Distributed Task-Based In Situ Data Analytics for High-Performance Simulations

Amal Gueroudji

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Analyses de données in situ par tâches distribuées pour les simulations haute performance

Distributed Task-Based In Situ Data Analytics for High-Performance Simulations

Présentée par :

Amal GUEROUDJI

Direction de thèse :

Bruno RAFFIN Directeur de recherche, INRIA GRENOBLE-RHONE-ALPES	Directeur de thèse
Julien BIGOT CEA	Co-encadrant de thèse

Rapporteurs :

GABRIEL ANTONIU
Directeur de recherche, INRIA CENTRE RENNES-BRETAGNE ATLANTIQUE
LAURENT COLOMBET
Directeur de recherche, CEA CENTRE DE DAM ILE-DE-FRANCE

Thèse soutenue publiquement le **26 mai 2023**, devant le jury composé de :

BRUNO RAFFIN Directeur de recherche, INRIA CENTRE GRENOBLE-RHONE-ALPES	Directeur de thèse
YVES DENNEULIN Professeur des Universités, GRENOBLE INP	Président du Jury
ROBERT ROSS Senior scientist, Argonne National Laboratory	Examinateur
GABRIEL ANTONIU Directeur de recherche, INRIA CENTRE RENNES-BRETAGNE ATLANTIQUE	Rapporteur
LAURENT COLOMBET Directeur de recherche, CEA CENTRE DE DAM ILE-DE-FRANCE	Rapporteur

Invités :

VIRGINIE GRANDGIRARD
Directrice de recherche, CEA CENTRE DE CADARACHE
JULIEN BIGOT
Docteur en sciences, CEA CENTRE DE PARIS-SACLAY

To Her Majesty `Yemma`*,*
To my `Father`*,*
Sisters: `Sonia` *&* `Zahra`*,*
And beloved little Brother `Said Fouad`*.*

Acknowledgements

I was a little girl full of energy and thirst for knowledge. I was waiting to come to France and start my PhD, a new adventure, a new life, a new beginning in an infinite path... One afternoon in October 2019, I received an administrative email preventing me from working at CEA. I was broken. I cried like a kid whose toy was taken away. A few days later, I received another email from *Dr. Julien BIGOT* telling me that there was maybe a problem with the procedure and that there was a chance that I get the acceptance to join the CEA.

I smiled, I was happy again ... And a few weeks later it worked. Now I am here, becoming a Doctor.

So first of all, I would like to thank Julien for the human he is. I would like to thank you for your efforts to get me on your team even before you got to know me; your support every single day I felt down. Thank you for saying that "There is no PhD without Amal. Get her recovered first; then we will talk about work". I had the chance to meet another exceptional human, *Dr. Bruno RAFFIN*, my PhD director. He was always present to listen and guide me whenever I was lost in my thoughts.

I would like to express my sincere gratitude to my thesis advisors *Bruno RAFFIN*, for their guidance, and encouragement throughout my doctoral studies. Without their understanding and support, this thesis would not have been possible.

I would also like to thank the members of my thesis committee, Miss Virginie GRANDGIRARD, Mr. Gabriel ANTONIU, Mr. Laurent COLOMBET, Mr. Yves DENNEULIN and Mr. Robert ROSS, for accepting to evaluate my work and for their insightful comments and constructive criticism during the defence of my thesis.

I am deeply indebted to Her Majesty *Yemma* and all the family, to my A^2 and my friends that will recognize themselves for their love, encouragement, and unwavering support throughout my academic journey. Their constant encouragement and understanding have been a source of inspiration and motivation for me.

I would also like to extend my thanks to the staff and faculty of Grenoble Alpes University, Maison de la Simulation and DataMove teams, for providing an excellent academic environment that facilitated my research work and helped me develop both professionally and personally. This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 800945 — NUMERICS — H2020-MSCA-COFUND-2017, so I would like to deeply thank the Marie Skłodowska-Curie Actions (MSCA) for giving me a chance to join this incredible PhD program.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

What would life be if we had no courage to attempt anything?

Vincent Van Gogh

1.1 Introduction

Figure 1.1: ENIAC team, (Figure from [5]).

The term of *supercomputers* was used for the first time in March 1920, in the New York World, to refer to *"new statistical machines with the mental power of 100 mathematicians in solving even highly complex algebraic problems"* [150]. Since then and over the last 70 years, computing has grown from the first programmable electronic general-purpose computer: the Electronic Numerical Integrator and Computer (ENIAC, Figure 1.1 [5]) able to process 500 floating-point operations per second (flops), completed in 1945 to the first Exascale supercomputer in the world: Frontier (Figure 1.2 [67]) able to process 1.102 Exaflops [33, 6].

One can ask for a simple definition of high-performance computing and question the need for supercomputers while a laptop is enough for our daily tasks. There are several answers to the first question, and here we have chosen the two that seem to be the most relevant to us. A high-performance computer is defined in the JISC New Technology Initiative Proposal [2] and cited in [104] as: *"computing resources which provide more than an order of magnitude more computing power that is normally available on one's desktop"* And high-performance computing is defined on the IBM website [7]: *"HPC is technology that uses clusters of powerful processors, working in parallel, to process massive multi-dimensional datasets (big data) and solve complex problems at extremely high speeds. HPC systems typically perform at speeds more than one million times faster than the fastest commodity desktop, laptop or server systems."* The two definitions are complementary, and both of them mention the computing power and the memory size of supercomputers, which leads us to answer the second question regarding the need for supercomputing. It arises to solve complex, memory/compute-bound problems.

Today, supercomputing is involved in the research for solutions to an extensive list of challenges: green and renewable energy, nuclear fusion, solar and water energy, and medical concerns ranging from understanding the human body to drug discovery thanks to computing power that speeds up the research process. Examples from astrophysics, trying to understand our universe, chemistry and the creation of new materials, simulating natural phenomena or coupling real experiments and the Internet of Things (IoT) with HPC to form digital twins systems. Several legacy problems started to be resolved with the emergence of HPC, thanks to the computing power and memory they offer. For instance, the use of machine learning and artificial intelligence has gained in popularity since the appearance of the general-Purpose graphic processing unit (GPGPU). Similarly, alongside telescopes, HPC has been used to understand and explore the theoretical aspects of a black hole, and in 2019 the first-ever black hole image could be synthesized [91].

According to J. Dongarra [77], the value of a supercomputer derives from the value of the problem it solves. As such, supercomputing is tightly related to scientific applications that are usually simulations. Those programs that model physics phenomena are complex and need large amounts of both computing power and memory to run. To achieve such a performance, computer architecture has evolved from a simple implementation of a Von Neumann computer to millions of powerful cores and accelerators interconnected together, able to process hundreds of petaflops per second. Along with those complex architectures, different programming models are proposed to write efficient programs.

Figure 1.2: Frontier the first Exascale supercomupter, (Figure from [67]).

High-performance simulations are typically iterative programs that evolve over time and may produce, in some fields, such as weather forecasts, dozens of terabytes per hour. The generated data needs to be processed to understand the phenomenon under study. In the classical workflow, the data generated by the simulation is first written to disk and then read back for post-processing, also known as post hoc processing, usually on a different workstation. Data analytics are easily performed with sequential Python codes, recently scientists adopted available parallel libraries adapted for data and big data analytics because of the huge amount of data generated by simulations.

The size of the output is not the only challenge. While CPU performance has increased following Moore's law, disk bandwidth did not, and the gap between them is widening by a few orders of magnitude, creating what is known as the IO bottleneck. In situ workflows were introduced in 2008. They aim to process the data generated by large-scale simulations as close as possible to when (time) and where

(memory) it was generated. Such workflows bypass disk accesses by processing the data in the same supercomputer as the simulation, thus avoiding the previously mentioned IO bottleneck. Despite the performance shown by in situ workflows, they are not widely used in the community because of their setup complexity and the need for prior knowledge about data analytics to do.

Most of the existing in situ tools are built on the MPI programming model inherited from the host simulation. While this model, alongside others known as MPI+X, is well suited for scientific applications for their regularity, they are not adapted for data analytics purposes. Data analytics algorithms have a different structure compared to simulations. They are characterized by irregular data and control structures and communication patterns. Trying to write some of those algorithms in a static and synchronous model such as MPI is like putting the square peg in a round hole. Not only are they not compatible, but also, putting them together is complex.

In this work, we want to bring together post hoc simplicity and in situ performance. In other words, we will couple high-performance simulations parallelized in MPI with in situ analytics written in a more adapted model of data analytics algorithms, namely distributed task-based programming model.

The rest of the document is organized as follows.

- **Part I State of the Art.** It contains two chapters. In Chapter 2, we present the context and related work, namely in situ and distributed task-based frameworks. In Chapter 3, we present the tools used in this work, namely PDI and Dask distributed.
- **Part II Contributions.** It presents the main scientific contributions of this work and contains three chapters. Chapter 4 presents the approach that we propose to couple MPI simulations with distributed task-based analytics in an in situ fashion, called DEISA bridging model with the first elements of its implementation using Dask and PDI. Chapter 5 presents a complete implementation with the needed configuration and user API. Chapter 6 proposes conceptual improvements in DEISA and Dask distributed.
- **Part III Conclusion and Perspectives.** It summarizes the main conclusions and lessons learned from this work, and provides feedback about possible perspectives and projects.

1.2 Communications

- **COMPAS21: Conférence francophone d'informatique en Parallélisme, Architecture et Système 2021**

  Amal Gueroudji, Julien Bigot, Bruno Raffin. Preliminary Experiments in Coupling in situ Dask analytics with MPI Simulations. COMPAS 2021 - Conférence francophone d'informatique en Parallélisme, Architecture et Système, 2021, virtual

- **HiPC21: 28th International Conference on High-Performance Computing, Data, and Analytics 2021**

  Amal Gueroudji, Julien Bigot, Bruno Raffin. DEISA: Dask-enabled in situ analytics. HiPC 2021 - 28th International Conference on High-Performance Computing, Data, and Analytics, Dec 2021, virtual, India.

- **HPC/DA21: Workshop on the In Situ Co-Execution of High-Performance Computing & Data Analysis 2021**

  Amal Gueroudji, Julien Bigot, Bruno Raffin. Preliminary Experiments in Coupling in situ Dask analytics with MPI Simulations. HPC/DA 2021 - Workshop on the In Situ Co-Execution of High-Performance Computing & Data Analysis, July, 2021

- **Per3S 2022: 6th Workshop Performance and Scalability of Storage Systems 2022**

  Amal Gueroudji, Julien Bigot, Bruno Raffin. Handling IO data with PDI and Optimizing away IO with PDI/DEISA. Per3S 2022 - 6th Workshop Performance and Scalability of Storage Systems, June 2022

- **PASC22: Conference on The Platform for Advanced Scientific Computing 2022**

  Virginie Grandgirard, Kevin Obrejan, Dorian Midou, Y Asahi, PE Bernard, J Bigot, E Bourne, J Dechard, G Dif-Pradalier, P Donnel, X Garbet, A Gueroudji, G Hager, H Murai, Yacine Ould-Ruis, T Padioleau, L Nguyen, M Peybernes, Y Sarazin, M Sato, M Tsuji, P Vezolle. New advances to prepare GYSELA-X code for Exascale global gyrokinetic plasma turbulence simulations: porting on GPU and ARM architectures. PASC22 - Conference on The Platform for Advanced Scientific Computing, the Association for Computing Machinery (ACM); the Swiss National Supercomputing Centre (CSCS), Jun 2022, Bâle (virtual event), Switzerland. pp.1-21.

- **PDSW22 Work In Progress: 7th International Parallel Data Systems Workshop 2022**

  Amal Gueroudji, Julien Bigot, Bruno Raffin. Dask-Enabled External Tasks For In Transit Analytics. PDSW 2022 - 7th International Parallel Data Systems Workshop, Nov 2022

- **JLESC15 Short Talk: 15th Joint Laboratory for Extreme-Scale Computing Workshop 2023**

  Amal Gueroudji, Julien Bigot, Bruno Raffin. Dask-Enabled External Tasks For In Transit Analytics. JLESC15 2023 - 15th Joint Laboratory for Extreme-Scale Computing Workshop, Mar 2023

Part I

State of the Art

Chapter 2

Context and Related Work

Research is to see what everybody else has seen, and to think what nobody else has thought

Albert Szent-Gyorgyi

In this chapter, we provide a detailed context of our work, supercomputers, their architecture and parallel programming models. Then have a look at high-performance data analytics workflows, namely, post hoc workflows and in situ workflows, analysing the pros and cons of each. The second part of this chapter is dedicated to the related work on in situ data processing tools with comparative analysis. In addition, we present some existing big data frameworks that are of great interest to this work, as our goal is to bring their productivity to HPC workflows. Finally, we present the tools used for this work, namely PDI data interface for data handling and Dask distributed for data analytics.

2.1 High Performance Computing and Data Analytics

In this section, we define the context of this work, namely high-performance computing (HPC), supercomputers and their architecture. High-performance simulations running on those machines and high-performance and big data processing, including post hoc and in situ analytics. We dedicate a section to discuss parallel programming models focusing on Bulk Synchronous Parallel (BSP) and distributed task-based programming. Finally, we conclude with a discussion to highlight the need to consider a heterogeneous programming model scheme in the context of in situ processing and motivate our work.

2.1.1 High Performance Computing

A high-performance computer can be defined as a large set of powerful computing resources working in parallel, which provides more than an order of magnitude more computing power than is normally available on one's desktop. Thus it makes it possible to process massive multi-dimensional datasets (big data) and solve complex problems at extremely high speed [2, 7, 104].

2.1.1.1 Supercomputers' Architecture

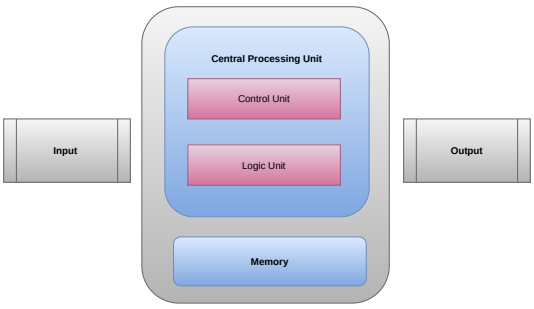


Figure 2.1: A schematic representation of a simplified Von Neumann architecture, inspired by paper [153]. The computer is composed of a CPU and a main memory with an IO mechanism.

In 1945, John Von Neumann proposed a digital electronic computer architecture composed of four main components: a central processing unit, a memory unit, Input/Output (IO) mechanisms and a storage system, as shown in Figure 2.1. Since then, this architectural view has not changed radically. However, the technologies behind the different components have never been the same. This minimalist view of a computer has been improved in design and performance, and the number of transistors per chip doubled every 18 months, following Moore's law until the late 2000s. Nowadays, instead of putting more electronics into the same chipset, constructors are increasing the computing power per processing unit. The number of cores or processing units (CPUs) keeps increasing, not only in the same nodes but also in distributed configurations, leading us to clusters and supercomputers (see Figure 2.4). A computing node may contain one CPU socket or more. A CPU is usually a multicore. A node may also contain accelerators such as graphic processing units (GPUs). Several nodes are interconnected with high-speed networks to process large problems. For instance, the Frontier supercomputer comprises 8 730 112 cores, spread in nodes of one 3^{rd} Gen AMD EPYC CPU and four Purpose-Built AMD Instinct 250X GPUs. CPUs and GPUs are interconnected with AMD Infinity Fabric, and the nodes are interconnected with multiple Slingshot NICs providing 100GB/s network bandwidth[6], Figure 2.2 shows a diagram of a Frontier node.

There are many other interconnect systems used in supercomputers. Figure 2.3 shows the interconnect family system used in the TOP500 supercomputers in November 2022. The most used is Ethernet with 46.6% usage, followed by Infiniband with a percentage of 38.8% and Omnipath with 7.2% and others. In Figure 2.5, we show the roadmap of the Infiniband for 1x, 2x, 4x, and 12x port widths with bandwidths reaching 600Gb/s data rate in the middle of 2018 and 1.2Tb/s data rate in 2020.

The computing power offered by supercomputers generates more precise and huge amounts of data. In addition to an eventual small local storage per node, we find parallel storage servers interconnected together to form a larger storage system. A parallel file system (PFS) is software that manages the storage and accesses the data in those distributed servers simultaneously and efficiently. The most used parallel file systems in supercomputers are the open-source Lustre[10] file system and IBM Spectrum Scale or previously named General Parallel File System (GPFS)[143].

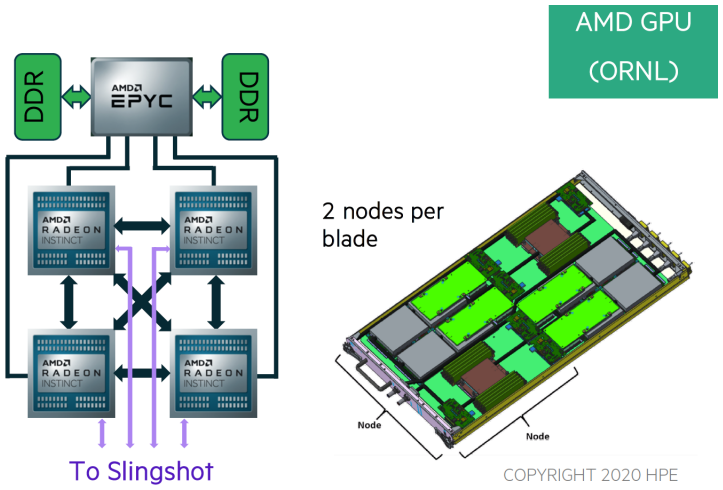


Figure 2.2: Frontier's node diagram, (Figure from [6]).

While computing power kept doubling each 12-18 months until the late 2000s, storage bandwidth did not. For a 3Ghz CPU whose average clock cycle is 0.3ns, the latency to access the main memory is between 70-100ns. Random Access Memory (RAM) accesses are two orders of magnitude slower than CPUs, and IO operations are even slower as they take between 1-10ms. Figure 2.6 shows the IO bandwidth for the IO500 supercomputers registered in November 2022. Note that most of the points are situated around 500GiB/s. In addition to the huge latency to access a PFS, it is usually shared between several applications running in the supercomputer. If those applications perform IOs simultaneously, then IO performance drops.

2.1.1.2 IO Bottleneck

The problem of IO performance comes from three main problems together: the gap between CPU performance and storage bandwidth, the fact that the parallel file system is shared between multiple users and the huge amount of data generated by simulations in some fields.

One can consider improving the IO performance or reducing the data size to write to reduce the IO bottleneck. We find already in the literature work that dedicates cores or staging nodes for IO operations to reduce the IO jitters [79] on the time-to-solution or send the data to faster memories such as local Solid-State Drives (SSDs) rather than the PFS. The second possibility is to reduce the amount of data to write to disk. The easiest way is to make selections on the global domain, such as time sampling, and write only what scientists think is relevant. Those decisions are made at the beginning of the simulation usually. Thus a good understanding of the simulated phenomena is necessary but not enough to capture all interesting or rare events. The last possibility is to process the data as it is generated and only keep interesting results. This approach is called in situ processing; it uses the same computing resources as the simulation. We have opted for this last possibility: to perform in situ processing to bypass frequent disk

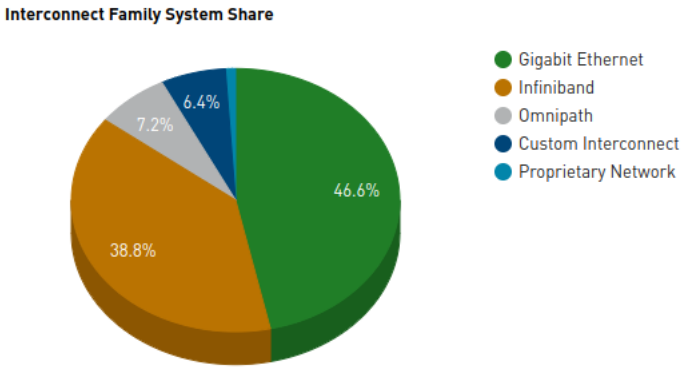


Figure 2.3: Interconnect family system share in TOP500 2022 list (Figure from Top500 [33]).

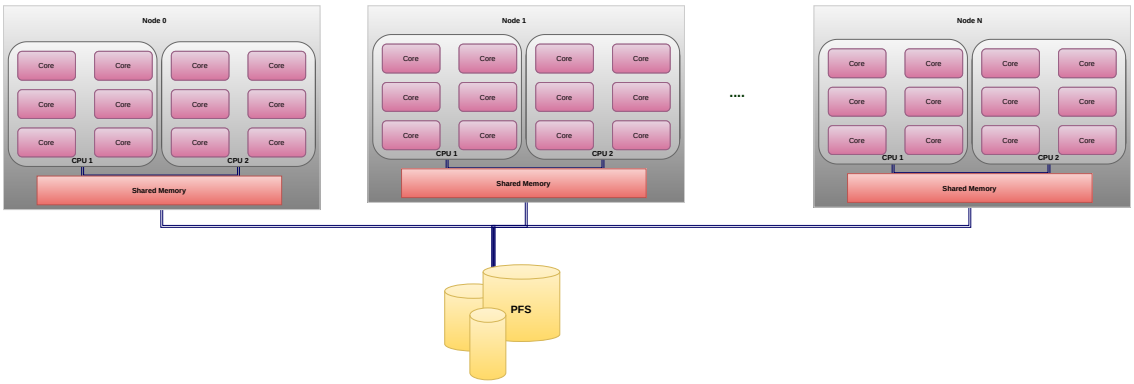


Figure 2.4: A schematic architecture of a supercomputer represented by compute nodes interconnected. Each node contains two CPUs with six cores. And all the nodes are connected to a parallel file system represented in this picture as shared disks. Figure inspired by [177].

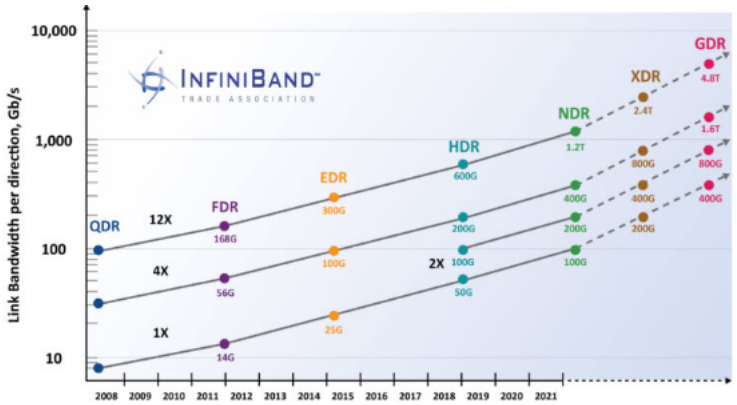


Figure 2.5: The InfiniBand trends for 1x, 2x, 4x, and 12x port widths with bandwidths reaching 600Gb/s data rate in the middle of 2018 and 1.2Tb/s data rate in 2020, (Figure from [149]).

accesses and avoid the IO bottleneck for several reasons, such as keeping only small and meaningful data, usually with enough information to keep the study interesting; being able to have online information about how the simulation progress, so activating steering or specific actions if needed; fully leverage the HPC platform and use the resources efficiently. Note that in in situ processing, we take advantage of the aggregated interconnect bandwidth of all used nodes instead of being limited to fixed bandwidth (storage bandwidth) as in post hoc processing.

2.1.2 High Performance Data Processing

An HPC numerical simulation is the execution of a program that models a phenomenon in general or the behaviour of an object under specific conditions. The objective is to study and to understand aspects of the problem and optimize the underlying industry or contribute with new findings to general science. The data generated by a scientific application needs to be processed to understand the phenomenon under study. The classic way to process that data is to save it first to disk and then read it back for post-processing (Section 2.1.2.1). However, simulations generate dozens of terabytes per hour in some fields, such as weather forecasts and nuclear fusion. Saving all the generated data is impossible both in terms of memory size and time to save to solution because of the IO bottleneck. Moreover, in traditional workflows, the generated data is often processed using sequential Python codes, which is worth replacing by parallel codes for larger datasets. Python is still one of the best languages and environments for engineering and scientific computing, as it helps to write nontrivial computational programs without getting too bogged down in syntax and compilation time lag [135]. To avoid the IO bottleneck and fully leverage the HPC platform, one can perform in situ processing (Section 2.1.2.2), which consists in processing the data as close as possible to where and when it is generated. In other words, the data is processed on the same platform where the simulation runs without going through the disk.

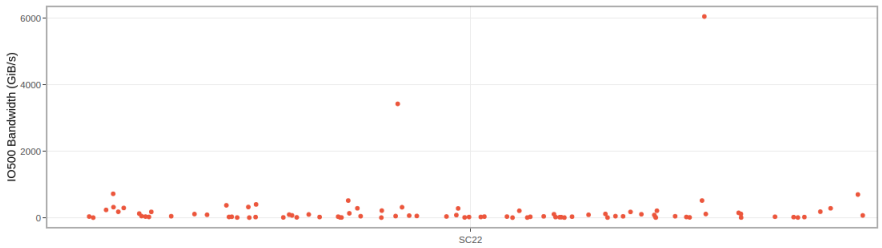


Figure 2.6: IO500 Bandwidth for November 2022, (Figure from IO500 list [8]).

2.1.2.1 Post Hoc Processing

Post-processing, also known as post hoc or offline processing, refers to analysing or visualizing the generated data at the end of the simulation. The generated data is saved into files in the PFS first, then read back for post-processing in a second step, usually in a separate computing environment. In post hoc workflows, the raw data written to disk has to be transformed to extract meaningful physically-based features of interest that will be visualized and analysed in the post-processing step.

The raw data can be written to one or several files, and one can consider one of those organizations of the data: one file per process over time, one file per process per timestep, one file per timestep for all the processes, and finally, one file for the whole simulation. Depending on the simulation duration, the number of processes collaborating, and the data size, the choices may be different. For instance, if we deal with a large simulation generating terabytes per timestep, using a single file of the whole simulation may not be the best solution.

Several IO libraries and file formats are used in HPC, such as MPI-IO [161], ADIOS [128, 98], HDF5 [93], NetCDF [125] and so on. However, HDF5[1] and NetCDF[2] are the most common for their performance, flexibility and the possibility to perform parallel IOs.

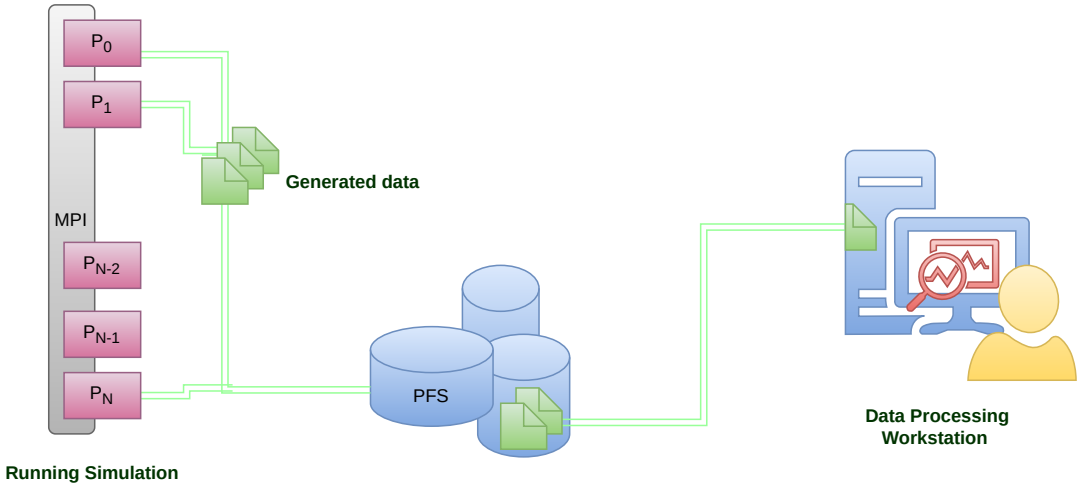


Figure 2.7: schematic view of post hoc processing workflow. In this figure, there are three main entities: the simulation represented by $N+1$ processes, the parallel file system, and a separate desktop computer where analytics are performed. The workflow is represented by two main steps: the simulation step, which is concluded by writing the generated data to the parallel file system, and the post-processing step, which starts at the end of the simulation. In this scheme, the generated data is sent to a separate desktop computer where analytics are performed.

Figure 2.7 shows a typical post hoc workflow where a running simulation saves the generated data into files in the PFS. Then the files are sent to a data analytics workstation to be processed. Scientists may perform either visualization or other any other analysis of the data. Usually, sequential Python codes using already available libraries, such as NumPy [167], Pandas [130], Scikit-learn [139, 116] and others, are common.

[1] https://www.hdfgroup.org/solutions/hdf5/
[2] https://www.unidata.ucar.edu/software/netcdf/

When the simulations generate large amounts of data, one can take advantage of the existing big data tools to process the data in parallel to reduce the time of the analysis. Nowadays, a large list of frameworks can be found, such as Dask, Ray, Parsl, Spark, Hadoop, PyCOMPSs and so on. In this work, we will focus on using the big data framework called Dask distributed for data analytics. However, depending on the needs, any other framework can be considered to process the generated data. Listing 2.1, shows an example of a sequential post hoc code that analyses an HDF5 dataset. It uses the incremental principal components analysis (IPCA) model from the scikit-learn Python library. Listing 2.2 shows the equivalent parallel code written using Dask distributed that offers an equivalent parallel implementation of scikit-learn called Dask_ml. Note that the Listings are almost similar, thus the easiness of porting sequential Python codes to parallel ones with Dask, which is one of the motivations to use Dask in this work. More details about Dask are given in Section 3.2.

```
from sklearn.decomposition import IncrementalPCA
import json
import h5py
# Load data from HDF5
ds = h5py.File('data.hdf5',mode='r')['dataset']
pca = IncrementalPCA(n_components=2, copy=False, svd_solver='randomized')
# process each time-step independently
for step in range(0, 10):
      pca.partial_fit(ds[step,:,:])
print(pca.explained_variance_)
```

Listing 2.1: Sequential post hoc data analysis with *scikit-learn*.

```
import dask.array as da
from dask_ml.decomposition import IncrementalPCA
import json
import h5py
# Connect to Dask
sched = json.load(open('sched.json'))
client = dask.distributed.Client(sched["address"])
# Build a lazy array descriptor from HDF5
ds = h5py.File('data.hdf5',mode='r')['dataset']
ds = da.from_array(ds, chunks=(1,4096,4096))
pca = IncrementalPCA(n_components=2, copy=False, svd_solver='randomized')
for step in range(0, 10):
      pca.partial_fit(ds[step,:,:])
print(pca.explained_variance_)
```

Listing 2.2: Parallel post hoc data analysis with Dask. Lines differing from the analysis of Listing 2.1 are highlighted.

To summarize, post hoc workflows are easy to set up. They keep decoupled the simulations and the data analytics by passing the data via files. Moreover, one can take advantage of the available ecosystems to process the data, especially using the Python libraries such as NumPy, Pandas, Scikit-learn and others, or pythonic parallel frameworks such Dask, Ray and so on when dealing with big data. However, today the IO bottleneck stands as a barrier between HPC and data analytics going through the disk (post hoc processing). In the following section, we present another data processing workflow scheme called In situ, characterized by its performance compared to post hoc workflows as it avoids unnecessary data communications and IOs.

2.1.2.2 In situ Processing

The in situ[129, 51] paradigm stands for processing the data generated by a simulation code as close as possible to when and where it is generated. In other words, data analytics are performed simultaneously in the same computing platform as the simulation. In situ workflows allow sharing of data between the simulation and the analytics. Thus, they reduce unnecessary data communications and IOs. Moreover, they allow data reduction and feature extraction to minimize the data memory size to save.

Figure 2.8 shows a typical in situ workflow, where the running simulation feeds the in situ analytics programs with the data it generates. Those analytics programs range from simple local Python code to

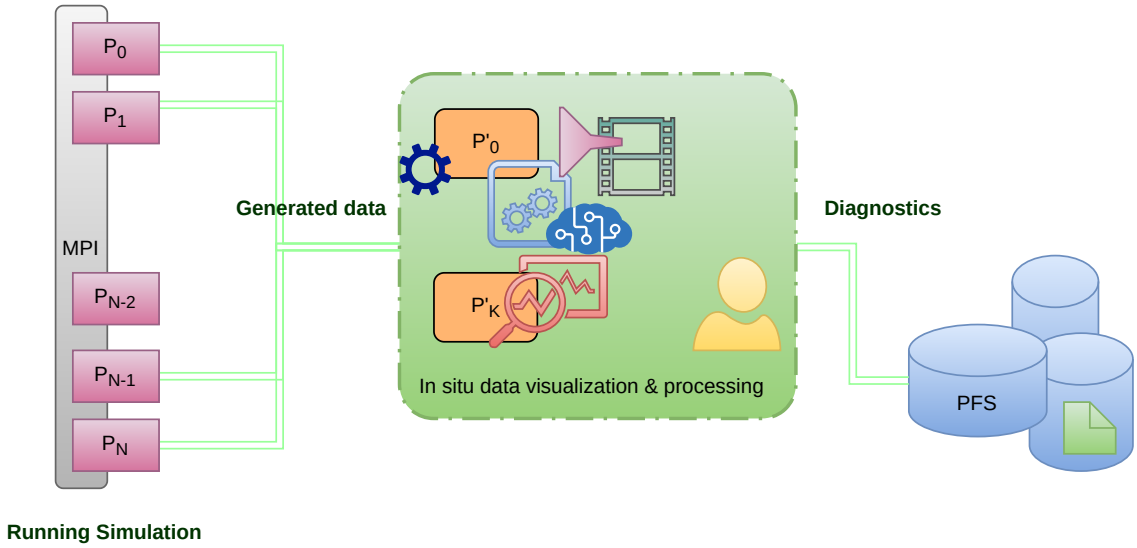


Figure 2.8: A schematic view of an in situ processing workflow. The simulation is represented by a set of MPI processes. They generate data which is sent to staging resources to perform analytics that generates diagnostics and reports that are written to the parallel file system. Note that in situ workflows happen in the same computing platform as the simulation but not necessarily in distinct nodes.

IA inference or 3D rendering and visualization. By the end of the analytics, only interesting results and diagnostics are saved to disk.

The simplest way to perform in situ analytics is to embed the analytics routines into the simulation code. However, this approach reduces the separation of concerns and may quickly produce an unmaintainable code. A cleaner way to do that is to decouple the analytics from the simulation. One can use data-handling libraries to access the simulation data and share it with external codes for analytics.

There are several other ways to perform in situ analysis, either by time (Section 2.1.2.2.1) or space (Section 2.1.2.2.2) sharing with the simulation processes. Analytics can share the same thread or process or just the same node with simulation; the processing in those cases is called in situ. If they only share the same supercomputer, and the data is sent to staging nodes, then we say in transit processing. It is also possible to perform both in situ and in transit processing in the same workflow.

In situ analytics allow data reduction and feature extraction while the simulation is running, which makes it easy to steer the simulation and trigger further analysis if needed. For instance, if a rare event is detected in situ, the simulation can be restarted from the last checkpoint, and further analytics may be done to understand further the event formation. This use case may be possible in post hoc workflows only if all the data is kept on the disk, which is not always the case. Moreover, instead of automatically reducing the size of raw data, such as having an output for each N timestep, which may lead to missing interesting events, in situ workflows allow a smarter selection of output data. Consequently, in situ does reduce not only the size of the data but also allows a smarter selection of interesting ones.

2.1.2.2.1 Time Sharing In time-sharing scenarios, the same cores are used by both simulation and the in situ analytics processes. Synchronous and asynchronous executions may be performed.

- **Synchronous Execution** In synchronous scenarios, the simulation is stopped periodically to perform analytics tasks. In a typical cycle, the cores are used for the simulation first, and when the data is ready for analytics, those cores are used for analytics. These two steps are repeated until the end of the simulation (Figure 2.9).
- **Asynchronous Execution** In the asynchronous scenarios, the cores are over-subscribed for both simulations and in situ analytics. The operating system (OS) scheduler is in charge of co-scheduling the two processes (Figure 2.10). This approach is less efficient because of the contentions on shared resources (such as memory buses and caches.) [180, 70].

  In the rest of this section, we will focus only on synchronous scenarios.

In synchronous scenarios, analytics routines may be directly embedded in the simulation code. In this case, we say that the analytics and the simulation are tightly coupled. This approach is not recommended,

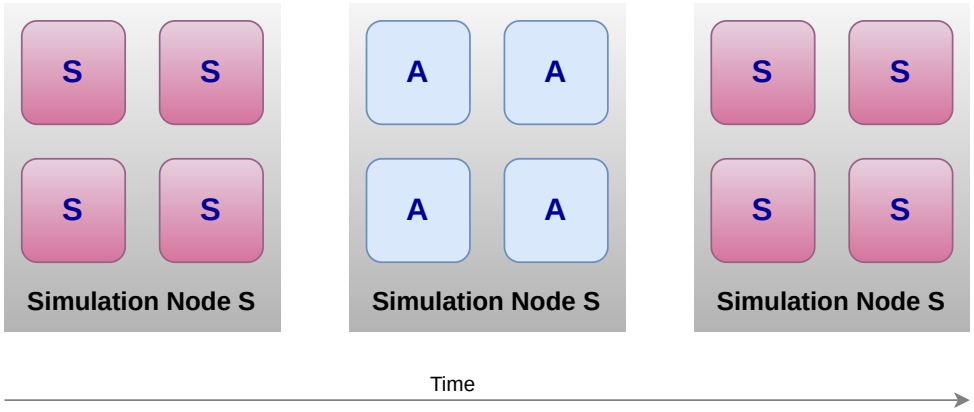


Figure 2.9: A schematic representation of synchronous in situ execution in a node with four cores. The simulation runs on the cores until an analytics step is reached. The simulation stops, and then analytics takes over. This cycle is repeated until the end of the workflow, (Figure inspired by [70]).

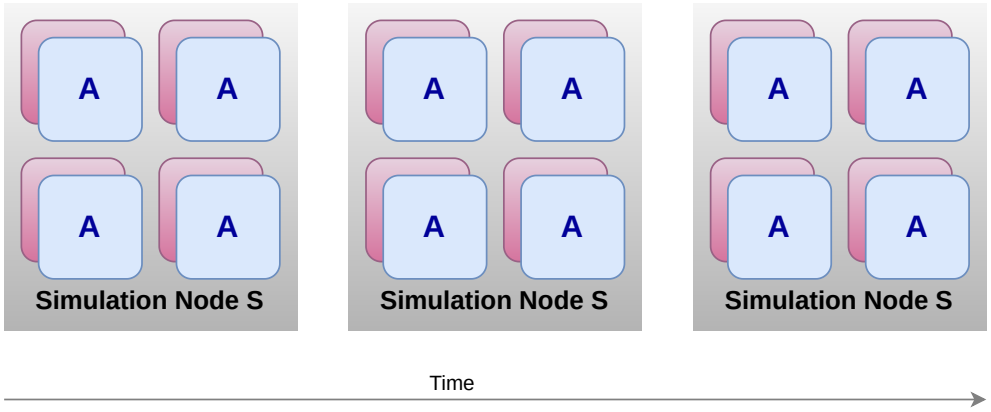


Figure 2.10: A schematic representation of an asynchronous in situ execution in a node with four cores. The simulation and the analytics are scheduled in the same over-subscribed cores, (Figure inspired by [70]).

as analytics routines will likely change very quickly, unlike the simulation. If the code is not well designed, such an approach may produce an unmaintainable code. To avoid such a situation, one may consider the separation of concerns design principle to provide a loosely coupled solution: each part of the workflow only computes what is designed for and provides an interface to communicate with other sections.

We have mentioned the separation of concerns here because it is one of the most important design principles and one of the objectives of this work.

2.1.2.2.2 Space Sharing In space-sharing scenarios, the resources are shared between simulation and analytics tasks. The analytics uses distinct cores (see Figure 2.11), which are usually called helper cores. The simulation always runs on fewer cores, but still, it uses more than the analytics. However, the performance loss is generally less than the ratio of confiscated cores [180, 79, 70].

Because distinct resources are used, the analytics are usually **asynchronous**, and the data to be analysed is copied/transferred at least once before the simulation resumes for the next step. The analytics cores used for analytics may be located on the same node as the simulation or in distinct nodes. In this last case, the analytics are called **in transit** analytics (Figure 2.12).

2.1.2.3 Heterogeneous Workflows

Because all configurations have their pros and cons, hybrid architectures are possible: either in terms of analytics placement or workflow synchronicity. One can perform synchronous in situ analytics, where the analysis routines are embedded in the simulation code, followed by in transit analytics on a staging area.

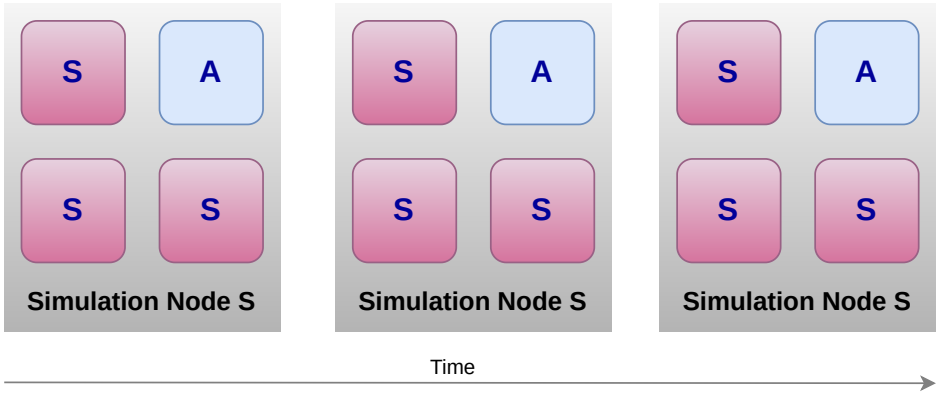


Figure 2.11: A schematic representation of asynchronous in situ execution in a node with four cores: the simulation runs on three cores, and the analytics on one helper core. The simulation and the analytics are collocated in the same node, (Figure inspired by [70]).

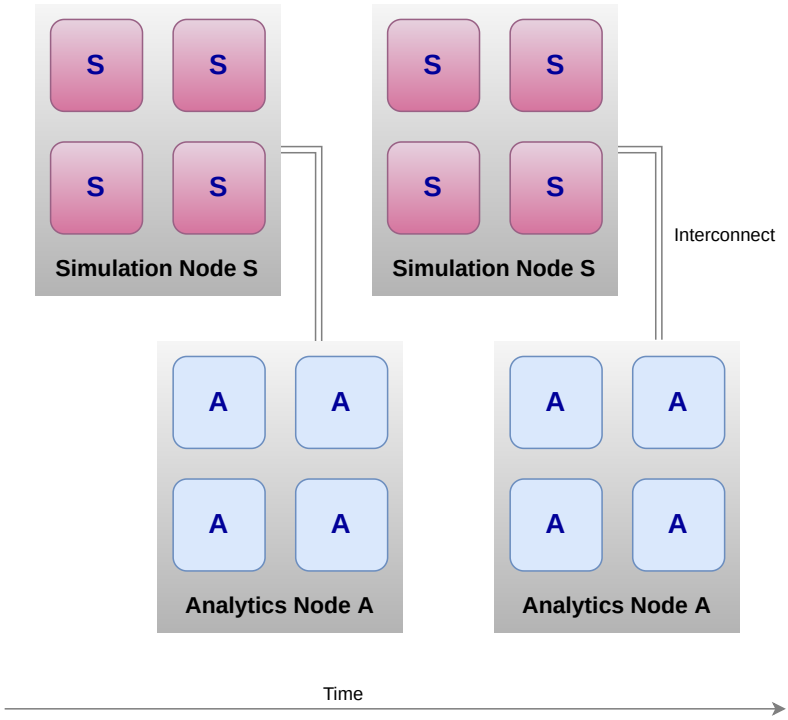


Figure 2.12: A schematic representation of in transit workflow. The simulation and the analytics run on distinct nodes. When a simulation step is completed, and data is ready to be processed, it is sent to analytics nodes over the network. Note that the simulation and the analytics node are located in the same computing platform, (Figure inspired by [70]).

This example is interesting, for instance, when the in situ routines reduce the size of data that needs to be sent to the staging nodes, and the in transit routines are heavy slow analytics algorithms.

2.1.2.4 Discussion

In situ processing is a good alternative to traditional post hoc workflows. They optimize several aspects of the HPC workflows, namely: the IOs, data communication and processing, the size of the output, and the workflow duration, thus, energy consumption. In situ workflows avoid the IO bottleneck by avoiding unnecessary IOs. They bypass disk accesses and avoid unnecessary data communications by sharing the same computing platform as the simulation. In addition, they allow data reduction and feature extraction while the simulation is running, which makes it easy to steer the simulation and trigger further analysis if needed. Moreover, instead of automatically reducing the size of raw data, such as having an output

for each N timestep, which may lead to missing interesting events, in situ workflows allow a smarter selection of output data. Consequently, in situ does reduce not only the size of the data but also allows a smarter selection of interesting ones.

As we have presented in the previous sections, in situ analytics can be performed synchronously or asynchronously with simulation, in the same cores/nodes or in different ones. Each variant has its pros and cons. For instance, synchronous in situ workflows may penalize the simulation if the analytics are relatively long and oversubscribing the core is not better because the OS scheduler may be inefficient on that. Dedicating cores or nodes is better from this point of view because of using distinct resources. However, data flow has to be managed carefully in those cases.

This work is devoted to working on another important aspect of in situ analytics, which is their setup complexity compared to post hoc workflows. This complexity usually derives from the programming model used in in situ tools, which is often the inherited message passing (MPI) model from the host simulation.

2.1.3 Parallel Programming Models

In this section, we present two parallel programming models: MPI, which is the most used to parallelize HPC simulations, and a higher-level programming model called distributed task-based paradigm, which is adapted for analytics. We finish with a discussion of how such a higher-level programming model could help to reduce the complexity of setting up in situ workflows.

A programming model is a set of concepts and program abstractions associated with a programming interface that is used for modelling and implementing algorithms. It is related to a programming paradigm that represents the theoretical concepts of the model, which may be built on hardware architecture or algorithmic specifications. Parallel programming models fit tasks from the parallel application to parallel hardware. They range from the application layer to programming languages, compilers, libraries, network communication, and IO systems[168].

Increasing the CPU frequency makes applications faster without changing the programming model; it consists in reducing the clock cycle, thus executing more instructions in the same duration. Putting more cores in CPUs, more CPUs in nodes, and pushing the limits to distributed configuration, with or without accelerators, introduce changes in the programming model and require new ones.

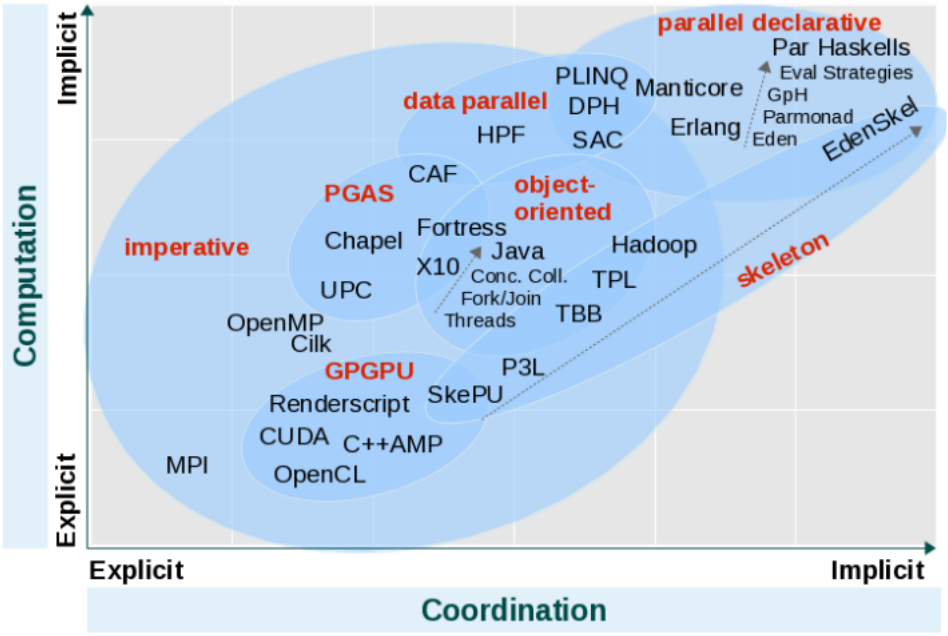


Figure 2.13: Programming model distribution according to their computation and coordination abstraction level: whether explicit or implicit, (Figure from [54]).

There are several attempts to classify programming models [54, 134, 114, 162, 113], focusing on several criteria such as the architecture (shared or distributed memory), type of parallelism, on abstraction level or productivity. The most relevant taxonomy to this work appears in [54]. It develops a classification over

several axes and shows a distribution of parallel programming models across two important dimensions, namely: computation (i.e. the algorithmic solution) and coordination (the management of parallelism), whether they are explicit or implicit, as represented in Figure 2.13.

We are interested in two main programming models in this work: the MPI programming model that can implement the Bulk-synchronous parallel (BSP) paradigm (Section 2.1.3.1). MPI is one of the most popular parallel programming models for distributed memory systems [61], and the distributed task-based programming model that is gaining more popularity for its simplicity and productivity. This last model intends to provide more and more implicitness in both parallelism and scheduling (computations and coordination). This is discussed in Section 2.1.3.2.

2.1.3.1 Bulk Synchronous Parallel Paradigm

In [166], *Leslie G. Valiant* introduced the bulk-synchronous parallel paradigm as a bridging model between software and hardware for parallel computing. He argued that such a model is analogous to the *Von Neumann* model and would get the same success as this last. A Bulk-Synchronous Parallel computer combines three attributes :

- a number of **components**, each performing processing and/or memory functions,
- a **router** that delivers messages point to point between pairs of components,
- facilities for **synchronizing** all or a subset of the components at regular intervals of L time units where L is the Periodicity parameter.

A BSP machine can be implemented using the well-known communication interface: Massage Passing Interface (MPI), where the components are defined as a set of P processes that share a common communicator. Each process has its separate local memory, performs a set of computations on its local data, and then exchanges some results with other processes. A local/global synchronization is performed periodically. Figure 2.14 shows an execution flow over time of an MPI program.

The MPI programming model suits very well the scientific applications where models are regular. The global domain is decomposed statically and explicitly within the processes. Each process has its local buffers updated as the simulation progresses, and parts of those buffers are exchanged with a subset of processes when needed. In this work, we focus on the MPI implementation of the BSP paradigm.

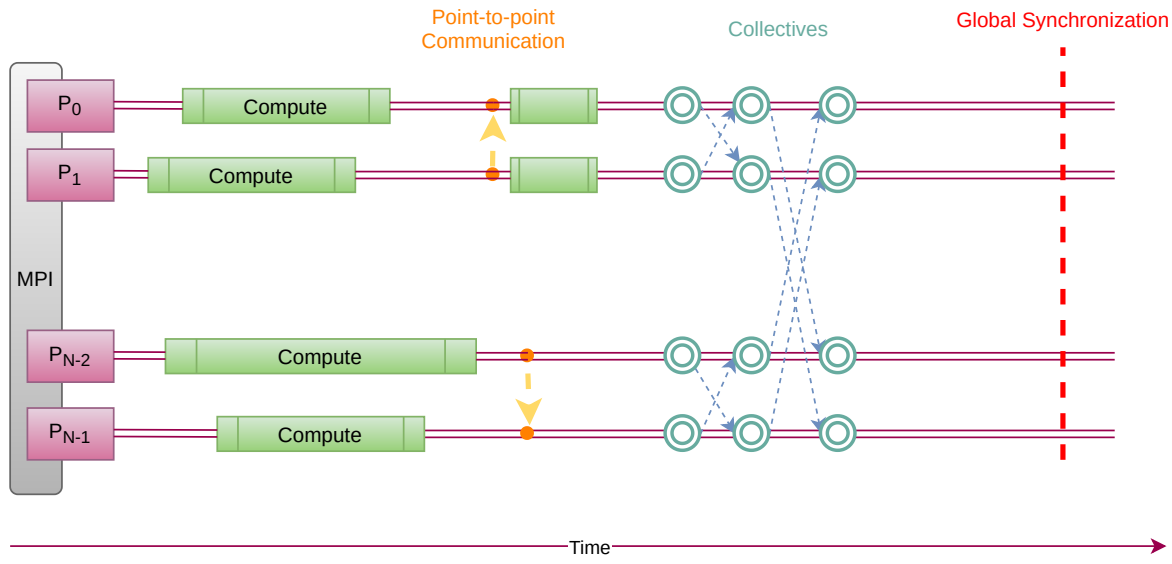


Figure 2.14: Execution flow of an MPI program represented by N processes. Compute, communication and synchronization regions have been represented over time, (Figure inspired by [112]).

Unlike several programming models that introduce new complex concepts, MPI was built using a relatively small number of well-defined and forward-looking concepts[165]. In the MPI-1 standard, an MPI process runs a program in its private address space and can communicate either through point-to-point message passing with another process or through collectives. The MPI standard has been then extended in several ways; however, using MPI may only require knowing a few concepts. In addition to

the strong base of MPI, it was designed to work with other tools. This characteristic is vital because the complexity of software and hardware keeps increasing. It also supports component-oriented software, thanks to communicators and groups, which is very important in designing modular or hierarchical code. Moreover, MPI is a complete model that can be used to write any parallel algorithm and is portable and can be used on platforms ranging from laptops to supercomputers[100]. All those characteristics make MPI perfectly fit high-performance applications and be a widely used model in HPC.

Note that, in MPI, the resource allocation and scheduling of the computations to processes are managed explicitly; while this is not a big issue for regular algorithms, it can quickly become complicated to manage for irregular ones.

2.1.3.2 Distributed Task-based Programming

Despite MPI success and efficiency, it is not the most suited model to parallelize irregular algorithms. This category of problems is characterized by irregular data structures and control patterns. Using static decomposition to parallelize them is not trivial. Higher-level programming models, such as task-based programming, reduce this complexity by cutting the computation into a set of tasks, and then a runtime dynamically schedules them.

Task-based paradigm has been introduced first in the shared memory context, Cilk [96], OpenMP [163] and Intel TBB [146] are examples of shared memory task-based tools. Task-based programming was then extended to distributed memory in [115], and used to design several frameworks and tools; we will give more examples in Section 2.3.

In the task-based paradigm, we define three main concepts:

- A number of **stateless tasks**: a task is defined as a sequence of instructions within a program that can be processed concurrently with other tasks in the same program[162]. A task has inputs and outputs. It can be either fine-grained or of a coarser granularity.
- **Dependencies** between tasks. A task can only be executed when all its dependencies are resolved,
- An **engine** that manages and schedules the execution of those tasks on a set of physical processes, which is usually called a scheduler,
- **Actor** is another concept that has emerged in several task-based systems. It is a stateful entity. It has internal attributes that may change when receiving messages from the environment. It may react by changing its internal state or/and sending responses.

In the task-based model, an application is split into tasks that are related to each other with dependencies to form a Directed Acyclic Graph (DAG): where the nodes are the tasks, and the edges represent the dependencies (Figure 2.15). A task with no input edges is called an entry task, while a task with no output edges is called an exit task[144]. The runtime scheduler analyses the DAG and decides which of the ready tasks to run on the available resources.

One of the main motivations for introducing higher-level models, such as the distributed task-based paradigm, is to create higher-level abstractions that make the design and implementation of parallel algorithms easier. The resource and the task scheduling (coordination) are managed by the runtime scheduler implicitly from the user's point of view. The task-based models can be less efficient in terms of performance compared to BSP-based models due to the overheads that may be introduced during runtime (due to the dynamic scheduling of tasks and message management at runtime). However, they widely increase productivity with the simplicity they introduce in designing and maintaining non-trivial applications.

In this work, we have chosen the Dask distributed framework as a distributed task-based tool to study and use, and we present in detail its architecture, task and data management and internal scheduling in Section 3.2

2.1.4 Discussion

While in situ processing is a relevant solution to avoid the IO bottleneck, it has several drawbacks, such as the setup complexity of in situ workflows and a priori knowledge needed to write relevant in situ analytics. In this work, we have a look at the complexity behind in situ workflows.

Most HPC simulations are written using the MPI programming model alongside other models, known as MPI+X, for their efficiency. Thus, most of the existing in situ tools are built on the MPI programming model, which is inherited from the host simulation. However, while MPI+X is the most suited model for

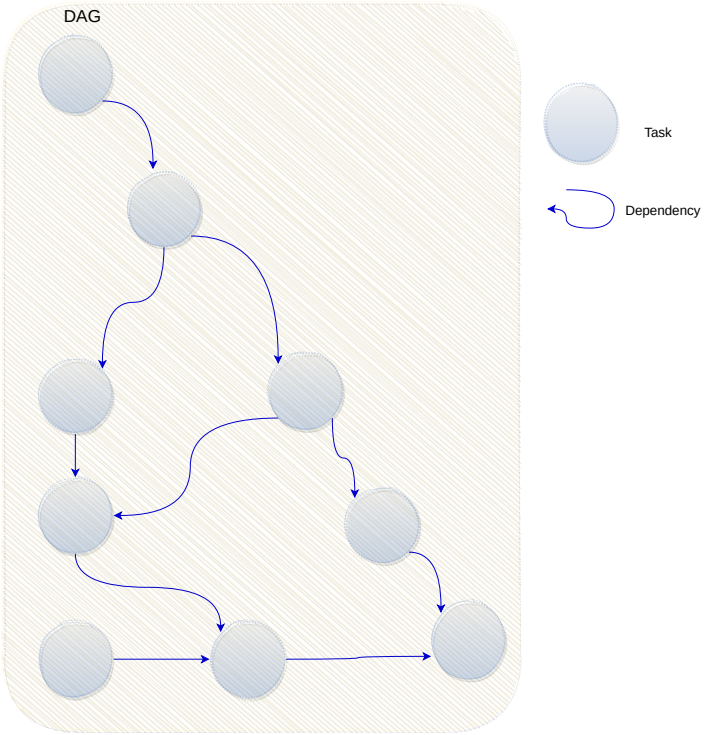


Figure 2.15: A directed acyclic graph: the nodes represent tasks, and the edges are the dependencies.

the HPC simulations, it does not suit well data processing algorithms for several reasons. First of all, HPC applications are usually characterized by a regular structure, where the program is a loop where processes usually run the same code and exchange data when necessary with neighbours, update the distributed data structure and synchronize globally. Adopting static and explicit parallelism using models such as MPI is relatively easy to apply and very efficient in terms of performance for such regular programs. On the other hand, data analytics algorithms are usually irregular; they may be characterized by one or more of the three types of irregularities, namely: data structures, control structures and communication patterns irregularity[117]. When the algorithm manipulates an irregular data structure, such as unbalanced trees, a need for dynamic scheduling and load balancing arises. When the algorithm presents irregular control patterns, synchronous models are not the most efficient. And when the algorithm shows irregular communication patterns, which is usually a result of the two precedent irregularities, non-determinism appears, and static models are not the most suited for such problems. Moreover, HPC simulations are programs that model a phenomenon, and the physics or logic behind it is less likely to change over time. It may be subject to improvement but rarely to core changes. Thus spending time/money on improving performance thanks to complex programming models is relevant because such codes are used for a couple of dozen years. On the other hand, data analytics algorithms may change from one study to another and are likely to change more frequently than the simulations themselves; moreover, they are less costly in computation. One would rather choose a less complex or easier model for such programs, even if the model is less efficient. In this work, we have opted for a distributed task-based framework that is both dynamic and asynchronous, with implicit parallelism usually ensured at runtime by a scheduler (see the used tools Chapter 3).

All in one, data analytics algorithms are different from simulations in their structure and execution; it is better to adopt a more suited model rather than taking the default option inherited from the host simulation.

2.2 In Situ Analytics

In this section, we present in the following sections some well-known in situ tools and frameworks, how the task-based paradigm has been used in the in situ workflows and finally, have a look at the big data frameworks already used for in situ analytics. We will have a discussion after each section to compare the presented tools.

2.2.1 General In Situ Frameworks

The in situ paradigm, as already presented, appeared for the first time in the paper of Kwan-Liu Ma *et al.* and has been applied first to scientific visualisation [129]. In this paper, which appeared in 2007, the authors already observed and declared that with the growing power of supercomputers and to be able to maximize the utilization of the data generated by simulations, one has to minimize or avoid IOs that are becoming a performance bottleneck. Thanks to this work, most scientific visualization frameworks that are meant for high-performance computing support in situ visualization. Paraview [38] and VisIt [63, 11, 172], which are built on the visualization toolkit VTK[26, 151], both support in situ visualization thanks to the Catalyst [92, 27] and Libsim [173] extensions, respectively.

ParaView is an open-source post-processing visualization tool built on MPI. It can run on computing platforms ranging from laptops to Exascale machines and process small to large datasets. Catalyst is an in situ library that enables easy integration of analysis routines within simulation codes. ParaView Catalyst [53, 24, 17, 68, 47] is the name given to the Catalyst implementation that uses ParaView for in situ analytics. It allows in situ processing and visualization workloads to run synchronously with the simulation by sharing the same data. This data needs to be transformed into VTK data structures by implementing an *Adaptor* to be understandable by ParaView. For most simulation codes, the coupling between the main simulation code and the adaptor will only involve three function calls. The first call initializes Catalyst and the pipelines; the second call performs any requested co-processing; and the third call finalizes Catalyst [4, 36].

VisIt is another well know visualization tool. It has a plugin architecture that allows to perform a wide variety of data processing operations and also import data from several data formats. It provides an interface for in situ analytics through libsim [64]. Libsim ensures two main functions: it creates an interface to map simulation data to VisIt format and manage VisIt events [84]. An *Adaptor* is implemented here also if the simulation is not compatible with the VTK data structures used by VisIt. Similarly to ParaView, Visit is built on MPI. It has a client/server architecture where one or more clients connect to a viewer, and remote servers run the in situ routines on the HPC plateform [25]. Figure 2.16 shows VisIt architecture where the input data is either the PFS or a running simulation.

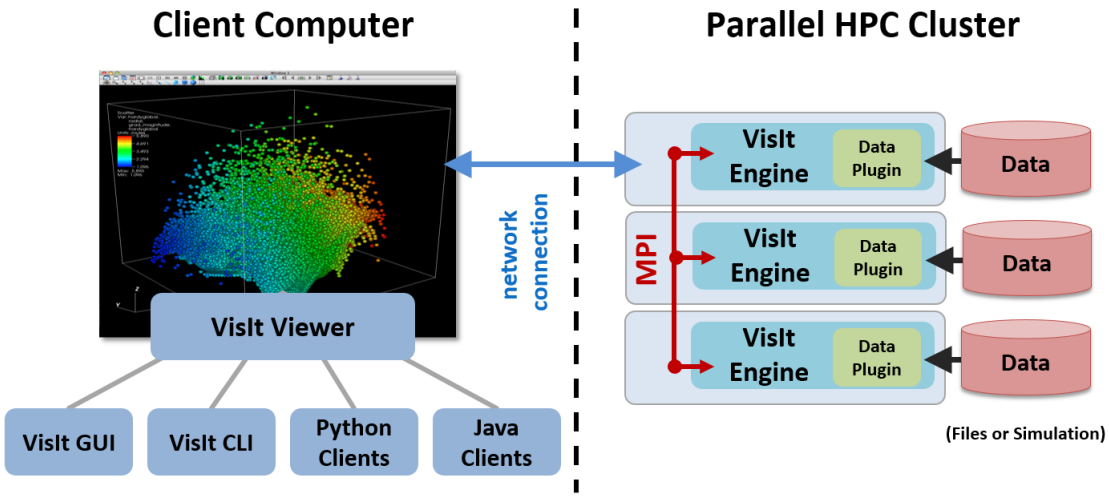


Figure 2.16: VisIt architecture showing client connected to the viewer, with remote engines on the HPC platform running within an MPI communicator. Since we are interested in in situ workflows, we suppose that the data is gotten from a running simulation rather than files, (Figure from [25]).

We also find several tools that were meant for IO management and then adapted for in situ analytics. ADIOS1[128, 110, 60, 51] and Damaris[79] are examples of frameworks in that category. FlowVR[87] on its side was developed for virtual reality and has been adapted to support both in situ and in transit analytics.

The Adaptable Input Output System (ADIOS) is a middleware that provides a generic interface to use transparently different IO transport layers and data handlers. It has a set of built-in IO systems, including HDF5, NetCDF, POSIX, and MPI-IO. ADIOS is configurable through an XML file where one can describe the data and select a library to handle it (write, read, or process) outside of the running simulation. FlexIO [179] is an example of an in situ tool built on ADIOS. It is a middleware for coupling

simulations with in situ analytics in ways that offer placement flexibility to those online analytics and visualization codes.

ADIOS2, the second generation of ADIOS, supports in situ processing as a built-in functionality thanks to the Sustainable Staging Transport engine (SST) that allows a direct connection between the simulations and the in situ processing codes. The SST buffers and sends the requested data over network using the same ADIOS write/read API as the file-based IO systems. The in situ workflows in ADIOS2 are set up similarly to IO workflows through the XML configuration file [124].

Damaris[79, 80] is another library that was designed originally for IO and then used for in situ processing. Damaris leverages helper cores and shared memory to reduce the IO jitter. It usually dedicates one core per node for IO operation, which is called a *server*. Damaris is a set of MPI processes running on a set of dedicated cores called severs. Just like ADIOS, Damaris uses an external file for configuration. In [78, 82, 83, 81] Dorier *et al.* use Damaris for in situ analyics. They keep the data in shared memory segments and perform in situ analytics and visualizations, filtering, indexing, and finally, IO in response to user-defined events sent either by the simulation or by external tools.

FlowVR [40, 39, 43] is a middleware dedicated to virtual reality (VR) that is built on the dataflow[156] paradigm and has been used for scientific visualization. An application in FlowVR is composed of modules exchanging data through the network. A module is a code that has been augmented with FlowVR methods. There is no explicit dependency between modules; they only exchange data with a daemon that runs on the same host. Once modules are defined, they are assembled, connecting their input and output ports. The application is thus represented as a dataflow where nodes are the modules, and the edges are First In First Out (FIFO) communication channels. With its dataflow architecture, FlowVR has been explored for in situ, in transit, and heterogeneous workflows in the work of M. Dreher *et al.* [84, 87, 88].

Due to its efficiency, in situ processing emerged quickly for general-purpose usage rather than focusing on specific tasks and visualization. Sensei [46, 56, 22, 34], is a generic in situ interface that focuses on having a unified API to instrument the simulation codes and making use of several external tools to handle data like Libsim, Catalyst, or user-defined analysis codes. Its architecture is built on three main components: a data adaptor to map simulation data to the VTK data model, an analysis adaptor to map the VTK data model to the model used on the analytics side, and a bridge to link the two adaptors and provide the methods called by the simulation to trigger in situ analytics. The bridge can be the VTK data model itself. Sensei already supports several backends, such as Alpine/Ascent/VTK-m[119, 132], ADIOS[128, 60] as well as for Paraview/Catalyst and VisIt/Libsim. Figure 2.17 shows a schematic architecture of Sensei.

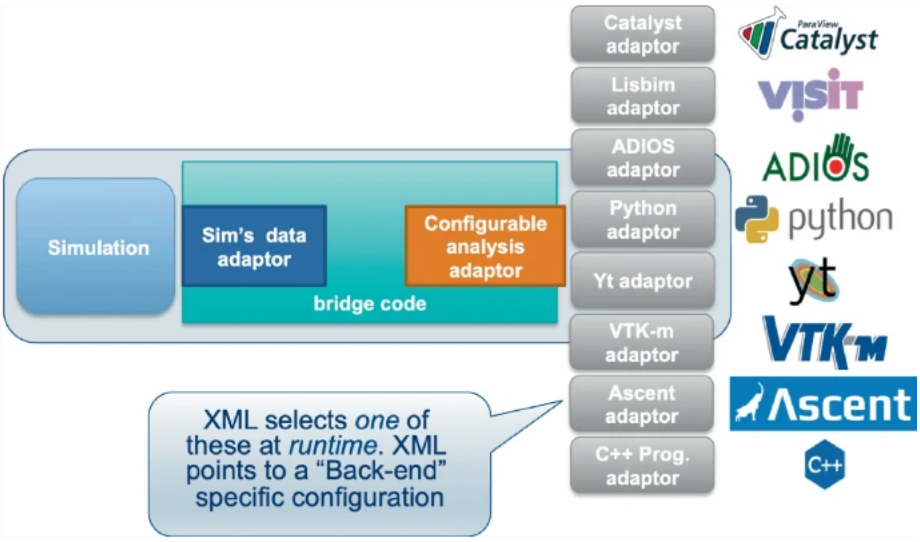


Figure 2.17: Sensei architecture showing a single data producer that has access to any number of potential in situ or in transit methods. The runtime choice, along with its associated parameters, is specified in an XML configuration file, (Figure from [56]).

Ascent[12, 119, 120, 121] is a lightweight in situ library that is designed to run in the same resources as the simulation. It integrates with many technologies (ADIOS, Babel Flow [141], ParaView/Catalyst and Python), supports both visualization and analysis routines, and provides support for modern su-

percomputers [65]. It uses conduit [118, 14] as a bridging data model between the simulations and the supported backends that simplifies describing hierarchical scientific data. It also implements VTK-h, to add a distributed memory layer to VTK-m, which already minimizes memory usage and execution time, to support both efficient shared and distributed memory configurations.

SmartSim[32, 30, 137] is a library dedicated to enabling in situ analysis and machine learning (ML) for traditional HPC simulations. It provides a different way to couple simulations with analytics by using the Redis[28] in-memory key-value store [109]. The Smartsim architecture is built on two main components: the SmartSim Infrastructure Library (IL) and SmartRedis. The first is a python-based workflow library that launches in-memory storage alongside HPC applications and facilitates the dynamic execution of simulations and ML infrastructure. The second consists of a lightweight client library used in applications to communicate with infrastructure launched by SmartSim. The data generated by the simulation by the clients into the Redis in-memory store. Then the SmartSim IL gets the data from the store and uses them to feed the ML models.

Decaf[86], a dataflow middleware for in situ workflows. Similarly to FlowVR, Decaf composes multiple executables to form a workflow. However, it does not rely on a separate daemon to manage the graph execution. A Decaf workflow is composed of several dataflows. Each dataflow is the association of a producer, a consumer, and a communication object called a *link*. The *link* is deployed in a set of separate resources where operation on the data structure can be performed. Bredala[85] is built on Decaf; it provides an API to construct a data model with enough information to keep its semantics while splitting and merging. The work on Decaf and Bredala was a strong base to propose a new concept to automatically extract needed data for analytics at the producer, called contracts[131].

DataSpaces [76] is a distributed virtual shared memory space implemented on staging nodes. In a client/server fashion where the running simulations are clients, and DataSpaces nodes are servers. The goal of DataSpaces is to enable the data of interest, which is extracted from a running application, to be efficiently indexed and asynchronously accessed and processed by other components in the simulation workflow. Data extraction is performed by the Decoupled and Asynchronous Remote Transfers (DART) library [74, 75]. DART is built on Remote Direct Memory Access (RDMA) technology to enable fast, low-overhead and asynchronous access to data from a running simulation, and support high-throughput, low-latency data transfers [55]. DataSpaces extends existing parallel programming models, such as MPI and Partitioned Global Address Space (PGAS), with a simple set of APIs. In order to enable the coupling of workflow component applications, DataSpaces provides the `put/get` operators to access the virtual shared store. As already mentioned DataSpaces servers are launched in staging nodes. Thus it enables in transit workflow configurations rather than in situ.

Wu *et al.* in [175] propose a declarative and reactive language and runtime for in situ visualization called DIVA that can extend existing in situ systems such as VTK. DIVA is built on MPI and consists of two main components: functional reactive programming (FRP) visualization-specific language and a low-level C++ dataflow API. FRP[170, 58, 140] is a programming paradigm that describes systems that operate on time-varying data, which is well adapted for in situ visualization purposes. The users describe their codes using the declarative API. Then the language parser translates it into an internal DAG. This last is finally interpreted then to the low-level C++ dataflow API for execution. DIVA language does not support directly in transit workflows.

2.2.1.1 Discussion

In this section, we have presented a set of in situ tools, ranging from visualization and AI-specific to general-purpose in situ workflow management systems. We had a variety of tools that could be classified according to multiple axes. For instance, ParaView Catalyst, and VisIt Libsim are synchronous, whereas the other presented tools may run asynchronous analytics too. One can compare how in situ analytics are coupled to the simulation, either embedded in the code or less intrusive such as in FlowVR and DataSpaces, where simulations are not aware of other applications. The data model is also different. Several tools are based on VTK or support it (ParaView Catalyst, VisIt Libsim, ADIOS, Sensei and Ascent), SmartSim uses Redis in-memory store, and Bredala implements its own data model. The in situ workflows may be statically generated and configured, or in situ tasks may be created in response to specific events. For instance, Decaf or FlowVR workflows are created with a static configuration where we know a priori the producer and the consumer that we link together, unlike DataSpaces and SmartSim, where the clients may request analytics tasks as moving on.

Despite all these differences, most of those tools share a common specification which is relying on static parallelization. Mapping the analysis workflow tasks to compute resources statically often leads to high performance but requires the user to control this mapping explicitly. The underlying transport

layer is often based on MPI which simplifies the coupling with the simulations, but may imply rewriting the parallel in situ analytics with MPI too, which is very complicated for the reasons we have already mentioned in Section 2.1.4.

2.2.2 Task-based Programming for In Situ Workflows

In this section, we will have a look at the use of the task-based programming model in the in situ workflows and discuss the reasons why we need to work more on this topic.

Task-based programming in shared memory is commonly used today in scientific applications to efficiently leverage nodes architecture, using OpenMP[163] or Intel OneAPI Threading Building Blocks (TBB)[146], respectively in GoldRush and TINS that we detail here.

GoldRush[180] is an in situ technique that exploits idle node resources for in situ data analytics. It is built on ADIOS IO system and FlexIO transport, already presented in section2.2.1. Unlike other in situ methods, GoldRush detects when the simulation does not use all available cores to launch in situ tasks in idle threads as long as there is enough memory. Typically this is possible within non-parallelized computations, MPI communications, and file IOs. At the end of an OpenMP region, a SIGCONT signal is emitted to start in situ analytics, and a SIGSTOP signal stops it when the simulation reaches the next OpenMP parallel region. This approach has several benefits, including the efficient use of compute node resources and reductions in data movement overheads.

TBB library provides a task-based programming model and a work-stealing scheduler for shared memory. It has been used in TINS[73, 71] for in situ analytics. In this work, both simulation and analytics are task-based, and TBB is used to dynamically distribute the simulation and the analytics tasks to the available cores. Even if TINS and GoldRucsh use similar approaches, they act on two different levels: TINS at the task level and GoldRush at the system level. TINS has been compared to both GoldRush and Damaris with static and dynamic helper core strategies in [72], and the results show that TINS outperforms both tools.

Task-based programming has been used in an in transit configuration in Sun *et al.* paper [157]. The proposed work is built on the DataSpaces system. It proposes an asynchronous coupling of distributed task-based scientific workflow where both simulation and in situ analytics are parallelized in a task-based fashion.

2.2.2.1 Discussion

In this section, we have focused on the usage of task-based programming for in situ workflows. The utilization is restricted to shared memory with TINS and GoldRush in addition to eventual embedded in situ routines in simulation codes. The only attempt to use task-based programming for in transit analytics was in Sun *et al.* paper, where both the simulation and the analytics were parallelized in tasks.

To the best of our knowledge, there is no work trying to couple distributed task-based programming with MPI for in situ/in transit analytics. Shared memory task-based models are usually easy to couple with other programming models, such as MPI. As their utilization is encapsulated in one MPI process, (like if it was just a sequential program), which does not create integration complexity or incompatibilities between the two models' concepts. Consequently, using them does not raise the same challenges as coupling distributed task-based programming with MPI, which we will discuss in Chapter 4.

2.2.3 Big Data Frameworks for In Situ Workflows

In this section, we present the attempts to use big data frameworks for in situ analytics.

Big Data models are built on several other programming models such as functional, SQL-based, and Actor models. The current defacto framework/model for writing data-centric applications is Map-reduce [174]. SMART [171] (inSitu MApReduce liTe) proposes a Map-Reduce [69] interface for programming in situ analysis on top of MPI/OpenMP. A Map-Reduce program is composed of a map and reduce procedures. The map method performs filtering and sorting, and the reduce method performs a summary operation. SMART supports a variety of scientific analytics on simulation nodes, with minimal modification of simulation code. It supports efficient in situ processing by accessing simulated data directly from memory in each node of a cluster or a distributed memory parallel machine. SMART is the first in situ framework based on a Map-Reduce-like model.

Paper [178] appeared in 2018 and uses another big data tool in in situ workflows. It takes benefit of Flink [62, 95] stream processing support for enabling in transit analysis. The architecture of the proposed framework connects the simulation parallelized with MPI to Flink worker nodes using ZeroMQ [105]. Flink

executes the analysis scripts in parallel, then injects results to the HBase distributed database [169], which takes care of storing the results using its local disks. The model provides a loose control on data partitioning that is not well adapted to support efficient parallelization of patterns such as stencil computations[176] or large-scale linear algebra.

2.3 Distributed Task-based Frameworks

Distributed task-based models alongside Python-based programming are gaining more and more popularity for the intuitive programming interface and orchestration they provide together. In this section, we present some of those systems and highlight their differences.

Parallel Scripting Library (Parsl) [48, 50, 49] is a Python-based scripting library that can express parallelism between both Python code and components written in other languages. Parsl uses `Apps` decorators to intercept and modify the behaviour of Python functions: the `@python_app` decorator is used for pure Python functions and `@bash_app` decorator for shell commands. When `App` decorator is invoked, an asynchronous task is registered, and a `Future` object is returned immediately. `Futures` can be passed as an argument to `App`, indicating a dependency between the two tasks. Parsl runtime manages the execution of the parcel-annotated programs on the configured resources by creating a dynamic task graph, where nodes are programs (tasks) and edges are the input/outputs exchanged between tasks. The data can be any serializable Python object, files or `Futures`. Parsl provides several executors, depending on applications, which allows running those tasks on one or more target execution resources. Figure 2.18 shows how Parsl programs are transformed into DAGs and then run in the executors.

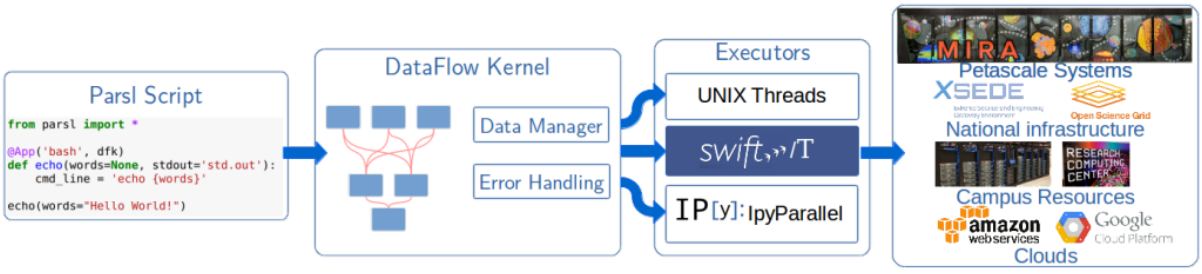


Figure 2.18: Parsl architecture: DataFlow Kernel (DFK) maps parsl-annotated scripts to Executors that support diverse computational platforms, (Figure from [50]).

Writing parallel programs in Legion[52] needs both programs to be expressed in terms of `Tasks` and data into `Regions` to be distributed across several machines. Logical regions are the fundamental abstraction used for describing program data in Legion applications. Pygion[155] is the Python high-level interface of Legion. Here again, Python decorators are used to mark functions for parallel execution. A program is divided into several tasks using `@task` to be executed in parallel and program data into `Regions` and `Subregions` to express data parallelism. The arguments to `Tasks`, and the `privileges` requested on those arguments (read, write, etc.) are used to compute a dependency graph between tasks that guides the parallel and distributed execution of the program. There are also dependencies between two tasks if they access overlapping data, and at least one of the tasks wants to write in that region. Early experiments for in situ visualization subsystem were prototyped using Legion in [103]. It shows Legion runtime manages to interleave simulation and visualization tasks without reducing the simulation throughput. However, using the implemented tool requires code modifications to redesign an MPI+X application into a Legion.

Several other interesting task-based frameworks are found in the literature. For instance, StarPU [44, 45, 42] is a runtime system for scheduling a graph of tasks onto a heterogeneous set of processing units. It provides a unified execution model and data management library for heterogeneous systems. Parallel Runtime Scheduling and Execution Controller (PaRSEC) [59, 107] is a task-based runtime for distributed architectures capable of tracking and moving data from different nodes. Dependencies between tasks are explicitly described, and the task graph is generated automatically from a domain-specific language (DSL). These approaches can improve usability within a domain as long as the target programs are well supported by the domain-specific semantics [155]. In addition to the DSLs, PaRSEC runtime has several components namely: schedulers, communication engines and data interfaces. Despite the similarities between Those tools, only Pygion supports data partitioning [164] through `subregions`; the others require

users to reorganize data in applications that use multiple access patterns explicitly.

PyCOMPSs[159, 145] is the Python binding of the COMPSs[158] task-based system. Similarly to Parsl and Pygion, the user has to identify the task that may be run in parallel and annotate them using decorators. And then, a runtime system analyses the script to identify the dependencies between those tasks depending on the arguments of the `@task` decorator. Among other arguments, there are the parameters of the decorated function, whose name is the formal parameter's name and whose value defines the type and direction of the parameter. Those arguments are needed to construct the dependency graph. PyCOMPSs added support to a distributed data structure, namely *ds-array* [66] that offers a parallel and almost the same API as NumPy[3]. It also supports MPI programs [90] as tasks to be integrated into the task graph. PyCOMPSs is part of the eFlow4HPC[4] project to provide HPC workflows as a service. It allows streaming communication between different parts of a workflow that can be used to evaluate intermediate results and enables the implementation of in situ optimization algorithms [89].

Until now, an MPI application can be integrated into PyCOMPSs as a task. In other words, the MPI applications are encapsulated into tasks, so they are started when the task is launched, and once they are finished, they share the produced data with that task, which is another possible way to couple MPI with task-based programming.

In the context of Parallel libraries, Dask [147] is one of the most interesting task-based Python frameworks because it offers access to parallel versions of well-known libraries such as NumPy and Pandas[5] with `dask.array` and `dask.dataframe`, respectively. Those two alongside `dask.bags` are called collections. It also provides a parallel version of Scikit-learn[6] called `dask_ml`. Dask is able to construct task graphs automatically thanks to blocked algorithms, that resolve small problems to compute a larger one. For instance, in order to compute the sum of a large array, we can compute the sum of smaller blocks and sum all intermediate sums. In addition to those high-level collections, there is also a possibility to construct the graph manually using a lower-level API using `Futures` and decorated functions.

Ray [133], another well-known task-based framework, has a distributed scheduler. It is used to scale both artificial intelligence (AI) and Python applications. The system architecture can be structured into two main components: the Ray core, which enables scalable applications to be built in pure Python, and Ray AIR [20], which provides several libraries (*Datasets* for distributed data preprocessing, *RLlib* for Reinforcement Learning [126], *Tune* for scalable hyperparameter tuning[127] and others) that enables simple scaling of AI workloads [21]. Ray provides a scheduler for Dask: Dask _on_ray[23]. This takes advantage of both frameworks: Dask collections to write analytics using familiar APIs and Ray-specific features such as the distributed scheduler, shared-memory store and others. Ray is less mature compared to Dask and does not have built-in primitives for partitioned data. However, it is more suited for heavy workloads and AI, where it has been shown that it outperforms Dask [142].

An interesting taxonomy of task-based programming models with recommendations is already done in [102].

2.3.1 Discussion

In this section, we have presented some of the well-known distributed task-based frameworks. We tried to pick the most relevant to our work either for their ease of use, related tentative for in situ usage or coupling with MPI applications.

Each of the presented frameworks has interesting features that make it relevant to specific needs. For instance, Parsl is a good candidate for general-purpose usage thanks to the different executors it offers. Legion has the particularity of being able to express data parallelism in addition to tasks which makes it more interesting to use in data-driven workflows. PaRSEC generates its task graph from a DSL which is an interesting approach too, because it takes advantage of the dynamicity of task-based programming at a low level and the abstraction level of DSL at a higher level. PaRSEC scheme is very interesting as it could be used for domain-specific in situ analytics, coupled with MPI simulations. Such configuration would be relevant for domain scientists that are used for DSL usage. The issue with such a scheme is the need for the development of new task-based DSLs. PyCOMPSs and other development in the eFlow4HPC is also interesting as it considers providing HPC workflows as a service, which would likely hide all coupling complexity and allows using PyCOMPSs to write task-based in situ analytics, but for the moment, no work has been published on this topic. From the user API perspective, Dask seems to be the most relevant thanks to the provided distributed libraries, which allow writing almost sequential

[3] https://numpy.org/
[4] https://eflows4hpc.eu/
[5] https://pandas.pydata.org/
[6] https://scikit-learn.org/stable/

Python codes that run in parallel. Ray on its side may be the best tool for AI workloads as it provides both interesting API and performance compared to Dask.

The rest of this discussion is driven by two main aspects: coupling distributed task-based frameworks with MPI, and the most relevant tool to use for in situ processing among all cited examples. First, there were relevant attempts to couple distributed task-based tools with MPI: in PyCOMPSs [90], an MPI application can be launched in a PyCOMPSs task. This is a way to perform the coupling. However, it is not possible to use it in our case for several reasons: first, in our work, we consider scientific applications that generate huge amounts of data; launching them within a task and then gathering the results in the same node may not be possible for memory reasons. The second issue is that scientific applications are usually iterative; it is not possible to extract the data at each timestep from a task (to perform in situ analytics) and keep it running. This violates the definition of a task. Note that we want to simplify writing in situ analytics using distributed task-based frameworks while keeping the MPI simulation as is, so we don't consider rewriting those simulations in other paradigms.

Another attempt was to use MPI with Dask in [152], but it was as a transport layer rather than coupling with MPI programs. Mainly, in this work, the authors have added a new implementation using MPI to the already-used RPC communication system to replace TCP or UCX. This work can be considered to optimize a solution based on Dask to unify the transport layer between simulations and analytics written in Dask.

The second aspect is to pick the most relevant tool to use for in situ processing, and this time, from the user perspective. We want to propose a solution that minimizes the changes in the existing post hoc analytics codes to work in situ and to facilitate the development of new ones. Given that Python is the most used language for data analytics, Python-based tools are more advantageous, but it is not enough, as most of the presented tools are. The interesting aspect to consider is the available distributed APIs to write analytics and whether they are similar to sequential Python APIs, which scientists were used to to write post hoc analytics. And here, PyCOMPSs and Dask already provide parallel versions of some known libraries, which makes them more favourable.

2.4 Summary

In this Chapter, we have defined our context and discussion the related work to our topic. We have defined high-performance computing and justified the need for it to solve complex problems. We have presented two of the programming models: message passing and distributed task programming, which are used to program those huge machines, and then we raised the IO bottleneck issue that faces large-scale simulations in domains such as nuclear fusion. We have presented two analytics workflows used to process the data generated by HPC simulations, namely: post hoc and in situ workflows. The first suffers from the IO bottleneck, and the second is quite complicated to set up.

Our goal is then to provide a solution that brings together the in situ performance and the post hoc ease of use. Thus in the second part, the related work, we address mainly two topics. The first presents and discusses the existing in situ tools, how they work and their specificity, and eventual attempts to use task-based programming or other big data models for in situ analytics. The second presents task-based frameworks and where they have been used for in situ analytics or at least coupled with message-passing programs.

Chapter 3

Used Tools: Dask Distributed and PDI

If I have seen further than others, it is by standing upon the shoulders of giants.

Isaac Newton

Now that we have had a look at some of the existing in situ platforms and tools and have an idea about the emergence of distributed task-based programming in big data and some attempts at using it for in situ analytics, we are going to present the tools we will use in this work.

We have opted for Dask Distributed (which is the distributed version of Dask) to use as a task-based framework for in situ analytics. This choice has been motivated by the ease of use of Dask, from the user perspective, and we target a more general-purpose tool rather than a specialized one.

In addition, we have chosen to provide a clean solution that separates concerns, namely the physics under study and the way we handle data. We have opted for PDI data interface as a data handler (which is developed in Maison de la Simulation). It is used in multiple production codes such as Gysela[1], ARK[2], Alya[3] and GYM-DSSAT[4].

[1] https://gyselax.github.io/
[2] https://gitlab.erc-atmo.eu/erc-atmo/ark
[3] https://compbiomedeu.github.io/applications/Alya/Alya.html
[4] https://rgautron.gitlabpages.inria.fr/gym-dssat-docs/

3.1 Overview

In this section, we present the tools that we consider in the approach that we propose in Part II. Our goal is to provide an in situ approach that is based on a distributed task-based framework to take advantage of its dynamicity. For that, we have opted for Dask Distributed framework, which is the distributed version of Dask (presented in Section 2.3). Our choice has been motivated by several aspects; first of all, the paradigm itself suits data analytics better than MPI and static models. It simplifies the implementation of non-trivial and irregular algorithms and hides all communication and scheduling aspects. The second reason is that Dask distributed is Python-based, thus, we take advantage of Python simplicity, productivity and all the Zen of Python and bring them to he in situ context. The third and most important reason which distinguishes Dask distributed from the other tools is the set of well-known parallel libraries it provides, such as parallel `Numpy`, `Pandas` known for `dask.array` and `dask.dataframe` respectively. It is important as domain experts are used to these tools, thus, it is easy for them to jump to the parallel version that has almost the same API as the sequential ones.

PDI data interface is the data handler we have chosen to use in this work. It will extract the data from the simulation and make it available for in situ processing. We have opted for it rather than other data interfaces for several reasons. First, it is process-local, extra resources are not needed to extract the data for the simulation. It already provides an interface with Python through the lightweight library `Pybind11`. PDI provides an outstanding separation of concerns (simulation and data handling): this is an important pattern we wanted to consider in this work, as our objective is to provide an elegant (but simple) way to process data in situ. Its data model is well adapted for scientific data and a set of types are already available to use. Moreover, PDI design allows the implementation of modular software without developing efforts, where data exchange can be easily handled through its data store and data handling through an extensive set of plugins. PDI is developed in our lab, thus, we have more expertise and support on it, moreover, it has already been used in the two production codes that we present in Chapter 6. Moreover, as our production use cases already use PDI, there is no further needed implementation in the simulation codes to support in situ analytics. It is just about configuration.

Note that some sections are technical. However, they are important to understand our contributions and implementations in Chapter 5 and Chapter 6.

3.2 Dask Distributed

In this section, we will present the Dask distributed framework and how it operates internally. When we started working on the project, the documentation regarding the scheduling, the internals and how the Dask distributed operates and manages tasks internally, and the different actors were not as developed as they are today [16]. So most of the concepts that are presented are coming mostly from the paper [147], Dask GitHub repository code [41].

3.2.1 Overview

Dask is a Python framework that enables parallel and out-of-core computations. This work is based on the distributed version of Dask, named Dask Distributed because our goal is to perform in situ analytics for large-scale distributed simulations. By abuse of language, we will say Dask instead of Dask Distributed in the rest of this document.

Dask is built on the client/server scheme. It has three main components: one or more *clients*, a *scheduler*, and one or more *workers*, as shown in Figure 3.1. The client is the entry point to the Dask cluster; it represents an interface between the end-user and Dask. It submits analytics to the scheduler as a task graph. The scheduler analyses the task graph and checks for any connected workers; if so, it sends the ready tasks to the idle workers. The workers are multi-threaded processes that perform the computations and store or share the data. The source of the data in Dask is usually a storage system.

The code in Listing 3.2 is typical client code in a Dask workflow. It runs following these steps that are also represented in Figure 3.1: the client first connects to the scheduler. It reads the metadata of the HDF5 file `"data.hdf5"` from the parallel file system. A `dask.array` object is created from the descriptor returned in the previous step. The `mean()` method and the multiplication operation create a task graph that is submitted to the scheduler by the call to `compute()`. The task graph generated by Listing 3.2 is shown in Figure 3.2 in Page 37.

The scheduler then sends it to the workers as they become ready. The first ready tasks are `getitem` tasks that read data from the file. Note that it is at this stage (when tasks are sent to the workers) that

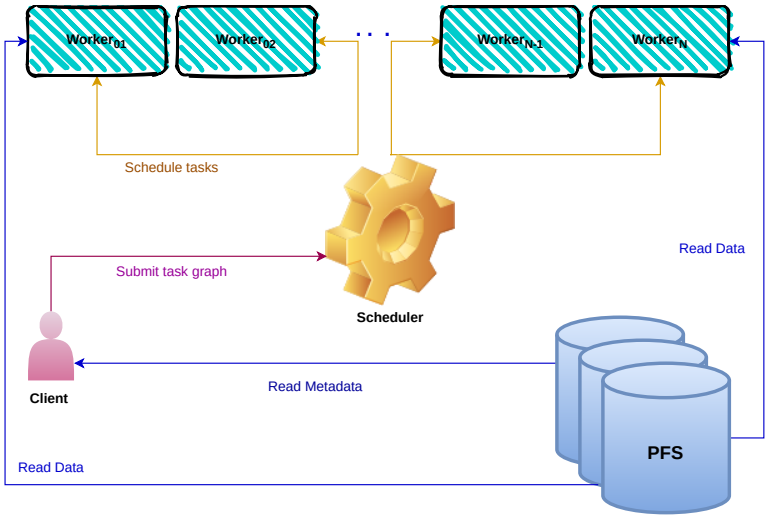


Figure 3.1: Dask distributed architecture in a typical post hoc workflow. A client and N workers are connected to the scheduler. 1) The client reads small metadata regarding the needed files from the PFS, 2) creates the Dask data structure and submits a task graph to the scheduler, 3) the scheduler analysis the graph and submits tasks to the workers, 4) the workers execute the tasks, 5) some of them read data blocks in parallel from the PFS.

the data is read from the file system, which makes it possible to process data larger than the memory of one node in parallel. Once all the tasks are computed, the result is returned to the client.

Listing 3.1 represents a typical python analysis without Dask; Listing 3.2 illustrates a Dask parallel equivalent.

3.2.2 Tasks in Dask Distributed

The tasks are a central concept in Dask and all task-based frameworks; it is the smallest piece of work that can be submitted to the scheduler and run by a worker.

In this part, we detail how Dask handles and implements this concept, starting with task graph creation, the task state transition, and finally, task scheduling. We define and discuss *pure data tasks* and how they are used in our work. Alongside those important low-level details, we present Dask collections that will strengthen our choice by showing the ease of use of Dask and its underlying submodules.

3.2.2.1 Task Graph

Every script submitted to Dask scheduler is first translated to a task graph, specifically a directed acyclic graph of tasks with data dependencies. The *graph* is represented as a dictionary, where the keys (identifiers) are any hashable value that is not a task, and the values are computations. A *computation* can be a key present in the graph, a value such as an integer, a task, or a list of computations. A *task* is described in the dictionary as a Python tuple that has a callable as a first element, such as a function or an object of a class that implements the *__call__()* magic method; followed by a list of arguments, and an argument may be any valid computation. For instance `(function1, arg1, arg2, arg3)` is a task that applies `function1` to `(arg1, arg2, arg3)` where the arguments are valid computations such as: `arg1: 1, arg2: (function2, arg4), arg3: [(function3, arg5, 0.2), 5]`.

When executing the task described by `(function1, arg1, arg2, arg3)` in the dictionary, the worker runs `function1(arg1, arg2, arg3)`, by moving the opening parenthesis one term to the left, the execution of `function1` is delayed. This representation allows Dask to store this computation as data that can be analyzed by the scheduler later rather than cause immediate execution.

The user can use the `Delayed` API to customize and create his own task graph. It may be advantageous when the user has particular processing which is not covered by the high-level collections. Moreover, Dask implements a `future` API, which is immediate rather than lazy. Listing 3.3 shows an example of task graph creation in Dask using the low-level `Delayed` decorator. The created task graph is shown in Figure 3.3. We use the `delayed` decorator to declare lazy functions to Dask scheduler, and then we use them to create a task graph.

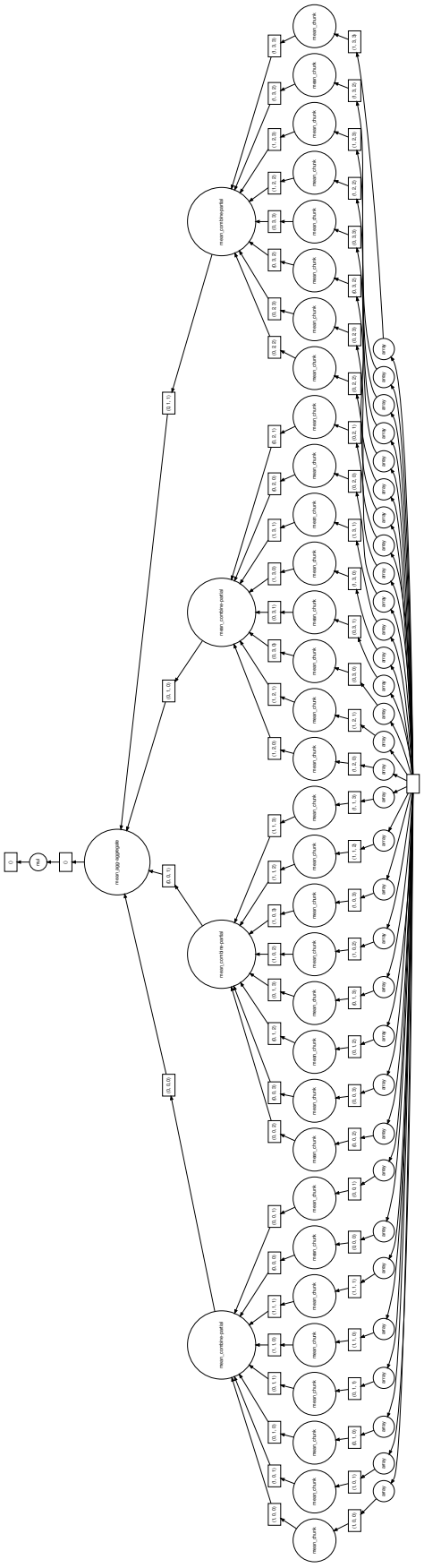


Figure 3.2: Dask graph generated in Listing 3.2. The HDF5 dataset size is $(2, 20, 20)$ and chunk size is $(1, 5, 5)$. From the bottom to the top of the graph, we have 'array' tasks that correspond to reading the chunks from the file, followed by local 'mean' computations and then the 'mean' aggregations. And finally, a 'mul' operation corresponds to '*200' in the script.

```
import numpy as np
import h5py
# Load data from HDF5
data = h5py.File('data.hdf5',mode='r')['dataset1']
# Compute the mean of the array
computed_mean = np.array(data).mean()*200
print("Computed mean : ", computed_mean)
```

Listing 3.1: Sequential post hoc mean using *Numpy*.

```
import dask.array as da
import h5py
# Connect to Dask
client = dask.distributed.Client(address)
# Build a lazy array descriptor from HDF5
data = h5py.File('data.hdf5',mode='r')['dataset1']
daskdata = da.from_array(data, chunks=(1, 5, 5))
mean = daskdata.mean()*200
computed_mean = mean.compute()
print("Computed mean : ", computed_mean)
```

Listing 3.2: Parallel post hoc mean with Dask. Lines differing from the analysis of Listing 3.1 are highlighted.

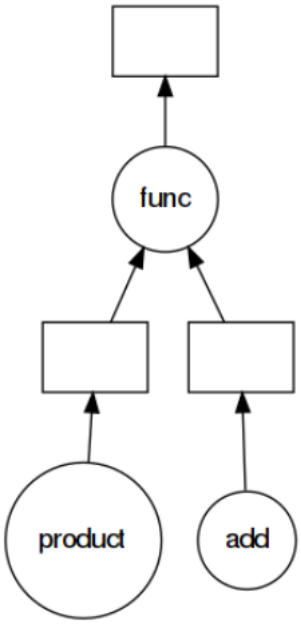


Figure 3.3: Dask graph generated in Listing 3.3. From the bottom to the top, we first compute the 'product' and 'add' functions asynchronously, and then we apply the 'func' function to their results.

It is represented as a dictionary (Line22) where we can see the keys generated by Dask, the functions and the arguments, for instance: `'func-f0310b3c-f9ed-4199-b3bb-dda81384823a': (<function __main__.func(a, b)>, 'add-e11deea3-1568-4abf-9ae5-a6491a7f46d8', 'product-e7451475-cfe2-4ae5-9358-191db215ea3b')` is the node in the task graph created by line19, `'func-f0310b3c-f9ed-4199-b3bb-dda81384823a'` is the key of this task, `<function __main__.func(a, b)>` is the callable, `'add-e11deea3-1568-4abf-9ae5-a6491a7f46d8', 'product-e7451475-cfe2-4ae5-9358-191db215ea3b'),` are the keys of arguments.

3.2.2.2 Dask Collections and Futures

In addition to the `@delated` decorator, used to create task graphs, Dask supports parallel versions of familiar libraries such as `Numpy` and `Pandas`, respectively, `dask.array` and `dask.dataframe`, with almost similar APIs to thier sequential counterparts. Thus, they become usable in larger-than-memory problems. In this work, we have been particularly interested in *dask.array*. A *dask.array*[147] is a potentially larger-than-memory numpy-like array; it is constructed by aggregating blocks of `numpy` arrays called chunks. Operation on a *dask.array* generates a task graph automatically, thus called a high-level collection.

The user writes code very close to sequential one using those available APIs. A task graph can be constructed and then submitted to the scheduler by calling specific functions such as `compute`. Listing 3.4 shows a Dask code, where the `dask.array` API is used alongside `@delayed`. In this example, we compute the sum of a *lazy* random array. Then we apply the decorated `product` function we created in the previous example to this sum and `p`, then compute the result using the *compute* method. The generated task graph from 3.4 is shown in Figure 3.4 in page 41.

```
import dask

# Define needed functions
@delayed
def add(a, b):
    return a+b

@delayed
def product(a, b):
    return a*b

@delayed
def func(a, b):
    return product(a, b) - add(a, b)

# Create task graph using our functions
a =  add(1, 2)
p =  product(3, 4)
f =  func(a, p)

# Show the created task graph as a dictionary; the result is in the following comment
dict(f.dask)

"""
dict(f.dask) shows the dictionary created by the lazy computation `func` applied to `a`
     and `p`
{'func-f0310b3c-f9ed-4199-b3bb-dda81384823a': (<function __main__.func(a, b)>,
  'add-e11deea3-1568-4abf-9ae5-a6491a7f46d8',
  'product-e7451475-cfe2-4ae5-9358-191db215ea3b'),
 'add-e11deea3-1568-4abf-9ae5-a6491a7f46d8': (<function __main__.add(a, b)>,
  1,
  2),
 'product-e7451475-cfe2-4ae5-9358-191db215ea3b': (<function __main__.product(a, b)>,
  3,
  4)}
"""

# Visualize the task graph
f.visualize()
```

Listing 3.3: Task graph creation with *Delayed*.

```
import dask
import dask.array as da

# Create a random dask array of size 20*20 chunked into 5*5 blocks
darray = da.random.random([20, 20], chunks=(5,5))

# Compute the sum of all elements in the array
s = darray.sum()

# Visualize the created task graph
s.visualize()

# compute the product of `s` and the computed `p` from delayed in the previous example
c = product(s, p)
result = c.compute()
```

Listing 3.4: Dask example using the *dask.array* submodule.

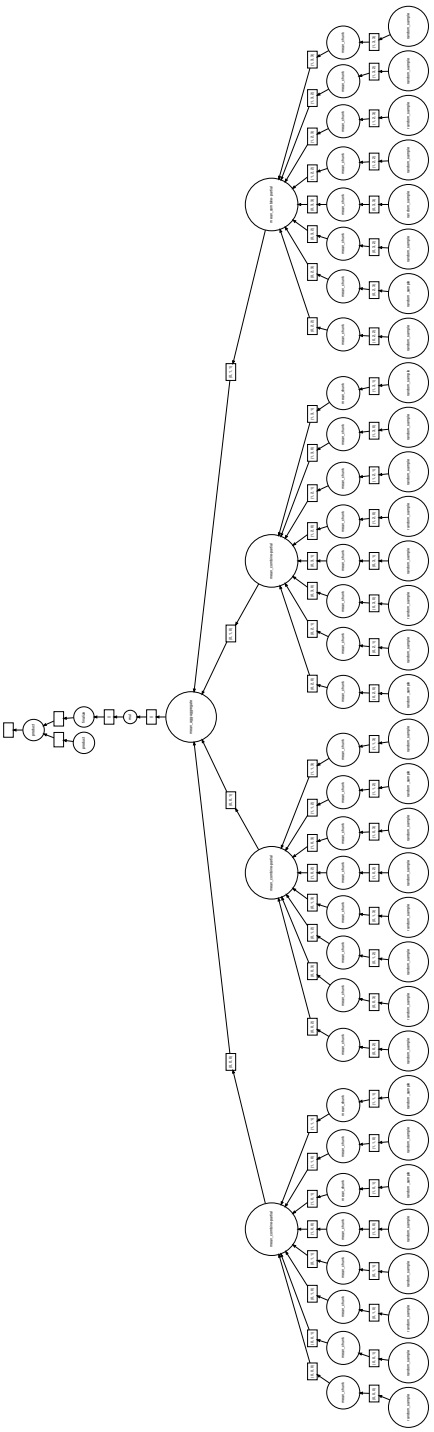


Figure 3.4: Dask graph generated in Listing 3.4. The circles at the bottom of the graph represent the task that will generate random data chunks. They will then be summed into partial sums, which will be aggregated to the total sum, which is then multiplied by the ‘p‘ computed in the previous example.

3.2.2.3 Task journey

Task graph creation is not the only step that is done before getting to the scheduler. Either the graph is constructed using high-level APIs or `@delayed` it goes through the following steps:

- graph creation: as already presented in 3.2.2.1, the graph is encoded using a Python dictionary, and it may include millions of entries. This step is done on the client side,
- graph optimization: Dask tries to optimize the graph and eliminate unnecessary work. This may take some time if the graph is large,
- graph serialization: the graph needs to be sent from the client to the scheduler and then to the workers, so it must be converted into bytes before sending it. This is done in the serialization step.
- graph communication: once the graph is serialized on the client side, it is sent to the scheduler.

These steps are done after calling the `compute/persist` and may take some time if the graph is very large. Once the task graph is on the scheduler side, it populates its internal data structures to be able to analyze it and schedule it efficiently on the workers.

3.2.2.4 Task states

This section is important to understand the contributions proposed in Chapter 6.

In the Dask scheduler, a task can be in one of these six states: `"released", "waiting", "no-worker", "processing", "erred", "memory"`[5]:

- `released`: the task is known but not actively computing or in memory. Usually, a task is created in the `releard` state,
- `waiting`: the task is waiting on dependencies to be computed,
- `no-worker`: the task is ready to be computed, but no appropriate worker exists (for example, because of resource restrictions or because no worker is connected at all),
- `processing`: all task dependencies are available, and the task is assigned to a worker for computing (the scheduler doesn't know whether it is in a worker's queue or actively being computed),
- `memory`: the task is available in the memory of one or more workers,
- `erred`: Task computation, or one of its dependencies, has encountered an error
- `forgotten`: the task is no longer needed by any client or dependent task, so it disappears from the scheduler as well. As soon as a task reaches this state, it is immediately dereferenced from the scheduler.

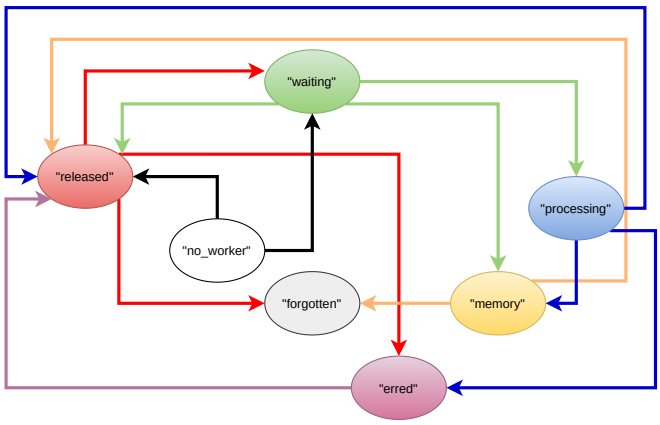


Figure 3.5: Dask task states and transitions. Figure reconstructed from the code in [41].

A task can move from one to another following stimulus coming either from a client or a worker. Figure 3.5 shows the different states and transitions in Dask. This figure has been reconstructed from the code in [41].

And here, we will detail some transition examples. When a task is created, it is in the `"released"` state. It passes to the `"waiting"` state that indicates that this task is waiting for dependencies to be available in memory or it is an entry task (a task without dependencies). If a task is an entry task, and if there are connected workers, it passes to the `"processing"` state when it is assigned to a worker. A task passes from the `"released"` to the `"erred"` state if a task on whom it depends erred and to the

[5] https://distributed.dask.org/en/stable/scheduling-state.html

`"forgotten"` state if it is not needed anymore by an alive client or dependent task. When the *"processing"* state is completed, it passes to the `"memory"` state and unlocks all dependent tasks.

3.2.2.5 Pure Data Tasks

As already mentioned, usually, the source of the data in Dask is a storage system. However, it is also possible for a client to send data to connected workers, either by passing by the scheduler or not. This can be done using the `scatter` method in the client API. It takes as a mandatory parameter: the data that needs to be sent and other optional parameters, such as a boolean that expresses whether or not to pass by the scheduler or a list of workers to whom the data will be sent. `scatter` returns a `future` to that sent data. The key of this `future` is the key of the equivalent *pure data* task in the Dask scheduler. That is, the data sent via a `scatter` is also considered a task with the specificity of only containing data (without a function to be called).

3.2.3 Scheduler Internal State

The scheduler keeps track of the tasks, alive clients and connected workers in its internal data structures that consist of four main objects: the `SchedulerState`, `TaskState`, `ClientState` and `WorkerState`. The `SchedulerState` object contains a global view of the internal state and uses the different other classes. The `TaskState` keeps the state of each task (key, state, dependencies, dependents, ...), `ClientState` and `WorkerState` keep the state of a client and a worker, respectively. Figure 3.6 shows the four main classes in the Dask scheduler internal state and the relevant attributes to this work.

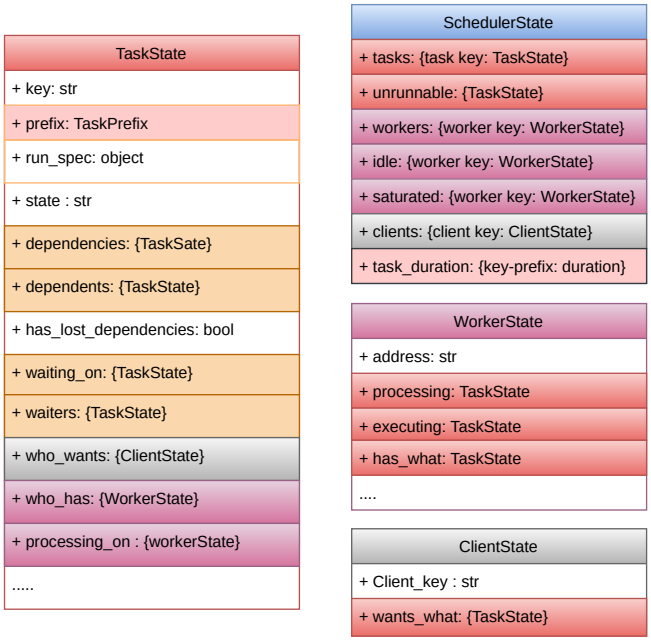


Figure 3.6: Dask internal classes. Figure reconstructed from the code in [41].

We will not detail all the attributes of those classes, but understanding how they are related is important. For instance, in the `ClientState` class, the `wants_what` attribute maps a client to a task from which it expects a result. The scheduler also keeps track of the tasks assigned to workers through the `WorkerState` class. The `"has_what"` attributed, for instance, refers to task results that are in the memory of a given worker.

The `TaskState` class is the most used in the scheduling as it keeps for each task: its dependencies (`"dependencies"`), tasks it depends on (`"dependents"`), clients that want its results (`"who_wants"`), workers that have its result in memory is it is processed (`"who_has"`), and the worker it is processing on if it is currently running (`"processing_on"`). The `"waiting_on"` is a subset of the `"dependencies"` attribute (equals at the beginning), from which we remove computed tasks. When this list is empty, and all the dependencies are completed correctly, this task passes from `"waiting"` to the *"processing"*. The `"waiters"` keeps track of the tasks that still need this task's results. When it is empty, and no client wants it, then its results can be removed from the Dask memory.

3.2.4 Scheduling in Dask Distributed

This section does not discuss Dask scheduling policies in Dask but the `transition` algorithm, which performs state transition. A task state transition occurs from stimuli. A stimulus is a state-changing message from a worker or a client to the scheduler. The scheduler handles those messages by triggering a `transition` function. Every state transition is implemented as a separate method in the scheduler. For

instance, `transition_processing_memory` is the name of the function that performs the transition from the *"processing"* to the `"memory"` state.

When a transition function is called, it mainly changes the state of the given task and constructs a dictionary of recommendations for the state transitions of other tasks, usually the depending tasks. In addition to the specific transition functions, the `transitions` method is called. As its name indicates, it triggers several transitions by iterating over the recommendations returned by the transition method.

For example, when the scheduler receives the `"task-finished"` stimuli, the `transition_processing_memory` is called. It switches the task state from `"processing"` to `"memory"` and recommends all dependent tasks to switch to the `"processing"` state. These recommendations are passed to the `transitions` function that iterates over until no more recommendation is added.

A set of messages to the clients and the workers is also constructed and sent. For instance, those may contain results needed by a client or a task to be run by a worker.

3.2.5 Memory Management in Dask

Dask keeps track of all the data that are available in the workers, distributed memory. Dask has an experimental component that optimizes the memory usage of workers across, which is called Active Memory Manager[6]. We did not explore this functionality in Dask. However, we had to take into consideration the operation of the Dask's included garbage collector. If data is not needed anymore by any client and does not appear anymore as a dependency of any other task, then it can be deleted. To do so, Dask checks its internal data structures and updates them whenever a client or worker sends updates on it. For instance, if the scheduler does not receive heartbeats from a client that submitted a task graph, and those tasks are not used in other computations of other clients, the scheduler cancels them. And the garbage collector deletes the data related to those tasks from workers' memory.

3.2.6 Communications in Dask

All information about where data lives in the distributed memory is kept in the scheduler's data structures. If data is needed by a worker different from the one owning it, communication between the two workers is initiated. But keep in mind that Dask scheduling policies try to schedule tasks in workers that minimize data copies. And worker-to-worker communications are hidden from the user's point of view, which is advantageous compared to message-passing explicit communications.

Communication between the clients is different. They are initiated by the end-user explicitly in the code. Dask provides several ways to ensure communication between clients. We focus on the coordination primitives used in our work, namely `Queues`, `Variables` and `Events`[7]:

- `Queue`: Dask queues follow the API for the standard Python `Queue`, but now move futures or small messages between clients. Queues send small pieces of information that are `msgpack` encodable (ints, strings, bools, lists, dictionaries, etc.). They can be used to send small metadata, and they are not adapted for sending large datasets because they are mediated by the scheduler.
- `Variable`: are like `Queues` in that they communicate futures and small data between clients. However, variables hold only a single value. The value can be `get` or `set` at any time. Variables are interesting in sharing small configurations or parameters between clients. They are also stored in the scheduler, so it is recommended to share only small data through variables.
- `Event`: hold a single flag which can be set or cleared. Clients can wait until the event flag is set. All clients can set or clear the flag, and there is no "ownership" of an event. They can be used to synchronize multiple clients.

[6] https://distributed.dask.org/en/stable/active_memory_manager.html
[7] https://docs.dask.org/en/stable/futures.html#queues

3.3 PDI Data Interface

In this section, we present the PDI data interface, its architecture and it is used to handle our simulation data.

3.3.1 Overview

PDI [148] data interface is a lightweight library for data handling. It offers a declarative API to expose the data of the simulation without specifying what to do with it. It proposes a way to call external libraries from a configuration file. PDI is built around three core concepts:

- data store: when data is shared from the simulation, it is made available to PDI through the data store,
- event subsystem: once the data is available in the data store, the event system notifies the data handler plugin,
- plugins: they access the data available in the store and process it.

The data layout, the orchestration of the exchanges, and the configuration of the different plugins are described in the specification tree; Figure 3.7 (in Page 45) shows a structure scheme of PDI [18].

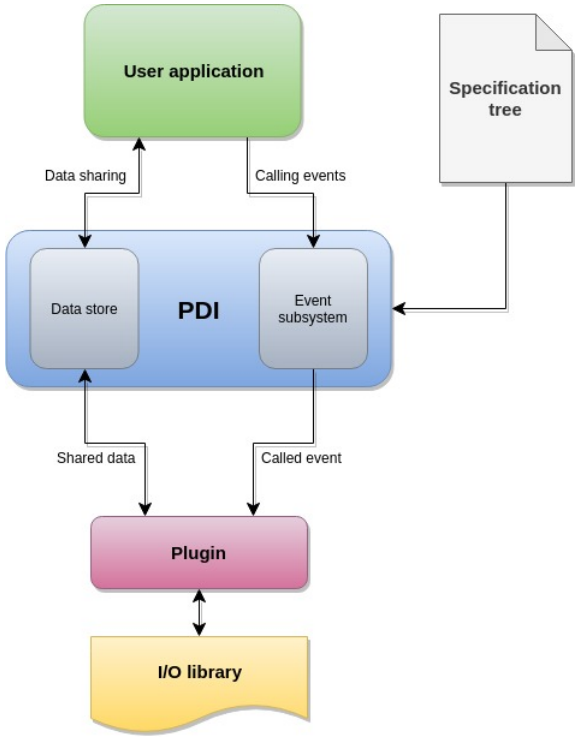


Figure 3.7: PDI architecture. Figure from PDI documentation [3].

3.3.2 PDI API and Simulation Instrumentation

PDI offers a very simple API; its functions can be grouped into three categories. The initialization and finalization functions, respectively `PDI_init` and `PDI_finalize`, are used to set up and finalize PDI by releasing its resources. The second category consists of a set of functions used to annotate the code. `PDI_share`, `PDI_reclaim` are used to respectively share data with PDI to be used by external plugins and reclaim it back by the simulation at the end of the plugin operation. `PDI_expose` does both sharing and reclaiming the data from the data store. *PDI_event* triggers a PDI event and finally, `PDI_multi_expose` exposes several variables and triggers an event. Listing 3.5 shows an example of a c code instrumentation with PDI, PDI API calls are highlighted.

```
1  int main( int argc, char* argv[] ) {
2    MPI_Init(&argc, &argv);
3    PDI_init(PC_parse_path("pdi_spec.yml"));
4    int rank; PDI_Comm_rank(MPI_COMM_WORLD, &rank);
5    config_t cfg = read_config("simulation.yml");
6    // share one-off configuration
7    PDI_multi_expose("init",
8        "cfg",  &cfg,  PDI_OUT,
9        "rank", &rank, PDI_OUT,
10       NULL);
11   // our temperature field
12   double* temp = malloc(sizeof(double) * cfg.loc[0] * cfg.loc[1]);
13   initialize(temp);
14   // main loop
15   for (int step=0; ii<nb_steps; ++step) {
16     do_compute(temp, MPI_COMM_WORLD);
17     // share data at every iteration
18     PDI_multi_expose("iter",
19         "step", &step, PDI_OUT,
20         "temp", temp,  PDI_OUT,
21         NULL);
22     MPI_Barrier(MPI_COMM_WORLD);
23   }
24   free(temp);
25   PDI_finalize();
26   MPI_Finalize();
27 }
```

Listing 3.5: PDI instrumentation of the C simulation code.

3.3.3 PDI Specification Tree

As mentioned in Section 3.3.1, the specification tree describes the data layout, orchestrates interactions between the code and PDI, and contains the plugin's configurations. It is specified in a `yaml` file and is provided to PDI at the initialization. For instance, in Listing 3.5 line3 the configuration file name is `"pdi_spec.yml"`.

The specification tree contains three main sections: the `metadata`, `data` and `plugins` sections. The `metadata` section contains small variables for which PDI keeps a copy that can, for example, include the sizes of a given array. The `data` section contains the data layout description. It defines the types of data expected in the store. Those data are not copied by PDI; only pointers to the data are shared. Finally, the `plugins` section lists the plugins that will be loaded and their configurations.

Listing 3.6 shows an example of a PDI `yaml` configuration file where a `types` section is added, where we can define new types such as `structures`. In this example, we find the `metadata` section in line 2. The `data` section starts in line 3. It provides a description of the `temp` field, (line 4) including its type, subtype and size (respectively in lines 5, 6, 7). Finally, the `plugin` section starts at line 8, loading one plugin: the `decl_hdf5` plugin in line 9. The PDI plugin system is detailed in the following.

```
1  types: #[...] including config_t description
2  metadata: {step: int, cfg: config_t, rank: int}
3  data:
4    temp: # the main temperature field
5      type: array
6      subtype: double
7      size: [ '$cfg.loc[0]', '$cfg.loc[1]' ]
8  plugins:
9    decl_hdf5:
10   - file: data-$step-$rank.h5
11     write:
12       temp:
13         when: '$step>0'
```

Listing 3.6: Data description in PDI YAML file.

3.3.4 PDI Plugins

PDI supports loose coupling of simulation codes with external libraries. Those libraries are supported in PDI as plugins and are configured through the specification tree. PDI offers a list of built-in plugins ranging from IO-specific ones to more generic data handling tools. It also allows the creation of user-specific plugins if the built-in ones are not enough.

The built-in plugins PDI provide can be grouped into four categories:

- general purpose: include `mpi`, `trace`, `set_value`, `pycall`, `user_code`, `serialize`,
- IO: include `decl'hdf5`, `decl'NetCDF`, `SIONlib` plugins,
- fault tolerance: `FTI`plugin.

In Listing 3.6, one plugin have been used the `decl_hdf5` plugin in line 9. In this example, a file named *data-step-rank.h5* is written by each process in every step greater than 0.

PDI is extensible, and the user can add new plugins. For instance, `FlowVR`, `Melissa`, and `Sensei` plugins have been added. `FlowVR` and `Sensei` have been already presented in Section 2.2.1, and `Melissa`[160] is an in situ framework for sensibility analysis.

3.4 Summary

In this chapter, we have presented the tools we chose to use in our work, namely Dask distributed and PDI data interface. Dask distributed has been selected for its ease of use thanks to the high-level parallel libraries it supports. We have prioritized the practicality as our goal is to bring together in situ performance and post hoc ease of use. We have decided to keep a good separation of concerns while handling the data to maintain good habits while keeping a simple declarative interface. Thus we have opted for PDI data handling tool to extract data from the simulation and PDI plugins to process it.

Part II

Contributions

Chapter 4

Approach: A Bridging Model Between MPI and Dask Distributed

Nothing in life is to be feared, it is only to be understood. Now it is the time to understand more, so that we may fear less.

Marie Curie

In this chapter, we present our approach to bringing together in situ performance and post hoc ease of use. We consider a producer-consumer scheme, where the producer is an MPI simulation, and the consumer is a Dask distributed analytics code. Our approach consists of proposing a bridging model between MPI and Dask distributed that hides code coupling complexity and all underlying differences between the two models. Then we propose an implementation using MPI, PDI and its plugins and Dask.

4.1 Overview

The in situ paradigm is an interesting alternative to post hoc processing. However, it sometimes becomes less relevant due to its setup complexity. The previous chapter shows that most in situ tools are built on the MPI model. While MPI is one of the most used programming models for scientific applications, it is not the most suited for data analytics algorithms for several reasons. Mainly the structure of those algorithms is different from the simulations' structure. Moreover, the life span of the analytics algorithms is usually shorter than simulations, as they are used to explore and understand parts of the simulations themselves. One would rather prefer a simpler prototyping model to write them instead of spending a long time optimizing an algorithm that may become useless in a short period.

To be able to explore the in situ paradigm with reduced complexity, we have mainly two options: build both the simulation and the in situ tool with a higher-level programming model that makes both the simulation and the analytics easier to design; or keep the simulation built on MPI and choose a programming model which is more adapted for analytics. The first possibility would require rewriting simulation codes using another programming model, likely higher-level and slower, which may conflict with the high performance we are looking for. In addition, our goal is to simplify the in situ analytics workflow setup, not to change the programming model used for simulations.

The second possibility is more attractive as it keeps using the MPI programming model for the simulations and considers a better-suited model for the analysis, such as the task-based programming shown earlier. One may opt for Python-based tools to smooth the transition from sequential post hoc Python code to in situ. Moreover, designing a solution that keeps the data handling well separated from the simulation concerns aims for a clean solution where switching between in situ and post hoc is almost or completely transparent in the simulation code. Such an approach takes advantage of all available data analytics libraries, which avoids recreating the stack from scratch.

In this category, we can consider two other possibilities: either considering intermediate storage to host the data generated by the simulation before it is consumed by the analytics or directly coupling the two models. The first possibility would be similar to DataSpaces-based and SmartSim solutions in the way they handle data (e.g. using an in-memory data store): the simulation and the analytics are completely decoupled, and the data is stored in a separate distributed shared memory. Both applications can read from and write to the data store. This approach is relevant as long as there is no added complexity to managing the distributed data structures or writing distributed algorithms. The second possibility is more attractive as it avoids the extra communications to/from the staging nodes.

In our work, we have chosen this last option. Our goal is to provide an in situ solution where simulations are parallelized in MPI, and analytics are parallelized using distributed task-based programming without saving data in intermediate storage. Hence, our work can be seen as a code coupling problem, where we have two codes written in two different programming models, and we want to make them communicate in a producer/consumer fashion, where the simulation is the producer, and the analytics is the consumer. In this chapter, we propose a bridging model between MPI and Dask distributed task-based programming systems. Among other data analytics frameworks, we have opted for Dask because it offers distributed APIs for well-known Python libraries such as `numpy`, `pandas` and `scikit-learn`. Hence, a post hoc sequential Python code is easily ported to in situ distributed Dask code thanks to the bridging model.

But before getting into the details about the bridging model, we summarize the most challenging points of this problem.

4.2 Challenges

The BSP and the Task-based paradigms differ not only in terms of abstraction levels, development effort, and performance but also in defining key concepts such as parallelism and the data and how they are managed. To concertize the challenges, we will use MPI and Dask terminology, but most of them are related to the paradigms rather than the implementations.

First of all, the type of parallelism: the application is represented as a set of P collaborating processes for a common job from the beginning until the end of the program. The task-based model parallelism is expressed in tasks interrelated via dependencies to form a task graph; each task performs a small part of the work. While the user is responsible for managing the processes: communications, and synchronizations in MPI, a runtime ensures that in task-based frameworks. We talk about explicit parallelism in MPI and implicit one in the task-based programming model.

Secondly, the data semantics and representation in those models are different too. Each process has its local memory and buffers. As defined in MPI, the data are manipulated through buffers that keep the same name during execution and whose values are updated as the simulation progresses. Hence the data in MPI can be defined as the value of a given variable at a specific moment. In the Dask model, data is defined as immutable inputs and/or outputs of a given task. While the time dimension is important to identify data in MPI, it does not have a similar signification in a task-based model. The third big difference is related to the view of the execution we may have about an application in both cases. While a task graph describes all the tasks that will be run in a task-based model, it is complex and sometimes impossible for MPI applications to have such a deterministic view from the beginning. For instance, the time to reach a stable state is not known at the beginning of the simulation. And this makes the coupling more challenging, as we do not have any a priori idea about what will be executed at runtime, the data that will be generated, and when it could be extracted.

Those conceptual differences make communicating two applications coming from those paradigms challenging. We propose a concrete bridging model between MPI and the Dask paradigms to reduce this complexity.

4.3 DEISA Model: A Bridging Model Between MPI and Dask Distributed

In the scope of this work, we only consider a bridging model that couples BSP and distributed task-based models in a producer/consumer scheme where the producer is parallelized using the MPI model and the consumer is parallelized using Dask distributed. In the next section, we define the terminology we use in this model and then resume the assumptions for a working bridging model.

4.3.1 Terminology

4.3.1.1 DEISA Components

A DEISA component is a running distributed application: the union of a code and all its necessary resources (processing and memory units). The components live in a distributed environment and manipulate distributed data structures.

The producer component in our work corresponds to a running simulation parallelized with MPI. The task-based component is a complete Dask distributed cluster running an analytics job. Unlike the simulation, where we only deal with the MPI processes, we deal with three entities in the task-based component: the analytics client, the centralized scheduler and distributed workers.

In this chapter, we will make an abstraction of the concept of DEISA tasks and DEISA task graphs for simplification; we will need it back in Chapter 5.

4.3.1.2 Internal and External Events

An internal event happens internally to a component and does not have any impact on the environment or other components. An external event can be described as an event emitted/received by a component to/from another component. For instance, a remote procedure call is an external event that is initiated by a component that will trigger an action in another component. An external event can be any operation, function or communication that is initiated by a component and is observable in another component.

4.3.1.3 DEISA Tasks

A task is defined as all the component's internal operations and events that are delimited by two external events. The smallest task that can be defined in an MPI process is the union of all computations that is delimited by two communications. The inputs of this task are all needed data to perform the computation. Its output is the resulting data structures. In MPI, a DEISA task can be defined as all the computations and internal (to the MPI process) communication delimited by two external communications.

In Dask, we distinguish the internal (to the component) input/output data from the external ones (where the source/destination) is another component. Hence, a DEISA task can be seen here as a Dask subgraph that is delimited by two external inputs/outputs.

Coupling two components results in a task graph. Figure 4.1 shows an example of two coupled components represented by a set of tasks. Each task has either an external input (tasks in the task-based

component) or an external output (in the MPI component). We distinguish the internal dependencies and the external ones as well with different colours.

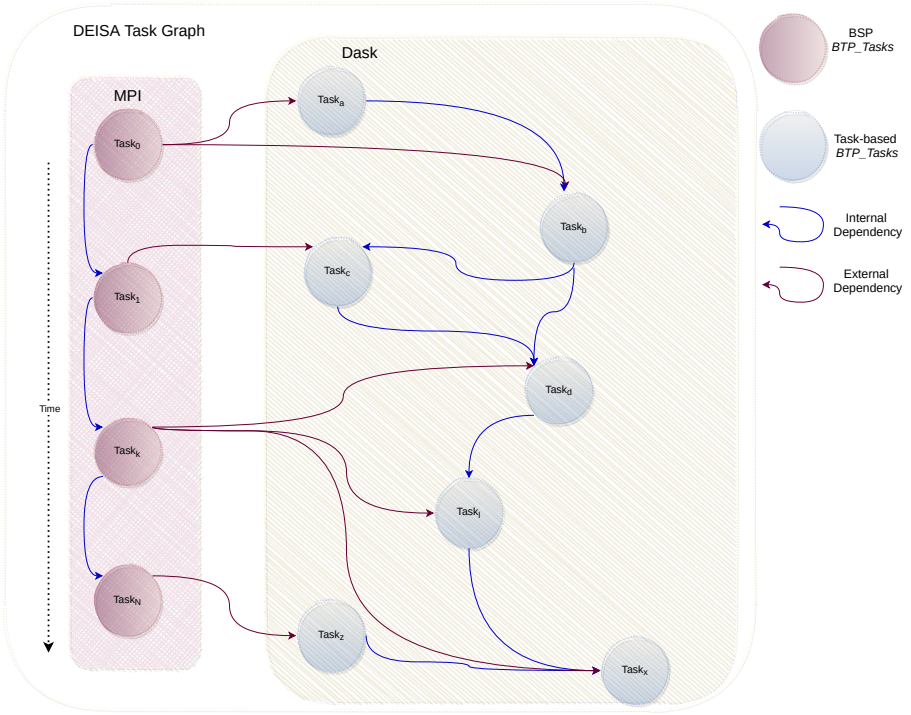


Figure 4.1: DEISA task graph of two coupled components: an MPI component in the left and a Dask component in the right. Note that all the tasks have either an external output on input.

4.3.1.4 DEISA Delivery Facility

In addition to ensuring the establishment of connections between components, the *Delivery Facility* (DF) ensures a global identification and redistribution of data between coupled components. It identifies a piece of data internally in a component and translates its identifier to a global ID understandable in other components. This step is essential due to the different ways data is defined in the two paradigms. The DF also implements data redistribution schemes, as the components deal with distributed data structures that are not necessarily distributed in the same way.

4.3.2 Full Producer/Consumer Example

We only show data communication between two DEISA components in this example to simplify the workflow. Figure 4.2 shows an example coupling $A^{R_n}_{MPI}$ and $B^{R_m}_{Dask}$ where A is a producer and B is a consumer. The MPI component has R_n resources. Each $Task_k$ is scheduled explicitly to a set of resources R_k. In scientific applications, we usually have an iterative code. Each task generates a block of data at a given moment t (only one task is shown in the figure, with a loop mark). Those blocks of the data (small blue boxes $D_{i,j}$) are shared and sent to the *Delivery Facility*. Component B is the consumer of the data. It is a task-based component that has R_m resources that are managed implicitly by a runtime (blue boxes with grey hachures). A task graph is represented as a DEISA task graph (yellow graph), with dependencies on external inputs. Those inputs (in red) are data with new keys (IDs) that are easily recognizable, thus usable in B.

The data is sent through the network between components. The *DFs* ensure the connection to a distant component, identifying and redistributing data between components. In this figure, the data identification is made in two steps from each side. The small blue boxes $D_{i,j}$ are identified by three elements: D is the name of the data, i for instance, the position of this block of data in the global distribution, and the j corresponds to the timestep or the iteration. These keys can be considered as being local to the component A. The same component has created a new key: $I_{l,m}$. It is a global key recognizable in the DF of B. In the component B, those keys are translated to new keys internally

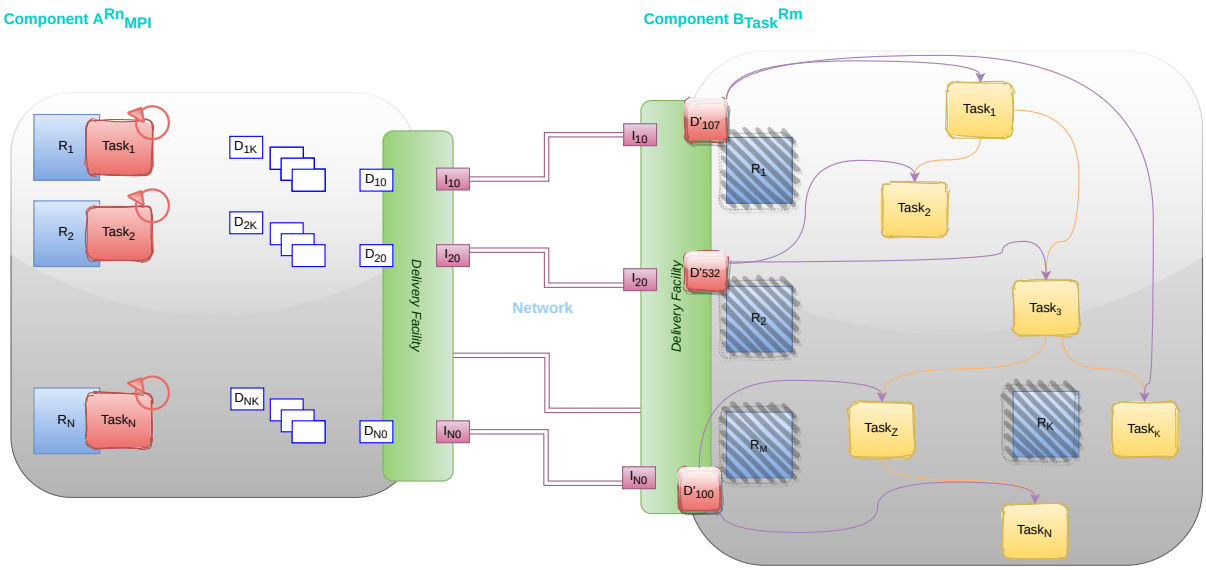


Figure 4.2: Producer Consumer Example.

understandable D'_k. This identification and translation process is mandatory but can be done in fewer steps. For example, if $I_{l,m}$ is recognizable by B, then there is no need for further translation at reception.

4.4 DEISA Bridging Model Implementation

Several choices have been made regarding the implementation of the bridging model. We have focused on three main goals: performance and separation of concerns on the simulation side and productivity on the analytics side. Those goals have guided all our choices to propose a solution that responds the most to our problematic: bringing together the performance of in situ and the productivity of post hoc workflows.

For that, we have opted for MPI for its success and popularity in the HPC community; the huge majority of legacy and new production codes are parallelized in MPI+X. We have opted for PDI data interface to handle data and implemented the *Data Facility* on top of the pycall plugin in Chapter 5 and then through a new PDI plugin named DEISA in Chapter 6. Moreover, PDI allows switching between different plugins easily, thus switching between in situ and post hoc modes if needed. This is crucial to support heterogeneous workflows to be able to keep analytics results along with raw data in case further analytics are needed. Finally, Dask distributed is used as a task-based framework and an *Adaptor* to implement the *Delivery facility* from the analytics component side.

4.4.1 External Events

We have respected the information hiding and separation of concerns modularity principles. All internal events are hidden at this point, and the components may share data only through external events. In the MPI simulation, we have introduced external events thanks to PDI data interface that allows sharing of internal data for external use. Once the data is shared via PDI, it is handled by our dedicated plugin.

In Dask, an external event is usually input data coming from the simulation and integrated into the task graph. We will detail that in the next section.

4.4.2 MPI Simulation Delivery Facility: Bridge

The delivery facility is the most important entity, as it ensures the real coupling and communication between the different components. The delivery facility in the simulation component is called the *Bridge*, It is built on a lightweight Dask client that is only used to send data to the workers and communicates metadata with the analytics client. It is implemented as a Python class.

We have implemented a PDI plugin that creates a bridge in each MPI process. Thus, the delivery on the MPI side can be seen as a set of distributed lightweight Dask clients. They ensure the connection to the Dask scheduler (so the task-based component). They also ensure the identification (Section 4.4.6.2.3)

and communication of the data. We have opted to use `scatter` to send data from the bridges to the Dask workers. To communicate metadata, we have used Dask `Queues` and `Variables`.

4.4.3 Dask Analytics Delivery Facility: Adaptor

The delivery facility in the Dask component is called the *Adapter*. It is associated with the main Dask client. It ensures communication with the DEISA bridges through the scheduler via `Queue`s and `Variable`s Sending the data to the workers is not enough; metadata has to be sent to the adaptor to keep the semantics of the data blocks sent independently and use them in meaningful analysis.

4.4.4 Data and Control Communication

To ensure the coupling of MPI with Dask, both data and control need to be exchanged. Several communication schemes can be considered. For instance, in a producer/consumer scheme, data can be pushed by the producer into the consumer's memory, or the consumer can pull the data from the producer. A staging memory can be deployed to host generated data before the consumer retrieves it. This works for the control as well. In this work, we wanted a direct coupling between the MPI and Dask to avoid eventual extra communications between the staging nodes and the two components. Moreover, we wanted to take advantage of Dask memory management system that takes care of optimizing data movement and deletes non-needed data. A pull approach could be interesting and safer, as the consumer will control when it receives the data. However, it may penalize the simulation if the analytics are longer. We have opted for pushing data rather than pulling it from Dask, as it is simpler to identify the end of an MPI task (through an external event). We suppose that the user puts enough memory in Dask at least to host one timestep.

In this work, the delivery facilities ensure the data and control communications.

4.4.4.1 Control Communication and Synchronization

Control in in situ coupling can be defined as events that are exchanged between data producers and consumers to trigger actions. We have used that to coordinate several operations. In this work, all control goes through the Dask scheduler. The scheduler's memory can be seen as a staging area between the two components. We have used available data structures in Dask to communicate control, namely `Queues` and `Variables`.

Dask `Queues` are similar to Python ones, where data is stored in a First In First Out (FIFO) fashion. The main difference is that in Dask, the `Queue`s can be accessed by multi-producers and multi-consumers Dask clients. We have used the `Queue`s to communicate metadata that have to be accessed only once and only by one consumer. A typical use is to store the metadata associated with the data generated by an MPI process.

Unlike the `Queue`s, the data stored in a Dask `Variable` can be accessed and modified by several clients at the same time. The way they are used in our implementation avoids any possible race conditions that may occur; we have used them only for immutable data, and we have only one writer and several readers.

We have used `Variable`s to share metadata that have to be accessed by all the bridges. The adaptor is the only writer in the `Variable`s. For instance, the list of connected workers is stored in one shared `Variable`.

The `Variables` have an interesting property: they block the readers while not initialized. We use this property to synchronize the simulation and Dask. For instance, the analytics client waits actively until all workers connect; we assume that the analytics client already knows the number of needed workers. Once those workers get connected, it puts a list of all the workers' IP addresses in a variable called `Workers`. The bridges (MPI processes) are blocked until this variable is initialized, waiting to get the list of workers.

The way we implement the synchronization is not different from making all bridges wait until all workers are connected and get the list from the scheduler directly. Both solutions imply requesting data from the scheduler, which may become a bottleneck when increasing the number of MPI processes. This is because both pieces of information come from the scheduler. Trying to get the list of connected workers or the content of a variable implies requests to the same entity, which is the scheduler. However, the implementation of the bridges may be changed, and communications between those and the adaptor may be improved to bypass the scheduler. And at that moment, our current implementation will make a lot of sense.

4.4.4.2 Data Communication

Data communication is the most important part of the in situ paradigm. The solution is built around how data is communicated between the producer and the consumer. We have opted for a direct coupling between MPI simulations and Dask in an in transit configuration where we separate the two components' resources. We have chosen to push generated data by a simulation process to Dask workers' memory and the associated metadata to the analytics client, through the bridges (Section4.4.2) and the adaptor (Section4.4.3).

The bridges are built on lightweight Dask clients. We have used the `scatter` operation available in Dask client class to send the pieces of data from the MPI processes (through the bridges) to the Dask workers. The metadata are sent through the Dask `Queue` and `Variable` mechanisms.

4.4.5 Control Flow

In a producer/consumer architecture, data flow should be controlled correctly. For instance, if the producer is faster than the consumer, the system should block the producer until there is enough memory in the consumer. Other solutions may propose having a staging area to store the data if the producer is full or just writing the data to disk. In this work, we have opted for the trivial solution. The simulation is blocked while Dask is still processing data from previous timesteps.

We use Dask distributed `Queue` to manage the data flow thanks to the `Queue.size` attribute. We can set the size of the `Queue` according to the total memory size of the workers and the size of the data generated per timestep. This solution is effective if metadata are submitted at each timestep.

4.4.6 Data Model

Data is one of the core concepts of our work; its definition, identification, representation, and communication are as important as its processing. To smooth differences between MPI and Dask, the delivery facilities from both sides, being aware of the semantics of data, add a layer to make it understandable to other components. We detail in the following sections how we managed the data in the DEISA bridging model regardless of all the differences in data semantics and definitions between MPI Dask.

4.4.6.1 Data Semantics

In order to couple a MPI component and a task-based one, we have to map data generated by the first one to the task graph managed by the second, while preserving their semantics. As DEISA components hold distributed data structures and exchange blocks rather than the whole data, the semantics may easily be lost while communicating. A possible way to preserve the semantics of the data is by uniquely identifying the blocks generated by the simulation (as defined in section4.4.6.1.1) and mapping them as inputs to the task graph as a pure data task (as defined in section3.2.2.5). In this section, we provide a definition of the data in each model.

4.4.6.1.1 Definition: In the MPI model, a data can be defined as a **value** of a buffer at a given timestep. In MPI, the name of this buffer, an MPI communicator, the rank of the process, and the timestep enable to identify a data in the global data structure generated by the simulation.

4.4.6.1.2 Definition: Data in a task-based model can be defined as an **input** of one or many tasks. Data may be ordinary objects, files, or pure data arrays from external resources. Here, we are mainly interested in simulation-generated data provided as inputs to the task-based component. In Dask, our solution is built on the pure data tasks (as defined in section 3.2.2.5), which are nothing else than pure data arrays.

4.4.6.2 Data Representation and Identification

A DEISA component can manage distributed data structures and process them. Distributed data structures are builtin concepts in several tools nowadays; high-level APIs already exist to manage those such as `dask.array`, `dask.dataframe` in Dask, `regions` in Pygion and *Ds-array* in PyCOMPSs. The common characteristic of those data structures is that they are virtual; they may be distributed over several nodes, and the whole data structure may not exist simultaneously in the memory. Each array represents a coherent multidimensional field. An array is split into blocks. Each block should be identified in a way

that allows a receiver component to know the component it comes from, the field it belongs to and its position in the spatio-temporal decomposition.

4.4.6.2.1 Data Representation One of our motivations for using Dask, was the relevant high-level APIs it offers. The one that corresponds the most to our need is the `dask.arrays` API. A `dask.arrays` is a virtual representation of a multidimensional array broken down into several blocks called chunks. It can be used, for instance, to read an HDF5 dataset in post hoc case. We have chosen to provide a `dask.array` per generated data field. This representation can preserve decomposition information of out-of-memory problems through chunks definition. Note that the user can jump from `dask.array` to `dask.dataframe` easily, for instance, when they want to use specific methods from `Pandas` API (`dask.dataframes`).

While we had a simple choice for the Dask side, we have two equivalent data representations for the simulation: either keeping a basic representation, where we associate each generated block with a set of metadata, and we push them to Dask; or proposing a virtual data structure, similar to `dask.arrays` to represent the global view of the generated blocks. For both possibilities, all the changes will be added in the configuration rather than in the simulation. So the user will not need to change the way he implements the simulation.

We have provided both implementations of data representations; technical details are discussed in Chapter 5.

4.4.6.2.2 Data Definition When a data is shared through PDI, the plugin code only gets a pointer to that data.The bridge is built on PDI with Python support and uses `pybind11` to expose C++ types to Python. When data is shared with PDI and has to be handled by the bridge, a pointer to that data is passed to PDI and transformed into a `numpy` array by `pybind11` then passed to the bridge alongside the metadata allowing the identification of this array.

On Dask side, we define our data as `dask.arrays`. We use the metadata we get in the adaptor to reconstruct the global view of the domain decomposition. Each block in `dask.array` corresponds to the `numpy` array generated per an MPI process in the simulation and sent to Dask.

4.4.6.2.3 Data Identification We uniquely identify each value of a given buffer produced in the MPI model by associating a unique *key* to a buffer's value at a given timestep and using this same *keys* for the corresponding `dask.arrays` in Dask. We have two main possibilities here: to use Dask key generation system or to develop our key generation algorithm. Both solutions have pros and cons, and opting for one or the other changes the design of our coupling system. Using the Dask key generation system gives rise to a multi-graph implementation of DEISA, and the second gives rise to a single-graph one.

Dask Key Management This solution is based on Dask *key* generation system. Once data is generated by an MPI process, the `scatter` method is used to send it to the Dask workers. The `Future` returned by the `scatter` function is appended to metadata, including the data name and its position in the global decomposition and sent to the scheduler. The *key* returned by `scatter` is unique; thus, the data can be identified in Dask by that unique *key*.

In this solution, one has to wait for the data to be generated before building the analytics task graph using the key identifying it. We call this approach the *multi-graph* approach, as during an execution multiple task graphs are generated, one per analyzed simulation iteration.

Automatic Key Management In the previous version, the keys associated with the data generated by the simulation were generated by Dask. In this version, we provide our key generation algorithm that is common to both components' delivery facilities. Thus, it is possible to know all the keys of the data that the MPI component will generate in advance. Having this information allows us to create the associated `dask.arrays` and submit tasks on them in advance (as soon as we know that the data will be generated and before they are actually generated). We call this approach the *Single-graph* approach, as a single analytics graph is submitted at the start of the simulation execution. This graph contains analysis to perform over all the timesteps. We expect better performance from this approach as it enables the reduction of traffic to the scheduler and further optimization in the global task graph. We have introduced a new concept in Dask to implement this solution: namely external tasks. We will provide more details on that in the next sections.

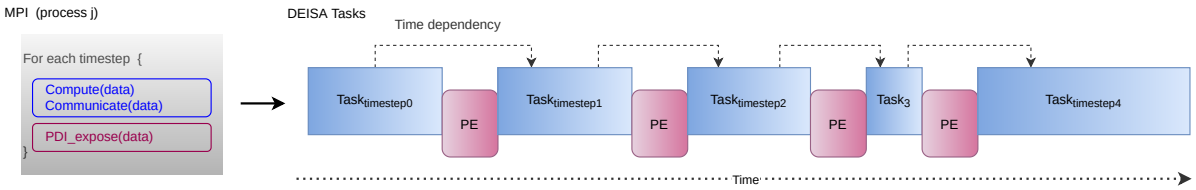


Figure 4.3: Example of DEISA representation for an iterative MPI code.

4.4.7 Porting an MPI Code to DEISA semantics

The MPI code does not need to be rewritten to be DEISA compatible. What needs to be done is only to tag the data we want to share externally and specify when this will be done. A DEISA task, in MPI words, is a set of computations and internal communications delimited by two external events; concretely, it is all the code that is delimited by two calls of `pdi_expose`. The call to `pdi_expose` that is added at the end of each iteration in the loop in fig. 4.3 is enough to construct a list of DEISA *tasks*, one per each time step. It can be seen as an unrolled loop over time.

4.5 Summary

This chapter introduces our approach to couple MPI simulations with Dask distributed analytics in transit as a producer-consumer scheme. It consists of a bridging model between the two different programming paradigms called DEISA. We have introduced three main concepts in this model, namely: components that refer to applications in one of the two models, tasks and external events to unify the terminology between the models and the delivery facilities to perform the data and control communication. We have implemented the bridging model using different tools. In addition to MPI and Dask distributed that implement the components, we use PDI data interface to express the external events in MPI, and implement the data facilities as Python libraries.

Chapter 5

DEISA1: Multi-Graph Implementation of DEISA

Everything is theoretically impossible until it is done

Robert A. Heinlein

This chapter is an extended version of our HiPC21 publication[101], where we have implemented the multi-graph version of the DEISA bridging model. In this version, we have used the Dask key generation system, the available `scatter` method to send data to Dask workers and a task graph is submitted at each time step.

5.1 Architecture

Figure 5.1 shows the architecture of the DEISA prototype. We couple a running simulation represented by $M+1$ MPI processes with a Dask instance comprising a scheduler, an analytics client and N workers. Simulation data is handled with the PDI data interface. It is usually shared at each timestep (or periodically every K timesteps) through the `pdi_expose` function with the DEISA plugin that instantiates one bridge object per MPI process.

Each bridge connects a Dask client to the scheduler. The bridge sends the data to the workers and metadata to the scheduler but does not submit any task graph to the scheduler. At each timestep, the data generated by each MPI process is sent to a pre-selected Dask worker with a round-robin fashion at the initialization step (step 1 in Figure 5.1). Each bridge constructs metadata related to the data block that includes the name of the data, its type and subtype, its size and the timestep. That metadata is sent to the scheduler in a Dask `Queue` associated with the bridge (step 2 in Figure 5.1).

The delivery facility on the Dask side is called a DEISA *Adaptor* or also called metadata adaptor. At each timestep, it requests the metadata from the scheduler (which is/will be available in the queues); uses it to create a `dask.array`. It creates one *dask.array* per block of data received from an MPI process, representing the process-local array. Then it gathers all the blocks in one larger array using the available `dask.array.block` method (step 3 in Figure 5.1), to get one that represents the global distributed array.

The client retrieves this `dask.array`, which is only a descriptor of the actual data that resides in the workers' distributed memory and submits a task graph that processes this `dask.array` (step 4 in Figure 5.1).

This process is done at each timestep, only after the communication of the data to Dask workers: the analytics client submits one task graph per iteration of the simulation.

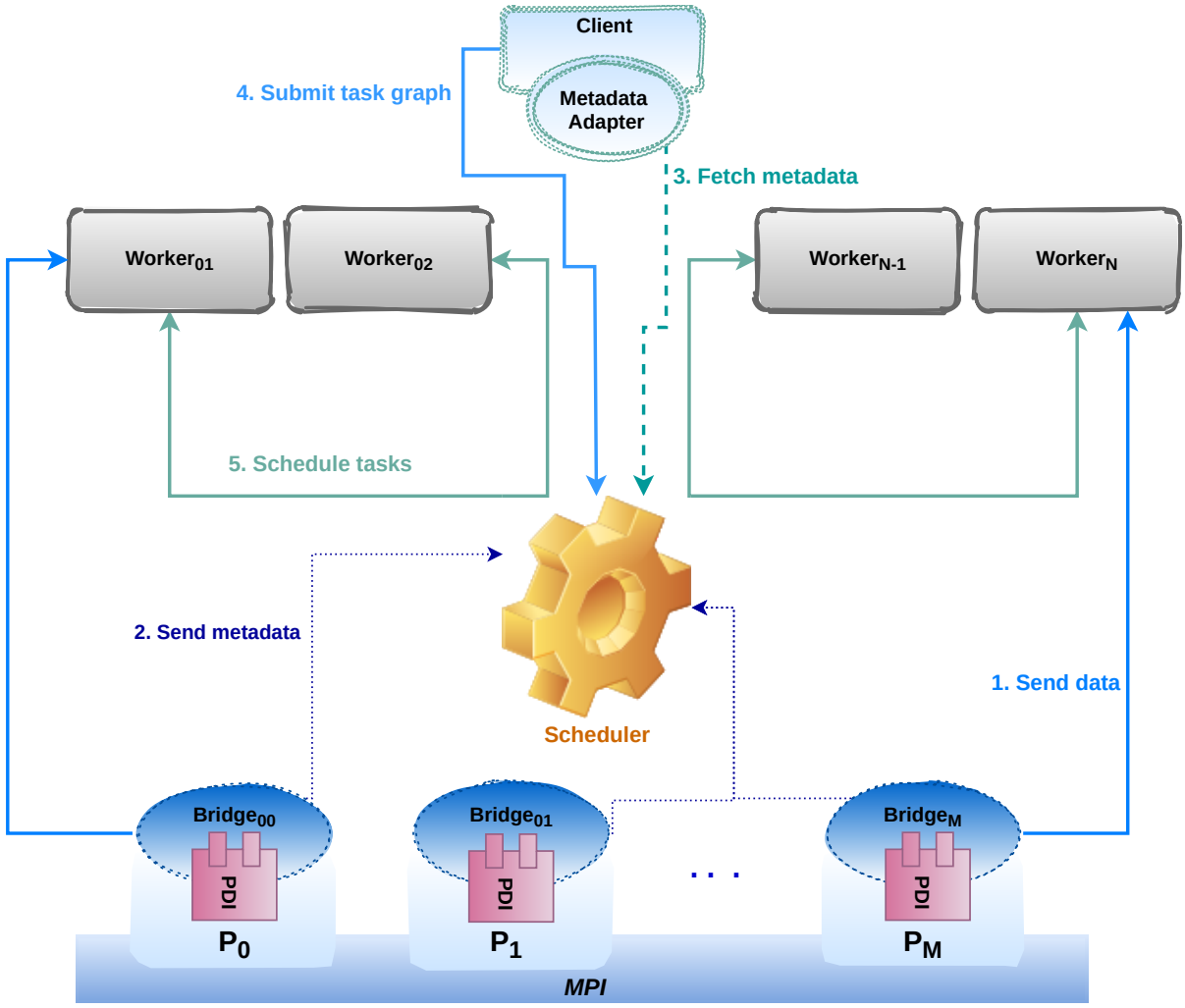


Figure 5.1: DEISA1 architecture.

5.2 Implementation

A deep understanding of Dask was required to ensure the coupled distributed systems' operation. We detail in the next sections the operation of the coupling of the two distributed applications.

5.2.1 Data and Metadata Communication

When data is available in an MPI process, it is shared with PDI via `pdi_expose`. The pointer of that data is transformed into a `numpy` array by `pybind11`, and passed to the DEISA plugin that implements the delivery facility from the simulation side. The array is sent to a pre-selected Dask worker via an available method called `scatter`. `scatter` is a method in the Dask `Client` class. It is meant to send pieces of data from the client to the workers directly or by passing through the scheduler. Here are the most important arguments of the `scatter` method: `scatter(data, workers=None, broadcast=False, direct=None,...)`. It returns a `Future` object (or a structure of *Future*s matching the input data type). In our case, the data is the received `numpy` array available in the bridge. We enable *direct* mode to send data directly to the workers without passing by the scheduler, to avoid slowing down the scheduler and make it possible to transfer data larger than the scheduler memory. We specify the IP address of the worker destination. We get in return a `Future` object, including a unique key associated with this data block, generated by Dask.

The metadata associated with each generated block is gathered in a Python dictionary that includes: the name of the data, its size and type, and its position in the global distribution as a tuple (including the time dimension in position: 0). A dictionary is serializable by `msgpack`; thus it can be sent via a `Queue` to the DEISA adaptor in the analytics client of Dask. However, these metadata are not enough to identify data in Dask. The associated key (generated by Dask) with each block of data is required, so it has to be included in the metadata too. The `Future` returned by the `scatter` is also needed to identify the data.

The are three main possibilities to include the *Future* in the metadata:

- Include the *Future* in the same `Queue` as the other metadata, which is not possible because the *Future*s and the dictionaries are not serializable in the same way. So this solution is not possible with the available serialisers in Dask.

- Extract the key of the *Future* and append it to the metadata `Queue`. The key is enough to identify the data that has been sent to Dask. However, if we only send the key, which is a string, the scheduler will not know that the string included in the *Queue* refers to real data. And if, at that time, the bridge goes out of the scope, and the analytics client did not create a *Future* with the same key yet, then the scheduler may delete that data because it assumes that no client wants it. So this solution does not work.

- Use a different `Queue` to send the *Future* or send it as a separate object in the same `Queue`. This is the most costly solution in terms of communications. However, it ensures a rigorous operation of our coupling system, as sending the *Future* object via a `Queue` prevents the scheduler from deleting associated data with that *Future*.

We have chosen to use one `Queue` per MPI process and send the metadata in two steps. First, construct the metadata dictionary and put it in a `Queue`, and when the `scatter` returns a *Future*, we put it in the same `Queue`. Using one `Queue` per process instead of a unique `Queue` for all the processes avoids any confusion with the metadata sent in two steps that may happen because we don't have any control on the arrival of messages to the `Queue`. We have implemented the DEISA adaptor in the Dask side to be able to deal with that 2-step metadata communication.

5.2.2 Metadata Reception and `dask.array` Creation

At the reception of the metadata in the DEISA adaptor, a `dask.array` is created with the same *key* as the *Future* and is put in a list of lists in the position it is supposed to be in (that we get from the metadata in the dictionary). Once done, we create a global `dask.array` using the `block` method that receives a list of lists of `dask.arrays` and creates a global array with as many blocks as there is the lists are each dimension.

Creating a `dask.array` this way is not the most natural way to go in Dask. Usually, the user gives Dask the sizes of the array in each dimension and the way they want to chunk it, and Dask creates the

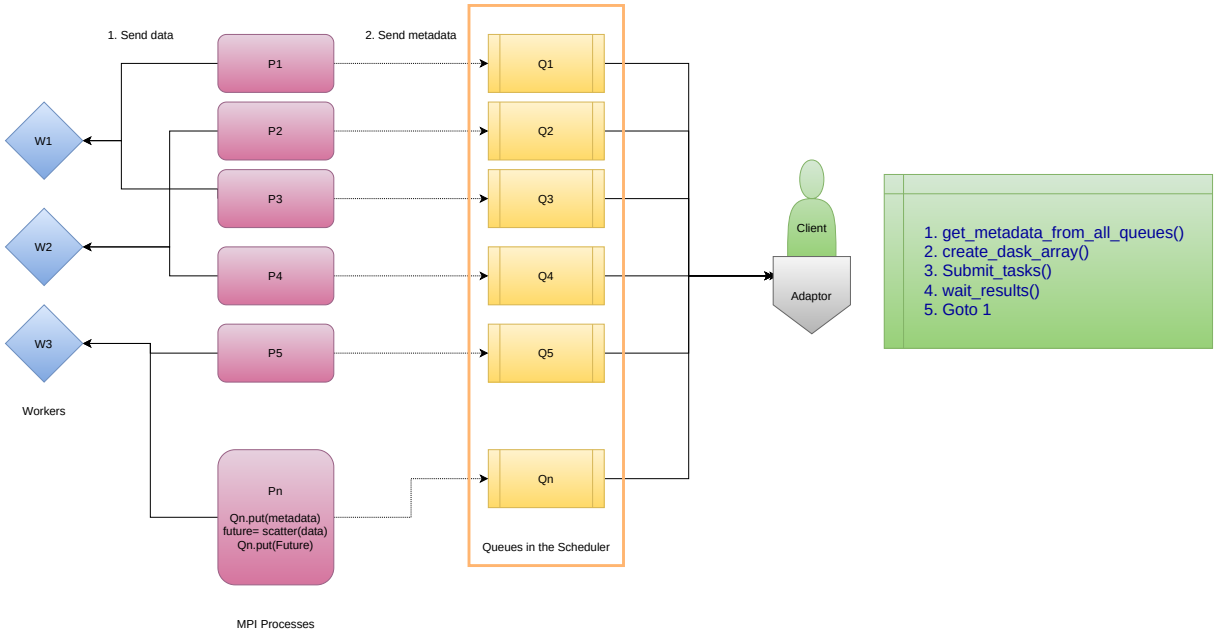


Figure 5.2: Control flow in DEISA1.

corresponding `dask.array`. In our work, we do the other way around where we first create the chunks with the keys that we want (keys we have gotten already from the `scatter`, to be able to identify the blocks), then we position them correctly in the array as real chunks then create it. The array creation is transparent to the user.

5.2.3 Control Flow

To manage the data flow in DEISA1 we used the `Queues` to stop the simulation from pushing data into the Dask workers. As already explained in Section 5.2.1, we use one `Queue` per bridge, where we put first the metadata and then the *Future* returned by the `scatter`. We have used the `Queue` sizes to control the flow. When a `Queue` is full, writing to it is blocked, suspending the activity of the MPI process.

The minimum size of a *Queue* is two because at each timestep we need to send 2 messages, and the maximum is $2N$ where N is the maximum of timesteps data the workers can host at the same time.

When a `Queue` is full, the bridges are blocked trying to add the next message containing the metadata dictionary in the queue. When they are unblocked, they can finally run the `scatter` and send the data.

Figure 5.2 shows how the data flow is implemented in the DEISA architecture. We have associated one `Queue` with each bridge. It will put its metadata into it until it is full: first, the metadata dictionary, then it performs the *scatter* and finally sends the `Future`. Note that if the bridge is blocked in the first step, `scatter` is not executed until it is unblocked (the adaptor gets data from the `Queue`).

The client gets two messages at once from each queue and submits the tasks graph corresponding to the timestep. It can wait or not until the results are computed. If the next metadata is not available yet, the client is blocked as long as the queues are empty.

5.2.4 Data Redistribution

The number of MPI processes can be different from the number of Dask workers. The generated blocks by the MPI processes are sent directly to the workers' memory following a round-robin scheme to load balance the distribution of the blocks workers. Each process obtains the list of connected workers and computes the position of the corresponding worker in the list. The position of the corresponding worker is computed using this formula: $worker = list(workers)[rank \ mod \ len(list(workers))]$. We suppose that the list of the connected workers will not change during runtime, so at the beginning, the adaptor gets the list of connected workers and puts it in a Dask `Variable` to be available for the bridges.

We assume that the user puts enough workers and enough memory in each worker to avoid memory overflows. There is no rule for that, as it depends on the simulation, its duration, the size of the data it generates and the time needed to process that data.

5.2.5 User API and Configuration

In this section, we present the DEISA plugin configuration and the user API in DEISA.

5.2.5.1 DEISA Bridge User API

There are three main Methods in the DEISA bridge object, and an initialization function:

Methods	Documentation
`Init(Nbr_bridges, rank, position, queue_size)`	Initialize the Bridge object.
`Bridge.__init__(Nbr_bridges, rank, position, queue_size)`	A bridge object is created, and it initialize the number of bridges, the rank of the MPI process associated with this bridge, the position of the data in the global distribution, and the size of the metadata queue to control the flow.
`Bridge.publish_data(timestep, data)`	Take the timestep and the data as parameters, it send the data to the associated worker and the metadata to the corresponding queue.
`Bridge.finalize()`	Disconnect the bridge from the scheduler.

Table 5.1: Bridge User API methods.

In this version, we use the PDI `pycall` plugin to call the Python API of the bridge. Three main events trigger the bridge methods: `Init, Available` and `Finalize`. At the initialization step (`Init` event here), the simulation (Listing 5.2) shares needed metadata with PDI to initialize the bridge object. In the `pycall` specification tree (Listing5.1), we handle the `Init` event by calling the *Init* function, offered by Python DEISA API (in Table 5.1), that initializes the bridge object.

When the simulation sends the `Available` event along side the data it wants to share for external use, the `publish_data()` method handles it in the `pycall` specification tree. This method sends the data and the metadata to Dask. This part is done inside the main loop, so each time the simulation reaches this `pdi_multi_expose` function, the available data it sent to Dask.

Finally, when the simulation is finished, we disconnect all the bridges from the Dask scheduler by emitting the `Finalization` event that it handled by calling the *finalize* method of the bridge.

All in all, the user has to keep in mind three main steps: at the beginning of the simulation, they have to initialize the coupling by calling `Init()` that returns a Bridge instance. Every time they want to share data with Dask, they call `Bridge.publish_data`. And once finished, they disconnect the bridges from Dask by calling `Bridge.finalize()`.

5.2.5.2 DEISA Adaptor User API

Methods	Documentation
`Initialization(Nbr_bridges, Nbr_workers)`	Initialization is a wrapper to initialize the Adaptor object.
`Adaptor.__init__(Nbr_bridges, Nbr_workers)`	An Adaptor object is created; it takes the number of associated bridges and the number of the workers that will be expected.
`Adaptor.get_client()`	This method returns the analytics client created by the Adaptor
`Adaptor.get_data()`	This method returns the `dask.array` of a given timestep when available.
`Adaptor.finalization()`	This method disconnects the clients from the scheduler and shutdown the dask cluster.

Table 5.2: Adaptor User API methods.

The user has to initialize the coupling from Dask side too. In their analytics script, as shown in Listing 5.4, they call the `Initialization()` function that returns an Adaptor object. The adaptor

```yaml
# ...
pdi:
  metadata:
    timestep:     int
    MaxtimeSteps: int
    pcoord_1d:     int
    pcoord: { type: array, subtype: int, size: 2 }
    dsize:  { type: array, subtype: int, size: 2 }
    psize_1d: int
    gmax: int
  data:
    local_t:
        type: array
        subtype: double
        size: ['$dsize[0]', '$dsize[1]']
  plugins:
    pycall:
      on_event:
        Init: # Initialization step
          with: # Needed parameters
            position: $pcoord
            rank: $pcoord_1d
            mpi_size: $psize_1d
            queue_size: $gmax
          exec: | # Initialize deisa
            from deisa import Init
            Bridge = Init(mpi_size, rank, position, queue_size)
        Available: # Data communication at each `Available` event
          with:
            timestep: $timestep
            local_t: $local_t
          exec: | # Send data and metadata to the deisa adaptor
            Bridge.publish_data(timestep, local_t)
        Finalization: # Finalization step
          with: ~
          exec: |
            Bridge.finalize()
# ...
```

Listing 5.1: Deisa configuration file.

connects a client to the scheduler, and the `Adaptor.get_client()` can be called to access it. The metadata sent by the bridges in the queues is retrieved by the client and used to create a `dask.array`. To get access to it, the user can call `Adaptor.get_data()` that returns a `dask.array` object. This array can be used as any ordinary `dask.array`; thus, all the available API[1] is usable. To finalize the Dask cluster, the `Adaptor.finalization()` is called. This method makes sure that there are no connected bridges and then shuts down the cluster.

Note that in this multi-graph version, one `dask.array` is created per timestep. Thus if several steps are waited, the *Adaptor.get_data()* has to be called as many times as needed. We suppose that the total number of timesteps is known in advance.

5.3 Experiments and Evaluation

Now that we have explained how our DEISA prototype operates, we evaluate its performance in two supercomputers: Ruche (Section 5.3.1.1) and Irene (Section 5.3.1.2). This section details how DEISA can be used shows the API and configuration, and compares its performance to post hoc analytics.

5.3.1 Launching Experiments

There are several ways to launch a Dask cluster. However, since our work aims to use Dask for in situ analytics, we opt to use the command line to include launching both Dask and the simulation in the same submission script and the same job. One of the most important motivations for that is the fact that we need both Dask and the simulation launched together for in situ analytics. If we launch them

[1] https://docs.dask.org/en/stable/array-api.html

```
int main( int argc, char* argv[] )
{
    /*...*/
    // Loop counter referring for the timestep
    int ii=0;

    // Share useful metadata for initialization step
    PDI_multi_expose( "Init",
                      "pcoord",       pcoord,       PDI_OUT,
                      "pcoord_1d",    &pcoord_1d,   PDI_OUT,
                      "dsize",        dsize,        PDI_OUT,
                      "psize",        psize,        PDI_OUT,
                      "psize_1d",     &psize_1d,    PDI_OUT,
                      "timestep",     &ii,          PDI_OUT,
                      "gmax",         &gmax,        PDI_OUT,
                      "MaxtimeSteps", &generations, PDI_OUT,
                      NULL);

    // Main loop
    for (; ii<generations; ++ii) {

        // control simulation duration by performing more substeps
        for (int jj=0; jj<200; ++jj){
            // Compute the values for the next iteration
            iter(dsize, cur, next);

            // Exchange data with the neighbours
            exchange(cart_comm, next);

            // Swap the current and next values
            double (*tmp)[dsize[1]] = cur; cur = next; next = tmp;
        }

        // Send the `Available` event and share available data with PDI
        PDI_multi_expose("Available",
                         "timestep",   &ii, PDI_OUT,
                         "local_t",    cur, PDI_OUT,
                         NULL);
        // A barrier to synchronising the processes
        MPI_Barrier(cart_comm);
    }

    // Send the `Finalization event`
    PDI_multi_event("Finalization");

    // Finalize PDI
    PDI_finalize();
/*...*/
}
```

Listing 5.2: Simulation main loop.

in two different jobs, we do not have control over the starting moment of each. For instance, if data is already available in the simulation, but Dask is in the waiting queue to be started, then we may lose the data after a timeout (if we didn't activate any write). That is, after the timeout, an exception is raised and PDI returns but the simulation can resume correctly.

To do so, we use commands already available to launch the different parts of a Dask cluster. Namely, `dask-scheduler` to start the scheduler process, `dask-worker` to start worker processes in one or more nodes. The analytics client is launched as a Python process also.

At the end, four steps are launched in one script; the scheduler first creates a file containing the connection information. Once this is done, the client and the workers can be connected, and the simulation is launched. Launching the steps in the background and waiting for all the steps is mandatory, else the first finished step will stop the job.

5.3.1.1 Ruche Cluster

We used for our experiments the Ruche[29] cluster (Moulon mesocentre, Paris-Saclay). The cluster is composed of 216 ThinkSystem SD530 server nodes; each with 2 Intel Xeon Gold 6230 20C @ 2.1GHz

```python
import os
import h5py
import dask.array as da
from dask_ml.decomposition import IncrementalPCA

# Connect the analytics client to the scheduler
def connect('scheduler.json'):
    # ...
    return client

client = connect(scheduler_info)

# Dask Configuration
dask.config.set({"distributed.deploy.lost-worker-timeout": 60, "distributed.workers.
    memory.spill":0.97, "distributed.workers.memory.target":0.95, "distributed.workers.
    memory.terminate":0.99 })

# Get HDF5 dataset information
f = h5py.File('data.h5', 'r')
ds = f['local_t']

# Create a dask array from the dataset with the same chunking as in deisa
gt = da.from_array(ds, chunks=(1, chunkx, chunky))

# Initialize and compute the incremental PCA
for i in range(len(gt)):
    if i==0:
       ipca=IncrementalPCA(n_components=2,copy=False, svd_solver='randomized')
    ipca.partial_fit(gt[i])

print("IPCA Algorithm, ", ipca, flush=True)
print("[explained_variance , singular_values]: [",
      ipca.explained_variance_ , ", ",  ipca.singular_values_], "]",
      flush=True)

# Clean up
os.remove('data.h5')
client.shutdown()
```

Listing 5.3: Dask IPCA code.

CPUs and 180GB of maximum user-allocatable memory. Each CPU has 20 cores. The interconnect uses Omni-Path 100 Gbit/s and the parallel file system the Spectrum Scale GPFS (IOs rate: 9 GB/s). Ruche uses the Slurm job management system.

Listing 5.5 shows an example of a submission script on Ruche (see Section 5.3.1.1). This script has four steps, as already described. Each step is launched with `srun`. To make sure that the steps are not launched in the same nodes, we use the `--relative=<n>`[31] option of `srun` that launches the job step relative to node n of the current allocation. In our case, we make sure that each job step is launched in a distinct subset of nodes.

5.3.1.2 Irene Supercomputer

We used the Irene supercomputer in the CEA TGCC centre. We used the skylake partition. It has 1653 nodes, each with 2 CPUs: CPU: 2x24-cores Intel Skylake@2.7GHz (AVX512), 180GB memory per node. Irene has a total of 79344 cores. The compute nodes are connected through an EDR InfiniBand network. This high throughput (100Gb/s) and low latency network is used for I/O and communications among nodes of the supercomputer. Irene uses a Lustre parallel distributed file system. Script 5.6 show the equivalent submission script in Irene Supercomputer. The main difference is the `ccc_mprun` command which is the equivalent of `srun` in Irene.

5.3.2 Heat2D Mini-App

We evaluated our prototype using an implementation of a modified 2D explicit finite difference heat solver. It has been parallelized with MPI using a block domain decomposition, also called a cartesian topology of processes, because the subdomains are squares or rectangles (Listing 5.2). It is representative

```
from deisa import Initialization
import dask.array as da
from dask_ml.decomposition import IncrementalPCA

# Get configuration data
with open(r'config.yml') as file:
    data = yaml.load(file, Loader=yaml.FullLoader)
    Ssize = data["parallelism"]["height"]*data["parallelism"]["width"]
    generations = data["MaxtimeSteps"]
    Sworkers = data["workers"]
    timeStep = 1

# Initialize the Adaptor
Adaptor = Initialization(Ssize, Sworkers)

# Get a dask client
Adaptor.client.get_versions(check=True)

# At each timestep we apply a partial_fit
for g in range(0, generations, timeStep):
    # Get dask array from the Adaptor
    arrays = Adaptor.get_data()

    # Reshape the data if necessary
    arrays = da.reshape(arrays, (arrays.shape[1], arrays.shape[2]))

    # Initialize and compute the Incremental PCA
    if g==0:
        ipca=IncrementalPCA(n_components=2, copy=False,
                            svd_solver='randomized')

    ipca.partial_fit(arrays)

    # Delete arrays if they are no more needed
    arrays=None

print("IPCA Algorithm, ", ipca, flush=True)
print("[explained_variance , singular_values]: [",
      ipca.explained_variance_ , ", ",  ipca.singular_values_], "]",
      flush=True)

# Finalization of dask instance
Adaptor.finalization()
```

Listing 5.4: Deisa Client interface.

of a typical 2D Eulerian simulation with stencil computation pattern and MPI ghosts data exchange. Outputs, consisting of the temperature field on the 2D domain, are produced periodically after a fixed number of iterations set to represent a realistic compute-to-output time ratio.

5.3.3 Principal Component Analysis

The Principal Component Analysis (PCA), is an important statistical method for dimensionality-reduction. Historically in has been invented in 1901 by *Karl Pearson* [138] and named by *Harold Hotelling* [108]. It is used to reduce the dimensionality of a dataset, while preserving as much variability (statistical information) as possible[111]. This is accomplished by linearly transforming the data into a new coordinate system where most of the variation in the data can be described with fewer dimensions than the initial data.

The operation of a PCA can be summarized in those steps [19, 106, 13]:

- Standardization: it aims to standardize the range of the initial variables so that each one of them contributes equally to the decomposition.
- Covariance Matrix Computation: in this step, the covariance matrix of the initial variables is computed to see all the possible pairs of correlated variables. Note that highly correlated variables contain redundant information.

```
#!/bin/bash

# Configure the different parameters
SCHEFILE=scheduler.json
NNODES=$(($SLURM_NNODES-2))
WORKERNODES=$(($NNODES/3))
SIMUNODES=$(($WORKERNODES*2))
NPROC=$(($SIMUNODES*2))
NWORKER=$(($WORKERNODES*2))

# Launching the scheduler
srun -N 1  -n 1 -c 20 --relative 0  dask-scheduler   --protocol tcp  --scheduler-file=
    $SCHEFILE 1>> scheduler.o  2>> scheduler.e  &

# Wait for the SCHEFILE to be created
while ! [ -f $SCHEFILE ]; do
    sleep 3
    echo -n .
done

# Connect the client to the Dask scheduler
srun  -N 1 -n 1 -c 20 --relative 0   `which python` -m trace -l -g client.py 1>> client
    .o 2>> client.e &
client_pid=$!

# Launch Dask workers in the rest of the allocated nodes
srun  -N $WORKERNODES -n $NWORKER -c 20 --relative 1   dask-worker --local-directory /
    tmp  --scheduler-file=${SCHEFILE} 1>> worker.o 2>>worker.e  &

REL=$(($WORKERNODES+1)
# Launch the simulation code
ccc_mprun  -N $SIMUNODES -n $NPROC -c 20 --relative $REL  ./simulation 1>> simulation.o
     2>> simulation.e &

# Wait for the client process to be finished
wait $client_pid
wait
```

Listing 5.5: Submission script of simulation and in situ analytics in Ruche supercomputer.

- Eigenvectors and Eigenvalues Computation: here, the eigenvectors and the eigenvalues of the covariance matrix are computed to identify the principal components. Geometrically, principal components represent the directions of the data that explain a maximal amount of variance. The covariance matrix's eigenvectors are the axes' directions where there is the most variance (most information), and they are called *Principal Components*. The eigenvalues are simply the coefficients attached to eigenvectors, which give the amount of variance carried in each principal component, and the eigenvectors, in the order of their eigenvalues, highest to lowest, are the principal components in order of significance.
- Feature Vector: the feature vector is the matrix that has as columns the eigenvectors of the components that we decide to keep
- Final Dataset Computation: by recasting the original data along the computed principal components axes. This can be done by multiplying the original dataset's transpose by the feature vector's transpose.

`Dask-ML`[2] library provides scalable machine learning algorithms in Python using 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 (as data comes as the simulation progresses). The 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

```
#!/bin/bash

# Configure the different parameters
SCHEFILE=scheduler.json
NNODES=$(($SLURM_NNODES-2))
WORKERNODES=$(($NNODES/3))
SIMUNODES=$(($WORKERNODES*2))
NPROC=$(($SIMUNODES*2))
NWORKER=$(($WORKERNODES*2))

# Prepare the spack environment
ml purge
source $CCCWORKDIR/env_spackuser
unset LD_PRELOAD
unset SELFIE_MPRUN
unset SELFIE_MSUB
export OMP_NUM_THREADS=24

# Launching the scheduler
ccc_mprun -N 1  -n 1 -c 24 -r 0  dask-scheduler --protocol tcp --scheduler-file=
    $SCHEFILE 1>> scheduler.o  2>> scheduler.e  &

# Wait for the SCHEFILE to be created
while ! [ -f $SCHEFILE ]; do
    sleep 3
    echo -n .
done

# Connect the client to the Dask scheduler
echo Connect Master Client
ccc_mprun  -N 1 -n 1 -c 24 -r 1   `which python3` ipca.py 1>> client.o 2>> client.e &
client_pid=$!

# Launch Dask workers in the rest of the allocated nodes
echo Scheduler booted, Client connected, launching workers
ccc_mprun  -N $WORKERNODES -n $NWORKER -c 24 -r 2   dask-worker --local-directory /tmp
     --scheduler-file=${SCHEFILE} 1>> worker.o 2>>worker.e  &

REL=$(($WORKERNODES+2))
# Launch the simulation code
echo Running Simulation
ccc_mprun  -N $SIMUNODES -n $NPROC -c 24 -r $REL  ./simulation 1>> simulation.o 2>>
    simulation.e

# Clean up the files
`which python3` postscript.py

# Wait for the client process to be finished
wait $client_pid
wait
```

Listing 5.6: Submission script of simulation and in situ analytics in Irene supercomputer.

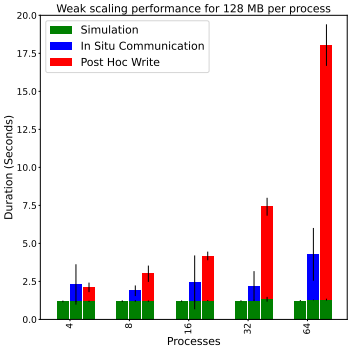
Listing 5.4 shows how the IPCA analysis in a DEISA client script.

5.3.4 Performance Evaluation

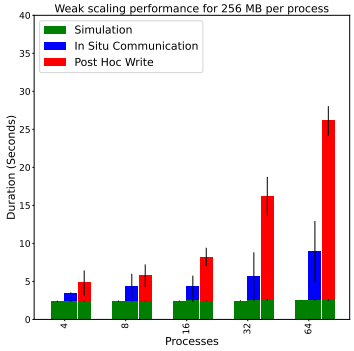
We performed two main experiments:

- **Experiment I** compares DEISA performance to a baseline with neither IO nor analysis, and to a version using a parallel post hoc analysis with plain Dask.

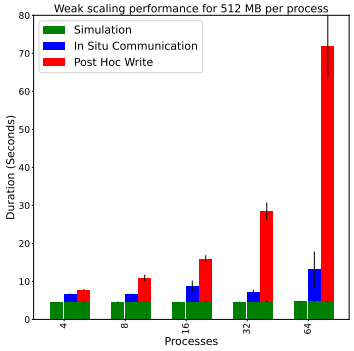
- **Experiment II** investigates the performance of DEISA more in-depth on large and small data sets to explain its behaviour.



(a) Average simulation, communication and IO times per iteration for case: 128 MiB per process



(b) Average simulation, communication and IO times per iteration for case: 256 MiB per process



(c) Average simulation, communication and IO times per iteration for case: 512 MiB per process

Figure 5.3: Weak scaling average simulation, communication and IO times per iteration for 128 MiB, 256 MiB and 512 MiB per process for three experiments: the first bar from the left of each scale represents the baseline (simulation time without any IOs), the second shows the results for DEISA (simulation and communication time over network), and the third bar represents results for the post hoc experiment (simulation and parallel HDF5 write).

5.3.4.1 Experiment I

Those experiments have been performed in the Irene supercomputer. We have used the developed MiniApp and the Incremental PCA. Table 5.3 and Table 5.4 summarize the parameters used and configurations in these experiments.

Parameter	Value
Number of runs	3
Number of iterations IPCA	10
MPI nodes / Dask worker node	2
MPI process / MPI node	2
Dask worker / Dask worker node	2
Thread / Dask worker	24
MPI process / Dask worker	2
DEISA client code	Listing 5.4
Dask client code	Listing 5.3

Table 5.3: Fixed parameters used in the Experiment 5.3.4.1.

Configuration	XP1:128 MiB	XP1:256 MiB	XP1:512 MiB
MPI block size	128	256	512
Dask chunk size	128	256	512
MPI Nodes	[4, 8, 16, 32, 64]		
Dask Nodes	[2, 4, 8, 16, 32]		

Table 5.4: The three configurations of Experiment 5.3.4.1.

Figure 5.3 shows the weak scaling results for the different configurations of **Experiment I**(Section 5.3.4.1). In the first subfigure from the left (5.3a), we have fixed the size of the data per MPI process to 128 MiB, to 256 MiB in the subfigure in the middle (5.3b), and to 512 MiB in the subfigure in the right (5.3c). The x-axis of each subfigure represents the variation of the MPI processes from 4 to 64, and y-axis represents the duration in seconds of the different steps. In each subfigure, we have the three cases: the first bar from the left, of each scale, is the baseline (simulation without IO). The stacked bar in the middle shows results for DEISA with the simulation time in green, and the in situ communication time over the network in blue. The last stacked bar of each scale show results for the post hoc experiment: simulation time in green and parallel HDF5 *write* time in red. The represented values are the maximum duration per

(a) Total analytics time for case: 128 MiB per process

(b) Total analytics time for the case: 256 MiB per process

(c) Total analytics time for case: 512 MiB per process

Figure 5.4: Weak scaling performance for the analytics. The first bar from the left shows the duration time in seconds of the in situ incremental PCA with DEISA, it includes waiting for the data to be available and the analytics duration. The second bar shows results for the post hoc version that includes both reading data from the PFS and processing it with the same algorithm.

iteration, averaged over all ranks of all runs. We have considered the maximum to give a representative value of iteration duration, as there is a synchronization barrier after communicating the data. The error bars are the standard deviation. The simulation time for all the experiments is almost the same with minimal variability, and it weak-scales perfectly when increasing the problem size.

The DEISA communication time is less than the HDF5 write time in all experiments, which is expected, since in DEISA we just send the data over the network, whereas in post hoc, we write data to the shared PFS.

In theory, DEISA will have better performance than post hoc, when the aggregated network bandwidth of Dask workers is greater than the data rate transfer of the Scratch disks of Irene. The network bandwidth is 100Gb/s, and the Scratch data rate transfer is 300GB/s. We can expect to be better than post hoc, starting at 24 worker nodes. However, with only two worker nodes, DEISA is already slightly better than post hoc performance in the three subfigures. This is explained by the fact that we never reach the theoretical disk transfer rate in real experiments. In theory, we expect DEISA to scale perfectly as the aggregated bandwidth increases in correlation with the problem size. However, we notice that the DEISA communication times increase slightly when increasing the number of processes. The small variation and the variability are due to the number of communications to the Dask scheduler that may slow it down.

The gap between in situ and post hoc performance increases when increasing the number of nodes (simulation and Dask worker). This is because the rate transfer is fixed. Thus, the post hoc analysis suffers from IO bottleneck, while the in transit analytics benefits from the aggregated network bandwidth that increases with the problem size. The variability in the red bars (HDF5 write) is likely due to sharing the parallel file system with other applications.

Figure 5.4 shows the weak scaling results for the analytics part of the different configurations of **Experiment I**. In the first subfigure from the left (5.4a), we fixed the chunk size, which is the size of the data per MPI process to 128 MiB, to 256 MiB in the subfigure in the middle (5.4b), and to 512 MiB in the subfigure in the left (5.4c). The x-axis of each subfigure represents the variation of the MPI processes from 4 to 64, equivalent to the variation of the Dask workers from 2 to 32. The y-axis represents the duration in seconds of the analytics. In each subfigure, we have the DEISA and the Dask analytics: the bar on the left of each scale is the DEISA analytics time that includes compute time and waiting for the data from the next step. The bar on the right of each scale shows the analytics time, which includes reading the data from the disk and analysing the data. The represented values are the mean duration over the three runs. The bar errors represent the standard deviation over the three runs. Results for DEISA are quite good compared to post hoc. Since the analytics time includes waiting for simulation data, we can not affirm that the IPCA algorithm scales perfectly. The variability is also limited.

When the chunks are 128 MiB, the post hoc times increase exponentially, starting from 8 workers to reach x25 times longer than in situ time at 16 workers. Similarly, when the chunks are 256 MiB, with 8 workers, the analytics time increases exponentially. For 512 MiB, the scaling is linear but not better than for the other sizes. Since we are performing the exact same algorithm as in the in situ case with the exact same configurations, the only reason to have such results may be the time spent by Dask workers

(a) Total analytics time for case: 128 MiB per process

(b) Total analytics time for the case: 256 MiB per process

(c) Total analytics time for case: 512 MiB per process

Figure 5.5: Weak scaling performance for the analytics with chunking activated while writing the HDF5 file. The chunking is equal to the size of data per MPI process and the chunking in Dask. The first bar from the left shows the duration time in seconds of the in situ incremental PCA, and the second bar shows results for the post hoc version that includes both reading data from the PFS and processing it with the same algorithm.

in reading the data. The Dask workers get the tasks at runtime; one reason may be the fact that the workers open and close the file for each *read* task. Another reason may be the chunking of the files: we have not specified any chunking of the PDI HDF5 plugin, so by default, no chunking is activated[35]. Dask may not be efficient in reading non-chunked files.

Note that we do not have values for some of the post hoc analytics experiments because the three runs crashed. The worker's and scheduler's logs show that the scheduler kills the workers after a given timeout without sending heartbeats. This is the case when the workers do long IOs.

We repeated the experiment by activating the chunking of the HDF5 files. It is equal at each time to the local block size of an MPI process which is equal to the chunking in Dask too. The results are represented in Figure 5.5. Post hoc performance is clearly better with the chunking activated. Even if the DEISA time includes the waiting for simulation data, it weak-scales better than post hoc, and the ratio between the DEISA performance and the post hoc one increases when increasing the number of Dask workers likely because of the shared parallel file system.

The missed values for post hoc here are due to crashes on the simulation side, likely because of the bug in HDF5 [9].

Figure 5.6 (in Page 73) and Figure 5.7 (in Page 5.7) show the two task streams generated by Dask for the Incremental PCA, with 64 MPI processes, 32 workers and 128 MiB, Dask post hoc version in the first and DEISA in situ version in the second. The x-axis represents the time, and the y-axis represents the worker cores. The small colored patches are tasks. Each color represents a type of task. We will not detail more those task streams, but they can be found online[6]. The colors in the task stream are not similar, as the submitted task graphs are different. In the in situ version, we do not perform reads, for instance. In the post hoc stream task, Dask generated more tasks compared to the in situ version because reading data from disk are considered tasks, so they are also included in the count. In both figures, we can notice that there are 10 steps (iterations). The algorithm computes the PCA incrementally. It submits a task graph for each iteration.

5.3.4.2 Experiment II

In this experiment, we perform a detailed study about the variability and performance of DEISA communications and how they affect the scheduler performance. This section has already been presented in paper [101].

Those experiments have been done on the Ruche cluster, where we used the same mini-app and analytics as for the previous experiment. We investigate the performance of DEISA in depth when either the data size or computer resources vary. Those two parameters are important because they may affect the Dask performance negatively.

[6] https://gitlab.maisondelasimulation.fr/agueroud/phd_xp

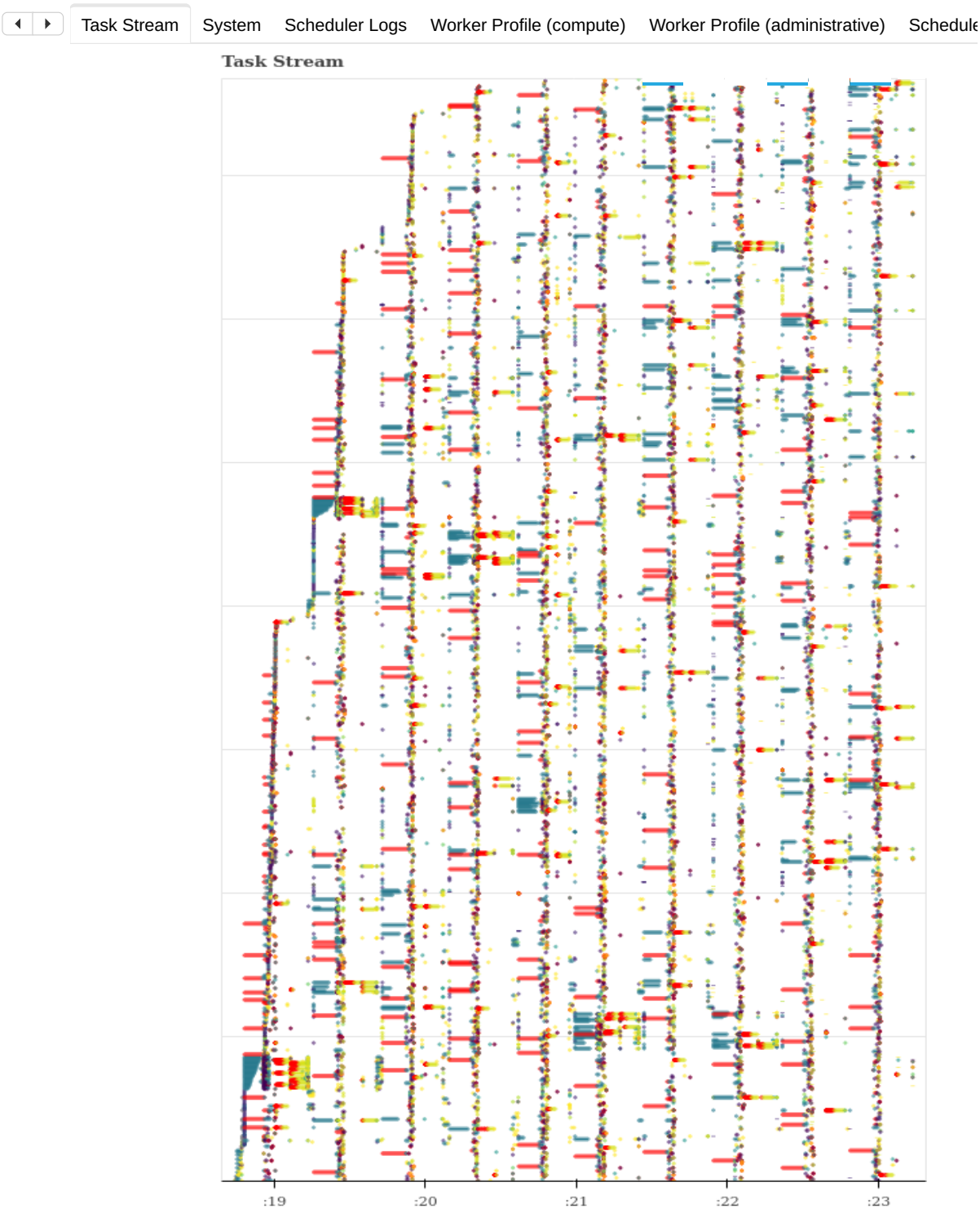


Figure 5.6: Task stream generated by Dask for post hoc incremental IPCA with chunking activated for 64 processes, 32 workers and 128 MiB per process. Number of tasks: 11565, Compute time: 3017.67s, Deserialize time: 25.39s, Disk-read time: 64.45ms, Transfer time: 2709.88s.

To show this impact we vary the size of the data per MPI process from 1 MiB to 256 MiB, and we use either 128 or 512 MPI processes. The size of the data per MPI process corresponds to the size of the chunks in the Dask analytics, and the number of MPI processes is the number of bridges connected to the scheduler.

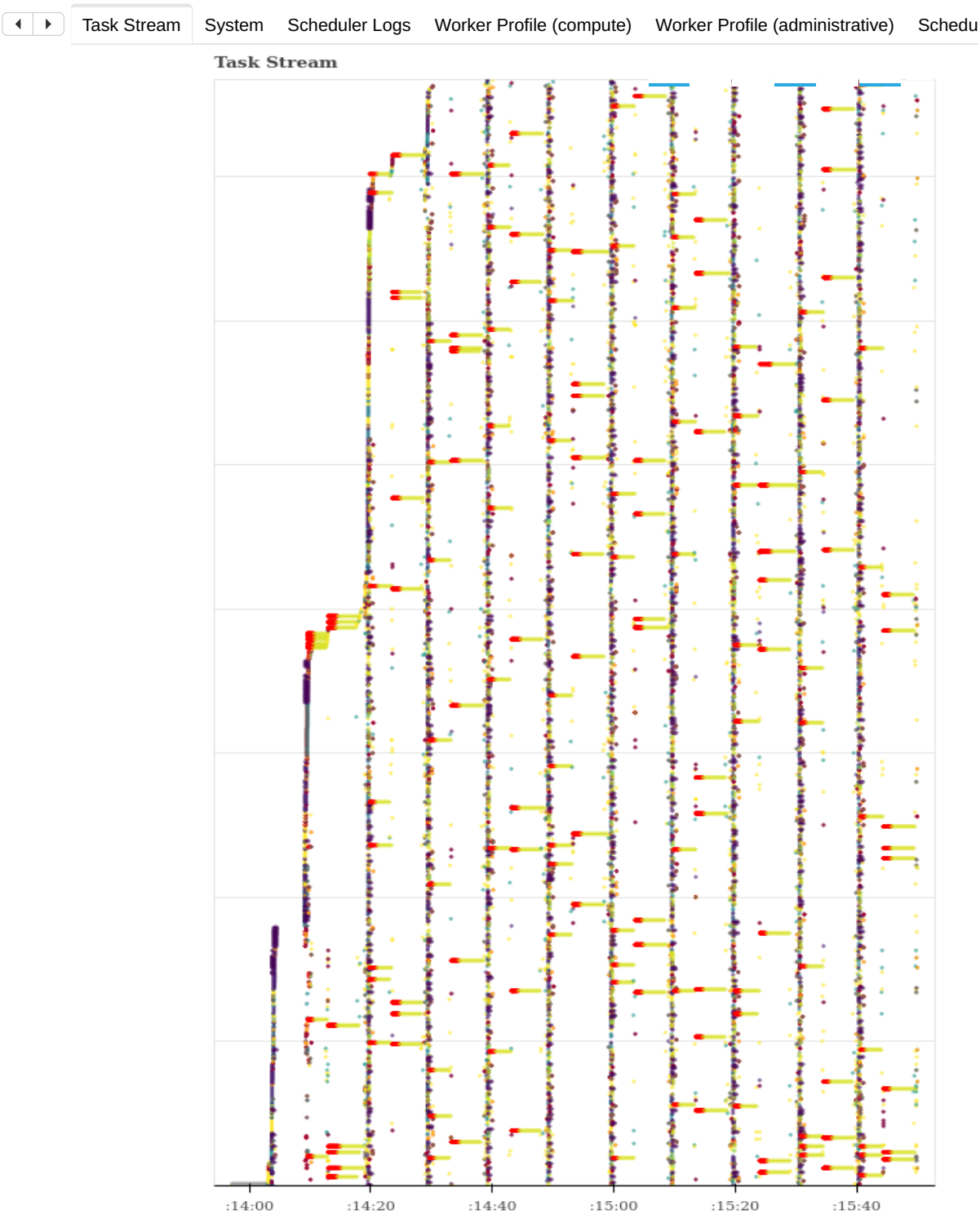


Figure 5.7: Task stream generated by Dask with in situ analytics enabled for the IPCA, for 64 processes, 32 workers and 128 MiB per process. Number of tasks: 9269, Compute time: 1104.79s, Deserialize time: 5.89s, Disk-read time: 63.47ms, Transfer time: 1007.39s.

Table 5.6 summarizes the four configurations tested, and Figure 5.8 (in Page 5.8) presents the average execution time over 3 runs of 10 iterations as well as an error bar representing the min and max values when standard deviation exceeds 2%. We excluded the first iteration here as it may include initialization time.

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

Table 5.5: Fixed parameters used in Experiment II 5.3.4.2.

Configuration	XP2:1 MiB:128	XP2:1 MiB:512	XP2:256 MiB:128	XP2:256 MiB:512
Data size / MPI process	1 MiB	1 MiB	256 MiB	256 MiB
Total nodes	6	21	6	21
MPI node	4	16	4	16
Dask worker nodes	1	4	1	4
client & scheduler node	1	1	1	1
Global data size	128 MiB	512 MiB	32 GiB	128 GiB
Data size / MPI node	32 MiB	32 MiB	8 GiB	8 GiB
Generated tasks	15330	55330	15210	55150

Table 5.6: Four configurations used for Experiment II 5.3.4.2.

On MPI side, we identify the compute time, the time to transfer the data from DEISA bridge to Dask workers, and that to send the required metadata to the scheduler. We also measure the duration of a barrier inserted just after these communications. This barrier captures the time required to re-synchronize the processes after potentially differing time in the communications. Without it, this time would be counted as part of the computation time.

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

At 1 MiB/process, the MPI simulation executes much faster than the analysis; this is reversed at 256 MiB/process. The task granularity has a high impact on Dask performance. At 1 MiB/process, the average time per task is at most 1ms, which is very small compared to the minimum recommended task duration of 100ms[15] in Dask. With a scheduling overhead of about 1ms per task, scheduling and communication overheads account for more than half the measured time at this granularity. With larger chunks, 256 MiB, and so longer tasks, Dask analytics become faster than the MPI part.

The experiment is run with no maximum `queue` size between DEISA bridges and the adapter. Hence, when the simulation produces data faster than the analysis can consume it (1 MiB/process configurations), data is buffered in the worker nodes' memory and processed even after the end of the simulation. The total time to solution is limited by the analytics part. On the other hand, when the simulation produces data slower than the analysis can consume it (256 MiB/processes configurations), Dask spends time waiting for metadata that is not yet produced by the simulation. The total time to solution is limited by the MPI part and Dask workers are idle for more than 4/5 of the iteration; a time that appears as part of the metadata fetch.

At 1 MiB/Process, data and metadata transfer costs are significant. This is mostly explained by the fact that at this scale, communication time is noticeably impacted by network latency. The data transfer performance is also explained by the behaviour of Dask `scatter` used by DEISA to transfer data to the workers. This method directly transfers data to the worker, but it also establishes an additional connection to the scheduler to notify it of the new data reference. For large enough data, this is negligible, but at this scale, this starts to be noticeable. In addition to the latency, another factor impacts network performance for small data sizes. When the data is small, simulation time is too, and data production frequency increases. The high number of requests sent to the scheduler per second can impact its response time. For the configuration XP2:1 MiB:128, more than 1920 requests/s are sent to the scheduler, and more than 9116 requests/s for configuration XP2:1 MiB:512. The time required to send metadata becomes almost 7 times longer in the configuration XP2:1 MiB:512 than in configuration XP2:1 MiB:128, while

the number and size of requests per MPI process are the same. At 256 MiB/process, this difference is still visible, but metadata handling represents less than 1.6% of an iteration in the worse case.

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

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

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

5.4 Limitations

This implementation suffers from several limitations. First, the centralized scheduler very quickly becomes a bottleneck due to the number of messages it receives while increasing the number of bridges. In addition to the heartbeat messages sent by each client to the scheduler every five seconds, sending metadata frequently slows down the scheduler. The second limitation is related to the quantity of data that is sent at each timestep. This implementation has no automatic way to select and send only needed data to the workers, even if eventually not used by the analytics. All generated data is sent to the workers. This limitation implies extra time, memory and energy at each timestep. Finally, because the task graphs are submitted at each timestep, all dependencies among the time dimension must be managed manually, making writing some algorithms complicated.

5.5 Summary

This chapter was dedicated to presenting the DEISA1 prototype, which implements a full in situ workflow based on the DEISA bridging model. We have presented its architecture and operation up to the real deployment on two supercomputers. Finally, we have evaluated the DEISA architecture and compared it to the post hoc analytics using Dask. We have shown that with almost the same coding efforts DEISA prototype shows better performance than post hoc. We have highlighted the different limitations of DEISA and the inherited ones from Dask.

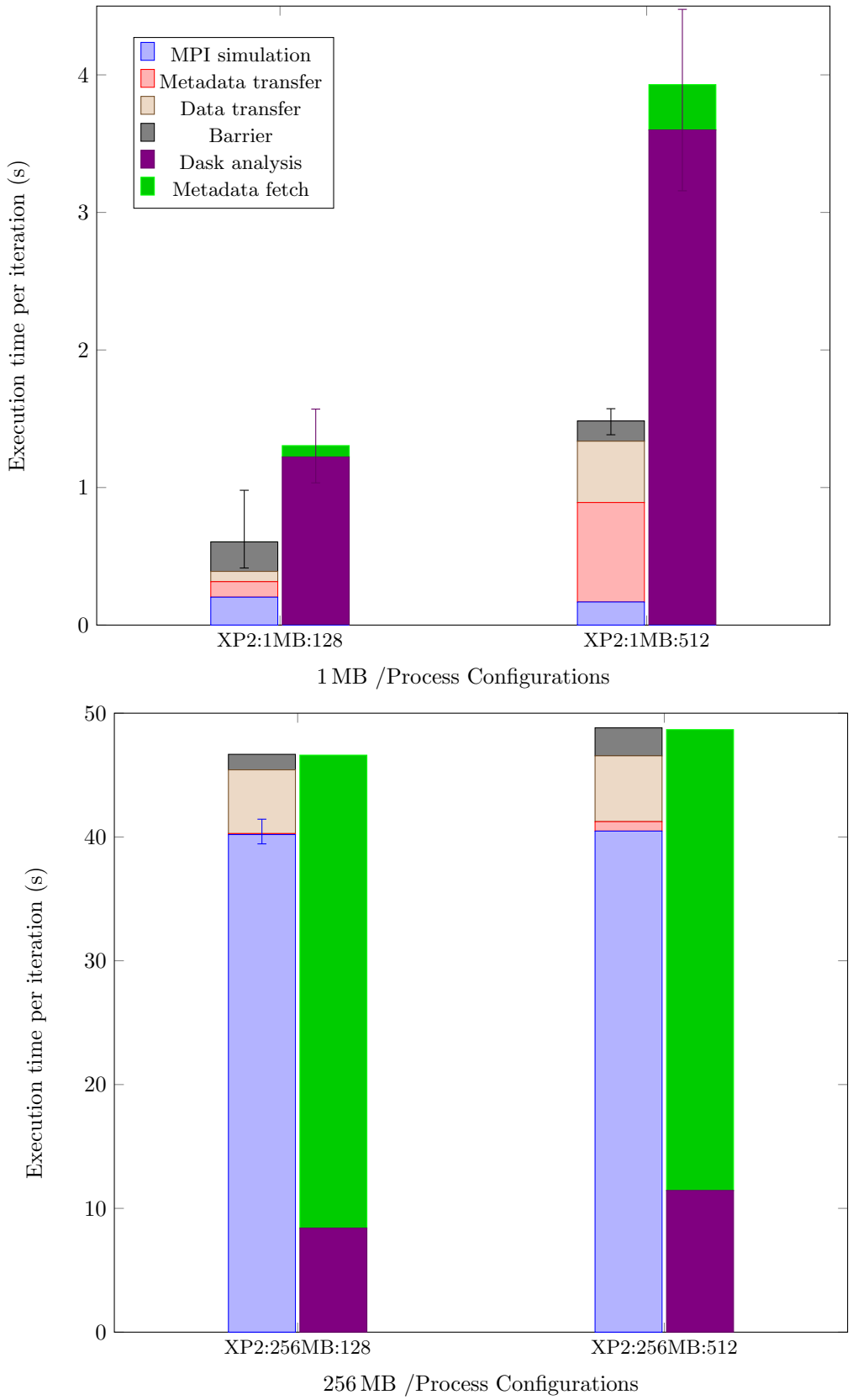


Figure 5.8: Detailed timing per time step for DEISA with 1 or 256 MiB /rank and 128 or 512 processes (6 or 21 nodes, respectively). For each configuration, the left bar shows the timing for the MPI simulation, and the right timings for Dask analytics.

Chapter 6

Dask-Enabled External Tasks for In Transit Analytics

Testing leads to failure, and failure leads to understanding

Burt Rutan

In this chapter, we address some limitations of DEISA1 by introducing three main concepts: DEISA virtual arrays, contracts, and external tasks. Thanks to those concepts, we implement a single-graph version of the DEISA bridging model, where one complete task graph can be submitted at the beginning of the simulation without waiting for the data to be available. This implementation improves both the performance and the user interface and takes advantage of Dask graph optimizations.

6.1 Overview

We address some limitations of DEISA1 by introducing three main concepts: DEISA virtual arrays (Section 6.3.1.3), contracts (Section 6.3.1.4), and external tasks in Dask distributed (Section 6.3.2.1). A DEISA virtual array describes the spatiotemporal domain decomposition of a generated data array. Describing the data in this way allows us to have a global view of the generated data. DEISA virtual arrays are used to create the `dask.arrays` in the analytics client. Contacts make selections on the DEISA virtual arrays to only send needed data to the workers. Those improvements allow us to implement the single-graph version of the DEISA bridging model.

The created `dask.arrays` are collections of external tasks, with the specific state `'deisa'` and particular keys. They refer to tasks computed by other applications outside Dask, and can be used as `dask.array`, thus integrated into Dask task graphs transparently. In Dask, data is considered as a specific task called pure data task, and the external tasks belong to that category. Those tasks only become visible and usable when the simulation sends the generated data with that specific keys to a worker. The tasks that depend on that specific external task can then be scheduled.

We have implemented these improvements on top of DEISA1[101]. We have added a new DEISA plugin in the PDI data interface and included our external tasks contribution into a forked version Dask distributed repository in [41].

6.2 Architecture

The architecture is similar to the previous DEISA version (see Figure 6.1). We still have 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. We improve and optimize the workflow operation by minimizing the load in the scheduler and providing a better user API. The main changes in the architecture are meant to minimise the load of the centralized scheduler and the way the two components communicate.

We have kept the implementation of the bridge built on the Dask client class. We still go through the scheduler for all communications between the bridges and the analytics client.

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, makes a data selection using `slice` objects in the DEISA array depending on the pieces needed for analytics. Then the client sends back the selections to the bridge. And this is done via a Dask *Variable*, which is accessible to all bridges afterwards. This is done at the beginning so that there is no need to send any metadata to the scheduler at each timestep, which improves the performance.

All the bridges are synchronized at this step and can go further as soon as the data they must send is known, or in other words, contracts are signed. Then the client submits the analytics. At each time step, each bridge checks if its block of data is needed. If so, it sends to the pre-selected worker.

We suppose in this version that the data sizes, including the time dimension, are known in advance. We are aware that this is not always possible.

6.3 Implementation

We implemented this version on top of DEISA. Most of the improvements are meant to achieve better performance and make Dask distributed support naively external tasks. To do so, we extended the `scatter` function, introduced `external` tasks in a forked version of Dask distributed to support external tasks and improved the DEISA plugin to describe virtual arrays easily and support contracts.

6.3.1 Data Communication and Metadata Management

In this section, we detail the data communication with our new implementation of `scatter`, and metadata management using contracts.

6.3.1.1 Data Communication With Scatter

`scatter` is a method in the Dask `Client` class. It is meant to send pieces of data from the client to the workers directly or by passing through the scheduler.

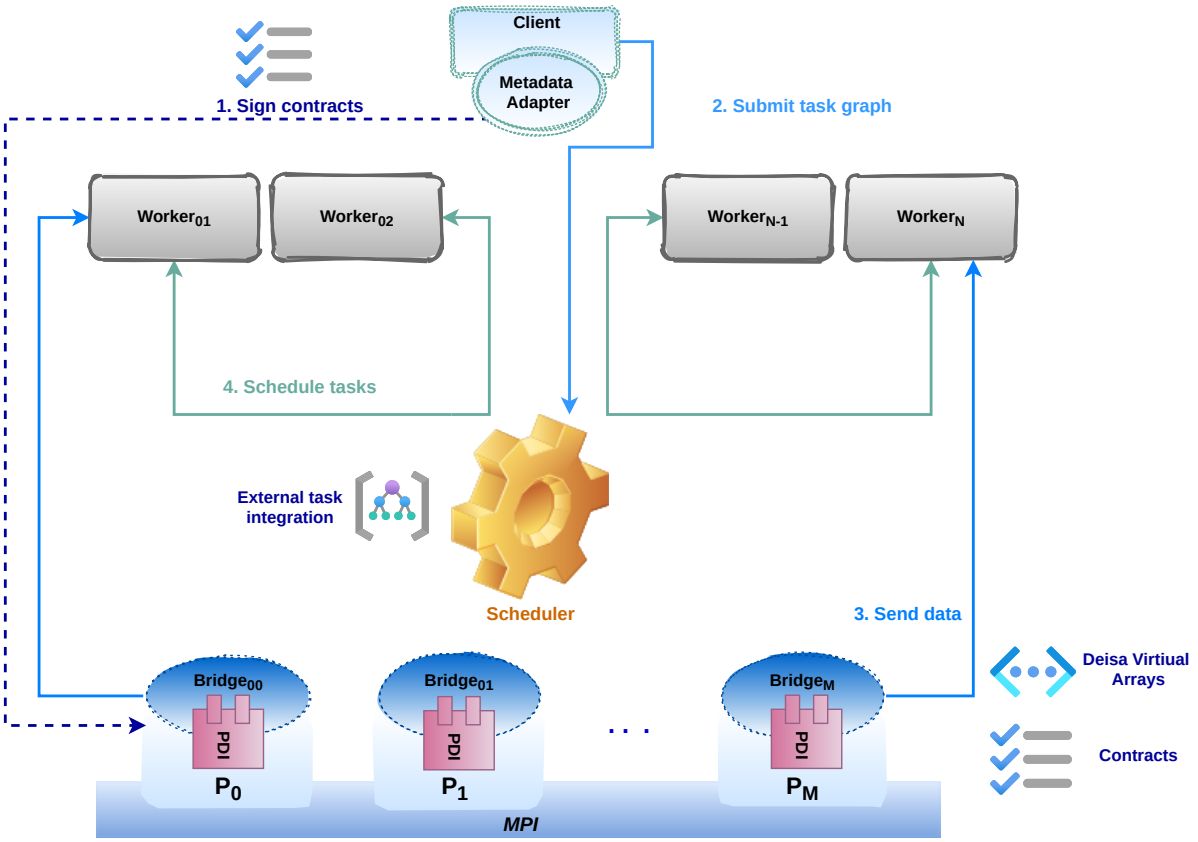


Figure 6.1: New DEISA architecture.

Figure 6.2 shows a simplified diagram of the pseudo-algorithm of the `scatter` method. The keys are created and associated with the data in a dictionary format. If the `Direct` option is activated, then the data is sent directly to the worker, which is relevant when large data is sent. If this option is not enabled, the data is sent to the scheduler first and then to the workers via `scatter_to_worker` function. In the case of direct communication, the `scatter_to_worker` function does not inform the scheduler that the current client desires this data. This is done by calling the `scheduler.update_data` that ensures relating the data identified by its key to the client that called the `scatter`. Once this is done, a `Future` object with the `finished` state is created.

Note that `scatter` is a complex process that involves the three entities in Dask distributed, namely the client, the scheduler and the worker. The `scatter_to_worker` function trigger a remote procedure call that calls the `worker.update_data` function that updates the internal data of a worker without informing the scheduler.

If the worker informs the scheduler that new data is available in its memory, but this data is not associated with any client in the scheduler's data structures, then it will be deleted by the garbage collector. To prevent this scenario from happening, the worker only updates its internal data structures to keep track of the data in its memory but does not inform the scheduler. Instead, the client initiates a new RPC to the scheduler to call `scheduler.update_data`, which will populate the internal data structures of the scheduler with the new data keys, associates them with the workers where they live (`who_has` mapping) and the client who wants them (`wants_what` mapping). This way, the data is not deleted since there is at least one client that desires it.

Finally, once this is done, a local `Future` is created with the same `key`, and returned to the client in the `"finished"` status.

6.3.1.2 DEISA Key Generation System

Unlike the multi-graph version where the `dask.arrays` are built on chunks whose keys are generated by Dask itself (that are returned by the `scatter` function). In this version, we created and used specific keys to identify our data.

Each key contains three main sections: the prefix `deisa`, the name of the data, and the position of the

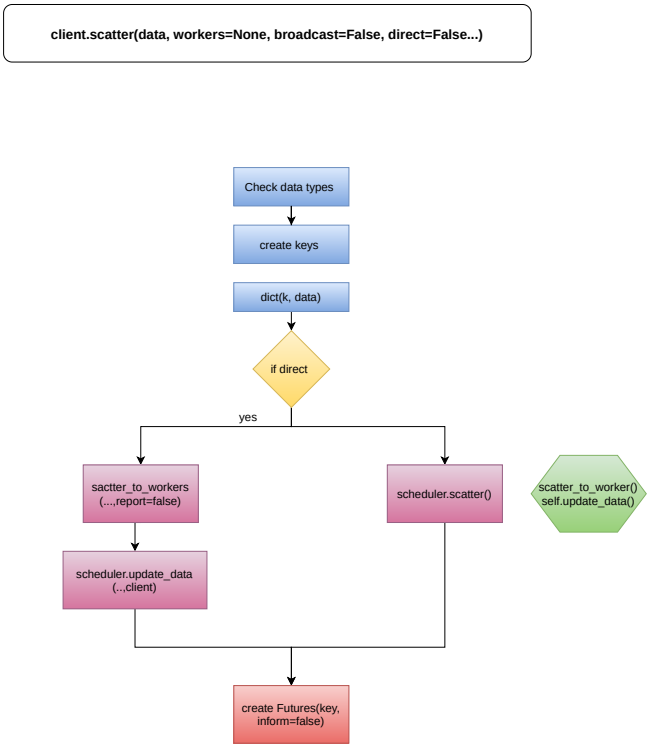


Figure 6.2: Simplified pseudo-algorithm of the Dask `scatter` method. Constructed diagram from code in [41].

chunk is the global array considering both time and space decomposition. For example, in `(deisa-temp, (1,3,5))`, `temp` is the name of the data, $(1, 3, 5)$ is its position in the global array. This naming scheme can be extended to support multiple simulations as producers of data, feeding the same Dask cluster, by adding an identifier for the simulation, for instance.

To avoid exchanging metadata between the two components at each timestep, and still be able to identify correctly the data generated by the simulation in the Dask component, we set up a protocol with minimum communications. We use the same domain decomposition in both components by sending to Dask all needed information for that namely: the name of the generated array, its sizes and sub-sizes in all dimensions. On the Dask side, one key per block is created, associated with an array having the corresponding sizes, and identified as already explained. From the MPI component side, every time a process has to send a block of data, it creates a key using the same identification scheme. The only requirement is that each process should know the position of the block it generates.

6.3.1.3 DEISA Virtual Arrays

A DEISA virtual array describes the spatiotemporal domain decomposition of a generated data array. It contains the global sizes in each dimension, including the time dimension, the size of each block (size of generated data by each MPI process), and the starting indexes of each block. Describing the data in this way allows us to have a global view of the generated data. DEISA virtual arrays are used to create the `dask.arrays` in the analytics client, based on the *external* task concept.

A `dask.array` that is created from a DEISA array descriptor contains only external data. We create a `"deisa"` 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 *Future* to create a `dask.array`, then gathering all of the arrays to create a global `dask.array`. The chunking of this last array corresponds to the spatiotemporal domain decomposition of the DEISA array.

Figure 6.3 shows an example of a DEISA virtual array construction, where we consider 2 main dimensions: space and time. Each MPI process generates a block per timestep, which is translated to a DEISA task and an external task too.

The DEISA virtual arrays have been added to the configuration file of the DEISA plugin. Listing 6.1 shows an example of a configuration file of the DEISA plugin. In line 15 starts the list of the DEISA virtual arrays. For instance, here we have only the `Gtemp` array that is constructed of blocks of the `temp` data.

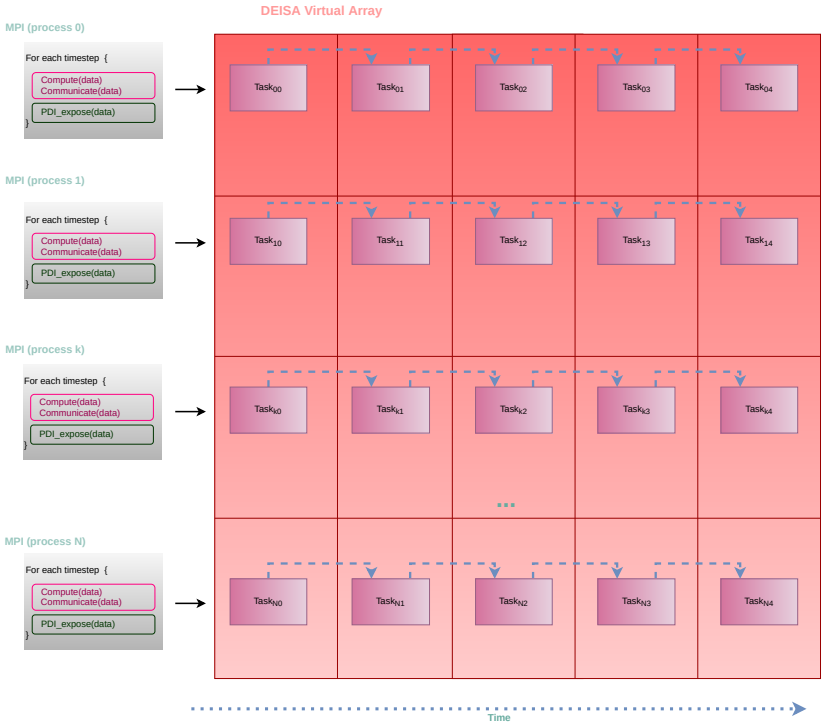


Figure 6.3: DEISA virtual arrays structure.

```yaml
pdi:
    types: #[...] including config_t description
    metadata: {step: int, cfg: config_t, rank: int}
    data:
        temp: # the main temperature field
            type: array
            subtype: double
            size: [ '$cfg.loc[0]', '$cfg.loc[1]' ]
    plugins:
        mpi: # get MPI rand and size
        deisa:
            scheduler_info: scheduler.json
            init_on: init
            time_step: $step
            deisa_arrays: # Deisa Virtual arrays
                G_temp: # Field name
                    type: array
                    subtype: double
                    size:
                        -'$cfg.maxTimeStep'
                        -'$cfg.loc[0] * ($rank % $cfg.proc[0])'
                        -'$cfg.loc[1] * ($rank / $cfg.proc[0])'
                    subsize: # Chunk size
                        -1
                        -'$cfg.loc[0]'
                        -'$cfg.loc[1]'
                    start:  # Chunk start
                        -$step
                        -'$cfg.loc[0] * ($rank % $cfg.proc[0])'
                        -'$cfg.loc[1] * ($rank / $cfg.proc[0])'
                    +timedim: 0 # A tag for the time dimension
            map_in: # Deisa array mapping
                temp: G_temp
```

Listing 6.1: Data description in PDI DEISA YAML file.

6.3.1.4 Contracts

A contract is concluded between the simulation and Dask at the beginning of the simulation. The bridge in MPI process 0 builds the DEISA virtual arrays descriptors using data from the simulation and sends them to the adaptor in Dask. The analytics client in Dask gets access to the equivalent `dask.array`, selects the data it is interested in from the available arrays, and sends back a selection request to the simulation bridge identifying the data it is actually interested in.

The contracts in the simulation side are checked every time data is available in an MPI process; if this block is included in the selection needed by the analytics, then the associated *key* is created, and the data (identified by this created *key*) is sent to a pre-selected worker. On Dask side, the contracts are used at the beginning to create the corresponding *dask.array*s to the DEISA virtual arrays. Similarly, *keys* are created the same way in the adaptor. Those *keys* are used to identify the chunks of the `dask.array` associated with that data. This way, only the description of the arrays sent in the contract process is performed; no need to send more metadata at each timestep. The contact process is summarized in Figure 6.4.

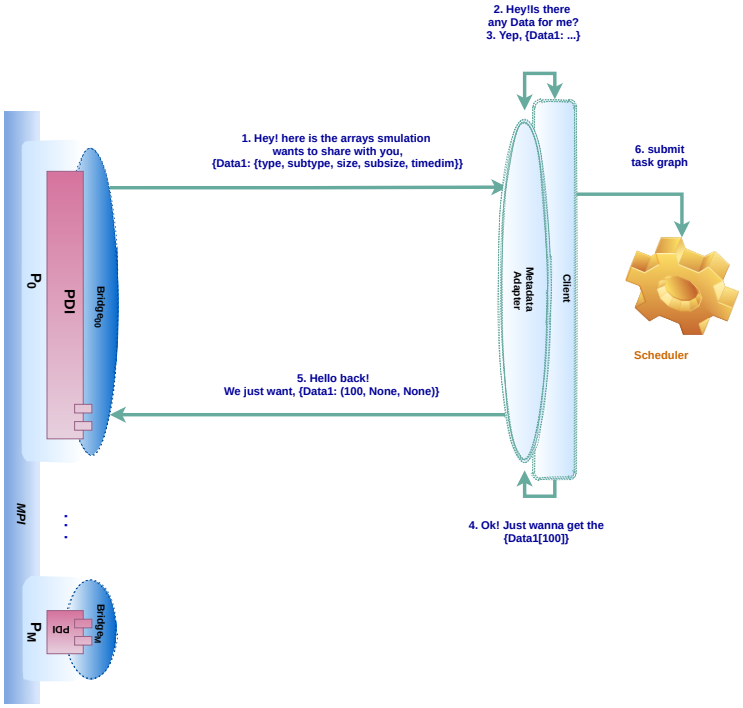


Figure 6.4: The Contracts operation that is done at the initialization step. Only rank 0 performs the contracts with the analytics client. Once the contacts are signed by the analytics client they are shared with all the bridges as metadata where they are saved locally. At each time step, each bridge checks if its data is included in the selection mentioned in the contracts, if so it sends its data to the workers, else it returns.

6.3.2 External Task and Asynchronous Scheduling

In this section, we detail the introduction of external tasks to Dask, and the asynchronous scheduling operation of those tasks.

6.3.2.1 External Tasks in Dask Distributed

Dask offers low and high-level libraries to create and submit tasks to the scheduler. The created tasks must be known and correctly defined to be used by Dask. Except for pure data tasks, a task has to include a callable (the function that the worker will execute). In this work, we want to use Dask for in situ analytics, so we suppose that the data is generated in another application, a running MPI simulation in this case, and we want to use that data as input to a Dask task graph.

We define an *external* task as tasks that run in an external environment rather than in Dask. 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 the first implementation of DEISA, we have used the pure data task concept (via `scatter`) to integrate external tasks in Dask graphs. However, by using it, we can only submit computations to Dask once data is sent to Dask via a `scatter`. Here we push this solution further to make Dask natively support external tasks, which makes us able to submit graphs on those external pure data tasks in advance (before data is available in the workers' memory).

We extended the Dask distributed task states with the new *external* task concept. We added a new task state called `"deisa"`. The task in a `"deisa"` state is identified by a unique key, and it is not schedulable nor runnable by Dask. Figure 6.5 shows the newly added `"deisa"` state and the two main corresponding transitions: to `"memory"` when the data with the same key becomes available in a worker's memory or to `"released"` when it needs to be deleted.

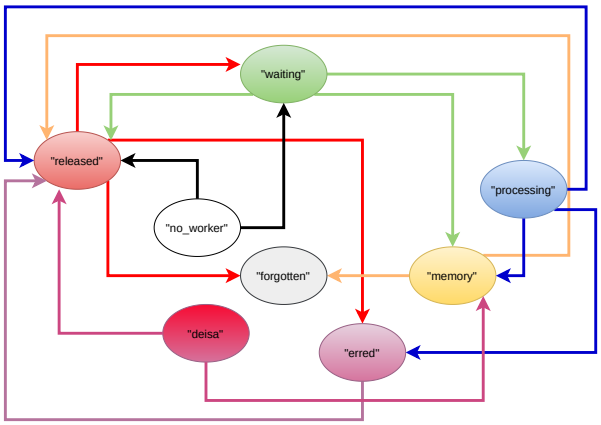


Figure 6.5: Dask task states and transitions with the newly added 'deisa' state.

The implementation of external tasks in Dask distributed mainly modify the client and the scheduler classes. The `Future` class in the client can be seen as a mirror of the tasks in the scheduler. In other words, the *Future* is an interface that can be used by the client to check the state of the task in Dask scheduler and can retrieve their results as well. Figure 6.6 shows `FutureStates` in the client and the corresponding `TaskStates` in the scheduler. In short, a `pending` *Future* may be a task in one of these states `"released", "retry", "waiting", "processing", "no-worker", "deisa"`, and so on. Thus to create an external task (a task in a *"deisa"* state), we need to create a `Future` by specifying a unique *key* and activating the DEISA mode by setting the `"deisa"` argument to *True*. This will trigger a RPC to the scheduler to create a `"deisa"` task.

Figure 6.7 shows a typical example of the task state transition of an external task. We consider in this figure one bridge that represents the external application, a Dask scheduler, a client and a worker. We do not represent all RPC between the different components. Our goal is to show that we are representing the exact same task in different places. We identify it by its unique key: `key1`. First, we create a *Future* in the client, which is identified by `key1` and enabling DEISA mode, the creation of the *Future* triggers the creation of a task in the `"deisa"` state in the scheduler. At some point, the data that is identified by the same `key1` is sent to worker1, so the state of this date becomes `"memory"` in worker1, then the `"deisa"` task in the scheduler switches to the `"memory"` state as well. Once this is updated in the scheduler, all clients and bridges desiring the `key1` are informed that the task is finished, and thus the *Future* switches to *finished* state.

This example will be explained in the next section, with all the remote procedure calls. Here the goal is to show the mapping between a task in the scheduler, a pure data task in the worker and the `Future` on the client side.

6.3.2.2 Asynchronous Scheduling of External Task in Dask

The goal of updating the default scatter system is to make Dask distributed support asynchronous scheduling. In other words, submit a task graph to Dask, that contains *external* tasks.

Figure 6.8 represents the updated version of `scatter`. We have added two main parameters to the method: `keys`, and `deisa`. Both of them are `None` by default. `keys` is a list of keys associated with a list

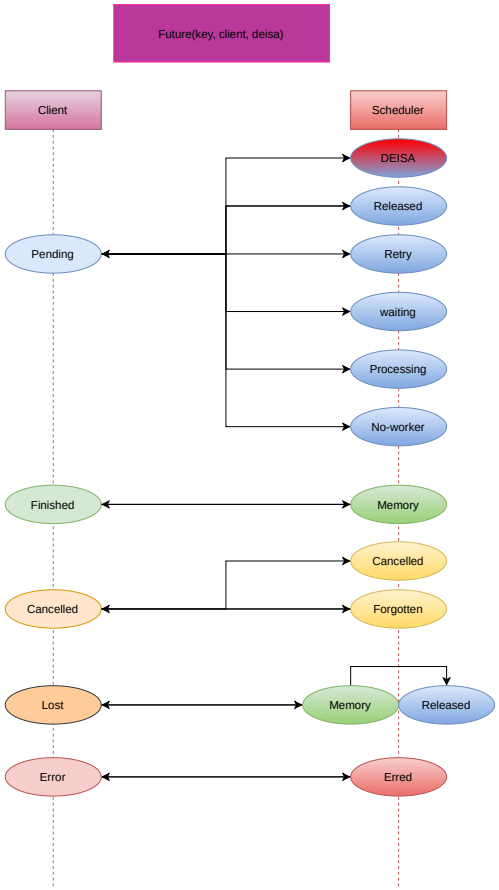


Figure 6.6: Futures states and corresponding task states in the scheduler.

of data we want to communicate. In our case, it is always a list of one element. `deisa` is an argument that is forwarded to the `scheduler.update_data` and `Future.__init__` methods that changes the default operation when `deisa` mode is activated.

The `Future.__init__` with `deisa` mode here does not matter because the status of the *Future* is turned to a `'finished'` status before the client returns from the `scatter`. What really matters is the `scheduler.update_data` method that needed to be changed. To understand the need for the made modifications in this method, we have to explain two main points.

First, let's recall how Dask scheduler manages finished tasks. When data is an output of a finished task in Dask, the worker sends a message to the scheduler, including, among other information, the key of the task and *"task-finished"* stimulus. Depending on the stimulus, the scheduler triggers different handlers, and in this particular case, it is `handle_task_finished` that is called. This handler triggers the transition process(see Section 3.2.4), which unblocks the dependent tasks, and the scheduling keeps going.

Now let's go back to the `scatter`. It has been introduced in Dask to send external data to the cluster. So by definition, this data does not exist in Dask before it is sent. The associated `key` with this data is created in the `scatter` function itself (as shown in the diagram in Figure 6.8), so this data can only be used in a task graph after the `scatter` is finished and returned. The way the scheduler manages the data it gets from a `scatter` is quite 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.

As shown in the diagram in Figure 6.8, to inform the scheduler about this new data, the method `scheduler.update_data` is called. The left part in Figure 6.9 shows a simplified pseudo-algorithm of the `scheduler.update_data`; mainly, only the scheduler's internal data structures are updated. So there is no transition process done in `scheduler.update_data`. This is a normal operation because we suppose that the data is not known before it is sent to Dask workers.

To support external tasks in Dask, we have to make it treat them as any other finished task. This means that Dask will not only update its internal data structures but it will also trigger the transition process to unblock the depending tasks. And that is mainly what has been done in the right part

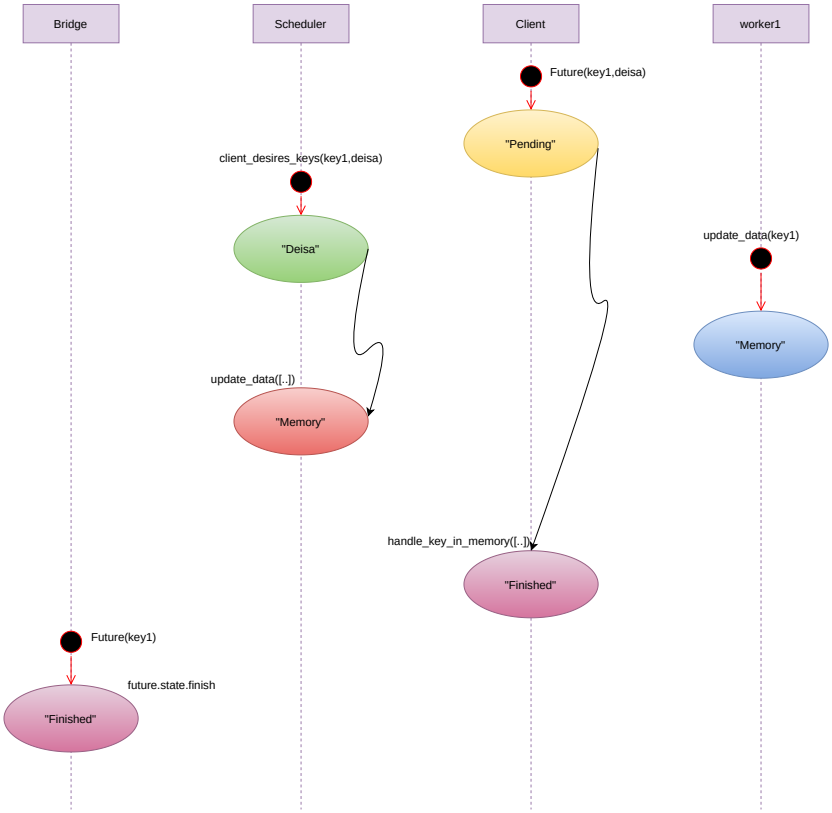


Figure 6.7: External task state scenario diagram.

of Figure 6.9. When DEISA mode is activated, we update the scheduler's internal data structures and trigger the transition process: starting by transiting the current task to `"memory"` state, then making all underlying transitions.

6.3.2.3 Full External Task Scenario

Now that we have explained how to support external tasks in Dask, here is a complete example (Figure 6.10) that shows and explains the example in Section 6.3.2.1.

In this diagram, there are two independent sequences. We suppose here that the top sequence is executed first, so we have only drawn methods called in that scenario. However, if the second sequence is launched first, then this workflow works but in a different scenario.

When the client creates the *Future* with `key1` and `deisa=True` as arguments, an RPC is done: the client sends a message to the scheduler saying that it desires the results of the task `key1` (or the data `key1` if it is a pure data task). The scheduler handles this message internally, creates a task in the `"deisa"` state because DEISA mode was enabled, and updates its internal data structures.

The second part of the diagram starts when the bridge calls the modified `scatter` method, with DEISA mode activated and using the same key: `key1` and specifying the worker `worker1`. The bridge calls `worker.update_data` (using the same key). This call makes the worker update its internal data and internal data structure to make `key1` to the `"memory"` in the worker1. Then returns some metadata to the bridge regarding the size and mapping of the `key1` to the *worker1*.

Once this is received in the bridge, it triggers an RPC again, and this time to the scheduler to update its data with the information it got from the worker and by activating the **deisa mode**. Now that the `scheduler.update_data` is modified to support external tasks, the scheduler updates its internal data structures and starts the transition process. At the end of this process, it sends messages to concerned clients and workers. For the client and the bridge, they are informed that the `key1` is now in memory, so they turn their *Futures* to the `"finished"` status, and the worker1 will likely run dependent tasks on `key1`.

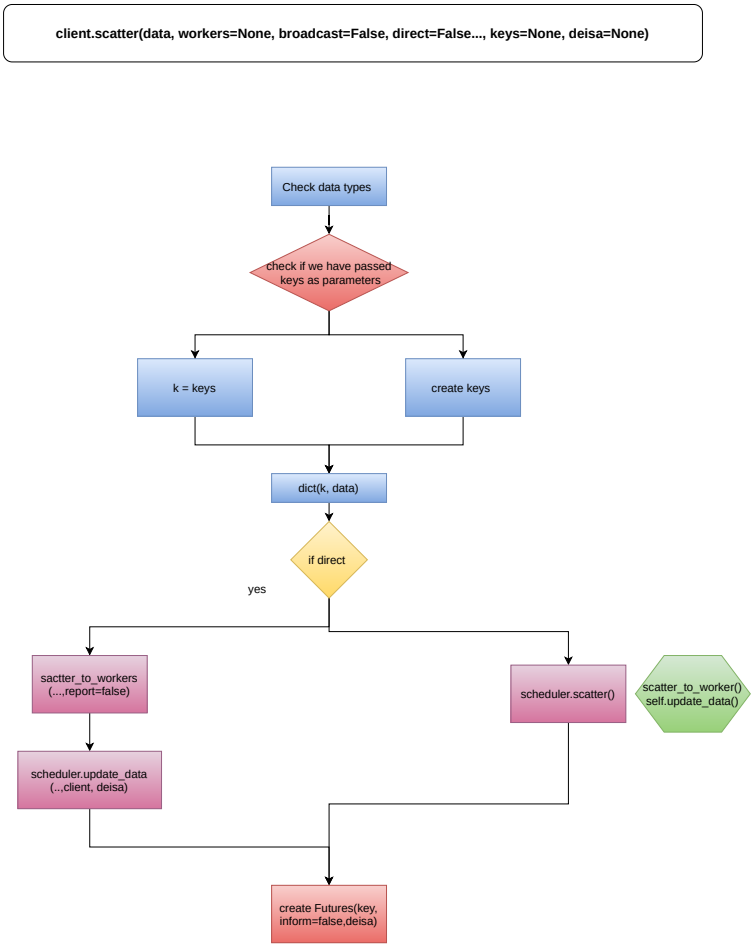


Figure 6.8: New `scatter` pseudo-algorithm diagram. Diagram reconstructed from the code in [41].

6.3.3 User API and Configuration

In this section, we present the new user API in DEISA and PDI specification tree to support the newly added concepts.

6.3.3.1 DEISA Plugin Configuration

We implemented a DEISA plugin to support the new functionalities. The plugin responds to both data sharing and PDI events. The specification tree of the plugin contains 5 main keywords:

- `scheduler_info`: takes a string that represents the path to the `json` scheduler file that contains connection information. When the scheduler is launched, it generates this file with the name passed to the option `--scheduler-file`.
- An `init_on` initialization event to gather all the needed metadata data and send them to the DEISA adaptor. The needed metadata have to be shared by the simulation before the `init_on` event is issued. At the `init_on` event, the contracts are signed between the simulation and Dask (Section 6.3.1.4).
- The *time_step* variable that represents the iterator over the time dimension. It is needed to create the unique key to the associated data.
- A `deisa_arrays` section that describes the global data arrays following time and space distributions, independently from local data name;
- a `map_in` section maps local buffer's name (defined in the `data` section of PDI) to the `deisa_arrays`'s name (Section 6.3.1.3).

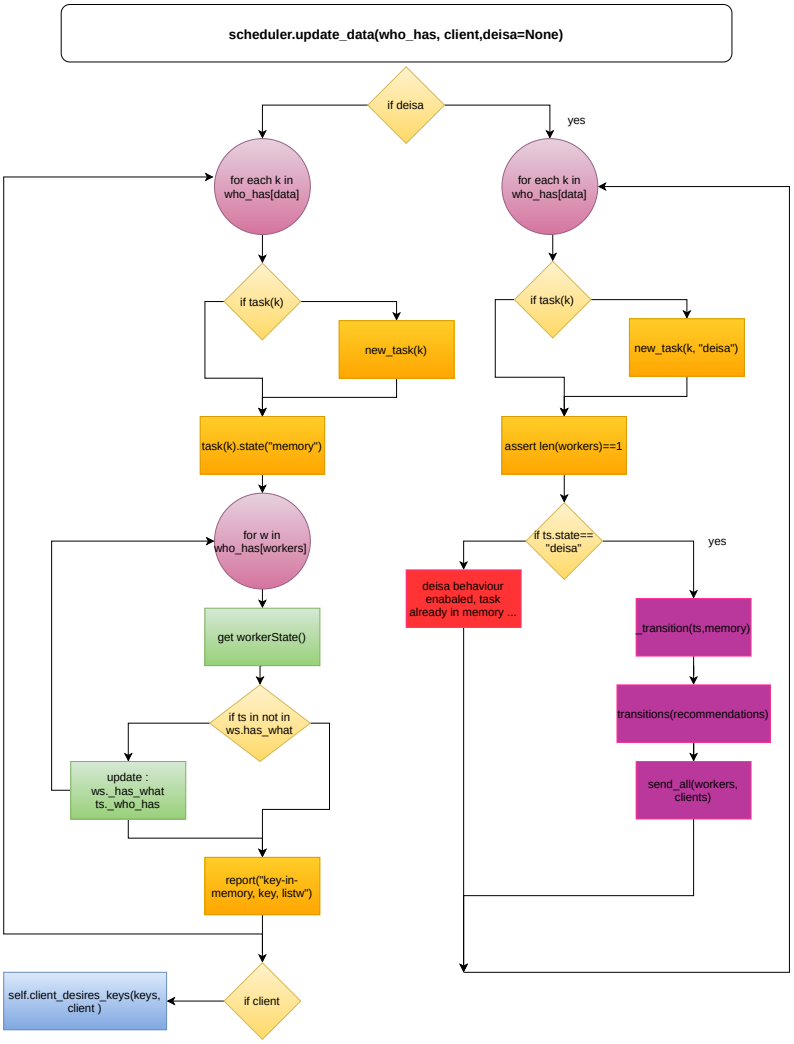


Figure 6.9: Scheduler update_data method. Diagram reconstructed from the code in [41].

6.3.3.2 User API

The bridge's user API is now hidden with the new declarative interface of the specification tree of the DEISA plugin. The new interface is easier and hides all implementation details.

We have kept recalling until now that the main goal of this work is to bring post hoc easiness to the in situ environment. Thus the user API has to be as similar as possible to the post hoc one or at least comparable. The user API in the new version of DEISA consists of four main functions:

- `Deisa(scheduler_file, nbr_workers)` returns a DEISA Adaptor object, the scheduler_file is the file returned by the scheduler when it is launched(see Section 5.3.1), and nbr_workers is the number of workers that need to be connected.
- `Adaptor.get_client()` returns a Dask client, it can be used to submit analytics to the scheduler and use all available Client API in Dask distributed.
- `Adaptor.get_deisa_arrays()` returns a dictionary where the keys are strings representing the names of available data, and the values are DEISA arrays. The square brackets operator can be used to select a DEISA array, and a square brackets operator is also implemented to make a selection in the DEISA array to get a `dask.array`. If all the array is needed, the Ellipsis (`...`) can be used to select all the array.
- `Deisa_arrays.validate_contracts()` to validate the selection that we did and ask for that data (Section 6.3.1.4).

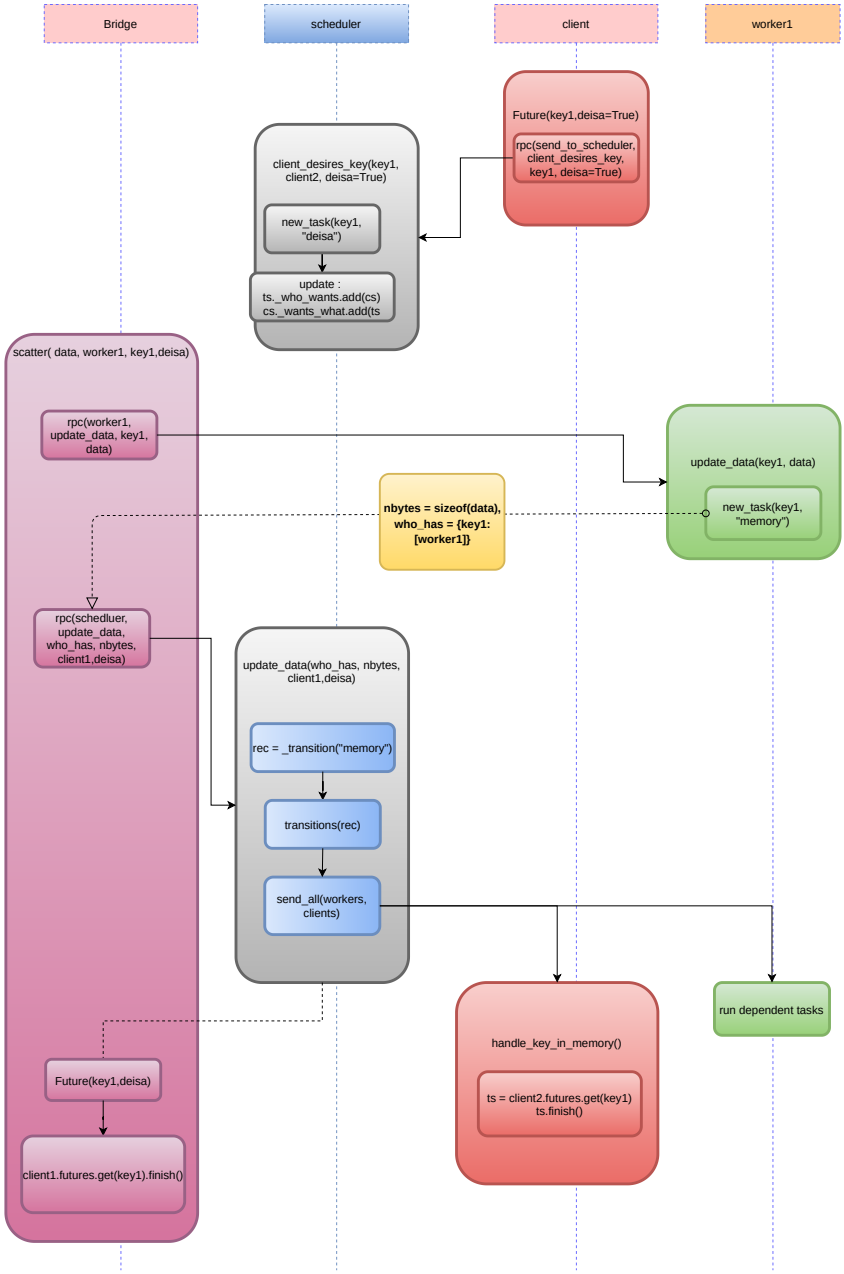


Figure 6.10: Activity diagram of a typical external tasks scenario.

6.4 Experiments and Evaluation

In this section, we present the experiments we have performed to evaluate our new prototype. First, we present our experimental environment in the Irene supercomputer. Then talk about the software we have used in the experiments, namely a new version of the incremental PCA and a temporal derivative. We show our comparisons to the previous version of DEISA and the post hoc experiments. We will discuss those results and explain eventual similarities and differences between the different versions.

6.4.1 Environment Installation

To ensure reproducibility, we have spack to install our environment with all needed dependencies instead of relying on the available module systems in the supercomputers. In some supercomputers like Irene, users do not have access to the internet, so we had to pre-fetch needed packages in a machine with internet access, then create a mirror and send it to Irene using `rsync`. Once this is done, you can use the local mirror in the supercomputer to install the packages. The configuration file in Listing 6.3 shows an

```python
from deisa import Deisa

# Derivative function
def Derivative(F, dt):
    """
    First Derivative
       Input: F        = function to be derivate
              dt       = step of the variable for derivative
       Output: dFdt = first derivative of F
    """
    c0 = 2. / 3.
    dFdt = c0 / dx * (F[3: - 1] - F[1: - 3] - (F[4:] - F[:- 4]) / 8.)
    return dFdt

# Scheduler file name and configuration file
scheduler_info = 'scheduler.json'

# Initialize Deisa
Deisa = Deisa(scheduler_info, nbr\_workers)

# Get client
client = Deisa.get_client()

# Get available deisa arrays
arrays = Deisa.get_deisa_arrays()

# Select data: 1/2 timesteps
gt = arrays["global_t"][::2]

# Construct a lazy task graph
cpt = derivative(gt, 1)

# Submit the task graph to the scheduler
s = client.persist(cpt)

# Sign contract
arrays.validate_contract()

print(client.compute(s).result(), flush=True)
client.shutdown()
```

Listing 6.2: In situ incremental temporal derivative.

example of a configuration file to create a Spack environment.

We also provide a Spack recipe for the cloned version of Dask distributed that supports the external tasks in Dask, and a recipe for DEISA plugin that can be found on GitHub repository[1]. DEISA Python API can be installed through Pypi[2].

6.4.2 Software

We used the same simulation HeatPDE MiniApp to evaluate the new version of DEISA, with two different analytics, an in situ version of the incremental PCA and a temporal derivative of the generated data.

6.4.2.1 In Situ Incremental PCA

We implemented a new version of the Incremental PCA [1], which takes a multidimensional array and computes its PCA incrementally. Thus it can be used for both post hoc and in situ. Moreover, we have provided a similar interface to the sequential PCA by hiding the incremental execution of the IPCA.

In addition to the multidimensional array, the `fit(ndarray, label_list, feature_labels, sample_labels)` method takes three new parameters:

- ndarray: array-like or sparse matrix that will be chunked to N chunks in the first dimension, where N is len(ndarray). We suppose that dimension zero is the time dimension.

[1] https://github.com/pdidev/spack
[2] https://pypi.org/project/deisa/

```yaml
spack:
  concretization: together
  specs:
  - netlib-lapack, pdi, pdiplugin-decl-hdf5, pdiplugin-deisa, py-dask+diagnostics py-h5py, pdiplugin-mpi, pdiplugin-pycall
  view: true
  packages:
    all:
      buildable: true
      permissions:
        write: group
        group: group001
      compiler:
      - gcc
      providers:
        mpi:
        - openmpi
    openmpi:
      buildable: false
      externals:
      - prefix: /path/to/products/openmpi-4.0.3/gcc--9.3.0/default/
        spec: openmpi@4.0.3%gcc@9.3.0+cuda+cxx~cxx_exceptions~java+lustre~memchecker+pmi+pmix~sqlite3~static~thread_multiple~wrapper-rpath
          fabrics=ucx schedulers=slurm
        modules: [gnu/9.0.3, openmpi/4.0.3]
  repos:
  - /path/to/spack/var/spack/repos/pdi
  mirrors:
    mirror-gysela-deisa:
      fetch:
        url: file:///path/to/mirror-gysela-deisa
```

Listing 6.3: Spack environment installation configuration file.

- `label_list`: a list of N strings that represents the labels of the N dimensions of the ndarray. It is used to create an `xarray`.
- `feature_labels`: a list of X strings, included in `label_list`, that represent the dimensions we want to consider as features. They are stacked into the features' dimension.
- `sample_labels`: a list of Y strings, included in `label_list`, that represent the dimensions we want to consider as samples, where $X + Y = N$. They are stacked into the samples' dimension.

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_transform(ndarray, label_list, feature_labels, sample_labels)` method takes the same parameters.

6.4.2.2 Incremental Time Derivative

Computing the derivative in post hoc is memory-consuming because we need several timesteps at once to compute a derivative at a given timestep. In this work, we are interested in the time derivative in particular because it demonstrates the variation over time. Since the simulations are discrete, to approximate the derivative, we use the finite difference [94].

Thanks to *dask.array* API and the new version of DEISA, an incremental in situ time derivative, is a one-line stencil computation (see Listing 6.2). The user does not need to manage memory or the incremental nature of the algorithm. Dask launches the tasks when the data is available.

6.4.3 Performance Evaluation

We evaluate the new version of DEISA compared to the previous prototype and post hoc analytics. We use the new interface of the IPCA for both DEISA and post hoc analytics and show the importance of the contract through the incremental time derivative. We used the chunking in HDF5 to improve post hoc performance in those experiments. In the different figures, the results for the previous DEISA prototype

are referenced with DEISA1, the results for the new version of DEISA without contracts and a heartbeat interval set to 1 min with DEISA2, and the full new version with a ∞ heartbeat interval is DEISA3.

Parameter	Value
Number of runs	3
Number of iterations IPCA	10
Number of iteration Derivative	12
MPI nodes / Dask worker node	2
MPI process / MPI node	2
Dask worker / Dask worker node	2
Thread / Dask worker	24
MPI process / Dask worker	2

Table 6.1: Fixed parameters used in Experiment III 6.4.3.1 and 6.4.3.2.

Configuration	XP1:128 MiB	XP1:256 MiB	XP1:512 MiB	XP1:1 GiB
MPI block size	128	256	512	1
Dask chunk size	128	256	512	1
MPI Nodes	[4, 8, 16, 32, 64, 128, 256]			
Dask Nodes	[2, 4, 8, 16, 32, 64, 128]			

Table 6.2: The three configurations of Experiment 6.4.3.1 and 6.4.3.2.

We have performed two main experiments with the parameters and configuration in Table 6.1 and Table 6.2:

- **Experiment III** compares the performance of DEISA with and without contracts (DEISA3 vs DEISA2) and the old version (DEISA1)
- **Experiment IV** compares the new version of DEISA (DEISA3) performance to the old version (DEISA1) and to parallel post hoc analysis with plain Dask (DASK) using the old version of the Incremental PCA presented in Section 5.3.3, and using the newly developed version presented in Section 6.4.2.1.

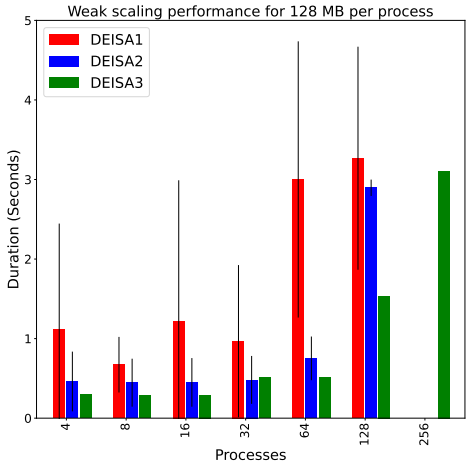
6.4.3.1 Experiment III

Those experiments have been performed on the Irene supercomputer. We used the heat equation solver MiniApp for the three implementations of DEISA. Table 6.1 and Table 6.2 show the parameters and configurations used in these experiments. In those experiments, we were just interested in the communication time. To show the importance of contracts, compute the analysis only every two timesteps.

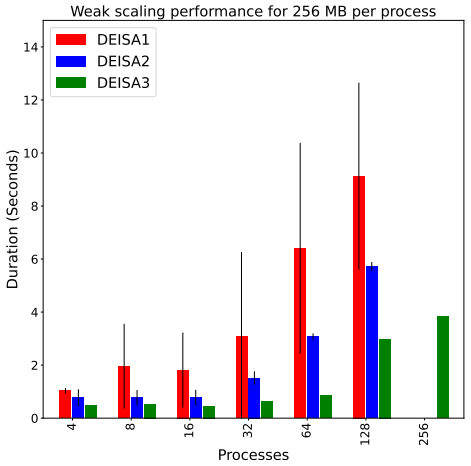
Figure 6.11 shows the weak scaling results for the different configurations of **Experiment III**. In the subfigure in the top left (Subfigure 6.11a), we have fixed the size of the data per MPI process to 128 MiB, to 256 MiB in the subfigure in the top right (Subfigure 6.11b), to 512 MiB in the subfigure in the bottom left (Subfigure 6.11c) and to 1 GiB in the bottom right (Subfigure 6.11d). The x-axis of each subfigure represents the variation of the processes for 4 to 128 for DEISA1 and DEISA2 and for 256 for DEISA3, and y-axis represents the duration in seconds of communication time. In each subfigure, we have the three cases of Experiment III. The first bar from the left of each scale is the communication time for DEISA1. The bar in the middle shows results for DEISA2 (the new version without contracts and the heartbeat interval set to one minute instead of 5s) in blue. The last bar of each scale shows results for the communication time of the new version of DEISA with contracts activated and the heartbeat interval set to ∞.

The represented values are the maximum value per iteration averaged over ranks and runs. The error bars represent the standard deviation. The standard deviation is not represented for DEISA3, as we do only send the data once every two iterations; thus, it is not representative of the real variability.

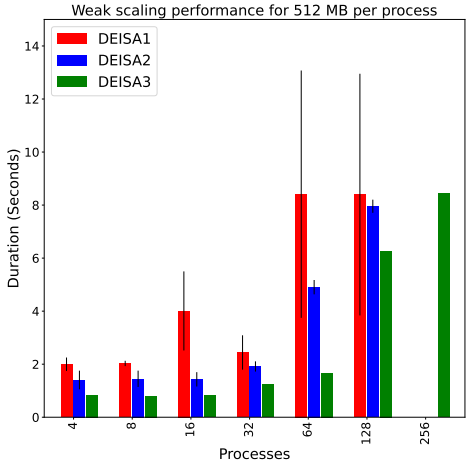
We expect DEISA1 to have the worst performance with more variability (because of the number of metadata we send at each timestep, and the heartbeat interval). DEISA2 should be less variable than DEISA1. DEISA3 should perform twice better than DEISA2 and be less variable. Since we have activated the contracts for DEISA3 (Listing 6.2). It prevents the bridges from sending the data to the workers if it is not needed in the analytics.



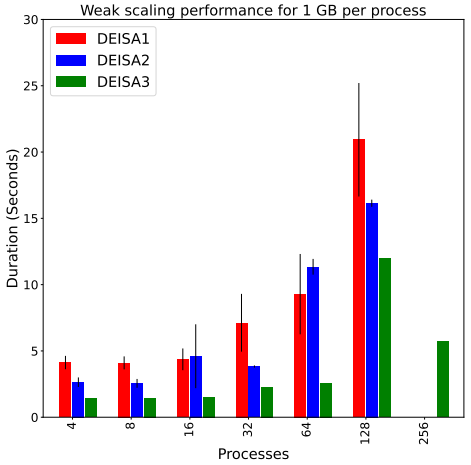
(a) Average communication times per iteration for 128 MiB per process



(b) Average communication times per iteration for 256 MiB per process



(c) Average communication times per iteration for 512 MiB per process



(d) Average communication times per iteration for 1 GiB per process

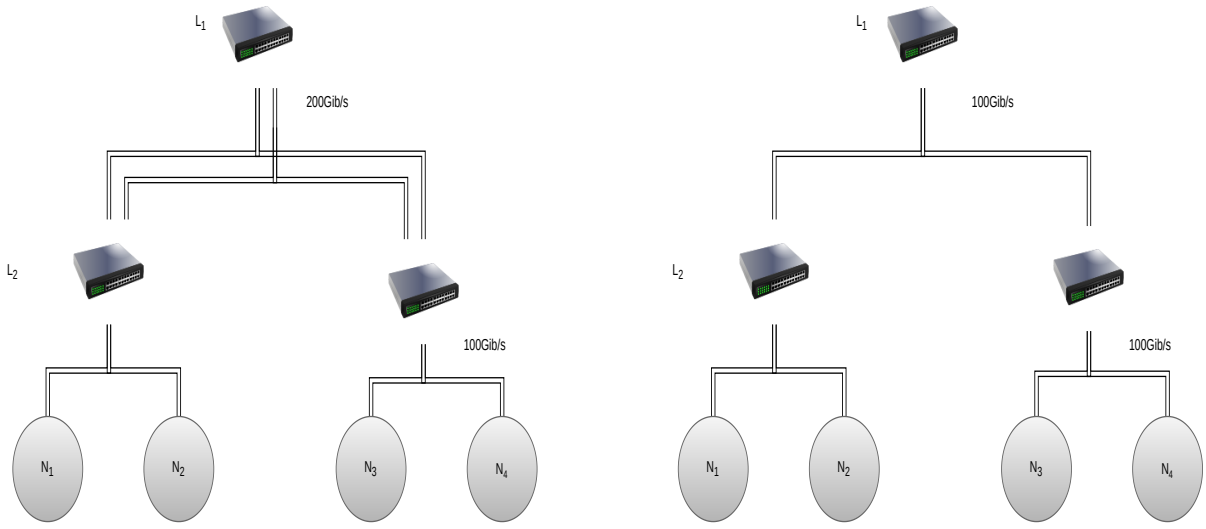
Figure 6.11: Weak scaling average communication for 128 MiB, 256 MiB, 512 MiB and 1 GiB per process per iteration for three experiments: the first bar from the left of each scale represents the communication time for the old version of DEISA (in red), the second shows the results for DEISA without contracts (in blue), and the third bar represents results for DEISA full options (in green).

Overall, our expectations are true for almost all cases. DEISA1 presents more variability than DEISA2. For all cases, DEISA2 is better than DEISA1 except in Subfigure 6.11d, when the number of processes is 64. Since we do not observe a big variability, this may be due to the node allocation of this experiment. We will have a look at this in detail in the last part of this discussion. DEISA3 also shows a strange duration when the number of processes is 128.

The three versions weak-scale almost perfectly until 64 processes. Then we start seeing an unpredictable variation in the duration for DEISA2 and DEISA3.

This may be due 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. To simplify the topology, we suppose that we have four nodes: N_1, N_2, N_3, N_4. Every 2 nodes are connected to a switch L_2 where we have N_1 with N_2 and N_3 with N_4, then the two switches L_2 are connected to a L_1 switch. Figure 6.12 shows the difference between a fat tree topology and a pruned fat tree topology which is used the topology of the Skylake partition we are using. The fat tree topology has 200 Gib/s in the switch L_1; this maintains the 100 Gib over all the nodes. However, in the pruned fat tree topology, a 100 Gib/s link has been used, which makes communications between N_1 and $[N_3, N_4]$ or N_2 and $[N_3, N_4]$ potentially longer, and vice versa.



(a) Fat Tree topology (b) Pruned Fat Tree topology

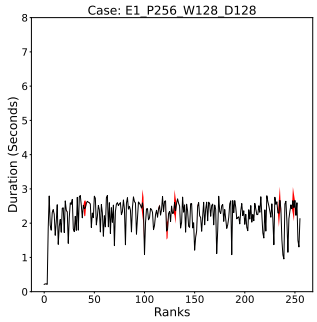
Figure 6.12: Fat Tree versus Pruned Fat Tree topology.

If the scheduler, which is always in the first node of our allocation, is connected to a different switch than 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.

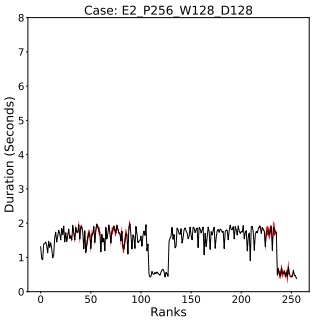
We investigated this variability by checking the mean duration of the communications per rank for DEISA3 and show the results in Figure 6.13. We fixed the number of processes to 256 and varied the size of the data per process: 128 MiB, 256 MiB, 512MiB and 1 GiB. We separated the results we got from each run. Each line in the subfigures corresponds to a fixed size of data. We submitted the runs independently, so we do not have control over the allocated nodes, but we may get the same allocation multiple times due to 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 (the black line). The standard deviation over iterations is represented as a red band. First of all, we notice that there is variability over the 3 runs for specific data size, but overall we do not really notice the red band, so there is minimal variability over iteration. In some subfigures, we have the same pattern of variability (for instance in Subfigures 6.13b and 6.13c, Subfigures 6.13e and 6.13e, Subfigures 6.13h and 6.13h, Subfigures 6.13k and 6.13k). This makes us think that they may have the same allocations of at least nodes connected to the same switches. We also notice the same patterns even when the size of the data changes. For instance, Subfigures 6.13h, 6.13i, 6.13j, 6.13k and 6.13l have all of them the same variability pattern. We have checked the logs and found that all of the four experiments in Subfigure 6.13h, Subfigure 6.13i, Subfigure 6.13k and Subfigure 6.13l have the exact same allocation, thus the similarity in the variations. Subfigure 6.13j allocation differs by two nodes compared to others. We had a look at the topology of the nodes in the Skylake partition (that we can not share externally) and found that the nodes of the five 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, 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. So here, in this particular 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.

We were also interested in the behaviour of DEISA2 and DEISA1, the variability over ranks, iterations and runs, which we represent in Figure 6.14 and Figure 6.15 respectively. Our expectations are much more related to the variability over iterations, which will be more visible in DEISA1. Remember that in DEISA2, we have fixed the heartbeat interval of the bridges to one minute, and in DEISA1, we have kept the value by default which was 5 seconds. This frequency, alongside the frequent metadata sent to the scheduler, may cause more variability per iteration because of the load in the scheduler in DEISA1.

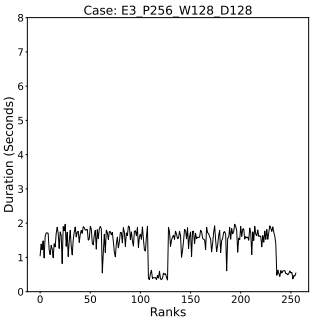
Indeed the red band, which represents the standard deviation per iteration, is more visible in DEISA1 experiments than in DEISA2 and almost absent in DEISA3. This is thanks to the improvements over



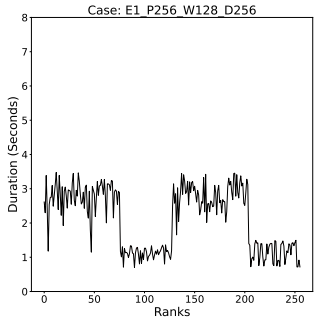
(a) Run 1, 128 MiB per process



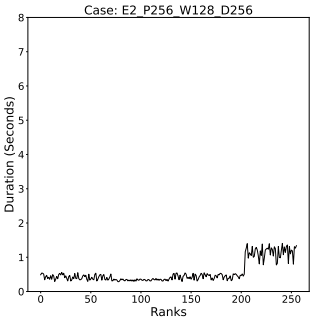
(b) Run 2, 128 MiB per process



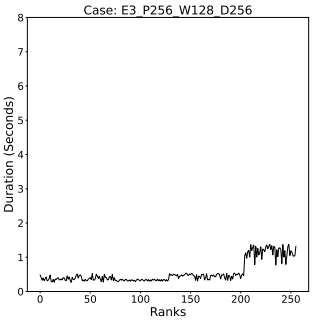
(c) Run 3, 128 MiB per process



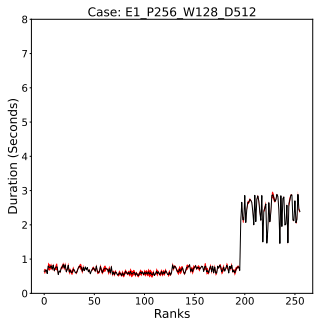
(d) Run 1, 256 MiB per process



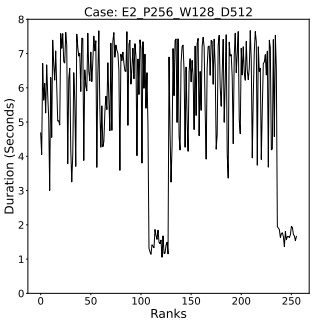
(e) Run 2, 256 MiB per process



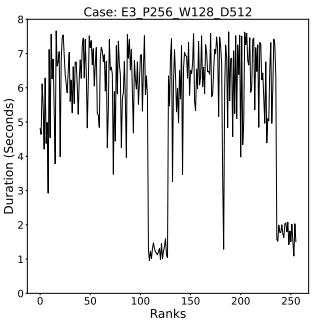
(f) Run 3, 256 MiB per process



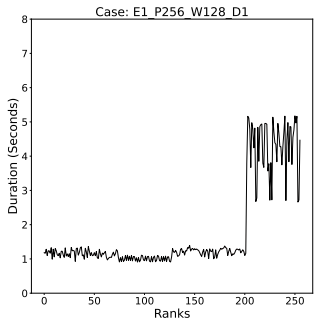
(g) Run 1, 512 MiB per process



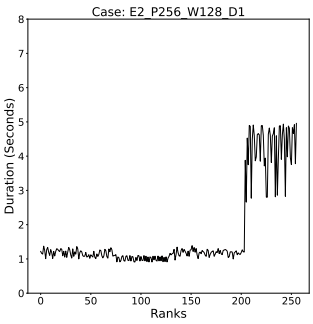
(h) Run 2, 512 MiB per process



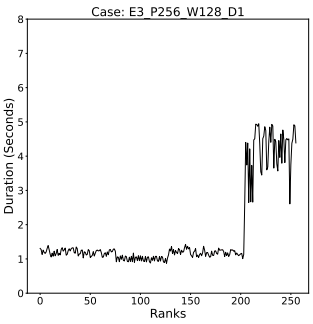
(i) Run 3, 512 MiB per process



(j) Run 1, 1 GiB per process



(k) Run 2, 1 GiB per process



(l) Run 3, 1 GiB per process

Figure 6.13: Average communication time per iteration for DEISA3 experiments, the number of processes is fixed to 256, we vary the size of the data from 128 MiB to 1 GiB, and we show results over the 3 runs.

the three versions. Less metadata and fewer heartbeat messages coming from the bridges help to reduce the variability of communication time. The takeaway from those experiments is that we could improve

performance by minimizing the frequency of messages sent to the scheduler from the bridges. This 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 if they still need results as they even don't wait for any. The only variability that we still encounter is the one related to the placement of the process, scheduler and workers.

In this last part, we are interested in the strange values we got in Figure 6.11. In Figure 6.16, the number of processes is 64, and the data size is 1 GiB for DEISA2. We notice that there is a big variability over processes and runs and a small one over iterations. In the best cases, we spend 2 seconds to send the data, and in the worst cases, we spend around 12 seconds. In Figure 6.17, the number of processes is 128, and the data size is 1 GiB for DEISA3. We notice that there is variability over ranks and runs but not over iterations (no red band). In the best cases, we spend around 1 second to send 1 GiB; in the worst ones, we spend more than 10 seconds. For both cases, since the simulation algorithm requires a global synchronization between all ranks at each timestep, we use the maximal value registered by all the ranks before we compute the mean over iterations and runs in Figure 6.11; we capture the worst cases.

We did not do further investigation related to these variabilities. However, we can consider optimizing communications with the placement of the workers and simulation processes as a perspective.

6.4.3.2 Experiment IV

In this section, we analyse the performance of the new version of DEISA (DEISA3) compared to the old version (DEISA1) and post hoc performance using Dask. We compare the results of the IPCA we had in Chapter 5 to the results we got with the new version of IPCA presented in Section 5.3.3.

Figure 6.18 summarizes weak scaling performance from the simulation side. The first subfigure from the left shows results for 128 MiB per process, in the middle for 256 MiB and the right for 512 MiB. 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 have 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 only computed the mean and the standard deviation over the remaining iterations.

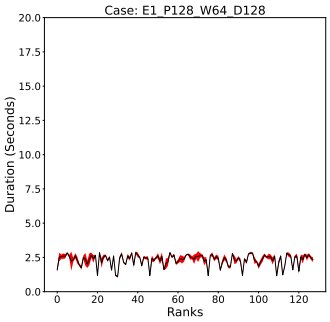
The different results in those subfigures were already discussed in the previous sections. The HDF5 writes are chunked, so we have almost the same HDF5 write time as in Chapter 5. The communication times for both DEISA versions are almost similar, with more variability in DEISA1. The missed values for post hoc here are due to crashes in the simulation side, likely due to a bug in HDF5 [9].

We focus more on the analytics part to analyse how the two versions of IPCA performed. Figure 6.19 shows the weak scaling results for the analytics part of the different configurations of **Experiment IV**. the first bar from the left of each scale (in red) represents analytics time for post hoc with the IPCA presented in Section 5.3.3. 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 6.4.2.1. 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.

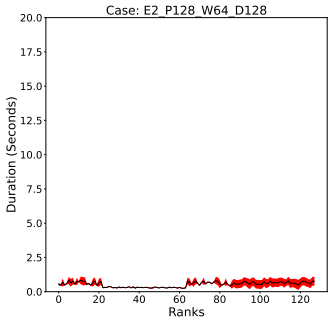
For the different chunk sizes, for small scales, DEISA versions are comparable to post hoc versions. Post hoc with our new IPCA is even a bit more efficient than DEISA when the number of Dask workers is two. When increasing the problem size, DEISA versions perform better than post hoc.

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 x1.8 faster when the chunk size is 256 MiB, and the number of workers is 16. We expect that this is due to the way we submit tasks to Dask in the version of IPCA. Instead of submitting the tasks for each `partial_fit`, in the new version of the 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. However, if those two tasks are in the same task graph, 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.

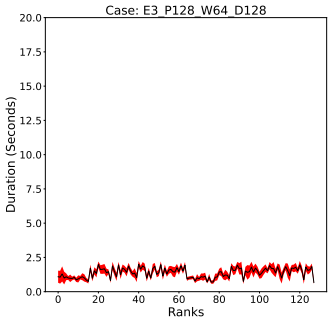
This is also beneficial for in situ analytics (in DEISA1, we have used the old version, and in DEISA3, the new one). However, it is less visible as the time spent waiting for the simulation data is included, and



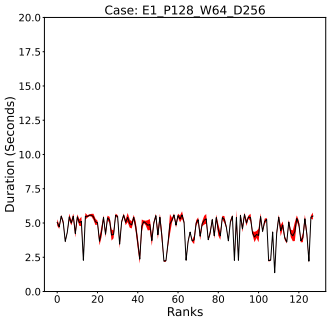
(a) Run 1, 128 MiB per process



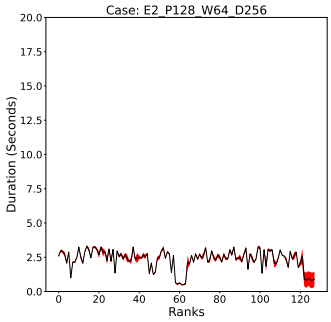
(b) Run 2, 128 MiB per process



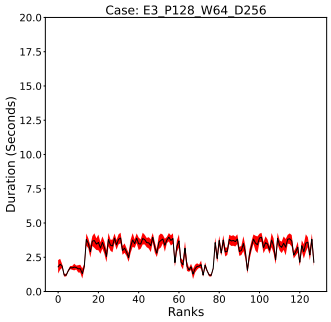
(c) Run 3, 128 MiB per process



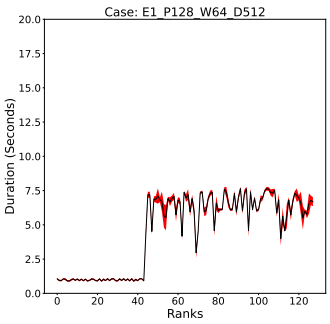
(d) Run 1, 256 MiB per process



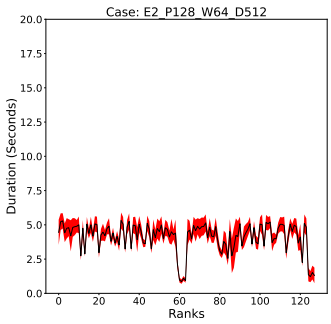
(e) Run 2, 256 MiB per process



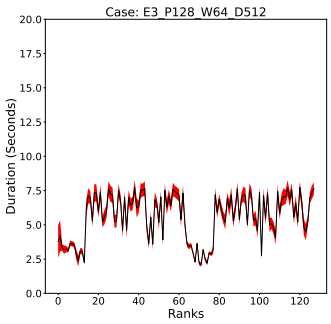
(f) Run 3, 256 MiB per process



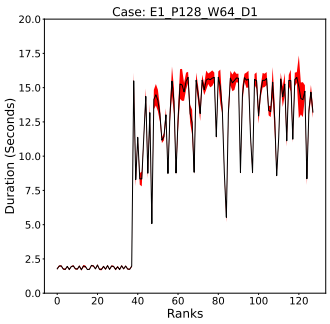
(g) Run 1, 512 MiB per process



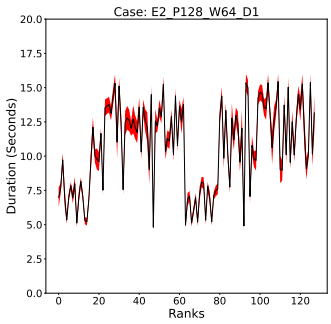
(h) Run 2, 512 MiB per process



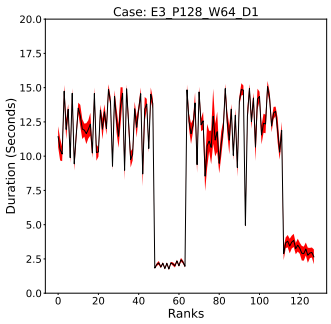
(i) Run 3, 512 MiB per process



(j) Run 1, 1 GiB per process



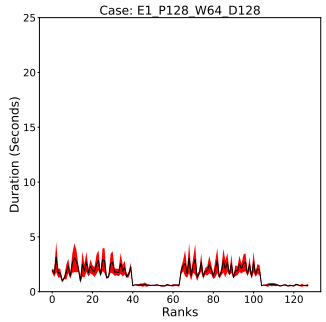
(k) Run 2, 1 GiB per process



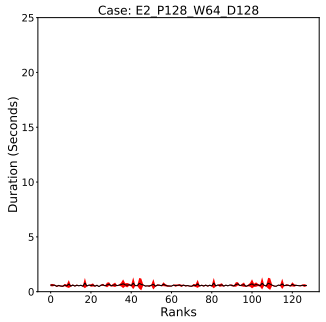
(l) Run 3, 1 GiB per process

Figure 6.14: Average communication time for DEISA2 experiments, the number of processes is fixed to 128, we vary the size of the data from 128 MiB to 1 GiB, and we show results over the 3 runs.

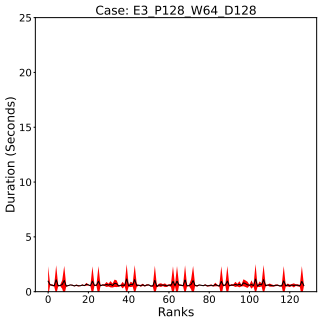
the time spent running tasks in in situ is usually short compared to the time spent reading data from disk.



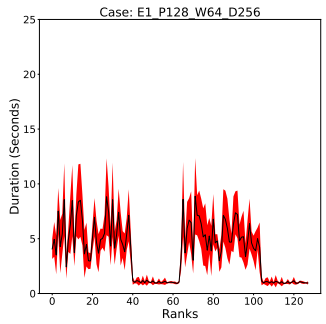
(a) Run 1, 128 MiB per process



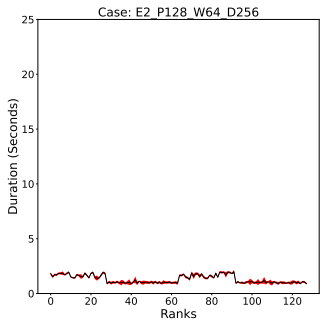
(b) Run 2, 128 MiB per process



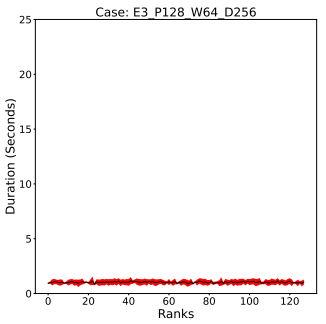
(c) Run 3, 128 MiB per process



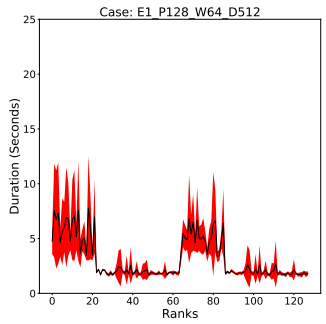
(d) Run 1, 256 MiB per process



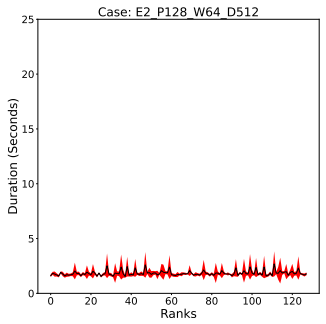
(e) Run 2, 256 MiB per process



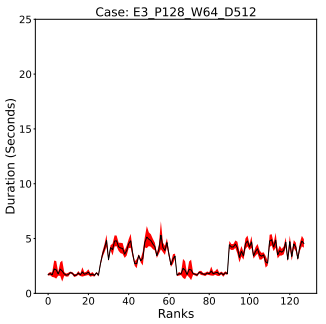
(f) Run 3, 256 MiB per process



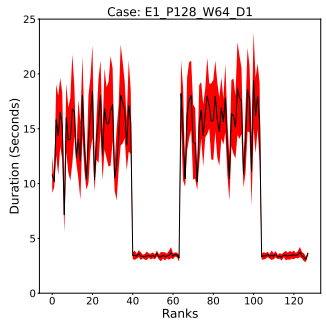
(g) Run 1, 512 MiB per process



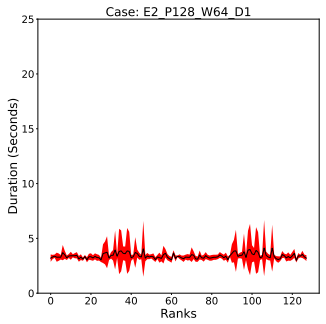
(h) Run 2, 512 MiB per process



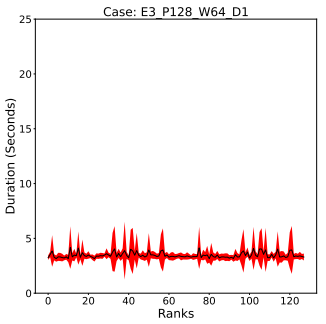
(i) Run 3, 512 MiB per process



(j) Run 1, 1 GiB per process



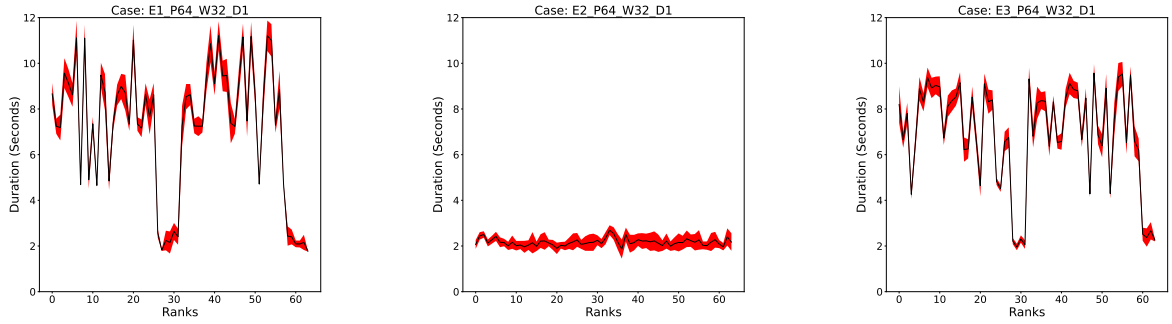
(k) Run 2, 1 GiB per process



(l) Run 3, 1 GiB per process

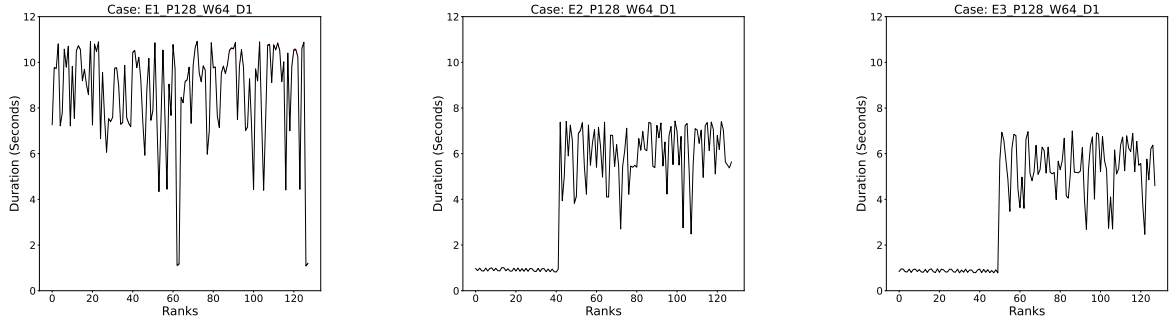
Figure 6.15: Average communication time for DEISA1 experiments, the number of processes is fixed to 128, we vary the size of the data from 128 MiB to 1 GiB, and we show results for the 3 runs.

We can not verify all the optimizations brought by the new version of the IPCA, but we can check the performance report and check the trend and some statistics about the task stream. Figure 6.21 and 6.20



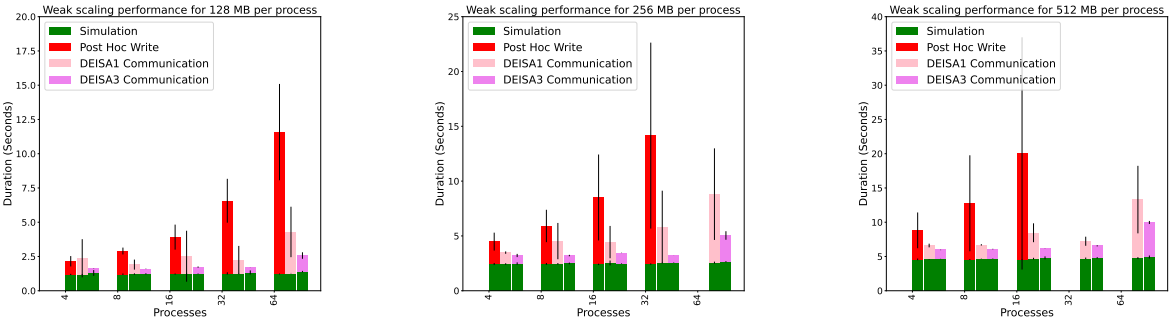
(a) Run 1, 1 GiB per process (b) Run 2, 1 GiB per process (c) Run 3, 1 GiB per process

Figure 6.16: Average communication time for DEISA2 experiments, the number of processes is fixed to 64 and the size of the data to 1 GiB, and we show results for the 3 runs.



(a) Run 1, 1 GiB per process (b) Run 2, 1 GiB per process (c) Run 3, 1 GiB per process

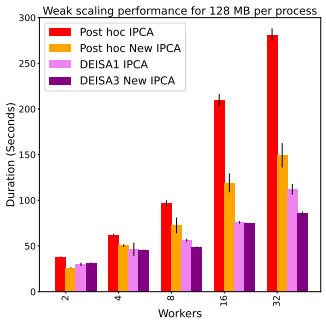
Figure 6.17: Average communication time for DEISA3 experiments, the number of processes is fixed to 128 and the size is 1 GiB, and we show results over the 3 runs.



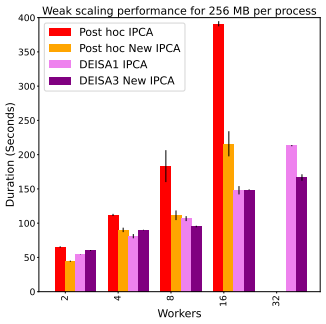
(a) Average simulation, communication and IO times per iteration for 128 MiB per process (b) Average simulation, communication and IO times per iteration for 256 MiB per process (c) Average simulation, communication and IO times per iteration for 512 MiB per process

Figure 6.18: Weak scaling average simulation, communication and IO times per iteration for 128 MiB, 256 MiB and 512 MiB per process for three experiments: the first stacked bar from the left of each scale represent results for post hoc version: simulation in green, communications in red. The second stacked bar shows results for the old version of DEISA (DEISA1): simulation in green and communication in pink. The third stacked bar shows results for the new version of DEISA (DEISA3): the simulation in green and the communications in violet.

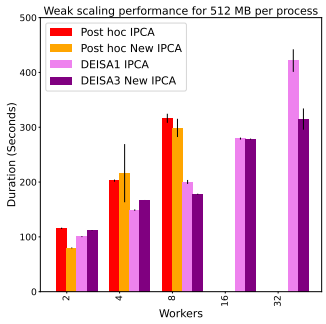
show the tasks streams for the DEISA3 and the Dask version with the new IPCA algorithm, which can be compared to the task stream in Figure 5.7 and 5.6 for the old versions of the IPCA. We can notice a



(a) Average analytics duration for 128 MiB chunk size



(b) Average analytics duration for 256 MiB chunk size



(c) Average analytics duration for 512 MiB chunk size

Figure 6.19: Weak scaling average analytics time for 128 MiB, 256 MiB and 512 MiB per process for three experiments: the first bar from the left of each scale (in red) represents analytics time for post hoc with the IPCA presented in Section 5.3.3. 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 6.4.2.1. The third bar from the left (in violet) shows analytics time for the old version of DEISA (DEISA1) with the old IPCa, and the last bar from the left (purple) shows results for the new version of DEISA (DEISA3) with the new IPCA .

Version	DEISA1	DEISA3	DASK1	DASK2
Duration (s)	118,39	81	270.27	135
Number of Tasks	9269	7090	11565	9693
Transfer time (s)	1007.39	1354.78	2709.88	1446

Table 6.3: Task Stream Summary for the 4 versions of the IPCA: DEISA1 version and DASK1 version both use the old iterative IPCA, DEISA3 and DASK2 use the new IPCA. DEISA versions are in situ, and DASK versions correspond to the post hoc versions.

difference, at least in trend. With the new version of the IPCA, a new part in the task stream appears that was not present in the old version of the IPCA. This appears before the larger task stream. It represents the computation of some bits of data necessary for the construction of the task graph. We also notice that there are fewer tasks as time progresses for both DEISA and Dask, because having the global view of the task graph at the beginning allows Dask to compute all ready tasks as soon as the data is available, and the more time progresses the fewer there are tasks to perform. This is possible thanks to optimizations that Dask applies to the graph.

Table 6.3 summarizes the statistics collected from the different task streams, and here we clearly notice that there are fewer generated tasks in the new version both for DEISA and post hoc. The transfer time for post hoc versions decreases, and this may be due to the efficiency of Dask in optimizing communications by reducing data transfer when it has a global view of all the tasks to run. For DEISA versions, we notice a small increase in the transfer time, which is justified by the imposed placement of simulation data in worker's memory that may trigger more transfers in the new version.

To check the efficiency of the different methods over configurations, we have fixed the number of processes and represented the efficiency in *MibiBytes per Second*. 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 6.22.

In the Subfigure 6.22a, 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), 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 rather stable until 64 processes. Remember that for the in situ cases, we measure the `scatter` method 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), which means that we can not achieve the

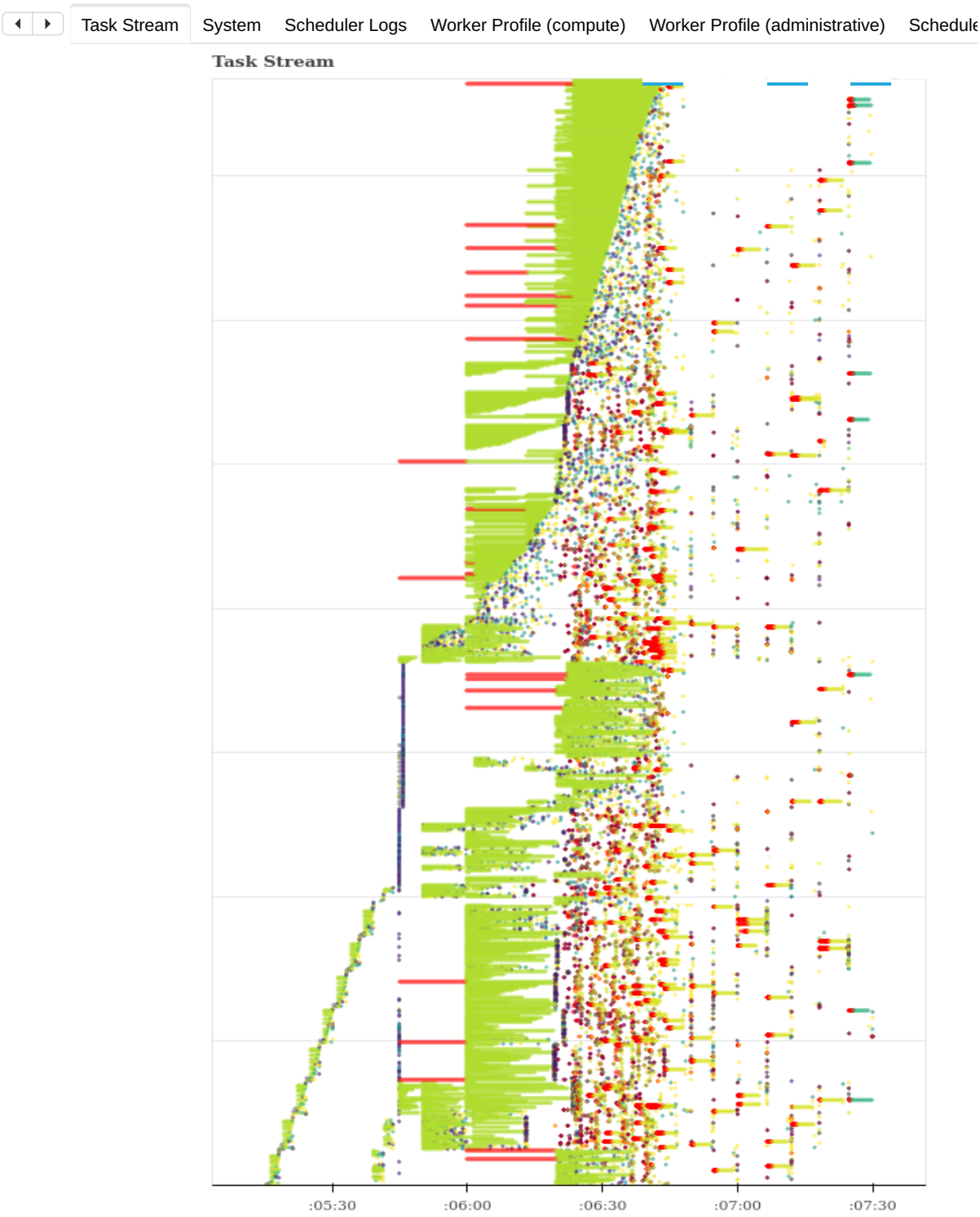


Figure 6.20: Task stream generated by Dask for post hoc the new IPCA with chunking activated for 64 processes, 32 workers and 128 MiB per process. Number of tasks: 9693 Compute time: 8359.85s Deserialize time: 26.64 s Disk-read time: 67.34 ms Transfer time: 1446.00 s.

theoretical performance of the aggregated bandwidth.

We can keep in mind three takeaways from those experiments.

- Post hoc performance gets worse when increasing the problem size because the PFS gets saturated

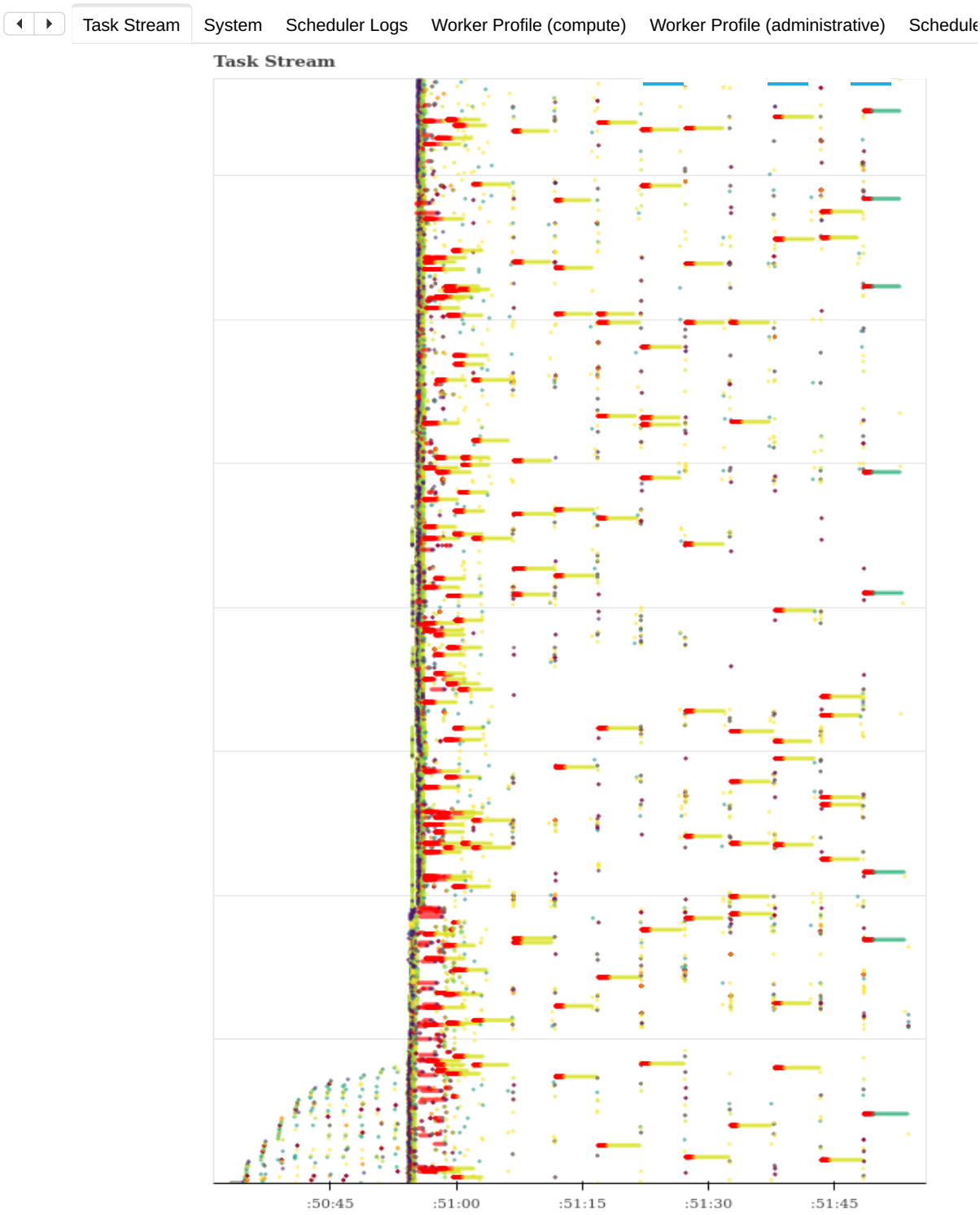
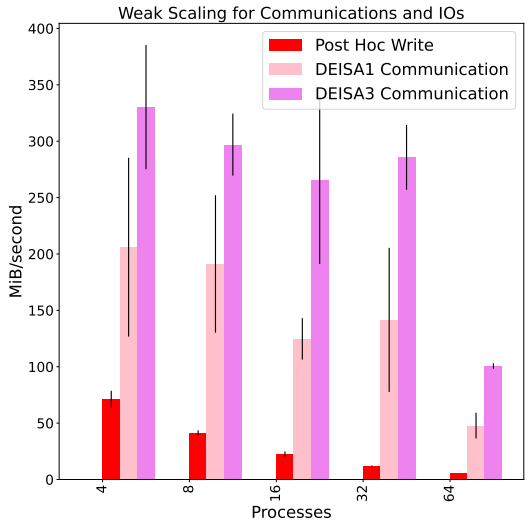


Figure 6.21: Task stream generated by Dask with in situ analytics enabled for the new IPCA, for 64 processes, 32 workers and 128 MiB per process. Number of tasks: 7090. Compute time: 837.69s. Deserialize time: 1.47s. Transfer time: 1354.78s.

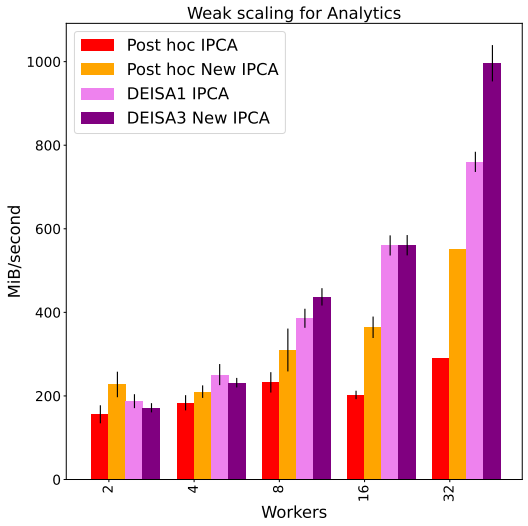
by the number of nodes writing at the same time.

- In situ results are better as they take advantage of the aggregated network bandwidth.
- In situ results are limited by the `scatter` operation that goes through the centralized scheduler and

the placement of the simulation processes and workers and scheduler, which may vary depending on the network topology.



(a) Bandwidth in MiB per second for the simulation side



(b) Bandwidth in MiB per second for the analytics side

Figure 6.22: Bandwidth in MiB per second for both the simulation and the analytics side. In the simulation, the HDF5 write for post hoc cases is represented in red and the communications over the network for the in situ cases (pink and violet bars). On the analytics side, the old IPCA version performance is represented in red, post hoc, with the new version in orange, the DEISA1 with the old PCA is represented in violet, and the DEISA3 with the new IPCA is in purple

.

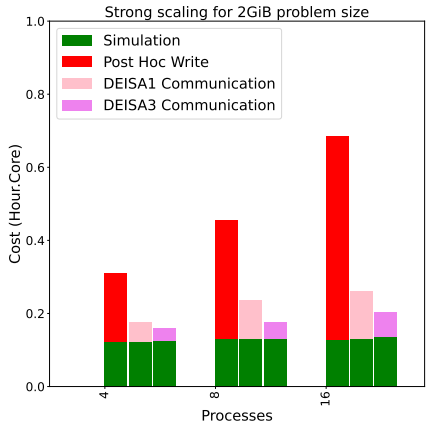
Subfigure 6.22b represents the computed bandwidth for the analytics part (MebiBytes computed per second) when the number of the Dask workers varies between 2 and 32. The x-axis represents the variation of Dask workers, and the y-axis is the bandwidth in MibiByte per second. Here again, the post hoc versions 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 (in red), the second bar represents the results of the post hoc with the new version of the IPCA (in 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).

First, in the first scale, the post hoc version with the new IPCA has a slightly better performance than all the others and starting for 4 workers, 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 discussed in the previous section. 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.

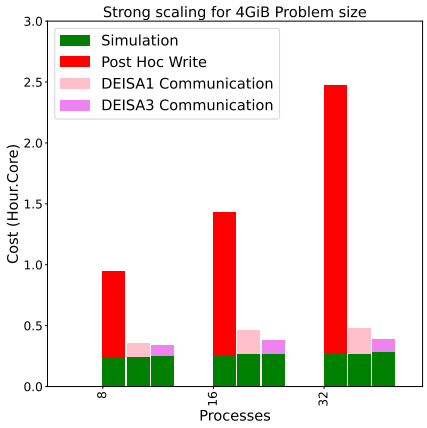
In this figure, we see that the post hoc with the old IPCA performance is almost stable when increasing the size of the problem, which is not the case for either the new version of the IPCA in post hoc or the in situ versions. But we can only affirm that the new IPCA in post hoc performs better when increasing the problem size. The exact reason for this behaviour still under investigation.

Figure 6.23 represents the strong scaling results in hour.core for the simulation side. We have fixed the problem size and varied configurations in each subfigure. In Subfigure 6.23a, we have fixed the problem size to 2 GiB and varied the processes from 4 to 16. In Subfigure 6.23b, we have fixed the problem size to 4 GiB and varied the processes from 8 to 32. In Subfigure 6.23c, we have fixed the problem size to 8 GiB and varied the processes from 16 to 64. In Subfigure 6.23d, we have fixed the problem size to 16 GiB and varied the processes from 32 to 64, we only represent the results for DEISA versions here because post hoc versions have crashed.

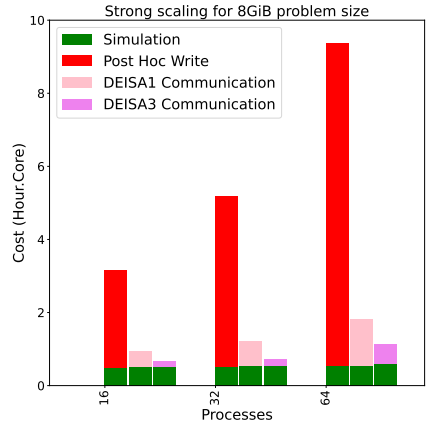
The simulation in all subfigures strong scales perfectly. In all cases, Post hoc writes are more costly than DEISA communications, and the cost increases with the number of processes. In the largest configuration, post hoc write per iteration is 18 times more costly than DEISA3: in situ workflows are less costly than post hoc workflows. In almost all configurations, DEISA3 is more efficient than DEISA1 and



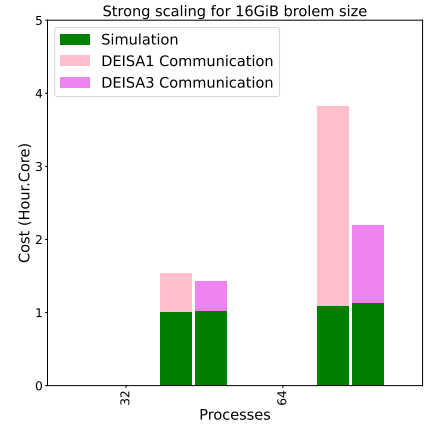
(a) Strong scaling results represented in hour-core for a 2 GiB problem size



(b) Strong scaling results represented in hour-core for a 4 GiB problem size



(c) Strong scaling results represented in hour-core for an 8 GiB problem size



(d) Strong scaling results represented in hour-core for a 16 GiB problem size

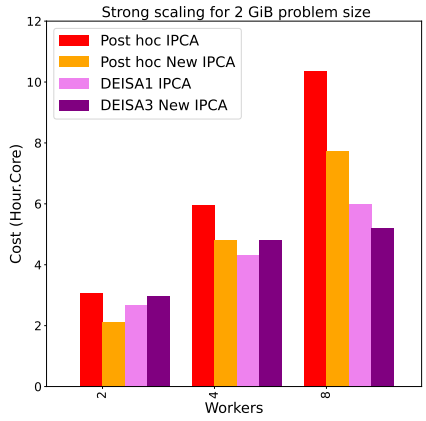
Figure 6.23: Strong scaling results represented in hour-core for the simulation side. The simulation is represented in green bars, and the post hoc HDF5 write in red. DEISA1 communication in pink and DEISA3 communications in violet

.

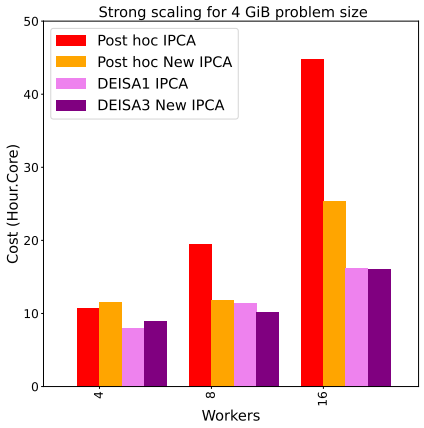
strong-scales better.

Figure 6.24 represents the strong scaling results in hour.core for the analytics side. In each subfigure, we have fixed the problem size and varied the configurations. In Subfigure 6.24a, we have fixed the problem size to 2 GiB and varied the processes from 2 to 8. In Subfigure 6.24b, we have fixed the problem size to 4 GiB and varied the processes from 4 to 16. In Subfigure 6.24c, we have fixed the problem size to 8 GiB and varied the processes from 8 to 32. In Subfigure 6.24d, we have fixed the problem size to 16 GiB and varied the processes from 16 to 32, we only represent the results for DEISA versions here because post hoc versions have crushed.

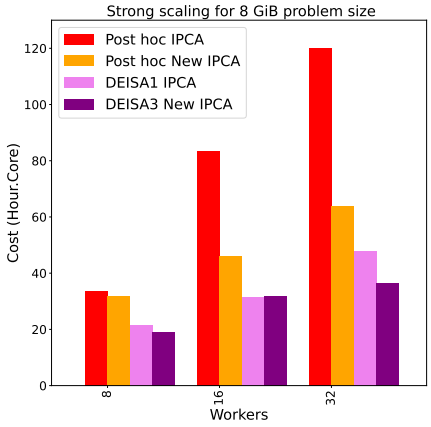
In all cases, post hoc versions are more costly compared to the in situ configuration again. The cost of the post hoc analytics with the old version of IPCA increases in linearly with the number of processes.For the new version of the IPCA in post hoc configuration, it strong-scale better and thus has less cost than the old version. The in situ version has the same cost in almost all configurations. The cost increases with the number of workers but is still better than post hoc versions. In other words, for a fixed problem size, if the algorithm is more costly when increasing the number of workers, it means that it is more efficient with a larger chunk size, likely because of communications. In the largest configuration, we have post hoc with the old version of IPCA is x3 times more costly than DEISA3 with the new version of the IPCA.



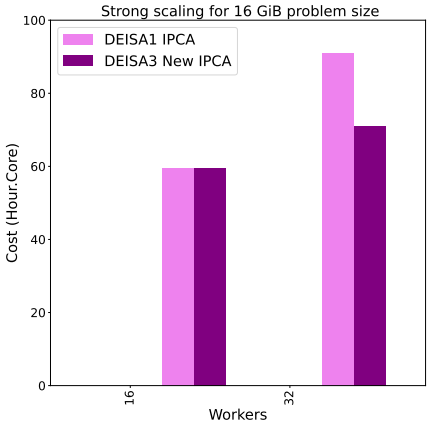
(a) Strong scaling results represented in hour-core for a 2 GiB problem size



(b) Strong scaling results represented in hour-core for a 4 GiB problem size



(c) Strong scaling results represented in hour-core for a 8 GiB problem size



(d) Strong scaling results represented in hour-core for a 16 GiB problem size

Figure 6.24: Strong scaling results represented in hour-core for the Analytics side. The red bar represents the results of the post hoc version with the old IPCA. The orange bar shows results of the post hoc analytics results with the new IPCA. The violet bar shows the results of the DEISA1 with the old IPCA, and in purple, the results of the DEISA3 with the new IPCA algorithm.

6.5 Production Use Cases

In this section, we present another aspect of the experiments, which is the integration of DEISA in production use cases, namely GYSELA [99, 123, 122, 57] and ARK [136].

6.5.1 GYSELA 5D

GYSELA (GYrokinetic SEmi-LAgrangian) is a global full-f [97] nonlinear gyrokinetic code that simulates electrostatic plasma turbulence and transport in the core of Tokamak devices. It evolves the complete 5-dimensional (three space coordinates, two velocity coordinates) particle distribution function. As it is impossible to save its full evolution over time, reductions from 0 to 3D data are performed. For instance, in a simulation where the 5D mesh grid has around $2.75 * 10^{11}$ points, only 30 TeraBytes are stored.

GYSELA was instrumented with PDI for IOs and checkpoints by Yacine Ould Rouis, which made the integration of DEISA easy. Preliminary experiments were run using 4 GiB chunks per process, without totally disabling heartbeats and by performing data selections to eliminate ghost zones. They have shown unexpectedly long transfer time to Dask workers in comparison to our previous experiments. The impact of heartbeats has already been discussed in the previous experiments. Our investigation has also shown

that Dask serialization is very inefficient for non-contiguous data that appears due to ghosts exclusion. Dask implementation adds multiple useless copies, which result in non-negligible overheads. Experiments are still in progress regarding this point.

6.5.2 ARK2-MHD

DEISA has also been integrated into the ARK2-MHD[3] code for the *Dynostar* Grand Challenge on ADASTRA[4] (PhD of Remi Bourgeois). A "Grand Challenge" is a period where selected scientists are allocated compute time to run full-scale experiments on new French supercomputers in order to validate them at scale before entering normal production. ARK2-MHD is a finite volume simulation code for turbulent convective dynamo. The resolution used in the Grand Challenge was 4096^3 cells with 9 variables (density ρ, pressure P, 3 components of velocity u_{xyz}, 3 components of the magnetic field B_{xyz}, and the mixing mass ratio X). This represents $\simeq$ 40 TeraByte, and it would be unreasonable to save it all before executing post-hoc reductions to follow the quantity of interest. Thus, the need for in situ approach.

ARK2-MHD has been instrumented with PDI to expose vertical and horizontal slices of the solution for both post hoc and in situ processing. DEISA has been used to compute the power spectrum of the kinetic energy in situ using a `fft2` on 2D slices of the 3D MHD to follow the evolution of the simulation. All this work has been done autonomously by the developers and users of ARK2-MHD, with only limited support from our part. They were really happy with both the performance and more importantly, the easiness to integrate in situ analytics using our approach.

6.6 Limitations

The single-graph implementation of DEISA, presented in this chapter, improves the performance compared to the multi-graph version presented in Chapter 5. Thanks to the newly introduced concepts in Dask to support the in situ analytics and in DEISA to improve the user experience, we could integrate DEISA into production use cases successfully. However, DEISA still has technical limitations related to the worker and scheduler placement compared to the simulation nodes. Not only for the reasons we already discussed in the previous chapters regarding performance variability but also regarding the supported configurations: for the moment, we have only implemented an in transit version. However, sometimes performing in situ reductions in the simulation nodes is better than sending large datasets and performing the reductions in transit. One can think about configurable workflows where the user can choose where to launch the worker processes: in situ and/or in transit. Another limitation is related to the serialization of the large non-contiguous array that is inefficient in Dask, likely because of the underlying memory copies they trigger. When the size of the data is relatively big, few GiBs, for instance, the time spent on serialization becomes non-negligible. An eventual solution could be the implementation of more adapted serialization algorithms or sending the contiguous data and making selections in the Dask worker side. Several other aspects may be improved in Dask that will boost the performance of DEISA at a large-scale related to the centralized scheduler and communication protocol. The centralized scheduler is a real bottleneck. We have limited the messages from the bridges, but still, a distributed scheduler may be a good move for extreme-scale simulations. A work in progress in already started to integrate DEISA with the Dask in Ray project[5] to take advantage of the interface of Dask (so DEISA) and the distributed scheduling of Ray.

The new implementation of DEISA signs a contract at the beginning of the simulation, where all the data that will be generated needs to be declared. This static way of defining the data that needs to be generated may be limiting, as we do not always know the number of timesteps to perform in advance. We may think of several interesting ways to bypass this limitation, such as hybridizing the two versions: by using the current implementation until a deterministic timestep and switching to the previous version where analytics are submitted only when the data arrives in memory (we do not have any information about it in advance). Or one can think about dynamic contacts that may change over time, so the bridges need to check the contacts at each timestep, and more analytics may be submitted by the client depending on that.

[3] https://gitlab.erc-atmo.eu/erc-atmo/ark
[4] https://www.genci.fr/en/node/1149
[5] https://docs.ray.io/en/latest/data/dask-on-ray.html

6.7 Summary

In this chapter, we introduced the in situ analytics in the Dask framework by introducing several concepts, mainly external tasks, that integrate the simulation tasks naively in a Dask task graph. We improved the interface and the performance and the operation of DEISA to offer a better experience for the users in terms of API, performance and functionality. We then performed several experiments and compared our results to post hoc and the old version of DEISA with and without contacts to show the advantages of the new operation.

The takeaways of this chapter are that DEISA scales and performs better than post hoc with now the exact same algorithms. DEISA is less costly in terms of hour.core, and storage space. Technically DEISA still can be improved regarding the way it sends the data to Dask, and the worker/processes placement to optimize communications, which can be investigated in future work.

Chapter 7

Conclusion and Perspectives

To know that we know what we know, and to know that we don't know what we don't know, that is the true knowledge

Nicolaus Copernicus

7.1 Conclusion and Perspectives

In this thesis, we have proposed an approach to bypass the IO bottleneck without adding complexity to the workflow setup by coupling MPI simulations with Dask distributed in an in transit configuration.

Our main contribution consists in the definition of a bridging model to couple MPI programs with Dask distributed in a producer-consumer configuration where the MPI program (simulation in our work) is the producer, and the distributed task-based application is the consumer. We have presented two implementations of our model using PDI for data handling. The multi-graph implementation is based on Dask internals and key management system. This work has been published and presented at the HiPC international conference. In the single-graph implementation of the bridging model, we have introduced new concepts in Dask to natively support in situ analytics and external tasks. In this version, we have improved the interface, the design, and the operation of DEISA compared to the multi-graph version. However, both implementations are complementary and could be used in the same workflow. For instance, the multi-graph implementation has the advantage of submitting a task graph at each time step, which may be used when the shape of the data that will be generated is not known in advance. In contrast, the single graph implementation has the advantage of simplifying task submission and taking advantage of Dask graph optimizations. We have evaluated DEISA versions compared to post hoc analytics and shown that with the exact same code, DEISA offers better performance since it avoids the IO bottleneck. The single-graph implementation presents less variability than the multi-graph one as it puts less strain on the scheduler.

DEISA has been integrated into two production codes: GYSELA 5D and ARK2-MHD. Both generate a massive amount of data and need in situ analytics. We have developed the time derivative and the IPCA algorithms that will be used in GYSELA. The integration into production codes was quite positive. It was a good step to interact with domain scientists so as to better understand their needs and expectations regarding the tools we provide, and get new ideas for usable and relevant software for them.

As already discussed, DEISA can still be enhanced in several directions, such as optimizing scheduler/-workers/processes placement to improve performance, reduce variability, and support more workflow configurations. For example, instead of sending data to the workers in a round-robin fashion, one could look for the worker physically closest to the sending process on the network to host its data. Another possibility is to distribute the workers over the list of all allocated nodes instead of putting them all in the first N nodes in the list. For instance, allocating a worker node every two simulation nodes should reduce the variability as there is a more significant probability that two communicating nodes are connected to the same switches.

DEISA bridges implementation can also be improved. Currently, the bridges are built on lightweight Dask clients. One could design a new class in Dask to implement the bridges that would communicate with each other without going through the scheduler. Using MPI for communication between the bridges could be better than the current version, where we use Dask variables that are hosted in the centralized scheduler. Note that MPI can also be used as a communication layer in the Dask cluster [152]. The `scatter` function could also be ameliorated. Instead of letting the bridges inform the scheduler about the new data sent to the workers, a possible improvement would be to make the worker do it. This is possible in the single-graph implementation thanks to the contract mechanism that ensures that the needed data *keys* are desired by the main analytics client a priori. Thus we make sure that the garbage collector will not delete the data since at least one client wants them. Another limitation we have encountered is related to the data serialization in the `scatter` operation, which is inefficient for non-contiguous data. To improve this aspect, one could implement a better serialization algorithm or send the complete contiguous array and make the selection on the worker side.

DEISA could be deployed in a larger spectrum of use cases, either in terms of software or hardware heterogeneity. It has been used together with post hoc in ARK2-MHD in an interesting hybrid in transit/post hoc workflow, which is quite common in HPC. This was made possible and easy without modifying the MPI code thanks to PDI and its configuration in a separate file. An improvement to consider is the support of hybrid in situ/in transit workflows to reduce the variability, as well as to construct more interesting workflows for certain analyses. For instance, if the first step of the analytics is a data reduction, it could be more interesting to perform those reductions in situ, so as to send smaller data for in transit processing. Several ways to support those workflows may be considered, such as running the reductions on the DEISA bridges (synchronous configuration) or in a worker deployed in a dedicated core (asynchronous configuration). The challenge here is to identify the tasks to run in situ and the tasks to run in transit. Depending on the task graph, the choice may be complicated. The simplest example is when the first tasks in the graph contain reduction tasks only. One could force all those

tasks without dependencies to run in the closest worker (dedicated worker or in the bridge if we don't dedicate a worker for those tasks) and then send data to in transit workers for further processing. Another example is when the reduction comes after other analytics, for instance, a local matrix multiplication followed by a reduction. In this case, one may run the first two layers in situ and then go in transit. However, it is not always trivial to distinguish in an optimal way when to start running tasks in transit. A possible solution is to make DEISA take action in the optimization of the task graph, after calling the `compute` methods, where it analyses and decides to force or not tasks to run on the in situ workers. The information regarding the closest worker to an MPI process needs to be available at this step.

Supercomputers are becoming increasingly heterogeneous, and considering this in software design is important. DEISA takes advantage of Dask in all its capabilities, like the possibility of using GPUs in the analytics. Dask can use GPUs in a few ways: such as using GPU-accelerated libraries (like Pytorch and TensorFlow) through `Delayed` and `Future` capabilities, or the `cuDF` Pandas-like library, that interoperates well and is tested against Dask `dataframe`[1]. In the in situ context, if the simulation is launched only in CPUs and the compute nodes also contain one or several GPUs, then one can easily use those for data analytics through DEISA. Note that all of these capabilities come for free with Dask, which is important to mention.

DEISA could be explored in a multi-producer/consumer configuration. For instance, to perform ensemble runs analytics [37], one could connect several simulation instances to the same Dask cluster and collect data from all of them. With the current version of DEISA, this can be implemented easily by simply adding the identifier of the simulation instance in data keys to recognize the source of the data every time it is received. Here again, one can take advantage of either `dask.array` API or `dask.dataframe` API and all the Dask ecosystem depending on the needs.

In situ analytics reduce the IO bottleneck problem. Still, it is not perfect because scientists have to know the analytics they want to perform in advance, which is not always easy or even possible. One could take advantage of *triggers* [120] alongside hybrid workflows in such situations. Triggers are mechanisms that customize the workflow depending on specific situations and events. They trigger distinct actions in response to specific conditions. The response can be any kind of analytics, control or IOs. DEISA could be used to catch rare events or strange behaviour in the simulations and trigger specific analyses if we already know which ones to apply, or generate a checkpoint to analyse post hoc when still in the discovery phase. If logical or physical errors are detected, then the simulation could be stopped to avoid wasting time and energy. The trigger mechanism could be implemented thanks to dynamic task creation in Dask using the `worker-client` functionality [2]. It creates a client within a worker (a server internally) and submits new tasks to the scheduler if a condition is fulfilled [3]. Such functionality could be easily used to trigger new online analytics or to steer the simulation when a rare event is detected.

This work has also brought closer the HPC and the High-Performance Data Analytics (HPDA) communities by coupling MPI programs with the HPDA tool Dask. With that, we participate with our bridging model in the convergence of those two communities without having to re-implement any of their stacks. This work shows a way to make HPC and HPDA work and cooperate together for a common finality in a single workflow while preserving the characteristics of each. Moreover, coupling such powerful programming models is a good step toward taking advantage of both, depending on the application requirement. For instance, the approach we provided can be applied outside of the context of in situ processing, by for example taking advantage of the dynamicity of the task-based programming for a given part of a large MPI code where high performance is not a must.

The DEISA approach could also be used in more research directions. For instance, it could be explored for HPC/AI convergence, where the already existing DEISA bridging model could be used to couple HPC simulations with ML models, either in the AI for HPC direction, where AI models are used to accelerate parts of the code or in the HPC for AI direction, where simulation data feeds the training of AI models. The bridging model solves some challenges in the HPC/AI convergence that are related to the coupling itself: one does not need to care anymore about the difference in programming languages: C/C++ or Fortran for HPC and Python/Dask for AI. Programming models are also hidden: parallelism, data distributions and communications are all managed by DEISA. Currently, an HPC/AI workflow, in the AI for HPC direction, can be easily set up by using existing pre-trained models. An example of using pre-trained models in PyTorch within Dask can be found in the Dask blog [4]. DEISA can also be used to feed the machine learning training process incrementally. This may need further expertise on AI because

[1] https://docs.dask.org/en/stable/gpu.html
[2] https://distributed.dask.org/en/stable/task-launch.html#connection-with-context-manager
[3] https://distributed.dask.org/en/stable/task-launch.html
[4] https://blog.dask.org/2021/03/29/apply-pretrained-pytorch-model

the data we get from a given timestep will not necessarily be saved to disk and thus available later. A possibility may be to consider a dimensionality reduction to reduce the data size to keep in memory. Note that the training time should be reduced because we do it online, and we avoid IOs.

DEISA could be used outside of data analytics; it could be deployed in supercomputers to collect real-time data about running applications and generates logs and reports to learn specific characteristics for data reproducibility or any other concern. A DEISA bridge could be associated with a running job to collect data from the job scheduler and any relevant real-time and raw data. Those data could then be transferred to the workers for processing before being included in diagnostic reports or just written to a database. In this use case, one can also imagine dynamic and reactive report generation if strange data has been detected. For example, if the computation duration takes more time than usual, then more data could be requested from the associated bridge about communications or IOs and so on. In more general words, DEISA could be used for online data collection and analytics since it can be integrated easily into an HPC platform with MPI applications and provides ease of use for data processing.

Another perspective that can be considered relates to the DEISA bridging model. It could be generalized for a larger range of code coupling configurations. The first step would to make the model work the other way around, where the task-based model produces the data and the MPI application is a consumer. Such a model should be interesting in workflows where the simulation is steered online by in situ analytics. Or in task-based codes where we want to accelerate a given part of the code using MPI. In such a configuration, more efforts should be made to ensure that external data coming from the task-based program will not deadlock the MPI program.

The bridging model could also be extended to other programming models with the idea of taking advantage of those models' capabilities. This can be relevant when dealing with heavy applications treating several different concerns. Such a possibility is interesting, for instance, to use in high-level code generation where users might want to write code using different programming models. Instead of only considering code generation for heterogeneous backends, one can imagine code generation for heterogeneous distributed programming models. This idea can be explored to generate skeletons of the desired workflows with the desired programming models and tools and all needed configurations for the code coupling. The generated code could then be used by domain scientists to develop the core functionalities separately.

Another perspective that can be considered is regarding the external tasks that we have introduced in Dask distributed. This concept is generic enough to receive data from any external source, not only MPI simulation data. One can think about using it in other contexts, such as the implementation of digital twins' workflows. The same Dask instance could be fed by both running simulations and real devices. An interesting state-of-the-art with both theoretical and practical study and challenges can be found in [154]. External tasks in Dask have no restrictions at the moment, and the received data is not verified. Thus they may present cybersecurity issues. Dask distributed already supports TLS/SSH communications between clients/scheduler/workers [5], but in the context of digital twins, further security validation may be required both at Dask and devices levels.

At the end of this thesis, we would like to highlight that DEISA is not just a tool but a pattern to keep in mind. In situ data analytics are only a specific case of data analytics. Instead of recreating a new data analytics stack from scratch for this case, it may be easier and less costly to make existing data analytics stacks work in situ. The idea can be generalised for other similar situations.

To know that we know what we know, and to know that we do not know what we do not know, that is the true knowledge, but the more we know, the more we feel that we don't know.

[5] https://distributed.dask.org/en/stable/tls.html

Part III

French Summary

Chapter 8

Résumé de la Thèse en Français

8.1 Introduction

Le terme *supercalculateur* a été utilisé pour la première fois en Mars 1920, dans le New York World, pour désigner *"de nouvelles machines statistiques dotées de la puissance de calcul de cent mathématiciens pour résoudre des problèmes algébriques même très complexes"* [150]. Depuis lors et au cours des 70 dernières années, l'informatique s'est développée du premier ordinateur électronique programmable à usage général, l'Electronic Numerical Integrator and Computer (ENIAC, Figure 1.1 [5]), capable de traiter 500 opérations en virgule flottante par seconde (flops), achevé en 1945, jusqu'au premier supercalculateur Exascale au monde : Frontier (Figure 1.2 [67]) capable de traiter 1.102 Exaflops (par seconde?) [33, 6].

Pour comprendre ce que c'est le calcul haute performance (en anglais, High Performance Computing HPC), le besoin d'avoir des supercalculateurs alors qu'un ordinateur de bureau suffit pour nos tâches quotidiennes, nous avons choisi ici les deux définitions qui nous semblent les plus pertinentes. Un supercalculateur est défini dans la proposition d'initiative de nouvelle technologie du JISC [2] et cité dans [104] comme étant: *"des ressources informatiques qui fournissent plus d'un ordre de grandeur de puissance de calcul que ce qui est normalement disponible sur un ordinateur de bureau"*. Le calcul haute performance est défini sur le site web d'IBM [7] comme suit: *"Le HPC est une technologie qui utilise des grappes de processeurs puissants, travaillant en parallèle, pour traiter des ensembles de données multidimensionnelles massives (big data) et résoudre des problèmes complexes à des vitesses extrêmement élevées. Les systèmes HPC fonctionnent généralement à des vitesses plus d'un million de fois supérieures à celles des systèmes d'ordinateurs de bureau, d'ordinateurs portables ou de serveurs les plus rapides"*. Les deux définitions sont complémentaires et mentionnent toutes deux la puissance de calcul et la taille de la mémoire des supercalculateurs, ce qui nous amène à répondre à la deuxième question concernant le besoin de supercalculateurs. Il s'agit de résoudre des problèmes complexes, limités en termes de mémoire et/ou de calcul.

De nos jours, les supercalculateurs participent à la resolution d'une longue liste de défis scientifiques comme ceux liés aux énergies vertes et renouvelables, fusion nucléaire, énergie solaire et hydraulique, et préoccupations médicales allant de la compréhension du corps humain à la découverte de médicaments, grâce à la puissance de calcul qui accélère le processus de recherche. L'astrophysique, qui tente de comprendre notre univers, la chimie et la création de nouveaux matériaux, la simulation de phénomènes naturels ou le couplage d'expériences réelles et l'internet des objets (IoT) avec le calcul intensif pour former des systèmes de jumeaux numériques sont d'autres exemples de problèmes à resoudre. Plusieurs anciens problèmes scientifiques ont commencé à être résolus avec l'émergence du HPC, grâce à la puissance de calcul et à la mémoire qu'il offre. Par exemple, l'utilisation de l'apprentissage automatique et de l'intelligence artificielle a gagné en popularité depuis l'usage du calcul générique sur processeur graphique (GPGPU). De même, parallèlement aux télescopes, le HPC a été utilisé pour comprendre et explorer les aspects théoriques d'un trou noir, et en 2019, la toute première image d'un trou noir a pu être synthétisée [91].

Selon J. Dongarra [77], la valeur d'un supercalculateur découle de la valeur du problème qu'il résout. En tant que tel, le calcul intensif est étroitement lié aux applications scientifiques qui sont généralement des simulations. Les programmes qui modélisent les phénomènes physiques sont complexes et nécessitent une grande puissance de calcul et de mémoire pour fonctionner. Pour atteindre de telles performances, l'architecture des ordinateurs a évolué, passant d'une simple implémentation de l'architecture de Von Neumann à des millions de cœurs et d'accélérateurs puissants interconnectés, capables de traiter des

centaines de pétaflops par seconde. Parallèlement à ces architectures complexes, différents modèles de programmation sont proposés pour implémenter des programmes efficaces. Les simulations haute performance sont généralement des programmes itératifs qui évoluent dans le temps et peuvent produire, dans certains domaines tels que les prévisions météorologiques, des dizaines de téraoctets par heure. Les données générées doivent être traitées pour comprendre le phénomène étudié. Dans les workflows classiques, les données générées par la simulation sont d'abord écrites sur des disques, puis relues pour un post-traitement, également connu sous le nom de traitement post hoc, généralement sur un autre poste de travail. Les analyses de données sont facilement réalisées avec des codes Python séquentiels, mais les scientifiques ont récemment adopté des bibliothèques parallèles adaptées aux analyses données et big data en raison de l'énorme quantité de données générées par les simulations.

La taille des résultats n'est pas le seul défi. Si les performances des processeurs ont augmenté selon la loi de Moore, ce n'est pas le cas de la bande passante des disques, et l'écart entre les deux s'élargit de quelques ordres de grandeur, créant ce que l'on appelle le goulot d'étranglement des entrées-sorties (IO bottleneck). Les workflows in situ ont été introduits en 2008. Ils visent à traiter les données générées par des simulations à grande échelle aussi près que possible du moment (temps) et de l'endroit (mémoire) où elles ont été produites. Ces workflows contournent les accès aux disques en traitant les données dans les mêmes resources de calcul que la simulation, évitant ainsi le goulot d'étranglement des entrées-sorties mentionné précédemment. Malgré les performances démontrées par les workflows in situ, ils ne sont pas largement utilisés par la communauté en raison de la complexité de leurs configurations et la nécessité de connaitre a priori les analyse de données à effectuer.

La plupart des outils in situ existants sont implementés audessus du modèle de programmation MPI hérité de la simulation. Si ce modèle, ainsi que d'autres connus sous le nom de MPI+X, sont bien adaptés aux applications scientifiques pour leur régularité, ils ne sont pas adaptés à l'analyse de données. Les algorithmes d'analyse de données ont une structure différente de celle des simulations. Ils sont caractérisés par des structures de données et de contrôle et des schémas de communication irréguliers. Implémenter certains de ces algorithmes dans un modèle statique et synchrone tel que MPI c'est comme tenter de faire entrer un carré dans un rond. Non seulement ils ne sont pas compatibles, mais leur implementation est complexe.

Dans ce travail, nous voulons réunir la simplicité du post hoc et la performance des workflows in situ. En d'autres termes, nous couplerons des simulations haute performance parallélisées en MPI avec des analyses in situ écrites dans un modèle plus adapté aux algorithmes d'analyse de données, à savoir le modèle de programmation par tâches distribuées.

Le reste du document est organisé comme suit:

- **Partie I État de l'art** contient deux chapitres. Dans le chapitre 2, nous présentons le contexte et les travaux connexes, à savoir les outils in situ et la programmation par tâches distribuées. Dans le chapitre 3, nous présentons les outils utilisés dans ce travail, à savoir PDI et Dask distribué.

- **Partie II Contributions** contient les principales contributions scientifiques de ce travail et contient trois chapitres. Le chapitre 4 présente l'approche que nous proposons, appelée DEISA bridging model et son implémentation en utilisant Dask et PDI. Le chapitre 5 présente une implémentation complète, avec la configuration nécessaire et l'API utilisateur. Le chapitre 6 propose des améliorations conceptuelles pour DEISA et Dask distribués.

- **Partie III Conclusion et perspectives** résume les principales conclusions et leçons tirées de ce travail, et fournit un retour sur les perspectives et projets possibles.

8.2 État de l'Art

Dans cette partie, nous présentons le contexte détaillé de ce travail, les supercalculateurs, leurs architectures et les modèles de programmation parallèle. Nous examinons ensuite les workflows d'analyse de données haute performance, à savoir les workflows post hoc et les workflows in situ, en analysant les avantages et les inconvénients de chacun. La deuxième partie de ce chapitre est consacrée aux travaux connexes sur les outils de traitement des données in situ, avec une analyse comparative. En outre, nous présentons certains frameworks big data existants qui sont d'un grand intérêt pour ce travail, car notre objectif est d'apporter leur productivité aux workflows HPC. Enfin, nous présentons les outils utilisés dans ce travail, à savoir l'interface de données PDI pour la gestion des données et Dask distribué pour leur analyse.

8.3 Contributions

8.3.1 Modèle de Couplage des Simulations MPI et des Analyses Dask

Dans cette partie, nous présentons notre approche pour combiner la performance in situ et la facilité d'utilisation post hoc. Nous considérons un schéma producteur-consommateur, où le producteur est une simulation MPI, et le consommateur est un code d'analyse distribué en Dask. Notre approche consiste à proposer un bridging model entre MPI et Dask distribué qui cache la complexité du couplage des codes et toutes les différences sous-jacentes entre les deux modèles. Nous avons défini un ensemble de concept afin de proposer ce model notamment: les composant DEISA, les tâches DEISA, les événements internes/externes et les Delivery Facility. Nous proposons ensuite une implémentation utilisant MPI, PDI et ses plugins et Dask.

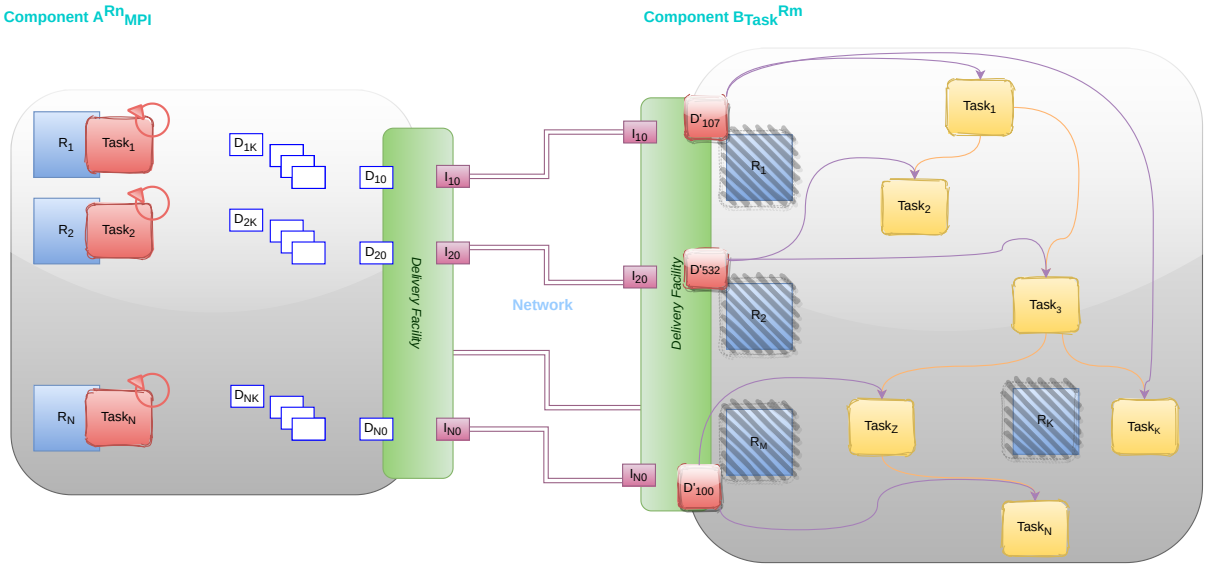


Figure 8.1: Exemple d'un workflow producteur-consommateur.

La figure 8.1 montre un exemple de couplage entre $A^{R_n}_{MPI}$ et $B^{R_m}_{Dask}$ où A est un producteur et B un consommateur. Le composant MPI dispose de R_n ressources. Chaque $Task_k$ est programmée explicitement sur un ensemble de ressources R_k. Dans les applications scientifiques, nous avons généralement un code itératif. Chaque tâche génère un bloc de données à un moment t (une seule tâche est représentée sur la figure, avec une marque de boucle). Ces blocs de données (petites bo5tes bleues $D_{i,j}$) sont partagés et envoyés a la Delivery Facility.

Le composant B est le consommateur des données. Il s'agit d'un composant dont l'implementation est par tâches distribuées qui dispose de R_m ressources gérées implicitement par un runtime (carés bleus avec des hachures grises). Un graphe de tâches est représenté comme un graphe de tâches DEISA (graphe jaune), avec des dépendances aux entrées externes. Ces entrées (en rouge) sont des données avec de nouvelles clés (ID) qui sont facilement reconnaissables, donc utilisables dans B.

Les données sont envoyées par le réseau entre les composants. Les DF assurent la connexion avec un composant distant, en identifiant et en redistribuant les données entre les composants. Dans cette

figure, l'identification des données se fait en deux étapes de chaque côté. Les petits rectangles bleus $D_{i,j}$ sont identifiées par trois éléments : D est le nom de la donnée, i par exemple, la position de ce bloc de données dans la distribution globale, et j correspond au pas de temps ou à l'itération. Ces clés peuvent être considérées comme locales au composant A. Le même composant a créé une nouvelle clé: $I_{l,m}$. Il s'agit d'une clé globale reconnaissable dans le DF de B. Dans le composant B, ces clés sont traduites en nouvelles clés compréhensibles en interne D'_k. Ce processus d'identification et de traduction est obligatoire mais peut être réalisé en moins d'étapes. Par exemple, si $I_{l,m}$ est reconnaissable par B, il n'est pas nécessaire de poursuivre la traduction à la réception.

8.3.2 DEISA1: Analyse In Situ en Dask

Cette partie est une version étendue de notre publication HiPC21, où nous avons mis en œuvre la version multi-graphe du bridging model DEISA. Dans cette version, nous avons utilisé le système de génération de clés Dask, la méthode `scatter` disponible pour envoyer des données aux workers Dask et un graphe de tâches est soumis à chaque iteration.

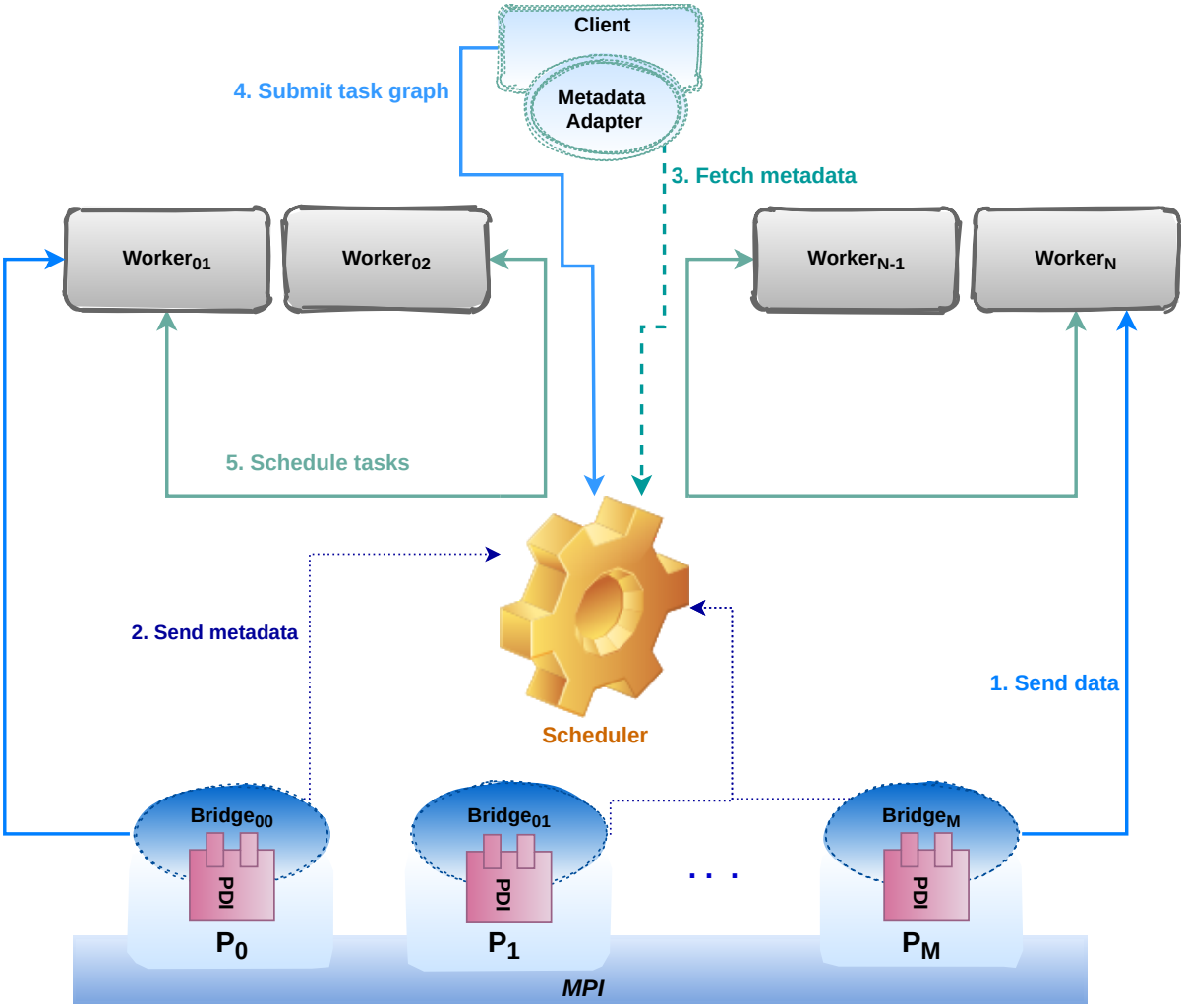


Figure 8.2: Architecture de DEISA1.

La figure 8.2 montre l'architecture du prototype DEISA. Nous couplons une simulation en exécution représentée par $M + 1$ processus MPI avec une instance Dask comprenant un scheduler, un client analytique et N travailleurs. Les données de simulation sont gérées avec l'interface de données PDI. Elles sont généralement partagées à chaque pas de temps (ou périodiquement tous les K pas de temps) via la fonction `pdi_expose` avec le plugin DEISA qui instancie un objet bridge par processus MPI.

Chaque bridge connecte un client Dask au scheduler. Le bridge envoie les données aux workers et les métadonnées au scheduler mais ne soumet aucun graphe de tâche au scheduler. À chaque pas de temps, les données générées par chaque processus MPI sont envoyées à un worker Dask présélectionné

avec un mode round-robin à l'étape d'initialisation (étape 1 dans la Figure 8.2). Chaque bridge construit des métadonnées relatives au bloc de données qui comprennent le nom des données, leur type et leur sous-type, leur taille et le pas de temps. Ces métadonnées sont envoyées au scheduler dans une Dask `Queue` associée au bridge (étape 2 dans la Figure 8.2).

Le delivery facility du côté de Dask est appelé DEISA *Adaptor* ou adaptor de métadonnées. À chaque pas de temps, il demande les métadonnées au scheduler (qui sont/seront disponibles dans les files d'attente) ; il les utilise pour créer un `dask.array`. Il crée un *dask.array* par bloc de données reçu d'un processus MPI, représentant le tableau local du processus. Il rassemble ensuite tous les blocs dans un tableau plus grand en utilisant la méthode `dask.array.block` disponible (étape 3 dans la figure 8.2), afin d'en obtenir un qui représente le tableau distribué global.

Le client récupère ce `dask.array`, qui n'est qu'un descripteur des données réelles résidant dans la mémoire distribuée des workers, et soumet un graphe de tâches qui traite ce `dask.array` (étape 4 dans la Figure 8.2).

Ce processus est effectué à chaque pas de temps, uniquement après la communication des données aux workers Dask: le client analytique soumet un graphique de tâches par itération de la simulation.

Le système proposé a été évalué et comparé au post hoc, l'analyse montre des performance intéressante avec des modifications minimales du code post hoc. Les expériences ont été faites sur deux supercalculateurs: Ruche et Irene en utilisant le code de simulation heat2D et une analyse en composantes principales incrémentale (IPCA).

8.3.3 Support des Tâches Externes en Dask

Dans cette partie, nous abordons certaines limites de DEISA1 en introduisant trois concepts principaux: les tableaux virtuels DEISA, les contrats et les tâches externes dans Dask. Grâce à ces concepts, nous mettons en œuvre une version à graphe unique du bridging model DEISA, où un seul graphe de tâches complet peut être soumis au début de la simulation sans attendre que les données soient disponibles. Cette implémentation améliore à la fois les performances et l'interface utilisateur et tire parti des optimisations du graphe des tâche dans Dask. La figure 8.3 montre la nouvelle architecture de DEISA.

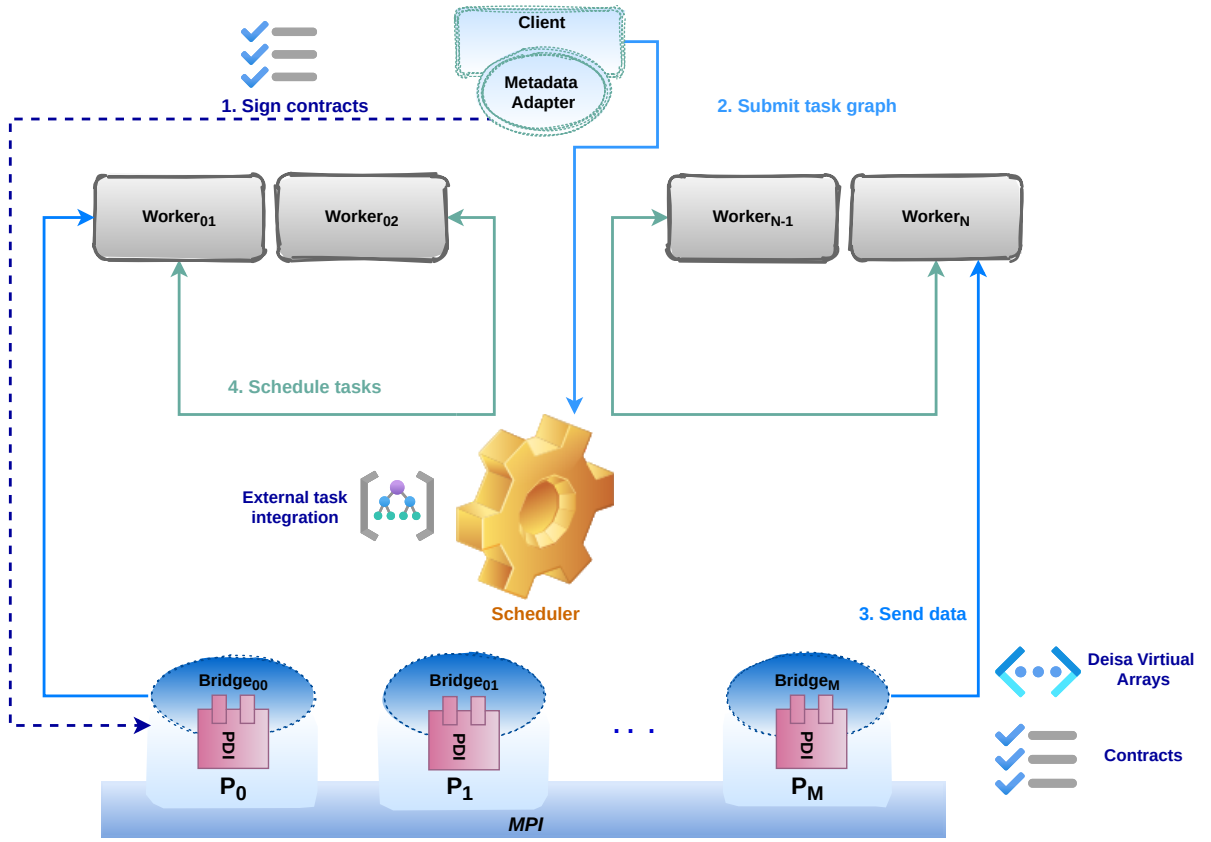


Figure 8.3: La nouvelle architecture de DEISA.

L'architecture est similaire à la version précédente de DEISA (voir Figure 6.1). Nous avons tou-

jours deux composants dans un schéma producteur/consommateur, où la simulation MPI en exécution représentée par $M+1$ processus est le producteur, et le cluster Dask est le consommateur. Nous améliorons et optimisons le fonctionnement du workflow en minimisant la charge sur scheduler et en fournissant une meilleure API pour l'utilisateur. Les principaux changements dans l'architecture visent à minimiser la charge du scheduler centralisé et la façon dont les deux composants communiquent.

Nous avons conservé l'implémentation du bridge au-dessus la classe client Dask. Nous passons toujours par le scheduler pour toutes les communications entre les bridges et le client d'analyse.

Au début de la simulation, la bride au rang 0 se connecte à l'ordonnanceur Dask et envoie la description des tableaux virtuels DEISA à l'adaptor. Le client d'analyse connecté à l'adaptor dans Dask, effectue une sélection de données en utilisant les objets `slice` dans le tableau DEISA en fonction des éléments nécessaires à l'analyse. Le client renvoie ensuite les sélections au bridge. Cela se fait par l'intermédiaire d'un Dask *Variable*, qui est ensuite accessible à tous les bridges. Cette opération est effectuée au début afin qu'il ne soit pas nécessaire d'envoyer des métadonnées au scheduler à chaque pas de temps, ce qui améliore les performances.

Touts les bridges sont synchronisées à cette étape et peuvent continuer dès que les données qu'elles doivent envoyer sont connues ou, en d'autres termes, que les contrats sont signés. Ensuite, le client soumet l'analyse. À chaque pas de temps, chaque bridge vérifie si son bloc de données est nécessaire. Si c'est le cas, il l'envoie au worker présélectionné.

Une analyse des performance a été effectué pour comparer la nouvelle version avec l'ancienne et le post hoc et montrer les nouvelles capacités du système. Les expériences ont été menées sur le supercalculateur Irene, en utilisant la simulation heat2D et deux analyses: la IPCA et une dérivée en temps.

Notre travail a été également intégré dans deux codes de production a savoir GYSELA 5D et ARK2-MHD.

8.4 Conclusion et Perspectives

Cette thèse a réuni la facilité du post hoc et la performance de l'in situ, ce qui est une nécessité pour les scientifiques dans certains domaines aujourd'hui. Tout d'abord parce qu'ils ne peuvent pas sauvegarder toutes les données générées par ces simulations sur disque, mais ont toujours besoin de les analyser. Ensuite, il serait préférable d'avoir accès au même écosystème qu'en post hoc, qui est à la fois adapté à l'analyse des données et caractérisé par une productivité élevée. Nous avons proposé une approche qui contourne le goulot d'étranglement des entrées-sorties sans ajouter une complexité à la configuration des workflows d'analyse en couplant les simulations MPI avec Dask distribué dans une configuration en transit.

Nos principales contributions consistent à définir un *bridging modèle* pour coupler les programmes MPI à Dask distribué dans une configuration producteur-consommateur où le programme MPI (simulation dans notre cas) est le producteur, et l'analyse par tâches distribuées est le consommateur. Nous avons présenté deux implémentations de notre modèle utilisant PDI pour la gestion des données et Dask distribué pour l'analyse in transit. L'implémentation multi-graphe du modèle est basée sur le design de Dask et son système de gestion des clés. Ce travail a été publié et présenté à la conférence internationale HiPC. Dans l'implémentation à graphe unique du modèle, nous avons introduit de nouveaux concepts dans Dask pour supporter nativement l'analyse in situ et les tâches externes. Dans cette version, nous avons amélioré l'interface, la conception et le fonctionnement de DEISA par rapport à la version multi-graphe. Cependant, les deux implémentations sont complémentaires et peuvent être utilisées dans le même workflow. Par exemple, l'implémentation multi-graphes présente l'avantage de soumettre un graphe de tâches à chaque pas de temps, ce qui peut être utilisé lorsque nous ne connaissons pas la durée de la simulation. En revanche, l'implémentation à graphe unique présente l'avantage de simplifier la soumission des tâches et de tirer parti des optimisations du graphe Dask. Nous avons évalué les versions de DEISA par rapport aux analyses post hoc et montré qu'avec exactement le même code, DEISA offre de meilleures performances, comme prévu, puisqu'il évite le goulot d'étranglement des entrées-sorties. L'implémentation d'un seul graphe présente moins de variabilité que l'implémentation de multi-graphes car il y'a moins de charge sur le scheduler Dask.

DEISA a été intégré dans deux cas d'utilisation en production : GYSELA 5D et ARK2-MHD. Ces deux projets génèrent une quantité massive de données et nécessitent des analyses in situ. Nous avons développé la dérivée temporelle et les algorithmes IPCA qui sont nécessaires et qui seront utilisés dans GYSELA. L'intégration dans les codes de production a été très positive. C'est une bonne étape pour interagir avec les physiciens, comprendre leurs besoins et leurs attentes concernant les outils que nous fournissons, et obtenir de nouvelles idées pour des logiciels utilisables et pertinents pour eux.

Comme nous l'avons déjà souligné, DEISA peut encore être amélioré de plusieurs points de vue, notamment en optimisant le placement des scheduler/workers/processus afin d'améliorer les performances, de réduire la variabilité et de prendre en charge un plus grand nombre de configurations de workflow. Par exemple, au lieu d'envoyer des données aux workers en round-robin (?), il peut être plus intéressant de chercher le worker le plus proche physiquement d'un processus donné pour recevoir ses données. Une autre possibilité consiste à répartir les workers sur la liste des nœuds alloués au lieu de les placer tous dans les N premiers nœuds de la liste. Par exemple, l'attribution d'un nœud de workers tous les deux nœuds de simulation devrait réduire la variabilité, car la probabilité que deux nœuds communicants soient connectés au même switch est plus importante.

La mise en œuvre des bridges DEISA peut également être améliorée. Actuellement, les bridges sont implementés audessus des clients Dask légers. On peut concevoir une nouvelle classe dans Dask pour implémenter les bridges et les faire communiquer sans passer par le scheduler. Par exemple, l'utilisation de MPI pour la communication entre les brdiges peut être meilleure que la version actuelle, où nous utilisons des variables Dask qui sont hébergées dans le scheduler centralisé. Il convient de noter que MPI peut également être utilisé comme couche de communication dans Dask [152]. Le `scatter` peut également être amélioré. Par exemple, au lieu de laisser les bridges informer le scheduler des nouvelles données envoyées aux workers, une amélioration possible consiste à faire en sorte que ce soit le worker qui s'en charge plutôt que la brdige. Cela est possible dans l'implémentation a graphe unique grâce au mécanisme de contrat qui garantit que les données nécessaires *keys* sont déclarées par le client d'analyse principal a priori. Nous nous assurons ainsi que le ramasse-miettes ne supprimera pas les données car un client au moins en a besoin. Une autre limitation que nous avons rencontrée est liée à la sérialisation des données dans le `scatter`, qui est inefficace pour les données non contiguës. Pour améliorer cet aspect, il est possible de mettre en œuvre un meilleur algorithme de sérialisation ou d'envoyer le tableau contigu complet et d'effectuer la sélection du côté des workers.

DEISA peut être déployé dans un plus grand nombre de cas d'utilisation, que ce soit en termes d'hétérogénéité logicielle ou matérielle. Elle a été utilisé avec le post hoc dans l'ARK2-MHD, qui est un cas d'utilisation intéressant de workflows hybride en transit/post hoc, ce qui est assez courant dans le HPC. Cela est possible et facile sans modifier le code MPI grâce à PDI et à sa configuration dans un fichier séparé. Une amélioration à envisager est la prise en charge des workflows hybrides in situ/in transit afin de réduire la variabilité et de construire des workflows plus intéressants pour certaines analyses. Par exemple, si la première étape de l'analyse est une réduction des données, il peut être plus intéressant d'effectuer ces réductions in situ, puis d'envoyer des données plus petites pour un traitement in transit. Plusieurs façons de prendre en charge ces workflows peuvent être envisagées, comme l'exécution des réductions dans les bridges DEISA (configuration synchrone) ou dans un worker déployé dans un cœur dédié (configuration asynchrone). Le défi consiste ici à identifier les tâches à exécuter in situ et les tâches à exécuter in transit. En fonction du graphe des tâches, le choix peut être compliqué. L'exemple le plus simple est celui où le premier niveau du graphe de tâches (feuilles) ne contient que des tâches de réduction. On peut forcer toutes les tâches sans dépendances à s'exécuter dans le worker le plus proche (worker in situ, ou dans le bridge si on ne dédie pas de resources à ces tâches) puis envoyer les autres dans les worker in transit. Un autre exemple est celui où la réduction vient après d'autres analyses, par exemple, une multiplication locale de matrice par un entier suivie d'une réduction. Dans ce cas, on peut exécuter les deux premières couches in situ, puis passer in transit. Cependant, il n'est pas toujours facile de déterminer de manière optimale quand commencer à exécuter les tâches in transit. Une solution possible est de faire en sorte que DEISA prenne main dans l'optimisation du graphe de tâches, après avoir invoquer les méthodes `compute`, où elle analyse et décide de forcer ou non les tâches à s'exécuter sur les workers in situ. Les informations concernant le worker le plus proche d'un processus MPI doivent être disponibles à ce stade.

Les supercalculateurs deviennent de plus en plus hétérogènes, et il est important d'en tenir compte dans la conception des logiciels. DEISA tire parti de toutes les capacités de Dask, comme la possibilité d'utiliser les GPU dans l'analyse. Dask peut utiliser les GPU de plusieurs façons : par exemple en utilisant des bibliothèques optimisées pour les GPU (comme Pytorch et TensorFlow) grâce aux `Delayed` et `Future`, ou la bibliothèque `cuDF` similaire a Pandas, qui interagit bien et est testée contre Dask `dataframe`[1]. Dans le contexte in situ, si la simulation est lancée uniquement dans les CPU et que les nœuds de calcul contiennent également un ou plusieurs GPU, il est alors possible d'utiliser ces derniers pour l'analyse des données grâce à DEISA. Il est important de noter que nous avons obtenu toutes ces fonctionnalités et d'autres capacités de Dask sans plus d'effort et uniquement grace au bridging model.

DEISA peut être exploré dans une configuration multi-producteurs/consommateurs. Par exemple, pour effectuer des analyses d'ensemble, nous connectons plusieurs instances de simulation au même cluster Dask et collectons les données de toutes ces instances. Avec la version actuelle de DEISA, cela peut être facilement mis en œuvre en ajoutant simplement l'identifiant de l'instance de simulation dans les clés de données afin de reconna5tre la source des données à chaque fois qu'elles sont reçues. Là encore, il est possible de tirer parti de l'API `dask.array` API ou de `dask.dataframe` API et de tout l'écosystème Dask en fonction des besoins.

L'analyse in situ réduit le problème du goulot d'étranglement des entrées-sorties. Elle n'est cependant pas parfaite, car les scientifiques doivent conna5tre à l'avance les analyses a effectuer, ce qui n'est pas toujours le cas lorsqu'ils essaient de comprendre le problème lui-même. On pourrait tirer parti de *triggers* [**?**] parallèlement aux workflows hybrides dans de telles situations. Les triggers in situ sont des mécanismes qui personnalisent le workflows en fonction de situations et d'événements spécifiques. Ils déclenchent des actions distinctes en réponse à des conditions satisfaites. La réponse peut être n'importe quel type d'analyse, de contrôle ou d'entrées-sorties. DEISA peut être utilisé pour détecter des événements rares ou des comportements étranges dans les simulations et déclencher des analyses spécifiques si nous les connaissons déjà, ou un checkpoint pour les analyser en post hoc. Si des erreurs logiques ou physiques sont détectées, la simulation peut être arrêtée pour éviter de perdre du temps et de l'énergie. Le mécanisme de trigger peut être mis en œuvre grâce à la création dynamique de tâches dans Dask en utilisant la fonctionnalité `worker-client` [2]. Elle crée un client au sein d'un worker (un serveur en interne) et soumet de nouvelles tâches au scheduler si une condition est remplie [3]. Cette fonctionnalité peut être facilement utilisée pour déclencher de nouvelles analyses en ligne lorsqu'un événement rare est détecté, ou pour piloter la simulation.

Ce travail a également rapproché les communautés HPC et Big Data en couplant les programmes

[1] https://docs.dask.org/en/stable/gpu.html
[2] https://distributed.dask.org/en/stable/task-launch.html#connexion-with-context-manager
[3] https://distributed.dask.org/en/stable/task-launch.html

MPI avec l'outil Big Data Dask. Nous participons ainsi, avec notre modèle, à la convergence de ces deux communautés sans avoir à réimplémenter l'une ou l'autre de leurs outils. Ce travail montre un moyen de faire travailler et coopérer HPC et Big Data pour une finalité commune dans un workflows unique tout en préservant les caractéristiques de chacun. En outre, le couplage de modèles de programmation aussi puissants est une bonne étape pour tirer parti des deux, en fonction des exigences de l'application. Par exemple, l'approche que nous avons proposée peut être appliquée en dehors du contexte du traitement in situ, notamment en tirant parti du dynamisme de la programmation par tâches pour une partie d'un code MPI volumineux où la haute performance n'est pas indispensable.

L'approche DEISA peut également être utilisée dans d'autres directions de recherche. Par exemple, elle peut être explorée dans la direction de la convergence HPC/IA, où le modèle DEISA déjà existant peut être utilisé pour coupler des simulations HPC avec des modèles ML, soit dans la direction de l'IA pour HPC, où les modèles IA peuvent être utilisés pour accélérer des parties du code HPC, soit dans la direction HPC pour IA, où les données de simulation peuvent alimenter les modèles d'entra5nement IA. Le modèle résout certains problèmes de convergence HPC/AI liés au couplage lui-même : il n'est plus nécessaire de se préoccuper de la différence entre les langages de programmation : C/C++ ou Fortran pour le calcul intensif et Python/Dask pour l'IA. Les modèles de programmation sont également cachés : le parallélisme, la distribution des données et les communications sont tous gérés par DEISA. Actuellement, un workflows HPC/AI, dans la direction de l'IA pour HPC, peut être facilement mis en place en utilisant des modèles pré-entra5nés existants. Un exemple d'utilisation de modèles pré-entra5nés dans PyTorch au sein de Dask peut être trouvé dans le blog Dask [4]. DEISA peut également être utilisé pour alimenter le processus d'apprentissage automatique de manière incrémentale. Cela peut nécessiter une expertise supplémentaire en matière d'IA car les données que nous obtenons à partir d'un pas de temps ne seront pas nécessairement sauvegardées sur le disque et donc disponibles plus tard. Il est possible d'envisager une réduction de la dimensionnalité afin de réduire la taille des données à conserver en mémoire. Notez que le temps d'apprentissage devrait être réduit parce que nous le faisons en ligne et que nous évitons les entrées-sorties.

DEISA peut être utilisé en dehors de l'analyse de données; elle peut être déployée dans des supercalculateurs pour collecter des données en temps réel sur les applications en cours d'exécution et générer des logs et des rapports pour apprendre des caractéristiques spécifiques pour la reproductibilité ou toute autre préoccupation. Un bridge DEISA peut être associé à une applications en cours, il peut collecter des données à partir du job scheduler, ainsi que toutes les données brutes et en temps réel pertinentes. Ces données peuvent ensuite être transférées aux workers pour un éventuel prétraitement avant d'être incluses dans des rapports de diagnostic ou simplement les écrire dans une base de données. Dans ce cas d'utilisation, on peut également imaginer la génération de rapports dynamiques et réactifs si des données étranges ont été détectées. Par exemple, si la durée de calcul est plus longue que d'habitude, des données supplémentaires peuvent être demandées au bridges associé concernant les communications ou les entrées-sorties, etc. En termes plus généraux, DEISA peut être utilisé pour la collecte de données et l'analyse en ligne car elle peut être intégré facilement dans une plateforme HPC avec des applications MPI et offre une facilité d'utilisation pour le traitement des données.

Une autre perspective qui peut être envisagée concerne le bridging model DEISA. Il peut être généralisé pour un plus grand nombre de configurations de couplage de code entre différents modèles de programmation. La première étape consiste à faire fonctionner le modèle dans l'autre sens, où le modèle par tâches produit les données et l'application MPI les consomment. ce model peut être intéressant dans les workflows où la simulation est pilotée en ligne par des analyses in situ. Ou dans les codes par tâches où nous voulons accélérer une partie donnée du code à l'aide de MPI. Dans une telle configuration, des efforts supplémentaires doivent être faits pour garantir que les données externes provenant du programme par tâches ne bloqueront pas le programme MPI.

Le modèle DEISA peut également être étendu à d'autres modèles de programmation dans l'idée de tirer parti des capacités de ces modèles. Cela peut s'avérer utile lorsqu'il s'agit d'applications lourdes traitant plusieurs préoccupations différentes. Une telle possibilité est intéressante, par exemple, pour la génération de code de haut niveau lorsque l'utilisateur souhaite écrire du code en utilisant différents modèles de programmation. Au lieu de considérer uniquement la génération de code pour des backends hétérogènes, on peut imaginer la génération de code pour des modèles de programmation distribués hétérogènes. Cette idée peut être explorée pour générer des squelettes des workflows souhaités avec les modèles de programmation et les outils souhaités et toutes les configurations nécessaires pour le couplage du code. Le code généré peut ensuite être utilisé par les scientifiques du domaine pour développer les fonctionnalités séparément.

[4] https://blog.dask.org/2021/03/29/apply-pretrained-pytorch-model

Une autre perspective qui peut être envisagée concerne les tâches externes que nous avons introduites dans Dask. Ce concept est suffisamment générique pour recevoir des données de n'importe quelle source externe, et pas seulement les données de simulation MPI. On peut envisager de l'utiliser dans d'autres contextes, tels que la mise en œuvre des workflows des jumeaux numériques. La même instance de Dask peut être alimentée à la fois par des simulations en cours et par des appareils réels. Un état de l'art intéressant avec des études et des défis théoriques et pratiques peut être trouvé dans [154]. Les tâches externes dans Dask n'ont aucune restriction pour le moment, et les données reçues ne sont pas vérifiées. Elles peuvent donc présenter des problèmes de cybersécurité. Dask distribué supporte déjà les communications TLS/SSH entre clients/scheduler/workers [5], mais dans le contexte des jumeaux numériques, une validation de sécurité supplémentaire peut être nécessaire à la fois au niveau de Dask et des appareils.

Au terme de cette thèse, nous voudrions souligner que DEISA n'est pas seulement un outil, mais un modèle à garder en tete. Les analyses de données in situ sont des analyses de données ; au lieu de recréer toute la pile d'analyse de données pour l'in situ, il peut être plus facile et moins coûteux de faire fonctionner la pile d'analyse de données in situ. Et l'idée peut être généralisée pour d'autres situations similaires.

[5] https://distributed.dask.org/en/stable/tls.html

Bibliography

[1] dask-ml 0.1 documentation - dask_ml.decomposition.IncrementalPCA. URL modules/generated/dask_ml.decomposition.IncrementalPCA.html.

[2] Jisc new technology initiative proposal.

[3] Pdi documentation. URL https://pdi.julien-bigot.fr/master/.

[4] catalyst user's guide paraview. URL https://www.paraview.org/files/catalyst/docs/ParaViewCatalystUsersGuide_v2.pdf.

[5] ENIAC team - Détours. URL https://detours.canal.fr/saviez-premier-ordinateur-de-lhistoire-a-ete-cree-femmes/eniac-team/.

[6] Frontier. URL https://www.olcf.ornl.gov/frontier/.

[7] What is HPC? Introduction to high-performance computing | IBM. URL https://www.ibm.com/topics/hpc.

[8] IO500 - Submissions. URL https://io500.org/submissions/graphs.

[9] large parallel/collective data set crashes and lustre striping - HDF5 - HDF Forum. URL https://forum.hdfgroup.org/t/large-parallel-collective-data-set-crashes-and-lustre-striping/10602/12.

[10] Lustre. URL https://www.lustre.org/.

[11] About VisIt, . URL https://visit-dav.github.io/visit-website/about/.

[12] Ascent — Ascent 0.8.0 documentation, . URL https://ascent.readthedocs.io/en/latest/.

[13] A Beginner's Guide to Eigenvectors, Eigenvalues, PCA, Covariance and Entropy, . URL http://wiki.pathmind.com/eigenvector.

[14] Conduit — Conduit 0.8.5 documentation, . URL https://llnl-conduit.readthedocs.io/en/latest/.

[15] Dask documentation - Chunks, . URL array-chunks.html.

[16] Dask.distributed — Dask.distributed 2022.10.2 documentation, . URL https://distributed.dask.org/en/stable/.

[17] ParaView: ParaView In Situ, . URL https://kitware.github.io/paraview-docs/latest/cxx/group__Insitu.html.

[18] PDI: Core Concepts, . URL https://pdi.dev/master/Concepts.html.

[19] Principal Component Analysis (PCA) Explained | Built In, . URL https://builtin.com/data-science/step-step-explanation-principal-component-analysis.

[20] Ray AIR Technical Whitepaper, . URL https://docs.google.com/document/d/1bYL-638GN6EeJ45dPuLiPImA8msojEDDKiBx3YzB4_s/preview?usp=embed_facebook.

[21] Ray v2 Architecture, . URL https://docs.google.com/document/d/1tBw9A4j62ruI5omIJbMxly-la5w4q_TjyJgJL_jN2fI/preview?usp=embed_facebook.

[22] sensei, . URL https://sensei-insitu.org/index.html.

[23] Using Dask on Ray — Ray 2.2.0, . URL https://docs.ray.io/en/latest/data/dask-on-ray.html.

[24] Using ParaView for High-Performance Computing + In Situ, . URL https://www.paraview.org/hpc-insitu/.

[25] 1. Understanding how VisIt works — VisIt User Manual 3.2.2 documentation, . URL https://visit-sphinx-github-user-manual.readthedocs.io/en/develop/intro/Understanding_how_VisIt_works.html.

[26] VTK - The Visualization Toolkit, . URL https://vtk.org/.

[27] What is Catalyst? — Catalyst documentation, . URL https://catalyst-in-situ.readthedocs.io/en/latest/introduction.html.

[28] Redis. URL https://redis.io/.

[29] Cluster overview - Mesocentre Documentation. URL https://mesocentre.pages.centralesupelec.fr/user_doc/ruche/01_cluster_overview/.

[30] Introduction — SmartSim 0.4.1 documentation. URL https://www.craylabs.org/docs/overview.html.

[31] Slurm Workload Manager - srun. URL https://slurm.schedmd.com/srun.html#OPT_relative.

[32] Using Machine Learning at scale in numerical simulations with SmartSim: An application to ocean climate modeling | Elsevier Enhanced Reader. URL https://reader.elsevier.com/reader/sd/pii/S1877750322001065?token=3C2209B5F0A6FC2C970A8433BDA25929224F81C74C20F9A352B3AF7661C49FBD9AC6152AB10B92F8CD6F436671CA3565originRegion=eu-west-1&originCreation=20221027124459.

[33] June 2022 | TOP500. URL https://www.top500.org/lists/top500/2022/06/.

[34] SENSEI-insitu/SENSEI, Dec. 2022. URL https://github.com/SENSEI-insitu/SENSEI. original-date: 2021-06-03T15:25:01Z.

[35] pdidev / PDI Data Interface · GitLab, Dec. 2022. URL https://gitlab.maisondelasimulation.fr/pdidev/pdi.

[36] Kitware/ParaViewCatalystExampleCode-MOVED-, Nov. 2022. URL https://github.com/Kitware/ParaViewCatalystExampleCode-MOVED-. original-date: 2013-03-12T15:18:32Z.

[37] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with nimrod/g: killer application for the global grid? In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 520–528, 2000. doi: 10.1109/IPDPS.2000.846030.

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

[39] J. Allard and B. Raffin. Distributed Physical Based Simulations for Large VR Applications. In *IEEE Virtual Reality Conference*, Alexandria, USA, Mar. 2006.

[40] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. FlowVR: a Middleware for Large Scale Virtual Reality Applications. In *Proceedings of Euro-par 2004*, Pisa, Italia, Aug. 2004.

[41] G. Amal. Distributed, Jan. 2022. URL https://github.com/GueroudjiAmal/distributed. original-date: 2021-12-02T09:44:00Z.

[42] S. Archipoff, C. Augonnet, O. Aumage, G. Beauchamp, B. Bramas, A. Buttari, A. Cassagne, J. Clet-Ortega, T. Cojean, N. Collin, et al. Starpu. 2017.

[43] T. Arcila, J. Allard, C. MA(C)nier, E. Boyer, and B. Raffin. FlowVR: A Framework For Distributed Virtual Reality Applications. In *1iAre journA(C)es de l'Association FranAaise de RA(C)alitA(C) Virtuelle, AugmentA(C)e, Mixte et d'Interaction 3D*, Rocquencourt, France, Nov. 2006.

[44] 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*, pages 863–874. Springer, 2009.

[45] C. Augonnet, S. Thibault, and R. Namyst. *StarPU: a runtime system for scheduling tasks over accelerator-based multicore machines*. PhD thesis, INRIA, 2010.

[46] U. Ayachit, B. Whitlock, M. Wolf, B. Loring, B. Geveci, D. Lonie, and E. W. Bethel. The SENSEI Generic In Situ Interface. In *2016 Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*, pages 40–44, Salt Lake City, UT, USA, Nov. 2016. IEEE. ISBN 978-1-5090-3872-5. doi: 10.1109/ISAV.2016.013. URL http://ieeexplore.ieee.org/document/7836400/.

[47] U. Ayachit, A. C. Bauer, B. Boeckel, B. Geveci, K. Moreland, P. O'Leary, and T. Osika. Catalyst revised: Rethinking the paraview in situ analysis and visualization api. In *International Conference on High Performance Computing*, pages 484–494. Springer, 2021.

[48] Y. Babuji, A. Brizius, K. Chard, I. Foster, D. S. Katz, M. Wilde, and J. Wozniak. Introducing parsl: a python parallel scripting library. *Zenodo2017*, 2017.

[49] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard. Parsl: Pervasive Parallel Programming in Python. *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 25–36, June 2019. doi: 10.1145/3307681.3325400. URL http://arxiv.org/abs/1905.02158. arXiv: 1905.02158.

[50] Y. N. Babuji, K. Chard, I. T. Foster, D. S. Katz, M. Wilde, A. Woodard, and J. M. Wozniak. Parsl: Scalable parallel scripting in python. In *IWSG*, 2018.

[51] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel. In situ methods, infrastructures, and applications on high performance computing platforms. *Computer Graphics Forum*, 35(3):577–597, 2016. doi: https://doi.org/10.1111/cgf.12930. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12930.

[52] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Salt Lake City, UT, Nov. 2012. IEEE. ISBN 978-1-4673-0805-2 978-1-4673-0806-9. doi: 10.1109/SC.2012.71. URL http://ieeexplore.ieee.org/document/6468504/.

[53] y. Bauer, r. Wissink, M. Potsdam, and B. J. on. ParaView Catalyst Computes Particle Paths In Situ, Oct. 2016. URL https://blog.kitware.com/paraview-catalyst-computes-particle-paths-in-situ/.

[54] E. Belikov, P. Deligiannis, P. Totoo, M. Aljabri, and H.-W. Loidl. A survey of high-level parallel programming models. *Heriot-Watt University, Edinburgh, UK*, 1(2):2–2, 2013.

[55] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–9, Nov. 2012. doi: 10.1109/SC.2012.31. ISSN: 2167-4337.

[56] E. W. Bethel, B. Loring, U. Ayachit, D. Camp, E. P. N. Duque, N. Ferrier, J. Insley, J. Gu, J. Kress, P. O'Leary, D. Pugmire, S. Rizzi, D. Thompson, G. H. Weber, B. Whitlock, M. Wolf, and K. Wu. The SENSEI Generic In Situ Interface: Tool and Processing Portability at Scale. In H. Childs, J. C. Bennett, and C. Garth, editors, *In Situ Visualization for Computational Science*, pages 281–306, Cham, 2022. Springer International Publishing. ISBN 978-3-030-81627-8.

[57] J. Bigot, V. Grandgirard, G. Latu, C. Passeron, F. Rozar, and O. Thomine. Scaling gysela code beyond 32k-cores on blue gene/q. In *ESAIM: PROCEEDINGS*, volume CEMRACS 2012 of *43*, pages 117–135, Luminy, France, July 2012. doi: 10.1051/proc/201343007. URL https://hal.inria.fr/hal-01050322.

[58] S. Blackheath. *Functional reactive programming*. Simon and Schuster, 2016.

[59] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[60] D. A. Boyuka, S. Lakshminarasimham, X. Zou, Z. Gong, J. Jenkins, E. R. Schendel, N. Podhorszki, Q. Liu, S. Klasky, and N. F. Samatova. Transparent in Situ Data Transformations in ADIOS. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 256–266, May 2014. doi: 10.1109/CCGrid.2014.73.

[61] M. Brinskiy, M. Lubin, and J. Dinan. Chapter 16 - mpi-3 shared memory programming introduction. In J. Reinders and J. Jeffers, editors, *High Performance Parallelism Pearls*, pages 305–319. Morgan Kaufmann, Boston, 2015. ISBN 978-0-12-803819-2. doi: https://doi.org/10.1016/B978-0-12-803819-2.00025-2. URL https://www.sciencedirect.com/science/article/pii/B9780128038192000252.

[62] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[63] H. Childs. VisIt: An End-User Tool for Visualizing and Analyzing Very Large Data. page 17.

[64] H. Childs. *In Situ Visualization for Computational Science*. Springer Nature, 2022. ISBN 978-3-030-81627-8. Google-Books-ID: JURuEAAAQBAJ.

[65] H. Childs, J. C. Bennett, and C. Garth. In situ visualization for computational science: Background and foundational topics. In *In Situ Visualization for Computational Science*, pages 1–8. Springer, 2022.

[66] J. A. Cid-Fuentes, S. Sola, P. Alvarez, A. Castro-Ginard, and R. M. Badia. dislib: Large scale high performance machine learning in python. In *2019 15th International Conference on eScience (eScience)*, pages 96–105, Sept. 2019. doi: 10.1109/eScience.2019.00018.

[67] S. M. B. t. f. t. comment. Oak Ridge's exascale 'Frontier' system named world's most powerful supercomputer on Top500. URL https://www.datacenterdynamics.com/en/news/oak-ridges-exascale-frontier-system-named-worlds-most-powerful-supercomputer-on-top500/.

[68] cscsch. In Situ Analysis and Visualization with ParaView Catalyst and Ascent - Part 2, 2022. URL https://www.youtube.com/watch?v=SFgest3c-ck.

[69] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[70] E. Dirand. *Integration of High-Performance Task-Based In Situ for Molecular Dynamics on Exascale Computers*. Theses, Université Grenoble Alpes, Nov. 2018. URL https://hal.archives-ouvertes.fr/tel-01949170. Issue: 2018GREAM065.

[71] E. Dirand. *Integration of high-performance task-based in situ for molecular dynamics on exascale computers*. PhD thesis, Université Grenoble Alpes, 2018.

[72] 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*, pages 159–178. Springer, 2018.

[73] E. Dirand, L. Colombet, and B. Raffin. TINS: A Task-Based Dynamic Helper Core Strategy for In Situ Analytics. In R. Yokota and W. Wu, editors, *Supercomputing Frontiers*, volume 10776, pages 159–178. Springer International Publishing, Cham, 2018. ISBN 978-3-319-69952-3 978-3-319-69953-0. doi: 10.1007/978-3-319-69953-0_10. URL http://link.springer.com/10.1007/978-3-319-69953-0_10. Series Title: Lecture Notes in Computer Science.

[74] C. Docan, M. Parashar, and S. Klasky. Dart: A substrate for high speed asynchronous data io. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, page 219–220, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939975. doi: 10.1145/1383422.1383454. URL https://doi.org/10.1145/1383422.1383454.

[75] C. Docan, M. Parashar, and S. Klasky. Enabling high-speed asynchronous data extraction and transfer using dart. *Concurrency and Computation: Practice and Experience*, 22(9):1181–1204, 2010.

[76] C. Docan, M. Parashar, and S. Klasky. DataSpaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 15(2):163–181, June 2012. ISSN 1573-7543. doi: 10.1007/s10586-011-0162-y. URL https://doi.org/10.1007/s10586-011-0162-y.

[77] J. Dongarra. Trends in high-performance computing: A historical overview and examination of future developments. *Circuits and Devices Magazine, IEEE*, 22:22–27, Feb. 2006. doi: 10.1109/MCD.2006.1598076.

[78] M. Dorier. *Addressing the Challenges of I/O Variability in Post-Petascale HPC Simulations*. phdthesis, Ecole Normale Supérieure de Rennes, Dec. 2014. URL https://tel.archives-ouvertes.fr/tel-01099105.

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

[80] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf. Damaris: Leveraging Multicore Parallelism to Mask I/O Jitter. Research Report RR-7706, INRIA, Apr. 2012. URL https://hal.inria.fr/inria-00614597.

[81] 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)*, pages 67–75, 2013. doi: 10.1109/LDAV.2013.6675160.

[82] M. Dorier, M. Dreher, T. Peterka, J. M. Wozniak, G. Antoniu, and B. Raffin. Lessons Learned from Building In Situ Coupling Frameworks. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization - ISAV2015*, pages 19–24, Austin, TX, USA, 2015. ACM Press. ISBN 978-1-4503-4003-8. doi: 10.1145/2828612.2828622. URL http://dl.acm.org/citation.cfm?doid=2828612.2828622.

[83] M. Dorier, G. Antoniu, F. Cappello, M. Snir, R. Sisneros, O. Yildiz, S. Ibrahim, T. Peterka, and L. Orf. Damaris: Addressing performance variability in data management for post-petascale simulations. *ACM Trans. Parallel Comput.*, 3(3), oct 2016. ISSN 2329-4949. doi: 10.1145/2987371. URL https://doi.org/10.1145/2987371.

[84] M. Dreher. *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), Feb. 2015. URL https://www.theses.fr/2015GREAM076.

[85] M. Dreher and T. Peterka. Bredala: Semantic Data Redistribution for In Situ Applications. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 279–288, Sept. 2016. doi: 10.1109/CLUSTER.2016.30. ISSN: 2168-9253.

[86] M. Dreher and T. Peterka. Decaf: Decoupled Dataflows for In Situ High-Performance Workflows. Technical Report ANL/MCS-TM-371, Argonne National Lab. (ANL), Argonne, IL (United States), July 2017. URL https://www.osti.gov/biblio/1372113-decaf-decoupled-dataflows-situ-high-performance-workflows.

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

[88] M. Dreher, J. Prevoteau-Jonquet, M. Trellet, M. Piuzzi, M. Baaden, B. Raffin, N. Férey, S. Robert, and S. Limet. ExaViz: a Flexible Framework to Analyse, Steer and Interact with Molecular Dynamics Simulations. *Faraday Discussions*, 169:119–142, May 2014. doi: 10.1039/C3FD00142C. URL https://hal.inria.fr/hal-00942627.

[89] J. Ejarque, R. M. Badia, L. Albertin, G. Aloisio, E. Baglione, Y. Becerra, S. Boschert, J. R. Berlin, A. D'Anca, D. Elia, F. Exertier, S. Fiore, J. Flich, A. Folch, S. J. Gibbons, N. Koldunov, F. Lordan, S. Lorito, F. Løvholt, J. Macías, F. Marozzo, A. Michelini, M. Monterrubio-Velasco, M. Pienkowska, J. d. l. Puente, A. Queralt, E. S. Quintana-Ortí, J. E. Rodríguez, F. Romano, R. Rossi, J. Rybicki, M. Kupczyk, J. Selva, D. Talia, R. Tonini, P. Trunfio, and M. Volpe. Enabling dynamic and intelligent workflows for HPC, data analytics, and AI convergence. *Future Generation Computer Systems*, 134:414–429, 2022. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2022.04.014. URL https://www.sciencedirect.com/science/article/pii/S0167739X22001364.

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

[91] S. Erotokritou. HPC supports first black hole image, Nov. 2019. URL https://prace-ri.eu/hpc-supports-first-black-hole-image/.

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

[93] M. Folk, A. Cheng, and K. Yates. Hdf5: A file format and i/o library for high performance computing applications. In *Proceedings of Supercomputing*, volume 99, pages 5–33, 1999.

[94] B. Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of Computation*, 51(184):699–706, 1988. ISSN 0025-5718, 1088-6842. doi: 10.1090/S0025-5718-1988-0935077-0. URL https://www.ams.org/mcom/1988-51-184/S0025-5718-1988-0935077-0/.

[95] E. Friedman and K. Tzoumas. *Introduction to Apache Flink: stream processing for real time and beyond.* " O'Reilly Media, Inc.", 2016.

[96] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, may 1998. ISSN 0362-1340. doi: 10.1145/277652.277725. URL https://doi.org/10.1145/277652.277725.

[97] X. Garbet, Y. Idomura, L. Villard, and T. Watanabe. Gyrokinetic simulations of turbulent transport. *Nuclear Fusion*, 50(4):043002, mar 2010. doi: 10.1088/0029-5515/50/4/043002. URL https://dx.doi.org/10.1088/0029-5515/50/4/043002.

[98] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, K. Wu, and S. Klasky. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. *SoftwareX*, 12:100561, July 2020. ISSN 2352-7110. doi: 10.1016/j.softx.2020.100561. URL https://www.sciencedirect.com/science/article/pii/S2352711019302560.

[99] V. Grandgirard, J. Abiteboul, J. Bigot, T. Cartier-Michaud, N. Crouseilles, G. Dif-Pradalier, C. Ehrlacher, D. Esteve, X. Garbet, P. Ghendrih, G. Latu, M. Mehrenberger, C. Norscini, C. Passeron, F. Rozar, Y. Sarazin, E. Sonnendrücker, A. Strugarek, and D. Zarzoso. A 5d gyrokinetic full-f global semi-lagrangian code for flux-driven ion turbulence simulations. *Computer Physics Communications*, 207:35–68, 2016. doi: 10.1016/j.cpc.2016.05.007. URL http://dx.doi.org/10.1016/j.cpc.2016.05.007.

[100] W. D. Gropp. Learning from the Success of MPI. In G. Goos, J. Hartmanis, J. van Leeuwen, B. Monien, V. K. Prasanna, and S. Vajapeyam, editors, *High Performance Computing — HiPC 2001*, volume 2228, pages 81–92. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-43009-4 978-3-540-45307-9. doi: 10.1007/3-540-45307-5_8. URL http://link.springer.com/10.1007/3-540-45307-5_8. Series Title: Lecture Notes in Computer Science.

[101] 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)*, pages 11–20, 2021. doi: 10.1109/HiPC53243.2021.00015.

[102] J. Gurhem. *Paradigmes de programmation répartie et parallèle utilisant des graphes de tâches pour supercalculateurs post-pétascale*. PhD thesis, 2021. URL http://www.theses.fr/2021LILUI005. Thèse de doctorat dirigée par Petiton, Serge Informatique et applications Université de Lille (2018-2021) 2021.

[103] A. Heirich, E. Slaughter, M. Papadakis, W. Lee, T. Biedert, and A. Aiken. In situ visualization with task-based parallelism. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, pages 17–21. 2017.

[104] A. J. G. Hey. High Performance Computing – Past, Present and Future. page 11.

[105] P. Hintjens. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

[106] S. M. Ho. PRINCIPAL COMPONENTS A N ALYSI S (PCA). *Principal Components Analysis*.

[107] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra. Dynamic task discovery in parsec: A data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 1–8, 2017.

[108] H. Hotelling. Relations between two sets of variates. *Biometrika*, 28(3/4):321–377, 1936. ISSN 00063444. URL http://www.jstor.org/stable/2333955.

[109] S. Idreos, N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, A. Ross, J. Lennon, V. Jain, H. Gupta, D. Li, et al. Learning key-value store design. *arXiv preprint arXiv:1907.05443*, 2019.

[110] C. Jin, S. Klasky, S. Hodson, W. Yu, J. Lofstead, H. Abbasi, K. Schwan, M. Wolf, W.-k. Liao, A. Choudhary, et al. Adaptive io system (adios). *Cray User's Group*, 2008.

[111] I. T. Jolliffe and J. Cadima. Principal component analysis: a review and recent developments. *Philosophical transactions of the royal society A: Mathematical, Physical and Engineering Sciences*, 374(2065):20150202, 2016.

[112] S. Kamburugamuve, P. Wickramasinghe, S. Ekanayake, and G. Fox. Anatomy of machine learning algorithm implementations in mpi, spark, and flink. Technical report, 01 2017.

[113] H. Kasim, V. March, R. Zhang, and S. See. Survey on Parallel Programming Model. In J. Cao, M. Li, M.-Y. Wu, and J. Chen, editors, *Network and Parallel Computing*, Lecture Notes in Computer Science, pages 266–275, Berlin, Heidelberg, 2008. Springer. ISBN 978-3-540-88140-7. doi: 10.1007/978-3-540-88140-7_24.

[114] R. Ketata, L. Kriaa, and L. A. Saidane. Parallel Programming Models: A Survey. *International Journal of Engineering Research*, 4(04), 2016.

[115] T. Kooburat and M. Hwang. Extending Task-based Programming Model beyond Shared Memory Systems. page 7.

[116] O. Kramer. Scikit-learn. In *Machine learning for evolution strategies*, pages 45–53. Springer, 2016.

[117] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 65–76, 2009. doi: 10.1109/ISPASS.2009.4919639.

[118] M. Larsen, E. Brugger, H. Childs, J. Eliot, K. Griffin, and C. Harrison. Strawman: A Batch In Situ Visualization and Analysis Infrastructure for Multi-Physics Simulation Codes. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 30–35, Austin TX USA, Nov. 2015. ACM. ISBN 978-1-4503-4003-8. doi: 10.1145/2828612.2828625. URL https://dl.acm.org/doi/10.1145/2828612.2828625.

[119] M. Larsen, J. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison. The alpine in situ infrastructure: Ascending from the ashes of strawman. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, ISAV'17, page 42–46, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351393. doi: 10.1145/3144769.3144778. URL https://doi.org/10.1145/3144769.3144778.

[120] M. Larsen, A. Woods, N. Marsaglia, A. Biswas, S. Dutta, C. Harrison, and H. Childs. A flexible system for in situ triggers. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV '18, page 1–6, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365796. doi: 10.1145/3281464.3281468. URL https://doi.org/10.1145/3281464.3281468.

[121] M. Larsen, E. Brugger, H. Childs, and C. Harrison. Ascent: A Flyweight In Situ Library for Exascale Simulations. In *In Situ Visualization For Computational Science*, pages 255 – 279. Mathematics and Visualization book series from Springer Publishing, Cham, Switzerland, May 2022.

[122] G. Latu, J. Bigot, N. Bouzat, J. Gimenez, and V. Grandgirard. Benefits of smt and of parallel transpose algorithm for the large-scale gysela application. In *PASC '16 - Proceedings of the Platform for Advanced Scientific Computing Conference*, Lausanne, France, June 2016. ACM Press. doi: 10.1145/2929908.2929912. URL https://hal.archives-ouvertes.fr/hal-01834323.

[123] G. Latu, Y. ASAHI, J. Bigot, T. Fehér, and V. Grandgirard. 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, September 2018. URL https://hal.inria.fr/hal-01719208.

[124] M. Laufer and E. Fredj. High performance parallel i/o and in-situ analysis in the wrf model with adios2. *arXiv preprint arXiv:2201.08228*, 2022.

[125] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pages 39–39, 2003. doi: 10.1109/SC.2003.10053.

[126] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062. PMLR, 2018.

[127] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[128] J. Lofstead and R. Ross. Insights for exascale IO APIs from building a petascale IO API. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov. 2013. doi: 10.1145/2503210.2503238. ISSN: 2167-4337.

[129] K.-L. Ma, C. Wang, H. Yu, and A. Tikhonova. In-situ processing and visualization for ultrascale simulations. *Journal of Physics: Conference Series*, 78(1):012043, 2007. URL http://stacks.iop.org/1742-6596/78/i=1/a=012043.

[130] W. McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

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

[132] K. Moreland, C. Sewell, W. Usher, L. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K. Ma, H. Childs, M. Larsen, C. Chen, R. Maynard, and B. Geveci. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications*, 36(3): 48–58, May 2016. ISSN 1558-1756. doi: 10.1109/MCG.2016.48. Conference Name: IEEE Computer Graphics and Applications.

[133] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, Oct. 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/moritz.

[134] M. Nestmann. Building a Consistent Taxonomy for Parallel Programming Models. 2017. ISSN 1617-5468. doi: 10.18420/IN2017_245. URL https://dl.gi.de/handle/20.500.12116/4020. ISBN: 9783885796695 Publisher: Gesellschaft für Informatik, Bonn.

[135] T. E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007. doi: 10.1109/MCSE.2007.58.

[136] T. Padioleau, P. Tremblin, E. Audit, P. Kestener, and S. Kokh. A high-performance and portable all-mach regime flow solver code with well-balanced gravity. application to compressible convection. *The Astrophysical Journal*, 875(2):128, apr 2019. doi: 10.3847/1538-4357/ab0f2c. URL https://dx.doi.org/10.3847/1538-4357/ab0f2c.

[137] S. Partee, M. Ellis, A. Rigazzi, S. Bachman, G. Marques, A. Shao, and B. Robbins. Using machine learning at scale in hpc simulations with smartsim: An application to ocean climate modeling. *arXiv preprint arXiv:2104.09355*, 2021.

[138] K. Pearson. LIII. On lines and planes of closest fit to systems of points in space, Nov. 1901. URL https://doi.org/10.1080/14786440109462720.

[139] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[140] I. Perez, M. Bärenz, and H. Nilsson. Functional reactive programming, refactored. *ACM SIGPLAN Notices*, 51(12):33–44, 2016.

[141] S. Petruzza, S. Treichler, V. Pascucci, and P.-T. Bremer. Babelflow: An embedded domain specific language for parallel analysis and visualization. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 463–473, 2018. doi: 10.1109/IPDPS.2018.00056.

[142] A. Puurula. Benchmarking Python Distributed AI Backends with Wordbatch, June 2019. URL https://towardsdatascience.com/benchmarking-python-distributed-ai-backends-with-wordbatch-9872457b785c.

[143] D. Quintero, L. Bolinches, P. Chaudhary, W. Davis, S. Duersch, C. H. Fachim, A. Socoliuc, and O. Weiser. IBM Spectrum Scale (formerly GPFS). page 550.

[144] A. Rădulescu and A. J. Van Gemund. On the complexity of list scheduling algorithms for distributed-memory systems. In *Proceedings of the 13th international conference on Supercomputing*, pages 68–75, 1999.

[145] C. Ramon-Cortes, F. Lordan, J. Ejarque, and R. M. Badia. A Programming Model for Hybrid Workflows: combining Task-based Workflows and Dataflows all-in-one. *Future Generation Computer Systems*, 113:281–297, Dec. 2020. ISSN 0167739X. doi: 10.1016/j.future.2020.07.007. URL http://arxiv.org/abs/2007.04939. arXiv: 2007.04939.

[146] A. D. Robison. *Intel® Threading Building Blocks (TBB)*, pages 955–964. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4. doi: 10.1007/978-0-387-09766-4_51. URL https://doi.org/10.1007/978-0-387-09766-4_51.

[147] M. Rocklin. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. pages 126–132, Austin, Texas, 2015. doi: 10.25080/Majora-7b98e3ed-013. URL https://conference.scipy.org/proceedings/scipy2015/matthew_rocklin.html.

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

[149] sadmin. InfiniBand Roadmap - Advancing InfiniBand. URL https://www.infinibandta.org/infiniband-roadmap/.

[150] P. B. Schneck. *Supercomputer Architecture*, volume 31 of *The Kluwer International Series in Engineering and Computer Science*. Springer US, Boston, MA, 1987. ISBN 978-1-4615-7959-5 978-1-4615-7957-1. doi: 10.1007/978-1-4615-7957-1. URL http://link.springer.com/10.1007/978-1-4615-7957-1.

[151] W. J. Schroeder, L. S. Avila, and W. Hoffman. Visualizing with VTK: a tutorial. *IEEE Computer Graphics and Applications*, 20(5):20–27, Sept. 2000. ISSN 1558-1756. doi: 10.1109/38.865875. Conference Name: IEEE Computer Graphics and Applications.

[152] A. Shafi, J. Hashmi, H. Subramoni, D. K., and Panda. *Efficient MPI-based Communication for GPU-Accelerated Dask Applications*. Jan. 2021.

[153] S. Shankar. Looking into the Black Box: Holding Intelligent Agents Accountable [NUJS Law Review]. 10:451, Jan. 2017.

[154] A. Sharma, E. Kosasih, J. Zhang, A. Brintrup, and A. Calinescu. Digital twins: State of the art theory and practice, challenges, and open research questions. *Journal of Industrial Information Integration*, 30:100383, 2022. ISSN 2452-414X. doi: https://doi.org/10.1016/j.jii.2022.100383. URL https://www.sciencedirect.com/science/article/pii/S2452414X22000516.

[155] E. Slaughter and A. Aiken. Pygion: Flexible, Scalable Task-Based Parallelism with Python. In *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, pages 58–72, Denver, CO, USA, Nov. 2019. IEEE. ISBN 978-1-72815-979-9. doi: 10.1109/PAW-ATM49560.2019.00011. URL https://ieeexplore.ieee.org/document/9062721/.

[156] T. B. Sousa. Dataflow programming concept, languages and applications. In *Doctoral Symposium on Informatics Engineering*, volume 130, 2012.

[157] Q. Sun, M. Romanus, T. Jin, H. Yu, P.-T. Bremer, S. Petruzza, S. Klasky, and M. Parashar. In-Staging Data Placement for Asynchronous Coupling of Task-Based Scientific Workflows. In *2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2)*, pages 2–9, Nov. 2016. doi: 10.1109/ESPM2.2016.006.

[158] E. Tejedor. Infrastructure-Agnostic Programming and Interoperable Execution in Heterogeneous Grids. page 12.

[159] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta. Pycompss: Parallel computational workflows in python. *The International Journal of High Performance Computing Applications*, 31(1):66–82, 2017.

[160] T. Terraz, A. Ribes, Y. Fournier, B. Iooss, and B. Raffin. Melissa: Large Scale In Transit Sensitivity Analysis Avoiding Intermediate Files. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, pages 1 – 14, Denver, United States, Nov. 2017. URL https://hal.inria.fr/hal-01607479.

[161] R. Thakur, W. Gropp, and E. Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, IOPADS '99, page 23–32, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581131232. doi: 10.1145/301816.301826. URL https://doi.org/10.1145/301816.301826.

[162] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4):1422–1434, Apr. 2018. ISSN 0920-8542, 1573-0484. doi: 10.1007/s11227-018-2238-4. URL http://link.springer.com/10.1007/s11227-018-2238-4.

[163] tim.lewis. Home. URL https://www.openmp.org/.

[164] S. Treichler, M. Bauer, R. Sharma, E. Slaughter, and A. Aiken. Dependent partitioning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 344–358, 2016.

[165] J. L. Träff, S. Benkner, J. J. Dongarra, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, and G. Weikum, editors. *Recent Advances in the Message Passing Interface: 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, volume 7490 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-33517-4 978-3-642-33518-1. doi: 10.1007/978-3-642-33518-1. URL http://link.springer.com/10.1007/978-3-642-33518-1.

[166] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[167] S. Van Der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in science & engineering*, 13(2):22–30, 2011.

[168] A. Vitorović, M. V. Tomašević, and V. M. Milutinović. Chapter five - manual parallelization versus state-of-the-art parallelization techniques: The spec cpu2006 as a case study. volume 92 of *Advances in Computers*, pages 203–251. Elsevier, 2014. doi: https://doi.org/10.1016/B978-0-12-420232-0.00005-2. URL https://www.sciencedirect.com/science/article/pii/B9780124202320000052.

[169] M. N. Vora. Hadoop-hbase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 601–605. IEEE, 2011.

[170] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 242–252, 2000.

[171] Y. Wang, G. Agrawal, T. Bicer, and W. Jiang. Smart: a MapReduce-like framework for in-situ scientific analytics. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov. 2015. doi: 10.1145/2807591.2807650. ISSN: 2167-4337.

[172] B. Whitlock. Getting data into visit. *Lawrence Livermore National Laboratory*, 2010.

[173] B. Whitlock, J. M. Favre, and J. S. Meredith. Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System. In *11th Eurographics conference on Parallel Graphics and Visualization*, pages 101–109, 2011. ISBN 978-3-905674-32-3.

[174] D. Wu, S. Sakr, and L. Zhu. *Big Data Programming Models*, pages 31–63. Springer International Publishing, Cham, 2017. ISBN 978-3-319-49340-4. doi: 10.1007/978-3-319-49340-4_2. URL https://doi.org/10.1007/978-3-319-49340-4_2.

[175] Q. Wu, T. Neuroth, O. Igouchkine, K. Aditya, J. H. Chen, and K.-L. Ma. Diva: A Declarative and Reactive Language for In-Situ Visualization. *arXiv:2001.11604 [cs]*, Sept. 2020. URL http://arxiv.org/abs/2001.11604. arXiv: 2001.11604.

[176] X. Xing, B. Dong, J. Ajo-Franklin, and K. Wu. Automated parallel data processing engine with application to large-scale feature extraction. In *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, pages 37–46, 2018. doi: 10.1109/MLHPC.2018.8638638.

[177] B. Yan and R. Regueiro. Comparison between pure MPI and hybrid MPI-OpenMP parallelism for Discrete Element Method (DEM) of ellipsoidal and poly-ellipsoidal particles. *Computational Particle Mechanics*, 6, Nov. 2018. doi: 10.1007/s40571-018-0213-8.

[178] H. C. Zanúz, B. Raffin, O. A. Mures, and E. J. Padrón. In-transit molecular dynamics analysis with Apache flink. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 25–32, Dallas Texas USA, Nov. 2018. ACM. ISBN 978-1-4503-6579-6. doi: 10.1145/3281464.3281469. URL https://dl.acm.org/doi/10.1145/3281464.3281469.

[179] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.-A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu. FlexIO: Location-flexible Execution of In Situ Data Analytics for Large Scale Scientific Applications.

[180] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky. GoldRush: Resource Efficient in Situ Scientific Data Analytics Using Fine-grained Interference Aware Execution. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 78:1–78:12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2503279. URL http://doi.acm.org/10.1145/2503210.2503279.

École Doctorale
Mathématiques, Sciences et Technologies de l'Information, Informatique

Titre: Analyse de Données In Situ par Tâches Distribuées pour les Simulations Haute Performance

Mots clés: Programmation par tâche, Traitement in situ, simulation haute performance, Dask, Couplage de code, MPI

Résumé: Sur les systèmes à grande échelle, l'écart entre les performances des CPU et la de bande passante des disques ne cesse d'augmenter. Dans certains domaines, tels que les prévisions météorologiques et la fusion nucléaire, les modèles numériques génèrent des grandes quantités de données qu'un traitement post hoc classique n'est plus possible en raison des limites de la capacité de stockage et de la performance des entrées-sorties. Les approches in situ sont intéressantes pour contourner les accès aux disques dans ces cas et tirer pleinement parti de la plateforme HPC. Cependant, elles sont souvent complexes à mettre en place et peuvent nécessiter de redévelopper des versions parallèles des analyses. Dans notre travail, nous proposons un modèle qui est bien adapté aux traitements in situ ou nous couplons le modele MPI pour la simulation avec un paradigme par tâches distribuées pour l'analyse. Cela permet de réduire la complexité et de tirer le meilleur parti de chacun de ces deux paradigmes puissants. Nous proposons un modèle de couplage des deux paradigmes et le validons à l'aide d'un prototype appelé DEISA, qui permet de coupler des codes parallèles MPI avec des analyses écrites en Dask distribué. Le modèle ne nécessite que des modifications minimales des codes de simulation et d'analyse par rapport à leurs équivalents post hoc. Il donne accès à tout l'écosystème déjà existant à utiliser en in situ, comme les versions parallèles de Numpy, Pandas et scikit-learn. Nous introduisons de nouveaux concepts dans Dask distribué pour prendre en charge les analyses in situ de manière native. L'approche a été évaluée et comparée à des analyses post hoc sur deux supercalculateurs, et DEISA a été utilisé dans des cas de production. Les résultats sont intéressants et montrent de bonnes performances avec un minimum d'efforts de codage.

Title: Distributed Task-Based In Situ Data Analytics for High-Performance Simulations

Keywords: Task-based programming, In situ processing, High performance simulations, Dask, Code coupling, MPI

Abstract: A widening performance gap is separating CPU performance and IO bandwidth on large-scale systems. In some fields, such as weather forecast and nuclear fusion, numerical models generate such amounts of data that classical post hoc processing is not feasible anymore due to the limits in both storage capacity and IO performance. In situ approaches are attractive to bypass disk accesses in these cases and fully leverage the HPC platform. They are, however, often complex to set up and can require to re-develop parallel versions of the analysis from scratch. In our work, we propose a hybrid model that is well-suited for in situ workflows that combine regular simulations and irregular analytics. We couple the bulk synchronous parallel paradigm for simulation with a distributed task-based one for analysis. This reduces complexity and leverages the best of each of these two powerful paradigms. We propose a bridging model between the two paradigms and validate it through a prototype called DEISA, which supports coupling MPI parallel codes with analyses written using Dask. The bridging model requires minimal modifications of both the simulation and analysis codes compared to their post hoc counterpart. It gives access to an already existing rich ecosystem to be used in situ, such as the parallel versions of Numpy, Pandas and scikit-learn. We introduce new concepts in Dask distributed to support the in situ analytics natively. The approach has been evaluated and compared to post hoc analytics in two supercomputers, and DEISA has been used in production use cases. The results are quite interesting and show good performance with minimum coding efforts.

Université Grenoble Alpes
École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble
Titre: Analyses de données in situ par tâches distribuées pour les simulations haute performance