IO, restart, in situ or coupling: PDI, a single interface to decouple data handling from computation in numerical simulation

October 4th, 2023 – SISMA

Julien Bigot[2], François-Xavier Morvan[2], Pierre-Antoine Raclius[2], Yushan Wang[2]

Thanks to:
Leonardo Bautista Gomez[1], Sebastian Friedemann[6], Amal Gueroudji[2], Virginie Grandgirard[2], Karim Hasnaoui[3], Francesco Iannone[4], Kai Keller[1], Guillaume Latu[2], Yacine Ould Rouis[3], Bruno Raffin[6], Karol Sierocinski[7], Kacper Sinkiewicz[7], Benedikt Steinbusch[5], Christian Witzler[5]

[1] BSC, [2] CEA, [3] CNRS, [4] ENEA, [5] FZJ, [6] Inria, [7] PSNC

Initial Motivation: the I/O Issue

- We want it easy to use
- We want it fast
- We want a portable library
- We want large language support
- We want parallelization independent file format
- We want a portable file format
- We want to leverage the underlying hardware
- We want…

Initial Motivation: the I/O Issue

- We want it easy to use
- We war
- We war
- We want large language support
- We want parallelization independent file format
- We want a portable file format
- We want to leverage the underlying hardware
- We want…

Handling I/O is complex
Optimizing I/O is a job on its own

Initial Motivation: the I/O Issue

PDI

- We want it easy to use
- We war
- We war
- We want large language support
- W
- We want a portable file format
- We want to … e
- We want…

Handling I/O is complex
Optimizing I/O is a job on its own

Complex but common problem,
A community with dedicated expert

Let's use **libraries**

The I/O Issue: the library ecosystem

PDI

- FTI
- MPI I/O
- pNetCDF
- SIONlib
- FlowVR
- Adios
- Hercule
- NetCDF
- HDF5
- XIOS
- IME
- POSIX
- SCR
- pHDF5
- Damaris
- PaDaWAn

The I/O issue: 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
  - Parallelism level, replicated / distributed data, I/O frequency, …
  - Initialization data reading, result writing (small or large), checkpoint writing, coupling related I/O
- The specific execution
  - Small case / large case, debug / production, …
- The specific hardware available
  - I/O bandwidth, intermediate storage, …

The I/O issue: Choosing a library

Choosing the best library: a problem on its own

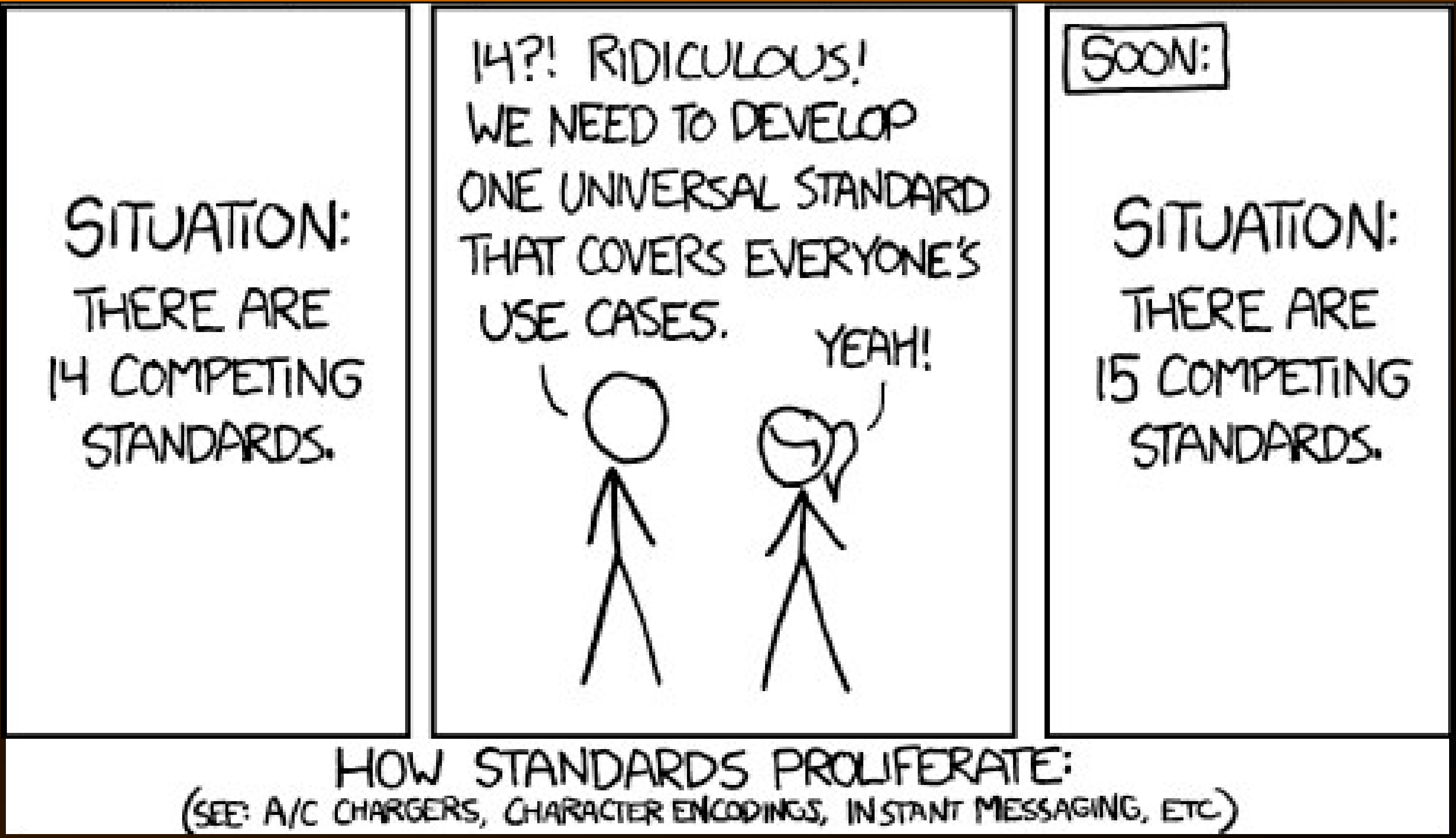
The best library depends on…

- The code specifics, the type of I/O
  - Parallelism le… …equency, …
  - Initialization data reading, result writing (small or large), checkpoint writing, coupling related I/O
- The specific execution
- The specific hardware available
  - I/O bandwidth, intermediate storage, …

Not one-size fits all library

Many codes end-up with an IO abstraction layer

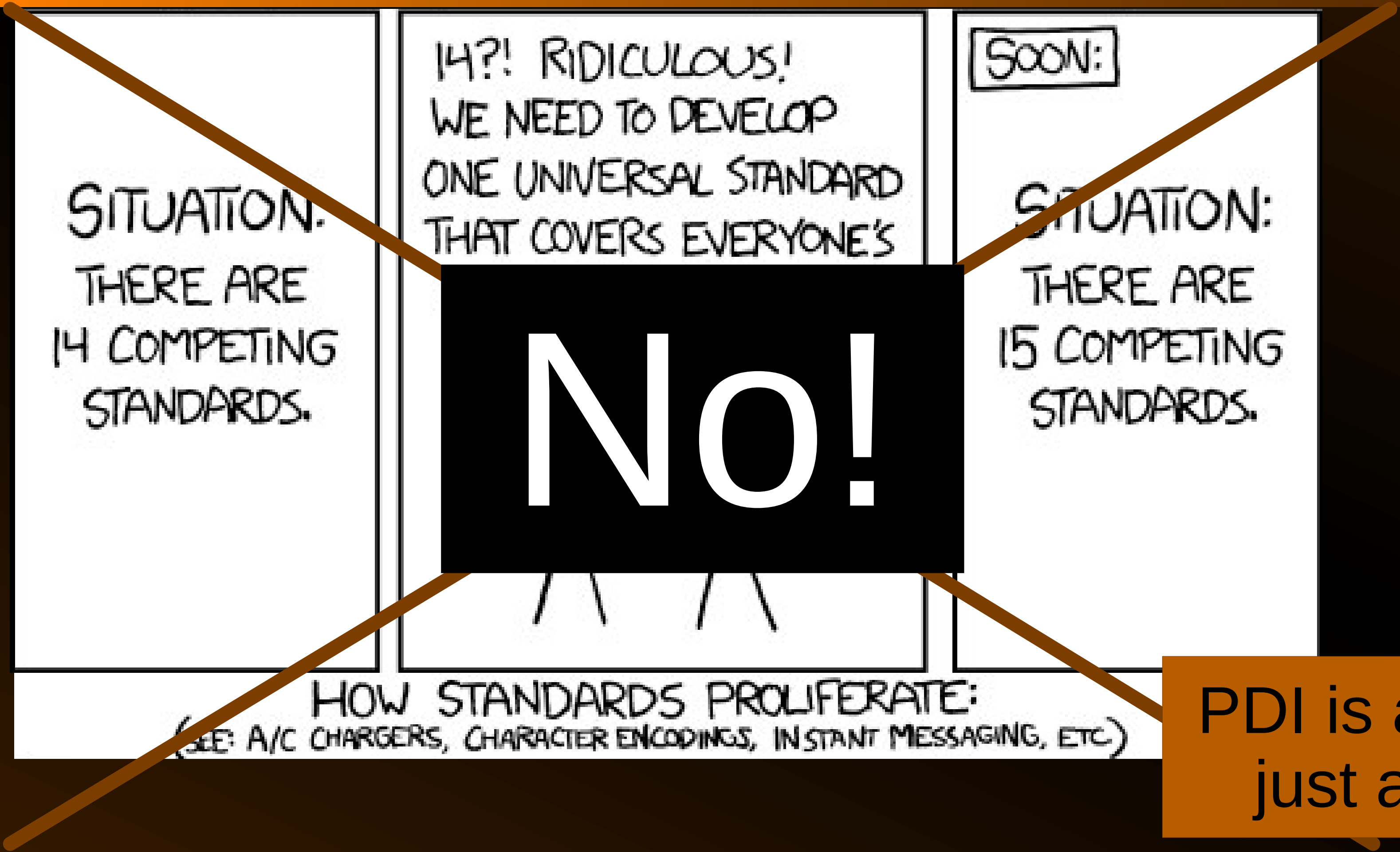
Introducing PDI



© XKCD
https://xkcd.com/927/

Introducing PDI

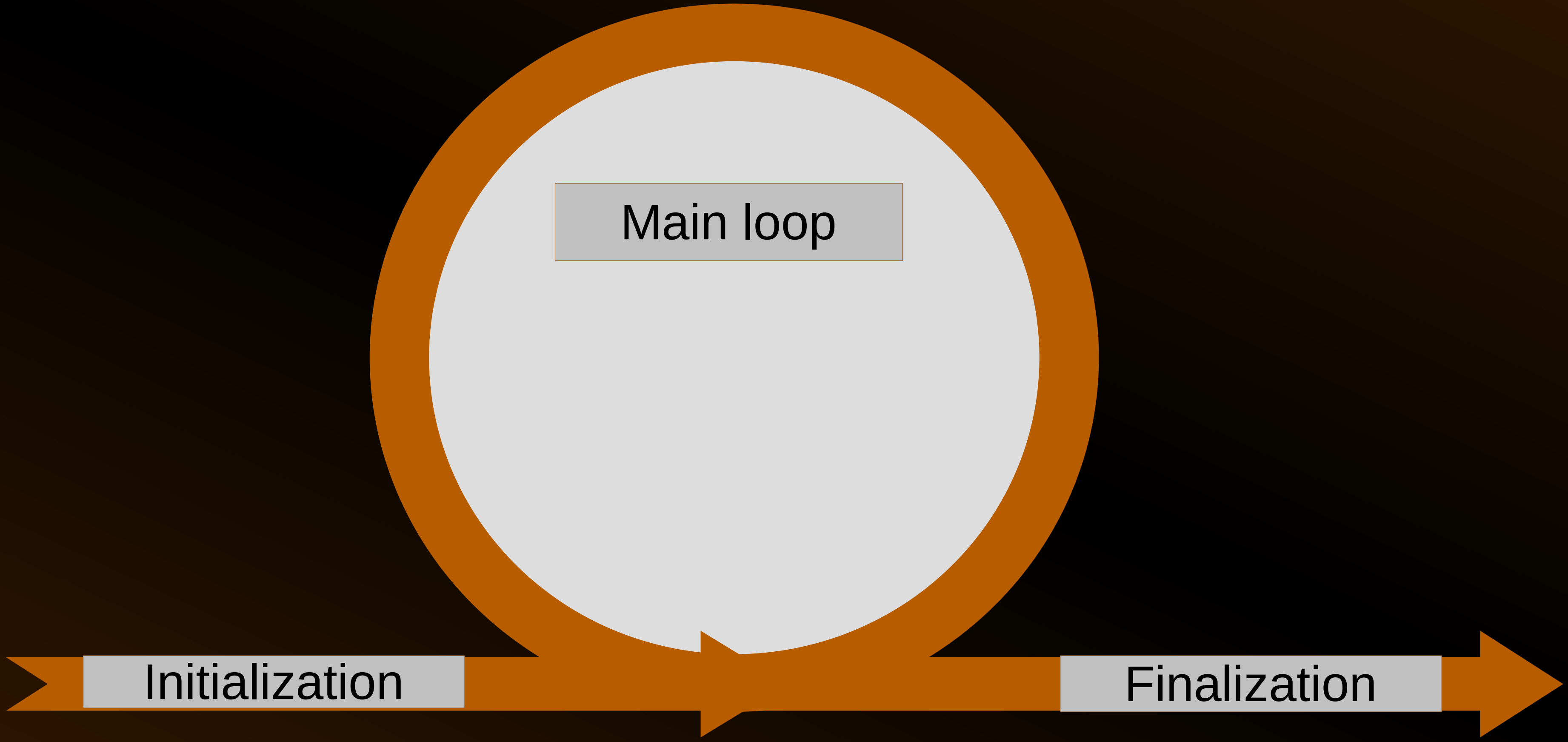
PDI



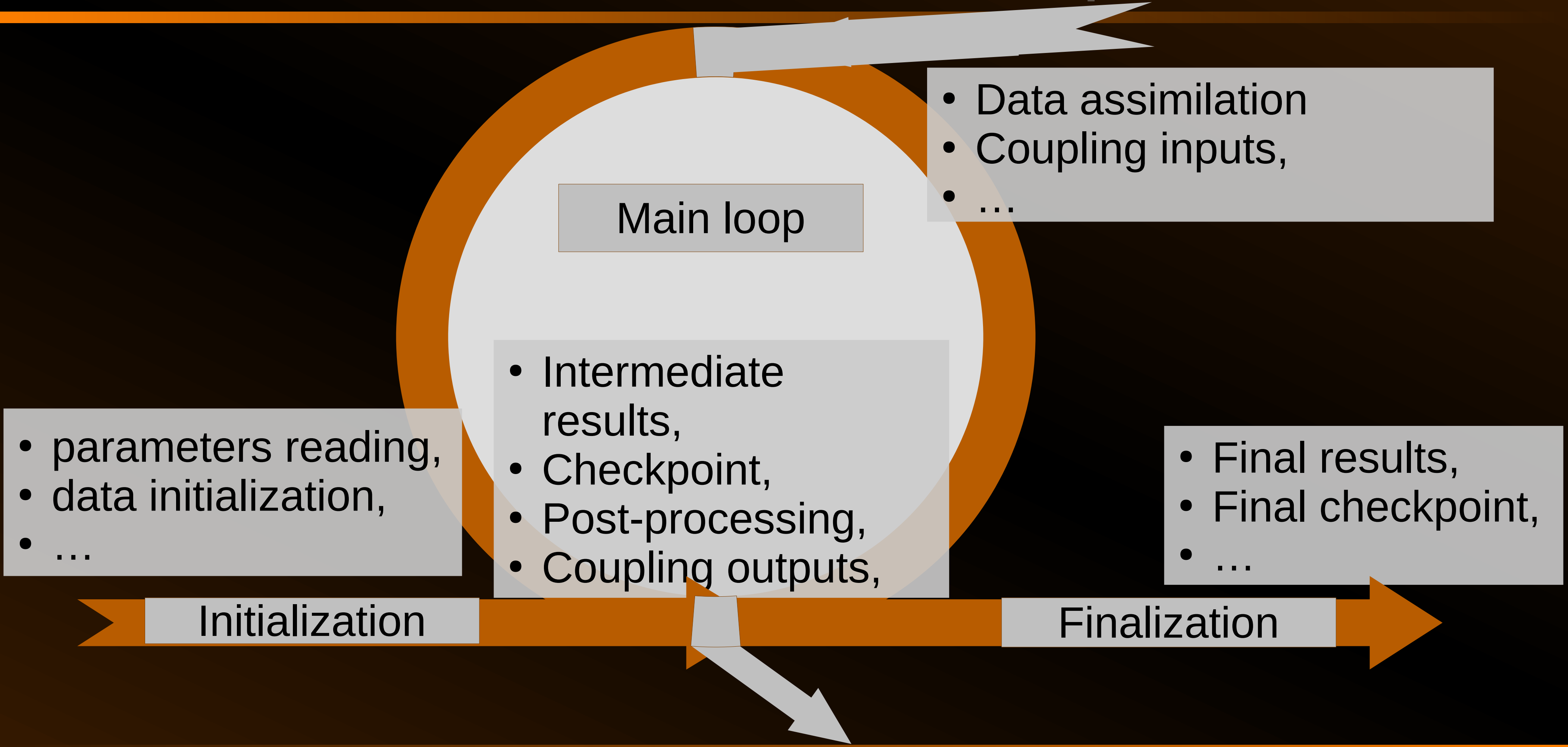
https://xkcd.com/927/

PDI is an Interface...
just an interface!

I/O in codes: let's take a step back



I/O in codes: let's take a step back



I/O in codes: let’s take a step back

Similar from the code point of view:

- Import or export data

But… different libraries needed

- Data assimilation
- Coupling inputs,
- …

Main loop

- parameters reading,
- data initialization,
- …

- Intermediate results,
- Checkpoint,
- Post-processing,
- Coupling outputs,

- Final results,
- Final checkpoint,
- …

Initialization

Finalization

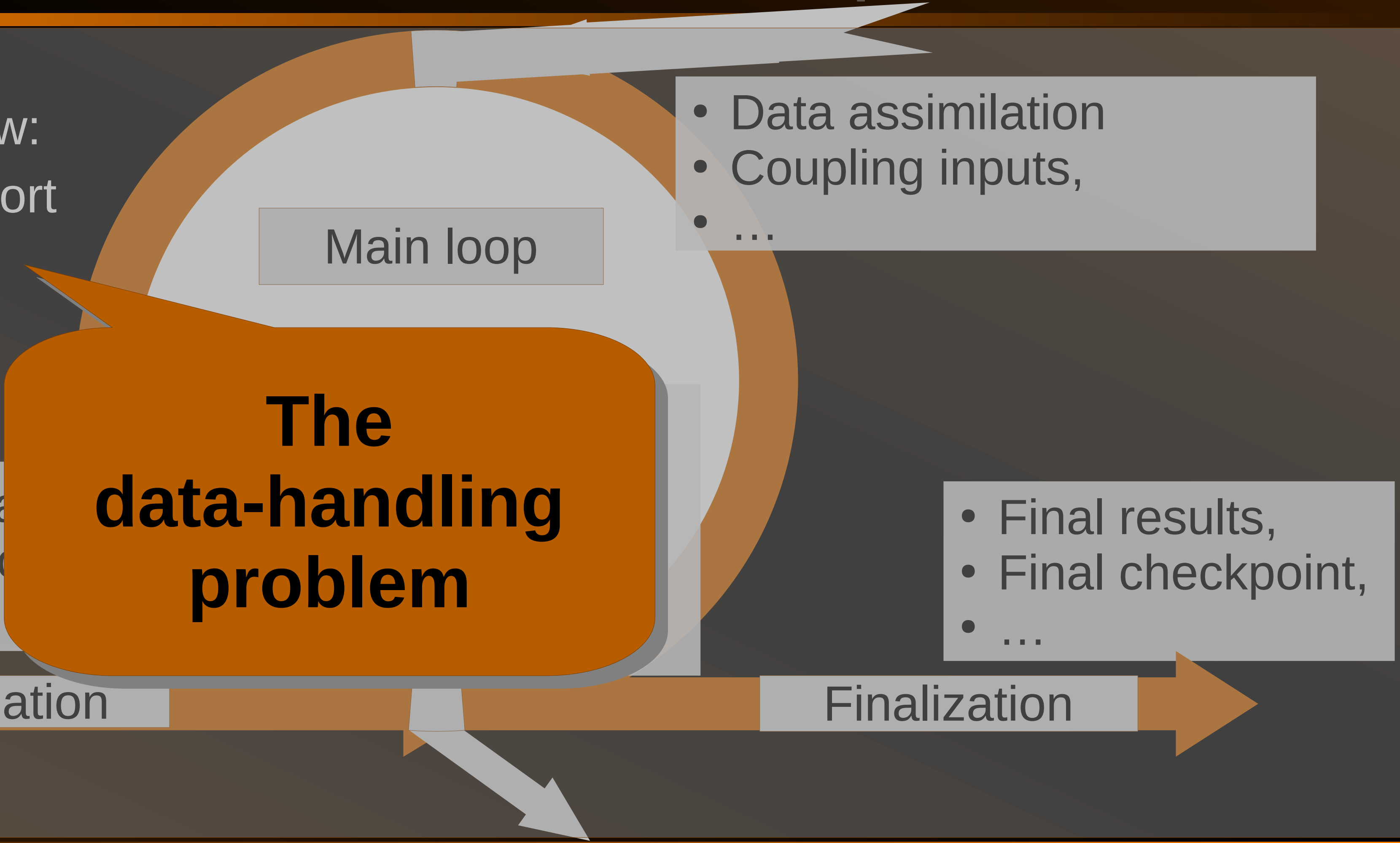
I/O in codes: let’s take a step back

Similar from the code point of view:

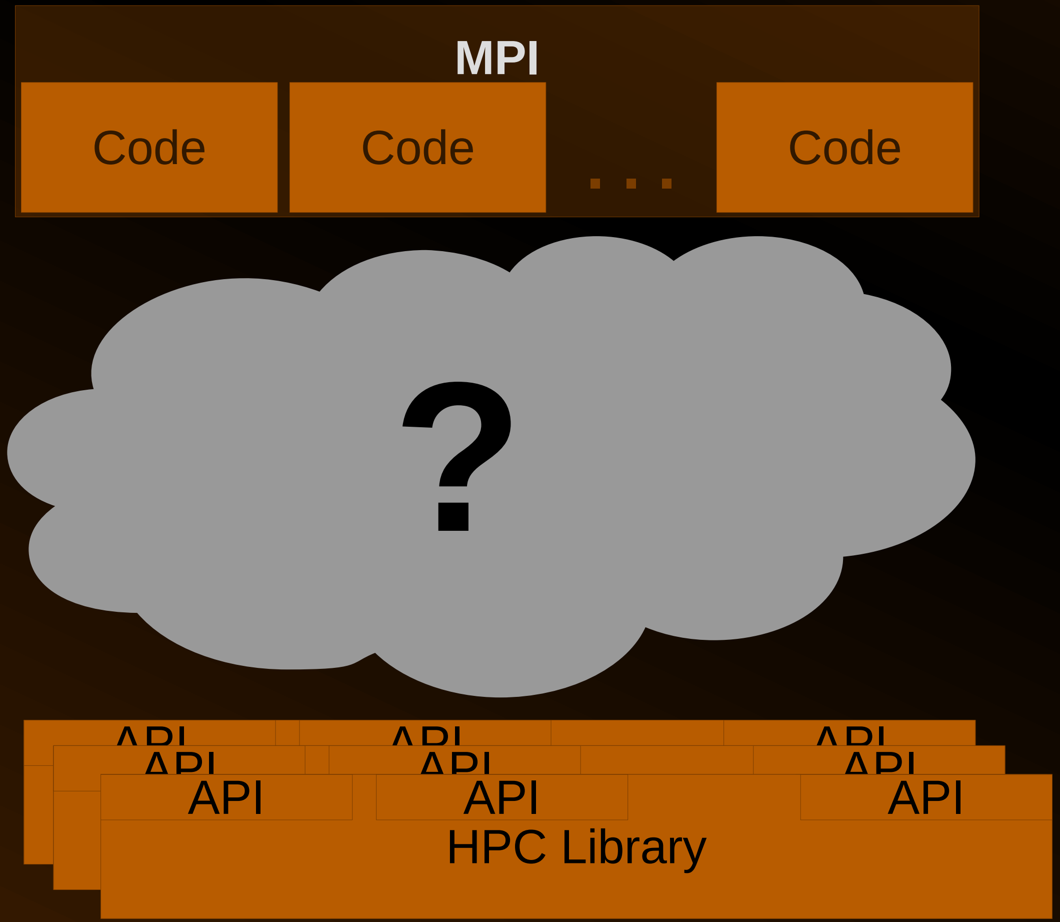
- Import or export data

But… different libraries needed

- parameters rea
- data initializati
- …

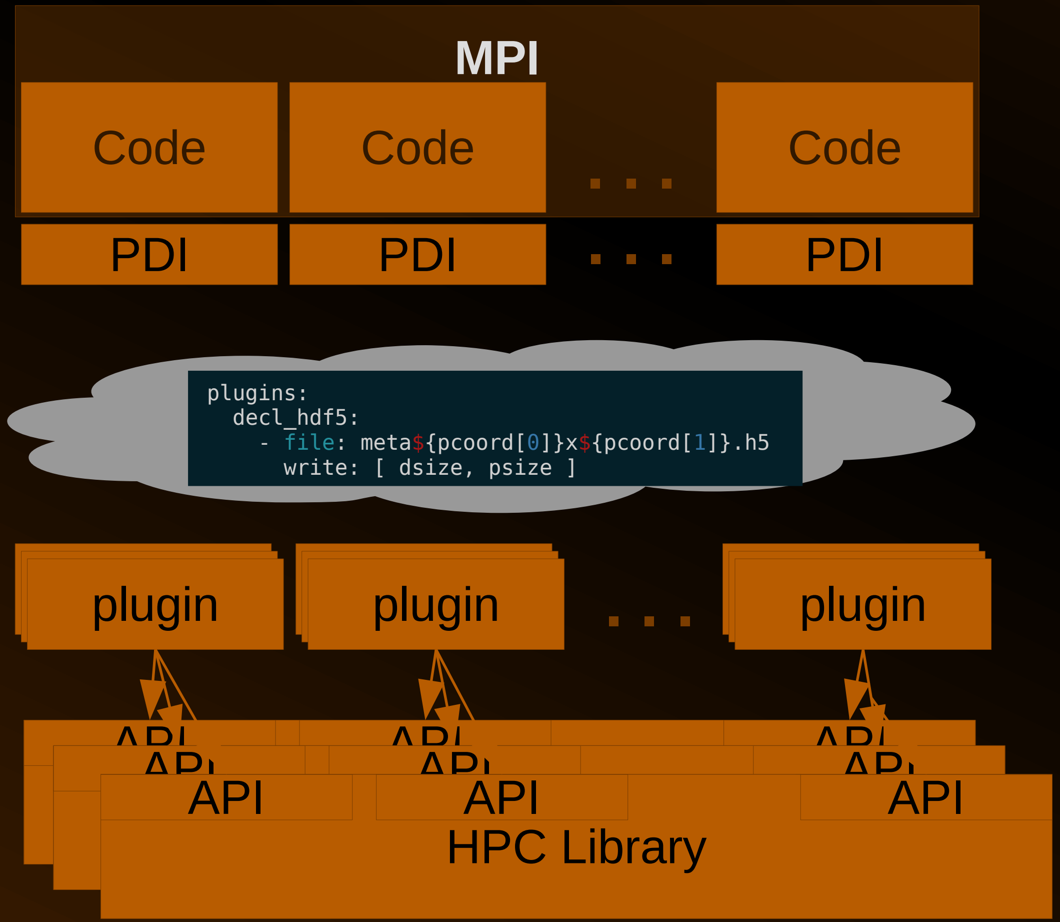


What is PDI?



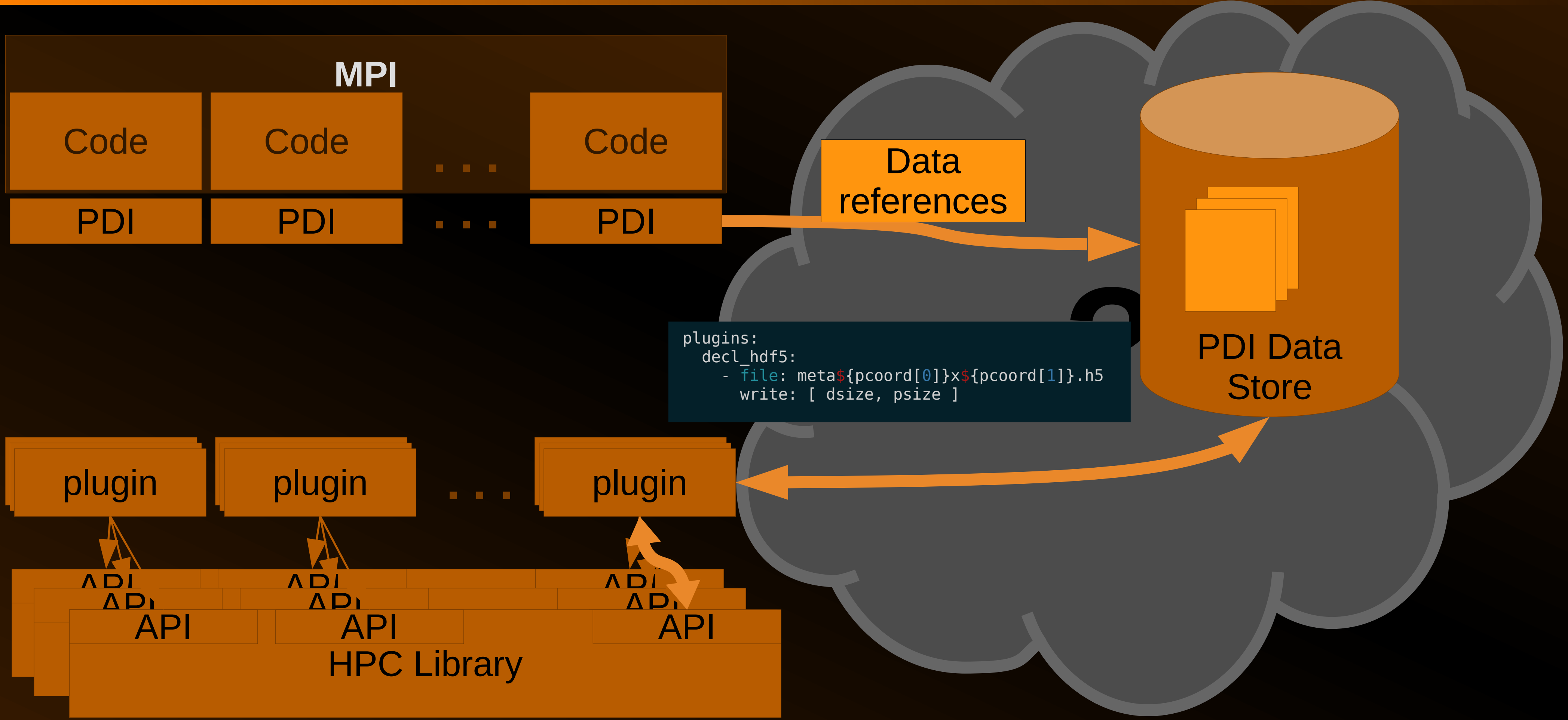
- PDI annotations: a purely declarative API
- Plugins for access to existing libraries

What is PDI?



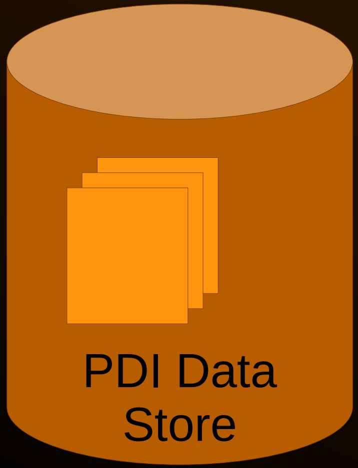
- PDI annotations: a purely declarative API
- PDI YAML spec. tree: What to do with data
- Plugins for access to existing libraries

What is PDI?


MPI
Code
Code
Code
PDI
PDI
PDI
Data references
PDI Data Store
plugins:
  decl_hdf5:
    - file: meta${pcoord[0]}x${pcoord[1]}.h5
      write: [dsize, psize]
plugin
plugin
plugin
API
API
API
HPC Library

Inside PDI: The shared data store

- PDI data store: a dict of buffer references
  - Name ⇒ unique identifier
  - Reference
    - Ownership & locking information
      - RW-lock: Single Writer / Multiple Readers
      - Memory ownership : Strong or Semi-weak
    - Type ⇒ memory layout and interpretation
    - Buffer address ⇒ pointer to user memory (CPU/GPU[WIP])



PDI data model

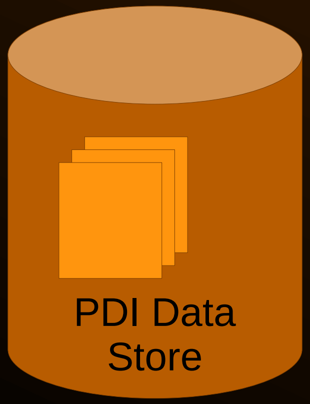
```
#pragma pdi metadata
int buffer_size;
#pragma pdi size:[$buffer_size+1]
double *main_buffer;
```

- Data type: memory layout & sematics
  - Annotations (C/C++), fully automatic (Python), or YAML (Fortran)
  - MPI / HDF5 inspired model: scalar / array / record
- “Data” vs. “Metadata”
  - PDI only handles the pointer for “data”
    - Minimal overhead
  - PDI keeps a copy of “metadata”
    - Can be used in $-expressions

Kevin Barre

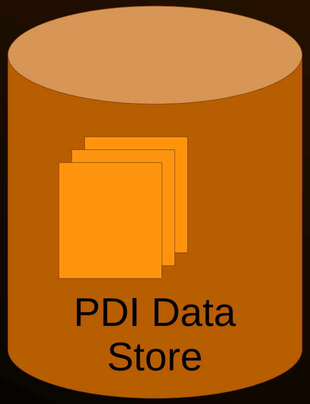
Inside PDI: the store + notifications

- PDI data store: a dict of buffer references
  - Name ⇒ unique identifier
  - Reference
    - Ownership & locking information
      - RW-lock: Single Writer / Multiple Readers
      - Memory ownership : Strong or Semi-weak
    - Type ⇒ memory layout and interpretation
    - Buffer address ⇒ pointer to user memory (CPU/GPU[WIP])



Inside PDI: the store + notifications

- PDI data store: a dict of buffer references
  - Name ⇒ unique identifier
  - Reference
    - Ownership & locking information
      - RW-lock: Single Writer / Multiple Readers
      - Memory ownership : Strong or Semi-weak
    - Type ⇒ memory layout and interpretation
    - Buffer address ⇒ pointer to user memory (CPU/GPU[WIP])



- Notification system: plugins register to be called
  - On data share / access
  - On arbitrary locations in code (named “events”)

PDI: A simple API

```c
/** Initializes PDI */
PDI_status_t PDI_init(PC_tree_t yaml_conf);

/** Finalizes PDI */
PDI_status_t PDI_finalize();
```

a C / C++ API
Also available for:
- Fortran
- Python

- **Init** takes the specification tree as parameter
  - The YAML is parsed using the *paraconf* library
- **Finalize** releases all PDI-related resources

PDI Annotation API

```c
typedef enum { PDI_IN, PDI_OUT, PDI_INOUT } PDI_inout_t;

// A data buffer is ready (filled)
PDI_status_t PDI_share(const char *name, void *data, PDI_inout_t access);

// A buffer will be reused
PDI_status_t PDI_reclaim(const char *name);
```

a C / C++ API
Also available for:
- Fortran
- Python

- Share
  - A buffer is in a coherent consistent state
  - Reference the buffer in PDI store
- Reclaim
  - The buffer will be reused for a different use
  - Un-reference the buffer in PDI store

PDI: Annotation API usage


```c
double* data_buffer = malloc( buffer_size*sizeof(double) );

while ( !computation_finished )
{
    compute_the_value_of( data_buffer, /*...*/ );
    PDI_share("main_buffer", data_buffer, PDI_OUT);
    do_something_without_data_buffer();
    do_something_reading( data_buffer, /*...*/ );
    PDI_reclaim("main_buffer");
    update_the_value_of( data_buffer, /*...*/ );
}
```

buffer is shared
- between here

…
- and here

- Creates a “shared region” in code where
  - Data referenced in PDI store
  - Plugins can use it
- Code should refrain from
  - modifying it (**PDI_IN|OUT**)
  - accessing it (**PDI_IN**)

PDI Additional API

PDI

```c
typedef enum { PDI_IN, PDI_OUT, PDI_INOUT } PDI_inout_t;

// Combine share & expose for 1 piece of data
PDI_status_t PDI_expose(const char *name, void *data, PDI_inout_t access);

// When there is no data… (an interesting location has been reached)
PDI_status_t PDI_event(const char* event);

// And when there is multiple data
PDI_status_t PDI_multi_expose(const char *event_name,
    void *data, PDI_inout_t access,
    …,          …,
    NULL );
```

a C / C++ API
Also available for:

- Fortran
- Python

- Expose = share + reclaim
- Events: similar to exposing empty data

- Multi-expose:
  - All share
  - An event
  - All reclaims

PDI approach: wrap-up

PDI

In code

- Write & annotate your code
- Annotate buffers availability (share / reclaim)
- Compile and… DONE! (on the code side)

In YAML

- Use pre-made plugins or write your own code to choose I/O libraries, describe behavior
  - React to events
  - Access data in the store

PDI in practice: Decl'HDF5

```
PDI_expose("buffer_size", &buffer_size, PDI_OUT);
double* data_buffer = malloc( buffer_size*sizeof(double) );

while ( iteration_id < max_iteration_id )
{
    compute_the_value_of( data_buffer, /*...*/ );
    PDI_share("main_buffer", data_buffer, PDI_OUT);
    do_something_reading( data_buffer, /*...*/ );
    PDI_reclaim("main_buffer");
}
```

- Write data in the HDF5 format
- Heavily relies on
  - $-expressions
  - default configuration values
- Makes
  - *Simple* things **easy**
  - *Complex* things **possible**

Decl'HDF5: the YAML

```yaml
plugins:
  decl_hdf5:
    file: 'my_file_${iteration_id}x${rank}.h5'
    write: main_buffer
```

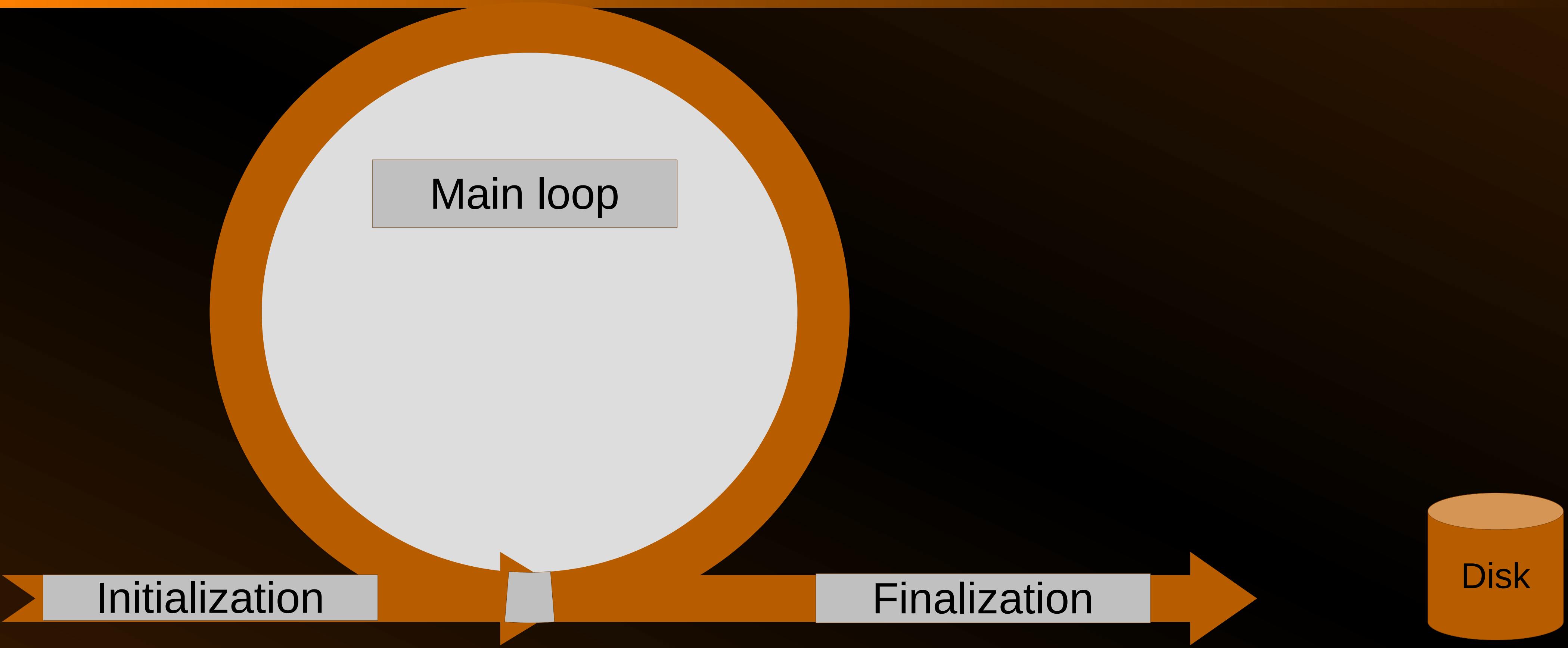
- Simple to just dump data as HDF5

Decl'HDF5: a complex YAML example

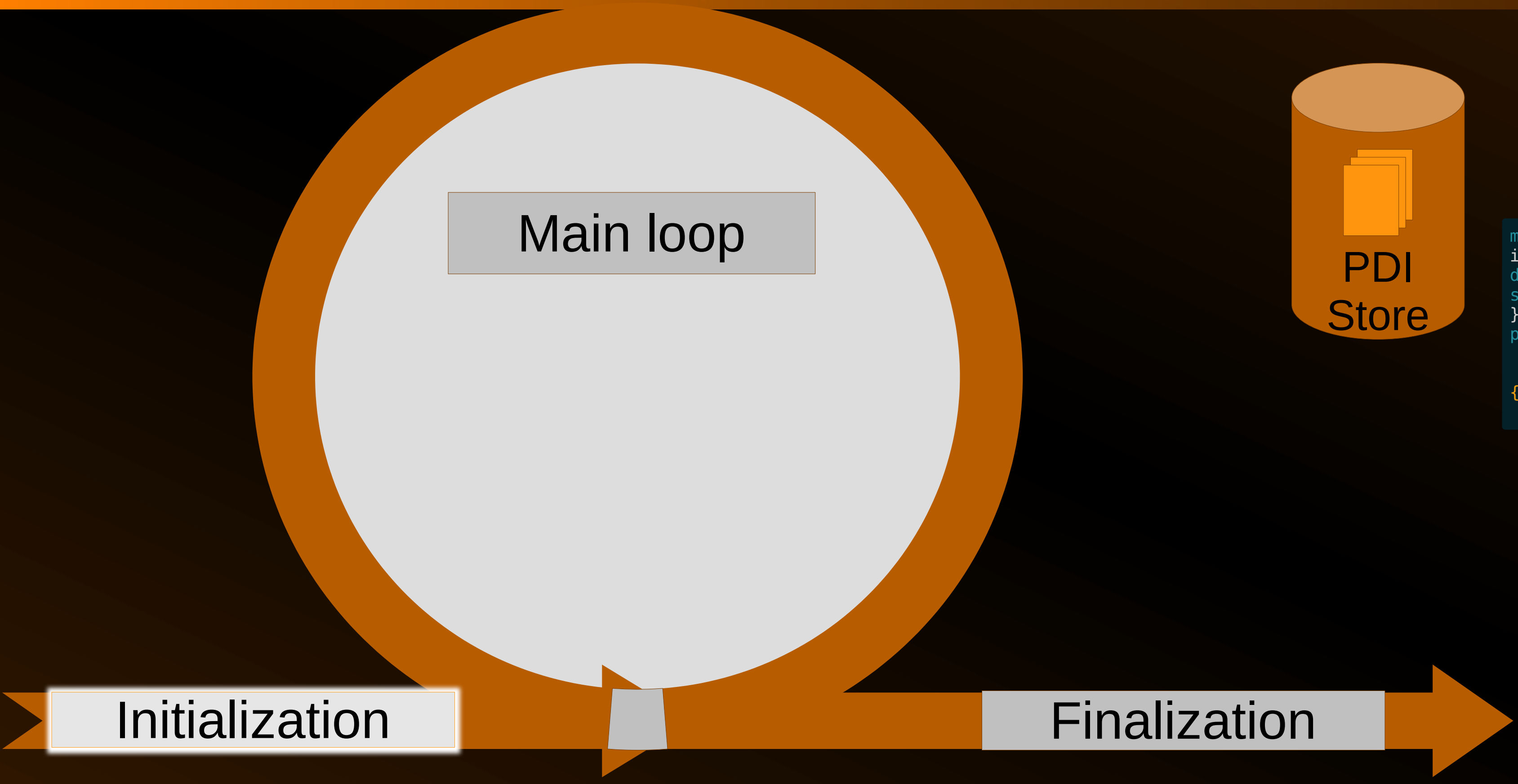
```yaml
plugins:
  decl_hdf5:
    file: 'my_file.h5'
    when: '$iteration_id % 100 = 0 & $iteration_id < 10000'
    datasets:
      main_dset:
        type: array
        subtype: double
        Size: [ '($buffer_size - 2) * $np', 100 ]
    write:
      main_buffer:
        memory_selection: { start: 1, size: '$buffer_size - 2' }
        dataset: main_dset
        dataset_selection:
          start: [ '($buffer_size - 2) * $iteration_id', '$iteration_id/100' ]
          size: [ '$buffer_size - 2', 1 ]
    communicator: $MPI_COMM_WORLD
  mpi:
```

- Possible to do complex rearranging of data in parallel

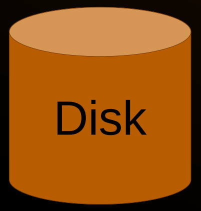
PDI: behind the scene



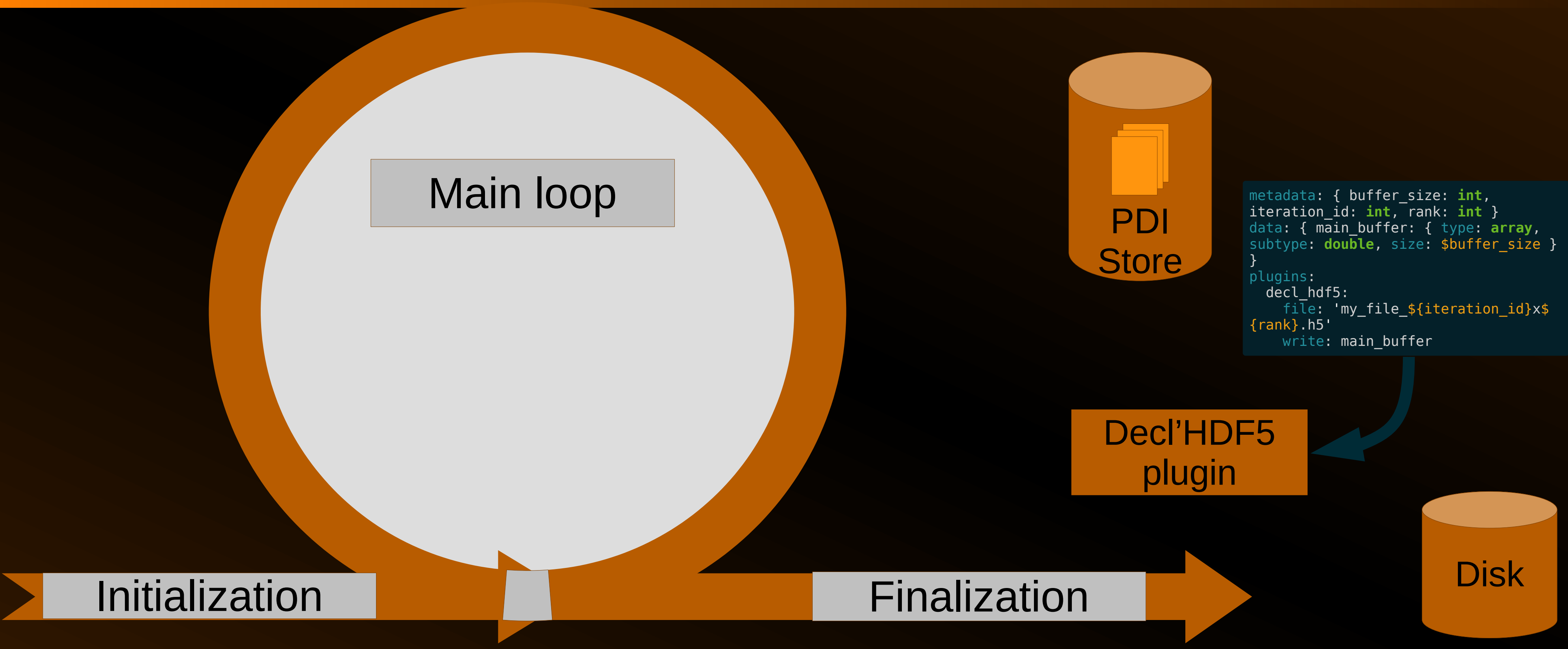
PDI: behind the scene




```yaml
metadata: { buffer_size: int,
iteration_id: int, rank: int }
data: { main_buffer: { type: array,
subtype: double, size: $buffer_size }
}
plugins:
  decl_hdf5:
    file: 'my_file_${iteration_id}x$
{rank}.h5'
    write: main_buffer
```

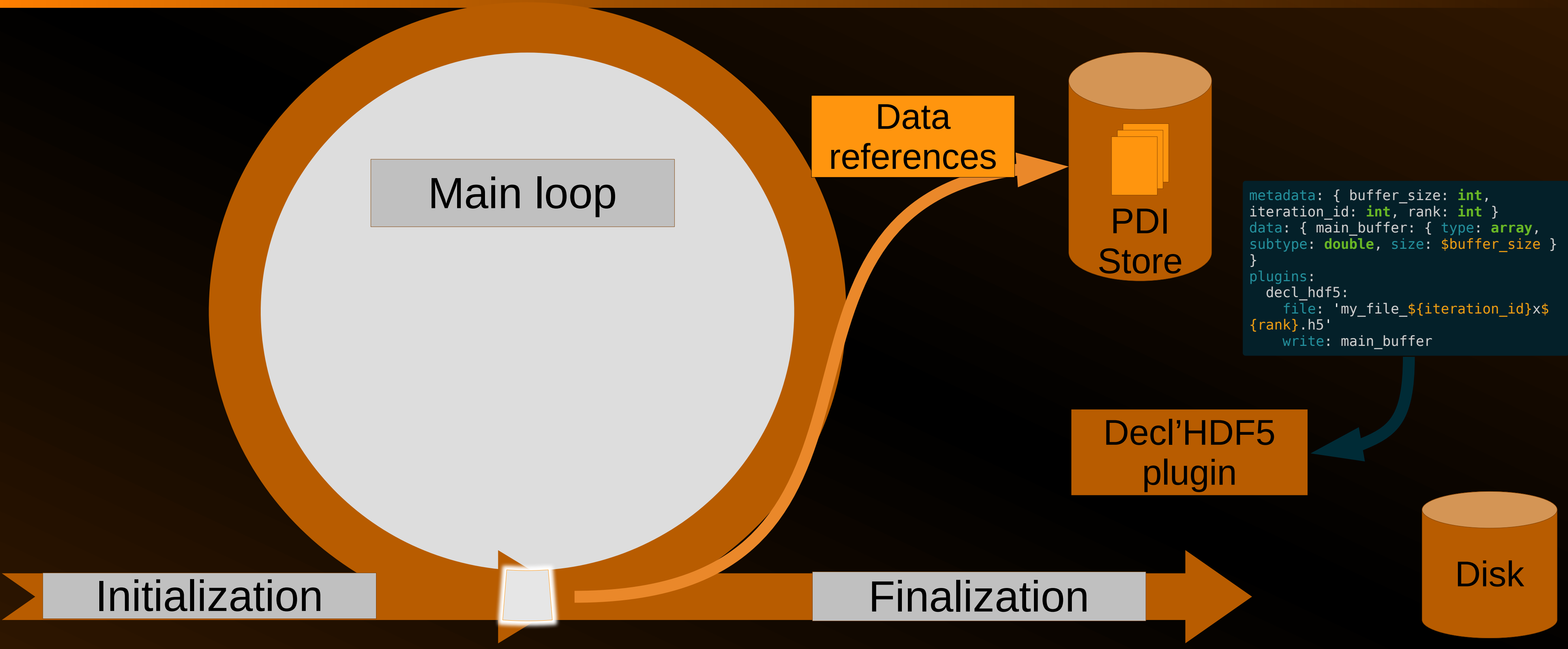


PDI: behind the scene



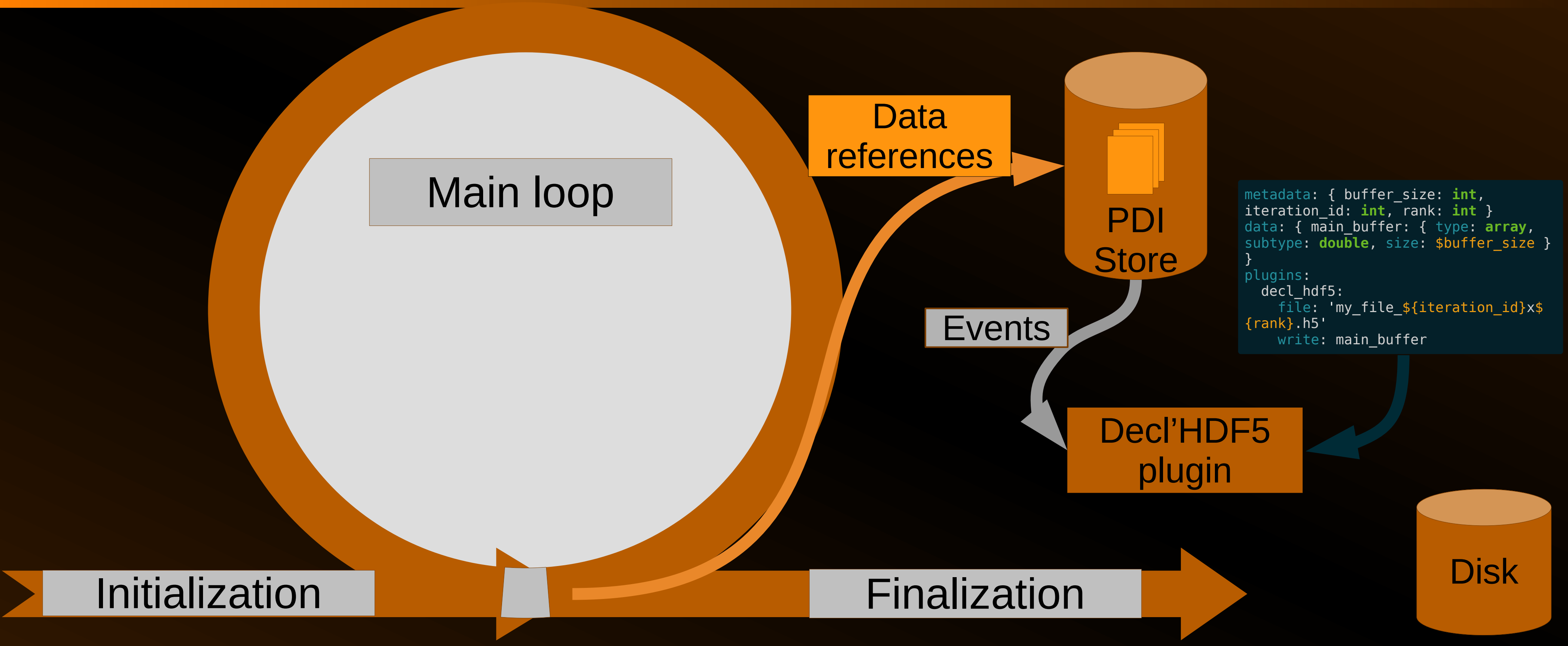

```yaml
metadata: { buffer_size: int,
iteration_id: int, rank: int }
data: { main_buffer: { type: array,
subtype: double, size: $buffer_size }
}
plugins:
  decl_hdf5:
    file: 'my_file_${iteration_id}x$
{rank}.h5'
    write: main_buffer
```

PDI: behind the scene

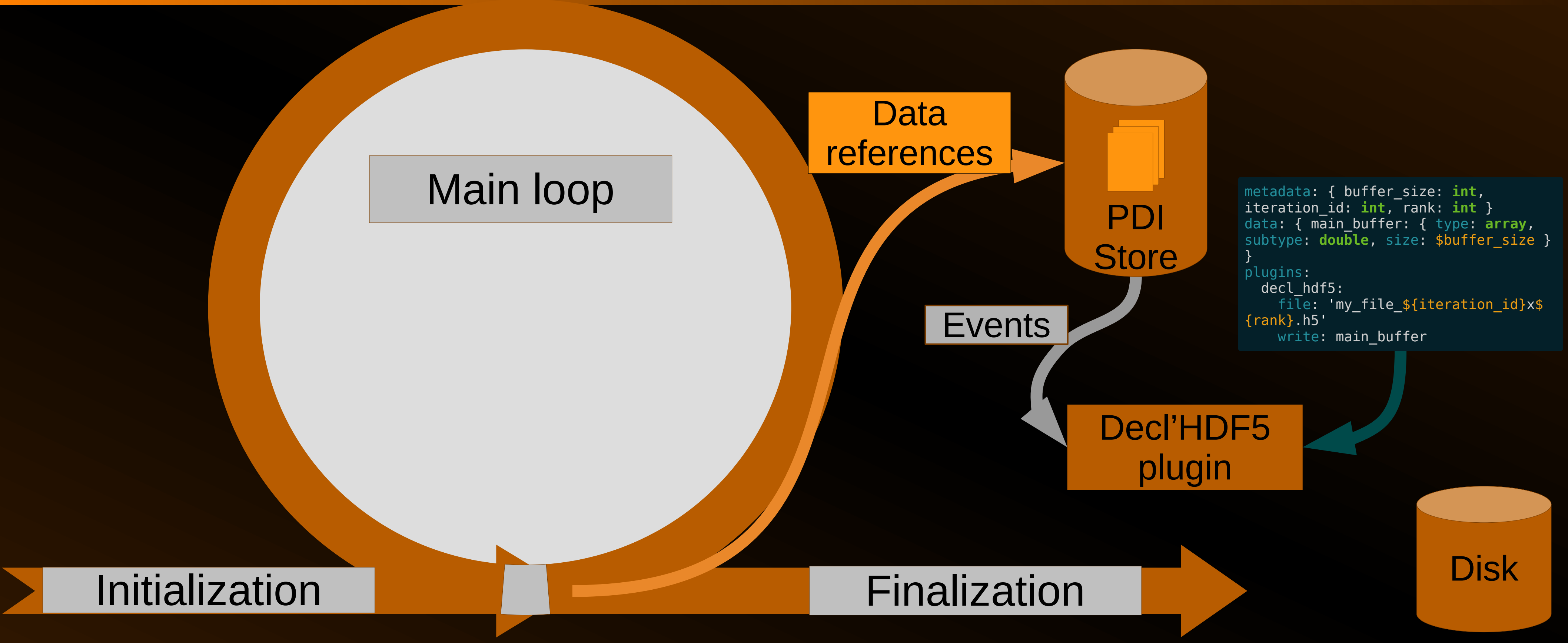



```yaml
metadata: { buffer_size: int,
iteration_id: int, rank: int }
data: { main_buffer: { type: array,
subtype: double, size: $buffer_size }
}
plugins:
  decl_hdf5:
    file: 'my_file_${iteration_id}x$
{rank}.h5'
    write: main_buffer
```

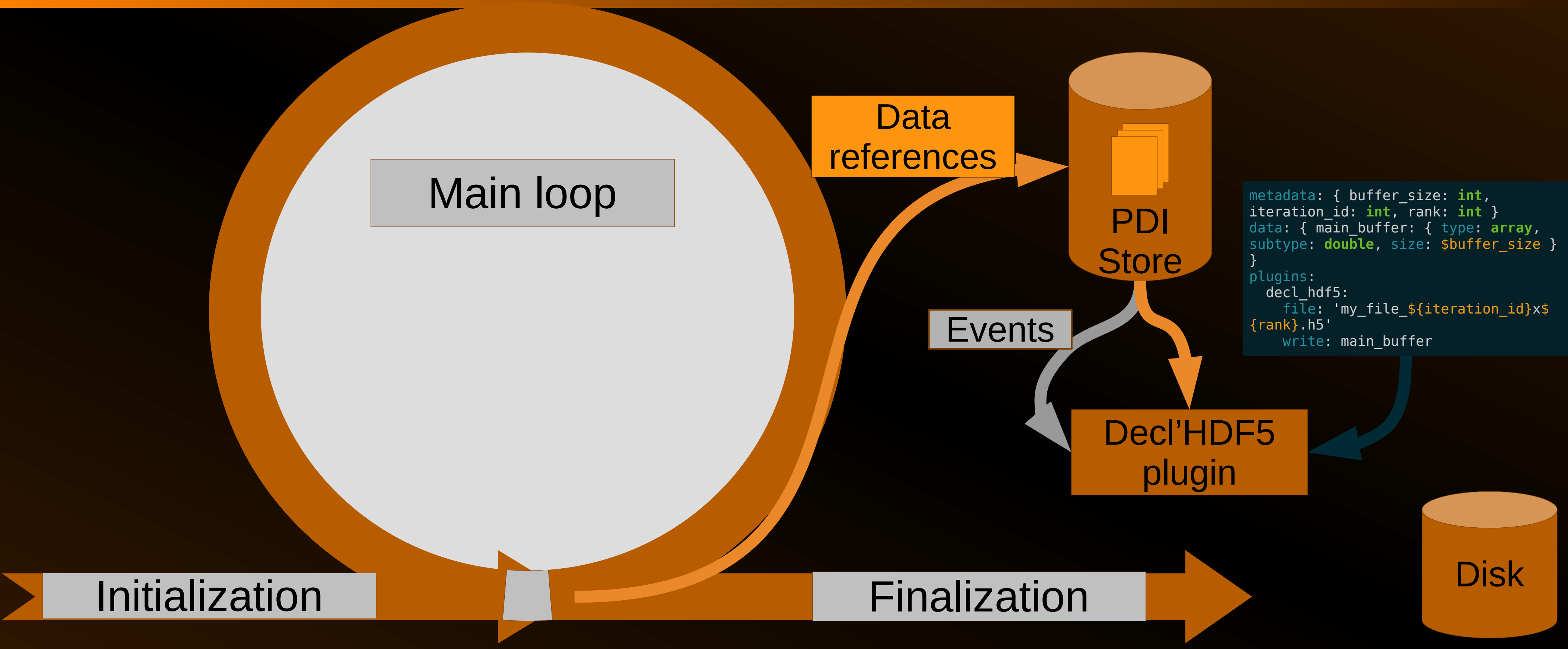
PDI: behind the scene



PDI: behind the scene

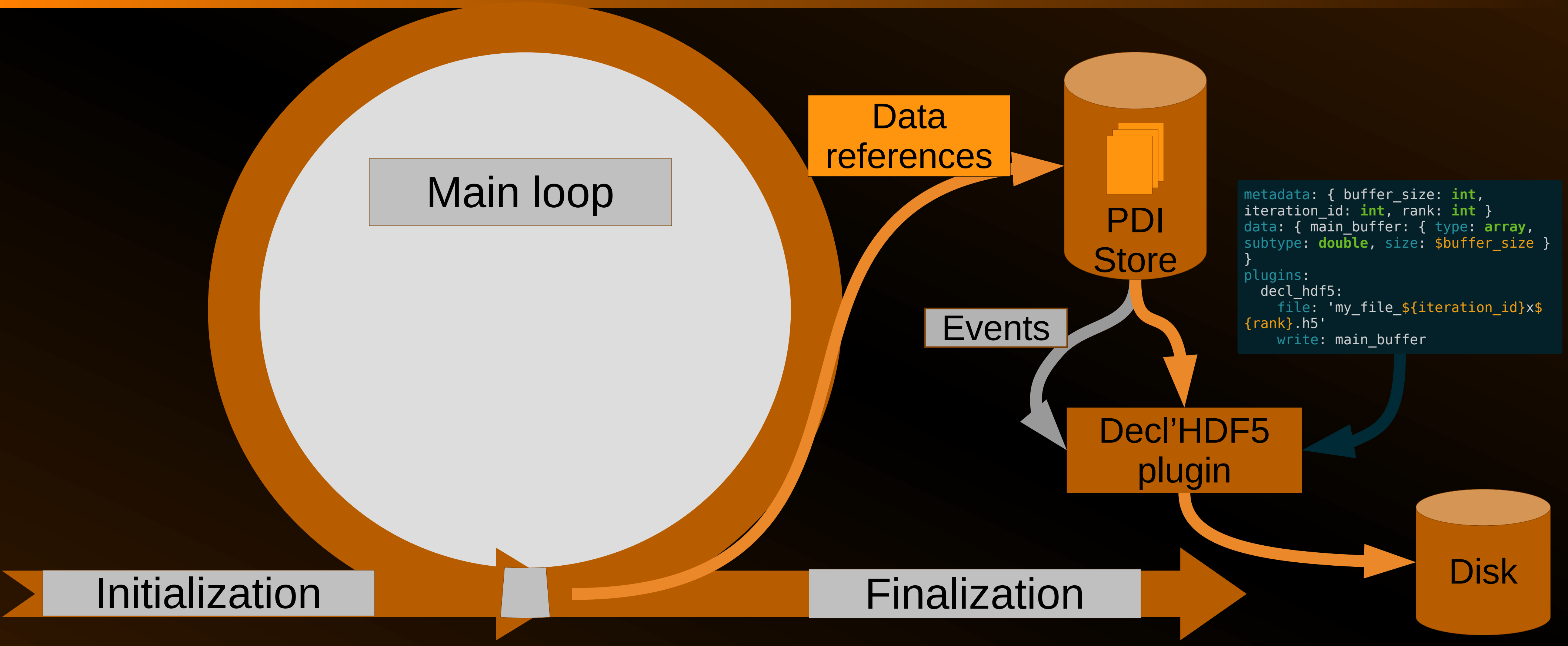


PDI: behind the scene




```yaml
metadata: { buffer_size: int,
iteration_id: int, rank: int }
data: { main_buffer: { type: array,
subtype: double, size: $buffer_size }
}
plugins:
  decl_hdf5:
    file: 'my_file_${iteration_id}x$
{rank}.h5'
    write: main_buffer
```

PDI: behind the scene



PDI: The plugins

- IO libraries
  - HDF5 / parallel HDF5, NetCDF4 / pNetCDF4, SIONlib
- Special purpose IO
  - FTI, ADIOS / SENSEI
- Workflow integration
  - Dask w. Deisa, FlowVR, Melissa
- Your own code
  - $-expressions based language, Python, C, C++, Fortran

Data coupling with PDI: pycall

```yaml
plugins:
  pycall:
    on_event:
      trigger_event_name: # event that triggers the call
        with: { iter: $iteration_id, original_data: $main_field }
        exec: |
          if iter<1000:
               new_data = original_data*4 # uses numpy
               pdi.expose('new_data', new_data, pdi.OUT);
```

- Let you call your own Python code
  - Data is exposed as numpy arrays
  - Numpy arrays can be re-exposed

  ⇒ In-process post-processing and data transformation

Data coupling with PDI: user-code

PDI

```yaml
plugins:
  user_code:
    on_event:
      trigger_event_name: # event that triggers the call
        function_name { in1: $iteration_id, in2: $main_field }
```

```c
void function_name(void)
{
     int* iter = NULL; PDI_access("in1", &iter, PDI_IN);
     double* main_field = NULL; PDI_access("in2", &iter, PDI_IN);
     // …
     PDI_release("in2");
     PDI_release("in1");
}
```

- Let you call your own (C/Fortran) functions
  - When performance matters
  - To call library APIs not covered by plugins

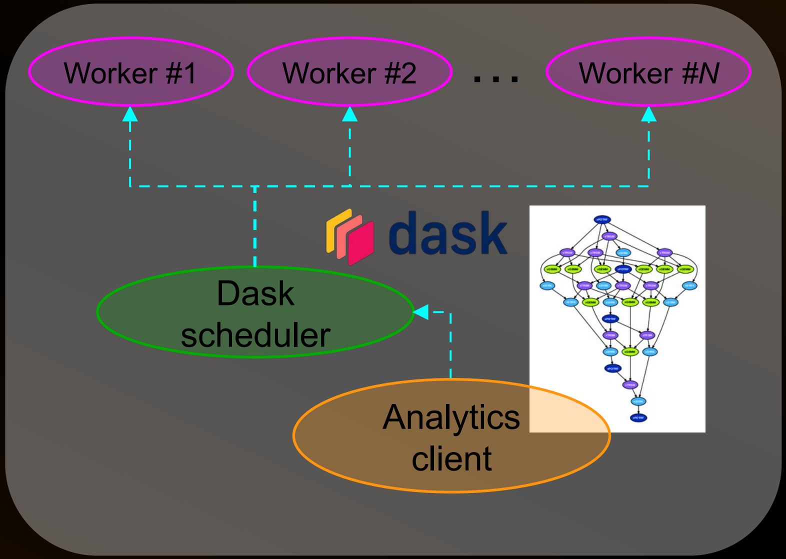
Post hoc data analytics with python

```python
from sklearn.decomposition import IncrementalPCA
import yaml, json
import h5py
# load the simulation configuration
simu = yaml.load(open('simulation.yml'))
# Load data from HDF5
gtemp = h5py.File('data.hdf5',mode='r')['gtemp']
# process each time-step independently
for step in range(0, simu['timesteps']):
  pca = IncrementalPCA(n_components=2, copy=False,
                       svd_solver='randomized')
  pca.fit(gtemp[step,:,:])
  print(pca.explained_variance_)
```

Asahi, Y. & Fujii, K. & Heim, D. & Maeyama, S. & Garbet, X. & Grandgirard, V. & Sarazin, Y. & Dif-Pradalier, G. & Idomura, Y. & Yagi, M. (2021). “**Compressing the time series of five dimensional distribution function data from gyrokinetic simulation using principal component analysis**”. *Physics of Plasmas*. 28. 012304. 10.1063/5.0023166.

Dask distributed?

- A scheduler/workers (+client) model to run work (each on its own process/node)
- A task-based model to describe work
- Many tools ported to dask for ease of use
  - Numpy / SciPy
  - Scikit-learn
  - Pandas
  - ...



Post hoc data analytics with python

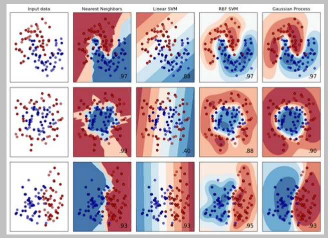
PDI

```python
from sklearn.decomposition import IncrementalPCA
import yaml, json
import h5py
# load the simulation configuration
simu = yaml.load(open('simulation.yml'))
# Load data from HDF5
gtemp = h5py.File('data.hdf5',mode='r')['gtemp']
# process each time-step independently
for step in range(0, simu['timesteps']):
  pca = IncrementalPCA(n_components=2, copy=False,
                          svd_solver='randomized')
  pca.fit(gtemp[step,:,:])
  print(pca.explained_variance_)
```

Asahi, Y. & Fujii, K. & Heim, D. & Maeyama, S. & Garbet, X. & Grandgirard, V. & Sarazin, Y. & Dif-Pradalier, G. & Idomura, Y. & Yagi, M. (2021). “**Compressing the time series of five dimensional distribution function data from gyrokinetic simulation using principal component analysis**”. *Physics of Plasmas*. 28. 012304. 10.1063/5.0023166.

Post hoc data analytics with Dask

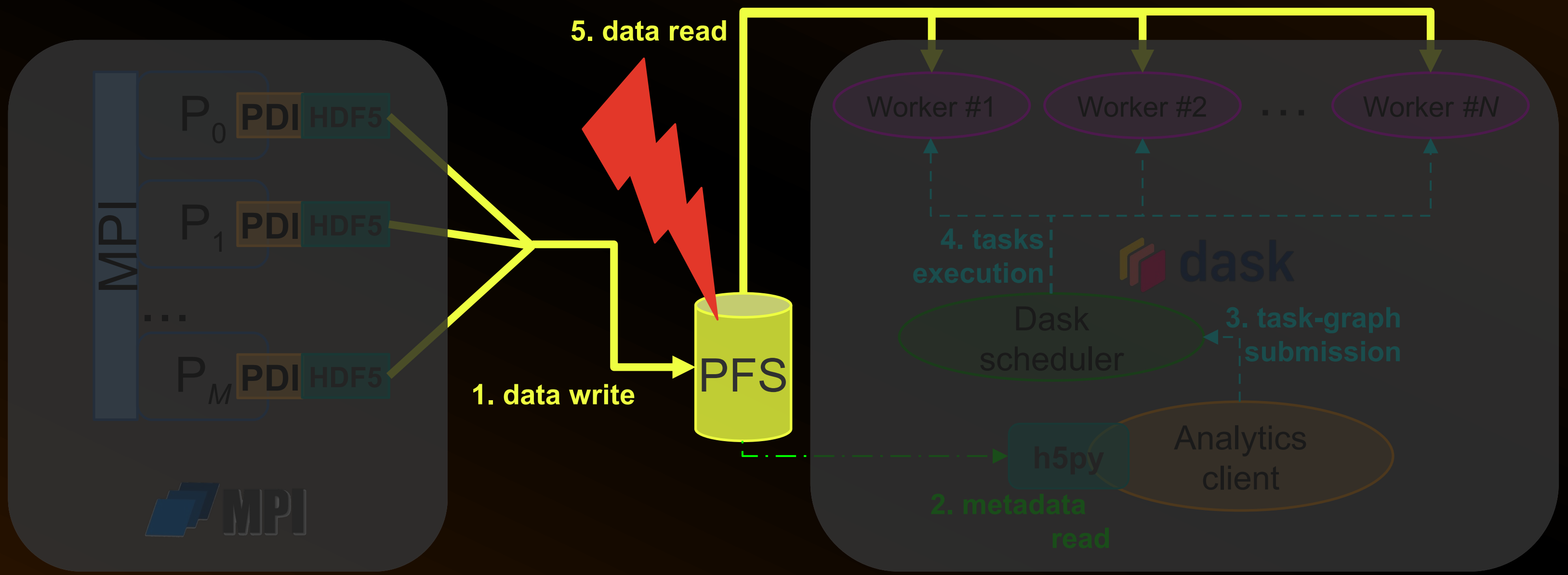
```python
import dask.array as da
from dask_ml.decomposition import IncrementalPCA
import yaml, json
import h5py
# Connect to Dask
sched = json.load(open('sched.json'))
client = dask.distributed.Client(sched["address"])
# load the simulation configuration
simu = yaml.load(open('simulation.yml'))
# Build a lazy array descriptor from HDF5
gtemp = h5py.File('data.hdf5',mode='r')['gtemp']
gtemp = da.from_array(gtemp, chunks=(1,4096,4096))
for step in range(0, simu['timesteps']):
  pca = IncrementalPCA(n_components=2, copy=False,
                       svd_solver='randomized')
  pca.fit(gtemp[step,:,:])
  print(pca.explained_variance_)
```



Asahi, Y. & Fujii, K. & Heim, D. & Maeyama, S. & Garbet, X. & Grandgirard, V. & Sarazin, Y. & Dif-Pradalier, G. & Idomura, Y. & Yagi, M. (2021). “**Compressing the time series of five dimensional distribution function data from gyrokinetic simulation using principal component analysis**”. *Physics of Plasmas*. 28. 012304. 10.1063/5.0023166.

Dash for post hoc analytics

```yaml
plugins:
  decl_hdf5:
    file: 'my_file_${iteration_id}x${rank}.h5'
    write: main_buffer
```

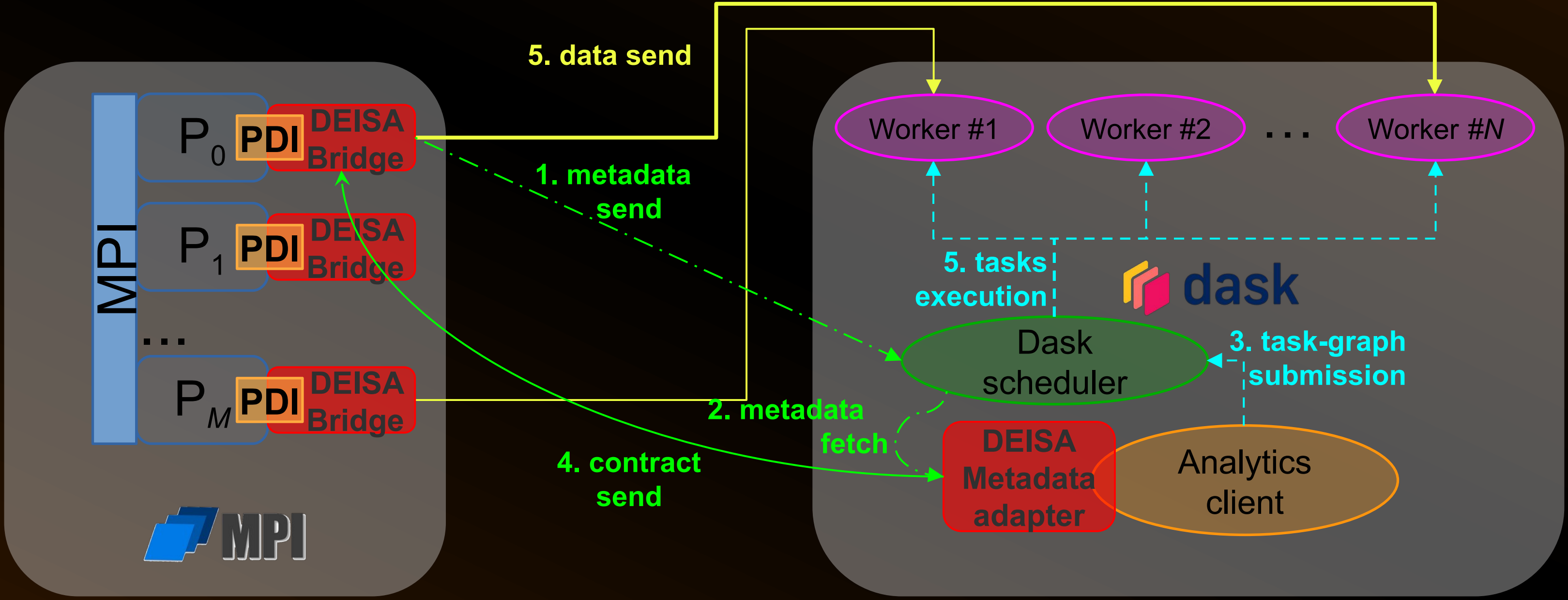


- File-system requirements are huge
  - Let's run simulation & analysis at the same time
  - Erase files as soon as they are not required anymore
- File-system performance is still an issue

Yuuichi Asahi (JAEA)
Antoine Lavandier (MdlS)

Dask in situ with Deisa

```yaml
plugins:
  deisa:
    scheduler_file: "/home/user/xp/sched.json"
    transfer: { main_field: { when: "$iteration_id>0" } }
```



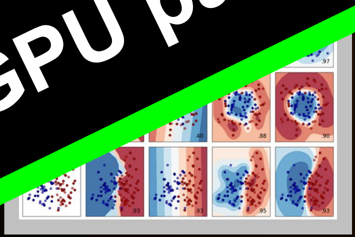
Amal Gueroudji, Julien Bigot, Bruno Raffin. **"DEISA: dask-enabled in situ analytics."** *HiPC 2021 - 28th International Conference on High Performance Computing, Data, and Analytics*, Dec 2021, virtual, India

Amal Gueroudji. "**Distributed Task-Based In Situ Data Analytics for High-Performance Simulations**". *PhD Thesis*, Université Grenoble Alpes [2020-..], 2023. English.

Amal Gueroudji (MdlS)

Deisa: The analytics code

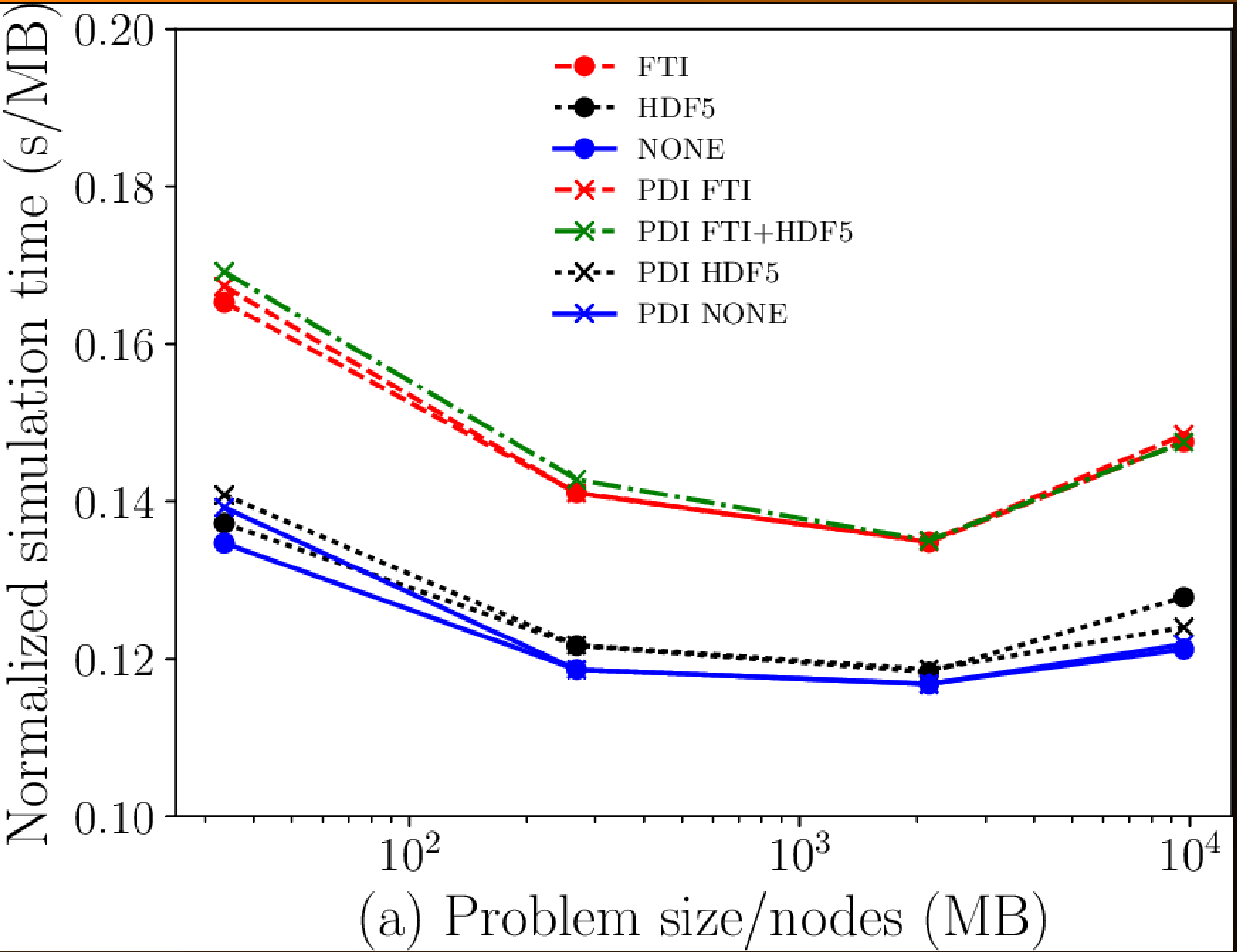
```
import dask.array as da
from dask_ml.decomposition import Increme
import yaml, json
import deisa
# Connect to Dask
sched = json.load(open('s
client = dask.distrib
# load the simulat
simu = yaml.lo
# Get data
gtemp
fo
                                  , copy=False,
                                  ='randomized')

                              ance_)
```



Used in production for grand-challenge on Adastra (CINES) #10 Top500
Multi-day full-scale run on the whole GPU partition

PDI: Perf. Evaluation 1/2

PDI



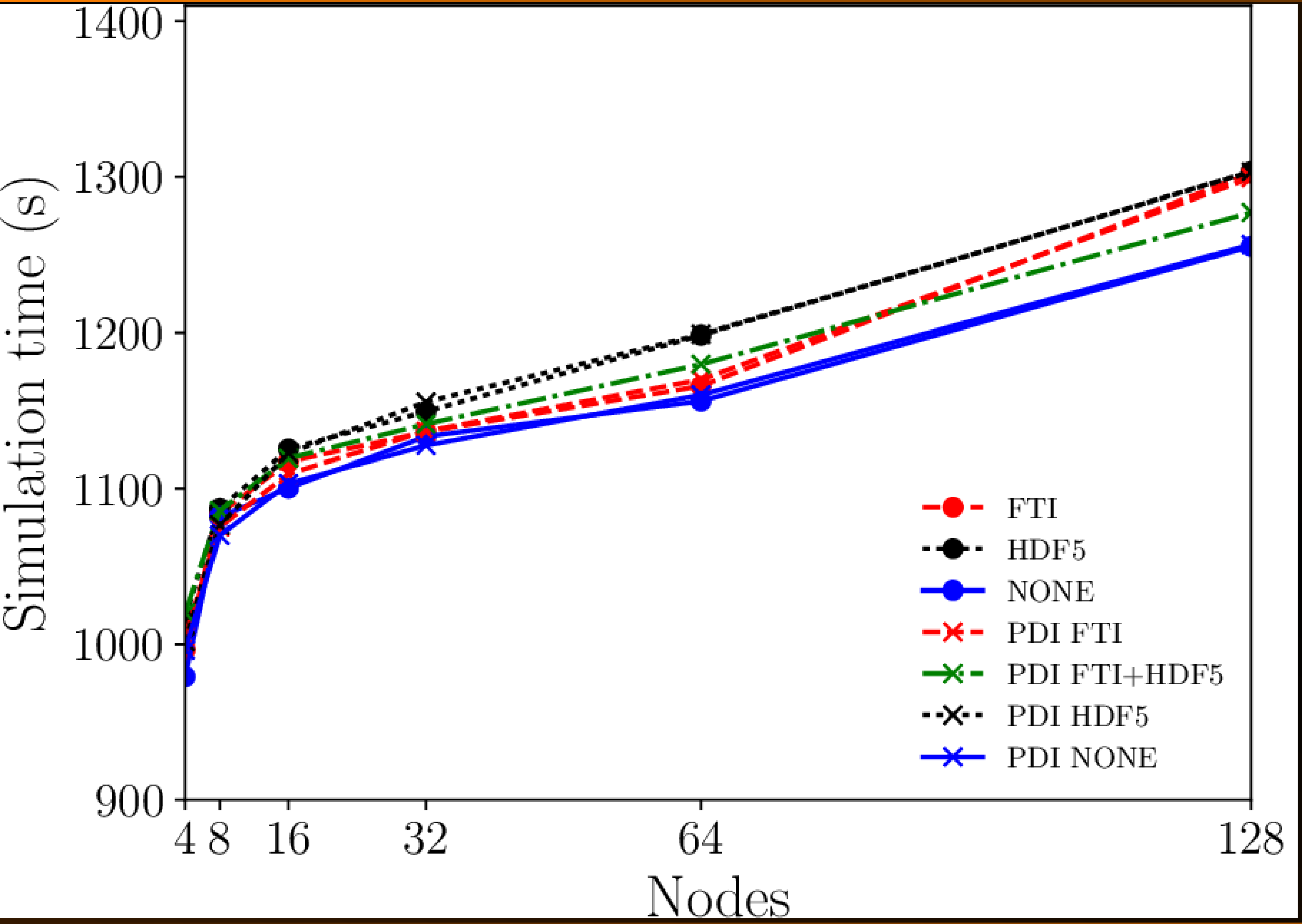
Corentin Roussel (MdlS)
Kai Keller (BSC)

- 4 versions of Gysela
  - No checkpoint
  - HDF5 checkpoints
  - FTI fault-tolerance
  - PDI (none / HDF5 / FTI / HDF5+FTI)

Execution time by MB of checkpointed data on 4 MareNostrum
Nodes with and without PDI

PDI: Perf. Evaluation 2/2

Corentin Roussel (MdlS)
Kai Keller (BSC)

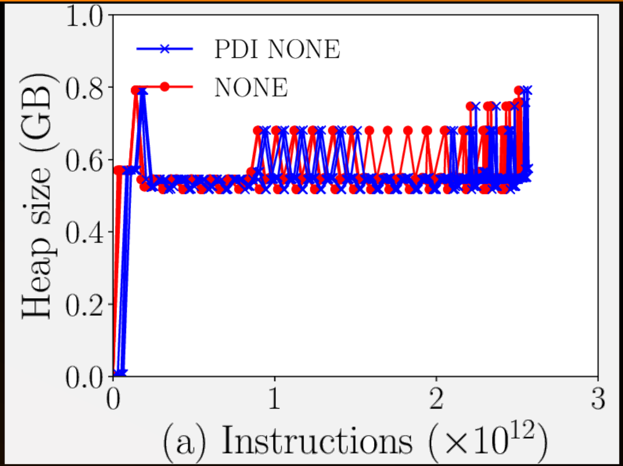


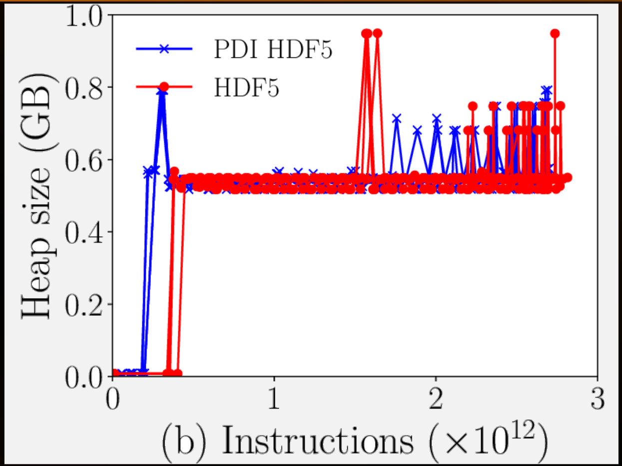
Gysela Wallclock time in weak scaling on Curie (TGCC – France) with and without PDI

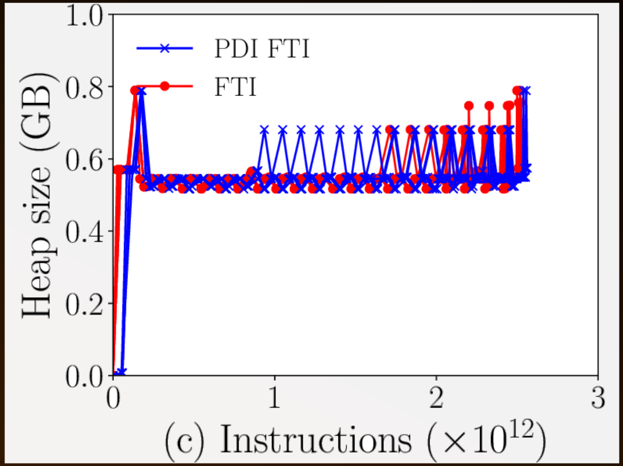
Checkpointed data ~2.1GB/node

PDI: Memory overhead

Corentin Roussel (MdlS)
Kai Keller (BSC)







Memory usage during a Gysela execution with and without PDI on 4 nodes of MareNostrum (BSC – Spain)

PDI In practice

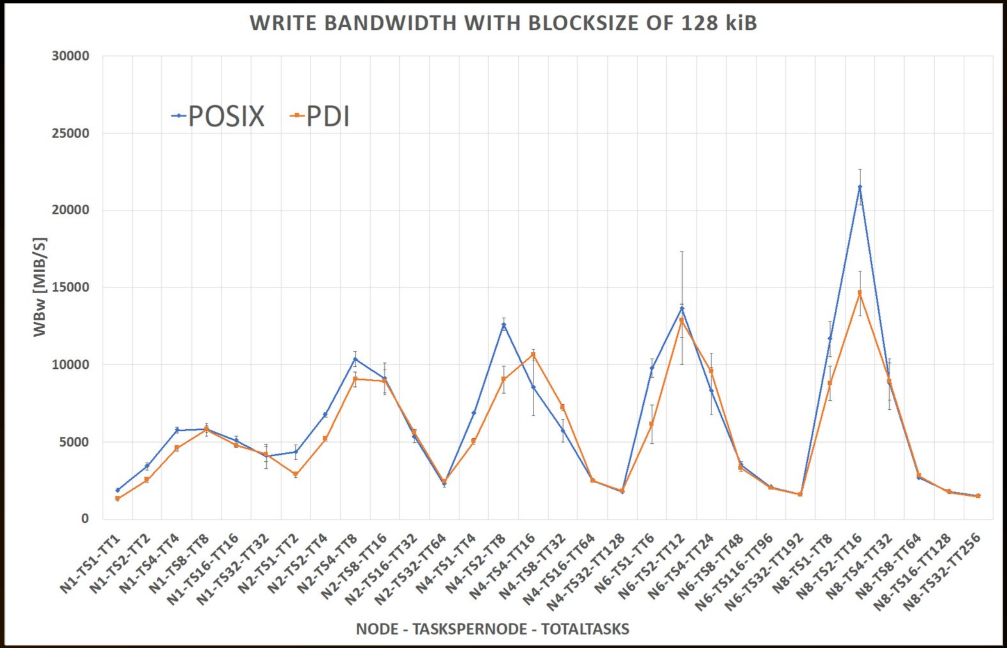
- PDI is publicly available (BSD 3-clause license)
  - Regular releases since 2014
  - Packages available for Debian, Fedora, Ubuntu, Spack
  - Documentation & tutorials available @ https://pdi.dev/1.6/
  - Heavily tested & validated
    - more than 1500 tests…
    - …running on more than 12 platforms each
- Integration in production codes
  - Gysela, Parflow, ESIAS, Manta?, …
- Part of NumPEx software stack

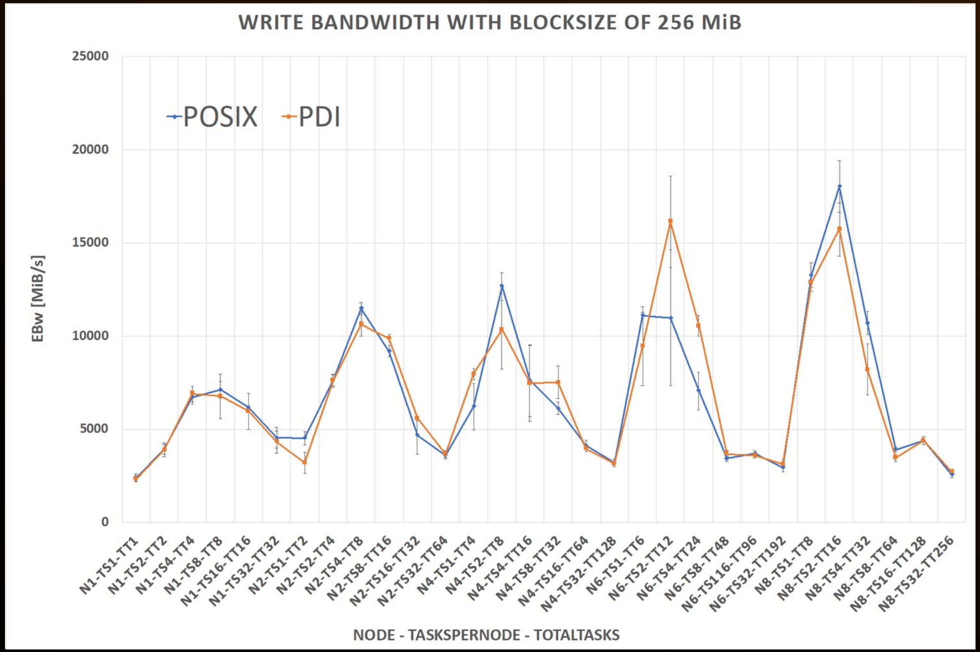
Conclusion

- An API for Data Coupling, Not an IO library
  - A declarative annotation API
  - Multiple plugins for and data processing
    - Describe your IO from YAML
    - Switch to in situ processing or more without even recompiling
- Your turn now!
  - Get the doc: https://pdi.dev/master/
  - Join the fun on slack https://join.slack.pdi.dev/

Perf evaluation: IOR

PDI





IOR IO Benchmark PDI integration

Scaling with small (128k) & large (256M) data blocks on CRESCO6

Francesco Iannone (ENEA)

Preliminary performance evaluation

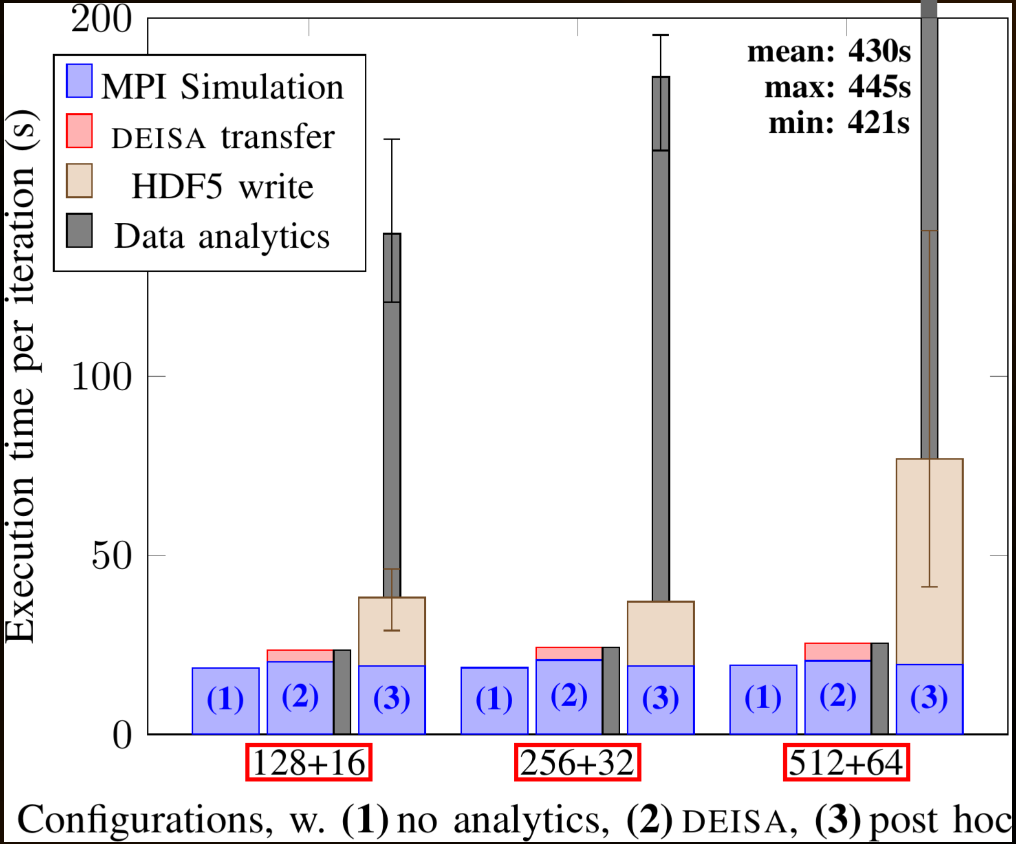
Setup:

- Ruche cluster
  - 192 nodes (2 CPUs 20 cores each, 180 GB)
  - Omni-Path 100 Gbit/s
  - Spectrum Scale GPFS (IOs rate: 9 GB/s)
- Mini-app
  - 2D heat solver
  - Incremental Principal Component Analysis

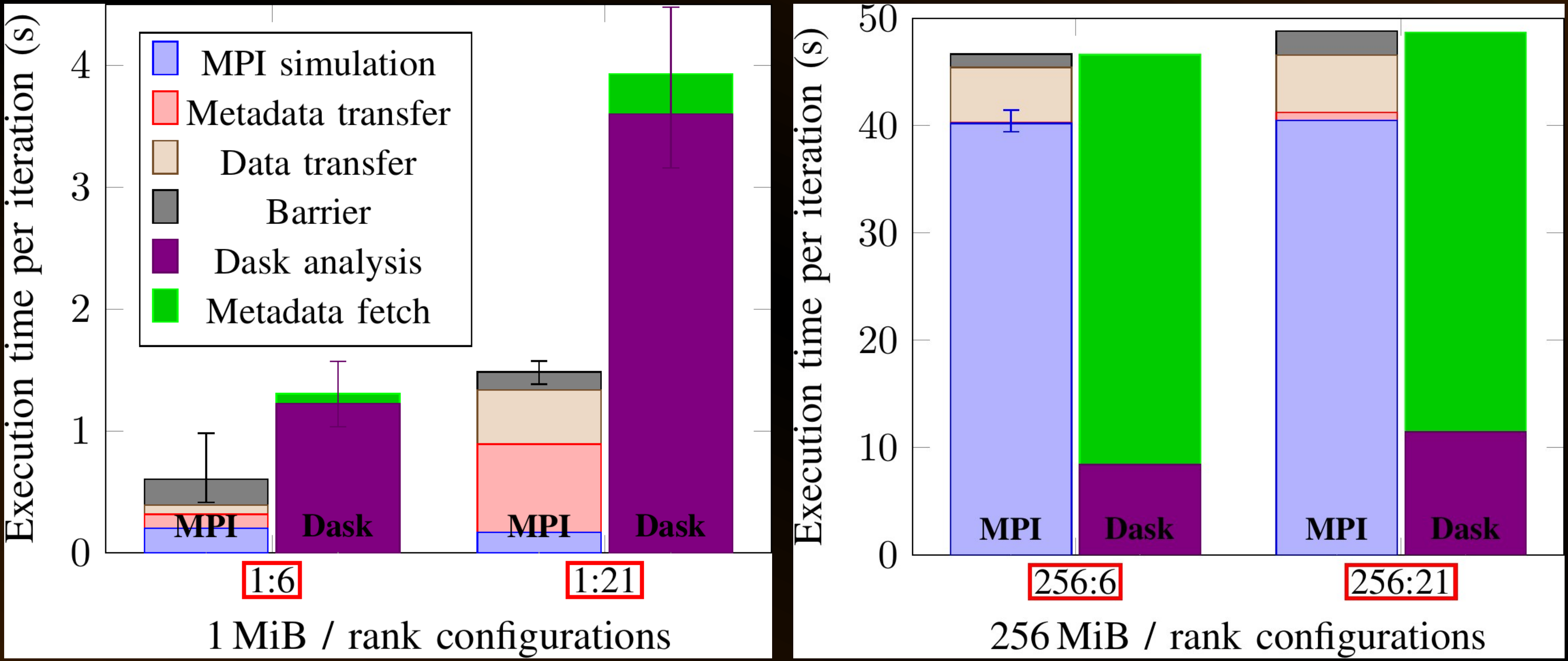
Preliminary performance evaluation

- Weak scaling
  - X + Y cores
  - X cores for MPI simu.
  - Y cores for Dask analytics
- No analytics
- vs. Post-hoc
- vs. DEISA

Configuration	128+16	256+32	512+64
MPI processes	128	256	512
Dask workers	16	32	64
MPI nodes	4	8	16
Dask worker nodes	1	2	4
Global data size	16 GiB	32 GiB	64 GiB
Dask generated tasks	15210	29010	55150



Preliminary performance analysis



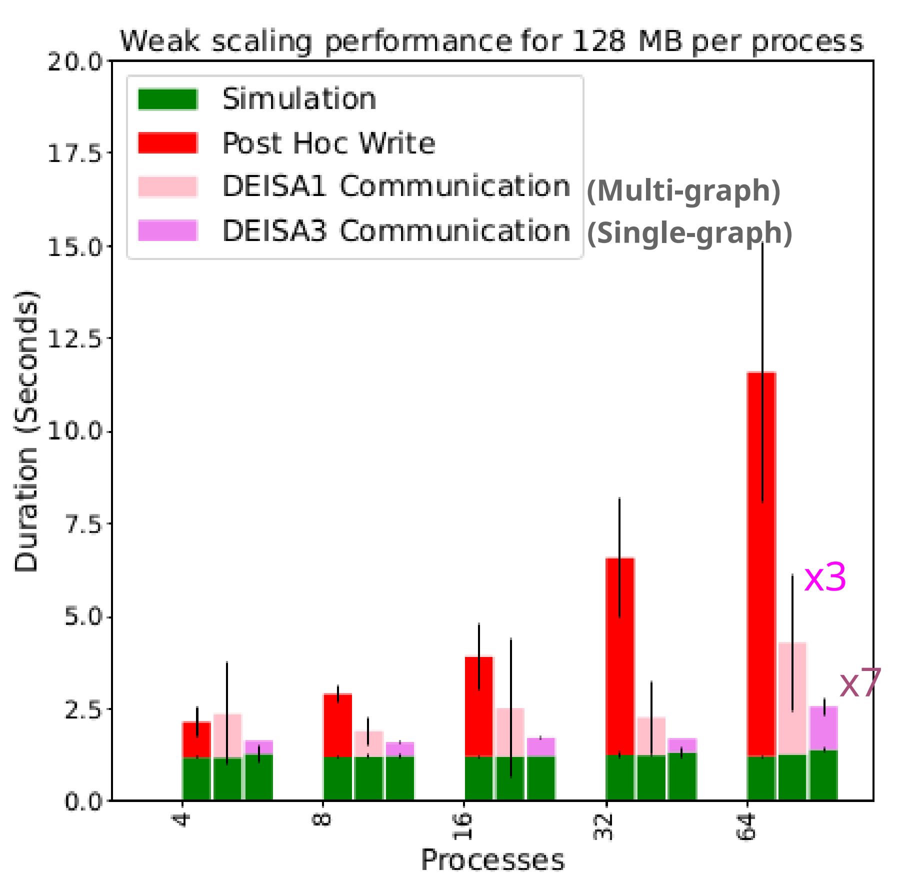
Deisa Performance evaluation

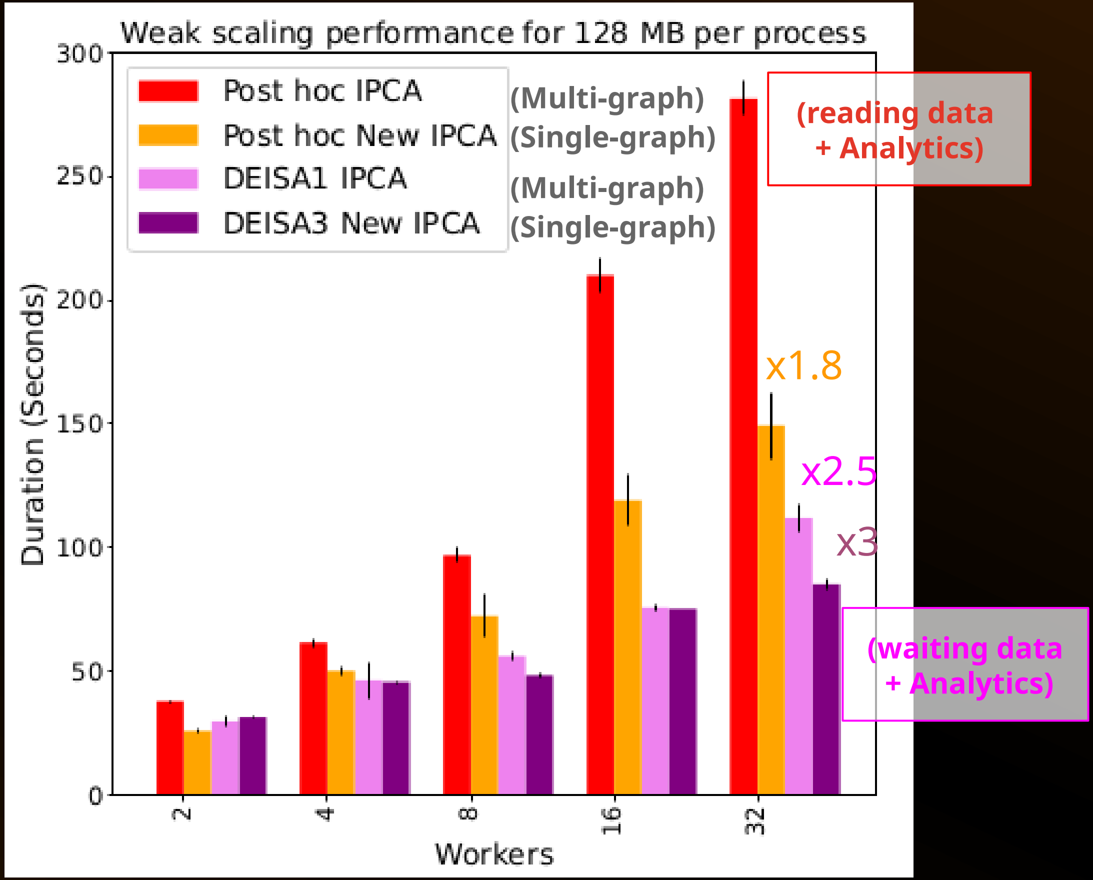
- IRENE supercomputer @ TGCC, France,
- Nodes:
  - 2x24-cores Intel Skylake@2.7GHz
  - 180GB RAM
- InfiniBand network (100Gb/s),
- Scratch disks: 300GB/s transfer rate
- Mini App 2D heat solver

Parameter	Value
Number of runs	3
Number of iterations IPCA	10
Number of iteration Derivative	12
MPI nodes / Dask worker node	2
MPI process / MPI node	2
Dask worker / Dask worker node	2
Thread / Dask worker	24
MPI process / Dask worker	2

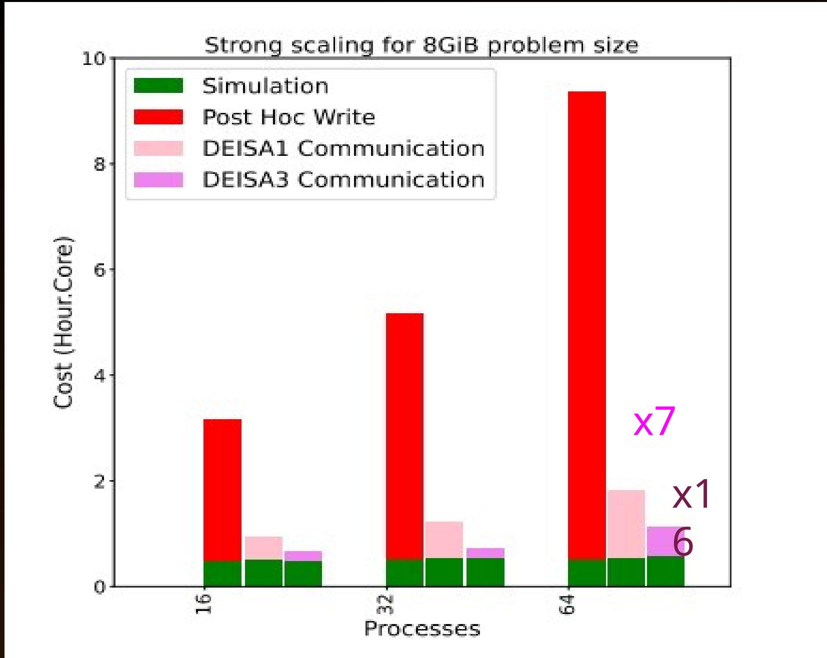
Configuration	XP1:128 MiB	XP1:256 MiB	XP1:512 MiB	XP1:1 GiB
MPI block size	128	256	512	1
Dask chunk size	128	256	512	1
MPI Nodes	[4, 8, 16, 32, 64, 128, 256]			
Dask Nodes	[2, 4, 8, 16, 32, 64, 128]			

DEISA vs Post hoc Weak Scalability

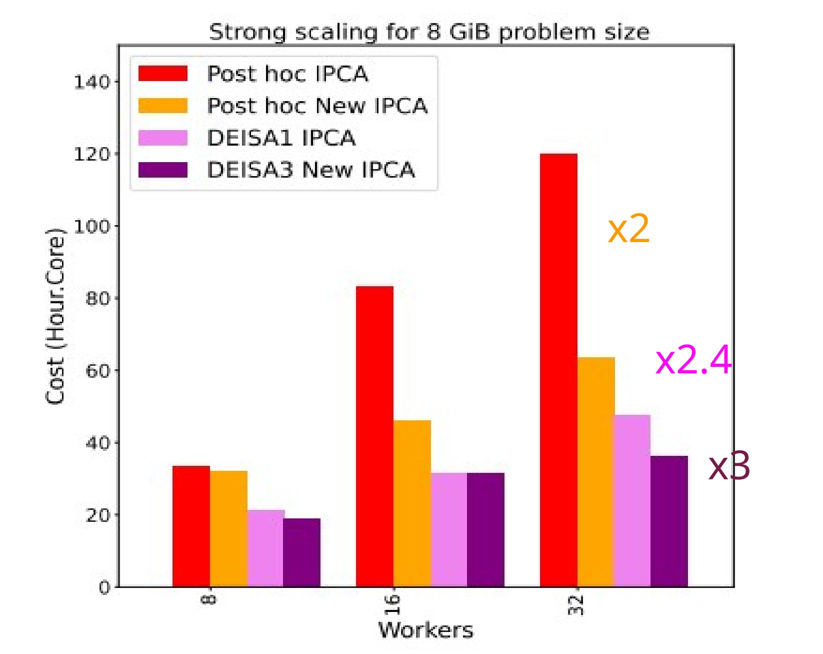




DEISA vs Post hoc efficiency in hour.core

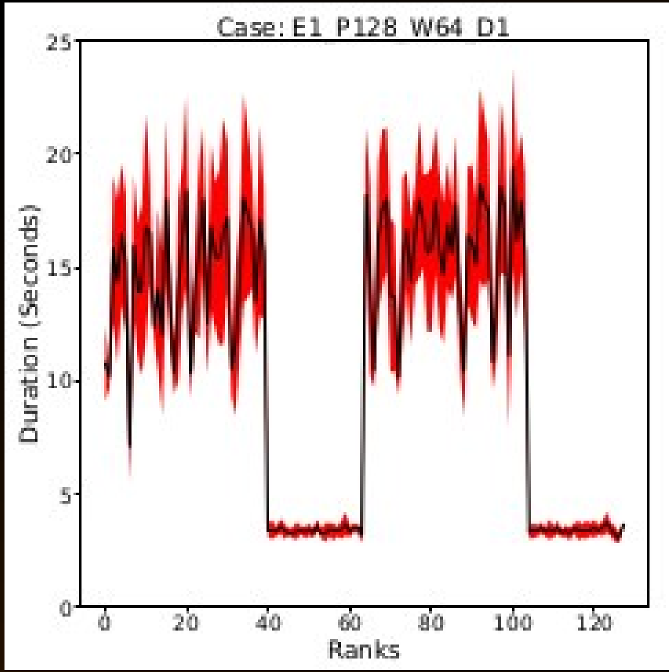


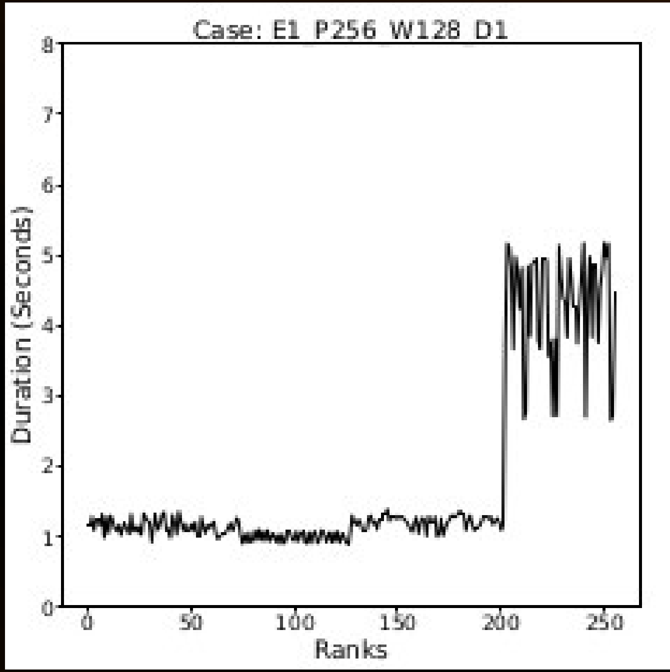
(c) Strong scaling results represented in hour-core for an 8 GiB problem size



(c) Strong scaling results represented in hour-core for a 8 GiB problem size

Deisa scheduler-related jitter





Multi-graph
-lot
metadata
-heartbit=5s

Single-graph
less
metadata
heartbit=∞

60