Deisa

Dask-enabled in situ analytics

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

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% on 1.46M threads on CEA-HF (AMD EPYC 7763)
- Intensive use of petascale resources:
  - ~ 150M hour.core / year
  - (GENCI + PRACE + HPC Fusion resources)
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

0D to 3D diagnostics

Integrals in phase space

Maxwell equations

Poisson equation

- adiabatic electrons

$\phi(r, \theta, \varphi)$

Diagnostics instead

Full-f

Semi-lagrangian

Collision operator

Source terms

Flux-driven

Gyrokinetic Boltzmann 5D equations

- Multi-ion-species:
  - GK eq. with arbitrary mass and charge
- Kinetic electrons in progress

Gyroaverage

E, B

Derivatives

3D

Global

(r, \theta, \varphi) geometry
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

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
gtmp_s = h5py.File('data.hdf5',mode='r')['gtmp_s'][

# process each time-step independently
for step in range(0, simu['timesteps']):
    pca = IncrementalPCA(n_components=2, copy=True,
                         svd_solver='randomized')
    pca.fit(gtmp_s[step,:,:])
    print(pca.explained_variance_)
```

Requires a single node computer with ~100TB RAM
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
gtmp = h5py.File('data.hdf5', mode='r')['gtmp']

for step in range(0, simu['timesteps']):
    pca = IncrementalPCA(n_components=2, copy=False,
                          svd_solver='randomized')
    pca.fit(gtmp[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
- ...

Worker #1  Worker #2  Worker #N

Dask scheduler

Analytics client
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

Running Simulation

Generated data

Diagnostics

PFS

In situ data visualization & processing

Pv₀

Pvᵦ

Damaris
Sensei
SmartSim
Visit[libsim]
Paraview[catalyst]
ADIOS [I, II]
…
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-Enabled In Situ Analytics

- 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

MPI

Code

Code

Code

API

API

API

API

HPC Library
What is PDI?

PDI annotations: a purely declarative API

PDI YAML spec. tree:
- What to do with data

Plugins for access to existing libraries

---

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

---

What is PDI?

MPI

Code Code ...

PDI PDI ...

PDI

HPC Library

API API API

MPI

...
What is PDI?

PDI Data Store

Data references

Events

MPI

Code

Code

Code

PDI

PDI

PDI

plugin

plugin

plugin

HPC Library

plugin

plugin

plugin

metadata:

- int
dsize: { type: int, size: 2 }
psize: { type: int, size: 2 }
data:

main_field:

- double
sizes: ['$dsize[0]', '$dsize[1]']

plugins:

decl_hdf5:

- file: meta$\{pcoord[0]\}x$\{pcoord[1]\}.h5
  write: [ dsize, psize ]
PDI: Annotation API usage

- 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)

```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
Decl’HDF5: the YAML

```
plugins:
dcl_hdf5:
  file: 'my_file_\${iteration_id}\${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();
}
```

```json
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
```

20
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.yaml"));
    int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    config_t cfg = read_config("simulation.yaml");
    // 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);
}
```

```json
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:
    deisa:
        scheduler_file: "/home/user/xp/sched.json"
        transfer: { gtemp: { 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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>128+16</th>
<th>256+32</th>
<th>512+64</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI processes</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>Dask workers</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>MPI nodes</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>Dask worker nodes</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Global data size</td>
<td>16 GiB</td>
<td>32 GiB</td>
<td>64 GiB</td>
</tr>
<tr>
<td>Dask generated tasks</td>
<td>15210</td>
<td>29010</td>
<td>55150</td>
</tr>
</tbody>
</table>
Preliminary performance analysis

1 MiB / rank configurations

256 MiB / rank configurations
Deisa v1: still some limitations

- Data send
- Metadata send
- Metadata fetch
- Task-graph submission
- Task execution
- Worker #1, #2, ..., #N

**Contention on the scheduler**

- Can be large, some might not even be used
- Loop must match the simu
Deisa v3

1. metadata send
2. metadata fetch
3. task-graph submission
4. contract send
5. data send

Worker #1
Worker #2
... Worker #N

Dask scheduler

MPI

DEISA Bridge

Analytics client

DEISA Metadata adapter
Introducing Deisa v3 (single graph)

Metadata sent from simulation to dask ahead:

- A single task-graph constructed encompassing all time-steps
  - Requires the addition of the “external tasks” concept to dask
- Time is a dimension like any other
  - More expressivity (e.g. one graph for time derivative)
- Reduced metadata transfer
  - Less contention on the scheduler
- Contracts
  - Detect data actually required by the graph, do not transfer useless data
  - Better performance

But… only for “regular” applications

Used in production for grand-challenge on Adastra (CINES) #10 Top500

Multi-day full-scale run on the whole GPU partition applications
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of runs</td>
<td>3</td>
</tr>
<tr>
<td>Number of iterations IPCA</td>
<td>10</td>
</tr>
<tr>
<td>Number of iteration Derivative</td>
<td>12</td>
</tr>
<tr>
<td>MPI nodes / Dask worker node</td>
<td>2</td>
</tr>
<tr>
<td>MPI process / MPI node</td>
<td>2</td>
</tr>
<tr>
<td>Dask worker / Dask worker node</td>
<td>2</td>
</tr>
<tr>
<td>Thread / Dask worker</td>
<td>24</td>
</tr>
<tr>
<td>MPI process / Dask worker</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>XP1:128 MiB</th>
<th>XP1:256 MiB</th>
<th>XP1:512 MiB</th>
<th>XP1:1 GiB</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI block size</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1</td>
</tr>
<tr>
<td>Dask chunk size</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1</td>
</tr>
<tr>
<td>MPI Nodes</td>
<td>[4, 8, 16, 32, 64, 128, 256]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dask Nodes</td>
<td>[2, 4, 8, 16, 32, 64, 128]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DEISA vs Post hoc Weak Scalability

Weak scaling performance for 128 MB per process

- Simulation
- Post Hoc Write
- DEISA1 Communication (Multi-graph)
- DEISA3 Communication (Single-graph)

![Graph showing weak scaling performance with DEISA vs Post hoc Weak Scalability](image)

(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
- heartbeat=5s

Single-graph
- less metadata
- heartbeat=∞
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
  - ...