Deisa
Dask-**e**nabled **in** **s**itu **a**nalytics

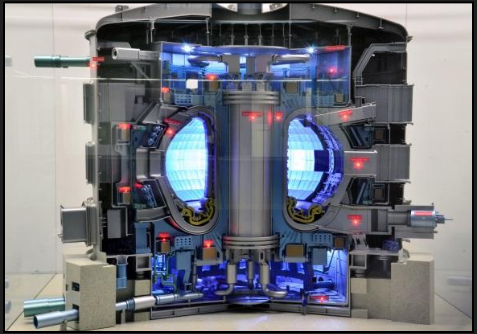
Analyze your MPI Simulation Outputs with Dask

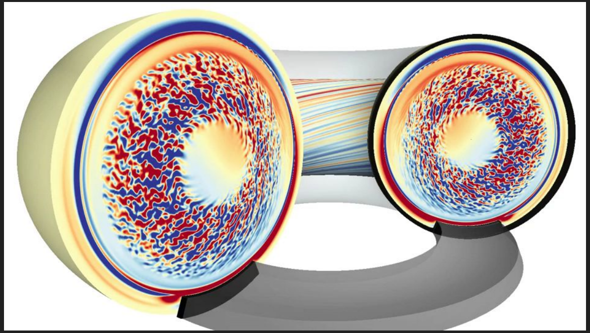
Julien Bigot
Virginie Grandgirard
Amal Gueroudji
Bruno Raffin

HPC Numerical simulations

- Typically numerical simulation
  - Not data analysis
  - Number crunching
- Written in Fortran (now, more & more C++)
  - Using MPI for parallelization over multiple nodes
    - and OpenMP for shared memory parallelism
    - … and GPU
- Iterate over time
- Manipulate very structured data
  - multi-dimensional arrays
  - compute the next state from time-step to time-step

The example of GYSELA


ITER project

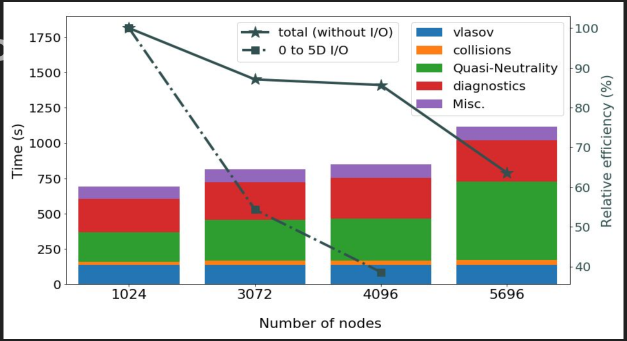

GYSELA simulation

Developed @ CEA/IRFM, lead developer: Virginie Grandgirard

- To optimize performance and minimize risks, each ITER scenario will have to be numerically validated.
- A complete chain of numerical tools will be required, ranging from scale models, which can be used in real time, to first-principles simulations, which are more costly but more reliable.
- Turbulent transport mainly governs confinement in Tokamaks
- Tokamak plasmas weakly collisional □ Kinetic approach mandatory
  - Fusion plasma turbulence is low frequency □ fast gyro-motion is averaged out
  - Gyrokinetic approach: phase space reduction from 6D to 5D

The example of GYSELA

- Gyrokinetic codes require state-of-the-art HPC techniques and must run efficiently on several thousand processors
  - Non-linear 5D simulations (3D in space + 2D in velocity) + multi-scale problem in space and time
- Even more resources required when modelling both core & edge plasmas like GYSELA
- GYSELA = Fortran 90 code with hybrid MPI/OpenMP parallelisation optimized up to 1,460,000 threads
  - Relative efficiency of **85% on more than 1M threads** and **63% o** **1.46M threads** on CEA-HF (AMD EPYC 7763)
- **Intensive use of petascale resources**:
  - ~ 150M hour.core / year
  - (GENCI + PRACE + HPC Fusion resources)



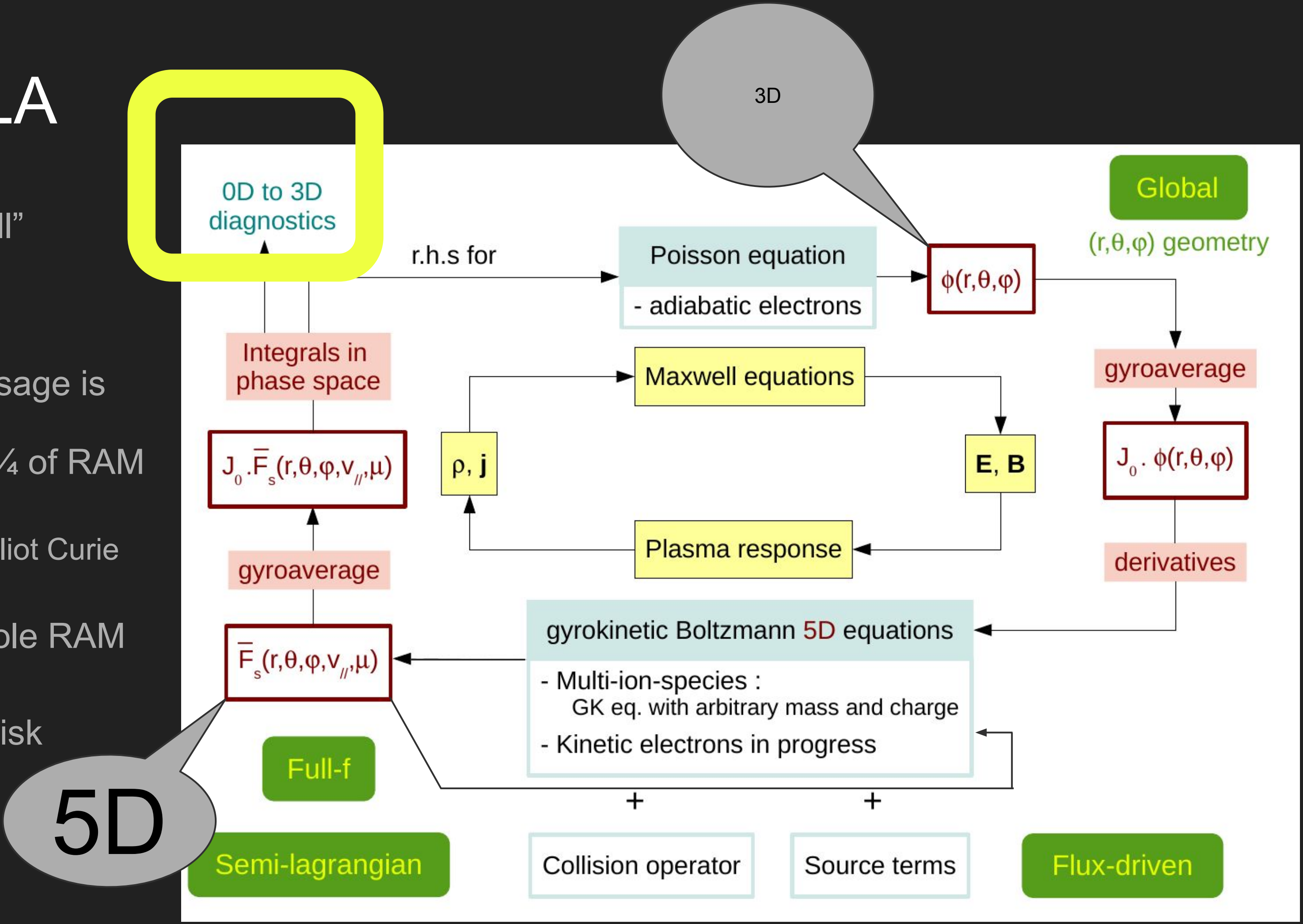
Data in GYSELA

In GYSELA, 3D means "small"

- ~GB or so

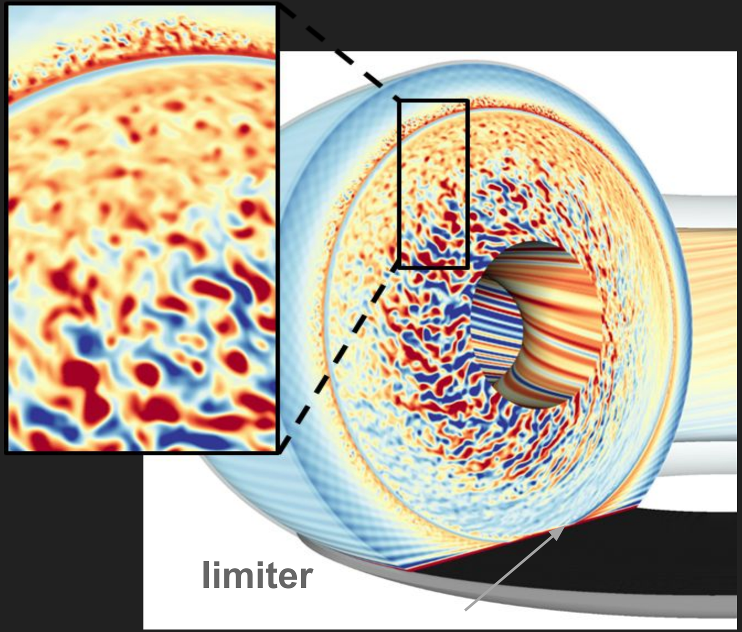
5D is where the real space usage is

- 1 single variable `f` fills ¼ of RAM
  - Of the full cluster
  - That's ~100TB on Joliot Curie
- 2 or 3 copies fill the whole RAM
- You don't write that to disk
  - (or not too often)
  - Diagnostics instead


3D
5D
0D to 3D diagnostics
r.h.s for
Poisson equation
- adiabatic electrons
$\phi(r,\theta,\varphi)$
Global
(r,θ,φ) geometry
Integrals in phase space
Maxwell equations
gyroaverage
$J_0.\bar{F}_s(r,\theta,\varphi,v_{//},\mu)$
ρ, j
E, B
$J_0 . \phi(r,\theta,\varphi)$
Plasma response
gyroaverage
derivatives
gyrokinetic Boltzmann 5D equations
$\bar{F}_s(r,\theta,\varphi,v_{//},\mu)$
- Multi-ion-species : GK eq. with arbitrary mass and charge
- Kinetic electrons in progress
Full-f
+
+
Semi-lagrangian
Collision operator
Source terms
Flux-driven

Diagnostics in GYSELA

- In the code (in Fortran)
  - Reduce data from 5D to 3D, 2D, 1D, 0D…
  - To a single node each
- Write the result to files
  - HDF5
- Analyze the files post hoc
  - In python
  - Interactively
  - FFTs, more reductions, combining data
  - generating graphs, images, videos, …



In 2020, a new diagnostic?

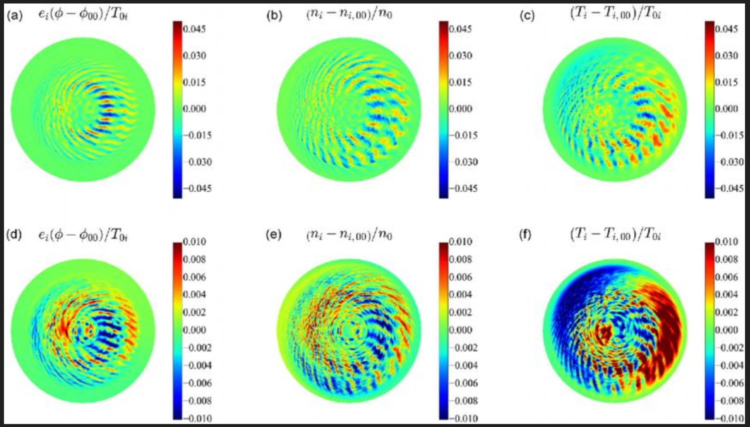
Principal Component Analysis computation on 5D distribution function

- Yuuichi Asahi et al.
- Done on GT5D

Asahi, Yuuichi & Fujii, Keisuke & Heim, Dennis & Maeyama, Shinya & Garbet, Xavier & Grandgirard, Virginie & Sarazin, Yanick & Dif-Pradalier, G. & Idomura, Yasuhiro & Yagi, Masatoshi. (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.

Hard to implement in Fortran+MPI+OpenMP

- Parallel PCA already available in Scikit-learn
- => Let's reuse it!



Post hoc data analytics with python

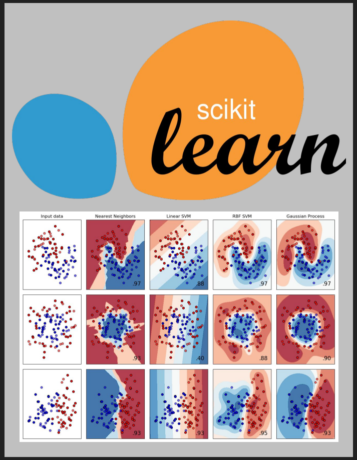
```python
from sklearn.decomposition import IncrementalPCA
import yaml, json
import h5py
# load the simulation configuration
simu = yaml.load(open('simulation.yml'))
# Load data from HDF5
gtemp = h5py.File('data.hdf5',mode='r')['gtemp']
# process each time-step independently
for step in range(0, simu['timesteps']):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

Requires a single node computer with ~100TB RAM

Sequential python usin kit-learn for PCA

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_)
```



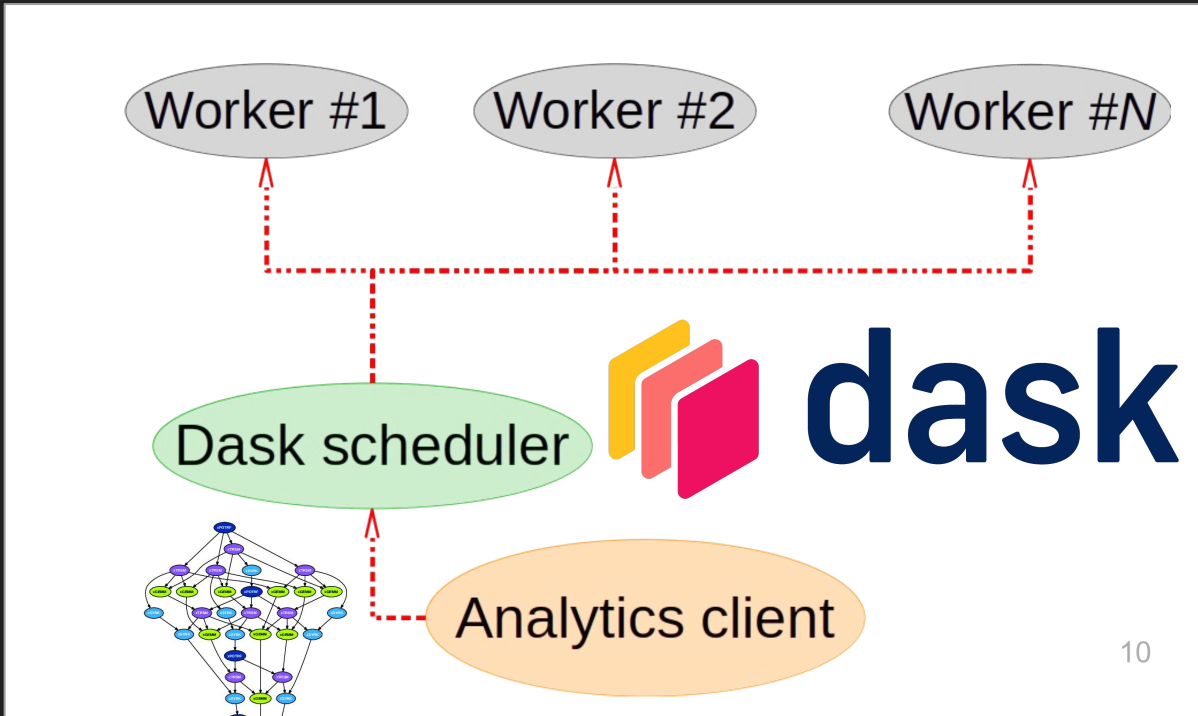
Dask distributed?

A scheduler/workers (+client) model to run work (each on its own process/node)

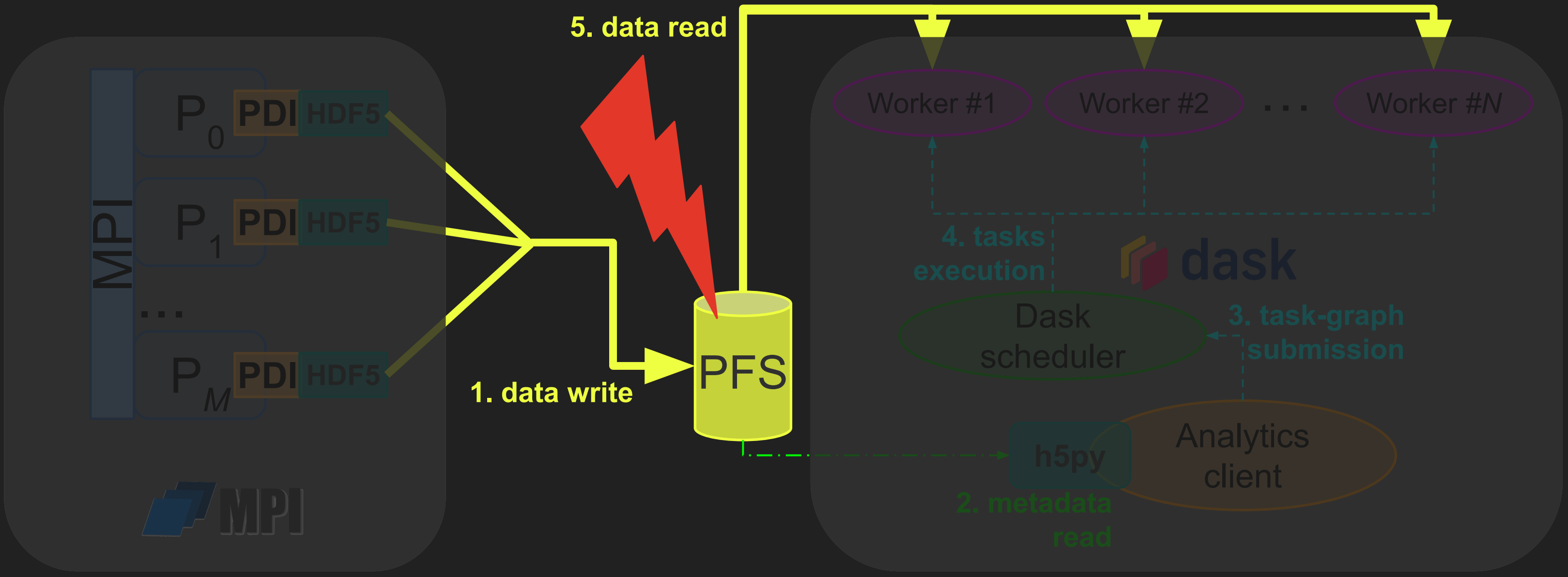
A task-based model to describe work

Many tools ported to dask for ease of use

- Numpy / SciPy
- Scikit-learn
- Pandas
- ...



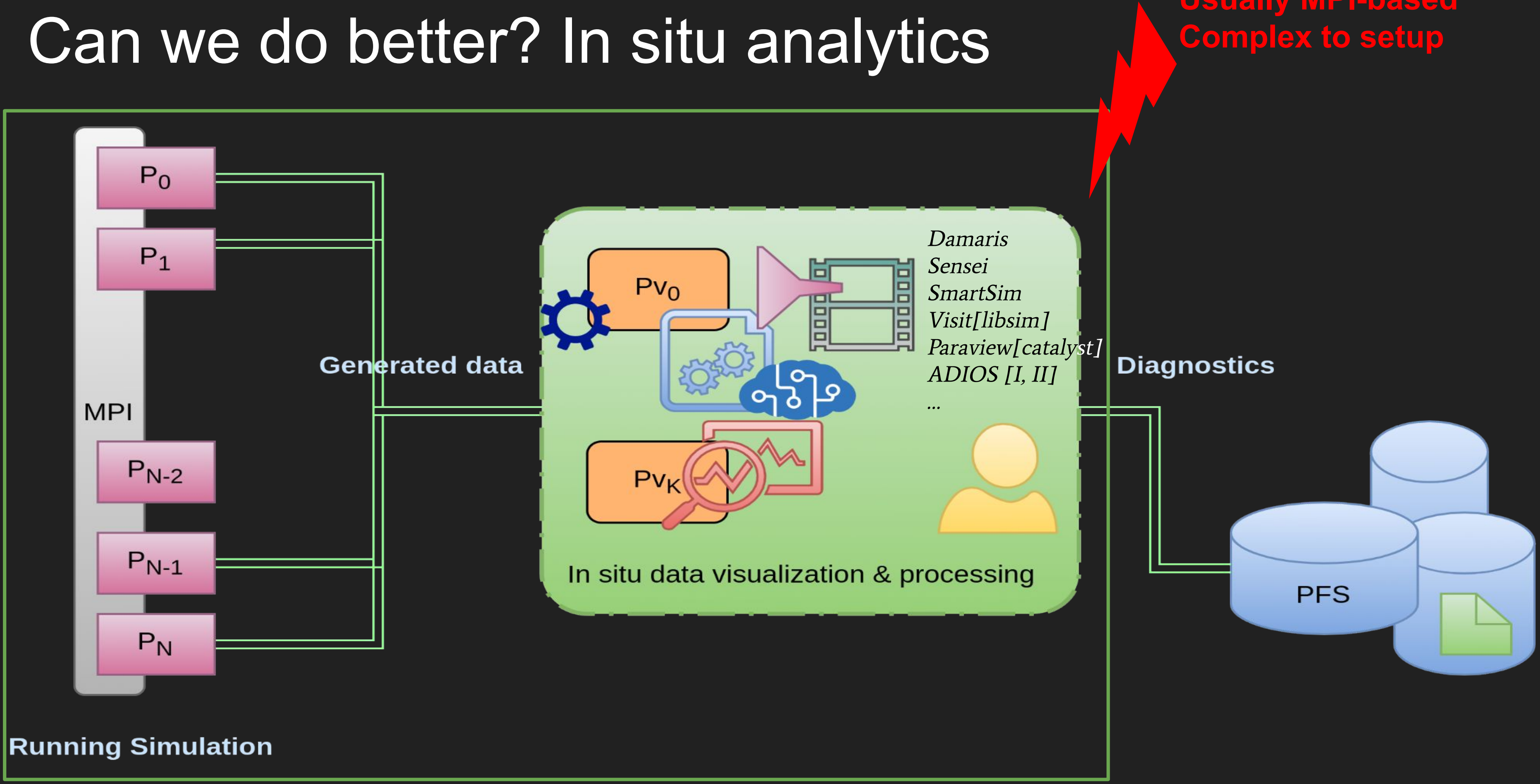
Dask for post hoc analytics



- 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 IO performance is still an issue

Can we do better? In situ analytics

Usually MPI-based
Complex to setup



Can we do even better? Deisa!

General context

- Python analytics are nice and many tools are available :)
  - Dask offers a great parallel task-based programming model :)
  - But file-system performance is a bottleneck :/
- In situ analytics solve performance issues :)
  - Typically close to the application (MPI) programming mode
  - MPI is not well suited to writing data analytics :/

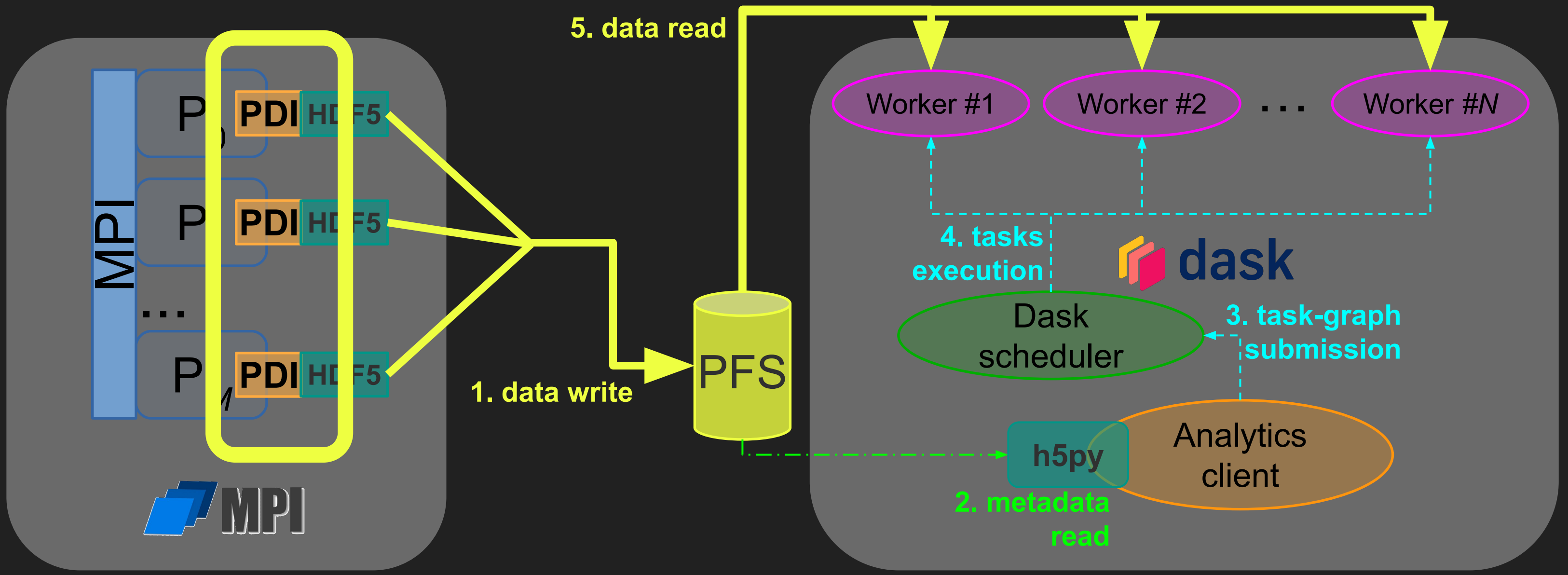
Let's combine these!

Dask-**E**nabled **I**n **S**itu **A**nalytics

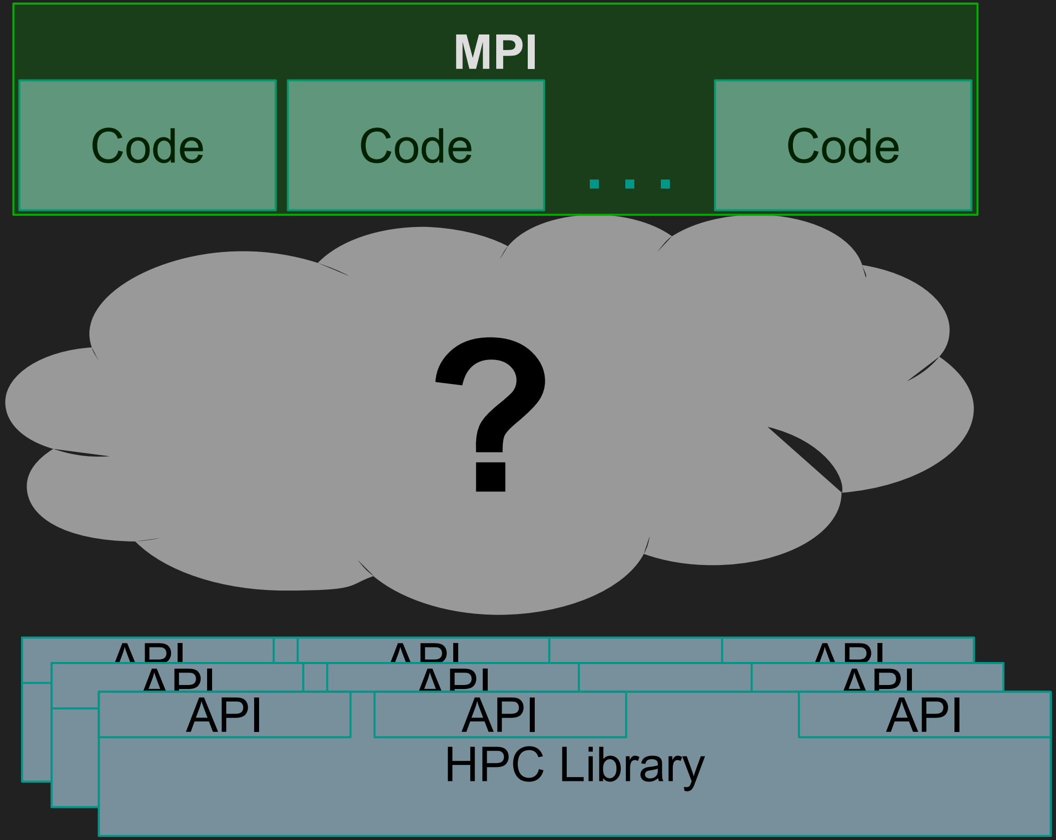
Amal Gueroudji. Distributed Task-Based In Situ Data Analytics for High-Performance Simulations. Université Grenoble Alpes [2020-..], 2023. English.

- PhD work by Amal Gueroudji, advised by J. Bigot & B. Raffin

Dask for post hoc analytics



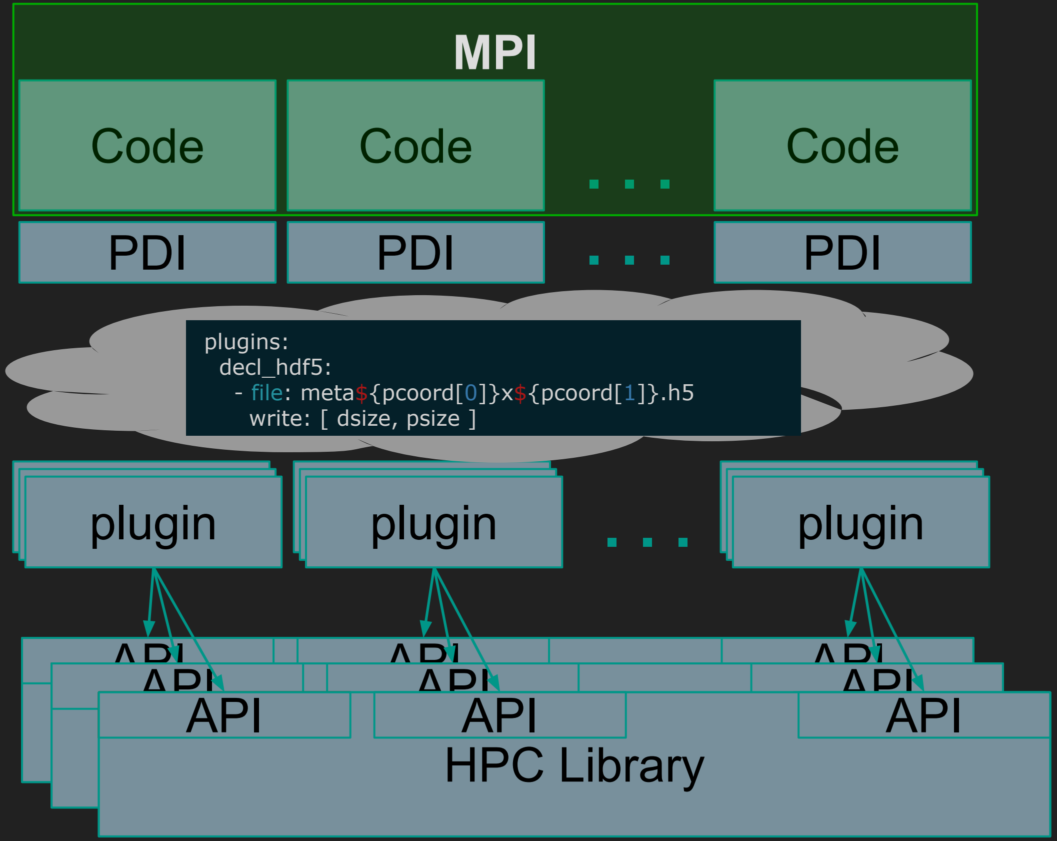
What is PDI?



PDI annotations: a purely declarative API

Plugins for access to existing libraries

What is PDI?



```
plugins:
  decl_hdf5:
    - file: meta${pcoord[0]}x${pcoord[1]}.h5
      write: [ dsize, psize ]
```

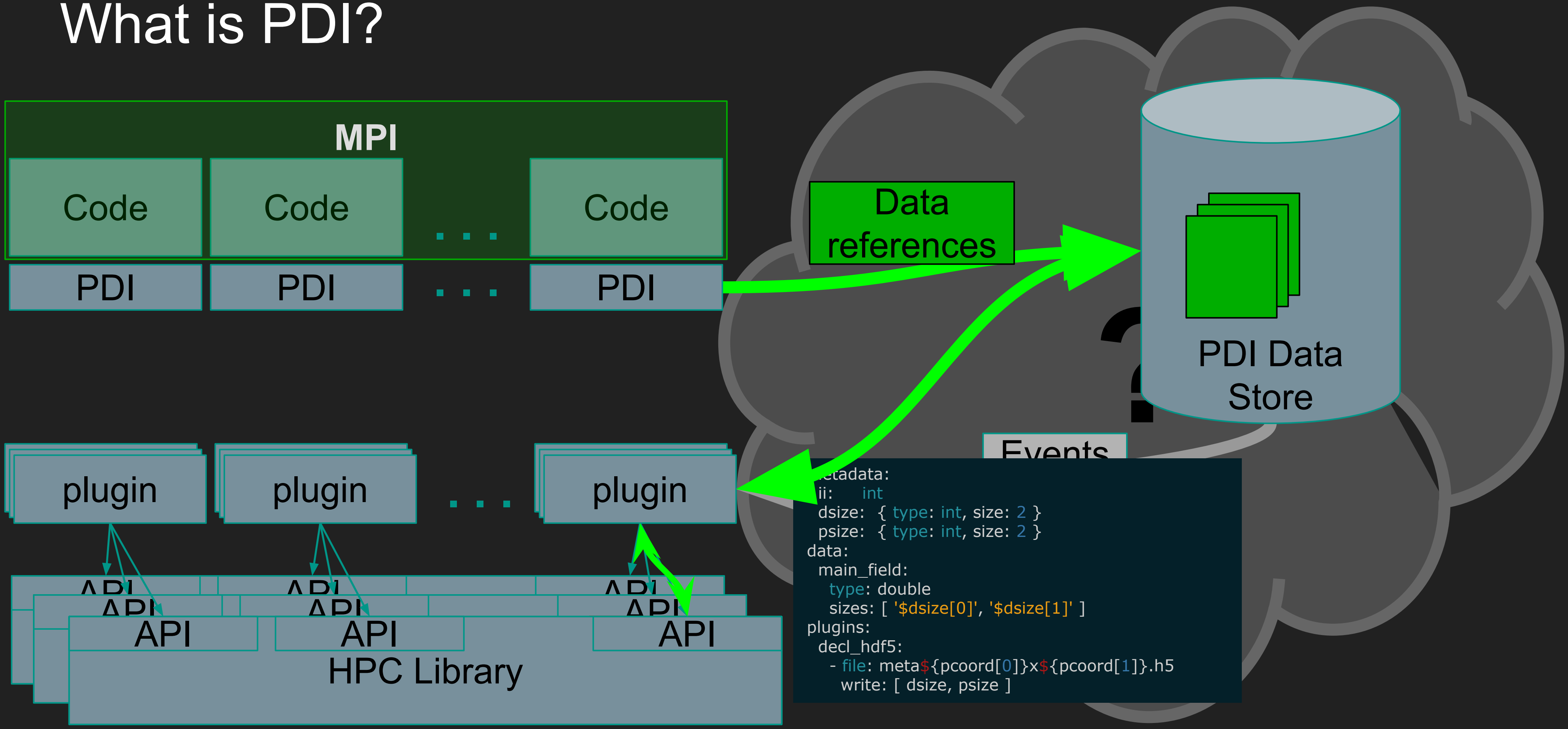
PDI annotations: a purely declarative API

PDI YAML spec. tree:

- What to do with data

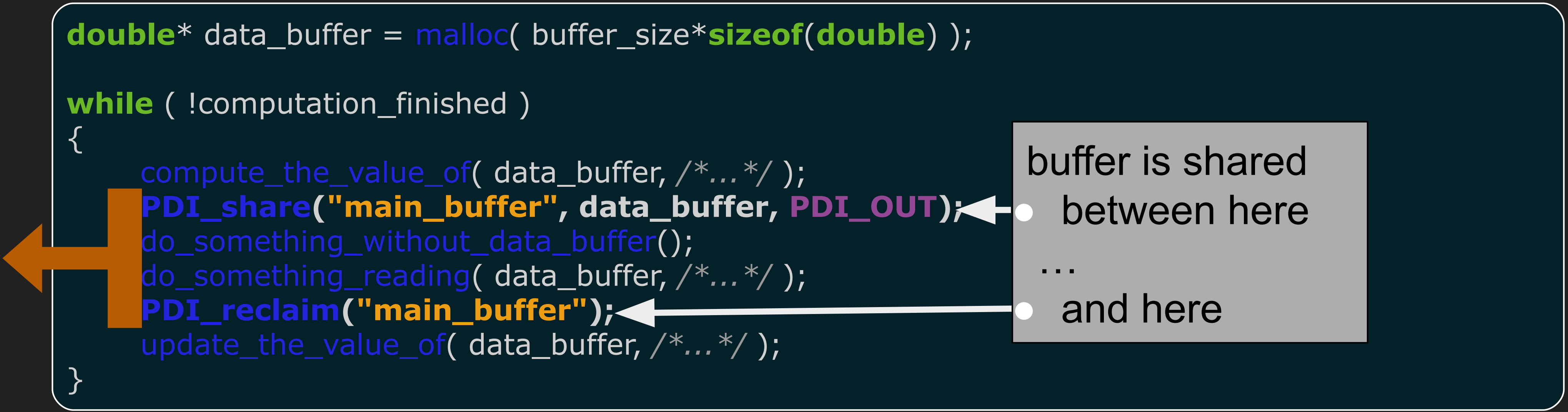
Plugins for access to existing libraries

What is PDI?



```
etadata:
ii:     int
 dsize:  { type: int, size: 2 }
 psize:  { type: int, size: 2 }
data:
 main_field:
   type: double
   sizes: [ '$dsize[0]', '$dsize[1]' ]
plugins:
 decl_hdf5:
   - file: meta${pcoord[0]}x${pcoord[1]}.h5
     write: [ dsize, psize ]
```

PDI: Annotation API usage



```
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, /*...*/ );
}
```

- 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)

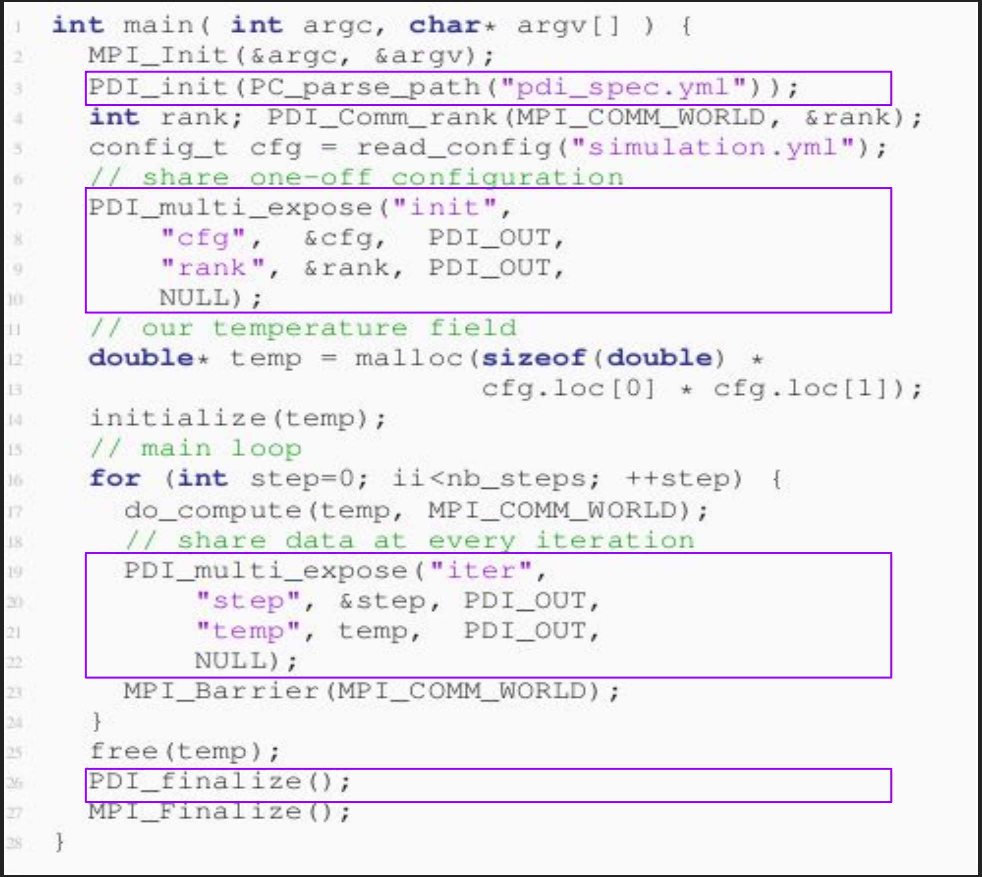
Decl’HDF5: the YAML

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

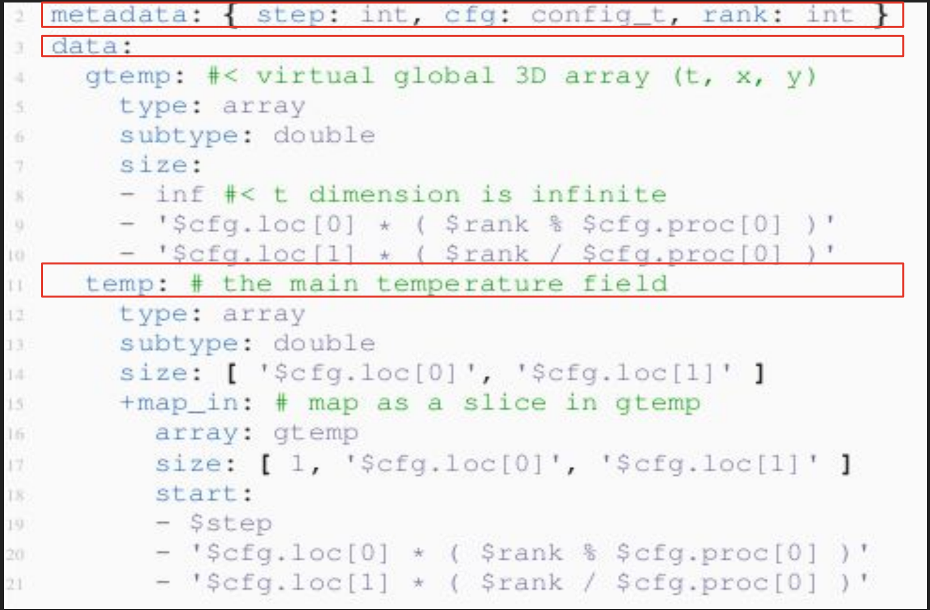
- Write data in the HDF5 format
- Heavily relies on
  - $-expressions
  - default configuration values

- Makes
  - Simple things easy
  - Complex things possible

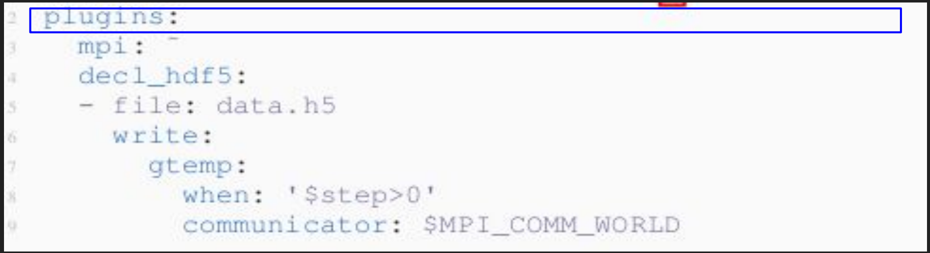
PDI for PCA: Simulation instrumentation



```c
int main( int argc, char* argv[] ) {
  MPI_Init(&argc, &argv);
  PDI_init(PC_parse_path("pdi_spec.yml"));
  int rank; PDI_Comm_rank(MPI_COMM_WORLD, &rank);
  config_t cfg = read_config("simulation.yml");
  // share one-off configuration
  PDI_multi_expose("init",
      "cfg",  &cfg,  PDI_OUT,
      "rank", &rank, PDI_OUT,
      NULL);
  // our temperature field
  double* temp = malloc(sizeof(double) *
                        cfg.loc[0] * cfg.loc[1]);
  initialize(temp);
  // main loop
  for (int step=0; ii<nb_steps; ++step) {
    do_compute(temp, MPI_COMM_WORLD);
    // share data at every iteration
    PDI_multi_expose("iter",
        "step", &step, PDI_OUT,
        "temp", temp,  PDI_OUT,
        NULL);
    MPI_Barrier(MPI_COMM_WORLD);
  }
  free(temp);
  PDI_finalize();
  MPI_Finalize();
}
```

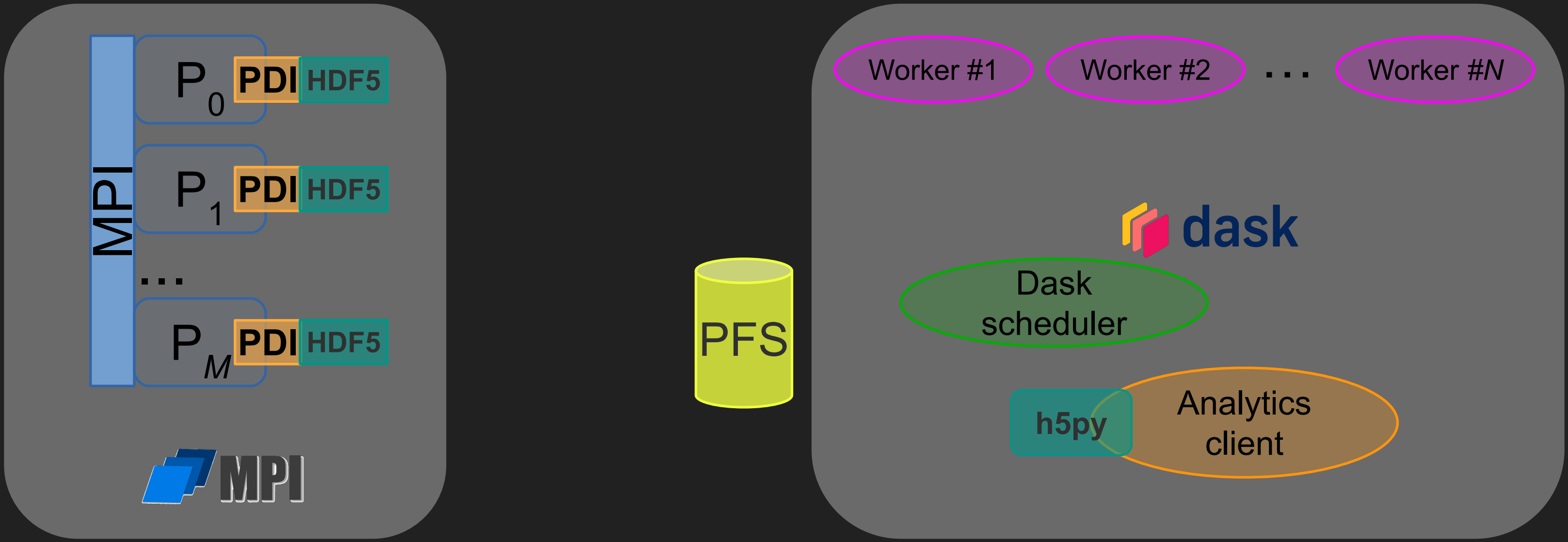


```yaml
metadata: { step: int, cfg: config_t, rank: int }
data:
  gtemp: #< virtual global 3D array (t, x, y)
    type: array
    subtype: double
    size:
    - inf #< t dimension is infinite
    - '$cfg.loc[0] * ( $rank % $cfg.proc[0] )'
    - '$cfg.loc[1] * ( $rank / $cfg.proc[0] )'
  temp: # the main temperature field
    type: array
    subtype: double
    size: [ '$cfg.loc[0]', '$cfg.loc[1]' ]
    +map_in: # map as a slice in gtemp
      array: gtemp
      size: [ 1, '$cfg.loc[0]', '$cfg.loc[1]' ]
      start:
      - $step
      - '$cfg.loc[0] * ( $rank % $cfg.proc[0] )'
      - '$cfg.loc[1] * ( $rank / $cfg.proc[0] )'
```

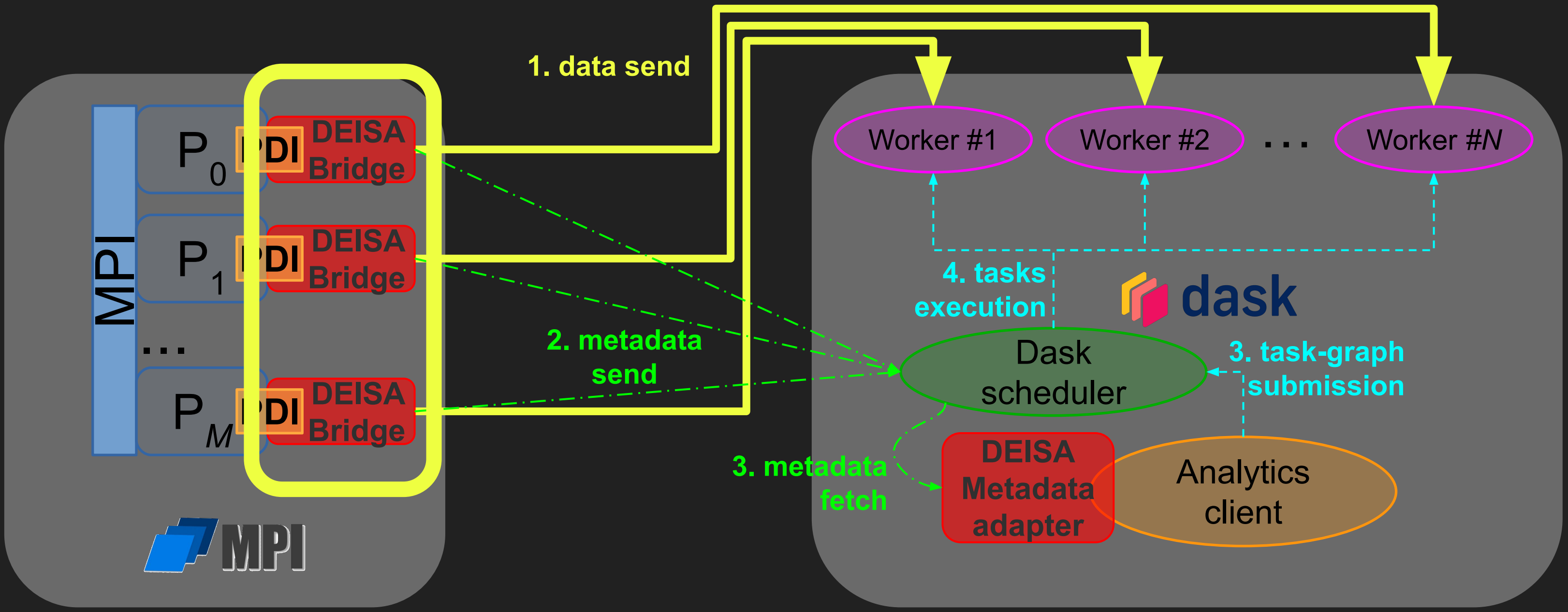


```yaml
plugins:
  mpi: ~
  decl_hdf5:
  - file: data.h5
    write:
      gtemp:
        when: '$step>0'
        communicator: $MPI_COMM_WORLD
```

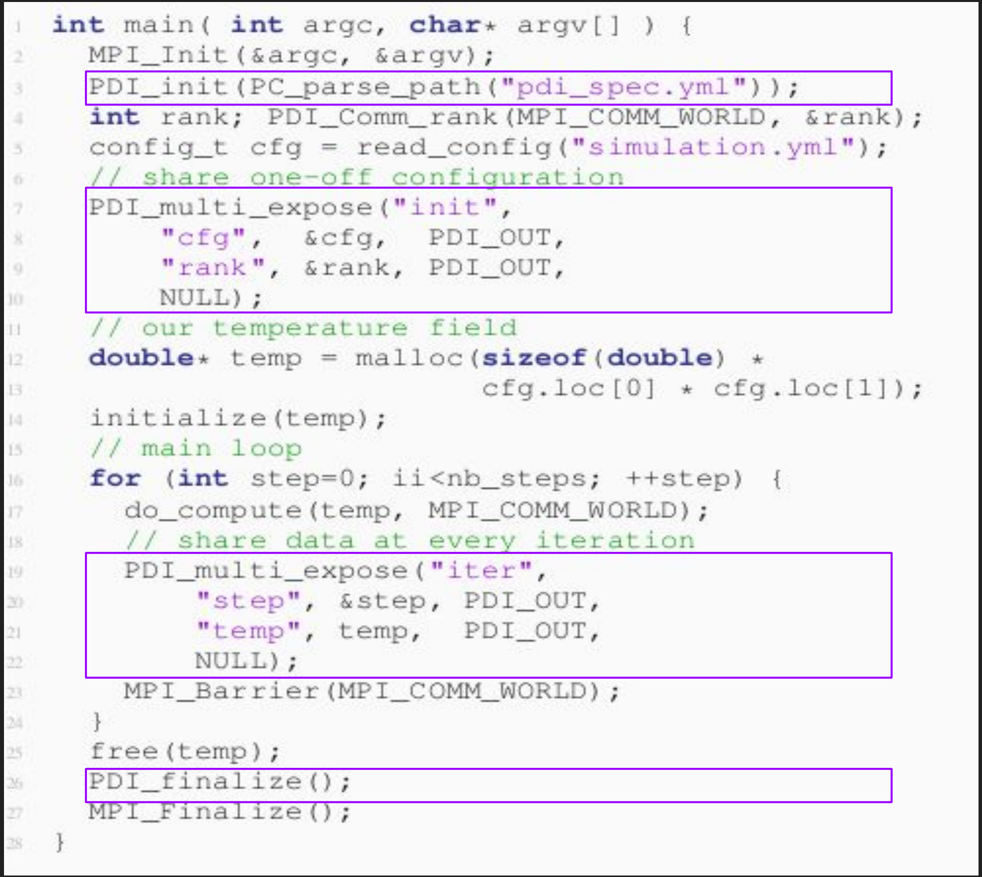
Dask for post hoc analytics



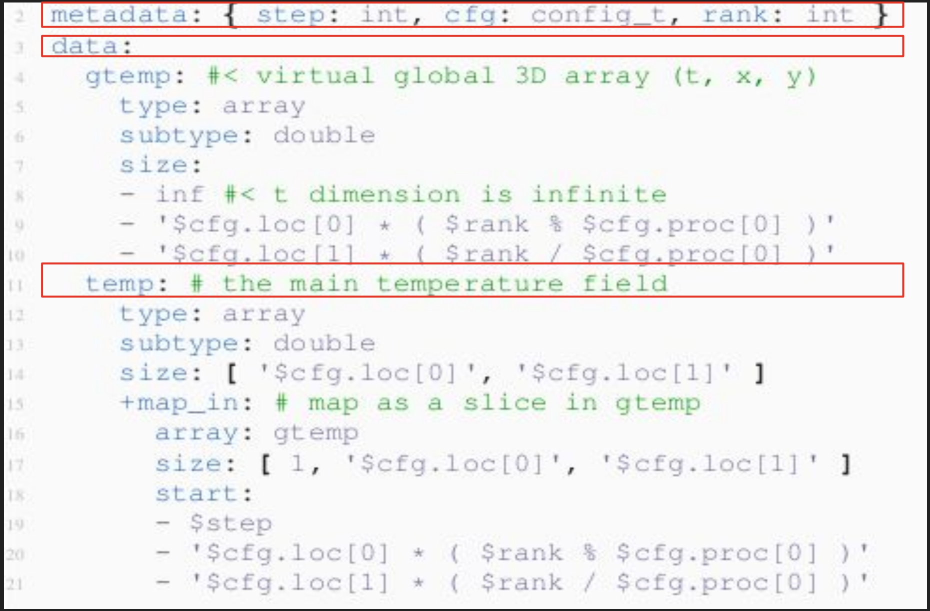
Introducing Deisa v1 for in situ analytics



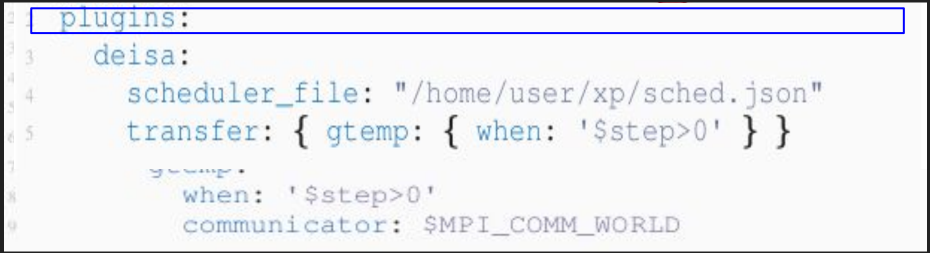
Deisa: Simulation instrumentation



```c
int main( int argc, char* argv[] ) {
  MPI_Init(&argc, &argv);
  PDI_init(PC_parse_path("pdi_spec.yml"));
  int rank; PDI_Comm_rank(MPI_COMM_WORLD, &rank);
  config_t cfg = read_config("simulation.yml");
  // share one-off configuration
  PDI_multi_expose("init",
      "cfg",  &cfg,  PDI_OUT,
      "rank", &rank, PDI_OUT,
      NULL);
  // our temperature field
  double* temp = malloc(sizeof(double) *
                        cfg.loc[0] * cfg.loc[1]);
  initialize(temp);
  // main loop
  for (int step=0; ii<nb_steps; ++step) {
    do_compute(temp, MPI_COMM_WORLD);
    // share data at every iteration
    PDI_multi_expose("iter",
        "step", &step, PDI_OUT,
        "temp", temp,  PDI_OUT,
        NULL);
    MPI_Barrier(MPI_COMM_WORLD);
  }
  free(temp);
  PDI_finalize();
  MPI_Finalize();
}
```

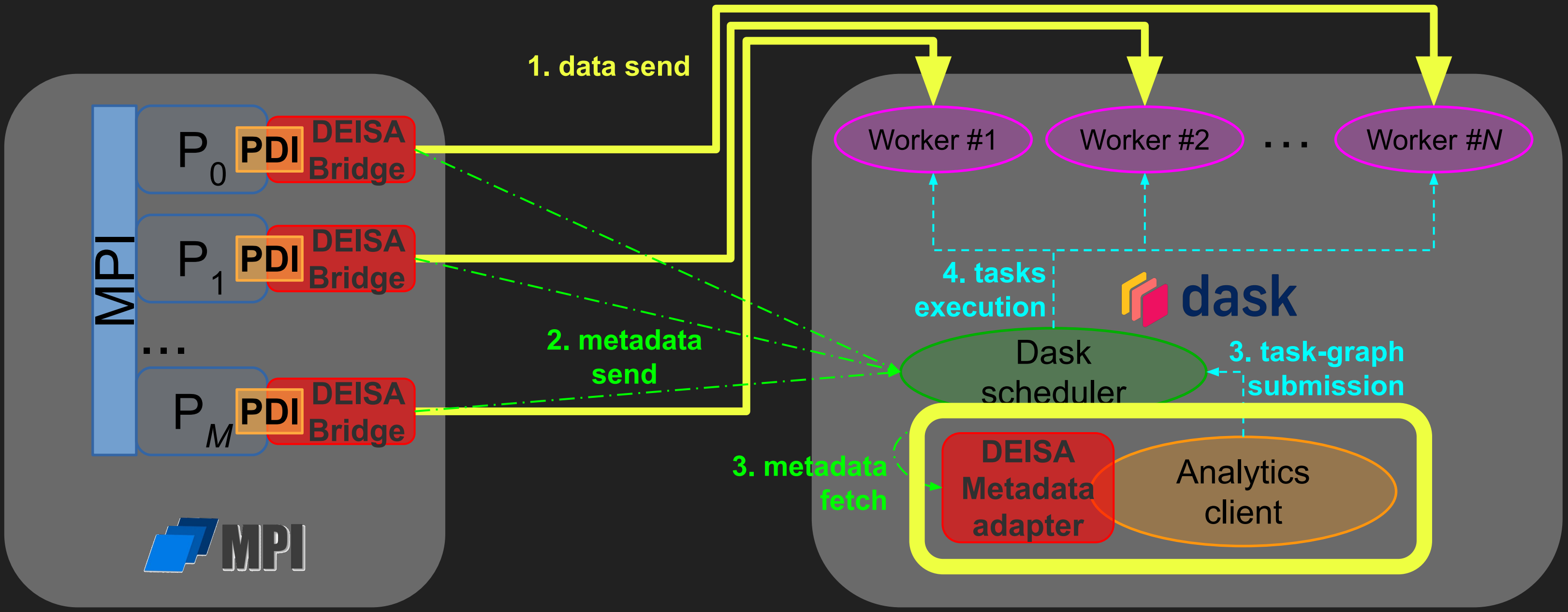


```yaml
metadata: { step: int, cfg: config_t, rank: int }
data:
  gtemp: #< virtual global 3D array (t, x, y)
    type: array
    subtype: double
    size:
    - inf #< t dimension is infinite
    - '$cfg.loc[0] * ( $rank % $cfg.proc[0] )'
    - '$cfg.loc[1] * ( $rank / $cfg.proc[0] )'
  temp: # the main temperature field
    type: array
    subtype: double
    size: [ '$cfg.loc[0]', '$cfg.loc[1]' ]
    +map_in: # map as a slice in gtemp
      array: gtemp
      size: [ 1, '$cfg.loc[0]', '$cfg.loc[1]' ]
      start:
      - $step
      - '$cfg.loc[0] * ( $rank % $cfg.proc[0] )'
      - '$cfg.loc[1] * ( $rank / $cfg.proc[0] )'
```

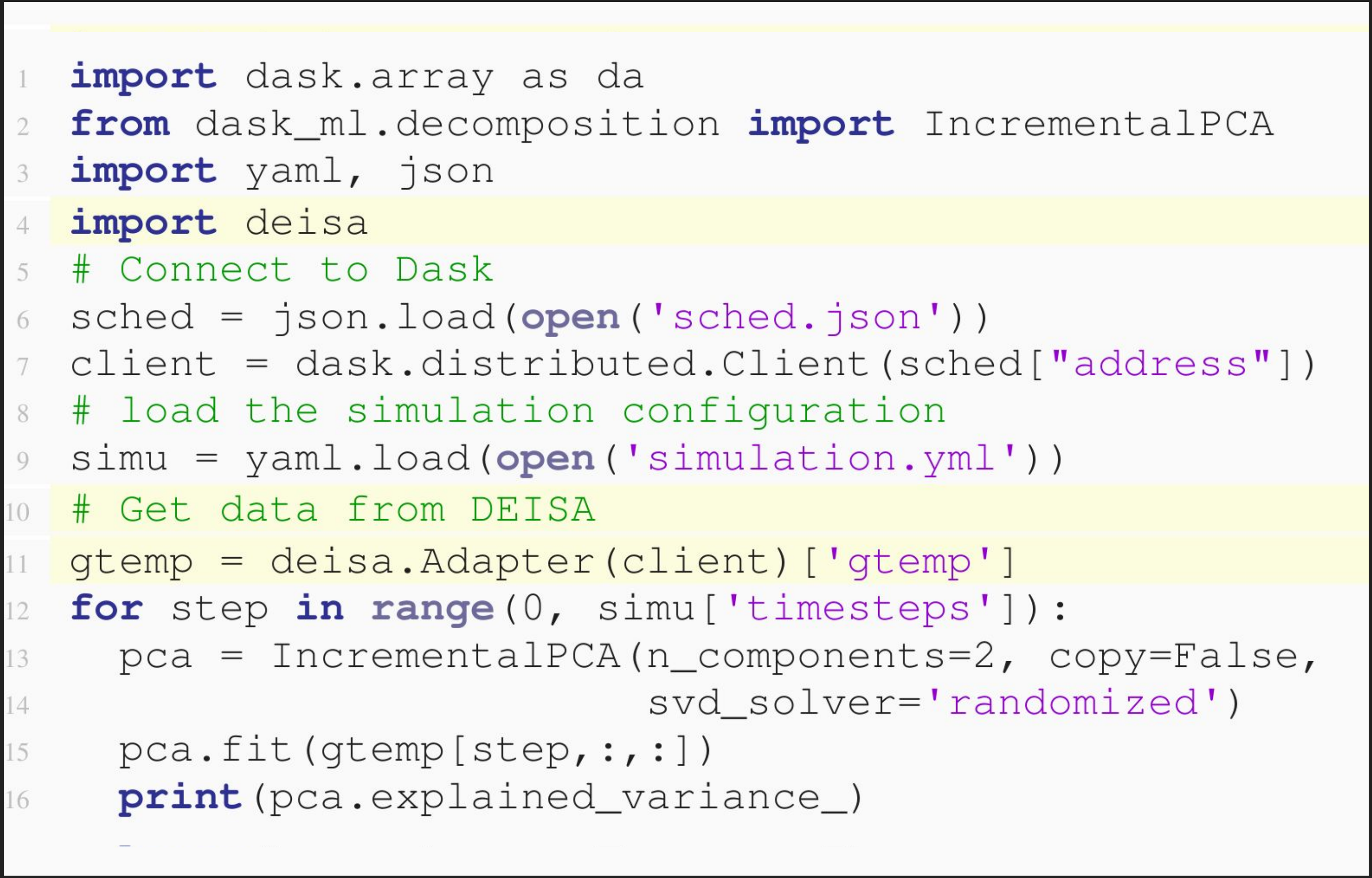


```yaml
plugins:
  deisa:
    scheduler_file: "/home/user/xp/sched.json"
    transfer: { gtemp: { when: '$step>0' } }
        when: '$step>0'
        communicator: $MPI_COMM_WORLD
```

Introducing Deisa v1 for in situ analytics



Deisa: The analytics code



```python
import dask.array as da
from dask_ml.decomposition import IncrementalPCA
import yaml, json
import deisa
# 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'))
# Get data from DEISA
gtemp = deisa.Adapter(client)['gtemp']
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_)
```

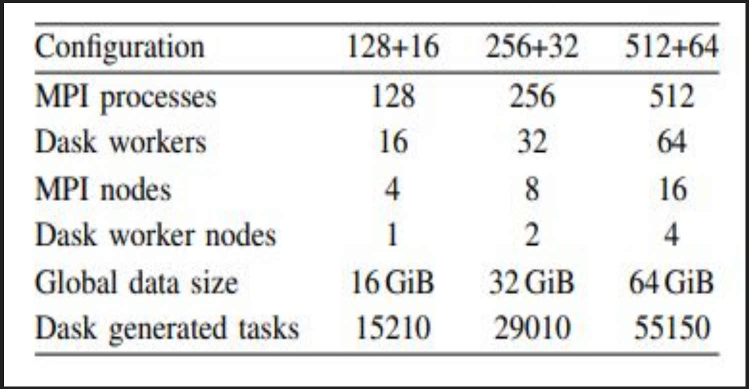
Preliminary performance evaluation

Setup:

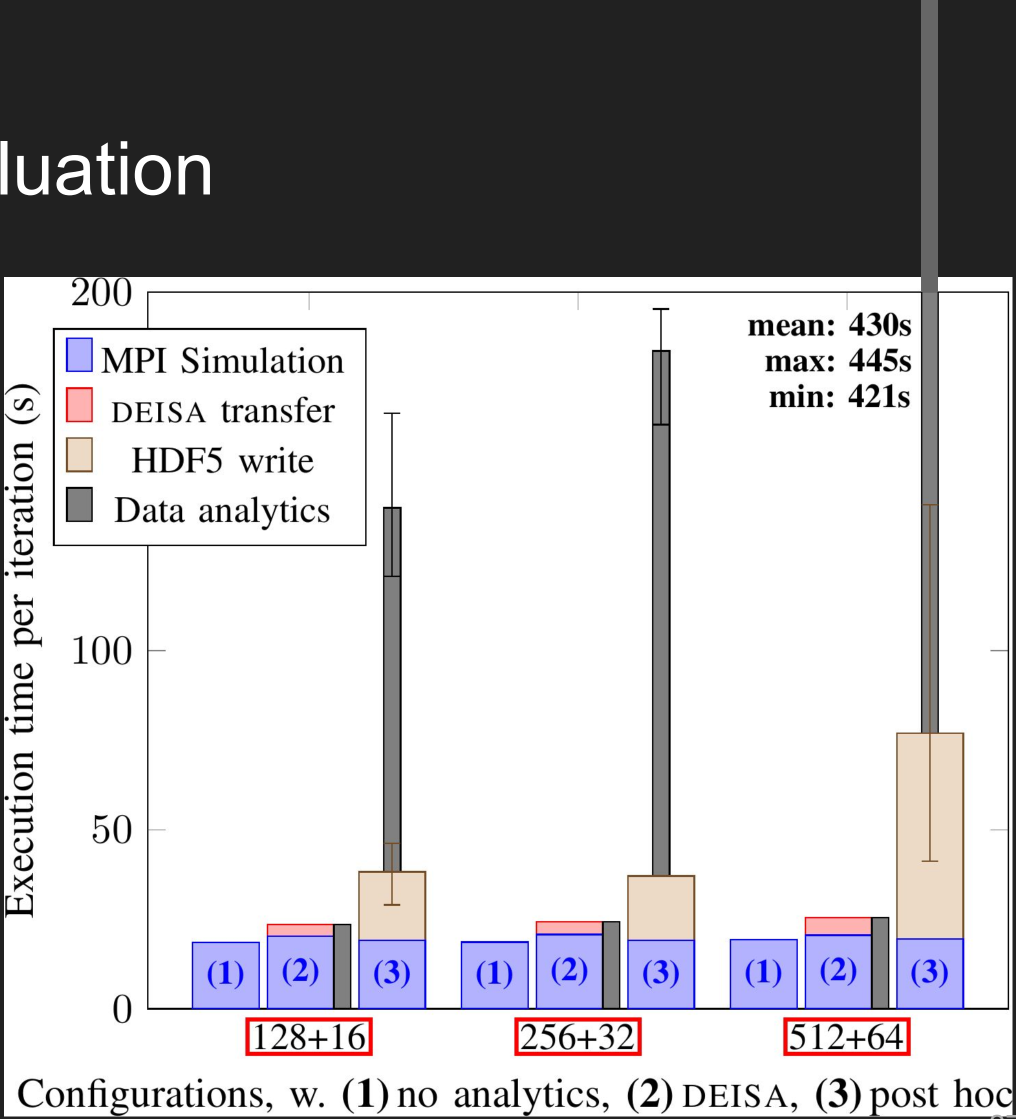
- Ruche cluster
  - 192 nodes (2 CPUs 20 cores each, 180 GB)
  - Omni-Path 100 Gbit/s
  - Spectrum Scale GPFS (IOs rate: 9 GB/s)
- Mini-app
  - 2D heat solver
  - Incremental Principal Component Analysis

Preliminary performance evaluation

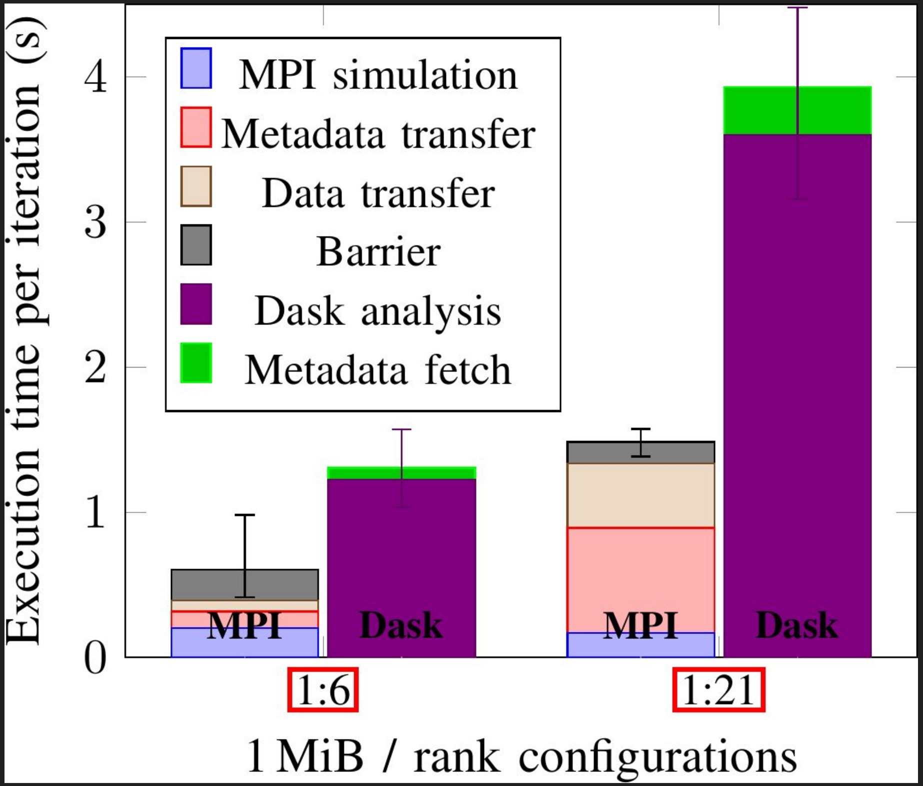
- Weak scaling
  - X + Y cores
  - X cores for MPI simu.
  - Y cores for Dask analytics
- No analytics
- vs. Post-hoc
- vs. DEISA

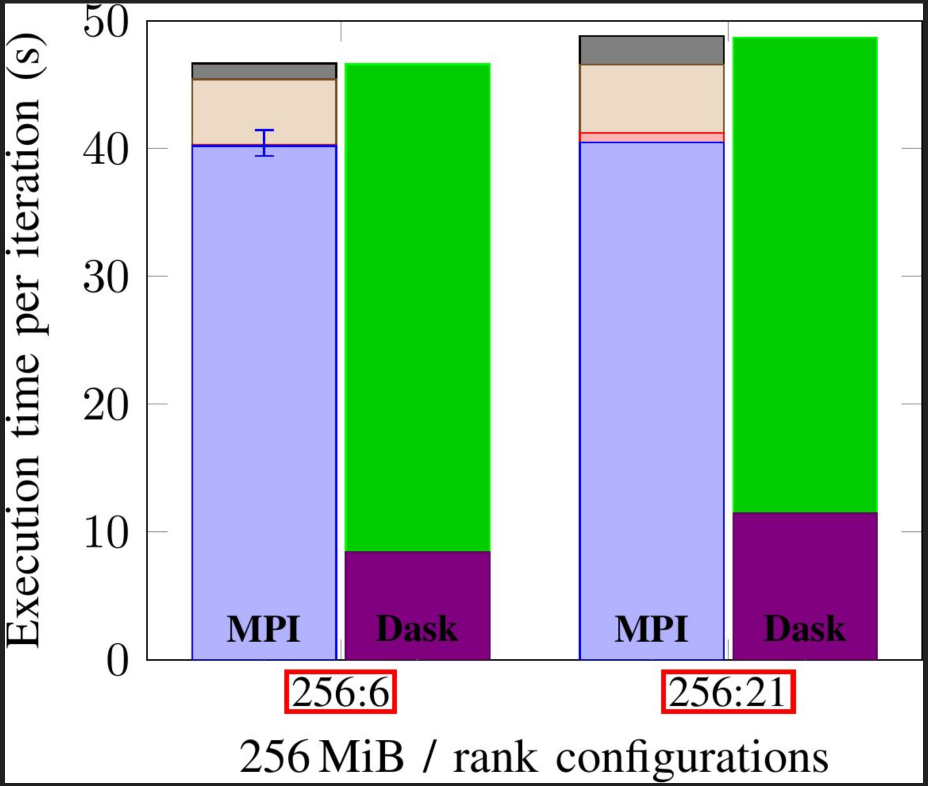


Configuration	128+16	256+32	512+64
MPI processes	128	256	512
Dask workers	16	32	64
MPI nodes	4	8	16
Dask worker nodes	1	2	4
Global data size	16 GiB	32 GiB	64 GiB
Dask generated tasks	15210	29010	55150

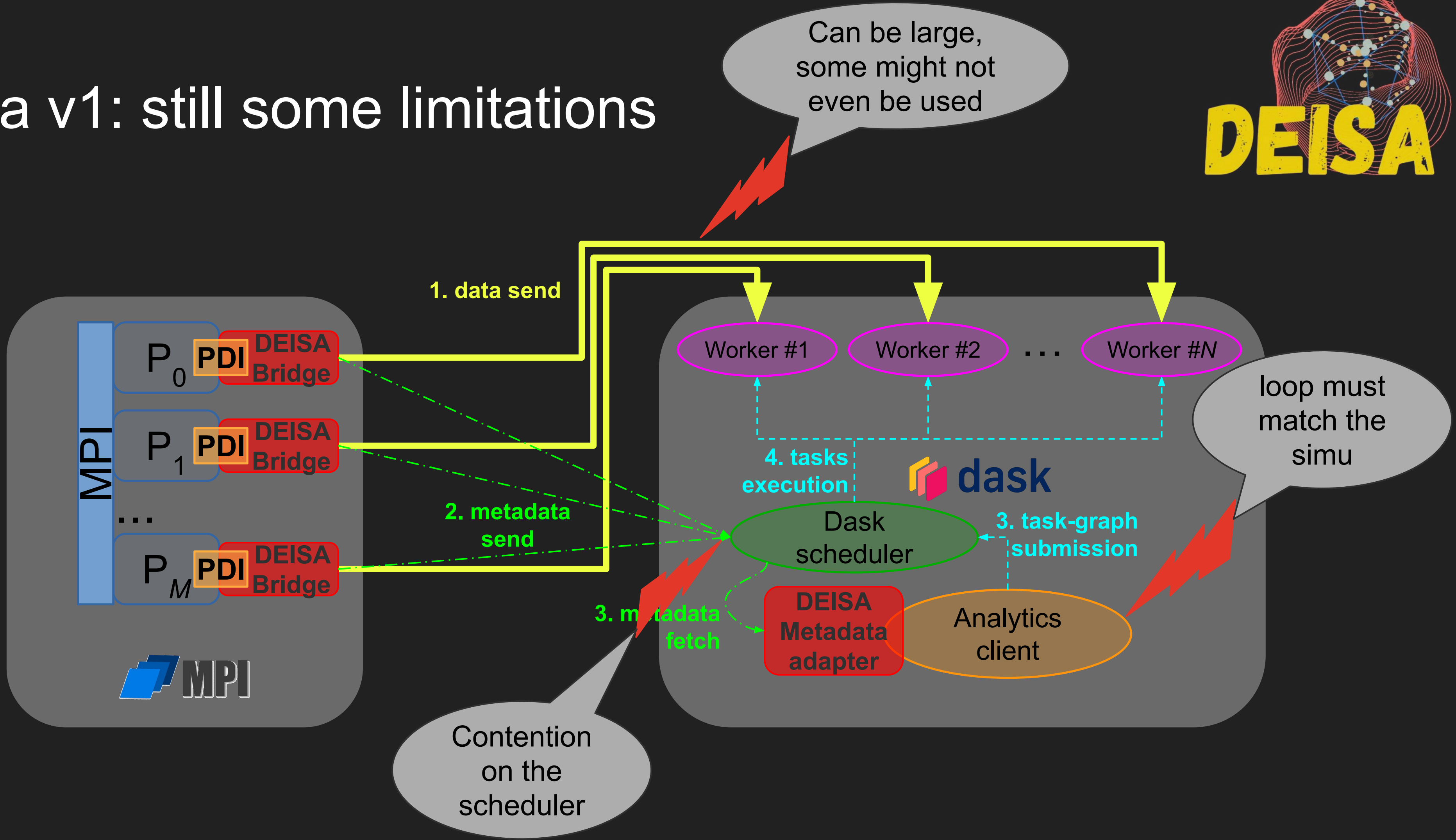


Preliminary performance analysis

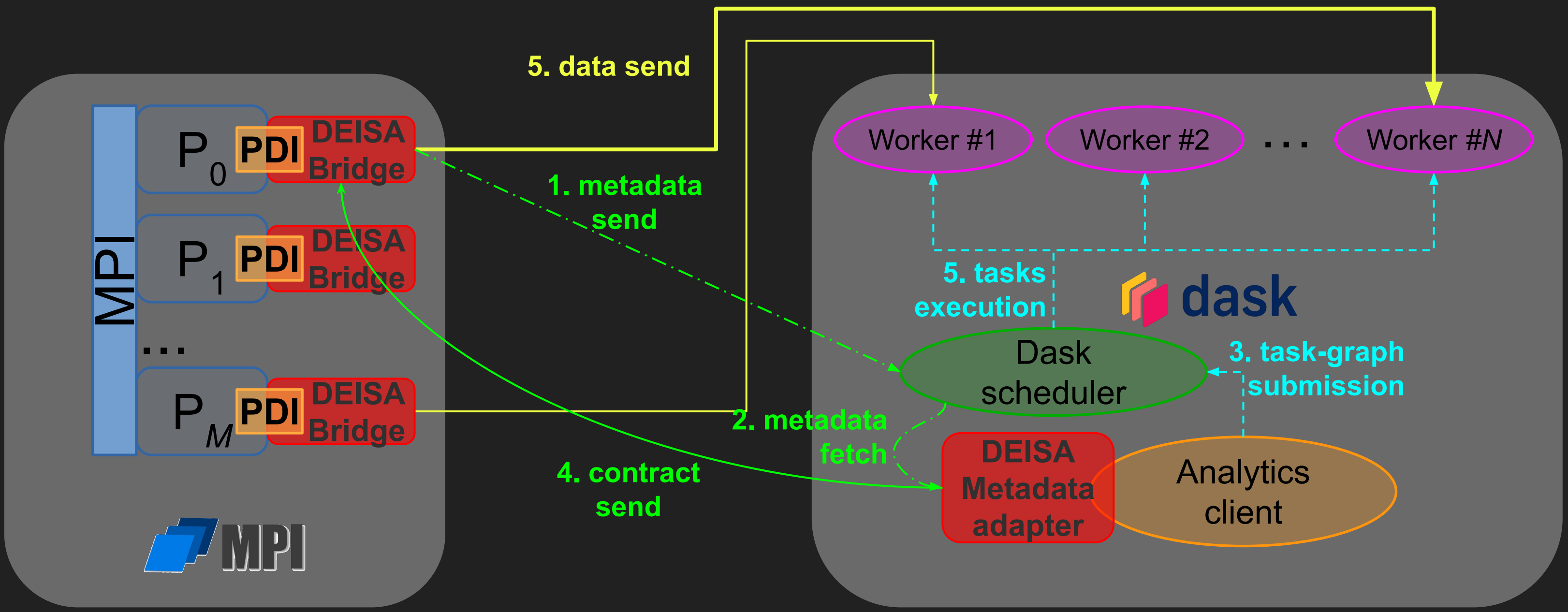




Deisa v1: still some limitations



Deisa v3



5. data send
MPI
P0
PDI
DEISA Bridge
P1
PDI
DEISA Bridge
...
PM
PDI
DEISA Bridge
1. metadata send
4. contract send
MPI
Worker #1
Worker #2
...
Worker #N
5. tasks execution
dask
Dask scheduler
3. task-graph submission
2. metadata fetch
DEISA Metadata adapter
Analytics client

Introducing Deisa v3 (single graph)

Metadata sent from simulation to dask ahe

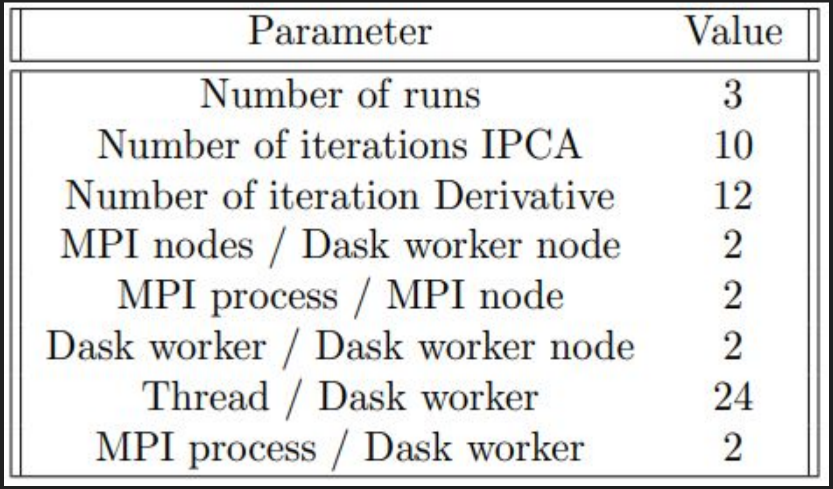
- A single task-graph construct
  - Requires the addition of th
- Time is a dimensio
  - More express
- Reduced
  -
-
  - n, do not transfer useless data

pplications

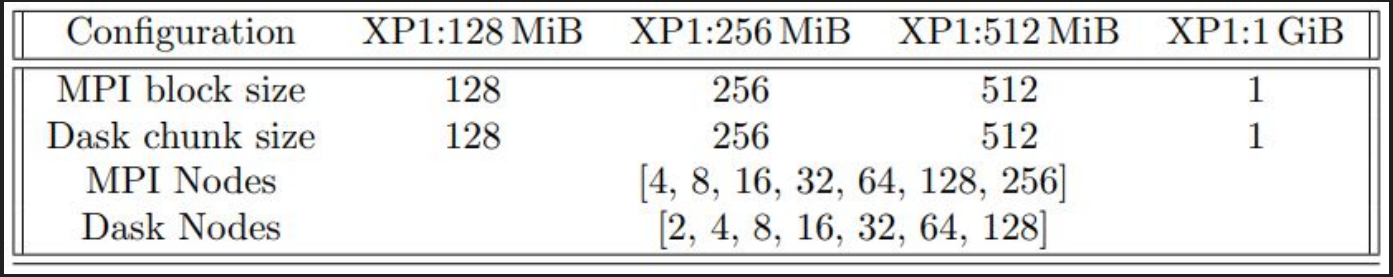
Used in production for grand-challenge on Adastra (CINES) #10 Top500 Multi-day full-scale run on the whole GPU partition

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

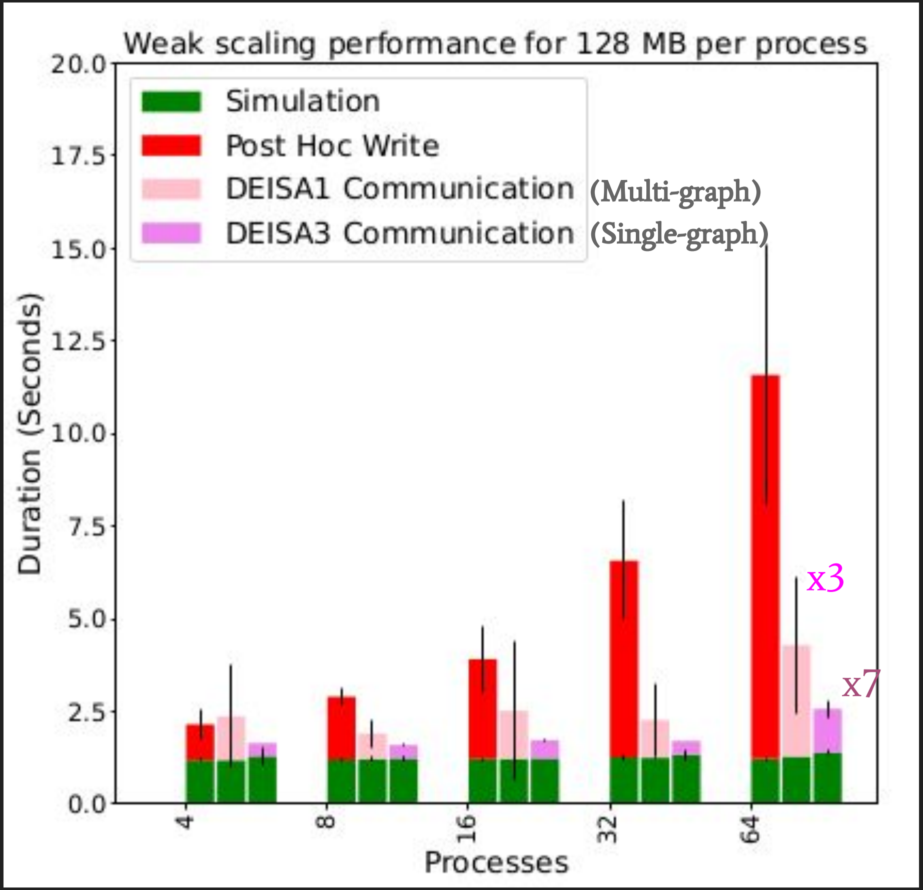


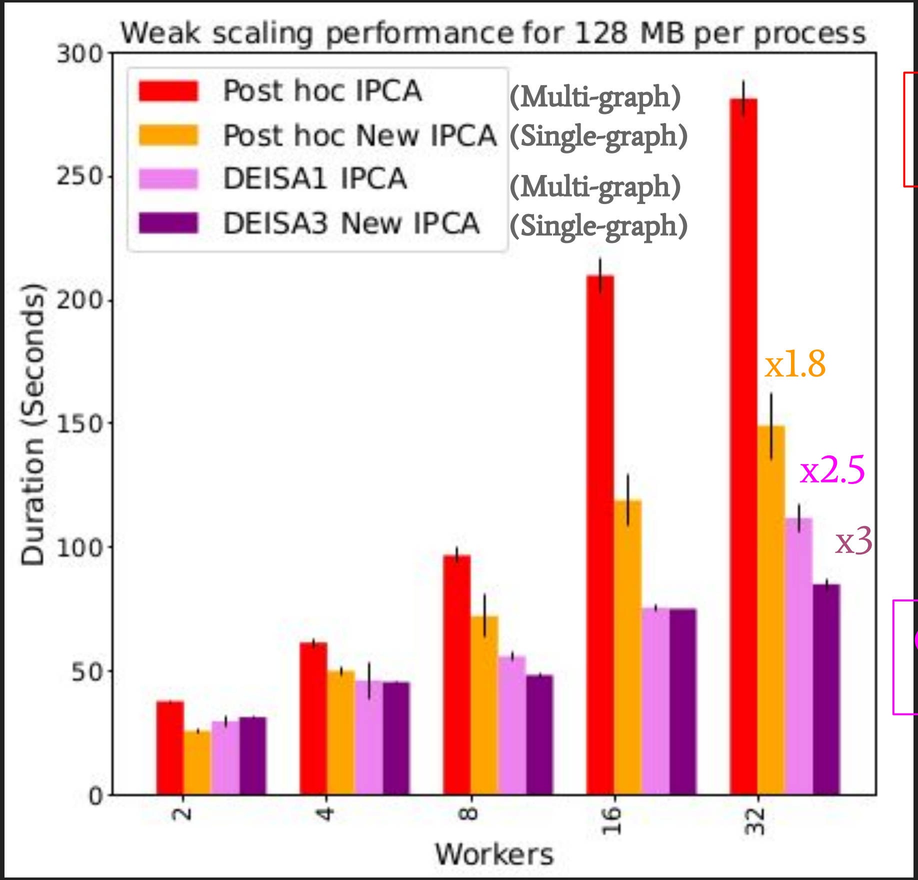
Parameter	Value
Number of runs	3
Number of iterations IPCA	10
Number of iteration Derivative	12
MPI nodes / Dask worker node	2
MPI process / MPI node	2
Dask worker / Dask worker node	2
Thread / Dask worker	24
MPI process / Dask worker	2



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

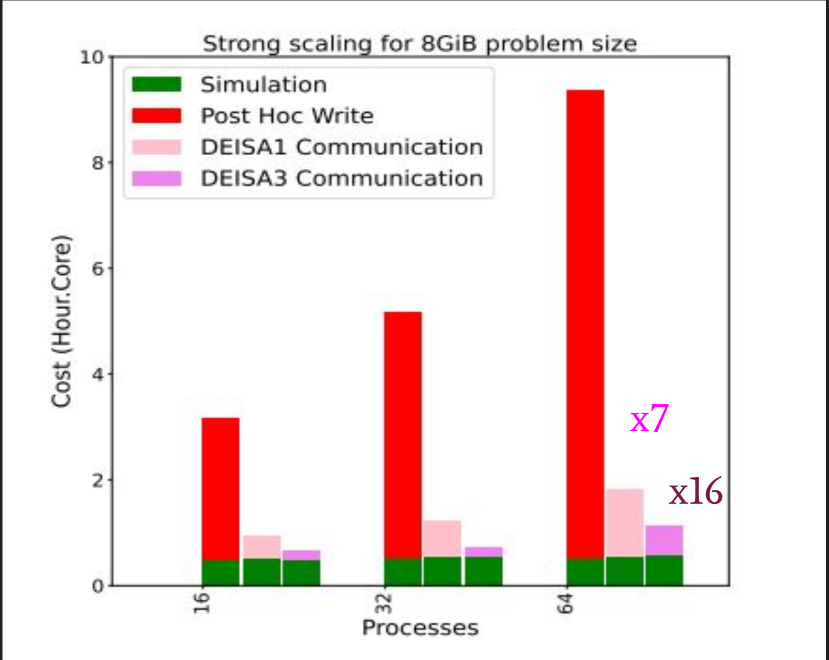




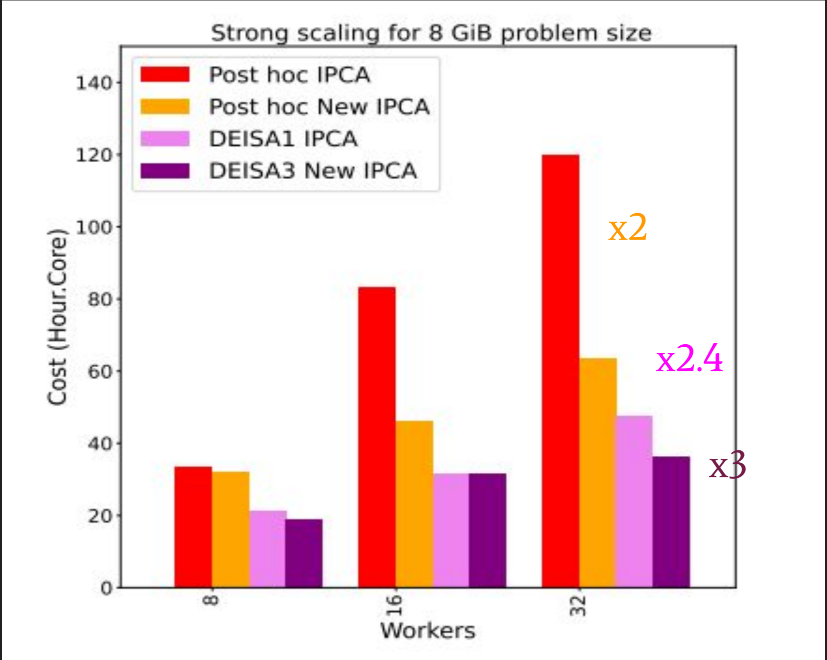
(reading data + Analytics)

(waiting data + Analytics)

DEISA vs Post hoc efficiency in hour.core

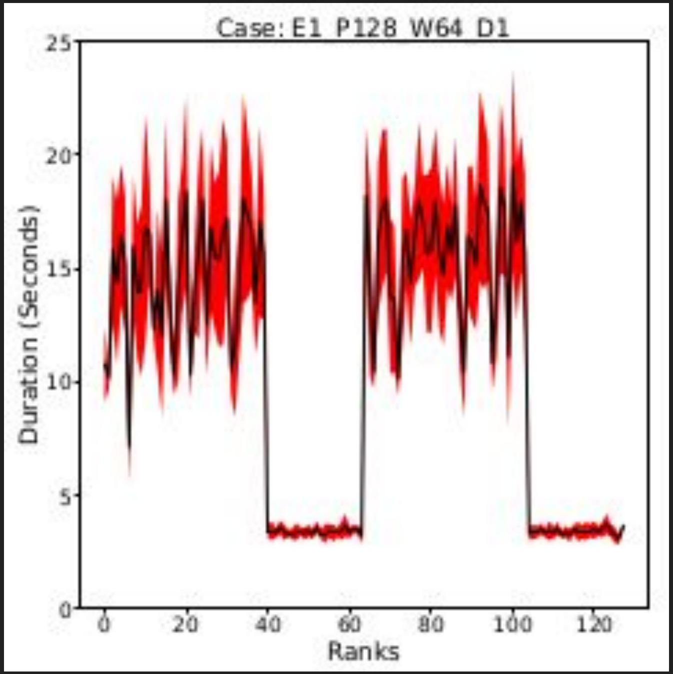


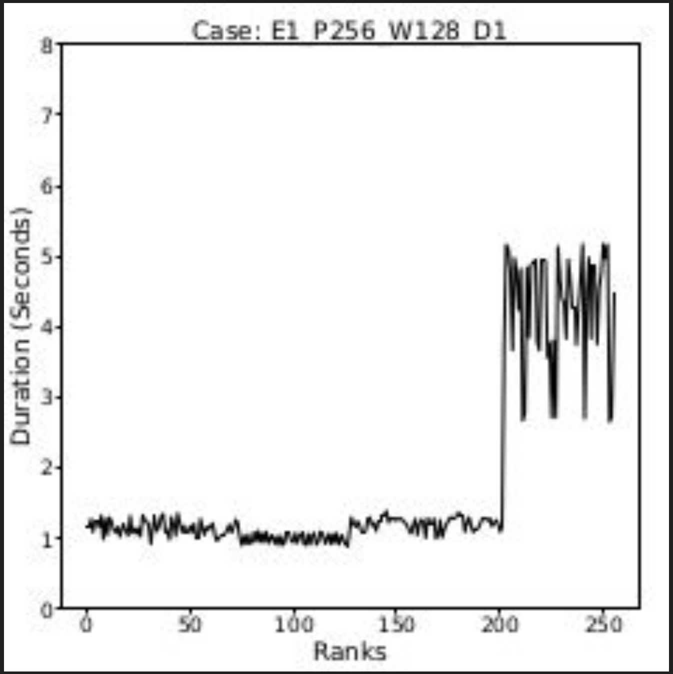
(c) Strong scaling results represented in hour-core for an 8 GiB problem size



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

Variability evaluation over iterations and processes





Multi-graph
-lot metadata
-heartbit=5s

Single-graph
less metadata
heartbit=∞

To summarize

Deisa v1

- In situ data transfer
  - from PDI instrumented simulation
  - to Dask cluster
  - without going through disk
- Dynamically at each time-step
- Pushed by the simulation

Deisa v3

- All Deisa v1
- But see time as any other dimension
- Data pulled by Dask (contracts)
- … but all metadata must be known ahead of time

Deisa

- For now, a proof of concept
- Result of a PhD. thesis

PDI

- A production software
- Documentation available
  - https://pdi.dev/
- Heavily tested & validated
  - >700 tests on 14 platforms
- Regular releases & packages
  - Debian, Ubuntu, Fedora, Spack

What's next in Deisa? **NumPEx** !

- Make Deisa production-grade (in progress)
  - Improve scalability & performance
  - Upstream dask modifications
  - Improve packaging
- Integrate in GYSELA rewrite
  - GyselaX++ => C++-based, GPU-first rewrite, using DDC (xarray for C++/GPU)
  - New analytics based on PDI/Deisa + xarray, support post hoc / in situ transparently
- Modularize and combine with other tools
  - Combine with Damaris for node-local reductions
  - Could a Melissa-like be based on this architecture?
- New features
  - Triggers & feedback from analytics to simulation
  - Support hybrid Dask-graph execution
    - Firsts tasks run in simulation process to prevent data copy
  - …