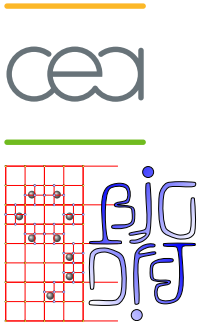


HPC and Multi-GPU

École "Programmation Hybride": une étape vers le many-cœurs

L'ÉSCANDILLE – AUTRANS, FRANCE

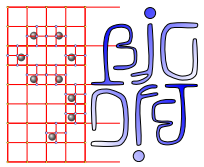
Parallel Codes and High Performance Computing: Massively parallelism and Multi-GPU

Luigi Genovese

L_Sim – CEA Grenoble

October 10, 2012

A basis for nanosciences: the BigDFT project



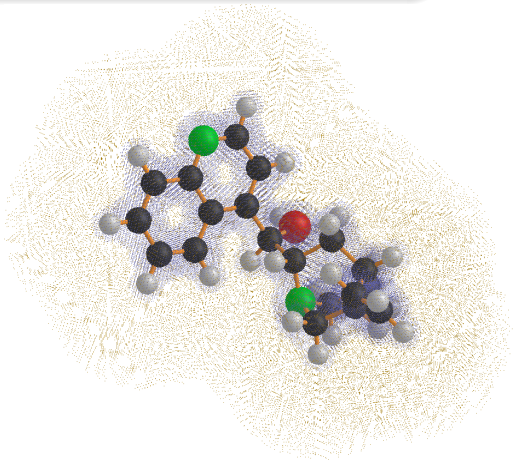
STREP European project: BigDFT(2005-2008)

Four partners, 15 contributors:
CEA-INAC Grenoble (T.Deutsch), U. Basel (S.Goedecker),
U. Louvain-la-Neuve (X.Gonze), U. Kiel (R.Schneider)

Aim: To develop an ab-initio DFT code based on Daubechies Wavelets, to be *integrated in ABINIT*.

BigDFT 1.0 $\longrightarrow$ January 2008

. . . Not only a DFT adventure.



In this presentation

- Present HPC scenario
- Developers' and *users'* challenges
- Outcomes and general considerations

Ab initio calculations with DFT

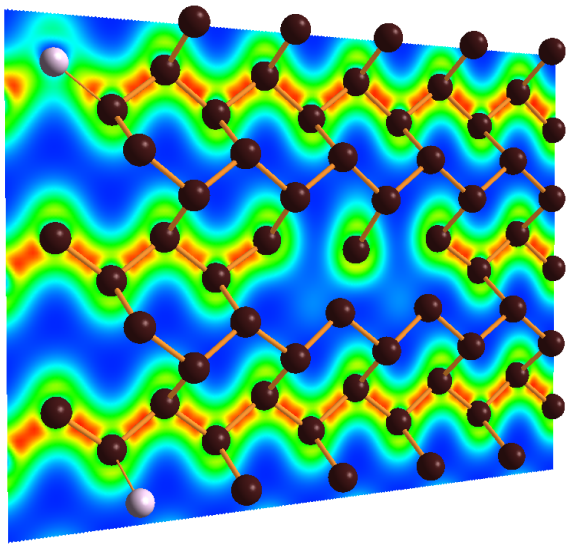
Several advantages

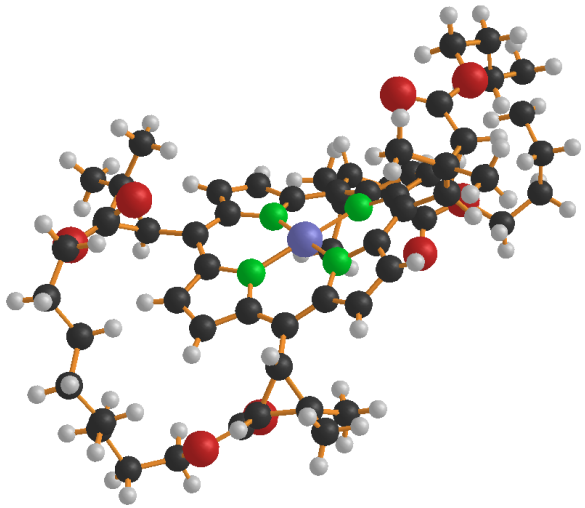
- ✔ Ab initio: No adjustable parameters
- ✔ DFT: Quantum mechanical (fundamental) treatment

Main limitations

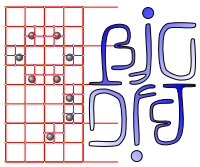
- ✘ Approximated approach
- ✘ Requires high computer power, limited to few hundreds atoms in most cases

Wide range of applications: nanoscience, biology, materials



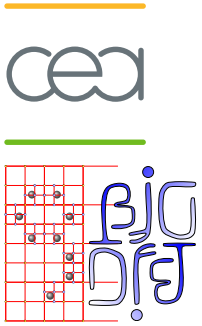


Outline



HPC and Multi-GPU

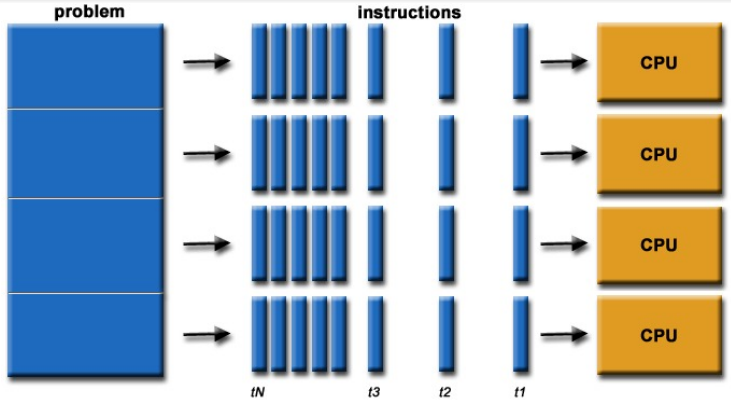
What is Parallel Computing?



HPC and Multi-GPU

Easy to say...

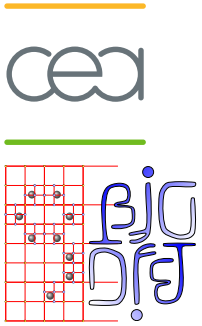
Simultaneous use of multiple compute resources to solve a computational problem



...but not so easy to implement

- A problem is broken in multiple parts which can be solved concurrently
- Each part is associated to a series of instructions
- Instruction from each part are executed simultaneously on different Compute Processing Units

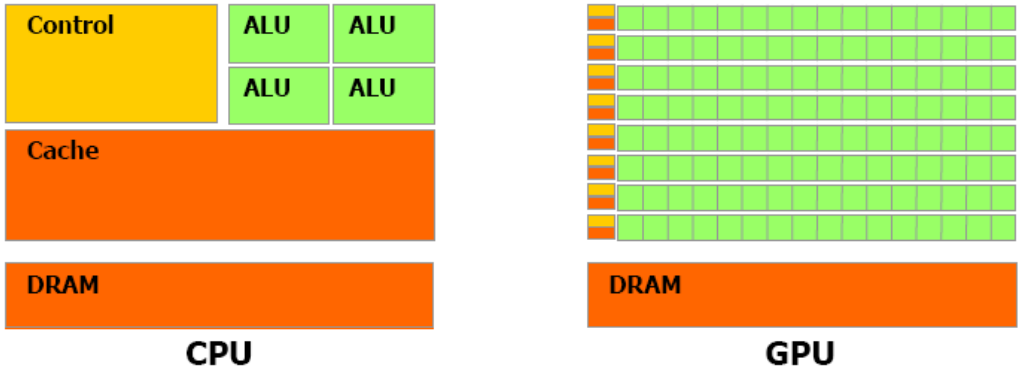
The Compute Processing Unit(s)



A computing machine (node) is made of:

- Control Unit
- Arithmetic Logic Unit
- Memory Unit

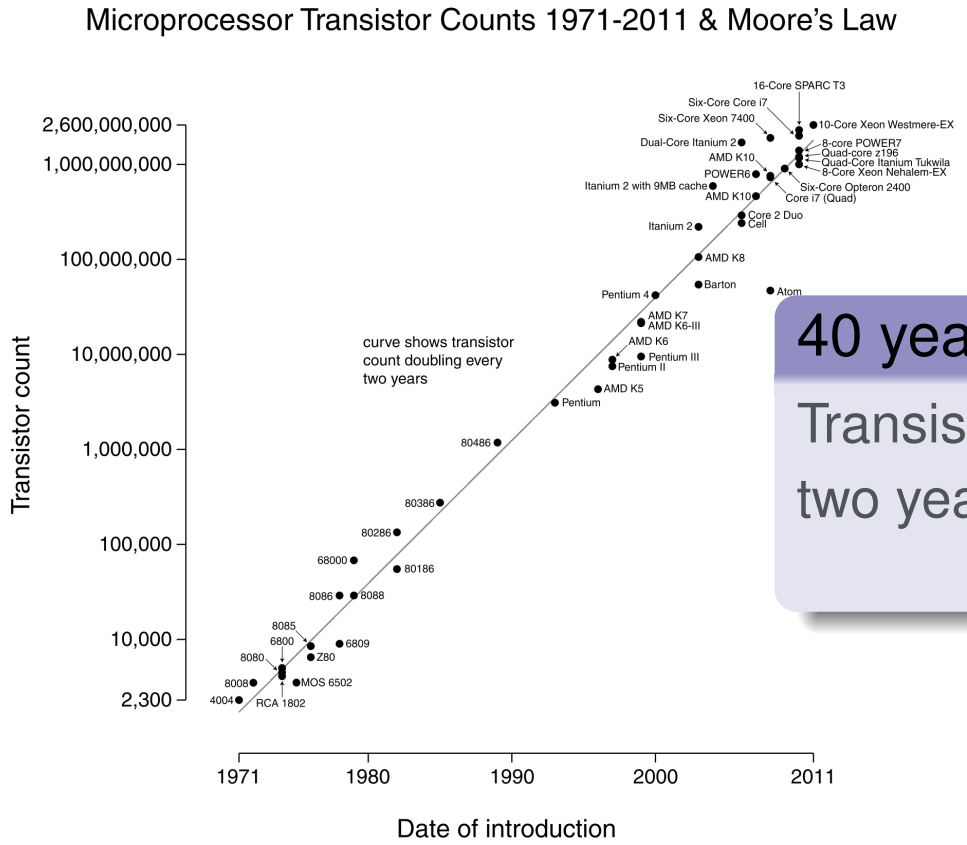
They might exist in different ratio of different architectures



After all, they are transistors

What does technology offer us?

Moore's law



40 years of improvements

Transistor counts double every two years...

...but how?

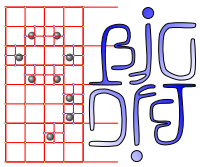
Power is the limiting factor (around 100 W nowadays)

Power $\propto$ Frequency^3 ☛ Clock rate is limited

Multiple slower devices *preferable* than one superfast device

☛ More performance with less power $\rightarrow$ software problem?

Why software problem?



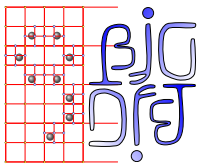
The power cost of frequency

	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

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?

HPC thumb-rules have changed



Frequency-dominated era

- ✔ Parallelism is not improved by the architecture
- ✔ Frequency increases → No. Flop/s increases
- ☛ Code runs faster

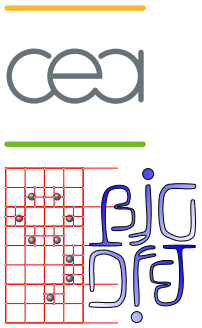
Manycore era

- ✔ Parallelism is *dramatically changed* in the architecture
- ✔ Frequency *decreases*
- ☛ Code runs slower
- ✖ The code should be changed

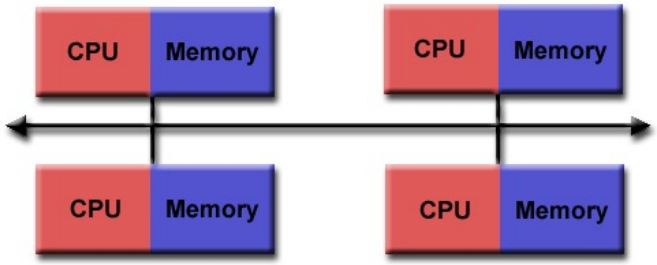
The parallel behaviour of a code (oversimplification)

- Capacity computing: many independent jobs
- Capability computing: single job, parallel intensive

How to parallelize your data?

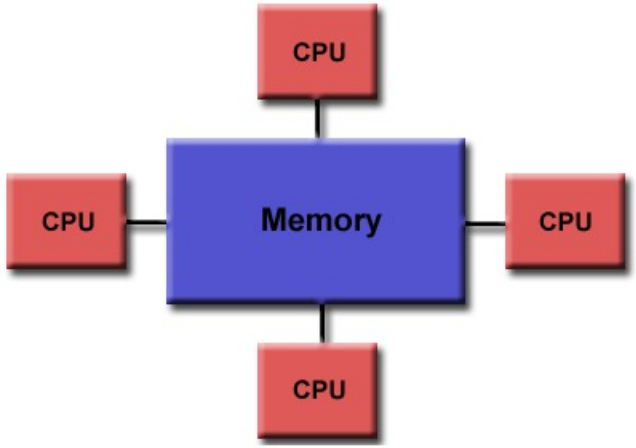


Distributed Memory



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

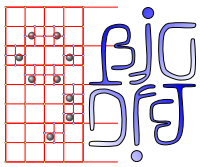
Shared Memory



- Memory is common to all processors
- Threads operate concurrently on data
- Relies on bandwidth perfomances

Memory operations are crucial for parallelism

The cost of the memory transfer



1W $\times$ 1 Year = 1$ (neglecting cooling and storage)

Some facts about memory transfer: Memory bandwidth

40 GB/s (CPU); 20 GB/s (RAM); 3.5 GB/s (interconnect)

Bandwidth evolves less faster than computational power:

✔ $\sim$90's (Math co-processor): 1 Flop/s each 4 Bytes transferred

✖ Nowadays: 62 Flop/s per Bytes transferred

The cost in energy of data movement

Computation: a FMA costs now 100 pJ (10 pJ in the future)

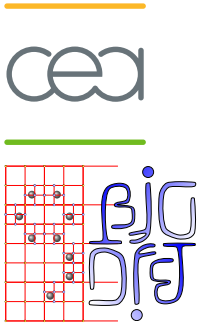
- Move data in RAM costs 4.8 nJ (1.92 nJ)
- Communicating data (MPI) costs 7.5 nJ (2.5 nJ)

☛ Moore's law revisited:

Thread number executions will double each year

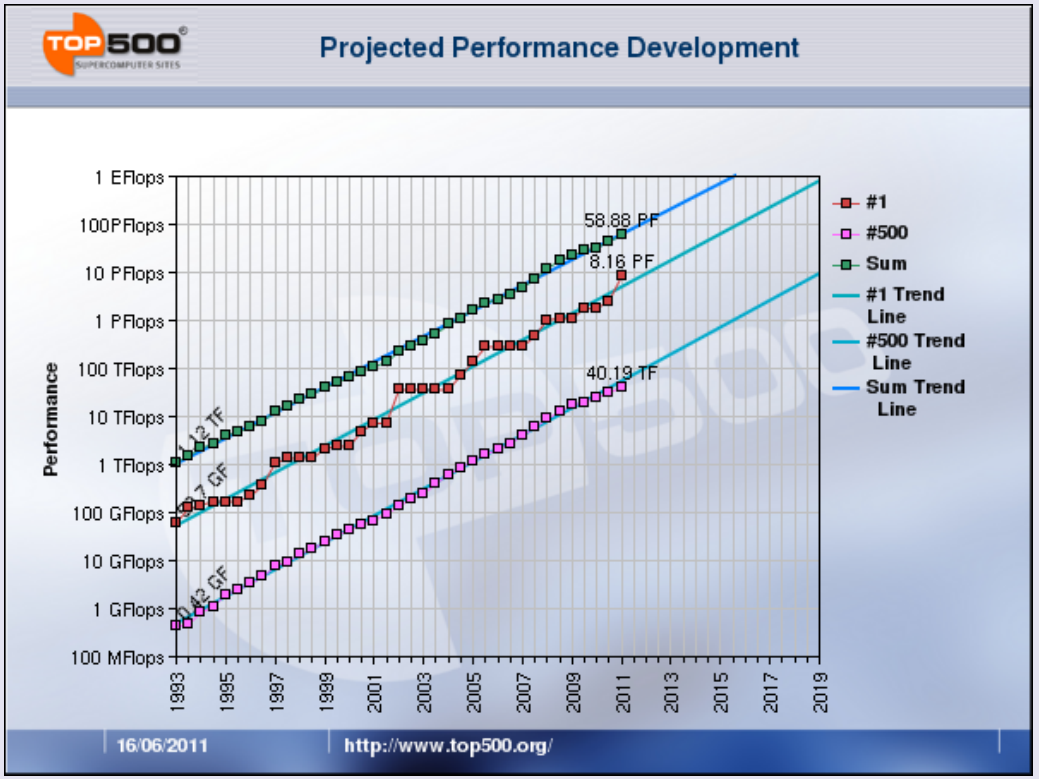
A complicated scenario for HPC (with ab initio) codes

Top500



HPC and Multi-GPU

World Top Supercomputer Ranking



Some considerations

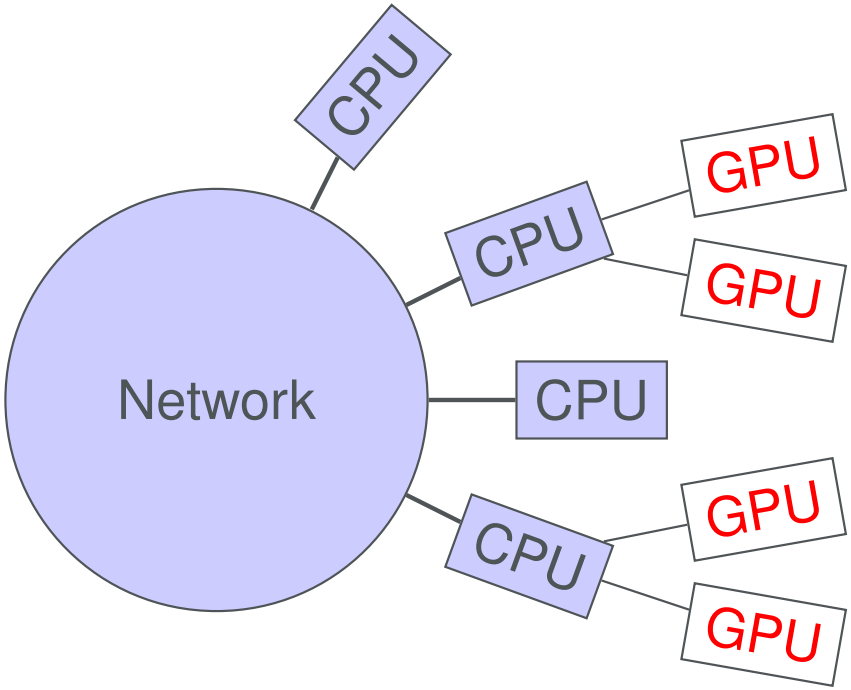
- Faster than Moore's law (doubles every 14 months)

☛ In 8 years top 1 goes off the list

- Hybrid (CPU/GPU) architectures are emerging

Distribute the data on hybrid supercomputer

How a code can be executed on hybrid CPU-GPU architectures?

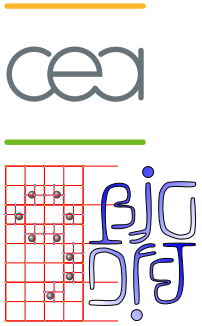


Data transfer is still MPI-based

Only on-board communication between GPU and CPU

Data distribution should depend on the presence of GPUs on the nodes $\rightarrow$ Multilevel parallelization required

Hybrid Supercomputing nowadays



GPGPU on Supercomputers

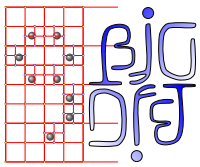
- Traditional architectures are somehow saturating
  More cores/node, memories (slightly) larger but not faster
- Architectures of Supercomputers are becoming hybrid
  3 out to 5 Top Supercomputers are hybrid machines
- Extrapolation: In 2015, No. 500 will become petafloppic
  Likely it will be a hybrid machine

Codes should be conceived differently

- # MPI processes is limited for a fixed problem size
- Performances increase only by enhancing parallelism
- Further parallelisation levels should be added (OpenMP, GPU)

Does (electronic structure calculations) codes are suitable?

Future scenarios for the supercomputing



Exascale is foreseen for 2018: we cannot wait

Simulation: limited money (200 M$) and power (20 MW)
How can you get exascale (1000 times more powerful)?

- 100 times more memory
- 100$\times$ for bandwidth
- Interrupt time 10 times smaller (one each day)

The Blue-Gene like scenario

100 thousands nodes with 1000 cores on it

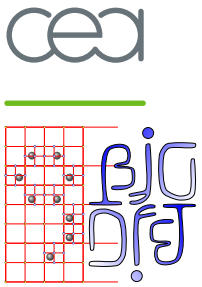
The GPU-like scenario

10 thousands node with 10 thousands "cores"

First observations:

- Huge thread concurrency
- Fault resiliance might become crucial

Future problems from the supercomputing



Architectures change *way* faster than codes

- Number of CPU hours is increasing
- ☛ Should not be scared in asking Mhours

Which *scientific* codes are exempted?

Might we ignore this? What is the price to pay?

- Produce new science by preserving system size (possible only for new scientific domains)
- Stop coding $\rightarrow$ Parallelism is not altered

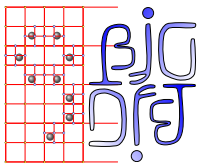
"Easy" things have already been done $\rightarrow$ life is hard

This approach *cannot* last: a (yet) new challenge

- Architectures for HPC are *market* driven
- Low-power is now dominating (smartphones)

BigDFT on ARM architecture: 1/30 of Power, 10 times slower!

How *far* is petaflop (for DFT)?



At present, with traditional architectures

Routinely used DFT calculations are:

- Few dozens (hundreds) of processors
- Parallel intensive operations (blocking communications, 60-70 percent efficiency)
- Not *freshly* optimised (legacy codes, monster codes)

☛ Optimistic estimation: 5 GFlop/s per core $\times$ 2000 cores $\times$ 0.9 = 9 TFlop/s = 200 times less than Top 500's #3!

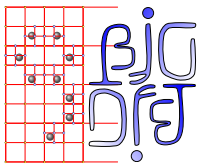
It is such as

Distance Earth-Moon = 384 Mm

Distance Earth-Mars = 78.4 Gm = 200 times more

Moon is reached. . . can we go to Mars? (. . . in 2015?)

Using GPUs in a given (DFT) code



HPC and Multi-GPU

Developer and user dilemmas

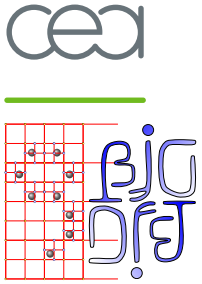
- Does my code fits well? For which systems?
- How much does porting costs?
- Should I always use GPUs?
- How can I interpret results?

Evaluating GPU convenience

Three levels of evaluation

1. Bare speedups: GPU kernels *vs.* CPU routines
   Does the operations are suitable for GPU?
2. Full code speedup on one process
   Amdahl's law: are there hot-spot operations?
3. Speedup in a (massively?) parallel environment
   The MPI layer adds an extra level of complexity

Case study: 1D convolutions (BigDFT code)



Initially, naive routines (FORTRAN?)

$$y(j,I) = \sum_{\ell=L}^{U} h_\ell x(I+\ell, j)$$

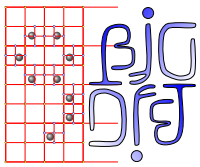
- Easy to write and debug
- Define reference results

```
do j=1,ndat
  do i=0,n1
    tt=0.d0
    do l=lowfil,lupfil
tt=tt+x(i+l,j)*h(l)
    enddo
    y(j,i)=tt
  enddo
enddo
```

Optimisation can then start (Ex. X5550,2.67 GHz)

Method	GFlop/s	% of peak	SpeedUp
Naive (FORTRAN)	0.54	5.1	1/(6.25)
Current (FORTRAN)	3.3	31	1
Best (C, SSE)	7.8	73	2.3
OpenCL (Fermi)	97	20	29 (12.4)

How to optimize?



A trade-off between benefit and effort

FORTRAN based

- ✔ Relatively accessible (loop unrolling)
- ✔ Moderate optimisation can be achieved relatively fast
- ✖ Compilers fail to use vector engine efficiently

Push optimisation at the best

- About 20 different patterns have been studied for one 1D convolution
- Tedious work, huge code $\longrightarrow$ Maintainability?

☛ Automatic code generation?

Consider new programming paradigms

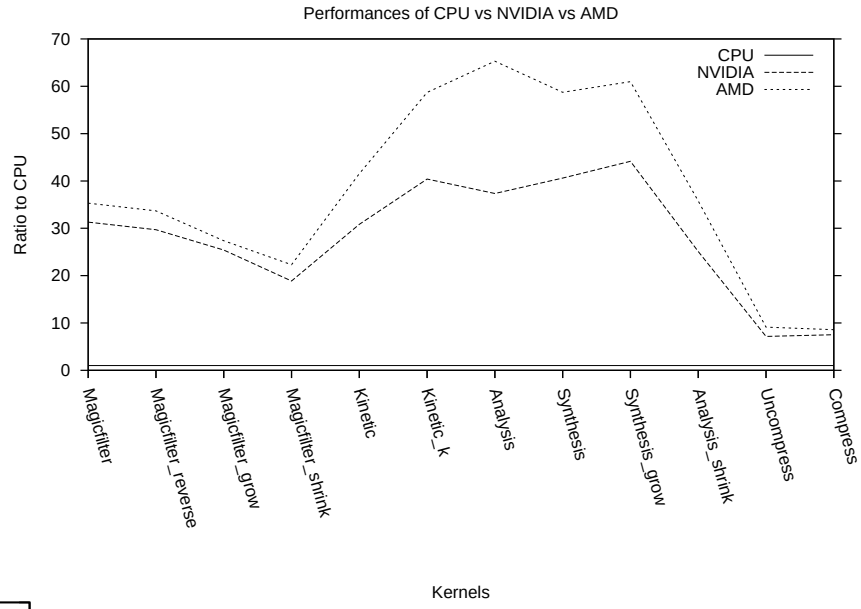
New coding approaches are *most* welcome

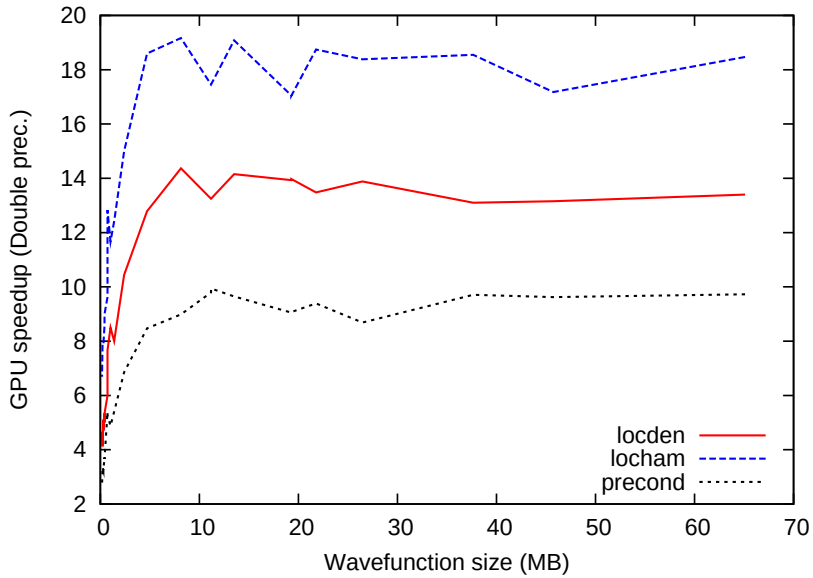
$\rightarrow$ Kronos' OpenCL standard

GPU-ported operations in BigDFT (double precision)

Convolutions Kernels

- ☛ (OpenCL (re)written)
- ✔ Fully functional (all BC)
- ✔ Based on the former CUDA version
- ✔ A 10 to 60 speedup



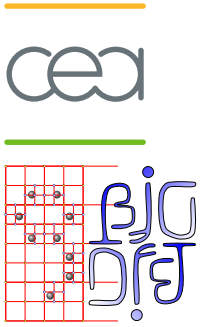


GPU BigDFT sections

GPU speedups between 5 and 20, depending of:

- ✔ Wavefunction size
- ✔ CPU-GPU Architecture

Interpretation of HPC behaviour:



Evaluate the code behaviour:

A not so easy task (especially nowadays)

Frequent mistakes:

- Parallel efficiency is not walltime
- Scalability is not only communication-driven
- Performance evaluation is a multicritierion evaluation process
  - Best scalability (Machine point of view)
  - Best acceleration efficiency (Vendor point of view)
  - Best walltime (User point of view)

  But also robustness, fault tolerance, best Flop/W ratio

Anticipated messages

Far from trivial situation:

- No golden rule
- HPC Optimal Stategies should be *interpreted*

Amdahl's law

A basic concept

The speedup with N cores depends of the parallel fraction (P) of the code:

$$\text{speedup} = \frac{1}{\frac{P}{N} + (1 - P)}$$

It represents the limits to the scalability of a given code

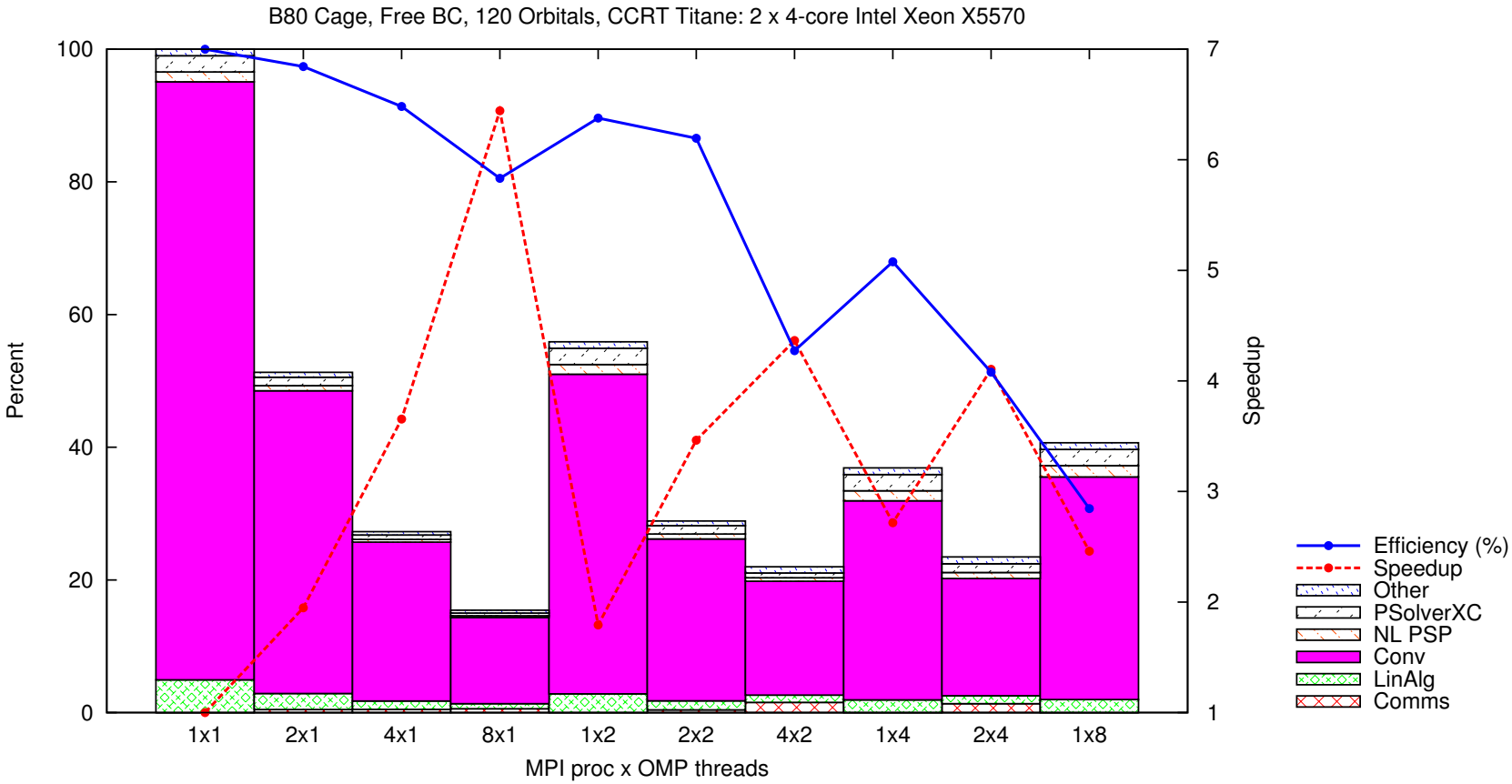
An important definition

Parallel Efficiency $= \frac{Time(N_{ref})}{Time(N)} \frac{N}{N_{ref}}$ Often used as a benchmark of a code in a parallel environment

Lots of factors involved

- Scalability of the problem
- Communication performances
- Computational cost of operations

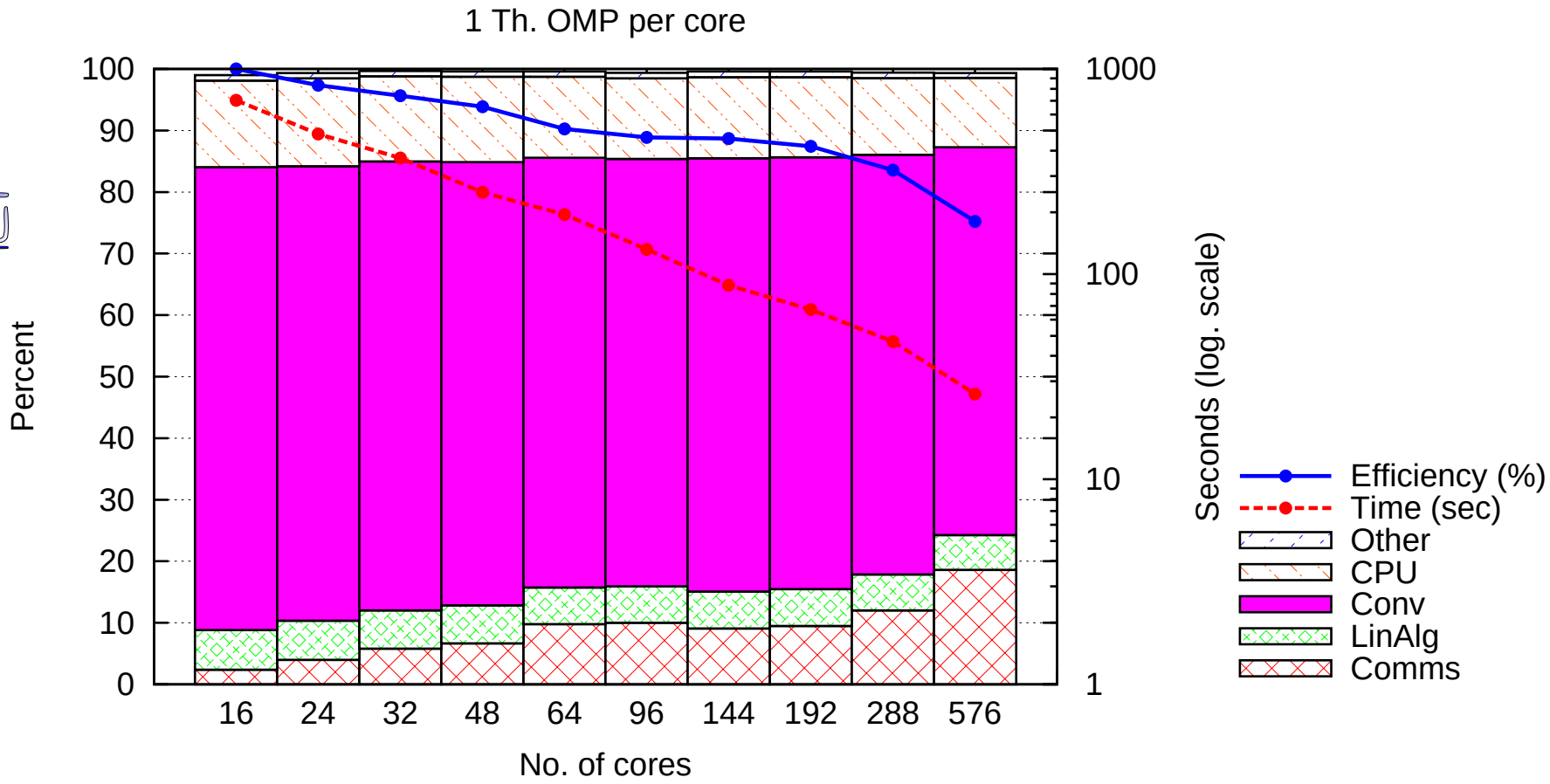
Intranode bandwidth problem



Scalability does not depend only on communication

Amdahl's law is a upper limit!

Task repartition for a small system (ZnO, 128 atoms)



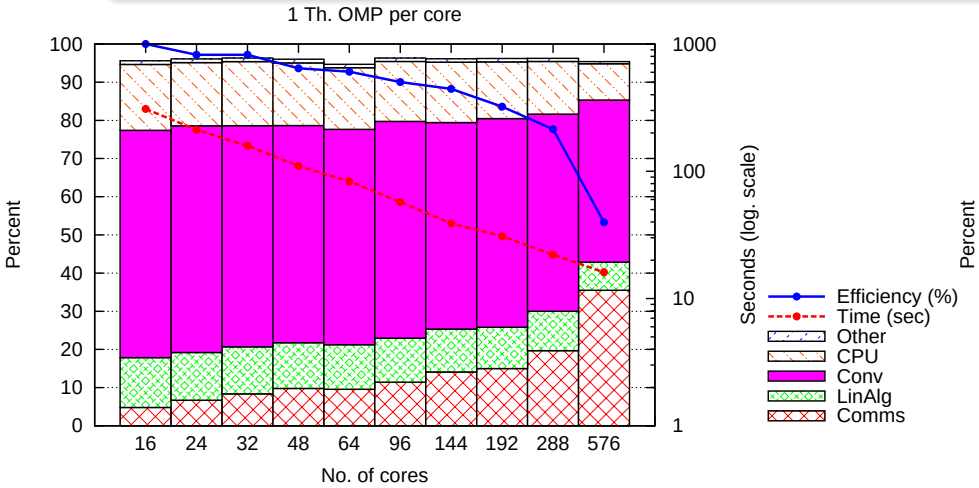
What are ideal conditions for acceleration (e.g. GPU)

To-be-accelerated routines should take the majority of the time

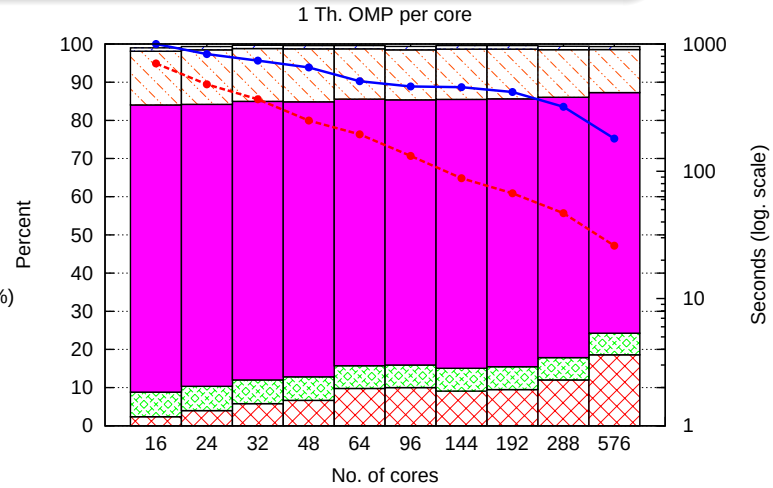
What happens to parallel efficiency?

Parallelisation and architectures

Same code, same runs. Which is the best?



CCRT Titane (Nehalem, Infiniband)



CSCS Rosa (Opteron, Cray XT5)

Titane is 2.3 to 1.6 times faster than Rosa!

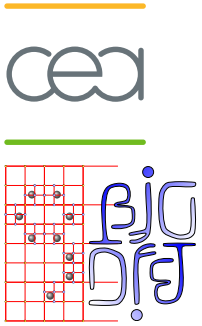
Degradation of parallel performances: why?

1. Calculation power has increased more than networking
2. Better libraries (MKL)

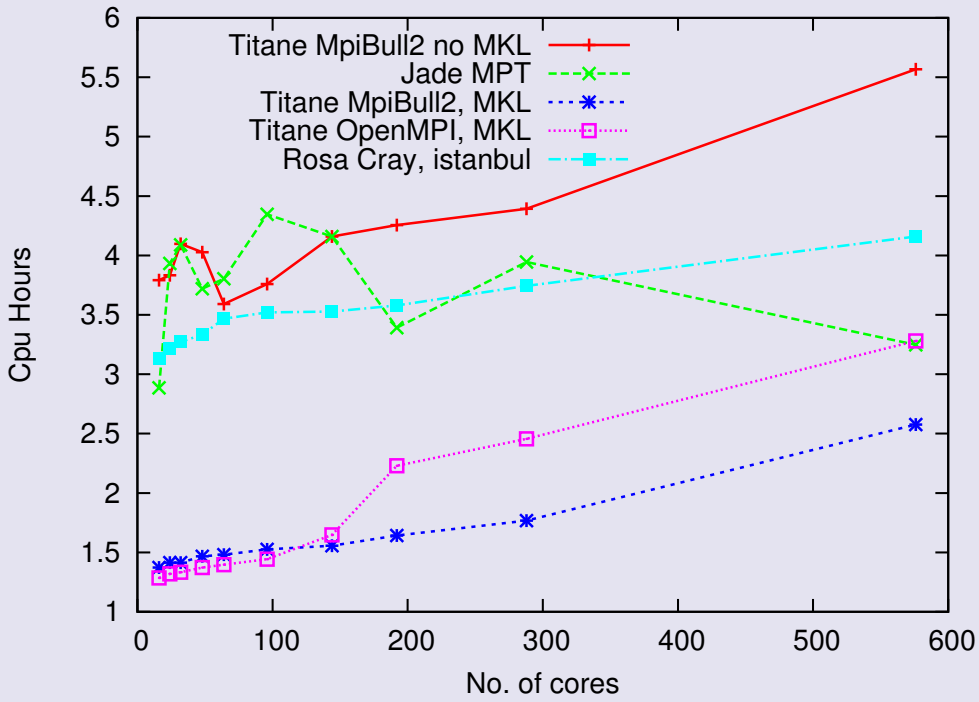
☛ Walltime reduced, but lower parallel efficiency

This will always happen while using GPU!

Architectures, libraries, networking



Same runs, same sources; different user conditions



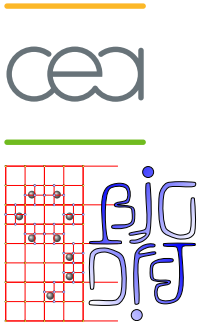
Differences up to a factor of 3!

A case-by-case study

Consideration are often system-dependent, a thumb rule not always exists.

☛ Know your code!

A (even more) frequent mistake



Example: two DFT codes running on the same system.

Naive question: Which one is *faster*?

The running conditions

- Machine generation (CPU, cores, cache,. . .)
- Parallel environment (MPI procs, OMP threads, GPUs)
- Binaries (libraries, compiler,. . .)
- Network vs. Computation performance

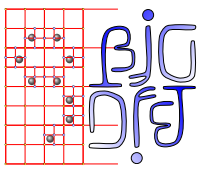
The code conditions (DFT example)

- Basis set (formalism, cut-off, . . .)
- Self-Consistentcy (Input Guess, minimization scheme)

How this question should be posed?

- Which is lowest time-to-solution possible for this system on a given machine?
- Which is the fastest machine for this system?

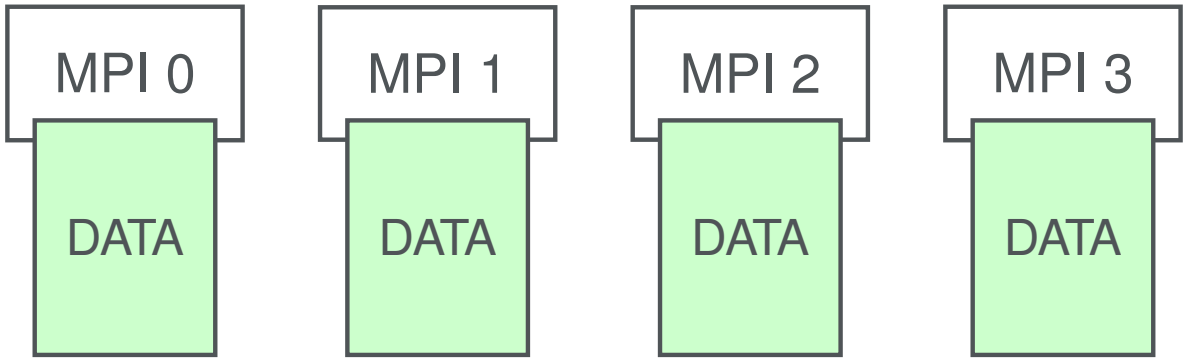
Data repartition on a node



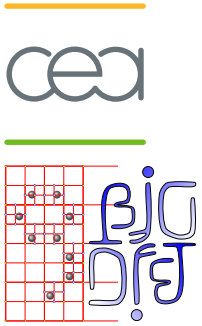
HPC and Multi-GPU

Non-hybrid case

GPU not used $\rightarrow$ homogeneous repartition

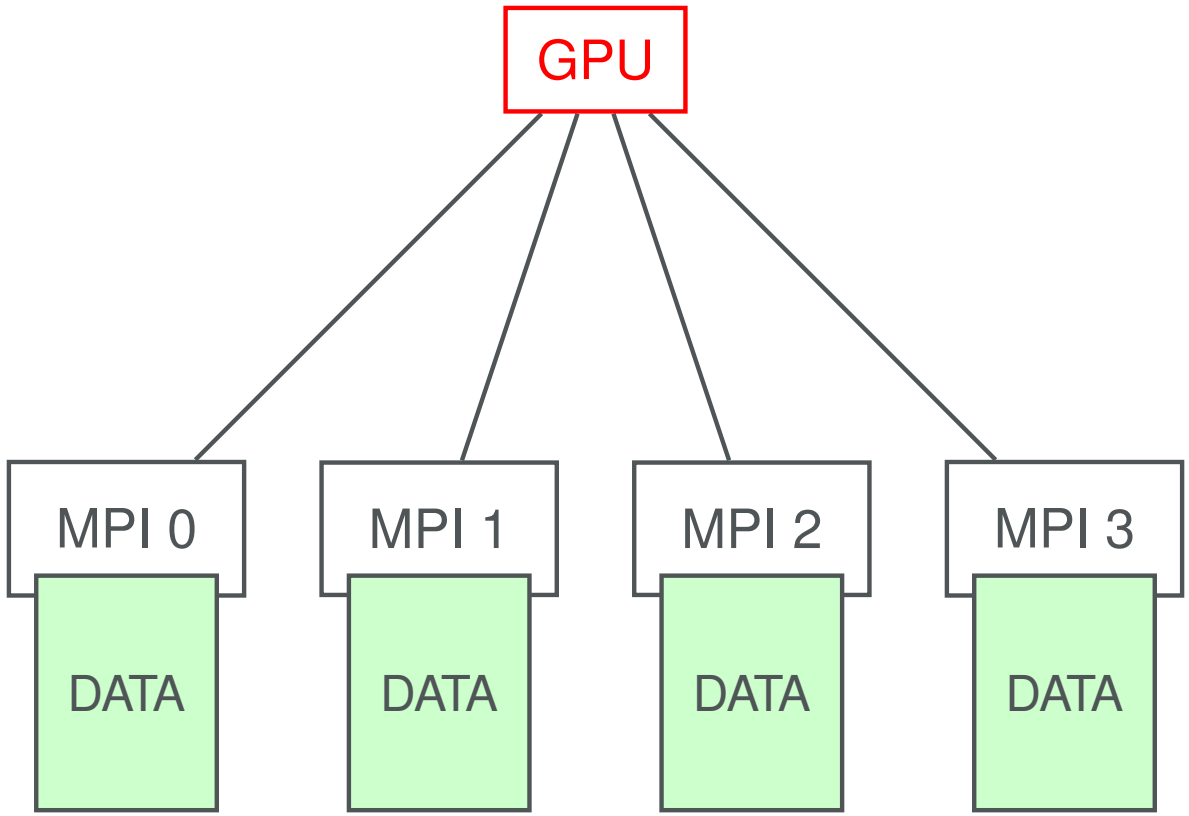


Data repartition on a node

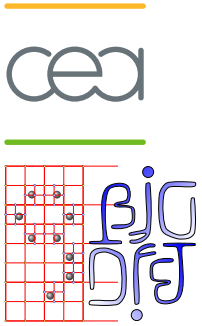


"Naive" repartition

All the cores use the GPU at the same time

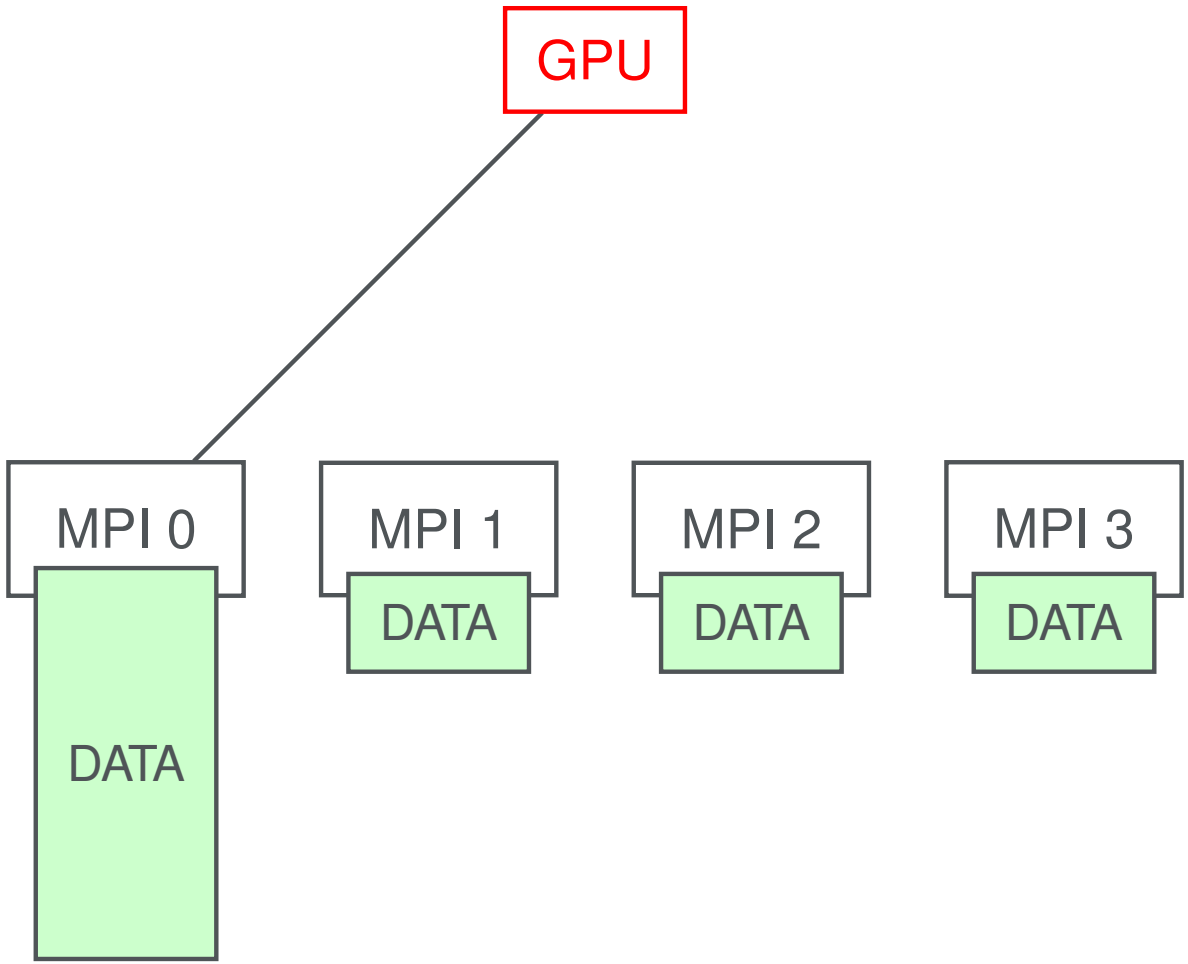


Data repartition on a node

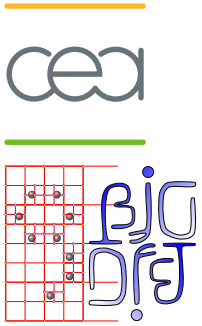


Inhomogeneous repartition

Only one node use the GPU with more data

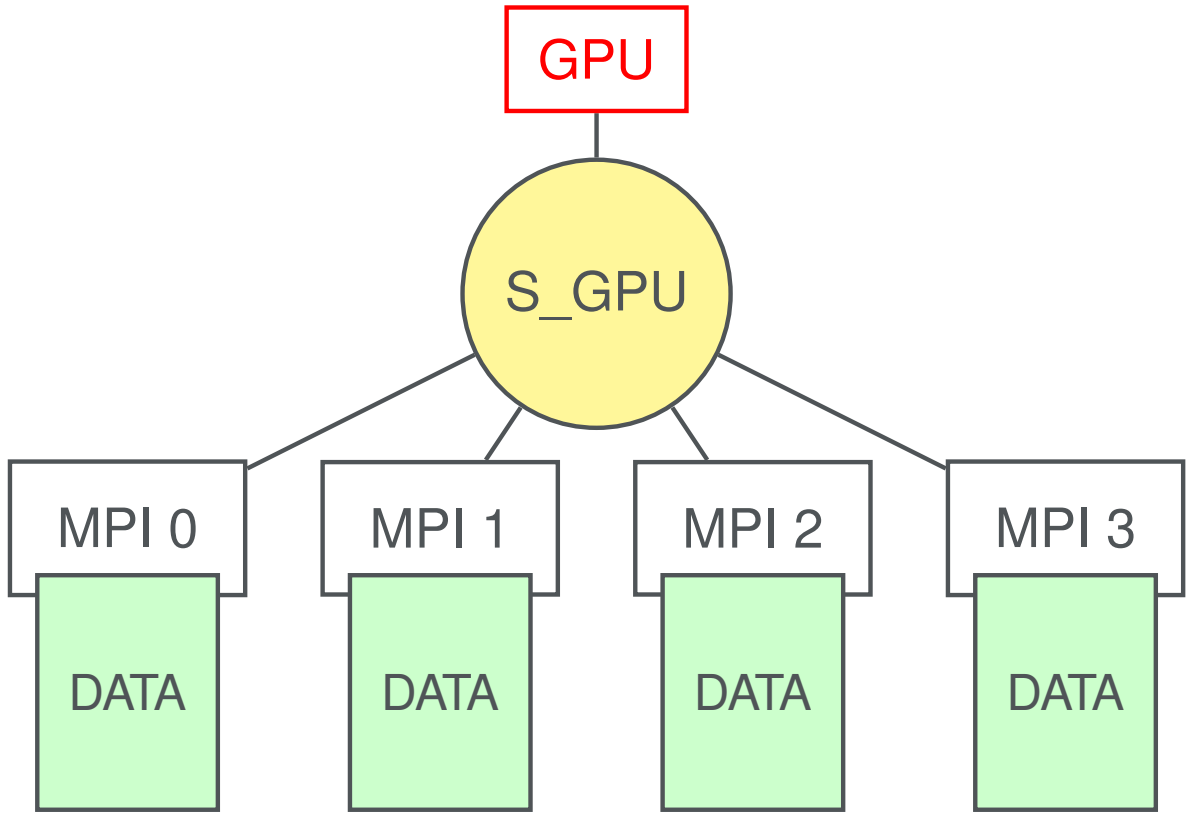


Data repartition on a node

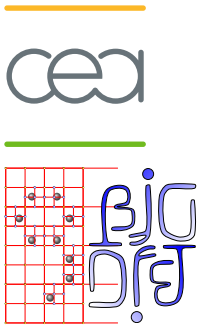


The S_GPU approach

S_GPU library manages GPU resource within the node



S_GPU library (M. Ospici, LIG / Bull / CEA)



De-synchronisation of operations

Two semaphores are activated for each card on the node:

- Data transfer (CPU $\rightarrow$ GPU and GPU $\rightarrow$ CPU)
- Calculation on the GPU

Each operation (e.g. convolution of a wavefunction) is associated to a stream.

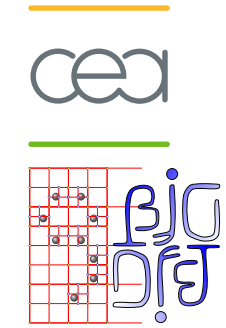
Operation overlap

Calculation and data transfer of different stream may overlap
Operation are scheduled on a first come - first served basis

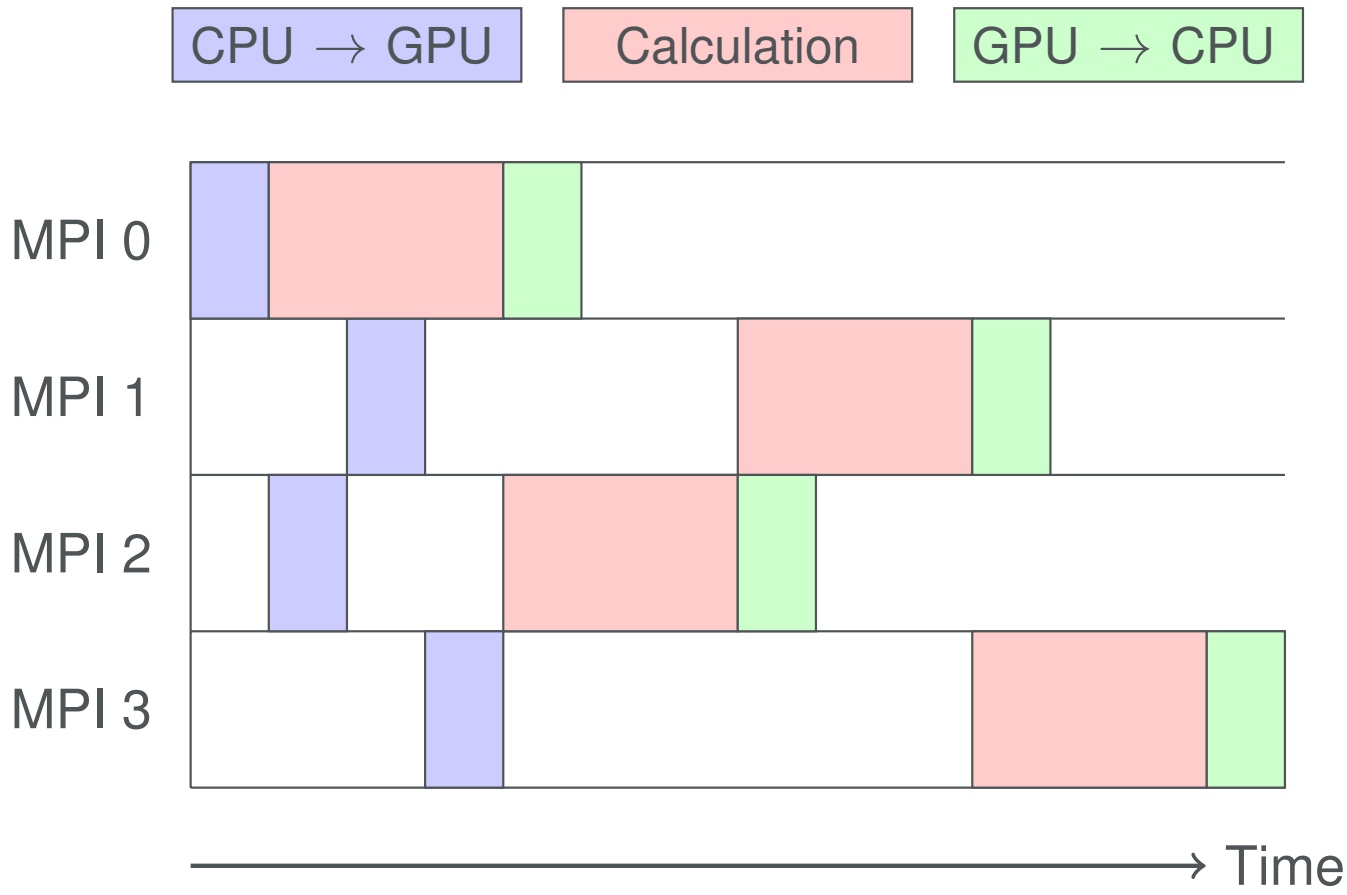
Several advantages

- The time for memory transfers is saved
- Heavy calculation can be passed to the card one - by - one, avoiding scheduling problems

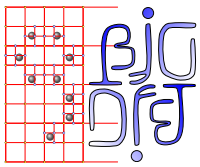
Example of a time chart



The GPU can be viewed as a shared co-processor



Convenience of S_GPU approach (end of 2009)



HPC and Multi-GPU

Different tests thanks to BigDFT flexibility

We have performed many tests, with different ratios GPU/CPU on the same node

Speedup on the full code (examples)

S_GPU is the best compromise speedup/easiness

Examples:

CPU -GPU	8 - 1	8 - 2	4-2	2-2
S_GPU	1.96	3.69	3.73	5.09
Inhomogeneous (best)	2.08	2.64	2.32	2.40

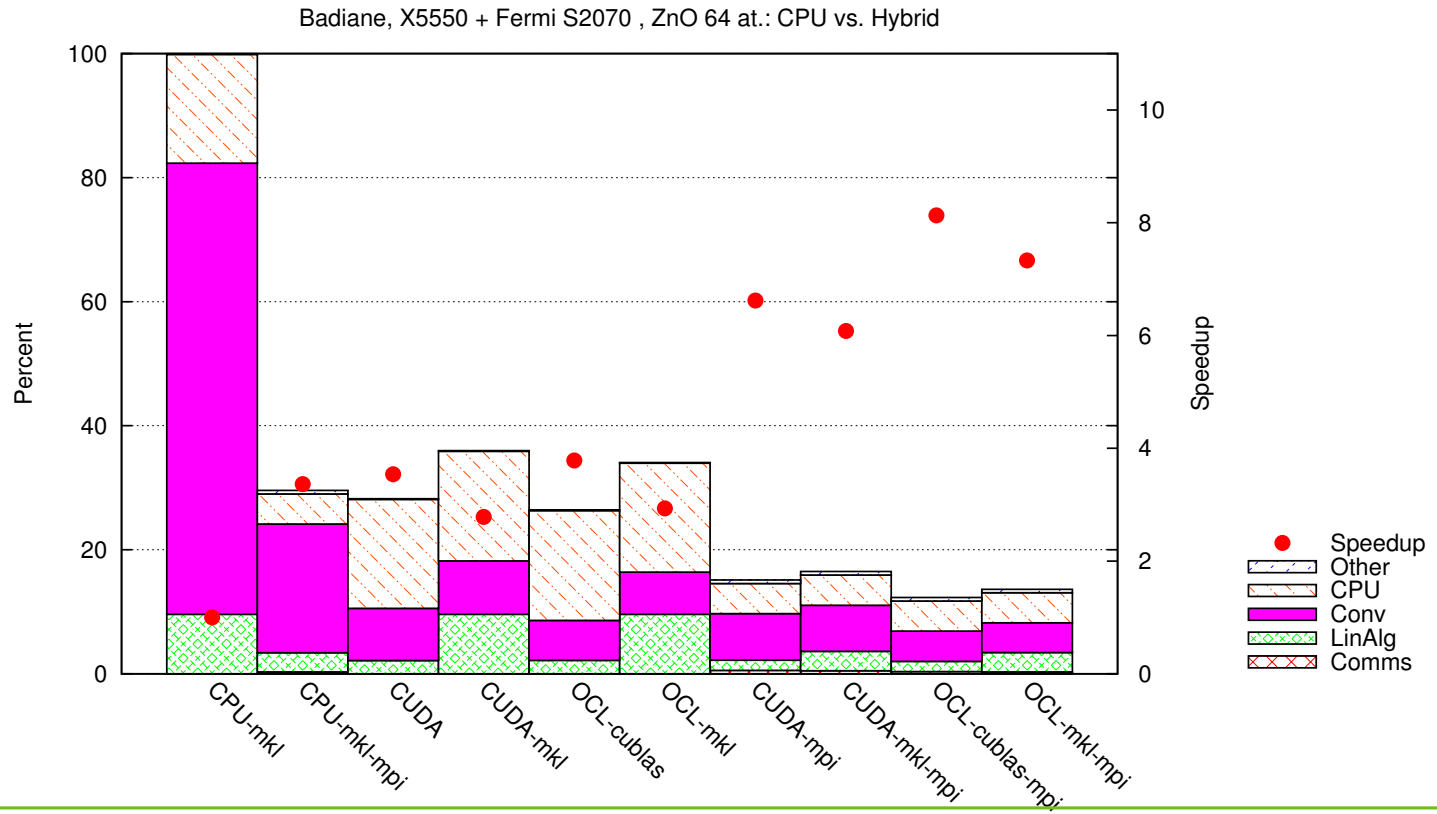
Full code tested on Multi-GPU platforms

- CINES -Iblis
  48 GPU, Prototype calculations
- CCRT - Titane
  Up to 196 GPU (Grand challenge 2009)

Case study: BigDFT in hybrid codes

Acceleration of the full BigDFT code

- Considerable gain may be achieved for suitable systems
  Amdahl's law should always be considered
- Resources can be used concurrently (OpenCL queues)
  More MPI processes may share the same card!



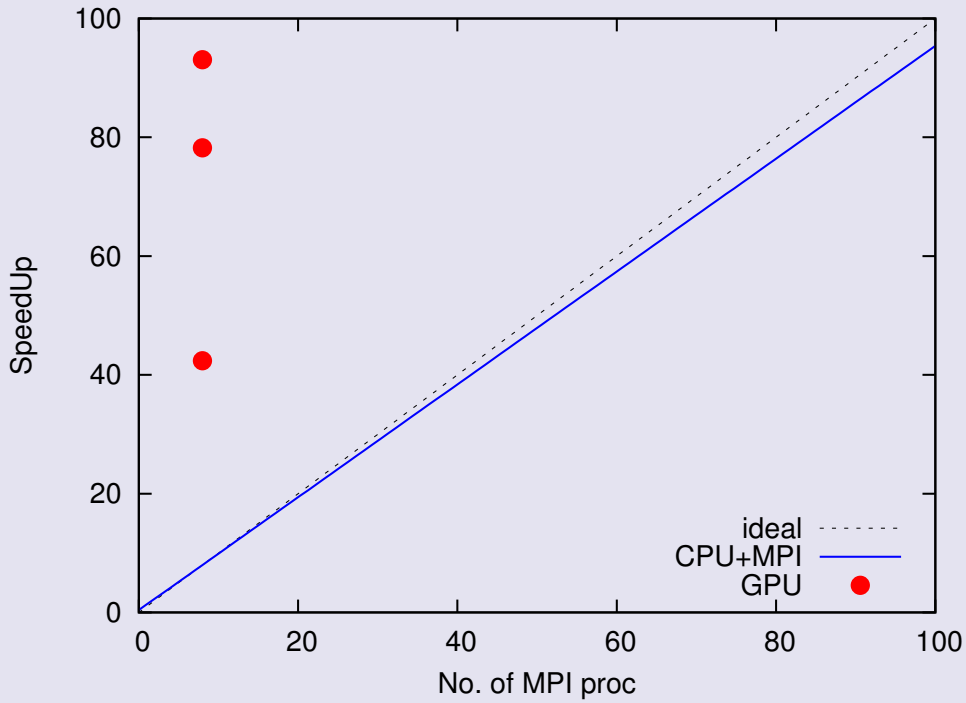
The time-to-solution problem I: Efficiency

Good example: 4 C at, surface BC, 113 Kpts

Parallel efficiency of 98%, convolutions largely dominate.

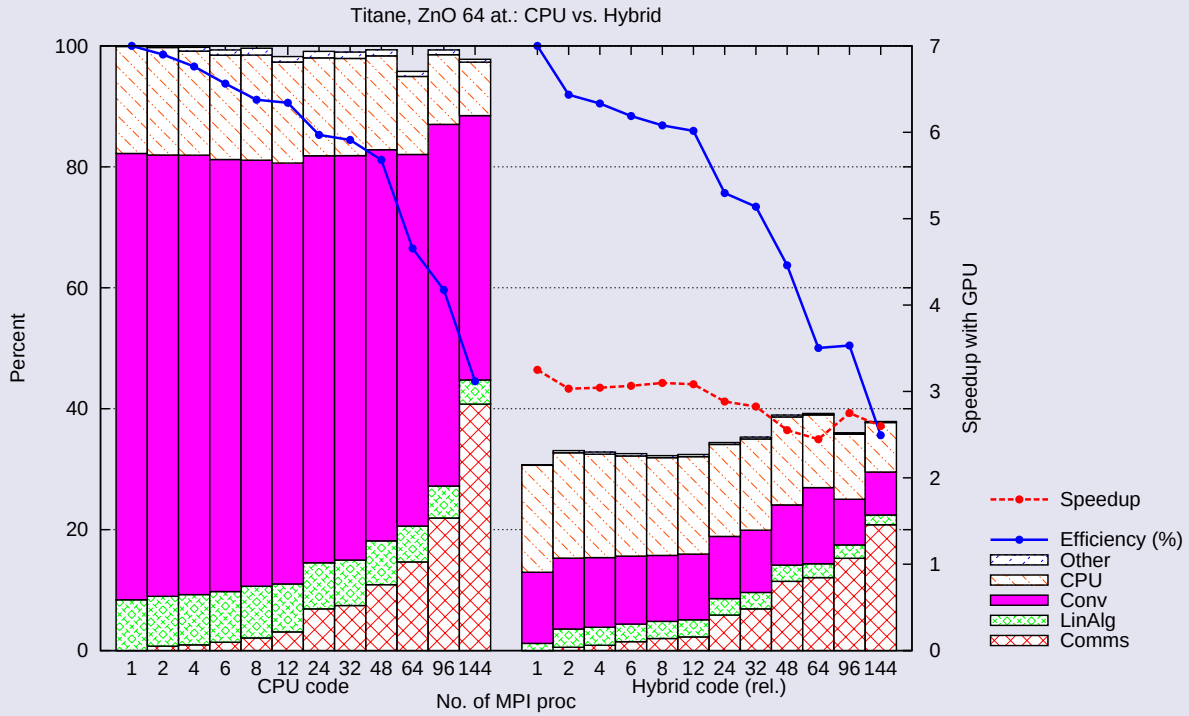
Node:
2× Fermi + 8 × Westmere
8 MPI processes

# GPU added	2	4	8
SpeedUp (SU)	5.3	9.8	11.6
# MPI equiv.	44	80	96
Acceler. Eff.	1	.94	.56



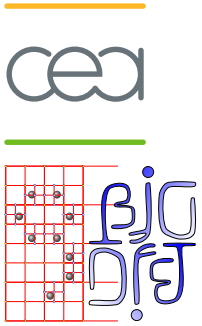
The time-to-solution problem II:Robustness

Not so good example: A too small system



- ✘ CPU efficiency is poor (calculation is too fast)
- ✘ Amdahl's law not favorable (5x SU at most)
- ✔ GPU SU is almost independent of the size
- ✔ The hybrid code *always* goes faster

A look in near future: science with HPC codes



A concerted set of actions

- Improve codes functionalities for present-day and next generation supercomputers
- Test and develop new formalisms
- ☛ Transform challenges in opportunities (needs work!)

The Mars mission

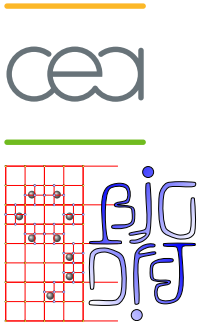
Is Petaflop performance possible?

- Multilevel parallelization $\rightarrow$ one order of magnitude
- Bigger systems, heavier methods $\rightarrow$ (more than) one order of magnitude bigger

Two challenges comes from HPC

- Conceive unprecedented things on new machines
- Preserve and maintain to-date functionalities on *future* machines

A rapidly evolving situation



Architecture evolutions

- Manycore era (multilevel parallelisation)
- Memory traffic as *the* limiting factor

Software evolutions

- Superposition of parallelization layers
- Optimization issues: maintainability vs. robustness

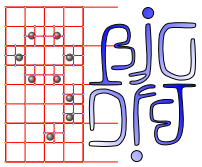
Users ability

- Architecture dimensioning: adapt the runs to the system
- Performance evaluation approach

And it is not going better:

- New set of architectures (GPU, MIC, BG/Q,...)
- New development paradigms (MPI, OpenMP, OpenCL,...)
- HPC codes must follow (HPC projects, Users how-to,...)

General considerations



What is desirable? (Does it *open* new directions?)

Performance should lead to improvements

Optimisation effort

- Know the code behaviour and features
  Careful performance study of the complete algorithm
- Identify and *make modular* critical sections
  Fundamental for mainainability and architecture evolution
- Optimisation cost: consider *end-user* running conditions
  Robustness is more important than best performance

Performance evaluation know-how

- No general thumb-rule: what means High Performance?
  A multi-criterion evaluation process
- Multi-level parallelisation always to be used
  Your code will not (anymore) become faster via hardware