ABINIT School on ground state, linear response properties and dynamics
Prague, Sept. 2-6, 2019

FROM RESEARCH TO INDUSTRY

Parallelization in ABINIT

Speed-up Efficiency Manycore
MPI OPENMP

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

OUTLINE

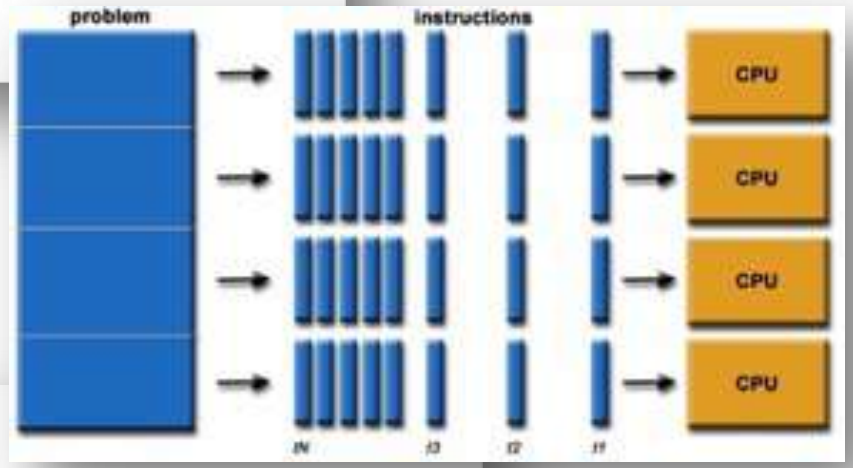
- What are super-computers made of?
- How to measure the parallel efficiency
- ABINIT parallelization strategy
- Parallelism inside ABINIT
- Performance (examples)

What are supercomputers made of?
Ho to use them?

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

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 solved in multiple parts which can be solved **concurrently**
  - Each part is associated to a series of **instructions**; instructions are compute processes or memory transfers
  - Instructions of each part are executed simultaneously on different Compute Processing Units

IMPROVE 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 floating point operations simultaneously (overlap)
    → 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

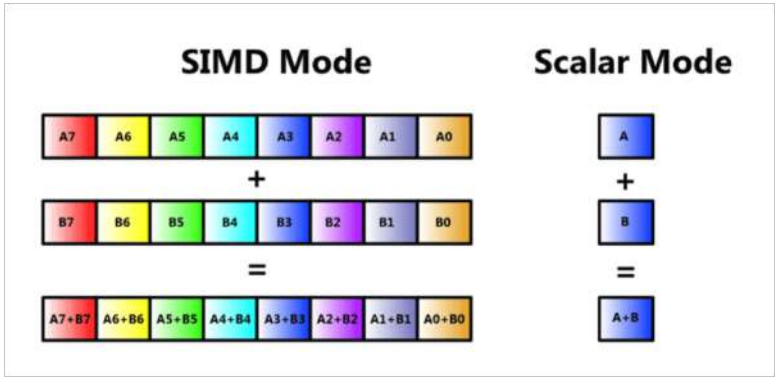
- *Power increase as Frequency^3*
  → Clock rate is limited
- *Power is a limited factor for supercomputers*
  → Around 3-5W per CPU nowadays
- *Multiple slower devices are preferable than one superfast!*
  → Multiples computing units per CPU!
- *More performance with less power?*
  → software problem!

VECTORIZATION

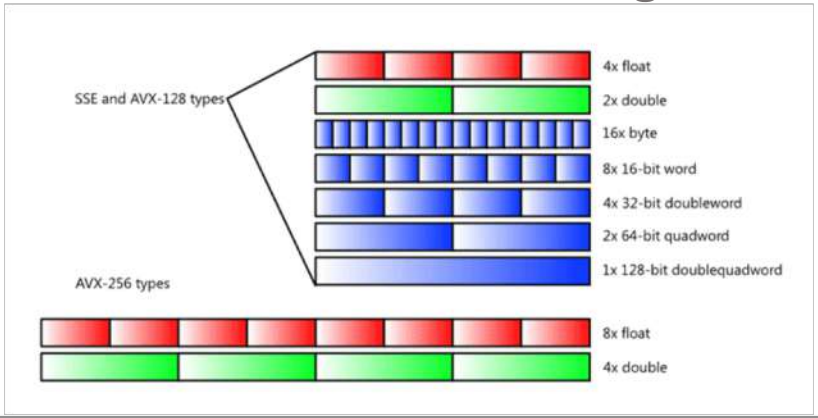
- **What is vector computing?**
  - Vectorization can be considered as a “hardware parallelization”, directly implemented in the processor unit.
  - It generalizes operations on scalars to apply to vectors.
  - Operations apply at once to an entire set of values.
  - The processor uses a specific set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

- **Common vector operations**
  - Addition, Multiplication
  - FMA (Fuse Multiply-Add) $a \longleftarrow a + (b \times c)$

- **Example for addition**



- **Size of "vector" in recent hardware is increasing**



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

IMPROVE PERFORMANCE OF MEMORY TRANSFER

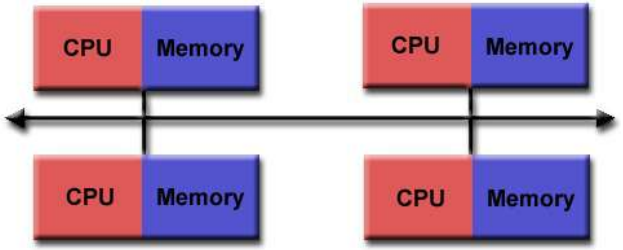
- **Traditional Measure of memory performance:**
  - Access time (ps)
  - Transfer speed, bandwidth (Byte/s)
  - Latency (ns)
- **How to speed up memory access?**
  - Speed up the access
    → Change the technology
    → Implant the memory closer to processing unit!
  - Increase bandwidth
    → Increase memory clock rate, increase number of “channels”!
  - Decrease the latency
    → Improve “switches”, improve network speed

THE POWER COST OF MEMORY TRANSFER

- **Some facts about memory transfer:**
  - 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 FLOPS/Byte transferred
- **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
- **Recomputing data is faster than fetching it randomly in memory**

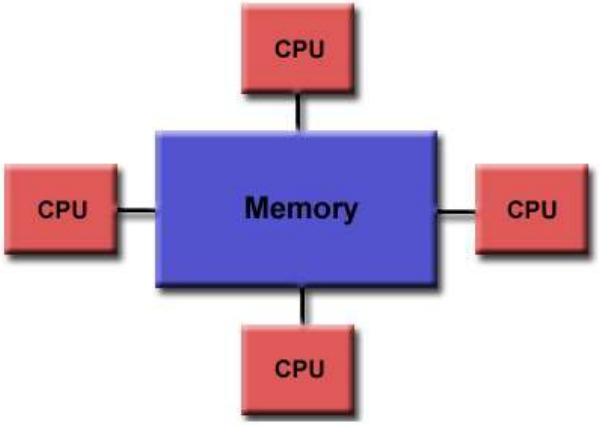
HOW TO DISTRIBUTE DATA IN MEMORY

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?

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

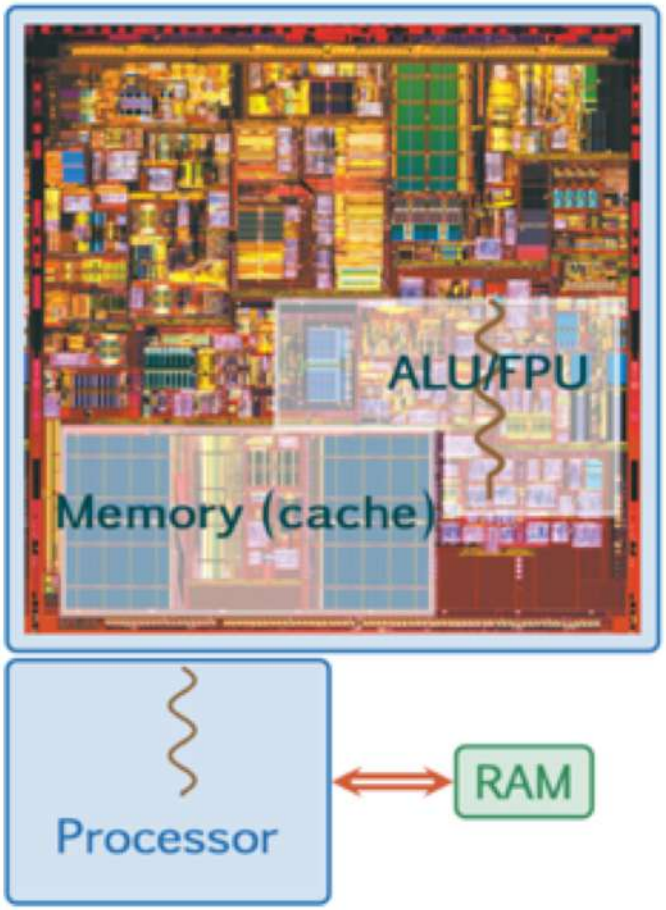
2 workers are twice faster than one!

What is a worker on a (super)computer?

A BASIC PROCESSOR DESIGN (OLD FASHIONED)

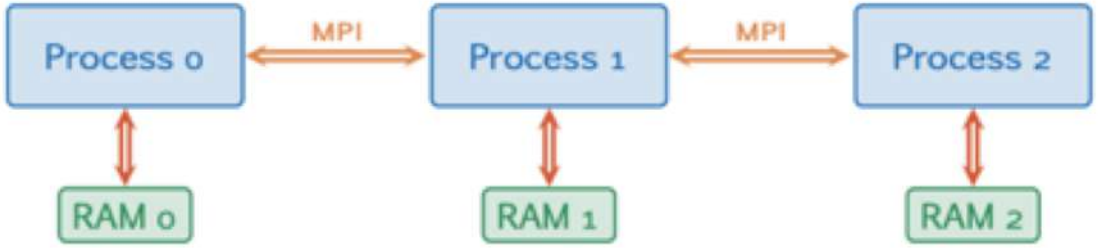
- Arithmetic and Logic Unit
- Floating-Point Unit
- Memory (small)
- Controllers
- …
- RAM is far from the processor

- 1 processor (CPU) has 1 core!



BUILDING A SUPERCOMPUTER WITH OLD CPUS

MPI = Message Passing Interface
A communication protocol



Message Passing paradigm

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

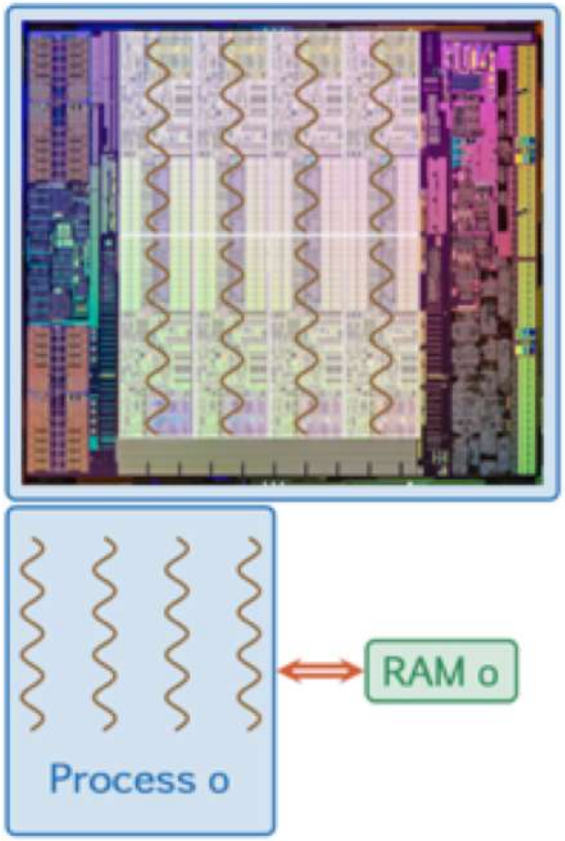
How to use it?

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

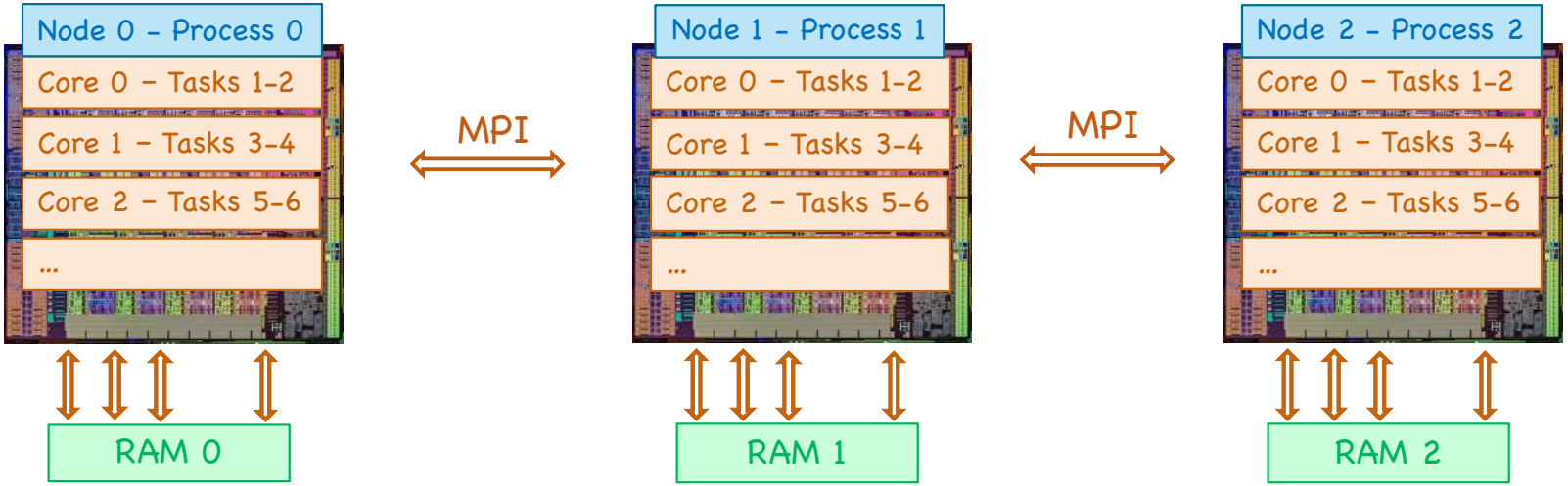
A MODERN PROCESSOR DESIGN: « MANYCORE »

- 1 processor has several cores (nowadays: 4 to 68)
- 1 core = ALU/FPU/cache memory
- Each core may have 2 (or more) 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!

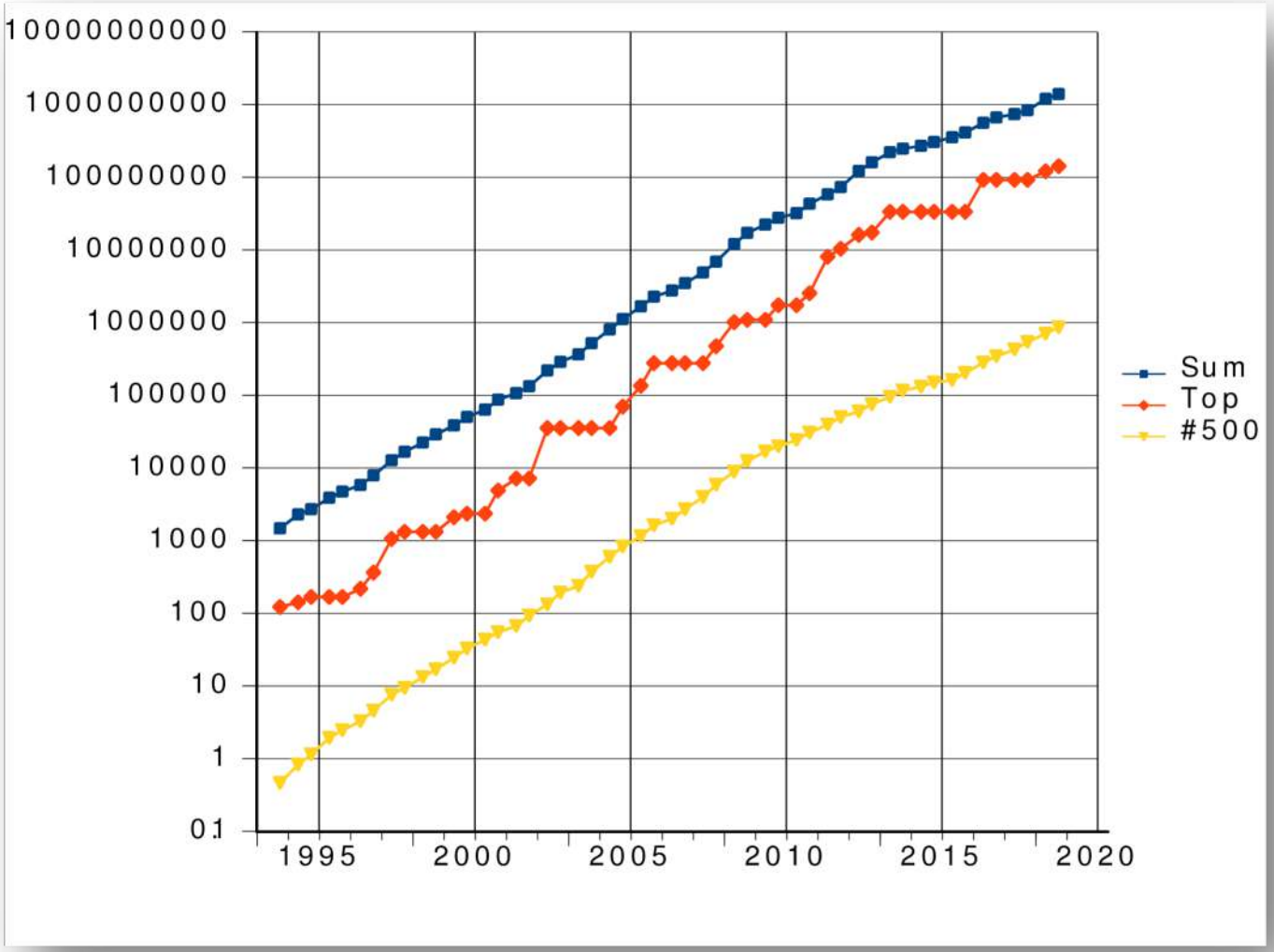
How to use it?

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

SUPER-COMPUTERS EFFICIENCY INCREASES CONTINUOUSLY …

Super-computers world TOP 500

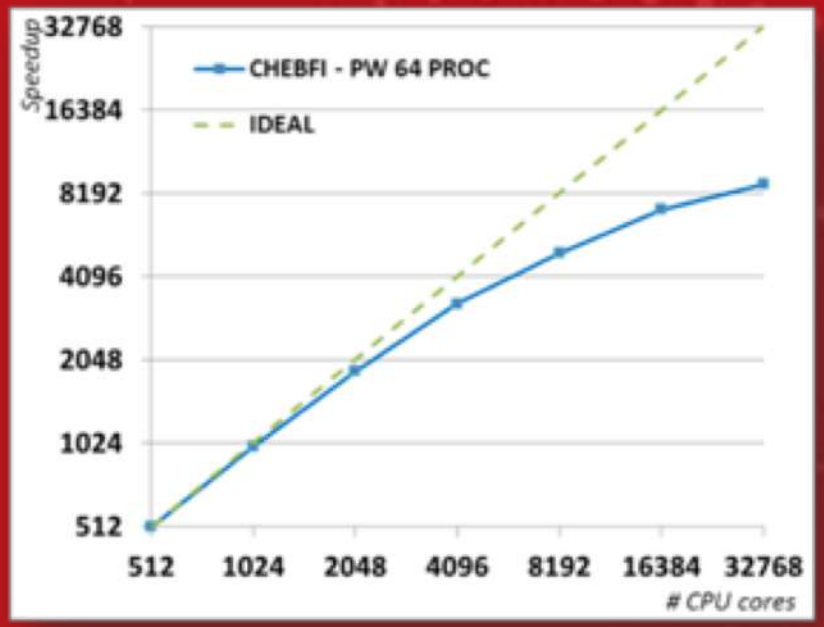
Perf. In GFLOPS



...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 the efficiency of a code on a super-computer?

3 WAYS TO DEAL WITH PARALLELISM

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

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

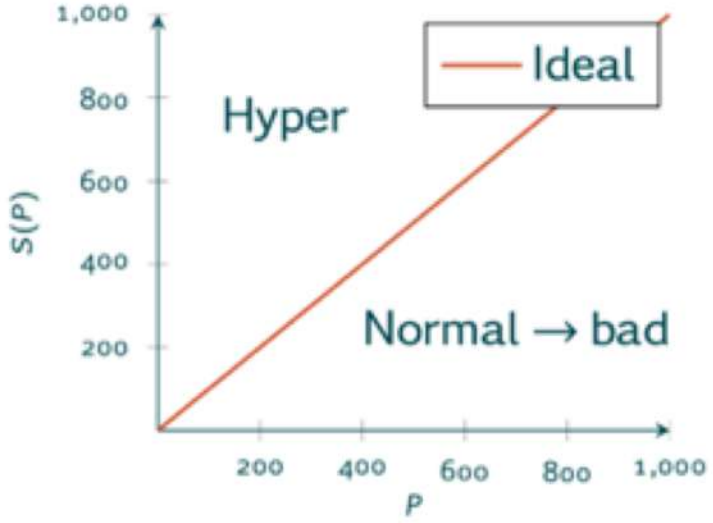
SPEEDUP

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, ...

SPEEDUP – AMDAHL'S LAW

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 :

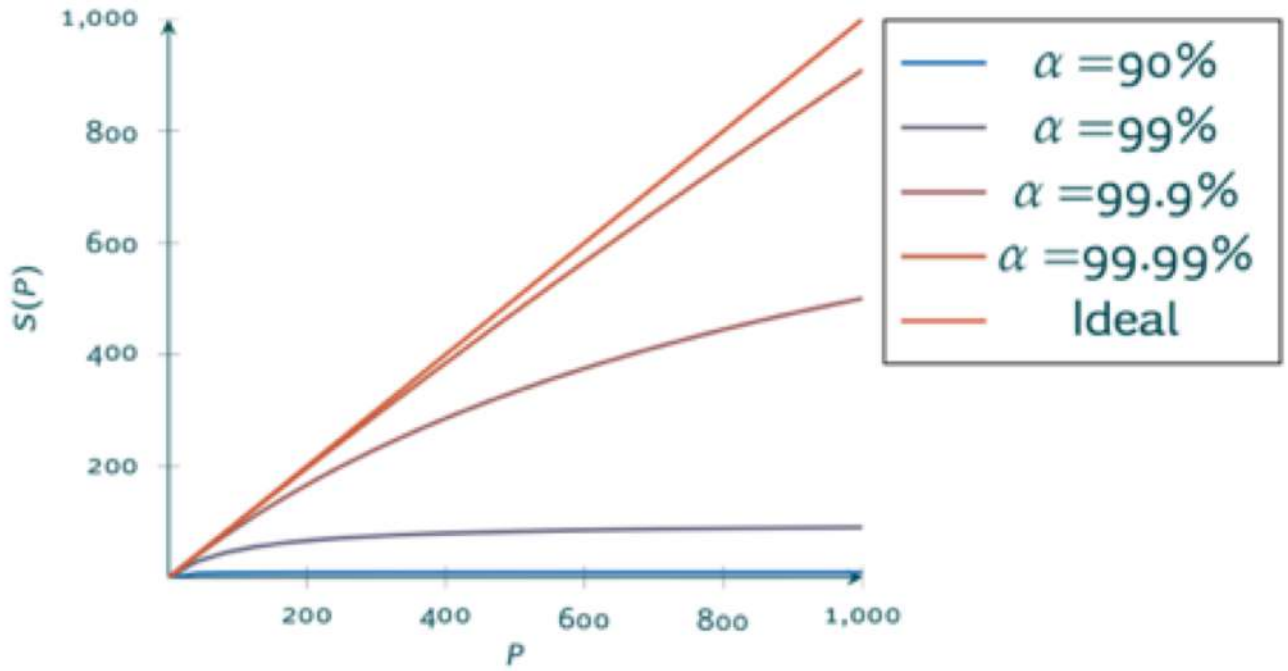
$$T(P) = \underbrace{(1-\alpha)T}_{\text{sequential}} + \underbrace{\frac{\alpha}{P}T}_{\text{parallel}}$$

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

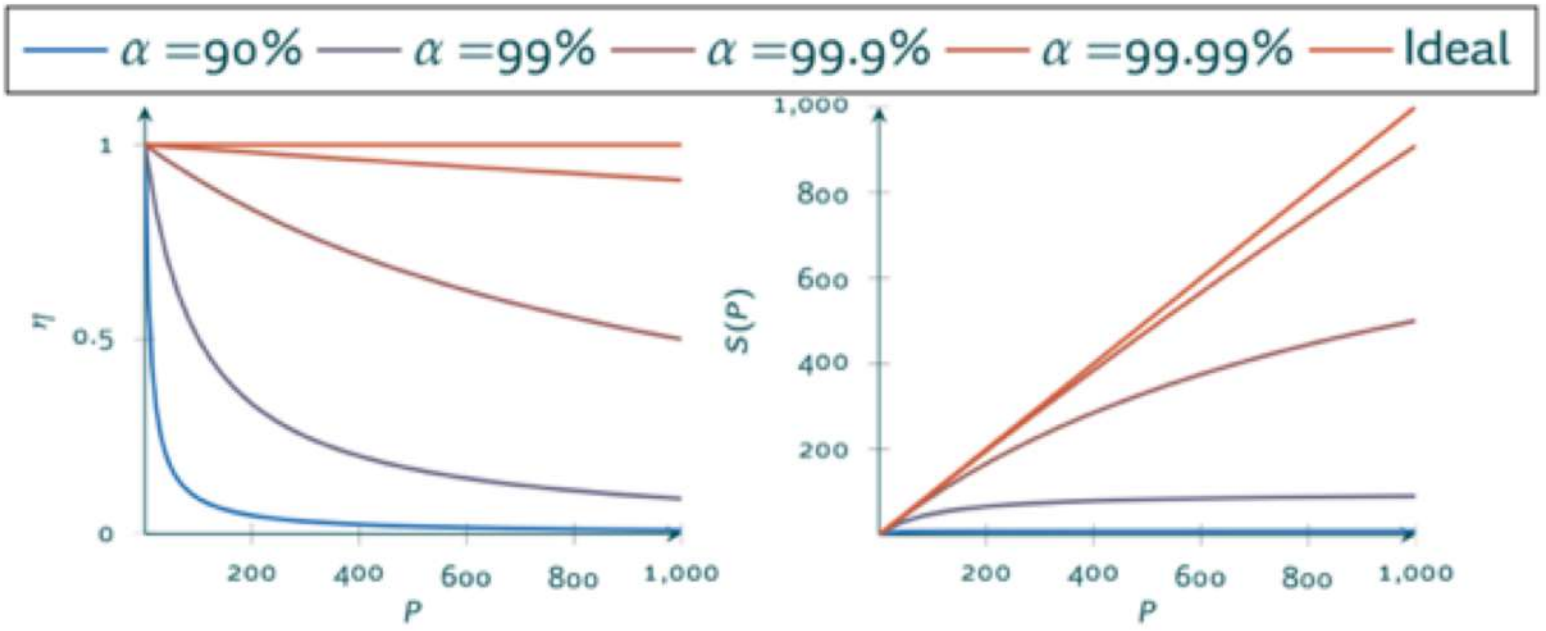
$$S(P) = \frac{1}{1 - \alpha + \frac{\alpha}{P}} \qquad (2)$$



SCALING EFFICIENCY

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

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 :

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

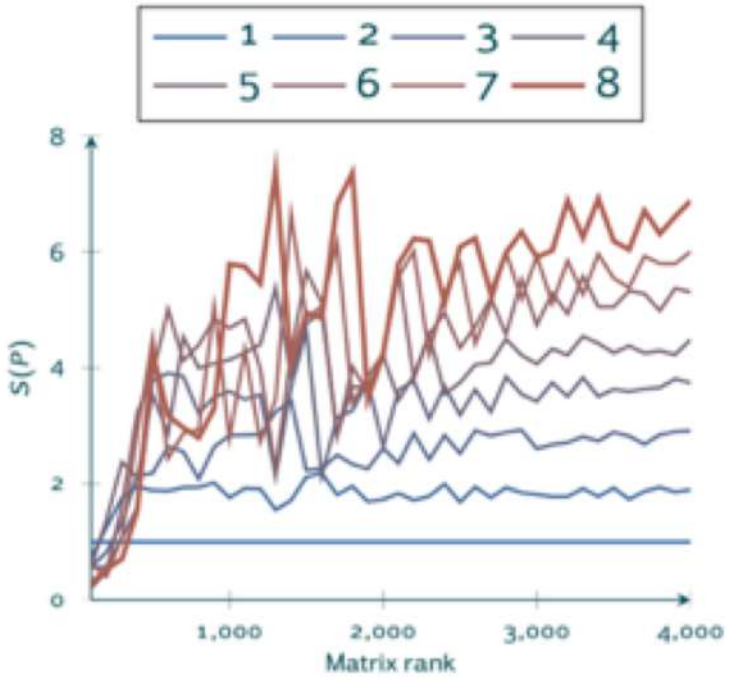
- Weak scaling : The work is increased in the same way as the number of workers :

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

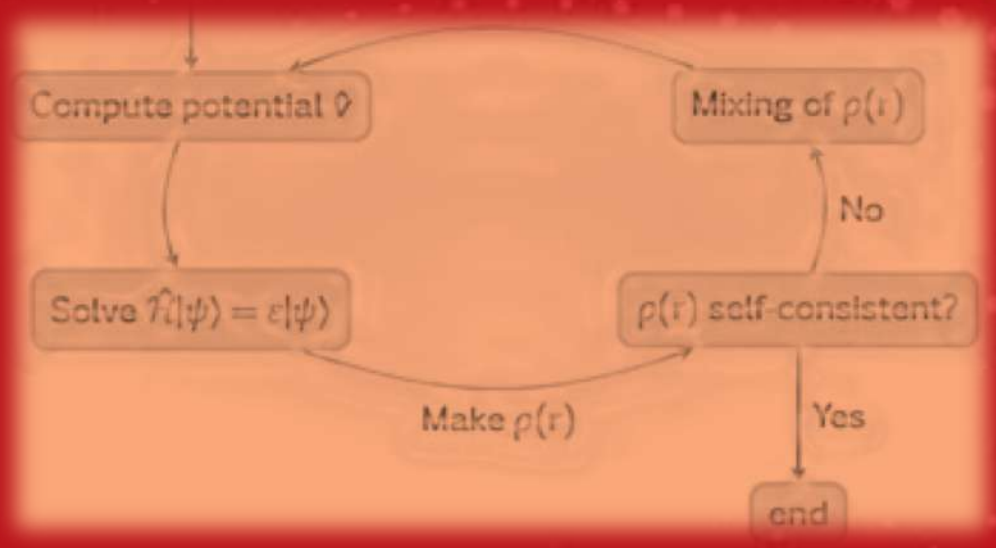
TEST ON MATRIX-MATRIX MULTIPLICATION

Test of the so called GEMM

GEneral Matrix Matrix product

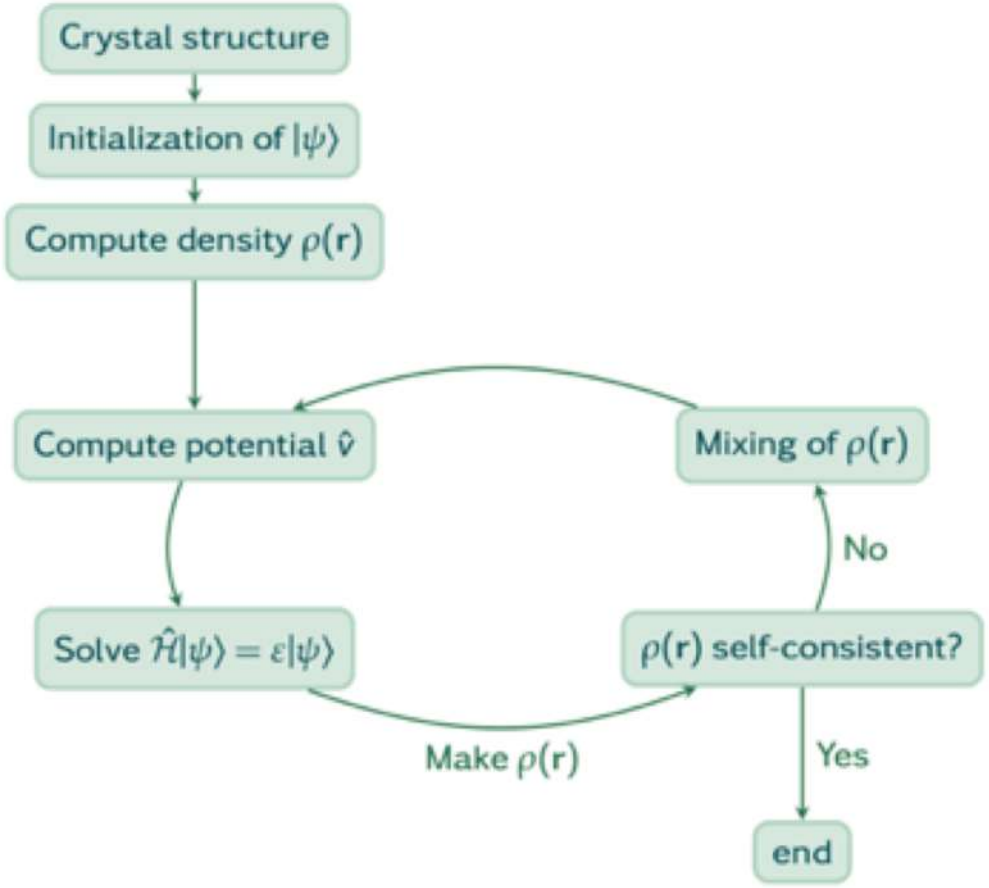


Good scaling is reached only if each "worker" has enough data to work on !

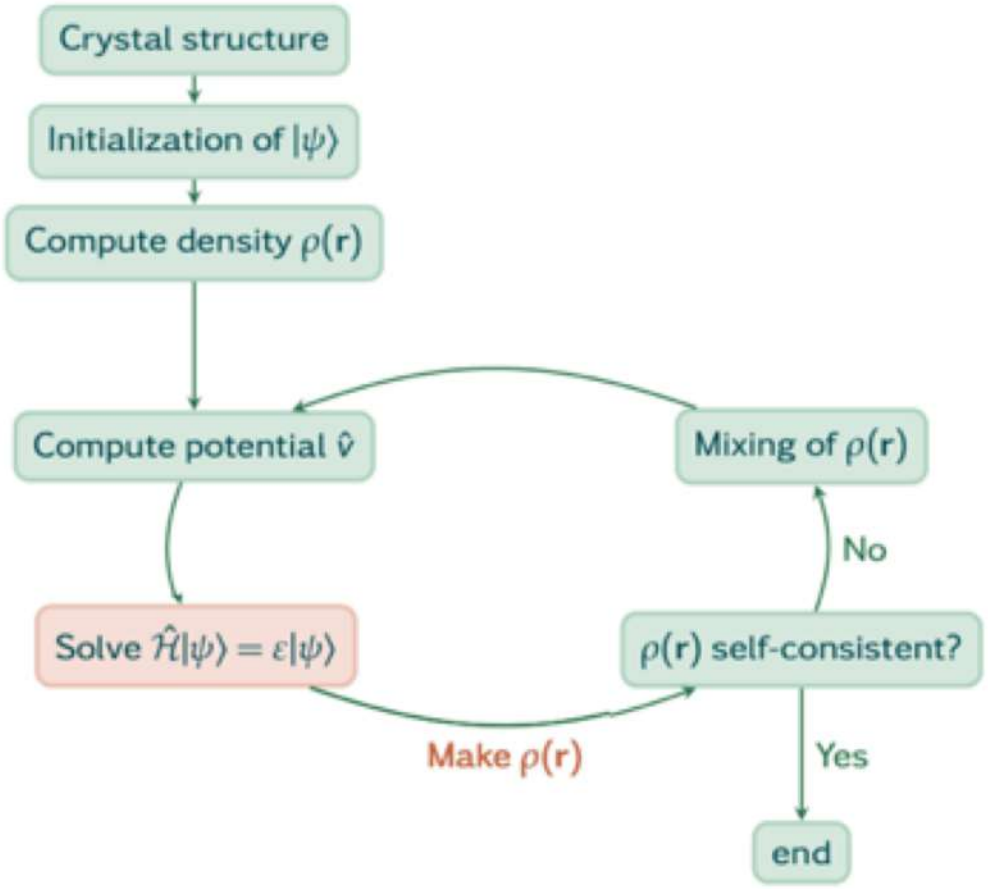


ABINIT parallelization strategy

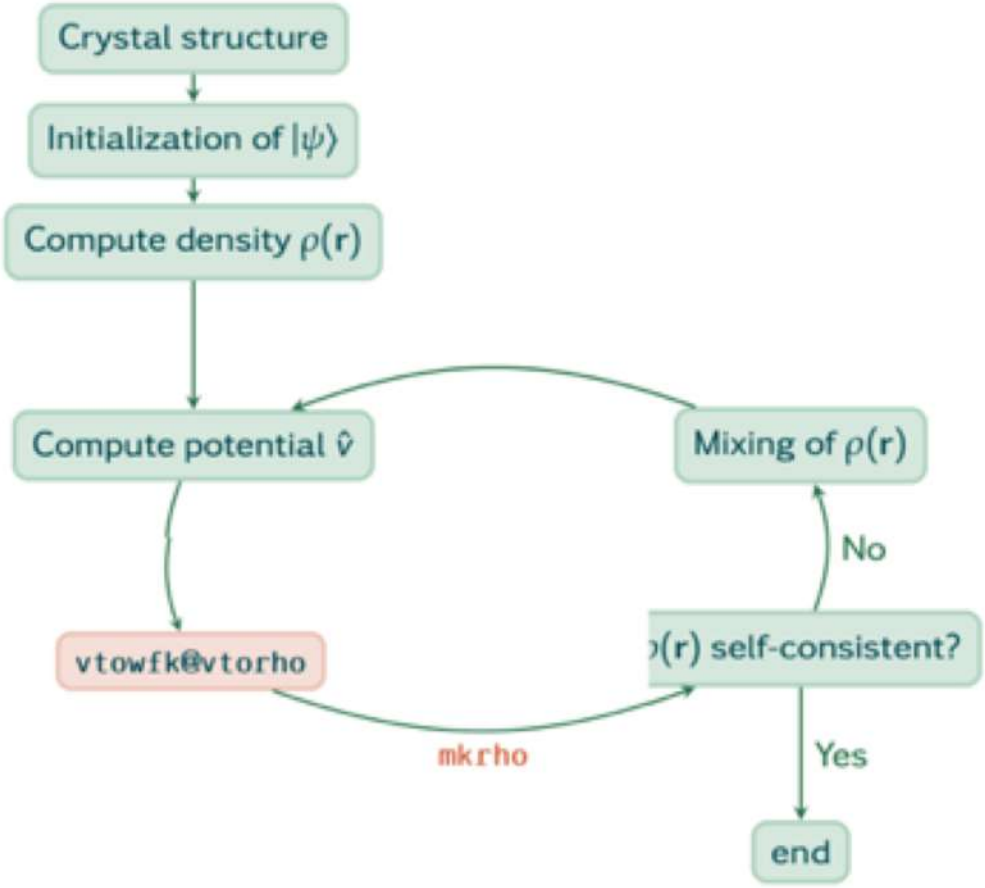
The main self-consistent loop



The main self-consistent loop



The main self-consistent loop



MOST TIME CONSUMING PARTS

- Use `timopt=-3` in input file to obtain detailed summary of time spent in ABINIT
- A typical time analysis:

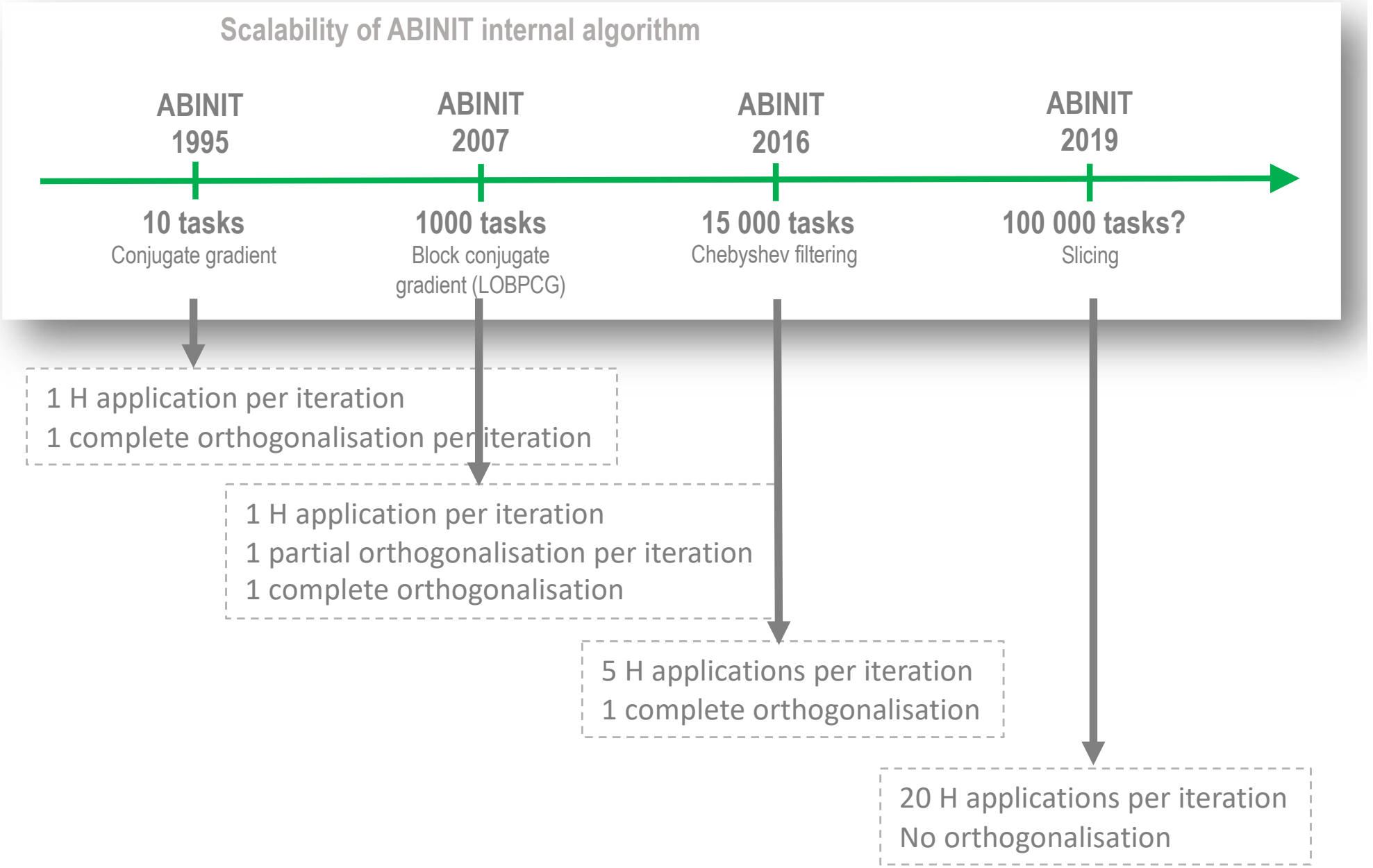
- vtorho: $v \rightarrow \rho(\mathbf{r})$ (98%)
  - vtowfk: $v \rightarrow |\psi_{\mathbf{k}}\rangle : \hat{\mathcal{H}}_{\mathbf{k}}|\psi_{\mathbf{k}}\rangle = \varepsilon|\psi_{\mathbf{k}}\rangle$ (97%)
    - cgwf, lobpcg: Diagonalization (7%)
    - getghc: $\hat{\mathcal{H}}|\psi\rangle$ (90%)
  - mkrho: $n(\mathbf{r}) = \sum_i^N \langle\psi_i(\mathbf{r})|\psi_i(\mathbf{r})\rangle$
- rhotov: $n(\mathbf{r}) \rightarrow v$

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

INTERNAL ALGORITHM SCALABILIYTY

- Target computations :
  `1000<Nband<50000`, Npw~100 000…250 000
- Direct diagonalization unachievable (~10^{12})
- In search of the eigenvectors associated with the lowest eigenvalues
- Need an **iterative algorithm**
- Different kind of algorithms but the elementary bricks are the same:
  - Hamiltonian application (linear algebra, FFT) ➢ computation
  - Rayleigh-Ritz procedure (linear algebra, diago, ortho) ➢ communication

INTERNAL ALGORITHME SCALABILITY



ABINIT STRONG SCALING - 2016

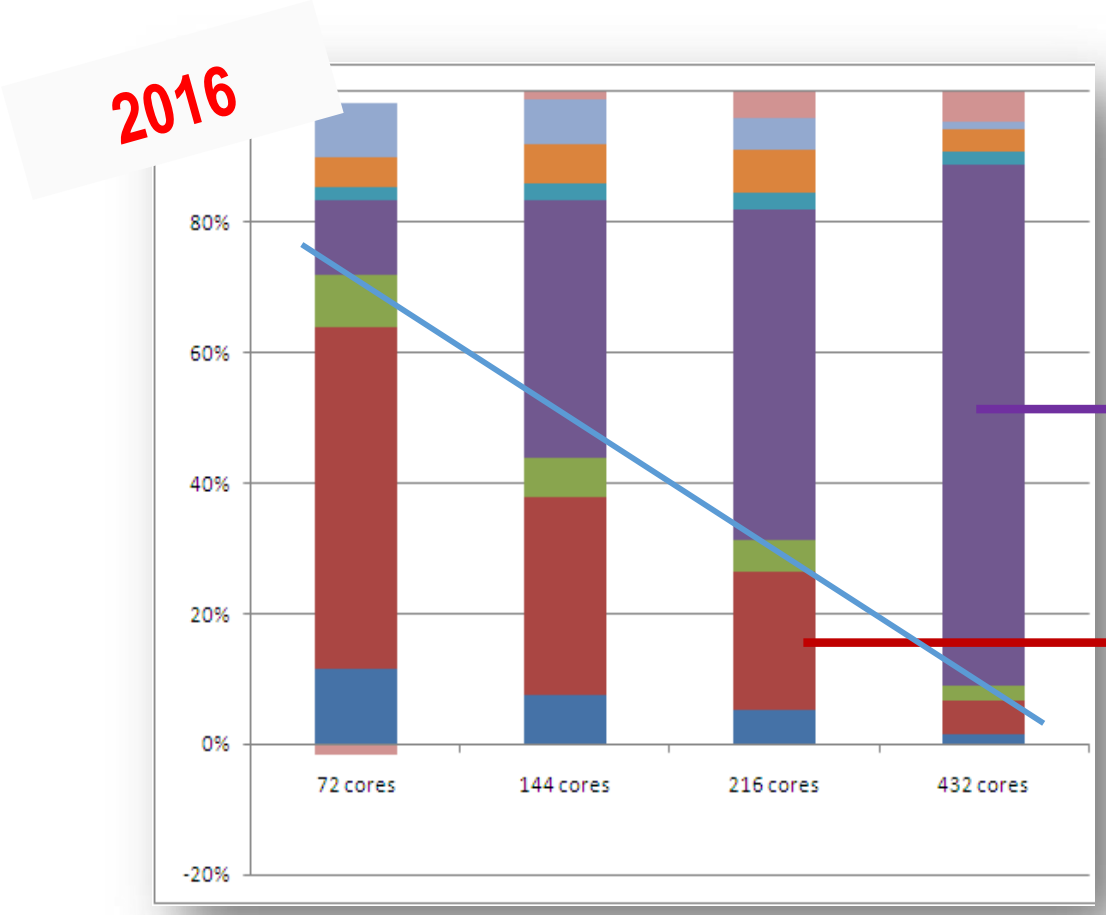
Repartition of time in in a ground-state calculation varying the number of band CPU cores (MPI, strong scaling)

TEST CASE

A vacancy in a 108 atoms cell (gold)

Gamma k-point only, PAW

Computation of total energy and forces



Iterative solver (eigenvalues)

Without H application

Hamiltonian application

Linear algebra, FFT

ABINIT ON MANY CORE ARCHITECTURES - STRATEGY

1. Improve the **scalability** of the diagonalization algorithm
   More calculation, less storage, less communications

2. Efficiently use the **shared memory** (*openMP*)
   in a « medium grained » mode
   Adapt the code to the hardware **topology**
   Give longer tasks to the elementary computing units
   Decrease the data movements

3. ***Externalize*** the elementary operations
   Express the physics in terms of elementary operations
   Use vendor (or optimized) libraries

4. Add an ***abstraction*** layer
   Isolate low level optimized operations

ABINIT PERFORMANCES - 2019

Uranium
128 atoms
1600 bands (**3200 electrons**)

2016

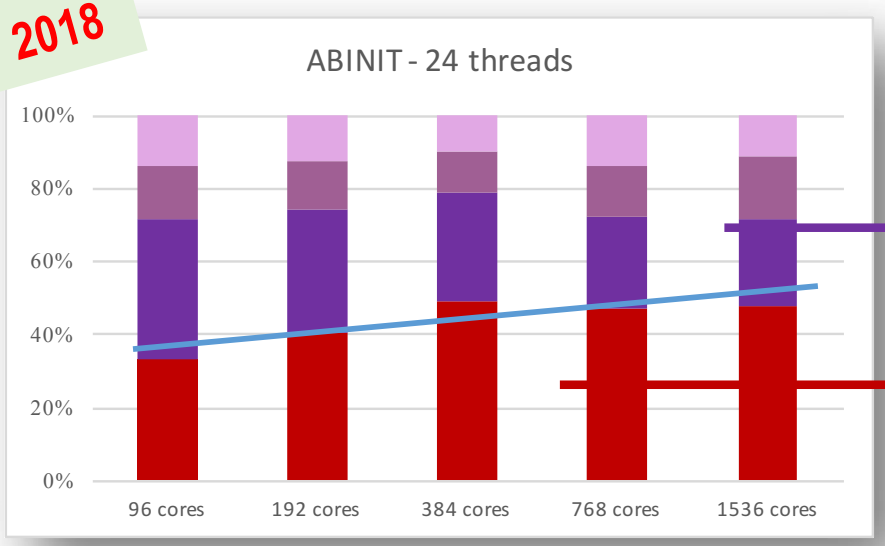
Iterative solver

Hamiltonian

Old implementation
MPI only
CEA-TGCC Curie
Nehalem
16 cores/node

A perfect example of Amdahl's law

2018



Iterative solver

Hamiltonian

CEA Tera1000-2
Intel KNL
64 cores/node

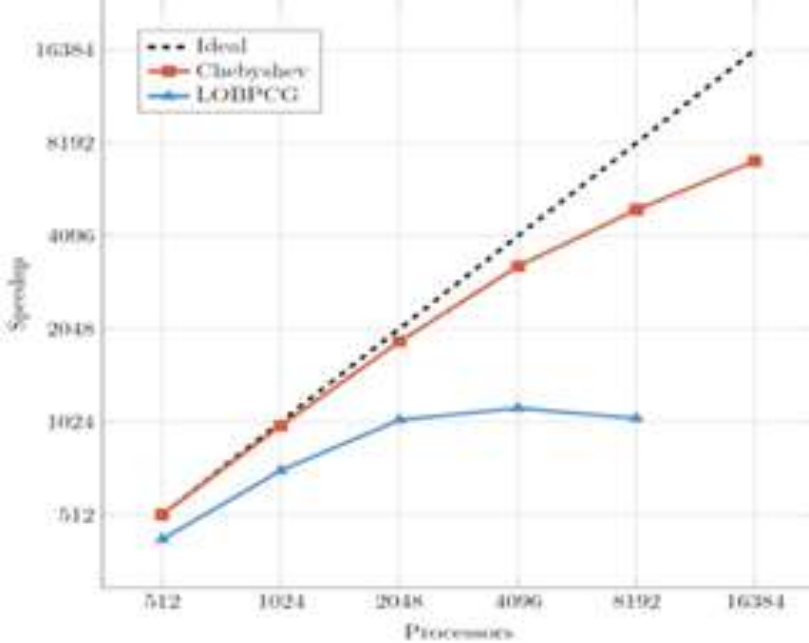
New implementation
MPI x 24 threads

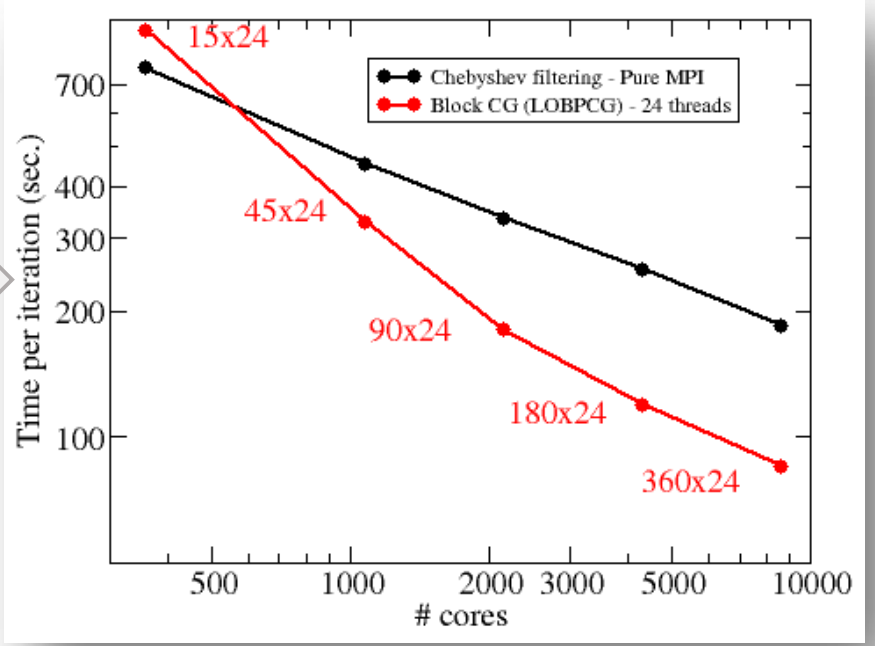
FROM DISTRIBUTED TO HYBRID PARALLELISM

Distributed (pure MPI) → ***Hybrid (MPI+openMP)***

2016 — *TGCC – Curie*
Intel Nehalem

2019 — *TGCC – Joliot-Curie*
Intel Skylake – 48 cores/node – 2 sockets





Titanium
512 atoms (**8200 electrons**)
Speedup

Gallium oxide Ga_2O_3
1960 atoms (**17400 electrons**)
Time per SCF iteration

```
o iblockbd=1,nblockbd
  iband=1+(iblockbd-1)*blocksz
  if (proc_distrb_cycle) then
    if (.not.cg_band_distributed) then
      icgb=icgb+npw_k*my_nspinor*blocksz
    endif
    if (.not.cprj_band_distributed) then
      ibgb=ibgb+my_nspinor*blocksz
      cycle
    end if
```

Parallelism inside ABINIT

ELECTRONIC DENSITY AND PARALLELISM

Electronic density formula within PAW+plane-wave DFT

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

	Spins	Bands	k vectors	Plane waves	Atoms
Parallelization levels:					
System-size scaling:	≤ 2	$\propto S$	$\propto 1/S$	$\propto S$	$\propto S$
Parallel speedup:	Linear	???	Linear	Poor	Linear

- Each k independent from the others ⟶ **Try first parallelization over k points**
- In case of polarization (nsppol=2) ⟶ **parallelization over spins**

PARALLELIZATION OF EIGENSOLVER

- Eigenvectors are orthogonal to each other ⟶ non trivial parallelization
- Can parallelize over bands and/or plane waves
- 2 kinds of algorithms :
  - Treat blocks of eigenvectors concurrently
  - Filter independently sets of eigenvector

- Possible choices of algorithm:
  - **Conjugate Gradient**
    Default when k-points parallelization only
  - **Block conjugate gradient** (LOBPCG)
    Default when Band-FFT parallelization
  - **Chebyshev Filtering**
    For a very large number of processors

ABINIT INPUT VARIABLES FOR PARALLELISM

- By default, only parallelization over k/spins is activated
  - **paral_kgb**: **1** → To activate k/plane-waves/bands parallelism

- Choice of iterative eigensolver
  - **wfoptalg**: **0** → conjugate gradient
    114 → block conjugate gradient
    1 → Chebyshev filtering

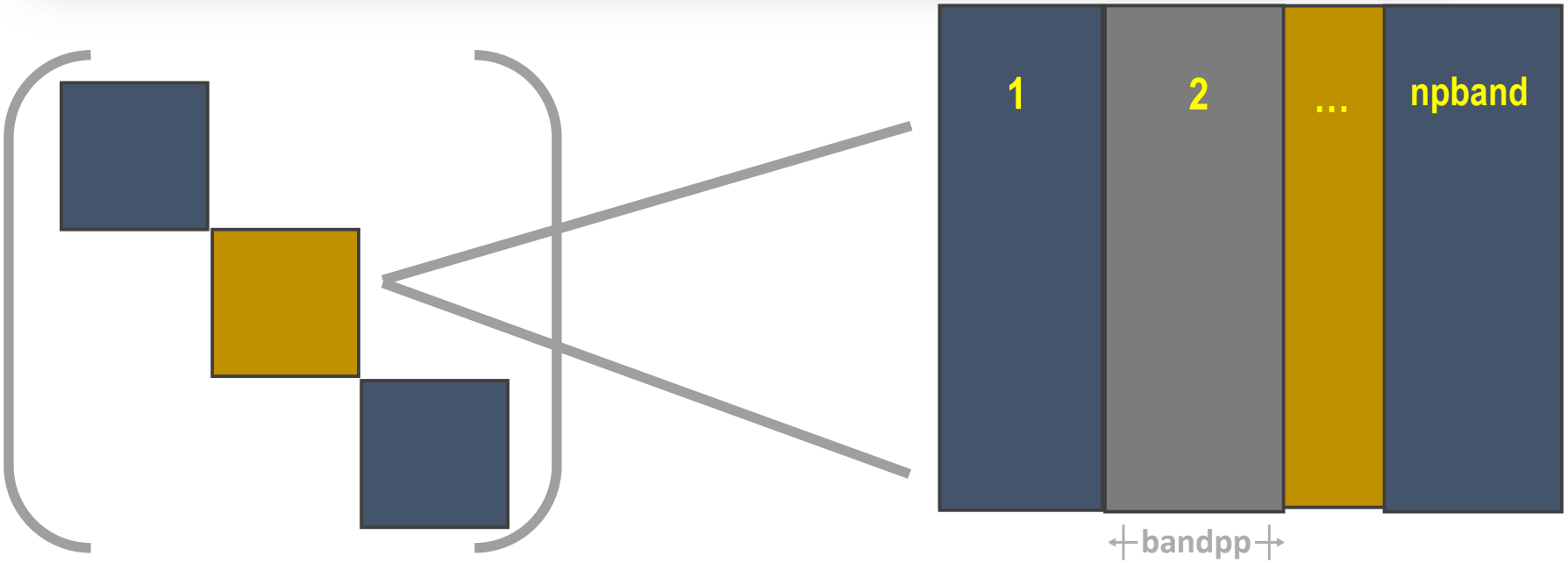
- 4 basic input variables to control how things are distributed

npkpt	npband	bandpp	npfft or tasks (openMP)
# procs for k-points	# procs for bands	bands	# procs or # tasks for plane-wave/FFT

```
nproc = npkpt x npband x npfft
```

PARALLELISM OF BLOCKED ALGORITHM (LOBPCG)

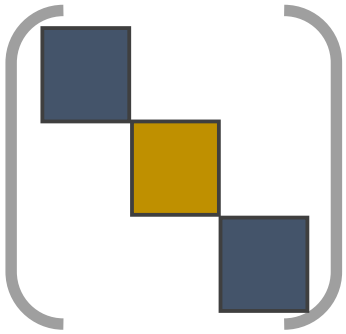
- **Diagonalization per block:**
  - Each block of eigenvectors is concurrently diagonalized by the `npband` processes. One bloc after the other.
  - Each process handles `bandpp` bands
    → The size of a bloc is `bandpp` *x* `npband`



PARALLELISM OF BLOCKED ALGORITHM (LOBPCG)

- The **accuracy** of the diagonalization depends on the block size:
  - Eigenvectors are orthogonal inside a block;
    Then blocks are orthogonalized
    The orthogonalization is better for large blocks
  - → ABINIT converges better with large blocks

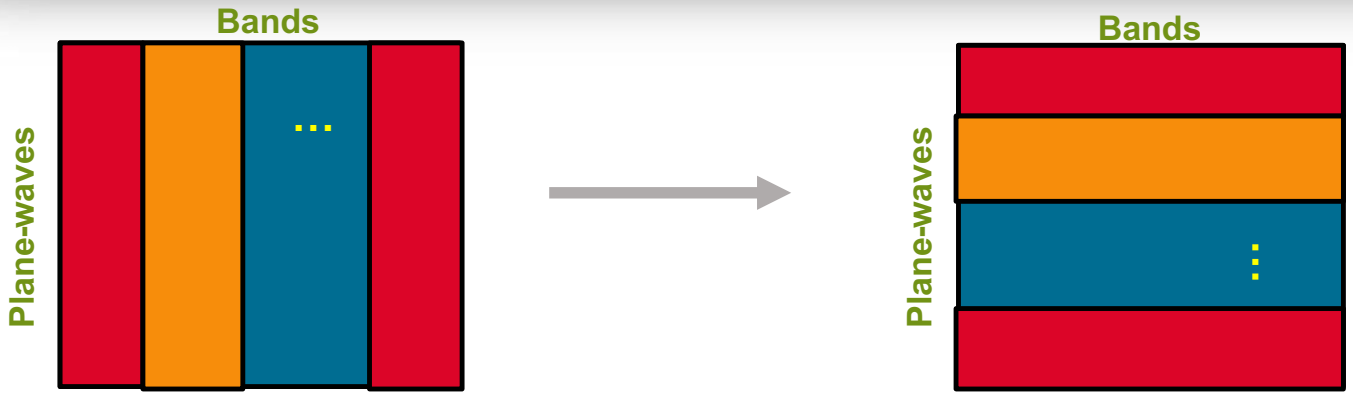
- The **speed** of the diagonalization depends on the block size:
  - Blocks are diagonalized one after the other
    Parallelization is more efficient for one block
  - → ABINIT converges faster with large blocks



- But large blocks need memory!

TWO INTERNAL REPRESENTATIONS

- Bands are distributed in memory.
  Plane waves (FFT components) are distributed in memory
- To perform a **scalar product** $\langle \Psi_i | \Psi_j \rangle$
  a process needs the Ψ_i and Ψ_j vectors and a few plane-waves
- To perform a **Fast Fourier Transform** (FFT)
  a process need all the components of a vector Ψ_i
- We **change the representation** by applying a transposition
  $\rightarrow$ Needs a lot of communications



FIRST SCENARIO: PURE MPI

- First, if possible, parallelize over k-points/spins (**npkpt**)
- Then parallelize the diagonalization over `npband x npfft` processes
  - Each block diagonalization is made by **npband** processes in parallel
  - Each process handles **bandpp** bands sequentially
  - Scalar products and FFTs are done by **npfft** processes
- Distribute the workload as follows:
  - **npfft** should be >1, but should be small enough (<10)
  - **npband** sets the # of procs: **nproc = npkpt x npband x npfft**
  - Increase the convergence speed by increasing the block size
    → increase **bandpp** but not to much (sequential part)
  - **nband** has to be a multiple of **npband x bandpp**
- In any case, **the ideal distribution is system dependent**!

2ND SCENARIO: HYBRID MPI+OPENMP - 1

- Launch the calculation using a maximum number of openMP tasks, Ideally the size of a “socket” on the compute node
  `export OMP_NUM_THREADS=xx`
- First, if possible, distribute MPI processes over k-points/spins (**`npkpt`**)
- Then distribute the reminding MPI processes over bands (**`npband`**)
  - Each block diagonalization is made by **`npband`** processes in parallel
  - Each process handles **`bandpp`** bands in parallel using the *openMP* tasks
  - Scalar products and FFTs are done by the *openMP* tasks (**`npfft`** not used)
  - **`nproc = npkpt x npband`**

2ND SCENARIO: HYBRID MPI+OPENMP - 2

- Distribute the workload as follows:
  - **npband** x **bandpp** (size of a block) should be maximalized and has to divide the number of bands (**nband**)
    Ideally it should be **nband** or **nband/2** or **nband/3**
  - **bandpp** has to be a multiple of the number of *openMP* tasks
  - **nband** has to be a multiple of **npband** x **bandpp**
- In any case, **the ideal distribution is system dependent**!

WORKLOAD DISTRIBUTION – KEY IDEAS

- Main **rules** (for large systems):
  - Use *openMP* as soon as possible; maximalize the # threads
  - Use k-points/spins parallelism first
  - Maximalize the block size (increase **`npband`** and **`bandpp`**)
  - Follow the multiplicity rules for **`nband`**, **`nthreads`**, **`npband`**, **`bandpp`**…
- Make **performance comparisons** for every system
- Use `autoparal` and `max_ncpus` variable as a starting guess:
  - **`autoparal=1`**: ABINIT tries to find automatically a suitable distribution
  - **`max_ncpus=N`**: ABINIT prints all possible distributions with **`nproc<=N`** and stops

ABINIT+MPI+OPENMP - EXAMPLES

Varying the block size

# MPI proc. = npband	# threads = bandpp	# blocks	CPU (sec.)
36	32	1	**188**
36	16	2	205
36	8	4	231
36	4	8	360

Test case

64 Pu atoms - 1200 electronic bands

Tera1000 CEA super-computer (Intel KNL)

Changing the MPI – openMP distribution

# MPI proc. = npband	# threads = bandpp	# blocks	CPU (sec.)
570	2	1	1091
285	4	1	918
143	8	1	320
72	16	1	248
36	32	1	**188**
18	64	1	197

OTHER LEVELS OF PARALLELIZATION

- **npimage** → For images : NEB, string method, PIMD
- **npspinor** → spinorial components in case of spin-orbit coupling
- **nppert** → in a DFPT calculation (response) to parallelize over perturbations
- **nphf** → In case of Hartree-Fock calculation
- **paral_atom** → Enable of disable parallelization over atoms (automatically set up)

Performances

SCALAR VS MANYCORE ARCHITECTURE

Test case

64 Pu atoms

1200 electronic bands

Tera1000 CEA supercomputer

	Nehalem	***Haswell***	***KNL***
Process x threads	500 x 1	500 x 1	50 x 4
Number of nodes	63	16	3
CPU time Old implementation	45 min. 36 iterations	27 min. 36 iterations	N/A
CPU time New implementation	10 min. 8 iterations	5 min. 8 iterations	8 min. 8 iterations

ABINIT – MEDIUM SIZE SYSTEM

Gallium oxide Ga_2O_3

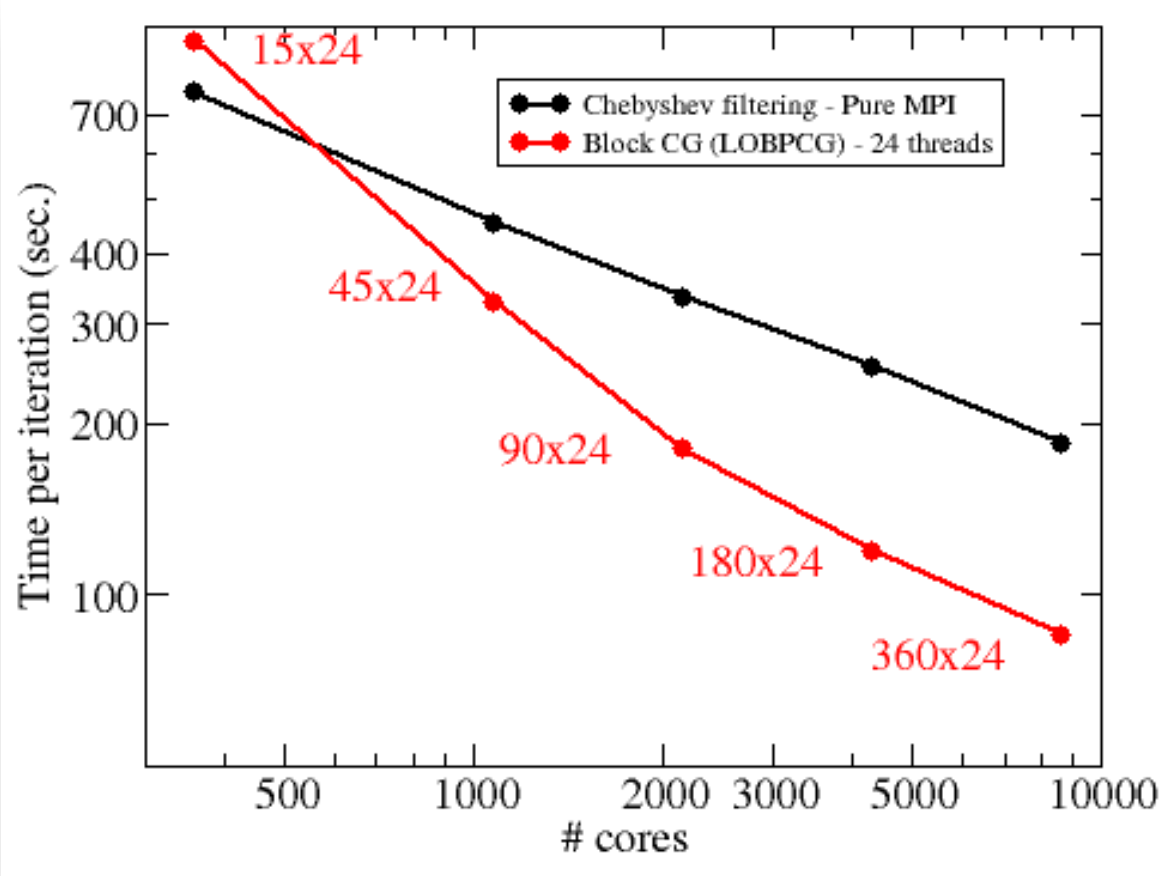
1960 atoms

8700 bands (17400 electrons)

Time per SCF iteration

Bands + FFT parallelism only

Can be mixed to other parallelism levels



TGCC –Joliot-Curie

Intel Skylake

48 cores/node – 2 sockets

ABINIT - LARGE SYSTEM

Gallium oxide Ga_2O_3

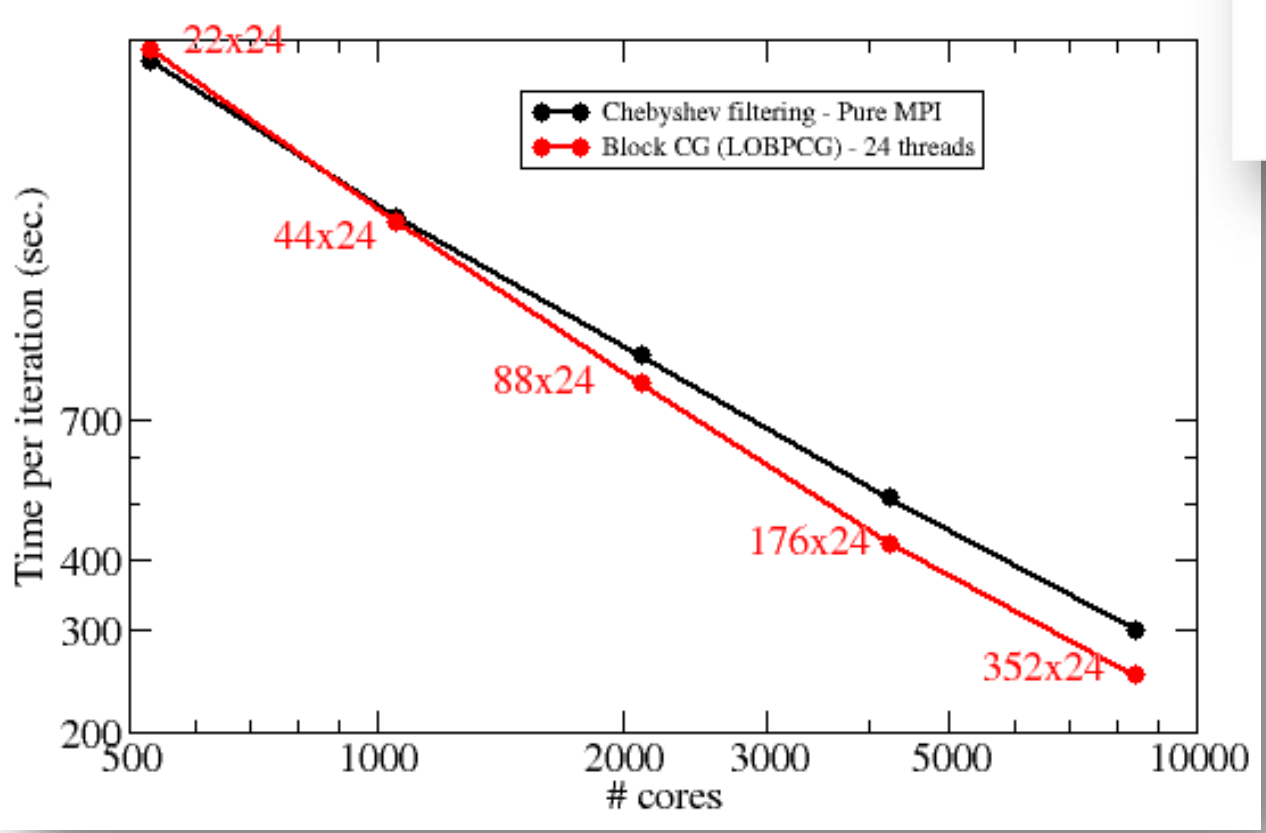
1960 atoms

18400 bands (36800 electrons)

Time per SCF iteration

Bands + FFT parallelism only

Can be mixed to other parallelism levels



TGCC –Joliot-Curie

Intel Skylake

48 cores/node – 2 sockets

Conclusion

ABINIT IN PARALLEL – KEYS POINTS

- ABINIT parallel efficiency is strongly **system dependent**
- It is highly recommended to use **hybrid parallelism**
- ABINIT cannot be used without a **minimum knowledge** of…
  - The computer architecture (nodes, CPUs/node, …)
  - The iterative diagonalization algorithm
- `autoparal` keyword can help…

  But this is only a starting point!

  Manual tuning is always better

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