2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-1016-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/HIPC53243.2021.00015

DEISA: Dask-Enabled In Situ Analytics

Amal Gueroudji*†, Julien Bigot*‡ and Bruno Raffin§
* Université Paris-Saclay, UVSQ, CNRS, CEA, Maison de la Simulation, 91191, Gif-sur-Yvette, France
† Email: amal.gueroudji@cea.fr
‡ Email: julien.bigot@cea.fr, ORCID: https://orcid.org/0000-0002-0015-4304
§ Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
Email: bruno.raffin@inria.fr, ORCID: https://orcid.org/0000-0002-7980-4946

***Abstract*—A widening performance gap is separating CPU performance and IO bandwidth on large scale systems. In some fields such as weather forecast and nuclear fusion, numerical models generate such amounts of data that classical post hoc processing is not feasible anymore due to the limits in both storage capacity and IO performance. In situ approaches are attractive to bypass disk accesses in these cases and fully leverage the HPC platform. They are however often complex to set up and can require to re-develop parallel versions of the analysis from scratch.**

In this paper we propose a hybrid model that is well suited for in situ workflows that combine regular simulations and irregular analytics. Our model couples the bulk synchronous parallel paradigm for simulation with a distributed task-based one for analysis. This reduces complexity and leverages the best of each of these two powerful paradigms. We validate the model with a prototype, called DEISA, that supports coupling MPI parallel codes with analyses written using Dask. This implementation requires minimal modifications of both the simulation and analysis codes compared to their post hoc counterpart. It give access to an already existing rich ecosystem to be used in situ such as the parallel versions of Numpy, Pandas and scikit-learn.

Experiments in configurations up to 1024 cores show that DEISA can improve the simulation wallclock time (excluding analysis) by a factor up to 3 and the total experiment (including analysis) hour.core cost by a factor of up to 5 compared to parallel post hoc with plain Dask while requiring the modification of only two lines of python code, three of YAML, and none at all in a C simulation code already instrumented with PDI Data Interface.

***Index Terms*—In situ processing, code coupling, task-based programming, MPI, Dask**

I. Introduction

The classical way to handle simulation data is by saving it to disk to read it back later for *Post Hoc* processing. In many fields such as weather forecast and fusion plasma, simulations can generate dozens of terabytes of data per hour. At this scale, performance is limited by the file system, in what is known as the IO bottleneck. *In Situ* processing proposes to process the data as soon as produced by the simulation, bypassing the disk to avoid the bottleneck. The performance of both the simulation and analysis are improved. However processing data in situ is usually more complex to setup than post hoc. In situ tools are typically built following the MPI programming model inherited from the host simulation.

In MPI, codes are expressed as a set of statically placed, long-lived processes that explicitly communicate during their life-time and manipulate data through variables whose value evolve along the execution. The Bulk Synchronous Parallel (BSP) paradigm and MPI in particular are well adapted for massively parallel codes where performance is crucial, thus it is widely used for simulation codes. Users do however need to explicitly specify all resource allocations, communications and computation ordering which make it difficult to compose and complex to use for data analysis.

In task-based systems, applications are expressed as task graphs whose tasks are short-lived and communicate through immutable data values that also express dependencies. Task execution and placement is automatically and dynamically handled by the runtime, as are communications and the life-time of data. Task-based programming has proven well suited for data analysis where easy adaptation is key and performance is usually less critical than for the simulation itself.

In order to offer the best of both worlds, we propose a new hybrid model that combines the BSP and the distributed task-based paradigms. This reduces the complexity and takes advantage of these two powerful paradigms in the same workflow. In this paper we have chosen Dask distributed framework as a task-based environment because of the rich ecosystem it offers in particular the distributed versions of well known libraries such as numpy and scikit-learn.

Overall, this paper makes the following contributions: 1) it proposes a new paradigm that combines BSP and task-based programming, 2) it implements this paradigm in a prototype, called DEISA, coupling MPI with Dask, 3) it validates the approach with an application example that highlights the ease of use from the user perspective and compares it to a post hoc version that requires similar development efforts, 4) finally, it evaluates its performance and shows gains when using DEISA compared to a pure Dask post hoc version that can reach a factor up to 5 in execution time.

The remaining of the paper is organized as follow, Section II presents a brief review of the related work on in situ data analytics and task-based systems. Section III presents Dask. Section IV introduces our approach and the architecture of DEISA. Section V evaluates its performance for different configurations before Section VI concludes the paper.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 824158 (EoCoE-2).

†AG was supported by the CEA NUMERICS program, which has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement No 800945

2640-0316/21/$31.00 ©2021 IEEE
DOI 10.1109/HiPC53243.2021.00015

II. Related Work

The in situ paradigm was applied first to visualisation [1], before being extended to more general purpose data processing. As such, most of scientific visualization frameworks meant for high performance computing support in situ visualization. Paraview [2], built on top of VTK [3] or Visit/Libsim [4], are both supporting in situ processing through the extensions Catalyst [5] and Libsim [6] respectively. In situ visualization comes as a built-in feature in the latest developments such as Alpine/Ascent/VTKm [7], [8] or SENSEI [9].

Other frameworks take a more generic approach to support any kind of data processing in situ, in transit (data are first moved to extra nodes not running the simulation) or a mix of both. Analysis results are eventually saved to disk rather than directly visualized. FlowVR [10], Decaf [11] and Bredala [12], in situ extensions for ADIOS1 [13], [14] and ADIOS2 [15], Damaris [16] or Dataspaces [17] are examples of frameworks in that category. All these tools rely on a static parallelization, derivative of the data-flow model: the tasks of the analysis workflow are mapped to compute resources statically. This often leads to high performance, but require the user to explicitly control this mapping. The underlying transport layer is often based on MPI, simplifying the coupling with the simulation code also based on MPI, or the introduction of an analysis algorithm parallelized with MPI in the workflow.

The map/reduce model, supported by frameworks like Spark or Flink, is a popular parallelization model for data analysis tasks for Big Data oriented applications. A few attempts have been made with this model for in situ processing: SMART [18] proposes a map/reduce interface for programming analysis on top of MPI/OpenMP, while [19] takes benefit of Flink stream processing support for enabling in transit analysis. But the model provides a loose control on data partitioning that is not well adapted to support efficient parallelization of patterns such as stencil computations [20] or large scale linear algebra.

Task-based programming where the tasks are dynamically distributed to compute resources is today classical for shared memory programming using for instance OpenMP or Intel TBB. TINS [21] leverages this approach as long as the simulation is also parallelized on each node with tasks. TINS relies on the TBB work-stealing scheduler to dynamically distribute the tasks on the cores, being simulation or analytics tasks. The benefits are twofold: performance is improved as cores are not assigned exclusively to analysis or simulation workload, and the user does not have to take care of task to core mapping. Goldrush investigates a similar approach in the context of OpenMP [22].

Extensions of task-based programming to distributed programming, such as PyCOMPSs [23], [24], Dask [25], Ray [26], Parsl [27], and Pygion [28] are gaining popularity for scientific data analysis for the mix of performance and simplicity they offer. They provide a Python interface and often the transparent parallelization of some classical APIs (or part of them) like Numpy or Pandas. But direct coupling of MPI based parallel simulation with such task-based system is not directly supported. Either it needs to go through a file interface with the associated performance impact or requires to aggregate data on the application side to mask the parallelization (this also has a performance impact), if not completely rewriting the application. Some early work has been done with Legion for in situ visualization [29]. In this paper we propose a direct coupling solution for Dask.

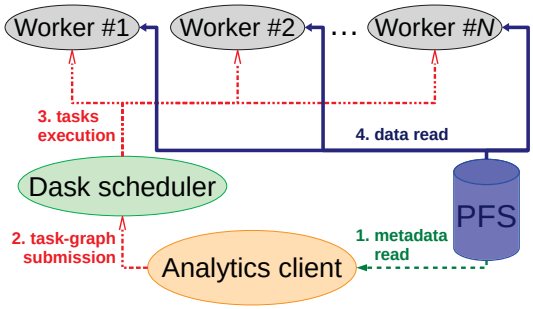

Fig. 1: Dask architecture in a typical post hoc context, one client, N workers connected to the scheduler. 1) The client reads small metadata regarding the needed files from the PFS, 2) creates the Dask data structure and submits a task graph, 3) the workers execute the tasks, 4) some of which read data blocks in parallel from the PFS.

III. Dask

Dask is a tool for distributed parallel execution of python code that is built on task-based programming model. Its architecture is organized around three components: a centralized scheduler, multiple workers and one or more clients as illustrated in Figure 1. At the lowest level, the scheduler and workers support remote procedure calls using Tornado web services.

The workers offer services to store data and to remotely execute code on this data. The scheduler offers services to execute complete task graphs submitted by the client, delegating tasks execution to the workers. It manages the graph to identify and submit to the workers those tasks whose dependencies are fulfilled. It also takes care of data management based on reference counting. When some data stops being referenced, a garbage collector on the scheduler triggers its deallocation on the workers. From the client point of view, task programming is based on python *Delayed* and *Future* interfaces. One can build and submit a task graph using the usual python API for asynchronous code execution. The resulting future acts as a reference to the actual data on a worker. This prevents its deallocation and supports fetching the value back to the client.

Higher-level APIs such as *Dask Array*, *Dask DataFrame* and *Dask-ML* are also available. They are mostly drop-in replacement for the popular *Numpy*, *Pandas* and *scikit-learn* APIs. For example, `dask.array` supports a distributed data structure representing an array split in blocks distributed on multiple workers. When instantiating an `Array` from HDF5, as illustrated in lines 11 and 12 of Listing 2, the client only

```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_)
```

Listing 1: Sequential post hoc data analysis with scikit-learn

```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_)
```

Listing 2: Parallel post hoc data analysis with Dask. Lines differing from the analysis of Listing 1 are highlighted

reads the metadata required to create the tasks that will actually read data from the file chunk by chunk on the workers as illustrated by Figure 1. This way, data can be larger than the memory of a single node. With these high-level APIs, the users can write analysis codes that are very similar to the sequential codes. Listings 2 and 1 present an example of post hoc analysis with a sequential NumPy code and the equivalent parallel code using Dask-ML.

IV. DEISA Approach

Unlike the post hoc, in the in situ paradigm, the source of data is not a file anymore, but the parallel simulation code itself. In order to support this, DEISA adds two main components to Dask post hoc architecture, as illustrated in Figure 2: the DEISA *bridge* and DEISA *metadata adapter*. The bridge is instantiated by each MPI process of the simulation and is responsible for sending the data to Dask workers and the metadata to the scheduler. The metadata adapter is executed by the Dask analytics client that fetches this metadata to reconstruct a global descriptor for the distributed data. This can then be used by Dask client in the construction and submission of a standard task graph. After this step, the execution of the task graph by Dask proceeds normally and similarly to a standard post hoc execution.

```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();
}
```

Listing 3: PDI instrumentation of the C simulation code

```yaml
types: #[...] including config_t description
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] )'
```

Listing 4: Data description in PDI YAML file

a) Simulation Code Instrumentation: To extract the data from the simulation, DEISA relies on the PDI Data Interface (PDI) [30], [31]. PDI is a thin interface designed to move the IO concerns that often pollute the simulation code out of it. It supports C, C++, Fortran & python. As illustrated in Listing 3 (lines 7 and 19), PDI calls are introduced in the simulation code to identify when and where internal data structures become accessible to external tools. A YAML file, selected at initialization (line 3), describes the data structures and specifies what actions (provided by PDI plugins) to trigger on the exposed data.

For example, Listing 4 provides the YAML description of

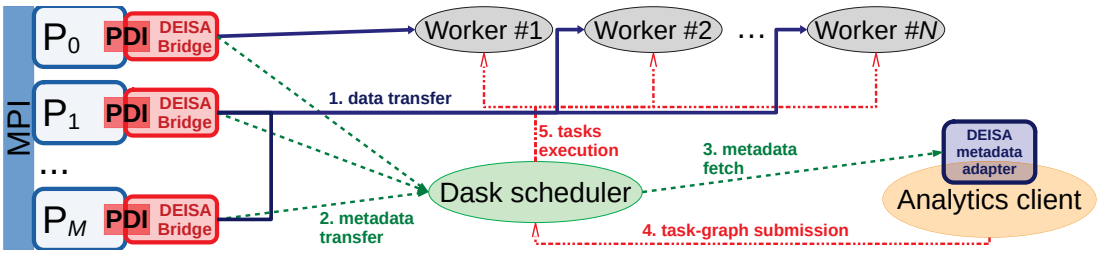

Fig. 2: DEISA example coupling an MPI application with M processes to a complete Dask instance running N workers

the data exposed by the code of Listing 3. The type of small metadata is described on line 2 and a copy of these values is kept by PDI. For the data itself, PDI only manipulates pointers on the user data so as to limit memory and execution time overheads to a minimum. The description of the data structures, such as `temp` on line 11, can use the value of metadata exposed by the code through *$-expressions*. This is for example illustrated for the size of the array on line 14. This reduces information redundancy between the code and YAML and ensures information used by PDI reflects that from the simulation.

In addition to the data directly exposed by the code, a 3D array (`gtemp`) is defined on line 4. The size of this array is infinite in the time dimension and the product of the local array size (`$cfg.loc`) by the number of processes in the two other dimensions. It represents the logical global array distributed over MPI and the time-steps; it is never fully stored in memory or exposed by the code. Instead, the `temp` array is mapped in `gtemp` as a time-slice and a block in the other two dimensions on line 15) to give PDI a global view of the distributed data.

PDI is typically used to push data to files as illustrated for the HDF5 case in Listing 5. The Decl'HDF5 plugin is loaded (line 4) and used to write the `gtemp` array to the `data.h5` file. A filter expression introduced by the `when` keywords on line 8 is used to select the data to actually write. The MPI plugin is also loaded on line 3 and parallel HDF5 with collective IO over the `MPI_COMM_WORLD` is selected on line 9. Whenever the `temp` array is exposed from the code, an HDF5 operation is triggered. The first time this happens, a 3D dataset whose shape matches that of `gtemp` is created. Then, HDF5 hyperslabs are used to describe the part of `gtemp` available in the local MPI process according to the mapping information of Listing 4, line 15. This data is written to the file using HDF5 collective IOs over the communicator provided as is usual in parallel HDF5. This specification results in the writing of a 3D array in the HDF5 file independent of the MPI distribution and including all time-steps in a dedicated dimension.

In addition to Decl'HDF5, other PDI plugins support IO in formats like NetCDF, SIONlib, FTI, etc. Existing plugins also support data serialization or integration in FlowVR workflows. Finally, plugins exist to call code that manipulates and transforms the data directly from the YAML file in any language supported by PDI (C, C++, Fortran, python). In particular,

```
1 #[...] data description from Listing4
2 plugins:
3   mpi: ~
4   decl_hdf5:
5   - file: data.h5
6     write:
7       gtemp:
8         when: '$step>0'
9         communicator: $MPI_COMM_WORLD
```

Listing 5: PDI YAML file to write to HDF5

```
1 #[...] data description from Listing4
2 plugins:
3   deisa:
4     scheduler_file: "/home/user/xp/sched.json"
5     transfer: { gtemp: { when: '$step>0' } }
```

Listing 6: PDI YAML file to analyze data through DEISA

this allows python code to be called from a C, C++ or Fortran simulation code.

DEISA leverages PDI to limit its intrusiveness in the simulation code. It requires neither modification nor recompilation of a code already instrumented with PDI for file output. Only the YAML file must be changed as illustrated in Listing 6. The `deisa` plugin is loaded instead of `decl_hdf5` (in fact, both can be loaded alongside each other without interference). In the DEISA-specific configuration, on line 4, one gives the path to the JSON encoded scheduler information file, which is generated by the Dask scheduler at initialization. The `transfer` keyword on line 5 selects the data to make available for analysis and can include a filter expression as in the HDF5 case. When the DEISA plugin is loaded, it instantiates a DEISA bridge and passes the `scheduler_file` path to it. Then, similarly as in Decl'HDF5, whenever the `temp` array is exposed from the code, if the `when` filter condition is true, a DEISA operation is triggered. The plugin gathers the information about the buffer (name, type and size of the local block and its position in the global array) as well as the data itself and calls the `publish_data` method of the bridge.

b) DEISA *bridge:* DEISA bridge is a python class that handles the data exposed through PDI and that interfaces with Dask. It uses Dask client API and behaves as a regular client from Dask point of view. It does however limit its role to the transfer of data to Dask workers and metadata to the scheduler, without submitting any task.

At initialization, the class retrieves the information required to connect to the scheduler from the provided `scheduler_file` file. It queries the scheduler about the list of workers available. Since in the Dask model, clients can not directly communicate with each others, the bridge (that is a client from Dask point of view) initialize a Dask *Queue* on the scheduler to support information transfer to DEISA metadata adapter.

When the `publish_data` method is called, the bridge starts by sending the metadata (name, type and size of the local block and its position in the global array) to the queue on the scheduler; this is a very small message (typically a few bytes). It then selects a peer worker where it sends the data using Dask `scatter` function. The function is called with the `direct` parameter set to `True` in order to enable transfer data to the worker directly, without going through the scheduler. The peer worker is selected in a round robin fashion based on the local MPI rank to ensure a balanced memory load on the different workers that are often less numerous than MPI processes.

The scatter function returns a Dask future that acts as a reference whose existence keeps the data alive and prevents its deallocation by Dask garbage collector. The bridge then sends this future to the queue on the scheduler. Sending the future in a separate message enables the scheduler to include it in the list of references keeping the data alive in the workers. The bridge then destroys its local future and completely forgets about the data it just sent. By default, the queue size on the scheduler is unlimited. It can however be limited to setup a flow control: a bridge is blocked when intending to write in the queue if this queue is full. A dedicated `buffer_nb` keyword in the YAML enables to set the max queue size.

c) DEISA *metadata adapter:* DEISA metadata adapter is a python class instanciated from the analytics code in Dask client that gives access to the data produced by the simulation for use in task graphs. This adapter implements a subscript operator (line 11 of Listing 7) that takes a name as parameter and returns a descriptor associated to the array provided with that name by the MPI simulation code. This descriptor is returned as an instance of a dedicated class that also implements the subscript operator for slicing (line 15 of Listing 7). When called, this second subscript operator returns the `Array` describing the data on the workers matching the described selection and that can be used in the construction of a normal Dask task graph.

The first subscript operator call (line 11) only stores the name and a reference to the adapter in the returned descriptor. The implementation lies entirely in the subscript operator of this returned descriptor. When called (line 15), it fetches the metadata and futures from the queues on the scheduler and stores them on the client. This empties the queues on the scheduler and enables the MPI processes to send more data if this was blocking them. It also creates a new reference to the data on the analytics client, hence ensuring the data on the workers is not deallocated by Dask garbage collector. As soon as all blocks that are part of the requested selection are retrieved, the adapter combines them into a single `Array` descriptor that represents the global distributed array. The resulting descriptor can be seamlessly used as input to Dask parallel algorithms.

```
1  import dask.array as da
2  from dask_ml.decomposition import IncrementalPCA
3  import yaml, json
4  import deisa
5  # Connect to Dask
6  sched = json.load(open('sched.json'))
7  client = dask.distributed.Client(sched["address"])
8  # load the simulation configuration
9  simu = yaml.load(open('simulation.yml'))
10 # Get data from DEISA
11 gtemp = deisa.Adapter(client)['gtemp']
12 for step in range(0, simu['timesteps']):
13   pca = IncrementalPCA(n_components=2, copy=False,
14                        svd_solver='randomized')
15   pca.fit(gtemp[step,:,:])
16   print(pca.explained_variance_)
```

Listing 7: Parallel in situ data analysis with DEISA. Lines differing from the analysis of Listing 2 are highlighted

An example of use of this API is illustrated in Listing 7. This example performs the exact same analysis using Dask as in Listing 2, but it uses DEISA for in situ execution instead of loading data from HDF5 for post hoc execution. Apart from importing DEISA module on line 4, the only difference with the post hoc version is that a DEISA metadata adapter instance must be initialized with a reference to the Dask client on line 11 instead of opening a HDF5 file. The actual fetching of metadata and construction of the `Array` descriptor occurs when `gtemp` is in turn subscripted on line 15. This final descriptor can be used in the `IncrementalPCA` in the exact same way it was used post hoc.

In this section, we have focused on the implemented prototype on top of Dask, however a similar approach can be adapted for other distributed task-based frameworks such as PyCompSs, Ray and so on. Each of them has its strengths and may be the best framework for a particular need. In our case we have chosen Dask because it's widely used by the data analytics community in several domains for the rich ecosystem it offers, in particular the parallel versions of numpy, pandas and scikit-learn.

V. Evaluation

To evaluate DEISA, we have implemented a mini-app coupling a MPI simulation code to parallel data analytics. The simulation code is illustrated in Listing 3 and relies on a modified 2D explicit finite difference heat solver parallelized in MPI using a block domain decomposition. It is representative of a typical 2D Eulerian simulation with stencil computation pattern and MPI ghosts data exchange. Outputs, consisting in the temperature field on the 2D domain, are produced periodically after a fixed number of iterations set to represent a realistic compute-to-output time ratio. Outputs are either written to parallel HDF5 files (then reread for post processing

with Dask) or sent directly to DEISA for in situ processing as discussed in the previous section.

The analytics code, illustrated in Listings 2 for its post hoc version and 7 for DEISA version, computes a principal component analysis on the 2D temperature field at each time-step. The PCA is computed using the parallel incremental PCA provided by Dask [32] that is more memory efficient than the basic PCA algorithm. This analysis has not been selected for its physical significance in this specific case, but because it is representative of analytics used in real simulations [33].

A. Ease of use

To switch from Dask + PDI/Decl'HDF5 for parallel post hoc analytics to DEISA for in situ analytics, the simulation code does not have to be modified or even recompiled. PDI YAML file must be modified to load the DEISA plugin instead of Decl'HDF5, however the configuration format of both plugins is very similar and this is only a minor change. The python analytics code must be changed also, but once again, the interface offered by `deisa.Adapter` is close enough to that of `h5py` that this change is very limited.

For our PCA example, only four lines change in the YAML file between Listing 5 and 6. A more complex simulation would only require one more line by result exported from the simulation. For this same example, only two lines change in the python analysis code between Listing 2 and 7. Once again, a more complex simulation would only require one more line by result imported from the simulation.

Execution setup is also simple. At start-up, the scheduler generates a `json` file to a user-specified path. Providing this same path to the client and simulation is enough for all the elements to auto-configure themselves.

In DEISA, data buffering between the simulation and analysis happens in the the Dask workers memory. In case the analysis is slower than the simulation, DEISA also provides a `buffer_nb` parameter to provide basic flow control and wait for the analysis instead of filling the memory. As in any in situ analytics solution however, the analysis must consume the data in the order it is produced, which can somewhat constrain the way the analysis code must be written, potentially leading to changes with respect to the post hoc version. In our experience however, many scripts can be reused as-is as they use independent analyses in the time dimension.

The PDI based architecture of DEISA makes it possible to extract metadata directly from the simulation code and transfer it to the analysis client. All parameters controlling the coupling (Dask nodes addresses, data description, extraction frequency, etc.) are either directly extracted from the simulation code or set up only once, in the YAML or python. This unicity of information ensures that the simulation and analysis remain in sync and that a small modification of the code does not lead to an invalid analysis.

Like any in situ approach, DEISA makes it possible to only store the final analysis results to disk while post hoc approaches requires to additionally store potentially huge intermediate results. In order to achieve this, the user must however define all its analysis before in situ execution. On the contrary, in post hoc, the analysis can be tuned and adapted once the data is available. By providing a framework making it easy to switch between both approaches, DEISA takes the best of both worlds. It for example supports a workflow where the user can tune its analysis post hoc on the first simulation runs before switching to in situ for later production runs.

Overall DEISA presents a comparable ease of use as the post hoc version. The user neither needs to explicitly set up the parallelization used for the analytics nor any kind of communication between workers. With Dask high-level API there is even no need to parallelize the analysis explicitly by defining the task graph.

Parameter	Value
MPI nodes / Dask worker node	4
MPI process / MPI node	32
Dask worker / Dask worker node	16
Thread / Dask worker	2
MPI process / Dask worker	8
Data size / MPI process	128 MiB
Data size / MPI node	4 GiB
Mean data size / Dask worker node	16 GiB
Data analysis	Listing 7

TABLE I: Fixed parameters used in Experiment #1

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

TABLE II: The three configurations of Experiment #1

B. Performance evaluation

To evaluate the performance of our approach, we have run experiments on the RUCHE cluster (Moulon mesocentre, Paris-Saclay). The cluster is composed of 192 ThinkSystem SD530 servers nodes; each with 2 Intel Xeon Gold 6230 20C @ 2.1GHz CPUs and 180GB of maximum user-allocatable memory. The interconnect uses Omni-Path 100 Gbit/s and the parallel file system the Spectrum Scale GPFS (IOs rate: 9 GB/s).

We have performed three experiments:

- Experiment #1 compares DEISA performance to a baseline with neither IO nor analysis, and to a version using a parallel post hoc analysis with plain Dask.
- Experiment #2 investigates the performance of DEISA more in depth on large and small data sets to explain its behaviour.

For all experiments Dask scheduler and the analytics client are run on the same dedicated node.

a) Experiment #1: Tables I and II summarize the parameters used in the 3 configurations of Experiment #1.

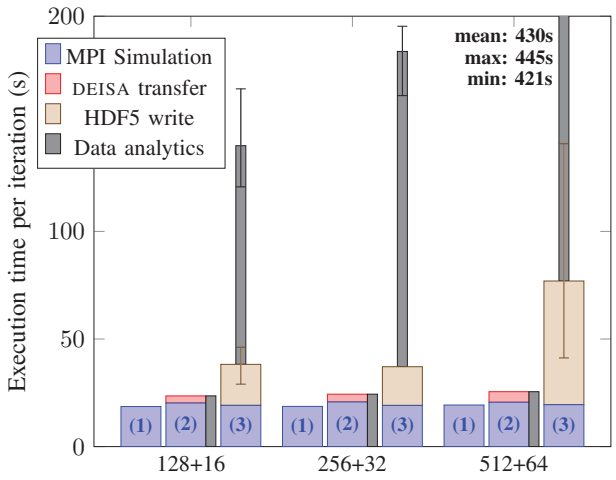

Fig. 3: Execution time for the three configurations of Experiment #1 in three analytics modes each. The width of each bar represents the relative amount of nodes used for each step (simulation vs. analytics).

We use a ratio of Dask workers to simulation processes of 1/8 as it gives a good load balancing between simulation and analysis with enough memory on each Dask worker. We use two compute threads per Dask worker that we found to be the best from a performance point of view in this case. For a different experiment, these values would have to be adjusted by the user depending on the results size and generation frequency, as well as the total compute cost and intrinsic parallelism of the analysis.

Figure 3 presents the simulation execution time in the three configurations of Table II. We present the average execution time over 3 runs as well as an error bar representing the min and max values when standard deviation exceeds 2%. Each run comprises 10 iterations for the cases with no analytics or DEISA, but only 5 in post hoc due to disk space limitations. We excluded the first iteration from the figure since it includes some setup time.

While running the simulation with no output is useless, it provides a baseline for the simulation execution time. This time is relatively stable over the three analysis strategies. DEISA overhead accounts for 21, 23 & 24% of an iteration in Configuration 128+16, 256+32 and 512+64 respectively. HDF5 writing accounts for 36 to 60%, 45 to 52% and 53 to 86% of an iteration in the same three configurations. DEISA performs and scales better than the post hoc version, with an almost constant overhead where post hoc performance decreases rapidly when increasing the problem size.

This can be explained by the scalability of the peak data transfer bandwidth between the simulation and analysis, as illustrated in Table III. For DEISA, data transfers use the network and the available bandwidth is limited by the aggregate bandwidth of all worker nodes 100 Gb/s interfaces. In post hoc,

Case	Configuration	Bandwidth measured	Bandwidth peak	BW ratio
DEISA	128+16	53 Gb/s	100 Gb/s	53%
network	256+32	111 Gb/s	200 Gb/s	55%
bandwidth	512+64	199 Gb/s	400 Gb/s	49%
post hoc	128+16	7.21 Gb/s	72 Gb/s	10%
disk	256+32	15.29 Gb/s	72 Gb/s	21%
bandwidth	512+64	9.56 Gb/s	72 Gb/s	13%

TABLE III: Measured network and disk bandwidth compared to theoretical peak.

data transfers go through disk and the available bandwidth is limited by the 9 GB/s parallel file system bandwidth. For DEISA, the bandwidth scales linearly with the number of nodes, while for post hoc, it does not scale at all and is lower than for DEISA, even with a single node.

Beyond the peak bandwidth consideration, the measures show that DEISA manages to use at least 49% of the peak bandwidth. This is a good performance even though Dask uses Tornado on top of the socket API, which does not take advantage of the full capabilities of OmniPath. In [34] only 37.5 Gb/s out of 100 Gb/s could be used for IPoFabric. In future work we will intend to use UCX [35], supported by Dask, to better leverage high performance network capabilities.

HDF5 performance on the other hand, only reaches between 10 and 21% of the theoretical peak disk bandwidth, with high variability as highlighted in Figure 3. This is partially due to the interference due to other users in a shared cluster. Since DEISA relies on the network and not the file system, it is much less affected by this issue.

Figure 3 also presents the analytics execution time performed in parallel with the MPI solver in DEISA, or post hoc after using data from HDF5 files. We use the same number of Dask nodes for in situ and post hoc, as specified in Table I. This measures include the total time required to execute the analysis task graph, including post hoc specific tasks to read HDF5 files or network transfer in DEISA. Post hoc analytics duration is surprisingly long compared to DEISA analytics. Since we performed the exact same analysis in both cases, we expect this to be due to the HDF5 reads that may be desynchronized, thus less efficient compared to the synchronized writing part. We used HDF5 chunks with a size matching those in Dask (128 MiB), to reduce the risk of performance gaps due to different configurations. Since Dask applies specific optimizations on the task graph, for instance fusing some tasks, it is impossible to distinguish the reading tasks from the computations, thus identify the reading time for these experiments. We have investigated more on that by creating a task-graph that just reads the file and the reading time is almost the difference between the analytics part of post hoc experiments and analytics part of DEISA. Further specific fine tuning may enable to improve post hoc performance, but this is beyond the scope of this paper.

Enabling in situ processing with DEISA shows a significant performance benefit, both on the simulation and analytics side.

Configuration	Total cost per iteration (core.h) Post hoc	DEISA	Cost ratio	Simulation cost per iteration (core.h) Post hoc	DEISA	Cost ratio
128+16	3.16	1.25	39%	1.35	0.83	61%
256+32	6.54	2.38	36%	2.63	1.73	65%
512+64	26.63	4.76	17%	10.94	3.63	33%

TABLE IV: Comparison of the compute resource costs for the whole experiment (simulation + analytics) and for the simulation only in core.hours and cost ratio of the DEISA experiment compared to the post hoc one. For example in Configuration 512+64, DEISA simulation costs 33% of the post hoc simulation

Configuration	1:6	1:21	256:6	256:21
Data size / MPI process	1 MiB	1 MiB	256 MiB	256 MiB
Total nodes	6	21	6	21
MPI node	4	16	4	16
Dask worker nodes	1	4	1	4
client & scheduler node	1	1	1	1
Global data size	128 MiB	512 MiB	32 GiB	128 GiB
Data size / MPI node	32 MiB	32 MiB	8 GiB	8 GiB
Generated tasks	15330	55330	15210	55150

TABLE V: Four configurations used for Experiment #2. All other parameters are kept the same as in Experiment #1.

In Configuration 512+64, DEISA is 3 times faster than post hoc on MPI side (MPI solver plus data transfer over network for DEISA, MPI solver plus parallel HDF5 write for post hoc). The wallclock time of the whole experiment (simulation+analysis) is more than 16 times faster for DEISA than post hoc.

Table IV shows that in terms of compute resource usage, DEISA is already cheaper for Configuration 128+16 as it only uses 39% of the resources required for post hoc. Even without considering the analytics part DEISA is cheaper by using only 61% of the resources required for post hoc for the same experiment. The cost ratio increases with the size, due to a better scalability of the data transfer on the network with DEISA than through disk in post hoc. In Configuration 512+64 DEISA uses only 17% of the resources required for post hoc, for the whole experiment and only 33% while considering only the simulation part.

b) Experiment #2: In this experiment, we investigates the performance of DEISA in depth when either the data size or computer resources vary. We keep most parameters from Experiment #1, but we vary the size of the data per MPI rank from 1 MiB to 256 MiB and we use either 6 or 21 nodes. Table V summarizes the four configurations tested and Figure 4 presents the average execution time over 3 runs of 10 iterations as well as an error bar representing the min and max values when standard deviation exceeds 2%. We also excluded the first iteration here.

On MPI side, we identify the time due to the simulation (line 17 of Listing 3), the time to transfer the data from DEISA adapter to Dask workers, and that to send the required metadata to the scheduler. We also measure the duration of

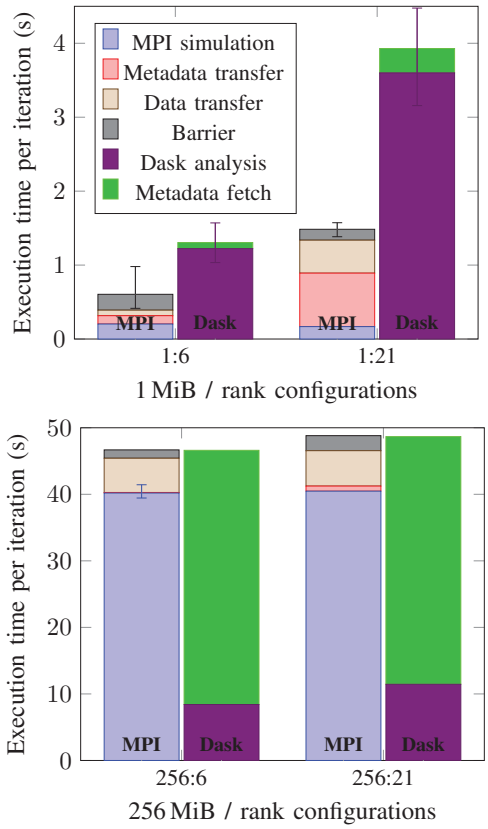

Fig. 4: Detailed timing per time step for DEISA with 1 or 256 MiB/rank and 6 or 21 nodes. For each configuration, the left bar shows timing for the MPI side, and the right timings for Dask side

a barrier inserted just after these communications (line 23 of Listing 3). This barrier captures the time required to re-synchronize the processes after potentially differing time in the communications. Without it, this time would be counted as part of the computation time.

On Dask side, we identify the time required by DEISA metadata adapter to gather all metadata from the scheduler (line 15 of Listing 7) on the one hand and the time required for the submission and actual execution of the task graph on the other hand.

At 1 MiB/rank, the MPI simulation executes much faster than the analysis; this is reversed at 256 MiB/rank. The task granularity has a high impact on Dask performance. At 1 MiB/rank, the average time per task is at most 1ms, which is very small compared to the minimum recommended task duration of 100ms [36] in Dask. With a scheduling overhead of about 1ms per task, scheduling and communication overheads account for more than half the measured time at this granularity. With larger chunks, 256 MiB/rank, and so longer tasks, Dask analytics become faster than the MPI part.

The experiment is run with no maximum `queue` size between DEISA bridges and metadata adapter. Hence, when the

simulation produces data faster than the analysis can consume it (1 MiB/rank configurations), data is buffered in the worker nodes memory and processed after the end of the simulation. The total time to solution is limited by the analytics part. On the other hand, when the simulation produces data slower than the analysis can consume it (256 MiB/rank configurations), Dask spends time waiting for metadata that is not yet produced by the simulation. The total time to solution is limited by the MPI part and Dask workers are idle for more than 4/5 of the iteration; a time that appears as part of the metadata fetch.

At 1 MiB/rank, data and metadata transfer costs are significant. This is mostly explained by the fact that at this scale, communication time is noticeably impacted by network latency. The data transfer performance is also explained by the behaviour of Dask `scatter` used by DEISA to transfer data to the workers. This function directly transfers data to the worker, but it also establishes an additional connection to the scheduler to notify it of the new data reference. For large enough data, this is negligible, but at this scale, this starts to be noticeable. In addition to the latency, another factor impacts network performance for small data sizes. When the data is small, simulation time is too, and data production frequency increases. The high number of requests sent to the scheduler per second can impact its response time. For Configuration 1:6, more than 1920 requests/s are sent to the scheduler, and more than 9116 requests/s for Configuration 1:21. The time required to send metadata becomes almost 7 times longer in Configuration 1:21 than in Configuration 1:6 while the number and size of requests per MPI rank is the same. At 256 MiB/rank, this difference is still visible, but metadata handling represents less than 1.6% of an iteration in the worse case.

The variation in data and metadata transfer time between MPI ranks is measured by the barrier we inserted. For small sizes, this can represent as much time as the mean duration of data + metadata transfer. For larger sizes however, the transfer time becomes more stable and the barrier represents a lower relative amount of time. This can be explained by the existence of a time spent waiting for the availability of Dask network thread on the server when making a request. This time is very irregular and does not seem to depend on the data size.

This bad network performance does not only affect DEISA at the interface between the MPI simulation and Dask analysis. Communications also happen in Dask execution of the task graph. The number of communications grows with the number of tasks, and their efficiency improves with the size of data. Hence, with 4 times more compute resources to compute a graph 4 times bigger, Dask task graph execution is 2.9 times slower at 1 MiB/rank, while this ratio is only of 1.36 at 256 MiB/rank.

Overall, data granularity must be set to a large enough value for DEISA to be efficient. This is however not a DEISA specificity and plain Dask post hoc usage must follow the same rules.

VI. Conclusion

In this paper we have introduced a programming paradigm that combines the BSP model from MPI and the distributed task-based paradigms from Dask. We have presented its implementation in DEISA, and stressed the advantages this introduces in development efforts, performance for in situ simulation data processing. Because the expression of parallelism is abstracted from the actual mapping of data and tasks to compute nodes, an analytics task graph requires minor modifications to move from a sequential post hoc to an in situ execution context.

DEISA combines PDI and Dask features to minimize the required code changes in both the simulation and the analytics code. Turning a post hoc Dask analysis in situ, is both easy from the user point of view and leads to significant performance gain as demonstrated by the experiments. We have shown that even considering only the simulation part is more efficient than post hoc and up to 3 times cheaper.

This paper also stresses some limitations in the prototype mainly related to the centralized Dask architecture that relies on a single scheduler becoming a bottleneck at very large scale or when handling small data sets.

Future work will focus on developing solutions to reduce the pressure on the scheduler, for instance by aggregating metadata on the MPI side between bridges before forwarding the result to the scheduler. For the next steps we will also investigate how to deploy tasks in situ and in transit, and not only in transit as done in this paper, and we will collaborate with computational physicists to develop production use cases, in particular using the large scale plasma simulation code Gysela [37].

References

[1] K.-L. Ma, C. Wang, H. Yu, and A. Tikhonova, "In-situ processing and visualization for ultrascale simulations," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012043, 2007. [Online]. Available: http://stacks.iop.org/1742-6596/78/i=1/a=012043

[2] J. Ahrens, B. Geveci, and C. Law, "ParaView: An End-User Tool for Large Data Visualization," *Visualization Handbook*, Jan. 2005.

[3] M. D. Hanwell, K. M. Martin, A. Chaudhary, and L. S. Avila, "The Visualization Toolkit (VTK): Rewriting the rendering code for modern graphics cards," *SoftwareX*, vol. 1-2, pp. 9–12, Sep. 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352711015000035

[4] H. Childs, "VisIt: An End-User Tool for Visualizing and Analyzing Very Large Data," p. 17.

[5] N. Fabian, K. Moreland, D. Thompson, A. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. Jansen, "The Paraview Coprocessing Library: a Scalable, General Purpose In Situ Visualization Library," in *Large Data Analysis and Visualization Workshop(LDAV'11)*, 2011, pp. 89–96.

[6] B. Whitlock, J. M. Favre, and J. S. Meredith, "Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System," in *11th Eurographics conference on Parallel Graphics and Visualization*, 2011, pp. 101–109.

[7] M. Larsen, J. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison, "The alpine in situ infrastructure: Ascending from the ashes of strawman," in *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, ser. ISAV'17. New York, NY, USA: Association for Computing Machinery, 2017, p. 42–46. [Online]. Available: https://doi.org/10.1145/3144769.3144778

[8] K. Moreland, C. Sewell, W. Usher, L. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K. Ma, H. Childs, M. Larsen, C. Chen, R. Maynard, and B. Geveci, "VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures," *IEEE Computer Graphics and Applications*, vol. 36, no. 3, pp. 48–58, May 2016, conference Name: IEEE Computer Graphics and Applications.

[9] U. Ayachit, B. Whitlock, M. Wolf, B. Loring, B. Geveci, D. Lonie, and E. W. Bethel, "The SENSEI Generic In Situ Interface," in *2016 Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*. Salt Lake City, UT, USA: IEEE, Nov. 2016, pp. 40–44. [Online]. Available: http://ieeexplore.ieee.org/document/7836400/

[10] M. Dreher and B. Raffin, "A Flexible Framework for Asynchronous In Situ and In Transit Analytics for Scientific Simulations." IEEE Computer Science Press, May 2014. [Online]. Available: https://hal.inria.fr/hal-00941413

[11] M. Dreher and T. Peterka, "Decaf: Decoupled Dataflows for In Situ High-Performance Workflows," Argonne National Lab. (ANL), Argonne, IL (United States), Tech. Rep. ANL/MCS-TM-371, Jul. 2017. [Online]. Available: https://www.osti.gov/biblio/1372113-decaf-decoupled-dataflows-situ-high-performance-workflows

[12] ——, "Bredala: Semantic Data Redistribution for In Situ Applications," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2016, pp. 279–288, iSSN: 2168-9253.

[13] J. Lofstead and R. Ross, "Insights for exascale IO APIs from building a petascale IO API," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2013, pp. 1–12, iSSN: 2167-4337.

[14] D. A. Boyuka, S. Lakshminarasimham, X. Zou, Z. Gong, J. Jenkins, E. R. Schendel, N. Podhorszki, Q. Liu, S. Klasky, and N. F. Samatova, "Transparent in Situ Data Transformations in ADIOS," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2014, pp. 256–266.

[15] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, K. Wu, and S. Klasky, "ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, Jul. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711019302560

[16] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O," in *CLUSTER 2012 - IEEE International Conference on Cluster Computing*. Beijing, China: IEEE, Sep. 2012. [Online]. Available: https://hal.inria.fr/hal-00715252

[17] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: an interaction and coordination framework for coupled simulation workflows," *Cluster Computing*, vol. 15, no. 2, pp. 163–181, Jun. 2012. [Online]. Available: https://doi.org/10.1007/s10586-011-0162-y

[18] Y. Wang, G. Agrawal, T. Bicer, and W. Jiang, "Smart: a MapReduce-like framework for in-situ scientific analytics," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2015, pp. 1–12, iSSN: 2167-4337.

[19] H. C. Zanúz, B. Raffin, O. A. Mures, and E. J. Padrón, "In-transit molecular dynamics analysis with Apache flink," in *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. Dallas Texas USA: ACM, Nov. 2018, pp. 25–32. [Online]. Available: https://dl.acm.org/doi/10.1145/3281464.3281469

[20] X. Xing, B. Dong, J. Ajo-Franklin, and K. Wu, "Automated parallel data processing engine with application to large-scale feature extraction," in *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, 2018, pp. 37–46.

[21] E. Dirand, L. Colombet, and B. Raffin, "TINS: A Task-Based Dynamic Helper Core Strategy for In Situ Analytics," in *Supercomputing Frontiers*, R. Yokota and W. Wu, Eds. Cham: Springer International Publishing, 2018, vol. 10776, pp. 159–178, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-319-69953-0_10

[22] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky, "GoldRush: Resource Efficient in Situ Scientific Data Analytics Using Fine-grained Interference Aware Execution," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 78:1–78:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503279

[23] C. Ramon-Cortes, F. Lordan, J. Ejarque, and R. M. Badia, "A Programming Model for Hybrid Workflows: combining Task-based Workflows and Dataflows all-in-one," *Future Generation Computer Systems*, vol. 113, pp. 281–297, Dec. 2020, arXiv: 2007.04939. [Online]. Available: http://arxiv.org/abs/2007.04939

[24] J. A. Cid-Fuentes, S. Sola, P. Alvarez, A. Castro-Ginard, and R. M. Badia, "dislib: Large scale high performance machine learning in python," in *2019 15th International Conference on eScience (eScience)*, Sep. 2019, pp. 96–105.

[25] M. Rocklin, "Dask: Parallel Computation with Blocked algorithms and Task Scheduling," Austin, Texas, 2015, pp. 126–132. [Online]. Available: https://conference.scipy.org/proceedings/scipy2015/matthew_rocklin.html

[26] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 561–577. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/moritz

[27] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive Parallel Programming in Python," *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 25–36, Jun. 2019, arXiv: 1905.02158. [Online]. Available: http://arxiv.org/abs/1905.02158

[28] E. Slaughter and A. Aiken, "Pygion: Flexible, Scalable Task-Based Parallelism with Python," in *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*. Denver, CO, USA: IEEE, Nov. 2019, pp. 58–72. [Online]. Available: https://ieeexplore.ieee.org/document/9062721/

[29] A. Heirich, E. Slaughter, M. Papadakis, W. Lee, T. Biedert, and A. Aiken, "In situ visualization with task-based parallelism," in *Workshop on In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization (ISAV'17)*, ser. ISAV'17. New York, NY, USA: ACM, 2017, pp. 17–21. [Online]. Available: http://doi.acm.org/10.1145/3144769.3144771

[30] "Pdi documentation." [Online]. Available: https://pdi.julien-bigot.fr/master/

[31] C. Roussel, K. Keller, M. Gaalich, L. Bautista Gomez, and J. Bigot, "PDI, an approach to decouple I/O concerns from high-performance simulation codes," Sep. 2017, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01587075

[32] "dask-ml 0.1 documentation - dask_ml.decomposition.IncrementalPCA." [Online]. Available: modules/generated/dask_ml.decomposition.IncrementalPCA.html

[33] Y. Asahi, K. Fujii, D. M. Heim, S. Maeyama, X. Garbet, V. Grandgirard, Y. Sarazin, G. Dif-Pradalier, Y. Idomura, and M. Yagi, "Compressing the time series of five dimensional distribution function data from gyrokinetic simulation using principal component analysis," *Physics of Plasmas*, vol. 28, no. 1, p. 012304, 2021. [Online]. Available: https://doi.org/10.1063/5.0023166

[34] V. C. Barroso, U. Fuchs, and A. Wegrzynek, "Benchmarking message queue libraries and network technologies to transport large data volume in the ALICE O system," in *2016 IEEE-NPSS Real Time Conference (RT)*. Padova, Italy: IEEE, Jun. 2016, pp. 1–5. [Online]. Available: http://ieeexplore.ieee.org/document/7543162/

[35] "OpenUCX — OpenUCX documentation." [Online]. Available: https://openucx.readthedocs.io/en/master/

[36] "Dask documentation - Chunks." [Online]. Available: array-chunks.html

[37] J. Bigot, V. Grandgirard, G. Latu, C. Passeron, F. Rozar, and O. Thomine, "Scaling gysela code beyond 32k-cores on blue gene/q," in *ESAIM: PROCEEDINGS*, ser. 43, vol. CEMRACS 2012, Luminy, France, July 2012, pp. 117–135. [Online]. Available: https://hal.inria.fr/hal-01050322