Python Data Processing on Supercomputers for Large Parallel Numerical Simulations

Adrien Vannson
EPFL, ENS de Lyon

Master's Project advised by:

Bruno Raffin
Inria Grenoble, DataMove team

and supervised by:

Sanidhya Kashyap
EPFL

and

Aurélien Garivier
ENS de Lyon

10 February 2025 – 8 August 2025

Contents

I Introduction ... 3
I 1 Structure of this report ... 3

II State of the art ... 3
II 1 Task-based programming ... 3
II 1.1 Shared memory task-based programming ... 4
II 1.2 Distributed task-based programming ... 4
II 2 *In situ* data processing ... 6
II 2.1 PDI ... 7
II 2.2 Deisa ... 7
II 2.3 Reisa ... 8

III Experiments ... 8
III 1 Jean-Zay ... 8
III 2 Leonardo ... 8
III 3 Grid5000, `gros` cluster ... 8

IV Doreisa: Dask-on-Ray Enabled In-Situ Analytics ... 9
IV 1 Doreisa v1: Deisa-like system using Dask-on-Ray ... 9
IV 1.1 Design ... 9
IV 1.2 User API ... 9
IV 1.3 Performance evaluation ... 10
IV 2 Building the Dask array ... 11
IV 3 Doreisa v2: Distributed scheduler ... 12
IV 3.1 Design ... 12
IV 3.2 `ObjectRefs` sharing, nested `ObjectRefs` ... 13
IV 3.3 Evaluation ... 13
IV 3.4 Task graph partitioning ... 14
IV 3.5 Finding the bottleneck ... 15
IV 4 Doreisa v3: Asynchronous task graph processing ... 15
IV 4.1 User API ... 16
IV 4.2 Performance evaluation ... 16
IV 5 Doreisa v4: *In transit* analytics ... 17

V Performance evaluation ... 18
V 1 Simulation producing bigger chunks ... 18
V 2 Forced data movements ... 18
V 3 Integration with Parflow ... 19

VI Development of Doreisa ... 19
VI 1 Software Engineering practices ... 19
VI 2 Challenges ... 20
VI 2.1 Technical issues ... 20

VII Conclusion and future work ... 20

Bibliography ... 20

I Introduction

As the world's most powerful supercomputers have reached a new computational power threshold, now exceeding the exaflop [1], it has become possible to run large-scale simulations of complex phenomena with unprecedented precision. This has opened new applications in a wide range of scientific domains. Large simulation codes such as Gysela [2] – a gyrokinetic code for plasma simulation – and Parflow [3], [4], [5], [6] – a physics-based integrated hydrologic model simulating hydrological, groundwater and land-surface processes – leverage this computational power.

These simulations typically follow an iterative process, generating a large amount of data at each iteration. The data produced is then used by scientists for visualization and/or analysis.

Post hoc analytics involves storing simulation data to disk and loading it later for analysis. It is straightforward to implement as the simulation and analysis workflows are separated, avoiding the need for a complex coupling. However, the performance of *post hoc* analytics is fundamentally limited by disk I/O bandwidth.

Between 2009 and 2022, the ratio of storage I/O bandwidth to computing power of the first three supercomputers listed in Top500 [1] has been divided by 25 [7]. The Jean Zay supercomputer [8] offers a peak disk write speed of 1.1 TB/s [9]. On the other hand, modern simulations produce unprecedented volumes of data. Gysela generates 5D arrays of about 100 TB at each iteration [10], highlighting the need for more efficient alternatives.

To avoid this I/O bottleneck, *in situ* analytics consists of performing the analysis on the fly, directly on the nodes where the simulation runs. *In situ* solutions can be complex to implement since simulation and data analysis programs rely on fundamentally different technologies and paradigms. Simulation codes are performance-critical and are developed using efficient technologies such as `C++` with MPI. On the other hand, data analysis pipelines are often designed using high-level programming languages such as Python, relying on a rich ecosystem of efficient libraries.

This Master's project introduces Doreisa – Dask-on-Ray Enabled In Situ Analytics – a scalable and efficient system for in situ analysis of simulation data. Doreisa offers a Dask-based interface, allowing a simple definition of analysis tasks. Experiments show that Doreisa scales well to hundreds of nodes, allowing it to run on the Jean Zay supercomputer. Doreisa has been integrated with Parflow, ensuring that it can effectively analyze the data produced by large parallel numerical simulations.

I 1 Structure of this report

Section II presents the research projects, tools that will be useful for Doreisa. Section III describes the supercomputers used for development and performance evaluation. The first section of Section IV introduces a functional Deisa-like system with limited performance. The following subsections detail improvements made to this solution, until the most advanced version. Section V evaluates the performance of Doreisa in various scenarios. Section VI briefly mentions practical aspects of the development of Doreisa.

The Doreisa implementation is available on Github [11]. All the experiments are available on Github as well [12].

II State of the art

II 1 Task-based programming

Task-based programming is a high-level programming model allowing the definition of a computation as a set of tasks that need to be executed and dependencies between them. The tasks are scheduled dynamically at runtime, taking advantage of the parallelism offered by the system executing the computation.

More specifically, the tasks can be represented in a directed acyclic graph where each node represents a task, and each edge represents a dependency between two tasks. The graph may be fully known at the beginning of the execution, or eagerly discovered as the

computation progresses. A scheduling method is used to determine the execution order and place of the tasks, potentially taking into account the availability of resources and data locality. The tasks are executed in parallel, if possible. Once a task completes, its result can be passed to other tasks, as needed for the computation.

Shared memory task-based programming systems are designed to run on a system where the memory is shared between all the processing units. Distributed task-based programming systems extend task-based programming to distributed systems: several nodes cooperate to execute the computation, and need to communicate to each other as they do not directly share any memory.

II 1.1 Shared memory task-based programming

Shared memory task-based programming systems often rely on the *work stealing* scheduling algorithm: each processor has a list of tasks to perform. When a task creates subtasks, they are added to the list of the processor. Idle processors attempt to steal pending tasks from busy processors to execute them.

The following paragraphs describe two systems for shared memory task-based programming.

Cilk [13] is an extension of the C language providing support for task-based programming. Users define tasks as C functions that are able to start new tasks themselves. The tasks are scheduled with a work-stealing algorithm: by default, sub-tasks are executed on the same processor as their parent task, but idle processors can "steal" tasks from other processors.

TBB [14] is a C++ library allowing the development of parallel applications. As it is based on standard C++ and requires only a runtime library to run, it does not rely on a custom compiler. It includes both high-level parallel algorithms and low-level abstractions. The scheduler is based on the work stealing algorithm.

II 1.2 Distributed task-based programming

Distributed task-based programming extends task-based programming to distributed systems. As the memory is not shared between the processing units, scheduling the tasks efficiently becomes crucial to avoid costly data movements. To work at scale, these systems often implement fault tolerance techniques.

Parsl [15] targets scientific workloads. It allows the user to couple a Python code with dependencies (Python functions or external tasks) that can be executed remotely on various resources. A centralized scheduler, called the `DataFlowKernel`, coordinates the execution of the tasks.

StarPU [16] focuses on supporting heterogeneous architectures. Tasks can have several hardware-specific implementations such as CPU and GPU, the most adapted one being chosen at runtime. It is efficient and supports various scheduling strategies.

The following subsections introduce more extensively task-based programming tools used by Doreisa.

II 1.2.a Dask

Dask [17] is a Python framework aiming at making distributed computing easy. The basic workflow to run computations with Dask is composed of two steps:

- **Building a task graph.** The distributed computation is represented as a task graph, as the one in figure 1. The task graph is a directed acyclic graph where each node represents a computation. There is an edge from a node T_1 to a node T_2 if T_1 needs the result of T_2 to be executed. Internally, a task graph is represented as a simple Python dictionary.
- **Running the computation.** The graph is shipped to a scheduler, which is in charge of executing all the tasks and returning the final result of the computation. Dask provides several schedulers suited to different use cases: a threaded scheduler, a multiprocessing scheduler as well as a distributed scheduler. The threaded scheduler and the multiprocessing scheduler work for single-

node computations, while the distributed scheduler is able to schedule tasks across a large cluster, taking into account resource availability as well as data locality. The threaded scheduler is more lightweight, but some workloads need multiprocessing to be truly parallelized due to Python's `GIL` (Global Interpreter Lock).

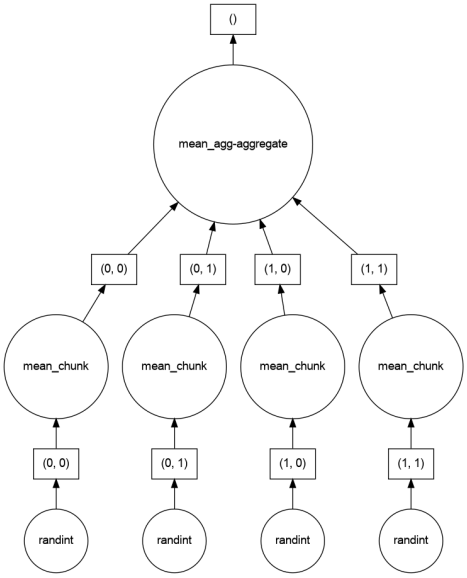

Figure 1: Example of a Dask task graph

Dask provides APIs similar to pandas and numpy to support working with distributed arrays and dataframes. A Dask array is a distributed implementation of numpy arrays. It is composed of several *chunks*, each chunk being a numpy array containing a part of the full array. Performing computations on a Dask array produces a task graph that can be executed in a distributed manner, hiding all the complexity from the final user.

Dask suffers from several limitations impacting performance and the ability to scale:

- Data cannot be shared between worker processes without being copied, even when the workers are running on the same node.
- Task execution is managed by a centralized scheduler. It enables dynamic load balancing and optimization of data movements, but becomes a bottleneck at scale. According to Dask's documentation [18], it can handle at most around 4000 tasks per second.

II 1.2.b Ray

Ray [19], [20] is a large Python framework, developed for AI and machine learning applications. One of its building blocks, Ray Core, offers a low level API for task-based distributed computing. Ray also provides modules for AI training, hyperparameter search, model serving, reinforcement learning, etc, that are implemented on top of Ray Core. In the rest of this report, Ray will only stand for Ray Core.

Ray's scheduler is distributed: each node of the Ray cluster is able to schedule new tasks. This allows tasks to create subtasks efficiently, without having to contact a centralized scheduler as in Dask. Contrary to Dask, no description of the full task graph is available ahead of the execution.

Ray's API is lower level than Dask's: it does not include convenient abstractions such as distributed arrays and DataFrames.

The following sections briefly introduce the main abstractions provided by Ray: object references, tasks and actors. Ray also provides advanced options to choose how tasks are scheduled, the lifetime of actors, support asynchronous code with `asyncio`, etc, but introducing them is out of the scope of this report.

II 1.2.b.a Ray `ObjectRefs`

Ray relies on a distributed shared memory system [21]: each node of the cluster runs an *object store*, where Ray stores data. The data is directly accessible by the workers running on the node, and can also be retrieve remotely.

A Ray `ObjectRef` is a small Python object that points to data living anywhere in the Ray cluster, acting as a distributed pointer. The data can be retrieved using the `ray.get` function.

An `ObjectRef` can be created by placing data directly in the Ray object store using the `ray.put` function. It can also point to the result of a call to a remote function or method, possibly before the function execution terminates. In that case, the `ObjectRef` is used as a *future*.

`ObjectRefs` can be freely shared across the nodes of the Ray cluster: thanks to a distributed reference counting system, the memory is freed automatically when no `ObjectRef` pointing to it exists anymore.

```python
import ray

object_ref: ray.ObjectRef = ray.put([1, 2])
assert ray.get(object_ref) == [1, 2]
```

Listing 1: Usage of Ray's `ObjectRefs`. Putting an object to the object store produces a reference. Its value can be retrieved using `ray.get`

II 1.2.b.b Ray tasks

In Ray, a remote function is a function that can be executed remotely, on any available machine in the cluster. Calling a remote function will create a remote task whose result is represented by an `ObjectRef`. Calling `ray.get` on this reference will wait for the task to finish and return the result of the task.

```python
import time
import ray

@ray.remote
def f(n):
  # Some expensive computation
  time.sleep(3)

  return 2 * n

object_ref: ray.ObjectRef = f.remote(6)
print(ray.get(object_ref))  # 12
```

Listing 2: Execution of a Ray remote function. Calling `.remote()` returns an `ObjectRef` immediately, `ray.get` waits for the result and retrieves it.

II 1.2.b.c Ray actors

Ray actors are instances of classes defined with the `ray.remote` decorator. They allow *stateful* computations.

```python
import ray

@ray.remote
class Actor:
  def __init__(self) -> None:
    self.n = 0

  def increase_counter(self) -> int:
    self.n += 1
    return self.n

actor = Actor.remote()
ray.get(actor.increase_counter.remote())  # 1
ray.get(actor.increase_counter.remote())  # 2
ray.get(actor.increase_counter.remote())  # 3
```

Listing 3: Example of a Ray actor. The actor's internal state is preserved across method calls, allowing stateful computations.

II 1.2.c Dask-on-Ray

Dask-on-Ray [22] is a project aiming at bringing the best of Dask and Ray together. It makes it possible to execute a Dask task graph on a Ray cluster, allowing the use of the simple Dask API and abstractions such as Dask Arrays and Dask DataFrames, while taking advantage of the better performance and flexibility offered by Ray. As it does not take advantage of Ray's distributed scheduler (all the Ray tasks are created by one single node), the Dask-on-Ray scheduler can become a bottleneck when executing large task graphs.

To use it, a Dask task graph is first created, as with standard Dask. Then, Dask-on-Ray is called to execute it. Dask-on-Ray is actually a Dask scheduler, that is a function taking two main parameters: a Dask task graph and a list of tasks whose result should be returned. For each task, the Dask-on-Ray scheduler performs a Ray `remote` call to execute it on the Ray cluster.

As the arguments to the functions in the graph are passed directly to the Ray remote function, it is possible to put Ray's `ObjectRefs` as values in the task graph. The compute function will dereference the `ObjectRef` automatically.

II 2 *In situ* data processing

To avoid having to store all the data produced by the simulation on the disk at each iteration, the analysis can be performed online, as soon as the data is available. Data should be

processed as close as possible to where it is produced [23]. As the analysis is performed online, it becomes possible to monitor the simulation in real time, allowing an early detection of potential problems.

It is possible to take advantage of *in process*, *in situ* and *in transit* paradigms.

- *In process* analysis consists of analyzing the data directly in the process that produced it. No data movement is required at all.
- *In situ* analysis consists of analyzing the data on the node that produced it. The data may be copied, but the copy remains local: it is not transmitted across the network.
- *In transit* analysis consists of analyzing the data on another node than the one that produced it. Sending the data across the network is necessary.

In process analytics avoids data movements, but forces the simulation to wait during the analysis. Communications between nodes may be difficult to implement efficiently. *In situ* analytics avoids useless data movements and allows the simulation to run while the data is being analyzed. However, it can steal resources such as CPU, memory or cache from the simulation, slowing it down. With *in transit* analytics, the simulation is not disturbed as the data is analyzed remotely. The remote copy of all the data can be expensive, and additional nodes are required for the analytics.

[24] explores a flexible solution to perform *in situ* and *in transit* analytics by allowing the user to define a graph of tasks to be executed for the analytics. The model is simple: a task corresponds to a process or a thread running an infinite loop. It has some input and some output, and is connected to other tasks. The computations are realized on resources that the simulation does not need, reducing the impact on the performances.

Damaris/Viz [25] focuses on *in situ* visualization. It supports coupling the simulation with standard analysis pipelines as well as custom Python scripts for simple local analysis. Cores are dedicated to the analysis in order to make the execution time predictable.

Tins [26] is a task-based *in situ* framework. The analysis is performed on dedicated helper cores that can also perform simulation tasks when no analysis is needed. Tins is developed with Tbb [14]: the simulation should be adapted to support it, and the analytic code needs to be written in C++. It focuses on optimizing the analysis inside a node, but does not support inter-node communications.

II 2.1 PDI

PDI (the PDI Data Interface) [27] is a project aiming at coupling C / C++ programs (typically MPI simulations) with plugins in charge of using the data for various tasks. Plugins make it possible to save the data to HDF5 files, export it to JSON, etc. The user needs to write a YAML file to choose how to use the data, without having to recompile the simulation code each time the usage changes.

In this project, we will use the `Pycall` plugin, which allows making the data available to a Python script as a Numpy array without copying it.

II 2.2 Deisa

Deisa [28], [29] is a Dask-based solution for *in situ* analytics. It can be coupled with a simulation without any code change, provided that the simulation supports PDI. The data produced by the simulation is represented as a Dask array that the user manipulates to define analytic tasks. Deisa supports a *contract*-based mechanism avoiding the overhead of handling the data chunks that are not required by the analysis.

Deisa lacks the flexibility of dynamically adapting the analysis to the simulation results as the simulation runs: all the analytic tasks need to be defined at the beginning of the execution. The number of iterations that can be analyzed is also fixed, which can be problematic for systems where the number of iterations is not known in advance.

Its peak performance is inherently limited by the uses the Dask "distributed" scheduler, which is centralized and cannot handle more than 4000 tasks per second [18].

Deisa patches the Dask scheduler to add support for *external tasks*. These tasks allow using the data produced by the simulation in the analysis: the simulation processes notify the scheduler when their chunks are ready, allowing it to start the tasks depending on these chunks. This patch is costly to maintain, as it needs to be updated each time internal changes are made to the Dask distributed scheduler.

II 2.3 Reisa

Reisa [30] is an attempt to solve the limitations of Deisa by using Ray instead of Dask. One of the main limitations of this approach is the lack of native array support. In Reisa, users no longer have a global view on the data as a Dask array: they have to manually define their analytics with callbacks taking the `numpy` arrays produced by the simulation and the corresponding ranks as parameters. This lower-level approach makes it harder for the user to write the analytic script.

Reisa is not optimized to work with large-scale simulations, and it was evaluated with at most 16 nodes.

III Experiments

All the experiments presented in this report were performed on the Jean Zay or Leonardo supercomputers, or on the `gros` cluster of Grid5000 located in Nancy. The following sections introduce these systems.

All the experiments are available on Github [12].

III 1 Jean-Zay

The Jean-Zay supercomputer, shown in Figure 2, is a supercomputer located in Saclay, France. Following its extension in 2024, it has a peak computing power of 125.9 PFlop/s [8], which places it among the most powerful supercomputers. A benchmark realized before the extension ranks it 35$^{\text{th}}$ in the Top500 list of June 2025 [1].

Figure 2: The Jean-Zay supercomputer

The experiments presented in this report were carried out using the CPU partition of Jean-Zay. This partition is composed of 720 nodes (at most 256 bookable at the same time), each having:

- 2 Intel Cascade Lake 6248 processors (2 x 20 cores)
- 192 Go of RAM

The tasks are submitted using SLURM.

III 2 Leonardo

Leonardo [31] is a supercomputer located in Bologna (Italy). It was ranked 10$^{\text{th}}$ in the Top500 list of June 2025 [1]. Its CPU partition [32] is composed of 1536 nodes, each having:

- 2 56-core Intel Xeon Platinum 8480+ CPUs
- 16x 32 GB DDR5-4800 (512 GB)

III 3 Grid5000, `gros` cluster

Grid5000 [33] is a large-scale testbed for research. It is composed of thousands of nodes, distributed across sites in France and Luxembourg.

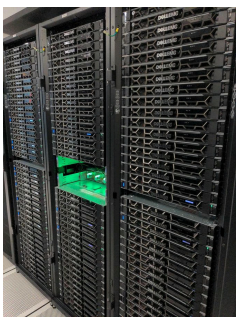

Figure 3: The `gros` cluster of Grid5000, in Nancy

The `gros` cluster, located in Nancy, is composed of 124 nodes. Each node has an Intel Xeon Gold 5220 CPU with 18 cores, and 96 GB of RAM. The detailed specifications are available online [34].

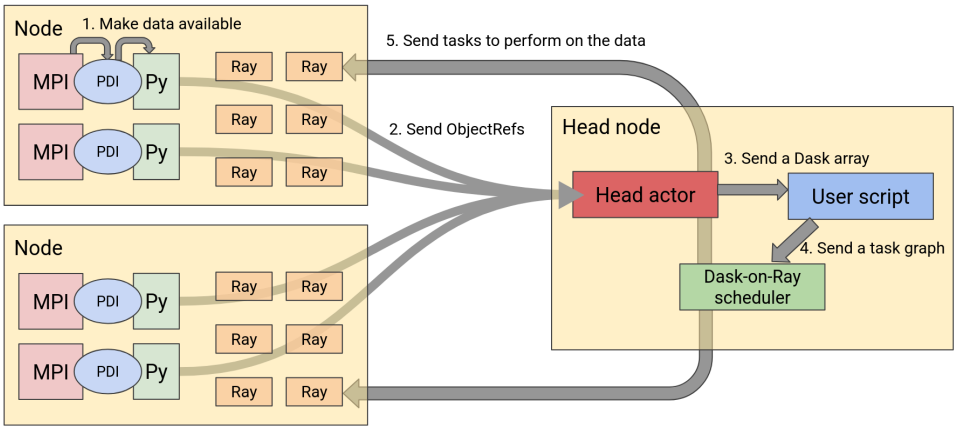

Figure 4: Architecture of the first Doreisa proof of concept

IV Doreisa: Dask-on-Ray Enabled In-Situ Analytics

IV 1 Doreisa v1: Deisa-like system using Dask-on-Ray

This section describes the first working prototype of Doreisa: we will call it *Doreisa v1*. The goal is validate the choice of Dask-on-Ray to execute Dask computation on a Ray cluster with data produced by an HPC simulation. Performance is not taken into account yet. This first proof of concept works by placing the data produced by the simulation into Ray's distributed memory. For each iteration, an actor collects the references to the data from all the nodes and schedules the task graph.

This solution made it possible to analyze data produced by HPC simulations easily. However, its design remains centralized, with one main actor quickly becoming the bottleneck of the analytics.

IV 1.1 Design

The general design of the first version of Doreisa is shown in figure Figure 4.

Each iteration of the analysis pipeline consists of the following steps:

1. The MPI processes terminate one iteration of the simulation. The data, a `numpy` array corresponding to a chunk (ie part) of the full distributed array, is ready for analytics. It is made available to Doreisa using PDI, which serves as an interface between Doreisa and the simulation code.
2. The chunk produced by the simulation is placed in the Ray object store. An `ObjectRef` is produced: this reference to the data is sent to a main actor running on the head node.
3. The head actor collects the references of all the processes. Once it has received all of them, it builds a Dask array. Each chunk of the array is represented by the corresponding `ObjectRef`.
4. The user script receives the Dask array. It can be used as a standard Dask array, to perform any kind of analysis. Performing operations on the Dask array produces a task graph.
5. The task graph is passed to the Dask-on-Ray scheduler that is in charge of executing it. Each Dask task is converted to a Ray task, and scheduled by Ray. Ray takes into account data locality when scheduling tasks [35], so unnecessary data movements can be avoided.

IV 1.2 User API

Doreisa offers a simple Python API, allowing the users to concisely define their analytics.

Listing 4 shows an example of an analytic code. In this example, the simulation produces two arrays at each iteration: `temperature` and `pressure`. The user defines and registers a callback that is called every iteration, as soon as the data is available. It is given Dask arrays representing the data produced by the simulation, as well as the current timestep. The

```python
def callback(temperature: list[da.Array], pressure: da.Array, *, timestep: int):
    if len(temperature) == 2:
        diff = temperature[1] - temperature[0]
        print("Mean temperature difference:", diff.mean().compute())

    print("Max pressure:", pressure.max().compute())

run_simulation(callback, [
    ArrayDefinition("temperature", window_size=2),
    ArrayDefinition("pressure", preprocessing_callback=lambda array: 10 * array),
])
```

Listing 4: Doreisa user API

user can perform computations by calling the `compute` method.

A sliding window mechanism allows keeping several versions of an array in memory. A *preprocessing callback* can be defined to transform the chunks *in simulation*, before placing the data to Ray's distributed memory.

IV 1.3 Performance evaluation

This first solution has the drawback of being centralized: the head actor needs to collect an `ObjectRef` for each chunk produced by the simulation. Plus, the number of tasks represented in the Dask task graph will be of the same order of magnitude as the number of chunks, and the same Python process has to schedule all of them. For big simulations running on hundreds of nodes, the head node has to process tens of thousands of references and tasks at each iteration.

Doreisa v1 is evaluated on Jean Zay. The same experiment is repeated several times, with a varying number of nodes. Each node is in charge of 40 chunks per iteration, so the total problem size grows linearly with the number of nodes (*weak scaling*: with a well-parallelized system, one would expect the execution time to remain constant or only slightly increase with the total number of nodes). The analysis consists of computing the mean of the distributed array. The execution time is averaged across 200 iterations, the first iterations being ignored to allow a warm-up phase. To avoid interfering with Doreisa, no actual MPI simulation is executed: the chunks of data are random numpy arrays. The size of the chunks is very small (10×10) to ensure that the cost of generating them and computing operations on them is negligible: the experiment aims at measuring the overhead of Doreisa (handling the task graph, scheduling the tasks, …): if the analysis is too heavy, the actual computations will hide this overhead, which will only be noticed on large problem sizes. The same evaluation protocol will be used in the following sections, to ensure comparable results.

Figure 5 shows the results obtained. The execution time is proportional to the number of nodes. In this situation, the centralized actor is clearly the bottleneck. More precisely, the analysis is composed of the following main parts:

- Collecting the `ObjectRefs` produced by the workers.
- Creating the Dask array as well as the task graph. For such small graphs, the time is negligible.
- Executing the task graph using the Dask-on-Ray scheduler.

Both the reference gathering and the task graph execution are time-consuming processes, neither one being negligible relative to the other. To further improve the performance, both need to be optimized.

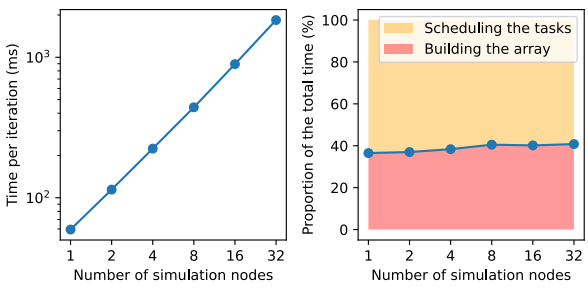

Figure 5: Performance of the naive method (weak scaling with a simple analysis)

IV 2 Building the Dask array

In Doreisa v1, at each iteration, each simulation process sends an `ObjectRef` to its data to the head actor. This centralized approach is not scalable enough: gathering all these references is a costly operation whose execution time is proportional to the number of references. When many processes send their references one-by-one, at most around three thousand references can be collected each second (figure 6).

To solve this problem, a simple idea consists of sending first the references to intermediate actors that then transmit them to the head node in a single message.

We evaluated this optimization on the *gros* cluster of the Nancy site of Grid5000 as follows. Python processes sent `ObjectRefs` to a centralized Ray actor. The process was repeated with a varying number of processes from 1 to 512, as well as a number of references sent by each process at each iteration varying from 1 to 256. During each measurement, 200 iterations were performed. Note that this experiment only benchmarked `ObjectRefs` collecting in Ray, without using the whole Doreisa implementation.

To avoid having to deploy the experiment on a very large cluster, several Python scripts were started on each core. The measurement was repeated two times: one time with two nodes sending references, the other time with four. As the total execution time in each scenario varied by less than 10%, it was confirmed that the bottleneck actually came from the head node and not the simulation node.

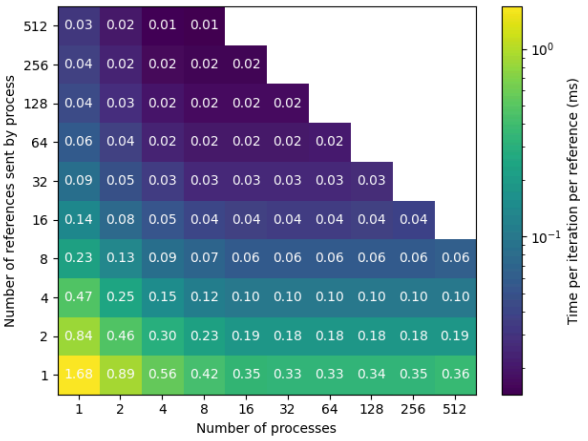

Figure 6: Time (ms) needed to collect *one* reference depending on the number of processes and references sent by each process

We can notice (figure 6) that the measured values are higher when less than 8 processes are used. This is expected: with a small number of processes, the measured time includes some time where, for instance, the centralized actor is idle, waiting for data. These measurements do not correspond to a realistic use case, as HPC simulations involve a much higher number of processes. When more processes are sending data to the head node, their number does not matter anymore and the measured values stabilize. In the next paragraph, we will focus on the results obtained with at least 8 processes.

With at least 8 processes, the total number of processes does not impact the time needed to send one reference. However, sending several references at each request greatly reduces the time needed to send one reference: it becomes possible to reduce the total time by around 20 times with this optimization.

In practice, to reduce network usage, a good compromise is to place one actor on each simulation node. This actor has the responsibility to collect all the chunk references produced by the simulation process, and to send them all at once to the head node.

Another solution could have been to rely on collective operations such as MPI's `gather` to collect the references, but these operations are not directly supported by Ray's distributed reference counting system.

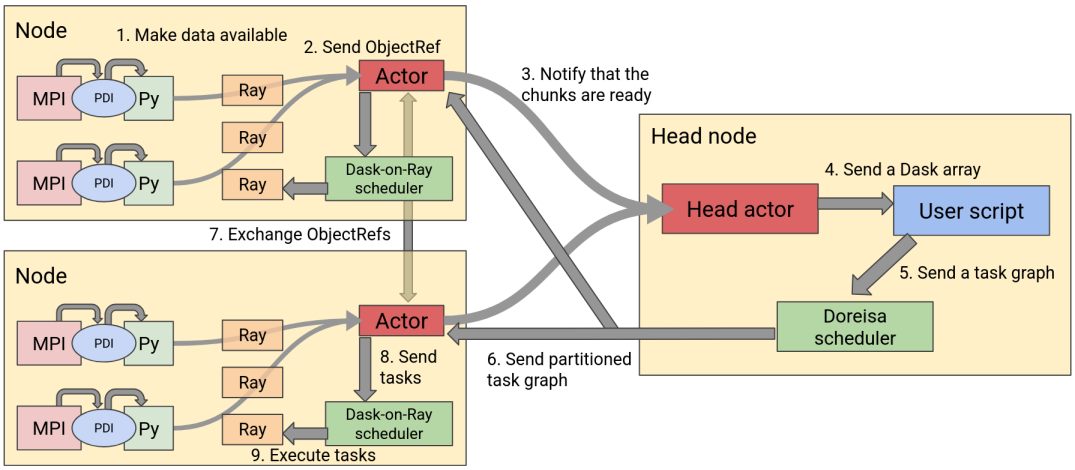

Figure 7: Architecture of the Doreisa distributed scheduler

To conclude, to reduce the time taken to collect all the `ObjectRef`, it is possible to use intermediate actors on each node to collect the references first, and send them in batches to the head node. However, in the end, this optimization was not integrated to Doreisa: the approach presented in the next section makes this optimization useless.

IV 3 Doreisa v2: Distributed scheduler

IV 3.1 Design

In Doreisa v1, the head node becomes the bottleneck of the whole system when too many nodes are added to the cluster. It has the responsibility to communicate with all the workers to collect references to the chunks, build the Dask array, schedule the tasks…

The optimization presented in Section IV 2 reduces the time needed to collect the references from the worker processes, but it does not improve tasks scheduling: all the tasks are still started by the same Python process.

This section describes a method to distribute the scheduling of the tasks, taking advantage of Ray's distributed scheduler and actor model. The goal is to avoid forcing the head node to communicate with all the data-producing processes as well as the Ray worker processes, but instead limit the communication to lightweight exchanges with actors running on each simulation node. These actors will be called *scheduling actors*.

Figure 7 shows the architecture of the Doreisa distributed scheduler. The main difference with the proof-of-concept version is that an additional actor – that we will call a `SchedulingActor` – is started on each simulation node. The head actor only communicates with the simulation nodes using this actor. It has the responsibility to collect all the `ObjectRefs` produced by the simulation node, and schedule the tasks sent by the head node.

The steps of one iteration of the analysis are as follows:

1. The simulation processes produce chunks of data made available to Doreisa using PDI.
2. The chunks are put in the Ray object store of the node, and the `ObjectRefs` are sent to the scheduling actor.
3. When all the chunks of the node have been collected, the head actor is notified.
4. The head actor builds a Dask array. The array does not contain the `ObjectRefs` pointing to the data directly: for performance reasons, they are replaced by a small object indicating which object reference should be used.
5. The user defines computations on the Dask array, which produces a task graph. The task graph is passed to the Doreisa scheduler: the Dask-on-Ray scheduler is not called yet.
6. The Doreisa scheduler partitions the graph, creating one partition for each simulation node. The partitions are sent to the scheduling actors.

7. After receiving its partition of the task graph, each scheduling actor prepares the execution of the tasks. It sends messages to the other actors to collect the missing `ObjectRefs` (these `ObjectRefs` correspond to tasks scheduled by other scheduling actors). See section IV 3.2 for more details. The references are placed in the task graph directly, replacing the placeholder object.
8. Each scheduling actor sends its task graph to the local Dask-on-Ray scheduler for execution.
9. The Dask-on-Ray scheduler schedules the tasks using the local Ray scheduler. Depending on resource availability and data locality, the Ray scheduler may choose to execute the tasks anywhere in the cluster. For example, if a task consisting of computing the mean of a chunk is scheduled by an actor running on a different node than the one storing the data, Ray will likely execute the task where the chunk is to avoid useless data movements.

With this approach, all the simulation nodes are in charge of scheduling a part of the task graph, effectively distributing the scheduling.

IV 3.2 `ObjectRefs` sharing, nested `ObjectRefs`

From an implementation perspective, there is one major difference compared to the proof-of-concept version: it is no longer possible to simply put `ObjectRefs` pointing to the data directly in the task graph that will be executed by the Dask-on-Ray scheduler. Indeed, as the scheduling is now distributed, a scheduling actor does not own all the `ObjectRefs` that are needed to perform the computation: it may need to ask other scheduling actors to send `ObjectRefs` corresponding to results of tasks that they scheduled (see step 7 of figure 7).

When asking another scheduling actor for an `ObjectRef`, the remote call produces an `ObjectRef` containing the result of the call, which is itself an `ObjectRef`. It is not possible to call `ray.get` on the actual `ObjectRef` from the second reference since it may not be ready at that time. Trying to call it anyway can result in deadlocks: an actor A might require an `ObjectRef` from another actor B to schedule its task graph, while B also requires an `ObjectRef` from A to do so.

For this reason, we need to store the `Objectref` returned by the remote call, which itself contains an `ObjectRef` pointing to the data. However, the Dask-on-Ray scheduler does not work with nested `ObjectRefs`: it expects the references to directly contain the data.

To solve this issue, it was necessary to:

- Patch a small part of the Dask-on-Ray scheduler to make it work with nested `ObjectRefs`. Dask-on-Ray relies on a call to a remote function to execute the task. In standard situations, the arguments of the function are automatically dereferenced by Ray, but only the first reference is dereferenced when using nested `ObjectRefs`. The patch makes this function recursive so that it calls itself a second time to dereference the `ObjectRefs` to the actual data (it was not possible to simply `get` the data as it would have led to unnecessary data movements).
- Force all the references in the dictionary to be nested `ObjectRefs`, even when it is not necessary. This may require artificially wrapping an `ObjectRef` inside another one by calling the identity function remotely. This avoids useless data copies when calling twice the remote function mentioned earlier.

IV 3.3 Evaluation

We benchmarked Doreisa v2 with up to 256 nodes, following the protocol defined in section IV 1.3.

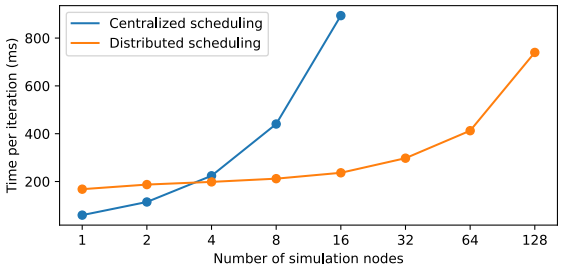

Figure 8: Time per iteration with the distributed scheduler

With less than 4 simulation nodes in the cluster, we notice a small overhead of the

distributed scheduler. When more nodes are added to the cluster, the distributed scheduler becomes much more efficient than the centralized scheduler: its total execution time grows slowly with the number of nodes until a new bottleneck appears at about 32 nodes.

IV 3.4 Task graph partitioning

The Doreisa distributed scheduler needs to partition the task graph in different subsets and send these subsets to the scheduling actors in the cluster.

The partitioning strategy has an impact on the performance of the application: if one scheduling actor has too many tasks to schedule, it can become a bottleneck similarly to what happened with Doreisa v1. If many directly dependent tasks are put in different partitions, many messages will be exchanged between the scheduling actors to schedule the tasks. A good strategy needs to:

- Produce a balanced partition (with subsets of a comparable size).
- Minimize the number of dependencies between two tasks that are part of different subsets.

Since the chunks are produced by the simulation, the partition of each leaf node corresponding to a chunk is imposed.

Note that the partitioning of the task graph only has an impact on which scheduling actor will have the responsibility to schedule each task. It is still the Ray scheduler of the node on which the scheduling actor runs that will eventually be in charge of scheduling the task on any node of the cluster.

The problems of graph partitioning and acyclic directed acyclic graph partitioning (partitioning a directed acyclic graph, ensuring that the quotient graph remains acyclic) have been studied in the literature [36], and are known to be APX-hard [37].

IV 3.4.a Partitioning strategies

We developed two partitioning strategies:

- **Random partitioning.** Each task is randomly assigned to a scheduling actor, subject to the constraint that the resulting partition is balanced: the sizes of the subsets differ by at most one. This strategy does not try to minimize the number of *cut edges*, that is the number of edges connecting two vertices in different components.

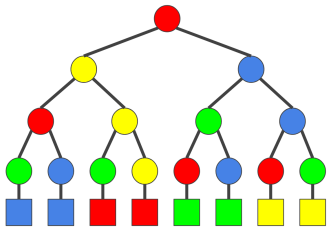

Figure 9: Random task graph partitioning

- **Greedy partitioning.** A task T is assigned to the scheduling actor that schedules the greatest number of tasks on which T directly depends. This strategy is optimal on trees.

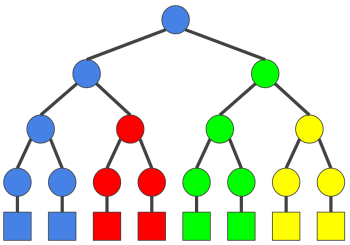

Figure 10: Greedy task graph partitioning

IV 3.4.b Evaluation

We evaluate the two strategies using the same protocol as before (see section IV 1.3). The task consists of computing the mean of the Dask array. The task graph is a tree: leaf vertices represent tasks computing the mean of each chunk, and inner vertices represent tasks merging partial means together to compute the mean of a bigger block of the array (cf figure 9 and figure 10, squares correspond to data chunks and circles to tasks).

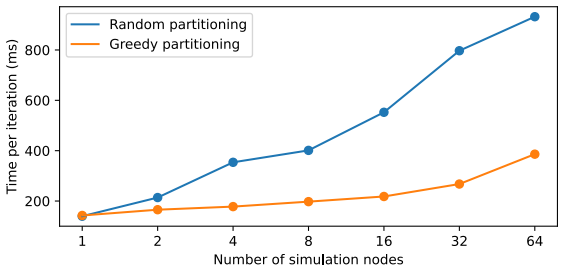

Figure 11: Performance impact of task graph partitioning

Since the partitioning does not determine the nodes executing each task, since the communication cost between the actors is small and since all the communications happen in parallel, we expected the choice of the graph partitioning strategy to have a relatively small impact on the performance as long as the subsets of the partition have comparable sizes.

Figure 11 shows the time taken to complete the analytics using each graph partitioning strategy. The greedy strategy scales very well, while the time per iteration needed with the random strategy increases with the number of nodes, reaching almost one second with 64 nodes. This behavior will require further investigation.

IV 3.5 Finding the bottleneck

As we saw in the previous section, the current system is able to scale well until about 64 nodes. Once this threshold is reached, a new bottleneck appears and the execution time starts being proportional to the number of nodes in the Ray cluster.

To understand where this problem comes from, a more detailed analysis was performed. The total execution time of a simple data analysis was measured with a varying number of nodes: 32, 64, 128 and 255. Four execution times were measured:

1. **Array creation.** This is the time taken by the head node to receive the information by the scheduling actors that the chunks are ready, and to build the Dask array as well as the task graph.
2. **Graph partitioning.** Executing the greedy scheduling algorithm without sending the task graph to the scheduling actors.
3. **Partitioned graph sending.** Sending the partitioned task graph to the scheduling actors, without having the actors perform any computation at all.
4. **Graph scheduling.** Performing all the computations required for the analysis. The scheduling actors schedule the tasks, which are executed on the Ray cluster. Note that due to the very small size of the data chunks, the execution time of each task is negligible.

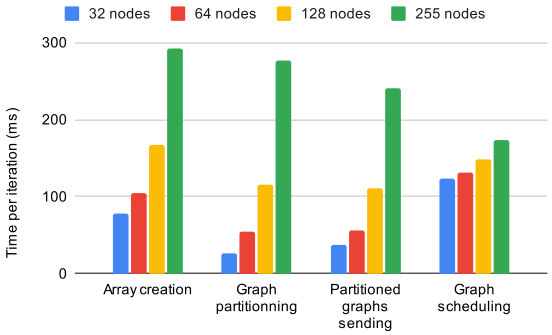

Figure 12: Time per iteration per action

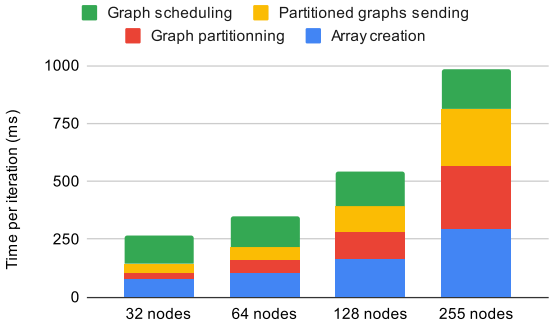

Figure 13: Decomposition of the time per iteration depending on the cluster size

Figure 12 shows the time per iteration for each step, depending on the cluster size. Figure 13 shows, for each cluster size, the proportion of the time spent in each step of the computation.

We notice that the distributed scheduler designed in the previous section scales very well: the execution time per iteration only increases by a constant amount each time the cluster size doubles.

The high execution times for the bigger cluster sizes are due to the first three tasks, which had a limited impact on smaller runs: creating the array, partitioning the task graph and sending it to the scheduling actors. These steps correspond to the centralized part of Doreisa that is executed on the head node. To further optimize the system, we need to focus on these steps.

IV 4 Doreisa v3: Asynchronous task graph processing

Thanks to the previous improvements, we managed to make the performance of Doreisa acceptable in most situations. Even with a

large number of nodes in the cluster, an iteration takes less than one second to be executed. However, we might want to further reduce this latency to make the system work efficiently with even more chunks per node.

The idea is now to hide the time taken to execute the centralized tasks mentioned in the previous section by executing them in advance, before the data is available.

We let the user define the tasks that need to be performed in advance by defining an optional callback, called a few iterations before the data is actually available. The Doreisa scheduler immediately starts shipping the task graphs to the scheduling actors, without having to wait for the data to be ready. The user can prepare several iterations in parallel: this pipelining prevents the centralized processing of the task graph from becoming a bottleneck.

The definition of the analytic tasks before the availability of the data relies on Dask's `persist` API. Instead of calling the `compute` method that executes the computation and returns its result, the `persist` method starts the computation in the background and returns a Dask array. The internal representation of this new array is simple: it contains `ObjectRefs` to the result of the computation. Calling the `compute` method on this array will retrieve the result from the `ObjectRef`, without handeling the whole original task graph.

The Doreisa scheduler needs to be updated to support this feature: if the scheduler is called from a `persist` call, it directly returns `ObjectRefs` to the final result, without getting their value.

IV 4.1 User API

Listing 5 shows what the iteration preparation interface looks like from a user perspective: the user defines a standard callback as well as a preparation callback. The return value of the preparation callback is passed as an argument to the simulation callback.

IV 4.2 Performance evaluation

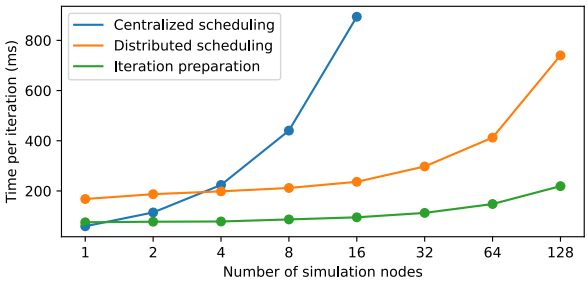

Figure 14: Performance improvement of iteration preparation

The performance improvement of the iteration preparation mechanism is evaluated with the same protocol as before (cf section IV 1.3). The experiment is repeated five times, with a number of iterations prepared in advance varying from 0 to 8.

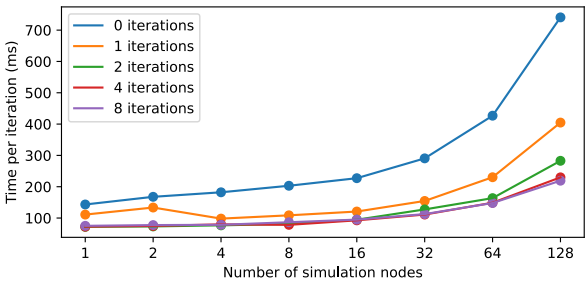

Figure 15: Performance improvement of iteration preparation: varying number of iterations prepared in advance

```python
def prepare_iteration(array: da.Array, *, timestep: int) -> da.Array:
    # We cannot use compute here since the data is not available yet
    return array.sum().persist()

def simulation_callback(array: da.Array, *, timestep: int, preparation_result: da.Array):
    print(preparation_result.compute())

run_simulation(
    simulation_callback,
    [ArrayDefinition("array")],
    max_iterations=NB_ITERATIONS,
    prepare_iteration=prepare_iteration,
    preparation_advance=10,
)
```

Listing 5: Task pipelining example. The task graph is prepared in the first callback. Its return value is passed as a parameter to the second callback, which retrieves the result.

Figure 15 shows the results obtained with this experiment.

When no iterations are prepared in advance (which approximately corresponds to not using the pipelining mechanism), the execution time ultimately starts increasing linearly with the number of nodes.

As we increase the number of iterations prepared in advance, we notice that the execution time becomes smaller. When enough iterations are prepared, the expensive tasks fully overlap with the previous iterations and stop being a bottleneck: the performance stops improving. Pipelining over several iterations is necessary since the execution time of the analysis is very short. Using a more expensive simulation and analysis would reduce the number of iterations to prepare in advance.

With 128 simulation nodes, more than 30000 tasks are executed each second. This is an order of magnitude above the peak performance of the Dask centralized scheduler, which can handle at most 4000 tasks per second according to Dask's documentation [18].

IV 5 Doreisa v4: *In transit* analytics

Until now, the simulation nodes of the cluster were also in charge of analysing the data. Performing the analysis *in situ* – directly on the nodes producing the data – can be a good solution, especially in situations where the simulation code runs on the GPU of the machine. In this case, it usually lets CPU cores idle, so they can be used by the analytics without overhead.

However, some simulations run only on CPU, and performing the analysis of the data on the simulation nodes would disturb them in an unacceptable manner [25]. With *in transit* analytics, instead of having the simulation processes perform a local copy of the data, they now send it directly to *in transit* nodes. They then return to the simulation, without being pertubed by the analytic tasks.

Figure 16 shows how Doreisa works for *in transit* analytics. It is similar to *in situ* analytics, with an important difference: simulation nodes send all their data to analytic nodes. The analytic nodes will write the data to their Ray object store, and then use it as for *in situ* analytics.

Ray can be heavy to start on a machine, with several processes needed: the raylet, the plasma store, the workers, etc. To avoid disturbing the simulation, in the case of *in transit* analytics, Ray is not started at all on simulation nodes. The simulation processes are simply given an IP address and a port that they use to communicate (ie get the preprocessing callbacks and send the chunks of data) with the analytic node using ZeroMQ [38].

The chunks of data are sent using ZeroMQ over a TCP connection, which prevents taking full advantage of the high-performance network available on the supercomputer. An improvement could be to take advantage of RDMA (Remote Direct Memory Access)

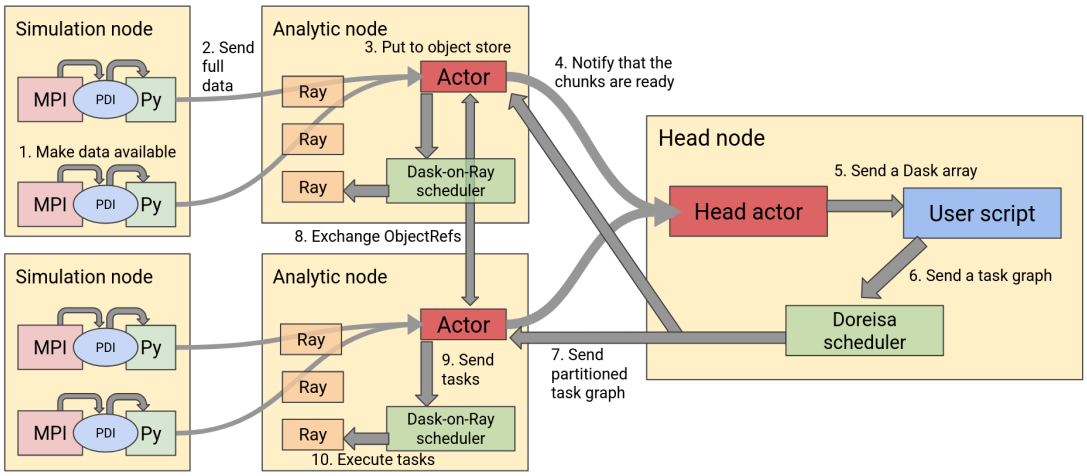

Figure 16: Architecture of the Doreisa for *in-transit* analytics

to send the data more efficiently using the supercomputer's high-performance network, for instance using UCX [39].

V Performance evaluation

We evaluate the performance of Doreisa in more various scenarios, closer to real-life applications.

V 1 Simulation producing bigger chunks

All the experiments presented in the previous section were realized using chunks of data with a negligible size, to avoid having the effective computation influence the results. In this section, Doreisa is evaluated performing an analysis on an array composed of a varying number of 1000×10000 chunks (40 chunks per node in the cluster). The analysis is also more expensive: we compute the mean of the values obtained after calling the function $x \mapsto \sin\left(\sqrt{x+1}\right)$ element-wise on the array.

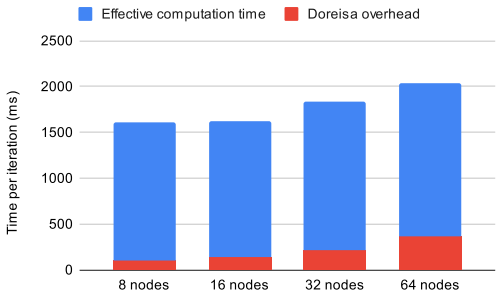

Figure 17: Analysis with big chunks of data

We notice (figure 17) that the overhead introduced by Doreisa is negligible compared to the effective computation time.

The experiment was realized before the development of the iteration preparation mechanism. With this optimization, the Doreisa overhead could be further reduced.

V 2 Forced data movements

For some task graphs, it is impossible to perform all the computations where the data is produced, and data movements are required. Suppose that M is a Dask array composed of three chunks (the chunk shape is 3×1). Consider the task of computing the sum of the coefficients of $\sin\left(M + \widetilde{M}\right)$, where $\widetilde{M}$ is M flipped on its first axis.

To perform this computation, moving the chunk at position $(0, 0)$ and $(2, 0)$ is the same node is necessary. This data movement can be seen on the task graph produced by Dask (figure 18), where two `add` nodes need both chunks.

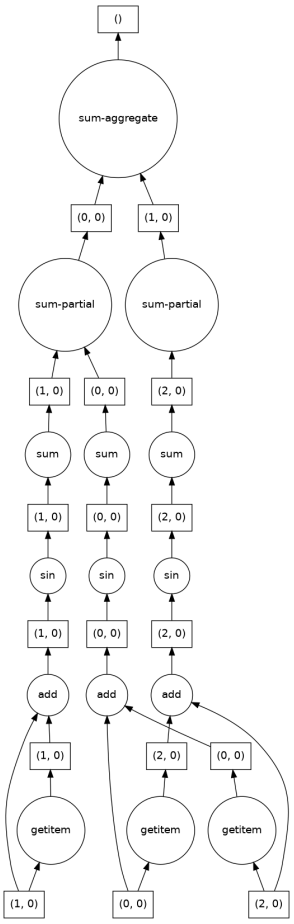

Figure 18: Task graph of a computation requiring data movements

Figure 19 shows the time taken to perform the same computation with a chunk shape of $40N \times 1$, where N is the number of nodes in the cluster. Due to the data movements required, each iteration takes more time than before, even if the computation is simpler.

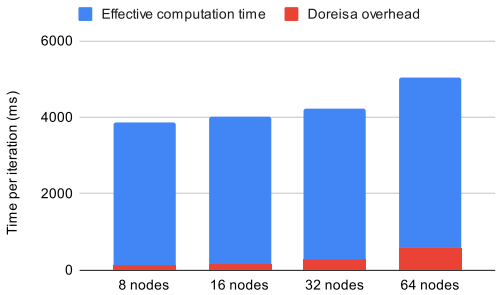

Figure 19: Analyzing big chunks of data, data movements required

Even if the total amount of transmitted data per iteration is proportional to the number of nodes in the cluster, the time per iteration does not grow: the high-performance network connecting the nodes of the supercomputer is performant enough not to become a bottleneck.

V 3 Integration with Parflow

Doreisa was integrated to Parflow [3], [4], [5], [6] and evaluated with it [40] by Andrès Bermeo Marinelli and Hugo Strappazzon, engineers in the team, using the *Leonardo* supercomputer. As Parflow already supports PDI, the integration was straightforward and no change to the simulation code were required. This section describes the results of the evaluation.

The experiment consists of running Parflow with Doreisa on four simulation nodes and one head node. The simulation runs on the CPUs: the 112 cores of the node are used as follows:

- 100 cores for the Parflow simulation (Parflow requires a square number).
- 11 cores for the Doreisa analysis.
- 1 core for measurements.

Each node is in charge of executing the simulation on a $240 \times 240 \times 240$ grid composed of $10 \times 10 \times 1$ chunks of size $24 \times 24 \times 240$. The analysis consists of computing the mean of the *pressure* array produced by Parflow at each iteration.

Figure 20 shows the time spent by the simulation and the analytics, from the sixth to the ninth iteration of the simulation. The first iterations are not included since their duration is not stable enough, as parts of the system are still starting. Since the simulation is distributed, all the nodes may not start a new iteration exactly at the same time: the times are only measured on the head node and the simulation worker with rank 0.

The overhead of analyzing the data with Doreisa is very small: at each iteration, the simulation is paused from about 2% to 10% of the time (this corresponds to the PDI line on figure 20). During this time, the chunks of data are copied to the Ray object store of the node. After this, the simulation starts its next iteration while the analysis runs in parallel.

VI Development of Doreisa

VI 1 Software Engineering practices

Doreisa was developed following standard Software Engineering practices, to ensure that the project is maintainable in the long run by several engineers and researchers.

- All the aspects of the implementation are tested. The tests are executed automatically on Github at each pull-request and on the main branch.
- The code is typechecked with `Pyright`. The code quality is checked, and the code is formatted using `Ruff`.

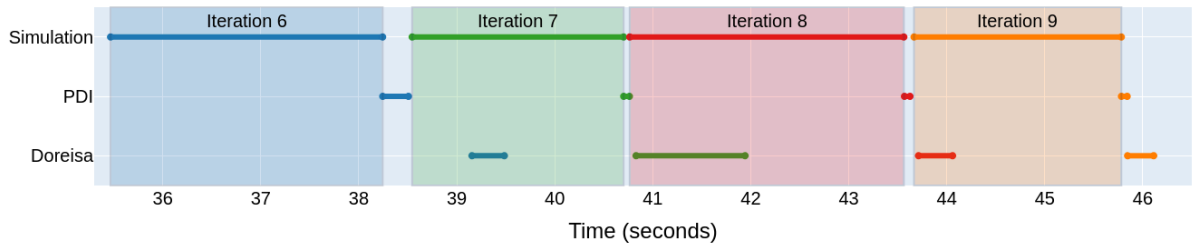

Figure 20: Benchmark of Parflow with Doreisa with 4 simulation nodes

- Releases are published regularly on PyPI following the *semantic versioning* scheme.
- The documentation is generated and published online automatically at each release [41].

VI 2 Challenges

VI 2.1 Technical issues

During the development of Doreisa, I came across several problems that took me a lot of time to fully understand and solve. This section briefly describes some of them.

- **Deployment on SLURM.** Supercomputers typically rely on SLURM [42] to manage their resources. To use Doreisa on such supercomputers, it was necessary to start a Ray cluster with SLURM. When it is starting on a node, Ray starts the worker processes that will be used to execute remote tasks. The number of such processes corresponds to the number of available cores on the machine. Since supercomputers are optimized for efficient computations, each machine typically has several CPUs, each one having tens of cores. As a consequence, a lot of Ray workers can be started at the same time (40 for Jean Zay). Each of these processes performs operations on Numpy arrays. Numpy internally relies on OpenBLAS, which itself starts many threads to take advantage of the parallelism offered by the machine. This high number of threads made SLURM kill Ray processes.

VII Conclusion and future work

This Master's project introduced Doreisa, a new distributed framework for *in situ* analytics of data produced by large parallel numerical simulations running on supercomputers. Doreisa helps bridge the gap between simulation and analysis workflows, allowing the analysis of the data on the fly and avoiding costly disk access.

Future work will involve optimizing the memory usage, which can be crucial for memory-bound simulations. More evaluation is also needed to make sure that Doreisa works well even on large-scale simulations. Doreisa only supports analyzing array-based data. Supporting more flexible structures such as meshes would allow it to analyze the data of more simulations.

Bibliography

[1] "Top500 list - June 2025." Accessed: Aug. 01, 2025. [Online]. Available: https://top500.org/lists/top500/list/2025/06/

[2] V. Grandgirard *et al.*, "A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations," *Computer physics communications*, vol. 207, pp. 35–68, 2016.

[3] J. E. Jones and C. S. Woodward, "Newton–Krylov-multigrid solvers for large-scale, highly heterogeneous, variably saturated flow problems," *Advances in water resources*, vol. 24, no. 7, pp. 763–774, 2001.

[4] S. F. Ashby and R. D. Falgout, "A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations," *Nuclear science and engineering*, vol. 124, no. 1, pp. 145–159, 1996.

[5] S. J. Kollet and R. M. Maxwell, "Integrated surface–groundwater flow modeling: A free-surface overland flow boundary condition in a parallel groundwater flow model," *Advances in Water Resources*, vol. 29, no. 7, pp. 945–958, 2006.

[6] R. M. Maxwell, "A terrain-following grid transform and preconditioner for parallel, large-scale, integrated hydrologic modeling," *Advances in Water Resources*, vol. 53, pp. 109–117, 2013.

[7] J. Monniot, F. Tessier, M. Robert, and G. Antoniu, "Supporting dynamic allocation of heterogeneous storage resources on HPC systems," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 28, p. e7890, 2023.

[8] IDRIS, "Jean Zay : Présentation." Accessed: Jul. 22, 2025. [Online]. Avail-

able: http://www.idris.fr/jean-zay/jean-zay-presentation.html

[9] IDRIS, "Jean Zay: the disk spaces." Accessed: Aug. 07, 2025. [Online]. Available: http://www.idris.fr/eng/jean-zay/cpu/jean-zay-cpu-calculateurs-disques-eng.html

[10] "Deisa tutorial." Accessed: Aug. 07, 2025. [Online]. Available: https://indico.math.cnrs.fr/event/12030/attachments/5028/9435/Deisa%20tutorial.pdf

[11] "Doreisa GitHub repository." [Online]. Available: https://github.com/deisa-project/doreisa

[12] "Doreisa experiments GitHub repository." [Online]. Available: https://github.com/AdrienVannson/doreisa-internship/

[13] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, 1998, pp. 212–223.

[14] A. D. Robison, "Intel® Threading Building Blocks (TBB)," in *Encyclopedia of Parallel Computing*, D. Padua, Ed., Boston, MA: Springer US, 2011, pp. 955–964. doi: 10.1007/978-0-387-09766-4_51.

[15] Y. Babuji *et al.*, "Parsl: Pervasive parallel programming in python," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 25–36.

[16] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," in *European Conference on Parallel Processing*, 2009, pp. 863–874.

[17] "Dask." Accessed: Apr. 18, 2025. [Online]. Available: https://www.dask.org/

[18] "Dask Actors motivation." Accessed: Jun. 03, 2025. [Online]. Available: https://distributed.dask.org/en/latest/actors.html#motivation

[19] P. Moritz *et al.*, "Ray: A distributed framework for emerging AI applications," in *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, 2018, pp. 561–577.

[20] "Ray." Accessed: Apr. 18, 2025. [Online]. Available: https://www.ray.io/

[21] S. Wang *et al.*, "Ownership: A distributed futures system for Fine-Grained tasks," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 671–686.

[22] "Using Dask-on-Ray." Accessed: Aug. 04, 2025. [Online]. Available: https://docs.ray.io/en/latest/ray-more-libs/dask-on-ray.html

[23] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K.-L. Ma, "In situ visualization for large-scale combustion simulations," *IEEE computer graphics and applications*, vol. 30, no. 3, pp. 45–57, 2010.

[24] M. Dreher and B. Raffin, "A flexible framework for asynchronous in situ and in transit analytics for scientific simulations," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014, pp. 277–286.

[25] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro, "Damaris/viz: a nonintrusive, adaptable and user-friendly in situ visualization framework," in *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, 2013, pp. 67–75.

[26] E. Dirand, L. Colombet, and B. Raffin, "Tins: A task-based dynamic helper core strategy for in situ analytics," in *Asian Conference on Supercomputing Frontiers*, 2018, pp. 159–178.

[27] C. Roussel, K. Keller, M. Gaalich, L. B. Gomez, and J. Bigot, "PDI, an approach to decouple I/O concerns from high-performance simulation codes," 2017.

[28] A. Gueroudji, J. Bigot, and B. Raffin, "DEISA: dask-enabled in situ analytics,"

in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2021, pp. 11–20.

[29] A. Gueroudji, J. Bigot, B. Raffin, and R. Ross, "Dask-extended external tasks for HPC/ML in transit workflows," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 831–838.

[30] "Reisa." Accessed: Jun. 03, 2025. [Online]. Available: https://github.com/xicko7/reisa

[31] "Leonardo Pre-exascale Supercomputer." Accessed: Aug. 05, 2025. [Online]. Available: https://leonardo-supercomputer.cineca.eu/

[32] "Leonardo HPC System." Accessed: Aug. 05, 2025. [Online]. Available: https://leonardo-supercomputer.cineca.eu/hpc-system/#jump-partition

[33] R. Bolze *et al.*, "Grid'5000: a large scale and highly reconfigurable experimental grid testbed," *The International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, 2006.

[34] "Nancy:Home - Grid5000." Accessed: Aug. 04, 2025. [Online]. Available: https://www.grid5000.fr/w/Nancy:Home

[35] "Scheduling – Ray 2.48.0." Accessed: Aug. 04, 2025. [Online]. Available: https://docs.ray.io/en/latest/ray-core/scheduling/index.html#locality-aware-scheduling

[36] J. Herrmann, J. Kho, B. Uçar, K. Kaya, and Ü. V. Çatalyürek, "Acyclic partitioning of large directed acyclic graphs," in *2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID)*, 2017, pp. 371–380.

[37] A. E. Feldmann and L. Foschini, "Balanced partitions of trees and applications," *Algorithmica*, vol. 71, no. 2, pp. 354–376, 2015.

[38] "ZeroMQ." Accessed: Jul. 24, 2025. [Online]. Available: https://zeromq.org/

[39] "UCX - Unified Communication X." Accessed: Aug. 06, 2025. [Online]. Available: https://openucx.org/

[40] "Parflow benchmark." Accessed: Aug. 01, 2025. [Online]. Available: https://github.com/theabm/bench-parflow/tree/spack

[41] "Doreisa documentation." Accessed: Aug. 07, 2025. [Online]. Available: https://deisa-project.github.io/doreisa/analytics/

[42] "Slurm workload manager." Accessed: Jun. 04, 2025. [Online]. Available: https://slurm.schedmd.com/