Introduction to PDI & hands-on

Yushan Wang
Benoît Martin
Julien Bigot

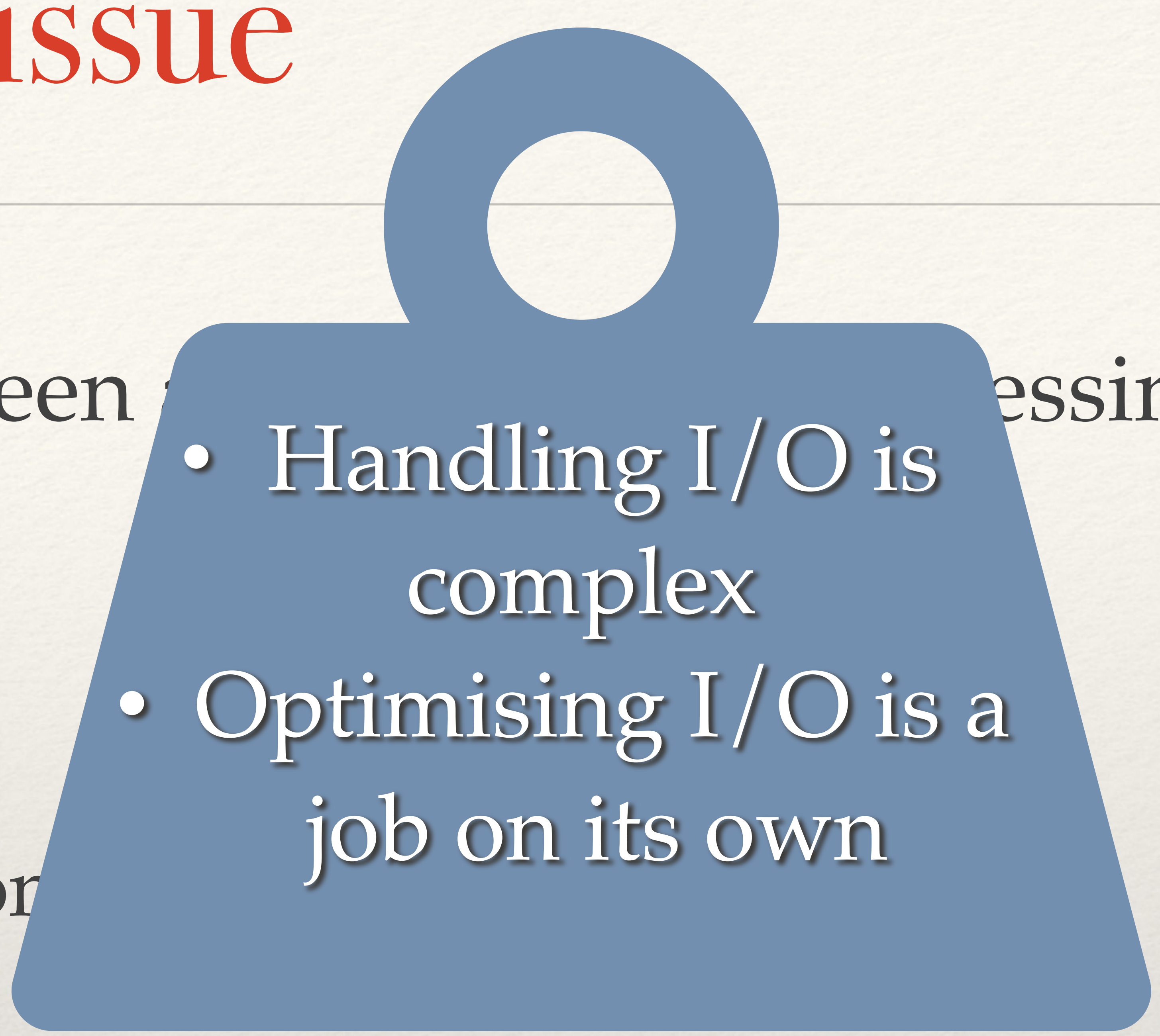
MAISON DE LA SIMULATION

Initial Motivation: the I/O issue

- I/O refers to the communication between an information processing system and the outside world.
- **Ease of use**: user-friendly interfaces
- **Performance**: high speed, latency reduction
- **Portability**: cross-platform compatibility, ease of migration
- **Large language support**: diverse ecosystem, broad adoption
- **Parallelisation-independent file format**: concurrent processing, scalability
- **Portable file format**: interoperability, future-proofing
- **Leverage underlying hardware**: hardware optimisation, resource utilisation

Initial Motivation: the I/O issue

- I/O refers to the communication between … essing system and the outside world.
- **Ease of use**: user-friendly interfaces
- **Performance**: high speed, latency reduction
- **Portability**: cross-platform compatibility, ease of migration
- **Large language support**: diverse ecosystem, broad adoption
- **Parallelisation-independent file format**: concurrent processing, scalability
- **Portable file format**: interoperability, future-proofing
- **Leverage underlying hardware**: hardware optimisation, resource utilisation



The library ecosystem

SIONlib

FTI

MPI I/O

NetCDF

pNetCDF

Damaris

HDF5

POSIX I/O

Adios

Hercule

XIOS

FlowVR

…

…

…

How to choosing a library

- Choosing the best library: a problem on its own
- The best library depends on :
  - The code specifics, the type of I/O operations, computational intensive? I/O bound? Read/write-heavy? Sequential? Random? …
  - Parallelism level, replicated / distributed data, I/O frequency, …
  - Data lifecycle, Initialisation data reading, result writing (small or large), checkpointing writing, coupling related I/O
  - The specific execution, small/large case, debug/production
  - …

How to choosing a library

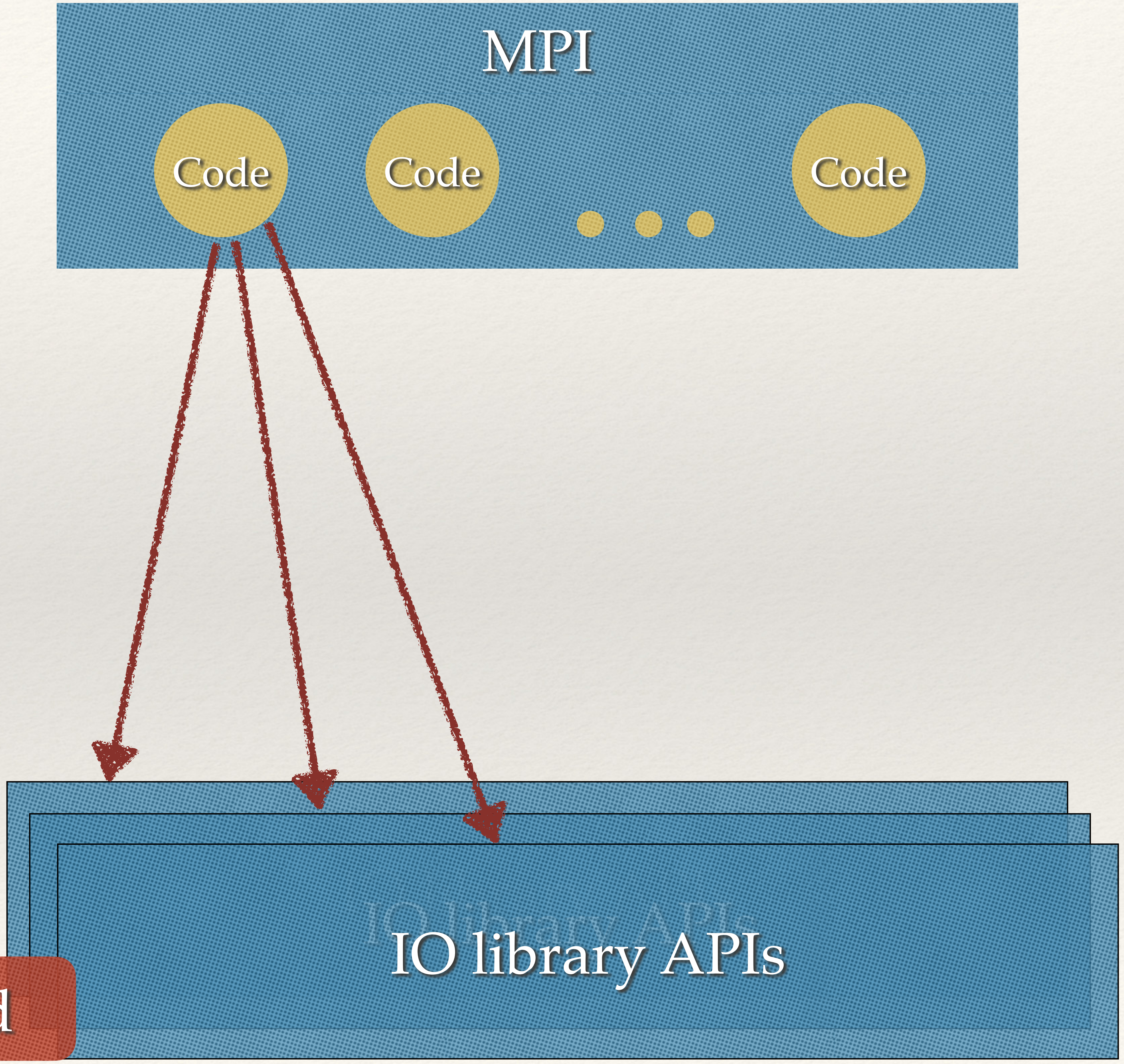
- Choosing the best library: a problem on its own
- The best library depends on :
  - The code specifics, the type of I/O operations, computational intensive? I/O bound? Read/write-heavy? Sequential? Random? …
  - Parallelism level, replicated / distributed data, I/O frequency, …
  - Data lifecycle, Initialisation data reading, result writing (small or large), checkpointing writing, coupling related I/O
  - The specific execution, small/large case, debug/production
  - …

No one-size-fits-all library

Many codes end up with an I/O abstraction layer, BUT …

Why PDI?

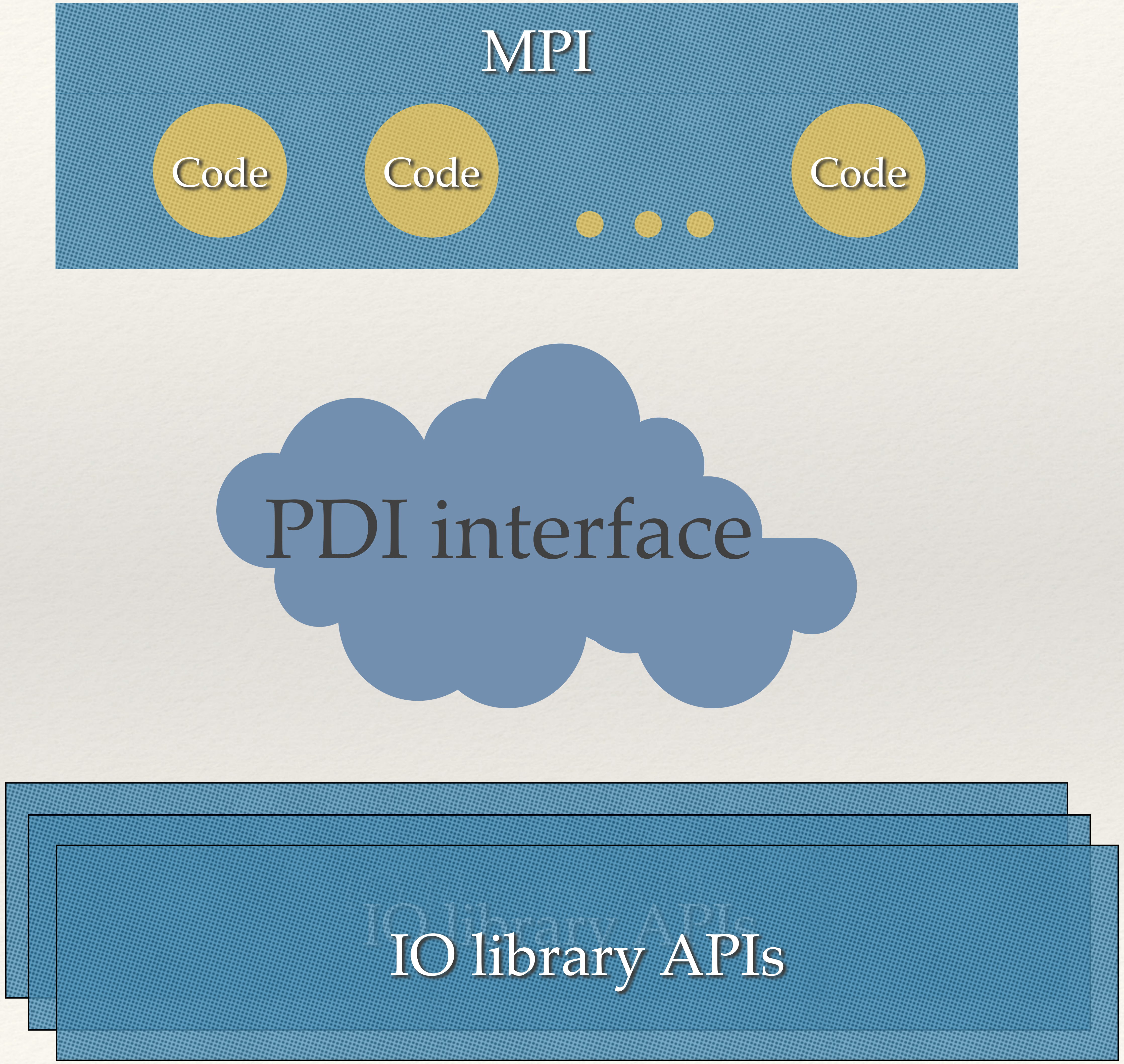
```
Void MyApp::write( some_grids,   some_params,
                   some_strings, some_fields, …)
{
    if (IO_lib == HDF5)
    {
        h5_write(some_grids,
                 some_strings,
                 some_fields);
    }
    else if (IO_lib == FTI)
    {
        fti_checkpoint(some_grid, some_fields);
    }
    else if (IO_lib == VTK)
    {
        write_vtk(some_params, some_strings,
                  some_fields);
    }
    ...
}
```



Different library, different API, different standard

Why PDI?

```
Void MyApp::write( some_grids,   some_params,
                   some_strings, some_fields, …)
{
    magic_IO(some_grids,   some_params,
             some_strings, some_fields, …);
}
```

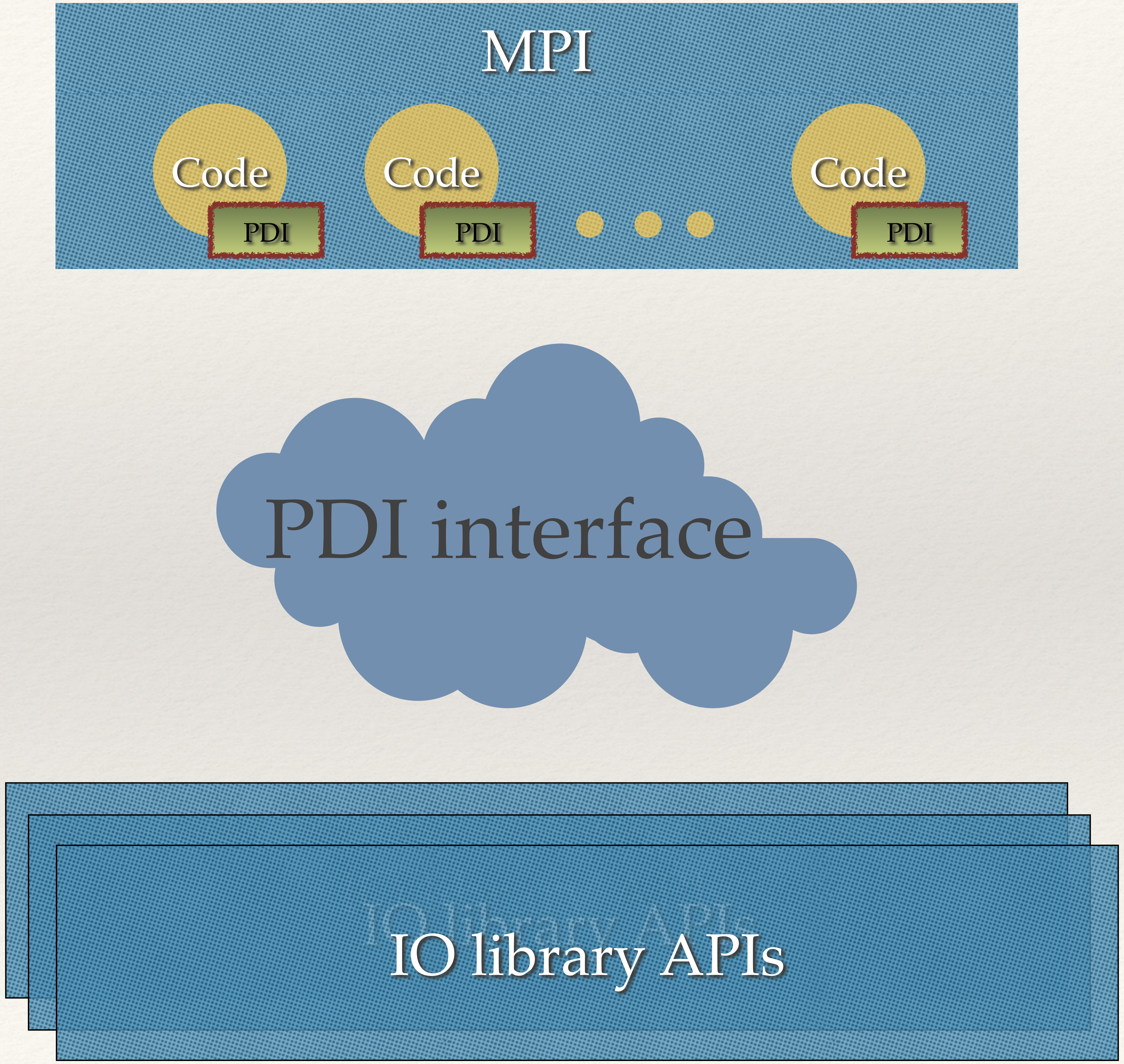


What is PDI?

- PDI has a pure declarative API

```
double* my_data = malloc(N * sizeof(double));

while (temporal_loop)
{
    update_data(my_data);

    PDI_share("main_data", my_data, PDI_OUT);

    other_compute(…);

    other_reading(my_data);

    PDI_reclaim("main_data");

}
```

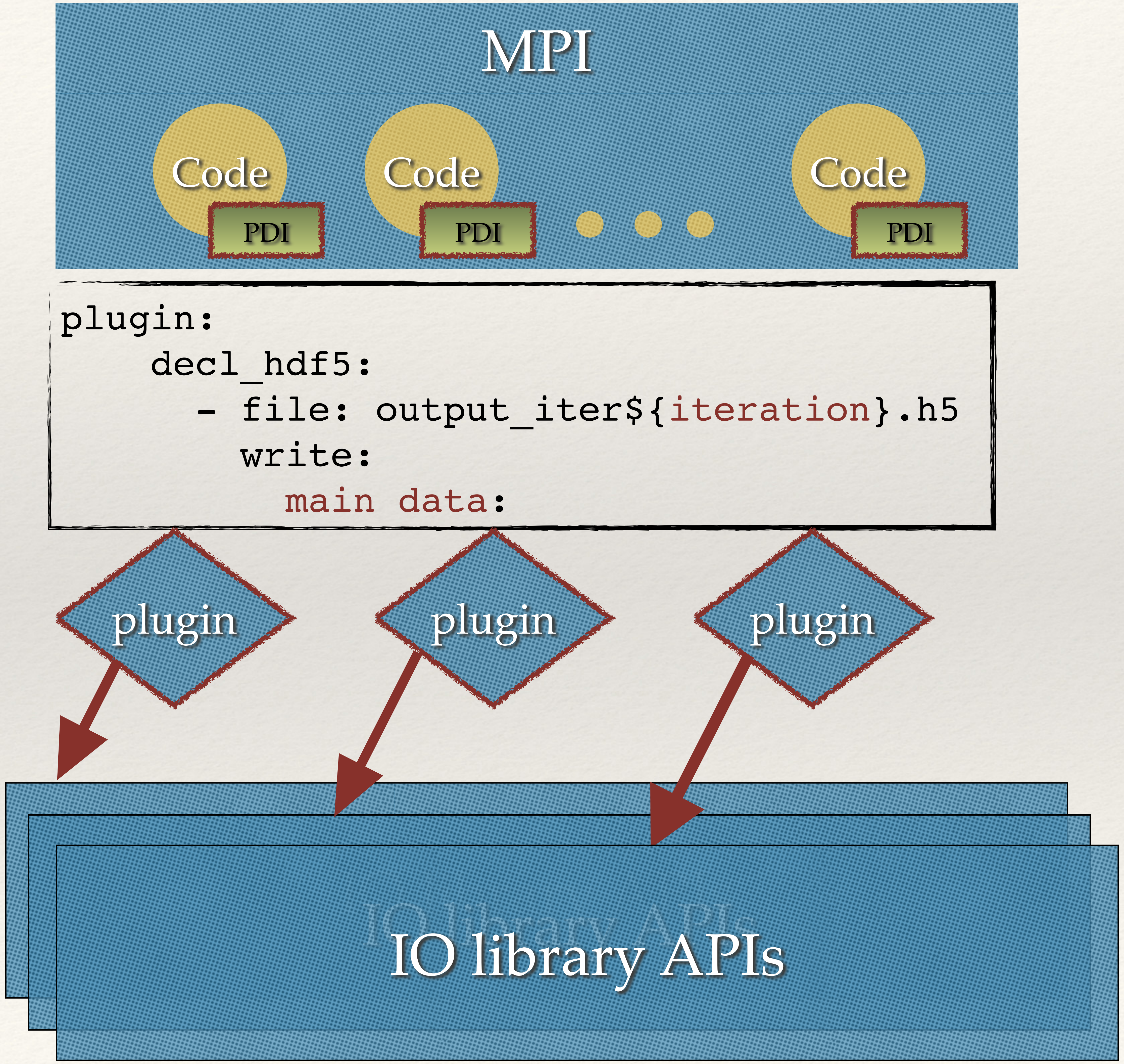
Buffer is available to PDI plugins



What is PDI?

```yaml
metadata:
    iteration: int
    domain_size: {type: int, size: 2}
data:
    main_data:
        type: double,
        size: ['$domain_size[0]', '$domain_size[1]']
```

- PDI YAML config file:
    - How data is represented
    - What to do with data
    - Modify yaml configuration without recompiling the application



How to use PDI

In code

- Enable PDI environment
- Annotate buffers availability (share / reclaim/ …)
- Compile and … Done!

In yaml

- Specify metadata and data representation
- Use pre-made plugins or write your own code to choose I/O libraries, describe behaviour

How to use PDI

In code

- Enable PDI environment
- Annotate buffers availability (share / reclaim/…)
- Compile and … Done!

In yaml

- Specify metadata and data representation
- Use pre-made **plugins** or write your own code to choose I/O libraries, describe behaviour

HDF5, NetCDF, FlowVR, FTI, Pycall, Deisa, User-code, Damaris, Melissa, JSON…

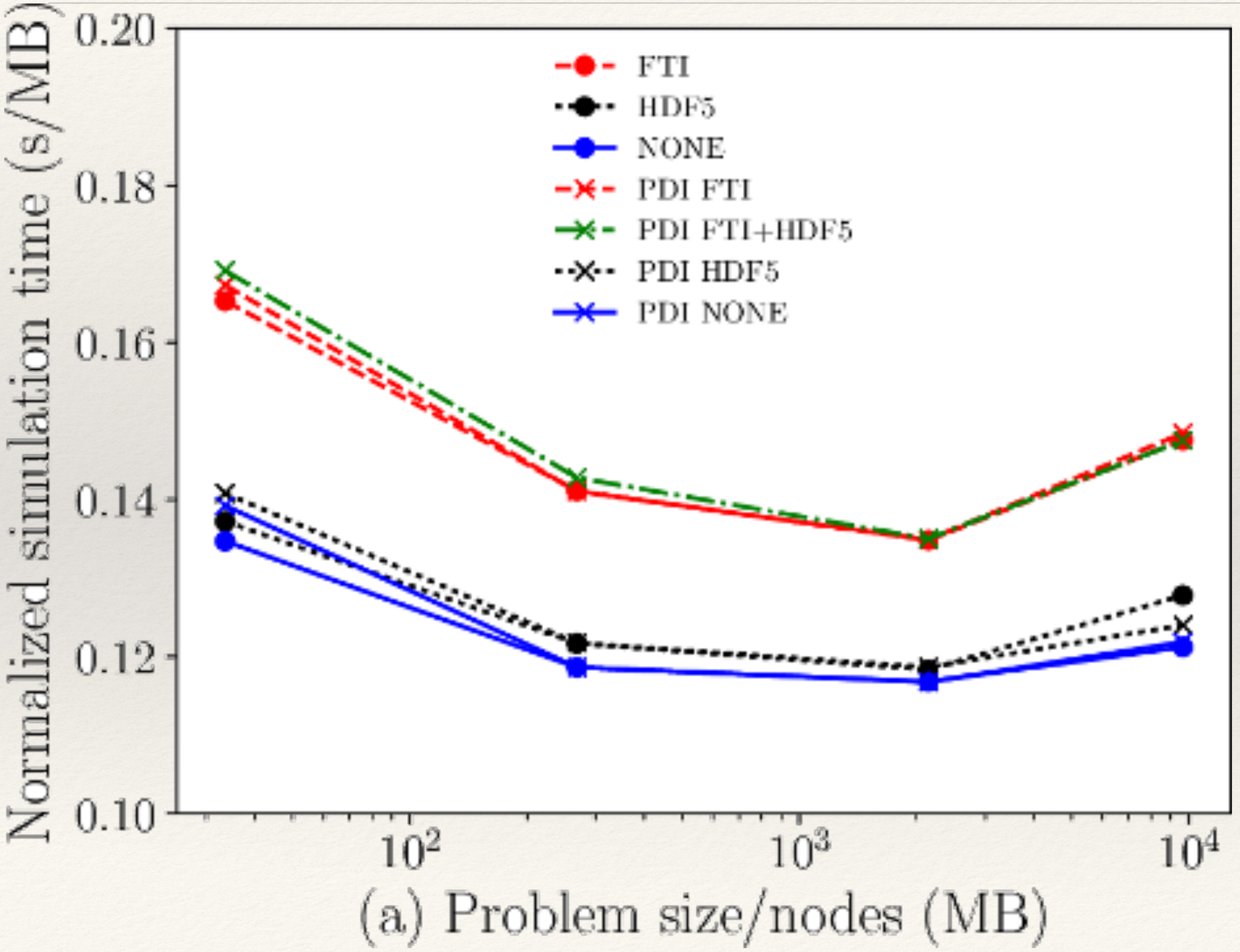
PDI's API : 12 functions

Full description at https://pdi.dev/master/modules.html	
PDI_init	Initialize PDI
PDI_finalize	Finalize PDI
PDI_share	Share data with PDI
PDI_relaim	Reclaim ownership of a data buffer shared with PDI
PDI_access	Request for PDI to access a data buffer
PDI_release	Release ownership of a data shared with PDI
PDI_event	Trigger a PDI event
PDI_expose	PDI_share + PDI_reclaim
PDI_multi_expose	PDI_share(s) + PDI_event+PDI_reclaim(s)

PDI in practice

- PDI is open-source (BSD 3-clause license)
- Regular releases since first commit in 2014
- Packages available for Debian, Fedora, Ubuntu, Spack
- Documentation available @ `https://pdi.dev/1.6/`
- Heavily tested & validated (more than 700 tests, on 14 platforms)
- Integration in production codes (Gysela, Parflow, ESIAS, …)
- Slack channel for user support
- **PDI team is recruiting! Please spread the news!**

PDI performance evaluation



- 4 versions of Gysela
  - No checkpointing
  - HDF5 checkpointing
  - FTI
  - PDI (none, HDF5, FTI, FTI+HDF5)

PDI hands-on

- PDI configuration and installation
- Hands-on setup
  - 2D heat equation
  - Written in C
- Go through several PDI plugins

PDI installation

- Recommend to use pre-compiled binary packages, or Spack recipe.
  - PDI is already installed on Ruche: `. /gpfs/workdir/shared/pdi-deisa/setup-env.sh`
- Download PDI source distribution
  - https://gitlab.maisondelasimulation.fr/pdidev/pdi/-/releases
- Most Dependencies are embedded, required external dependencies are:
  - CMake 3.10+
  - C 99 & C++17 compiler
  - Fortran 03 compiler
  - Python 3.6+
  - MPI

PDI installation

```
wget https://gitlab.maisondelasimulation.fr/pdidev/pdi/-/archive/
1.6.0/pdi-1.6.0.tar.bz2

tar -xjf pdi-1.6.0.tar.bz2

mkdir pdi-1.6.0/build

cd pdi-1.6.0/build

cmake -DCMAKE_INSTALL_PREFIX="${HOME}/pdi_install/" ..

make install
```

PDI installation

BUILD_FORTRAN	ON
BUILD_HDF5_PARALLEL	ON
BUILD_MPI_PLUGIN	ON
BUILD_NETCDF_PARALLEL	ON
BUILD_TRACE_PLUGIN	ON
BUILD_USER_CODE_PLUGIN	ON
BUILD_PYTHON	OFF
BUILD_FTI_PLUGIN	OFF

USE_Doxygen	AUTO
USE_FTI	AUTO
USE_HDF5	AUTO
USE_paraconf	AUTO
USE_pybind11	AUTO
USE_yaml	AUTO
USE_Zpp	EMBEDDED

Compiling with PDI

- C/C++ compilation using CMake

```
find_package(PDI REQUIRED COMPONENTS C)


add_executable(exec_file source_files.c)

target_link_libraries(exec_file PDI::PDI_C)
```

Your turn

- Connection to ruche

```
ssh ruche
```

- Activate PDI environment
  - Using pre-installed PDI on ruche

    ```
    . /gpfs/workdir/shared/pdi-deisa/setup-env.sh
    ```

  - Or you can install your own

```
wget https://gitlab.maisondelasimulation.fr/pdidev/pdi/-/archive/1.6.0/pdi-1.6.0.tar.bz2
tar -xjf pdi-1.6.0.tar.bz2
mkdir pdi-1.6.0/build && cd pdi-1.6.0/build
cmake -DCMAKE_INSTALL_PREFIX="${HOME}/pdi_install/" ..
make install
```

PDI Hands-on

- Get the sources from GitHub:

```
git clone https://github.com/pdidev/tutorial.git
cd tutorial
```

- Compile:

```
cmake .
```

```
make ex1
```

- Execution:

```
srun --nodes=1 --cpus-per-task=20 --reservation=bigotj_136 --pty /bin/bash
```

```
mpirun -np 4 ./ex1
```

PDI hands-on

- `https://pdi.dev/master/Hands_on.html`

- ex1: no PDI at all
    - Marked with `//***`
    - load / unload configuration file ()
    - add PDI init and finalise
    - read data from yml config file

PDI hands-on

- `https://pdi.dev/master/Hands_on.html`
- ex2: expose some data to PDI
  - Use **`PDI_share`** and **`PDI_reclaim`** to expose data to PDI :
    - Domain configuration information : global size, local size, etc.
    - iteration number
    - current data
  - Before / during / after the iteration loop
  - Observe the PDI trace using **`trace_plugin`**

PDI hands-on

- `https://pdi.dev/master/Hands_on.html`
- ex3: first try with HDF5 plugin
  - Describe the metadata in ex3.yml :
    - Domain configuration information : global size, local size, etc.
    - iteration number
  - Let PDI know what to do with the **`decl_hdf5`** plugin
  - Use one MPI process !!!
  - Look at the output file

PDI hands-on

- `https://pdi.dev/master/Hands_on.html`
- ex4: write data with HDF5 plugin
  - Describe the data in ex3.yml :
    - Temperature data on the current iteration
    - Use $-expression to define the size
  - Write one file per process
  - Look at the output file

PDI hands-on

- `https://pdi.dev/master/Hands_on.html`
- ex5-1: use PDI_event
  - To prevent redundant open/close of the output file
- ex5-2: add a **`when`** clause
  - Write for 2 iterations only
- ex5-3: write data in different dataset

PDI hands-on

- `https://pdi.dev/master/Hands_on.html`
- ex6: a tidier way of coding
  - **`PDI_expose = PDI_share + PDI_reclaim`**
  - **`PDI_multi_expose = PDI_share + PDI_event + PDI_reclaim`**
- ex7: get ride of the ghost layer in output
  - Define **`datasets`** inside of the **`decl_hdf5`** plugin
  - Use **`memory_selection`** to skip the ghost cells

PDI hands-on

- `https://pdi.dev/master/Hands_on.html`
- ex8: write 2D array as a slice of 3D dataset including a dimension on time
  - Add one dimension to **`datasets`** to represent the iteration
  - Use **`dataset_selection`** to specify where to write in the dataset

PDI hands-on

- `https://pdi.dev/master/Hands_on.html`
- ex9: let's bring in the parallelism
  - Load **`mpi`** plugin
  - Set the communicator for **`decl_hdf5`** plugin
  - Modify the size from local to global in **`datasets`** (use `psize`)
  - Modify the **`dataset_selection`** to ensure no overlap (use `pcoord`)

PDI hands-on

- `https://pdi.dev/master/Hands_on.html`
- ex10: post-processing in Python (need Python environment, we will do tomorrow)
  - Load **`pycall`** plugin
  - Enable the plugin upon the **`loop`** event
  - Fill the input arguments
  - Implement the transformation and expose data to PDI
  - Write the transformed data to hdf5

PDI hands-on

- `https://pdi.dev/master/Hands_on.html`
- ex11: user_code plugin
  - The user-code plugin enables one to call a user-defined function when a specified event occur or certain data becomes available.
  - `set_target_properties(exe PROPERTIES ENABLE_EXPORTS TRUE)`
  - At event “initialization”, open a file for recording the total mass
  - At event “finalization”, close the file
  - When “main_field” is shared to PDI, compute the total mass and write to file

PDI hands-on

- `https://pdi.dev/master/Hands_on.html`
- ex12: set_value plugin
- Action triggered upon: **`on_init, on_finalize, on_data, on_event`**
- A random integer is exposed to PDI at each iteration
- Start the output once the integer passes 50
- and stop output when it's below 25

`share`	share new allocated data with given values
`release`	release shared data
`expose`	expose new allocated data with given values
`set`	set given values to the already shared data
`event`	call an event