

Input-Output (IO) strategies and HDF5 format

(Pencil Code School, Geneva/CERN, 21st of September 2025)

Philippe-A. Bourdin

Philippe.Bourdin@uni-graz.at

Overview:

- * Comparison of IO strategies
- * Available IO modules
- * Features of HDF5
- * Conversion tool => pc_convert_hdf5.pro

Collective IO and ghost cells

Collective IO and ghost cells

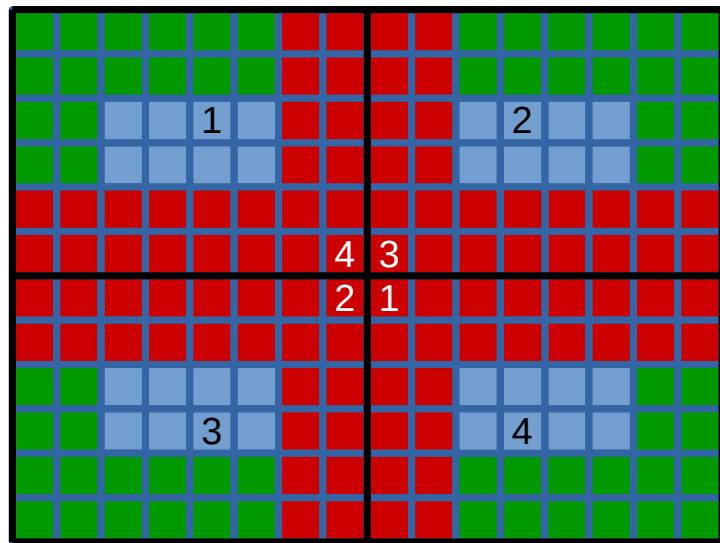
Possibility to save storage space and improve writing speed?

Collective IO and ghost cells

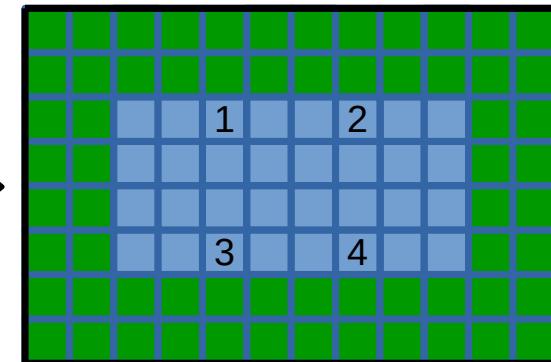
Possibility to save storage space and improve writing speed?

- Removal of inner ghost cells:

io_dist



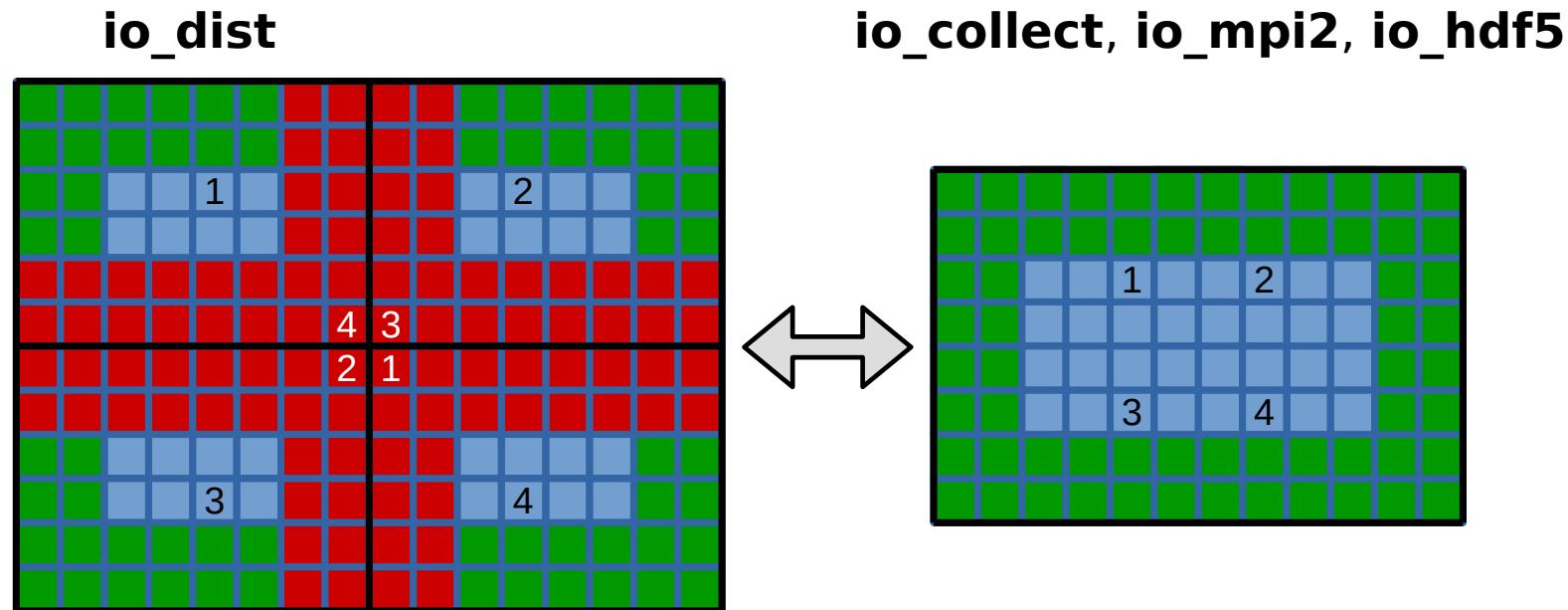
io_collect, io_mpi2, io_hdf5



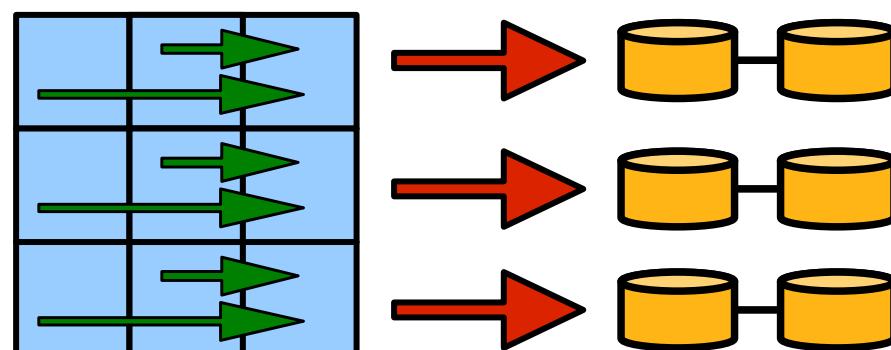
Collective IO and ghost cells

Possibility to save storage space and improve writing speed?

- Removal of inner ghost cells:



- **io_collect_xy**: collect data on special IO nodes; not all ghost cells removed:



How to install and use HDF5?

- On ubuntu systems, just install the packages:
“libhdf5-openmpi-dev” (mandatory)
“h5tools” (optional), **“hdfview”** (optional)
- On a supercomputer:
“module load ...hdf5...” (check „**module avail**“ for available packages)

How to install and use HDF5?

- On ubuntu systems, just install the packages:
“**libhdf5-openmpi-dev**” (mandatory)
“**h5tools**” (optional), “**hdfview**” (optional)
- On a supercomputer:
“**module load ...hdf5...**” (check „**module avail**“ for available packages)
- Change “src/Makefile.local” (see “samples/corona” for an example):
IO = io_hdf5
=> „pc_build“ will then automatically find the HDF5 compiler wrapper.

Massive parallel file access

Massive parallel file access

What is possible regarding IO?

1024*1024*256:

- “io_dist.f90“ writes distinct files from each processor **2 s**
(fastest, filesystem heavily loaded, stores all inner ghost cells)
- “io_collect.f90“ collects everything on one processor **70 s**
(efficient only for small setups, stores no inner ghost cells)

Massive parallel file access

What is possible regarding IO?

1024*1024*256:

- “io_dist.f90“ writes distinct files from each processor **2 s**
(fastest, filesystem heavily loaded, stores all inner ghost cells)
- “io_collect.f90“ collects everything on one processor **70 s**
(efficient only for small setups, stores no inner ghost cells)
- “io_collect_xy.f90“ collects everything on the xy-leading processor **10 s**
(still fast also for many processors, stores some inner ghost cells)
- “io_mpi.f90“ collects everything using available processors **8 s**
(fast, binary format, requires self-written PC reading routines)
- “io_hdf5.f90“ collects everything using all processors **9 s**
(fast, self-explaining extendible data format, readable everywhere)

Massive parallel file access

What is possible regarding IO?

512*512*512:

Run #	io_ref	io_dist	io_hdf5	io_collect	io_collect_xy	io_mpi2	wall times [h]
1	0.12600	0.08774	0.27500	0.16100	0.08233	0.36800	
2	0.06799	0.08130	0.31300	0.22100	0.14100	0.34600	
3	0.06032	0.07303	0.29600	0.21900	0.09754	0.32100	
4	0.03970	0.06528	0.24700	0.22200	0.10700	0.33400	
5	0.04528	0.08753	0.28700	0.18300	0.04515	0.34800	
mean value	0.06786	0.07898	0.28360	0.20120	0.09460	0.34340	[h]
standard deviation	0.03442	0.00973	0.02471	0.02779	0.03504	0.01749	[h]
relative fluctuation	51 %	12 %	9 %	14 %	37 %	5 %	
fastest run	0.0397	0.06528	0.247	0.161	0.04515	0.321	[h]
snapshots		20	20	20	20	20	
time per snapshot		4.6044	37.314	21.834	0.981	50.634	[s]
speed loss		1,00	8,10	4,74	0,21	11,00	
confidence interval	0,01995	0,00564	0,01433	0,01611	0,02031	0,01014	

=> large fluctuations & **io_collect_xy** faster than **io_dist**!?

Massive parallel file access

What is possible regarding IO?

512*512*512:

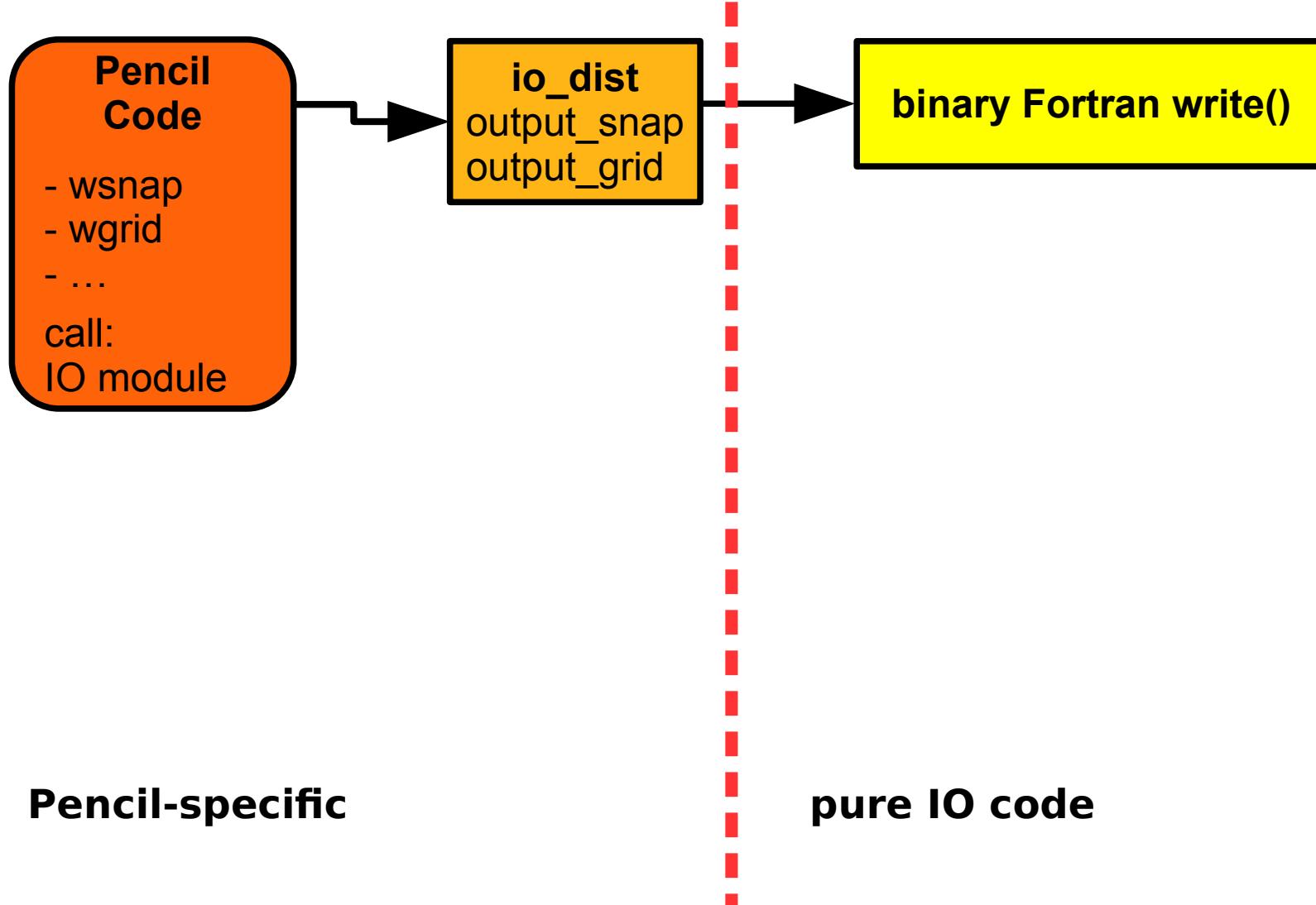
Run #	io_ref	io_dist	io_hdf5	io_collect	io_collect_xy	io_mpi2	wall times [h]
1	0.006466	0.01236	0.19500	0.102	0.02359	0.230	
2	0.005980	0.01193	0.19400	0.102	0.02694	0.226	
3	0.005525	0.01374	0.19800	0.103	0.02512	0.222	
4	0.005762	0.01163	0.19500	0.106	0.03220	0.224	
5	0.005981	0.01229	0.19300	0.106	0.03808	0.223	
6	0.005197	0.01308	0.18900	0.106	0.02451	0.230	
7	0.005073	0.01443	0.19100	0.106	0.03448	0.226	
8	0.005651	0.01263	0.19400	0.105	0.04865	0.224	
9	0.004900	0.01278	0.19600	0.102	0.02783	0.224	
10	0.005846	0.01178	0.20200	0.103	0.03184	0.224	
mean value	0.00564	0.01267	0.19470	0.10410	0.03132	0.22530	[h]
standard deviation	0.00048	0.00089	0.00359	0.00185	0.00770	0.00275	[h]
relative fluctuation	8 %	7 %	2 %	2 %	25 %	1 %	
fastest run	0.0049	0.01163	0.189	0.102	0.02359	0.222	[h]
snapshots		20	20	20	20	20	
time per snapshot		1.2114	33.138	17.478	3.3642	39.078	[s]
speed loss		1,00	27,36	14,43	2,78	32,26	
confidence interval	0,00028	0,00051	0,00208	0,00107	0,00446	0,00159	

=> less fluctuations, but obviously **not an optimal setup** (huge differences)...

IO abstraction layer for HDF5

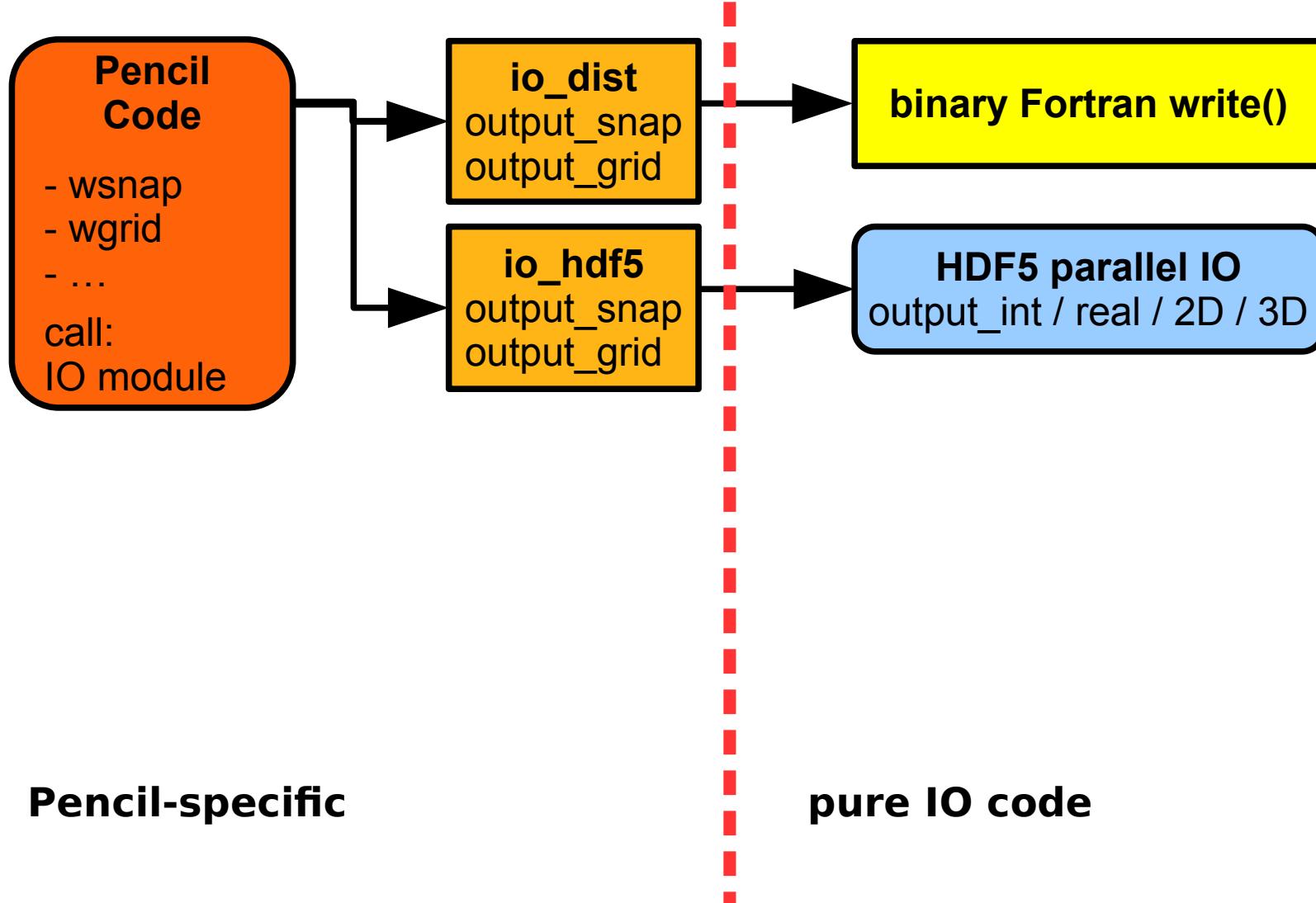
IO abstraction layer for HDF5

Separation of Pencil-specific code from IO code:



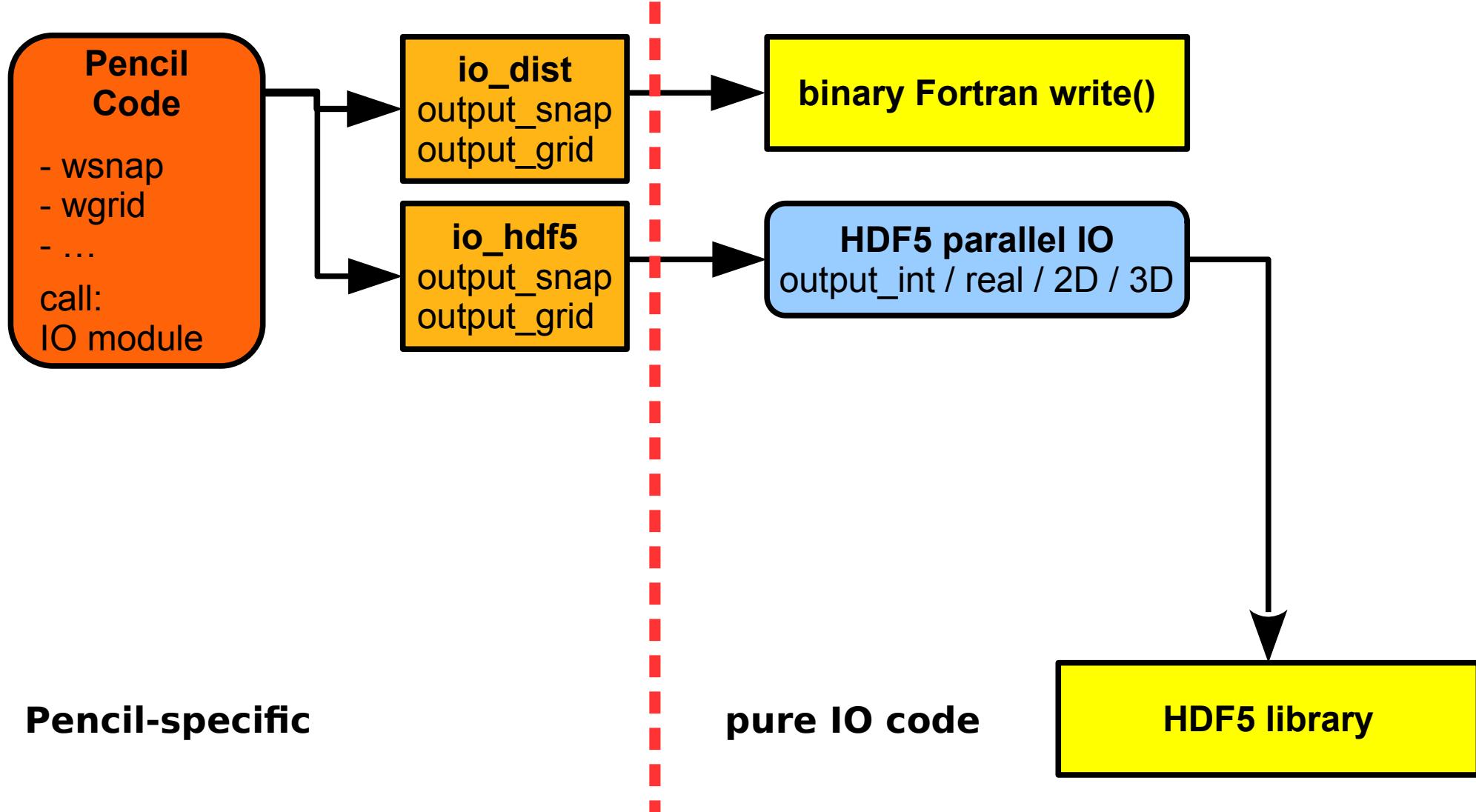
IO abstraction layer for HDF5

Separation of Pencil-specific code from IO code:



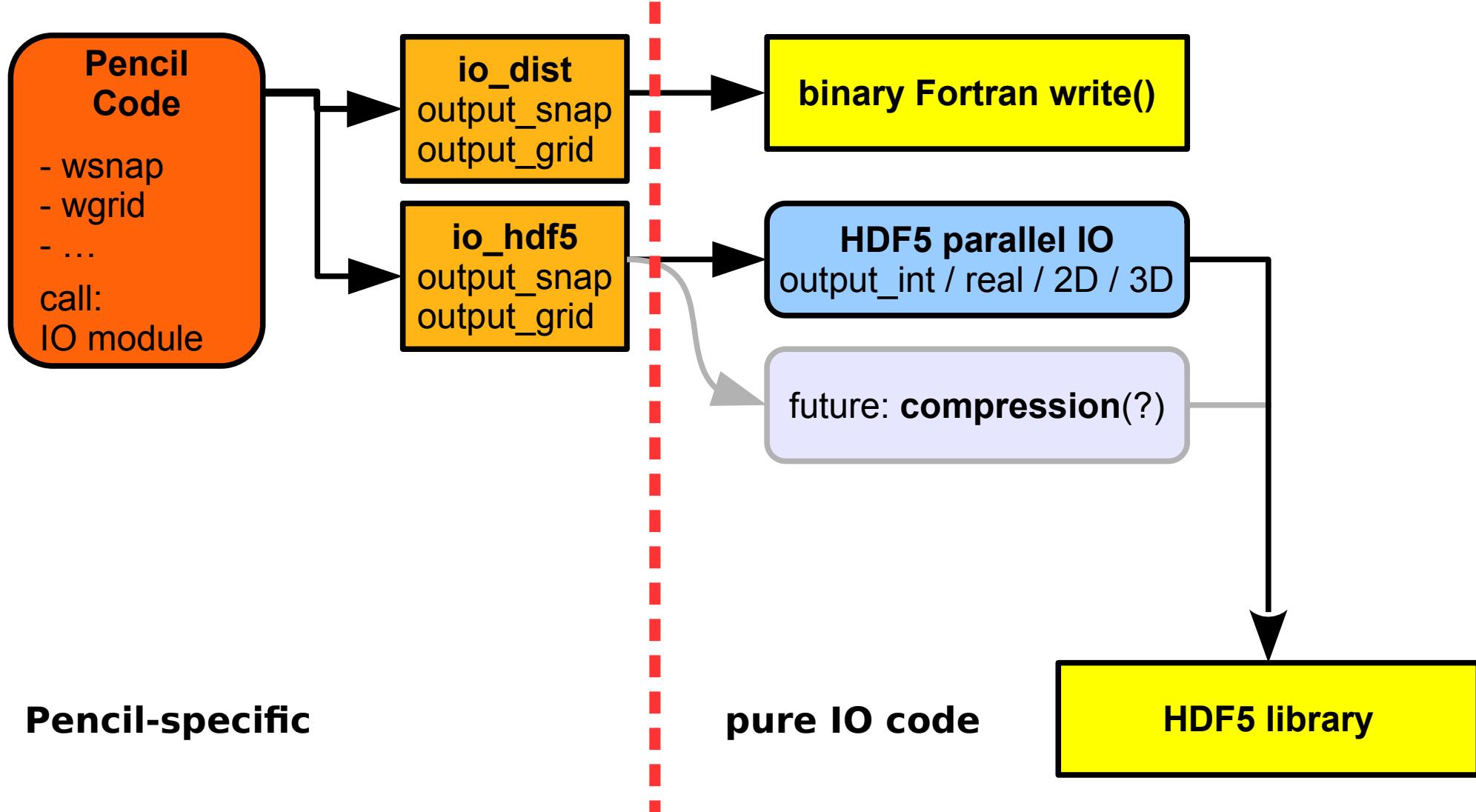
IO abstraction layer for HDF5

Separation of Pencil-specific code from IO code:



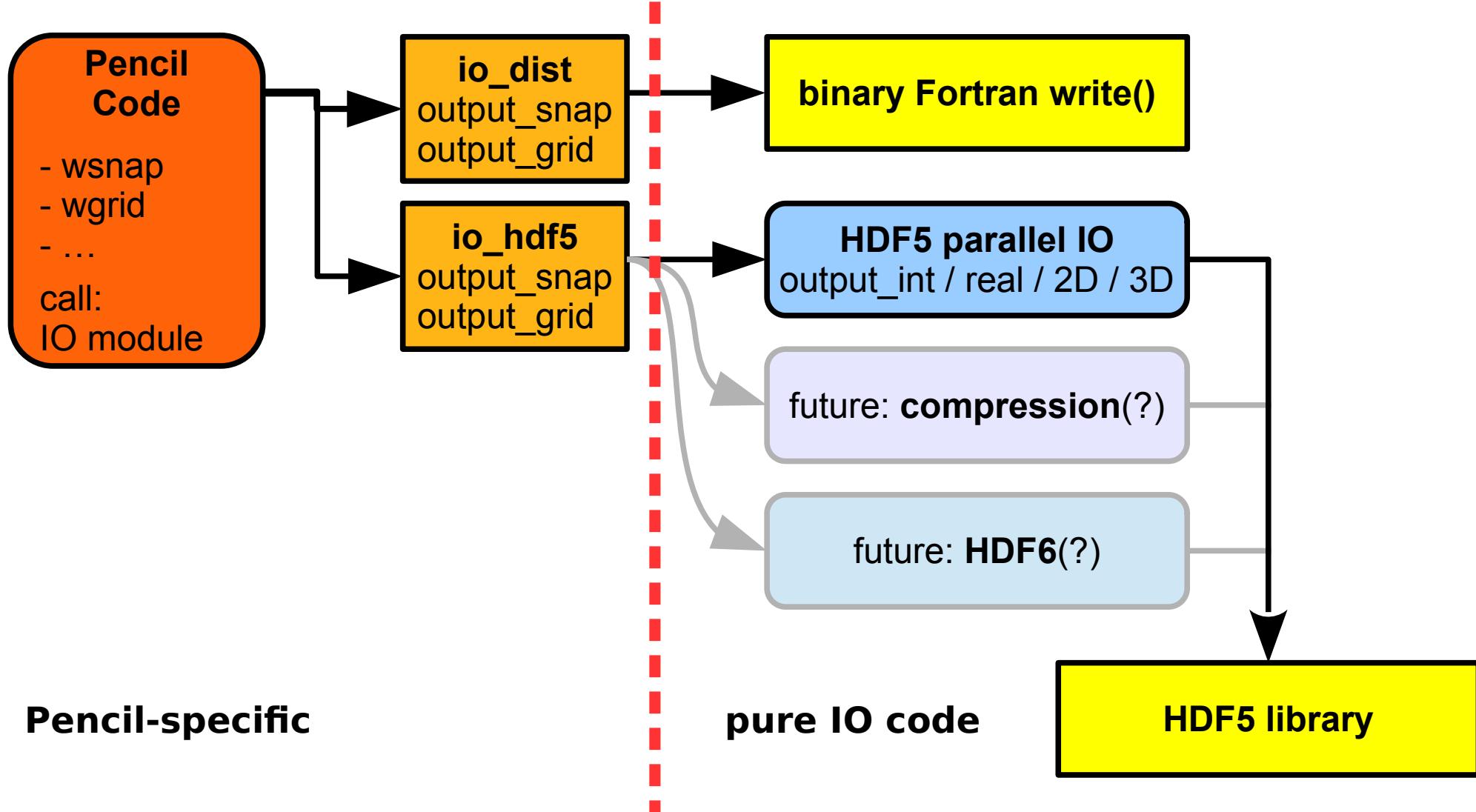
IO abstraction layer for HDF5

Separation of Pencil-specific code from IO code:



IO abstraction layer for HDF5

Separation of Pencil-specific code from IO code:



Large-scale data processing

Large-scale data processing

Why using collective snapshot files?

- IDL is slow in reading and combining distributed varfiles
- Using structures in IDL requires much more resources (memory, CPU)
- Inner ghost layers don't need to be stored (can save up to 50%)

Large-scale data processing

How to read HDF5 snapshots in IDL?

- Important IDL routines automatically switch to use HDF5, if available:

```
pc_read_var, pc_read_var_raw, pc_read_slice, pc_read_subvol_raw,  
pc_read_grid, pc_read_dim, pc_read_ts, pc_read_video,  
pc_read_pvar, pc_read_qvar, pc_read_pstalk,  
pc_get_quantity, pc_read_xyaver, pc_read_phiavg, ...
```

- New unified IDL reading routine “pc_read” for HDF5 snapshots:

```
Ax = pc_read ('ax', file='var.h5')      ;; open file 'var.h5' and read Ax  
Ay = pc_read ('ay', /trim)              ;; read Ay without ghost cells  
Az = pc_read ('az', processor=2)        ;; read data of processor 2  
ux = pc_read ('ux', start=[47,11,13], count=[16,8,4]) ;; read subvolume  
aa = pc_read ('aa')                   ;; read all three components of vector-field A  
xp = pc_read ('part/xp', file='pvar.h5')    ;; get x position of particles  
ID = pc_read ('stalker/ID', file='PSTALK0.h5')  ;; stalker particle IDs
```

Large-scale data processing

Read and write HDF5 files IDL

- Low-level routines for basic needs:

idl/read/hdf5/	h5_open_file	open HDF5 file (read, write, or truncate)
	h5_contains()	returns true if a given dataset exists
	h5_content()	returns all dataset names in a group
	h5_get_size()	returns the size of a dataset
	h5_get_type()	returns the IDL type of a dataset
	h5_read()	returns the content of a dataset
	h5_write	write a dataset
	h5_create_group	create a dataset group
	h5_close_file	close HDF5 file

=> If possible, use a high-level function like

„pc_read()“ or „pc_get_quantity()“ instead!

Large-scale data processing

Improved capabilities on secondary outputs:

- Averages are always consistent after restarts from earlier snapshots.
- Videofiles are always consistent, too.
- No need anymore for „pc_read_all_videofiles“ and similar.

Large-scale data processing

Improved capabilities on secondary outputs:

- Averages are always consistent after restarts from earlier snapshots.
- Videofiles are always consistent, too.
- No need anymore for „pc_read_all_videofiles“ and similar.

Convert old data:

- Conversion tool exists:
„pc_convert_hdf5“ converts a whole data directory to HDF5
(including varfiles, averages, slices, particles, etc.)

Large-scale data processing

How to view HDF5 snapshots directly?

- In a terminal:

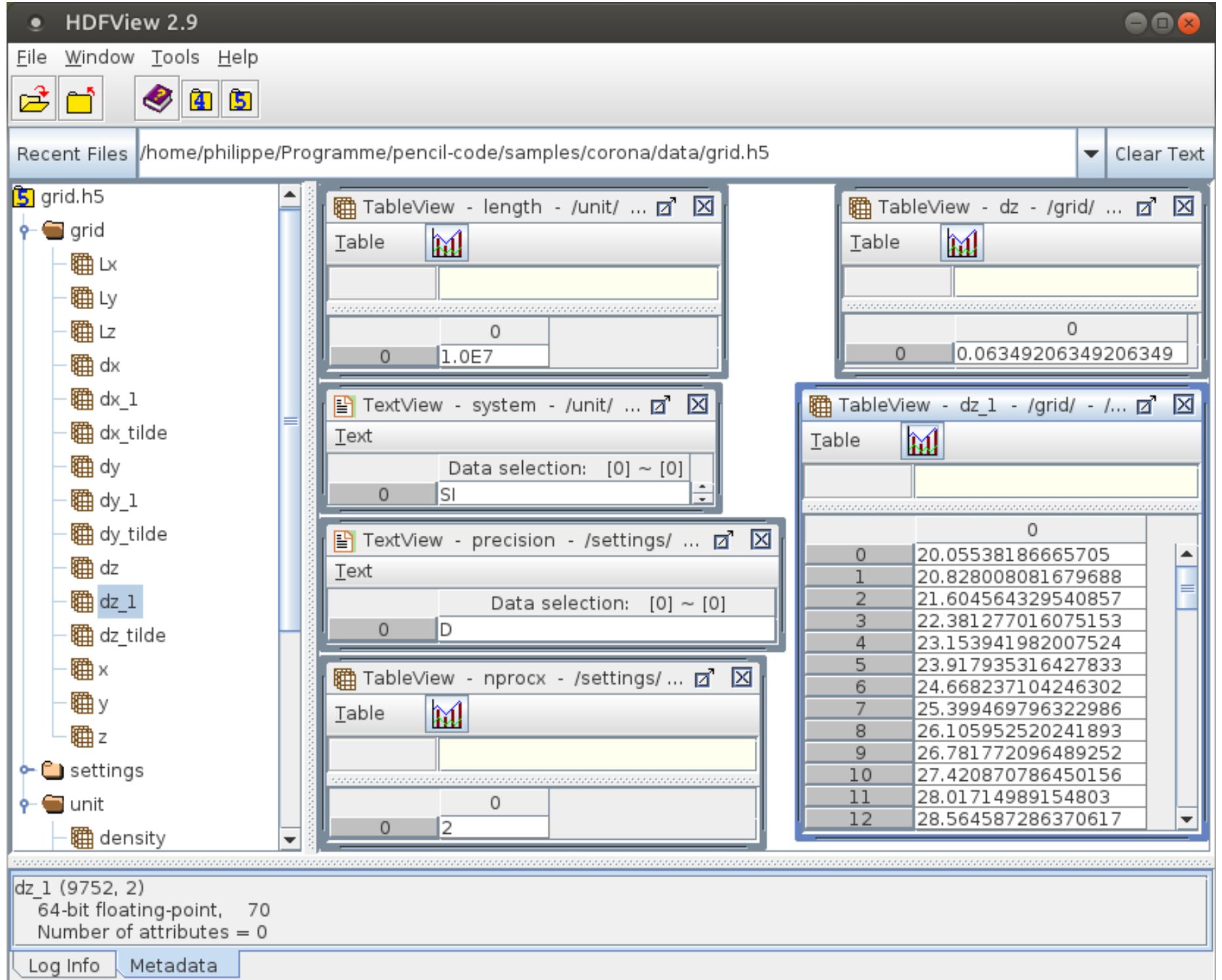
```
h5dump -H file.h5
```

- Graphical tool:

```
hdfview file.h5
```

- Other:

Matlab, Tecplot, ParaView, etc. directly load and display HDF5 data



HDFView 2.9



File Window Tools Help



Recent Files /home/philippe/Programme/pencil-code/samples/corona/data/allprocs/var.h5 ▾ Clear Text

• data
 ax
 ay
 az
 InTT
 Inrho
 ux
 uy
 uz
• grid
• settings
• time
• unit
 density

TableView - time - / - ...

Table

0	9.250866949169088E-4

TableView - density - /unit/ - ...

Table

0	1.0000000000000005E-8

TableView - Inrho - /data/ - /home/philippe/Programme/pencil-code/samples/corona/...

Table 0 70

	0	1	2	3	4	5	6	7
0	32.68667...	32.68667...	32.68667...	32.68667...	32.68667...	32.68667...	32.68667...	32.68667...
1	31.49853...	31.49853...	31.49852...	31.49853...	31.49852...	31.49852...	31.49852...	31.49852...
2	30.18198...	30.18196...	30.18194...	30.18200...	30.18187...	30.18190...	30.18193...	30.18195...
3	16.796295	16.796295	16.796295	16.796295	16.796295	16.796295	16.796295	16.796295

density (21971584, 2)
 64-bit floating-point, 1
 Number of attributes = 0

Log Info Metadata

How to install and use HDF5?

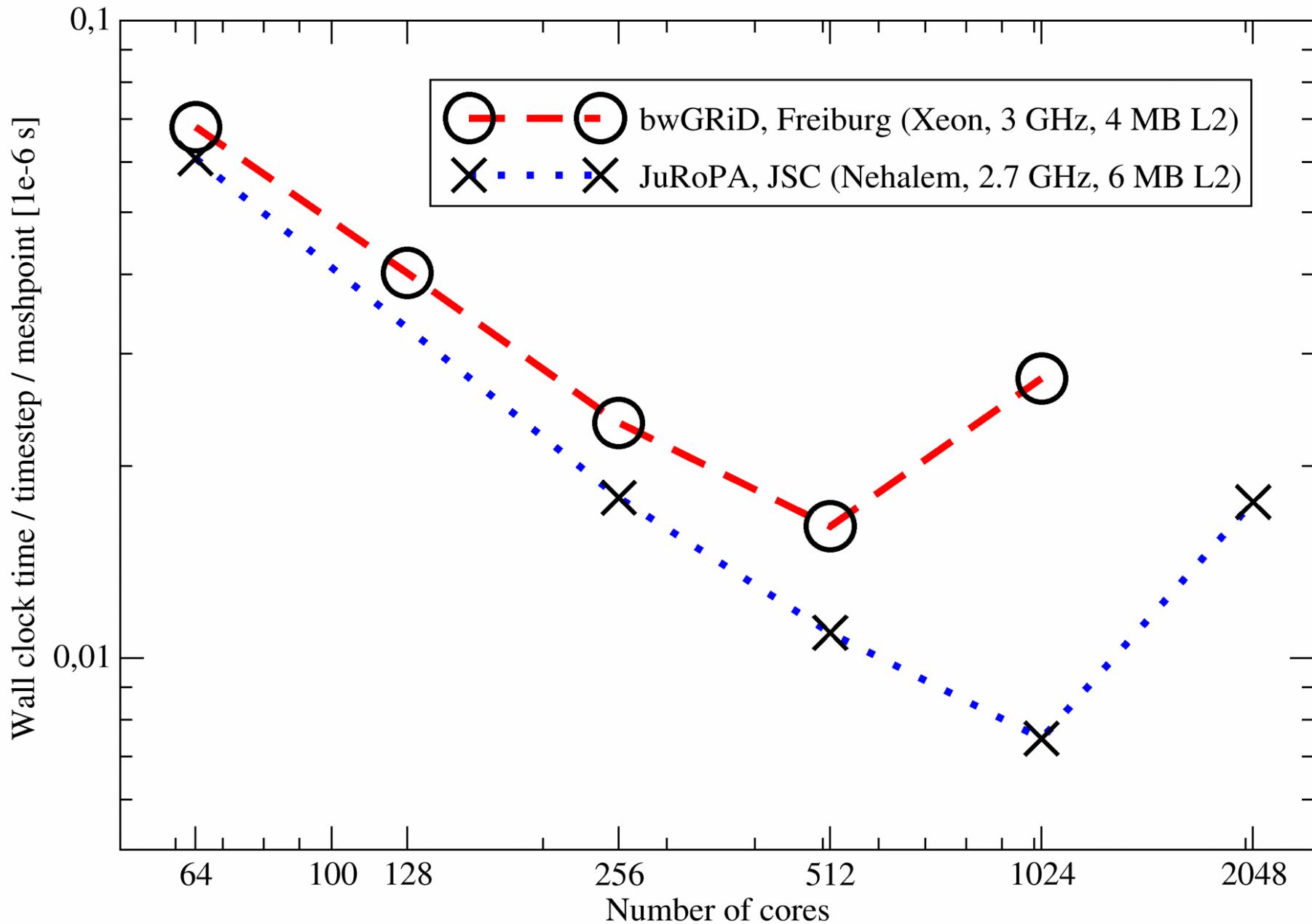
- On ubuntu systems, just install the packages:
“**libhdf5-openmpi-dev**” (mandatory)
“**h5tools**” (optional), “**hdfview**” (optional)
- On a supercomputer:
“**module load ...hdf5...**” (check „**module avail**“ for available packages)
- Change “src/Makefile.local” (see “samples/corona” for an example):
IO = io_hdf5
=> „pc_build“ will then automatically find the HDF5 compiler wrapper.

Thank you for supporting the HDF5 transition!

Scaling by a factor of 10'000

Scaling by a factor of 10'000

Scalability regarding number of cores for a fixed setup on different computing clusters



Scaling by a factor of 10'000

Major steps:

~ 1 CPU core: Code need to run reliably first... (create auto-test data!)

Scaling by a factor of 10'000

Major steps:

- ~ 1 CPU core: Code need to run reliably first... (create auto-test data!)
- ~ 10 CPU cores: Parallelization in 1-2 coordinate directions

Scaling by a factor of 10'000

Major steps:

- ~ 1 CPU core: Code need to run reliably first... (create auto-test data!)
- ~ 10 CPU cores: Parallelization in 1-2 coordinate directions
- ~ 100 CPU cores: Parallelization with >4 cores per coordinate direction

Scaling by a factor of 10'000

Major steps:

- ~ 1 CPU core: Code need to run reliably first... (create auto-test data!)
- ~ 10 CPU cores: Parallelization in 1-2 coordinate directions
- ~ 100 CPU cores: Parallelization with >4 cores per coordinate direction
- ~ 1'000 CPU cores: Parallel file Input/Output (Lustre, GPFS, BeeGFS)
 Optimize all global communication
 Avoid global arrays

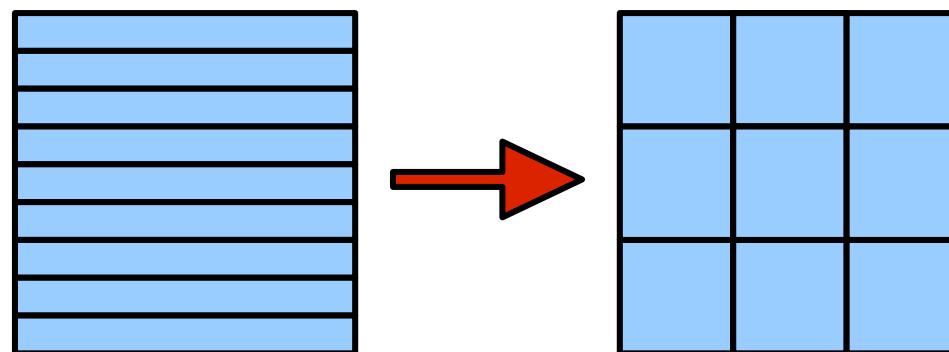
Scaling by a factor of 10'000

Major steps:

- ~ 1 CPU core: Code need to run reliably first... (create auto-test data!)
- ~ 10 CPU cores: Parallelization in 1-2 coordinate directions
- ~ 100 CPU cores: Parallelization with >4 cores per coordinate direction
- ~ 1'000 CPU cores: Parallel file Input/Output (Lustre, GPFS, BeeGFS)
 Optimize all global communication
 Avoid global arrays
- ~ 10'000 CPU cores: Find weak parts (bounday condition?) and optimize them
 Divide into sub-sections
 Avoid global communication

Scaling by a factor of 10'000

Cubic subdomains:



- better volume/surface ratio
- less communication for subdomain boundaries

But:

- fill up vector unit (SIMD, SSE, Altivec, ...) with fastest dimension

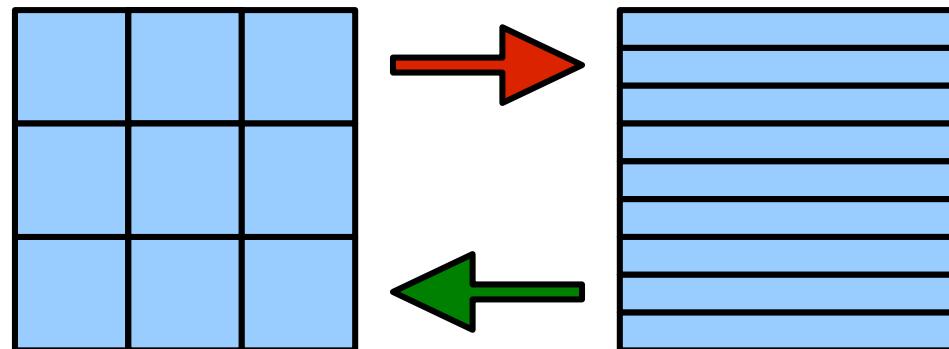
=> Due to trade-off, **intermediate subdomains may perform best!**

Scaling by a factor of 10'000

Massive parallel Fourier-transformation:

- Boundary condition requires FFT to be executed each timestep, overall performance is limited by the weakest part

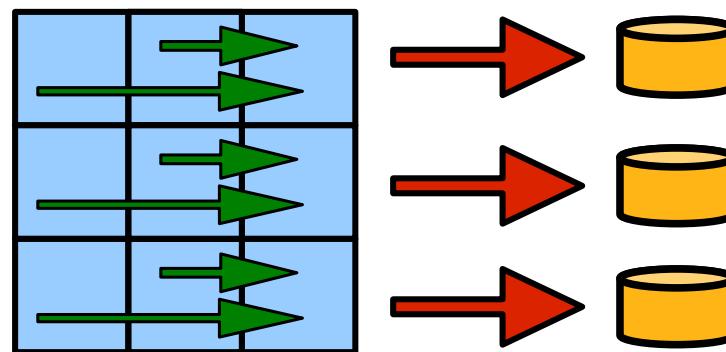
=> remap domain decomposition to let processors compute in parallel



Scaling by a factor of 10'000

Massive parallel file input/output:

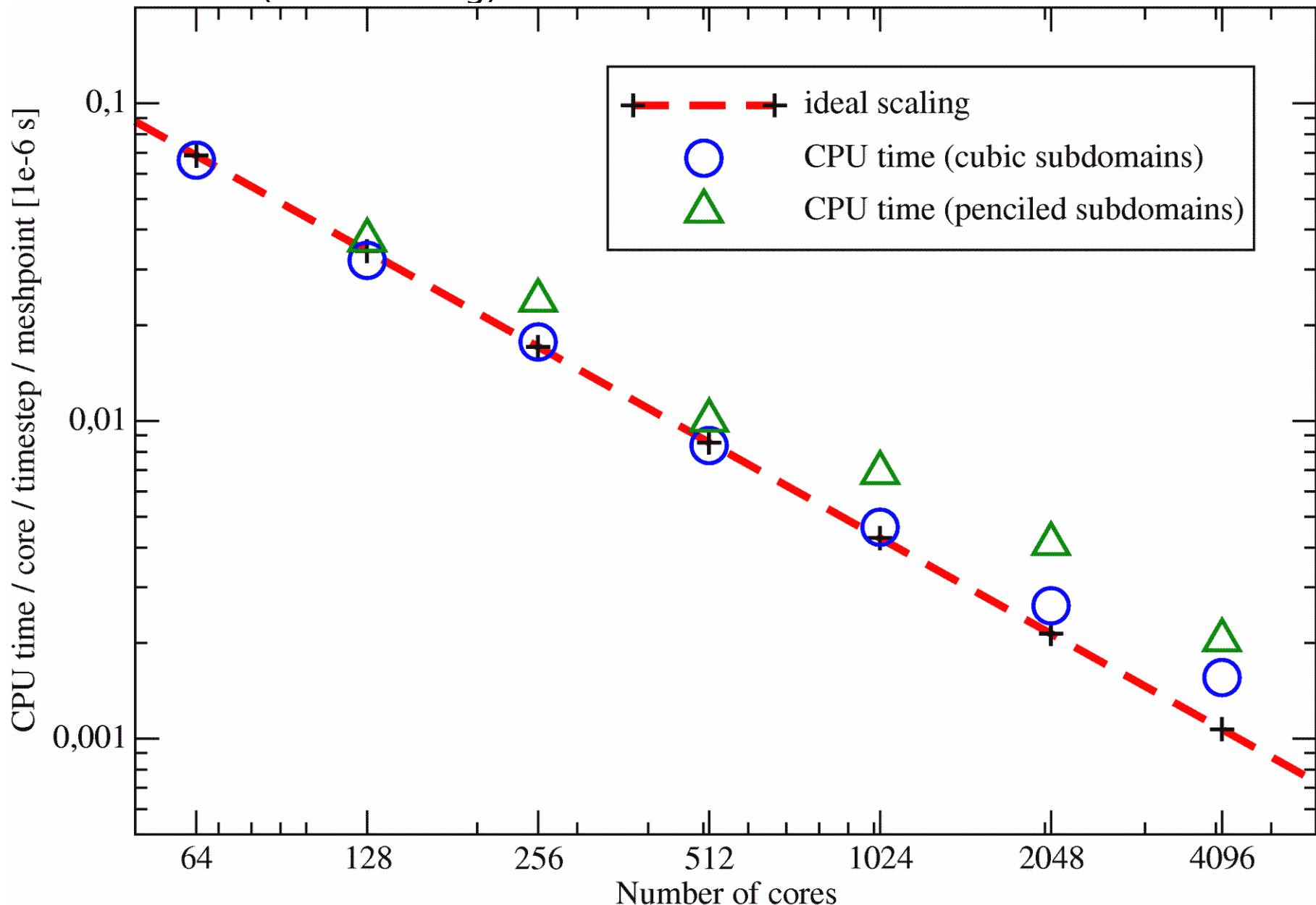
- Writing from each processor may overload the filesystem
- => communicate sets of subdomains to specialised I/O-nodes



=> self-organized writing of combined subsets,
alternatively use “MPI-IO” or “parallel HDF-5“

Scaling by a factor of 10'000

Benchmarks: (hard scaling)



Scaling by a factor of 10'000

Benchmarks: (hard scaling, “before & after” optimization work)

