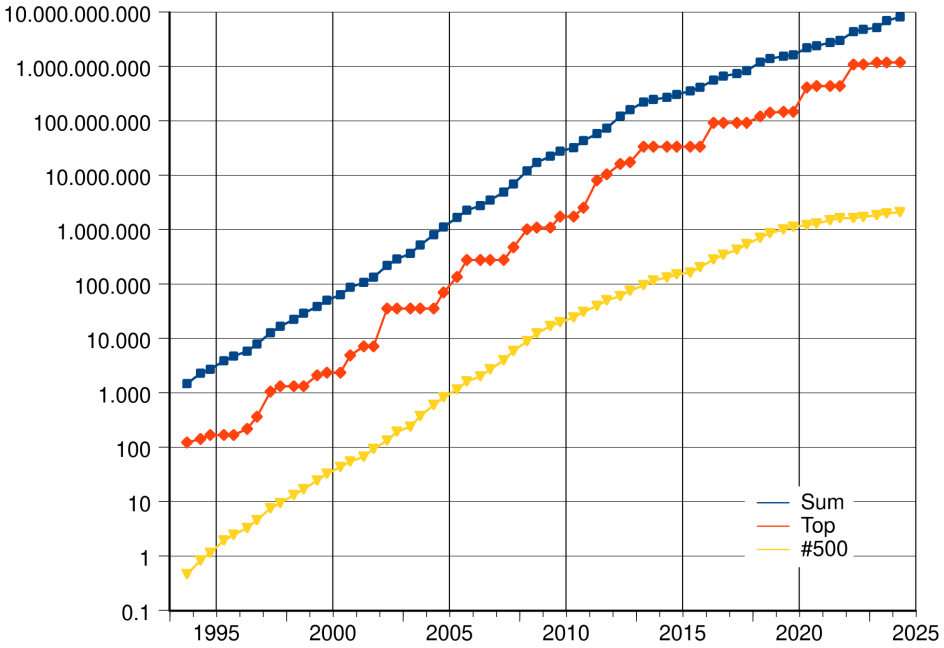
NVIDIA and AMD, INTEL soon

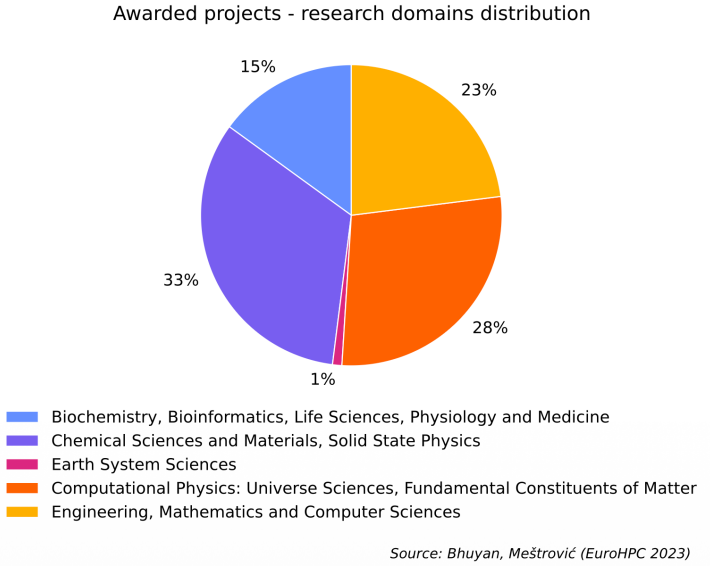
`CUDA, ROCm`

`nvhpc,llvm,cray`

Supercomputer efficiency increases continuously

Supercomputer TOP 500 – Moore's law



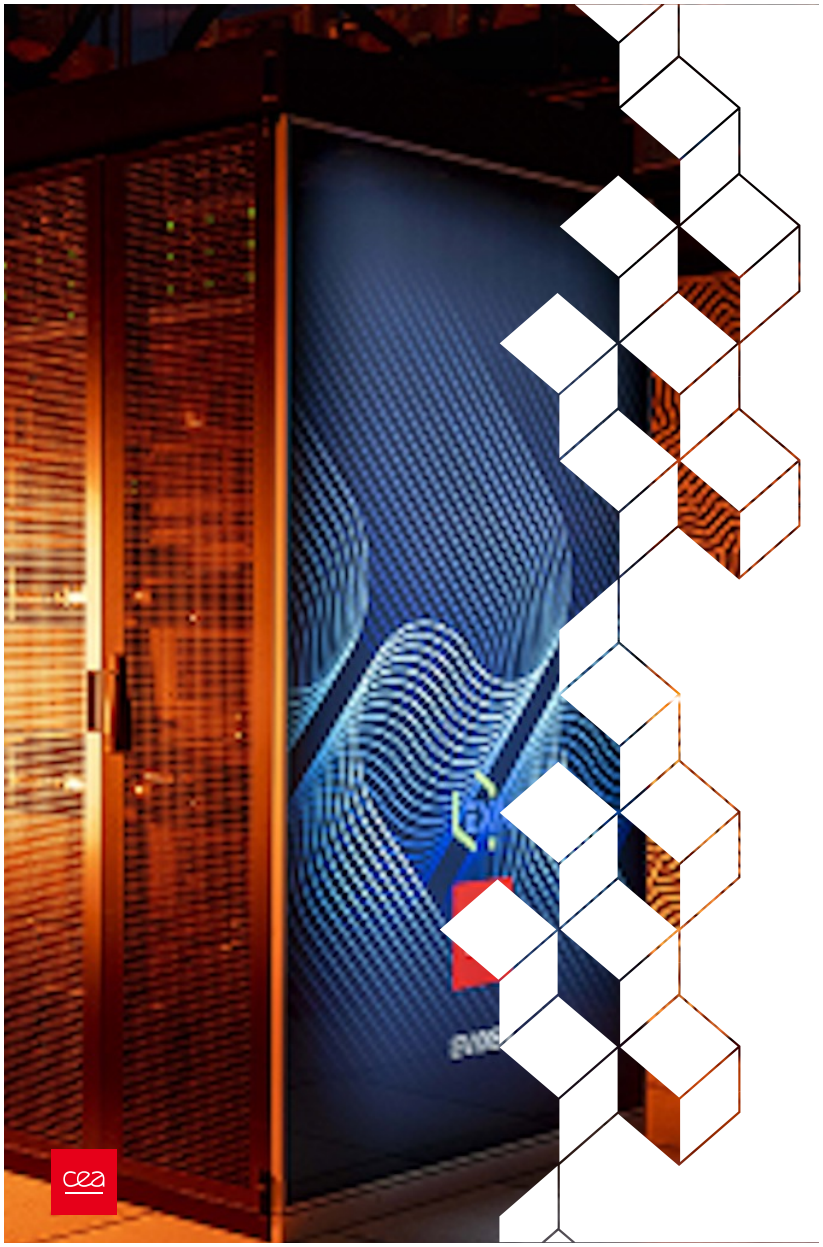


...Codes must be in continuous evolution

Hardware and software are closely related

Exercise:

- Take a given computational problem
- Write a code at a time t_0.
  Solve the problem on a computer.
- Freeze your code and wait some time $t_1 - t_0$
- Take a new computer at time t_1.
  Solve again the same problem.
- What happens to your performances?



How to measure a code efficiency?

Metrics, laws, ...

3 ways to measure code performances

Complementary indicators

Speedup

Scaling efficiency

Scalability

These performance indicators tell us how efficient the parallelism is.

- Is the code adapted to massive parallelism?
- Do we correctly use it?

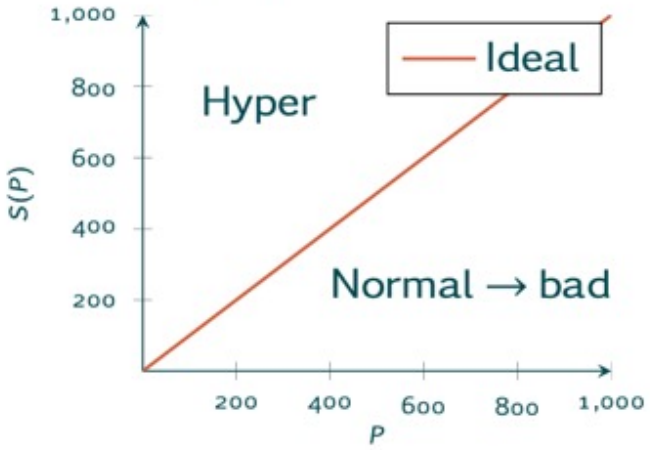
Speedup

Relative performance

The speedup is defined by

$$S(P) = \frac{T(1)}{T(P)} \qquad (1)$$

where $T(P)$ is the execution time on P cores.



- The closer to the straight line, the better
- Hyper speedup : cache/memory effect
- Bad speedup : time consuming communication, not enough parallel parts, ...

Speedup – Amdahl's law

Theoretical speedup

Gene Amdahl
(1922-2015)

What if the code is only parallelized at α% ?
The sequential execution time is:

$$T = (1-\alpha)T + \alpha T$$

On P cores the time will be at best :

$$T(P) = \underbrace{(1-\alpha)T}_{\text{sequential}} + \underbrace{\frac{\alpha}{P}T}_{\text{parallel}}$$

Thus, the speepup will be :

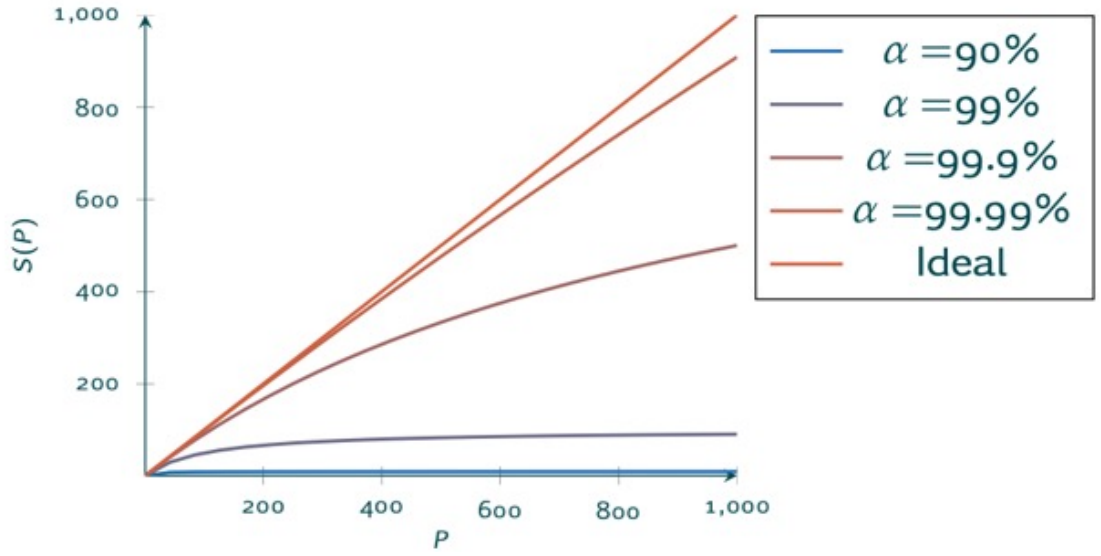
$$S(P) = \frac{T(1)}{T(P)} = \frac{T(1)}{(1-\alpha)T(1) + \frac{\alpha}{P}T(1)} = \frac{1}{1-\alpha+\frac{\alpha}{P}}$$

Speedup – Amdahl's law

Speedup limit not easy to reach

Theoretical limit of the speedup

$$S(P) = \frac{1}{1 - \alpha + \frac{\alpha}{P}} \qquad (2)$$

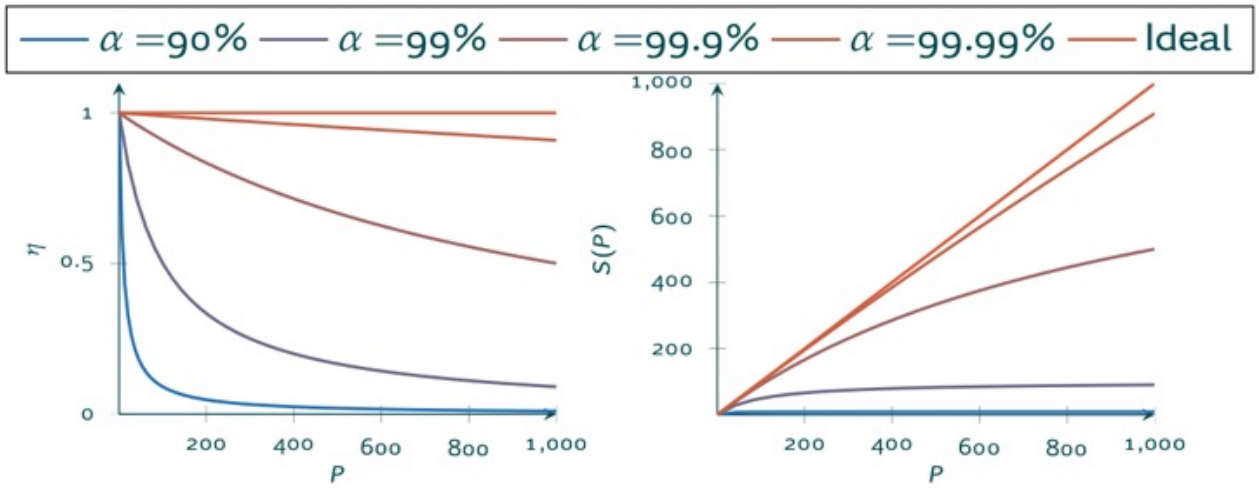


Parallel efficiency

Real vs ideal speedup

The scaling efficiency η is defined as :

$$\eta = \frac{S(P)}{P} = \frac{T(1)}{PT(P)} \begin{cases} \in [0;1] & \rightarrow \text{normal} \\ > 1 & \rightarrow \text{hyper} \end{cases} \qquad (3)$$



Scalability

Increasing work AND ressources

What is a scalable code ?

There are 2 ways of defining the scalability :

- Strong scaling : The work load is the same but the number of workers increase :

$$\eta = \frac{S(P)}{P} = \frac{T(1)}{PT(P)} \text{ should stay close to } 1.$$

- Weak scaling : The work is increased in the same way as the number of workers :

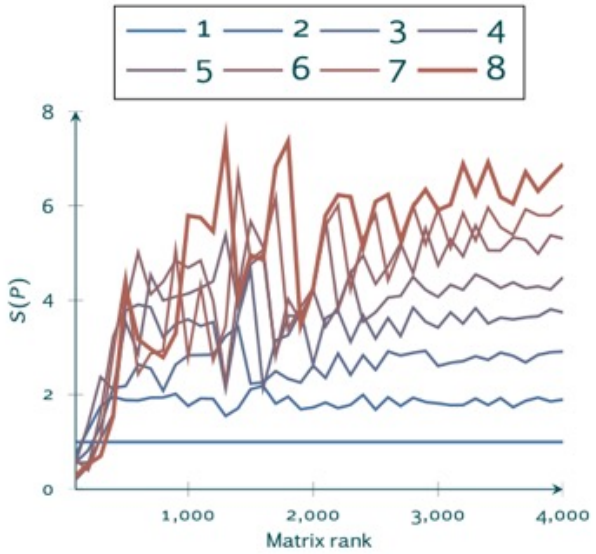
$$S(P) = \frac{T(1)}{T(P)} \text{ should stay close to } 1$$

Parallel performance: a real case

Testing Matrix-Matrix multiplication

Test of the so called GEMM

GEneral Matrix Matrix product



Good scaling is reached only if each "worker" has enough data to work on !

Limiting factors of a code

Code boundaries

A code can be:

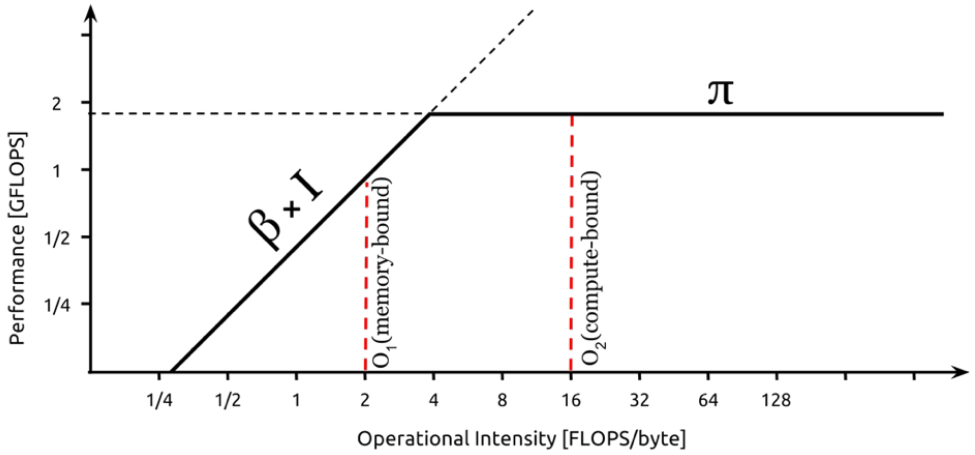
- **Compute-bound**
  Time is principally determined by the speed of the processor
  High processor utilization
- **Memory-bound**
  Time is principally determined by the amount of free memory required
  Too intensive memory access, full memory, too low memory bandwidth
- **I/O-bound**
  Time is principally determined by the period spent waiting for input/output
  Processes are waiting for data
  RAM I/O bound (~memory-bound), Disk I/O bound

Roofline model

An intuitive and visual performance model for a code

- Work (W): number of operations
- Memory traffic (Q): number of bytes of memory transferred
- Arithmetic intensity (I) : $I = W/Q$

- Peak performance (π): expressed in FLOPS, derived from benchmarking
- Peak bandwidth (β): expressed in bit/s, derived from architecture manual



How to check a parallel code?

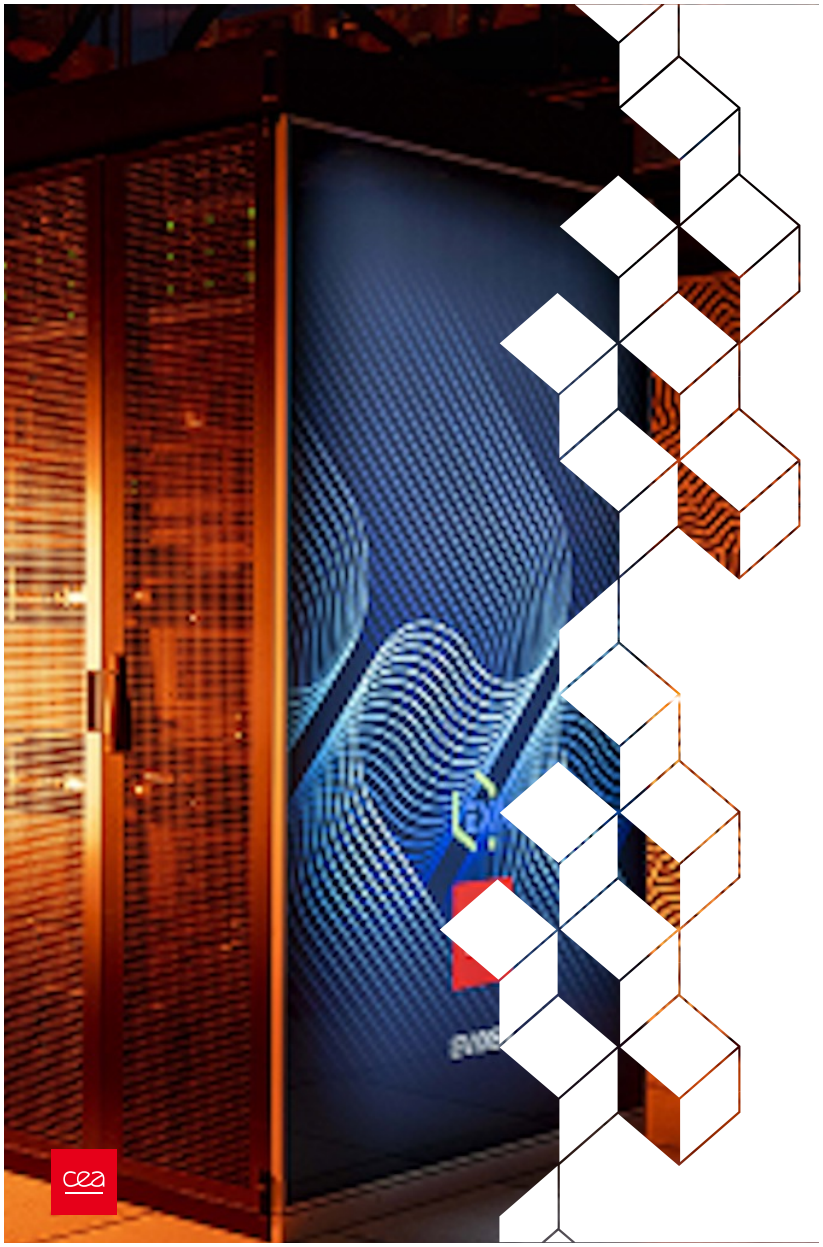
How to deal with the performance indicators

Use all indicators

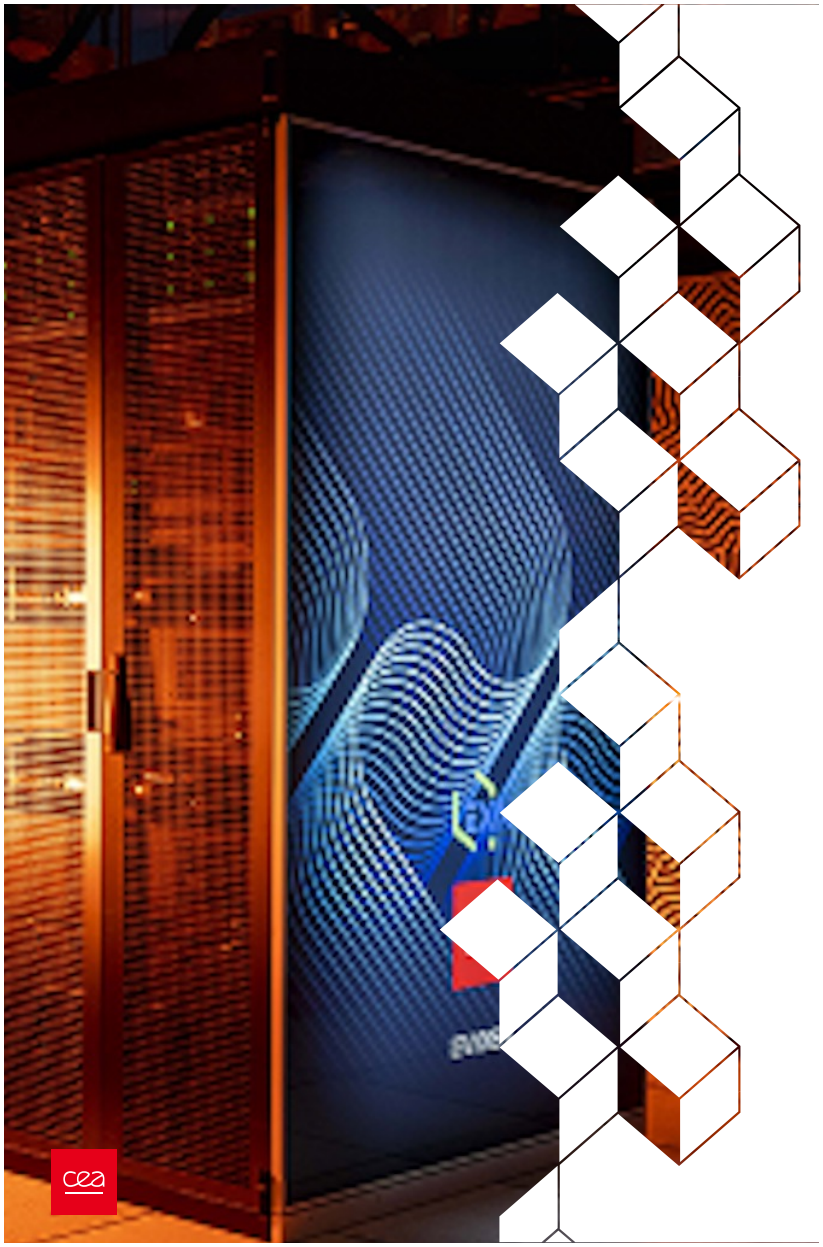
- A computationally inefficient code has a (artificial) high speedup but low efficiency

Define significant tests

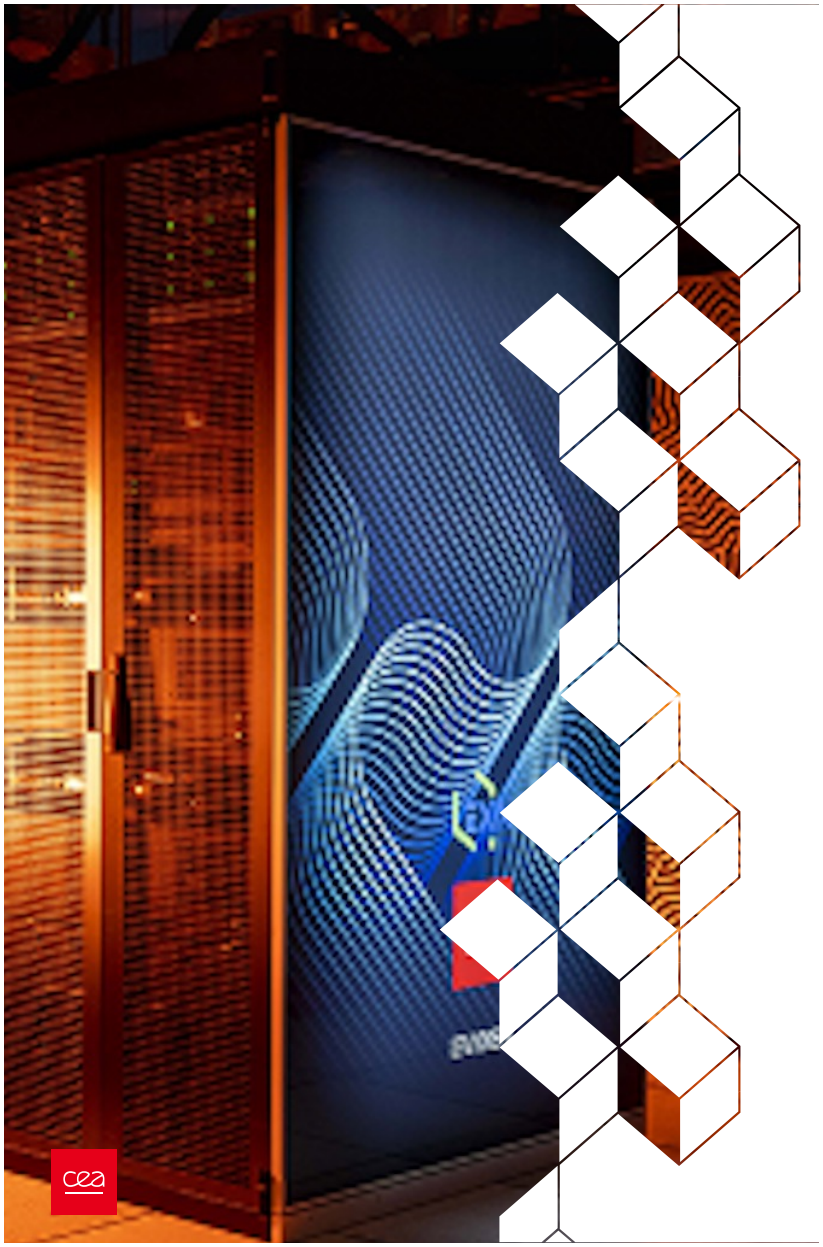
- Low scaling, strong scaling
- Compute bound, memory bound, …



DFT parallelization strategy

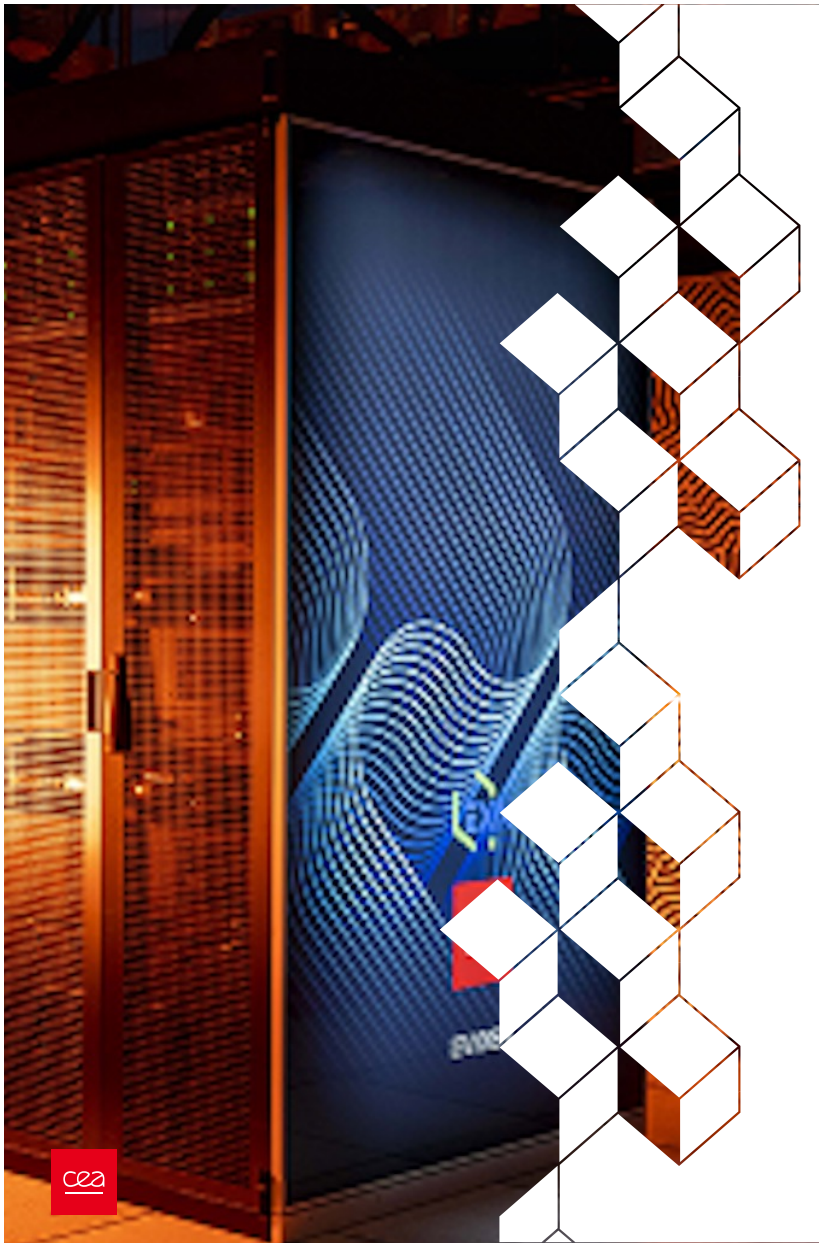


Iterative diagonalization algorithms



Parallelism with ABINIT

How to ...



ABINIT parallel performances