UNIVERSITY OF TORINO

Ph.D. in Modeling and Data Science

XXXIV cycle

Final dissertation

Workflow models for heterogeneous distributed systems

Supervisor: Marco Aldinucci

Candidate: Iacopo Colonnelli

ACADEMIC YEAR 2020/2021

Summary

The role of data in modern scientific workflows becomes more and more crucial. The unprecedented amount of data available in the digital era, combined with the recent advancements in Machine Learning and High-Performance Computing (HPC), let computers surpass human performances in a wide range of fields, such as Computer Vision, Natural Language Processing and Bioinformatics. However, a solid data management strategy becomes crucial for key aspects like performance optimisation, privacy preservation and security.

Most modern programming paradigms for Big Data analysis adhere to the principle of data locality: moving computation closer to the data to remove transfer-related overheads and risks. Still, there are scenarios in which it is worth, or even unavoidable, to transfer data between different steps of a complex workflow.

The contribution of this dissertation is twofold. First, it defines a novel methodology for distributed modular applications, allowing topology-aware scheduling and data management while separating business logic, data dependencies, parallel patterns and execution environments. In addition, it introduces computational notebooks as a high-level and user-friendly interface to this new kind of workflow, aiming to flatten the learning curve and improve the adoption of such methodology.

Each of these contributions is accompanied by a full-fledged, Open Source implementation, which has been used for evaluation purposes and allows the interested reader to experience the related methodology first-hand. The validity of the proposed approaches has been demonstrated on a total of five real scientific applications in the domains of Deep Learning, Bioinformatics and Molecular Dynamics Simulation, executing them on large-scale mixed cloud-High-Performance Computing (HPC) infrastructures.

Acknowledgements

A Ph.D. program is like a long and non-linear path that asymptotically approaches scientific, professional, and human maturity. Following such a path translates into a long journey, where pride and satisfaction for newly acquired notions entangle with discouragement and frustration for those still to be learned. Although one's thirst for knowledge can play a role in reaching the final destination of this journey, what mattered in my experience was travelling with the right crew. The current Section is dedicated to all its members.

First of all, this entire dissertation would have never seen the light without Prof. Marco Aldinucci. Marco has been my supervisor in so many directions that listing all of them would require a new dissertation. He taught me to think critically about distributed computing and to reason about theoretical concepts rather than blindly follow new trendy terms and software. He always encouraged me to adopt my own research approach, even when we had different views on a given topic. Furthermore, he always helped me when needed, especially for those problems that could not be solved with algorithms and coding.

Looking at the world through different eyes is the only way to approach the truth. A special thanks goes to the reviewers of this dissertation, Rosa M. Badia and Ian T. Foster, my industrial tutor Claudia Misale, and the dissertation chair Marco Danelutto for their precious suggestions and words of encouragement. I promise I will do my best to "keep up with the good work" as long as I can. Another heartfelt thanks goes to my colleagues: Yasir Arfat, Barbara Cantalupo, Bruno Casella, Alberto Riccardo Martinelli, Doriana Medić, Gianluca Mittone, and Alberto Mulone. You shared with me the joys and sorrows of research. Neither a pandemic kept us from working together, sharing ideas, drinking alcohol and having fun.

Neither discovery nor research can exist without inspiration. Mom, I have to thank you for always supporting me during my entire life, even when this meant putting my needs before yours. Thanks to you, I have the best family one could wish for. And you, Alice, have been my primary source of inspiration during these last years. I watched you fight against the enemies inside you, day after day, and nothing gave me more strength than seeing a smile on your face at the end of the battle. What I learned during these years is that if we fight together, we can win. But "if I cannot break your fall, I'll pick you up right off the ground. If you felt invisible, I won't let you feel that now".

Contents

Chapter 1

Introduction

When considering data-oriented workflows, all the aspects of data management become crucial for performance optimisation, privacy preservation and security. Modern programming paradigms for Big Data analysis, such as MapReduce [1] and Resilient Distributed Datasets [2], focus on the principle of *data locality*: moving computation closer to the data to remove data transfer overheads and risks. More recently, the successful adoption of *federated learning* approaches [3], [4] to data analytics shifted the support for distributed environments from a scalability-related feature to an unwaivable requirement, as data cannot be moved by contract.

Besides, there are scenarios in which it is worth, or even unavoidable, to transfer data between different modules of a complex application. For example, in HPC centres, the combination of centralised shared file systems and bursty I/O patterns constitutes scalability's main bottleneck. Recently, HPC facilities started to equip worker nodes with local, high-end *burst buffers* [5] where each process of a distributed application can read and write intermediate results or persist its state to implement checkpoint/restart. This strategy allows reducing pressure on global storage nodes and channels, improving performance and scalability at the cost of increased complexity. Modern deep neural networks [6], [7] provide another use case. Their training phase requires computing power amounts and interconnection speeds that only the biggest HPC centres in the world can offer. However, input datasets are commonly generated, analysed, and pre-processed on much more comfortable cloud resources, and trained models are transferred back to long-lived cloud services for inference purposes.

When performing data transfers, an educated application of the *security-by-design* principle requires encrypting any sensitive content with an adequately robust cypher suite every time it moves through an insecure channel or is sent to an untrusted party. However, effectively dealing with security-related aspects is not trivial, especially for users without a strong Computer Science background.

In addition, modern applications require *portability* of deployments, avoiding vendor and technology lock-in and fostering cross-stack executions to handle time-critical emergency computations. Indeed, efficiently implementing *urgent computing*

[8] in a single infrastructure is not trivial. Shared HPC facilities require the manual intervention of system administrators to arrange a high-priority queue, reserve some hardware resources to that queue, and notify other users that their jobs will likely be preempted or killed for the duration of the operation. On the other hand, cloud orchestrators provide *elasticity* mechanisms to dynamically change the number of agents involved in a distributed execution, allowing users to request additional resources to face load peaks. Nevertheless, the cloud virtualisation layer prevents high-performance libraries, e.g. Message Passing Interface (MPI) or linear algebra libraries, relying on hardware-specific optimisations and low latency network protocols commonly available on HPC bare-metal computing nodes.

Cross-stack (e.g. cloud-HPC) infrastructures can solve these problems, offloading computation to the best suitable architecture as long as computing power is available and using different stacks only as backup solutions. The main drawbacks of these configurations are their need for complex deployment strategies, cross-stack life-cycle management, and explicit data transfers between the different modules of a distributed application. Plus, guaranteeing *reproducibility* of executions requires a detailed description of the execution environment, the data transfer operations and the mapping between application tasks and processing locations in charge of running them.

Moving all those management features from the host application to the workflow coordination plane has undoubted advantages in portability, maintainability, and soundness. Indeed, it allows moving from a *tightly coupled* approach, where business logic is interleaved with platform-specific optimisations, to a *loosely coupled* one, where those concerns are clearly separated. In addition, a Workflow Management System (WMS) receiving a *topology-aware* workflow representation can automatically adopt scheduling optimisation techniques that take data locality into account, prevent unallowed data movements by executing steps in a confined environment, and adequately protect data during transfers, letting the data scientists focus on the business code.

Also, modern modular applications based on software containers [9] and their related architecture paradigms (e.g. the *microservices* pattern) can highly benefit from a topology-aware WMS. Indeed, if containers introduce unprecedented benefits in software portability and experiments reproducibility [10], successfully orchestrating multi-container applications and data transfers between their components is a complex task. In particular, the ephemeral nature of file systems makes it crucial to wisely manage the life-cycle of each container, ensuring data availability while avoiding resource wastes.

Still, programming paradigms and tools are useless if people are not willing to adopt them. Despite all the advantages of a loosely coupled approach, domain experts often prefer to use general-purpose languages to develop their applications, i.e. stick with a tightly coupled strategy. Likely reasons behind this behaviour are the additional effort required to learn a full-fledged WMS or coordination language, the increased difficulty in maintaining coherence between host and coordination logic in a still-developing application, and the lack of a de-facto standard toolchain for scientific workflows.

With their capability to unify imperative code and declarative metadata in a unique format, computational notebooks [11] are halfway between high-level coordination languages and low-level distributed computing libraries, the two alternative ways to develop distributed scientific applications. In addition, the widespread diffusion of Jupyter Notebooks [12] in almost all areas of computational science (particularly in the data science field) implies that many domain experts are already familiar with them, removing the need to learn an additional, workflow-specific tool.

The contribution of this dissertation is twofold. First, it defines a novel methodology to describe distributed modular applications, allowing topology-aware scheduling and data management while separating business logic, data dependencies, parallel patterns and execution environments. In addition, it introduces computational notebooks as a high-level and user-friendly interface to this new kind of workflow, aiming to flatten the learning curve and improve the adoption of such methodology. Each of these contributions is accompanied by a full-fledged, Open Source implementation, which has been used for evaluation purposes and allows the interested reader to experience the related methodology first-hand. These contributions are summarised in detail in Sec. 1.1.

This dissertation is the final result of a three-year Ph. D. programme, during which the author published 11 research articles in journals and conferences. Some are directly related to this thesis, while the others refer to different topics, mainly parallel computing, Deep Learning (DL) and medical statistics. Sec. 1.2 contains a list of all author's publications. Plus, the author took part to several research projects and activities, which are listed in Sec. 1.3. Finally, Sec. 1.4 lists the funding sources that made this work possible.

1.1 Results and contributions

The first two contributions of this work are a *hybrid workflow* abstraction capable of expressing topology-aware workflows and the implementation of a novel WMS, called *StreamFlow*, capable of orchestrating hybrid workflows across mixed cloud-HPC architectures. These aspects are introduced in Sec. 1.1.1 and expanded in Chapters 3 and 4. Then, the second part of the thesis introduces *literate workflows*, a novel paradigm to model (hybrid) workflows as computational notebooks, and *Jupyter-workflow*, which extends the Jupyter stack to support such paradigm. Sec. 1.1.2 summarises these last contributions, which are then discussed in detail in Chapters 5 and 6.

1.1.1 Hybrid workflows and the StreamFlow framework

A *hybrid workflow* model combines a standard workflow model, expressing dependencies among different steps of a modular application, and a *topology of deployment locations*, describing a set of heterogeneous execution environments and the communication channels between them. Each workflow step can be *mapped* onto one or more locations for execution, and the same location can be contended by multiple steps.

There are several advantages in including execution environments directly in the workflow model. A topology-aware WMS can statically evaluate the soundness of execution plans and implement scheduling policies based on data locality. A domain expert can select the best environment for executing each workflow step and easily shift from one configuration to another by modifying the step-location mapping. A researcher willing to reproduce an experiment has a clear vision of the execution environment used in the original work and can port the application onto a different architecture without touching the workflow logic.

The *StreamFlow* framework serves as runtime support for hybrid workflows [13]. It fully supports the Common Workflow Language (CWL) open standard [14] to model workflows and several well-known external formats to model topologies of deployment locations (e.g. Helm charts for Kubernetes deployments or Slurm scripts for HPC workloads). It has successfully orchestrated complex workflows in Bioinformatics and DL on mixed cloud-HPC environments, and further applications are currently under development.

1.1.2 Literate workflows and the Jupyter-workflow stack

The *literate workflows* paradigm [15] treats each cell of a computational notebook as a workflow step, using the related metadata to express dependencies, parallel patterns and, in the case of hybrid workflows, location topologies and the step-location mapping relation. Literate workflows interleave host and coordination logic in the same document but at the same time keep them well separated, promoting clarity and maintainability.

Execution semantics of standard computational notebooks are inherently sequential: cells are executed one at a time, either in the order of appearance (*bulk execution*) or on the user's command (*interactive execution*). The proposed methodology can extract a workflow Directed Acyclic Graph (DAG) by defining *sequentially equivalent* parallel semantics so that independent cells can be executed concurrently, obtaining the same output of a sequential case.

Jupyter-workflow extends the IPython software stack to support hybrid literate workflows, relying on StreamFlow for scheduling, orchestration and fault-tolerance. It has been successfully evaluated on large-scale HPC and cloud environments with four practical applications in the domains of DL, scientific simulation and Bioinformatics.

1.2 List of publications

This section lists all the author's publications in reverse chronological order. Research works are organised along two dimensions: Sec. 1.2.1 categorises them based on the venue, and then Sec. 1.2.2 groups them by the targeted topic. Among them, J1, J5, C1 and C3 are direct results of this dissertation, but all cover topics and use-cases that served as inspirations for the main contributions.

1.2.1 Publications organised by venue

Book chapters

B1 M. Aldinucci, D. Atienza, F. Bolelli, M. Caballero, **I. Colonnelli**, J. Flich, J. A. Gómez, D. González, C. Grana, M. Grangetto, S. Leo, P. López, D. Oniga, R. Paredes, L. Pireddu, E. Quiñones, T. Silva, E. Tartaglione, and M. Zapater, «The DeepHealth Toolkit: A Key European Free and Open-Source Software for Deep Learning and Computer Vision Ready to Exploit Heterogeneous HPC and Cloud Architectures», in *Technologies and Applications for Big Data Value*, E. Curry, S. Auer, A. J. Berre, A. Metzger, M. S. Perez, and S. Zillner, Eds., Cham: Springer International Publishing, 2022, ch. 9, pp. 183–202, ISBN: 978-3-030-78307-5. DOI: 10.1007/978-3-030-78307-5_9

B2 E. Quiñones, J. Perales, J. Ejarque, A. Badouh, S. Marco, F. Auzanneau, F. Galea, D. González, J. R. Hervás, T. Silva, **I. Colonnelli**, B. Cantalupo, M. Aldinucci, E. Tartaglione, R. Tornero, J. Flich, J. M. Martinez, D. Rodriguez, I. Catalán, J. Garcia, and C. Hernández, «The DeepHealth HPC Infrastructure: Leveraging Heterogenous HPC and Cloud Computing Infrastructures for IA-based Medical Solutions», in *HPC, Big Data, and AI Convergence Towards Exascale: Challenge and Vision*, O. Terzo and J. Martinovič, Eds., Boca Raton, Florida: CRC Press, 2022, ch. 10, pp. 191–216, ISBN: 978-1-0320-0984-1. DOI: 10.1201/9781003176664

Journal papers

J1 **I. Colonnelli**, M. Aldinucci, B. Cantalupo, L. Padovani, S. Rabellino, C. Spampinato, R. Morelli, R. Di Carlo, N. Magini, and C. Cavazzoni, «Distributed workflows with Jupyter», *Future Generation Computer Systems*, vol. 128, pp. 282–298, 2022, ISSN: 0167-739X. DOI: 10.1016/j.future.2021.10.007

J2 O. D. Filippo, J. Kang, F. Bruno, J.-K. Han, A. Saglietto, H.-M. Yang, G. Patti, K.-W. Park, R. Parma, H.-S. Kim, L. D. Luca, H.-C. Gwon, M. Iannaccone, W. J. Chun, G. Smolka, S.-H. Hur, E. Cerrato, S. H. Han, C. di Mario, Y. B. Song, J. Escaned, K. H. Choi, G. Helft, J.-H. Doh, A. T. Giachet, S.-J. Hong, S. Muscoli, C.-W. Nam, G. Gallone, D. Capodanno, D. Trabattoni, Y. Imori, V. Dusi, B. Cortese, A. Montefusco, F. Conrotto, **I. Colonnelli**, I.

Sheiban, G. M. de Ferrari, B.-K. Koo, and F. D'Ascenzo, «Benefit of Extended Dual Antiplatelet Therapy Duration in Acute Coronary Syndrome Patients Treated with Drug Eluting Stents for Coronary Bifurcation Lesions (from the BIFURCAT Registry)», *The American Journal of Cardiology*, 2021, issn: 0002-9149. doi: 10.1016/j.amjcard.2021.07.005

J3 M. Aldinucci, V. Cesare, **I. Colonnelli**, A. R. Martinelli, G. Mittone, B. Cantalupo, C. Cavazzoni, and M. Drocco, «Practical Parallelization of Scientific Applications with OpenMP, OpenACC and MPI», *Journal of Parallel and Distributed Computing*, vol. 157, pp. 13–29, 2021. doi: 10.1016/j.jpdc.2021.05.017

J4 F. D'Ascenzo, O. De Filippo, G. Gallone, G. Mittone, M. A. Deriu, M. Iannaccone, A. Ariza-Solé, C. Liebetrau, S. Manzano-Fernández, G. Quadri, T. Kinnaird, G. Campo, J. P. Simao Henriques, J. M. Hughes, A. Dominguez-Rodriguez, M. Aldinucci, U. Morbiducci, G. Patti, S. Raposeiras-Roubin, E. Abu-Assi, G. M. De Ferrari, F. Piroli, A. Saglietto, F. Conrotto, P. Omedé, A. Montefusco, M. Pennone, F. Bruno, P. P. Bocchino, G. Boccuzzi, E. Cerrato, F. Varbella, M. Sperti, S. B. Wilton, L. Velicki, I. Xanthopoulou, A. Cequier, A. Iniguez-Romo, I. Munoz Pousa, M. Cespon Fernandez, B. Caneiro Queija, R. Cobas-Paz, A. Lopez-Cuenca, A. Garay, P. F. Blanco, A. Rognoni, G. Biondi Zoccai, S. Biscaglia, I. Nunez-Gil, T. Fujii, A. Durante, X. Song, T. Kawaji, D. Alexopoulos, Z. Huczek, J. R. Gonzalez Juanatey, S.-P. Nie, M.-a. Kawashiri, **I. Colonnelli**, B. Cantalupo, R. Esposito, S. Leonardi, W. Grosso Marra, A. Chieffo, U. Michelucci, D. Piga, M. Malavolta, S. Gili, M. Mennuni, C. Montalto, L. Oltrona Visconti, and Y. Arfat, «Machine learning-based prediction of adverse events following an acute coronary syndrome (PRAISE): a modelling study of pooled datasets», *The Lancet*, vol. 397, no. 10270, pp. 199–207, 2021, issn: 0140-6736. doi: 10.1016/S0140-6736(20)32519-8

J5 **I. Colonnelli**, B. Cantalupo, I. Merelli, and M. Aldinucci, «StreamFlow: cross-breeding cloud with HPC», *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 4, pp. 1723–1737, 2021. doi: 10.1109/TETC.2020.3019202

Conference papers

C1 **I. Colonnelli**, B. Cantalupo, C. Spampinato, M. Pennisi, and M. Aldinucci, «Bringing AI pipelines onto cloud-HPC: setting a baseline for accuracy of COVID-19 diagnosis», in *ENEA CRESCO in the fight against COVID-19*, F. Iannone, Ed., ENEA, 2021. doi: 10.5281/zenodo.5151511

C2 G. Agosta, W. Fornaciari, A. Galimberti, G. Massari, F. Reghenzani, F. Terraneo, D. Zoni, C. Brandolese, M. Celino, F. Iannone, P. Palazzari, G. Zummo, M. Bernaschi, P. D'Ambra, S. Saponara, M. Danelutto, M. Torquati,

M. Aldinucci, Y. Arfat, B. Cantalupo, **I. Colonnelli**, R. Esposito, A. R. Martinelli, G. Mittone, O. Beaumont, B. Bramas, L. Eyraud-Dubois, B. Goglin, A. Guermouche, R. Namyst, S. Thibault, A. Filgueras, M. Vidal, C. Alvarez, X. Martorell, A. Oleksiak, M. Kulczewski, A. Lonardo, P. Vicini, F. L. Cicero, F. Simula, A. Biagioni, P. Cretaro, O. Frezza, P. S. Paolucci, M. Turisini, F. Giacomini, T. Boccali, S. Montangero, and R. Ammendola, «TEXTAROSSA: Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale», in *Proc. of the 24th Euromicro Conference on Digital System Design (DSD)*, Palermo, Italy: IEEE, Aug. 2021. DOI: 10.1109/DSD53832.2021.00051

C3 **I. Colonnelli**, B. Cantalupo, R. Esposito, M. Pennisi, C. Spampinato, and 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, January 19, Budapest, Hungary*, ser. OASIcs, vol. 88, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 5:1–5:13. DOI: 10.4230/OASIcs.PARMA-DITAM.2021.5

C4 V. Cesare, **I. Colonnelli**, and M. Aldinucci, «Practical Parallelization of Scientific Applications», in *28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2020, Västerås, Sweden, March 11-13*, IEEE, 2020, pp. 376–384. DOI: 10.1109/PDP50117.2020.00064

C5 M. Drocco, P. Viviani, **I. Colonnelli**, M. Aldinucci, and M. Grangetto, «Accelerating Spectral Graph Analysis Through Wavefronts of Linear Algebra Operations», in *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019, Pavia, Italy, February 13-15*, IEEE, 2019, pp. 9–16. DOI: 10.1109/EMPDP.2019.8671640

C6 P. Viviani, M. Drocco, D. Baccega, **I. Colonnelli**, and M. Aldinucci, «Deep Learning at Scale», in *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019, Pavia, Italy, February 13-15*, IEEE, 2019, pp. 124–131. DOI: 10.1109/EMPDP.2019.8671552

1.2.2 Publications organised by topic

Workflow modelling and management

This topic is the main subject of the dissertation, and related works are discussed extensively in the following Chapters. In particular, J5 introduces the StreamFlow framework, a WMS that serves as runtime support for hybrid workflows. Chapter 4 is entirely dedicated to that topic. C1 and C3 describe the CLAIRE COVID-19 universal pipeline and its implementation with StreamFlow, extensively discussed in Sec. 4.2.2. J1 contains descriptions of hybrid literate workflows and the Jupyter-workflow WMS, introduced in Chapters 5 and 6, respectively. B2 and B1 introduce the software stack developed in the context of the DeepHealth European project, where StreamFlow is used to orchestrate DL training and inference pipelines on top of HPC4AI, a multi-tenant cloud-HPC system at Università di Torino. Finally, C2 describes the TEXTAROSSA European project, one of the projects containing StreamFlow in their software stack.

Parallel computing

The shift toward parallel computing platforms has many drivers likely to sustain this trend for several years to come. Consequently, parallel programming methodologies evolve in two directions: an enhancement in applications performances and a rise in the level of abstraction of concurrency management primitives.

Algorithmic skeletons [16], [17], i.e. standard parallel programming patterns for which one or more pre-defined and optimised implementations exist, play a crucial role in this process. When multiple implementations of the same pattern are available, choosing the best option is a non-trivial task. C5 empirically compares multiple implementations of a Generalised Fourier Transform kernel on heterogeneous machines equipped with one or more Graphics Processing Unit (GPU) accelerators, showing how the best performing library varies depending on the matrix size.

These paradigms work well because they are *explicitly parallel*: the programmers can directly design their applications within a specific programming model and verify the embedded sequential code's compliance with programming model constraints. The crucial aspect for optimal performances becomes the choice of the correct pattern to represent data/control communication patterns between subsequent parallel sections of an application.

As a rule of thumb, reducing the amount of communication and avoiding global synchronisations improves scalability and reduces overhead at the price of higher complexity. C6 proposes a novel approach to Deep Neural Network (DNN) training that substitutes the all-reduce operation of distributed Stochastic Gradient Descent with an asynchronous nearest-neighbour pattern, improving scalability while retaining good convergence properties.

Difficulty increases when dealing with legacy applications designed with a purely sequential programming mindset, possibly using global variables, aliasing, random number generators, and stateful functions. Re-designing from scratch with an explicitly parallel approach is still the most effective option to achieve scalable and efficient parallel codes. Still, this approach cannot effectively support the industrial adoption of parallel computing key technologies, where human productivity and time-to-solution are equally, if not more, essential aspects than performance.

C4 and J3 propose a semi-automatic methodology to modernise an existing sequential scientific code with a little re-designing effort, making it a parallel and robust code. Such methodology is then successfully tested on four real-world scientific codes using the shared memory model (via OpenMP), message-passing model (via MPI), and General Purpose Computing on GPU model (via OpenACC).

Statistics and Machine Learning in medicine

The combination of statistics and medicine has been ubiquitous in the scientific literature of the last century. The statistical analysis of clinical trials is fundamental to validating new treatments and drugs, but extracting knowledge from these datasets is complex, as they are often noisy, incomplete and unbalanced.

Techniques from classical statistics, such as propensity scores [18], [19] or Cox proportional hazard models [20], are still widely used in contemporary research. Indeed, J2 applies these techniques to a large dataset of patients treated with percutaneous coronary intervention for coronary bifurcations to evaluate the optimal duration of dual antiplatelet therapy.

In recent years, with the rise of Artificial Intelligence (AI) and Machine Learning (ML), medical research started to successfully apply more modern and sophisticated techniques to extract knowledge from clinical trials [21]. In this setting, J4 derives a risk stratification model based on AdaBoost [22] to predict all-cause death, recurrent acute myocardial infarction and major bleeding after an acute coronary syndrome.

The explosion of DL models to solve image recognition and object detection problems in the last decade gave a massive boost to AI-assisted diagnosis techniques based on images, typically X-Ray and Computed Tomography (CT) scans. These strategies rely on deep Convolutional Neural Networks (CNNs) to classify patients according to the presence or absence of disease-related features in medical images. However, the use of different datasets, pre-processing pipelines and DNNs makes it impossible to fairly compare the performance of different approaches. C1 and C3 describe the CLAIRE COVID-19 universal pipeline, a reproducible workflow capable of automating the comparison of state-of-the-art DL models to diagnose COVID-19. The interested reader can find an extensive discussion of this pipeline in Sec. 4.2.2.

1.3 Research projects and activities

This section lists the research projects and activities in which the author took part during his Ph. D. program. In particular, Sec. 1.3.1 lists the international research projects funded by the European Commission, Sec. 1.3.2 enumerates the national Italian projects, and Sec. 1.3.3 discusses other research activities.

1.3.1 European projects

DeepHealth (EC H2020 IA, ICT-2018-11, 2019, 36 months, 14.8M€, G.A. 825111). The “Deep-Learning and HPC to Boost Biomedical Applications for Health” project aims to offer a unified framework assembled with state-of-the-art techniques in DL and Computer Vision, completely adapted to exploit underlying heterogeneous HPC and Big Data architectures.

ACROSS (EC H2020 IA, EuroHPC-01-2019, 2021, 36 months, 8M€, G.A. 955648). The "HPC Big Data Artificial Intelligence cross-stack platform towards exascale" project will co-design and develop an HPC, Big Data, and AI convergent platform, supporting complex workflows in the aeronautics, weather and energy domains and their execution on top of the next generation of pre-exascale infrastructures, still being ready for exascale systems, and on.

TEXTAROSSA (EC H2020 RIA, EuroHPC-01-2019, 2021, 36 months, 6M€, G.A. 956831). The "Towards extreme scale technologies and accelerators for Euro-HPC HW/SW supercomputing applications for exascale" project aims at applying a co-design approach to heterogeneous HPC solutions, supported by the integration and extension of intellectual properties, programming models and tools derived from European research projects.

EUPEX (EC H2020 RIA, EuroHPC-02-2020, 2021, 48 months, 41M€, G.A. 101033975): The "European Pilot for Exascale" project will pave the way for a self-reliant European HPC industry, capable of delivering exascale-class supercomputers designed in Europe.

1.3.2 National projects

HPC4AI (Regione Piemonte, POR FESR Regione Piemonte, 2018, 24 months, 4.5M€). The "High-PerformanceComputing for Artificial Intelligence" project aims at creating a federated competence centre on HPC, AI and Big Data analytics to co-design with industries and SMEs research and technology transfer projects.

QWaaS (CINECA ISCRA-C, 2021, 9 months). The "Quantum Workflows as a Service" project aims at investigating how to build hybrid computational workflows involving Quantum Computing and HPC in a picture where Quantum Computers acts as accelerators of HPC traditional platforms, but the two souls can remain as loosely coupled as possible.

1.3.3 Other research activities

HPC-Europa3 scholarship Participation at the HPC-Europa3 transnational access programme, under the supervision of Dr. Eduardo Quiñones Moreno, on the topic of I/O performance optimisation in large-scale DL workloads.

IBM Mentorship Participation at the IBM T.J. Watson mentorship programme, under the supervision of Dr. Claudia Misale, on the topic of large-scale distributed workflows over mixed cloud-HPC infrastructures.

CWL Technical Team Member of the Common Workflow Language Technical Team[1], with the aim of proposing, implementing and approving new features of the CWL Standard before the CWL Leadership Team final vote.

Workflow Benchmarking Group Co-chair of the Workflow Benchmarking Group (WfBG)[2], an international working group aiming at developing a community agreement on benchmarking suites and performance metrics for real-world workflows, and collaboratively maintaining a catalogue of state-of-art implementations of these suites for various workflow languages and frameworks.

1.4 Funding

This work has been partially supported by the DeepHealth project[3], *"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 [23], the ACROSS project[4], *"HPC Big Data Artificial Intelligence open Stack Platform Towards Exascale"* which has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No. 955648 [24], and the HPC4AI project[5], which has been funded by the Region Piedmont POR-FESR 2014-20 (INFRA-P) [25].

[1]https://www.commonwl.org/governance/

[2]https://workflows.community/groups/benchmarking/

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

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

[5]https://hpc4ai.it/

Chapter 2

Background

This chapter is devoted to a general introduction of the key concepts addressed in the thesis, together with a state-of-the-art discussion of research advancements and open problems in the covered scientific area. Sec. 2.1 tries to unify business and scientific workflows domains under a generic enough definition of workflows as directed bipartite graphs. Sec. 2.2 gives an overview of the main WMSs features for workflow modelling and runtime phases. Sec. 2.3 introduces distributed computing and describes the principal execution infrastructures for large-scale distributed applications. Sec. 2.4 presents the container-based virtualisation approach and its role in the modern workflows ecosystem. Finally, Sec. 2.5 introduces the literate computing paradigm and lists previous attempts to use notebooks as workflow modelling tools and high-level interfaces to HPC facilities.

2.1 Workflows

Workflow models, thanks to their generality, represent a powerful abstraction for designing complex applications and executing them on large-scale distributed architectures, such as HPC centres, Grid environments, and the cloud. A drawback of such generality is that no consistent and commonly agreed definition of workflow seems to exist in computer science literature. For instance, the Workflow Management Coalition [26] identifies a workflow as the (partial) automation of a business process, during which *data* or *tasks* are passed from one participant to another according to a set of *procedural rules*. Conversely, the Encyclopedia of Database Systems [27] defines a (scientific) workflow as the description of a process for accomplishing a scientific objective, usually expressed as *tasks* organised and orchestrated according to their data (and possibly other) *dependencies*. These definitions indirectly assume a specific paradigm to model and execute tasks: a control-driven one in the former and a dataflow-like one in the latter. Such paradigms are the most common ways to describe the business and scientific workflows, respectively, but an acceptable definition should subsume both scenarios.

In order to be as generic as possible, this work defines a workflow as a *directed* *bipartite graph*. The nodes of this graph can refer to either the computational *steps* of a modular application or the *ports* through which they communicate. Its edges encode *dependency relations* between steps as links from steps to (output) ports and from (input) ports to subsequent steps. More formally:

Definition 2.1.1. A workflow is a directed bipartite graph $W = (S, P, D)$, where S is the set of steps, P is the set of ports, and $D \subseteq (S \times P) \cup (P \times S)$ is the set of dependency links. □

Let $\mathit{In}(s), \mathit{Out}(s) \subseteq P$ be the sets of input and output ports of a step s. Their definitions can be formally written as follows:

$$\begin{aligned} \mathit{In}(s) &= \{p \in P : (p, s) \in D\} \\ \mathit{Out}(s) &= \{p \in P : (s, p) \in D\} \end{aligned} \tag{2.1}$$

In the simplest case, the behaviour of s can be described with a function f_s, taking arguments in the $\mathit{In}(s)$ domain and returning values in the domain of $\mathit{Out}(s)$. In this generic setting, a path connecting step s to step t introduces a partial execution order $s \prec t$, but no further assumption is made on the nature of dependencies. On the other hand, constraining the domain and codomain of f_s, its evaluation semantics, or its triggering strategy is equivalent to impose a specific paradigm to express the workflow [28]. Notice that the functional nature required for the workflow steps does not exclude stateful computations, as they can still be modelled using a *feedback loop*, i.e. a port $p \in \mathit{In}(s) \cap \mathit{Out}(s)$ carrying the current state of step s.

Directed bipartite graphs have already been proposed in the literature to model workflows. Petri Nets [29], widely used to express control flows, are bipartite graphs by definition, and also dataflow graphs can be seen as bipartite graphs [30]. Both models come with *token-pushing* semantics [31]: steps are enabled by the presence of *tokens* in their input ports. The main difference between the two families of models relies on the nature of such tokens, particularly their ability to carry data. Nevertheless, differences are more subtle than they look. Some extensions to the Petri Nets paradigm (e.g. Coloured Petri Nets [32]) support (typed) data tokens, which can be used to construct dataflow graphs. On the other hand, dataflow graphs can easily model the “pure communication” tokens of standard Petri Nets as empty data tokens. The bipartite graph model proposed in this work is somehow pluripotent, as it can be reconciled into each specific case by supplying additional constraints and details.

A formal definition of the workflow model allows to univocally determine its *expressive power*, i.e. the set of applications that can be described as workflows and the limitations introduced by additional constraints. For example, a directed graph is flexible enough to express *sequential*, *concurrent*, and *iterative* workflow patterns [33]. Conversely, the DAG abstraction adopted by many workflow systems drops support for iterative patterns (and therefore stateful computations), but it can prevent deadlocks whenever each step terminates in a finite amount of time. Furthermore, the model can be extended to support more complicated scenarios. For example, *conditional* workflow patterns can be supported by describing each step s with a tuple $(f_s^{(1)}, \dots, f_s^{(n)}, c_s)$ where c_s is a condition evaluated on $In(s)$ to determine which function $f_s^{(k)}$ should be executed to produce values on $Out(s)$.

2.2 Workflow management systems

When expressing an application in terms of a workflow model, it is necessary to distinguish between two different classes of semantics [34]:

- The *host semantics*, which define the subprogram in each workflow step (i.e. the body of f_s), usually expressed in a general-purpose programming language (e.g. Java, C++, or Python) or as a shell script;
- The *coordination semantics*, which define the interactions between steps through a declarative markup syntax, an imperative Domain-Specific Language (DSL), or a graph-based modelling interface.

Tools in charge of exposing coordination semantics to the users and orchestrating workflow executions are known as Workflow Management Systems (WMSs). The WMS landscape is quite variegated: it embraces both high-level tools, mainly focused on resolving typical domain-specific modelling issues, and low-level specifications, aimed at executing tasks at scale on multi-process infrastructures.

Several surveys exist on WMSs, comparing their different functionalities [35]–[37], focusing on their evolution [38], or providing classification based on their support for extreme-scale applications [39]. The current section contains a general overview of the WMSs features supporting the two main phases of a workflow life-cycle: the *modelling phase*, during which a domain expert describes the different functional components of an application, and the *runtime phase*, during which the WMS deploys and manages the computational units required for the workflow execution.

Standard WMSs are user-driven systems specifically developed to satisfy domain requirements. They provide domain experts with a paradigm to describe, manage and share complex business processes or scientific analyses to ensure reproducibility and scalability. Typically, workflows can either be described programmatically or modelled using high-level declarative DSLs or advanced Graphical User Interfaces (GUIs), more suitable for users with little programming experience.

Many scientific WMSs, such as Kepler[1] [40], Askalon[2] [41], Pegasus[3] [42], Taverna[4]

[1]https://kepler-project.org/

[2]http://www.askalon.org/

[3]https://pegasus.isi.edu/

[4]https://taverna.incubator.apache.org/

[43], Triana [44], and Galaxy[5] [45], emerged with the diffusion of web services and Grid technologies, which allow users to access robust services and infrastructures more naturally than before [46]. Therefore, they were mainly targeted towards these architectures and not focused on portability. Over time, by evolving in strict contact with the scientific community, they acquired maturity from the functional design point of view and started providing some additional features, e.g. workflow distribution formats and repositories, and supporting newer architectures, like cloud computing and software containers.

Being Grid native, most of these tools support distributed workflows out of the box, providing automatic scheduling and data transfers management in the absence of a unique storage space shared among all the worker nodes. However, each WMS adopts its own technological stack for the communication layer, often relying on low-level external libraries that must be properly installed and pre-configured on each node involved in the workflow execution. For instance, Triana, Askalon and Pegasus depend on Grid-oriented libraries and tools (the GAP interface [47], the GLARE library [48], and HTCondor [49], respectively) to perform tasks offloading and data transfers, limiting the spectrum of supported execution environments. Conversely, Kepler implements an actor model through the Ptolemy library [50], modelling communication channels between pairs of interacting actors as abstract FIFO queues called `receivers`. In principle this approach is more platform-independent, as each actor pair can specify a different receiver implementation to exchange data. However, the Ptolemy library is more focused on single-node parallelism, and the only available distributed receiver relies on the Java Remote Method Invocation (RMI) protocol [51].

Other approaches privilege portability by providing a set of pluggable *executors* targeting a diverse set of infrastructures, such as public cloud services, batch schedulers (e.g. HTCondor, PBS, Slurm) and Kubernetes clusters. In most cases, such executors are agentless and do not require any specific software installed on the worker nodes. Nevertheless, different steps of the same workflow cannot be managed by different executors, and the control plane (i.e. the WMS process itself) must communicate with all the workers directly. Within these products, workflows are usually described through a product-specific DSL. For example, Apache Airflow[6] and Snakemake[7] [52] workflows are essentially Python scripts extended by declarative code that can be executed on distributed infrastructures. Other systems adopt Unix-style approaches for defining workflows: in Makeflow[8] [53] the end-user expresses a workflow in a technology-neutral way using a syntax similar to Make, while the

[5]https://galaxyproject.org/learn/advanced-workflow/

[6]https://airflow.apache.org/

[7]https://snakemake.readthedocs.io/en/stable/

[8]http://ccl.cse.nd.edu/software/makeflow/

Nextflow[9] [10] framework builds workflows using the Unix pipe concept. Together with Taverna, Nextflow is one of the few products to support cycles in workflows, while the other WMSs constrain workflow models to be DAGs. Finally, other WMSs adopt a lower-level approach by exposing an Application Programmable Interface (API) in a general-purpose programming language. For example, Toil [54] and DagOnStar [55] allow users to model workflows as pure Python scripts. Also Pegasus [42] exposes Python, Java, and R APIs to express workflows, but then such high-level descriptions are translated in DAX, the Pegasus low-level XML-based representation of workflow DAGs.

Since product-specific DSLs tightly couple workflows to a single software, actually limiting portability and reusability, there are also efforts in defining workflow specification languages or standards. For example, the CWL[10] [14] is an open standard for describing workflow DAGs following a JSON or YAML syntax or a mixture of the two. One of the first and most used CWL implementations is CWL-Airflow [56], which adds support for CWL to Apache Airflow, but also other products (e.g. Galaxy, Snakemake, Toil, and Nextflow) offer some compatibility with CWL. Other examples of workflow modelling open standards are the Workflow Description Language (WDL)[11], which is similar to CWL in terms of expressive power, and the Serverless Workflow Specification[12], which aims to describe event-based interactions among serverless applications. The latter is much more flexible, supporting iterations, data filters, and service invocations, but it is more oriented to a control-driven paradigm, with limited support for dataflow patterns.

An alternative approach to complex and feature-rich WMSs privileges performance over accessibility, exposing lower-level programming models directly to the users and coordinating the execution of many fine-grained tasks on distributed architectures. Big Data frameworks like Spark [57], Flink [58], and Storm [59], and parallel programming libraries like HyperLoom [60], Dask [61], COMP Superscalar (COMPSs) [62], Ray [63], and Parsl [64], belong to this category.

These libraries allow users to parallelise existing sequential applications by identifying functions that can be executed as asynchronous remote tasks. Concurrency can be expressed either imperatively, by explicitly calling the library API, or declaratively, through annotations. Asynchronicity is typically implemented with the *futures* paradigm [65]: an asynchronous function immediately returns a future object that can be passed as argument to another asynchronous function. The workflow execution plan, typically a layered dataflow model [66], is automatically built just-in-time by the runtime layer of the framework. Each asynchronous function invocation

[9]https://www.nextflow.io/

[10]https://www.commonwl.org/

[11]https://openwdl.org/

[12]https://serverlessworkflow.io/

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.

These tools are often used for programming HPC workflows, being a reasonable middle ground between high-level WMSs and explicit message passing libraries in terms of complexity and performance. Since all worker nodes share a common file system in HPC facilities, most of these tools do not support automatic data transfers (COMPSs is an exception). The main drawback is that host and coordination logics are interleaved in the same program, which must be entirely written in one of the supported languages.

Several tools adopt a similar idea for the automatic collection of provenance data from scripts. For instance, yesWorkflow [67] 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 [68] allows users to initialise 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 [69] 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.

Other tools, such as noWorkflow [70], 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. [71] explore the AST of Ruby scripts targeting GridSpace execution environment [72] to extract workflow models. The CXXR project [73] extends the R interpreter to automatically retrieve provenance data, while the LLVM-SPADE stack [74] 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.

2.3 Distributed systems

The concept of distributed computing is so broad that a proper definition cannot but depend on the context in which it is inserted. In this work, a distributed system is regarded as a *collection of independent computing elements* [75], where the word independent means that each computing element has its own clock and its own memory address space. Plus, a distributed system can be *heterogeneous*, meaning that different computing elements can have different hardware architectures, e.g. in terms of endianness, and equip different kinds of *accelerators*, e.g. GPUs, neuromorphic devices, or manycore processors.

The absence of a shared address space implies that the computing elements can only communicate through the network, exchanging messages. Such communications can be explicit, as in message-passing libraries like MPI [76] and message queue implementations like Apache Kafka[13] [77], or hidden behind higher-level abstractions, such as Distributed File-Systems (DFSs) or Partitioned Global Address Spaces (PGASs). However, programming a distributed application requires a lot of effort and compromises, as it is theoretically impossible to have an available, consistent, and partition-tolerant application [78].

The two main reasons for adopting a distributed software stack are *high availability* and *scalability*. The former relies on redundant components to ensure service availability in case of failure. The latter offloads different portions of a workload to multiple, potentially interacting computing nodes to improve performances (*strong scalability*) or deal with larger problems (*weak scalability*). Recently, the diffusion of federated learning techniques [3], [4] for data analysis added *data federation* to the motivations behind distributed systems.

Workflow abstractions are inherently modular and require an explicit encoding of all the necessary communications between subsequent steps. For these reasons, workflows lend themselves well to distributed executions. Indeed, many of the most popular WMSs were specifically created to address the then-emerging Grid platforms (see Sec. 2.2). Later on, with the advent of cloud computing and HPC facilities, WMSs started to include these technologies in their range of supported execution environments. The rest of the current section is devoted to exploring the peculiarities of these architectures in more detail. In particular, Sec. 2.3.1 introduces Grid environments and their logical stack, Sec. 2.3.2 describes high-performance computing centres, and Sec. 2.3.3 deals with cloud platforms.

2.3.1 Grid computing

Grid computing aims to define and provide flexible, secure and coordinated resource sharing in dynamic collections of individuals or institutions, called Virtual Organisations (VOs) [79]. In contrast to the other computing technologies discussed below, a Grid is *fully decentralised* [80]: different members of a VO can live within different control domains, and they interact with each other without the mediation of a centralised service provider acting as a trusted third party.

Interoperability is the most crucial aspect of a solid Grid architecture, aiming to support relationships among any potential participants. The lack of a centralised infrastructure makes it necessary to rely on standardised open *protocols* for negotiating and sharing resources, authenticating and authorising users, and managing the life-cycle of VOs. Still, interoperability among different Grids requires them to implement the same protocols. A key role in inter-Grid compatibility is played by

[13]https://kafka.apache.org/

two factors: an *hourglass model* for the Grid architecture and a de-facto standard implementation of the Grid technological stack.

The stack of traditional Grid architectures contains five layers [79]. The lowest one is the *Fabric* layer. It contains physical and logical resources that must be shared between members of a VO, like computing power, storage and network resources, and some mechanisms for discovering their state and capabilities and managing quality of service. Right above, the *Connectivity* layer defines communication and authentication protocols for secure network transactions, supporting, for example, routing, naming, single sign-on and delegation. The *Resource* layer builds on the Connectivity layer to define protocols for more advanced features involving single resources, like life-cycle management, monitoring and accounting. Connectivity and Resource layers constitute the neck of the hourglass: they contain a few widely-used protocols that can easily be implemented on top of a diverse set of resources (in the Fabric layer) and form the basis for constructing advanced services in the upper portion of the stack. The *Collective* layer contains protocols that involve either a global state of Grid or a collection of resources, like directory services for resource discovery, co-allocation and distributed task scheduling. Grid-based WMSs introduced in Sec. 2.2 also belong to this layer. Finally, the *Applications* layer contains the user-level applications running in the context of a VO, which are expressed in terms of services and protocols defined at any lower layer.

The de-facto standard implementation of the Grid architecture is the Globus Toolkit [81], a set of Open Source protocols, services and libraries specifically developed to support Grid applications. Among the Toolkit components, it is worth mentioning the Grid Resource Allocation Management (GRAM) protocol [82] for secure allocation and monitoring of computational resources, the Metacomputing Directory Service (MDS) [83] for resource discovery, the Grid Security Infrastructure (GSI) protocols [84] for authentication, single sign-on and delegation, and the GridFTP protocol [85] for high-speed data transfers. The existence of a de-facto standard implementation immensely facilitates compatibility among different Grid implementations. In contrast, the lack of a standard interface is the main obstacle to interoperability between today's clouds [86].

Another fundamental difference between Grid and cloud architectures is the compute model: cloud resources are virtualised and provisioned to users on-demand, while Grids rely mainly on queue-based schedulers that provision bare-metal resources through the GRAM interface [86]. Nevertheless, Grid protocols are, in principle, extremely flexible. GRAM supports on-demand resource provisioning through advanced reservations, and there exist some efforts to provide virtualised Grid resources in the literature [87].

Even if in the last years the cloud paradigm has become the widely preferred solution to host and provide services in the enterprise community, several Grid environments still survive in the research field. One of the most famous examples is the Worldwide LHC Computing Grid (WLCG) [88], which aims at storing, distributing and analysing the data measured by the Large Hadron Collider (LHC) detectors ad CERN.

2.3.2 High-performance computing

In the traditional meaning, the terms High-Performance Computing (HPC) refer to clusters of tightly-coupled, massively parallel machines aiming to provide significantly greater sustained performance than mainstream computer systems [89]. Such clusters are often called *supercomputers*, and the terms *cluster computing* are often used as a synonym of HPC.

A typical HPC stack includes vast amounts of cores and memory per computing node, massively parallel hardware accelerators, low-latency networking technologies and storage systems, and highly-optimised software libraries. This entire stack's main (and often sole) aim is to push performances to the maximum achievable amount. At the time of writing, the most powerful supercomputers in the world reach hundreds of Pflop/s on the HPLinpack benchmark [90], and enormous research efforts focus on reaching exascale performances (i.e. 10^{18} flop/s) [91]–[93].

Providing such levels of performance unavoidably brings extreme complexity in both the usage and maintenance of these systems. Any virtualisation layer is forbidden, as the virtualised overlay introduces a significant overhead and hinders compilers from applying hardware-specific optimisations. Consequently, all the users of an HPC cluster directly access the same bare metal nodes, forcing system administrators to put massive efforts into configuring secure authentication and authorisation toolchains and properly segmenting data accesses.

If the presence of high-end hardware and specialised software allows applications to achieve extreme performances, pushing them to the limit requires a deep knowledge of the entire HPC stack. For instance, linear algebra libraries such as BLAS [94] and LAPACK [95] play a crucial role in scientific applications like Computational Fluid Dynamics (CFD) and DL. However, proper tuning requires a tight coupling with the underlying hardware, which is so complex that often the best way to achieve maximum performance is to switch between implementations, according to the matrix size, the type of operation, and the presence or absence of hardware accelerators [96].

Explicit message passing is the most common communication paradigm in HPC, as it provides programmers with the highest amount of flexibility, and the MPI [76] is the de-facto standard in this field. According to Flynn's taxonomy [97], MPI applications are classified as Single Program Multiple Data (SPMD). Indeed, the same code is executed on multiple, potentially distributed computing elements by a single `mpirun` command, which returns when the global execution terminates (either successfully or after a failure). Properly compiled implementations of MPI can take advantage of both the Remote Direct Memory Access (RDMA) capabilities of active network boards and the userspace hardware access brought by their related software stacks. At the same time, low-latency technologies and proper in-network aggregation mechanisms guarantee high performances also in the presence of small messages. Again, pushing performances to the limit requires complex compilation toolchains and deep knowledge of the HPC stack.

The MPI communication layer is usually combined with other libraries and tools, realising the so-called MPI/X programming model. For instance, MPI/OpenMP can improve intra-node parallelism management [98], while MPI/CUDA can offload multi-node computations to NVidia GPUs [99], [100]. Multiple versions of fine-tuned scientific libraries, frameworks, and compilers are usually made available to HPC users through environment modules[14], even if recently system administrators are exploring alternative mechanisms such as ad-hoc package managers (e.g. Spack [101]) or software containers (e.g. Singularity [102]). In any case, since HPC users do not have privileged rights, installing and configuring additional software often requires a direct intervention of system administrators.

Typical HPC architectures contain some publicly exposed *frontend nodes*, where users land after logging in, and several *worker nodes*, which are in charge of executing computations and cannot usually access the Internet. Frontend and worker nodes share some portions of the file system, where one or more shared DFS are mounted. Such DFSs are parallel network file systems designed with performance in mind, such as Lustre[15] [103], GPFS[16], or BeeGFS[17]. Plus, worker nodes can host local, high-end *burst buffers* [5], usually based on the Non-Volatile Memory Express (NVMe) protocol, where a program can temporarily persist its input and output data to reduce pressure on the global storage nodes and channels, especially during checkpointing phases.

Traditionally, all worker nodes on an HPC cluster were homogeneous in terms of hardware architectures and software stacks. With the approach of exascale, HPC facilities are starting to become more modular. Indeed, the number and variety of HPC applications significantly increased in the last decades, and different applications come with different requirements. For instance, Bioinformatics software usually needs vast amounts of memory, large-scale CFD simulations need sustained high performances on double-precision operations, and DL requires efficient I/O and powerful GPUs. Consequently, modern HPC infrastructures offer multiple modules, i.e. multiple sets of racks optimised for different kinds of workloads. Developing proper technologies to orchestrate complex workflows across multiple HPC modules is still an open research problem, but the hybrid workflow models proposed in this thesis (see Chapter 3) are a step in this direction.

HPC users can access frontend nodes through secure remote shells, commonly

[14]https://modules.readthedocs.io/

[15]https://www.lustre.org/

[16]https://www.ibm.com/docs/en/gpfs/

[17]https://www.beegfs.io/

based on the Secure SHell (SSH) protocol, but they cannot directly interact with worker nodes. Instead, they can assign *batch jobs* to one or more queues, specifying the resources required for execution. A queue-based manager will then schedule jobs executions, according to a First In First Out (FIFO) order or a priority-based policy, as soon as enough computing nodes are available. Slurm[18] [104], PBS[19], and LSF[20] are among the most widespread queue managers in modern HPC facilities. The absence of high-level GUIs, the batched scheduling of jobs, and the somehow exotic ways to manage software dependencies make HPC usage highly complex for domain experts without a strong computer science background. Indeed, finding an effective way to improve accessibility to HPC facilities is still an open problem in computer science. In recent years, the combination of interactive job queues and literate computing is spreading as an alternative way to offer HPC services, but the related solutions are still at an early stage (see Sec. 2.5.3).

2.3.3 Cloud computing

According to the classical NIST definition [105], cloud computing is "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources". In a sense, it is the antithesis of HPC, where resources are tightly coupled, homogeneous, and accessed according to queue-based scheduling.

Cloud providers offer portions of a large-scale distributed system to their users according to an Anything as a Service (XaaS) paradigm, where a technology managed by the provider is delivered to end-users over the Internet [106]. The former three layers of XaaS stack were: Software as a Service (SaaS), where clients merely utilise applications entirely managed by the cloud provider; Platform as a Service (PaaS), where clients deploy and configure applications on top of the provider-managed infrastructure; Infrastructure as a Service (IaaS), where clients have full control over operating systems and libraries and can partially configure hardware resources such as storage and networking [105]. With the spread of the XaaS paradigm, new and more specific acronyms have been added to the "anything" list, such as Machine Learning as a Service (MLaaS) [107] or Big Data Analytics as a Service (BDAaaS) [108].

Akin to HPC, the IaaS paradigm usually gives users access to computing resources through a remote shell. Apart from that, the two architectures are tremendously different from each other. HPC users share the entire computing stack, from bare metal resources to the operating system kernels. In contrast, within the IaaS paradigm, each user gains privileged access to its own virtualised resources, i.e.

[18]https://slurm.schedmd.com/

[19]https://www.openpbs.org/

[20]https://www.ibm.com/docs/en/spectrum-lsf/

hyper-converged Virtual Machines (VMs), virtualised storage, and Software-Defined Networks (SDNs).

At first glance, it could seem that the IaaS paradigm gives users more control over their infrastructure than the HPC approach, as the combination of on-demand access and privileged capabilities makes cloud resources very flexible. The downside is that a user has no control over whatever exists between the physical hardware and its VM. As a consequence, compilers cannot apply hardware-specific optimisations. Plus, with *overprovisioning*, multiple virtual cores are allocated to the same physical core so that each application can use only a fraction of the actual CPU time. Consequently, software performances can fluctuate significantly from one run to the other, and such fluctuations depend on factors that are not under users' control, such as the current load of the underlying infrastructure or the cloud control plane itself. Recently, with the diffusion of DL workloads, many cloud providers started to offer bare metal resources and hardware accelerators alongside the classical virtualised options, but the related costs are commonly very high.

On the other hand, a cloud environment is ideal for hosting long-lived public services, such as web servers or databases, which are not compatible with the air-gapped worker nodes and the one-shot allocation model of HPC. Moreover, *hybrid cloud* solutions allow for partitioning the modules of complex applications between public and private nodes, publishing only those services that need to be accessed and hiding the others behind configurable firewalls. Nevertheless, more flexibility necessarily comes with increased complexity in deploying, configuring and managing applications.

For this reason, several tools have been developed with the precise aim to seamlessly support hybrid cloud deployments of complex applications composed of heterogeneous intercommunicating services. These tools expose to the user just an agnostic and straightforward interface, usually a DSL. This approach is often referred to as Infrastructure as Code (IaC), and the tools in charge of translating infrastructure descriptions into their runtime deployment and life-cycle management are known as *orchestrators*.

There are two leading families of orchestrators: *agentless* and *agent-based*. The first family concentrates on the deployment and configuration phases, with no support for life-cycle management. Infrastructures are described through either a DSL [109]–[111] or a GUI [112], and interpreted by an agentless architecture. A centralised server communicates with both the cloud control plane to deploy IaaS instances and the instances themselves to configure them. At the time of writing, one of the most popular tools of this kind is Ansible[21] from Red Hat.

Agent-based orchestrators support both deployment and life-cycle management of infrastructures. The control plane installs an agent on each instance during the

[21] https://www.ansible.com/

configuration phase to continuously collect information on the system's state. When the system state differs from the one described in the infrastructure declaration, the orchestrator tries to restore the correct one. With this strategy, systems can provide advanced orchestration features, such as self-healing, auto-scaling, and garbage collection. The first implementations of this approach, such as LCFG [113] and CfEngine [114], date back to the first half of the '90s. More recent examples are present in scientific literature, such as Engage [115], SALSA [116], and Roboconf [117], and among industrial products, like Juju[22], Chef[23], and Puppet[24].

Workflows' reproducibility can enormously benefit from an IaC approach, as it offers a repeatable way to recreate the entire execution environment for an experiment. On the other hand, advanced orchestration technologies can rely on workflows for expressing complex deployment and management processes. For instance, Mietzner et al. [118] adopt WS-BPEL, the OASIS Web Services Business Process Execution Language [119], to describe the provisioning of complex SaaS architectures, while Brogi et al. [120] propose Petri Nets to define applications management protocols in TOSCA, the OASIS Topology and Orchestration Specification for Cloud Applications [121].

2.4 Container-based virtualisation

Operating system containers, or simply *containers* in modern terminology, are lightweight virtualised environments that trade isolation for efficiency [9]. While VMs rely on a *hypervisor* to implement virtualisation at the hardware abstraction layer, containers running on the same host share the operating system kernel, moving virtualisation at the operating system layer. This design undoubtedly allows for better peak performances, but guaranteeing isolation among different containers is much more complex than isolating VMs, both in terms of security and performance.

Akin to standard virtualisation terminology, a *container image* contains the instructions for starting a *container instance* (often simply called *container*). As containers are much more lightweight than VMs, each container is usually dedicated to a single application and its dependencies. This approach increases modularity, reduces the attack surface, and improves maintainability. A *container runtime* process running on the host is in charge of converting images into isolated execution environments and managing their life-cycle.

Early containerisation technologies, like OpenVZ[25] or Linux-VServer[26], were

[22]`https://jujucharms.com/`

[23]`https://www.chef.io/chef/`

[24]`https://puppet.com/`

[25]`https://openvz.org/`

[26]`http://linux-vserver.org/`

based on colossal kernel reengineering efforts to guarantee isolation at the process level, with each solution adopting a different (although similar) strategy to achieve this goal [9]. In recent years, the significant updates in Linux kernel isolation-related features and the creation of container-related standardisation communities, such as the Open Container Initiative[27], significantly reduced the implementation-specific differences in container runtime systems.

In containers ecosystems, the term isolation is commonly referred to two different contexts: *resources isolation*, which regulates the usage of hardware resources for each container instance, and *security isolation*, which segments access and visibility to global operating system's objects (e.g. PIDs, user and group ids, file descriptors). In modern implementations of Linux container runtimes, resources isolation is achieved through limits and control groups, while security isolation is obtained through namespaces, system call filtering (SecComp), and access control modules (SELinux, AppArmor).

Container file systems are usually *ephemeral*: file writes are allowed, but restarting the container instance restores the whole file system to the initial state, as described in its image. For permanent data storage, persistent *volumes* can be mounted to the file system of one or more containers. A volume can be managed directly by the container runtime or point to a portion of the host's file system. The second option must be used carefully because it significantly reduces the isolation level between host and container contexts.

Among modern implementations, Docker[28] containers are undoubtedly the most widespread. Their success is probably due to the relatively user-friendly specification format for creating Docker images (i.e. the Dockerfile), the support for image inheritance, which saves users from building everything from scratch every time, and the possibility to distribute images through public and private registries. The fundamental merit of Docker is to have pushed the containerisation idea to the mainstream, consequently generating significant improvements in the whole ecosystem. As an example, the aforementioned Open Container Initiative was established by Docker in cooperation with many of the major cloud providers.

Recently, containers gave birth to new software architecture paradigms. An example is the *microservices* pattern, where a modular application is decomposed into multiple interacting containers, each dedicated to a single module. The clear advantages of this approach are better maintainability, less intrusive system updates, and easy extensibility with new modules. A drawback comes from the complexity of deploying and connecting a potentially high number of containers, manage their life-cycle, and recover from failures. To ease these tasks, some container orchestrators

[27]https://opencontainers.org/

[28]https://www.docker.com/

started to flourish. Among them, Kubernetes[29] has become the de-facto standard for container orchestration during the last years, and the vast majority of cloud providers include a managed Kubernetes service in their offering. Kubernetes comes with a very flexible YAML-based DSL describing both deployment and runtime orchestration features for multi-agent applications.

Even if containers were initially thought of for performance-oriented platforms, as HPC systems are, their adoption in this field is still quite limited compared to the cloud. Sec. 2.4.1 gives an overview of this topic. Sec. 2.4.2 focuses instead on the workflow domain, where containers bring undoubted advantages in portability and reproducibility.

2.4.1 Containers on HPC

Regarding performance, overheads introduced by containerised software with no network and storage virtualisation are almost negligible compared to bare metal executions [122], [123]. Still, adoption of software containers in HPC environments is significantly limited, with most data centres stuck on legacy software distribution systems like environment modules.

There are two main obstacles to the widespread adoption of containers in HPC. The first one mainly affects Docker containers: if the container runtime daemon runs with root permissions in an HPC environment, a malevolent root user inside a container can manage to get privileged access to the host operating system. This is a serious hazard for HPC systems because they are typically shared by different groups of users (a.k.a. tenants). To date, this issue is solved by many *rootless* container stacks, either directly deriving from Docker (e.g. udocker [124], socker [125], and Occam [126]) or with entirely alternative implementations (e.g. Podman[30], Shifter [127], Charliecloud [128], and Singularity [102]).

The second issue, still unsolved, involves performances portability. As discussed above, HPC software stacks need complex compilation toolchains, tightly coupled with the underlying hardware, in order to achieve maximum performance. This requirement implies that container images should be built directly on the HPC nodes, allowing compilers to apply hardware-specific optimisations. Still, building containers requires almost always privileged permissions (Charliecloud and Podman are exceptions) and should be managed directly by system administrators. Plus, the exact configuration of a given compilation toolchain that maximises performance is strictly related to each specific HPC facility. Conversely, images built elsewhere or downloaded from public registries can significantly reduce peak performance.

[29]https://kubernetes.io/

[30]https://podman.io/

2.4.2 Container-based workflows

Portability and reproducibility have always been two fundamental aspects of scientific workflows. Nevertheless, the combination of the two is undoubtedly a non-trivial requirement to satisfy, as it is necessary to guarantee that a program running on top of potentially very diverse execution environments will give identical results. First, it is necessary to provide identical versions of all the libraries directly or indirectly involved in the computation. On top of that, some numerical stability problems can arise when running the same code on different platforms, e.g. on Linux and Mac OS X [10]. Fortunately, with the diffusion of lightweight containerisation technologies like Docker and Singularity [102], a straightforward solution for these issues finally appeared. Nowadays, container-based tasks are supported by many WMSs on the market, either as an alternative to native execution or as first-class citizens [129].

The typical way to support containerisation in WMSs is through a one-to-one mapping between steps and containers, i.e. a container image is associated with each step in the workflow graph. In this setting, the execution flow of a single step always consists of three sequential actions: deploy the container, execute the step inside it, and finally undeploy the container. Drawing a parallel with the famous Flynn's taxonomy [97], this execution pattern could be defined as Single-Task Single-Container (STSC).

Compared with a Multiple-Tasks Single-Container (MTSC) alternative, the STSC pattern comes with a decisive advantage. Since containers' file system is commonly ephemeral, every step execution runs inside a clean and consistent environment (except for potential temporary files saved into persistent folders). For its part, an MTSC execution can provide some performance improvements in those cases when the step execution is high-speed (comparable with the startup and shutdown overheads of a container, generally in the order of milliseconds). Moreover, an MTSC approach can also be useful when a process inside the container must complete a heavy initialisation phase before being ready to perform steps or when some data dependencies reside in the ephemeral file system, to avoid additional data transfers when recreating containers.

The Single-Task Multiple-Containers (STMC) setting is more interesting, as it allows using multiple, possibly heterogeneous environments to execute a single step. For example, with an STMC approach, it would be possible to run an MPI step on top of multiple nodes or a MapReduce-based step with multiple instances of Apache Spark.

Finally, the most general setting of Multiple-Tasks Multiple-Containers (MTMC) would also allow *concurrent* step executions, i.e. a configuration in which steps s_1 and s_2 run contemporarily on different resources and s_1 produces data consumed by s_2. The support for this last configuration becomes fundamental when dealing with stream-based workflows [39]. In principle, also an MTSC configuration enables the concurrent execution of steps into the same resource, but here the advantage is less valuable. Indeed, it is far easier to obtain the same behaviour in an STSC setting with a single step charged with launching and managing all the required processes.

Unfortunately, a many-to-many step-image association is not enough to model an *MC configuration, as it is also necessary to explicitly specify the connections among different containers. Some ways to define multi-agent environments with containers are already present on the market, from simple libraries like Docker Compose[31] and Singularity Compose[32] to complex orchestrators like Kubernetes. One option is to rely on them for the environment definition, substituting the original one-to-one step-container association with a many-to-one step-environment association and treating an entire multi-agent environment as the unit of deployment. Some WMSs natively designed on top of Kubernetes, e.g. Argo Workflows[33], adopt precisely this approach. Despite offering great flexibility in interacting with Kubernetes resources, these products are tightly coupled with such technology and do not allow for step offloading on different environments, such as HPC sites. Conversely, the hybrid workflows approach proposed in this work offers a flexible and technology-agnostic way to specify each of the *T*C patterns directly in the workflow model (see Chapter 3).

2.5 Literate computing

Donald Knuth introduced *literate programming* in the early '80s as a programming approach where code fragments and their documentation in natural language are interleaved in the same source file [130]. A literate program can be compiled by a specific tool to obtain two different products: it can either be *tangled* to a code file or *woven* into a documentation manual. The original language, called `WEB`, mixed PASCAL code with LaTeX documentation, but several other examples support additional programming languages and documentation frameworks.

Drawing inspiration from literate programming, Millman and Pérez developed the concept of *literate computing* [11]. The main difference between the two approaches lies in their execution model. A literate program can be tangled to produce code, but compilation and execution of such code are beyond the paradigm's scope. In contrast, literate computing targets *interactive environments*, in which users can immediately and repeatedly execute commands. Plus, the results of such executions are stored as part of the file format, together with source code and documentation. Literate computing files are called *computational notebooks*, or simply *notebooks* when the context is clear. In recent years, several literate computing technologies have seen widespread diffusion. Jupyter Notebooks [12], formerly known as IPython Notebooks, are undoubtedly the market leaders. Their ecosystem is described in

[31] https://docs.docker.com/compose/

[32] https://singularityhub.github.io/singularity-compose/

[33] https://argoproj.github.io/argo-workflows/

Listing 2.1: 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",
      ...
    }],
}
```

greater detail in Sec. 2.5.1. Among others, significant examples are knitr[34] and Apache Zeppelin[35] notebooks.

The capability to unify imperative code and declarative metadata in a unique format puts literate computing halfway between high-level coordination languages and low-level distributed computing libraries, the two classes of tools commonly used for workflow modelling. Furthermore, the widespread diffusion of notebook technologies implies that many domain experts are already familiar with them, removing the need to learn additional, workflow-specific tools. For such reasons, the literature contains some efforts to use Jupyter Notebooks for workflow modelling, even if, as far as the author knows, this thesis is the first attempt to establish explicit models and semantics for *literate workflows*. Sec. 2.5.2 briefly describes the most relevant approaches in this direction.

Literate computing (in particular Jupyter Notebooks) has also been evaluated as a high-level interface to HPC resources. Indeed, feature-rich, user-friendly web dashboards that serve as Integrated Development Environments (IDEs) for literate computing technologies are far more accessible for domain experts than SSH-based remote shells. Plus, market-leading cloud providers already offer computational notebooks as interfaces to compose, execute, and scale ML pipelines on cloud resources. Sec. 2.5.3 explores the state-of-the-art in this topic.

2.5.1 Jupyter Notebooks

Jupyter Notebooks have been designed to support scientific computing, from

[34] https://yihui.org/knitr/

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

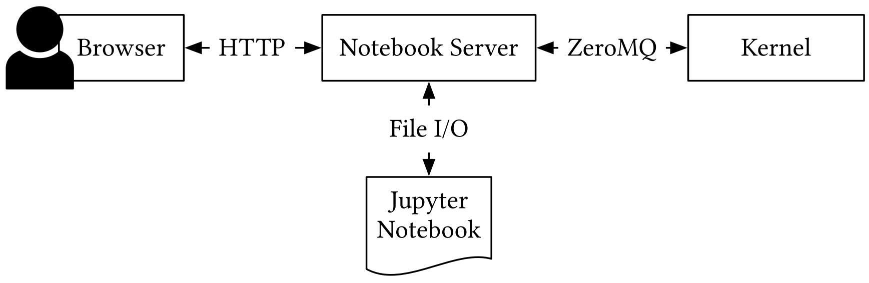


Figure 2.1: Sketch of the Notebook Server architecture.

interactive exploration to publication of a readable and replicable research object [131]. In practice, a Jupyter Notebook is a JSON document containing an ordered list of *cells* with code, explanatory text, mathematics, plots and rich media. The format of each cell type is regulated by a versioned JSON Schema[36], as reported in Listing 2.1 for generic and code cells.

As sketched in Fig. 2.1, users interact with Jupyter Notebooks through a *notebook server*, a web-based computational environment for creating and executing such documents. When a user triggers the execution of a cell, its code is sent to a dedicated, potentially remote process called *kernel*, which is in charge of interpreting the code, storing the notebook's global context, and managing the input and output streams. ZeroMQ[37], a TCP-based messaging library, implements the bidirectional communication protocol between the server and the kernel.

Initially thought as a high-level interface for the IPython project [132], nowadays the Jupyter ecosystem supports interactive programming in numerous languages with dedicated kernels and boasts a rich set of helper tools for, among other things, converting notebooks to standard file formats (nbconvert), publishing them on the web (nbviewer), and offering Jupyter as a service on cloud platforms (e.g. JupyterHub[38], Google Colaboratory[39]).

Although Jupyter Notebooks have seen a widespread diffusion in almost all areas of computational science, they are now ubiquitous in the data science field [133]. The reasons behind this trend are disparate. The possibility to discuss algorithms and visualise results in the same document is fundamental for data scientists, as every step in a data analysis pipeline can significantly affect its outcome. Plus, the long-standing compatibility with the Python ecosystem and the support offered by all the major DL frameworks on the market (e.g. PyTorch [134], TensorFlow [135], MxNet [136]) undoubtedly play a role. Another crucial aspect of the Jupyter

[36] `https://nbformat.readthedocs.io/en/latest/format_description.html`

[37] `https://zeromq.org/`

[38] `https://jupyterhub.readthedocs.io/en/stable/`

[39] `https://colab.research.google.com`

ecosystem is the possibility to run kernels on desktop machines, cloud resources or HPC facilities using the same high-level interface. This aspect allows data scientists to move the computation where the (large) datasets reside and to scale their experiments on huge amounts of computing power.

2.5.2 Workflows with Jupyter

Despite their attractive properties as workflow modelling tools, bare Jupyter Notebooks are unsuitable for complex workflows executions. Indeed, notebooks' purely sequential execution flow makes it impossible to exploit the inherent concurrency of workflow graphs. Nevertheless, some efforts to modify the Jupyter stack to overcome this limitation have been proposed in the literature.

Cloud-based scientific platforms like KBase [137], [138] and GenePattern 2.0 [139] have built their primary user interface on Jupyter. However, their main goals are to offer an integrated Bioinformatics programming environment, provide an extensive pipeline database, introduce user-friendly interfaces, and offload computation to their own target infrastructure, specifically an HTCondor cluster. The resulting notebooks are tightly coupled with the underlying software stack, preventing portability to the broader Jupyter ecosystem.

Netflix's nteract[40] framework adopts a more self-contained approach. It allows users to augment notebooks with input data and schedule them for batched execution. Outputs are also saved in notebook format, facilitating inspection and debugging. This approach's primary limitation is its coarse granularity, as it only allows to map an entire notebook to a single workflow step.

The Script-of-Scripts (SoS) project [140] moves the execution unit to the finer-grained level of single cells and introduces multi-language notebooks, called SoS Notebooks. Each cell of an SoS Notebook can declare a different host language, with the SoS runtime automatically handling object conversions during inter-cells communications. Dependencies between different steps can be specified directly in the code cells through a template-based mechanism. Despite its clarity and flexibility, this approach introduces a technology lock-in in notebooks, as their execution requires an interpreter able to disentangle the mix of coordination and application logic inside code cells.

The Notebooks into Workflows (NiW) project [141] adopts a different strategy. Instead of directly using notebooks as workflow descriptions, it statically translates a Jupyter Notebook into a WINGS workflow [142], 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.

[40] https://nteract.io/

Several attempts to extract the data stream through cell executions have appeared in the literature. For instance, Dataflow Notebooks [143] modify the behaviour 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[41] 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 [144] combines dataflow tracing with liveness and initialised variable analysis to detect staleness generated by cell re-executions in Python Notebooks. The Vizier framework [145] 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 aim to prevent the issues caused by out-of-order executions and repeated executions of Notebook cells, which are among the principal causes of reproducibility issues in the Jupyter ecosystem [146].

A different strategy to scale-up Jupyter Notebooks is to let users *explicitly* inject parallelism into their business code. This approach is proposed by `ipyparallel`, the official IPython parallel extension. However, several other task-based programming libraries allow an interactive construction of the task graph from Jupyter Notebooks (e.g. Techila, PyCOMPSS, Dask, Ray, and Parsl). The main disadvantage of this strategy is that it forces a tight coupling between business and parallel logic. The way such coupling is realised strongly depends on the chosen parallel library, making it impossible to migrate from one parallel solution to another without touching the business code.

2.5.3 Jupyter Notebooks on HPC

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.

Offloading standard Jupyter kernels from a server outside the data centre to the air-gapped worker nodes is not an option, as communications between notebooks and kernels require bidirectional connections. One of the most common approaches is to install a Jupyter-based service (e.g. a JupyterHub instance or a custom Jupyter Notebook spawner) directly on the login nodes of a data centre or on some publicly exposed instances of a tightly-coupled cloud service [147]–[152]. 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. At the same time, a custom job scheduling interface instantiates interactive Jupyter kernels on worker nodes.

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

Even if the software architecture of such approaches is always similar to the one sketched in Fig. 2.2, the implementation details differ from one solution to the other, being tightly coupled with the underlying HPC stack. The lack of a de-facto standard is undoubtedly one of the biggest obstacles to broad adoption. Other significant drawbacks are the need to modify the authentication mechanism, with all its security implications, and the setup and maintenance difficulties of such platforms, which are non-trivial even for expert system administrators.

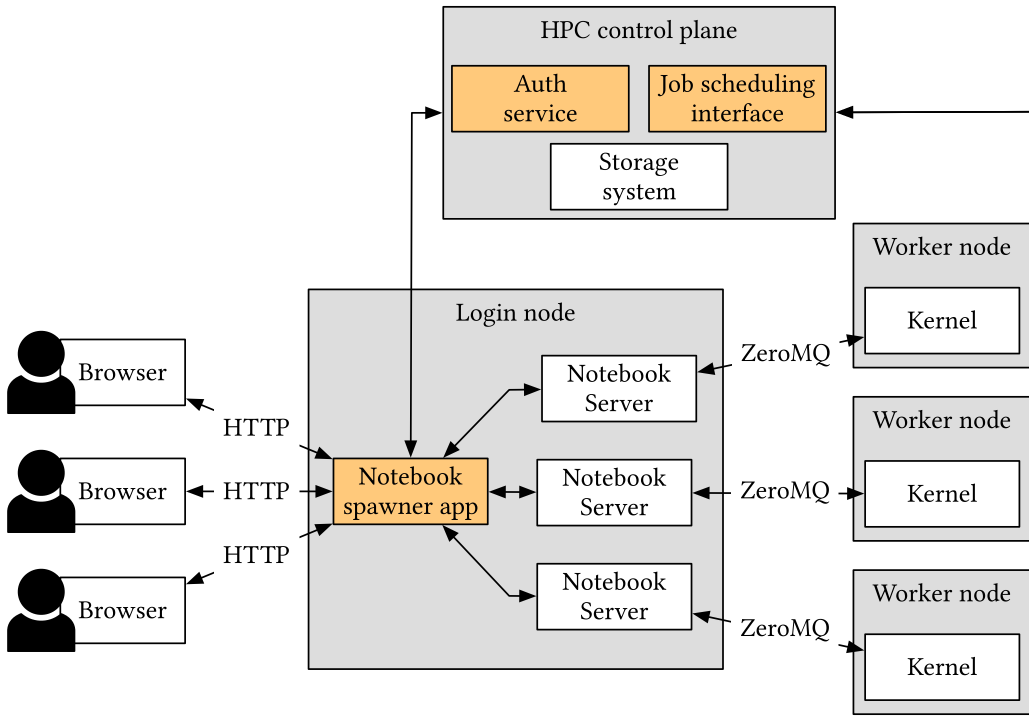


Figure 2.2: Sketch of the common architecture of a HPC Jupyter interface. Orange blocks usually need to be developed or extended to integrate Jupyter in the existing software stack.

Chapter 3

Hybrid workflows

In general, a *hybrid workflow* can be defined as a workflow whose steps can span multiple, heterogeneous, and independent computing infrastructures [39]. Each of these aspects has significant implications.

Support for *multiple* infrastructures implies that each step must potentially target a different *deployment location*[1] in charge of executing it. That location must have access to all the input dependencies, and it is likely to store all the output dependencies on its local scope. Locations can be *heterogeneous*, exposing different methods and protocols for authentication, communication, resource allocation and job execution. Plus, they can be *independent* of each other, meaning that direct communications and data transfers among them may not be allowed. A suitable model for hybrid workflows must then be *composite*, enclosing a specification of the workflow dependencies, a topology of the involved locations, and a mapping relation between steps and locations. This thesis aims at giving a formal definition of hybrid workflow models.

As discussed in Sec. 2.2, several products offer partial support for heterogeneous or distributed workflows, although with limitations. For example, many WMSs can execute workflows over heterogeneous architectures using connectors. However, they cannot split different portions of the same workflow among multiple execution environments, or they require the presence of unique storage space accessible from all compute units. More advanced WMSs like Triana, Askalon and Pegasus allow automated data transfers among different workers, but they require all compute units to be managed by a single orchestration tool. More importantly, all these details are not explicitly encoded in the workflow descriptions, making it difficult to extract and compare the runtime execution strategies of different WMSs.

A formal way to describe a workflow model not only in terms of steps and dependencies, but also in terms of runtime semantics, can serve as a low-level intermediate representation to compare different WMSs or migrate a model from

[1]In this dissertation, the term *location* always refers to the execution site of a step, as in e.g. [153], [154].

one WMS to another. Plus, it can serve as a coordination interface for a new class of *topology-aware* WMSs, where users can explicitly map steps onto (families of) processing elements to couple each step with the best-suited execution environment.

As described in Sec. 2.1, a workflow can be represented as a bipartite graph (see Def. 2.1.1), but the majority of WMSs on the market simplify this model to a DAG. Even if this choice strongly reduces the expressive power, DAGs are much easier to handle: start and end steps are always clearly identifiable, and there can be no deadlocks (at least at the coordination level). For this reason, the following discussion starts introducing hybrid acyclic workflows in Sec. 3.1, giving the fundamental definitions of location topology and mapping relations. Sec. 3.2 gives formal operational semantics for such models, discussing how a WMS can validate them through static soundness analysis and enact them by constructing a proper execution plan. Sec. 3.3 concludes the chapter by covering some advanced concepts.

3.1 Hybrid acyclic workflows

A hybrid acyclic workflow can be seen as the composition of two graphs: a bipartite workflow DAG specifying dependencies between its steps and a directed graph modelling the topology of (potentially heterogeneous) deployment locations. For each step, a many-to-many mapping relation $\mathcal{M}$ states the set of locations in charge of executing it. More formally:

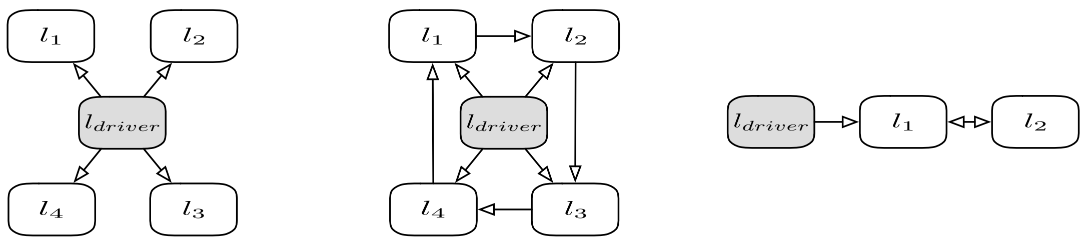
Definition 3.1.1. A hybrid acyclic workflow is a tuple $(W, \Gamma, \mathcal{M})$, where $W = (S, P, D)$ is a workflow DAG, $\Gamma = (L, C)$ is a topology of deployment locations, and $\mathcal{M} : S \to L$ is a many-to-many mapping relation. □

The workflow DAG is equivalent to the model described in Def. 2.1.1, with the additional constraint that dependency links cannot form cycles. Therefore, the following discussion will concentrate on the other two graphs: the *topology of deployment locations* and the *mapping relation*.

3.1.1 Topologies of deployment locations

Definition 3.1.2. A topology of deployment locations is a directed graph $\Gamma = (L, C)$, where the set of nodes L contains the locations and the set of links $C \subseteq (L \times L)$ contains communication channels between them. □

In the simplest case, all locations $l_i \in L$ are equal to each other, but the model can be refined to include *typed locations* to represent homogeneous sets of locations and *deployment groups* to model the co-allocation of multiple locations during the execution of a single step. Sec. 3.3.1 will further dig into these details. Also, communication channels can be typed to represent different communication technologies (e.g. SSH, HTTPS, FTP).



(a) Star topology. (b) Hybrid ring-star topology. (c) Linear topology.

Figure 3.1: Examples of location topologies. Control locations are coloured in grey, while processing locations are left blank.

Even if the majority of communication protocols are bidirectional, not always both parties can initiate a connection. For instance, a private server behind a NAT can connect to a public cloud instance, but the converse is not always true. In order to reflect these considerations, the direction of channel (l_1, l_2) in a topology refers to the capability of l_1 to initiate connections with l_2, even if the communication protocol is always assumed to be bidirectional.

Each location can have two distinct roles. A *control* location, which belongs to the WMS control plane, actively participates in management activities like data transfers, scheduling, or fault-tolerance. A *processing* location can only execute or *delegate* commands sent by control locations. A control location can also act as a processing location, sending commands to itself or executing commands received from other controllers. Let $\mathcal{C}(l_i, l_j)$ be a *communication path*, i.e. a sequence of channels $c_1, \ldots, c_m \in C$ connecting l_i to l_j. With this execution model, it is clear that a processing location l_p can execute a command only if a path $\mathcal{C}(l_c, l_p)$ exists, where l_c is a control location. In order to be consistent with this statement, assume each location l_i connected to itself with a channel $(l_i, l_i) \in C$. For the sake of brevity, such channels will not be explicitly mentioned when describing topologies henceforth.

When considering actual WMSs implementations, the most common topology is a *star*, with a single control location l_{driver} connected to all the other locations through unidirectional channels. Such architecture, represented in Fig. 3.1a, is general enough to support a wide range of real scenarios: l_{driver} can be a process running on a desktop machine, a public server on a cloud instance, or a login node in a data centre. The location topology representation of a star architecture with n processing locations is the following:

$$\Gamma_s = \left(\{l_{driver}, l_1, \ldots, l_n\}, \bigcup_{i=1}^{n} \{(l_{driver}, l_i)\}\right) \tag{3.1}$$

Data movements between different locations are always possible in a star topology, but they require two copy operations: a first one from the source to the driver and a second one from the driver to the destination.

There are other cases in which a topology-aware WMS can significantly optimise data transfers. For example, consider the hybrid ring-star topology of Fig. 3.1b:

$$\Gamma_{rs} = \left(\{l_{driver}, l_1, \ldots, l_n\}, \bigcup_{i=1}^{n} \left\{ (l_{driver}, l_i), \left(l_i, l_{(i+1) \bmod n}\right) \right\} \right) \tag{3.2}$$

In this case, a data transfer between locations l_i and l_{i+1} can pass through a single channel, which directly connects them. If channels are homogeneous and there is no contention on network resources, the cost of data transfers can be cut by half compared to a pure star topology. In addition, relying on direct channels help to reduce the contention on l_{driver}, further boosting the overall performance.

There are also situations in which a topology-aware WMS is necessary. Consider the linear topology of Fig. 3.1c:

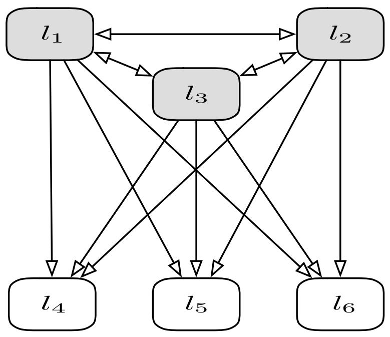
$$\Gamma_l = (\{l_{driver}, l_1, l_2\}, \{(l_{driver}, l_1), (l_1, l_2), (l_2, l_1)\}) \tag{3.3}$$

A WMS pretending to communicate directly with all the processing locations cannot transfer data from l_1 to l_2, as the latter is not reachable from l_{driver}. This is the case of low-level task-based libraries, such as Spark [57], Ray [63], or Parsl [64]. These libraries rely on all-to-all topologies in which a central control plane is connected to all the workers, and each worker can open connections with all the others. In this setting, data transfers between processing locations can always happen in a single step, but scenarios like the one in Fig. 3.1c require to instantiate a portion of the control plane (e.g. the Spark master or a Ray local scheduler) directly on location l_1, in fact transforming it to a control location.

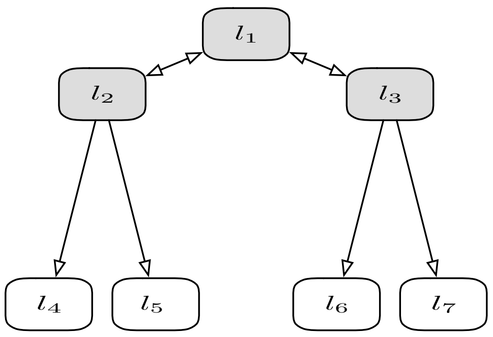
Conversely, in a topology-aware WMS a control location l_c can rely on *chained delegations* to communicate with every location l_i for which there exists a communication path $\mathcal{C}\,(l_c, l_i)$, without requiring a direct channel between them. Particularly, in the case of Fig. 3.1c l_{driver} can delegate to l_1 the execution of a command on l_2. This scenario is not uncommon in real cases: l_2 could be a private resource on a hybrid cloud or could not trust l_{driver}, but only l_1, in a federated environment. An actual WMS can implement this feature in several ways, e.g. with tunneling connections, nested remote command executions, or delegation certificates [155], choosing the best technique for each use case.

Location topologies can also include multiple control locations. One of the most common topologies in actual implementations of distributed WMSs is the *distributed star* depicted in Fig. 3.2a, where each control location is connected to all the others. Such topology can be used for either *active-active* configurations, where the workload is shared among all control locations, or *active-passive* scenarios, where there is only one active control location at a time, which can be replaced in case of failure. For instance, multiple Spark masters can be configured in an active-passive setting with the help of a Zookeeper service [156].

Another common scenario is the *hierarchical* topology shown in Fig. 3.2b, where a central control location is connected to multiple peripheral locations, managing



(a) Distributed star topology with 3 control locations and 3 processing locations.



(b) 2-level hierarchical topology with 3 control locations and 4 processing locations.

Figure 3.2: Examples of location topologies with multiple control locations. Control locations are coloured in grey, while processing locations are left blank.

disjoint subsets of processing locations to reduce scheduling and transfer overhead. The Ray distributed library adopts this topology [63]. Hybrid workflow models are general enough to describe all these architectures without forcing a topology-aware WMS to stick with a single configuration for all occasions.

3.1.2 Mapping relations

In hybrid workflow models, the mapping relation $\mathcal{M} : S \rightarrow L$ states which locations are in charge of executing each workflow step. In the simplest case, steps map onto single locations, but extending the mapping to more abstract scenarios is possible. For instance, steps can map onto location types, charging the WMS scheduler to choose the best location among the compatible ones (see Sec. 3.3.1).

The mapping relation forms a many-to-many relationship between steps and locations, but the semantics of one-to-many links is different in the two directions. Multiple locations related to a single step express a *spatial constraint*: all locations must be active before scheduling the step, and the execution will span all of them. Conversely, multiple steps related to a single location introduce a *temporal constraint*: the same location must execute all the steps one after the other, or even concurrently if it can handle multiple executions, and each step cannot start until its related location is available.

Note that the concept of *spatio-temporal composition model* has been already proposed in the literature to combine software components and workflow models in a unique programming paradigm [157], supporting explicit parallelism through skeletons [158], [159]. Among other advantages, these models can express the co-allocation of multiple workflow steps, which cannot be expressed in traditional workflow DAGs without violating the acyclic requirement. Hybrid workflows could easily be extended with an additional step-to-step mapping relation to support this feature, but for the sake of simplicity, this dissertation does not treat this more advanced scenario.

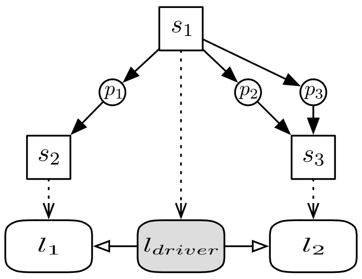


Figure 3.3: Example of a hybrid workflow. Steps are represented as squares and ports as circles. Dependency links between steps and ports are depicted as arrows with black-filled heads. Locations are represented as squashed rectangles and communication channels as arrows with white-filled heads. Control locations are coloured in grey, while processing locations are left blank. Finally, mapping relations are expressed as dotted arrows.

As a drawback, traditional spatio-temporal approaches collapse spatial and temporal planes into a single, component-based representation, making it difficult to properly manage the life-cycle of components with both spatial and temporal ports, especially in the presence of control structures in the coordination language. Instead, hybrid workflows keep the two dimensions separated and offer precise semantics for each life-cycle management operation, allowing a WMS implementation to infer an optimised execution plan for each well-formed workflow model (see Sec. 3.2.1).

Figure 3.3 shows a toy example of hybrid workflow. A step s_1 produces three different outputs, which correspond to ports p_1, p_2 and p_3. A second step s_2 depends on port p_1, and a third step s_3 depends on p_2 and p_3. None of them produces other outputs. This DAG is accompanied by a simple star location topology with a single control location l_{driver} and two processing locations l_1 and l_2. Step s_1 is executed locally by the driver, while s_2 and s_3 are offloaded to l_1 and l_2, respectively. Since the driver is directly connected to both l_1 and l_2, all data transfers and remote executions can be performed straightforwardly.

In symbolic notation, Fig. 3.3 can be written as follows:

$$\begin{aligned} W &= \left(\{s_1, s_2, s_3\}, \{p_1, p_2, p_3\}, \begin{Bmatrix} (s_1, p_1), (s_1, p_2), (s_1, p_3), \\ (p_1, s_2), (p_2, s_3), (p_3, s_3) \end{Bmatrix}\right) \\ \Gamma &= (\{l_{driver}, l_1, l_2\}, \{(l_{driver}, l_1), (l_{driver}, l_2)\}) \\ \mathcal{M} &= \{(s_1, l_{driver}), (s_2, l_1), (s_3, l_2)\} \end{aligned} \tag{3.4}$$

3.2 Operational semantics

Sec. 3.1 introduced the *syntax* of hybrid workflow models for both symbolic and graphical representations. The current section focuses instead on their *operational semantics*, describing how a topology-aware WMS is supposed to enact such models at runtime.

Consider the example in Fig. 3.3 again. Suppose that step s_1 completed successfully, producing data tokens on its output ports p_1, p_2, and p_3. In order to simplify the discussion, all tokens produced by a step are supposed to carry the related data directly. In addition, each step s is supposed to execute only once so that there is a one-to-one mapping between ports and tokens. A step can run multiple times in actual workflow executions, producing a list or even a stream of tokens into its output ports. The following discussion can be adapted to this scenario by explicitly creating a different version of a port $p \in Out(s)$ for each execution of s to recover a one-to-one mapping with tokens. Still, explicitly treating this scenario would only add unnecessary complexity to the discussion.

Let $\mathcal{P}(l_i) \subseteq P$ be the set of ports whose data tokens reside on $l_i \in L$. Since step s_1 has been executed on l_{driver}, then $Out(s_1) \subseteq \mathcal{P}(l_{driver})$. Indeed, each location is supposed to store its output data locally, or at least in a place that it can reach directly. Consider now step s_2. In a standard workflow DAG, s_2 would be ready to be executed, as its only input port p_1 contains a token. Things get a bit more complicated with hybrid models. Step s_2 is related to location l_1, but l_1 must be up and running prior to take part in data transfers and execute commands. Plus, p_1 data must reside on l_1 before executing s_2. In light of these considerations, it seems reasonable to impose three preconditions for executing a hybrid workflow step: all its input ports should contain values, its related location should be up and running, and all its input data should reside on that location. The first condition is implicit in the semantics of the standard workflow DAG, but a suitable operational semantics for hybrid workflows must reflect the other two.

Henceforth, let $\langle l, \mathcal{P}(l), s \rangle$ be the *location configuration* representing location l in state s containing data related to ports $\mathcal{P}(l)$. Let a location have two possible states, *active* (A) and *inactive* (I), and two related transitions, *deploy* and *undeploy*. Moreover, let $\langle l, \mathcal{P}(l), - \rangle$ refer to a location l in any allowed state. The deploy semantics is relatively straightforward:

$$\langle l, \mathcal{P}(l), - \rangle \xrightarrow{\text{deploy}} \langle l, \mathcal{P}(l), A \rangle \tag{3.5}$$

Inactive locations can neither execute steps nor take part in data transfers. Hence, the deployment operation would be preliminary to all other activities involving the location. It is reasonable to suppose that control locations can always deploy and undeploy processing locations, independently of the topology. However, all the control locations are always considered active, as a location cannot deploy itself. In addition, it is worth noting from Eq. (3.5) that deploying an already active location does not affect its configuration. Consequently, the deploy operation is reentrant.

The undeploy operation is quite delicate. There are cases in which location data are not persistent, as with container-based technologies like Docker and Kubernetes (see Sec. 2.4). This eventuality implies that when a location goes inactive, all its data must be considered lost:

$$\langle l, \mathcal{P}(l), - \rangle \xrightarrow{\text{undeploy}} \langle l, \emptyset, I \rangle \tag{3.6}$$

A positive side of this approach is that a location failure can be treated as a spurious undeploy operation without introducing additional semantics. As for deployments, the undeploy operation is reentrant and does not affect already inactive locations.

Data transfers are the most complicated operations in hybrid workflows, as they involve an analysis of the location topology. Even if processing locations can be instructed to move data autonomously in a realistic WMS implementation, only the control plane can initiate data transfers. Given that, when performing transfers between locations l_i and l_j, a control location must initiate connections with both l_i and l_j.

Within location topologies, such requirement assumes a precise meaning in terms of communication channels. Let $\mathcal{N}(l_i) \subseteq L$ be the set of *neighbours* of location $l_i \in L$, s.t.

$$\mathcal{N}(l_i) = \{l_j : (l_i, l_j) \in C\} \tag{3.7}$$

As discussed in Sec. 3.1.1, $l_j \in \mathcal{N}(l_i)$ means that l_i can initiate a bidirectional connection with l_j. Such connection can be used to perform three different actions: send data to l_j, receive data from l_j, or initiate a connection to a location $l_k \in \mathcal{N}(l_j)$ and delegate the data transfer operation. Each involved location must be active to participate in these actions. However, since deploy operations are always considered feasible, they can be safely excluded from the following discussion. The feasibility of a transfer operation can then be defined only in terms of communication paths.

Theorem 3.2.1. *A transfer operation between locations l_i and l_j is feasible iff there exist a control location l_c and two communication paths $\mathcal{C}(l_c, l_i)$ and $\mathcal{C}(l_c, l_j)$.*

Proof. With no loss of generality, consider a topology with a single control location l_c. If there are multiple control locations, the proof needs to be verified for at least one of them. Since only control locations can initiate data transfers, l_c must be able to initiate a connection with both l_i, to receive the data, and l_j, to send data. Since both operations require the same considerations, focus only on the former.

As discussed before, a location can only interact with its neighbours. If $l_i \in \mathcal{N}(l_c)$, then l_c can directly receive data from l_i. Otherwise, it can only open a connection with each $l_k \in \mathcal{N}(l_c)$ and implement a chained delegation according to a breadth-first search. If there is a communication path $\mathcal{C}(l_c, l_i)$, then such search will at a certain point find a location l_l s.t. $l_i \in \mathcal{N}(l_l)$, so that l_l can receive data from l_i and propagate them through the multi-hop connection spanning $\mathcal{C}(l_c, l_l)$. Otherwise, the receive operation cannot be performed. The same reasoning applies to the send operation, involving the existence of a path $\mathcal{C}(l_c, l_j)$.

An alternative strategy would be to send data directly from l_i to l_j. If $l_i = l_c$, then such operation is feasible, but it implies the existence of paths $\mathcal{C}(l_i, l_i)$, which always exists, and $\mathcal{C}(l_i, l_j)$. The same goes when $l_j = l_c$. Instead, if both l_i and l_j are processing locations, this strategy needs a path $\mathcal{C}(l_c, l_i)$ to initiate the send operation and a path $\mathcal{C}(l_i, l_j)$ to send data to l_j, implying the existence of a path $\mathcal{C}(l_c, l_j)$. □

Given a port $p \in P$ and a location $l_j \in L$, transferring p data to l_j is possible iff there is at least one feasible data transfer operation from a location l_i s.t. $p \in \mathcal{P}(l_i)$ to location l_j. The following corollary formally states this intuition:

Corollary 3.2.2. *Given a port p, a transfer of its related data to location l_j is feasible iff there exist a control location l_c, a location $l_i \in L$ s.t. $p \in \mathcal{P}(l_i)$ and two communication paths $\mathcal{C}(l_c, l_i)$ and $\mathcal{C}(l_c, l_j)$.*

Proof. If there exists no location l_i s.t. $p \in \mathcal{P}(l_i)$, it is evident that no transfer operation is allowed. Conversely, given $l_i \in L$ s.t. $p \in \mathcal{P}(l_i)$, the feasibility of the transfer operation is established by Theorem 3.2.1. □

Therefore, finding the optimal strategy for a data transfer is equivalent to finding the shortest path on a directed graph. A WMS can also rely on more detailed location topologies for better optimisation: channels can be weighted according to their performances, and such weight can be dynamically adjusted to take into account concurrent transfers that temporarily reduce the channel bandwidth.

When dealing with feasible operations, the transfer semantics for data related to port $p \in \mathcal{P}(l_s)$ are as follows:

$$\begin{aligned}\{\langle l, \mathcal{P}(l), A\rangle : l \in \mathcal{T}(l_s, l_d)\} \xrightarrow{\text{transfer}(p)} &\{\langle l, \mathcal{P}(l), A\rangle : l \in (\mathcal{T}(l_s, l_d) \setminus l_d)\} \cup \\ &\{\langle l_d, \mathcal{P}(l_d) \cup \{p\}, A\rangle\}\end{aligned} \tag{3.8}$$

where $l_s, l_d \in L$ are the source and destination locations and $\mathcal{T}(l_s, l_d)$ is the set of locations involved in the transfer operation. Contrary to what the reader could expect, the transfer(p) semantics is not simply defined in terms of l_d because this alternative hides many dangerous subtleties behind a more intuitive notation. The main flaw is a lack of unambiguity: multiple locations can store data from port p, and there can be multiple transfer strategies for a single pair of locations (l_s, l_d). By requiring the explicit set of involved locations $\mathcal{T}(l_s, l_d)$, the transfer semantics become unequivocal. Plus, note that transfer is always feasible when $p \in \mathcal{P}(l_d)$, as $l_d \in \mathcal{N}(l_d)$, but Eq. (3.8) does not affect the configuration of l_d. Consequently, also transfer operations are reentrant.

Eventually, the execute operational semantics for a mapping $\mathcal{M}(s)$:

$$\{\langle l, \mathcal{P}(l) \cup In(s), A\rangle : l \in \mathcal{M}(s)\} \xrightarrow{\text{execute}(s)} \{\langle l, \mathcal{P}(l) \cup Out(s), A\rangle : l \in \mathcal{M}(s)\} \tag{3.9}$$

This last rule explicitly states that all the output data produced by s reside on each related location, as assumed at the beginning of this section. Plus, it implies that each port $p \in In(s)$ contains a value, as p can be included in $\mathcal{P}(l)$ only due to either a transfer operation (which in turn requires p to be non-empty as per Corollary 3.2.2) or a previous execute operation.

Note that Eq. (3.9) remains deliberately general on the operational semantics of s itself. For example, it is not explicitly stated if data tokens in the set $In(s)$ are still present on a location l after the execution of s or if they are "consumed" during the operation. This choice is in line with the generality of the proposed approach, which aims at augmenting generic workflow models with topology-awareness without imposing strict requirements on such models. However, as a consequence, execute semantics cannot be considered reentrant.

Regarding the feasibility of execute operations, it is clear that a control location must send a command to all the involved locations to offload computation. The following theorem states precisely that:

Theorem 3.2.3. *An execute operation on a set of locations $\bar{l} = \mathcal{M}(s)$ is feasible iff there exists a control location l_c and a path $\mathcal{C}(l_c, l_i)$ for each location $l_i \in \bar{l}$.*

Proof. When a step $s \in S$ is mapped onto a single location $l_i \in L$, the proof is very similar to the one of Theorem 3.2.1 for transfer feasibility and is then omitted. Consider now a step s mapped onto a set of locations $\bar{l}$. In this case, there are two possible strategies for offloading computation. A first option is to directly send a command to each member of $\bar{l}$, which requires a path $\mathcal{C}(l_c, l_i)$ for each $l_i \in \bar{l}$. An alternative is to send the command only to a single location $l_i \in \bar{l}$, which executes it on the whole set of processing locations (as it commonly happens with SPMD libraries like MPI). For this operation to be feasible, a control location l_c must initiate a connection with location l_i, requiring a path $\mathcal{C}(l_c, l_i)$. Then, l_i must initiate a connection with all the other locations $\bar{l} \setminus l_i$, requiring the presence of a path $\mathcal{C}(l_i, l_j)$ for each $l_j \in (\bar{l} \setminus l_i)$. This condition again implies the existence of a path $\mathcal{C}(l_c, l_i)$ for each $l_i \in \bar{l}$. □

Now that the operational semantics have been defined, they can be used to derive an *execution plan* for a hybrid workflow, represented as a workflow DAG itself. Sec. 3.2.1 explores this topic in detail, comparing hybrid models and pure DAGs in terms of *suitability*.

3.2.1 Execution plan

Let us define as *static* a workflow DAG whose execution flow is entirely determined at compile time. For instance, the workflow DAG of Fig. 3.3 is static, as it contains no conditional or iterative branch whose behaviour depends on runtime values. If a hybrid model augments a static workflow DAG, it is possible to statically compile it into an *execution plan*, which takes the shape of a static workflow DAG of the same kind. This is an essential property of hybrid models, as such plans can be fully understood also by a topology-unaware WMS.

Definition 3.2.4. Given a hybrid model $(W, \Gamma, \mathcal{M})$ based on a static workflow DAG $W = (S, P, D)$, its execution plan can be statically inferred and expressed as another static workflow DAG $W' = (S', P', D')$, where S' contains the operational steps to execute the hybrid model, P' contains location configurations, and $D' = (S' \times P') \cup (P' \times S')$ contains the dependency links. □

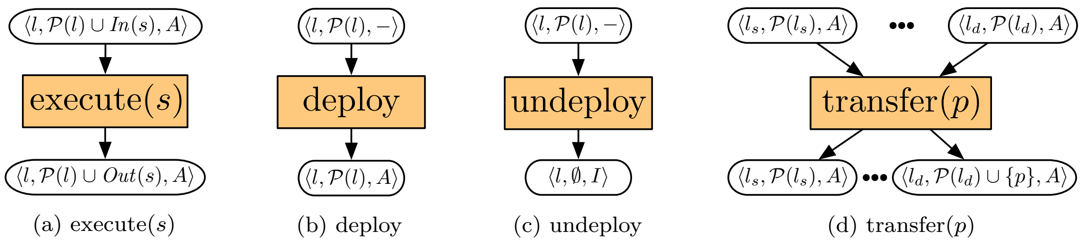

$\langle l, \mathcal{P}(l) \cup \mathit{In}(s), A\rangle$
execute(s)
$\langle l, \mathcal{P}(l) \cup \mathit{Out}(s), A\rangle$
(a) execute(s)
$\langle l, \mathcal{P}(l), -\rangle$
deploy
$\langle l, \mathcal{P}(l), A\rangle$
(b) deploy
$\langle l, \mathcal{P}(l), -\rangle$
undeploy
$\langle l, \emptyset, I\rangle$
(c) undeploy
$\langle l_s, \mathcal{P}(l_s), A\rangle$
$\langle l_d, \mathcal{P}(l_d), A\rangle$
transfer(p)
$\langle l_s, \mathcal{P}(l_s), A\rangle$
$\langle l_d, \mathcal{P}(l_d) \cup \{p\}, A\rangle$
(d) transfer(p)

Figure 3.4: Execution plan representations of hybrid workflows' operational semantics. Each operation becomes a step in a workflow DAG. The related steps are filled in orange to distinguish them from the steps of the original workflow, which are left blank.

An execution plan $W' = (S', P', D')$ can contain only four kinds of operations in S', corresponding to the four operational semantics described above. The shape of these steps in terms of their input and output ports is reported in Fig. 3.4. Note that each step receives one or more location configurations in input and returns different configurations for the same locations in output. Since DAGs do not allow feedback loops, this is the only way to represent stateful elements like locations. Plus, it is worth noting from Fig. 3.4d that the transfer block explicitly propagates all the locations in $\mathcal{T}(l_s, l_d)$ as outputs, even if the transfer semantics only affect the configuration of l_d. Such forwarding aims at making the locations available for further transfer or undeploy operations if needed.

Fig. 3.5 shows an example for the hybrid model of Fig. 3.3. Note that in the example the execute(s) operation is assumed to preserve data tokens related to its input ports $\mathit{In}(s)$ after execution, but the same plan is valid also in the contrary case. There can be multiple correct execution plans for a single hybrid workflow model: a possible strategy to build an execution plan is reported in Algorithm 1. Since the original workflow DAG is required to be static, it is always possible to:

- remove all the dead paths before building the execution plan, ensuring that each $s \in S$ will be executed at runtime (line 1);
- sort the steps according to a *topological order* that takes into account their dependencies (line 2).

The execution plan $W' = (S', P', D')$ is then initialised with empty steps and links sets, and the location configuration with the proper initial state is added to P' for each location $l \in L$ (lines 3–8). Then, each step $s \in S$ mapped onto a set of locations $\bar{l} = \mathcal{M}(s)$ generates its sub-plan (lines 9–22). First, it is necessary to transfer input data for each port $p \in \mathit{In}(s)$ to each location $l_i \in \bar{l}$ (lines 12–19). Note that, since steps are sorted according to valid execution order, if both the original workflow DAG and the topology of deployment locations are *sound*, there should always be a valid l_s at line 13. Even if formal requirements for soundness will be stated in Sec. 3.2.2, the reader can trust the author that all the examples presented in this section are correct. Before executing the transfer (line 18), all

Algorithm 1: A possible strategy for generating an execution plan.

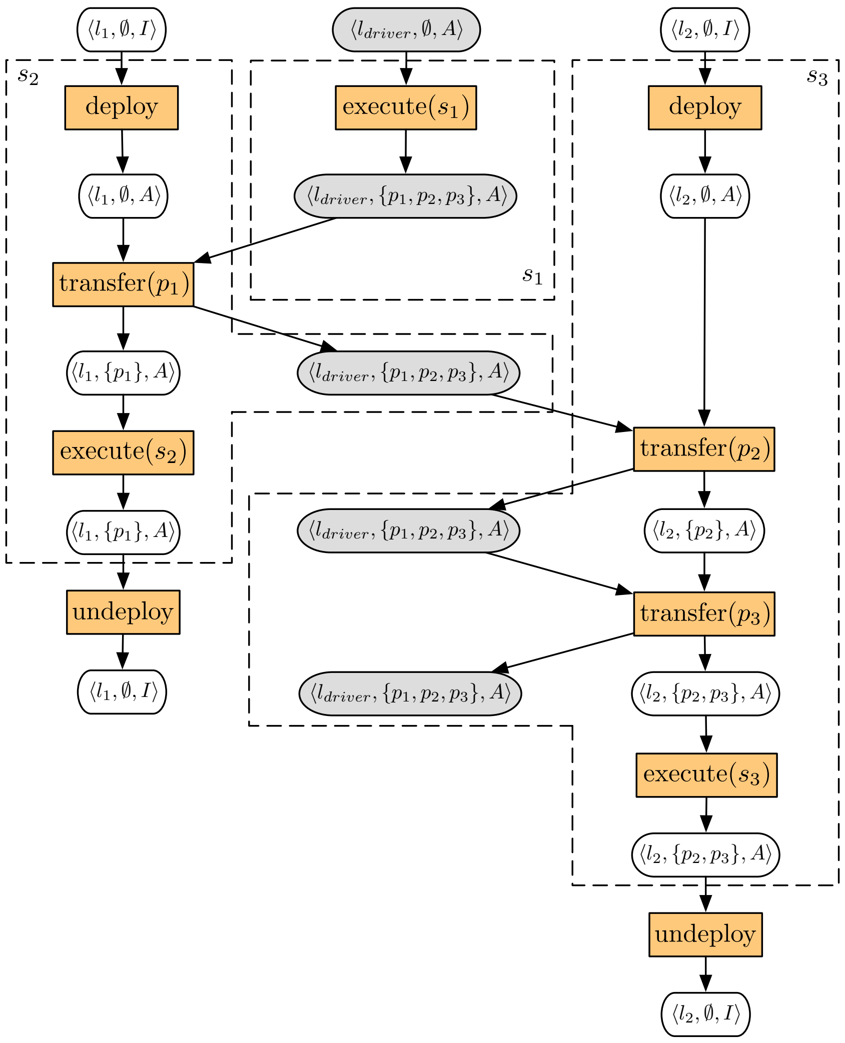
```
Data:
   Hybrid model $(W, \Gamma, \mathcal{M})$ where $W = (S, P, D)$ is a static DAG.
Result:
   Execution plan $W' = (S', P', D')$, again a static DAG.
Procedure:
1  Remove dead paths from $W$ s.t. each step is actually executed.
2  Sort $s \in S$ in a topological order according to the dependencies in $D$.
3  Initialise
4      $S' \leftarrow \emptyset$
5      $P' \leftarrow \{\langle l, \emptyset, A \text{ if } l \text{ is control else } - \rangle : l \in L\}$
6      $D' \leftarrow \emptyset$
7      $W' \leftarrow (S', P', D')$
8  end
9  For $s \in S$ do
10     $\bar{l} \leftarrow \mathcal{M}(s)$
11     For $l_i \in \bar{l}$ do
12         For $p \in In(s)$ do
13             For $l_j \in \mathcal{T}(l_s, l_i)$ do
14                 if $l_j$ is not control then
15                     $W' \leftarrow \text{add}(W', l_j \xrightarrow{\text{deploy}} l_j)$
16                 end
17             end
18             $W' \leftarrow \text{add}(W', \mathcal{T}(l_s, l_i) \xrightarrow{\text{transfer}(p)} \mathcal{T}(l_s, l_i))$
19         end
20     end
21     $W' \leftarrow \text{add}(W', \bar{l} \xrightarrow{\text{execute}(s)} \bar{l})$
22 end
23 For $\langle l_i, \mathcal{P}(l_i), A \rangle \in Out(W')$ do
24     if $\langle l_i, \mathcal{P}'(l_i), I \rangle \in In(W')$ then
25         $W' \leftarrow \text{add}(W', l_i \xrightarrow{\text{undeploy}} l_i)$
26     end
27 end
28 $W' \leftarrow \text{optimise}(W')$
29 return $W'$
```

processing locations in $\mathcal{T}(l_s, l_i)$ must be activated through a deploy operation (lines 13–17), while control locations are active by default. Finally, line 21 executes step s, propagating its outputs $Out(s)$ to each location $l_i \in \bar{l}$.

Let $In(W)$ and $Out(W)$ be the inputs and outputs of a workflow model $W = (S, P, D)$, i.e.

$$\begin{aligned} In(W) &= \{p \in P \colon\ p \notin Out(s)\ \forall s \in S\} \\ Out(W) &= \{p \in P \colon\ p \notin In(s)\ \forall s \in S\} \end{aligned} \tag{3.10}$$

When all the steps have been processed, all locations $l_i \in Out(W')$ that were initially inactive are connected to an undeploy operation for deactivation (lines 23–27). Indeed, the idea is that a sound execution plan should restore the initial state for each involved location. This simple linking strategy for undeploy steps works well as far as every step $s \in S'$ is actually executed at runtime. In the presence of dynamic workflows, proper placement of undeploy steps gets far more

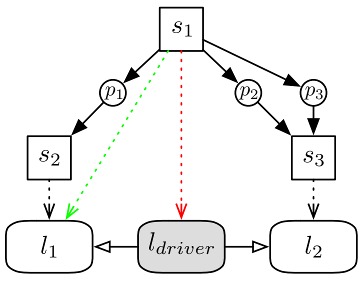


$\langle l_1, \emptyset, I\rangle$
$\langle l_{driver}, \emptyset, A\rangle$
$\langle l_2, \emptyset, I\rangle$
s_2
deploy
execute(s_1)
deploy
s_3
$\langle l_1, \emptyset, A\rangle$
$\langle l_{driver}, \{p_1, p_2, p_3\}, A\rangle$
$\langle l_2, \emptyset, A\rangle$
transfer(p_1)
s_1
$\langle l_1, \{p_1\}, A\rangle$
$\langle l_{driver}, \{p_1, p_2, p_3\}, A\rangle$
execute(s_2)
transfer(p_2)
$\langle l_1, \{p_1\}, A\rangle$
$\langle l_{driver}, \{p_1, p_2, p_3\}, A\rangle$
$\langle l_2, \{p_2\}, A\rangle$
undeploy
transfer(p_3)
$\langle l_1, \emptyset, I\rangle$
$\langle l_{driver}, \{p_1, p_2, p_3\}, A\rangle$
$\langle l_2, \{p_2, p_3\}, A\rangle$
execute(s_3)
$\langle l_2, \{p_2, p_3\}, A\rangle$
undeploy
$\langle l_2, \emptyset, I\rangle$

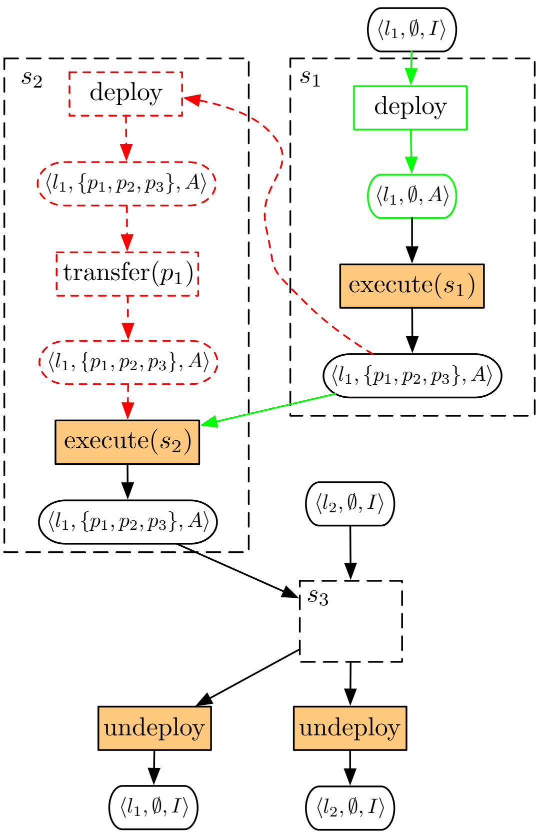
Figure 3.5: Execution plan for the hybrid workflow example of Fig. 3.3. Time flows from top to bottom. Sub-plans for each step are framed with dashed lines.

complicated. Sec. 3.3.2 contains a brief discussion on this topic. Finally, line 28 optimises the execution plan by removing redundant deploy, undeploy, and transfer operations. Note that execute operations can never be removed, as they are part of the original workflow.

As a concrete example, consider the execution plan in Fig. 3.5. Step s_1 is mapped onto the control location l_{driver} and takes no input. Therefore, the only action in its sub-plan is an execute(s_1), which produces values of ports p_1, p_2 and p_3 on location l_{driver}. Step s_2 is instead mapped onto the processing location l_1, which needs to be explicitly deployed. Then, its input port p_1 must be transferred to location l_1. Since l_{driver} stores the only copy of p_1 and can initiate a connection with l_1, a transfer(p_1) operation involving $\mathcal{T}(l_{driver}, l_1) = \{l_{driver}, l_1\}$ is the only feasible option. After that,



(a) Updated hybrid workflow model.


$\langle l_1, \emptyset, I\rangle$
s_2
deploy
$\langle l_1, \{p_1, p_2, p_3\}, A\rangle$
transfer(p_1)
$\langle l_1, \{p_1, p_2, p_3\}, A\rangle$
execute(s_2)
$\langle l_1, \{p_1, p_2, p_3\}, A\rangle$
s_1
deploy
$\langle l_1, \emptyset, A\rangle$
execute(s_1)
$\langle l_1, \{p_1, p_2, p_3\}, A\rangle$
$\langle l_2, \emptyset, I\rangle$
s_3
undeploy
undeploy
$\langle l_1, \emptyset, I\rangle$
$\langle l_2, \emptyset, I\rangle$

(b) Updated execution plan DAG.

Figure 3.6: The same example of Fig. 3.3, but with step s_1 mapped onto location l_1 instead of l_{driver}. Comparison of how such update reflects on the hybrid workflow model and the execution plan DAG. The sub-plan relative to step s_3 remains unchanged, so it is not reported.

step s_2 can run on l_1, but since it does not produce outputs, it does not affect the state of l_1. Finally, l_1 can be undeployed as no other step needs it. The sub-plan for step s_3 is similar.

Note that all transfers involving the same location are forced to be sequential. For example, all transfer operations involving l_{driver} depend on each other in Fig. 3.5. Multiple transfers between the same locations can easily be merged by a generalised transfer semantics transfer($\overline{p}$), where $\overline{p} = \{p_1, \ldots, p_n\}$ is a set of ports s.t. $p_i \in \mathcal{P}(l_s)$ for each $p_i \in \overline{p}$. Plus, if a location can handle multiple simultaneous connections, a proper WMS implementation can further increase the degree of parallelism.

It is also worth noting how the execution plan fails to explicitly represent the dependencies between steps and ports initially encoded in the workflow DAG, even if they somehow resurface at the sub-plans interconnection level. This flaw can make it hard to directly manage execution plans, as slight and intuitive modifications in the hybrid workflow model can translate into non-trivial reorganisations of the plan.

Consider, for example, the hybrid model in Fig. 3.6a. It is equal to the one in Fig. 3.3, but with step s_1 mapped onto l_1 instead of l_{driver}. This update is straightforward on the hybrid model, but it requires significant modifications to the execution plan, as inferred from Fig. 3.6b. First, it is necessary to modify the sub-plan of s_1 to deploy l_1 before launching the execute operation, but this is intuitive as the configuration of s_1 has been changed on the hybrid model. Then, the sub-plans of s_2 and s_3 can be attached by replacing l_{driver} with l_1 in their transfer operations. Nevertheless, since s_2 is executed on the same location as s_1, the plan can be dramatically simplified by removing unnecessary deployment and transfer operations.

Comparing the *suitability* of the two syntaxes, i.e. the modelling effort needed to describe the same workflow [160], it is clear how hybrid models are more user-friendly than pure execution plans. Still, even if suitability can be a good reason in itself to adopt a new workflow model [161], pure DAGs have the clear advantages to be well-known to domain experts, well-studied in literature, and compatible with whatever WMS on the market. Also, a WMS can provide users with a set of predefined deploy, undeploy, and transfer blocks for the most common technologies and scenarios. As an example, the Ystia suite[2] adopts this approach to orchestrate workflows on cloud-HPC infrastructures. The possibility to automatically translate a static hybrid workflow to an (optimised) execution plan combines the best of two worlds.

All the models presented in this section were supposed to be sound. However, it is still unclear how a user (or a WMS) can know if a hybrid model is sound. The straightforward but not so satisfying answer is: when at least one correct execution plan exists to enact it. Sec. 3.2.2 formally defines when a hybrid model can be considered sound and discusses how such property can be statically inferred.

3.2.2 Soundness

Automatic soundness analysis is a fundamental concept in workflow modelling, as a paradigm made to express large and complex applications cannot rely on runtime trial and error as the only debugging tool. Given that, the concept of soundness has been extensively treated in literature for most workflows abstractions, from Petri Nets and their extensions [162], [163] to dataflow models [164]. This section gives a formal definition of soundness for hybrid workflow models and their execution plans.

As discussed before, a hybrid workflow aims at augmenting a workflow DAG with topology awareness. Given that, the soundness of the original DAG cannot but be a requirement for the soundness of its hybrid counterpart. However, the way to prove it is strictly related to its semantics and is out of this dissertation's scope. In any case, when building an execution plan from a workflow DAG all the data dependencies between workflow steps must be preserved. Let $\mathcal{D}(s_1, s_2)$ be a

[2]https://ystia.github.io/

dependency path, i.e. a sequence of dependencies $d_1, \ldots, d_n \in D$ connecting s_1 to s_n. In a sound plan, the dependency paths connecting the execute operations in D' should reflect those connecting the related steps of the original workflow DAG in D.

In a sound binding graph, each step $s \in S$ should be mapped onto at least one location $l \in L$. Otherwise, it is not clear which locations are in charge of its execution. A WMS can provide a default binding, e.g. to the control location l_{driver}, when a user does not explicitly specify it for a step. Conversely, unbound locations are perfectly admissible: they can still serve as bridges for multi-hop data transfers.

Consider now location topologies. For sure, models requiring unfeasible transfers cannot be considered sound, as no proper execution plan can be generated for them. Plus, all execution operations should be feasible, too. According to Theorem 3.2.1, the latter requirement is satisfied iff there is a control location $l_c \in L$ able to initiate a communication with each location $l_i \in \mathcal{M}(s)$ for any step s. The feasibility of transfer operations is more complicated, as it also involves data tokens, but the exact set of locations storing data from a port p at a given time depends on each specific execution plan. Still, it is not a realistic option to require an exploration of the execution plans space to determine the soundness of a model. The following two statements ensure that, for static DAGs and typical WMS implementations, the feasibility condition for execute operations is also sufficient to ensure the feasibility of all transfer operations.

Theorem 3.2.5. *Let a hybrid workflow $(W, \Gamma, \mathcal{M})$ augment a static DAG $W = (S, P, D)$ and have a single control location $l_c \in L$. If there exists a path $\mathcal{C}\,(l_c, l_i)$ for each location l_i s.t. $l_i \in \mathcal{M}(s)$ for any step $s \in S$, then there exists an execution plan $W' = (S', P', D')$ in which all transfer operations are feasible.*

Proof. Consider the execution plan generation strategy of Algorithm 1. In order to prove the theorem, it is sufficient to show that this strategy always generates an execution plan in which all transfers are feasible. Consider a port $p_i \in (Out(s_i) \cap In(s_j))$, and let $l_i \in \mathcal{M}(s_i)$ and $l_j \in \mathcal{M}(s_j)$. First of all, line 12 requires the existence of at least one location l_s s.t. $p_i \in \mathcal{P}(l_s)$. Since steps are topologically sorted according to their dependencies in D, step s_j must be processed after s_i so that $p_i \in \mathcal{P}(l_i)$. Let then $l_s = l_i$. Since $l_i \in \mathcal{M}(s_i)$, then, by hypothesis, there exists a path $\mathcal{C}\,(l_c, l_s)$, and since also $l_j \in \mathcal{M}(s_j)$ there must be a path $\mathcal{C}\,(l_c, l_j)$, as there is only one control location $l_c \in L$. By Corollary 3.2.2, this is sufficient to prove the feasibility of the transfer operation. □

Corollary 3.2.6. *Let a hybrid workflow $(W, \Gamma, \mathcal{M})$ augment a static DAG $W = (S, P, D)$ and have a set of control locations $\bar{l}_c \subseteq L$ s.t. there exist a path $\mathcal{C}\,(l_c, l'_c)$ for each $l_c, l'_c \in \bar{l}_c$. If there exists a path $\mathcal{C}\,(l_c, l_k)$ between any $l_c \in \bar{l}_c$ and each location l_k s.t. $l_k \in \mathcal{M}(s)$ for any step $s \in S$, then there exists an execution plan $W' = (S', P', D')$ in which all transfer operations are feasible.*

Proof. Since each control location can initiate connections with all the others, the statement can be trivially proven by combining Corollary 3.2.2 with the fact that, for each $l_c, l'_c \in \bar{l}_c$, there exists a path $\mathcal{C}(l_c, l_k)$ iff there exists another path $\mathcal{C}(l'_c, l_k)$ for any $l_k \in L$. □

Even if limited to static DAGs, these statements strongly suggest the possibility that sufficient conditions for the existence of a feasible plan can be inferred directly from the hybrid workflow model. Guided by this assumption, which for sure needs to be tested on more general scenarios, the following statement introduces a precise definition for the soundness of a topology of deployment locations, which only relies on communication paths.

Definition 3.2.7. Given a hybrid workflow $(W, \Gamma, \mathcal{M})$, its topology of deployment locations $\Gamma = (L, C)$ is sound if, for each port $p_i \in (Out(s_i) \cap In(s_j))$, it contains at least a control location $l_c \in L$ s.t. there exist two communication paths $\mathcal{C}(l_c, l_i)$ and $\mathcal{C}(l_c, l_j)$ with $l_i \in \mathcal{M}(s_i)$ and $l_j \in \mathcal{M}(s_j)$. □

Note that the soundness requirement in Def. 3.2.7 is stricter than the hypotheses of Theorem 3.2.5 and Corollary 3.2.6, as in the former case the control location l_c must be *the same location* for both (s_i, l_i) and (s_j, l_j). Such requirement is general enough to cover the (probably purely theoretical) situation in which multiple control locations cannot communicate with each other. The following statements take into account all the previous considerations to define soundness for binding graphs and hybrid workflow models.

Definition 3.2.8. Given a hybrid workflow $(W, \Gamma, \mathcal{M})$, its mapping relation $\mathcal{M} : S \to L$ is sound if $\mathcal{M}(s) \neq \emptyset$ for every step $s \in S$. □

Definition 3.2.9. Given a hybrid workflow $(W, \Gamma, \mathcal{M})$, it is sound if the original workflow model $W = (S, P, D)$ is sound, its topology of deployment locations $\Gamma = (L, C)$ is sound, and its mapping relation $\mathcal{M} : S \to L$ is sound. □

One or more execution plans can be generated to enact a sound hybrid workflow model at runtime, but only a subset of them is sound. As discussed above, a sound execution plan $W' = (S', P', D')$ must execute all the steps $s \in S$, and the dependency paths connecting the related execute operations in D' must not violate the dependency paths between the steps themselves in D. Another fundamental aspect is that all the operations in S' should be feasible. Otherwise, the plan cannot be enacted. Plus, a plan that deploys initially inactive processing locations without undeploying them at the end should also be considered unsound, as it can lead to undesirable situations. For instance, consider a workflow requesting over half a thousand VMs to a cloud provider without releasing them after completion. The formal definition of soundness reported below captures all these considerations.

Definition 3.2.10. Given a sound hybrid model $(W, \Gamma, \mathcal{M})$, its execution plan $W' = (S', P', D')$ is sound if:

1. $\langle l, \mathcal{P}(l), s\rangle \in \mathit{In}(W')$ implies that $\langle l, \mathcal{P}'(l), s\rangle \in \mathit{Out}(W')$;
2. each operation $s' \in S'$ is feasible;
3. for each pair of steps $s_i, s_j \in S$, the existence of a dependency path $\mathcal{D}(s_i, s_j)$ in D implies the existence of $\mathcal{D}(\text{execute}(s_i), \text{execute}(s_j))$ in D'.

□

Even if limited to static workflows, the following statement establishes a fundamental property: there always exists a sound execution plan for a sound hybrid workflow model.

Theorem 3.2.11. *Given a sound hybrid model* $(W, \Gamma, \mathcal{M})$ *based on a static workflow DAG* $W = (S, P, D)$*, there always exists a sound execution plan* $W' = (S', P', D')$ *for it.*

Proof. Consider the execution plan generation strategy of Algorithm 1. In order to prove the theorem, it is sufficient to show that this strategy always generates a sound execution plan.

Requirement 1 can be proven by induction. As P' is initially populated with a configuration for each location and no operation can generate new locations, $\mathit{In}(W')$ contains a configuration $\langle l_i, \mathcal{P}(l_i), s\rangle$ for each $l_i \in L$. If there is no operation $s' \in S'$ s.t. $l_i \in \mathit{In}(s')$, then $\langle l_i, \mathcal{P}(l_i), s\rangle \in \mathit{Out}(W')$, and Requirement 1 is trivially satisfied. Otherwise, it is necessary to distinguish between two cases: $s = A$ and $s = I$. In the former case, the only operation that can change the state of l_i is the undeploy operation, but Algorithm 1 never binds l_i to an undeploy operation if $\langle l_i, \mathcal{P}(l_i), A\rangle \in \mathit{In}(W')$. Conversely, $\langle l_i, \mathcal{P}(l_i), I\rangle$ can only be attached to either a deploy or an undeploy operation, as transfers and executions require active locations. Since in Algorithm 1 an undeploy operation is always the last operation involving a location, either l_i is directly connected to an undeploy, or it is connected to a deploy, a (potentially empty) sequence of transfers and executions, and finally an undeploy. In any case, the undeploy operation returns $\langle l_i, \mathcal{P}'(l_i), I\rangle$, and no further operation modify its state, so $\langle l_i, \mathcal{P}'(l_i), I\rangle \in \mathit{Out}(W')$.

The proof of Requirement 2 is very similar to the one of Theorem 3.2.5, and it is therefore omitted. Requirement 3 states that dependency paths between execute operations in D' must mimic the dependency paths between the related steps in D. As Algorithm 1 explicitly sorts the steps in a topological order according to their dependencies in D, a step s_j can never be processed before step s_i if there exists a dependency path $\mathcal{D}(s_i, s_j)$. Consider a pair of steps $s_i, s_j \in S$ and suppose that exists a dependency path $\mathcal{D}(s_i, s_j)$ involving only one port p_i s.t. $\mathit{Out}(s_i) \cap \mathit{In}(s_j) = \{p_i\}$. Let $\mathcal{M}(s_i)$ contain only location l_i so that execute(s_i) adds p_i data to $\mathcal{P}(l_i)$. Before this operation, there was no location $\langle l_i, \mathcal{P}(l_i), A\rangle \in P'$ s.t. $p_i \in \mathcal{P}(l_i)$. In addition, since the DAG is static, p_i data token cannot be generated by multiple alternative steps. Consequently, every operation involving p_i

must depend, either directly or indirectly, on the execution of s_i. Let now $\mathcal{M}(s_j)$ contain only location l_j, s.t. the execution of s_j only requires p_i to be stored on l_j. The execute(s_j) operation must then depend on execute(s_i), directly if $l_i = l_j$ or indirectly through a transfer if $l_i \neq l_j$. In any case, there always exists a dependency path $\mathcal{D}\left(\text{execute}(s_i), \text{execute}(s_j)\right)$ in D', satisfying Requirement 3.

The same reasoning can trivially be extended to the case when $\mathcal{D}\left(s_i, s_j\right)$ contains one or more intermediate steps, each with one or more involved ports, as it is sufficient to repeat the proof for every pair of steps s_k, s_l s.t. $\mathit{Out}(s_k) \cap \mathit{In}(s_l) \neq \emptyset$. The case when a single step s is bound to multiple locations is also trivial. Indeed, even if the same data tokens are stored on multiple locations after the execution of s, this does not affect in any way the temporal dependencies between subsequent steps executions.

$\square$

3.3 Advanced topics

The previous sections formally introduced hybrid workflow models for static acyclic workflows with flat topologies of deployment locations. Being static, these models represent a perfect ground for unambiguous statements without the need to impose constraints or limitations on the underlying semantics. Conversely, the current section deals with more complex but more realistic models with a more relaxed approach.

The goal here is not to extend the formalism to cover all the features of hybrid workflows, which would be highly demanding and useless at the same time. Instead, the current section gives the reader an outlook of possible extensions of the baseline introduced so far, together with a brief discussion on their advantages and complexities. In particular, Sec. 3.3.1 introduces hierarchical location topologies, while Sec. 3.3.2 deals with dynamic workflow graphs.

3.3.1 Advanced topologies of deployment locations

In all hybrid models discussed so far, the mapping relation directly links steps with the locations in charge of executing them. An execution plan cannot prescind from such detailed information, but some cases could greatly benefit from more abstraction. Consider, for example, a hybrid model with two equal locations $l_1, l_2 \in L$ and three steps $s_1, s_2, s_3 \in S$ independent of each other. Since the two locations are equal, they could execute any of the steps, but such a detailed step-location mapping requires to statically assign two steps to the same location. There are cases in which a much better strategy would be to assign the three steps to *any* of the locations l_1 and l_2 and let a scheduler automatically decide the single mappings, either at runtime according to a FIFO policy or at compile time applying some pre-configured decision rules.

Location types serve precisely this purpose. In a *typed location topology*, each location comes with a type (or even multiple types if necessary), locations of the same type are considered interchangeable, and the mapping of a step onto a location type assumes the meaning of "execute the step on any location of the proper type". This kind of mapping must necessarily be resolved to a basic step-location mapping before executing the step so that all the concepts introduced in the previous sections are still valid. This resolution mechanism can safely be implemented either at compile time or directly at runtime, but static soundness analysis becomes much more cumbersome in the latter case.

Notice that the concept of type automatically brings to mind more powerful techniques, such as inheritance and composition. Here the golden rule is: location topologies can become as complex as needed, but the more complexity is preserved at runtime, the more difficult it becomes to validate them for soundness statically.

Another critical aspect concerns deployment and undeployment operations. Sec. 3.2 introduced semantics in which the deploy and undeploy granularity is the single location, but in actual implementations, there are cases in which the unit of deployment is a group of locations. Consider, for example, Kubernetes Pods, which can be composed of multiple and heterogeneous containers that are forced to co-exist for their entire life-cycle.

In order to represent such settings, multiple locations can be gathered in *deployment groups*, which represent the units of deployment for a location topology. This concept needs extended operational semantics, which are reported below for a deployment group $\bar{l} = \{l_1, \ldots, l_n\}$.

$$\left\{\langle l_i, \mathcal{P}(l_i), - \rangle : l_i \in \bar{l}\right\} \xrightarrow{\text{deploy}} \left\{\langle l_i, \mathcal{P}(l_i), A \rangle : l_i \in \bar{l}\right\} \tag{3.11}$$

$$\left\{\langle l_i, \mathcal{P}(l_i), - \rangle : l_i \in \bar{l}\right\} \xrightarrow{\text{undeploy}} \left\{\langle l_i, \emptyset, I \rangle : l_i \in \bar{l}\right\} \tag{3.12}$$

All the properties and theorems discussed in previous sections remain valid with deployment groups. The only thing that could look problematic at first glance is the proper placement of undeploy operations in an execution plan, but since Eq. (3.12) depends on all the involved locations, there is no risk of early deactivations.

As a final note, introducing both deployment groups and typed locations in a hybrid model is perfectly fine, and StreamFlow does precisely that (see Chapter 4). In this setting, it is necessary to distinguish three different layers in a location topology: the *units of deployment*, represented by the deployment groups, the *units of mapping*, i.e. the location types, and the *units of scheduling*, i.e. the single locations.

3.3.2 Dynamic and cyclic workflows

Sec. 3.1 introduced hybrid models for acyclic workflows, and Sec. 3.2.1 further constrained them to be static when dealing with execution plans. On the other hand, realistic workflows usually contain more complex constructs, such as choices and iterations with conditions depending on runtime data. It is then crucial to ensure that hybrid models can be coupled with more general workflows, but unfortunately the way to do it cannot prescind from the specific semantics of the workflow model.

Static workflows are never a problem, not even when they contain cycles. Indeed, a cycle can easily be unrolled into a DAG at compile time when the number of iterations can be determined a priori. Conversely, the way to resolve dynamic constructs varies from one representation to the other. Consider, for example, a port $p_i \in P$ and two different steps $s_i, s_j \in S$ s.t. $p_i \in (In(s_i) \cap In(s_j))$. In that case, a model can adopt XOR semantics, executing either s_i or s_j, and it can choose the candidate step randomly or according to a specific policy, implementing an *Exclusive choice* or a *Deferred choice* pattern [165]. This is what happens typically with Petri Nets [29]. Another model, such as the CWL dataflow model [14], can instead rely on AND semantics (implementing a *Parallel split* pattern), propagating a copy of the token to both s_i and s_j, while a third model can completely disallow a configuration with multiple steps depending on the same port.

Another important aspect, which is again related to the specific workflow model, is the discarded step of a conditional branch. Most models skip the related steps and the whole set of their successors, which are not executed. However, for instance, CWL propagates a `null` value to the outputs of a discarded sub-workflow so that if a step s_j depends on another step s_i and s_i is discarded, s_j is still executed, perhaps with a default input value. The same reasoning applies to the stopping conditions of non-determinate iterative branches.

On the one hand, imposing specific requirements on the workflow models semantics would strongly and unnecessarily limit the field of application of the proposed approach. On the other hand, an overarching analysis of its compatibility with all the existing workflow models is beyond the scope of this work, whose aim is introducing the general idea of hybrid workflows and not doing a roundup of their subtleties. Nevertheless, there is room for some general considerations, starting from dynamic settings.

The main issue with dynamic conditions is to find a proper way to handle deployments and undeployments in an execution plan. Indeed, some of the steps can be skipped in a dynamic workflow, leaving the entire branch of successors unexecuted when the workflow ends. Suppose that a step s_i belongs to one of these branches and that it is the unique step bound to a location $l_i \in L$. Deployment operational blocks, as conceived in Sec. 3.2.1, do not accept any input dependency other than the location to deploy. As a consequence, l_i should always be deployed, even when s_i is not executed.

The reverse situation, i.e. when the successor of an unexecuted step should undeploy an active location l_i, is also problematic. Indeed, it violates Requirement 1 for execution plans soundness (see Def. 3.2.10). The introduction of conditional deployment and undeployment blocks can seem a trivial solution, but it requires specifying semantics for conditions, which cannot but depend on the semantics of the coupled workflow. Another strategy could be to build a different execution plan for each static version of the execution flow, but there are cases in which this is either unfeasible or too complex to be managed in a reasonable amount of time.

Dynamic loops present similar issues: a loop with no iterations is equal to a discarded branch in a condition, and the loop termination logic contains nothing but another condition. Plus, care must be taken to place undeploy steps outside loop iterations. Otherwise, still necessary ports could be deleted, violating Requirement 2 of Def. 3.2.10. However, hybrid cyclic workflows require far deeper thought, as more subtle issues are to be faced. For instance, should different iterations of the same step be forced to run on the same location l_i, or can a runtime mapping operation modify the model according to some logic?

Note that the presence of dynamic workflows only affects static soundness analysis and compatibility with standard WMSs, as a topology-aware WMS can always generate an execution plan incrementally at runtime. Still, a further investigation of dynamic models is a must. In this thesis, the hybrid workflows approach is coupled with two paradigms: the CWL standard (see Sec. 4.1.5) and the literate computing (see Sec. 5.3). The former adopts a dataflow model with conditional branches, while the latter is inherently sequential, but none of them is iterative. Coupling hybrid workflows with a non-DAG paradigm is an unescapable path for future research.

Chapter 4

StreamFlow

The StreamFlow framework[1] has been created as runtime support for *hybrid acyclic workflows* on *multi-container environments* [13]. The interested reader is referred to Chapter 3 and Sec. 2.4.2 for a detailed discussion of such concepts. Being focused on the execution phase, StreamFlow does not offer any intrinsic way to model workflows. Instead, it has been designed to seamlessly integrate with external coordination semantics, allowing users to augment existing workflows with hybrid capabilities. In particular, it is fully compliant with the CWL open standard. The same design concept applies to most supported execution environments, which are described in an external, well-known format whenever such format exists (e.g. Helm[2] charts for Kubernetes deployments or Slurm scripts for HPC workloads). This chapter details the StreamFlow implementation (Sec. 4.1) and evaluates it, both in terms of programmability and performances, with two real applications in the domains of Bioinformatics and DL (Sec. 4.2).

4.1 Implementation

The StreamFlow framework's logical stack is depicted in Fig. 4.1. It should be clear from the higher portion of the figure that a StreamFlow execution needs three different types of inputs: workflow descriptions, location topology descriptions, and a StreamFlow file with the step-location mapping (Sec. 4.1.1). Such mapping relation translates into an MTSC pattern by default, but StreamFlow can be explicitly configured to adopt any task-container mapping pattern (Sec. 4.1.2). The first operational step is a *translation* of an external workflow format into StreamFlow's internal representation. Currently, only CWL-based workflows are supported, but the integration with additional design tools and formats is in plans (Sec. 4.1.3). Before actually executing a step, it is necessary to deploy the related location. The

[1]`https://streamflow.di.unito.it`

[2]`https://helm.sh/`

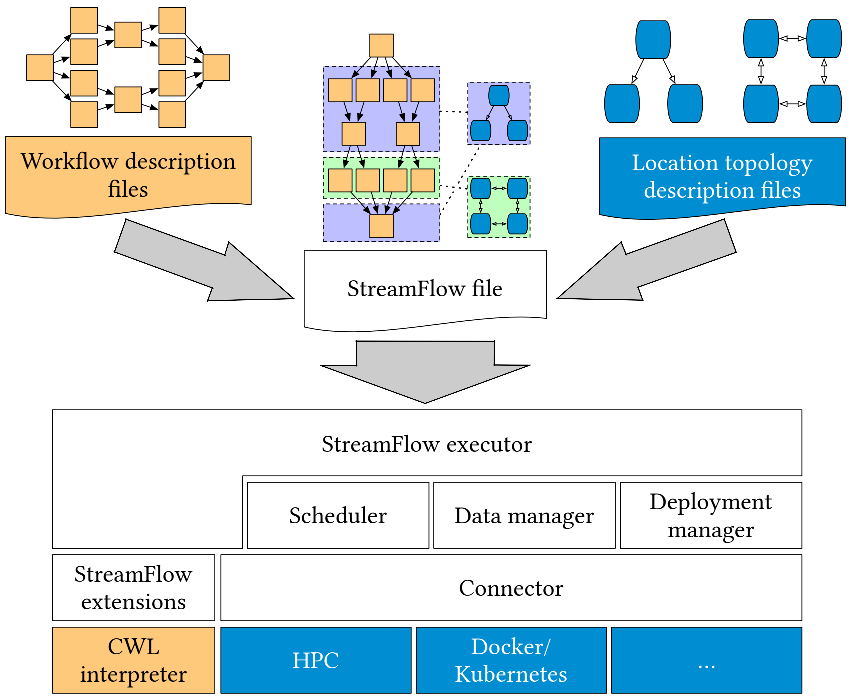


Figure 4.1: StreamFlow framework's logical stack. Coloured portions refer to existing technologies, while white ones are directly part of StreamFlow's codebase. In particular, the yellow area is related to the definition of the workflow's dependency graph, while the blue area refers to the execution environments.

`DeploymentManager` component has precisely the role of deploying locations when needed and destroying them as soon as they become useless (Sec. 4.1.4). The `Scheduler` component is then in charge of selecting the best locations to execute each step while guaranteeing that all requirements are satisfied. Each step can generate one or more *jobs*, i.e. runtime command instances running on top of their assigned locations to process a given input-output data partition (Sec. 4.1.5). Finally, the `DataManager` component, which knows where each job's input and output data reside, must ensure that each location can access all the data dependencies required to complete the assigned job, performing data transfers only when necessary (Sec. 4.1.6).

4.1.1 The StreamFlow file

When launching a StreamFlow execution from the Command Line Interface (CLI), its only argument is the path of a YAML file, conventionally called `streamflow.yml`. The crucial role of that file is to map each workflow step onto the location that should execute it. Moreover, to ensure this mapping is unambiguous, each location and each step should be *uniquely identifiable*.

In order to effectively represent complex location topologies, StreamFlow relies on a three-level hierarchical format (see Sec. 3.3.1). Each deployment group, called

Listing 4.1: StreamFlow file format

```
"version": "v1.0",
# Workflow descriptions
"workflows": {
  "workflow-name": {
    "cwl-example": {
      "type": "cwl", # Only the CWL type is supported
      "config": {
        # Pointers to the CWL workflow description files
        "file": "example.cwl",
        "settings": "input-values.yml"
      },
      "bindings": [
        {
          # Posix-based name of the selected workflow step
          "step": "/step1",
          "target": {
            # Model-service pair where the step must run
            "model": "model-name",
            "service": "service-name",
            # The step requires 2 instances of the selected service
            "locations": 2,
          }
        }
      ]
    }
  }
},
# Model descriptions
"models": {
  "model-name": {
    "type": "docker" | "singularity" | "docker-compose" | "helm" | "ssh" | "occam" | "slurm" | "...",
    "external": "true" | "false",
    "config": {
      # This schema depends on the selected model type
      ...
    }
  }
}
```

model, is managed independently by a dedicated implementation of a `Connector` interface, which acts as a proxy for the underlying orchestration library. Models constitute the *units of deployment*, as all their components are always co-allocated when one of them executes a step. A single model can include multiple location types, called *services*. Services are the *units of mapping*, as StreamFlow users can map each workflow step with a single service for execution. Finally, multiple replicas of the same service can coexist in a given model. Each service can then refer to one or more running instances, called *locations*[3], which constitute the *units of scheduling*. Indeed, each step can be offloaded to a configurable number of locations to be processed.

This section describes the StreamFlow file syntax and the strategies adopted to guarantee naming uniqueness. Listing 4.1 reports a commented example of a StreamFlow file to help the reader follow the discussion. Interested readers can find

[3]Locations were called *resources* in the original StreamFlow article [13]. Nevertheless, the term resources is commonly used to indicate hardware resources (e.g. cores, memory and disk space) provided by each unit in an execution environment. Therefore, their name has been changed to avoid potential ambiguities.

the complete and authoritative specification of the StreamFlow file format in the official JSON Schema document[4].

A valid StreamFlow file contains the version number (which currently only accepts the `v1.0` value) and two main sections: `workflows` and `models`. The `workflows` section consists of a dictionary of uniquely named workflow specifications, i.e. objects containing three fields: the `type` field identifies which language has been used to describe the workflow; the `config` field includes the paths to the files containing such descriptions; the `bindings` list contains the step-service mapping relations.

Many WMSs and coordination languages on the market express workflows as *nested dependency graphs*, in which each node can refer to either a simple step or a nested sub-workflow. Therefore, StreamFlow adopts a Posix-like naming scheme, mapping simple steps to files and sub-workflows to folders. In particular, the most external workflow description is mapped onto the root folder. This scheme allows for easy and unambiguous identification of steps, given that there exists an intuitive way to assign a name to each step in the workflow's graphical structure and that this name has the univocity constraint required by a typical file system representation. Fortunately, most coordination languages on the market satisfy these requirements, and the CWL standard is not an exception.

The `models` section contains a dictionary of uniquely named model specifications, i.e. objects with two distinct fields: the `type` field identifies which `Connector` implementation should be used for its creation, destruction and management; the `config` field contains a dictionary with configuration parameters for the corresponding `Connector`.

Usually, the `config` parameters are directly extracted from the tools used to interact with the underlying orchestration library (e.g. the `helm` CLI for Helm charts or the `docker-compose` CLI for Docker Compose). A user who is familiar with these libraries can easily understand the StreamFlow format. The best way to unambiguously identify services in a model strictly depends on the model specification itself. For instance, in Docker Compose models, it is sufficient to take a key in the `services` dictionary, while for Kubernetes and Helm, the user is explicitly required to fill in the `name` attribute of each container in a Pod template with a unique identifier.

The format adopted for the `bindings` list takes into account all previous considerations on unambiguous identification of steps and services. Each list element contains a `target` object, with a (`model`, `service`) pair that uniquely identifies a service and a `step` attribute containing a path in the aforementioned Posix-based naming scheme. A step can also be bound to multiple locations through the `locations` numeric field, which defaults to `1`. If the path resolves to a folder, i.e. to a sub-workflow, the same target is applied recursively in the file system hierarchy

[4]`https://raw.githubusercontent.com/alpha-unito/streamflow/master/streamflow/config/schemas/v1.0/config_schema.json`

unless a more specific configuration, i.e. another entry in the `bindings` list with a deeper path in its `step` field, overrides it. Bindings can also be grouped, i.e. a single element of the bindings list can, in turn, be a list of bindings. These groups model step-to-step bindings, translating themselves into co-allocations of the involved steps and locations at runtime.

4.1.2 Task-container mapping patterns

Since models are not redeployed after each step execution when multiple steps are bound to the same service, StreamFlow implements by default an MTSC pattern. This design choice minimises data transfers to achieve better performances, seamlessly reusing all the outputs stored on the ephemeral portion of a container file system. As a drawback, it gives up the clean and consistent execution environment commonly provided by containers, which can be problematic if traces left by previous jobs can have unexpected effects on the next ones.

If an STSC pattern is required, it can be induced by adding a `recycle` directive to a binding entry in the StreamFlow file. Such directive induces a redeployment of the related service before executing a job. In this case, StreamFlow will automatically handle all required data transfers, ensuring that at least one copy of each job output is stored in a persistent location before deleting the container.

If a single step is mapped onto multiple locations, StreamFlow implements an STMC pattern, executing the related job only on the first location. An additional `STREAMFLOW_HOSTS` environment variable contains the comma-separated list of allocated hostnames. This strategy allows for straightforward compatibility with the standard launcher-based SPMD libraries, such as MPI.

Finally, even if the CWL standard alone cannot explicitly describe co-allocations, the MTMC pattern can be expressed with the help of step-to-step bindings, forcing the co-allocation of multiple steps and their related locations.

4.1.3 The WMS integration layer

As stated before, one of the design choices for the StreamFlow approach is to rely on existing coordination languages instead of coming with yet another way to describe workflow models. In particular, it is fully compliant with the CWL open standard. Being a fully declarative language, CWL is far simpler to understand than its Make-like or dataflow-oriented alternatives. Moreover, some existing WMSs provide at least a partial compatibility with CWL format, even when it is not their primary coordination language. Last but not least, the CWL's reference implementation, called `cwltool`[5], is written in Python, allowing StreamFlow to use the official library to obtain a compiled workflow representation and handle some complex aspects of the standard (e.g. the evaluation of inline JavaScript code).

[5]https://github.com/common-workflow-language/cwltool

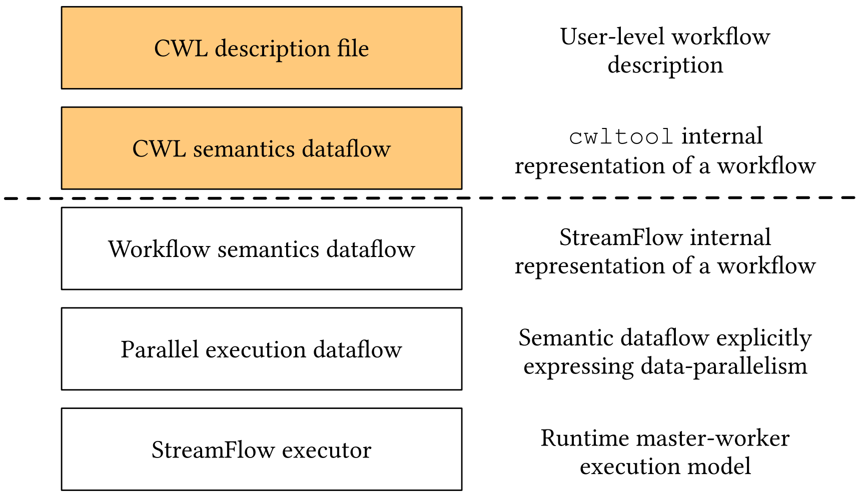

CWL description file
User-level workflow description
CWL semantics dataflow
cwltool internal representation of a workflow
Workflow semantics dataflow
StreamFlow internal representation of a workflow
Parallel execution dataflow
Semantic dataflow explicitly expressing data-parallelism
StreamFlow executor
Runtime master-worker execution model

Figure 4.2: StreamFlow framework's layered dataflow model. Yellow blocks refer to the CWL runtime library's workflow representations, called `cwltool`, while the white ones are internal representations adopted by the different layers of the StreamFlow framework.

CWL semantics can be used to describe a workflow through a declarative JSON or YAML syntax, written in one or more files with `.cwl` extension. Optionally, an additional configuration file contains a list of input parameters to initialise a workflow execution. The `cwltool` library natively translates the CWL semantics in a low-level macro dataflow graph [166] and implements multi-threaded runtime support. Nevertheless, even if CWL is the primary coordination language in StreamFlow, it is still worthwhile to avoid too tight coupling between CWL logics and the StreamFlow runtime to support additional coordination languages in the future if needed.

For this reason, the StreamFlow framework adopts a layered dataflow model [66], as depicted in Fig. 4.2. First, a `Translator` component compiles the CWL dataflow semantics into an internal macro dataflow graph representation. Notice that this representation supports much broader semantics, including loops, stream-based input ports and from-any activation policies. This double step in the compilation process can ensure an adequate decoupling of the StreamFlow's workflow representation to any specific coordination language, making it easier to include new languages in the future. On the other hand, a strongly modular structure of the code allows the co-existence of a robust core implementation of the dataflow model with some CWL-specific components (e.g. a `CWLTokenProcessor` object in charge of build tokens from raw output data), enabling full CWL support.

CWL workflows are not entirely static, as they support conditional branches through the `when` directive. As mentioned in Sec. 3.3.2, a `null` value is propagated to the outputs of a discarded sub-workflow so that the resulting token still activates subsequent steps while skipping the internal ones. This aspect can be problematic for the proper treatment of undeploy operations in an execution plan. However, the issue can trivially be solved by introducing conditional deploy and undeploy semantics so that the related operations are executed only when the branch condition is satisfied.

When dealing with explicit parallel semantics, whether data-parallel constructs as Scatter/Gather or stream-parallel patterns like pipeline executions, the same node of a dataflow graph can be executed multiple times. Therefore, the runtime support needs a lower, parallelism-aware layer that represents each workflow step as the set of its execution units. In StreamFlow, such execution units are called *jobs* and are the only entities directly visible to the underlying runtime components for scheduling, execution and fault tolerance purposes. The unique parallel execution pattern natively included in CWL standard is the `scatter`, in which a list of input data is partitioned among multiple, identical tasks that can be executed in parallel by multiple nodes. Still, in principle, the StreamFlow runtime can support stream processing pipelines, as the related parallel patterns can be trivially implemented through token-pushing semantics.

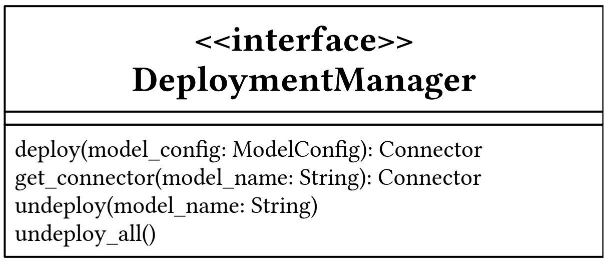
The StreamFlow control plane implements a master-worker pattern, i.e. a centralised control node manages all the aspects of workflow executions, including data transfers, jobs scheduling, and fault tolerance. Regarding availability and performance, the pros and cons of such design choice are discussed in detail in Sec. 4.1.6.

4.1.4 Model life-cycle management

In StreamFlow, the model deployment and the subsequent step execution happen in two distinct phases, leaving the models' life-cycle management to an external orchestration library whenever possible. This strategy allows StreamFlow users to rely on all the orchestration features provided by a mature product (e.g. autoscaling, restarting policies, affinity-based scheduling) without additional constraints. Moreover, they can adopt the original deployment description language, avoiding the extra effort needed to learn a new syntax.

As discussed in Sec. 3.2.1, deployment and undeployment steps can be explicitly embedded in an execution plan DAG. A potential drawback of this approach is that, since deployment steps have no dependencies, a standard scheduler will try to execute them as soon as possible, according to an *eager* allocation strategy. Some models can be up and running long before they are needed in this setting, wasting both energy and money. Even worse, in the case of conditional branches, a model could be deployed and never be used. For such reasons, StreamFlow adopts a more practical *lazy* approach, letting a model be deployed by the first fireable step requiring it.

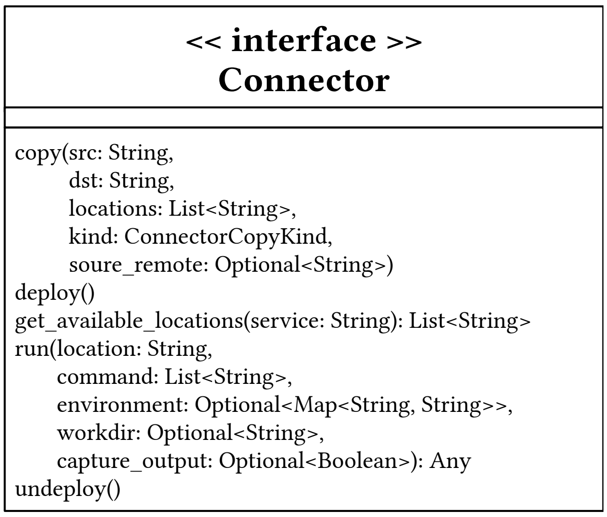
Since a single model instance can execute multiple steps, a consensus strategy is needed when concurrent steps require the same model. Centralising the deployment and life-cycle management to a unique `DeploymentManager` component is undoubtedly the easiest way to guarantee consensus. The Unified Modeling Language (UML)



```
<<interface>>
DeploymentManager

deploy(model_config: ModelConfig): Connector
get_connector(model_name: String): Connector
undeploy(model_name: String)
undeploy_all()
```

Figure 4.3: UML class diagram for the `DeploymentManager` interface.



```
<< interface >>
Connector

copy(src: String,
        dst: String,
        locations: List<String>,
        kind: ConnectorCopyKind,
        soure_remote: Optional<String>)
deploy()
get_available_locations(service: String): List<String>
run(location: String,
      command: List<String>,
      environment: Optional<Map<String, String>>,
      workdir: Optional<String>,
      capture_output: Optional<Boolean>): Any
undeploy()
```

Figure 4.4: UML class diagram for the `Connector` interface.

diagram of this component is reported in Fig. 4.3. Being unique and centralised, it can easily handle multiple concurrent deployment requests, i.e. calls to the `deploy` method, by deploying each model once and returning a pointer to each caller.

Under the hood, the `DeploymentManager` asynchronously offloads the actual orchestration operations to an underlying library through a pluggable implementation of the `Connector` interface, whose UML diagram is shown in Fig. 4.4. This design adheres to the separation of concerns principle, providing an easy way to add support for additional infrastructures if required. Several `Connector` implementations come out of the box with StreamFlow, supporting Docker and Singularity containers, Docker Compose files, Helm charts, SSH-accessible machines, queue-based HPC workload managers (Slurm and PBS), and Occam, the Docker-based supercomputing centre of Università di Torino [126].

As discussed in Sec. 3.2.1, a model should be undeployed as soon as the last task needing it has been completed. This logic is relatively easy to implement when dealing with static coordination models as CWL, but things get more complicated in the dynamic setting. Probably the best strategy for the second case would be to set a grace period, after which the model is undeployed if no new step requires it.

For now, in dynamic scenarios, StreamFlow undeploys all models at the end of the entire workflow execution through the `undeploy_all` method. Moreover, the same method is invoked in case of unrecoverable failures. This approach is very straightforward, but it can lead to resource wastes if some models remain unused for a long time.

Finally, there are some cases in which a workflow step should be executed on a given target location, but the life-cycle of such location is managed externally by an independent orchestration infrastructure. In those cases, StreamFlow can be instructed to skip deployment and undeployment phases for a model by marking it as `external` in the StreamFlow file.

4.1.5 Workflow scheduling

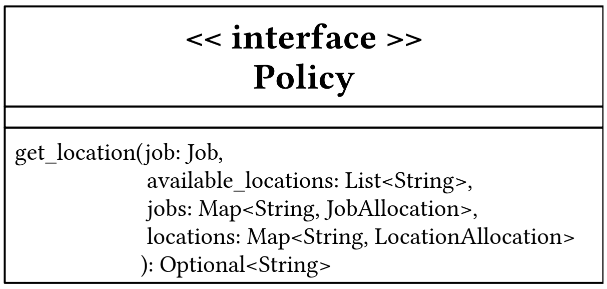
The workflow scheduling strategy is a fundamental component of a WMS, mainly for the significant impact on the overall execution performances. It is common for WMSs to allow users to specify some minimum hardware requirements for a step, e.g., the number of cores or the amount of memory. Such requirements are generally configurable using optional parameters in the coordination language, while the actual mapping on top of adequate worker nodes is left to the specific executor implementation.

It is much easier for a scheduling algorithm to work with *homogeneous* location pools, in which all the nodes have the same characteristics in terms of cores, memory, network and persistence. Nevertheless, different steps can require diverse resources in a real scenario, resulting in sub-optimal workloads for homogenous pools. The case of hybrid workflows is even more complicated. The non-uniform data access makes data locality a crucial aspect in scheduling optimisation. Plus, it is no longer true that a job can be executed on any worker node equipped with enough hardware.

In StreamFlow, the services exposed by each model can be identified as *capabilities*, and a job can be executed on top of it only if all its *requirements* are satisfied. StreamFlow straightforwardly manages this association by identifying each location type with a single service and specifying which service is required by each step (through the `bindings` list described in Sec. 4.1.1).

The model life-cycle is managed by an external orchestration library, so that resources can either be inferred from the environment description file or obtained by querying the orchestrator's control plane. The step-related resource constraints (specified in the workflow description) and the requirement-capability associations (i.e. mapping relations specified in the StreamFlow file) are then directly managed by the `Scheduler` component when selecting the target location.

Even if only a single target service can be specified for each step, multiple replicas of the same service could exist at the same time and, if the underlying orchestrator provides auto-scaling features, their number could also change in



```
get_location(job: Job,
             available_locations: List<String>,
             jobs: Map<String, JobAllocation>,
             locations: Map<String, LocationAllocation>
            ): Optional<String>
```

Figure 4.5: UML class diagram for the `Policy` interface.

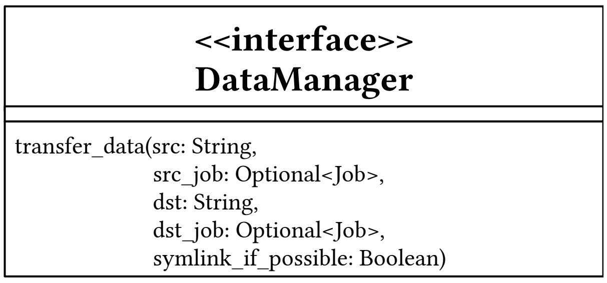
time. Extracting the list of compatible locations for a given step (by calling the `get_available_locations` method of the appropriate `Connector` instance) and applying a scheduling policy to find the best target are `Scheduler` component's responsibilities.

Given the very complex nature of the execution environments managed by StreamFlow, it is improbable that a universally best scheduling strategy exists. Indeed, many factors (e.g. costs, computing power, data locality, load balancing) can affect workflow execution efficiency. For this reason, StreamFlow implements a `Policy` interface to allow users to implement their custom strategies.

As can be seen from the UML class diagram shown in Fig. 4.5, the `Policy` interface only contains a single method, called `get_location`, with four input arguments: the `job` argument, containing a characterisation of the current job in terms of resource requirements and data dependencies; the `available_locations` argument, which is the list of all the locations which satisfy the requirement-capability association for the current step; the list of previously allocated `jobs` and the respective involved `locations`, to support load-balancing features.

The StreamFlow `Scheduler` component processes fireable steps according to a simple FIFO order, with no support for preemption. Moreover, since each scheduling policy can only process one task at a time, all those strategies requiring global knowledge of the tasks queue (e.g. the various flavours of backfilling or a "shortest job first" approach) cannot currently be implemented. Even if this can result in sub-optimal scheduling solutions, the proposed approach drastically reduces the implementation complexity.

A very general scheduling policy, serving as a default strategy, comes out of the box with StreamFlow. When a step becomes fireable, the algorithm iterates over all available locations, starting from those containing at least one of its data dependencies to privilege data locality. Then, it reserves the first one that does not contain jobs in the `running` state and satisfies all the constraints. If the search fails, the task is inserted into a waiting queue: a new scheduling attempt will be performed as soon as a `running` job notifies its termination.



```
transfer_data(src: String,
              src_job: Optional<Job>,
              dst: String,
              dst_job: Optional<Job>,
              symlink_if_possible: Boolean)
```

Figure 4.6: UML class diagram for the `DataManager` interface.

4.1.6 Data transfers

Hybrid workflow executions make it necessary to waive the comfort brought by a globally shared data space, leaving to the WMS the task of explicitly moving the data whenever required. Since large data transfers are very time-consuming operations, especially for long distances and in the absence of dedicated high-throughput communication networks, the WMS should always select the best communication channel between two endpoints and avoid unnecessary data movements.

As mentioned in Sec. 4.1.3, StreamFlow adopts a star topology of deployment locations (see Sec. 3.1.1), i.e. it implements a master-worker pattern for the control plane, where the driver acts as the master [167], [168]. The reader's first impression could be that imposing a star topology is a substantial limitation to the flexibility of a hybrid workflow model. Indeed, in principle, it can describe any topology and statically validate its soundness.

In reality, such topology is much less limiting than it could appear. Despite some potential performance and robustness issues of the centralised control plane, the master-worker is a widely adopted pattern for the runtime support of parallel and distributed systems. Indeed, a centralised 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 [169], [170].

Indeed, the master-worker pattern is adopted by many WMSs and task-based parallel libraries, such as Makeflow [53], Pegasus [42], COMPSs [62], HyperLoom [60], and other well-known systems, such as Kubernetes and Apache Spark [57]. In reality, most of the weaknesses of the master-worker schema can be mitigated with little efforts, such as excluding the master from large data exchanges by allowing direct data movements among workers. This task can be done 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 StreamFlow, where a `DataManager` component, whose UML diagram is in Fig. 4.6, has been designed to orchestrate data transfers, avoid redundant movements, and keep the driver outside the data path whenever possible. In particular, transfers can always be avoided when both tasks run on the same location, but this can also happen when two locations share a data space (e.g. a persistent volume) or if a step explicitly performs a data transfer before completing. The `transfer_data` method investigates the actual need for a transfer by checking if the destination path exists on the target service and computing digests of both source and destination paths. If the destination path does not exist or digests are different, then a data movement is unavoidable.

Plus, two locations in the same model could directly communicate through a channel, removing the need for a double copy operation through the control plane. For example, since all Occam nodes share the `/archive` and `/scratch` portions of the file system, only a local copy on the target location is required to transfer a data dependency in one of such folders. Such optimisations are managed by the `copy` method of the corresponding `Connector` implementation. Conversely, locations belonging to different models are supposed to be independent of each other, so they always communicate through the driver.

Notice that the star topology adopted by StreamFlow is only the upper layer of a *hierarchical* topology. Indeed, the driver location is supposed to communicate directly with each model through a `Connector` interface, but this does not imply that the driver location must directly communicate with every location in a model. Instead, each `Connector` implementation can rely on local communication channels to implement multi-step connections or even nested connections through inner `Connector` instances, e.g., offloading tasks to a Docker container running on top of a VM exposing an SSH port.

Summing up, the only actual limitations of this approach are the absence of inter-model channels and the single point of failure represented by the single driver location. Improvements in both directions are currently under development and represent a crucial plan for the StreamFlow evolution. In particular, additional communication channels can easily be modelled by users in a dedicated section of the StreamFlow file. At the same time, 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.

4.2 Evaluation

The current section describes the experimental evaluation of the StreamFlow approach on two real scientific pipelines in the fields of Bioinformatics and DL. In particular, Sec. 4.2.1 shows how hybrid workflows can free HPC facilities from processing the low demanding steps of an RNA sequencing pipeline, offloading them to cheaper cloud locations without significantly increasing the overall time-to-solution. Then, Sec. 4.2.2 seamlessly offloads a highly parallel and computationally demanding DNN hyperparameter search to a large HPC facility, while the other steps of a complex DL pipeline are processed by a single cloud VM.

4.2.1 Single-cell RNA sequencing

This section showcases the advantages of hybrid workflows' flexibility, allowing users to choose the best target location for each step. The selected use case is a pipeline for single-cell RNA sequencing (scRNA-seq). The idea behind the scRNA-seq technique is to isolate single cells through microfluidic approaches by capturing their transcripts through emulsion droplets loaded with chemical reagents and generating sequencing libraries in which the transcripts are tagged to track their cell of origin. One of the most popular platforms for single-cell analysis, marketed by 10X Genomics, can analyse from 500 to 20,000 cells in each run. Combined with high-throughput sequencing producing billions of reads, scRNA-seq allows the assessment of fundamental biological properties of cells populations and biological systems at unprecedented resolution. The problem with this technique is the noise, which is exaggerated by the need for very high amplification from the small amounts of RNA found in each cell. Denoising the data and estimating an adequate amount of sequencing reads covering each gene in the cell is crucial to define a reliable RNA *count matrix*, representing how many transcripts have been captured for each cell and each gene.

The count matrix creation is performed according to the adopted scRNA-seq experimental technology and the used sequencing approach. For example, consider a typical 10X Genomics experiment followed by an Illumina Novaseq sequencing. The first part of the analysis is performed by a tool called CellRanger [171], which deals with two substeps: the creation of the fastq files (the raw sequences of the four bases, called reads) from the flowcell provided in output by the sequencer and the alignment of the reads against the reference genome, counting for each gene how many reads have been captured. Once the count matrix has been computed, a quantitative analysis of the results is performed: cells with similar transcriptomic profiles are clustered and characterised according to some reference databases. This operation can be performed using ad-hoc software developed in Python or R. This pipeline uses two main R packages to analyse the count matrix: Seurat [172], [173] for normalisation, dimensionality reduction and clustering of cells, and SingleR [174] for labelling the clusters, i.e. identifying the cell type according to public single-cell data annotation databases.

The pipeline relies on a published dataset [175] concerning Gene Editing in Hematopoietic Stem Cell as a test case. In particular, this dataset was produced to compare the efficiency of different gene-editing approaches and, for this reason, the whole experiment is composed of 6 different single-cell samples sequenced independently. This complex experimental design resulted in a particularly challenging and time-consuming dataset, making a flexible, automated and scalable WMS particularly desirable.

This section describes two experiments with two different combinations of Occam and Helm environments. Fig. 4.7 provides a graphical representation of the hybrid workflow model for a single-cell pipeline. The workflow dependency graph is a simple

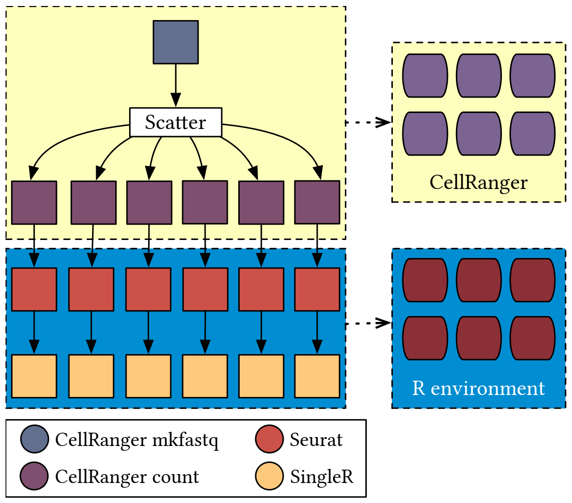


Figure 4.7: Dependency graph and model bindings for the single-cell workflow. In this case, the first step creates six different sequences, which can then be processed independently of each other for the remaining three steps.

DAG with four different steps, while the topology contains two distinct location types (i.e. container images): a CellRanger image for the first two steps and an R image with Seurat, SingleR, and their dependencies for the last two steps.

The first step produces six outputs, which can be processed independently by the rest of the pipeline. In CWL, this can be easily implemented using the `scatter` directive, which generates six jobs for each related step. Since these jobs cannot be executed in a distributed fashion, the maximum number of nodes from which the workflow execution can take some benefit is equal to the fan-out of the initial parallel split. Therefore, if enough hardware resources are available, the best strategy would be to allocate six locations of each type, implementing an MTSC mapping (see Sec. 2.4.2).

A hybrid workflow model can be beneficial to perform a data preprocessing phase on a dedicated HPC structure before moving data to the cloud to complete the remaining steps. Indeed, in the examined case, the total size of the initial data is almost 60GB, but modern sequencing machines can achieve 10 billion sequences per flowcell, corresponding to about 3TB of data. Plus, the `cellranger count` command requires a pretty high amount of resources to be performed: the official documentation reports 8 cores and 32GB of memory as minimum requirements, but a significant speedup can be appreciated until up to 32 cores and 128GB of memory.

Without hybrid workflows, the best strategy would be to execute the entire set of tasks on top of six HPC nodes to take full advantage of the available grade of parallelism while avoiding data transfers. Moreover, this solution is better than hybrid alternatives when using total wall clock time as the only evaluation metric. Therefore, it is worth using this setting as a baseline to evaluate the significance of

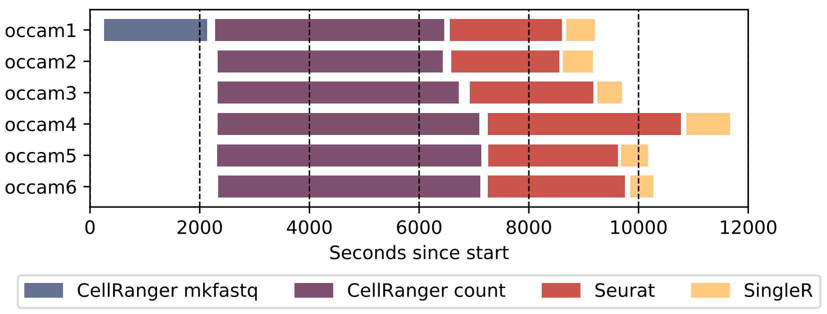


Figure 4.8: Execution timeline for the StreamFlow single-cell application on six Occam nodes, each allocated to both a CellRanger and an R environment containers.

performance loss when switching to a mixed cloud-HPC configuration.

The first experimental setting reserves six Light nodes on the Occam facility, each of which having 2x Intel Xeon E5-2680 v3 (12 core each, 2.5GHz) CPUs and 128GB (8x16, 2133MHz) of memory, and allocates each node to both a CellRanger and an R environment containers. An additional Occam node has been reserved to the StreamFlow control plane. This architecture is described by a topology containing a single model with two services (the two Docker images), each of which exposes six locations. As mentioned in Sec. 4.1.6, all Occam nodes share the `/archive` folder, mounted as an NFS export, and the `/scratch` folder, with a LUSTRE parallel file system. The input data of the pipeline have been manually copied on the `/archive` file system, and StreamFlow has been configured to use a folder on the `/scratch` hierarchy as its output folder so that no automatic data transfers are needed. Fig. 4.8 shows the timeline for this execution. Its whole duration is more or less 3h15m, dominated by the CellRanger count and Seurat steps. White space between subsequent bars represents the time needed by StreamFlow to perform internal tasks, which is negligible compared to the time needed to complete the workflow steps.

The second experimental setting dedicates the HPC structure to the first two steps, offloading the rest of the workflow to a cloud environment. This configuration makes sense for three reasons. First of all, the third and fourth steps need less computing power, and they strongly underexploit the computing resources available in an HPC facility. Plus, the outputs of the last step of a pipeline must often be stored in a database or visualised in a web application, and the cloud is undoubtedly the most natural place to host these services. Finally, by observing intermediate data in the workflow model, it is possible to notice that output data of the second step have a total size of about 15-30MB, allowing to transfer them to a remote infrastructure without introducing significant overhead.

In this setting, the third and fourth steps of the pipeline are offloaded to a

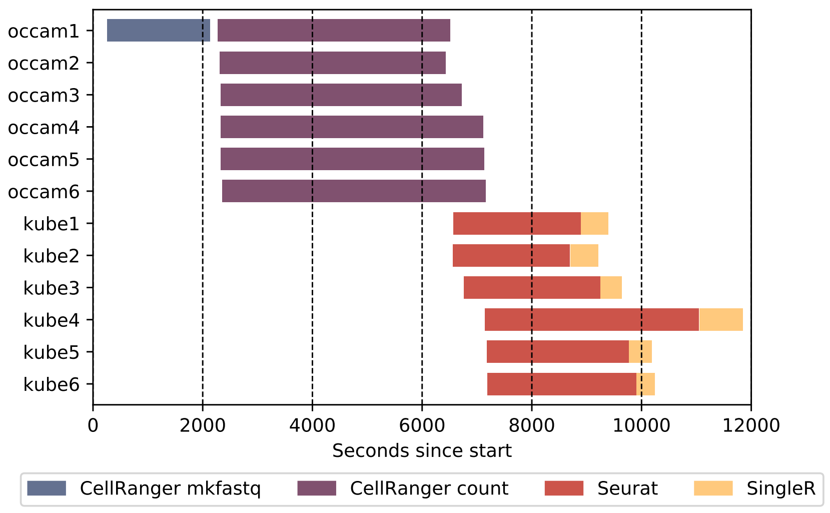


Figure 4.9: Execution timeline for the StreamFlow single-cell application in a hybrid configuration, with six Occam nodes allocated to CellRanger as many replicas and six Kubernetes worker nodes allocated to as many R environment containers.

virtualised Kubernetes cluster running on top of the GARR[6] cloud, based on OpenStack[7], containing six worker nodes with 4 virtual CPUs and 8GB of memory each. The related location topology contains two different models: the first one with six Occam nodes, with an instance of the CellRanger container allocated on each of them, and the second one with six Kubernetes Pods, each with an instance of the R environment container and a `podAntiAffinity` parameter to ensure that each Pod is allocated on a different worker node. Note that there is no need to modify the CWL description of the workflow to run it on the new environment: changes only involve model descriptions and the `streamflow.yml` file.

On Kubernetes, the StreamFlow output folder of each container has been mapped onto a persistent volume managed by Cinder, the OpenStack's block storage service, configured with a `readWriteOnce` access mode. Therefore, no shared data space exists between different worker nodes, but the scheduling policy described in Sec. 4.1.5 ensures that each SingleR task is executed by the node where its required input data already reside, removing the need for data transfers. Since the StreamFlow control plane still runs inside an Occam node, the only unavoidable data movement is from Occam to Kubernetes, between the second and the third steps.

The timeline for this second run is reported in Fig. 4.9. The first important thing to observe is how the whole duration of this hybrid execution is comparable with the previous full-HPC configuration. This result is mainly due to the combination

[6]https://garr.it/it/

[7]https://www.openstack.org/

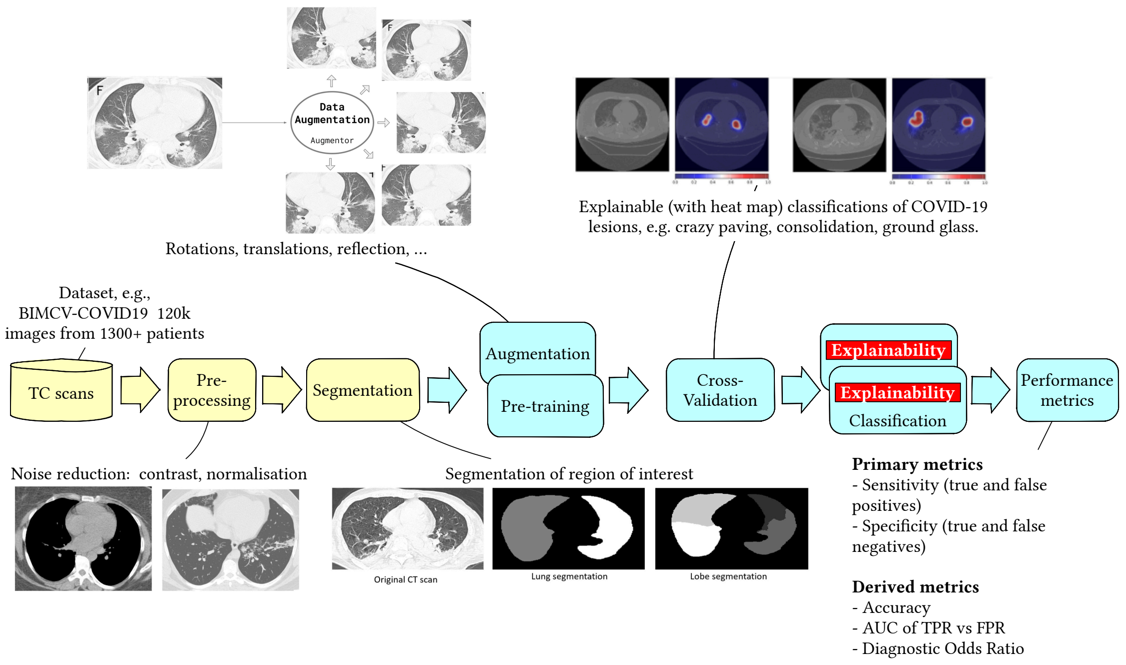

Data Augmentation
Augmentor
Rotations, translations, reflection, ...
Explainable (with heat map) classifications of COVID-19 lesions, e.g. crazy paving, consolidation, ground glass.
Dataset, e.g., BIMCV-COVID19 120k images from 1300+ patients
TC scans
Pre-processing
Segmentation
Augmentation
Pre-training
Cross-Validation
Explainability
Explainability
Classification
Performance metrics
Noise reduction: contrast, normalisation
Segmentation of region of interest
Original CT scan
Lung segmentation
Lobe segmentation
Primary metrics
- Sensitivity (true and false positives)
- Specificity (true and false negatives)
Derived metrics
- Accuracy
- AUC of TPR vs FPR
- Diagnostic Odds Ratio

Figure 4.10: The CLAIRE COVID-19 universal pipeline. Yellow steps belong to the data preparation phase of the pipeline, which is executed only once per dataset. Blue steps belong to the core training phase, which is repeated multiple times for each hyperparameters configuration to implement cross-validation.

of two factors. Firstly, the time needed to transfer data from the Occam facility to the GARR cloud is negligible compared to the time needed to complete the steps themselves. Moreover, the Seurat task seems not to benefit from additional computing power, making it quite useless to commit HPC machines for its execution. In a situation like this, it is pretty clear that the hybrid workflows approach can be beneficial to obtain a more efficient resource allocation without significant performance drops.

4.2.2 The CLAIRE COVID-19 universal pipeline

This section explores how StreamFlow can help bridge HPC and DL workloads. The related use case is the CLAIRE COVID-19 universal pipeline [176], [177], sketched in Fig. 4.10. When the COVID-19 pandemic broke out, among the initiatives aimed at improving the knowledge of the virus, containing its diffusion, and limiting its effects, the Confederation of Laboratories for Artificial Intelligence Research in Europe (CLAIRE)[8] task force on AI & COVID-19 supported the set up of a novel European group to study the AI-assisted diagnosis of COVID-19 pneumonia [178].

At the pandemic's start, several studies outlined the effectiveness of radiology

[8]https://https://claire-ai.org/

imaging for COVID-19 diagnosis through chest X-Ray and mainly CT, given the pulmonary involvement in subjects affected by the infection. Even if X-Ray scans represent a cheaper and most effective solution for large-scale screening, their low resolution led DL models to show lower accuracy than those obtained with CT data. Consequently, CT scans have become the gold standard for the investigation of lung diseases. Several research groups worldwide began to develop DL models for diagnosing COVID-19, mainly in the form of deep CNNs, applying lung disease analysis from CT scan images.

Despite the large number of proposed models and techniques, different and not comparable architectures, pipelines and datasets make it impossible to select the most promising ones. Trying to solve this problem, the CLAIRE task force on AI & COVID-19 started working on the definition of a reproducible workflow capable of automating the comparison of state-of-the-art DL models to diagnose COVID-19. This workflow subsequently evolved towards the CLAIRE COVID-19 universal pipeline, composed of two main parts:

- a *data preparation* phase (yellow blocks in Fig. 4.10), comprising *pre-processing*, where standard techniques for cleaning the training images are applied, and *segmentation*, for extracting and selecting the region of interest through an autoencoder DNN (e.g. DeepLabV3 [179], U-Net [180], or Tiramisu [181]) to improve the quality of the data. This phase is performed once for each dataset;

- a *core training* phase (blue blocks in Fig. 4.10), composed of standard DL steps such as *data augmentation*, to generate image variants, model *pre-training*, to generate an initial set of weights for initialisation, and eventually *classification*, during which a CNN (e.g. GoogLeNet [182], AlexNet [183], ResNet [184], DenseNet [185], or Inception-ResNet [186]) labels each image with a class. In this setting, each class is identified with a kind of lesion typical of the disease. This phase is performed once per hyperparameters configuration.

The pipeline robustness is further increased by applying *cross-validation* to the classification step, i.e. repeating the training process on different portions of the dataset. *Performance metrics* are then obtained by collecting and averaging all the measures from all the trained instances.

The universal pipeline aims to analyse some of the best DNNs in the literature, together with a systematic exploration of networks hyperparameters, allowing a deeper search for the best model. Each of these configurations generates a different, independent variant of the pipeline. The resulting number of the CLAIRE COVID-19 pipelines variants is 990. Exploring the entire spectrum of variants requires two non-trivial ingredients: a supercomputer of adequate computational power, equipped with many latest generation GPUs, and a mechanism capable of unifying and automating the execution of all variants of the workflow on a supercomputer.

The input of the universal pipeline is the most extensive dataset publicly available

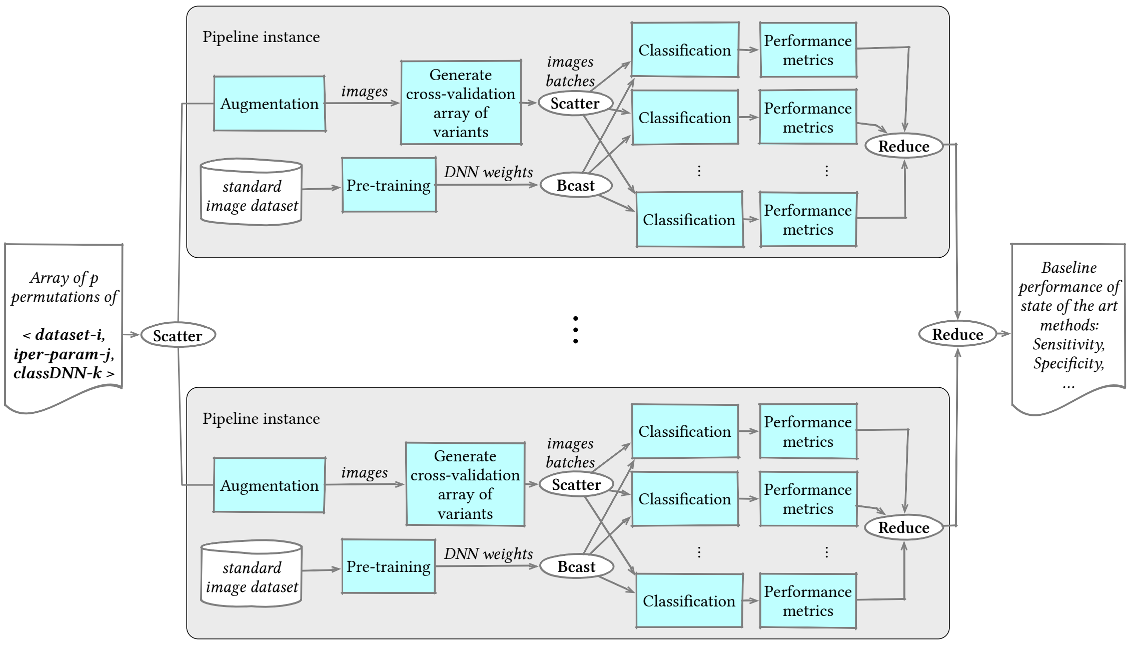

Pipeline instance
Augmentation
images
Generate cross-validation array of variants
images batches
Scatter
Classification
Performance metrics
Reduce
standard image dataset
Pre-training
DNN weights
Bcast
Array of p permutations of
< dataset-i, iper-param-j, classDNN-k >
Scatter
Reduce
Baseline performance of state of the art methods: Sensitivity, Specificity, ...

Figure 4.11: Unfolded core training phase of the CLAIRE-COVID19 universal pipeline.

related to COVID-19's pathology course, i.e. the BIMCV-COVID19+ dataset[9] [187], with more than 120k images from 1300 patients. Supposing to train each pre-trained model for 20 epochs on such dataset, a single variant of the pipeline takes over 15 hours on a single NVidia V100 GPU, one of the most powerful accelerators in the market. Therefore, exploring all the 990 pipeline variants would take over two years on the most powerful GPU currently available.

Since the universal pipeline has an embarrassingly parallel structure (see Fig. 4.11), using a supercomputer can reduce the execution time to one day. In the best case, running all the variants concurrently on 990 different V100 GPUs only takes 15 hours of wall-clock time. Nevertheless, pre-training and post-training steps like performance metrics extraction and comparison can safely run locally on the practitioner's desktop machine or a cloud-hosted VM, as they do not require much computing power. Therefore, the optimal execution of this pipeline advocates a hybrid workflow model.

In practice, for this experiment, the StreamFlow control plane has been launched on a cloud-based host node. The entire data preparation phase and the post-training steps have been executed locally, while different portions of the training spectrum have been offloaded to three heterogeneous architectures:

- the ENEA CRESCO D.A.V.I.D.E. HPC cluster, composed of 45 nodes with

[9] https://bimcv.cipf.es/bimcv-projects/bimcv-covid19/

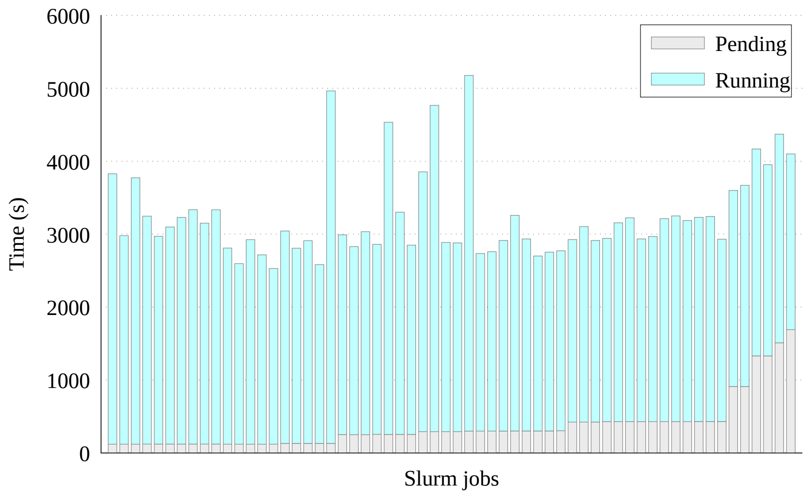


Figure 4.12: Execution times of the Classification step for 60 variants of DenseNet-121 on top of the CINECA MARCONI100 cluster.

2 IBM POWER8 sockets, 256 GB of RAM and 4 NVidia P100-SMX2 GPUs each, provided on-demand through bare SSH connections;

- the CINECA MARCONI100 cluster, a Slurm-managed HPC facility with 32 IBM POWER9 cores, 256 GB of RAM and 4 NVidia V100 GPUs per node;
- the High-Performance Computing for Artificial Intelligence (HPC4AI) infrastructure at Università di Torino, a multi-tenant cloud-HPC system with 80 cores and 4 GPUs per node, managed by OpenStack [25].

As an interface towards cloud-HPC infrastructures, StreamFlow seamlessly managed data movements and remote step execution with each of these infrastructures, automatically transferring the training results to the control node and rescheduling failed jobs through its fault tolerance layer.

Fig. 4.12 shows the time taken to train in parallel 60 DenseNet-121 models with different hyperparameters on top of MARCONI100. Each training has been configured to run for 50 epochs, with early stopping after 10 epochs without accuracy improvements. Detailed information on the experiment's results in terms of DNN accuracy are available elsewhere [188]. Excluding the queueing times, training the 60 models on a single V100 GPU would have required two days, while MARCONI100 only took slightly more than 80 minutes to complete the entire step. Scheduling the entire workload on the Cloud would have been highly costly: reserving a `p2.16xlarge` EC2 instance with 16 V100 GPUs on AWS costs 14.4$/hour. Moreover, note that 60 models constitute only a tiny portion of the solution space covered by the CLAIRE COVID-19 pipeline. In this scenario, hybrid workflows can significantly drop the total cost without affecting the overall performance.

Chapter 5

Distributed literate workflows

Despite all the advantages of scientific WMSs in modularity, portability, and reproducibility, domain experts often prefer to stick with standard, general-purpose languages to develop and publish their experiments. Some apparent reasons behind this choice are the additional effort required to learn a new framework, the increased difficulty in maintaining coherence between host and coordination logic, and the lack of a de-facto standard WMS that everyone should know and use.

In order to overcome this problem, this thesis introduces *literate workflows* as a new paradigm, able to interleave host and coordination logic in the same document but at the same time to keep them separated. The name derives from the concept of *literate computing* (see Sec. 2.5), whose primary implementation nowadays is the Jupyter stack. Several attempts have been made to model workflows with Jupyter Notebooks (see Sec. 2.5.2), but as far as the author knows, this thesis is the first attempt to derive a general methodology.

A computational notebook is essentially a list of code cells executed sequentially in a given order (Sec. 5.1). The idea is to treat each cell as a workflow step, using the related metadata to express input and output dependencies. A workflow DAG can be extracted from this representation, and independent steps can be executed concurrently (Sec. 5.2). Plus, notebook metadata can describe topologies of deployment locations, and each cell can be mapped onto a different location, modelling hybrid literate workflows that can be executed in a distributed fashion (Sec. 5.3).

5.1 Literate computing semantics

A computational notebook can be seen as an ordered list of cells, each containing code, code executions output, or natural language documentation. In this work, code cells are treated as the *atomic execution units* of a notebook. This assumption is valid also for actual implementations whenever a cell ends successfully. Conversely, documentation and output cells are ignored, as they do not affect the notebook's operational semantics. Some implementations could allow documentation cells to

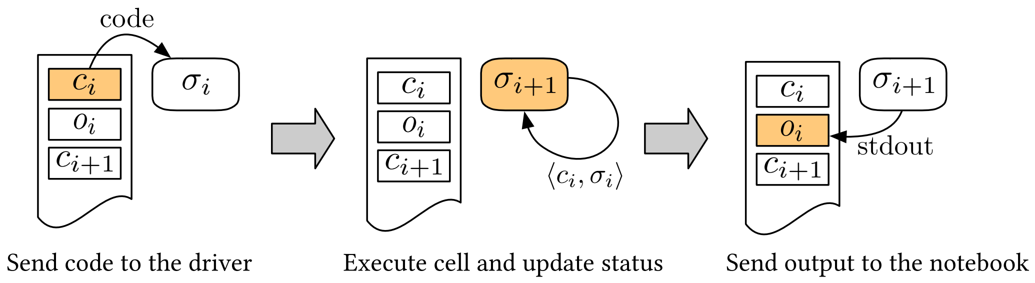

code
c_i
o_i
c_{i+1}
σ_i
σ_{i+1}
$\langle c_i, \sigma_i \rangle$
stdout
Send code to the driver
Execute cell and update status
Send output to the notebook

Figure 5.1: Execution steps of a cell c_i in a computational notebook. Its code is sent to the driver process, which updates its state from σ_i to σ_{i+1} and returns the execution stdout to the output cell o_i for visualisation.

reference variables from the notebook state, but the following discussion can easily be extended to include such cases.

Let then a notebook $\boldsymbol{N}$ be composed of a sequence of code cells $c_1, \ldots, c_n$, where every c_i is, in turn, a sequence of instructions in the notebook's host language. Each cell is executed in a *state*, a partial mapping $\sigma : \text{Ide} \rightarrow \text{Obj}$ from host language identifiers to first-class objects (e.g. values, expressions, functions). In most implementations, the state is preserved across subsequent cell executions, and a unique global state exists at a given point in time. A dedicated *driver* process (e.g. the kernel in Jupyter Notebooks or the interpreter in Apache Zeppelin) manages that state, guaranteeing a *unique total order* of cells executions.

Fig. 5.1 depicts the steps of a cell execution. Let $In(c)$ and $Out(c)$ be the sets of identifiers read and written by the instructions of cell c, respectively. By representing a cell waiting to be evaluated as a *configuration* $\langle c, \sigma \rangle$, s.t. $In(c) \subset \text{dom}(\sigma)$, the *execution relation*

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

expresses that the execution of cell c in state σ produces a new state σ', s.t. $Out(c) \subset \text{dom}(\sigma')$. A cell execution can also produce something on the driver's `stdout`, which is flushed back to the notebook to be visualised in the related output cell o. With this formalism, if $\langle c_i, \sigma_i \rangle \rightarrow \sigma_{i+1}$ for every $i \in [1, n]$, the execution of a sequence $c_1; \ldots; c_n$ of cells can be written as follows

$$\langle c_1; \ldots; c_n, \sigma_1 \rangle \rightarrow \sigma_{n+1} \tag{5.1}$$

so that each cell in the sequence is executed in the state produced by the previous one.

Computational notebooks usually support two different execution modes: *interactive execution* and *bulk execution*. In both cases, the execution of cells is sequential, and each cell is executed in the state resulting from the execution of the last cell in temporal order, as described in Eq. (5.1). For bulk mode, which executes all the code cells in the same order in which they appear in the notebook, the execution order specified by the operator ‘;’ is well defined. Conversely, the interactive execution mode allows users to execute cells repeatedly, out-of-order, or even ignore some of them. Nevertheless, given that a unique total order of cell executions is always guaranteed by hypothesis, it is always possible to treat the history of cell executions as a (potentially infinite) notebook executed in bulk mode, recovering Eq. (5.1).

Requiring a total execution order for cells contrasts with the flexibility of standard workflow abstractions, where independent steps can always run concurrently. Sec. 5.2 introduces a methodology to relax this constraint while preserving consistency with a fully sequential execution.

5.2 Literate workflows semantics

The bulk execution mode enables a concurrent distributed execution of the cells by defining *sequentially equivalent* parallel semantics. Since the cells $c_1, \dots, c_n$ are totally ordered by their position in the notebook, the control-flow graph is the linear chain of cells. Given that, it is possible to statically compute the execution graph with the highest degree of parallelism by Bernstein’s conditions [189], initially designed for parallelising compilers. Such conditions describe the three cases that induce data dependencies between pairs of cells, which prevent their parallel execution. When $i < j$, c_j *depends on* c_i if at least one of the following conditions holds:

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

In all other cases, namely when

$$\mathit{Out}(c_i) \cap \mathit{In}(c_j) = \mathit{Out}(c_i) \cap \mathit{Out}(c_j) = \mathit{In}(c_i) \cap \mathit{Out}(c_j) = \emptyset$$

the cells c_i and c_j do not interfere and can be executed in any order, hence also in parallel.

As discussed below, this work relaxes 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 produces an output on the initial state of the notebook, where indeed the identifier representing the standard output conflicts. The proposed relaxation aims at preserving the output of all cells, which in the sequential execution involve the same identifier (the notebook’s output), but in different moments.

First, it is necessary to know the sets $\mathit{In}(c)$ and $\mathit{Out}(c)$ for all the cells $c \in \boldsymbol{N}$ to evaluate Bernstein’s conditions. An interpreter can automatically extract these sets from the cell's code, akin to what noWorkflow does for scripts [70], or a user can explicitly list their members in the cell's metadata, similarly to the yesWorkflow approach [67]. In any case, this discussion assumes $\mathit{In}(c)$ and $\mathit{Out}(c)$ to list all input and output dependencies for cell c correctly.

In preparation for their parallel execution, all notebook cells are arranged in a DAG from the control-flow chain, defining a workflow in which nodes are configurations and edges are data dependencies. Let *DAG evaluation* be the order derived from applying data dependencies. A DAG evaluation sequence can then be described as a composition of atomic cells c using ';' (sequential composition) and '|' (parallel composition). For example,

$$c_1; (c_2 \mid (c_3; c_4))$$

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

DAG evaluation supports the automatic parallelisation of independent cells. In particular, given a notebook $\boldsymbol{N} = [c_1, \ldots, c_n]$, a simple strategy to obtain a valid DAG evaluation consists in connecting c_i to c_j, with $i < j$, whenever $\exists x \in \mathit{In}(c_j)$ s.t. $x \in \mathit{Out}(c_i)$ and $x \notin \mathit{Out}(c_k)$ for each $k \in (i, j)$. It is possible to prove by induction that such strategy always complies with the Bernstein's conditions.

Theorem 5.2.1. *Given a notebook $\boldsymbol{N} = [c_1, \ldots, c_n]$, the DAG obtained by connecting c_i to c_j, with $i < j$, whenever $\exists x \in \mathit{In}(c_j)$ s.t. $x \in \mathit{Out}(c_i)$ and $x \notin \mathit{Out}(c_k)$ for each $k \in (i, j)$ preserves sequential consistency.*

Proof. Sequential consistency is guaranteed by the validity of Bernstein's conditions. True data dependencies are trivially preserved by the strategy, as it explicitly adds a link $c_i \to c_j$ whenever $\exists x \in \mathit{Out}(c_i) \cap \mathit{In}(c_j)$. Output and anti-dependencies are instead preserved by creating a separate context σ_i for each cell c_i. Anti-dependencies can then be ignored, since if there exists an $x \in \mathit{In}(c_i) \cap \mathit{Out}(c_j)$, the cell c_i will still receive $\sigma_h(x)$ with $h < i$, and any cell c_l with $l > j$ will receive $\sigma_k(x)$ with $k \geq j$, independently of the actual order of execution between c_i and c_j. The discussion for output dependencies is similar, but in addition, the strategy ensures that if $x \in \mathit{Out}(c_i) \cap \mathit{Out}(c_k) \cap \mathit{In}(c_j)$, with $i < k < j$, then the cell c_j always receives $\sigma_k(x)$, independently of the actual order of execution between c_i and c_k. □

This strategy is effective when outputs of subsequent cells are alternative to each other, i.e. whenever $x \in \mathit{Out}(c_i) \cap \mathit{Out}(c_j)$ means that $\sigma_j(x)$ overwrites the previous value. Still, all those cases when the resulting x is a combination of the two values require explicitly putting x into $\mathit{In}(c_j)$, inducing a true data dependency between the cells. A typical case is the `stdout`, which is populated according to the total execution order of cells in the sequential case, but in the parallel case would induce a true data dependency between any pair of cells. In order to overcome this problem, cells are assumed independent 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

$$\langle c_1 \mid \cdots \mid c_n, \sigma\rangle \to \biguplus_{1\leq i\leq n} \sigma'_i \quad \text{if } \langle c_i, \sigma_i\rangle \to \sigma'_i \;\; \forall i \in [1, n] \tag{5.2}$$

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

$$(\sigma \uplus \sigma')(x) = \begin{cases} \sigma(x) & \text{if } x \in \mathrm{dom}(\sigma) \setminus \mathrm{dom}(\sigma') \\ \sigma'(x) & \text{if } x \in \mathrm{dom}(\sigma') \setminus \mathrm{dom}(\sigma) \\ \sigma(x) \uplus \sigma'(x) & \text{if } x \in \mathrm{dom}(\sigma) \cap \mathrm{dom}(\sigma') \end{cases} \tag{5.3}$$

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 parallelisation accumulation behaviour. 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 behaviour is inspired by merging `stdout` of remotely executing processes aiming to preserve single-cell outputs rather than overwriting them.

Theorem 5.2.2. *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 true data dependency cannot be executed in parallel, and anti-dependencies can be ignored, as discussed while proving Theorem 5.2.1. 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, as per Eq. (5.3), and all the conflicting identifiers of σ_1 and σ_2 computed by the user-defined associative operator $\uplus$. The existence and 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. Since $\uplus$ is an associative operator, the same argument scales to the transitive closure of the merged state of a sequence of cells executed in parallel, including 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, the parallel execution is considered sequentially equivalent. □

The DAG evaluation can also support the data parallelism paradigm – the *Map*/*ApplyToAll* functions – by way of an *explicit* metadata annotation on a cell c

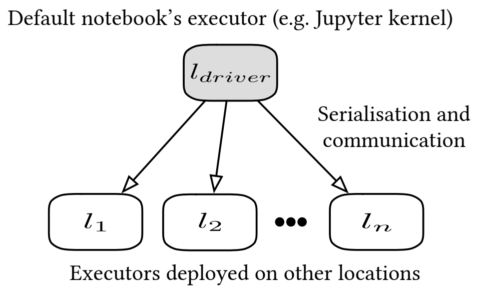

Default notebook's executor (e.g. Jupyter kernel)
l_{driver}
Serialisation and communication
l_1
l_2
l_n
Executors deployed on other locations

Figure 5.2: Runtime architecture of a computational notebook's distributed execution under a star location topology.

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, such cells are denoted as $Map(c)$.

Consider, for instance, a Scatter/Gather pattern. The semantics of a *Map* cell depend on two (here unspecified) functions: *Scatter*, which splits states into lists of states, and *Gather*, which recombines lists of states into states. The execution semantics of *Map* cells can then be described as follows

$$\langle Map(c), \sigma \rangle \rightarrow Gather(\sigma'_1, \dots, \sigma'_n) \quad \text{if } \langle c_1, \sigma_i \rangle \rightarrow \sigma'_i \;\; \forall i \in [1, n] \tag{5.4}$$

where $Scatter(\sigma) = [\sigma_1, \dots, \sigma_n]$. If two cells $Map(c_1)$ and $Map(c_2)$ are executed sequentially in bulk mode and $Out(c_1) = In(c_2)$, it is possible to rely on a *map fusion* transformation [190] to reduce the communication overhead, rewriting each sequence of type *Scatter*/*Gather*/*Scatter*/*Gather* as *Scatter*/*LocalCopy*/*Gather*. This kind of optimisation can be expressed as the following equivalence between configurations:

$$\langle Map(c_1); Map(c_2), \sigma \rangle = \langle Map(c_1; c_2), \sigma \rangle \tag{5.5}$$

Theorem 5.2.3. *The Map function preserves sequential equivalence of successfully terminating parallel executions.*

Proof. The *Map* function generates a list of independent replicas of a cell that satisfy Bernstein's conditions by construction. □

5.3 Hybrid literate workflows

In order to model the distributed execution of hybrid workflows, configurations $\langle c, \sigma \rangle$ can be extended to *global configurations* $\langle [c, l], \sigma \rangle$, where the l component indicates the location in which the execution of c takes place. The *global execution relation* can then be specified as

$$\langle [c_1, l_1]; \dots ; [c_n, l_n], \sigma_1 \rangle \rightarrow \sigma_{n+1} \quad \text{if } \langle [c_i, l_i], \sigma_i \rangle \rightarrow \sigma_{i+1} \;\; \forall i \in [1, n] \tag{5.6}$$

Eq. (5.6) has the same meaning as the Eq. (5.1), except that it carries the additional information that a cell c_i is deployed and executed on $l_i \in L$, where L is the set of locations in a hybrid workflow model. Assuming a star topology, Eq. (5.1) is equivalent to Eq. (5.6) when $l_i = l_{driver}$ for every $1 \leq i \leq n$, where l_{driver} refers to the default notebook's kernel (see Fig. 5.2). The binding $(c_i, l_i) \in B$ between a cell c_i and a location l_i can be directly specified in the metadata field of cell c_i itself or determined through more complex policies, as discussed in Sec. 3.3.1.

Just like the local execution relation, also the global execution relation is partial. However, the remote execution of a cell is assumed to be independent of the location in which the execution takes place, i.e. $\langle [c_i, l_i], \sigma \rangle \rightarrow \sigma'$ and $\langle [c_i, l_j], \sigma \rangle \rightarrow \sigma''$ implies $\sigma' = \sigma''$. This means that 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. Such assumption guarantees deterministic semantics of global execution, modulo errors, and it allows to straightforwardly extend all the properties of DAG evaluation discussed in Sec. 5.2 to the global case. Since the model resulting from DAG evaluation is a deterministic acyclic graph, it is always possible to obtain an execution plan (see Sec. 3.2.1) and apply static soundness analysis to the related hybrid workflow model (see Sec. 3.2.2).

Chapter 6

Jupyter-workflow

Jupyter-workflow[1] extends the IPython software stack to support *hybrid literate workflows* [15] (see Chapter 5). In particular, the code cells of a Jupyter Notebook are interpreted as the steps of a workflow DAG, while metadata encode dependencies between cells, topologies of deployment locations and mapping relations. Even if Jupyter-workflow is limited to IPython-compatible programs, the metadata format is general enough to be reused with any kernel in the Jupyter ecosystem or with other computational notebooks frameworks, e.g. Apache Zeppelin. The current chapter details the Jupyter-workflow implementation (Sec. 6.1) and evaluates it on large-scale HPC and cloud environments with four practical applications in the domains of DL, scientific simulation and Bioinformatics (Sec. 6.2).

6.1 Implementation

The Jupyter-workflow logical architecture, reported in Fig. 6.1, consists of three main components:

- a *coordination metadata format* to model global cells configurations and location topologies (see Sec. 6.1.1);
- a *dependency resolver* component to help users identify the input dependencies of each cell (see Sec. 6.1.2);
- a *Jupyter stack extension* to handle coordination metadata, execute cells remotely and manage data transfers (see Sec. 6.1.3).

Moreover, Jupyter-workflow relies on the dill library [191] to perform data serialisation (see Sec. 6.1.4) and the StreamFlow framework to coordinate hybrid workflows (see Chapter 4). The rest of the current section is devoted to a detailed analysis of each component, discussing the most significant design and implementation choices.

[1] `https://github.com/alpha-unito/jupyter-workflow`

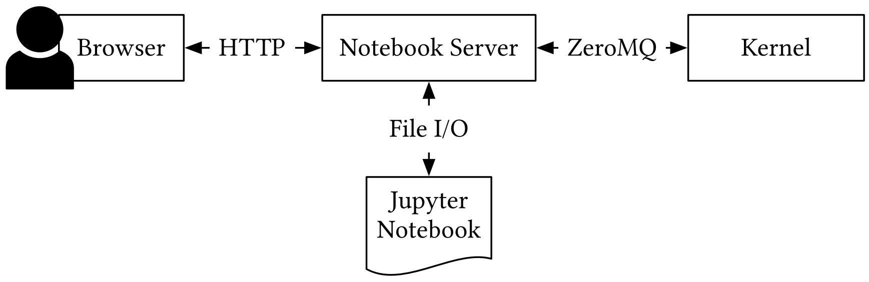


Figure 6.1: Jupyter-workflow logical stack. White blocks refer to existing technologies (except for StreamFlow and its connectors, which are coloured in blue), while yellow-ocher ones are directly part of the Jupyter-workflow codebase.

Listing 6.1: 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
      "deployment": {# Description of the execution environment
          ...
      },
      "service": "target service inside the model",
      "locations": "number of workers to reserve"
  },
  "version": "v1.0"
}
```

6.1.1 Coordination metadata format

In the Jupyter Notebook format, each cell is accompanied by a `metadata` field containing custom values (see Listing 2.1). Jupyter-workflow extends the code cell metadata format with a `workflow` section to express the dataflow dependencies, i.e. the set $In(c_i)$ of input dependencies and the set $Out(c_i)$ of return values, and the location binding (c_i, l_i) for each cell c_i. This approach is similar to the one adopted by YesWorkflow [67], but the main difference is that, in Jupyter-workflow, the dataflow description is separated from the host code so that the same format can be used in combination with any underlying kernel. A high-level schema of the `workflow` section is reported in Listing 6.1.

A `step` subsection contains two lists, called `in` and `out`, describing input dependencies and return values, respectively. Jupyter-workflow 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 Jupyter-workflow automatically manages both data transfers and path remappings on the target location;
- *controls*, which are only used to force additional dependency relations between workflow steps without carrying any data value.

Note that control dependencies are necessary whenever a cell updates the state of an external location, e.g. inserts a record into a database, and this update does not modify any variable in the program state.

The cell-location binding is expressed in a `target` subsection, while locations are described in a `model` subsection. Both of them are exact transpositions of their corresponding StreamFlow directive (see Sec. 4.1.1), as well as the distinction between units of deployment, binding, and scheduling. Iterable cell inputs, e.g. lists or dictionaries, can be scattered across multiple locations for parallel execution. Scattering schemes can be specified through a dedicated `scatter` section in the `step` metadata (Listing 6.1). In particular, an `items` list contains the elements to scatter, while the `method` entry specifies which operator (e.g. `dotproduct` or `cartesian`) should be used when scattering over multiple `items`. The `items` list can contain names, file paths, or nested scatter schemes, providing great flexibility in modelling complex parallel patterns. Rewriting rules can optimise the execution plan by removing unnecessary directives (see Sec. 5.2).

6.1.2 The DependencyResolver component

In many cases, input dependencies can be automatically inferred with an inspection of the cell code. Therefore, drawing inspiration from noWorkflow [70], Jupyter-workflow includes 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 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 that reside in its global scope, whose first operation is a `Load` (i.e. a read from σ), and do not come from Python builtins or IPython standard namespace. The fact that the dill serialisation library can autonomously deal with transitive dependencies dramatically simplifies this task, as it is not necessary 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. On the one hand, each name that does not appear in a node of the AST representation returned by the `ast` module cannot be recognised by the `DependencyResolver` component. This set contains, among others, variables dynamically loaded in `eval` constructs, variables accessed directly from the `locals` and `globals` dictionaries and modules dynamically imported by the `importlib` package. On the other hand, as the code is 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, Jupyter-workflow lets users combine automatic dependency induction with explicit metadata to correct potentially wrong behaviours, print the list of automatically identified dependencies, or even fully disable the `DependencyResolver` for a step by setting the `autoin` field to `False`.

6.1.3 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 this 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[2] technology needs a kernel-agnostic frontend plugin, which is currently under development.

When receiving coordination metadata, the kernel backend must correctly process them during both interactive and bulk execution flows. The current Jupyter-workflow implementation only extends the IPython kernel. Indeed, it is the most widely used backend in the Jupyter ecosystem and, since StreamFlow is also implemented in Python, the integration with the underlying WMS layer is much more manageable. However, support for other Jupyter kernels is undoubtedly in plans, and most aspects discussed in this section are still perfectly applicable to a language other than Python. 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 DAG (see Sec. 5.2). The resulting DAG can then be orchestrated by the StreamFlow runtime support, whose

[2] `https://jupyterlab.readthedocs.io/en/stable/`

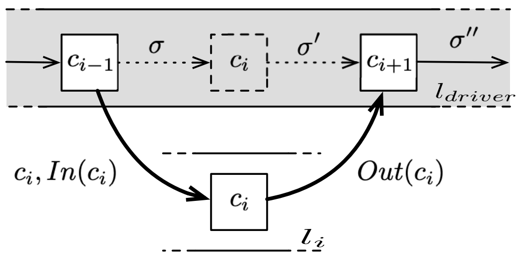


Figure 6.2: Interactive remote execution of a configuration $\langle [c_i, l_s], \sigma \rangle$. When $l_s \neq l_{driver}$, both code c_i and true data dependencies $In(c_i)$ must be transferred to l_s. After that, the return values $Out(c_i)$ can be transferred back to l_{driver}.

control plane runs in the same process as the default Notebook kernel. In particular, when cell c_i enters the fireable state, its code and input dependencies $In(c_i)$ are serialised and transferred to its bound location l_s so that the program state can be reconstructed from them (see Fig. 6.2). StreamFlow manages all the computation movement aspects, i.e., data transfer, path remapping, locations deployment, and task scheduling. In particular, an `executor` script, automatically transferred to each remote location, is in charge of recreating the program state, executing the code, and serialising the return values.

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 the cell execution order cannot be determined a priori in an interactive scenario, all the components of $Out(c_i)$ are always transferred to the local kernel and merged into the program state after the execution of cell c_i.

6.1.4 Serialisation

When dealing with computation movement, serialisation 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 unserialisable 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 serialiser capable of producing a suitable byte stream for every object and pair of locations 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 optimisations. Things worsen when the source and destination locations exhibit significant differences in operating systems or hardware architectures.

A possible approach to mitigate this kind of problem is to serialise some entities *by reference*, i.e., recreate them remotely following the standard procedure instead of

Listing 6.2: Example of Jupyter-workflow Notebook with scatter pattern

```
# Cell 1 code
import time

lrs = [0.1, 0.001, 0.0001]
wds = [1e-05, 1e-06, 1e-07]

start_time = time.perf_counter()
# --------------------------------
# Cell 2 metadata
"workflow": {
  "step": {
    "in": [],
    "autoin": "true",
    "out": [],
    "scatter": {
      "items": ["lrs", "wds"],
      "method": "cartesian"
  },
  "version": "v1.0"
}

# Cell 2 code
for lr in lrs:
  for wd in wds:
    print("Train model with lr=" + str(lr) + " and wd=" + str(wd))
    time.sleep(5) # Simulate model train
# --------------------------------
# Cell 3 code
end_time = time.perf_counter()
print(end_time - start_time)
```

reconstructing them from a marshalled internal state. The dill library [191] relies on this strategy for Python modules, 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 a pair of marshalling and unmarshalling routines for a particular object type, augmenting a baseline of standard cases directly handled by the library. This approach has been adopted in the distributed version of the FastFlow framework [192] and also dill comes with a `@register` decorator to extend the standard set of serialisable types. Jupyter-workflow sticks with this last strategy, adopting dill as the base serialisation library and adding a dedicated `serializer` subsection in the input and output dependencies description, as shown in Listing 6.1. Each entry in this subsection lets users specify `predump` and `postload` routines to transform unsuitable values before marshalling and reobtaining the original object after unmarshalling, respectively.

6.1.5 Examples

This section presents some toy examples of Jupyter-workflow Notebooks, allowing the reader to understand better how they are programmed and executed. Note that users can program the cell metadata with the help of a high-level GUI accessible from each cell's toolbar, which is much more intuitive than dealing with JSON directly.

Listing 6.2 provides a simple example of a parallel parameter search for a DNN, which will be expanded in Sec. 6.2.1. In order to keep things as simple as possible, the complex training code has been substituted with a `sleep` of 5 seconds, which is enough to appreciate the presence of parallelism in the execution flow. The Notebook is composed of two code cells. The first one is executed locally in the kernel's context, and it simply initialises the two lists `lrs` and `wds` containing learning rate and weight decay values, respectively. Plus, it initialises a timer to measure the execution time of cell 2, which is then printed by cell 3.

In this simple example, Jupyter-workflow only manages the execution of cell 2, which is decorated with `workflow` metadata. It contains a nested `for` loop that iterates over the defined parameter lists, producing all possible combinations. This pattern is equivalent to a *cartesian product* between `lrs` and `wds`. In particular, cell 2 must process nine combinations of parameters.

Supposing that each execution lasts 5 seconds (as proxied by the `sleep` instruction), the sequential execution of cell 2 will last at least 45s. However, using the `scatter` directive in the cell metadata, the total execution time is reduced to a theoretical optimum of 5s whenever enough hardware resources are available, i.e. when the target locations provide at least nine cores. The actual execution time will be slightly higher because of overheads introduced by the workflow construction and orchestration machinery ($\sim$ 1s) and, in the case of remote executions, the latency and bandwidth of the involved communication channel.

Two crucial aspects emerge from the simple example in Listing 6.2. First, the execution has been parallelised without modifying the sequential code but simply working at the metadata level. As a consequence, the Notebook is compatible with any kernel able to correctly understand the `workflow` metadata, independently of the underlying technology stack used for parallelisation, data serialisation and distributed execution.

Second, the input dependencies of cell 2 have not been explicitly listed, but they have been automatically inferred by the system, as specified by the `autoin` option. This feature can save users from writing a lot of boilerplate code, which is error-prone and sometimes counter-intuitive. For example, in cell 2, the `time` module is also an input dependency, as it must be explicitly imported in the program context prior to calling the `sleep` function.

Listing 6.3 provides a different example, in which the code inside cells cannot be parallelised, but there is still room for optimisation by concurrently executing cells without inter-dependencies. Note that this kind of parallelism cannot be exploited in the interactive execution mode but only using the DAG-based bulk execution provided by Jupyter-workflow.

Again, the first cell initialises the input parameters and starts a timer, which will be used by cell 4 to print the total execution time of the Notebook. Cells 2 and 3 process two different parameters, i.e. `a` and `b`, and are not dependent on each other, so they can be executed in parallel if there are enough resources. Their

Listing 6.3: Example of Jupyter-workflow Notebook with concurrent cell execution

```
# Cell 1 code
import time

a = 1
b = 2

start_time = time.perf_counter()
# ---------------------------------
# Cell 2 metadata
"workflow": {
  "step": {
    "in": [],
    "autoin": "true"
    "out": [
      {
        "name": "control_a",
        "type": "control"
      }
    ]
  },
  "version": "v1.0"
}

# Cell 2 code
print("Processing " + str(a))
time.sleep(5)
# ---------------------------------
# Cell 3 metadata
"workflow": {
  "step": {
    "in": [],
    "autoin": "true"
    "out": [
      {
        "name": "control_b",
        "type": "control"
      }
    ]
  },
  "version": "v1.0"
}

# Cell 3 code
print("Processing " + str(b))
time.sleep(7)
# ---------------------------------
# Cell 4 metadata
"workflow": {
  "step": {
    "in": [
      {
        "name": "control_a",
        "type": "control"
      },
      {
        "name": "control_b",
        "type": "control"
      }
    ],
    "autoin": "true",
    "out": []
  },
  "version": "v1.0"
}

# Cell 4 code
end_time = time.perf_counter()
print(end_time - start_time)
```

execution time is proxied by two `sleep` instructions of 5 and 7 seconds, respectively, leading to a sequential execution time of at least 12s. However, Jupyter-workflow can reduce the theoretical optimum to 7s, i.e. to the longest step's execution time.

Listing 6.3 also provides a use case where explicit input dependencies are needed for proper workflow execution. Indeed, by considering only code-related dependencies cell 4 could be executed in parallel with cells 2 and 3, as it does not require input variables produced by those cells. However, this is not the desired behaviour. To handle this case, two artificial `control` dependencies can be injected as output variables of cells 2 and 3 and as input variables of cell 4. This setting forces cell 4 to wait for cells 2 and 3 to complete before executing, leading to correct time measurement.

Note again that no modification has been made to the business code inside cells. Control dependencies have been injected at the coordination (i.e., metadata) level, and concurrency has been automatically inferred from the code structure. A much more complex application of the same concepts is described in Sec. 6.2.4 below.

6.2 Evaluation

The current section contains an experimental evaluation of how the Jupyter-workflow approach can be effectively applied to standard scientific pipelines in DL, scientific simulation, and Bioinformatics, enabling interactive analysis at scale and promoting literate computing as a full-fledged workflow modelling paradigm.

In particular, Sec. 6.2.1 analyses a Notebook-based implementation of the CLAIRE COVID-19 universal pipeline classification step. Sec. 6.2.2 describes 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 XaaS paradigm for inference. In Sec. 6.2.3, a Quantum ESPRESSO[3] [193] simulation workflow is used to represent a broad class of traditional HPC molecular dynamics simulation tools to investigate how Jupyter-workflow can enable interactive simulations at scale. Finally, in Sec. 6.2.4, a Bioinformatics pipeline based on the 1000 Genomes project [194] is used to analyse performances in the cloud.

6.2.1 DNN hyperparameter search

This section relies on a Jupyter-workflow implementation of the most computationally intensive portion of the COVID-19 universal pipeline to demonstrate how the proposed approach effectively combines usability and scalability. The pipeline is composed of a data processing section and a core training workflow, and its goal is to perform a metrics assessment on 11 variants of DNN models, each with its

[3]http://www.quantum-espresso.org

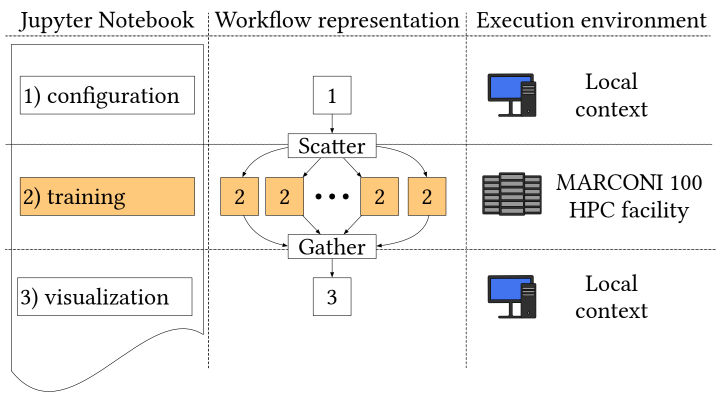


Figure 6.3: Graphical representation of the Jupyter-workflow execution plan for the CLAIRE-COVID19 Notebook.

hyper-parameters. The interested reader can find a detailed description of the pipeline in Sec. 4.2.2.

The Notebook version of the classification step used in this experiment performs a hyperparameter search for 4 different DenseNet models [185]: DenseNet-121, DenseNet-161, DenseNet-169, and DenseNet-201. In order to improve classification performances, it adopts a *transfer learning* approach: weights pre-trained on the ImageNet dataset [195] are fine-tuned on a pre-processed subset of the BIMCV-COVID19 dataset [187] using a standardly configured Adam optimiser [196] ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$). Such transfer learning process is configured to run for at most 50 epochs, with early stopping after 10 epochs without improving the validation loss.

The Notebook explores 12 different configurations for each model 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, it performs 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 Jupyter-workflow, the code can be easily split into three main sections (as shown in Fig. 6.3):

- an initial configuration section, containing the module imports and the hyper-parameters' grid;
- a training section, i.e. a single Notebook cell containing the main training loop;
- a visualisation section, where the metrics of each training experiment can be efficiently analysed 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, Jupyter-workflow automatically generates the cartesian product of input configurations and schedules them for

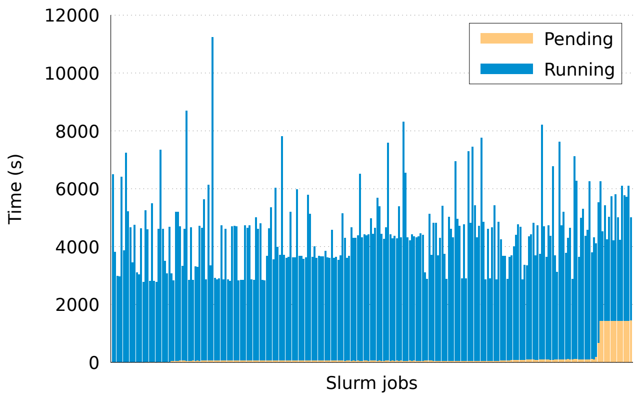


Figure 6.4: Execution times for the 240 DenseNet training experiments running in parallel on 240 GPU-equipped nodes of the CINECA MARCONI 100 facility.

concurrent execution. Nevertheless, the actual amount of concurrency strongly depends on the chosen execution environment. In order to fully take advantage of their embarrassingly parallel nature, the 240 training steps are offloaded 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, only a single GPU has been requested 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 Jupyter-workflow kernel, running on a desktop machine equipped with an Intel i7-7700K CPU (4 cores, 8 threads, 4.20 GHz). Notice 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, the serialisation and deserialisation 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. 6.4 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 actual 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 minutes. Nevertheless, a cell's global execution (scatter) can complete only after the tail of the slowest jobs, which took more than 3 hours 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 hours to complete the training. Conversely, Jupyter-workflow allows a $\times 92$ faster execution without sacrificing the Jupyter

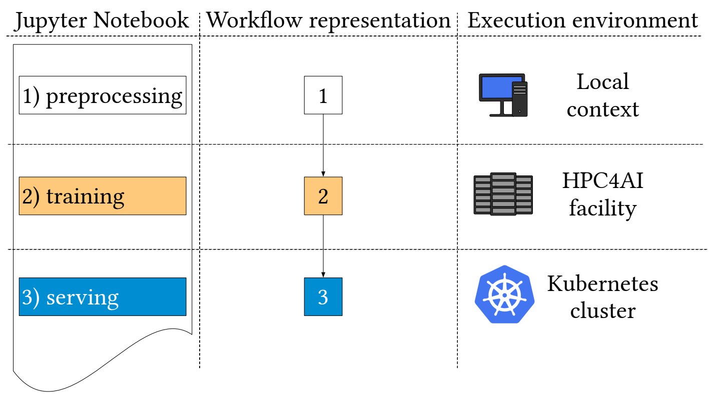


Figure 6.5: Graphical representation of the Jupyter-workflow execution plan for the TensorFlow training+serving Notebook.

high-level interactive visualisation tools.

6.2.2 Training and serving DNNs

In the DL field, training+serving pipelines can highly benefit from a mixed cloud-HPC execution. Indeed, even if HPC facilities with heterogeneous computing nodes are ideal for model training, their queue-based workload management and limited Internet access are unsuitable for the serving phase, as inference usually comes with strict real-time requirements and needs a publicly exposed REST API. This section describes how Jupyter-workflow can efficiently orchestrate pre-processing, training, and serving tasks for a DNN using TensorFlow [135], offloading the execution of each step to the most suitable infrastructure (see Fig. 6.5).

Since the experiment evaluates design-related aspects rather than performances, the Notebook only sets up a playground with a very small CNN trained on the Fashion-MNIST dataset [197]. After a local data pre-processing phase, the training step is offloaded to a bare metal node of the HPC4AI facility [25], 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, the already pre-processed dataset can be treated as a `name` dependency, letting StreamFlow manage serialisation and transfer operations. Conversely, it is more efficient for massive datasets to move pre-processing and training steps close to the data. 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 6.4. 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.

Listing 6.4: Data handling strategies in Jupyter-workflow

```
/************************************************************
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) */
```

Concerning serialisation, some internal data structures prevent the dill library from successfully parsing TensorFlow networks. However, Jupyter-workflow easily solves this issue by putting some custom logic in the related `serializer` section. In particular, it is possible 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. Moreover, if necessary, the deserialisation logic can be extended to upload the model on one or more GPU devices.

As soon as the training step terminates, the resulting model is stored in a Docker container, 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 (CI) purposes, strongly reducing the gap between prototyping and deployment phases in the development life-cycle.

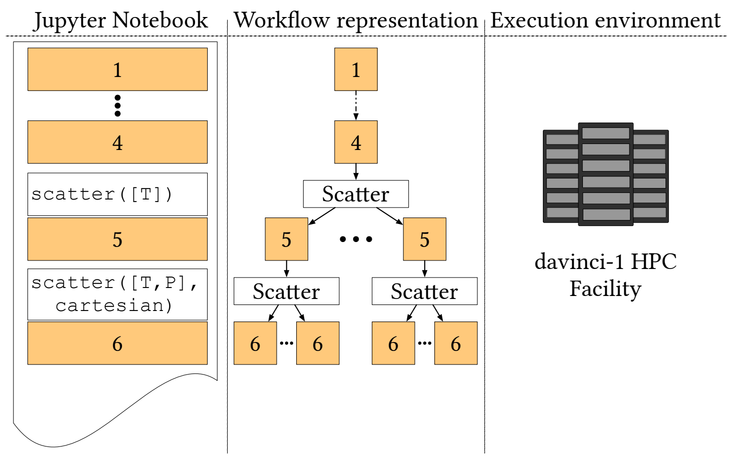


Figure 6.6: Graphical representation of the Jupyter-workflow execution plan for the Quantum ESPRESSO Notebook.

Notice 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 initialisation step because the `DependencyResolver` cannot discriminate between strings and file paths.

6.2.3 Interactive simulations at scale

This section empirically evaluates the Jupyter-workflow capabilities to enable interactive simulations of realistic, large-scale systems. In particular, it analyses the weak scalability of a Notebook containing a multi-step simulation workflow in Quantum ESPRESSO, which implements a Car-Parrinello simulation of a mixture of H_2O, NH_3 and CH_4 molecules to represent the so-called primordial soup. Such simulation explores 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 represented in Fig. 6.6, the workflow proceeds as follows. Common to all configurations, the first four cells 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 T, the simulation forks again to simulate each temperature at several P values using the Parrinello-Rahman constant pressure Lagrangian (cell 6).

The following discussion focuses on the last two steps, as others are trivial. Using the Jupyter-workflow metadata format, cell 5 can be parallelised by scattering on T, while cell 6 can use a cartesian product operator to scatter over all (P, T) combinations. In the interactive execution 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 (see Sec. 5.2).

Table 6.1: Weak scalability for the Quantum ESPRESSO simulation when executed manually on PBS and in both Jupyter-workflow (Notebook) execution modes.

Step	Nodes	PBS (s)	Notebook Interactive (s)	Notebook 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

Each workflow step is offloaded to two CPU nodes of davinci–1, the Leonardo S.p.A. HPC system, each equipped with 2 Intel Xeon Platinum 8260 sockets (24 cores, 2.40 GHz each) and 1 TB of RAM. Weak scalability of the application is evaluated 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. The results are reported in Table 6.1. Notice how the overhead introduced by the interactive execution mode becomes predominant with 16 (P, T) points, while it remains negligible in the bulk evaluation mode. These results empirically confirm the effectiveness of the parallel patterns rewriting rules in optimising the execution plan.

The example Notebook discussed here 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 adapted for any simulation of the P-T phase diagram of any material, scaling a single (P, T) point simulation up to several thousands of nodes. Peak performances are not an issue: the Quantum ESPRESSO suite has been shown to scale well to petascale systems, and it is currently addressing the exascale challenges [198]. However, much of the Quantum ESPRESSO performance derives from the linked matrix multiplication libraries, tightly coupled with the underlying hardware technology at compile time.

Tweaking performances of these 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 [94] and LAPACK [95] can have a massive impact on the time-to-solution (see Sec. 2.3.2). With its capability to seamlessly offload computation to optimised execution environments on HPC facilities, Jupyter-workflow

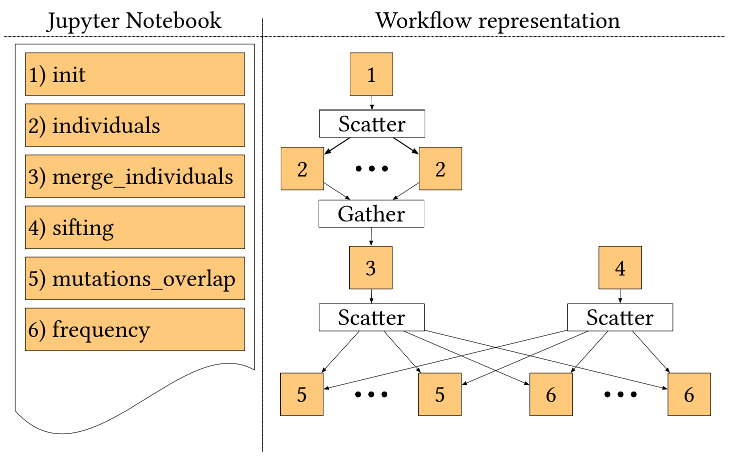


Figure 6.7: Graphical representation of the Jupyter-workflow execution plan for the 1000-genome workflow.

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.

6.2.4 The 1000-genome literate workflow

This section investigates Jupyter-workflow strong scalability on distributed infrastructures. In particular, a 1000-genome workflow instance is executed on a Kubernetes cluster running on top of the HPC4AI OpenStack-based cloud. A detailed description of the 1000-genome workflow initially implemented in Pegasus is available in the literature [199]. Fig. 6.7 shows the Jupyter Notebook representation of the workflow (as a list of 6 cells) and the corresponding DAG automatically extracted by the Jupyter-workflow runtime. Notice how the amount of concurrency enabled by the DAG evaluation strategy becomes significant when dealing with complex workflows.

The 1000-genome pipeline has been selected as a representative of large-scale scientific workflows for three main reasons:

- Pegasus is a state-of-the-art representative of High Throughput Computing (HTC) WMSs, supporting distributed execution environments without a unique shared data space (via HTCondor);
- the code of each workflow step is written in Bash or Python, both supported by the IPython kernel;
- the critical portion of the pipeline is a highly parallel step, composed of 2000 independent short tasks (∼120s each), which are unsuitable for queue-based

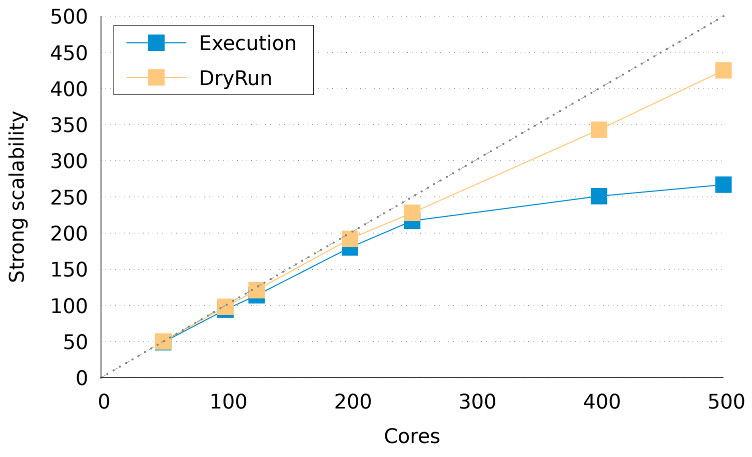


Figure 6.8: Speedup obtained executing the 1000-genome workflow on the HPC4AI cloud. The blue curve refers to the actual execution. The orange curve shows a DryRun of the same workflow to assess Jupyter-workflow overhead (the business code is substituted with sleeps matching execution time).

batch workload managers but can efficiently be executed at scale by on-demand cloud locations (e.g. Kubernetes).

The porting of the host code to Jupyter-workflow merely requires creating a cell for each step by copy-pasting the original code. Concerning coordination logic, the two WMSs adopt a diverse approach. Pegasus requires users to model a static workflow graph, specifying all the input and output dependencies of each step at compile time. This strategy is compelling in terms of expressiveness, as expressible graphs are not limited to the composition of a predefined set of patterns. The Jupyter-workflow approach favours simplicity and usability, but it is limited by the original sequential nature of Jupyter Notebooks, even if parallel patterns and DAG evaluation strongly mitigate the constraints.

Jupyter-workflow can seamlessly deal with dynamic outputs. This requirement is fundamental 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 perform entire families of similar experiments by simply changing the input values (as analysed in Sec. 6.2.3). Conversely, encoding dynamic data dependencies in a Pegasus workflow requires users to embed complex just-in-time compilation steps in the workflow graph explicitly.

The experiment discussed in this section measures the strong scaling of an 8-chromosomes instance of the 1000-genome workflow running on up to 500 concurrent Kubernetes Pods. In particular, the underlying Kubernetes cluster comprises 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 reserves 1 core and 2GB of RAM and mounts a 1GB tmpfs. Only the `individuals` step is taken into account for performance evaluation, as it constitutes by far the bottleneck of the workflow. Plus, the experiment relies on a slightly modified version of the pipeline, which better addresses distributed architectures. Indeed, in the original implementation, a chromosome input file is made available to all involved workers in its entirety, and each of them selects a different partition. This operation generates massive traffic on the networking layer when workers do not share the file system. Instead, the Jupyter-workflow implementation scatters the dataset, transferring to each worker only the required data.

On cloud architectures, all the resources (CPUs, memory, network, disks) are typically overprovisioned and subject to load generated by other users. Therefore, a DryRun version of the code has been developed to serve as a baseline. The DryRun simulates the workflow behaviour 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. Fig. 6.8 compares the strong scalability of real and DryRun executions. The former scales reasonably well up to 250 containers, then it starts suffering from the data distribution bottleneck introduced by the Kubernetes control plane. For its part, the DryRun shows that the intrinsic overhead introduced by the Jupyter-workflow runtime synchronisations (orange curve) keeps a reasonably linear gap against ideal speedup (at least up to 500 Pods).

As theoretically expected, the central aspect of performance in a cloud/Kubernetes setting is tuning the communication/computation ratio at the Kubernetes control plane, which does not leave much room for optimisation in I/O-bound problems like the 1000-genome workflow. Indeed, in this experiment, data transfers are performed via WebSocket using the `kubectl` CLI, a portable but suboptimal baseline strategy to communicate with isolated Kubernetes Pods using the controller nodes as a bridge.

In these cases, viable optimisation paths concern the distribution or elimination of data movements. The former can be realised by implementing direct communication channels between worker Pods, e.g. through the SCP protocol, leaving the Kubernetes control plane out of the critical path. The latter is enabled by rewriting rules such as Map fusion (see Sec. 5.2). Undoubtedly a more detailed analysis of the different overhead sources in Kubernetes-based cloud environments would be an essential topic for further research. Still, a proper monitoring setting would require a more isolated environment to suppress or artificially inject the performance volatility generated by resource sharing in a controlled way.

Another crucial analysis would be comparing state-of-art implementations of the 1000-genome use case with Pegasus and Jupyter-workflow. The main complexity of such aexperiment derives from the different target execution architectures that the two WMSs are optimised for. Indeed, Pegasus has been developed to run static workflow diagrams on top of HTC architectures managed by HTCondor, while StreamFlow (and consequently Jupyter-workflow) is designed for container-native cloud applications and queue-based HPC clusters. A proper ad-hoc configuration of these execution environments and the fine-tuning of all the WMS parameters to achieve the best performances on top of them are complex activities which require expert users to reach state of the art. Conversely, comparing an optimised version of one tool with a suboptimal version of the other will lead to biased and unfair results. However, developing community consensus on workflow benchmarking applications and developing reproducible and agnostic methodologies to collect and report benchmark results are two of the main goals of the Workflow Benchmarking Group (WfBG). Therefore, a further investigation of this use case is in plan from this perspective.

Chapter 7

Conclusion

7.1 Conclusion and remarks

This thesis introduced two methodological contributions in the field of large-scale distributed workflows and two full-fledged WMSs implementations designed and developed according to such methodologies.

Hybrid workflows, introduced in Chapter 3, augment standard workflow models with topology awareness, incorporating detailed representations of execution environments in terms of computing locations and communication channels directly in the workflow definition. This knowledge allows for compile-time soundness evaluation of the whole model, efficient scheduling strategies based on data-locality, portability and reproducibility of experiments promoted by the clear separation of workflow logic and execution architecture.

The StreamFlow framework, described in Chapter 4, provides runtime support for CWL-based hybrid workflows running on mixed cloud-HPC execution environments. It has been evaluated on two real scientific pipelines in the fields of Bioinformatics and DL, showing how the hybrid workflows approach can be beneficial to improve the resource allocation strategy without significant performance drops, e.g. by offloading computationally heavy steps to HPC facilities while executing less demanding ones on cheaper cloud VMs.

Distributed literate workflows, discussed in Chapter 5, augment the inherently sequential execution model of computational notebooks with sequentially equivalent parallel semantics and global configurations to represent notebook cells as the steps of a hybrid workflow DAG. In addition, notebook semantics are augmented to express explicit parallel patterns (e.g. Scatter/Gather), which can be further optimised with proper rewriting rules to increase the amount of concurrency in workflow executions.

The Jupyter-workflow framework, introduced in Chapter 6, extends the IPython software stack to support distributed literate workflows in Jupyter Notebooks. It has been evaluated on large-scale HPC and cloud environments with four practical applications in the domains of DL, scientific simulation and Bioinformatics, showing how literate workflows can evolve computational notebooks from a prototyping technology to a high-level programming paradigm for scientific applications.

In the author's opinion, the proposed methodologies bring significant advances in modelling and orchestrating modern workflows. The heterogeneity and complexity of modern applications force monolithic approaches to give way to modular architectures and patterns for designing and developing software, of which workflow models are first-class representatives. However, in the same way, the heterogeneity in contemporary hardware resources and their features (e.g. highly parallel hardware accelerators, low energy consuming FPGAs, or application-specific quantum solvers) fosters modular approaches also in the design of execution environments for such applications.

Hybrid workflows represent an essential methodological step in this direction. An explicit representation of the entire environment generalises the concepts of portability and reproducibility from the application plane to the entire execution process. The separation of concerns brought by hybrid workflow models promotes cooperation between domain experts, who write the application logic, and computer scientists, who find the best execution environment for each workflow step according to specific requirements (e.g. in terms of cost, time-to-solution or energy consumption). At the same time, both of them are free from the burdens of managing applications deployment and life-cycle and writing explicit data transfers logics, enhancing productivity. In addition, the flexibility of a loose mapping relation between steps and locations allows for automatic cross-stack executions of independent steps, providing a trivial way to offload tasks in urgent computing scenarios.

Finally, the high-level programming paradigm and the user-friendly IDE provided by computational notebooks, together with a static soundness evaluation, facilitate the adoption of the proposed methodology among a broad class of users, lowering the technical barriers to model hybrid workflows and providing a unique interface to access heterogeneous execution infrastructures.

7.2 Future work

Future research paths from this work can be classified into two main families: methodology extensions and further empyrical evaluations.

Regarding methodologies, the formalisation of hybrid workflow models must unavoidably be extended to treat dynamic workflows containing conditional branches and iterative patterns, exploring at least the primary workflow modelling tools like Petri Nets [29] and dataflow graphs [34]. In addition, literate workflows could also greatly benefit from iterative coordination patterns, as many scientific applications contain iterations in their business logic (e.g. DNN training or molecular dynamics simulation). The first step in this direction, which is ongoing, is adding a structured, iterative construct to the CWL standard. Such a construct has been already proposed to the CWL Leadership Team, and StreamFlow will be the first WMS to implement it.

In parallel, the actual effectiveness of the proposed methodology should be tested on a broader set of applications. Extending StreamFlow to support other execution environments, like modular extreme-scale HPC facilities and quantum solvers, and Jupyter-workflow to integrate other kernels, such as Julia and R, is the first step to broadening the class of compatible applications. Plus, the addition of an iterative construct will enable both tools to manage all optimisation-based scientific workloads, like large-scale simulations based on iterative solvers, distributed and federated DL workloads, and NISQ-based quantum computing algorithms.

On the other hand, proper dissemination of the proposed methodologies and tools is essential to find additional use cases in various academic and industrial domains. In this direction, StreamFlow is currently part of the software stack in several ongoing European projects (DeepHealth, ACROSS, EUPEX, and others), it is officially recognised as a fully-compliant implementation of the CWL standard[1], and it is part of the Workflow Community Initiative (WCI)[2], a community-supported common knowledge-base for workflow research and development.

[1]https://www.commonwl.org/implementations/

[2]https://workflows.community/systems/streamflow/

References

[1] J. Dean and S. Ghemawat, «MapReduce: Simplified Data Processing on Large Clusters», in *Usenix OSDI '04*, Dec. 2004, pp. 137–150.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, «Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing», in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, S. D. Gribble and D. Katabi, Eds., USENIX Association, 2012, pp. 15–28.

[3] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, «Communication-Efficient Learning of Deep Networks from Decentralized Data», in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*, 2017, pp. 1273–1282.

[4] S. Warnat-Herresthal, H. Schultze, K. L. Shastry, S. Manamohan, S. Mukherjee, V. Garg, R. Sarveswara, K. Händler, P. Pickkers, N. A. Aziz, S. Ktena, F. Tran, M. Bitzer, S. Ossowski, N. Casadei, C. Herr, D. Petersheim, U. Behrends, F. Kern, T. Fehlmann, P. Schommers, C. Lehmann, M. Augustin, J. Rybniker, J. Altmüller, N. Mishra, J. P. Bernardes, B. Krämer, L. Bonaguro, J. Schulte-Schrepping, E. De Domenico, C. Siever, M. Kraut, M. Desai, B. Monnet, M. Saridaki, C. M. Siegel, A. Drews, M. Nuesch-Germano, H. Theis, J. Heyckendorf, S. Schreiber, S. Kim-Hellmuth, P. Balfanz, T. Eggermann, P. Boor, R. Hausmann, H. Kuhn, S. Isfort, J. C. Stingl, G. Schmalzing, C. K. Kuhl, R. Röhrig, G. Marx, S. Uhlig, E. Dahl, D. Müller-Wieland, M. Dreher, N. Marx, J. Nattermann, D. Skowasch, I. Kurth, A. Keller, R. Bals, P. Nürnberg, O. Rieß, P. Rosenstiel, M. G. Netea, F. Theis, S. Mukherjee, M. Backes, A. C. Aschenbrenner, T. Ulas, A. Angelov, A. Bartholomäus, A. Becker, D. Bezdan, C. Blumert, E. Bonifacio, P. Bork, B. Boyke, H. Blum, T. Clavel, M. Colome-Tatche, M. Cornberg, I. A. De La Rosa Velázquez, A. Diefenbach, A. Dilthey, N. Fischer, K. Förstner, S. Franzenburg, J.-S. Frick, G. Gabernet, J. Gagneur, T. Ganzenmueller, M. Gauder, J. Geißert, A. Goesmann, S. Göpel, A. Grundhoff, H. Grundmann, T. Hain, F. Hanses, U. Hehr, A. Heimbach, M. Hoeper, F. Horn, D. Hübschmann, M. Hummel, T. Iftner, A. Iftner, T. Illig, S. Janssen, J. Kalinowski, R. Kallies, B. Kehr, O. T. Keppler, C. Klein, M. Knop, O. Kohlbacher, K. Köhrer, J. Korbel, P. G. Kremsner, D. Kühnert, M. Landthaler, Y. Li, K. U. Ludwig, O. Makarewicz, M. Marz, A. C. McHardy, C. Mertes, M. Münchhoff, S. Nahnsen, M. Nöthen, F. Ntoumi, J. Overmann, S. Peter, K. Pfeffer, I. Pink, A. R. Poetsch, U. Protzer, A. Pühler, N. Rajewsky, M. Ralser, K. Reiche, S. Ripke, U. N. da Rocha, A.-E. Saliba, L. E. Sander, B. Sawitzki, S. Scheithauer, P. Schiffer, J. Schmid-Burgk, W. Schneider, E.-C. Schulte, A. Sczyrba, M. L. Sharaf, Y. Singh, M. Sonnabend, O. Stegle, J. Stoye, J. Vehreschild, T. P. Velavan, J. Vogel, S. Volland, M. von Kleist, A. Walker, J. Walter, D. Wieczorek, S. Winkler, J. Ziebuhr, M. M. B. Breteler, E. J. Giamarellos-Bourboulis, M. Kox, M. Becker, S. Cheran, M. S. Woodacre, E. L. Goh, J. L. Schultze, COVID-19 Aachen Study (COVAS), and Deutsche COVID-19 Omics Initiative (DeCOI), «Swarm

Learning for decentralized and confidential clinical machine learning», *Nature*, vol. 594, no. 7862, pp. 265–270, Jun. 2021, ISSN: 1476-4687. DOI: 10.1038/s41586-021-03583-3.

[5] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez, «Storage challenges at Los Alamos National Lab», in *IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012, April 16-20, 2012, Asilomar Conference Grounds, Pacific Grove, CA, USA*, IEEE Computer Society, 2012, pp. 1–5. DOI: 10.1109/MSST.2012.6232376.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, «Language Models are Few-Shot Learners», in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M.-F. Balcan, and H.-T. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

[7] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, «Zero-Shot Text-to-Image Generation», in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, M. Meila and T. Zhang, Eds., ser. Proceedings of Machine Learning Research, vol. 139, PMLR, 2021, pp. 8821–8831.

[8] P. H. Beckman, S. Nadella, N. Trebon, and I. Beschastnikh, «SPRUCE: A System for Supporting Urgent High-Performance Computing», in *Grid-Based Problem Solving Environments - IFIP TC2/ WG 2.5 Working Conference on Grid-Based Problem Solving Environments: Implications for Development and Deployment of Numerical Software July 17-21, 2006, Prescott, Arizona, USA*, P. W. Gaffney and J. C. T. Pool, Eds., ser. IFIP, vol. 239, Springer, 2006, pp. 295–311. DOI: 10.1007/978-0-387-73659-4_16.

[9] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. C. Bavier, and L. L. Peterson, «Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors», in *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, 2007, pp. 275–287. DOI: 10.1145/1272996.1273025.

[10] P. Di Tommaso, M. Chatzou, E. W. Floden, *et al.*, «Nextflow enables reproducible computational workflows», *Nature Biotechnology*, vol. 35, no. 4, pp. 316–319, Apr. 2017, ISSN: 1087-0156. DOI: 10.1038/nbt.3820.

[11] K. J. Millman and F. Pérez, «Developing open-source scientific practice», in *Implementing Reproducible Research*, Chapman and Hall/CRC, 2014, pp. 149–183.

[12] H. Shen, «Interactive notebooks: Sharing the code», *Nature*, vol. 515, no. 7525, pp. 151–152, Nov. 2014, ISSN: 0028-0836. DOI: 10.1038/515151a.

[13] I. Colonnelli, B. Cantalupo, I. Merelli, and M. Aldinucci, «StreamFlow: cross-breeding cloud with HPC», *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 4, pp. 1723–1737, 2021. DOI: 10.1109/TETC.2020.3019202.

[14] 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, and L. Stojanovic, *Common Workflow Language, v1.0*, 2016. DOI: 10.6084/m9.figshare.3115156.v2.

[15] I. Colonnelli, M. Aldinucci, B. Cantalupo, L. Padovani, S. Rabellino, C. Spampinato, R. Morelli, R. Di Carlo, N. Magini, and C. Cavazzoni, «Distributed workflows with Jupyter», *Future Generation Computer Systems*, vol. 128, pp. 282–298, 2022, ISSN: 0167-739X. DOI: 10.1016/j.future.2021.10.007.

[16] M. Cole, «A Skeletal Approach to Exploitation of Parallelism», in *Proc. of CONPAR 88*, ser. British Computer Society Workshop Series, Cambridge University Press, 1989, pp. 667–675.

[17] M. Cole, «Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming», *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.

[18] P. R. Rosenbaum and D. B. Rubin, «The Central Role of the Propensity Score in Observational Studies for Causal Effects», *Biometrika*, vol. 70, no. 1, pp. 41–55, 1983, ISSN: 00063444.

[19] P. R. Rosenbaum and D. B. Rubin, «Reducing Bias in Observational Studies Using Subclassification on the Propensity Score», *Journal of the American Statistical Association*, vol. 79, no. 387, pp. 516–524, 1984, ISSN: 01621459.

[20] D. R. Cox, «Regression Models and Life-Tables», *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 34, no. 2, pp. 187–220, 1972, ISSN: 00359246.

[21] Y. Arfat, G. Mittone, R. Esposito, B. Cantalupo, G. M. De Ferrari, and M. Aldinucci, «A Review of Machine Learning for Cardiology», *Minerva cardiology and angiology*, 2021. DOI: 10.23736/s2724-5683.21.05709-4.

[22] Y. Freund and R. E. Schapire, «A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting», *J. Comput. Syst. Sci.*, vol. 55, no. 1, pp. 119–139, 1997. DOI: 10.1006/jcss.1997.1504.

[23] M. Caballero, J. A. Gómez, and A. Bantouna, «Deep-Learning and HPC to Boost Biomedical Applications for Health (DeepHealth)», in *32nd IEEE International Symposium on Computer-Based Medical Systems, CBMS 2019, Cordoba, Spain, June 5-7, 2019*, IEEE, 2019, pp. 150–155. DOI: 10.1109/CBMS.2019.00040.

[24] 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, and 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. DOI: 10.1145/3457388.3458508.

[25] 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, and F. Galeazzi, «HPC4AI, an AI-on-demand federated platform endeavour», in *ACM Computing Frontiers*, Ischia, Italy, May 2018. DOI: 10.1145/3203217.3205340.

[26] WFMC, «Workflow Management Coalition Terminology & Glossary, Document Number WFMC-TC-1011, Document Status – Issue 3.0», in *Document Status–Issue 3.0*. Brussels: Workflow Management Coalition, Feb. 1999.

[27] B. Ludäscher, S. Bowers, and T. M. McPhillips, «Scientific Workflows», in *Encyclopedia of Database Systems, Second Edition*, Springer, 2018. DOI: 10.1007/978-1-4614-8265-9_1471.

[28] W. M. P. van der Aalst, «Three good reasons for using a Petri-net-based workflow management system», English, in *Information and process integration in enterprises: Rethinking documents*, ser. The Kluwer International Series in Engineering and Computer. Netherlands: Kluwer Academic Publishers, 1998, pp. 161–182.

[29] W. Reisig and G. Rozenberg, Eds., *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, vol. 1491, Lecture Notes in Computer Science, Springer, 1998, ISBN: 3-540-65306-6. DOI: 10.1007/3-540-65306-6.

[30] K. M. Kavi, B. P. Buckles, and U. N. Bhat, «A Formal Definition of Data Flow Graph Models», *IEEE Trans. Computers*, vol. 35, no. 11, pp. 940–948, 1986. DOI: 10.1109/TC.1986.1676696.

[31] J. D. Brock, «A formal model of non-determinate dataflow computation», Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[32] K. Jensen, «Coloured Petri nets: A high level language for system design and analysis», in *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, 1989, pp. 342–416. DOI: 10.1007/3-540-53863-1_31.

[33] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, «Workflow Patterns», *Distributed Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003. DOI: 10.1023/A:1022883727209.

[34] E. Lee and T. Parks, «Dataflow Process Networks», *Proc. of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[35] J. Yu and R. Buyya, «A Taxonomy of Workflow Management Systems for Grid Computing», *Journal of Grid Computing*, vol. 3, no. 3-4, pp. 171–200, 2005. DOI: 10.1007/s10723-005-9010-8.

[36] J. Liu, E. Pacitti, and V. P. et al, «A Survey of Data-Intensive Scientific Workflow Management», *Journal of Grid Computing*, vol. 13, no. 4, pp 457–493, Dec. 2015.

[37] S. C. Boulakia, K. Belhajjame, O. Collin, J. Chopard, C. Froidevaux, A. Gaignard, K. Hinsen, P. Larmande, Y. L. Bras, F. Lemoine, F. Mareuil, H. Ménager, C. Pradal, and C. Blanchet, «Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities», *Future Generation Comp. Syst.*, vol. 75, pp. 284–298, 2017. DOI: 10.1016/j.future.2017.01.012.

[38] M. P. Atkinson, S. Gesing, J. Montagnat, and I. J. Taylor, «Scientific workflows: Past, present and future», *Future Generation Comp. Syst.*, vol. 75, pp. 216–227, 2017. DOI: 10.1016/j.future.2017.05.041.

[39] R. F. da Silva, R. Filgueira, I. Pietri, M. Jiang, R. Sakellariou, and E. Deelman, «A characterization of workflow management systems for extreme-scale applications», *Future Generation Comp. Syst.*, vol. 75, pp. 228–238, 2017.

[40] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. B. Jones, E. A. Lee, J. Tao, and Y. Zhao, «Scientific workflow management and the Kepler system», *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006. DOI: 10.1002/cpe.994.

[41] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H. L. Truong, A. Villazón, and M. Wieczorek, «ASKALON: A Development and Grid Computing Environment for Scientific Workflows», in *Workflows for e-Science, Scientific Workflows for Grids*, 2007, pp. 450–471. DOI: 10.1007/978-1-84628-757-2_27.

[42] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, and R. K. Wenger, «Pegasus, a workflow management system for science automation», *Future Generation Comp. Syst.*, vol. 46, pp. 17–35, 2015. DOI: 10.1016/j.future.2014.10.008.

[43] 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, and C. Wroe, «Taverna: lessons in creating a workflow environment for the life sciences», *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1067–1100, 2006. DOI: 10.1002/cpe.993.

[44] I. J. Taylor, M. S. Shields, I. Wang, and A. Harrison, «The Triana Workflow Environment: Architecture and Applications», in *Workflows for e-Science, Scientific Workflows for Grids*, Springer, 2007, pp. 320–339. DOI: 10.1007/978-1-84628-757-2_20.

[45] E. Afgan, D. Baker, M. van den Beek, D. J. Blankenberg, D. Bouvier, M. Cech, J. Chilton, D. Clements, N. Coraor, C. Eberhard, B. A. Grüning, A. Guerler, J. Hillman-Jackson, G. V. Kuster, E. Rasche, N. Soranzo, N. Turaga, J. Taylor, A. Nekrutenko, and J. Goecks, «The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update», *Nucleic Acids Research*, vol. 44, no. Webserver-Issue, W3–W10, 2016. DOI: 10.1093/nar/gkw343.

[46] R. Badia, E. Ayguade, and J. Labarta, «Workflows for Science: A Challenge When Facing the Convergence of HPC and Big Data», *Supercomput. Front. Innov.: Int. J.*, vol. 4, no. 1, pp. 27–47, Mar. 2017, ISSN: 2409-6008. DOI: 10.14529/jsfi170102.

[47] I. J. Taylor, M. S. Shields, I. Wang, and O. F. Rana, «Triana Applications within Grid Computing and Peer to Peer Environments», *J. Grid Comput.*, vol. 1, no. 2, pp. 199–217, 2003. DOI: 10.1023/B:GRID.0000024074.63139.ce.

[48] M. Siddiqui, A. Villazón, J. Hofer, and T. Fahringer, «GLARE: A Grid Activity Registration, Deployment and Provisioning Framework», in *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA, CD-Rom*, 2005, p. 52. DOI: 10.1109/SC.2005.30.

[49] D. Thain, T. Tannenbaum, and M. Livny, «Distributed computing in practice: the Condor experience», *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005. DOI: 10.1002/cpe.938.

[50] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. R. Sachs, and Y. Xiong, «Taming heterogeneity - the Ptolemy approach», *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003. DOI: 10.1109/JPROC.2002.805829.

[51] D. L. Cuadrado, A. P. Ravn, and P. Koch, «Automated distributed simulation in PTOLEMY II», in *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, as part of the 25th IASTED International Multi-Conference on Applied Informatics, February 13-15 2007, Innsbruck, Austria*, H. Burkhart, Ed., IASTED/ACTA Press, 2007, pp. 138–143.

[52] J. Köster and S. Rahmann, «Snakemake - a scalable bioinformatics workflow engine», *Bioinformatics*, vol. 28, no. 19, pp. 2520–2522, 2012. DOI: 10.1093/bioinformatics/bts480.

[53] M. Albrecht, P. Donnelly, P. Bui, and 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. DOI: 10.1145/2443416.2443417.

[54] J. Vivian, A. A. Rao, F. A. Nothaft, *et al.*, «Toil enables reproducible, open source, big biomedical data analyses», *Nature Biotechnology*, vol. 35, no. 4, pp. 314–316, Apr. 2017, ISSN: 1087-0156. DOI: 10.1038/nbt.3772.

[55] D. D. Sánchez-Gallegos, D. D. Luccio, S. Kosta, J. L. G. Compeán, and R. Montella, «An efficient pattern-based approach for workflow supporting large-scale science: The DagOnStar experience», *Future Gener. Comput. Syst.*, vol. 122, pp. 187–203, 2021. DOI: 10.1016/j.future.2021.03.017.

[56] M. Kotliar, A. V. Kartashov, and A. Barski, «CWL-Airflow: a lightweight pipeline manager supporting Common Workflow Language», *GigaScience*, vol. 8, no. 7, Jul. 2019, ISSN: 2047-217X. DOI: 10.1093/gigascience/giz084.

[57] 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, and I. Stoica, «Apache Spark: a unified engine for big data processing», *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016. doi: 10.1145/2934664.

[58] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, «Apache Flink™: Stream and Batch Processing in a Single Engine», *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015. [Online]. Available: http://sites.computer.org/debull/A15dec/p28.pdf.

[59] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini, «The power of both choices: Practical load balancing for distributed stream processing engines», in *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, Eds., IEEE Computer Society, 2015, pp. 137–148. doi: 10.1109/ICDE.2015.7113279.

[60] V. Cima, S. Böhm, J. Martinovic, J. Dvorský, K. Janurová, T. V. Aa, T. J. Ashby, and 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. doi: 10.1145/3183767.3183768.

[61] Dask Development Team, *Dask: Library for dynamic task scheduling*, 2016. [Online]. Available: https://dask.org.

[62] F. Marozzo, F. Lordan, R. Rafanell, D. Lezzi, D. Talia, and R. M. Badia, «Enabling Cloud Interoperability with COMPSs», in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 16–27, isbn: 978-3-642-32820-6.

[63] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, «Ray: A Distributed Framework for Emerging AI Applications», in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, 2018, pp. 561–577.

[64] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, «Parsl: Pervasive Parallel Programming in Python», in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '19, Phoenix, AZ, USA: ACM, 2019, pp. 25–36, isbn: 978-1-4503-6670-0. doi: 10.1145/3307681.3325400.

[65] H. G. Baker and 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. doi: 10.1145/800228.806932.

[66] C. Misale, M. Drocco, M. Aldinucci, and G. Tremblay, «A Comparison of Big Data Frameworks on a Layered Dataflow Model», *Parallel Processing Letters*, vol. 27, no. 01, pp. 1–20, 2017. doi: 10.1142/S0129626417400035.

[67] 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, and B. Ludäscher, «YesWorkflow: A User-Oriented, Language-Independent Tool for Recovering Workflow Information from Scripts», *CoRR*, vol. abs/1502.02403, 2015. arXiv: 1502.02403.

[68] B. Lerner and E. R. Boose, «RDataTracker: Collecting Provenance in an Interactive Scripting Environment», in *6th Workshop on the Theory and Practice of Provenance, TaPP'14, Cologne, Germany, June 12-13, 2014*, A. Chapman, B. Ludäscher, and A. Schreiber, Eds., USENIX Association, 2014. [Online]. Available: https://www.usenix.org/conference/tapp2014/agenda/presentation/lerner.

[69] L. A. M. C. Carvalho, K. Belhajjame, and C. B. Medeiros, «Converting scripts into reproducible workflow research objects», in *12th IEEE International Conference on e-Science, e-Science 2016, Baltimore, MD, USA, October 23-27, 2016*, 2016, pp. 71–80. DOI: 10.1109/eScience.2016.7870887.

[70] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, «noWorkflow: a Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts», *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1841–1844, 2017. DOI: 10.14778/3137765.3137789.

[71] M. Baranowski, A. Belloum, M. Bubak, and M. Malawski, «Constructing workflows from script applications», *Sci. Program.*, vol. 20, no. 4, pp. 359–377, 2012. DOI: 10.3233/SPR-120358.

[72] M. Malawski, T. Gubala, M. Kasztelnik, T. Bartynski, M. Bubak, F. Baude, and 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. DOI: 10.1007/978-0-387-78448-9_25.

[73] A. R. Runnalls and C. A. Silles, «Provenance Tracking in R», in *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*, ser. Lecture Notes in Computer Science, vol. 7525, Springer, 2012, pp. 237–239. DOI: 10.1007/978-3-642-34222-6_25.

[74] D. Tariq, M. Ali, and A. Gehani, «Towards Automated Collection of Application-Level Data Provenance», in *4th Workshop on the Theory and Practice of Provenance, TaPP'12, Boston, MA, USA, June 14-15, 2012*, U. A. Acar and T. J. Green, Eds., USENIX Association, 2012. [Online]. Available: https://www.usenix.org/conference/tapp12/workshop-program/presentation/tariq.

[75] A. S. Tanenbaum and M. van Steen, *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007, ISBN: 978-0-13-239227-3.

[76] M. P. Forum, «MPI: A Message-Passing Interface Standard», University of Tennessee, USA, Tech. Rep., 1994.

[77] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V. Y. Ye, «Building LinkedIn's Real-time Activity Data Pipeline», *IEEE Data Eng. Bull.*, vol. 35, no. 2, pp. 33–45, 2012. [Online]. Available: http://sites.computer.org/debull/A12june/pipeline.pdf.

[78] S. Gilbert and N. A. Lynch, «Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services», *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002. DOI: 10.1145/564585.564601.

[79] I. Foster, C. Kesselman, and S. Tuecke, «The Anatomy of the Grid: Enabling Scalable Virtual Organization», *The Intl. Journal of High Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, Fall 2001, ISSN: 1094-3420.

[80] I. Foster, «What is the Grid? A Three Point Checklist», Jul. 2002.

[81] I. T. Foster and C. Kesselman, «Globus: a Metacomputing Infrastructure Toolkit», *Int. J. High Perform. Comput. Appl.*, vol. 11, no. 2, pp. 115–128, 1997. DOI: 10.1177/109434209701100205.

[82] K. Czajkowski, I. T. Foster, N. T. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, «A Resource Management Architecture for Metacomputing Systems», in *Job Scheduling Strategies for Parallel Processing, IPPS/SPDP'98 Workshop, Orlando, Florida, USA, March 30, 1998, Proceedings*, D. G. Feitelson and L. Rudolph, Eds., ser. Lecture Notes in Computer Science, vol. 1459, Springer, 1998, pp. 62–82. DOI: 10.1007/BFb0053981.

[83] S. Fitzgerald, I. T. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke, «A Directory Service for Configuring High-Performance Distributed Computations», in *Proceedings of the 6th International Symposium on High Performance Distributed Computing, HPDC '97, Portland, OR, USA, August 5-8, 1997*, IEEE Computer Society, 1997, pp. 365–376. DOI: 10.1109/HPDC.1997.626445.

[84] I. T. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, «A Security Architecture for Computational Grids», in *CCS '98, Proceedings of the 5th ACM Conference on Computer and Communications Security, San Francisco, CA, USA, November 3-5, 1998*, L. Gong and M. K. Reiter, Eds., ACM, 1998, pp. 83–92. DOI: 10.1145/288090.288111.

[85] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, and I. T. Foster, «Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing», in *18th IEEE Symposium on Mass Storage Systems, MSS 2001: Large Scale Storage in the Web, San Diego, CA, USA, April 17-20, 2001*, IEEE Computer Society, 2001, pp. 13–28. DOI: 10.1109/MSS.2001.10001.

[86] I. Foster, Y. Zhao, I. Raicu, and S. Lu, «Cloud Computing and Grid Computing 360-Degree Compared», in *2008 Grid Computing Environments Workshop*, IEEE Computer Society, 2008, pp. 1–10. DOI: 10.1109/GCE.2008.4738445.

[87] K. Keahey, I. T. Foster, T. Freeman, and X. Zhang, «Virtual workspaces: Achieving quality of service and quality of life in the Grid», *Sci. Program.*, vol. 13, no. 4, pp. 265–275, 2005. DOI: 10.1155/2005/351408.

[88] J. Shiers, «The Worldwide LHC Computing Grid (worldwide LCG)», *Comput. Phys. Commun.*, vol. 177, no. 1-2, pp. 219–223, 2007. DOI: 10.1016/j.cpc.2007.02.021.

[89] J. J. Dongarra, «Trends in High-Performance Computing», in *Handbook of Nature-Inspired and Innovative Computing - Integrating Classical Models with Emerging Technologies*, Springer, 2006, pp. 511–520. DOI: 10.1007/0-387-27705-6_15.

[90] J. J. Dongarra, P. Luszczek, and A. Petitet, «The LINPACK Benchmark: past, present and future», *Concurr. Comput. Pract. Exp.*, vol. 15, no. 9, pp. 803–820, 2003. DOI: 10.1002/cpe.728.

[91] J. Shalf, S. S. Dosanjh, and J. Morrison, «Exascale Computing Technology Challenges», in *High Performance Computing for Computational Science - VECPAR 2010 - 9th International conference, Berkeley, CA, USA, June 22-25, 2010, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 6449, Springer, 2010, pp. 1–25. DOI: 10.1007/978-3-642-19328-6_1.

[92] S. Borkar and A. A. Chien, «The future of microprocessors», *Commun. ACM*, vol. 54, no. 5, pp. 67–77, 2011. DOI: 10.1145/1941487.1941507.

[93] J. J. Dongarra, P. H. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. M. Chapman, X. Chi, A. N. Choudhary, S. S. Dosanjh, T. H. Dunning, S. Fiore, A. Geist, B. Gropp, R. J. Harrison, M. Hereld, M. A. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. E. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Müller, W. E. Nagel, H. Nakashima, M. E. Papka, D. A. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. L. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. M. Tang, J. Taylor, R. Thakur, A. E. Trefethen, M. Valero, A. J. van der Steen, J. S. Vetter, P. Williams, R. W. Wisniewski, and K. A. Yelick, «The International Exascale Software Project roadmap», *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, 2011. DOI: 10.1177/1094342010391989.

[94] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, «An extended set of FORTRAN basic linear algebra subprograms», *ACM Trans. Math. Softw.*, vol. 14, no. 1, pp. 1–17, 1988. DOI: 10.1145/42288.42291.

[95] E. Anderson, Z. Bai, C. H. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen, *LAPACK Users' Guide, Third Edition*, ser. Software, Environments and Tools. SIAM, 1999, ISBN: 978-0-89871-447-0. DOI: 10.1137/1.9780898719604.

[96] P. Viviani, M. Aldinucci, M. Torquati, and R. d'Ippolito, «Multiple back-end support for the armadillo linear algebra interface», in *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, ACM, 2017, pp. 1566–1573. DOI: 10.1145/3019612.3019743.

[97] M. J. Flynn, «Some Computer Organizations and Their Effectiveness», *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sep. 1972. DOI: 10.1109/TC.1972.5009071.

[98] R. Rabenseifner, G. Hager, and G. Jost, «Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes», in *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2009, Weimar, Germany, 18-20 Febuary 2009*, IEEE Computer Society, 2009, pp. 427–436. DOI: 10.1109/PDP.2009.43.

[99] S. J. Pennycook, S. D. Hammond, S. A. Jarvis, and G. R. Mudalige, «Performance analysis of a hybrid MPI/CUDA implementation of the NASLU benchmark», *SIGMETRICS Perform. Evaluation Rev.*, vol. 38, no. 4, pp. 23–29, 2011. DOI: 10.1145/1964218.1964223.

[100] A. Sergeev and M. D. Balso, «Horovod: fast and easy distributed deep learning in TensorFlow», *CoRR*, vol. abs/1802.05799, 2018. arXiv: 1802.05799.

[101] T. Gamblin, M. P. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, «The Spack package manager: bringing order to HPC software chaos», in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, ACM, 2015, 40:1–40:12. DOI: 10.1145/2807591.2807623.

[102] G. M. Kurtzer, V. Sochat, and M. W. Bauer, «Singularity: Scientific containers for mobility of compute», *PLOS ONE*, vol. 12, no. 5, pp. 1–20, May 2017. DOI: 10.1371/journal.pone.0177459.

[103] P. Braam, «The Lustre Storage Architecture», *CoRR*, vol. abs/1903.01955, 2019. arXiv: 1903.01955.

[104] A. B. Yoo, M. A. Jette, and M. Grondona, «SLURM: Simple Linux Utility for Resource Management», in *Job Scheduling Strategies for Parallel Processing, 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003, Revised Papers*, ser. Lecture Notes in Computer Science, vol. 2862, Springer, 2003, pp. 44–60. DOI: 10.1007/10968987_3.

[105] P. Mell and T. Grance, *The NIST Definition of Cloud Computing*, en, Sep. 2011. DOI: 10.6028/NIST.SP.800-145.

[106] C. Miyachi, «What is "Cloud"? It is time to update the NIST definition?», *IEEE Cloud Comput.*, vol. 5, no. 3, pp. 6–11, 2018. DOI: 10.1109/MCC.2018.032591611.

[107] M. Ribeiro, K. Grolinger, and M. A. M. Capretz, «MLaaS: Machine Learning as a Service», in *14th IEEE International Conference on Machine Learning and Applications, ICMLA 2015, Miami, FL, USA, December 9-11, 2015*, IEEE, 2015, pp. 896–902. DOI: 10.1109/ICMLA.2015.152.

[108] C. A. Ardagna, P. Ceravolo, and E. Damiani, «Big data analytics as-a-service: Issues and challenges», in *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, IEEE Computer Society, 2016, pp. 3638–3644. DOI: 10.1109/BigData.2016.7841029.

[109] T. C. Chieu, A. A. Karve, A. Mohindra, and A. Segal, «Simplifying solution deployment on a Cloud through composite appliances», in *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Workshop Proceedings*, 2010, pp. 1–5. DOI: 10.1109/IPDPSW.2010.5470721.

[110] A. Lenk, C. Dänschel, M. Klems, D. Bermbach, and T. Kurze, «Requirements for an IaaS deployment language in federated Clouds», in *2011 IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2011, Irvine, CA, USA, December 12-14, 2011*, 2011, pp. 1–4. DOI: 10.1109/SOCA.2011.6166249.

[111] M. Caballer, J. D. S. Quilis, G. Moltó, and I. Blanquer, «A platform to deploy customized scientific virtual infrastructures on the cloud», *Concurr. Comput. Pract. Exp.*, vol. 27, no. 16, pp. 4318–4329, 2015. DOI: 10.1002/cpe.3518.

[112] A. V. Konstantinou, T. Eilam, M. H. Kalantar, A. Totok, W. C. Arnold, and E. C. Snible, «An architecture for virtual solution composition and deployment in infrastructure clouds», in *Proceedings of the 3rd International Workshop on Virtualization Technologies in Distributed Computing, VTDC@ICAC 2009, Barcelona, Spain, June 15, 2009*, 2009, pp. 9–18. DOI: 10.1145/1555336.1555339.

[113] P. Anderson, «Towards a High-Level Machine Configuration System», in *Proceedings of the 8th Conference on Systems Administration (LISA 1994), San Diego, California, USA, September 19-23, 1994*, USENIX, 1994. [Online]. Available: http://www.usenix.org/publications/library/proceedings/lisa94/anderson.html.

[114] M. Burgess, «A Site Configuration Engine», *Comput. Syst.*, vol. 8, no. 2, pp. 309–337, 1995. [Online]. Available: http://www.usenix.org/publications/compsystems/1995/sum%5C_burgess.pdf.

[115] J. Fischer, R. Majumdar, and S. Esmaeilsabzali, «Engage: a deployment management system», in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, ACM, 2012, pp. 263–274. DOI: 10.1145/2254064.2254096.

[116] D.-H. Le, H. L. Truong, G. Copil, S. Nastic, and S. Dustdar, «SALSA: A Framework for Dynamic Configuration of Cloud Services», in *IEEE 6th International Conference on Cloud Computing Technology and Science, CloudCom 2014, Singapore, December 15-18, 2014*, 2014, pp. 146–153. DOI: 10.1109/CloudCom.2014.99.

[117] L. M. Pham, A. Tchana, D. Donsez, N. D. Palma, V. Zurczak, and P.-Y. Gibello, «Roboconf: A Hybrid Cloud Orchestrator to Deploy Complex Applications», in *8th IEEE International Conference on Cloud Computing, CLOUD 2015, New York City, NY, USA, June 27 - July 2, 2015*, 2015, pp. 365–372. DOI: 10.1109/CLOUD.2015.56.

[118] R. Mietzner and F. Leymann, «Towards Provisioning the Cloud: On the Usage of Multi-Granularity Flows and Services to Realize a Unified Provisioning Infrastructure for SaaS Applications», in *2008 IEEE Congress on Services, Part I, SERVICES I 2008, Honolulu, Hawaii, USA, July 6-11, 2008*, IEEE Computer Society, 2008, pp. 3–10. DOI: 10.1109/SERVICES-1.2008.36.

[119] *Web Services Business Process Execution Language Version 2.0*, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, Apr. 2007.

[120] A. Brogi, A. Canciani, J. Soldani, and P. Wang, «A Petri Net-Based Approach to Model and Analyze the Management of Cloud Applications», *Trans. Petri Nets Other Model. Concurr.*, vol. 11, pp. 28–48, 2016. DOI: 10.1007/978-3-662-53401-4_2.

[121] *Topology and orchestration specification for cloud applications version 1.0*, http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html, Nov. 2013.

[122] J. Zhang, X. Lu, and D. K. Panda, «Is Singularity-based Container Technology Ready for Running MPI Applications on HPC Clouds?», in *Proceedings of the 10th International Conference on Utility and Cloud Computing, UCC 2017, Austin, TX, USA, December 5-8, 2017*, ACM, 2017, pp. 151–160. DOI: 10.1145/3147213.3147231.

[123] G. R. Alles, A. Carissimi, and L. M. Schnorr, «Assessing the Computation and Communication Overhead of Linux Containers for HPC Applications», in *Symposium on High Performance Computing Systems, WSCAD 2018, São Paulo, Brazil, October 1-3, 2018*, IEEE, 2018, pp. 116–123. DOI: 10.1109/WSCAD.2018.00027.

[124] J. A. T. Gomes, E. Bagnaschi, I. C. Plasencia, M. David, L. Alves, J. Martins, J. M. Pina, Á. L. García, and P. O. Fernández, «Enabling rootless Linux Containers in multi-user environments: The *udocker* tool», *Comput. Phys. Commun.*, vol. 232, pp. 84–97, 2018. DOI: 10.1016/j.cpc.2018.05.021.

[125] A. Azab, «Enabling Docker Containers for High-Performance and Many-Task Computing», in *2017 IEEE International Conference on Cloud Engineering, IC2E 2017, Vancouver, BC, Canada, April 4-7, 2017*, IEEE Computer Society, 2017, pp. 279–285. DOI: 10.1109/IC2E.2017.52.

[126] M. Aldinucci, S. Bagnasco, S. Lusso, P. Pasteris, and S. Rabellino, «OCCAM: a flexible, multi-purpose and extendable HPC cluster», in *Journal of Physics: Conf. Series (CHEP 2016)*, vol. 898, San Francisco, USA, 2017, p. 082 039. DOI: 10.1088/1742-6596/898/8/082039.

[127] D. M. Jacobsen and R. S. Canon, «Contain This, Unleashing Docker for HPC», in *Cray User Group (CUG 2015), Chicago, IL*, Apr. 2015.

[128] R. Priedhorsky and T. Randles, «Charliecloud: unprivileged containers for user-defined software stacks in HPC», in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*, ACM, 2017, 36:1–36:10. DOI: 10.1145/3126908.3126925.

[129] N. Kulkarni, L. Alessandrì, R. Panero, M. Arigoni, M. Olivero, G. Ferrero, F. Cordero, M. Beccuti, and R. A. Calogero, «Reproducible bioinformatics project: a community for reproducible bioinformatics analysis pipelines», *BMC Bioinformatics*, vol. 19, no. 10, p. 349, 2018, ISSN: 1471-2105. DOI: 10.1186/s12859-018-2296-x.

[130] D. E. Knuth, «Literate Programming», *Comput. J.*, vol. 27, no. 2, pp. 97–111, 1984. DOI: 10.1093/comjnl/27.2.97.

[131] 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, and 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. DOI: 10.3233/978-1-61499-649-1-87.

[132] F. Pérez and B. E. Granger, «IPython: A System for Interactive Scientific Computing», *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 21–29, 2007. DOI: 10.1109/MCSE.2007.53.

[133] J. M. Perkel, «Why Jupyter is data scientists' computational notebook of choice», *Nature*, vol. 563, no. 7729, pp. 145–146, Nov. 2018. DOI: 10.1038/d41586-018-07196-1.

[134] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, «PyTorch: An Imperative Style, High-Performance Deep Learning Library», in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 8024–8035. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[135] 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, and 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*, K. Keeton and T. Roscoe, Eds., USENIX Association, 2016, pp. 265–283.

[136] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, «MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems», *CoRR*, vol. abs/1512.01274, 2015. arXiv: 1512.01274.

[137] 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. DOI: 10.1145/2808719.2811433.

[138] A. P. e. a. Arkin, «KBase: The United States Department of Energy Systems Biology Knowledgebase», *Nature Biotechnology*, vol. 36, no. 7, pp. 566–569, 2018. DOI: 10.1038/nbt.4163.

[139] M. Reich, T. Liefeld, J. Gould, J. Lerner, P. Tamayo, and J. P. Mesirov, «GenePattern 2.0», *Nature Genetics*, vol. 38, no. 5, pp. 500–501, 2006. DOI: 10.1038/ng0506-500.

[140] G. Wang and B. Peng, «Script of Scripts: A pragmatic workflow system for daily computational research», *PLoS Comput. Biol.*, vol. 15, no. 2, 2019. DOI: 10.1371/journal.pcbi.1006843.

[141] L. A. M. C. Carvalho, R. Wang, Y. Gil, and D. Garijo, «NiW: Converting Notebooks into Workflows to Capture Dataflow and Provenance», in *Proceedings of Workshops and Tutorials of the 9th International Conference on Knowledge Capture (K-CAP2017), Austin, Texas, USA, December 4th*, I. Tiddi, G. Rizzo, and Ó. Corcho, Eds., ser. CEUR Workshop Proceedings, vol. 2065, CEUR-WS.org, 2017, pp. 12–16. [Online]. Available: http://ceur-ws.org/Vol-2065/paper04.pdf.

[142] Y. Gil, V. Ratnakar, J. Kim, P. A. González-Calero, P. Groth, J. Moody, and E. Deelman, «Wings: Intelligent Workflow-Based Design of Computational Experiments», *IEEE Intell. Syst.*, vol. 26, no. 1, pp. 62–72, 2011. DOI: 10.1109/MIS.2010.9.

[143] D. Koop and 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.

[144] S. Macke, A. G. Parameswaran, H. Gong, D. J. L. Lee, D. Xin, and A. Head, «Fine-Grained Lineage for Safer Notebook Interactions», *Proc. VLDB Endow.*, vol. 14, no. 6, pp. 1093–1101, 2021.

[145] M. Brachmann, W. Spoth, O. Kennedy, B. Glavic, H. Mueller, S. Castelo, C. Bautista, and 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, Online Proceedings*, www.cidrdb.org, 2020.

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

[147] D. Yin, Y. Liu, A. Padmanabhan, J. Terstriep, J. Rush, and S. Wang, «A CyberGIS-Jupyter Framework for Geospatial Analytics at Scale», in *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*, D. L. Hart and M. Dahan, Eds., ACM, 2017, 18:1–18:8. DOI: 10.1145/3093338.3093378.

[148] 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, and 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. DOI: 10.1109/HPEC.2017.8091097.

[149] M. Milligan, «Interactive HPC Gateways with Jupyter and Jupyterhub», in *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*, D. L. Hart and M. Dahan, Eds., ACM, 2017, 63:1–63:4. DOI: 10.1145/3093338.3104159.

[150] B. Glick and J. Mache, «Jupyter Notebooks and User-Friendly HPC Access», in *2018 IEEE/ACM Workshop on Education for High-Performance Computing, EduHPC@SC, Dallas, TX, USA, November 12, 2018*, IEEE, 2018, pp. 11–20. DOI: 10.1109/EduHPC.2018.00005.

[151] R. C. Thomas, S. Cholia, K. Mohror, and J. M. Shalf, «Interactive Supercomputing With Jupyter», *Comput. Sci. Eng.*, vol. 23, no. 2, pp. 93–98, 2021. DOI: 10.1109/MCSE.2021.3059037.

[152] T. E. Odaka, A. Banihirwe, G. Eynard-Bontemps, A. Ponte, G. Maze, K. Paul, J. Baker, and 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*, G. Juckeland and S. Chandrasekaran, Eds., ser. Communications in Computer and Information Science, vol. 1190, Springer, 2019, pp. 190–204. DOI: 10.1007/978-3-030-44728-1_12.

[153] G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn, «Observing Localities», *Theor. Comput. Sci.*, vol. 114, no. 1, pp. 31–61, 1993. DOI: 10.1016/0304-3975(93)90152-J.

[154] P. Krishnan, «Distributed CCS», in *CONCUR '91, 2nd International Conference on Concurrency Theory, Amsterdam, The Netherlands, August 26-29, 1991, Proceedings*, J. C. M. Baeten and J. F. Groote, Eds., ser. Lecture Notes in Computer Science, vol. 527, Springer, 1991, pp. 393–407. DOI: 10.1007/3-540-54430-5_102.

[155] M. Gasser and E. McDermott, «An Architecture for Practical Delegation in a Distributed System», in *Proceedings of the 1990 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 7-9, 1990*, IEEE Computer Society, 1990, pp. 20–30. DOI: 10.1109/RISP.1990.63835.

[156] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, «ZooKeeper: Wait-free Coordination for Internet-scale Systems», in *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, P. Barham and T. Roscoe, Eds., USENIX Association, 2010.

[157] H.-L. Bouziane, C. Pérez, and T. Priol, «A Software Component Model with Spatial and Temporal Compositions for Grid Infrastructures», in *Euro-Par 2008 - Parallel Processing, 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, August 26-29, 2008, Proceedings*, 2008, pp. 698–708. DOI: 10.1007/978-3-540-85451-7_75.

[158] M. Aldinucci, M. Danelutto, H. L. Bouziane, and C. Pérez, «Towards Software Component Assembly Language Enhanced with Workflows and Skeletons», in *Proc. of the ACM SIGPLAN Component-Based High Performance Computing (CBHPC)*, Karlsruhe, Germany: ACM, Oct. 2008, pp. 1–11, ISBN: 978-1-60558-311-2. DOI: 10.1145/1456190.1456194.

[159] M. Aldinucci, H.-L. Bouziane, M. Danelutto, and C. Pérez, «Stkm on Sca: A Unified Framework with Components, Workflows and Algorithmic Skeletons», in *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings*, 2009, pp. 678–690. DOI: 10.1007/978-3-642-03869-3_64.

[160] B. Kiepusewski, «Expressiveness and suitability of languages for control flow modelling in workflows», Ph.D. dissertation, Queensland University of Technology, Brisbane, Feb. 2003.

[161] W. M. P. van der Aalst and A. H. M. ter Hofstede, «YAWL: Yet Another Workflow Language», *Inf. Syst.*, vol. 30, no. 4, pp. 245–275, Jun. 2005, ISSN: 0306-4379. DOI: 10.1016/j.is.2004.02.002.

[162] W. M. P. van der Aalst, «Verification of Workflow Nets», in *Application and Theory of Petri Nets 1997, 18th International Conference, ICATPN '97, Toulouse, France, June 23-27, 1997, Proceedings*, P. Azéma and G. Balbo, Eds., ser. Lecture Notes in Computer Science, vol. 1248, Springer, 1997, pp. 407–426. DOI: 10.1007/3-540-63139-9_48.

[163] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn, «Soundness of workflow nets: classification, decidability, and analysis», *Formal Aspects Comput.*, vol. 23, no. 3, pp. 333–363, 2011. DOI: 10.1007/s00165-010-0161-4.

[164] S. Lerner, T. D. Millstein, E. Rice, and C. Chambers, «Automated soundness proofs for dataflow analyses and transformations via local rules», in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, J. Palsberg and M. Abadi, Eds., ACM, 2005, pp. 364–377. DOI: 10.1145/1040305.1040335.

[165] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros, «Workflow Patterns», *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, Jul. 2003, ISSN: 1573-7578. DOI: 10.1023/A:1022883727209.

[166] M. Aldinucci, M. Danelutto, L. Anardu, M. Torquati, and P. Kilpatrick, «Parallel patterns + Macro Data Flow for multi-core programming», in *Proc. of Intl. Euromicro PDP 2012: Parallel Distributed and network-based Processing*, Garching, Germany: IEEE, Feb. 2012, pp. 27–36. DOI: 10.1109/PDP.2012.44.

[167] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computations*, ser. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.

[168] M. Danelutto, R. D. Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi, «A methodology for the development and the support of massively parallel programs», *Future Generation Compututer Systems*, vol. 8, no. 1-3, pp. 205–220, 1992, ISSN: 0167-739X. DOI: 10.1016/0167-739X(92)90040-I.

[169] H. González-Vélez and M. Leyton, «A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers», *Softw., Pract. Exper.*, vol. 40, no. 12, pp. 1135–1160, 2010. DOI: 10.1002/spe.1026.

[170] 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, and A. Visa, «Programming languages for data-Intensive HPC applications: A systematic mapping study», *Parallel Computing*, p. 102 584, 2020, ISSN: 0167-8191. DOI: 10.1016/j.parco.2019.102584.

[171] G. X. Y. Zheng, J. M. Terry, P. Belgrader, P. Ryvkin, Z. W. Bent, R. Wilson, S. B. Ziraldo, T. D. Wheeler, G. P. McDermott, J. Zhu, M. T. Gregory, J. Shuga, L. Montesclaros, J. G. Underwood, D. A. Masquelier, S. Y. Nishimura, M. Schnall-Levin, P. W. Wyatt, C. M. Hindson, R. Bharadwaj, A. Wong, K. D. Ness, L. W. Beppu, H. J. Deeg, C. McFarland, K. R. Loeb, W. J. Valente, N. G. Ericson, E. A. Stevens, J. P. Radich, T. S. Mikkelsen, B. J. Hindson, and J. H. Bielas, «Massively parallel digital transcriptional profiling of single cells», *Nature communications*, vol. 8, pp. 14 049–14 049, Jan. 2017, ISSN: 2041-1723. DOI: 10.1038/ncomms14049.

[172] A. Butler, P. Hoffman, P. Smibert, E. Papalexi, and R. Satija, «Integrating single-cell transcriptomic data across different conditions, technologies, and species», *Nature Biotechnology*, vol. 36, no. 5, pp. 411–420, 2018, ISSN: 1546-1696. DOI: 10.1038/nbt.4096.

[173] T. Stuart, A. Butler, P. Hoffman, C. Hafemeister, E. Papalexi, W. M. I. Mauck, Y. Hao, M. Stoeckius, P. Smibert, and R. Satija, «Comprehensive Integration of Single-Cell Data», *Cell*, vol. 177, no. 7, 1888–1902.e21, Jun. 2019, ISSN: 0092-8674. DOI: 10.1016/j.cell.2019.05.031.

[174] D. Aran, A. P. Looney, L. Liu, E. Wu, V. Fong, A. Hsu, S. Chak, R. P. Naikawadi, P. J. Wolters, A. R. Abate, A. J. Butte, and M. Bhattacharya, «Reference-based analysis of lung single-cell sequencing reveals a transitional profibrotic macrophage», *Nature Immunology*, vol. 20, no. 2, pp. 163–172, 2019, ISSN: 1529-2916. DOI: 10.1038/s41590-018-0276-y.

[175] G. Schiroli, A. Conti, S. Ferrari, L. della Volpe, A. Jacob, L. Albano, S. Beretta, A. Calabria, V. Vavassori, P. Gasparini, E. Salataj, D. Ndiaye-Lobry, C. Brombin, J. Chaumeil, E. Montini, I. Merelli, P. Genovese, L. Naldini, and R. Di Micco, «Precise Gene Editing Preserves Hematopoietic Stem Cell Function following Transient p53-Mediated DNA Damage Response», *Cell Stem Cell*, vol. 24, no. 4, 551–565.e8, Apr. 2019, ISSN: 1934-5909. DOI: 10.1016/j.stem.2019.02.019.

[176] I. Colonnelli, B. Cantalupo, R. Esposito, M. Pennisi, C. Spampinato, and 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, January 19, Budapest, Hungary*, ser. OASIcs, vol. 88, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 5:1–5:13. DOI: 10.4230/OASIcs.PARMA-DITAM.2021.5.

[177] I. Colonnelli, B. Cantalupo, C. Spampinato, M. Pennisi, and M. Aldinucci, «Bringing AI pipelines onto cloud-HPC: setting a baseline for accuracy of COVID-19 diagnosis», in *ENEA CRESCO in the fight against COVID-19*, F. Iannone, Ed., ENEA, 2021. DOI: 10.5281/zenodo.5151511.

[178] M. Aldinucci, *High-performance computing and AI team up for COVID-19 diagnostic imaging*, https://aihub.org/2021/01/12/high-performance-computing-and-ai-team-up-for-covid-19-diagnostic-imaging/, Accessed: 2021-01-25, Jan. 2021.

[179] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, «Rethinking Atrous Convolution for Semantic Image Segmentation», *CoRR*, vol. abs/1706.05587, 2017. arXiv: 1706.05587.

[180] O. Ronneberger, P. Fischer, and T. Brox, «U-Net: Convolutional Networks for Biomedical Image Segmentation», in *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015 - 18th International Conference Munich, Germany, October 5 - 9, 2015, Proceedings, Part III*, ser. Lecture Notes in Computer Science, vol. 9351, Springer, 2015, pp. 234–241. DOI: 10.1007/978-3-319-24574-4_28.

[181] S. Jégou, M. Drozdzal, D. Vázquez, A. Romero, and Y. Bengio, «The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation», in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2017, Honolulu, HI, USA, July 21-26, 2017*, IEEE Computer Society, 2017, pp. 1175–1183. DOI: 10.1109/CVPRW.2017.156.

[182] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, «Going deeper with convolutions», in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, IEEE Computer Society, 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594.

[183] A. Krizhevsky, I. Sutskever, and G. E. Hinton, «ImageNet classification with deep convolutional neural networks», *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017. DOI: 10.1145/3065386.

[184] K. He, X. Zhang, S. Ren, and J. Sun, «Deep Residual Learning for Image Recognition», in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, IEEE Computer Society, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[185] G. Huang, Z. Liu, L. van der Maaten, and 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. DOI: 10.1109/CVPR.2017.243.

[186] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, «Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning», in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, AAAI Press, 2017, pp. 4278–4284. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14806.

[187] 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, and J. M. Salinas, «BIMCV COVID-19+: a large annotated dataset of RX and CT images from COVID-19 patients», *CoRR*, vol. abs/2006.01174, 2020. arXiv: 2006.01174.

[188] M. Pennisi, I. Kavasidis, C. Spampinato, V. Schinina, S. Palazzo, F. P. Salanitri, G. Bellitto, F. Rundo, M. Aldinucci, M. Cristofaro, *et al.*, «An Explainable AI System for Automated COVID-19 Assessment and Lesion Categorization from CT-scans», *Artificial Intelligence in Medicine*, p. 102 114, 2021. DOI: 10.1016/j.artmed.2021.102114.

[189] A. J. Bernstein, «Program Analysis for Parallel Processing», *IEEE Trans. on Electronic Computers*, vol. EC-15, no. 5, pp. 757–762, 1966.

[190] J. Darlington, Y. Guo, H. W. To, and J. Yang, «Functional Skeletons for Parallel Coordination», in *Euro-Par '95 Parallel Processing, First International Euro-Par Conference, Stockholm, Sweden, August 29-31, 1995, Proceedings*, ser. Lecture Notes in Computer Science, vol. 966, Springer, 1995, pp. 55–66. DOI: 10.1007/BFb0020455.

[191] M. M. McKerns, L. Strand, T. Sullivan, A. Fang, and M. A. G. Aivazis, «Building a Framework for Predictive Science», *CoRR*, vol. abs/1202.1056, 2012.

[192] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati, «Targeting Distributed Systems in FastFlow», in *Euro-Par 2012 Workshops, Proc. of the CoreGrid Workshop on Grids, Clouds and P2P Computing*, ser. LNCS, vol. 7640, Springer, 2013, pp. 47–56. DOI: 10.1007/978-3-642-36949-0_7.

[193] 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, and R. M. Wentzcovitch, «QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials», *Journal of Physics: Condensed Matter*, vol. 21, no. 39, p. 395 502, Sep. 2009. DOI: 10.1088/0953-8984/21/39/395502.

[194] Auton *et al.*, «A global reference for human genetic variation», *Nature*, vol. 526, no. 7571, pp. 68–74, Oct. 2015, ISSN: 1476-4687. DOI: 10.1038/nature15393.

[195] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and L. Fei-Fei, «ImageNet Large Scale Visual Recognition Challenge», *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015. DOI: 10.1007/s11263-015-0816-y.

[196] D. P. Kingma and J. Ba, «Adam: A Method for Stochastic Optimization», in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980.

[197] H. Xiao, K. Rasul, and R. Vollgraf, «Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms», *CoRR*, vol. abs/1708.07747, 2017.

[198] 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, and S. Baroni, «Quantum ESPRESSO toward the exascale», *The Journal of Chemical Physics*, vol. 152, no. 15, p. 154 105, 2020. DOI: 10.1063/5.0005082.

[199] R. F. da Silva, R. Filgueira, E. Deelman, E. Pairo-Castineira, I. M. Overton, and M. P. Atkinson, «Using simple PID-inspired controllers for online resilient resource management of distributed scientific workflows», *Future Generation Computer Systems*, vol. 95, pp. 615–628, 2019. doi: 10.1016/j.future.2019.01.015.

Acronyms

AI Artificial Intelligence

API Application Programmable Interface

AST Abstract Syntax Tree

BDAaaS Big Data Analytics as a Service

CFD Computational Fluid Dynamics

CI Continuous Integration

CLI Command Line Interface

CNN Convolutional Neural Network

CT Computed Tomography

CWL Common Workflow Language

DAG Directed Acyclic Graph

DFS Distributed File-System

DL Deep Learning

DNN Deep Neural Network

DSL Domain-Specific Language

FIFO First In First Out

GPU Graphics Processing Unit

GUI Graphical User Interface

HPC High-Performance Computing

HTC High Throughput Computing

IaaS Infrastructure as a Service

IaC Infrastructure as Code

IDE Integrated Development Environment

ML Machine Learning

MLaaS Machine Learning as a Service

MPI Message Passing Interface

MTMC Multiple-Tasks Multiple-Containers

MTSC Multiple-Tasks Single-Container

NVMe Non-Volatile Memory Express

PaaS Platform as a Service

PGAS Partitioned Global Address Space

RDMA Remote Direct Memory Access

SaaS Software as a Service

scRNA-seq single-cell RNA sequencing

SDN Software-Defined Network

SPMD Single Program Multiple Data

SSH Secure SHell

STMC Single-Task Multiple-Containers

STSC Single-Task Single-Container

UML Unified Modeling Language

VM Virtual Machine

VO Virtual Organisation

WDL Workflow Description Language

WMS Workflow Management System

XaaS Anything as a Service