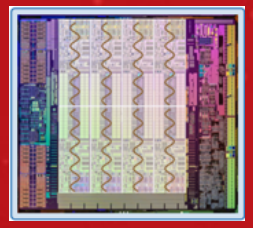
DFTK school 2022
Numerical methods for density-functional theory simulations
August 29-31, 2022, Paris, France

High-performance DFT simulations

FROM RESEARCH TO INDUSTRY

Marc Torrent

CEA, DAM, DIF. F-91297 Arpajon France
Matter under Extreme Conditions Laboratory, Paris-Saclay university, France



PERFORMANCE AT WORK – FASTER, BETTER, …

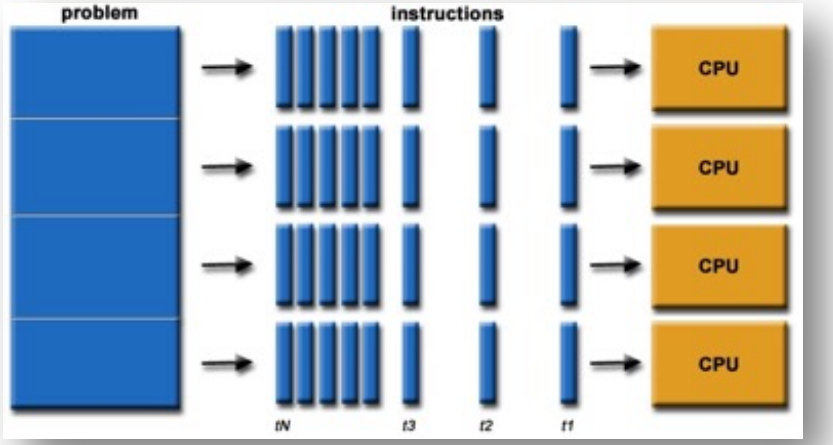
OUTLINE

- High performance computing?
- How to measure a code efficiency?
- Plane-wave DFT parallelization strategy
- Iterative diagonalization algorithms
- Examples

Supercomputers?
Key concepts

WHAT IS PARALLEL COMPUTING?

- **Easy to say…**
  - Simultaneous use of **multiple compute resources** to solve a **computational problem**
- **… but not so easy to implement!**
  - The problem has to be splittable in multiple parts which can be solved **concurrently**
  - Parts are executed simultaneously on different Compute Processing Units
  - Each part is made of a series of instructions; Instructions are **compute operations** or **memory transfers**



IMPROVING THE PERFORMANCE OF COMPUTE PROCESSING

- **Traditional Measure of computing performance: FLOPS**
  - **FL**oating point **OP**erations per **S**econd

- **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!

THE POWER COST OF FREQUENCY

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

- *Power increases as Frequency*^{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!

IMPROVING THE PERFORMANCE OF MEMORY TRANSFER

- **Measure of memory performance**
  - Access time (ps)
  - Transfer speed, bandwidth (Byte/s)
  - Latency (ns)

- **How to speed up memory access?**
  - Speed up the access
    - → Change the technology
    - → Implant the memory closer to processing unit!
  - Increase bandwidth
    - → Increase memory clock rate,
    - → Increase number of “channels”!
  - Decrease the latency
    - → Improve “switches”, improve network speed

THE POWER COST OF MEMORY TRANSFER

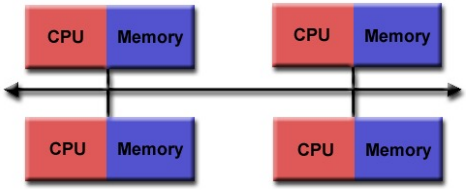
- **Memory transfer, some data:**
  - Bandwidths : CPU cache: 40 GB/s, RAM: 20 GB/s, network: 3.5 GB/s
  - Memory evolves less than computational power:
    90's: 0.25 FLOPS/Byte transferred, nowadays: 100/Byte transferred
- **Power cost of data movement**
  - Computation of a FMA costs 50 pJ
  - Move data in RAM costs 2 nJ
  - Communicating data (network) costs 3 nJ
- **Random vs strided access**
  - Random access is very low ~ equivalent to 200 CPU cycles
  - Strided access triggers prefetchers, reduces the latency

Computing a data is cheaper than fetching it in memory

Recomputing data is faster than fetching them randomly in memory

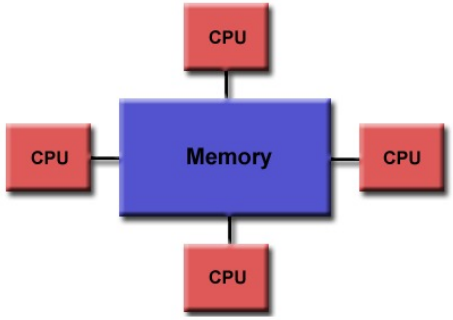
HOW TO DISTRIBUTE DATA IN MEMORY

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

WHAT DO WE WANT?

Solve our problem in multiple concurrent tasks

Save data access and data transfer

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 workers are twice faster than one!

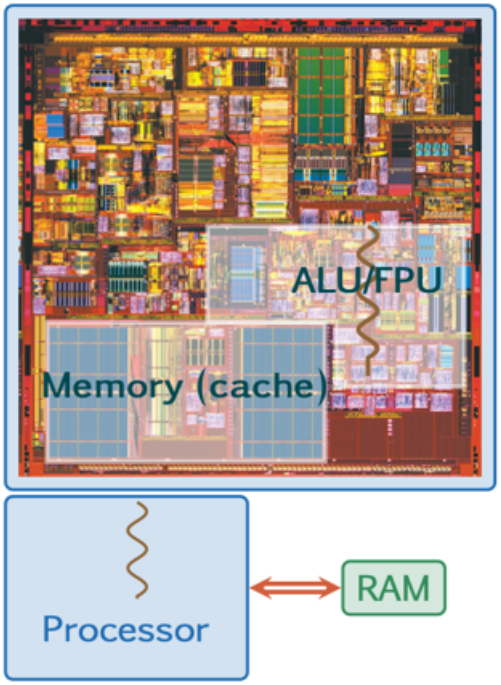
1 worker working twice faster spend 8 times more power.

But, what is a worker on a (super)computer?

A BASIC PROCESSOR DESIGN (OLD FASHIONED)

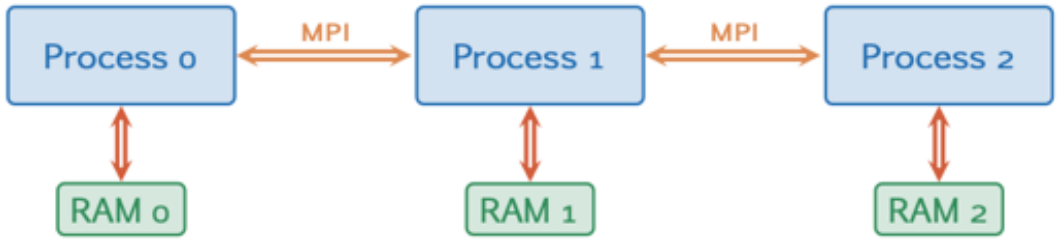
- Arithmetic and Logic Unit
- Floating-Point Unit
- Memory (small)
- Controllers
- …
- RAM is far from the processor

- 1 processor (CPU) has 1 core!



MPI = Message Passing Interface

A communication protocol



Message Passing paradigm

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

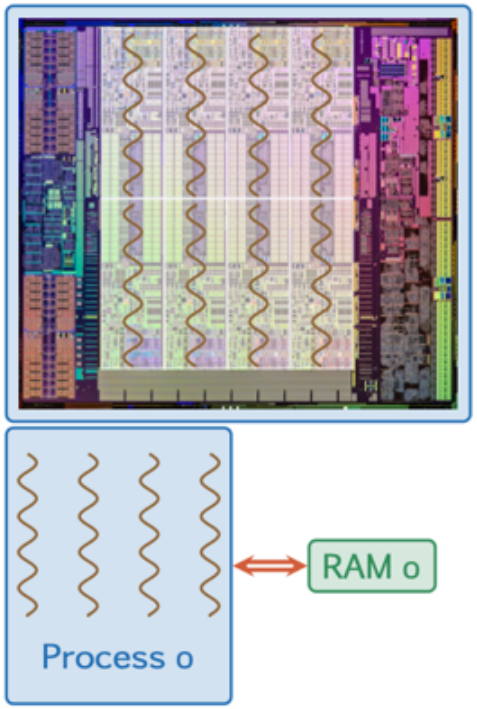
How to use it?

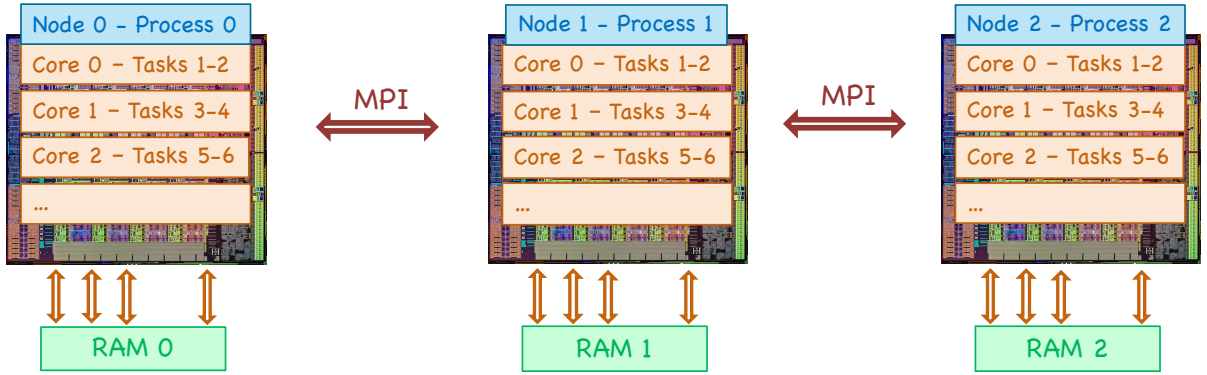
- Install a MPI library and compile the code with it.
- Launch:
  `mpirun -n N executable`

A MODERN PROCESSOR DESIGN : « MANYCORE »

- 1 processor has several cores
  (nowadays: 8 to 256)
- 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)





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!

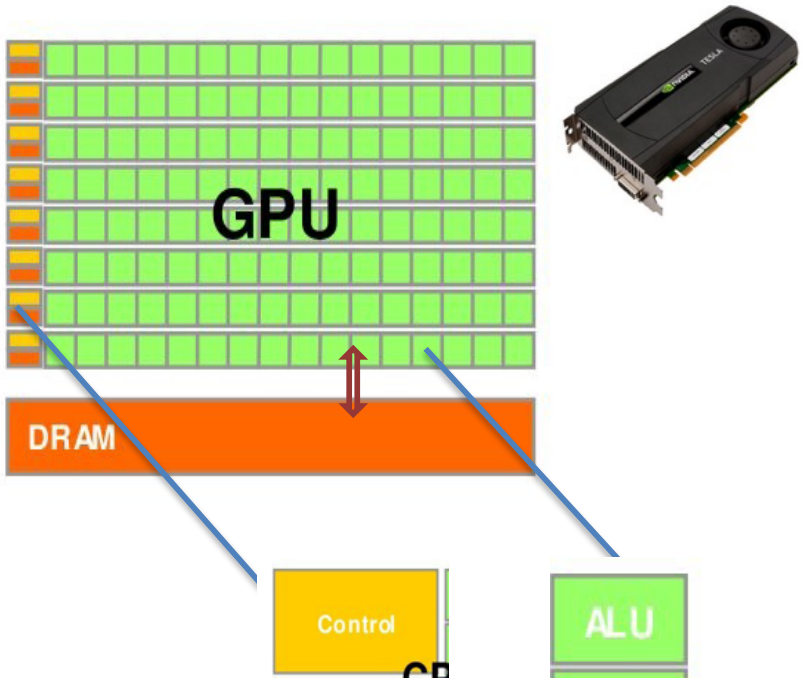
How to use it?

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

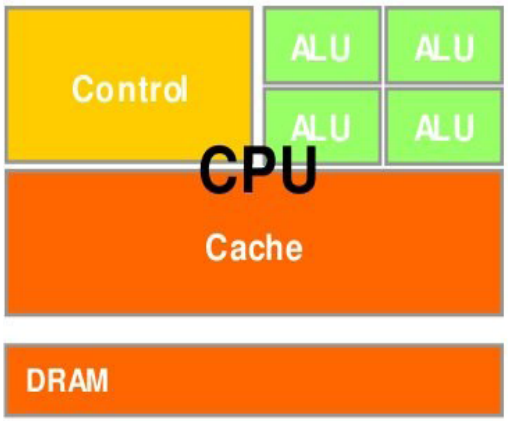
GPU: GRAPHICS PROCESSING UNIT

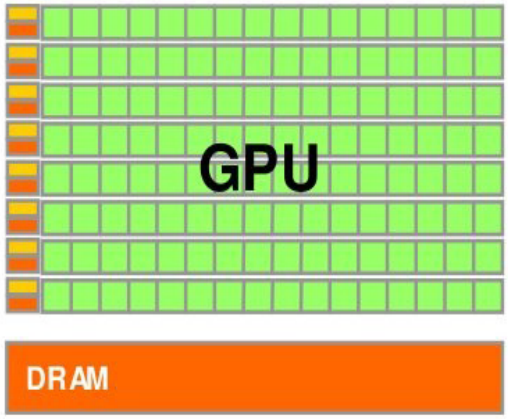
- A GPU is a Highly parallel multi-processor
- It has its own memory
- It is defined by the system as an external device
- It is optimized for doing compute-intensive and highly-parallel computation

- It has less control, more computation units than a CPU



CPU *vs* GPU



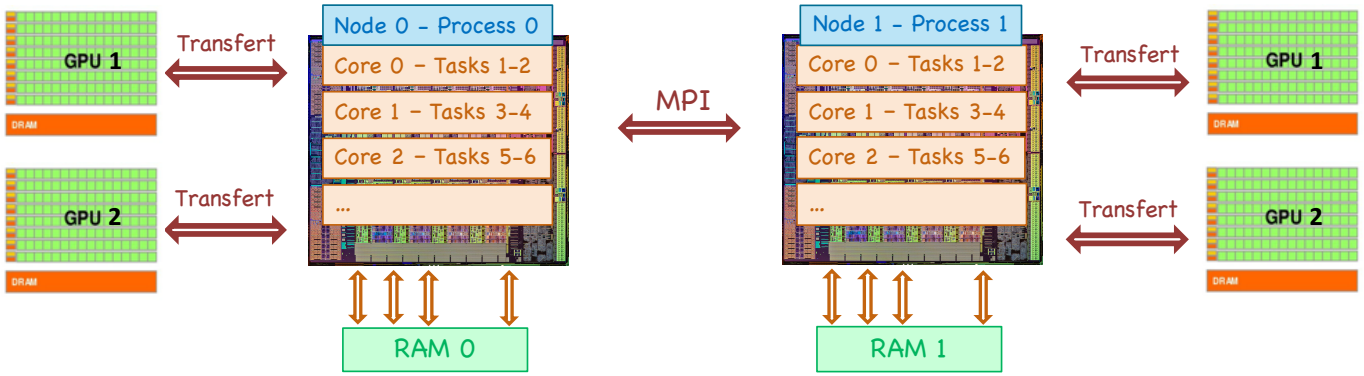


- Graphics processors are more and more sophisticated
- Software development tools are more and more accessible
- Huge computation power, at low energetic and financial costs

- CPU: optimized to execute a list of tasks as fast as possible
  Needs sophisticated flow control for synchronization
- GPU: optimized for compute-intensive and highly-parallel computation
  Less control, more computation units

GPU PROGRAMMING MODEL

- **Massively parallel computation**
  - SIMT: one single instruction is executed simultaneously by thousands of GPU threads (tasks) on different data
  - Few collaboration between threads
- **Explicit memory management**
- **Global scheme of a GPU code**
  - Memory allocation on GPU
  - Data transfer from CPU to GPU memory
  - Code execution by thousands of GPU threads
  - Recover results from GPU memory to standard memory



Hybrid CPU+GPU parallelism

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

How to use it?

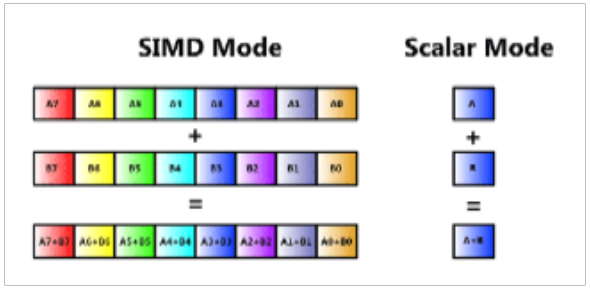
- Just run your code as for the hybrid CPU parallelism
- Activate GPU use at compilation level

What is vector computing?

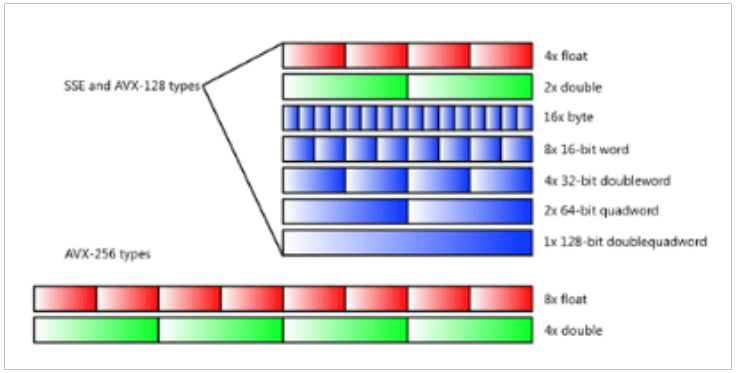
- Vectorization can be considered as a “hardware parallelization”, directly implemented in the processor unit.
- It generalizes operations on scalars to apply to vectors.
- Operations apply at once to an entire set of values.
- The processor uses a specific set of instructions: Advanced Vector Extensions (AVX)
- The size of vectors is hardware dependent.
  Recent processors use 512 bits vectors

VECTORIZATION

- Common vector operations (Implemented in all Hardware)
  - Addition, Multiplication
  - FMA (Fuse Multiply-Add) a ⟵ a + (b x c)
- Example for addition:



- Size of “vector” in recent hardware is increasing



VECTORIZATION

- **Vectorization improve performance but…**
  - Needs more transistors per surface unit in the chip
  - Power and heat accumulation increase linearly with vector size
  - → Frequency needs to be reduced!

- Needs suitable code!

Vectorizable

```
DO II=2,NMAX
  A(II) = B(II)+C(II)
  D(II) = E(II)-A(II-1)
END DO
```

Not vectorizable

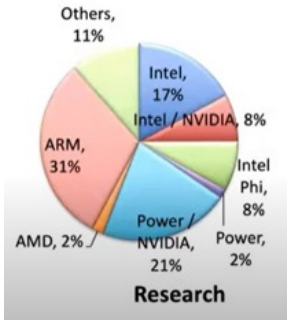
```
DO II=2,NMAX
  D(II) = E(II)-A(II-1)
  A(II) = B(II)+C(II)
END DO
```

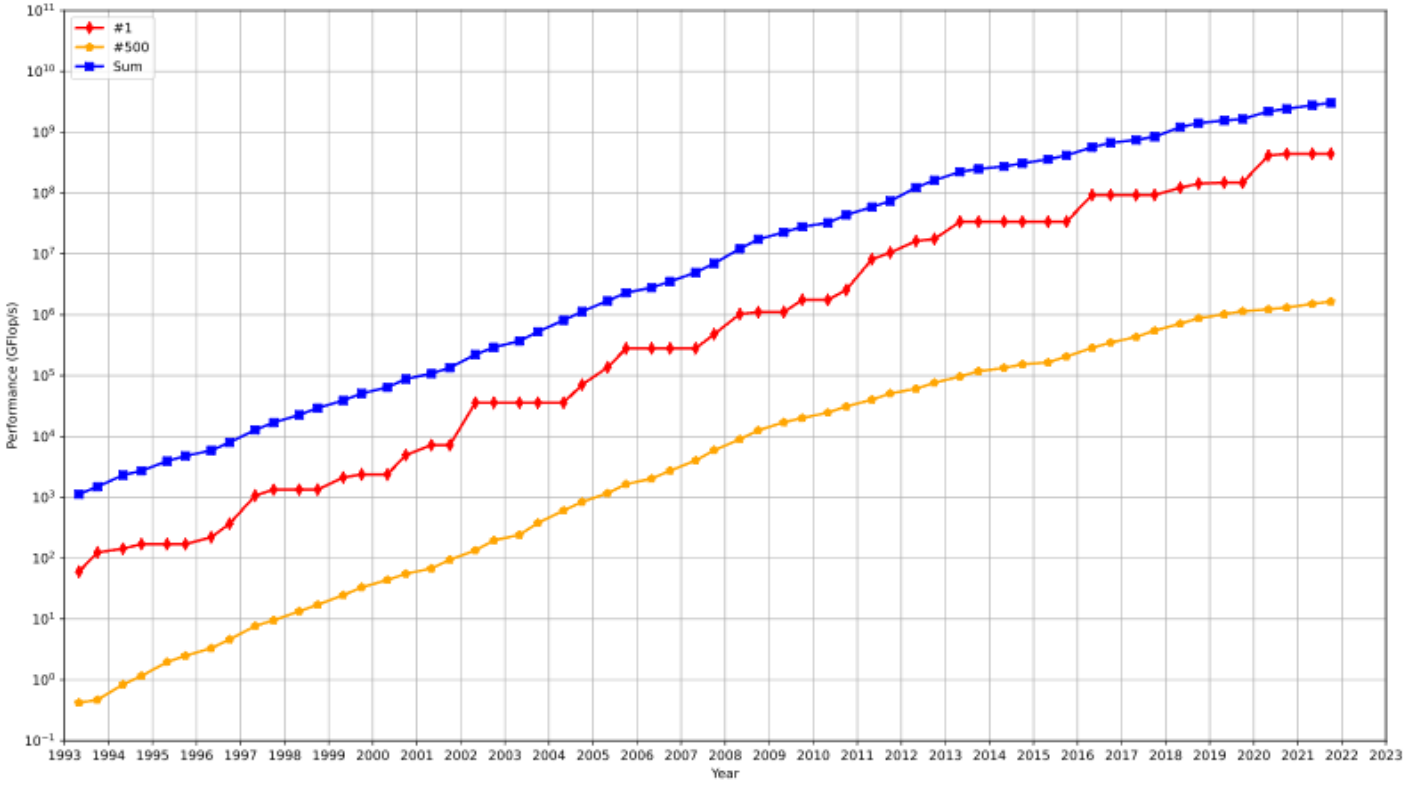
Need `A` before it has been computed

- Needs code changes to help the compiler!
  → Beware to data interdependency
- Order of operations is non deterministic
  → Round-off errors are unpredictable

SUPER-COMPUTERS EFFICIENCY INCREASES CONTINUOUSLY…

Super-computers world TOP 500





...CODES MUST BE IN CONTINUOUS EVOLUTION

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?

3 WAYS TO MESURE PARALLEL PERFORMANCES

- **Speedup**
- **Scaling efficiency**
- **Scalability**

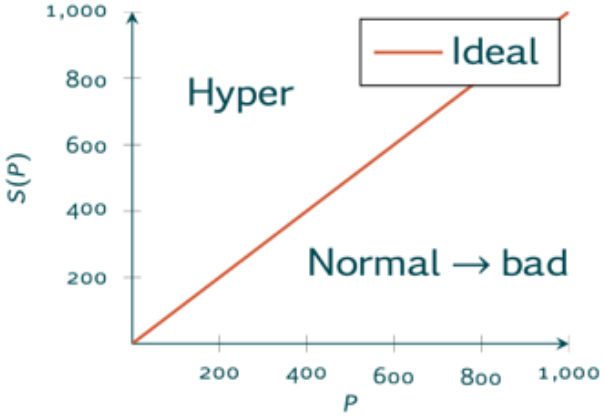
- These performance indicators tell us how good/efficient the parallelism is.
- Is the code adapted to massive parallelism?
- Do we correctly use it?

For a given case test

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, ...

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 :

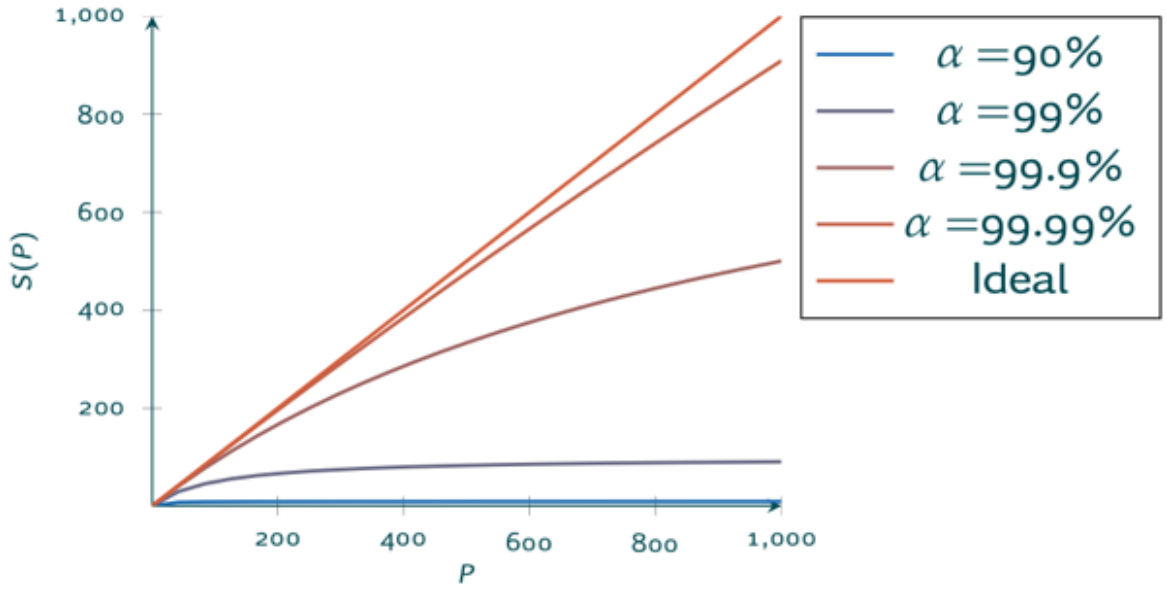
$$T(P) = \underbrace{(1-\alpha)T}_{\text{sequential}} + \underbrace{\frac{\alpha}{P}T}_{\text{parallel}}$$

Thus, the speepup will be :

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

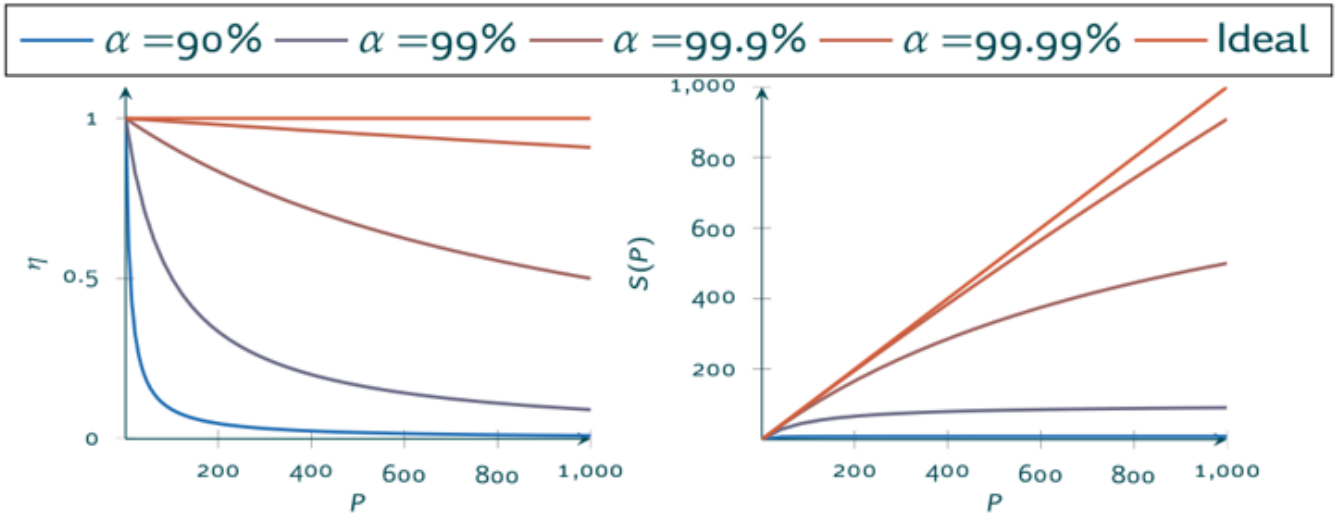
Theoretical limit of the speedup

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



The scaling efficiency η is defined as :

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



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 :

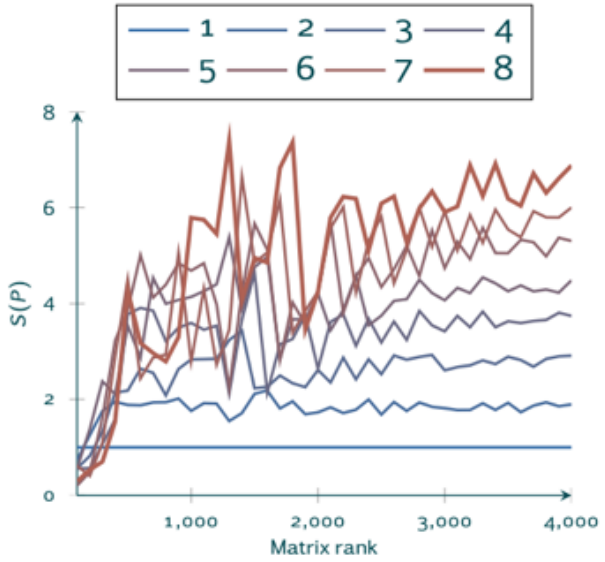
$$\eta = \frac{S(P)}{P} = \frac{T(1)}{PT(P)} \text{ should stay close to } 1.$$

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

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

Test of the so called GEMM

GEneral Matrix Matrix product



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

A code can be:

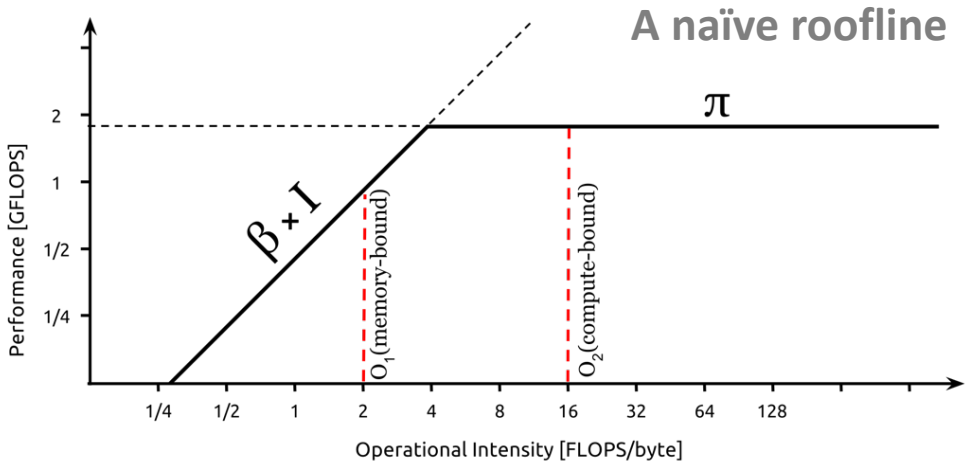
- **Compute-bound**
  Time is principally determined by the speed of the processor
  High processor utilization
- **Memory-bound**
  Time is principally determined by the amount of free memory required
  Too intensive memory access, full memory, too low memory bandwidth
- **I/O-bound**
  Time is principally 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/compute kernel

- 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

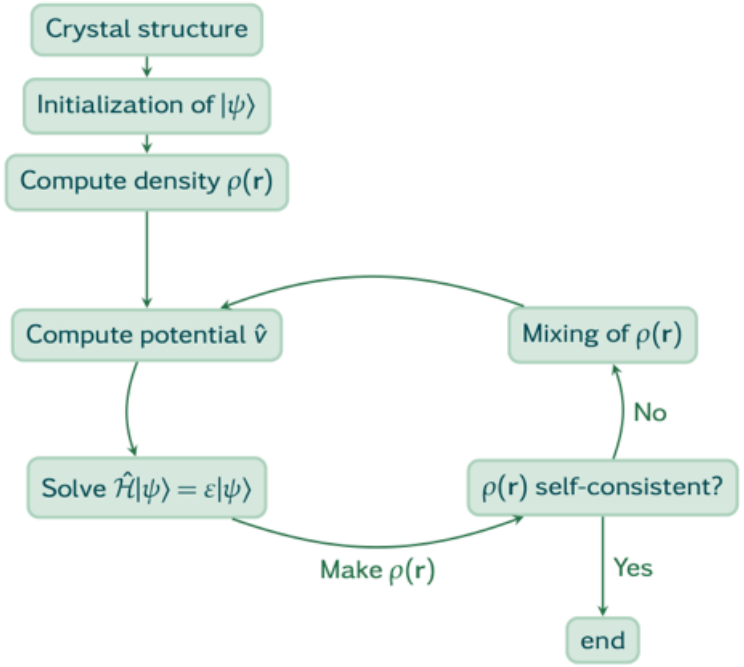


HOW TO DEAL WITH THE PERFORMANCE INDICATORS

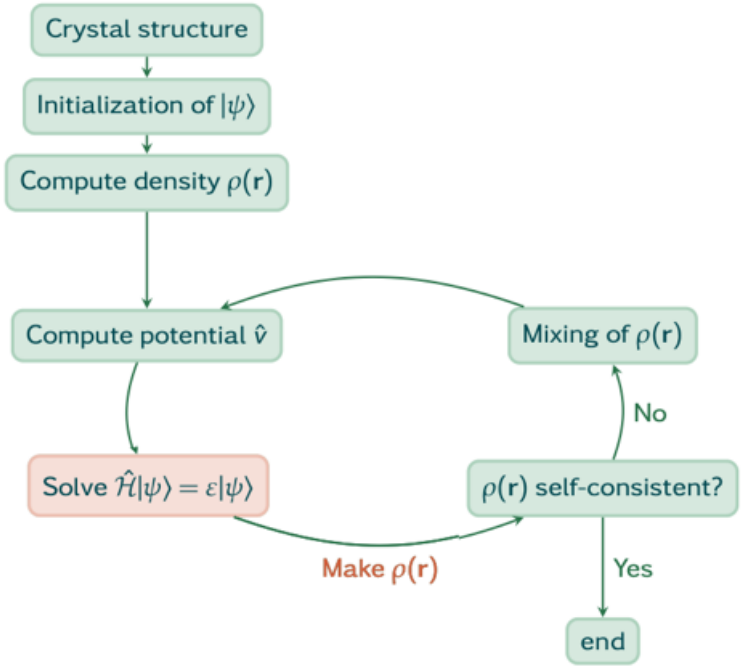
- **Use all indicators**
  - A computationally inefficient code has a (artificial) high speedup but low efficiency
- **Define significant tests**
  - Low scaling, strong scaling
  - Compute bound, memory bound, …

Plane-wave DFT parallelization strategy

The main self-consistent loop



The main self-consistent loop



A typical time analysis – Plane-wave DFT code

Sequential mode

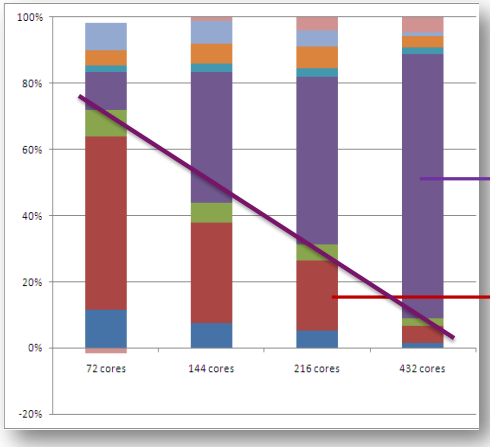
Initialization		1%
Computation of new density	$V(r) \longrightarrow \rho(r)$	98%
– Wave equation	$H\|\psi_k> = \varepsilon_k\|\psi_k>$	97%
– Iterative solver		7%
– Application of H	: $H\|\psi_k>$	90%
– Building of density	$\rho(r) = \sum_{ik}\|\psi_{ik}\|^2$	1%
Computation of new potential	$\rho(r) \longrightarrow V(r)$	0.5%
Finalization		0.5%

Parallel efficiency is dominated by:

- Algorithm to find eigenvectors $\hat{\mathcal{H}}_{\mathbf{k}}|\psi_{\mathbf{k}}\rangle = \varepsilon|\psi_{\mathbf{k}}\rangle$
- Hamiltonian application $\hat{\mathcal{H}}_{\mathbf{k}}|\psi_{\mathbf{k}}\rangle$:

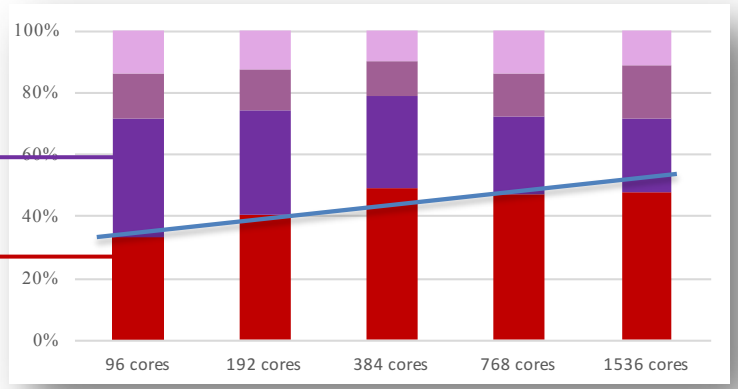
DENSITY-FUNCTIONAL THEORY – MOST CONSUMING PARTS

- Parallel performances are strongly dependent on algorithm used to solve the wave equation



Iterative solver (eigenvalues)
Without Hamiltonian application

Hamiltonian application
Linear algebra, FFT



Algorithm 1
Block conjugate gradient

Algorithm 2
Spectrum projection

Repartition of time in in a DFT calculation
varying the number of CPU cores (MPI, strong scaling)

PLANE-WAVE DFT - WHAT CAN WE USE TO PARALLELIZE?

Electronic density formula

$$\rho(\vec{r}) = \sum_{\substack{\sigma \\ spins}} \sum_{\substack{n \\ Bands}} \left[\int_{\substack{\vec{k} \\ Reciprocal \\ space}} \left(\left| \sum_{\substack{\vec{g} \\ Plane \\ waves}} \left(C_{n,k}(\vec{g}) \cdot e^{i\cdot(\vec{k}+\vec{g})\vec{r}} \right) \right|^2 \cdot d\vec{k} \right) \right]$$

Parallelization level:	*Spins (σ)*	*Electronic states (n)*	*Brillouin zone sampling (k)*	*Plane waves*
System-size scaling:	*≤ 2*	*$\propto S$*	*$\propto 1/S$*	*$\propto S$*
Parallel speedup:	*Linear*	*???*	*Linear*	*Poor*

Each level of parallelization has its own parallel efficiency

PLANE WAVE HAMILTONIAN APPLICATION – A SCALABLE COMPUTE BOUND PROCEDURE!

$$H = -\frac{1}{2}\nabla^2 + \underbrace{(V_{ext} + V_H + V_{xc})}_{V_{local}} + \underbrace{\sum_{ij} |p_i\rangle D_{ij} \langle p_j|}_{V_{non-local}}$$

Kinetic operator

Dot product

$$\left\langle \vec{g} \left| -\frac{1}{2}\nabla^2 \right| \psi \right\rangle = \frac{1}{2} \sum_g g^2 C_{\vec{g}}$$

Local operator

Two Fast Fourier Transform

$$\langle \vec{g} | \tilde{v}_{eff} | \psi \rangle = \underbrace{\int d\vec{r} \cdot e^{-i\cdot\vec{g}\cdot\vec{r}} \cdot \left(\tilde{v}_{eff}(\vec{r}) \cdot \underbrace{\left[\sum_{\vec{g}'} \left(c_{\vec{g}'} e^{i\cdot\vec{g}'\cdot\vec{r}} \right) \right]}_{\text{FFT}} \right)}_{\text{FTT}}$$

Non-local operator

Matrix-matrix multiplications

$$\langle \tilde{p}_i | \psi \rangle = \sum_{\vec{g}'} \langle \tilde{p}_i | \vec{g}' \rangle \langle \vec{g}' | \psi \rangle$$

p g X g ψ

$$\langle \vec{g} | V_{NL} | \psi \rangle = \sum_{i,j} \langle \vec{g} | \tilde{p}_i \rangle D_{ij} \langle \tilde{p}_j | \psi \rangle$$

g p X p ψ

- *Batch processing* can be applied
  i.e. several ψ_{ik} concurrently
  Highly scalable
- Use of external libraries (FFT, BLAS)
- **Need all plane-waves ($\vec{g}$) for some ψ**

EIGENSOLVER – A NON-SCALABLE MEMORY-BOUND & I/O-BOUND PROCEDURE

- Target computations : $1000<N_{states}<50\,000$, N_{PW}~100 000…250 000
- Direct diagonalization unachievable (~10^{12})
- In search of the eigenvectors associated with the lowest eigenvalues
- Need an iterative algorithm
- Different kinds of algorithms but all rely on linear/matrix algebra:

  $\langle\psi_{ik}|\psi_{jk}\rangle$, *matrix orthogonalization*, *matrix diagonalization*

  ➢ **communication + data movement in memory**

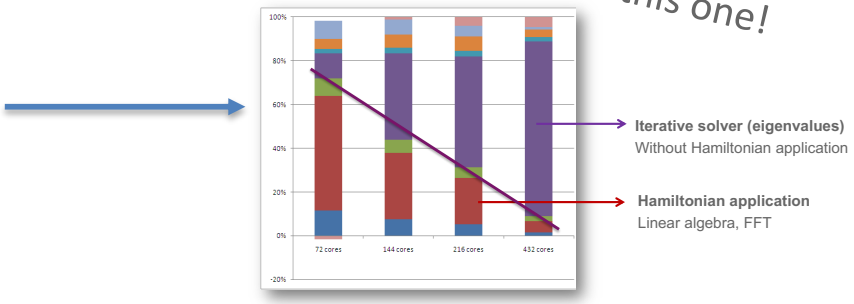
- Use of external libraries (BLAS, LAPACK)

See later

Need for an algorithm minimizing data movement and favoring computation!

- **Need all some plane-waves ($\vec{g}$) for all ψ**

Not this one!



TWO INTERNAL REPRESENTATIONS OF WAVE FUNCTIONS

- **Wave function components have to be distributed among processors**

  A too large array

- **Need for two data distributions for the wave functions**
  - Hamiltonian application ➠ **Need all plane-waves ($\vec{g}$) for some ψ**
  - Linear algebra ➠ **Need all some plane-waves ($\vec{g}$) for all ψ**
  - **Need for two data distributions**

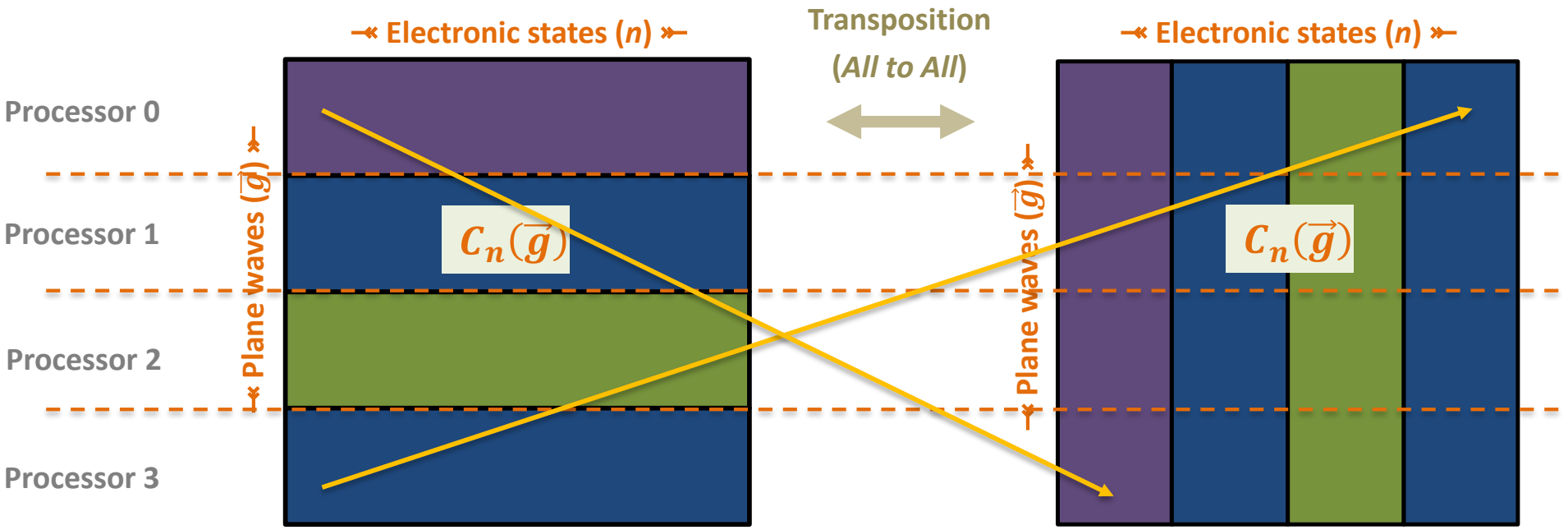
- **It is not possible to store twice the wave functions**

  ➠ Need to way to move from one distribution to the other

TWO INTERNAL REPRESENTATIONS OF WAVE FUNCTIONS

$$\psi_n(\vec{r}) = \sum_{\vec{g}} C_n(\vec{g})\, e^{i\vec{g}\cdot\vec{r}}$$

All processors communicate
with all processors!
Do it as few times as possible!



Iterative eigensolvers for DFT

Commissariat à l'énergie atomique et aux énergies alternatives - www.cea.fr

INTRODUCTION: RAYLEIGH-RITZ PROCEDURE

- Rayleigh-Ritz method: direct numerical method of **approximating eigenvalues/eigenvectors**
- Starting from a subset of vectors ϕ_n close to the eigenvectors ψ_n ,
  it allows to approximate these eigenvectors :

0. Find ϕ_n

1. Orthogonalize $\{\phi_n\}$ (if necessary)
2. Form the matrix $H = \phi^* H \phi$
3. Diagonalize H, get the eigenpairs $\left(\tilde{\lambda}_n, \tilde{\phi}_n\right)$
   This is a "small" eigenproblem
4. Compute the Ritz vectors $\tilde{\psi}_n = \phi_n \tilde{\phi}_n$
5. $\left(\tilde{\lambda}_n, \tilde{\psi}_n\right)$ are a good approximation of the (λ_n, ψ_n)
   Accuracy determined with: $\left\| H\tilde{\psi}_n - \tilde{\lambda}_n \tilde{\psi}_n \right\|$

STRUCTURE OF AN ITERATIVE SOLVER FOR DFT SELF-CONSISTENT CYCLE

- Find trial wave functions ϕ_n
  Iterative method

Application of **H**
Linear algebra

- Apply Rayleigh-Ritz procedure

Orthogonalization
Diagonalization

- Update the density
  i.e. the Hamiltonian operator

DFT self-consistent cycle

Each loop doesn't need to be individually converged

WHICH PROPERTIES FOR A HIGH PERFORMANCES ITERATIVE SOLVER

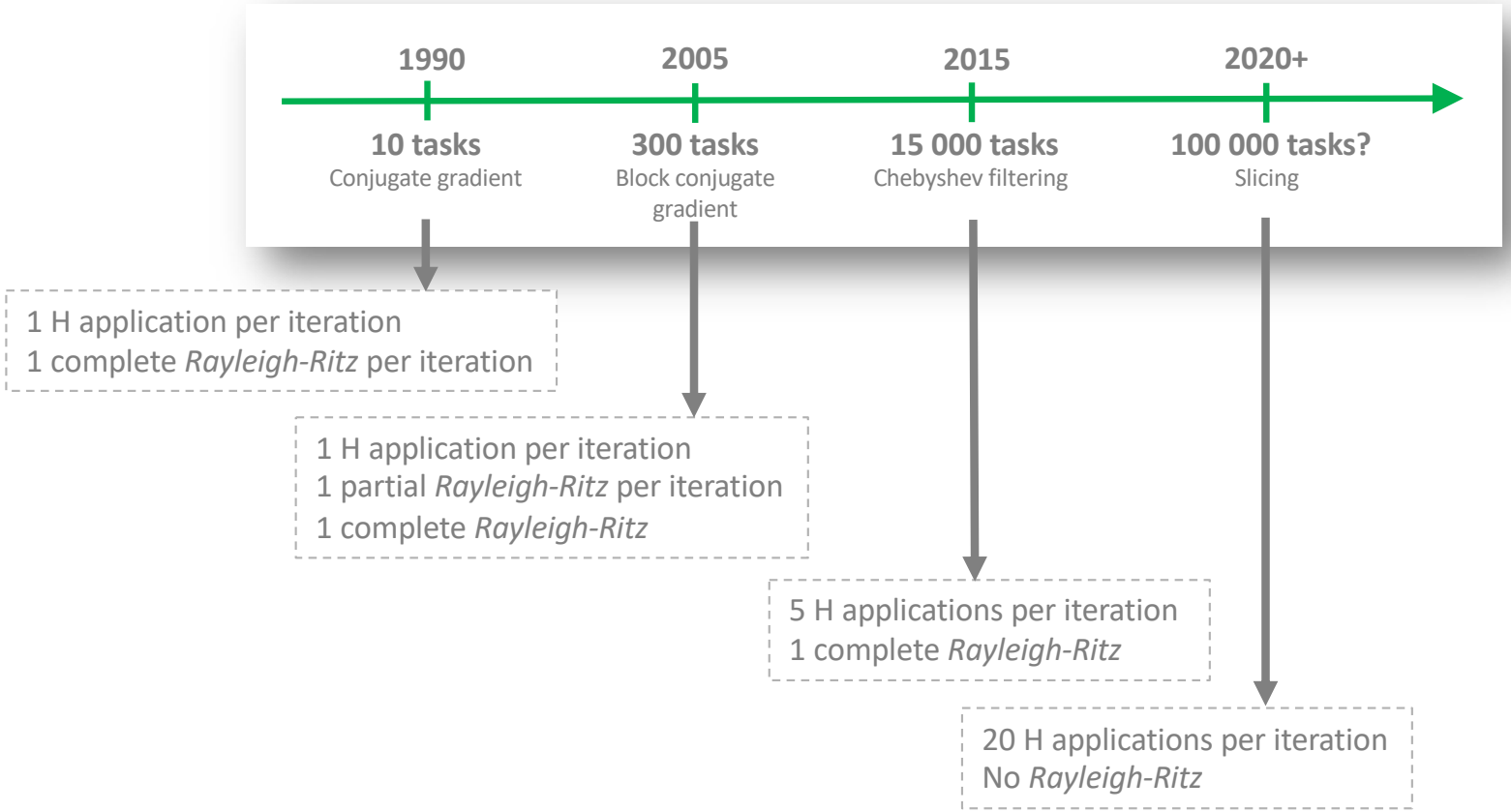
A solver should be compute-bound!

- Favor computation and distribute it.
  Redundant computations are not limiting
- Save communications between parallel processes.
- Optimize the data locality.
  Avoid data movement.
- Use standardized basic operations.
  Linear algebra, FFT, …

AVAILABLE TYPES OF ITERATIVE SOLVERS

- **Which differences between the algorithms**
  - Number of applications of the Hamiltonian during the search for trial wave functions
    Cost : computation
  - Size of space on which apply the Rayleigh-Ritz procedure
    Cost : memory access, communications
- **Available types of algorithms**
  - Minimization or diagonalization?
  - Number of Hamiltonian applications : 1 to 20 per iteration
  - Size of the Rayleigh-Ritz subspace : N_{states} to 1

EVOLUTION OF ITERATIVE SOLVERS



CONGUGATE GRADIENT – 1 TO 10 PROCESSORS

For j=1 to N_{iter}

  For i=1 to N_{states}

    - Compute the "conjugate gradient"
      Choose a minimization direction as the opposite to the gradient
      1 Hamiltonian application

    - Orthogonalize the gradient with respect to all ψ

    - Apply of the gradient to the current $\boldsymbol{\psi}$

  End For

  Apply Rayleigh-Ritz of the entire $\boldsymbol{\psi}$ space

End For

- 1 ***H*** application per ψ per iteration
- 1 complete Rayleigh-Ritz procedure per iteration

BLOCK CONGUGATE GRADIENT – 1 TO 300 PROCESSORS

Based on the conjugate gradient

The eigenvectors a grouped in blocks.
Each block is updated with a parallel procedure.

At the end, a full Rayleigh-Ritz procedure is applied to ensure
the complete orthogonality between blocks

BPCG : Block–Preconditionned Conjugate Gradient
In each block, the search direction is orthogonal
to already updated blocks

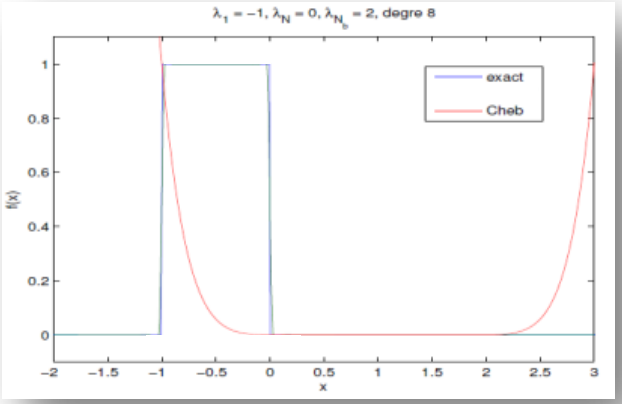
LOBPCG : Locally Optimal Block-Preconditionned Conjugate Gradient
In each block, the search direction is determined
with a Rayleigh-Ritz procedure

With block size=1, this is standard conjugate gradient

- 1 ***H*** application per iteration, done on several ψ in parallel
- 1 partial Rayleigh-Ritz procedure per iteration
- 1 complete Rayleigh-Ritz procedure at the end

SPECTRUM FILTERING – 100 TO 15000 PROCESSORS

- Algorithm based on a spectral projection of the eigenvalues
- Relies on the diagonalization of an auxiliary operator with a filtered (localized) spectrum of eigenvalues
- Several methods exists. One is based on a “Chebyshev filtering”
- Eigenvalues of H^n are λ^n $\longmapsto$ use a localized polynomial to filter



Yunkai Zhou *,**, Yousef Saad *, Murilo L. Tiago b, James R. Chelikowsky b
Journal of Computational Physics 219 (2006) 172–184

Antoine Levitt*, Marc Torrent
Computer Physics Communications 187 (2015) 98–105

CHEBYSHEV FILTERING – 100 TO 15000 PROCESSORS

For j=1 to N_{iter}

  For i=1 to N_{states}

    - Computation of $P_{cheb}(\boldsymbol{H})\psi$
    (polynomial of order 5 to 8)

  End For

End For

Apply Rayleigh-Ritz of the entire $P_{cheb}(\boldsymbol{H})\psi$ space
which has "filtered" eigen-components

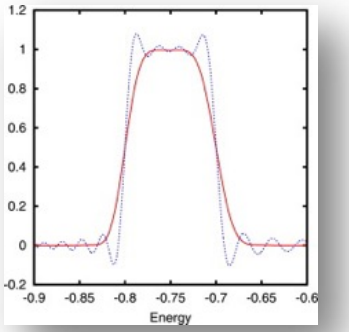
- 5 to 8 ***H*** applications per iteration, done on several ψ in parallel
- No Rayleigh-Ritz procedure during iteration
- 1 complete Rayleigh-Ritz procedure at the end

SPECTRUM SLICING METHOD – EXASCALE?

The eigenvalue spectrum is divided in several domains containing a small number of eigenvalues (possibly only one)

In each domain, an independent search of eigenvalues is performed. No global communication required.

Two known methods: *FEAST* or *Chebyshev* filter operator
Some dozen of Hamiltonian application to apply the filter



- Some 10 ***H*** applications per iteration, done on several ψ in parallel
- No Rayleigh-Ritz procedure

Schofield G., Chelikowsky J.R., Saad Y., *Computer Physics Communications* **183**, 497 (2012)

ITRATIVE SOLVERS - ISSUES

Convergence speed

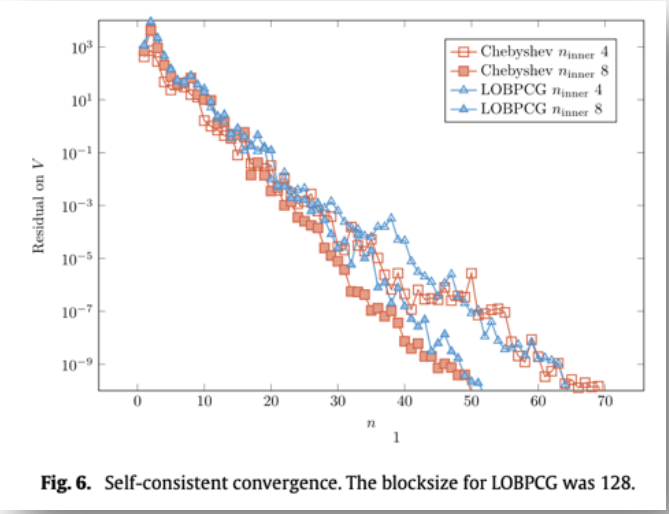
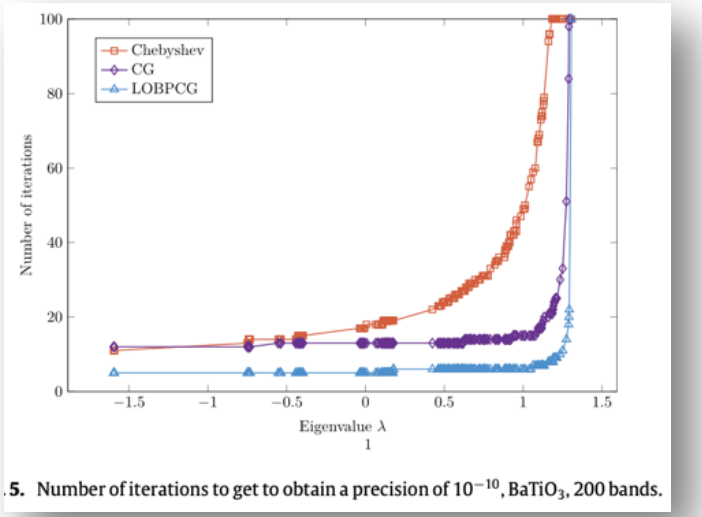


Fig. 6. Self-consistent convergence. The blocksize for LOBPCG was 128.

Precision on eigenvalues



5. Number of iterations to get to obtain a precision of 10^{-10}, $BaTiO_3$, 200 bands.

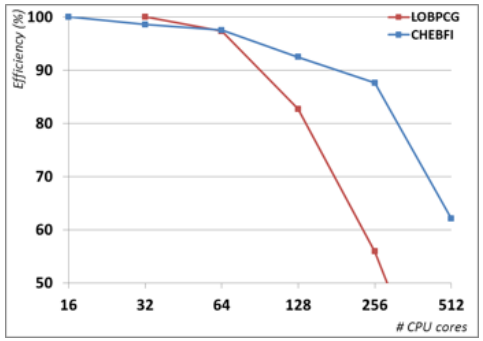
Performances

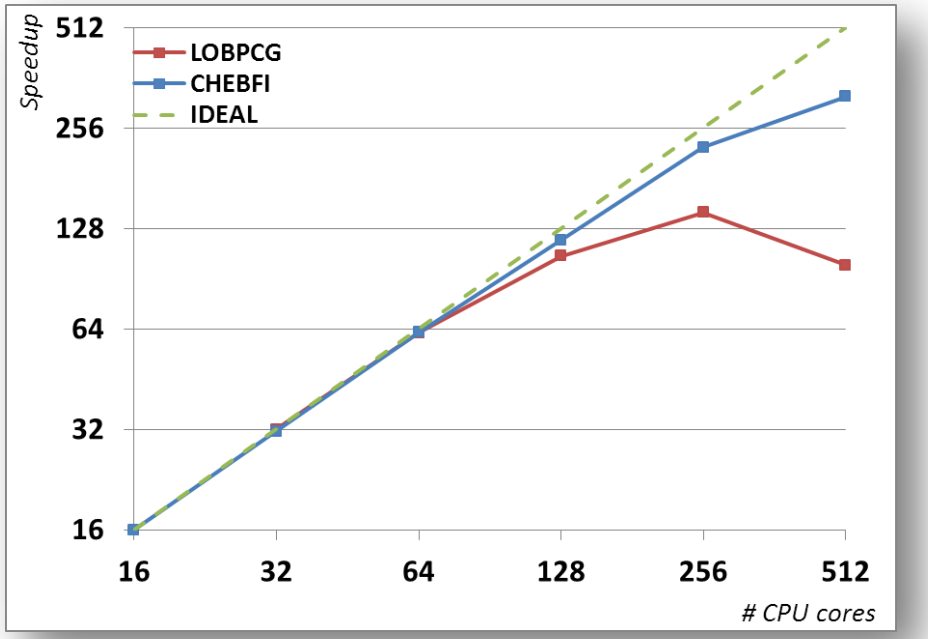
COMPARING ALGORITHMS – « SMALL SYSTEM »

CHEBFI vs BLOCK CG

UO_2 crystal

98 atoms, **512 bands**



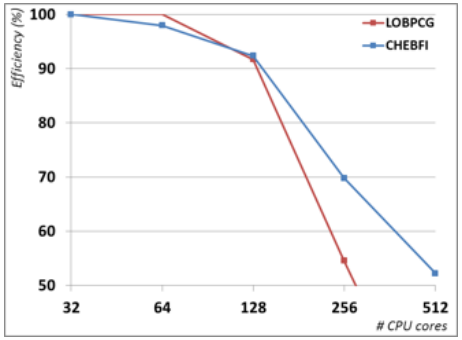


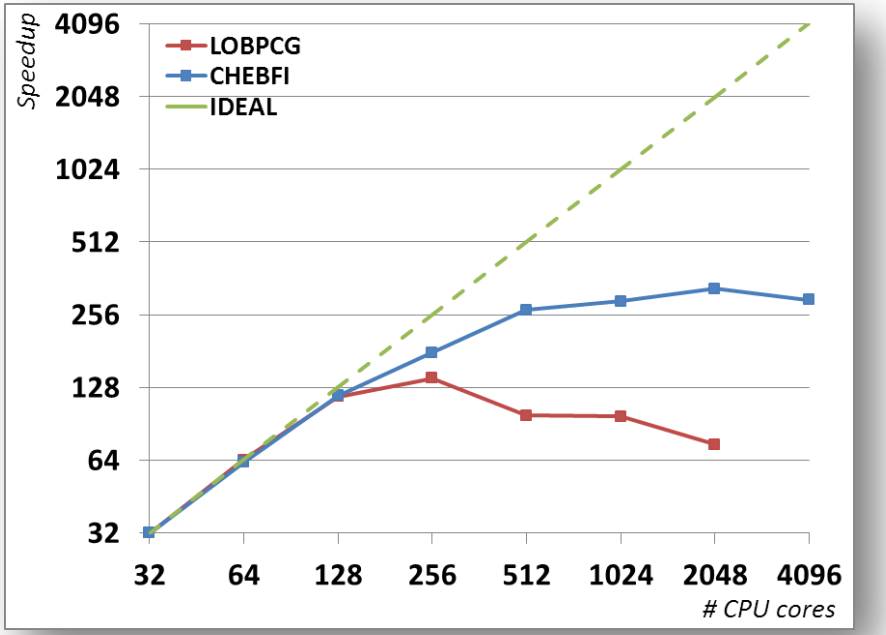
COMPARING ALGORITHMS – « MEDIUM-SIZED SYSTEM »

CHEBFI vs BLOCK CG

Au crystal + vacancy

107 atoms, **1024 bands**



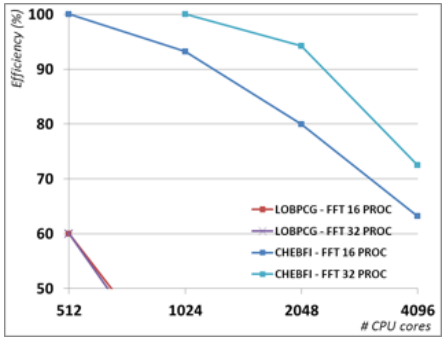


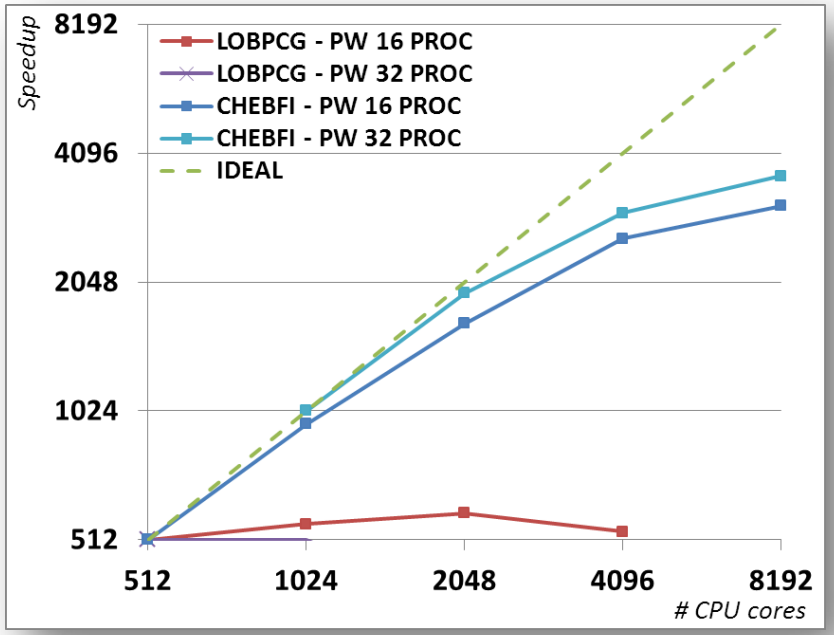
COMPARING ALGORITHMS – « MEDIUM-SIZED SYSTEM »

CHEBFI vs BLOCK CG

Ti crystal

256 atoms, **2048 bands**

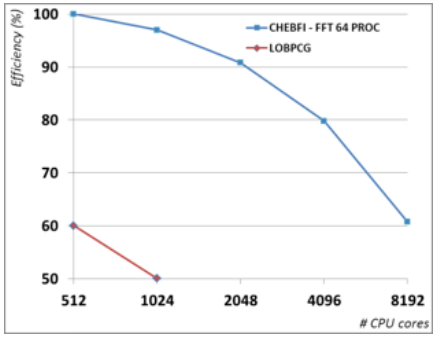




COMPARING ALGORITHMS – « LARGE SYSTEM »

Ti crystal

512 atoms, **4096 bands**



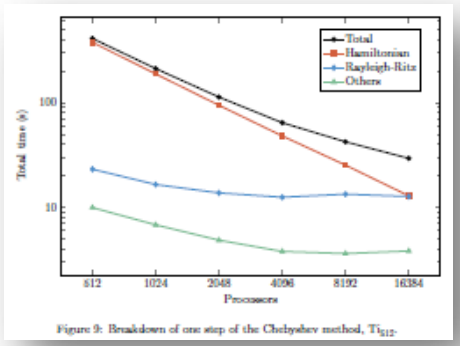
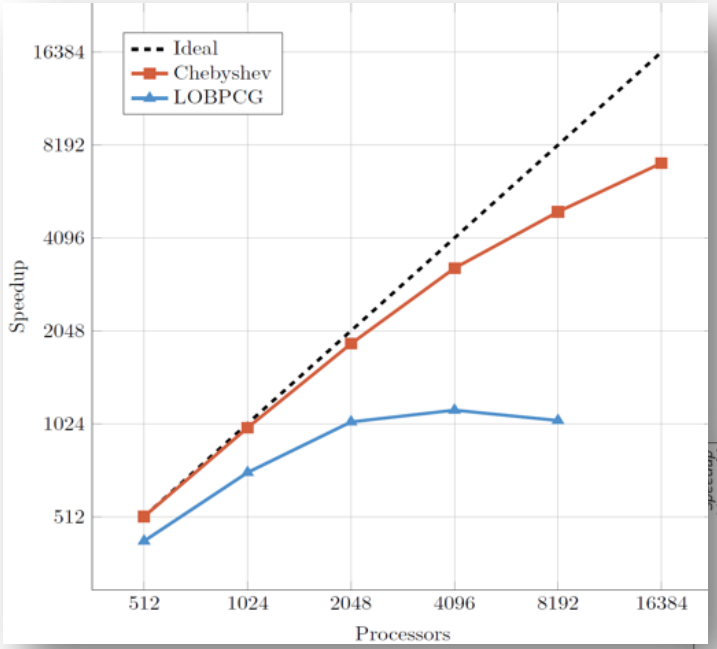
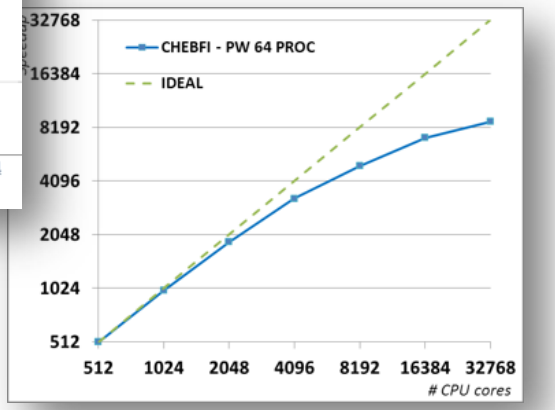


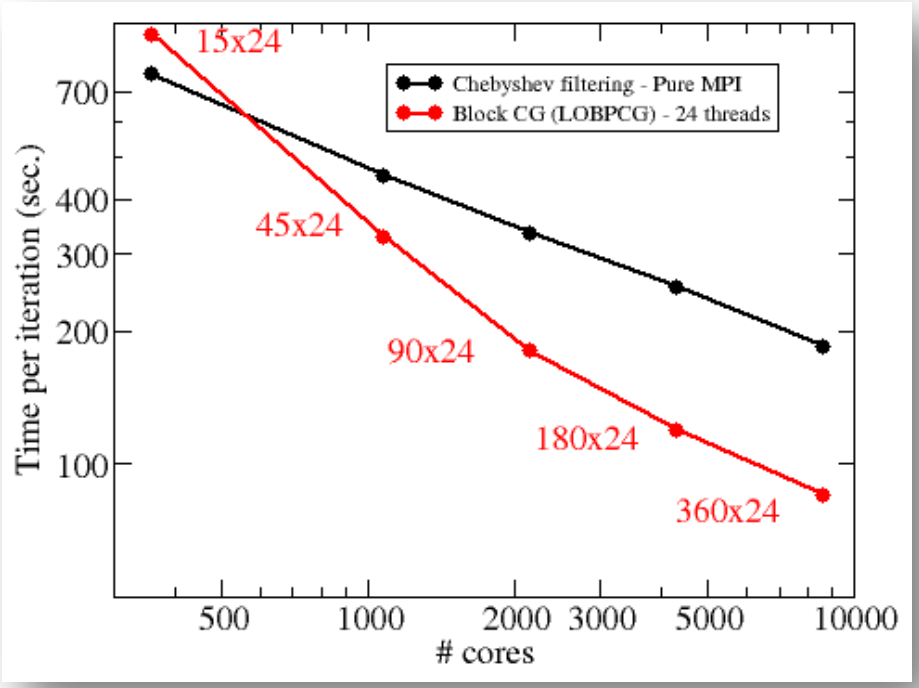
Figure 9: Breakdown of one step of the Chebyshev method, Ti_{512}.

CHEBFI vs BLOCK CG





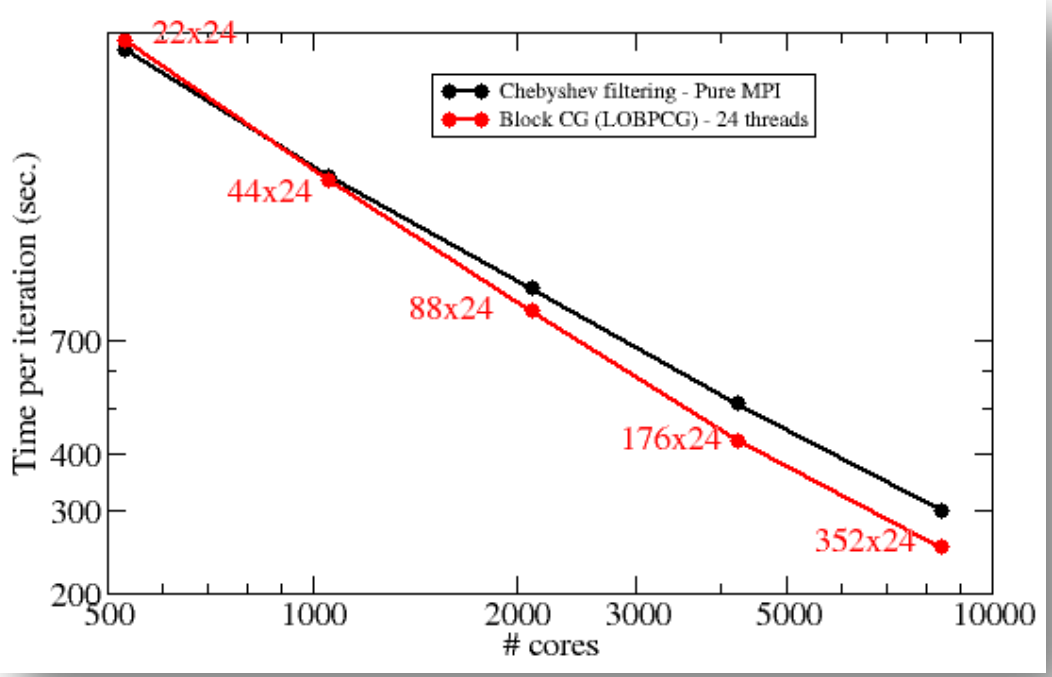
Block CG is back!



Gallium oxide Ga_2O_3
1960 atoms
8700 bands (**17400 electrons**)

TGCC –Joliot-Curie
Intel Skylake

Block CG is back!



Gallium oxide Ga_2O_3

4160 atoms

18400 bands (**36800 electrons**)

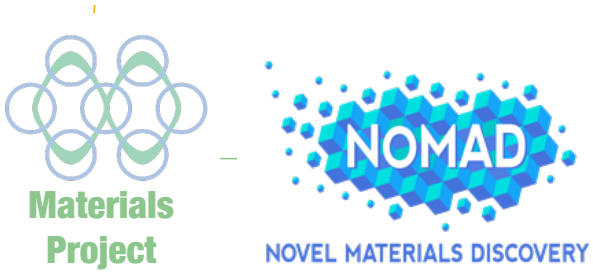
TGCC –Joliot-Curie

Intel Skylake

Summary

- **High Performance Computing is a great opportunity to…**
  - Compute properties of larger systems
  - Add more physics in DFT
- **DFT parallel efficiency is strongly system dependent**
- **It is highly recommended to use hybrid parallelism**
- **A DFT code cannot be used on a supercomputer without a minimum knowledge of…**
  - The computer architecture (nodes, CPUs/node, …)
  - The iterative diagonalization algorithm

AND MORE…



- **High Throughput computing**
  - Materials genomics
- **Coupling of DFT with Machine Learning**
  - Data-driven potentials
  - Stochastic/canonical sampling
- **Coupling DFT with other scales**