Future Generation Computer Systems 128 (2022) 282–298

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Distributed workflows with Jupyter

Iacopo Colonnelli [a,1,*], Marco Aldinucci [a,1], Barbara Cantalupo [a,1], Luca Padovani [a], Sergio Rabellino [a,1], Concetto Spampinato [b,1], Roberto Morelli [c], Rosario Di Carlo [c], Nicolò Magini [c], Carlo Cavazzoni [c]

[a] University of Torino, Computer Science Dept., Corso Svizzera 185, 10149, Torino, Italy
[b] University of Catania, Electrical Engineering Dept., Viale Andrea Doria 6, 95125, Catania, Italy
[c] Leonardo S.p.A., Piazza Monte Grappa 4, 00195, Roma, Italy

ARTICLE INFO

Article history:
Received 13 July 2021
Received in revised form 1 October 2021
Accepted 4 October 2021
Available online 14 October 2021

Keywords:
Distributed computing
Jupyter notebooks
Streamflow
Workflow management systems

ABSTRACT

The designers of a new coordination interface enacting complex workflows have to tackle a dichotomy: choosing a language-independent or language-dependent approach. Language-independent approaches decouple workflow models from the host code's business logic and advocate portability. Language-dependent approaches foster flexibility and performance by adopting the same host language for business and coordination code. Jupyter Notebooks, with their capability to describe both imperative and declarative code in a unique format, allow taking the best of the two approaches, maintaining a clear separation between application and coordination layers but still providing a unified interface to both aspects. We advocate the Jupyter Notebooks' potential to express complex distributed workflows, identifying the general requirements for a Jupyter-based Workflow Management System (WMS) and introducing a proof-of-concept portable implementation working on hybrid Cloud-HPC infrastructures. As a byproduct, we extended the vanilla IPython kernel with workflow-based parallel and distributed execution capabilities. The proposed Jupyter-workflow (Jw) system is evaluated on common scenarios for High Performance Computing (HPC) and Cloud, showing its potential in lowering the barriers between prototypical Notebooks and production-ready implementations.

1. Introduction

Jupyter Notebook's capability to unify imperative code and declarative metadata in a unique format puts them halfway between the two classes of tools commonly used for workflow modeling: high-level coordination languages and low-level distributed computing libraries. Also, Jupyter Notebooks come with a feature-rich, user-friendly web interface out-of-the-box, making them far more accessible for domain experts than the SSH-based remote shells commonly exposed by HPC facilities worldwide. Nevertheless, bare Jupyter Notebooks are unsuitable for both complex workflow modeling and large-scale executions. Indeed, the standard execution flow of Notebook cells is purely sequential, with all cells running on a single kernel instance and sharing a global program state. Furthermore, the standard transport layer, based on ZeroMQ, requires a bidirectional TCP connection between the frontend web server and the backend workers, an unlikely scenario in air-gapped data centers.

This work explores the Jupyter Notebook's potential to express complex workflows and coordinate their distributed execution on top of hybrid Cloud-HPC infrastructures. From the application perspective, we envision Jupyter Notebook as the first representative of a new class of interfaces to foster both portability and performance, mapping the Notebook cells to workflow steps to Cloud and HPC resources. This second passage leverages the existing state-of-the-art tools for both Cloud (e.g., Kubernetes and Dockers) and HPC (e.g., Slurm, PBS, and Singularity), inheriting their performance and portability. From the computational perspective, we extend the Jupyter Notebook kernel to support parallel and distributed execution of the Notebook cells, where a cell can drive the execution of a legacy parallel code, e.g., a Fortran+MPI application. Under the pressure of AI and data-driven applications, HPC systems will need to integrate with Cloud services and productivity-oriented software packages typical of these application domains. The challenge is to design an integration that supports productivity and portability without sacrificing HPC systems' performance.

This work improves the state of the art in two ways. First, we advocate a general methodology to transform the sequential execution model of a standard computational notebook into a global (parallel and distributed) one, general enough to express data and

[*] Corresponding author.
E-mail address: iacopo.colonnelli@unito.it (I. Colonnelli).

[1] *HPC Key Technologies and Tools* (HPC-KTT) national laboratory, CINI, Italy

https://doi.org/10.1016/j.future.2021.10.007

task parallelism. The global execution model is formally described in such a way to prove sequential equivalence, and therefore not jeopardize determinism and reproducibility. Concurrency is introduced in the Notebook by way of annotations in the cell's metadata.

Second, we introduce *Jupyter-workflow* (Jw), a novel Workflow Management System targeting Cloud-HPC infrastructures, with extends the Jupyter software stack to implement the proposed methodology. We experiment Jw for both expressivity and performance on four real-world pipelines from different scientific domains and different execution platforms (HPC and Cloud).

Notice that Jw is neither about running Jupyter on a remote platform nor using Jupyter as an interface to access a remote machine to run a third-party workflow. It is a workflow programming model for Notebooks, which are not as expressive as language-dependent workflows (e.g., iteration among cells and dynamic generation of new cells is not supported), but are the most used tools for interactive data analysis, which is destined to meet HPC.

This article is organized as follows. Section 2 is devoted to analyzing the related work, while Section 3 introduces the proposed methodology. After that, Section 4 describes the logical architecture and the software stack's implementation details, which is then evaluated on four real scientific workloads in Section 5. Finally, Section 6 concludes the article and outlines future research directions.

2. Related work

2.1. Workflow management systems

The workflow abstraction is widely used to model and implement complex applications, both in scientific and industrial domains, with a twofold advantage. Firstly, an explicit representation of true data dependencies among different steps, usually as the directed arcs of an acyclic graph, allows for an efficient distributed execution that reduces time-to-solution. Additionally, the strong decoupling between the application layer and the underlying computing architecture improves portability and reproducibility.

In this context, a Workflow Management System (WMS) is the high-level interface between the domain specialist and a set of execution infrastructures, ranging from a single desktop machine to an entire data center. The WMS landscape is very variegated, and the workflow modeling paradigms exposed to the users differ from one product to another, either focusing on generality and ease of use or privileging flexibility at the cost of increased complexity.

Many WMSs on the market, like Apache Taverna [1], Pegasus [2], Snakemake [3], Makeflow [4], and Nextflow [5], implement a strict separation of concerns between coordination and application layers. In this setting, the workflow graph is described with a dedicated, domain-specific *coordination language*. Each step can be specified in a *host language* of choice, usually a general-purpose programming language such as C++, Python, or R [6].

An intrinsic separation of concerns undoubtedly enhances the level of abstraction, with essential benefits in maintainability. Nevertheless, it introduces an additional (usually product-specific) coordination formalism that users must learn. This is why some products such as Galaxy [7], Kepler [8], and Knime [9], prefer to hide or replace the coordination language with a higher-level Graphical User Interface (GUI), trading off flexibility in favor of simplicity. Alternatively, other WMSs, e.g., CWL-Airflow [10], Toil [11], and StreamFlow [12], introduce support for product-agnostic coordination languages, such as CWL [13] or YAWL [14], to enhance portability and reproducibility and reduce the learning effort. Nevertheless, the actual effectiveness of a coordination standard is strictly related to its market adoption.

A strict separation of concerns is beneficial when workflows involve calls to third-party libraries or well-separated components of a software suite developed and maintained by third-party contributors. Conversely, it is not particularly suited for situations requiring co-design of different portions of a program or when the best granularity of steps is not clear in advance, e.g., when developing an entire application from scratch. Low-level distributed libraries, like PyCOMPSs [15], Ray [16], Dask [16], and Parsl [17], provide additional flexibility by modeling tasks and their dependencies directly in the host code. These systems allow users to parallelize existing sequential applications by identifying and annotating functions that should be executed as asynchronous parallel tasks. Asynchronicity is typically implemented with the *futures* paradigm [18]: an annotated function immediately returns a future object that can be passed as an argument to another annotated function. The workflow model, typically a layered dataflow model [19], is automatically built just-in-time by the runtime layer of the framework. Each annotated function invocation is a step of the workflow, and if it receives in input a future from another invocation, then there is a dependency between the two. This approach allows programmers to express concurrency at a much finer grain, well-fitting applications with low latency or high throughput requirements, but its higher complexity makes it better suited to computer scientists rather than domain experts. Moreover, libraries of this kind usually assume the existence of a shared data space and do not support file transfers between compute units (PyCOMPSs is an exception).

Several tools adopt a similar idea for the automatic collection of provenance data from scripts. For instance, yesWorkflow [20] allows users to insert special, language-independent comments in a script to explicitly describe the data flow. Its interpreter can then rely on such comments to generate a dataflow representation of the script. Similarly, RDataTracker [21] allows users to initialize and store automatic provenance collection for a portion of an R script by explicitly calling its APIs, possibly relying on more advanced functions to manipulate the output. The W2Share approach [22] takes a step further, trying to (semi-)automatically derive Taverna executable workflows from abstract representations extracted by yesWorkflow. However, the amount of human intervention required in each phase is still significant. The main drawback of these approaches is that provenance extraction logics are interleaved with business code, reducing readability and maintainability.

Other tools, such as noWorkflow [23], adopt a fully automatic strategy by extracting the data flow from a static analysis of the Python code's Abstract Syntax Tree (AST). Analogously, Baranowski et al. [24] explore the AST of Ruby scripts targeting GridSpace execution environment [25] to extract workflow models. The CXXR project [26] extends the R interpreter to automatically retrieve provenance data, while the LLVM-SPADE stack [27] augments binaries with provenance tracking logics at compile time. These approaches guarantee great flexibility in extracting the data flow and the environment state at any given time. However, it is difficult to properly set the granularity of retrieved information while operating at such a low and application-agnostic level. In addition, these solutions only target a single language and quite often only a specific version of it.

Due to the variety of proposals, no standard metrics exist to compare the different WMSs, and no silver-bullet solution has emerged so far. In [28], a holistic evaluation is suggested in terms of setup and deployment, workflow implementation and execution, and data management. As in other comparisons, e.g. [29], only a restricted subset of systems is considered.

As traditional WMSs, our approach decouples coordination logic (metadata) from application code (cells). No annotations or directives are required in the host code, ensuring portability and avoiding technology lock-in. In addition, metadata are also used to provide a description of the target infrastructure(s), which is then stored together with host and coordination codes for the benefit of reproducibility.

2.2. Workflows with Jupyter

The terms *Jupyter Notebook* [30] may refer to two different components: (1) a JSON document, following a versioned schema, containing an ordered list of cells with code, explanatory text, mathematics, plots, and rich media (2) a web-based interactive computational environment for creating such documents. In this work, we shall default to the former meaning.

Jupyter Notebooks, with their capability to unify code and metadata in a single document format, represent a valuable middle ground between the workflow modeling approaches introduced in Section 2.1. Furthermore, since Jupyter Notebooks have become a de-facto standard in several research fields, many domain experts are already familiar with them. Given that, some efforts to use Jupyter Notebooks for workflow modeling already exist in the literature.

Cloud-based scientific platforms like KBase [31,32] and GenePattern 2.0 [33] have built their primary user interface on Jupyter. However, their main goal is offering an effective bioinformatics programming environment, providing an extensive database of pipelines, introducing user-friendly interfaces, and offloading computation to their own specific target infrastructure, e.g., the HTCondor cluster. The resulting notebooks are tightly coupled with the underlying software stack, preventing portability to the broader Jupyter ecosystem.

The Netflix's nteract[2] framework adopts a self-contained approach, allowing users to augment Notebooks with input data and schedule them for batched execution. The outputs are also saved in Notebook format, facilitating inspection and debugging. This approach's primary limitation is its coarse granularity, as it only allows executing entire Notebooks from beginning to end.

The Script-of-Scripts (SoS) project [34] moves the unit of execution to the finer-grained level of single cells and introduces multi-language Notebooks, called SoS Notebooks. Each cell of a SoS Notebook can declare a different host language, with the SoS runtime automatically handling object conversions during inter-cells communications.

Both SoS and our approach allow offloading entire Notebooks or single cells to a queue-based HPC center, automatically handling input and output variables, path translations, and data-parallel patterns. However, while the latter adopts a purely metadata-based coordination language, SoS comes with a template-based mechanism that mixes up coordination and application logic inside code cells. This strategy introduces additional complexity, reducing maintainability and tightly coupling the host code with the SoS interpreter.

The Notebooks-into-Workflows (NiW) project [35] adopts a different approach. Rather than directly using a Jupyter notebook as a workflow description, the notebook is translated into a WINGS workflow [36], which can be independently executed, published, and reproduced. Even if the extraction of the workflow structure is fully automatic, there are strict compatibility requirements for notebooks: any newly generated data must be written into files, and all the code using the same file must be placed in a single cell.

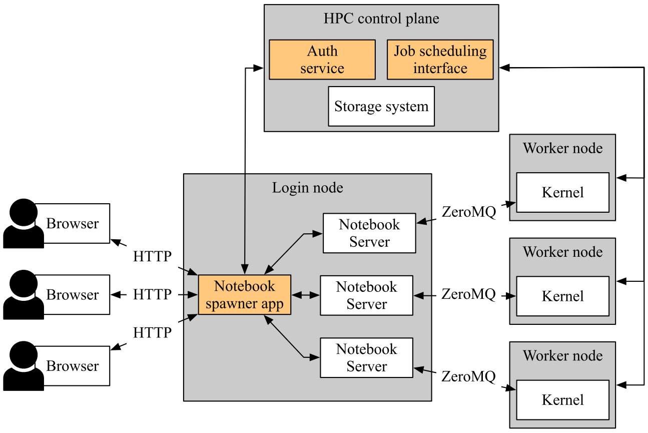


Fig. 1. Sketch of the common architecture of a HPC Jupyter interface. Orange blocks usually need to be developed or extended to integrate Jupyter in the existing software stack.

Several attempts to extract the data stream through cell executions have appeared in the literature. For instance, Dataflow Notebooks [37] modify the behavior of the IPython `Out` dictionary, indexing each cell execution with a unique persistent identifier and allowing users to refresh all the dependent cells after a cell re-execution. Nodebook[3] automatically tracks dependencies across IPython cells through static AST analysis and caches their outputs, enforcing a consistency based on the position of the cells in the notebook. The NBSAFETY Jupyter kernel [38] combines dataflow tracing with liveness and initialized variable analysis to detect staleness generated by cell re-executions in Python Notebooks. The Vizier framework [39] implements an entirely new computational notebook stack, in which each cell is executed on a separated program context, and communications between cells are explicitly handled by producing and consuming datasets, i.e., sets of named relational tables. All these approaches mainly aim to prevent the issues caused by out-of-order executions and repeated executions of Notebook cells, which are considered among the principal causes of reproducibility issues in the Jupyter ecosystem [40].

2.3. Jupyter notebooks on HPC

Finding an effective way to improve accessibility to HPC facilities is still an open problem in computer science. Indeed, if the advent of Cloud-based *-as-a-Service approaches contributed to a substantial lowering of the technical barriers to IT systems, the vast majority of data centers are still anchored to queue management systems, SSH-based remote shells, and air-gapped worker nodes.

Given their widespread diffusion and their relatively high-level interface, Jupyter Notebooks have already been investigated as a way to bridge the gap between non-IT practitioners and HPC infrastructures [41–44] to make interactive workflows mainstream in HPC centers. As an example, the National Energy Research Scientific Computing Center (NERSC) is currently integrating Jupyter as an interface to the CORI supercomputer [45], and the PANGEO platform component for HPC is based on the integration between Jupyter and Dask [46].

One of the most common approaches in this direction is to install a Jupyter-based service, e.g., a JupyterHub[4] instance or a custom Jupyter Notebook spawner, on the login nodes of a data

[2] https://nteract.io/

[3] https://multithreaded.stitchfix.com/blog/2017/07/26/nodebook/

[4] https://jupyterhub.readthedocs.io/en/stable/

center or on some publicly-exposed instances of a tightly-coupled Cloud service. Appropriate authentication chains can then percolate the user identity from the web interface down to the HPC worker nodes, ensuring secure access to the computing resources running the Jupyter kernels. Fig. 1 sketches this kind of architecture, highlighting in orange the blocks that usually need customizations.

If these approaches can ease access to HPC facilities for practitioners, their setup and maintenance are non-trivial tasks even for expert system administrators. One of the biggest obstacles is undoubtedly the need to modify the authentication mechanism, with all its security implications. An even more complicated scenario occurs when trying to offload Jupyter kernels from outside the data center because the ZeroMQ[5] message broker implementing Jupyter's communication layer needs bidirectional TCP connections between Notebooks and kernels. Such a requirement does not combine well with the air-gapped HPC facilities.

Aiming at freeing the end-users from system administrators' choices, and at the same time trying to save the latter from modifying the security layer, the proposed approach relies on the StreamFlow[6] WMS [12] to implement its communication layer. StreamFlow only requires unidirectional connections to the execution environment because the controller infrastructure leads both data and control planes. Therefore, users can connect to the remote execution environment using ordinary methods, with no additional installation required on the system administrators' side. Moreover, since it does not need a single data space shared among all the worker nodes, it can also deal with hybrid Cloud-HPC scenarios.

As a matter of fact, scientific workflows and HPC communities are converging on the same objectives to provide effective workflow management in a combined HPC and distributed environment [47], and Jupyter is an ideal tool for both communities. Our approach is designed to address the main requirements across these domains, providing a modeling methodology based on metadata enrichment, leveraging on Jupyter usability, and enabling execution on hybrid HPC and Cloud environments without any particular requirement on the remote execution infrastructures. Moreover, the strict separation of coordination and host code and the explicit definition of a high-level metadata format to describe application dependencies prescind from any specific implementation, allowing different runtime libraries to co-design the best-suited execution strategy with the peculiarities of their reference execution environment.

3. Methods

We envision Jupyter Notebooks as the first representative of a new class of interfaces to foster both portability and performance, which traditionally map onto different classes of tools (as discussed in Section 2.1). This goal is achieved by mapping the Notebook cells into a workflow graph's nodes, enabling a parallel and distributed execution over a broad set of infrastructures.

In this section we provide a formal account of the execution model of Jupyter Notebooks. The formalization allows us to describe in precise terms – and state the equivalence of – their (local and distributed) sequential and parallel semantics.

3.1. Notebooks as workflows

A Jupyter Notebook document can be seen as an ordered list of cells, each of which contains either code or Markdown text. For simplicity, we will only consider the former, as Markdown cells do not affect the Notebook's operational semantics. A Notebook document's low-level format is defined with a JSON schema,[7] used for validation by Jupyter tools. In such format, both the whole Notebook and every single cell are accompanied by a metadata field, containing arbitrary optional JSONable information about the related element (see the example in Listing 1).

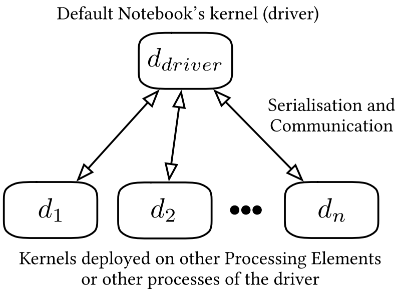

Default Notebook's kernel (driver)
d_{driver}
Serialisation and Communication
d_1
d_2
d_n
Kernels deployed on other Processing Elements or other processes of the driver

Fig. 2. Runtime architecture of a Jupyter Notebook's distributed execution.

In this work, we consider a single code cell as the atomic execution unit of a Notebook. Its content is mapped into a workflow step, while its metadata field describes *where* and *how* such step should be executed. Let a Notebook $\boldsymbol{N}$ be composed of a sequence of cells $c_1, \dots, c_n$, where every c_i is a sequence of commands in the Notebook host language (e.g., Python). A cell is executed in a *state*, that is a partial map $\sigma : \mathrm{Ide} \to \mathrm{Obj}$ from host language identifiers to host language first-class objects (e.g., values, expressions, functions). For any cell c_i, its metadata includes two (possibly empty) sets of identifiers $In(c_i)$ and $Out(c_i)$, respectively representing its *inputs*, namely the identifiers whose value is necessary for its execution, and its *outputs*, namely the identifiers whose value is affected by its execution.

If we represent a cell waiting to be evaluated as a *configuration* $\langle c, \sigma \rangle$ such that $In(c) \subseteq \mathrm{dom}(\sigma)$, the *execution relation*

$$\langle c, \sigma \rangle \to \sigma'$$

expresses the fact that the execution of c in the state σ produces a new state σ' such that $Out(c) \subseteq \mathrm{dom}(\sigma')$.

We are not interested in formalizing the execution relation of every single cell of a Jupyter Notebook, which depends on the host language. We just point out that the execution is partial (it might diverge).

Hereafter we specify the sequential, distributed and parallel semantics of Jupyter Notebooks, considering each cell as an atomic command whose semantics is left unspecified.

The local execution of a sequence $c_1; \dots; c_n$ of cells can be formalized as follows

$$\langle c_1; \dots; c_n, \sigma_1 \rangle \to \sigma_{n+1} \quad (1)$$
$$\text{if } \langle c_i, \sigma_i \rangle \to \sigma_{i+1} \text{ for every } 1 \leq i \leq n$$

so that each cell in the sequence is executed in the state produced by the previous one. The execution order specified by the operator ';' is driven by the user and might differ from the textual order of the cells in the Notebook.

3.2. Distributed sequential execution

In order to model the distributed execution of a Jupyter Notebook, we extend configurations $\langle c, \sigma \rangle$ to *global configurations* $\langle [c, d], \sigma \rangle$, where the d component indicates the location in which

[5] https://zeromq.org/
[6] https://streamflow.di.unito.it/
[7] https://nbformat.readthedocs.io/en/latest/format_description.html

Listing 1: Jupyter Notebook document cell format (general and code)

```
# Generic cell metadata
{
  "cell_type": "type",
  "metadata": {},
  "source": "single string or [list, of, strings]",
}

# Code cell metadata
{
  "cell_type": "code",
  "execution_count": 1,# Integer or null
  "metadata": {
    "collapsed": True, # Whether the output of the cell is
          collapsed
    "scrolled": False, # Any of true, false or "auto"
  },
  "source": "[some multi-line code]",
  "outputs": [{
    # List of output dicts (described below)
    "output_type": "stream",
    ...
  }],
}
```

the execution of c takes place. The *global execution relation* can then be specified as

$$\langle [c_1, d_1]; \ldots ; [c_n, d_n], \sigma_1 \rangle \rightarrow \sigma_{n+1} \quad (2)$$

$$\text{if } \langle [c_i, d_i], \sigma_i \rangle \rightarrow \sigma_{i+1} \text{ for every } 1 \leq i \leq n$$

Eq. (2) has the same meaning as the Eq. (1), except that it carries the additional information that a cell c_i is deployed and executed on $d_i \in \boldsymbol{D}$, where $\boldsymbol{D}$ is a set of predefined deployments. In particular, Eq. (1) is equivalent to Eq. (2) when $d_i = d_{driver}$ for every $1 \leq i \leq n$, where d_{driver} refers to the default Notebook's kernel (see Fig. 2).

The location in which a given cell is meant to be executed can be encoded in the metadata field of cell or it can be determined through more complex policies as we will see in Section 3.3. We envision a deployment onto processing elements, either local or remote, capable of executing pre-installed applications or containers (e.g., a Kubernetes running Dockers or a Slurm manager scheduling Singularity images) and reachable from the Notebook executor.

Just like the local execution relation, also the global execution relation is partial. However, we assume that the remote execution of a cell is independent of the location in which the execution takes place. That is, we assume that $\langle [c, d], \sigma \rangle \rightarrow \sigma'$ and $\langle [c, d'], \sigma \rangle \rightarrow \sigma''$ implies $\sigma' = \sigma''$. So, the execution of a cell that succeeds locally might fail remotely, but it must produce the same output in each deployment whenever the execution is successful. This requirement guarantees deterministic semantics of global execution, modulo errors.

The global execution relation unleashes a cell's execution from the local Notebook kernel, explicitly addressing a targeted worker in a set of remote resources by making the configuration itself explicitly aware of the place where it should happen. Global execution in a global address space has been longly experimented (with little success) in the last two decades within the *Partitioned Global Address Space* (PGAS) programming model, which has been implemented both as new languages (e.g., X10 [48], Chapel [49]) and libraries (e.g., UPC++ [50], DASH [51]). Many of them rely on Remote Memory Access protocols, allowing some form of sharing (e.g., GASNet [52]). Recently, dryish versions of global sharing under a simple Single-Writer-Multiple-Reader model with locality awareness have been exploited in the GAM [53], and Ray [16] runtime supports.

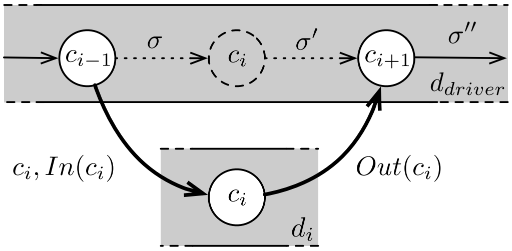


Fig. 3. Interactive remote execution of a configuration $\langle [c_i, d_s], \sigma \rangle$. When $d_s \neq d_{driver}$, both code c_i and true data dependencies $In(c_i)$ must be transferred to d_s. After that, the return values $Out(c_i)$ can be transferred back to d_{driver}.

3.3. Parallel execution

A Notebook can be executed interactively or in bulk mode (*Run All* command). In both cases, the execution of cells is sequential, and each cell is executed in the state resulting from the execution of the previous cell in temporal order, as described in Eq. (1). The same execution sequence can be extended to a distributed sequential execution as described in Eq. (2), where cells can be executed in different places following the same sequential order.

In the interactive mode, the execution of a cell is triggered by the user, which directly operates on the web interface and the host language interpreter running the cells one by one on $d_{driver} \in \boldsymbol{D}$ (see Fig. 3). In this case, consistency is trivially guaranteed.

The bulk execution mode enables a concurrent distributed execution of the cells by defining a *sequentially equivalent* parallel semantics. Since the cells $c_1, \ldots, c_n$ are totally ordered by their position in the Notebook, the control-flow graph is the linear chain of cells. Given that, we can statically compute the execution graph with the highest degree of parallelism by way of Bernstein's conditions [54], initially designed for parallelizing compilers. They describe the three cases that induce data dependencies among commands (cells in our case).

The presence of a data dependency between two cells prevents their parallel execution. When $i < j$, we say that *c_j depends on c_i* if at least one of the following conditions holds:

- $Out(c_i) \cap In(c_j) \neq \emptyset$. In this case we have a *true data dependency*, whereby c_i modifies an identifier read by c_j;
- $Out(c_i) \cap Out(c_j) \neq \emptyset$. In this case we have an *output dependency*, whereby c_i and c_j modify the same identifier;
- $In(c_i) \cap Out(c_j) \neq \emptyset$. This is a so-called *anti-dependency*, whereby c_j modifies an identifier read by c_i.

In all other cases, namely when

$$Out(c_i) \cap In(c_j) = Out(c_i) \cap Out(c_j) = In(c_i) \cap Out(c_j) = \emptyset$$

the cells c_i and c_j do not interfere with each other and can be executed in any order, hence also in parallel. As we shall see, we relax Bernstein's conditions by proposing a strategy to reconcile clashes due to output dependencies. Notice the importance of avoiding output dependencies in designing concurrent semantics for Notebooks: the execution of all cells produce an output on the initial state of the Notebook, where surely the identifier representing the standard outputs conflict. In reality, the interactive execution of the Notebook makes it possible to preserve the output of all cells. The proposed relaxation aims to preserve the output of all cells, which in the sequential execution happens in the same state (the Notebook's output) in different moments.

In preparation for their parallel execution, all cells are arranged in a direct acyclic graph (DAG) from the control-flow chain, thereby defining a workflow in which nodes are configurations and edges are data dependencies. Let us call *DAG* *evaluation* the order derived applying anti-dependency and true data dependencies. We can describe a DAG evaluation sequence as a term built using atomic cells c composed using ';' (sequential composition) and '|' (parallel composition). For example,

$$c_1; (c_2 | (c_3; c_4))$$

describes the execution of c_1 followed by the parallel execution of c_2 and the sequential execution of c_3 and c_4.

In this setting, the DAG evaluation order is defined by Eq. (2). Jupyter-workflow supports the two main parallelism exploitation paradigms: (1) explicit *data parallelism* on a single cell starting from lists of data elements in both interactive and bulk execution modes; (2) automatic *task parallelism* on independent cells in the bulk execution mode. We now describe the semantics of these two forms of parallelism more in detail.

Data parallelism. Jw implements data parallelism – that is the *Map/ApplyToAll* functions – by way of an *explicit* metadata annotation on a cell c receiving as input one or more lists $L_1, \dots, L_k$ and a list operator such as the *dot product* or the *cartesian product*. Hereafter we denote such cells with $Map(c)$.

The semantics of a *Map* cell depends on two (here unspecified) functions: *Scatter*, which splits states to lists of states, and *Gather*, which recombines lists of states into states. We can then describe the execution semantics of *Map* cells as follows

$$\begin{aligned}&\langle [Map(c), d], \sigma\rangle \rightarrow Gather(\sigma'_1, \dots, \sigma'_n)\\ &\text{if } \langle [c_1, d_i], \sigma_i\rangle \rightarrow \sigma'_i \text{ for every } 1 \le i \le n\\ &\text{where } Scatter(\sigma) = [\sigma_1, \dots, \sigma_n]\end{aligned} \tag{3}$$

where the d_i's are selected from the set of unallocated locations that satisfy user-specified requirements.

In case two cells $Map(c_1)$ and $Map(c_2)$ are executed sequentially in bulk mode and $Out(c_1) = In(c_2)$, we leverage on the *map fusion* transformation [55] to reduce the communication overhead, rewriting a *Scatter-Gather-Scatter-Gather* sequence as *Scatter-LocalCopy-Gather*. We can express this optimization as the following equivalence between configurations:

$$\langle Map(c_1); Map(c_2), \sigma\rangle = \langle Map(c_1; c_2), \sigma\rangle \tag{4}$$

Lemma. *The Map function preserves sequential equivalence of successfully terminating global parallel executions.*

Proof. By construction, *Map* generates a list of independent replicas of a cell that satisfy Bernstein's conditions. □

Task parallelism. Jw also supports the automatic parallelization of independent cells in the bulk execution mode. We define independent cells according to a relaxation of Bernstein's conditions that allows output conflicts (thus removing the output dependency case), assuming the existence of a user-defined associative operator $\uplus$ that reconciles all conflicting identifiers while avoiding output dependencies. Formally, given $c_1, \dots, c_n$ independent cells, their parallel execution is described as

$$\begin{aligned}&\langle [c_1, d_1] | \cdots | [c_n, d_n], \sigma\rangle \rightarrow \biguplus_{1\le i\le n} \sigma'_i\\ &\text{if } \langle [c_i, d_i], \sigma_i\rangle \rightarrow \sigma'_i \text{ for every } 1 \le i \le n\end{aligned} \tag{5}$$

where each σ_i is the restriction of σ to $In(c_i)$ and the reconciled state $\biguplus_{1\le i\le n} \sigma'_i$ is the union of the states for non-conflicting identifiers or the reduction of objects for conflicting identifiers. That is,

$$(\sigma \uplus \sigma')(x) = \begin{cases}\sigma(x) & \text{if } x \in \text{dom}(\sigma) \setminus \text{dom}(\sigma')\\ \sigma'(x) & \text{if } x \in \text{dom}(\sigma') \setminus \text{dom}(\sigma)\\ \sigma(x) \uplus \sigma'(x) & \text{if } x \in \text{dom}(\sigma) \cap \text{dom}(\sigma')\end{cases} \tag{6}$$

where the $\uplus$ operator on objects is provided by the user.

Even though the $\uplus$ operator resembles a *Reduce* operator, its pragmatics is not the parallelization accumulation behavior. In the sequential evaluation, accumulation generally induces a true data dependency, which cannot be easily removed without changing the cell business code. The $\uplus$ operator aims at carrying the merging of the cell's outputs in a single, adequately typed identifier. This behavior is inspired by merging stdout of remotely executing processes aiming to preserve single-cell output rather than overwriting them. The Jupyter Notebook preserves a private copy of the sequentially evaluated cells' output.

Theorem. *Given an associative operator $\uplus$ that correctly reconciles conflicting identifiers of states $\sigma_1, \dots, \sigma_n$, the DAG evaluation preserves sequential equivalence of successfully terminating global parallel executions.*

Proof. Two cells with either anti-dependency or true data dependency cannot be executed in parallel. The parallel execution of two cells generates a *reconciled state* $\sigma_1 \uplus \sigma_2$ that includes all the non conflicting identifiers of σ_1 and σ_2 with the same value of the sequential evaluation (see Eq. (6)) and all the conflicting identifiers of σ_1 and σ_2 computed by the user-defined associative operator $\uplus$. The existence and the correctness of the (user-defined) $\uplus$ operator is an assumption, which is satisfied by common operators such as list and string concatenation and value reduction. The merged state subsumes that the result of the execution of the two cells is executed in any sequential order. Being $\uplus$ an associative operator, the same argument scales to the transitive closure of the merged state of a sequence of cells executed in parallel, which includes all their output identifiers. All the identifiers available in the last state of the sequential execution are also available in the merged state of the parallel evaluation, with identical or equivalent values for conflicting identifiers. For this, we consider the parallel execution sequentially equivalent. □

4. Design and implementation

In compliance with the methodology discussed in Section 3, we realized an extension of the Jupyter software stack, named *Jupyter-workflow* (Jw)[8]. Its logical architecture, depicted in Fig. 4, consists of five main components:

- A *coordination metadata format* to express cell configurations;
- A *frontend extension* to send such metadata to the backend kernel;
- An *extended kernel* capable of managing remote cell executions;
- An *executor script* to remotely execute a cell and serialize its return values;
- A *dependency resolver* component to help users identifying the input dependencies of each cell.

Moreover, we rely on the dill library [56] to perform data serialization and deserialization (SerDes) and on the StreamFlow WMS to coordinate workflows. The rest of the current section is devoted to a detailed analysis of each component, discussing the most significant design and implementation choices.

4.1. Coordination metadata format

We added a `workflow` section to the Jupyter code cell metadata format to configure all the aspects introduced in Section 3,

[8] https://jupyter-workflow.di.unito.it

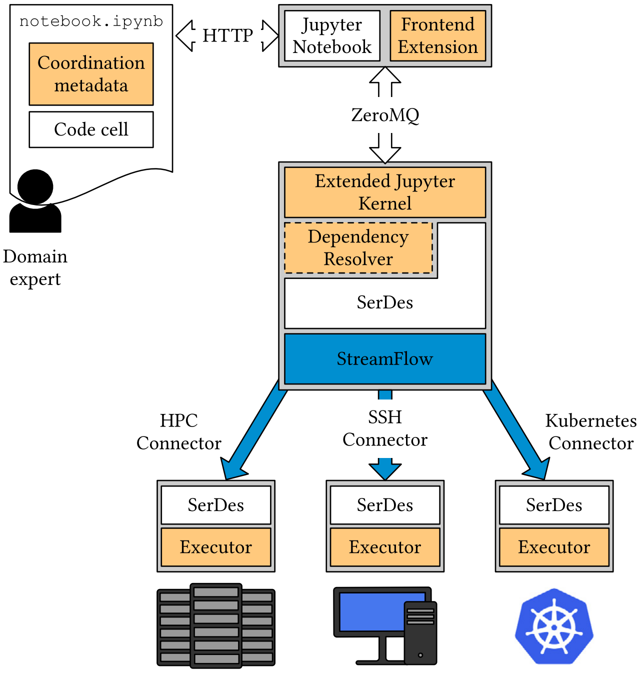


Fig. 4. Jupyter-workflow logical stack. White blocks refer to existing technologies (except for StreamFlow and its connectors, which are colored in blue), while yellow-ocher ones are directly part of the Jupyter-workflow codebase.

Listing 2: Jupyter-workflow metadata format

```
# Workflow metadata
{
  "step": {
    "in": [{  # List the members of In(c_i)
      "type": "name" | "env" | "file" | "control",
      "name": "variable name",
      "serializer": {
        "predump": "code executed before serializing",
        "postload": "code executed after serializing"
      },
      "value": "value to assign to the name",
      "valueFrom": "can take value from a different variable"
    }],
    "autoin": True | False, # Resolve In(c_i) automatically
    "out": [  # List the members of Out(c_i)
      ...
    ],
    "scatter": {
      "items": ["variable name" | "scatter subscheme"],
      "method": "dotproduct" | "cartesian" | ...
    }
  },
  "target": {# Part of the StreamFlow format
    "model": {# Description of the execution environment
      ...
    },
    "service": "target service inside the model",
    "resources": "number of workers to reserve"
  },
  "version": "v1.0"
}
```

i.e.: (1) The set $In(c_i)$ of input dependencies, potentially augmented with data-parallel descriptors; (2) The set $Out(c_i)$ of cell return values, potentially augmented with custom reduction descriptors; (3) A deployment d_s where to schedule the cell execution. A high-level schema, with some details omitted due to limited space, is reported in Listing 2.

A `step` subsection contains two lists, called `in` and `out`, describing input dependencies and return values, respectively. Currently, Jw supports four different families of dependencies in the `type` field:

- *names*, which are transferred to the destination program's state;
- *environment variables*, which are added to the target shell's environment;
- *files*, for which Jw automatically manages both data transfers and path remappings on the target executor;
- *controls*, which are only used to force additional dependency relations between workflow steps, without carrying any data value.

The deployment that will host the cell execution is described in a `target` subsection, which is an exact transposition of StreamFlow's `target` directive. StreamFlow relies on three different levels of granularity to represent remote execution environments:

- A `model` is an entire multi-agent infrastructure, e.g., a Helm release on Kubernetes or a Slurm-managed data center;
- A `service` is a specific agent in a model, e.g., a head node in a Ray cluster or a Kafka server in a microservices architecture;
- A `resource` is a single instance of a replicated service, e.g., an MPI node in a cluster or a single Pod managed by a ReplicaSet on Kubernetes.

A model is usually described in an external format, e.g., a Docker Compose file, a Helm chart, or a Slurm sbatch script, and constitutes the unit of deployment in StreamFlow. Conversely, the unit of scheduling is usually a single resource. However, a step can explicitly reserve multiple resources of the same `service` for execution, specifying a value greater than one in the `resources` field. Moreover, multiple instances of the same step can spawn concurrently on multiple resources when using data-parallel directives.

The Jw metadata format allows iterable cell inputs, e.g., a Python list or dictionary, to be scattered across multiple replicas of d_s for parallel execution. Scattering schemes can be specified through a dedicated `scatter` section in the `step` metadata (Listing 2). In particular, an `items` list contains the elements to scatter, while the `method` entry specifies which operator (e.g. `dotptoduct` or `cartesian`) should be used when scattering over multiple `items`. The `items` list can container either variable names, file paths or, nested scatter schemes, giving great flexibility to the developer. For example, a configuration like the following is perfectly fine whenever ``c'' and ``d'' are of the same length.

```
"scatter": {
    "items": ["a", "b", {"items": ["c", "d"],
                         "method": "dotproduct"}],
    "method": "cartesian"
}
```

4.2. Jupyter stack extension

Custom metadata are generally not propagated to the backend kernel by the Jupyter web interface. Therefore, an extension for the frontend stack is required to include the `workflow` metadata section when sending messages to the kernel. The technology used by such component depends on the adopted frontend. For the classical Jupyter interface, a `kernel.js` file in a kernel package allows kernel-specific frontend extensions. Conversely, the newer JupyterLab technology needs a kernel-agnostic frontend plugin, and it is currently under development.

After implementing a sending mechanism for coordination metadata, we need a kernel backend capable of correctly processing them during both interactive and bulk execution flows.

We started extending the IPython kernel, both because it is by far the most widely used backend in the Jupyter ecosystem and because, since StreamFlow is also implemented in Python, the integration with the underlying WMS layer was much more manageable. Nevertheless, support for other Jupyter kernels is undoubtedly in our plans. Indeed, the strict separation between host code and coordination metadata makes it possible to reuse the same metadata format independently of the host language, while the message-oriented nature of the Jupyter stack significantly simplifies language interoperability.

When executing a Notebook in bulk mode, the first step is to obtain a dataflow representation of its code cells, which takes the form of a Directed Acyclic Graph (DAG). Since the original execution flow is sequential, cells must be processed in order, extracting `workflow` metadata and input dependencies for each of them. The result of this operation is an ordered list of cells $[c_1, \dots, c_n]$ with the related $In(c_i)$ and $Out(c_i)$ sets. Such cells constitute the nodes of a dataflow graph, where a directed arc connects c_i to c_j (with $i < j$) whenever $\exists\, v \in In(c_j)$ s.t. $v \in Out(c_i)$ and $v \notin Out(c_k)$ for each k in the open interval (i, j). The resulting DAG can then be orchestrated by the StreamFlow runtime support.

The interactive execution flow is much more straightforward. Cells are sequentially processed one by one so that there is no need to construct dataflow-based intermediate representations, and the consistency of the program state is trivially preserved. However, since in an interactive scenario the cell execution order cannot be determined a priori, all the components of $Out(c_i)$ are always transferred to the local kernel and merged into the program state before proceeding with computation.

4.3. Executing cells remotely

When a cell enters the fireable state, all its input dependencies are transferred from the related predecessor in a serialized form so that the program state can be reconstructed from them. StreamFlow manages all the computation movement aspects, i.e., data transfer, path remapping, resource deployment, and task scheduling. In particular, an `executor` script, automatically transferred to each remote resource, is in charge of recreating the program state, executing the code, and serializing the return values.

Unlike most alternatives on the market, StreamFlow drops the requirement for a single data space shared among the entire set of workers. This feature allows it to support distributed workflow executions over hybrid architectures, e.g., with some steps scheduled on an HPC facility and the others offloaded to a Cloud computing infrastructure. Besides, StreamFlow can manage the automatic deployment and deletion of complex execution environments (e.g., an entire Spark cluster over Kubernetes), including their description directly in its workflow specification format. Moreover, if a scatter step input is a list of files, each of its elements will be transferred in parallel to the assigned remote resource by the underlying StreamFlow runtime.

StreamFlow also supports a pluggable checkpointing mechanism to store such dependencies on a persistent location for fault tolerance purposes. Indeed, if a remote resource fails to perform a step, it can be rescheduled for execution only if all its input dependencies are still available. Otherwise, it is necessary to travel back in the workflow DAG until a reschedulable step is met and replay the entire chain of steps up to the failed one before retrying it, introducing significant overhead.

The main obstacle for Jupyter Notebooks fault-tolerance is that a Python kernel cannot be fully serialized, stored on disk, and deserialized (see Section 4.5). This constraint prevents live migrations and high-availability of kernels. Storing all outputs of remote cells on a persistent location eases the re-execution of notebooks. When the node hosting Jw restarts after a crash during a bulk Notebook execution, only local cells must run to populate the local program context. At the same time, StreamFlow can automatically load output dependencies of completed remote cells from their checkpoint location. In principle, similar functionality could be extended to interactive notebooks by keeping the history of cell executions, but Jw does not yet provide such a feature.

4.4. The DependencyResolver component

In many cases, input dependencies can be automatically inferred with an inspection of the cell code. Therefore, we developed a `DependencyResolver` component to save practitioners the burden of manually listing every input name for every cell execution.

In Python, the `ast` module allows exploring the Abstract Syntax Tree (AST) of a code fragment. Therefore, since the Python language is lexically scoped, it is possible to obtain the set of input dependencies of a cell by seeking all the names: (1) that reside in its global scope; (2) whose first operation is a `Load`, i.e., a read from σ; (3) do not come from Python builtins or IPython standard namespace.

The fact that the serialization library can autonomously deal with transitive dependencies dramatically simplifies this task, as it is unnecessary to explore names' definitions external to c_i itself. Nevertheless, it is worth noting that the proposed strategy cannot entirely cover all possible scenarios, as both false positives and false negatives can occur.

In general, each name that does not appear in a node of the AST representation returned by the `ast` module cannot be recognized by the DependencyResolver. This set contains, for example, variables dynamically loaded in `eval` constructs, variables accessed directly from the `locals` and `globals` dictionaries and modules dynamically imported by the `importlib` package.

Conversely, since the code can be statically evaluated without knowing a priori the exact value of each name, some names can be marked as true dependencies even when they are never accessed. This case includes variables loaded in untaken paths of conditional branches, `except` branches of exception handling patterns or locations of a container (e.g. a list or a dictionary) that are never accessed.

Given that, we let users combine automatic dependency induction with explicit metadata to correct potentially wrong behaviors, print the list of automatically identified dependencies, or even to completely disable the `DependencyResolver` for a step by setting the `autoin` field to `False`.

4.5. Serialization

When dealing with computation movement, serialization is undoubtedly one of the most critical aspects to take into account. Indeed, considering a subprogram c_i with a set $In(c_i)$ of input dependencies and a set $Out(c_i)$ of return values, the presence of even a single unserializable element in $In(c_i) \cup Out(c_i)$ is sufficient to prevent c_i from being executed remotely.

On the other hand, pretending to reason about a perfect serializer capable of producing a suitable byte stream for every object and every pair of resources is quite unrealistic. Indeed, it is challenging, if not impossible, to produce a reversible external representation for some objects, e.g., when their content includes handlers to kernel objects, system libraries, or hardware-specific, low-level optimizations. Things worsen when the source and destination workers exhibit significant differences in operating systems or hardware architectures.

A possible approach to mitigate this kind of problem is to serialize some entities *by reference*, i.e., to recreate them remotely following the standard procedure instead of reconstructing them from a marshaled internal state. The dill library [56] relies on this strategy for Python modules, which are regularly imported in the destination program's state. Nevertheless, this strategy cannot be applied to stateful objects, which need information about the internal state to be coherently reconstructed.

A more flexible technique allows developers to register, for a particular object type, a pair of marshaling and unmarshaling routines, augmenting a baseline of standard cases directly handled by the library. This approach has been adopted in the distributed version of the FastFlow framework [57], and also dill comes with a `@register` decorator to extend the standard set of serializable types. We choose to stick with this last strategy for our implementation, adopting dill as the base serialization library and adding a dedicated `serializer` subsection in the description of input and output dependencies, as shown in Listing 2. Each entry in this subsection lets users specify `predump` and `postload` routines to transform unsuitable values before marshaling and reobtaining the original object after unmarshaling, respectively.

4.6. Centralized and distributed control plane architecture

The current Jw implementation adopts a master–worker pattern for the control plane, where the driver acts as the master [58, 59]. Despite some potential performance and robustness issues of the centralized control, the master–worker is a widely adopted pattern for the runtime support of parallel and distributed systems. A centralized architecture makes it possible to effectively maintain a coherent global knowledge of the system, simplifying the implementation of algorithms for task graphs unfolding, scheduling, load-balancing, and fault-tolerance [60,61].

The master–worker pattern is largely adopted by many WMSs and task-based parallel libraries, such as Makeflow [4], Pegasus [2], COMPSs [62], HyperLoom [63], and other well-known systems, such as Kubernetes, and Spark [64]. In reality, most of the weaknesses of the master–worker schema can be mitigated with little effort, such as excluding the master from large data exchanges by allowing direct data movements among workers. This can be done either directly by the WMS (as in COMPSs) or by delegating data transfers to external software, e.g. HTCondor (as in Pegasus and Makeflow).

This solution is also adopted in Jw, where the driver is kept outside the data path. It can be used as a relay toward air-gapped or firewalled workers, but data will be moved from one deployment to the next without passing through the driver in the general case. This pushes scalability issues to corner cases such as very fine-grained cells and large-scale parallel executions, which can be approached by coalescing the execution of independent cells in a single macro-cell. Also, the master–worker pattern can be made robust by replicating the master using a third-party distributed coherent database, as it happens with etcd in the Kubernetes control plane. Both these features are currently under development.

5. Evaluation

In this section, we demonstrate how the Jw approach can be effectively applied to common scenarios in the fields of deep learning, scientific simulation, and bioinformatics, enabling interactive analysis pipelines at scale.

In particular, the first example showcases how Notebooks can be used for interactive hyperparameter tuning for DNN training, executing configurable grid search tasks in parallel on GPU-equipped HPC facilities. The second example implements a hybrid Cloud-HPC workflow to perform a training+serving pipeline of a DNN, relying on the HPC computing power for the training step and on the Cloud *-as-a-Service paradigm for inference. A Quantum ESPRESSO[9] [65] simulation workflow is used as representative of a broad class of traditional HPC Molecular Dynamics Simulation (MDS) tools in order to investigate how Jw can enable interactive simulations at scale. Finally, a Bioinformatics pipeline based on the 1000 Genomes project [66] is used to analyze performances in the Cloud.

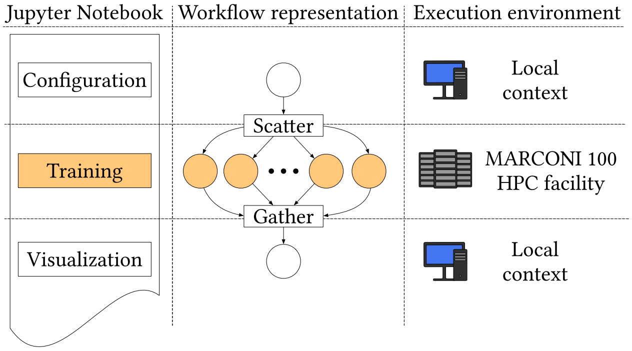


Fig. 5. Graphical representation of the Jw execution plan for the CLAIRE-COVID19 Notebook.

Both the examples and the entire framework's codebase are publicly available.[10]

5.1. Hyperparameter search for DNN assisted COVID-19 diagnosis

To demonstrate how the proposed approach effectively combines usability and scalability, we provide a Jw implementation of the most computationally intensive portion of the *COVID-19 universal pipeline*, developed by the Confederation of Laboratories for Artificial Intelligence Research in Europe (CLAIRE) task force on AI & COVID-19 [67].

The pipeline is composed of a preparatory data processing section and a core training workflow. The goal is to perform a metrics assessment on 11 variants of DNN models, each with its hyper-parameters. In the context of this work, we perform a hyperparameter search for 4 different DenseNet models [68]: DenseNet-121, DenseNet-161, DenseNet-169, and DenseNet-201. To improve classification performances, we adopt a *transfer learning* approach: weights pre-trained on the ImageNet dataset [69] are fine-tuned on a pre-processed subset of the BIMCV-COVID19 dataset [70] using a standardly configured Adam optimizer [71] ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$). Such transfer learning process is configured to run for at most 50 epochs, with early stopping after 10 epochs without improvements in the validation loss.

For each model, we explore 12 different configurations by varying 3 hyperparameters: learning rate (10^{-3}, 10^{-4}, 10^{-5}), weight decay ($5e^{-4}$, $5e^{-5}$), and LR decay step (10, 15). Plus, we perform 5-fold cross-validation on each configuration to reduce variability in the obtained classification metrics, with a total of 60 variants of each model's training process. With Jw, the code could be easily split into three main sections (as shown in Fig. 5):

- An initial configuration section, containing the module imports and the hyperparameters' grid;
- A training section, i.e., a single Notebook cell containing the main training loop;

[9] http://www.quantum-espresso.org
[10] https://github.com/alpha-unito/jupyter-workflow

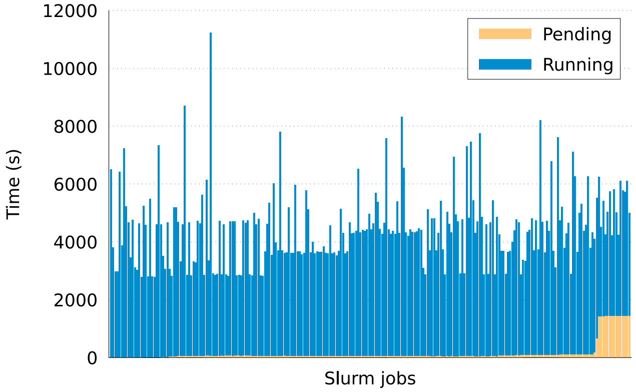


Fig. 6. Execution times for the 240 DenseNet training experiments running in parallel on 240 GPU-equipped nodes of the CINECA MARCONI 100 facility.

- A visualization section, where the metrics of each training experiment can be efficiently analyzed through the `matplotlib` interactive backend.

By marking each multi-valued hyperparameter and the set of cross-validation folds as `scatter` input parameters of the training cell, Jw automatically generates the cartesian product of input configurations and schedules them for concurrent execution. Nevertheless, the effective amount of concurrency strongly depends on the chosen execution environment.

In order to fully take advantage of their embarrassingly parallel nature, we offload the 240 training steps to 240 GPU nodes of the CINECA MARCONI 100 facility, equipped with 2 IBM POWER9 AC922 sockets (16 cores, 3.1 GHz each), 256 GB of RAM, and 4 NVIDIA V100 GPUs (16 GB of memory each). Moreover, we only request a single GPU for each job, trading off training speed for a shorter waiting time in the Slurm queue.

Conversely, the other cells are executed directly in the local context of the Jw kernel, running on a desktop machine equipped with an Intel i7-7700K CPU (4 cores, 8 threads, 4.20 GHz). It is worth noting that the DependencyResolver component correctly identified all the implicit input dependencies, containing aliases of Python modules (i.e., modules imported using the `import as` directive), and remote dataset paths. Moreover, also the serialization and deserialization of the program context worked properly even between two different hardware architectures (an `x86_64` Intel CPU on the local workstation and a `ppc64le` POWER9 on the MARCONI 100 nodes), without the need to implement any custom `predump` and `postload` logic.

Fig. 6 shows the execution time reported by the Slurm `sacct` command for each of the training jobs, including both the time spent in the waiting queue (Pending state) and the effective DNN training time (Running state). Overhead related to data transfers to and from the remote facility was negligible and has not been reported. The vast majority of configurations benefited from the early stopping criterion shortly after 10 epochs, lasting between 50 and 70 min. Nevertheless, the global execution (scatter) of a cell can complete only after the tail of the longest jobs, which took more than 3 h to terminate.

Despite this, the obtained speedup is substantial. Considering only the time spent in the Running state, a single V100 GPU would require about ∼288 h to complete the training. Conversely, Jw allows a ×92 faster execution without sacrificing the Jupyter high-level interactive visualization tools.

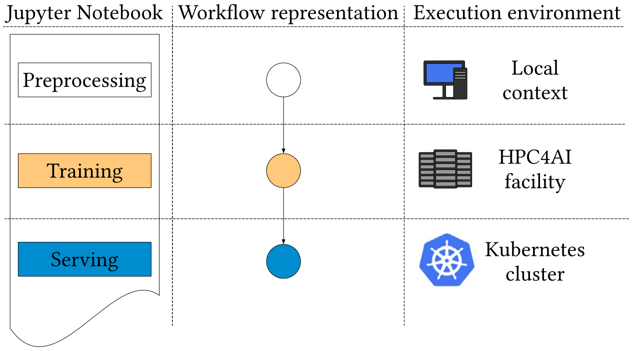


Fig. 7. Graphical representation of the Jw execution plan for the TensorFlow training+serving Notebook.

Listing 3: Data handling strategies in Jw

```
/*************************************************************
Small dataset --> name dependency

[1] dataset_path = "/home/myuser/dataset/path"
    dataset = Preprocess(Load(dataset))

Workflow metadata for cell [2]: */
{
  "step": {
    "in": [{
      "name": "dataset",
      "type": "name"
    }],
    ...
  },
  "target": {...}
}
/* [2] model_spec = ...
       model = Model(model_spec).fit(dataset)

**************************************************************
Huge dataset -> remote path injection

[1] dataset_path = "/home/myuser/dataset/path"

Workflow metadata for cell [2]: */
{
  "step": {
    "in": [{
      "name": "dataset_path",
      "type": "name",
      "value": "/remote/dataset/path"
    }],
    ...
  },
  "target": {...}
}
/* [2] dataset = Preprocess(Load(dataset_path))
       model = Model(model_spec).fit(dataset) */
```

5.2. Training and serving DNNs

In the deep learning field, training+serving pipelines can highly benefit from a hybrid Cloud-HPC execution. Indeed, if HPC facilities with heterogeneous computing nodes are ideal for model training, their queue-based workload management and limited Internet access are not suitable for the serving phase, as inference usually comes with strict real-time requirements and needs publicly exposed REST APIs. In this section, we describe how Jw can efficiently orchestrate pre-processing, training, and serving tasks for a Deep Neural Network (DNN) using TensorFlow [72] (Fig. 7). Since, in this case, we are interested in evaluating design-related aspects rather than performances, we set up a playground with a very small Convolutional Neural Network (CNN) trained on the Fashion-MNIST dataset [73].

After a local data pre-processing phase, the training step computation is offloaded to a node of the HPC4AI facility [74] equipped with 2 Intel Xeon Gold 6230 sockets (20 cores, 2.10 GHz each), 496 GB of RAM, and 4 NVIDIA V100-SXM2 GPUs (32 GB of memory each). Moving data from the local kernel to the remote HPC infrastructure is straightforward, as Fashion-MNIST is relatively small (less than 30 MB). Therefore, we can simply treat the already pre-processed dataset as a `name` dependency, letting StreamFlow manage serialization and transfer operations. It could be more efficient for massive datasets to move both pre-processing and training steps close to data, keeping only a small subset on the local environment for prototyping or debugging purposes. This scenario can be handled by explicitly modifying the dataset path through a `value` directive in the metadata. These two data management strategies are sketched in Listing 3. Even if the host code has been simplified for clarity, it is worth noting that switching between the two different scenarios only requires a regrouping of program instructions in the code cells, without modifying the business logic.

Concerning serialization, some internal data structures prevent the dill library from successfully parsing TensorFlow networks. Nevertheless, Jw allows to easily solve this issue by putting some custom logic in the related `serializer` section to explicitly save the model to a file using Keras utilities, transfer it to the remote executor and load it again in the target program's state. Such logic can also be extended to upload the model on one or more GPU devices if available.

The resulting model is stored in a Docker container when the training completes, which is then published as a Kubernetes Pod hosting the TensorFlow Serving framework. In this case, the Pod is automatically deployed on the HPC4AI Cloud infrastructure by the StreamFlow Helm connector, but cell executions can also be bound to externally managed models (i.e., marked as `external` in the coordination metadata). Therefore, the current example can be configured to send trained models directly to a production server for Continuous Integration purposes, strongly reducing the gap between prototyping and deployment phases in the development lifecycle.

It is worth noting that the DependencyResolver can correctly identify all the input dependencies (both Python modules and pre-processed datasets). Nevertheless, the trained model file needs to be explicitly listed in the input dependencies of the TensorFlow Serving initialization step, since the DependencyResolver cannot discriminate between strings and file paths.

5.3. Interactive simulation at scale: running quantum ESPRESSO on Jupyter

In order to assess the Jw capabilities to enable interactive simulations of realistic, large-scale systems, we implement a Notebook describing a multi-step simulation workflow in Quantum ESPRESSO. In particular, the analyzed workflow implements a Car-Parrinello simulation of a mixture of H_2O, NH_3 and CH_4 molecules to represent the basic ingredients of life (the so-called primordial soup). The simulation aims to explore the phase space to find where C–H, O–H and N–H bonds break up, forming more complex organic molecules. Several Car-Parrinello simulations at different pressure–temperature points (P, T) are needed to simulate the phase diagram.

As depicted in Fig. 8, the workflow proceeds as follows. The first four cells, common to all simulations, prepare a starting state at room temperature and pressure from a random distribution of the three molecules. Then the pipeline forks to simulate different temperatures through Nosé–Hoover thermostats (cell 5). Finally, for each temperature, the simulation forks again to simulate each temperature at several values of pressure using the Parrinello-Rahman constant pressure Lagrangian (cell 6).

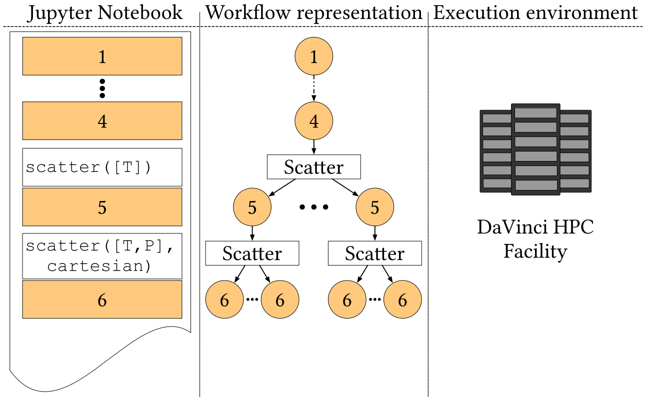


Fig. 8. Graphical representation of the Jw execution plan for the Quantum ESPRESSO Notebook.

Table 1
Weak scalability for the Quantum ESPRESSO Notebook when executed manually on PBS and in both Jw execution modes.

Step	Nodes	PBS (s)	Jw Interactive (s)	Jw Bulk (s)
Cell 5	2	408	461	413
	4	407	490	415
	8	407	553	418
Cell 6	2	459	536	465
	8	459	706	469
	32	461	1861	474

In the following discussion, we focus on the last two steps, as the others are trivial. Using the Jw metadata format, cell 5 can be parallelized by scattering on T, while cell 6 can use a cartesian product operator to scatter over all (P, T) combinations. In the interactive mode, where concurrency is confined inside single cells, cell 6 can start only when all cell 5 tasks terminate and all their outputs have been copied back to the driver node. This mode allows users to inspect cell outputs immediately, but it can introduce significant overhead. Conversely, in the bulk evaluation mode, data are moved only if necessary, and redundant Gather-Scatter combinations are removed to increase concurrency, as explained in Section 3.3.

We offload the execution of each step to two CPU nodes of davinci-1, the Leonardo S.p.A. HPC system. Each node is equipped with 2 Intel Xeon Platinum 8260 sockets (24 cores, 2.40 GHz each) and 1 TB of RAM. We analyze the weak scalability of the application by running it on 1, 4 and 16 (P, T) points, comparing for each setting the time to complete steps 5 and 6 with bare PBS, interactive notebooks and bulk evaluation. Results are reported in Table 1. It is worth noting how the overhead introduced by the interactive execution mode becomes predominant with 16 (P, T) points, while it remains totally negligible in the bulk evaluation mode.

This is a basic setup to test the effectiveness of the proposed approach. One can easily improve it, as the Notebook is general enough to be easily adapted to any simulation of P-T phase diagram of any material, scaling a single (p, T) point simulation up to several thousands of nodes.

Quantum ESPRESSO has been shown to scale well to petascale systems, and it is currently addressing the exascale challenges [75]. Therefore, peak performances are not an issue. Nevertheless, much of the Quantum ESPRESSO performance derives from the linked matrix multiplication libraries, which are tightly

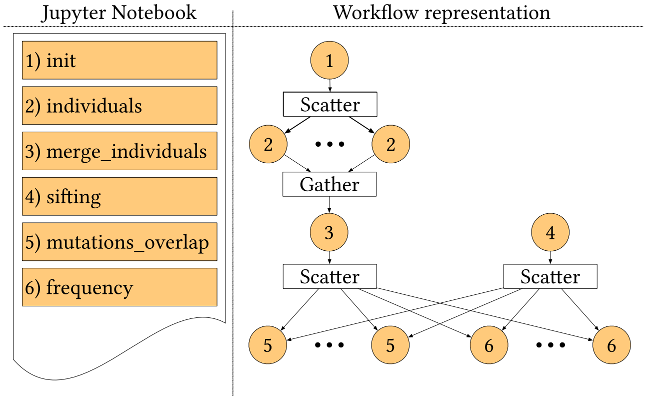


Fig. 9. Graphical representation of the Jw execution plan for the 1000-genome workflow.

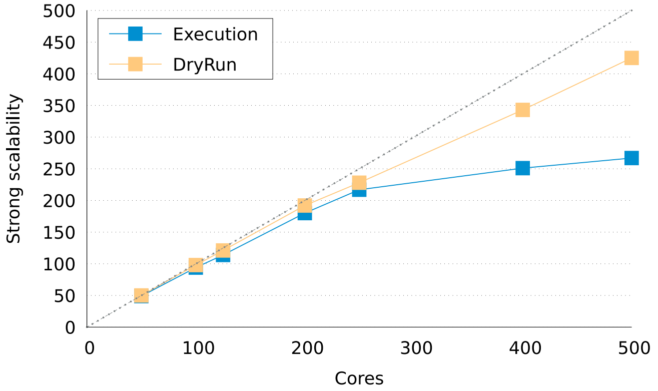


Fig. 10. Speedup obtained executing the 1000-genome workflow on the HPC4AI Cloud. The blue curve refers to a real execution. The orange curve shows a DryRun of the same workflow aimed at assessing Jw overhead (the business code is substituted with sleeps matching execution time).

coupled with the underlying hardware technology at compile time. Tweaking performances of this kind of libraries is out of reach of a large portion of domain experts, but linking Quantum ESPRESSO with low-performing or badly-compiled versions of BLAS and LAPACK can have a huge impact on the time-to-solution. With its capability to seamlessly offload computation to optimized execution environments on HPC facilities, Jw enables domain experts to run simulations interactively, exploring and validating the outputs of the first (lightweight) steps before proceeding with the heaviest portions of the pipeline.

5.4. Running the 1000-genome workflow on Kubernetes

To investigate Jw strong scalability on a distributed infrastructure, we execute an instance of the 1000-genome workflow on a Kubernetes cluster running on top of the HPC4AI OpenStack-based Cloud. A detailed description of the 1000-genome workflow, originally implemented in Pegasus, is available in literature [76]. Fig. 9 shows the Jupyter Notebook representation of the workflow (as a list of 6 cells) and the corresponding DAG automatically extracted by the Jw runtime. Notice that, when dealing with complex workflows, the amount of concurrency allowed by the DAG evaluation strategy becomes significant.

We selected 1000-genome workflow for three main reasons:

- Pegasus is a state-of-the-art representative of WMSs for High Throughput Computing (HTC), supporting execution environments without shared data spaces (via HTCondor);
- The host code of each step is written in either Bash or Python, both supported by the IPython kernel;
- The critical portion of the workflow is a highly-parallel step, composed of 2000 independent short tasks (∼120s each) which are critical for batch workload managers, but that can be executed at scale on on-demand Cloud resources (e.g., Kubernetes).

Porting the host code to Jw merely requires creating a cell for each step by copy-pasting the original code. Concerning the coordination layer, the two WMSs adopt a strictly diverse approach. Pegasus requires the user to manually compile a static workflow graph, specifying all the input and output dependencies of each step. This technique is extremely powerful in terms of expressiveness, as expressible graphs are not limited to the composition of a predefined set of patterns. Conversely, Jw is somewhat limited by the original sequential nature of Jupyter Notebooks, even if DAG evaluation strongly mitigates the constraints.

For its part, Jw can seamlessly deal with dynamic outputs. This is a fundamental requirement for a prototyping technology, where the exact structure of output dependencies is often not known a priori. Moreover, dynamically generated DAGs also increase reusability since the same notebook can be used to perform entire families of similar experiments by simply changing the input values (as analyzed in Section 5.3). Conversely, encoding dynamic data dependencies in a Pegasus workflow requires users to explicitly embed complex just-in-time compilation steps in the workflow graph.

For performance evaluation, we focus on the `individuals` step, which constitutes the bottleneck of the workflow. We measure the strong scaling of an 8-chromosomes instance of the 1000-genome workflow on 500 concurrent Kubernetes Pods. The underlying Kubernetes cluster is composed of 3 control plane VMs (4 cores, 8GB of RAM each) and 16 large worker VMs (40 cores, 120GB of RAM each), interconnected with a 10Gbps Ethernet. Each Pod requests 1 core and 2GB of RAM, and mounts a 1GB tmpfs. In the Pegasus workflow a chromosome input file is made available to all involved workers in its entirety, and each of them selects a different partition. Conversely, Jw scatters the dataset, transferring to each worker only the strictly required data.

To assess Jw performance on a Cloud, where all the resources (CPUs, memory, network, disks) are typically overprovisioned and subject to load generated by other users, we developed a DryRun version of the code serving as the baseline. The DryRun simulates the workflow behavior without actually using CPU cores and network bandwidth: the business code is substituted with sleeps of the expected average timespan of the task sampled from a normal distribution and communications are replaced with a small message. The Fig. 10 shows the strong scalability of real execution and DryRun. The real execution scales reasonably well up to 250 containers. Then it starts suffering from the Kubernetes master bottleneck for data distribution.

The DryRun shows that the intrinsic overhead introduced by the Jw runtime synchronizations (orange curve) keeps a reasonably linear gap against ideal speedup (at least up to 500 Pods). In the Cloud/Kubernetes setting (as theoretically expected) the crucial aspect for performance is the tuning of communication/-computation ratio at the Kubernetes master, which does not leave much room for optimization in I/O-bound problems (as the 1000-genome workflow).

In these cases, viable optimization paths concern the distribution or elimination of data movements. The former can be realized by implementing direct communication channels between worker nodes, leaving the Kubernetes master out of the critical path, while the latter is enabled by rewriting rules such as Map fusion (see Section 3.3). As described in Section 6 below, we are actively working on designing other cases.

6. Conclusion and future work

The widespread diffusion of Jupyter Notebooks (and similar software, such as Zeppelin[11]) in both academic and industrial workloads made them a de-facto standard for rapid prototyping and interactive data analysis. The reason for their success is their capability to support step-by-step execution and interactive tuning of software pipelines, boosting the productivity of scientists. A second reason is their portability, which is a prerequisite for reproducibility. However, the lack of support for complex workflows and the challenging integration with hybrid Cloud-HPC architectures undoubtedly hampered their adoption in production workloads.

After many years of evolution and application co-design, HPC systems, even at the extreme scale, are facing new challenges, inter-alia, the support for AI applications, and frameworks that are eager to compute power and natively run on GPUs. The exploitation of a growing industrial market is boosted by the ability of HPC solutions to process large volumes of data with speed and accuracy. Cloud providers have also started to offer turn-key HPC solution environments tailored to specific enterprise needs. A hybrid Cloud-HPC solution results in better efficiencies, but it requires a new generation of deployment tools to effectively support rapid prototyping, interactive tuning, and portability of *modern* HPC applications. Indeed, they are no longer a single co-designed kernel but complex pipelines (such as digital twins) embracing different applications and software stacks.

In this work, we introduced *Jupyter-workflow* (Jw), a novel methodology and a tool that unleash Jupyter Notebook's superior productivity and portability features in the HPC area, explicitly targeting Cloud resources and their workload managers (such as Kubernetes), HPC platforms together with their system software (such as SLURM), and their coupled exploitation usage. We advocate Jw as the first representative of a new class of interfaces for *modern* HPC applications. We formalized concurrent (parallel and distributed) semantics for Jupyter Notebooks supporting interactive and batch executions. We tested Jw on four different application pipelines and four different settings, with the driver running on a desktop and the computationally demanding cells offloaded to either HPC facilities or container-based Cloud environments. The Jw codebase is publicly available under the LGPLv3 license.

The theoretical speedup reachable executing a Jw Notebook depends on the *makespan*, i.e., the longest path of cells exhibiting a chain of data dependencies (see Sec. 3.3). The actual speedup of any specific execution also depends on the number of available processing elements, the scheduling strategy adopted for independent steps, and in the case of heterogeneous ones, also from the mapping of steps onto different executors. The problem of deriving an optimal scheduling/mapping for a Jw workflow can be reduced to a task graph execution problem, that is NP-complete in the most general case, but also extensively studied, and efficiently approximable [77]. StreamFlow does not implement predictive models to infer the duration of each task and, for this, adopts a version of first-come-first-served scheduling that considers the deployment declared in the cells. Notice that the Jw Scatter/Gather operators (and the related rewriting rules) introduce independent steps in the workflow, enhancing the ratio between the total number of steps and the number of steps in the longest path that eventually models the speedup. As common in parallel computing, the strong scalability (i.e., the speedup) and weak scalability are bound by the Amhdal's and Gustafson's laws, respectively [78].

In large-scale executions and when the available processing elements exceed the need, the Jw driver (that acts as a master) could become a bottleneck. However, DAG-evaluation coupled with rewriting rules can significantly reduce the overhead induced by the Jw driver (see Section 5.3). Conversely, a good scheduling policy becomes more important than a low overhead when there is more parallelism than available processing elements. The default mapping strategy of steps into processing elements adopted by StreamFlow privileges data-locality, in order to minimize data transfers [12]. However, the modular nature of StreamFlow makes it easy for users to implement a more sophisticated strategy. Still, with the default approach Jw can effectively deal with 2000 fine-grained concurrent tasks distributed among up to 500 homogeneous processing elements, with data movements constituting by far the bottleneck for strong scalability (see Section 5.4).

Future works. In the short term, we aim to apply the proposed methodology to other kernels, such as Julia and R. Also, we aim to extend the `DependencyResolver` to help the programmer in detecting critical data movements and to provide the users with explicit collective communications alternatives (such as Scatter/Gather and Reduce), which can be separately optimized. On the same ground, we will develop optimized dispatch/scheduling policies aiming to minimize the data movements. Plus, we are already testing a novel Cloud service (called Jupyter-as-a-Service) based on the Jw and JupyterHub as a modern frontend of an HPC-accelerated Cloud system for AI applications.

In the longer term, we aim at contributing to the ecosystems of tools addressing the vision of modular extreme-scale systems, where HPC and Cloud might play the role of modules in a broader computing continuum: an HPC cluster can play the role of an accelerator for an interactive data analysis pipeline running on the Cloud. This modular vision fosters co-design and early access to novel high-performance architectures, which are expected to be specialized and exotic (e.g., Quantum and Neuromorphic). Jupyter-workflow's clear separation of data flow and deployments makes it possible to support testing and integration of new modules into existing pipelines. This approach allows confining co-design effort within specific workflow steps, enabling separated optimization and testing of alternative implementations of the same step. As an example, we are currently designing a "quantum proxy" for Jw targeting different quantum machines available via Cloud services (D-wave and Pasqal).

CRediT authorship contribution statement

Iacopo Colonnelli: Conceptualization, Methodology, Software, Validation, Investigation, Writing – original draft, Visualization. **Marco Aldinucci:** Methodology, Formal analysis, Writing – original draft, Resources, Supervision, Funding acquisition. **Barbara Cantalupo:** Investigation, Writing – original draft, Project administration. **Luca Padovani:** Methodology, Formal analysis, Writing – review & editing. **Sergio Rabellino:** Investigation, Writing – review & editing. **Concetto Spampinato:** Investigation, Resources, Supervision. **Roberto Morelli:** Investigation. **Rosario Di Carlo:** Investigation. **Nicolò Magini:** Investigation. **Carlo Cavazzoni:** Investigation, Resources, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

[11] https://zeppelin.apache.org

Acknowledgments

This article describes work undertaken in the context of the DeepHealth project,[12] *"Deep-Learning and HPC to Boost Biomedical Applications for Health"* which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 825111 [79], and the ACROSS project,[13] *"HPC Big Data Artificial Intelligence Cross Stack Platform Towards Exascale"* which has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No. 955648 [80]. This work has been partially supported by the HPC4AI project[14] which has been funded by the Region Piedmont POR-FESR 2014-20 (INFRA-P) [74].

References

[1] T.M. Oinn, R.M. Greenwood, M. Addis, M.N. Alpdemir, J. Ferris, K. Glover, C.A. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P.W. Lord, M.R. Pocock, M. Senger, R. Stevens, A. Wipat, C. Wroe, Taverna: lessons in creating a workflow environment for the life sciences, Concurr. Comput.: Pract. Exper. 18 (10) (2006) 1067–1100, http://dx.doi.org/10.1002/cpe.993.

[2] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. Maechling, R. Mayani, W. Chen, R.F. da Silva, M. Livny, R.K. Wenger, Pegasus, a workflow management system for science automation, Future Gener. Comput. Syst. 46 (2015) 17–35, http://dx.doi.org/10.1016/j.future.2014.10.008.

[3] J. Köster, S. Rahmann, Snakemake - a scalable bioinformatics workflow engine, Bioinformatics 28 (19) (2012) 2520–2522, http://dx.doi.org/10.1093/bioinformatics/bts480.

[4] M. Albrecht, P. Donnelly, P. Bui, D. Thain, Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids, in: Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, SWEET@SIGMOD 2012, Scottsdale, AZ, USA, May 20, 2012, 2012, p. 1, http://dx.doi.org/10.1145/2443416.2443417.

[5] P. Di Tommaso, M. Chatzou, E.W. Floden, et al., Nextflow enables reproducible computational workflows, Nature Biotechnol. 35 (4) (2017) 316–319, http://dx.doi.org/10.1038/nbt.3820.

[6] E. Lee, T. Parks, Dataflow process networks, Proc. IEEE 83 (5) (1995) 773–801.

[7] E. Afgan, D. Baker, B. Batut, M. van den Beek, D. Bouvier, M. Cech, J. Chilton, D. Clements, N. Coraor, B.A. Grüning, A. Guerler, J. Hillman-Jackson, S.D. Hiltemann, V. Jalili, H. Rasche, N. Soranzo, J. Goecks, J. Taylor, A. Nekrutenko, D.J. Blankenberg, The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update, Nucleic Acids Res. 46 (Webserver-Issue) (2018) W537–W544, http://dx.doi.org/10.1093/nar/gky379.

[8] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M.B. Jones, E.A. Lee, J. Tao, Y. Zhao, Scientific workflow management and the Kepler system, Concurr. Comput.: Pract. Exper. 18 (10) (2006) 1039–1065, http://dx.doi.org/10.1002/cpe.994.

[9] M.R. Berthold, N. Cebron, F. Dill, T.R. Gabriel, T. Kötter, T. Meinl, P. Ohl, C. Sieb, K. Thiel, B. Wiswedel, KNIME: the Konstanz information miner, in: Data Analysis, Machine Learning and Applications - Proceedings of the 31st Annual Conference of the Gesellschaft für Klassifikation e.V., Albert-Ludwigs-Universität Freiburg, March 7–9, 2007, in: Studies in Classification, Data Analysis, and Knowledge Organization, Springer, 2007, pp. 319–326, http://dx.doi.org/10.1007/978-3-540-78246-9_38.

[10] M. Kotliar, A.V. Kartashov, A. Barski, CWL-airflow: A lightweight pipeline manager supporting common workflow language, GigaScience 8 (7) (2019) http://dx.doi.org/10.1093/gigascience/giz084.

[11] J. Vivian, A.A. Rao, F.A. Nothaft, et al., Toil enables reproducible, open source, big biomedical data analyses, Nature Biotechnol. 35 (4) (2017) 314–316, http://dx.doi.org/10.1038/nbt.3772.

[12] I. Colonnelli, B. Cantalupo, I. Merelli, M. Aldinucci, StreamFlow: cross-breeding cloud with HPC, IEEE Trans. Emerg. Top. Comput. (2020) http://dx.doi.org/10.1109/TETC.2020.3019202.

[13] P. Amstutz, M.R. Crusoe, N. Tijanić, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, J. Kern, D. Leehr, H. Ménager, M. Nedeljkovich, M. Scales, S. Soiland-Reyes, L. Stojanovic, Common workflow language, v1.0, 2016, http://dx.doi.org/10.6084/m9.figshare.3115156.v2.

[14] W.M.P. van der Aalst, A.H.M. ter Hofstede, YAWL: Yet another workflow language, Inf. Syst. 30 (4) (2005) 245–275, http://dx.doi.org/10.1016/j.is.2004.02.002.

[15] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R.M. Badia, J. Torres, T. Cortes, J. Labarta, PyCOMPSs: Parallel computational workflows in python, J. Supercomput. Appl. High Perform. Comput. 31 (1) (2017) 66–82, http://dx.doi.org/10.1177/1094342015594678.

[16] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M.I. Jordan, I. Stoica, Ray: A distributed framework for emerging AI applications, in: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Carlsbad, CA, USA, October 8–10, 2018, pp. 561–577.

[17] Y. Babuji, A. Woodard, Z. Li, D.S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J.M. Wozniak, I. Foster, M. Wilde, K. Chard, Parsl: Pervasive parallel programming in Python, in: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '19, ACM, New York, NY, USA, 2019, pp. 25–36, http://dx.doi.org/10.1145/3307681.3325400.

[18] H.G. Baker, C. Hewitt, The incremental garbage collection of processes, in: Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages, USA, August 15–17, 1977, ACM, 1977, pp. 55–59, http://dx.doi.org/10.1145/800228.806932.

[19] C. Misale, M. Drocco, M. Aldinucci, G. Tremblay, A comparison of big data frameworks on a layered dataflow model, Parallel Process. Lett. 27 (01) (2017) 1–20, http://dx.doi.org/10.1142/S0129626417400035.

[20] T.M. McPhillips, T. Song, T. Kolisnik, S. Aulenbach, K. Belhajjame, K. Bocinsky, Y. Cao, F. Chirigati, S.C. Dey, J. Freire, D.N. Huntzinger, C. Jones, D. Koop, P. Missier, M. Schildhauer, C.R. Schwalm, Y. Wei, J. Cheney, M. Bieda, B. Ludäscher, Yesworkflow: A user-oriented, language-independent tool for recovering workflow information from scripts, 2015, CoRR abs/1502.02403, arXiv:1502.02403.

[21] B. Lerner, E.R. Boose, Rdatatracker: Collecting provenance in an interactive scripting environment, in: A. Chapman, B. Ludäscher, A. Schreiber (Eds.), 6th Workshop on the Theory and Practice of Provenance, TaPP'14, Cologne, Germany, June 12–13, 2014, USENIX Association, 2014.

[22] L.A.M.C Carvalho, K. Belhajjame, C.B. Medeiros, Converting scripts into reproducible workflow research objects, in: IEEE International Conference on e-Science, e-Science 2016, Baltimore, MD, USA, October 23–27, 2016, 2016, pp. 71–80, http://dx.doi.org/10.1109/eScience.2016.7870887.

[23] J.F. Pimentel, L. Murta, V. Braganholo, J. Freire, noworkflow: a tool for collecting, analyzing, and managing provenance from python scripts, Proc. VLDB Endow. 10 (12) (2017) 1841–1844, http://dx.doi.org/10.14778/3137765.3137789.

[24] M. Baranowski, A. Belloum, M. Bubak, M. Malawski, Constructing workflows from script applications, Sci. Program. 20 (4) (2012) 359–377, http://dx.doi.org/10.3233/SPR-120358.

[25] M. Malawski, T. Gubala, M. Kasztelnik, T. Bartynski, M. Bubak, F. Baude, L. Henrio, High-level scripting approach for building component-based applications on the grid, in: Making Grids Work: Proceedings of the CoreGRID Workshop on Programming Models Grid and P2P System Architecture Grid Systems, Tools and Environments, 12–13 June 2007, Heraklion, Crete, Greece, 2007, pp. 309–321, http://dx.doi.org/10.1007/978-0-387-78448-9_25.

[26] A.R. Runnalls, C.A. Silles, Provenance tracking in R, in: P. Groth, J. Frew (Eds.), Provenance and Annotation of Data and Processes - 4th International Provenance and Annotation Workshop, IPAW 2012, Santa Barbara, CA, USA, June 19–21, 2012, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 7525, Springer, 2012, pp. 237–239, http://dx.doi.org/10.1007/978-3-642-34222-6_25.

[27] D. Tariq, M. Ali, A. Gehani, Towards automated collection of application-level data provenance, in: U.A. Acar, T.J. Green (Eds.), 4th Workshop on the Theory and Practice of Provenance, TaPP'12, Boston, MA, USA, June 14–15, 2012, USENIX Association, 2012.

[28] R. Mitchell, L. Pottier, S. Jacobs, R.F. da Silva, M. Rynge, K. Vahi, E. Deelman, Exploration of workflow management systems emerging features from users perspectives, in: 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, December 9-12, 2019, IEEE, 2019, pp. 4537–4544, http://dx.doi.org/10.1109/BigData47090.2019.9005494.

[29] E. Larsonneur, J. Mercier, N. Wiart, E.L. Floch, O. Delhomme, V. Meyer, Evaluating workflow management systems: A bioinformatics use case, in: IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2018, Madrid, Spain, December 3–6, 2018, IEEE Computer Society, 2018, pp. 2773–2775, http://dx.doi.org/10.1109/BIBM.2018.8621141.

[30] T. Kluyver, B. Ragan-Kelley, F. Pérez, B.E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J.B. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, et al., Jupyter notebooks - a publishing format for reproducible computational workflows, in: Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, June 7–9, 2016, 2016, pp. 87–90, http://dx.doi.org/10.3233/978-1-61499-649-1-87.

[12] https://deephealth-project.eu/

[13] https://www.acrossproject.eu/

[14] https://hpc4ai.it/

[31] R.W. Cottingham, The DOE systems biology knowledgebase (kbase): progress towards a system for collaborative and reproducible inference and modeling of biological function, in: Proceedings of the 6th ACM Conference on Bioinformatics, Computational Biology and Health Informatics, BCB 2015, Atlanta, GA, USA, September 9–12, 2015, ACM, 2015, p. 510, http://dx.doi.org/10.1145/2808719.2811433.

[32] A.P.e. a. Arkin, Kbase: The united states department of energy systems biology knowledgebase, Nature Biotechnol. 36 (7) (2018) 566–569, http://dx.doi.org/10.1038/nbt.4163.

[33] M. Reich, T. Liefeld, J. Gould, J. Lerner, P. Tamayo, J.P. Mesirov, Genepattern 2.0, Nature Genet. 38 (5) (2006) 500–501, http://dx.doi.org/10.1038/ng0506-500.

[34] G. Wang, B. Peng, Script of scripts: A pragmatic workflow system for daily computational research, PLoS Comput. Biol. 15 (2) (2019) http://dx.doi.org/10.1371/journal.pcbi.1006843.

[35] L.A.M.C Carvalho, R. Wang, Y. Gil, D. Garijo, Niw: Converting notebooks into workflows to capture dataflow and provenance, in: I. Tiddi, G. Rizzo, Ó. Corcho (Eds.), Proceedings of Workshops and Tutorials of the 9th International Conference on Knowledge Capture, K-CAP2017, Austin, Texas, USA, December 4th, in: CEUR Workshop Proceedings, vol. 2065, CEUR-WS.org, 2017, pp. 12–16.

[36] Y. Gil, V. Ratnakar, J. Kim, P.A. González-Calero, P. Groth, J. Moody, E. Deelman, Wings: Intelligent workflow-based design of computational experiments, IEEE Intell. Syst. 26 (1) (2011) 62–72, http://dx.doi.org/10.1109/MIS.2010.9.

[37] D. Koop, J. Patel, Dataflow notebooks: Encoding and tracking dependencies of cells, in: 9th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2017, Seattle, WA, USA, June 23, 2017, 2017.

[38] S. Macke, A.G. Parameswaran, H. Gong, D.J.L. Lee, D. Xin, A. Head, Fine-grained lineage for safer notebook interactions, Proc. VLDB Endow. 14 (6) (2021) 1093–1101.

[39] M. Brachmann, W. Spoth, O. Kennedy, B. Glavic, H. Mueller, S. Castelo, C. Bautista, J. Freire, Your notebook is not crumby enough, replace it, in: 10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12–15, 2020, www.cidrdb.org, 2020, Online Proceedings.

[40] J.F. Pimentel, L. Murta, V. Braganholo, J. Freire, A large-scale study about quality and reproducibility of jupyter notebooks, in: M.D. Storey, B. Adams, S. Haiduc (Eds.), Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26–27 May 2019, Montreal, Canada, IEEE/ACM, 2019, pp. 507–517, http://dx.doi.org/10.1109/MSR.2019.00077.

[41] D. Yin, Y. Liu, A. Padmanabhan, J. Terstriep, J. Rush, S. Wang, CyberGIS-jupyter framework for geospatial analytics at scale, in: D.L. Hart, M. Dahan (Eds.), Proceedings of the Practice and Experience in Advanced Research Computing 2017: Sustainability, Success and Impact, PEARC 2017, New Orleans, LA, USA, July 9–13, 2017, ACM, 2017, pp. 18:1–18:8, http://dx.doi.org/10.1145/3093338.3093378.

[42] A. Prout, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, M. Houle, M. Jones, P. Michaleas, L. Milechin, J. Mullen, A. Rosa, S. Samsi, A. Reuther, J. Kepner, MIT SuperCloud portal workspace: Enabling HPC web application deployment, in: 2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12–14, 2017, IEEE, 2017, pp. 1–6, http://dx.doi.org/10.1109/HPEC.2017.8091097.

[43] M. Milligan, Interactive HPC gateways with Jupyter and Jupyterhub, in: D.L. Hart, M. Dahan (Eds.), Proceedings of the Practice and Experience in Advanced Research Computing 2017: Sustainability, Success and Impact, PEARC 2017, New Orleans, LA, USA, July 9–13, 2017, ACM, 2017, pp. 63:1–63:4, http://dx.doi.org/10.1145/3093338.3104159.

[44] B. Glick, J. Mache, Jupyter notebooks and user-friendly HPC access, in: 2018 IEEE/ACM Workshop on Education for High-Performance Computing, EduHPC@SC, EduHPC@SC, Dallas, TX, USA, November 12, 2018, IEEE, 2018, pp. 11–20, http://dx.doi.org/10.1109/EduHPC.2018.00005.

[45] R.C. Thomas, S. Cholia, K. Mohror, J.M. Shalf, Interactive supercomputing with Jupyter, Comput. Sci. Eng. 23 (2) (2021) 93–98, http://dx.doi.org/10.1109/MCSE.2021.3059037.

[46] T.E. Odaka, A. Banihirwe, G. Eynard-Bontemps, A. Ponte, G. Maze, K. Paul, J. Baker, R. Abernathey, The pangeo ecosystem: Interactive computing tools for the geosciences: Benchmarking on HPC, in: Tools and Techniques for High Performance Computing - Selected Workshops, HUST, SE-HER and WIHPC, Held in Conjunction with SC 2019, Denver, CO, USA, November 17-18, 2019, Revised Selected Papers, in: Communications in Computer and Information Science, vol. 1190, Springer, 2019, pp. 190–204, http://dx.doi.org/10.1007/978-3-030-44728-1_12.

[47] E. Deelman, T. Peterka, I. Altintas, C.D. Carothers, K.K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, J.S. Vetter, The future of scientific workflows, Int. J. High Perform. Comput. Appl. 32 (1) (2018) 159–175, http://dx.doi.org/10.1177/1094342017704893.

[48] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, X10: An object-oriented approach to non-uniform cluster computing, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '05, Association for Computing Machinery, New York, NY, USA, 2005, pp. 519–538, http://dx.doi.org/10.1145/1094811.1094852.

[49] B. Chamberlain, D. Callahan, H. Zima, Parallel programmability and the chapel language, Int. J. Supercomput. Appl. High Perform. Comput. 21 (3) (2007) 291–312, http://dx.doi.org/10.1177/1094342007078442.

[50] Y. Zheng, A. Kamil, M.B. Driscoll, H. Shan, K. Yelick, Upc++: a pgas extension for c++, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, 2014, pp. 1105–1114, http://dx.doi.org/10.1109/IPDPS.2014.115.

[51] K. Fürlinger, C. Glass, J. Gracia, A. Knüpfer, J. Tao, D. Hünich, K. Idrees, M. Maiterth, Y. Mhedheb, H. Zhou, Dash: Data structures and algorithms with support for hierarchical locality, in: Euro-Par 2014: Parallel Processing Workshops, Springer International Publishing, Cham, 2014, pp. 542–552.

[52] C. Bell, D. Bonachea, A new DMA registration strategy for pinning-based high performance networks, in: 17th International Parallel and Distributed Processing Symposium, IPDPS 2003, 22–26 April 2003, Nice, France, in: CD-ROM/Abstracts Proceedings, IEEE Computer Society, 2003, p. 198, http://dx.doi.org/10.1109/IPDPS.2003.1213363.

[53] M. Drocco, Parallel Programming with Global Asynchronous Memory: Models, C++ APIS and Implementations (Ph.D. thesis), Computer Science Department, University of Torino, 2017, http://dx.doi.org/10.5281/zenodo.1037585.

[54] A.J. Bernstein, Analysis of programs for parallel processing, IEEE Trans. Electron. Comput. EC-15 (5) (1966) 757–763, http://dx.doi.org/10.1109/PGEC.1966.264565.

[55] J. Darlington, Y. Guo, H.W. To, J. Yang, Functional skeletons for parallel coordination, in: Euro-Par '95 Parallel Processing, First International Euro-Par Conference, Proceedings, Stockholm, Sweden, August 29–31, 1995, in: Lecture Notes in Computer Science, vol. 966, Springer, 1995, pp. 55–66, http://dx.doi.org/10.1007/BFb0020455.

[56] M.M. McKerns, L. Strand, T. Sullivan, A. Fang, M.A.G. Aivazis, Building a framework for predictive science, 2012, CoRR abs/1202.1056.

[57] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, M. Torquati, Targeting distributed systems in FastFlow, in: Euro-Par 2012 Workshops, Proc. of the CoreGrid Workshop on Grids, Clouds and P2P Computing, in: LNCS, vol. 7640, Springer, 2013, pp. 47–56, http://dx.doi.org/10.1007/978-3-642-36949-0_7.

[58] M. Cole, Algorithmic skeletons: Structured management of parallel computations, in: Research Monographs in Par. and Distrib. Computing, Pitman, 1989.

[59] M. Danelutto, R.D. Meglio, S. Orlando, S. Pelagatti, M. Vanneschi, A methodology for the development and the support of massively parallel programs, Future Gener. Comput. Syst. 8 (1–3) (1992) 205–220, http://dx.doi.org/10.1016/0167-739X(92)90040-I.

[60] H. González-Vélez, M. Leyton, A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers, Softw. - Pract. Exp. 40 (12) (2010) 1135–1160, http://dx.doi.org/10.1002/spe.1026.

[61] V. Amaral, B. Norberto, M. Goulão, M. Aldinucci, S. Benkner, A. Bracciali, P. Carreira, E. Celms, L. Correia, C. Grelck, H. Karatza, C. Kessler, P. Kilpatrick, H. Martiniano, I. Mavridis, S. Pllana, A. Respício, J. Simão, L. Veiga, A. Visa, Programming languages for data-intensive hpc applications: A systematic mapping study, in: Parallel Computing, 2020, 102584, http://dx.doi.org/10.1016/j.parco.2019.102584.

[62] F. Marozzo, F. Lordan, R. Rafanell, D. Lezzi, D. Talia, R.M. Badia, Enabling cloud interoperability with compss, in: Euro-Par 2012 Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 16–27.

[63] V. Cima, S. Böhm, J. Martinovic, J. Dvorský, K. Janurová, T.V. Aa, T.J. Ashby, V.I. Chupakhin, Hyperloom: A platform for defining and executing scientific pipelines in distributed environments, in: Proceedings of the 9th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and 7th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms, PARMA-DITAM@HiPEAC 2018, Manchester, United Kingdom, January 23–23, 2018, 2018, pp. 1–6, http://dx.doi.org/10.1145/3183767.3183768.

[64] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: a unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65, http://dx.doi.org/10.1145/2934664.

[65] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G.L. Chiarotti, M. Cococcioni, I. Dabo, A.D. Corso, S. de Gironcoli, S. Fabris, G. Fratesi, R. Gebauer, U. Gerstmann, C. Gougoussis, A. Kokalj, M. Lazzeri, L. Martin-Samos, N. Marzari, F. Mauri, R. Mazzarello, S. Paolini, A. Pasquarello, L. Paulatto, C. Sbraccia, S. Scandolo, G. Sclauzero,

A.P. Seitsonen, A. Smogunov, P. Umari, R.M. Wentzcovitch, QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials, J. Phys.: Condens. Matter 21 (39) (2009) 395502, http://dx.doi.org/10.1088/0953-8984/21/39/395502.

[66] Auton, et al., A global reference for human genetic variation, Nature 526 (7571) (2015) 68–74, http://dx.doi.org/10.1038/nature15393.

[67] I. Colonnelli, B. Cantalupo, R. Esposito, M. Pennisi, C. Spampinato, M. Aldinucci, HPC application cloudification: The StreamFlow toolkit, in: 12th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 10th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms, PARMA-DITAM 2021, in: Open Access Series in Informatics (OASIcs), vol. 88, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2021, pp. 5:1–5:13, http://dx.doi.org/10.4230/OASIcs.PARMA-DITAM.2021.5.

[68] G. Huang, Z. Liu, L. van der Maaten, K.Q. Weinberger, Densely connected convolutional networks, in: 2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21–26, 2017, IEEE Computer Society, 2017, pp. 2261–2269, http://dx.doi.org/10.1109/CVPR.2017.243.

[69] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A.C. Berg, L. Fei-Fei, ImageNet Large scale visual recognition challenge, Int. J. Comput. Vis. 115 (3) (2015) 211–252, http://dx.doi.org/10.1007/s11263-015-0816-y.

[70] M. de la Iglesia-Vayá, J.M. Saborit, J.A. Montell, A. Pertusa, A. Bustos, M. Cazorla, J. Galant, X. Barber, D. Orozco-Beltrán, F. García-García, M. Caparrós, G. González, J.M. Salinas, BIMCV COVID-19+: a large annotated dataset of RX and CT images from COVID-19 patients, 2020, CoRR abs/2006.01174.

[71] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, in: Y. Bengio, Y. LeCun (Eds.), 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015, in: Conference Track Proceedings, 2015.

[72] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P.A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, TensorFlow: A system for large-scale machine learning, in: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2–4, 2016, 2016, pp. 265–283.

[73] H. Xiao, K. Rasul, R. Vollgraf, Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms, 2017, CoRR abs/1708.07747.

[74] M. Aldinucci, S. Rabellino, M. Pironti, F. Spiga, P. Viviani, M. Drocco, M. Guerzoni, G. Boella, M. Mellia, P. Margara, I. Drago, R. Marturano, G. Marchetto, E. Piccolo, S. Bagnasco, S. Lusso, S. Vallero, G. Attardi, A. Barchiesi, A. Colla, F. Galeazzi, HPC4AI, an AI-on-demand federated platform endeavour, in: ACM Computing Frontiers, Ischia, Italy, 2018, http://dx.doi.org/10.1145/3203217.3205340.

[75] P. Giannozzi, O. Baseggio, P. Bonfà, D. Brunato, R. Car, I. Carnimeo, C. Cavazzoni, S. de Gironcoli, P. Delugas, F. Ferrari Ruffino, A. Ferretti, N. Marzari, I. Timrov, A. Urru, S. Baroni, Quantum espresso toward the exascale, J. Chem. Phys. 152 (15) (2020) 154105, http://dx.doi.org/10.1063/5.0005082.

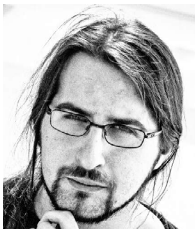
[76] R.F. da Silva, R. Filgueira, E. Deelman, E. Pairo-Castineira, I.M. Overton, M.P. Atkinson, Using simple pid-inspired controllers for online resilient resource management of distributed scientific workflows, Future Gener. Comput. Syst. 95 (2019) 615–628, http://dx.doi.org/10.1016/j.future.2019.01.015.

[77] M. Gallet, L. Marchal, F. Vivien, Efficient scheduling of task graph collections on heterogeneous resources, in: 2009 IEEE International Symposium on Parallel Distributed Processing, 2009, pp. 1–11, http://dx.doi.org/10.1109/IPDPS.2009.5161045.

[78] M. Aldinucci, V. Cesare, I. Colonnelli, A.R. Martinelli, G. Mittone, B. Cantalupo, C. Cavazzoni, M. Drocco, Practical parallelization of scientific applications with OpenMP, OpenACC and MPI, J. Parallel Distrib. Comput. 157 (2021) 13–29, http://dx.doi.org/10.1016/j.jpdc.2021.05.017.

[79] M. Caballero, J. Gomez, A. Bantouna, Deep-learning and hpc to boost biomedical applications for health (deephealth), in: 2019 IEEE 32nd International Symposium on Computer-Based Medical Systems, CBMS, IEEE Computer Society, Los Alamitos, CA, USA, 2019, pp. 150–155.

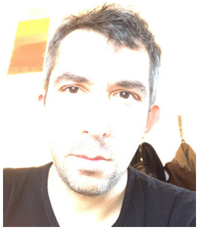
[80] M. Aldinucci, G. Agosta, A. Andreini, C.A. Ardagna, A. Bartolini, A. Cilardo, B. Cosenza, M. Danelutto, R. Esposito, W. Fornaciari, R. Giorgi, D. Lengani, R. Montella, M. Olivieri, S. Saponara, D. Simoni, M. Torquati, The italian research on HPC key technologies across EuroHPC, in: CF '21: Computing Frontiers Conference, Virtual Event, Italy, May 11–13, 2021, 2021, pp. 178–184, http://dx.doi.org/10.1145/3457388.3458508.



Iacopo Colonnelli is a Ph.D. student in Modeling and Data Science at Università degli Studi di Torino. He received his master's degree in Computer Engineering from Politecnico di Torino with a thesis on a high-performance parallel tracking algorithm for the ALICE experiment at CERN. His research focuses on both statistical and computational aspects of data analysis at large scale and on workflow modeling and management in heterogeneous distributed architectures.

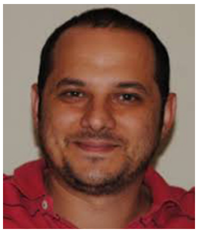
Marco Aldinucci is a full professor and the P.I. of the Parallel Computing research group at the University of Torino. He is the author of 120+ scientific articles and the recipient of the several research awards. He participated in over 15 EU-funded research projects on parallel and Cloud Computing (over 6M€). In 2018, he incepted the HPC4AI Turin's competency center on HPC-AI convergence. From 2021, he is the founding director of the CINI "HPC Key Technologies and Tools" national laboratory, gathering researchers from 35 Italian Universities. He is a member of the Governing Board of the EuroHPC Joint Undertaking.

Barbara Cantalupo is a Research Engineer at the Computer Science Department of the University of Torino. She received her master's degree in Computer Science from the University of Pisa (1994), and she has been a researcher in the parallel computing field at University and at the Italian National Research Council (CNR). Afterwards, she has worked for 15 years in several private companies in the area of supercomputing, mobile networks and space, acquiring knowledge on different application fields.



Luca Padovani received his Ph.D in Computer Science in 2003 from the University of Bologna. From 2005 to 2009 he was Assistant Professor at the University of Urbino and then he moved to the University of Torino. Since 2015 he is Associate Professor at the University of Torino, where he coordinates the research group on Formal Methods for Software Development. His research interests include concurrency theory, type systems for enforcing safety and liveness properties of distributed and concurrent programs and programming language semantics.

Sergio Rabellino, degree cum laude in Computer Science, is the head of the ICT research technical staff at the Department of Computer Science, University of Turin. He co-operates with the research groups in Security, Eidomatics, High Performance Computing, Artificial Intelligence and E-learning. He is a Moodle Developer and hardware/software architect of HPC, Cloud and e-learning platforms. Technical head and architect of the HPC4AI project, Moodle based projects Start@Unito, Orient@mente and many other IT services of the Turin University.



Dr. Concetto Spampinato is an Assistant Professor at the University of Catania. In 2014, he founded, and currently leads, the PeRCeiVe Lab at same university that hosts several Ph.D. students, RA assistants and assistant professors. Since April 2018 he is also affiliated as a "Courtesy Faculty Member" with the CRCV at the University of Central Florida (USA). His research interests lie mainly in learning-based computer vision and pattern recognition by publication in top-tier journals (IEEE TPAMI, IJCV, CVIU) and conferences (CVPR, ICCV, IROS). He has published over 160 papers on international journals or conferences (Google Scholar: 4373 citations, h-index 38).

Roberto Morelli is a researcher in the Artificial Intelligence team at Leonardo S.p.A., with on-going collaboration with INFN (Bologna) to pursue his Ph.D. in Data Science & Computation, with focus on AI unsupervised methods for beyond standard model searches. Other research interests regard Deep Learning and Computer Vision techniques applied to biomedical images to support Bologna's Physiology department. He graduated in Physics at the University of Bologna in 2017 with a thesis on electromagnetic finite-element analysis. He attended a Master in financial mathematics and worked in the Big Data and Analytics team at C.R.I.F. S.p.A.

Rosario Di Carlo is currently a research fellow in Leonardo Labs. Before joining Leonardo S.p.A. in January 2021, he was Research Fellow at the University of Modena and Reggio Emilia within the "AImageLab". He received his Bachelor's degree in Computer Engineering from the University of Palermo, and his Master's degree from the University of Modena and Reggio Emilia with a thesis on deep learning models for Visual Reasoning. During his activity at AimageLab, he has participated in several projects of Computer Vision and Machine Learning applied in the industrial environment, working with several companies including Tetra Pak and IMA.

Dr. Nicolò Magini received his Ph.D. in Physics in 2005 from the University of Florence. In 2007, he joined the CERN physics laboratory in Geneva to work on the worldwide computing grid infrastructure used by CERN experiments. At CERN, he worked on services for data management, workload management and monitoring. Recently, one of his main interests was the expansion of CERN computing to Cloud and HPC resources. In 2021, Dr. Magini joined Leonardo as Research Fellow in the HPC/Cloud Labs, where he is working on the management of the davinci-1 HPC, and on the deployment of a private Cloud infrastructure.

Carlo Cavazzoni is presently head of Cloud Computing in Leonardo S.p.A., and director of the Leonardo HPC Lab. Before joining Leonardo in May 2020, he spent more than 20 years in Cineca (Italian Supercomputing Centre), where he became head of HPC R&D, with responsibility for the evolution and exploitation of the National and European HPC infrastructure. He is a member of the EuroHPC Research and Innovation Advisory Board, steering board member of the ETP4HPC association, and Leonardo representative in GAIA-X. He is (co-)author of more than 100 peer review articles, including Science, Physical Review Letters, Nature Materials, and many others.