Handling IO data with PDI and Optimizing away IO with PDI/Deisa

• • •

GUEROUDJI Amal (CEA), BIGOT Julien (CEA), RAFFIN Bruno (INRIA),
Karol Sierociński (PSNC), Kacper Sinkiewicz (PSNC),
Yacine Ould-Rouis (CNRS)

Introduction :

>> Several ways/tools to handle generated data by scientific applications:

- >> IO tools (HDF5 / PHDF5, NetCDF4 / pNetCDF4, SIONlib, ...)
- >> Workflow management systems(FlowVR, Melissa, ...)
- >> Fault tolerance (FTI, ...)
- >> Data analysis frameworks(Dask, ...)

Introduction :

>> The good thing is that we have choice

>> The bad thing is that we need to change the application code every little change in the data we want to manage and the way we manage it.

Part I : PDI Data interface

PDI Data Interface :

>> PDI Data Interface decouples the simulation codes from data management (IOs, in situ/ in transit analytics, fault tolerance, workflow integration) concerns.

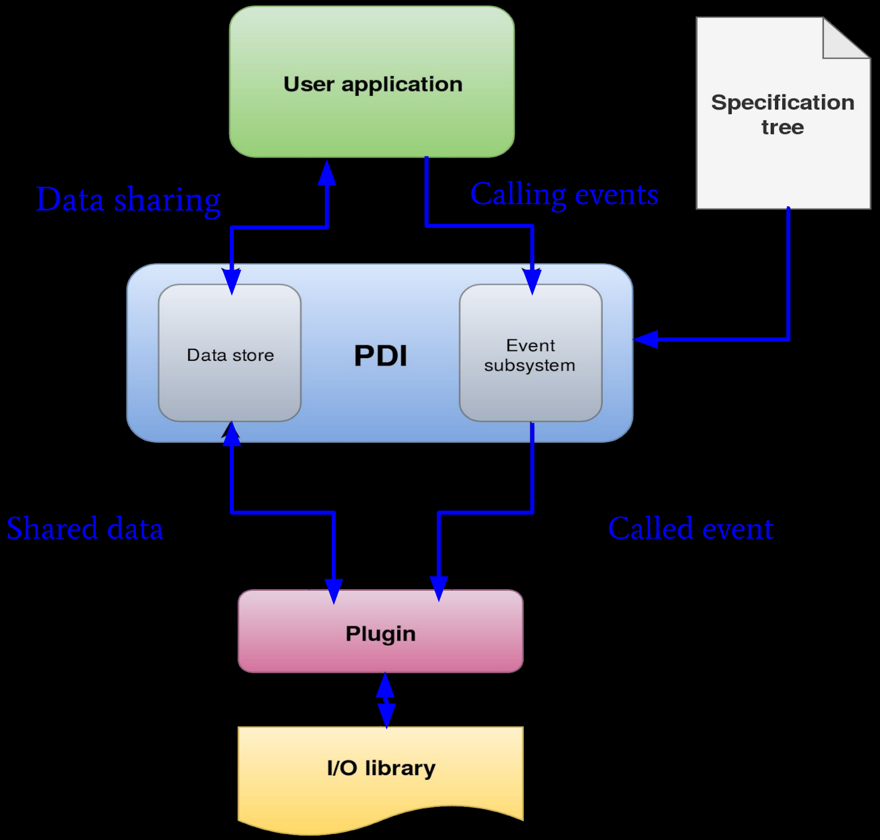
>> With PDI: Do it Once, Do It Right, Use it Everywhere

PDI Data Interface:

>> We have three main parts :

- Annotated code with call to PDI API(initialize, share, reclaim, ...)
- *YAML* configuration file (to describe the data layout and plugins)
- *Plugins* (to perform the needed job ex HDF5, Pycall ...)

PDI Data Interface Overview:



PDI user API:

>> All PDI functions that user can call are:

- PDI_init
- PDI_share
- PDI_reclaim
- PDI_release
- PDI_expose
- PDI_access
- PDI_event
- PDI_multiexpose
- PDI_finalize

PDI specification tree (Ymal):

- types: specifies user-defined datatypes,
- data & metadata: specify the type of the data in buffers exposed by the application; for metadata, PDI keeps a copy while it only keeps references for data,
- plugins: specifies the list of plugins to load and their configuration,
- plugin_path: specifies the path to a directory where PDI should search for plugins,
- logging: specify logger properties,
- additional sections are ignored.

PDI Plugins:

>> Builtin plugins:

- IOs: *decl'hdf5, decl'NetCDF, SIONlib*
- Fault tolerance: *FTI*
- Trace and debugging: *trace*
- Generic: mpi, *user-code, pycall, set-value, serialize*

>> User defined plugins:

- *Sensei, FlowVR, Melissa, Deisa*

Example:

```
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();
}
```

```
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] )'
```

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

Part II : DEISA Plugin

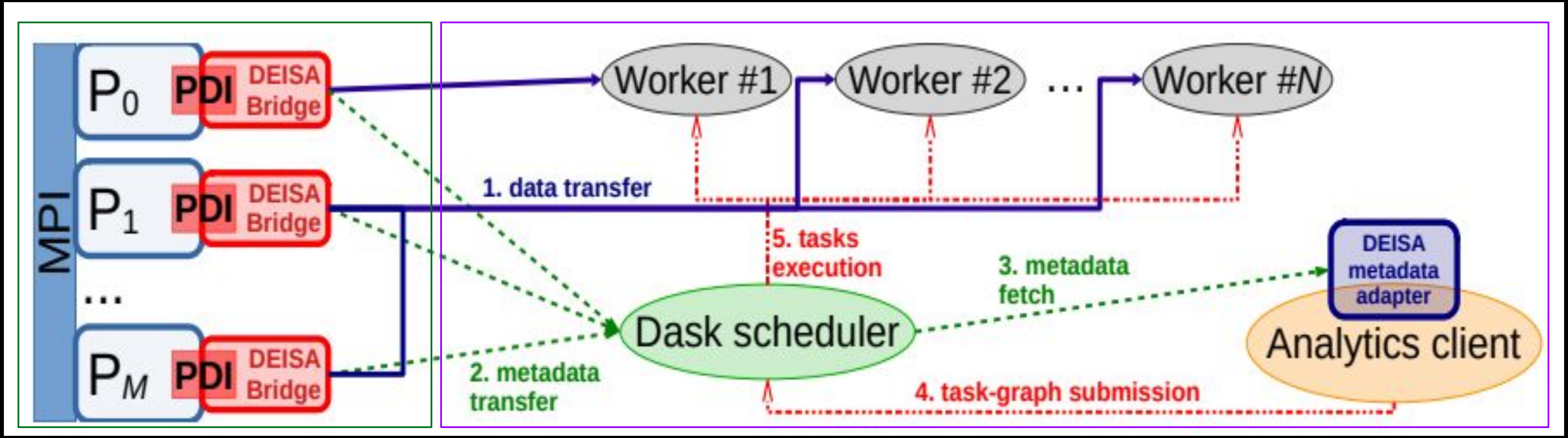
Dask-Enabled In Situ Analytics (DEISA):

>> Offers support for in situ analytics through Dask distributed

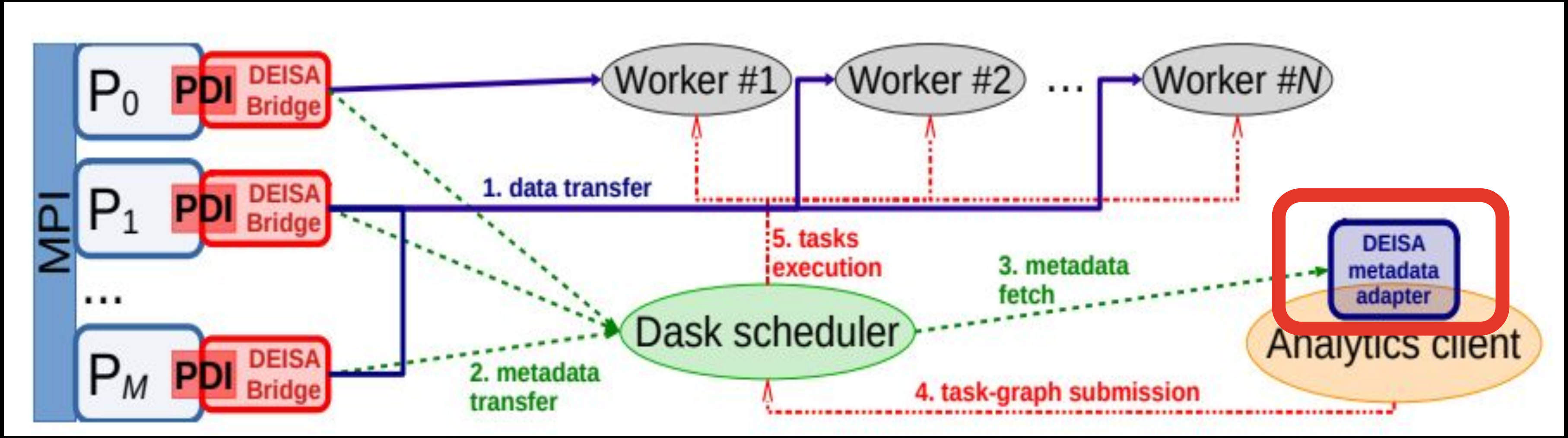
>> Brings the performance of in situ and the ease-of-use of post hoc processing together

>> Couples HPC and Big data fields

DEISA Overview:



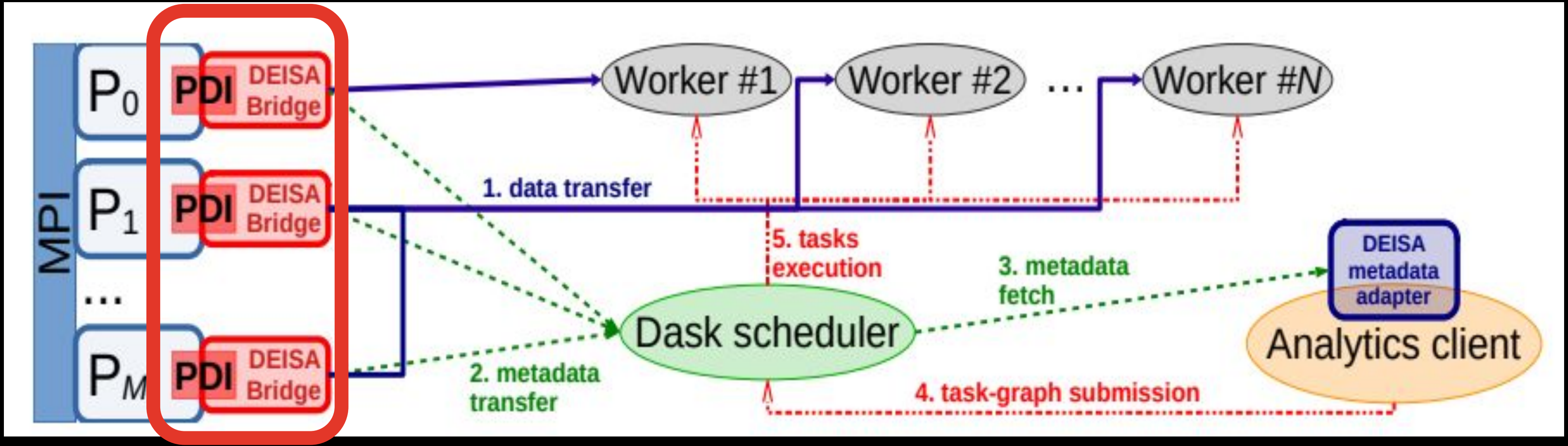
DEISA Overview:



Analytics with DEISA:

```
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_)
  print(pca.explained_variance_)
```

DEISA Overview:



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' } }
```

Performance evaluation:

>> Ruche supercomputer :

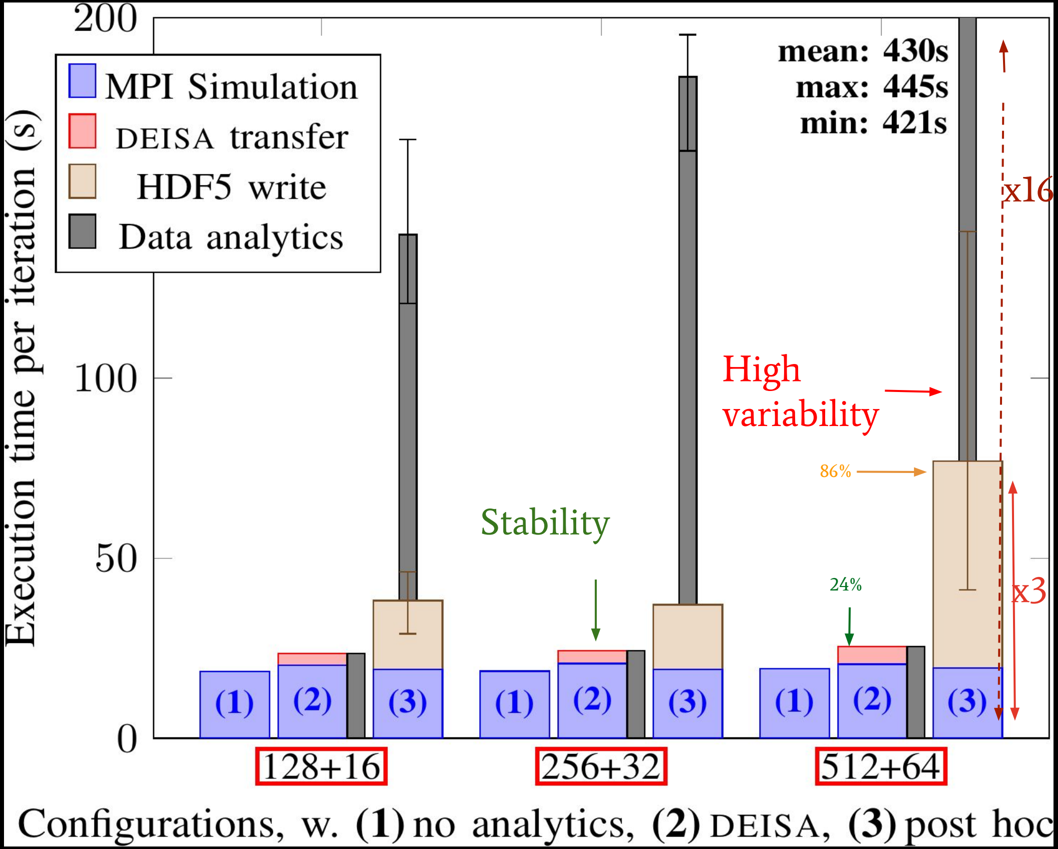
- 192 nodes (2 CPUs 20 cores each, 180 GB)
- Omni-Path 100 Gbit/s
- Spectrum Scale GPFS (IOs rate: 9 GB/s)

>> Mini-app :

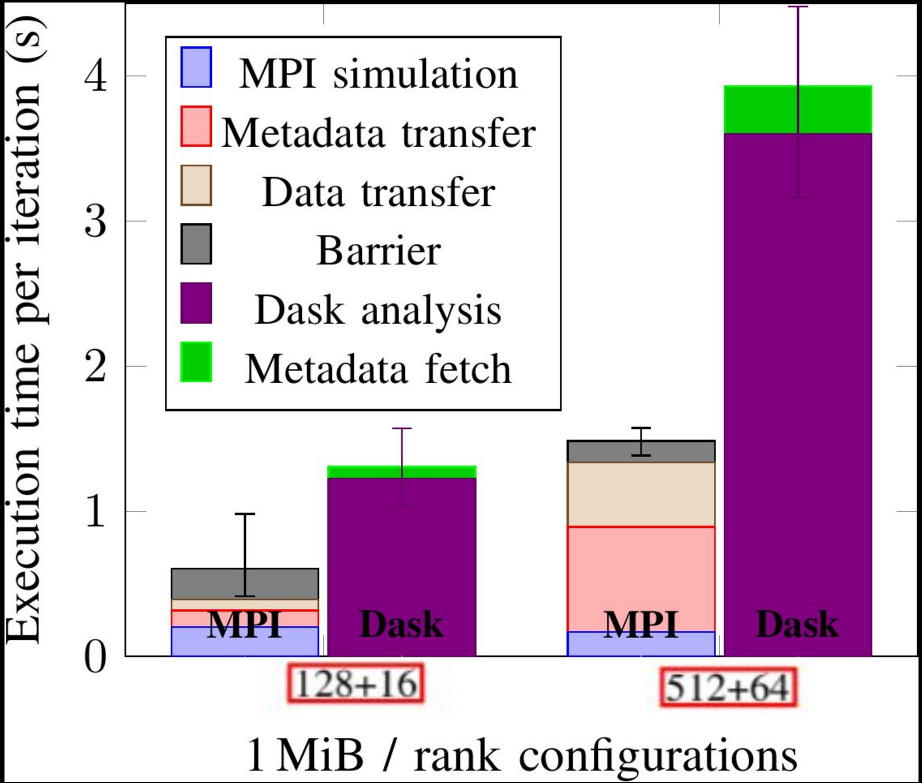
- 2D heat solver
- Incremental Principal Component Analysis

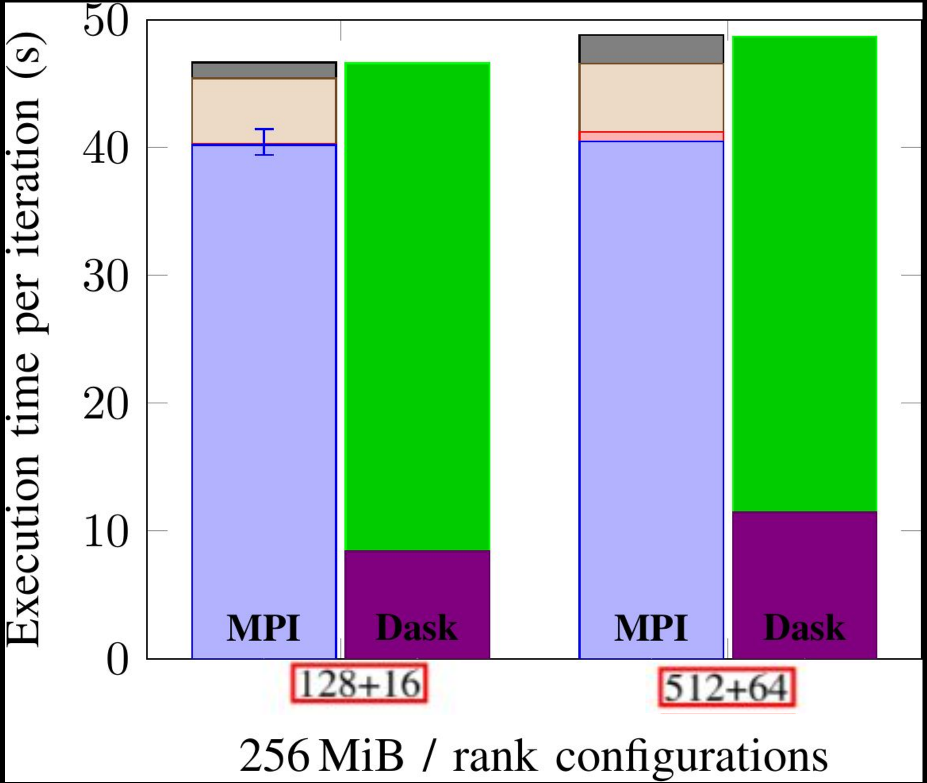
Performance evaluation

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



Detailed performance evaluation





Conclusion

>> PDI Data Interface :

>> Unified interface for IO and data handeling

>> Decouples data handling concerns from scientific applications

>> DEISA Dask-Enabled In Situ Analytics:

>> Leverages task-based programming model for in situ processing

>> Ease-of-use & performance gain

PDI documentation & support:

>> PDI official site: https://pdi.dev/master/index.html
>> PDI slack channel: https://join.slack.pdi.dev/
>> DEISA paper: Dask-Enabled In Situ Analytics (HAL)

- GUEROUDJI Amal(CEA)
  - amal.gueroudji@cea.fr
- BIGOT Julien (CEA)
  - julien.bigot@cea.fr
- RAFFIN Bruno (INRIA)
  - bruno.raffin@inria.fr