Dask-Extended External Tasks for HPC/ML In Transit Workflows

Amal Gueroudji, Julien Bigot, Bruno Raffin, Robert Ross

Dask-Extended External Tasks for HPC/ML In Transit Workflows

Amal Gueroudji
agueroudji@anl.gov
Argonne National Laboratory
Lemont, IL, USA

Julien Bigot
julien.bigot@cea.fr
Université Paris-Saclay, UVSQ, CNRS, CEA,
Maison de la Simulation
Gif-sur-Yvette, France

Bruno Raffin
bruno.raffin@inria.fr
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
Grenoble, France

Robert Ross
rross@mcs.anl.gov
Argonne National Laboratory
Lemont, IL, USA

ABSTRACT

In situ workflows are inescapable to fully leverage exascale architectures. They can be complex to build, however, because simulation and data analytics come from two different software ecosystems with their own paradigms and programming models. This work extends the DEISA bridging model between MPI+X simulations and distributed task-based analytics; it introduces the concept of *external tasks* to support the description of analytics graphs spanning multiple timesteps ahead of time while improving scalability. This new approach leads to a straightforward support for *contracts* between the simulation and analytics graph to limit the data transferred to that actually analyzed in a given execution. We implement this approach using Dask and MPI and evaluate it using an end-to-end in-transit workflow that uses an unsupervised ML model for dimensionality reduction. We compare our work with plain Dask postprocessing and with the previous version of DEISA. Our work performs better, up to ×7 for the simulation and ×3 for the analytics compared with DEISA, and is ×18 less costly compared with plain Dask—all of these with similar development efforts.

CCS CONCEPTS

• **Theory of computation → Distributed algorithms**; **Online learning algorithms**; **Massively parallel algorithms**; • **Computing methodologies → Parallel computing methodologies**; **Dimensionality reduction and manifold learning**.

KEYWORDS

DEISA, HPC, Machine Learning, Dask-ml, IPCA, Dask

1 INTRODUCTION

High-performance computing and data analytics (HPC/DA) workflows deal with two different types of applications and two different software ecosystems: MPI+X for the simulations and high-productivity Pythonic frameworks for data analytics. In order to fully leverage the exascale architectures, in situ workflows become inescapable; however, most of the in situ tools are built on the MPI+X model inherited from the host simulations, which makes them complicated to set up. In this work we couple MPI simulations with a data analytics tool to support in situ workflows rather than writing the in situ analytics using the MPI+X models. An earlier contribution already provided a method to couple MPI codes with Dask analytics [13], which is an important step in making in situ workflows easier to build and use. However, it has several limitations, such as scalability issues due to overloading the Dask scheduler with the quantity of metadata that has to be exchanged, as well as difficulties in implementing incremental algorithms.

In this paper we propose an approach that natively couples MPI+X codes and the Dask distributed task-based framework in a producer/consumer configuration to fit HPC workflows where simulation produces data and Dask consumes it. We introduce the *external* tasks concept in Dask to natively integrate simulation data into Dask task graphs. This approach associates an *external* task with each block of data that will be produced by the simulation. These tasks are not schedulable nor runnable by Dask; they are run by an external environment, and their results are sent to the workers. The data produced by the simulation is described as DEISA *virtual arrays* that protect the semantics of exchanged data and give Dask enough information about external tasks output. This approach leads to a straightforward implementation of *contracts* that easily and efficiently filter data. The introduction of external tasks in Dask improves performance and scalability of in-transit workflows compared with DEISA [13] because it reduces the amount of metadata that has to be sent to the scheduler at each timestep. Moreover, it makes abstraction of the time dimension and the fact that the data arrives incrementally, which makes the implementation of incremental algorithms trivial.

We provide an implementation of this approach in an end-to-end in-transit workflow and show how it easily resolves one of the complex challenges in the HPC/DA community, namely, building in situ workflows applying machine learning (ML) models. We have evaluated our work on the French TGCC Irene supercomputer using

a Heat2D miniapp coupled with an unsupervised machine learning model consisting of incremental principal component analysis (IPCA) used for dimensionality reduction. The evaluation has been performed along two axes, performance, and ease of use, compared with postprocessing with plain Dask and in transit with [13]. Our work performs better than these versions (×7, ×2 for the simulation part, and ×4, ×1.2 for the analytics, respectively) and is ×18 less costly than plain Dask—all with the same coding efforts.

The rest of the paper is organized as follows. Section 2 introduces our approach and its implementation. In Section 3 we evaluate our work and compare it with DEISA and plain postprocessing with Dask. In Section 4 we discuss related work and compare it with our work. In the final section, we summarize our conclusions and briefly discuss future work.

2 APPROACH AND IMPLEMENTATION

Before introducing our contributions we summarize important concepts about the Dask distributed and DEISA operation. A Dask distributed cluster has three main actors: client, scheduler, and workers. The client is a Python script using the available Dask APIs to create and submit task graphs to the scheduler. At the reception of the task graph, the scheduler populates its data structures to keep track of the state of the clients, tasks, and workers and then sends the ready tasks to the available workers. The actual work is done by these workers. The tasks created using the Dask APIs are known and correctly defined for use by Dask. A task must include a callable (the function the worker will execute) except for pure data tasks, [1] which are data sent to Dask workers by the clients using the `scatter` system. DEISA [13] uses scatter to send simulation data to a Dask worker. It can submit tasks dependent on that data only when the data is sent to the workers. Lots of metadata has to be sent to the scheduler at each timestep, and the incremental aspects of in situ algorithms must be managed manually. To ensure the coupling, each MPI process is associated with a *bridge*, and the Dask analytics client that submits the task graph is associated with an *adaptor*. The bridges and the adaptor ensure the communication and control of data and metadata between the simulation and the Dask cluster.

In this work, we extend the Dask distributed scheduler to support our newly defined `external` tasks. We define an `external` task as a task that runs in an external environment rather than in Dask. In this work, the external environment is an MPI simulation. From the Dask point of view, those tasks can be seen as pure data tasks because the only known information about them is their output data. In [13], where the task graph is only submitted to Dask once data is sent to Dask via a `scatter`. In contrast, in this work, we push this solution further to make Dask natively support external tasks, which makes it possible to submit graphs dependent on those external tasks in advance (before data is available in the workers' memory).

2.1 Our End-to-End Workflow Architecture

With all the previously defined concepts, our workflow comprises two components in a producer/consumer scheme, where the running MPI simulation represented by $M+1$ processes is the producer and the Dask cluster is the consumer.

[1] https://docs.dask.org/en/latest/futures.html#move-data

At the beginning of the simulation, the bridge at rank 0 connects to the Dask scheduler and sends the DEISA virtual arrays description to the adaptor. The analytics client, connected to the adaptor in Dask, creates Dask arrays that correspond to the DEISA arrays, then makes data selections using the `[]` operator on the DEISA array selecting the pieces needed for analytics. The client then sends the selections back to the bridges (Step 1, Sign contracts, in Figure 1). This is done at the beginning of the workflow, and there is no need to send any metadata to the scheduler at each timestep, thus improving performance compared with work in [13]. All the bridges and the adaptor are synchronized at this step, and they can proceed only when contracts are signed. In other words, the adaptor submits the analytics (Step 2, Submit task graph, in Figure 1) and the bridges know the data they must send for that. Each bridge checks the contracts locally at each timestep. If its data block is needed, the bridge sends it to the preselected worker (Step 3, Send data, in Figure 1). We assume in this work that the data sizes, including the time dimension, are known in advance and the contracts are signed once only at the beginning.

We improved the workflow operation by optimizing several aspects thanks to the newly introduced concepts. Compared with the work in [13], the main changes in the architecture are meant to minimize the load of the centralized scheduler and the way the two components communicate. We have kept the implementation of the bridge built in the Dask client class and set the heartbeat intervals to ∞ since there is no need to keep informing the scheduler about the bridges thanks to external tasks. Moreover, we are able to communicate small metadata, at the beginning of the workflow, with a minimum number of communications. Instead of sending lots of metadata every timestep from all the bridges in the order of $2 * Nbr\_timesteps * nbr\_ranks + 2s\_interval\_heartbeats$ messages, we now send only $1 + nbr\_ranks$ messages at the beginning of the workflow. Those communications correspond to setting up the contract process. Alongside the external tasks and the DEISA arrays, there is no need for more metadata communication. We still go through the scheduler for the communications between the bridges and the analytics client, but this is not a big issue anymore because the communications are done only once at the beginning of the workflow. This is done via two Dask `variables`, instead of *Nbr_ranks* distributed `queues` in [13].

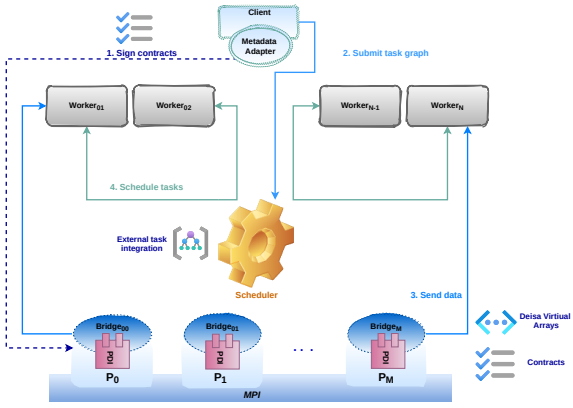


Figure 1: New DEISA end-to-end in-transit workflow

2.2 External Task Integration in Dask

We extended the Dask distributed scheduler with our newly defined *external task* concept. A task in an `external` state is identified by a unique key, and it is not schedulable nor runnable by Dask. Implementing external tasks in the Dask distributed scheduler mainly modifies the client and the scheduler's classes and operations. The `future` class in the client can be seen as a mirror of the tasks in the scheduler. In other words, to create an external task (i.e., a task in an `external` state), we need to create a `future` by specifying a unique `external_key` and activating the DEISA (in situ analytics in Dask) mode by setting the `external` argument to `true`. This will trigger a remote procedure call to the scheduler to create an `external` task. The work in [13] uses a `scatter` system to send data from the DEISA bridges to Dask workers. In this work we have updated its operation to support external tasks. We have added two main parameters to the method: `keys` and `external`. Both are set to `none` by default (to keep supporting the default operation). The `keys` parameter is a list of keys associated with a list of data we want to communicate; `external` is an argument that is forwarded to the `scheduler.update_data` and `Future.__init__` methods that changes the default operation when the `external` mode is activated.

We explain two main points to clarify the need for the modifications we have made to the `scatter` operation. First, let us recall how the Dask scheduler manages the finished tasks. When data is an output of a finished task in Dask, the worker sends a message to the scheduler, including the key of the task and `task-finished` stimulus. Depending on the stimulus, the scheduler triggers different handlers; in this case, it is `handle_task_finished` that is called. This handler triggers the task transition process, unblocking the dependent tasks, and the scheduling continues (waiting tasks become ready, and the scheduler eventually sends them to available resources). Now let us go back to the `scatter`. It has been introduced in Dask to send external data to the cluster. By definition, then, this data does not exist in Dask before it is sent. The associated `key` with this data is created in the `scatter` function itself, so this data can be used in a task graph only after the `scatter` is finished and returned. The way the scheduler manages the data it gets from a `scatter` is different from the way it manages data issued from an ordinary computed task by a worker, even if both of them are considered as tasks in the `memory` state (task result is finished and its result is in the distributed memory). To support external tasks in Dask, we have to make the scheduler handle them as any other finished task. This means that the scheduler will not only update its internal data structures but also trigger the task transition process to unblock the depending tasks. When the DEISA mode is activated, we update the scheduler's internal data structures and trigger the transition process: starting by transitioning the current task state from the `external` to the `memory` state, then making all underlying transitions of depending tasks.

Natively supporting external tasks reduces metadata exchanges between the DEISA bridges and the DEISA adaptor, and thus the load on the scheduler. Moreover, it allows submitting graphs on external data (not known by Dask) even before the external source generates them. This approach makes it possible to submit a whole task graph on simulation data generated over time without waiting for the data to be available, eliminating the need to implement incremental algorithms and manually manage time dependencies.

2.3 DEISA Plugin Configuration

To maintain the good separation of concerns proposed in [18] and used in [13], we have used the PDI data interface to extract simulation data independently from the data handling itself. Other tools such as Conduit [14] could also be used. We have developed and implemented a new PDI DEISA plugin that handles the data facility operation, including connection to Dask, data identification, and communication. Listing 1 represents a complete configuration of this plugin. The scheduler file generated by the scheduler at creation is expected as a value of the `scheduler_info` keyword to get the scheduler's address (Line 10). The `init_on` event corresponds to when we initialize the coupling (Line 11). The `time_step` expects the variable's name that corresponds to time progress in the simulation; it is needed because it will correspond to the index in the time dimension (Line 12). The `deisa_arrays` keyword expects a list of DEISA arrays descriptors, and the `map_in` is a mapping of which local data corresponds to which DEISA array.

2.4 Data Model

Both the simulation and Dask deal with distributed data in memory. On the simulation side, the user (statically) manages the distribution of the data blocks over the MPI ranks. Hence the data can be defined as the value of the corresponding buffer at a given timestep. In Dask, data represents the inputs/outputs of tasks that are dynamically scheduled over the workers. Thus the data in the Dask component can be seen as immutable. To maintain the coherence of the communicated data over the bridging model without violating the data definitions in the two ecosystems, we propose a naming scheme that keeps all the needed information used in the delivery facilities in both components. Instead of creating a new data format jointly used by the simulation and Dask, we propose a virtual data structure, DEISA arrays (Section 2.4.2), to describe the simulation data while maintaining its semantics; and we implement a protocol to communicate this descriptor to Dask and get back data filters called *contracts* (Section 2.4.3).

2.4.1 Naming Scheme. We use available information on the simulation side to create a unique key in Dask for each data block. This key is used when performing the `scatter` operation. Each key contains three main sections: the prefix `deisa`, the data's name, and the block's position in the spatiotemporal decomposition. For example, in `(deisa-temp, (1,3,5))`, `temp` is the name of the data, and $(1, 3, 5)$ is its position in the global decomposition, where the first dimension is time. We use the same domain decomposition in both components by sending to Dask all needed information: the name of the generated array, its sizes, and its subsizes in all dimensions (DEISA virtual array). A single key per block is created on the Dask side, associated with an array with the corresponding sizes, and identified as explained above. An eventual new decomposition is possible on the analytics side using the *rechunking* functionality of Dask arrays.

2.4.2 DEISA Virtual Arrays. A DEISA virtual array is a descriptor of a distributed multidimensional array. This concept is similar to

Dask arrays, HDF5 datasets, or *ds-array* in PyCOMPSs [4], but it is used only for configuration aspects. A DEISA virtual array describes the decomposition of the spatiotemporal domain of a data array generated by simulation. It contains the global sizes in each dimension, including the time dimension, the size of each block (size of the data generated by each MPI process), and the starting indexes of each block. Describing the data this way gives us a global view of the generated data. On the Dask side, a `dask.array` is created from a DEISA array descriptor containing only external data. We create an `external` task per MPI block per timestep. Technically, this is achieved by creating, in DEISA mode, a *future* with a specific key, per MPI block per timestep, and using the key to create a `dask.array`, then gathering all the arrays to create a global `dask.array`. The chunking of this last array corresponds to the spatiotemporal domain decomposition of the DEISA array, matching the distribution and evolution of data in the simulation.

The DEISA virtual arrays have been added to the configuration file of the DEISA plugin. Listing 1 shows an example of a configuration file of the DEISA plugin. On Line 13 the list of the DEISA virtual arrays starts. For instance, here we have only the `Gtemp` array constructed of blocks of the `temp` data. Each MPI process exposes its local 2D `temp` data to the DEISA plugin every timestep (without any copy). This data is mapped to the global deisa array `Gtemp`. From the `subsize` and `start` keywords, its position in the decomposition is known and used to create a unique key corresponding to a waited external task in Dask.

```
1  metadata: {step: int, cfg: config_t, rank: int}
2  data:
3    temp: # the main temperature field
4      type: array
5      subtype: double
6      size: [ '$cfg.loc[0]', '$cfg.loc[1]' ]
7  plugins:
8    mpi: # get MPI rand and size
9    PdiPluginDeisa:
10     scheduler_info: scheduler.json
11     init_on: init
12     time_step: $step
13     deisa_arrays: # Deisa Virtual arrays
14       G_temp: # Field name
15         type: array
16         subtype: double
17         size:
18           -'$cfg.maxTimeStep'
19           -'$cfg.loc[0] * ($rank % $cfg.proc[0])'
20           -'$cfg.loc[1] * ($rank / $cfg.proc[0])'
21         subsize: # Chunk size
22           -1
23           -'$cfg.loc[0]'
24           -'$cfg.loc[1]'
25         start:  # Chunk start
26           -$step
27           -'$cfg.loc[0] * ($rank % $cfg.proc[0])'
28           -'$cfg.loc[1] * ($rank / $cfg.proc[0])'
29         +timedim: 0 # A tag for the time dimension
30       map_in: # Deisa array mapping
31         temp: G_temp
```

Listing 1: Data description in DEISA YAML file.

2.4.3 Contracts. The concept of contracts was proposed in [17] for automatic data filtering for in situ analysis. We implement a similar protocol to communicate the DEISA arrays to the adaptor and return the data selection that the analytics client needs. We have a double synchronization between the two components in the contracts:

- The adaptor waits for the bridge in rank 0 to send the DEISA arrays, to check the data made available for sharing by the simulation. It creates the corresponding Dask arrays with the same naming scheme, makes a selection on needed data using the `[]` operator, and sends back the filters to all the bridges.
- All the bridges, including the bridge in rank 0, will be blocked before sending data to the workers until the reception of the filters. Each bridge checks whether its current data block is included or includes a part of the needed data. If this is the case, it will create a corresponding key to identify that data (with the naming scheme) and send it to a predefined worker.

The contract operation also checks whether the data needed for analytics is made available by the simulation and whether the selections are valid.

3 EVALUATION

We used the Irene supercomputer in the CEA TGCC center. We used the skylake partition with 1,653 nodes, each with 2 CPUs: CPU: 2x24-cores Intel Skylake @ 2.7 GHz (AVX512), 180 GB memory per node. Irene has a total of 79,344 cores. The compute nodes are connected through an EDR InfiniBand network. This high-throughput (100 Gb/s) and low-latency network is used for I/O and communications among supercomputer nodes. Irene uses a Lustre parallel distributed file system. We use a modified HeatPDE miniapp to evaluate our work along with an unsupervised learning model that consists of PCA.

Two factors motivate this choice. First, we show the ease of use and performance of a representative HPC/DA workflow operation with our newly developed system in general and HPC/ML integration in particular. Second, our choice is also motivated by the real need for PCA models in HPC workflows such as the work in [1], which uses this model to reduce the dimensionality of the five-dimensional array produced by Gysela fusion simulation [15] in a post hoc configuration.

3.1 Principal Component Analysis

The `Dask-ML`[2] library provides scalable machine learning algorithms in Python using the Dask framework and machine learning libraries such as `scikit-learn`.[3] Dask-ML provides a parallel implementation of the PCA based on the singular value decomposition (SVD) algorithm.[4] The PCA needs all the data to be processed in the main memory, which is impossible for large datasets or in situ processing (since data comes as the simulation progresses). Incremental PCA (IPCA) [5] responds to this limitation by processing the data in a minibatch fashion. Furthermore, the IPCA algorithm has a constant memory complexity.

[2] https://ml.dask.org/
[3] https://scikit-learn.org/stable/
[4] https://ml.dask.org/modules/generated/dask_ml.decomposition.PCA.html
[5] https://ml.dask.org/modules/generated/dask_ml.decomposition.IncrementalPCA.html

3.2 Multidimensional Incremental PCA

We implemented a new version of IPCA that takes a multidimensional array and computes its PCA incrementally. Thus it can be used for both the post hoc and in situ versions. Moreover, we have provided a similar interface to the sequential PCA by hiding the incremental execution of IPCA.[6]

```
from dask_ml.decomposition import InSituIncrementalPCA
from dask_interface import Deisa
# Initialize the Deisa
Deisa = Deisa(scheduler_info, config_file)
client = Deisa.get_client()
# Get data descriptor as a list of Deisa arrays object
arrays = Deisa.get_deisa_arrays()
# Filter data
gt = arrays["global_t"][...]
arrays.validate_contract()
ipca=InSituIncrementalPCA(n_components=2,copy=False,
     svd_solver='randomized')
ipca = ipca.fit(gt, ["t", "X", "Y"], ["X"], ["Y"])
# Submit the task graph to the scheduler
explained_variance ,singular_values  = client.persist([
      pca.explained_variance_ , pca.singular_values_])
```

Listing 2: In situ incremental PCA.

We have used the `xarray` library to stack the features' dimensions together and the samples' dimensions together to get a 2D array at the end and use the incremental PCA over the time dimensions. The `fit(ndarray, label_list, feature_labels, sample_labels)` method takes the same parameters. Listing 2 shows the in situ version of IPCA. The creation and the submission of the task-graph are the same for in-transit and post hoc versions.

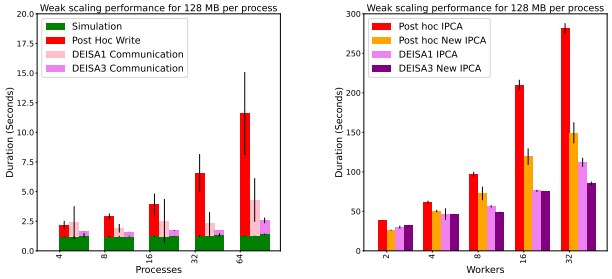
3.3 Experiments

We evaluate our work compared with the previous prototype [13] and post hoc analytics. In the different figures, the results for the DEISA prototype [13] are referenced with DEISA1, and the full new version with a 60*seconds* heartbeat interval is DEISA2 and a ∞ heartbeat interval in DEISA3. Those experiments have been performed on the Irene supercomputer. We used the heat equation solver miniapp for the three implementations of DEISA. For each experiment, we had three runs of 10 timesteps. We have fixed the number of processes per node to two.

- **Experiment I** compares our work (DEISA3) performance with DEISA1 and with parallel post hoc analysis with plain Dask (DASK) using the old version of the incremental PCA presented in Section 3.1 and using the newly developed version presented in Section 3.2.
- **Experiment II** investigates the variability in DEISA1, DEISA2, and DEISA3.

3.3.1 Experiment I. In this section we compare the performance of DEISA3 and DEISA1 and post hoc performance using plain Dask. We compare the results of the IPCA (named IPCA in the figures) with the results we got with the multidimensional version of IPCA presented in Section 3.1 (named new IPCA in the figures). We show here results only for 128 MiB block size because they are considered as the optimal size for the Dask tasks.

[6]https://github.com/GueroudjiAmal/dask-ml



(a) Simulation **(b) Analytics**

Figure 2: Weak-scaling average simulation, communication, and I/O times per iteration for 128 MiB per process on the left and the analytics on the right

Figure 2 summarizes weak-scaling performance for both the simulation and analytics. The subfigure on the left shows results for the simulation side. The x-axis represents the processes, and the y-axis shows the maximum duration per iteration averaged over ranks and runs. The error bar represents the standard deviation. We noticed that the first iteration of the post hoc version was longer than the others. We expect that it is due to file creation. We have computed only the mean and the standard deviation over the remaining iterations. The simulation (in green) weak scales perfectly while we have different patterns in the I/O phases as expected. The communication times for DEISA3 are better than for DEISA1 because of all the metadata that needs to be sent to the scheduler in DEISA1 that we have addressed in DEISA3 with our newly introduced concepts. The in situ versions weak scale better than the post hoc version, which gets limited by the PFS. In DEISA we take advantage of the aggregated bandwidth on compute nodes.

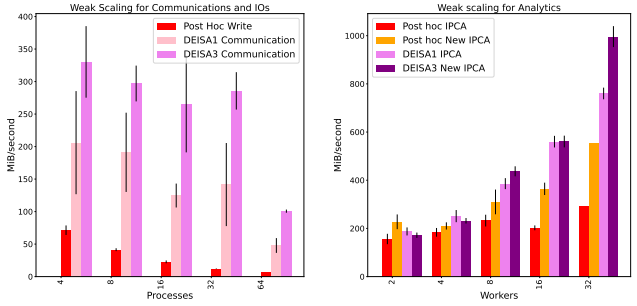
Subfigure 2b shows the weak-scaling results for the analytics part. The first bar from the left of each scale (in red) represents analytics time for the post hoc version with IPCA presented in Section 3.1. The second bar from the left (in orange) shows analytics time for the post hoc version with the new version of IPCA presented in Section 3.2. The third bar from the left (in violet) shows analytics time for the old version of DEISA (DEISA1), and the last bar from the left (purple) shows results for the new version of DEISA (DEISA3) The x-axis of each subfigure represents the variation of the Dask workers from 2 to 32. The y-axis represents the duration in seconds of the analytics. The DEISA analytics time includes compute time and waiting for the data from the next step. The post hoc time includes reading the data from the disk and analysing the data. We have chunked the HDF5 files and used the same chunking in the analytics. The represented values are the mean duration over the three runs. The bar errors are the standard deviation.

For the different chunk sizes, for small scales, DEISA versions are comparable to post hoc versions. Post hoc with our new IPCA is even a bit more efficient than DEISA when the number of Dask workers is two (in the same node, though). When increasing the problem size, DEISA versions perform better than post hoc.

Our new version of IPCA scales better than the old version, both in post hoc and in situ experiments. For post hoc cases, the new

IPCA version is almost twice as fast in some cases. We expect this is due to how we submit tasks to Dask in the new IPCA. Instead of submitting the tasks for each `partial_fit`, in the new version of IPCA, we create the graph of the `partial_fit` for all iterations and submit a single task graph to Dask. Doing so lets Dask optimize the execution of all the tasks over iterations and ovoids repetitive and unnecessary computations. For instance, if a given data is needed by two tasks submitted in two separate task graphs, Dask will perform two disk accesses, one for each submission. If those two tasks are in the same task graph, however, the data will be read only once and used by all the tasks present in the graph needing it. This is only one example, and Dask may perform more optimizations.

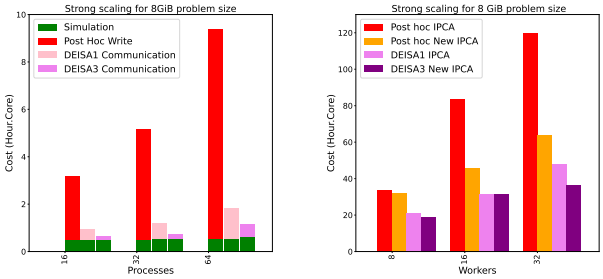
This is also beneficial for in situ analytics. However, it is less visible because the time spent waiting for the simulation data is included, and the time spent running tasks in situ is usually short compared with the time spent reading data from disk. To check the efficiency of the different methods over configurations, we have fixed the number of processes and represented the efficiency in *mebibytes per second* (MiB/s). The values represented are the mean and the standard deviation while changing the size of the data per MPI process, thus the size of the chunks in Dask analytics. The results are shown in Figure 3. In Subfigure 3a we have the bandwidth in MiB/s from the simulation side. The x-axis represents the processes, and the y-axis is the bandwidth in MiB/s. The first bar from the left for each scale (in red) represents the HDF5 write; in the middle (in pink) is DEISA1 communications; and in the right (in violet) is the DEISA3 communications. For the post hoc case, the bandwidth gets twice lower when doubling the number of processes, and this corresponds to our observations regarding the efficiency of post hoc while increasing the problem size. For the in situ cases, the bandwidth is fairly stable until 64 processes. Remember that for the in situ cases, we measure the `scatter` operation time that performs both one communication to the worker (sending data) and one communication to the scheduler (informing the scheduler about the new data in the worker memory). Thus, we cannot achieve the theoretical performance of the aggregated bandwidth.



(a) Simulation **(b) Analytics**

Figure 3: Bandwidth in MiB/s for both the simulation and the analytics side

Subfigure 3b represents the computed bandwidth for the analytics part (MiB/s) when the number of the Dask workers varies between 2 and 32. The x-axis represents the variation of Dask workers, and the y-axis is the bandwidth in MiB/s. Here again, the post hoc versions include reading data from the disk, and the in situ versions include waiting for simulation data to be computed. For each scale, the first bar from the left represents the results of the post hoc analytics with the old IPCA (red), the second bar represents the results of the post hoc with the new version of the IPCA (orange), the third bar represents the results of the DEISA1 with the old IPCA (violet), and the last bar the results of DEISA3 with the new IPCA (purple). In the first scale, the post hoc version with the new IPCA has a slightly better performance than all the others; and starting with 4 workers, the in situ versions become better. The new version of the IPCA is more efficient than the old version in the post hoc cases. This may be due to the optimizations in the task graph. For in situ cases, the two versions are comparable until the last scale (32 workers), where we see a big difference between the two versions. In this figure the post hoc with the old IPCA performance is almost stable when increasing the size of the problem, which is not the case for the new version of the IPCA in either the post hoc or the in situ versions. But we can only affirm that the new IPCA in post hoc performs better when increasing the problem size. The exact reason for this behavior still is under investigation.

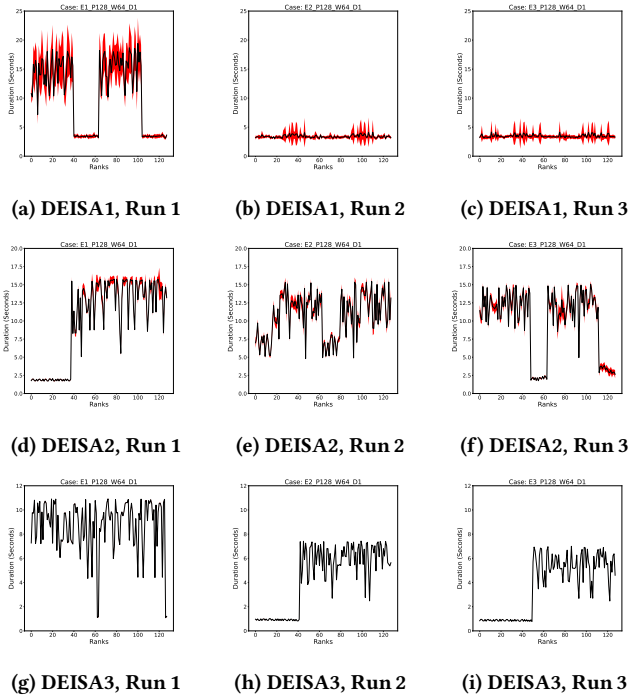


(a) Simulation **(b) Analytics**

Figure 4: Strong-scaling results represented in core-hours for the simulation and the analytics

Figure 4 represents the strong-scaling results in core-hours for the simulation and the analytics sides. we have fixed the problem size to 8 GiB and varied the processes from 16 to 64. Subfigure 4a shows results for the simulation side. The simulation strong scales perfectly. Post hoc writes are more costly than DEISA communications, and the cost increases with the number of processes. In the largest configuration, post hoc write per iteration is 18 times more costly than DEISA3: in situ workflows are less costly than post hoc workflows. DEISA3 is more efficient than DEISA1 and strong scales better. Subfigure 4b shows results for the analytics side. Post hoc versions are more costly compared with the in situ configuration again. The cost of the post hoc analytics with the old version of IPCA increases linearly with the number of processes. For the new version of IPCA in the post hoc configuration, it strong scales better and thus costs less than the old version. The in situ versions have almost the same cost for a fixed number of workers. The cost increases with the number of workers but is still better than post hoc versions; we expect that this is due to communications and the worker placement over the supercomputer. In the largest configuration, the post hoc with the old version of IPCA is

almost 3.5 times more costly than DEISA3 with the new version of IPCA.



(a) DEISA1, Run 1 **(b) DEISA1, Run 2** **(c) DEISA1, Run 3**

(d) DEISA2, Run 1 **(e) DEISA2, Run 2** **(f) DEISA2, Run 3**

(g) DEISA3, Run 1 **(h) DEISA3, Run 2** **(i) DEISA3, Run 3**

Figure 5: Average communication time per iteration for DEISA3, DEISA2, and DEISA1 experiments. The number of processes is fixed to 128, and the size of the data to 1 GiB per process. We show results for the 3 runs.

3.3.2 Experiment II. In these experiments, we were interested in the communication time to study the variability aspect in the in situ analytics. We investigated this variability by checking the mean duration of the communications per rank for DEISA1, DEISA2, and DEISA3 and show the results in Figure 5. We separated the results from each run. We submitted the runs independently, so we do not have control over the allocated nodes, but we may get the same allocation multiple times because of the way Slum works. The x-axis of each subfigure represents the MPI ranks, and the y-axis shows the communication time per rank averaged over iterations (black line). The standard deviation over iterations is represented as a red band. We notice that there is variability over the 3 runs for all versions, but overall we notice the red band more in DEISA1 and less in DEISA2, and we do not see it in DEISA3. The variability over ranks may be due to the node allocation of this experiment and to the physical distance of simulation nodes from the workers and the scheduler nodes, which may vary along allocations and affect the performance. The Skylake partition's compute nodes are connected through an EDR InfiniBand network in a pruned fat-tree topology. If the scheduler, which is always in the first node of our allocation, is connected to a switch different from some of the simulation nodes, the latency and hence the time to send the messages will increase with the distance (the number of switches that a message has to go through before getting to the workers and the scheduler), and the bandwidth may get smaller when we go higher in the tree. In some subfigures, we have the same pattern of variability (for instance in Subfigures 5h and 5i and Subfigures 5b and 5c). This makes us think that they may have the same allocations of at least nodes connected to the same switches. We have checked the logs and found that all four experiments in Subfigure 5h and Subfigure 5i have the exact same allocation. The nodes of the previous experiments are connected to two different switches, which may explain some of the observed variability over processes. Note that the scheduler is launched in the first node of the allocation and the client in the second node; the workers are launched starting from the third node, and then the simulation processes are launched in the rest of the nodes. In this case, the scheduler, the client, the workers, and some of the processes are connected to the same switch, while the rest are connected to another one. The centralized scheduler worsens the performance. Remember that in DEISA1 we have kept the heartbeat interval of the bridges at default of 5 seconds. This frequency, alongside the frequent metadata sent to the scheduler, causes more variability per iteration due to the load on the scheduler in DEISA1. Indeed the red band, which represents the standard deviation per iteration, is more visible in DEISA1 experiments, less in DEISA2, and absent in DEISA3. This is thanks to the improvements, mainly the external tasks in Dask. Less metadata and fewer heartbeat messages coming from the bridges reduce the communication time variability. The takeaway from those experiments is that we could improve performance by minimizing the frequency of messages sent to the scheduler from the bridges. Doing so does not affect the operation of Dask, because the role of the bridges is to send data to the workers only, without submitting any tasks to the scheduler. Thus the scheduler does not need to know whether they still need results as they do not wait for any. The only variability that we still encounter is the one related to the placement of the process, scheduler, and workers, which can be a subject of future contributions.

4 RELATED WORK AND DISCUSSIONS

This work is at the intersection of three main backgrounds: HPC, in situ, and big data analytics. Our goal is to bring big data productivity and HPC performance together in an in situ configuration for exascale workflows [12]. In this paper we follow up on the work proposed in [13] that already couples MPI simulations with Dask analytics in transit. By integrating the *external* tasks in the Dask philosophy, we overcome almost all the limitations found in DEISA and provide native support for simulation data in Dask task graphs.

In the literature, we find earlier attempts to couple MPI with big data tools such as SMART [19] that uses a MapReduce and [21] that uses Flink stream processing for in-transit analysis. However, the model provides poor control of data partitioning that is not well adapted to support efficient parallelization of patterns such as stencil computations. In contrast to those works, our proposition is built on a task-based model, which offers more flexibility and dynamicity in writing distributed algorithms. Distributed task-based frameworks such as Ray or Parsl [2] are examples of tools that can be used also.

The in situ/in-transit paradigm is related to the HPC community rather than the big data one. It is considered a good alternative to postprocessing in HPC workflows that bypasses the disk accesses

and thus the I/O bottleneck. Visualization tools with in situ support such as ParaView Catalyst and Visit-libsim11 [11, 20] or more generic tools [3, 6–9, 16] are all built on or use the MPI programming model, usually inherited from the HPC simulation. While MPI+X models are best suited for regular algorithms, which is usually the case for HPC simulations, using them to write irregular algorithms is not trivial, which is usually the case for data analytics pipelines.

Attempts to use task-based programming for in situ analytics are restricted to shared memory using Intel TBB in TINS [5] and OpenMP in Goldrush [22]. PyCOMPSs [10] couples MPI with distributed task programming; however, this is done by launching an MPI executable from a task rather than considering the two paradigms loosely coupled and running together in the same workflow.

5 CONCLUSION AND FUTURE WORK

In this work, we address the challenge of building in-transit HPC/ML workflows while keeping MPI simulation performance and bringing in the high productivity of data analytics tools such as Dask. We have proposed a generalized approach that can be applied to other task-based tools and have provided an end-to-end workflow implementation with Dask. Our main contribution is the introduction of external tasks that natively integrate simulation data on a Dask task graph. Our work outperforms postprocessing with plain Dask and the original in-transit DEISA work. Moreover, it allows ahead-of-time task submission on simulation data, which makes the implementation of incremental algorithms trivial. The next step is to use our work to couple real HPC simulations alongside other ML models. Note that the external tasks are more general and could be used for any external environment such as in digital twins workflows.

ACKNOWLEDGMENTS

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. It received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 800945 — NUMERICS — H2020-MSCA-COFUND-2017.

REFERENCES

[1] Yuuichi Asahi, Keisuke Fujii, Dennis Manuel Heim, Shinya Maeyama, Xavier Garbet, Virginie Grandgirard, Yanick Sarazin, Guilhem Dif-Pradalier, Yasuhiro Idomura, and Masatoshi Yagi. 2021. Compressing the time series of five dimensional distribution function data from gyrokinetic simulation using principal component analysis. *Physics of Plasmas* 28, 1 (2021), 012304. https://doi.org/10.1063/5.0023166 arXiv:https://doi.org/10.1063/5.0023166

[2] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. 2019. Parsl: Pervasive Parallel Programming in Python. *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (June 2019), 25–36. https://doi.org/10.1145/3307681.3325400 arXiv: 1905.02158.

[3] E. Wes Bethel, Burlen Loring, Utkarsh Ayachit, David Camp, Earl P. N. Duque, Nicola Ferrier, Joseph Insley, Junmin Gu, James Kress, Patrick O'Leary, David Pugmire, Silvio Rizzi, David Thompson, Gunther H. Weber, Brad Whitlock, Matthew Wolf, and Kesheng Wu. 2022. The SENSEI Generic In Situ Interface: Tool and Processing Portability at Scale. In *In Situ Visualization for Computational Science*, Hank Childs, Janine C. Bennett, and Christoph Garth (Eds.). Springer International Publishing, Cham, 281–306.

[4] J. Alvarez Cid-Fuentes, S. Sola, P. Alvarez, A. Castro-Ginard, and R. M. Badia. 2019. dislib: Large Scale High Performance Machine Learning in Python. In *2019 15th International Conference on eScience (eScience)*. 96–105. https://doi.org/10.1109/eScience.2019.00018

[5] Estelle Dirand, Laurent Colombet, and Bruno Raffin. 2018. TINS: A Task-Based Dynamic Helper Core Strategy for In Situ Analytics. In *Supercomputing Frontiers*, Rio Yokota and Weigang Wu (Eds.). Vol. 10776. Springer International Publishing, Cham, 159–178. https://doi.org/10.1007/978-3-319-69953-0_10 Series Title: Lecture Notes in Computer Science.

[6] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, Robert Sisneros, Orcun Yildiz, Shadi Ibrahim, Tom Peterka, and Leigh Orf. 2016. Damaris: Addressing Performance Variability in Data Management for Post-Petascale Simulations. *ACM Trans. Parallel Comput.* 3, 3, Article 15 (oct 2016), 43 pages. https://doi.org/10.1145/2987371

[7] Matthieu Dreher. 2015. *Méthodes In-Situ et In-Transit : vers un continuum entre les applications interactives et offines à grande échelle.* These de doctorat. Université Grenoble Alpes (ComUE). https://www.theses.fr/2015GREAM076

[8] Matthieu Dreher and Tom Peterka. 2016. Bredala: Semantic Data Redistribution for In Situ Applications. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. 279–288. https://doi.org/10.1109/CLUSTER.2016.30 ISSN: 2168-9253.

[9] M. Dreher and T. Peterka. 2017. *Decaf: Decoupled Dataflows for In Situ High PerformanceWorkflows.* Technical Report ANL/MCS-TM-371. Argonne National Lab. (ANL), Argonne, IL (United States). https://doi.org/10.2172/1372113

[10] H. Elshazly, F. Lordan, J. Ejarque, and R. M. Badia. 2020. Performance Meets Programmabilty: Enabling Native Python MPI Tasks In PyCOMPSs. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 63–66. https://doi.org/10.1109/PDP50117.2020.00016 ISSN: 2377-5750.

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

[12] Amal Gueroudji. 2023. *Distributed Task-Based In Situ Data Analytics for High-Performance Simulations.* Theses. Université Grenoble Alpes [2020-....]. https://theses.hal.science/tel-04194958

[13] Amal Gueroudji, Julien Bigot, and Bruno Raffin. 2021. DEISA: Dask-Enabled In Situ Analytics. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 11–20. https://doi.org/10.1109/HiPC53243.2021.00015

[14] Cyrus Harrison, Matthew Larsen, Brian S. Ryujin, Adam Kunen, Arlie Capps, and Justin Privitera. 2022. Conduit: A Successful Strategy for Describing and Sharing Data In Situ. In *2022 IEEE/ACM International Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*. 1–6. https://doi.org/10.1109/ISAV56555.2022.00006

[15] Guillaume Latu, Yuuichi ASAHI, Julien Bigot, Tamás Fehér, and Virginie Grandgirard. 2018. Scaling and optimizing the Gysela code on a cluster of many-core processors. In *SBAC-PAD 2018, WAMCA workshop (SBAC-PAD 2018 proceedings)*. Lyon, France. https://hal.inria.fr/hal-01719208

[16] Michael Laufer and Erick Fredj. 2022. High Performance Parallel I/O and In-Situ Analysis in the WRF model with ADIOS2. *arXiv preprint arXiv:2201.08228* (2022).

[17] C. Mommessin, M. Dreher, B. Raffin, and T. Peterka. 2017. Automatic Data Filtering for In Situ Workflows. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 370–378. https://doi.org/10.1109/CLUSTER.2017.35 ISSN: 2168-9253.

[18] Corentin Roussel, Kai Keller, Mohamed Gaalich, Leonardo Bautista Gomez, and Julien Bigot. 2017. PDI, an approach to decouple I/O concerns from high-performance simulation codes. (Sept. 2017). https://hal.archives-ouvertes.fr/hal-01587075 working paper or preprint.

[19] Y. Wang, G. Agrawal, T. Bicer, and W. Jiang. 2015. 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*. 1–12. https://doi.org/10.1145/2807591.2807650 ISSN: 2167-4337.

[20] Brad Whitlock, Jean M. Favre, and Jeremy S. Meredith. 2011. Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System. In *11th Eurographics conference on Parallel Graphics and Visualization* (Llandudno). 101–109.

[21] Henrique C. Zanúz, Bruno Raffin, Omar A. Mures, and Emilio J. Padrón. 2018. In-transit molecular dynamics analysis with Apache flink. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. ACM, Dallas Texas USA, 25–32. https://doi.org/10.1145/3281464.3281469

[22] Fang Zheng, Hongfeng Yu, Can Hantas, Matthew Wolf, Greg Eisenhauer, Karsten Schwan, Hasan Abbasi, and Scott Klasky. 2013. 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* (Denver, Colorado) *(SC '13)*. ACM, New York, NY, USA, Article 78, 12 pages. https://doi.org/10.1145/2503210.2503279