UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC)
BARCELONATECH

COMPUTER ARCHITECTURE DEPARTMENT (DAC)

Programming models to support Data Science workflows

PH.D. THESIS

2020 | SPRING SEMESTER

Author:

Cristián RAMÓN-CORTÉS VILARRODONA
cristian.ramoncortes@bsc.es

Advisors:

Dra. Rosa M. BADIA SALA
rosa.m.badia@bsc.es

Dr. Jorge EJARQUE ARTIGAS
jorge.ejarque@bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

"Apenas él le amalaba el noema, a ella se le agolpaba el clémiso y caían en hidromurias, en salvajes ambonios, en sustalos exasperantes. Cada vez que él procuraba relamar las incopelusas, se enredaba en un grimado quejumbroso y tenía que envulsionarse de cara al nóvalo, sintiendo cómo poco a poco las arnillas se espejunaban, se iban apeltronando, reduplimiendo, hasta quedar tendido como el trimalciato de ergomanina al que se le han dejado caer unas fílulas de cariaconcia. Y sin embargo era apenas el principio, porque en un momento dado ella se tordulaba los hurgalios, consintiendo en que él aproximara suavemente sus orfelunios. Apenas se entreplumaban, algo como un ulucordio los encrestoriaba, los extrayuxtaba y paramovía, de pronto era el clinón, la esterfurosa convulcante de las mátricas, la jadehollante embocapluvia del orgumio, los esproemios del merpasmo en una sobrehumítica agopausa. ¡Evohé! ¡Evohé! Volposados en la cresta del murelio, se sentían balpamar, perlinos y márulos. Temblaba el troc, se vencían las marioplumas, y todo se resolviraba en un profundo pínice, en niolamas de argutendidas gasas, en carinias casi crueles que los ordopenaban hasta el límite de las gunfias."

Julio Cortázar,
Rayuela

Dedication

This work would not have been possible without the effort and patience of the people around me. Thank you for encouraging me to get to this point and sharing so many great moments on the way.

To Laura, for her kindness and devotion, and for supporting me through thick and thin despite my difficult character.

Special thanks to my loving mother and father, who have always guided and encouraged me to never stop learning; and who bought me my first computer.

Last but not least, to my sweet little sister, Marta, who always stands on my side and whose affection and support keeps me always up.

Wholeheartedly,
Cristián Ramón-Cortés Vilarrodona

Declaration of authorship

I hereby declare that, except where specific reference is made to the work of others, this thesis dissertation has been composed by me and it is based on my own work. None of the contents of this dissertation has been previously published nor submitted, in whole or in part, to any other examination in this or any other university.

Signed: ______________________________

Date: ______________________________

Acknowledgements

I gratefully thank my supervisors Rosa M. Badia Sala and Jorge Ejarque Artigas for all their assistance during my career at the *Barcelona Supercomputing Center (BSC-CNS)* and for giving me the opportunity to collaborate on this project.

I also thank all my colleagues, current and former members of the *Workflows and Distributed Computing* team from the *Barcelona Supercomputing Center (BSC)*, for their useful comments, remarks, and engagement through the learning process of this thesis: Francesc Lordan, Javier Alvarez, Francisco Javier Conejero, Daniele Lezzi, Ramon Amela, Pol Alvarez, Sergio Rodríguez, Carlos Segarra, Marc Dominguez, Salvi Solà, Hatem Elshazly, and Nihad Mammadli.

On the other hand, this thesis has been financed by the Ministry of Economy and Competitiveness with a predoctoral grant under contract BES-2016-076791. This work has also been partially supported by the Spanish Government (SEV2015-0493); the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P); the Generalitat de Catalunya (contract 2014-SGR-1051); the Joint Laboratory for Extreme Scale Computing (JLESC); the Intel-BSC Exascale Lab; and the European Commission through the Horizon 2020 Research and Innovation program under contracts 687584 (TANGO project), 676556 (MuG project), 690116 (EUBra-BIGSEA project), 671591 (NEXTGenIO project), and 800898 (ExaQUte project).

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) BARCELONATECH
Computer Architecture Department (DAC)

Abstract

Programming models to support Data Science workflows

by Cristián RAMÓN-CORTÉS VILARRODONA

Data Science workflows have become a must to progress in many scientific areas such as life, health, and earth sciences. In contrast to traditional HPC workflows, they are more heterogeneous; combining binary executions, MPI simulations, multi-threaded applications, custom analysis (possibly written in Java, Python, C/C++ or R), and real-time processing. Furthermore, in the past, field experts were capable of programming and running small simulations. However, nowadays, simulations requiring hundreds or thousands of cores are widely used and, to this point, efficiently programming them becomes a challenge even for computer sciences. Thus, programming languages and models make a considerable effort to ease the programmability while maintaining acceptable performance.

This thesis contributes to the adaptation of High-Performance frameworks to support the needs and challenges of Data Science workflows by extending COMPSs, a mature, general-purpose, task-based, distributed programming model. First, we enhance our prototype to orchestrate different frameworks inside a single programming model so that non-expert users can build complex workflows where some steps require highly optimised state of the art frameworks. This extension includes the `@binary`, `@OmpSs`, `@MPI`, `@COMPSs`, and `@MultiNode` annotations for both Java and Python workflows.

Second, we integrate container technologies to enable developers to easily port, distribute, and scale their applications to distributed computing platforms. This combination provides a straightforward methodology to parallelise applications from sequential codes along with efficient image management and application deployment that ease the packaging and distribution of applications. We distinguish between static, HPC, and dynamic container management and provide representative use cases for each scenario using Docker, Singularity, and Mesos.

Third, we design, implement and integrate AutoParallel, a Python module to automatically find an appropriate task-based parallelisation of affine loop nests and execute them in parallel in a distributed computing infrastructure. It is based on sequential programming and requires one single annotation (the `@parallel` Python decorator) so that anyone with intermediate-level programming skills can scale up an application to hundreds of cores.

Finally, we propose a way to extend task-based management systems to support continuous input and output data to enable the combination of task-based workflows and dataflows (Hybrid Workflows) using one single programming model. Hence, developers can build complex Data Science workflows with different approaches depending on the requirements without the effort of combining several frameworks at the same time. Also, to illustrate the capabilities of Hybrid Workflows, we have built a Distributed Stream Library that can be easily integrated with existing task-based frameworks to provide support for dataflows. The library provides a homogeneous, generic, and simple representation of object and file streams in both Java and Python; enabling complex workflows to handle any data type without dealing directly with the streaming back-end.

Keywords: Distributed Computing, High-Performance Computing, Data Science pipelines, Task-based Workflows, Dataflows, Containers, COMPSs, PyCOMPSs, AutoParallel, Docker, Pluto, Kafka

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) BARCELONATECH
Computer Architecture Department (DAC)

Resumen

Programming models to support Data Science workflows

por Cristián RAMÓN-CORTÉS VILARRODONA

Los flujos de trabajo de Data Science se han convertido en una necesidad para progresar en muchas áreas científicas como las ciencias de la vida, la salud y la tierra. A diferencia de los flujos de trabajo tradicionales para la CAP, los flujos de Data Science son más heterogéneos; combinando la ejecución de binarios, simulaciones MPI, aplicaciones multiproceso, análisis personalizados (posiblemente escritos en Java, Python, C/C++ o R) y computaciones en tiempo real. Mientras que en el pasado los expertos de cada campo eran capaces de programar y ejecutar pequeñas simulaciones, hoy en día, estas simulaciones representan un desafío incluso para los expertos ya que requieren cientos o miles de núcleos. Por esta razón, los lenguajes y modelos de programación actuales se esfuerzan considerablemente en incrementar la programabilidad manteniendo un rendimiento aceptable.

Esta tesis contribuye a la adaptación de modelos de programación para la CAP para afrontar las necesidades y desafíos de los flujos de Data Science extendiendo COMPSs, un modelo de programación distribuida maduro, de propósito general, y basado en tareas. En primer lugar, mejoramos nuestro prototipo para orquestar diferentes software para que los usuarios no expertos puedan crear flujos complejos usando un único modelo donde algunos pasos requieran tecnologías altamente optimizadas. Esta extensión incluye las anotaciones de `@binary`, `@OmpSs`, `@MPI`, `@COMPSs`, y `@MultiNode` para flujos en Java y Python.

En segundo lugar, integramos tecnologías de contenedores para permitir a los desarrolladores portar, distribuir y escalar fácilmente sus aplicaciones en plataformas distribuidas. Además de una metodología sencilla para paralelizar aplicaciones a partir de códigos secuenciales, esta combinación proporciona una gestión de imágenes y una implementación de aplicaciones eficientes que facilitan el empaquetado y la distribución de aplicaciones. Distinguimos entre gestión de contenedores estática, CAP y dinámica y proporcionamos casos de uso representativos para cada escenario con Docker, Singularity y Mesos.

En tercer lugar, diseñamos, implementamos e integramos AutoParallel, un módulo de Python para determinar automáticamente la paralelización basada en tareas de nidos de bucles afines y ejecutarlos en paralelo en una infraestructura distribuida. AutoParallel está basado en programación secuencial, requiere una sola anotación (el decorador `@parallel`) y permite a un usuario intermedio escalar una aplicación a cientos de núcleos.

Finalmente, proponemos una forma de extender los sistemas basados en tareas para admitir datos de entrada y salida continuos; permitiendo así la combinación de flujos de trabajo y datos (Flujos Híbridos) en un único modelo. Consecuentemente, los desarrolladores pueden crear flujos complejos siguiendo diferentes patrones sin el esfuerzo de combinar varios modelos al mismo tiempo. Además, para ilustrar las capacidades de los Flujos Híbridos, hemos creado una biblioteca (*DistroStreamLib*) que se integra fácilmente a los modelos basados en tareas para soportar flujos de datos. La bilblioteca proporciona una representación homogénea, genérica y simple de secuencias continuas de objetos y archivos en Java y Python; permitiendo manejar cualquier tipo de datos sin tratar directamente con el back-end de streaming.

Palabras clave: Computación Distribuida, Computación de Altas Prestaciones, Flujos de Data Science, Flujos de Tareas, Flujos de Datos, Containers, COMPSs, PyCOMPSs, AutoParallel, Docker, Pluto, Kafka

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) BARCELONATECH
Computer Architecture Department (DAC)

Resum

Programming models to support Data Science workflows

per Cristián RAMÓN-CORTÉS VILARRODONA

Els fluxos de treball de Data Science s'han convertit en una necessitat per progressar en moltes àrees científiques com les ciències de la vida, la salut i la terra. A diferència dels fluxos de treball tradicionals per a la CAP, els fluxos de Data Science són més heterogenis; combinant l'execució de binaris, simulacions MPI, aplicacions multiprocés, anàlisi personalitzats (possiblement escrits en Java, Python, C / C ++ o R) i computacions en temps real. Mentre que en el passat els experts de cada camp eren capaços de programar i executar petites simulacions, avui dia, aquestes simulacions representen un repte fins i tot per als experts ja que requereixen centenars o milers de nuclis. Per aquesta raó, els llenguatges i models de programació actuals s'esforcen considerablement en incrementar la programabilitat mantenint un rendiment acceptable.

Aquesta tesi contribueix a l'adaptació de models de programació per a la CAP per afrontar les necessitats i reptes dels fluxos de Data Science estenent COMPSs, un model de programació distribuïda madur, de propòsit general, i basat en tasques. En primer lloc, millorem el nostre prototip per orquestrar diferent programari per a que els usuaris no experts puguin crear fluxos complexos usant un únic model on alguns passos requereixin tecnologies altament optimitzades. Aquesta extensió inclou les anotacions de `@binary`, `@OmpSs`, `@MPI`, `@COMPSs`, i `@MultiNode` per a fluxos en Java i Python.

En segon lloc, integrem tecnologies de contenidors per permetre als desenvolupadors portar, distribuir i escalar fàcilment les seves aplicacions en plataformes distribuïdes. A més d'una metodologia senzilla per a paral·lelitzar aplicacions a partir de codis seqüencials, aquesta combinació proporciona una gestió d'imatges i una implementació d'aplicacions eficients que faciliten l'empaquetat i la distribució d'aplicacions. Distingim entre la gestió de contenidors estàtica, CAP i dinàmica i proporcionem casos d'ús representatius per a cada escenari amb Docker, Singularity i Mesos.

En tercer lloc, dissenyem, implementem i integrem AutoParallel, un mòdul de Python per determinar automàticament la paral·lelització basada en tasques de nius de bucles afins i executar-los en paral·lel en una infraestructura distribuïda. AutoParallel està basat en programació seqüencial, requereix una sola anotació (el decorador @parallel) i permet a un usuari intermig escalar una aplicació a centenars de nuclis.

Finalment, proposem una forma d'estendre els sistemes basats en tasques per admetre dades d'entrada i sortida continus; permetent així la combinació de fluxos de treball i dades (Fluxos Híbrids) en un únic model. Conseqüentment, els desenvolupadors poden crear fluxos complexos seguint diferents patrons sense l'esforç de combinar diversos models al mateix temps. A més, per a il·lustrar les capacitats dels Fluxos Híbrids, hem creat una biblioteca (DistroStreamLib) que s'integra fàcilment amb els models basats en tasques per suportar fluxos de dades. La biblioteca proporciona una representació homogènia, genèrica i simple de seqüències contínues d'objectes i arxius en Java i Python; permetent gestionar qualsevol tipus de dades sense tractar directament amb el back-end de streaming.

Paraules clau: Computació Distribuïda, Computació d'Altes Prestacions, Fluxos de Data Science, Fluxos de Tasques, Fluxos de Dades, Containers, COMPSs, PyCOMPSs, AutoParallel, Docker, Pluto, Kafka

Contents

List of Figures

List of Listings

List of Tables

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
BDA	**Big Data Analytics**
COMPSs	**COMP S**uperscalar
CPU	**C**entral **P**rocessing **U**nit
DAG	**Directed Acyclic Graph**
GUI	**G**raphical **U**ser **I**nterface
HPC	**High Performance Computing**
I/O	**I**nput / **O**utput
MPI	**M**essage **P**assing **I**nterface
NEMS	**N**OAA **E**nvironmental **M**odeling **S**ystem
NMMB	**N**onhydrostatic **M**ultiscale **M**odel on the **B**-grid
NOAA	**N**ational **O**ceanic and **A**tmospheric **A**dministration
OmpSs	**O**pen**MP** Star**Ss**
OS	**O**perating **S**ystem
PyCOMPSs	**Py**thon binding for **COMP S**uperscalar
RAM	**R**andom **A**ccess **M**emory
SSH	**S**ecure **SH**ell
VM	**V**irtual **M**achine
WSDL	**W**eb **S**ervices **D**escription **L**anguage

Glossary

API
An Application Programming Interface (API) is a set of methods and functions that are used by other software to produce an abstraction layer.

Binary
A file containing a list of machine code instructions to perform a list of tasks.

Big Data Analytics
The process of analysing Big Data (large data sets) to find patterns or correlations.

Container
A lightweight, standalone, executable package of software. Their purpose is to contain applications with all the parts they need but keeping them isolated from the host system that they run on. They are designed to provide a consistent and replicable environment.

Core
An individual processor that actually executes program instructions. Current single chip CPUs contain many cores and are referred to as multi-processor or multi-cores.

CPU
The Central Processing Unit (CPU) is the part of the computer that contains all the elements required to execute the instructions of software programs. Its main components are the main memory, the Processing Unit (PU) and the Control Unit (CU). Modern computers use multi-core processors, which are a single chip containing one or more cores.

Distributed Computing
Field of computer science that studies Distributed Systems.

Distributed System
Set of components located on different networked computers that must communicate and coordinate to execute a common set of actions (e.g., application).

Environment Variable
In Linux systems, each process has an execution environment. This environment can be inherited from the user session environment and can be extended with specific process variables. An Environment Variable is a value stored in the process environment that can affect its execution.

Framework
A set of standardized concepts, practices or criterias used to face a given problem. Specifically, it defines a set of programs, libraries, languages, and programming models used jointly in a project.

Graphical User Interface
The Graphical User Interface (GUI) is the software that graphically interacts with the user of a computer to ease the data manipulation.

High Performance Computing
Refers to the practice of aggregating the power of several nodes achieving much higher performance in order to solve large problems in science, engineering, or business. The aggregated power is in terms of computation, memory, storage, etc.

Library
Collection of resources with a well-defined interface that can be used simultaneously by multiple independent computer programs and still exhibit the same behaviour. Usually, libraries contain a set of calls that higher level programs can use without implementing them repeatedly.

Master/Worker
Communication model where one process (master) controls and orchestrates one or more other processes (workers or slaves).

Node
A compute node refers to a single system within a cluster of many systems.

Operating System
System software that manages the hardware and provides services for computer software.

Parallel Computing
Computation where a set of processes inside the same machine carry out calculations simultaneously.

Programming Language
Set of instructions to execute specific tasks in a computing device. Usually, it refers to high-level languages (such as C, C++, Java, Python, and FORTRAN) that contain two main components: syntax (form) and semantics (meaning).

Programming Model
Programming style composed by a set of API calls where the execution model differs from the one of the base programming language in which the code is written. Often, the programming model exposes features of the hardware and is invoked as library calls.

Scratch Space
Supercomputers generally have what is called scratch space: disk space available for temporary use and only accessible from the compute nodes.

Script
A program stored in a plain file to automate the execution of tasks. Scripts are usually simple; combining elementary tasks and API calls to define more complex programs.

Software Stack
Set of software needed to create a complete solution (known as a platform) so that any additional software is required to execute the user applications.

SSH
A protocol to securely connect to a remote computer. This connection is generally for a command-line interface, but it is possible to use GUI programs through SSH.

Workflow
A composition of tasks and dependencies between tasks. Workflows are commonly represented as graphs, with the nodes being tasks and the arrows representing the dependencies. Somehow, tasks must represent an action that must be done (i.e. the execution of a binary), and the dependencies must represent the requirements that must be satisfied to be able to execute the task (i.e. the machine availability or the required data).

WSDL
Web Services Description Language (WSDL) is an Extensible Markup Language (XML) used to describe web services.

Part I

Introduction

Chapter 1

Introduction

1.1 Context

1.1.1 The Distributed Computing era

Several years ago, the IT community shifted from sequential programming to parallel programming [30] to fully exploit the computing resources of multi-core processors. The current generation of software applications requires more resources than those offered by a single computing node [88]. Thus, the community has been forced to evolve again towards distributed computing, that is, to obtain a larger amount of computing power by using several interconnected machines [129].

However, one of the major issues that arise from both parallel and distributed programming is that writing in parallel is not as easy as writing sequential programs [137] and, more often than expected, people that can develop useful and complete end-user applications are not capable of writing efficient parallel codes (and the other way around). We can somehow consider that scientists interpreting results do not care about the computational load or distribution (how results are computed), but do care of the results quality, the time spent to retrieve the results and the robustness of the system. In this line of "writing programs that scale with increasing number of cores should be as easy as writing programs for sequential computers" [29], several libraries, tools, frameworks, programming models, and programming languages have emerged to help the programmer to handle the underlying infrastructure.

1.1.2 The joint venture towards the Exascale Computing

Currently, the IT community requires parallelisation and distributed systems to handle large amounts of data [124, 205]. Towards this, *Big-Data Analytics (BDA)* [204] appeared some years ago; allowing the community to store, check, retrieve and transform enormous amounts of data in acceptable response times. In addition, several programming models have also arisen that are utterly different to the ones used by the *High Performance Computing (HPC)* community.

In the race towards *Exascale Computing* [74], the IT community has realised that unifying *HPC* platforms and *Big-Data (BD) Ecosystems* is a must. Currently, these two ecosystems differ significantly at hardware and software level, but "programming models and tools are perhaps the biggest points of divergence between the scientific-computing and *Big-Data Ecosystems*" [200]. In this respect, "there is a clear desire on the part of a number of domain and computer scientists to see a convergence between the two high-level types of computing systems, software, and applications: *Big Data Analytics* and *High Performance Computing*" [65].

A large number of libraries, tools, frameworks, programming models, and programming languages have appeared to fulfil the BDA and HPC needs. In fact, the contributions have gone one step further by building complex software stacks to provide a high-level abstraction for the development of distributed applications. From the point of view of the developers of distributed applications, this fact has lead to the possibility of choosing the most suitable software stack for the final application; promoting an (in)sane competition between them that has finally exploited in an uncontrollable and uncountable number of possibilities.

1.1.3 Task-based Workflows and Dataflows

In a general fashion, the distributed software can be classified into two different paradigms: *Task-based Workflows* and *Dataflows*. On the one hand, *Task-based Workflows* [67] orchestrate the execution of several pieces of code (*tasks*) that process and generate data values. These tasks have no state and, during its execution, they are isolated from other tasks. Hence, Task-based Workflows consist of defining the data dependencies among tasks. The users can explicitly or implicitly define these dependencies by means of a Graphical User Interface (GUI), a Command Line Interface, a receipt file, a programming model or even a programming language. However, all the software that belongs to this paradigm shares that the Data Dependency Graph is based on the task definition and does not vary depending on the input data. Notice that this does not mean that only static graphs can be managed; many software supports conditional flows, loop structures, and even dynamic graphs, but the Data Dependency Graph construction is based on the code that defines the tasks rather than in the input data. In one sentence, *Task-based Workflows* define a workflow of tasks to be executed in a certain order.

On the other hand, *Dataflows* [142] assume that tasks are persistent executions with state that continuously receive/produce data values (streams). Through dataflows, developers describe how the composing tasks communicate to each other. This means that tasks are treated as stages or transformations that data must go through to achieve the destination. Notice that, in contrast to the previous paradigm, the Data Dependency Graph varies depending on the input data, even if the code that defines the tasks is not modified. For instance, Task-based Workflows always execute one task per input data, while Dataflows execute a single task instance for several input data. Finally, in one sentence, *Dataflows* define a flow of data that is transformed from source to destination.

1.1.4 Batch Processing and Continous Processing

Some new concepts have become popular among the recent literature to emphasise the data continuity difference between the distributed software. Kindly stolen from the materials processing industry, many authors refer to *Batch Processing* and *Continous Processing* [208].

On the one hand, *Batch Processing* (Batching) refers to software that acts similarly to a traditional factory. First, a bulk of data is packed (batch) and then, it advances through each step (task) of the workflow until it is fully processed. Notice that all batches are treated separately and that a task can only process one batch at a time.

On the other hand, to keep the industry metaphor, *Continous Processing* refers to software that acts similarly to novel factories which involves moving one data at a time through each step (task) of the workflow without breaks in time. Notice that this implies adapting the number of workers (tasks) on-demand to fulfil the requirements. We must highlight that Continous Processing is a general term that is also known as *Real Time Processing* when the system reacts to the input data or as *Streaming* when the system acts (transforms) on the input data.

Finally, although it is easy to imagine Dataflows performing either Batch Processing or Continous Processing, it is hard it is hard to imagine Task-based Workflows performing Continous Processing since they do not depend on the input data and its definition is strongly related to the Batch Processing behaviour. For this purpose, many Task-based Workflow Managers have lowered down they batch size requirements to be as near as possible to Continous Processing, raising what is known as *Micro-Batching*.

1.2 Objectives and contributions

In the aforementioned context, an extense variety of software has appeared to fulfil different final end-user application requirements. However, application developers find hard to master many of this software, and little effort has been made to homogenise or integrate them. Hence, many developers spend valuable time translating applications from one model to another because of compatibility issues. Moreover, we consider that there is a need of such a framework because scientific simulations and complex applications often require combining different models because of the nature of its calculations; either because its most efficient implementation requires specific software, or either because of legacy compatibilities.

In this sense, this thesis benefits from the joint venture between HPC and BDA to investigate high-level abstraction frameworks capable of executing hybrid Data Science and HPC workflows. Figure 1.1 describes the four approaches used in this thesis to address the problem: (1) orchestrate the execution of multiple binaries, tools, and frameworks inside the same workflow, (2) ease the application packaging and deployment, (3) ease the development and execution of distributed applications, and (4) execute hybrid Task-based Workflows and Dataflows.

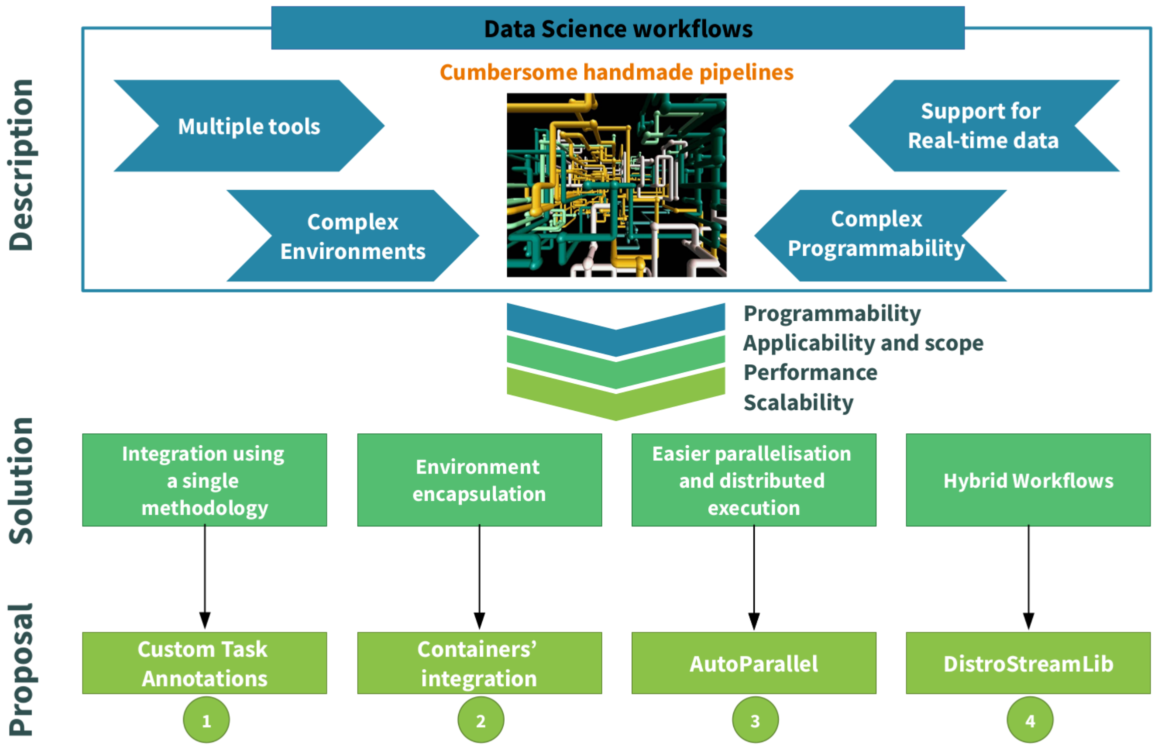


FIGURE 1.1: Lines of work to support Data Science workflows.

First, in contrast to HPC Workflows, Data Science Workflows are more likely to require hybrid executions, and their workload varies faster at execution time. We consider that a programming model capable of acting both as a Workflow Executor and Orchestrator eases the development of distributed applications significantly. Hence, using this model, the users must be capable of designing complex workflows that include native tasks, binaries, MPI [159] simulations, multi-threading applications (i.e. OpenMP [48] or OmpSs [75, 220]), nested COMPSs [32] workflows, and many others.

Second, containers provide an easy way to pack and distribute applications. Hence, by combining container technologies, our prototype enables developers to easily port, distribute, and scale their applications using distributed infrastructures. Moreover, due to the containers' light deployment times, our prototype will be able to adapt faster the resources to the application's remaining workload; reducing the execution cost significantly and allowing other jobs to consume the freed resources.

Third, although programming models have done a huge effort to overcome the distributed computing issues, we believe that there is still room for improvement. Considering that many parallel frameworks are capable of automatically parallelising some sequential code structures, it is easy to imagine that distributed frameworks should act similarly.

Fourth, we believe that Task-based Workflows and Dataflows can cohabit in a single model, allowing the users to change from one to another inside the same workflow depending on their needs. Furthermore, as far as we are concerned, no one has been working on such interaction. Notice that this extension expands from the traditional HPC requirements to the BDA needs; bringing our prototype closer to the Continuous Processing requirements.

Furthermore, our prototype is based on extending the COMPSs Programming Model. Considering that this thesis focuses on non-expert users, COMPSs provides a unique programmability and adaptability that no other state of the art framework can offer. On the one hand, it is based on sequential programming so the users do not need to deal directly with any parallelisation and distribution issue. Instead, COMPSs programmers select methods to be considered as tasks, and the COMPSs Runtime orchestrates its execution asynchronously. On the other hand, the COMPSs model abstracts the application from the underlying infrastructure so that they do not include any platform related detail. Hence, COMPSs applications are portable between any infrastructure and can be executed either locally or in clusters, clouds or containers. For a more extense overview of COMPSs, please refer to Chapter 3.

Finally, next subsections detail the specific objectives, the main contributions, and the associated publications of this thesis.

1.2.1 Research Questions

The following points summarise the research questions that this thesis tries to solve:

- **Q1:** How to orchestrate the execution of multiple binaries, tools, and frameworks inside the same Data Science workflow?
- **Q2:** How to use container technologies to ease the installation of software dependencies and better adapt the computing resources to the workload of the application?
- **Q3:** Can we automatically parallelise and distributedly execute sequential code without any user interaction?
- **Q4:** Can Task-based Workflows and Dataflows cohabit in a single programming model?

1.2.2 Detailed objectives

Related to the aforementioned research questions, the following points summarise the main goals of this project:

- **O1:** Create a high-level abstraction framework capable orchestrating complex workflows that include the execution of binaries, MPI simulations, multi-threading applications (i.e. OpenMP or OmpSs), nested COMPSs workflows, Big Data Frameworks, Machine Learning tools, and other frameworks.
- **O2:** Create a high-level abstraction framework capable orchestrating containerised applications and better adapt the computational resources to the pending workload.
- **O3:** Create a high-level abstraction framework to parallelise some sequential code structures automatically and execute them distributedly.
- **O4:** Create a high-level abstraction framework capable of executing hybrid Task-based Workflows and Dataflows.
- **O5:** Validate the proposal by porting real-world Life science and Earth science applications.

On the other hand, we want the end users to focus on the applications' development without dealing explicitly with the distributed challenges. In this sense, the proposal must support a sequential fashion (either in Java, Python, C or C++) to develop applications and trust in an underlying runtime to exploit the application's inherent parallelism and manage the available computing resources.

1.2.3 Contributions to the field

The main contributions of this thesis are:

- **C1:** A methodology and an implementation to transparently execute complex workflows by orchestrating different binaries, tools, and frameworks.
- **C2:** A methodology and an implementation to integrate container technologies as Resource Orchestration Providers (ROP).
- **C3:** A methodology and an implementation to distributedly execute automatically parallelised sequential code.
- **C4:** A methodology and an implementation to distributedly execute a combination of Task-based Workflows and Dataflows.

Contribution	Objectives	Research Questions
C1	O1, O5	Q1
C2	O2, O5	Q2
C3	O3, O5	Q3
C4	O4, O5	Q4

TABLE 1.1: Relation between research questions, objectives, and contributions.

Table 1.1 summarizes the relation between the research questions and detailed objectives listed in the previous sections and the contributions. Notice that the validation of our proposal by porting real-world Life science and Earth science applications (**O5**) is achieved transversely across the contributions of this thesis.

1.2.4 Publications

Next, we provide the list of publications that support each contribution of this thesis:

- State of the art
  - *A Survey on the Distributed Computing stack.*
    Cristian Ramon-Cortes, Pol Alvarez, Francesc Lordan, Javier Alvarez, Jorge Ejarque, Rosa M. Badia.
    Computer Science Review (COSREV).
    Submitted May 2020.
    Impact Factors - SJR: 1.431, CiteScore: 10.05.
- C1: Orchestration of complex workflows
  - *Boosting Atmospheric Dust Forecast with PyCOMPSs.*
    Javier Conejero, **Cristian Ramon-Cortes**, Kim Serradell, Rosa M. Badia.
    IEEE eScience.
    September 2018.
    Core Rank: A.
- C2: Dynamic computational resources using container techonologies
  - *Transparent Orchestration of Task-based Parallel Applications in Containers Platforms.*
    Cristian Ramon-Cortes, Albert Serven, Jorge Ejarque, Daniele Lezzi, Rosa M. Badia.
    Journal of Grid Computing (JoGC).
    December 2017.
    Impact Factor (JCR): 3.288 (Q1).
  - *Transparent Execution of Task-Based Parallel Applications in Docker with COMP Superscalar.*
    Victor Anton, **Cristian Ramon-Cortes**, Jorge Ejarque, Rosa M. Badia.
    25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP).
    March 2017.
    Core Rank: C.
- C3: Automatic parallelisation
  - *AutoParallel: Automatic parallelisation and distributed execution of affine loop nests in Python.*
    Cristian Ramon-Cortes, Ramon Amela, Jorge Ejarque, Philippe Clauss, Rosa M Badia.
    The International Journal of High Performance Computing Applications (IJHPCA).
    Accepted, May 2020.
    Impact Factor (JCR): 1.956 (Q2).

- *AutoParallel: A Python module for automatic parallelization and distributed execution of affine loop nests.*
  Cristian Ramon-Cortes, Ramon Amela, Jorge Ejarque, Philippe Clauss, Rosa M Badia.
  Proceedings of the 8th Workshop on Python for High-Performance and Scientific Computing (PyHPC - SC).
  November 2018.

- C4: Transparent execution of Hybrid Workflows
  - *A Programming Model for Hybrid Workflows: combining Task-based Workflows and Dataflows all-in-one.*
    Cristian Ramon-Cortes, Francesc Lordan, Jorge Ejarque, Rosa M Badia.
    Future Generation Computer Systems (FGCS), The International Journal of e-Science.
    Accepted July 2020.
    Impact Factor (JCR): 5.768 (Q1).

1.3 Tools and methodology

1.3.1 Tools

The principal tool used during this thesis has been COMPSs [32]. Since the COMPSs Runtime was developed in Java [121], the biggest part of the development has been performed in the Java language using the Eclipse [77] IDE and the Apache Maven [22] Software Project Management. The implementation has also required of C [126], C++ [212], and Python [196] implementations to extend the new features to the COMPSs bindings. Moreover, some BASH [95] scripts have been required for the COMPSs user commands, the deployment tools, and the experimentation testbed. Finally, for the results and the evaluation, we have used Extrae [83] and Paraver [183] to validate the parallel executions and GnuPlot [99] to illustrate simulation results.

1.3.2 Methodology

The methodology of this thesis has been based on the Design Research Method combined with a Test Driven Development strategy, always bearing in mind that the main goal of this thesis was to provide a generic approach to support Data Science workflows with high-level abstraction programming models. Next subsections provide in-depth information about the scientific method design (Subsection 1.3.2.1), the development strategy (Subsection 1.3.2.2) and the validation strategy (Subsection 1.3.2.3).

1.3.2.1 Scientific method design

In the first step, during the Relevance Cycle, we have analysed in-depth the Data Science workflows and the state of the art programming languages, programming models, frameworks, tools, and libraries (see Chapter 2 for more details). During this analysis, we have determined the main contributions of this thesis (see Section 1.2).

The second step, the Design Cycle, has fulfiled the needs of the Data Science workflows by proposing a generalizable extension of the COMPSs Programming Model to orchestrate the execution of complex workflows and to support the execution of hybrid Task-based Workflows and Dataflows. Notice that this step has required several iterations since every proposal must be implemented, documented, evaluated, and validated. Our plan was based on the Test Driven Strategy, where we validate our proposals against incremental use cases, from the simplest use cases up to the real world Data Science workflows.

Finally, during the Rigor Cycle, we have deployed the project in a production environment and largely documented the new artifacts to share the knowledge with the community.

1.3.2.2 Development strategy

The development has been based on the Test Driven Development Strategy, building applications from the simplest use cases up to the real world Data Science workflows. Form our experience, this is a robust and flexible methodology that keeps the project's goal in mind during the whole development of the project.

More in-depth, we have first defined all the Data Science workflows' requirements and created an easy-to-run test for each of them. Next, we have incrementally implemented the required features to fulfil every requirement. At this point, since the application requirements are more strong at the programming model's API level, we have followed a top-bottom implementation. Finally, once all the requirements are separately fulfiled, we have evaluated the proposal against more complex use cases and real-world Data Science workflows.

We would like to highlight that we have found the Test Driven Development significantly useful in making this process iterative, since all the use cases have been easily tested and have incrementally guided the development towards the end goal.

1.3.2.3 Validation strategy

Even if all the features have been tested locally, an extense validation has been performed against real-world Data Science workflows. This validation process has included porting the proposal and all the use cases to the MareNostrum 4 supercomputer [149] and performing an in-depth analysis of the application.

1.4 Dissertation structure

The rest of the document is structured as follows. First, regarding the rest of chapters of Part I, Chapter 2 describes the work that has been done in the field and, Chapter 3 introduces the starting point of our research.

Next, Part II details the contributions of this thesis; including a separated chapter for each contribution and a brief summary at the beginning of each chapter. This part contains the following chapters:

- Chapter 4 reports the integration with other frameworks to build complex Data Science workflows where some steps require highly optimised state of the art frameworks. Our solution provides several annotations (i.e., `@binary`, `@OmpSs`, `@MPI`, `@COMPSs`, and `@MultiNode`) for both Java and Python workflows so that non-expert users can orchestrate different frameworks within the same programming model.

- Chapter 5 describes the integration of container technologies as Resource Orchestration Providers (ROP). This combination provides a straightforward methodology to parallelise applications from sequential codes along with efficient image management and application deployment that ease the packaging and distribution of applications. We distinguish between static, HPC, and dynamic container management and provide representative use cases for each scenario using Docker, Singularity, and Mesos.

- Chapter 6 introduces the automatic parallelisation techniques included in our prototype. This extension provides a single Python annotation (the `@parallel` Python decorator) to automatically parallelise affine loop nests and execute them in distributed infrastructures so that anyone with intermediate-level programming skills can scale up an application to hundreds of cores.

- Chapter 7 presents the main characteristics of our solution to execute Hybrid Workflows: a combination of Task-based Workflows and Dataflows. This extension enables developers to build complex Data Science workflows with different approaches depending on the requirements without the effort of combining several frameworks at the same time. Moreover, this chapter describes the Distributed Stream Library that provides a homogeneous, generic and simple representation of object and file streams for both Java and Python.

Finally, in Part III, Chapter 8 concludes and gives some guidelines for future work.

Chapter 2

State of the art

This chapter provides an overview of the general state of the art and context of this thesis. The specific related work of each contribution is discussed in-depth at the begining of each chapter (see Sections 4.2, 5.2, 6.2, and 7.2).

2.1 Distributed Computing

The distributed computing premise is simple: solving a large problem with an enormous amount of computations as fast as possible by dividing it into smaller problems, dealing with them parallelly and distributedly, and gathering the results back. However, its implementation is not that simple because it can either lead to significant speed-ups or overheads due to the distributed computing challenges. These challenges range from the *Resource Management* to the *Data Distribution*, going through the *Coordination* and the *Monitoring* of the different distributed components.

In this sense, the community increasingly prefers to rely on high-abstraction frameworks to focus only on the application development by using any programming language, programming model or framework that fully abstracts the user from the distributed computing challenges; either by relying on other state of the art software, or by handling them explicitly. Moreover, these frameworks are expected to be easy to install, configure, and use so that they can be rapidly adapted to any application.

Representing the highest layer of the software stack and providing an almost ready-to-use option to implement distributed applications are the crucial points of the success of the *Application Development* software. However, at the technical level, they are also the worst black spots since high abstraction can only be achieved by building huge software stacks or extensive frameworks that are, in both cases, hard to maintain. Furthermore, ready-to-use

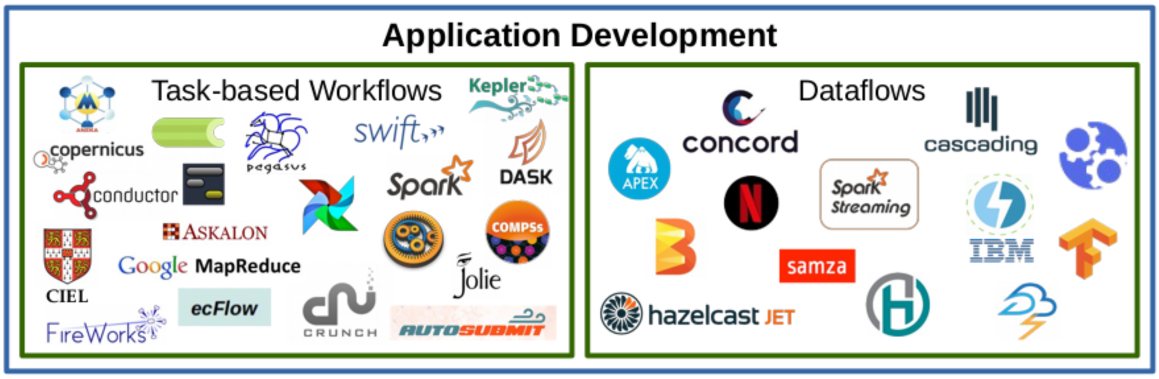


FIGURE 2.1: Classification of high abstraction frameworks for *Application Development*.

tools require automatic configurations that must support several heterogeneous underlying platforms that are continuously upgraded.

As shown in Figure 2.1, we have divided the high abstraction frameworks into two categories depending on the software purpose. Next subsections provide further information about each category and its latest software.

2.2 Task-based Workflows

The software targeting *Task-based Workflows* allows the users to define pieces of code to be remotely executed as *tasks* and dependencies between tasks to combine them together into *workflows*. The main common feature in this family of software is that the principal working unit is the *task*.

2.2.1 Software discussion and examples

Regarding the execution model, some frameworks, like Aneka [227] or Jolie [157], require application users to create tasks and add them to a bag explicitly. The tasks inside the bag are then selected to be executed by the model with equal probability. In this sense, the main drawback of using a *Bag of Tasks* is that users need to handle data dependencies between tasks before introducing a new task to the bag.

Other frameworks restrict the workflow to a predefined parallelism pattern (*Skeleton* programming), such as MapReduce [63]. In this kind of models, programmers only need to specify a set of methods that compose the predefined workflow. In contrast to the previous approach, skeleton models do handle data dependencies between tasks, but the users' application is pigeonholed into the predefined parallelism pattern.

Finally, other models go one step further by generalising the *Skeleton* model and allowing users to define Directed Acyclic Graph (DAG) of tasks. In this approach, applications are represented as DAGs, where tasks are represented by vertices, and data dependencies are represented by edges. In contrast with *Skeleton* models, DAG models allow application developers to describe any kind of workflow with any custom operation. Although all the frameworks within this group hide the data dependencies and the communication between the distributed processes, they can be classified by its workflow definition. On the one hand, some models require to explicitly define the workflow by means of a Graphical User Interface (such as Taverna [113], Kepler [219, 146] or Galaxy [5]), a Command Line Interface (such as Copernicus [190]), a receipt file (such as Askalon [84], AUTOSUBMIT [31], Fireworks [10] or Netflix Conductor [168]) or a language API (such as Pegasus [66], Apache Airflow [11] or ecFlow [76]). This methodology allows users to specifically control the dependencies between the different stages and have a clear overview of how the framework executes their application but makes tedious to design complex, large workflows. On the other hand, there are programming models and languages that opt to automatically infer the workflow from the user code, e.g., Spark [237], COMPSs [32, 144], Dask [202, 61], Apache Crunch [15], Celery [45], and Swift [232]. This workflow definition allows users to develop applications in an almost sequential manner, without explicitly handling the tasks spawned, and reducing the programming complexity to almost zero. However, the main disadvantage is that the users do not know beforehand how the framework will execute their application (for example, how many tasks will be created in a specific call).

2.2.2 Taxonomy

Table 2.1 presents our taxonomy of the surveyed task-based frameworks. Regarding the programming, we have categorised the different interfaces into Graphical User Interface

Software		Features																											
		Prog.		Workflow Definition						Workflow Exec.					Res. Management						F. Tolerance					Security			
		Interface	Language	Execution Model	Task Dependency Definition	Task Type	Dynamic	Nested	Stream Support	Parallel Execution	Load Balancing	Execution Analysis	Native Scheduling Policy	Configurable Sched. Policies	Num. Surrogates	Heterogeneous Surrogates	Cluster	Cloud	Container	Elasticity	Checkpointing	Lineage	Replication	Resubmission	Failover	Secure Communication	Data Encryption	User authentication	License
Task-based Workflows	Airflow [11]	GA	P	D	E	A	⋆	-	-	⋆	-	O	R	-	U		⋆	⋆	-	A	-	-	-	⋆	-	⋆	-	⋆	1
	Aneka [227]	G	V	B	/	A	-	-	-	⋆	⋆	O		-	U	⋆	⋆	⋆	-	A	-	-	-	⋆	⋆	⋆	⋆	-	8
	Askalon [84]	GR	/	D	I	A	-		-	⋆	⋆		O	⋆	U	⋆	⋆	-	-	A	A	-	⋆	⋆	-	-	-	-	8
	AUTOSUBMIT [31]	R	B	D	E	U	-	-	-	⋆	-	OP			U	⋆	⋆	-	-	-									3
	Celery [45]	PA	P	D	I	A	⋆	⋆	-	⋆		O			U	⋆	⋆	-	-	A	-	-	-	-	-	⋆	-	-	5
	CIEL [163]	LA	JO	D	I	A	⋆	⋆	⋆	⋆	⋆	-	O	-	U	⋆	⋆	-	-	-	-	⋆	-	⋆	⋆	-	-	-	5
	COMPSs [54]	PA	JPC	D	I	A	⋆	-	-	⋆	⋆	OP	O	⋆	U	⋆	⋆	⋆	⋆	A	-	-	-	⋆	-	-	-	-	1
	Copernicus [190]	R	PX	D	E	P	-	-	-	⋆		O			U	⋆	⋆	-	-	-	AM	-	-	⋆	⋆	⋆	-	⋆	2
	Crunch [15]	CA	J	D	I	A	⋆	-	⋆	⋆																			1
	Dask [202, 61]	GA	P	D	I	A	⋆	⋆	⋆	⋆	⋆	OP	L	⋆	U		⋆	-	-	-	-	-	-	⋆	M	⋆	⋆	⋆	5
	EcFlow [76]	CAG	PB	D	I	A	-	-	-	⋆	-				U	⋆	⋆	⋆	⋆	-	-	-	-	⋆	⋆	-	-	⋆	1
	FireWorks [10]	R	PNY	D	E	A	⋆	⋆	-	⋆	-	OP	F	-	U	⋆	⋆	-	-	-	-	-	-	⋆	-	-	-	-	5
	Galaxy [5]	G	/	D	E	S	-	-	-	⋆	-		F		U	/	/	/	/	/	-	-	-					⋆	7
	Google MapReduce [63]	A	C	S	I	U	-	-	-	⋆	⋆	-	O	-	U	⋆	⋆	-	-	-	-	-	⋆	⋆	⋆	-	-	-	8
	Jolie [157]	L	/	B	/	S	-	-	-	⋆	-	-	/	/	/	⋆	⋆	⋆	⋆	-	-	-	-	-	-	-	-	-	4
	Kepler [146]	G	/	D	E	A	-	⋆	-	-	-	-	/	/	/	/	/	/	/	/	-	-	-	-	-	-	-	-	5
	Netflix Conductor [168]	R	N	D	I	S	⋆	⋆	-	⋆		OP	R	-	U	⋆	/	/	/	/									1
	Pegasus [66]	RA	JPL	D	E	A	⋆	⋆	-	⋆	⋆	OP	F	⋆	U	⋆	⋆	⋆	⋆		A	⋆	-	⋆	-	-	-	-	1
	Spark [237]	CA	JSPR	D	I	A	⋆	⋆	⋆	⋆	⋆	O	F	⋆	U	⋆	⋆	⋆	⋆	A	-	⋆	⋆	⋆	⋆	-	⋆	⋆	1
	Swift [232]	L	/	D	I	A			-	⋆		OP			U	⋆	⋆	-	-	-	A	-	-	⋆	-	-	-	-	1
	Taverna [113]	G	/	D	E	S	-	⋆	⋆	⋆	-	OP	F	/	/	/	/	/	/	/	-	-	-	⋆	⋆	⋆	-	⋆	4

Legend: ⋆ Available - Not available / Not Applicable

TABLE 2.1: Comparison of the different software targeting *Task-based Workflows*.

(G in the table), Command Line Interface (C), receipt file (R), annotations or pragmas (P), programming API (A), or programming language (L). Also, we have distinguished the language supported by each framework: Java (J), Scala (S), Python (P), C++ (C), Visual C (V), R (R), Pearl (L), Bash (B), XML (X), JSON (N), YAML (Y), and OCaml (O).

Regarding the workflow definition, a distinction is made between the different execution model's types: bag of tasks (B in the table), *skeleton* (S), or DAG (D). We have also considered whether the users must explicitly (E) or implicitly (I) define the task dependencies and classified the supported task types into only pre-defined methods (P), only services (S), only user defined methods (U), or any (A). Also, we have distinguished the support for workflows that can vary during the application's execution (dynamic workflows), for nested executions, and for data streams.

When focusing on the application execution, the main difference is the framework's capability of executing in parallel or not. However, we have also distinguished more advanced features such as load balancing techniques, built-in tools for the application's execution analysis - online (O) or post-mortem (P) -, the native scheduling policy - ready (R), FIFO (F), LIFO (L) or optimised (O) -, and the support for customisable scheduling policies.

Regarding the resource management, we have distinguished between frameworks capable of handling a bounded or an unbounded number of surrogates. In this sense, we have also considered the capability of managing heterogeneous infrastructures, clusters, clouds, and containers. We have also evaluated the framework's capability to provide resource elasticity during the application's execution time, categorising this feature into automatic (A) or manual (M).

For advanced users, fault tolerance mechanisms and security might be an issue. For this purpose, we have also distinguished those frameworks that provide any kind of checkpointing - automatic (A) or manual (M) -, lineage (i.e., the ability to re-generate a lost or corrupt data by executing again the chain of operations that was used to generate it), replication, re-submission or fail-over, and those that provide secure communication, data encryption or user authentication.

Finally, we consider that the framework's availability is a high-priority issue for application developers. For this purpose, we also consider the license of each framework following the next nomenclature: (1 in the table) Apache 2.0 [21], (2) GNU GPL2 [96], (3) GNU GPL3 [97], (4) GNU LGPL2 [98], (5) BSD [42], (6) MIT License [156], (7) other public open-source software licenses (e.g., Academic Free License v3 [3], Mozilla Public License 2 [158], Eclipse Public License v1.0 [78]), and (8) custom private license or patent.

2.2.3 Analysis

First, the programming model's interface differs significantly between frameworks being the most common ones the programming API (A in the table) and the receipt file (R). However, many of them provide (or have planned to provide in the near future) an attractive easy-to-use Graphical User Interface (G) that includes advanced online and post-mortem execution analysis tools.

Second, although some frameworks include support for several languages (e.g., CIEL, COMPSs, Copernicus, EcFlow, FireWorks, Pegasus, and Spark), the rest of them only supports a single language. Hence, application developers should consider the application's language before selecting the appropriate framework. Also, it is worth mentioning that CIEL uses a custom language (Skywriting) but also provides APIs for Java and OCaml.

Third, most of the frameworks support user-defined (U) or any type (A) of tasks (except Copernicus which is developed for pre-defined methods and Galaxy, Jolie, Netflix Conductor, and Taverna which are developed for services) and are using the DAG execution model (except Aneka, Google MapReduce, and Jolie) to support complex workflows. Nevertheless, there are significant differences on the task dependency definition approach. From our point of view, the frameworks using explicit task dependency definition are more suitable for small applications while frameworks using implicit task dependency definition are better for large and complex application workflows.

Fourth, we are surprised by the lack of support for advanced workflow features (i.e., dynamic and nested workflows, and support for streams) and new infrastructures (mainly the cloud and containers). Although the software might still be evolving, modern applications require complex workflow features and elasticity mechanisms to automatically handle the application's resource usage (i.e., by managing the available computing resources). In this same line, we also believe that many frameworks have been designed for cluster infrastructures; which explains the lack of security mechanisms (i.e., secure communication, data encryption or user authentication).

In terms of fault tolerance, while re-submission and fail-over are common techniques among all the different software, only a few of them include checkpointing (Askalon, Copernicus, Pegasus, and Swift) or lineage (CIEL, Pegasus, and Spark). We know that fault tolerance comes up with a non-negligible performance degradation but, since application runs are lasting longer and longer, we believe that this is a key feature when selecting the appropriate framework.

Finally, most of the frameworks (except Aneka, Askalon, and Google MapReduce) are available through different public open licenses and are supported by large user communities, which allows developers to try different possibilities easily and without any cost before selecting the right framework for their application requirements.

2.3 Dataflows

Similarly to Task-based Workflows, *Dataflows* allow the application developers to define pieces of code to be remotely executed as *tasks*. However, Dataflows assume that tasks are persistent executions with state that continuously receive/produce data values (streams) and, therefore, tasks are treated as stages or transformations that data must go through to achieve the destination. Also, Dataflows are based on *Data Flow Graphs (DFG)* rather than Task Dependency Graphs (TDG). This difference mainly affects the way the task graph is constructed. On the one hand, TDGs define a task completion relation between tasks so that the only information travelling among the graph nodes is the task completion status and, thus, tasks need to share the data in a graph-independent way. On the other hand, DFGs define the data path so that the nodes of the graph represent stateful tasks waiting for the data travelling through the graph edges.

Closely following this definition, there are platforms that are specifically built for Dataflows such as TensorFlow [1]. However, this approach is also used for *stream processing*, *real-time processing* and *reactive programming* which, for the case, are basically subsets of each other. Thus, in stream processing words, a Dataflow is a sequence of data values (*stream*) where we apply a series of operations (*kernel* functions) to each element of the stream in a pipelined fashion.

2.3.1 Software discussion and examples

Although Task-based Workflows can target any type of computation, stream processing has become increasingly prevalent for processing social media and sensor devices data. On the one hand, many solutions such as Apache Samza [24], Apache Storm [225], Twitter Heron [135], IBM Streams [115, 110], Netflix Mantis [167], Cascading [43], or Apache Beam [13] have arose explicitly to solve this problem. On the other hand, other models have included stream processing while maintaining the functionalities of the rest of their framework through the micro-batching technique (e.g., Spark Streaming [235]) or by evolving from the databases environment to the in-memory computation (e.g., Hazelcast jet [104]).

2.3.2 Taxonomy

Software		Features																											
		Prog.		Stream Model							Workflow Exec.					Res. Management						F. Tolerance				Security			
		Interface	Language	Primitive	Multi-subscriber	Ordered	Window	Buffering	Drop Messages	Back Pressure	Process Model	Delivery Pattern	Stateful Operations	Load Balancing	Execution Analysis	Num. Surrogates	Heterogeneous Surrogates	Cluster	Cloud	Container	Elasticity	Checkpointing	Replication	Resubmission	Failover	Secure Communication	Data Encryption	User authentication	License
Dataflows	Apex [12]	CA	J	T	★	★	S	★	-	-	M	LME	★	★	OP	U	★	★	-	-	-	A	-	★	★	★	-	★	1
	Beam [13]	CA	JPG	B	★	-	TS	★	★	-	MO	LME	★	-		U	★	★	-	-	-	-	-	★	★				1
	Cascading [43]	CA	J	T	-	-					O					U	★	★	-	-	-	M	-	-	-				1
	Gearpump [17]	GCA	J	K	★	★	T	-	-	-	O	LE			OP	U	★	★	-	-	-	A	-	★	★				1
	Hazelcast jet [104]	CA	J	T	★	★	TS	★	★	★	O	L		★	-	U	★	★	★	-	★	-	-	★	★	-	-	-	1
	Heron [135]	GCA	JSP	T	★	★	TS	★		★	O	LME	★	-	OP	U	★	★	-	-	-	-	-	★	★				1
	IBM Streams [110]	GA		T	★						O	LE	-		OP	U	★	-	★	-	-	A	-		★				8
	Netflix Mantis [167]	GCA	J	S	★			★	★	★	MO		★		OP	U	★	★	★	-	★	-	-	★	★				8
	Samza [24]	CA	J	M	★	★	TS	★	★	★	O	LE	★	-	-	U	★	★	-	-	-	A	-	★	★	-	-	-	1
	Spark Streaming [235]	CA	JSP	D	★	-	TS	★		-	M	LME	★	★	OP	U	★	★	-	-	-	A	-	★	★	★	-	★	1
	Storm [225]	CA	JSLR	T	★	-	TS	★	★	-	O	LM	★	M	OP	U	★	★	-	★	★	A	-	★	★	★	-	-	1
	TensorFlow [1]	GCA	JPC	R	★						M	E	★	★	OP	U	★	★	-	-	-	M	-	★	★	-	-		1

Legend: ★ Available - Not available / Not Applicable

TABLE 2.2: Comparison of the different software targeting *Dataflows*.

Table 2.2 presents our taxonomy of the surveyed Dataflow frameworks. Regarding the programming, we have categorised the different interfaces into Graphical User Interface (G in the table), Command Line Interface (C), receipt file (R), programming API (A), or programming language (L). Also, we have distinguished the language supported by each framework: Java (J), Scala (S), Python (P), C++ (C), Go (G), Clojure (L), and JRuby (R).

For the application developers, we consider the main distinction between frameworks relies on the stream model. To this purpose, we distinguish the stream primitive between message (M in the table), tuple (T), bolt (B), DStream (D), source (S), task (K), and tensor (R). We also distinguish between single-subscriber and multi-subscriber models, and between ordered and unordered streams. We have also categorised the windowing support between time window (T) and size window (S) for those frameworks that allow to process a group of stream entries that fall within a window based on timers or data sizes. Moreover, we have evaluated other features such as buffering, message dropping, and back pressure.

Regarding the workflow execution, we distinguish the process model between one-record-at-a-time (O) or micro-batch (M). Furthermore, we have categorised the delivery pattern into at-least-once (L), at-most-once (M), and exactly-once (E) for those frameworks that ensure that messages are never lost, never replicated or both. We have also distinguished more advanced features such as support for stateful operations, load balancing techniques, and built-in tools for the application's execution analysis - online (O) or post-mortem (P) -.

We have considered the same features than in the previous case (see Section 2.2.2 for further details) when focusing on resource management, fault tolerance mechanisms, security, and licensing.

2.3.3 Analysis

First, all alternatives use a programming API (A in the table) combined with a supporting easy-to-use Graphical User Interface (G) or Command Line Interface (C) that includes advanced online and post-mortem execution analysis tools. Generally, we must highlight that the interfaces offered are more modern, attractive, and accessible than the ones offered by frameworks targeting Task-based Workflows, probably because the Dataflow software is newer.

Second, a narrow minority of frameworks offer support for several languages (e.g., Beam, Heron, Spark Streaming, Storm, and TensorFlow). As we stated for software targeting Task-based Workflows, we consider that the application developers should consider the application's language before selecting the appropriate framework.

Third, we observe that almost every framework has its own primitive, being the tuple (T in the table) the most commonly used. Although this may not be a problem when developing applications, it hardens the portability of applications between frameworks. Regarding the stream model, all the frameworks are multi-subscriber (except Cascading), allow the users to configure time and size windows (except Apex and Gearpump), and include buffering techniques (except Gearpump). However, only a few of them support advanced techniques such as ordered streams (Apex, Gearpump, Hazelcast jet, Heron, and Samza), message dropping (Beam, Hazelcast jet, Netflix Mantis, Samza, and Storm) or back pressure (Hazelcast jet, Heron, Netflix Mantis, and Samza).

Fourth, a large majority of the software has been explicitly developed for stream processing, what is reflected in a one-record-at-a-time (O in the table) process model. Only Apex, Spark Streaming, and TensorFlow rely exclusively on the micro-batching (M) technique. Also regarding the workflow execution, most frameworks include stateful operations (except IBM Streams) and load balancing techniques. Although all frameworks support the at-least-once delivery pattern (except TensorFlow), there is a significant variety when supporting the at-most-once, and exactly-once delivery patterns. We believe that this is a key feature to classify frameworks that application developers should consider to select the appropriate one.

Fifth, in terms of resource management, we are surprised by the lack of support for new infrastructures since all the software supports heterogeneous clusters, but almost none includes elasticity mechanisms for the cloud and containers. Similarly, the frameworks also lack security mechanisms (such as secure communication, data encryption or user authentication) because they are designed for clusters. Although software targeting Task-based Workflows presented the same issue, this should not be acceptable for novel frameworks as the ones covered in this section.

On the other hand, regarding fault tolerance, we are gratefully surprised to notice that the Dataflow frameworks are largely better. Generally speaking, the Dataflow software is not only including re-submission and fail-over (as the task-based frameworks do), but also checkpointing. As we previously stated, we consider that fault tolerance is a key feature for long-lasting applications.

Finally, as with task-based frameworks, we observe that most of the frameworks (except IBM Streams, and Netflix Mantis) are available through different public open licenses and are supported by large user communities.

Chapter 3

Background

This chapter provides an overview of the current state of the primary tools and frameworks on which this thesis relies. First, we introduce COMP Superscalar since, as we have already mentioned, this project extends its model. Next, we give a quick look at MPI because it is widely used in Data Science workflows and it highlights the benefits of the COMPSs programming model.

Some parts of this thesis use specific software that is not relevant for the rest of the thesis. Hence, the specific background is described at the begining of each contribution. For instance, Section 5.3 describes Resource Orchestration Platforms (ROP), Section 6.3 describes PLUTO, and Section 7.3 introduces Kafka.

3.1 COMPSs

COMP Superscalar (COMPSs) [32] is a task-based programming model that belongs to the family of Frameworks with implicit workflows. COMPSs applications consist of three parts: the application's code developed in a totally sequential manner, an application interface where the programmers specify which functions can be remotely executed (*tasks*) and

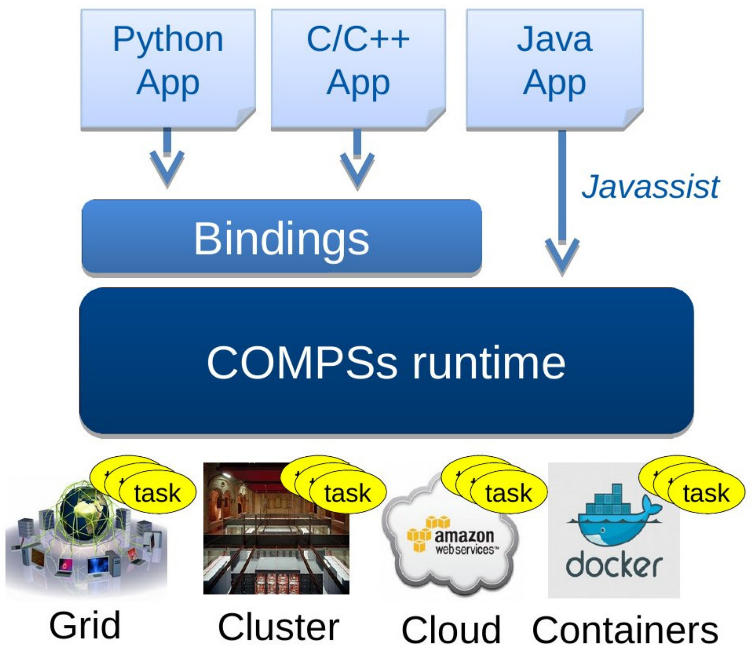


FIGURE 3.1: COMPSs overview.

a configuration file that describes the underlying infrastructure. With these three components, the COMPSs Runtime system exploits the inherent parallelism of the application at execution time by detecting the task calls and the data dependencies between them.

COMPSs natively supports Java applications but also provides bindings for Python (PyCOMPSs [215]) and C/C++. Furthermore, COMPSs allows applications to be executed on top of different infrastructures (such as multi-core machines, grids, clouds or containers) without modifying a single line of the application's code (see Figure 3.1). It also has fault-tolerant mechanisms for partial failures (with job resubmission and reschedule when task or resources fail), has a live monitoring tool through a built-in web interface, supports instrumentation using the Extrae [83] tool to generate post-mortem traces that can be analysed with Paraver [183], has an Eclipse IDE, and has pluggable cloud connectors and task schedulers.

Additionally, the COMPSs model has three key characteristics:

- **Sequential Programming:** The users do not need to deal with any parallelisation and distribution ascpect such as thread creation, synchronisation, data distribution, messaging or fault-tolerance. COMPSs programmers only select which methods must be considered tasks, and the COMPSs Runtime spawns them asynchronously on a set of resources instead of executing them locally and sequentially.

- **Infrastructure Agnostic:** COMPSs model abstracts the application from the underlying infrastructure. Hence, COMPSs applications do not include any platform related detail such as deployment or resource management. This feature makes applications portable between infrastructures with different characteristics.

- **No APIs:** When using COMPSs native language, Java, the model does not require any special API, pragma or construct in the program. Since COMPSs instruments the application's code at execution time to detect the tasks, everything can be developed in the standard Java syntax and libraries.

3.1.1 Programming model

The COMPSs programming model is based on sequential programming and mainly involves choosing the right methods as tasks. The users only need to annotate class and object methods so that they are asynchronosly spawned in the available resources at execution time. These annotations can be splited into two groups:

- **Method Annotations:** Annotations added to the sequential code methods to detect them as tasks and potentially execute them in parallel.

- **Parameter Annotations:** Annotations added to the parameters of an annotated method to handle data dependencies and transfers.

Since the annotation depends on the programming language, next sections provide in-depth details of the COMPSs programming model for Java and Python.

3.1.1.1 Java

A COMPSs application in Java is composed of three parts:

- **Main application:** Sequential code that defines the workflow of the application. It must contain calls to class or object methods annotated as tasks so that, at execution time, they can be asynchronously executed in the available resources.

- **Remote Methods:** Code containing the implementation of the tasks
- **Annotated Interface:** List of annotated methods to be run as remote tasks. This interface also contains the parameter annotations required by the COMPSs runtime to schedule the tasks properly.

Next, we illustrate how COMPSs applications are developed using the Increment: a didactic application that takes *N* counters, initialises them to a random value and increments them by *U* units. Listing 3.1 provides the main code, and Listing 3.2 provides the remote methods. Notice that these files contain all the code of the application and can be executed without COMPSs since they are written in purely sequential Java.

```java
public class Increment {

    public static void main(String[] args) {
        // Retrieve arguments
        if (args.length != 2) {
            System.err.println("[ERROR] Invalid number of arguments");
            System.err.println("    Usage: increment.Increment <N> <U>");
            System.exit(1);
        }
        int N = Integer.parseInt(args[0]);
        int U = Integer.parseInt(args[1]);
        // Initialize counters
        Integer[] counters = new Integer[N];
        for (int i = 0; i < N; ++i) {
            counters[i] = new java.util.Random().nextInt(
                                Integer.MAX_VALUE);
            System.out.println("[LOG] Initial Counter " + i
                                + " value is " + counters[i]);
        }
        // Increment U units each counter
        for (int i = 0; i < U; ++i) {
            for (int j = 0; j < N; ++j) {
                counters[j] = IncrementImpl.increment(counters[j]);
            }
        }
        // Show final counter values
        for (int i = 0; i < N; ++i) {
            System.out.println("[LOG] Final Counter " + i
                                + " value is " + counters[i]);
        }
    }
}
```

LISTING 3.1: COMPSs Java example: Increment main class.

```java
public class IncrementImpl {

    public static Integer increment(Integer counter) {
        // Return the increased counter
        return counter + 1;
    }

}
```

LISTING 3.2: COMPSs Java example: Increment helper methods class.

To enable COMPSs, the users must define the Annotated Interface and specify which methods must be considered *tasks*. The Annotated Interface must be defined inside a file with the same name than the main class of the users' application but with the "Itf" suffix (for instance, in the previous example, the Interface must be stored in the *IncrementItf.java* file). Regarding its content, the Annotated Interface must contain one entry per task. Each task must contain a Method annotation, the object or class method name, and one Parameter Annotation per method parameter. Section 3.1.1.3 lists more complex Method Annotations.

Listing 3.3 shows the Interface for the previous Increment application example. Notice that it only contains one task declaration of type $@Method$ called *increment*. Inside the $@Method$ annotation the users must also provide the *declaring class* which is the class containing the implementation of the task (*IncrementImpl* in the example). This value is required to link the task to the method implementation. The task definition also contains two Parameter Annotations that are required to build the task dependency graph since COMPSs uses data-flow graphs. The mandatory contents of the *Parameter* annotation are the *Type* (that must refer to any Java basic type, a string, an object or a file) and the *Direction* (where the only valid values are *IN*, *OUT* and *INOUT*). In the example, the first parameter is the return value of the function which has type *Integer* and, by default, has direction *OUT*. The second parameter is the *counter* argument which has type *Integer* and direction IN because the function requires the input value of the parameter to increase it but does not modify it.

```java
import es.bsc.compss.types.annotations.Parameter;
import es.bsc.compss.types.annotations.parameter.Direction;
import es.bsc.compss.types.annotations.parameter.Type;
import es.bsc.compss.types.annotations.task.Method;

public interface IncrementItf {

    @Method(declaringClass = "increment.IncrementImpl")
    Integer increment(
        @Parameter(type = Type.OBJECT, direction = Direction.IN)
            Integer counter
    );
}
```

LISTING 3.3: COMPSs Java example: Increment Interface.

As previously stated, the COMPSs annotations do not interfere with the applications' code. Thus, all COMPSs applications can be sequentially executed. To do so, Listing 3.4 compiles the previous code and executes the application with $N = 2$ counters that must be increased by $U = 3$.

```
$ javac increment/*
$ jar cf increment.jar increment/
$ java -cp increment.jar increment.Increment 2 3
[LOG] Initial Counter 0 value is 7
[LOG] Initial Counter 1 value is 1
[LOG] Final Counter 0 value is 10
[LOG] Final Counter 1 value is 4
```

LISTING 3.4: COMPSs Java example: Sequential execution of Increment.

On the other hand, the code can also be executed with COMPSs without recompiling the application's code. To do so, the users must invoke the *runcompss* command instead of the traditional *java* command. When done, the COMPSs Runtime will be setup and the application will be distributedly executed. Figure 3.2 provides the execution output of the Increment application and the task graph generated by its execution. Notice that the *runcompss* command has several command-line arguments that are fully detailed when executing `runcompss -h`.

```
$ runcompss -g increment.Increment 2 3
[  INFO] Using default execution type: compss
[  INFO] Using default location for project file
[  INFO] Using default location for resources file
[  INFO] Using default language: java

----------------- Executing increment.Increment -------------------------

WARNING: IT Properties file is null. Setting default values
[(790)    API]  -  Starting COMPSs Runtime v2.0
[LOG] Initial Counter 0 value is 5
[LOG] Initial Counter 1 value is 2
[LOG] Final Counter 0 value is 8
[LOG] Final Counter 1 value is 5
[(5090)    API]  -  Execution Finished

------------------------------------------------------------
```

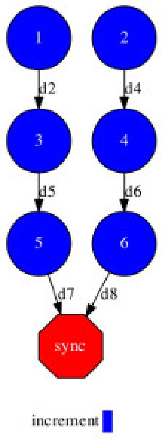


FIGURE 3.2: COMPSs Java example: COMPSs execution of Increment.

3.1.1.2 Python

The Python syntax in COMPSs is supported through a binding, PyCOMPSs. This Python binding is supported by a Binding-commons layer which focuses on enabling the functionalities of the Runtime to other languages (currently, Python and C/C++). It has been designed as an API with a set of defined functions. It is written in C and performs the communication with the Runtime through the JNI [141].

In contrast with the Java programming model, all the PyCOMPSs annotations are done inline. The Method Annotations are in the form of Python decorators. Hence, the users can add the `@task` decorator on top of a class or object method to indicate that its invocations will become tasks at execution time. Furthermore, the Parameter Annotations are contained inside the Method Annotation. For instance, the users can specify if a given parameter is read (IN), written (OUT) or both read and written in the method (INOUT).

Listing 3.5 shows an example of a task annotation. The parameter `c` is of type INOUT, and parameters `a`, and `b` are set to the default type IN. The directionality tags are used at execution time to derive the data dependencies between tasks and are applied at an object level, taking into account its references to identify when two tasks access the same object.

```python
@task(c=INOUT)
def multiply(a, b, c):
    c += a * b
```

LISTING 3.5: COMPSs Python example: Task annotation.

A tiny synchronisation API completes the PyCOMPSs syntax. As shown in Listing 3.6, the API function `compss_wait_on` waits until all the tasks modifying the `result`'s value are finished and brings the value to the node executing the main program. Once the value is retrieved, the execution of the main program code is resumed. Given that PyCOMPSs is used mostly in distributed environments, synchronising may imply a data transfer from remote storage or memory space to the node executing the main program.

```python
for block in data:
    presult = word_count(block)
    reduce_count(result, presult)
final_result = compss_wait_on(result)
```

LISTING 3.6: COMPSs Python example: Call to synchronisation API.

Finally, Table 3.1 summarises the available API functions.

API Function	Use
`compss_wait_on(obj, to_write=True)`	Synchronises for the last version of an object (or list of objects) and returns it.
`compss_open(file_name, mode='r')`	Synchronises for the last version of a file and returns it file descriptor.
`compss_delete_file(file_name)`	Notifies the Runtime to remove the given file.
`compss_delete_object(object)`	Notifies the runtime to delete the associated file to the given object.
`compss_barrier(no_more_tasks=False)`	Waits for the completion of all the previous tasks. If `no_more_tasks` is set to True, it blocks the generation of new tasks.

TABLE 3.1: List of PyCOMPSs API functions.

3.1.1.3 Annotations' summary

Apart from the task annotation, COMPSs provides a larger set of annotations for both Java and Python programming languages that include constraints to optionally define some hardware or software requirements, and scheduler hints to help the Scheduler component to schedule tasks in a certain predefined way. Next, Table 3.2 summarises the available Method and Parameter annotations for Python.

Method Annotation	Parameters			
	Name	**Value Type**	**Mandatory?**	**Default Value**
`@task(...)`	parameter name	`parameter_key`	Yes	
	returns	type | int	No	`0`
	isModifier	Boolean	No	`True`
	priority	Boolean	No	`False`
	isDistributed	Boolean	No	`False`
	isReplicated	Boolean	No	`False`
`parameter_key`	Type	`TYPE`	No	`None`
	Direction	`DIRECTION`	No	`IN`
`@constraint(...)`	*Same than Java*			
`@implement(...)`	source_class	String	Yes	
	method	String	Yes	

TABLE 3.2: Method and Parameter Annotations for Python.

Similarly, Table 3.3 summarises the available Method and Parameter annotations for Java.

Method Annotation	Parameters			
	Name	**Value Type**	**Mandatory?**	**Default Value**
`@Method(...)`	declaringClass	String	Yes	
	name	String	Yes	
	isModifier	String	No	"`true`"
	priority	String	No	"`false`"
	constraints	`@Constraints`	No	`NULL`
`@Service(...)`	namespace	String	Yes	
	name	String	Yes	
	port	String	Yes	
	operation	String	No	"`[unassigned]`"
	priority	String	No	"`false`"
`@Constraint(...)`	processors	`@Processor`	No	`NULL`
	computingUnits	String	No	"`[unassigned]`"
	processorName	String	No	"`[unassigned]`"
	processorSpeed	String	No	"`[unassigned]`"
	processorArchitecture	String	No	"`[unassigned]`"
	processorInternalMemorySize	String	No	"`[unassigned]`"
	processorType	String	No	"`CPU`"
	processorPropertyName	String	No	"`[unassigned]`"
	processorPropertyValue	String	No	"`[unassigned]`"
	memorySize	String	No	"`[unassigned]`"
	memoryType	String	No	"`[unassigned]`"
	storageSize	String	No	"`[unassigned]`"
	storageType	String	No	"`[unassigned]`"
	operatingSystemType	String	No	"`[unassigned]`"
	operatingSystemDistribution	String	No	"`[unassigned]`"
	operatingSystemVersion	String	No	"`[unassigned]`"
	appSoftware	String	No	"`[unassigned]`"
	hostQueues	String	No	"`[unassigned]`"
	wallClockLimit	String	No	"`[unassigned]`"
`@Processor(...)`	computingUnits	String	No	"`[unassigned]`"
	name	String	No	"`[unassigned]`"
	speed	String	No	"`[unassigned]`"
	architecture	String	No	"`[unassigned]`"
	type	String	No	"`[unassigned]`"
	internalMemorySize	String	No	"`[unassigned]`"
	propertyName	String	No	"`[unassigned]`"
	propertyValue	String	No	"`[unassigned]`"
`@SchedulerHints(...)`	isReplicated	Boolean	No	"`false`"
	isDistributed	Boolean	No	"`false`"
`@Parameter(...)`	type	Type	No	`Type.UNSPECIFIED`
	direction	Direction	No	`Direction.IN`
	prefix	String	No	"`null`"

TABLE 3.3: Method and Parameter Annotations for Java.

For more in-depth information about them readers can also check the *COMPSs User Guide: Application Development* [54].

3.1.2 Runtime system

To abstract applications from the underlying infrastructure, COMPSs relies on its Runtime System to spawn a master process on the machine where the application is running and a worker process per available resource (see Figure 3.3). These processes are communicated through the network (using different communication adaptors) and can send messages to each other to orchestrate the distributed execution of the application.

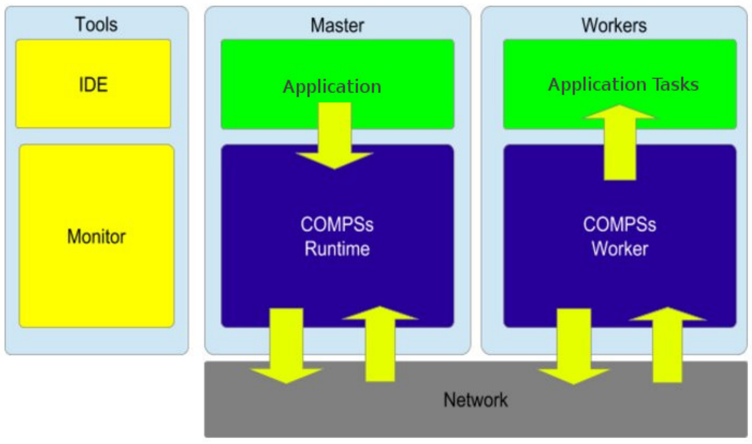


FIGURE 3.3: COMPSs structure.

Once a Java application starts, the COMPSs Runtime [144] triggers a custom Java ClassLoader that uses Javassist [50] to instrument the application's main class. The instrumentation modifies the original code by inserting the necessary calls to the COMPSs API to generate tasks, handle data dependencies and add data synchronisations. To achieve the same

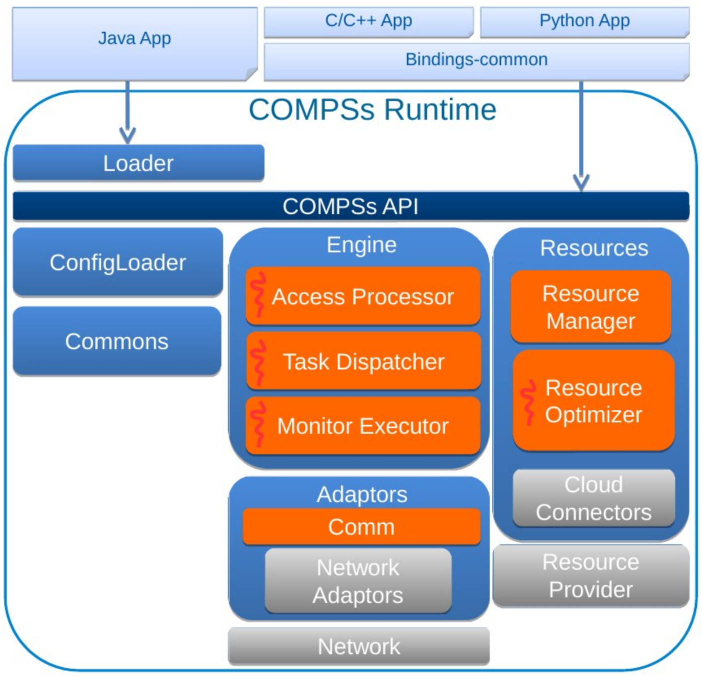


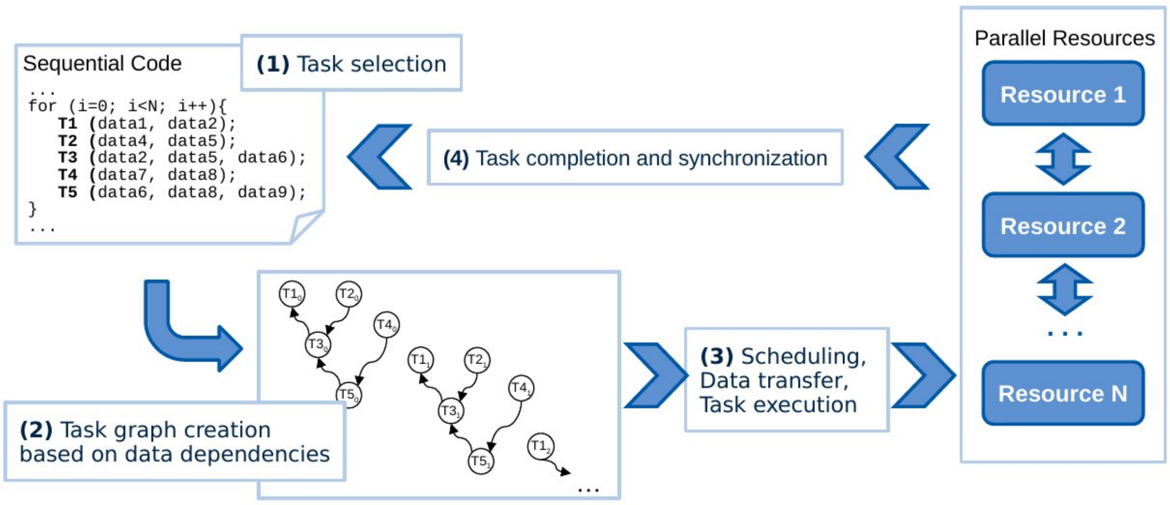
FIGURE 3.4: COMPSs Runtime overview.

purpose on Python applications, the Python Binding (PyCOMPSs) parses the decorators of the main code and adds the necessary calls to the COMPSs API. In the case of C/C++ applications, COMPSs also requires an Interface file that is used when compiling the application to generate stubs for the main code, add the required COMPSs API calls, and generate the code for the tasks execution at the workers. In any case, as shown on the top of the Figure 3.4, the interaction between the application and the COMPSs Runtime is always made through the COMPSs API.

More in-depth, the COMPSs Runtime has five main components:

- **Commons** Contains the common structures used by all the Runtime components
- **ConfigLoader** Loads the project and the resources configuration files, the command-line arguments, and the JVM configuration parameters.
- **Engine** Contains the submodules to handle the task detection, the data dependencies, and the task scheduling. More specifically, the *Access Processor* watches for the data accesses so that the Runtime can build the data dependencies between tasks, the *Task Dispatcher* controls the task life-cycle and the *Monitor Executor* controls the monitor structures for real-time and post-mortem monitoring.
- **Resources** Handles all the available resources in the underlying infrastructure. This component creates, destroys and monitors the state of all the available resources. Since COMPSs supports elasticity through cloud and SLURM [234, 211] connectors, this component contains a *Resource Optimiser* subcomponent that takes care of creating and destroying resources.
- **Adaptors** Contains the different communication adaptors implementations. This layer is used to communicate the COMPSs Master and the COMPSs Workers and abstracts the rest of the Runtime from the different network adaptors.

3.1.3 Task life-cycle



```
...
for (i=0; i<N; i++){
    T1 (data1, data2);
    T2 (data4, data5);
    T3 (data2, data5, data6);
    T4 (data7, data8);
    T5 (data6, data8, data9);
}
...
```

Sequential Code
(1) Task selection
(4) Task completion and synchronization
Parallel Resources
Resource 1
Resource 2
. . .
Resource N
(3) Scheduling, Data transfer, Task execution
(2) Task graph creation based on data dependencies
T1
T2
T3
T4
T5
...

FIGURE 3.5: COMPSs task execution workflow.

To clarify how COMPSs works when executing an application Figure 3.5 describes the task life-cycle. From the application's main code the COMPSs API registers the different tasks. Considering the registered tasks, COMPSs builds a task graph based on the data dependencies. This graph is then submitted to the *Task Dispatcher* that schedules the data-free tasks when possible. This means that a task is only scheduled when it is data-free, and there are enough free resources to execute it (each task can have different constraints, and thus, it is not scheduled if there is not a resource that satisfies the requirements).

Eventually, a task can be scheduled and, then, it is submitted to execution. This step includes the job creation, the transfer of the input data, the job transfer to the selected resource, the real task execution on the worker and the output retrieval from the worker back to the master. If any of these steps fail, COMPSs provides fault-tolerant mechanisms for partial failures.

Once the task has finished, COMPSs stores the monitoring data of the task, synchronises any data required by the application, releases the data-dependent tasks so that they can be scheduled, and deletes the task.

3.2 MPI

The Message-Passing Interface (*MPI*) standard "includes point-to-point message-passing, collective communications, group and communicator concepts, process topologies, environmental management, process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, some miscellaneous topics, and a profiling interface" [159]. The MPI standard includes bindings for C and Fortran, and its goal is "to develop a widely used standard for writing message-passing programs" [159]. It has several implementations but the best known are OpenMPI [176], IMPI [116], MPICH [160], and MVAPICH [164].

For the end user, using MPI requires handling explicitly the spawn of the processes, the code executed by each process and the communication between them. All this management is done by using API calls, and thus, the application code must be compiled and executed with the MPI compiler of the specific MPI implementation. The main advantage is that the users have full control of all the processes and the communication between them which, for experienced users, leads to high efficient codes. However, for inexperienced users, an efficient code can become unreadable and handle many processes can become tedious work. Moreover, when porting a sequential application to an MPI application, the users must explicitly distribute the data between the processes and retrieve back the results (which can lead to load imbalance or inefficient communications). Additionally, another inconvenience of MPI is that, once the application's execution has started, the number of processes cannot be changed dynamically, limiting the malleability of the applications.

Listing 3.7 shows an example of an MPI application written in C. The code spawns a given number of processes, sets up one process as coordinator and the rest as slaves that send back a message to the coordinator saying that they are ready to work. As seen in the figure, each process has a unique identifier, and the communication between them is done by using the MPI_Send or MPI_Receive API calls (obviously, more complex programs will require more complex API calls).

```c
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <mpi.h>
int main(int argc, char **argv) {
    char buf[256];
    int my_rank, num_procs;
    /* Initialize the infrastructure necessary for communication */
    MPI_Init(&argc, &argv);
    /* Identify this process */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    /* Find out how many total processes are active */
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    /* Until this point, all programs have been doing exactly the same.
       Here, we check the rank to distinguish the roles of the programs */
    if (my_rank == 0) {
        int other_rank;
        printf("We have %i processes.\n", num_procs);
        /* Send messages to all other processes */
        for (other_rank = 1; other_rank < num_procs; other_rank++) {
            sprintf(buf, "Hello %i!", other_rank);
            MPI_Send(buf, sizeof(buf), MPI_CHAR, other_rank,
                     0, MPI_COMM_WORLD);
        }
        /* Receive messages from all other process */
        for (other_rank = 1; other_rank < num_procs; other_rank++) {
            MPI_Recv(buf, sizeof(buf), MPI_CHAR, other_rank,
                     0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", buf);
        }
    } else {
        /* Receive message from process #0 */
        MPI_Recv(buf, sizeof(buf), MPI_CHAR, 0,
                 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        assert(memcmp(buf, "Hello ", 6) == 0),

        /* Send message to process #0 */
        sprintf(buf, "Process %i reporting for duty.", my_rank);
        MPI_Send(buf, sizeof(buf), MPI_CHAR, 0,
                 0, MPI_COMM_WORLD);
    }
    /* Tear down the communication infrastructure */
    MPI_Finalize();
    return 0;
}
```

LISTING 3.7: Hello MPI example in C.
Source: Wikipedia, Message Passing Interface.

Listing 3.8 shows the command-line used to spawn this proces and its result, and Figure 3.6 shows a diagram of executing the aforementioned code with 4 processes. Notice that the spawn time of the processes and the communication between them is not always done at the same time and thus, the diagram is only one of the possible execution diagrams of the same code. For instance, Process 0 will always send messages to Processes 1, 2 and 3

```
$ mpicc example.c
$ mpirun -n 4 ./a.out
We have 4 processes.
Process 1 reporting for duty.
Process 2 reporting for duty.
Process 3 reporting for duty.
```

LISTING 3.8: Hello MPI: Execution example.

in the same order and will receive the messages back in the same order, but Processes 1, 2 and 3 can receive the message in different orders and send the reply in different orders. This issue is one of the hardest things to overcome when developing applications with MPI because blocking processes in a receive call can lead to significant overheads. For instance, in our diagram, Processes 2 and 3 have sent all their data, but Process 0 does not receive the message until the data from Process 1 is received.

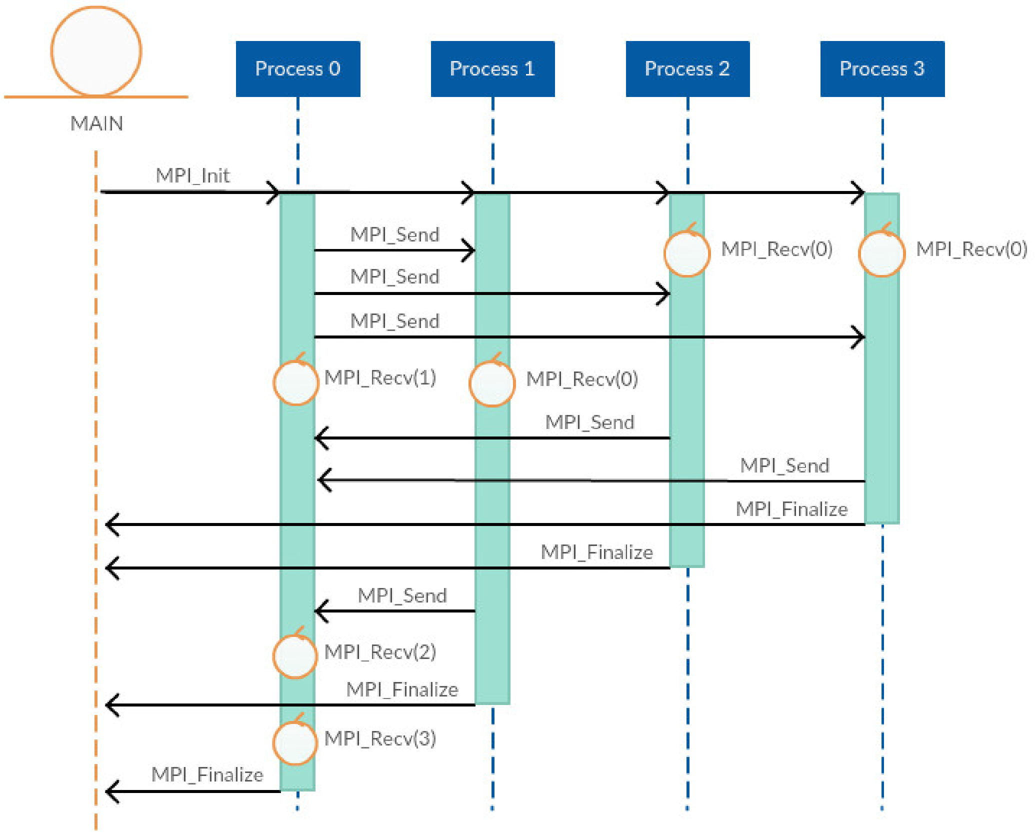


FIGURE 3.6: Hello MPI: Diagram of execution.

Part II

Contributions

Chapter 4

Orchestration of complex workflows

SUMMARY

In contrast to traditional HPC workflows, Data Science applications are more heterogeneous; combining binary executions, MPI simulations, multi-threaded applications, and custom analysis (possibly written in Java, Python, C/C++ or R). This chapter focuses on solving the research question **Q1**; proposing a methodology and an implementation to transparently execute complex workflows by orchestrating different binaries, tools, and frameworks. Our proposal integrates different programming models into a single complex workflow where some steps require highly optimised state of the art frameworks. Hence, this part of the thesis introduces the `@binary`, `@OmpSs`, `@MPI`, `@COMPSs`, and `@MultiNode` annotations to easily orchestrate different programming models inside a single Java or Python workflow. These annotations come along with some scheduling and execution improvements inside the Runtime that make our design easily extensible to include new frameworks in the future. Moreover, the NMMB-MONARCH application demonstrates how to port a real-world use case to our prototype; defining the workflow in both Java and Python languages. During the evaluation, both versions demonstrate a huge increase in programmability while maintaining the same performance.

4.1 General overview

This chapter focuses on solving the research question **Q1**; proposing a methodology and an implementation to transparently execute complex workflows by orchestrating different binaries, tools, and frameworks. Our proposal integrates the heterogeneity of Data Science workflows into a single programming model capable of orchestrating the execution of the different frameworks in a transparent way and without modifying nor its behaviour, nor its syntax. Moreover, it is designed to allow non-expert users to build complex workflows where some steps require a highly optimised state of the art frameworks.

For that purpose, we extend the COMPSs framework to act as an orchestrator rather than a regular application executor. Next subsections provide in-depth details about the modifications of the programming model annotations, the Runtime master, and worker executors. Furthermore, we demonstrate the capabilities of our proposal porting the NMMB-MONARCH application.

4.2 Related Work

Previous research has been conducted regarding the interoperability of programming models. Within the context of the European project Programming Model INTERoperability ToWards Exascale (INTERTWinE) [117], researchers have focused on the interoperability between seven key programming APIs (i.e., MPI, GASPI, OpenMP, OmpSs, StarPU, QUARK and PaRSEC). For instance, C. Simmendinger et al. [209] develop a strategy to significantly improve the performance and the interoperability between the GASPI and MPI APIs by leveraging GASPI shared windows and shared notifications. Also, K. Sala et al. [206] improve the interoperability between MPI and Task-based Programming Models by proposing an API to pause and resume tasks depending on external events (e.g., when an MPI operation blocks, the task running is paused so that the runtime system can schedule a new task on the core that became idle).

Regarding Task-based frameworks for distributed computing, there are two major trends. On the one hand, the frameworks that explicitly define tasks and their dependencies are usually designed to define and orchestrate the workflow and do not contain any code related to the actions performed by the application. Hence, all these frameworks support the execution of external binaries, tools, and libraries since it is the only way to define a task. For instance, Kepler [146], Taverna [113], and FireWorks [10] define custom runners (*ExternalExecution* actor, *ExternalTool*, or *ScriptTask*, respectively) that allow users to run external binary commands and handle their command output, error, and exit code. Kepler and FireWorks cannot build dependencies from the external binary input and output data, but users can define explicit transfer tasks to do so. Conversely, Taverna users can define input and output data and, additionally, they can pack and unpack it before and after the external binary execution. Galaxy [5] requires users to define custom job runners; which can be tedious but provides a larger set of features. Moreover, the custom job runner can be made available for any project. Finally, Pegasus [66] was originally designed to execute binaries. In this case, the users can link the standard output and error of the external binary and mark the required data transfers when defining the job.

On the other hand, the frameworks that implicitly define tasks and their dependencies do not separate the workflow description from the application's functionality and, therefore, the tasks executed by these frameworks are usually defined in the workflow language (i.e., Java, Python, Scala). Although the programming languages them-selves provide multiple ways to execute external binaries, the complexity of executing them (i.e., serialising the data, invoking the external binary, handling the output and error, retrieving the exit code) lies on the users. For instance, Apache Spark [237] does not provide any official way to execute external binaries on RDDs.

4.3 Programming model annotations

As shown in Section 3.1, the COMPSs Programming model defines annotations that must be added to the sequential code in order to run the applications in parallel. These annotations can be split into Method Annotations and Parameter Annotations. Our prototype extends the programming model by providing a new set of Method Annotations and Parameter annotations to support the execution of binaries, multi-threaded applications (OmpSs), MPI simulations, nested COMPSs applications, and multi-node tasks inside a workflow. From now on, tasks that must execute external frameworks are called *Non-Native Tasks*.

4.3.1 Method annotations

A new Method Annotation is defined for each supported non-native task. Notice that the Method Annotation must contain framework related parameters and, thus, its content varies depending on the target framework. Next, we list the currently supported frameworks and their specific parameters.

- **Binaries:** Execution of regular binaries (e.g., BASH, SH, C, C++, FORTRAN)
  - Binary: Binary name or path to a executable file
  - Working Directory: Working directory for the final binary execution
- **OmpSs:** Execution of OmpSs binaries
  - Binary: Path to the execution binary
  - Working Directory: Working directory for the final binary execution
- **MPI:** Execution of MPI binaries
  - Binary: Path to the execution binary
  - MPI Runner: Path to the MPI command to run
  - Computing Nodes: Number of required computing nodes
  - Working Directory: Working directory for the final binary execution
- **COMPSs:** Execution of nested COMPSs workflows
  - Application Name
  - Runcompss: Path to the runcompss command
  - Flags: Extra flags for the nested runcompss command
  - Computing Nodes: Number of required computing nodes
  - Working Directory: Working directory for the nested COMPSs application
- **Multi-node:** Execution of native Java/Python tasks that require more than one node
  - Computing Nodes: Number of required computing nodes

Next, we describe how to define and use each new Method annotation inside Java and Python workflows.

4.3.1.1 Java

Java applications need to be instrumented in such a way that method invocations inside the main code are intercepted by the COMPSs Runtime and substituted by remote task calls when required. In fact, the COMPSs Loader cross-validates the signatures of each method invocation with the tasks defined in the Interface. Notice that, when using regular methods, the annotation contains a mandatory methodClass field so that the signature (method class, method name, and parameters) can be built.

Except for multi-node tasks, the rest of the non-native tasks do not contain the method class. Thus, we force the users to define the remote methods inside specific packages and classes. This design decision is motivated by the fact that we consider that the annotation of non-native tasks must only refer to the real execution and, thus, we want to avoid a *declaringClass* field in the new annotation. Hence, non-native tasks methods must be defined inside the packages and classes defined in Table 4.1.

External Framework	**Remote Methods' package and class**
Binaries	binary.BINARY
OmpSs	ompss.OMPSS
MPI	mpi.MPI
COMPSs	compss.NESTED
MultiNode	-

TABLE 4.1: Java package and class for non-native tasks' remote methods.

On the other hand, Table 4.2 shows the syntax to define the different non-native tasks inside the Interface file. Notice that the task parameters must be annotated as with regular tasks.

External Framework	Method Annotation	Parameters		
		Name	**Value Type**	**Mandatory?**
Binaries	`@Binary(...)`	binary	String	Yes
		workingDir	String	No
		priority	Boolean	No
		constraints	@Constraints	No
OmpSs	`@OmpSs(...)`	binary	String	Yes
		workingDir	String	No
		priority	Boolean	No
		constraints	@Constraints	No
MPI	`@MPI(...)`	binary	String	Yes
		mpiRunner	String	Yes
		computingNodes	String	Yes
		workingDir	String	No
		priority	Boolean	No
		constraints	@Constraints	No
COMPSs	`@COMPSs(...)`	appName	String	Yes
		runcompss	String	Yes
		computingNodes	String	Yes
		flags	String	No
		workingDir	String	No
		priority	Boolean	No
		constraints	@Constraints	No
MultiNode	`@MultiNode(...)`	declaringClass	String	Yes
		computingNodes	String	No
		isModifier	Boolean	No
		priority	Boolean	No
		constraints	@Constraints	No

TABLE 4.2: Definition of external tasks (Method Annotations and their parameters) for Java workflows.

Finally, the calls to non-native tasks in the main workflow code are similar to regular method calls. For instance, Listings 4.1, 4.2, and 4.3 show, respectively, the annotation of one binary task, its remote method definition, and the main method calling it.

```java
package app;

import es.bsc.compss.types.annotations.task.Binary;

public interface MainItf {

    @Binary(binary = "path_to_bin")
    void myBinaryTask(
    );
}
```

LISTING 4.1: Binary task definition example in Java.

```java
package binary;

public class BINARY {

    public static void myBinaryTask() {
    }
}
```

LISTING 4.2: Binary remote method definition example in Java.

```java
package app;

import binary.BINARY;

public class Main {

    public static void main(String[] args) {
        BINARY.myBinaryTask();
    }
}
```

LISTING 4.3: Binary task invocation example in Java.

4.3.1.2 Python

Table 4.3 shows the syntax to define non-native tasks inside a Python workflow. Notice that the Method Annotations must be placed on top of the task decorator (`@task`) that annotates the object or class method that they refer.

External Framework	Method Annotation	Parameters		
		Name	**Value Type**	**Mandatory?**
Binaries	`@binary(...)`	binary	String	Yes
		workingDir	String	No
		priority	Boolean	No
OmpSs	`@ompss(...)`	binary	String	Yes
		workingDir	String	No
		priority	Boolean	No
MPI	`@mpi(...)`	binary	String	Yes
		mpiRunner	String	Yes
		computingNodes	String	Yes
		workingDir	String	No
		priority	Boolean	No
COMPSs	`@compss(...)`	appName	String	Yes
		runcompss	String	Yes
		computingNodes	String	Yes
		flags	String	No
		workingDir	String	No
		priority	Boolean	No
MultiNode	`@multinode(...)`	computingNodes	String	Yes
		priority	Boolean	No

TABLE 4.3: Definition of external tasks (Method Annotations and their parameters) for Python workflows.

On the other hand, the calls to non-native tasks in the main workflow code are similar to regular method calls. Moreover, in contrast to Java workflows, non-native tasks can have any signature when defining Python workflows since they are defined in-place using a decorator. For instance, Listing 4.4 shows the definition of one binary remote method (without any parameter) and its invocation from the main code.

```python
@binary(binary = "path_to_bin")
@task()
def myBinaryTask():
    pass

def main():
    myBinaryTask()

if __name__ == '__main__':
    main()
```

LISTING 4.4: Binary task definition, remote method and invocation example in Python.

4.3.2 Parameter annotations

In order to input and output data from the execution of non-native tasks the Parameter Annotation also needs to be enhanced. When using multi-node tasks the parameters and the return value of the task are the same than when using a regular method task. However, when executing standalone binaries, OmpSs processes, MPI processes, or COMPSs applications the exit value of the processes is used as the return value. Thus, we have decided that the COMPSs non-native tasks must use the exit value of their internal binary as the return value of the task. In this sense, our prototype allows the users to capture this value by defining the return type of the non-native task as an *int* (for implicit synchronisation), as an *Integer* (for post-access synchronisation) or to forget it (declaring the function as *void*). Listing 4.5 shows an Interface example of the three return types.

```java
public interface MainItf {

    @Binary(binary = "${BINARY}")
    int binaryTask1();

    @Binary(binary = "${BINARY}")
    Integer binaryTask2();

    @Binary(binary = "${BINARY}")
    void binaryTask3();
}
```

LISTING 4.5: Example of the different return types of the non-native tasks.

However, the users not only need the process exit value to work with this kind of applications but need to set the Standard Input (*stdIn*) and capture the Standard Output (*stdOut*) and Error (*stdErr*). For this purpose, our prototype includes a new parameter annotation, `StdIOStream`, that allows the users to set some parameters as standard I/O streams for the non-native tasks. Standard I/O Stream parameters are not passed directly to the binary command but rather they are set as *stdIn*, *stdOut*, or *stdErr* of the binary process. Since this kind of redirection is restricted to files in *LINUX* Operating Systems, we have decided to keep the same restrictions to the annotation. Consequently, **all *StdIOStream* parameters must be files**.

Listing 4.6 shows the Interface of a Java application with two tasks that have a normal parameter (the first one, that will be sent directly to the binary execution), a file parameter to be used as *stdIn* of the process, a file parameter to be used as *stdOut* and a last file parameter to be used as *stdErr*. The difference between `task1` and `task2` in this example is that the first task will overwrite the *fileOut* and *fileErr* content (since the files are opened in *write* mode), and the second task will append the *fileOut* and *fileErr* content at the end of the file (since the files are opened in *append* mode).

```java
public interface StreamItf {

    @Binary(binary = "${BINARY}")
    Integer task1(
        @Parameter(type = Type.STRING, direction = Direction.IN)
            String normalParameter,
        @Parameter(type = Type.FILE, direction = Direction.IN,
                   stream = StdIOStream.STDIN)
            String fileIn,
        @Parameter(type = Type.FILE, direction = Direction.OUT,
                   stream = StdIOStream.STDOUT)
             String fileOut,
        @Parameter(type = Type.FILE, direction = Direction.OUT,
                   stream = StdIOStream.STDERR)
             String fileErr
    );

    @Binary(binary = "${BINARY}")
    Integer task2(
        @Parameter(type = Type.STRING, direction = Direction.IN)
            String normalParameter,
        @Parameter(type = Type.FILE, direction = Direction.IN,
                   stream = StdIOStream.STDIN)
            String fileIn,
        @Parameter(type = Type.FILE, direction = Direction.INOUT,
                   stream = StdIOStream.STDOUT)
            String fileOut,
        @Parameter(type = Type.FILE, direction = Direction.INOUT,
                   stream = StdIOStream.STDERR)
            String fileErr
    );
}
```

LISTING 4.6: Example of the different standard I/O stream annotations for non-native tasks in Java.

Listing 4.7 shows the exact same application but in Python. For the sake of clarity, we have added the verbose annotations (tasks `task1` and `task2`) and its compact form (tasks `task1_bis` and `task2_bis`).

```python
@binary(binary = "${BINARY}")
@task(file_in={Type:FILE_IN, Stream:STDIN},
      file_out={Type:FILE_OUT, Stream:STDOUT},
      file_err={Type:FILE_OUT, Stream:STDERR})
def task1(normal_parameter, file_in, file_out, file_err):
    pass

@binary(binary = "${BINARY}")
@task(file_in={Type:FILE_IN, Stream:STDIN},
      file_out={Type:FILE_INOUT, Stream:STDOUT},
      file_err={Type:FILE_INOUT, Stream:STDERR})
def task2(normal_parameter, file_in, file_out, file_err):
    pass

# Compact form
@binary(binary = "${BINARY}")
@task(file_in={Type: FILE_IN_STDIN},
      file_out={Type: FILE_OUT_STDOUT},
      file_err={Type: FILE_OUT_STDERR})
def task1_bis(normal_parameter, file_in, file_out, file_err):
    pass

@binary(binary = "${BINARY}")
@task(file_in={Type: FILE_IN_STDIN},
      file_out={Type: FILE_INOUT_STDOUT},
      file_err={Type: FILE_INOUT_STDERR})
def task2_bis(normal_parameter, file_in, file_out, file_err):
    pass
```

LISTING 4.7: Example of the different stream annotations for non-native tasks in Python.

Finally, to summarise the last information retrieved from this example, Table 4.4 show the available modes for each stream type.

Type	Stream	Direction	Description
FILE	StdIOStream.STDIN	Direction.IN	Sets the process *stdIn*. The file is opened in *read* mode
FILE	StdIOStream.STDOUT	Direction.OUT	Sets the process *stdOut*. The file is opened in *write* mode
FILE	StdIOStream.STDOUT	Direction.INOUT	Sets the process *stdOut*. The file is opened in *append* mode
FILE	StdIOStream.STDERR	Direction.OUT	Sets the process *stdErr*. The file is opened in *write* mode
FILE	StdIOStream.STDERR	Direction.INOUT	Sets the process *stdErr*. The file is opened in *append* mode

TABLE 4.4: Available stream types with their valid directions and execution behaviour.

4.3.2.1 Prefix parameter annotation

COMPSs builds the data dependence graph taking into account the parameters annotated in the application Interface. Analysing several binaries, we have found out that a non-negligible part of them use prefixes for each parameter. The prefixes used by binaries can be divided into two types:

- **Separated Prefix** A prefix that is written separately before the parameter value. This type of prefixes are of the form:

  ```
  $ ./binary -param1 value --param2 value -k value
  ```

  In fact, there is not a strong need that the parameter prefix starts with a dash but its the common behaviour for Linux binaries.

- **Joint Prefix** A prefix that is written with the parameter value without being separated or with a separation character that it is not an empty space. This types of prefix vary a lot but are of the form:

```
$ ./binary -pValue -q=value --r=value s=value
```

The separated prefixes do not represent a problem for the COMPSs programming model since they can be defined as a standalone string parameter that is finally passed to the binary. However, the joint prefixes do represent a problem for COMPSs since the users must prepend the prefix to the parameter, breaking the data dependencies between the tasks. For the sake of clarity, consider the two tasks shown in Listing 4.8 and the code shown in Listing 4.9. Since the second task requires a joint prefix, when calling it from the main code the users must modify its value and prepend the prefix to the *fileName* variable. This string modification causes a synchronisation in the application's main code instead of creating a data dependency between the two tasks.

```java
// Must execute: ./tmp/bin1 fileName
@Binary(binary = "/tmp/bin1")
Integer task1(
@Parameter(type = Type.FILE, direction = Direction.INOUT)
    String fileName
);
// Must execute: ./tmp/bin2 --file=fileName
@Binary(binary = "/tmp/bin2")
Integer task2(
@Parameter(type = Type.FILE, direction = Direction.INOUT)
    String fileName
);
```

LISTING 4.8: Binary tasks example in Java for joint prefixes.

```java
String fileName = "/tmp/file";
BINARY.task1(fileName);
BINARY.task2("--file=" + fileName);
```

LISTING 4.9: Main code example in Java for joint prefixes.

Consequently, for this second type of prefixes, our prototype implements a new parameter annotation *prefix* that allows the users to define the prefix separately to the parameter value and its prepended to the parameter value just before the binary execution. This modification allows COMPSs to handle the data dependencies between parameters (since prefixes are immutable strings that do not define data dependencies) and allows the binaries to receive the parameter prefixes and its value together as a single parameter.

Listing 4.10 shows the main code of an application containing three binary tasks. Listing 4.11 shows the final binary command, and Listing 4.12 the definition of the binary tasks in the Interface file. Notice that the first task (*task1*) only uses separated prefixes; the second task (*task2*) uses only joint prefixes and the third task (*task3*) is a hybrid example of both separated and joint prefixes.

```java
public static void main(String[] args)
    String file1 = "file1.in"
    String file2 = "file2.inout"
    int kValue = 10;
    // Launch task 1
    task1("-p", file1, "--q", file2, "k", kValue);
    // Launch task 2
    task2(file1, file2, kValue);
    // Launch task 3
    task3("-p", file1, file2, kValue);
}
```

LISTING 4.10: Example of the main code calls to tasks with prefixes.

```
# TASK 1
./binaryExample -p file1.in --q file2.inout k 10
# TASK 2
./binaryExample -p=file1.in --q=file2.inout k10
# TASK 3
./binaryExample -p file1.in --q=file2.inout k10
```

LISTING 4.11: Example of the command executed inside each task using prefixes.

```java
@Binary(binary = "binaryExample")
void task1(
    @Parameter(type = Type.STRING, direction = Direction.IN)
        String pPrefix,
    @Parameter(type = Type.FILE, direction = Direction.IN)
        String fileIn,
    @Parameter(type = Type.STRING, direction = Direction.IN)
        String qPrefix,
    @Parameter(type = Type.FILE, direction = Direction.INOUT)
        String fileInOut,
    @Parameter(type = Type.STRING, direction = Direction.IN)
        String kPrefix,
    @Parameter(type = Type.INT, direction = Direction.IN)
        int k
);

@Binary(binary = "binaryExample")
void task2(
    @Parameter(type = Type.FILE, direction = Direction.IN,
        prefix = "-p=") String fileIn,
    @Parameter(type = Type.FILE, direction = Direction.INOUT,
        prefix = "--q=") String fileInOut,
    @Parameter(type = Type.INT, direction = Direction.IN,
        prefix = "k") int k
);

@Binary(binary = "binaryExample")
void task3(
    @Parameter(type = Type.STRING, direction = Direction.IN)
        String pPrefix,
    @Parameter(type = Type.FILE, direction = Direction.IN)
        String fileIn,
    @Parameter(type = Type.FILE, direction = Direction.INOUT,
        prefix = "--q=") String fileInOut,
    @Parameter(type = Type.INT, direction = Direction.IN,
        prefix = "k") int k
);
```

LISTING 4.12: Interface example of an application with prefixes.

4.4 Runtime master

At the master side, the Runtime works with an abstract entity known as `Task` that allows the Task Dispatcher and the Task Scheduler to be independent of the final execution framework while the Task object still contains information about the final execution framework. However, our proposal enhances the task detection to support new programming model annotations introduced by the non-native tasks and the Task Scheduler to support multi-node tasks.

4.4.1 Task detection

On the one hand, as previously explained, Java applications need to be instrumented in order to detect the potential tasks. Since non-native tasks do not define a method class to build the method signature, the COMPSs Loader looks for a pre-defined set of signatures to detect the execution of non-native tasks. Thus, the Loader of our prototype has been extended to consider as potential task any method defined inside the `binary.BINARY`, `ompss.OMPSS`, `mpi.MPI`, and `compss.COMPSS` classes.

On the other hand, the Python binding registers the annotated tasks into the Runtime with no further handling. The tasks are detected by means of Python decorators on top of the object or class method. Hence, our prototype only extends the previous COMPSs version by adding new decorators for the non-native tasks (i.e., `@binary`, `@ompss`, `@mpi`, `@compss`, and `@multinode`).

4.4.2 Task scheduler

As previously explained in Section 3.1, the COMPSs Runtime instruments the application's main code looking for invocations to the methods defined as tasks in the application interface. When these methods are detected, COMPSs creates a task that is submitted to the Task Analyser component and substitutes the method call by an *executeTask()* call. When the Task Analyser receives a new task, it computes its data dependencies and submits it to the Task Scheduler. Next, the Task Scheduler creates an Execution Action associated with the task and adds it to the execution queue. Eventually, the Execution Action will be scheduled and launched (this mechanism requires the task to be data-free and to have enough free resources to fulfil the task constraints). When the Execution Action is launched, a Job is created to monitor the task execution. This job includes the transfer of the job definition and all the input data to the target COMPSs Worker, the real task execution in the worker and the transfer of the output data back to the COMPSs Master. Once the job is completed, its data dependent Execution Actions are released (if any), and the job is destroyed (or, depending on the debug level, stored for post-mortem analysis).

Since non-native tasks only represent a new way of executing tasks in the COMPSs Worker, the Scheduler component does not require any modification to handle binary, and OmpSs tasks. However, our prototype includes support for tasks using more than one computational node (i.e., MPI, COMPSs, and multi-node) which requires to support Multi-node execution actions inside the Scheduler. To do so, we have associated several execution actions to the same task so that the Scheduler can map the data-dependencies and the resource consumption as it was done before. However, the execution actions associated with the same task must have different behaviours during the execution phase because only one of the actions must really launch the job.

Consequently, we have extended the *ExecutionAction* in a *MultiNodeExecutionAction* class that is only used when a task requires more than one computing node (otherwise the previous *ExecutionAction* implementation is used). When the task scheduler receives a new multi-node task (*ExecuteTaskRequest*) it creates a new *MultiNodeGroup* instance and *N MultiNodeExecutionAction* instances (being N the number of nodes requested by the task). The *MultiNodeGroup* instance is shared among all the actions assigned to the same task execution, and it handles the actions' id within the group. More in-depth, when the *MultiNodeExecutionActions* are created the *MultiNodeGroup* assigns a nullable identifier to all of them. Once the actions are scheduled and launched, the *MultiNodeGroup* assigns a unique valid identifier between 1 and N. This action identifier is used during the action execution to act as an execution slave node (when the assigned identifier is different to 1) or to act as an execution master node (when the assigned identifier is 1). When the *MultiNodeExecutionAction* is identified as a slave, it no longer triggers a job execution, but rather reserves the requested resources and waits for its master action to complete. When the *MultiNodeExecutionAction* is identified as a master, it retrieves all the hostnames of its slave actions (for the MPI command) and behaves as a normal ExecutionAction (launches a job to monitor the input data transfers, the real task execution on the node and the output data transfers).

On the one hand, Figure 4.1 shows an example of the normal process. A task *T1* requiring 1 node (normal task) is submitted to the scheduler through the *ExecuteTaskRequest* request. The request is then processed and an *ExecutionAction* is created as it was done before. Eventually, the action is scheduled, launched and finally executed, creating a new job that will monitor the task execution in the target node.

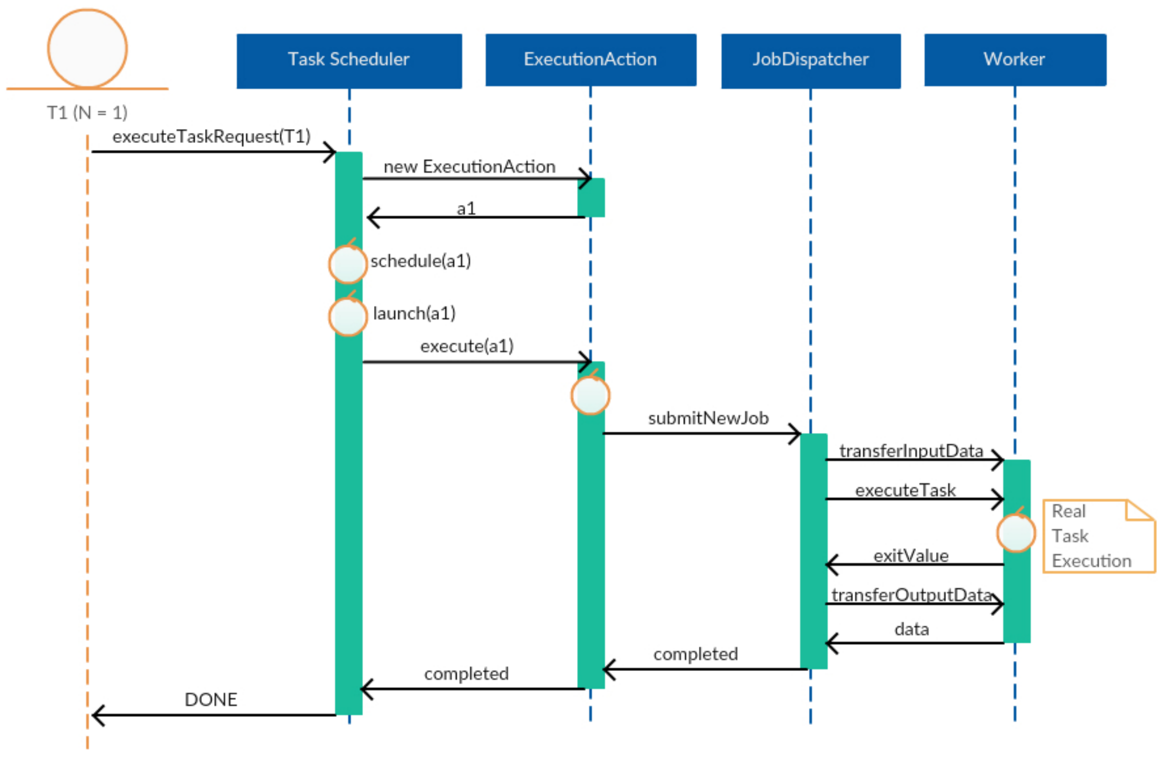


FIGURE 4.1: Example of a single node task flow.

On the other hand, Figure 4.2 shows an example of the Multi-Node process. A task *T2* requiring 3 nodes (multi-node task) is submited to the scheduler through the same *ExecutionTaskRequest* request. The request is then processed: a new action group (lets say *g1*) is created (a new instance of the *MultiNodeGroup*) and 3 *MultiNodeAction* instances (lets say *a1*, *a2* and *a3*) are created. The action group *g1* is shared among all the three actions and assigns a nullable action identifier to all of them.

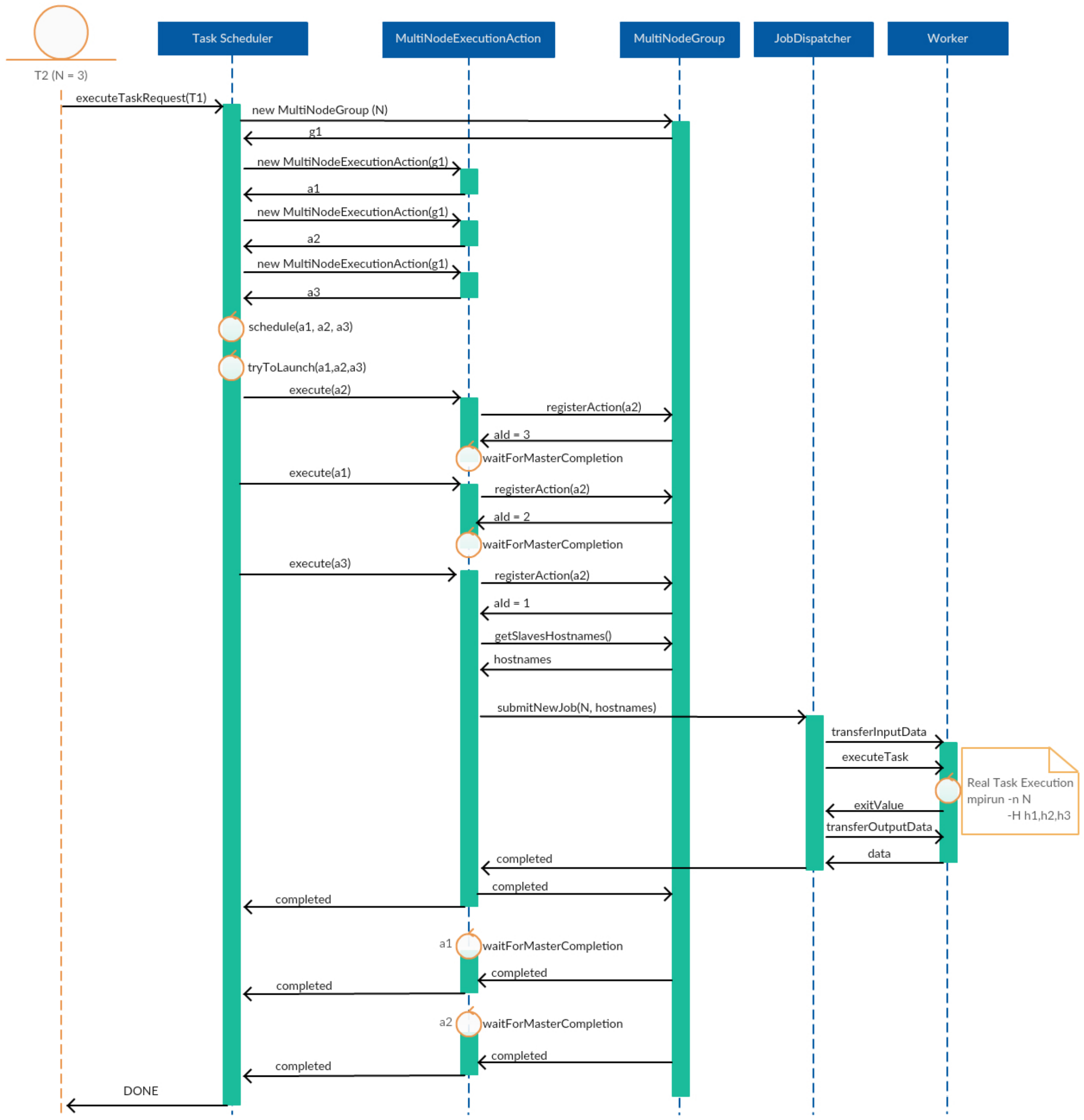


FIGURE 4.2: Example of a multi-node task flow.

Eventually, *a2* is scheduled, launched and finally executed. On the execution phase, the action asks for an action identifier and the action group *g1* assigns it an $actionId = 3$ (because the group size is 3 and no action has previously requested an identifier). Since the action identifier classifies *a2* as a slave action, the execute phase only reserves the task constraints and waits for the master action completion.

Eventually, *a1* is also scheduled, launched and finally executed. Following the same process than the previous action, *a1* is granted with $actionId = 2$ (because the group size is 3 and only one action has previously requested an identifier). Since the action *a1* is also classified as a slave, it reserves the task constraints and waits for the master action completion.

Finally, *a3* will also be scheduled, launched and finally executed. In this case, the action group assigns it an *actionId* = *1*. Since it is the last action, it is now identified as master and during its execution phase it retrieves the hostnames of the resources assigned to all the actions inside the *g1* group (lets say, *h1* for *a1* and *h2* for *a2*) and launches the execution job. The job will be then executed (lets say that the host assigned to this action *a3* is *h3*) monitoring the input data transfers, performing the real task execution (for example, calling the MPI command inside the host *h3* with 3 nodes *h1*, *h2* and *h3*) and retrieving back the output data from *h3*.

Once the job is completed, the action *a3* is marked as completed (freeing all the reserved resources) and, then, it triggers its completion to all the slave actions registered in the group *g1*. Consequently, *a1* and *a2* are also marked as completed (and its resources are also freed). When all the actions within the group are marked as completed, the task is registered as *DONE* and follows the usual process: frees its data dependent tasks and it is stored for post-mortem analysis.

Finally, notice that this process could lead to deadlocks or unused resources when multiple multi-node tasks are being scheduled at the same time. To avoid so, the Task Scheduler component keeps track of the action groups that have tasks that have already been scheduled and priorises the tasks belonging to those groups in front of the rest.

4.5 Worker executors

The Communication layer abstracts the Master node from the specific Communication Adaptors and thus, from the underlying infrastructure. However, the worker processes spawned by this layer are dependent on each Adaptor implementation. Currently, COMPSs supports the *NIO* and the *GAT* Communication Adaptors.

On the one hand, the *GAT* Adaptor is built on top of the Java Grid Application Toolkit (JavaGAT) [189] which relies on the SSH connection between nodes. During the application execution, the Runtime spawns a new worker process per task execution. More specifically, when a task must be executed, the GAT Communication Adaptor creates a GAT Job, sends the job and the required data through SSH to the worker's resource, starts the worker process, executes the task itself, closes the worker process, and retrieves the job status, the job's log files, and the required output data. The *worker.sh* script orchestrates all the processes and launches a language dependent script for the real task execution (*GATWorker.java* for Java, *worker.py* for Python and *Worker* for C/C++). Although the implementation suffers from some performance overheads (because the overhead of spawning a new process on each task execution becomes non-negligible for small duration tasks), it provides a high connectivity interface since it only requires the SSH port to be opened.

On the other hand, the *NIO* Adaptor is a more sophisticated implementation based on Java New I/O (NIO) library [179]. This adaptor spawns a persistent Java Worker Process per resource, rather than one per task execution, and the communication between Master and Workers is then made through Sockets. Hence, this Adaptor provides better performance than the *GAT* Adaptor but requires extra open ports between the available resources. Furthermore, the Worker processes persist during the full execution of the application, what also lets us have an object cache per worker, data communications between workers (rather than handling all the data in the Master resource) and thread binding mechanisms to map threads to specific cores of the machine. Finally, for the task execution, each worker has several *Executor* threads that can execute natively Java applications, or Python and C/C++ applications using a *ProcessBuilder*.

4.5.1 Invokers

Before executing any non-native task, the worker sets up some environment variables so that the users can retrieve, if necessary, information about the assigned resources inside the task code. Table 4.5 shows the defined variables.

Environment Variable	Description
`COMPSS_NUM_NODES`	Number of computing nodes
`COMPSS_HOSTNAMES`	List of computing node names or IPs
`COMPSS_NUM_THREADS`	Number of threads per process
`OMP_NUM_THREADS`	Number of threads per process

TABLE 4.5: Environment variables defined by the worker.

Furthermore, to enable the execution of non-native tasks for all the communication adaptors, we have implemented a *GenericInvoker* class that provides an API for executing standard, MPI, COMPSs, and OmpSs binaries. This API is built on top of a *BinaryRunner* class that spawns, runs and monitors the execution of any binary command. MultiNode tasks are executed as regular method tasks, with the difference that the scheduler has previously reserved other slave nodes.

More specifically, the *BinaryRunner* class has two methods. Firstly, *createCMDParametersFromValues* serializes the received parameters to construct the binary arguments. This method is also in charge of processing the *StdIOStream* annotations and redirecting the StdIn, StdOut, and StdErr when required. Secondly, *executeCMD* executes the received binary command (with all its parameters), monitors its execution and, finally, returns the exit value of the process.

On the other hand, the *GenericInvoker* class provides four functions: *invokeBinaryMethod*, *invokeMPIMethod*, *invokeCOMPSsMethod*, and *invokeOmpSsMethod* to invoke respectively standard, MPI, COMPSs, and OmpSs binaries. The four methods receive the binary path and the argument values, construct and execute the command by calling the *BinaryRunner* functions, and return the exit value of the binary execution.

Finally, our prototype also adapts each of the Communication Adaptor (*GAT* and *NIO*) to call this *GenericInvoker* when needed. In both cases, we have substituted the normal task execution by a *switch-case* that selects the required invoker considering the task's implementation type.

4.6 Use Case: NMMB-MONARCH

The NMMB-MONARCH application is an example of any application containing (i) multiple calls to any kind of binaries and scripts (e.g., R, BASH, PERL), (ii) multi-core or multi-node simulations (e.g., using OpenMP or MPI), and (iii) custom analysis written in Java or Python. Hence, the outcomes described in this section can be applied to any other application with similar requirements. However, we have chosen NMMB-MONARCH because of its availability and impact since it is used in production by another BSC department.

4.6.1 Application overview

The NMMB-MONARCH is a fully online multiscale chemical weather prediction system for regional and global-scale applications. The system is based on the meteorological Non-hydrostatic Multiscale Model on the B-grid [119], developed and widely verified at the National Centers for Environmental Prediction (NCEP). The model couples online the NMMB with the gas-phase and aerosol continuity equations to solve the atmospheric chemistry processes in detail. It is also designed to account for the feedbacks among gases, aerosol particles, and meteorology.

As shown in Figure 4.3, the NMMB-MONARCH workflow is composed by five main steps, namely *Initialisation*, *Fixed*, *Variable*, *UMO Model*, and *Postprocess*.

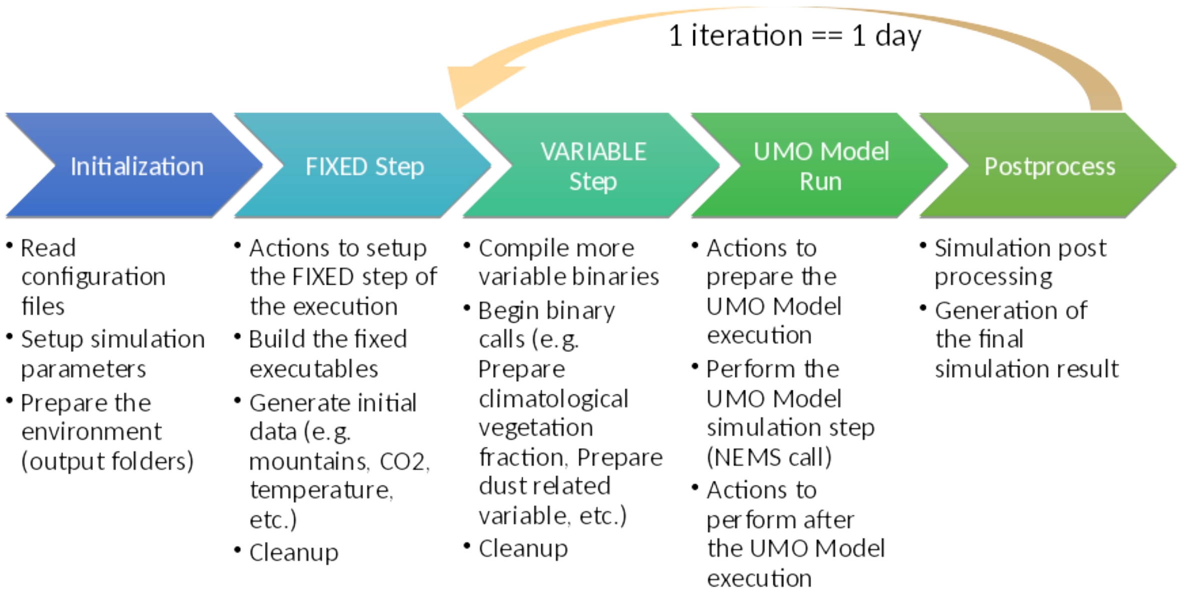

1 iteration == 1 day
Initialization
FIXED Step
VARIABLE Step
UMO Model Run
Postprocess
• Read configuration files
• Setup simulation parameters
• Prepare the environment (output folders)
• Actions to setup the FIXED step of the execution
• Build the fixed executables
• Generate initial data (e.g. mountains, CO2, temperature, etc.)
• Cleanup
• Compile more variable binaries
• Begin binary calls (e.g. Prepare climatological vegetation fraction, Prepare dust related variable, etc.)
• Cleanup
• Actions to prepare the UMO Model execution
• Perform the UMO Model simulation step (NEMS call)
• Actions to perform after the UMO Model execution
• Simulation post processing
• Generation of the final simulation result

FIGURE 4.3: NMMB-MONARCH: Structure diagram.

The first two steps are performed only once per execution since their objective is to set up the configuration. The *Initialisation* step prepares the environment, and the *Fixed* step performs the global actions that will be necessary for the whole simulation. More in detail, this phase compiles the main binaries for the simulation and generates the initial data. This includes reading global datasets with different parameters such as landuse, topography, vegetation, soil, etc. to build the initial files for the model execution and the definition of the simulation (horizontal resolution, vertical levels, and region if it is not a global run).

The *Variable*, *UMO Model* and *Postprocess* are performed iteratively per initial simulation day. During the *Variable* step, some particular binaries are compiled to generate the conditions of the day to simulate, such as the meteorological input conditions or boundary conditions.

During the *UMO Model* step, the simulation main binary (NEMS) is invoked. In contrast to the previous binaries used in the process, the NEMS is a Fortran 90 application parallelised with MPI. Therefore, NEMS can be executed in multiple cores and multiple nodes, relying on the MPI paradigm. The model uses the Earth System Modelling Framework library as the main framework [108]. Finally, after performing the *UMO Model* step, the *Postprocess* step converts the binary simulation results from the NEMS output to a more friendly and usable format such as NetCDF, and stores them in a particular folder.

In a nutshell, the NMMB-MONARCH application is structured in five main steps and composed by a set of binaries that perform individual actions in a coordinated way for achieving the dust simulation results. Among these binaries, the one that stands out from the rest is the NEMS binary, which is implemented in MPI.

4.6.2 Parallelisation design

The original NMMB-MONARCH application consists in a BASH script defining the main workflow and a set of Fortran binaries. Since our prototype supports Java and Python workflows, we have ported two different versions of the NMMB-MONARCH. Both of them parallelise the main workflow while keeping the Fortran binaries.

Regarding the parallelisation, all the binaries have been considered as tasks. For instance, Listings 4.13 and 4.14 show the annotation of the `deeptemperature` binary in Java and Python respectively.

```java
@Binary(binary = "/path/to/deeptemperature.x")
Integer deeptemperature(
    @Parameter(type = Type.FILE, direction = Direction.IN)
        String seamask,
    @Parameter(type = Type.STRING, direction = Direction.IN)
        String soiltempPath,
    @Parameter(type = Type.FILE, direction = Direction.OUT)
        String deeptemperature
);
```

LISTING 4.13: NMMB-MONARCH: Annotation of the `deeptemperature` binary in Java.

```python
@binary(binary='/path/to/deeptemperature.x')
@task(returns=int,
      seamask=FILE_IN,
      deep_temperature=FILE_OUT)
def deeptemperature():
    pass
```

LISTING 4.14: NMMB-MONARCH: Annotation of the `deeptemperature` binary in Python.

As previously explained, the NEMS simulator invoked within the *UMO Model* step is implemented using MPI. Thus, in contrast to the rest of binaries, we have annotated it as an MPI task. Listings 4.15 and 4.16 show the annotation of the `NEMS` binary in Java and Python respectively. Notice that, considering that the NEMS execution can be performed with a different number of nodes, the constraint decorator attached to the MPI task may vary between executions. Hence, to ease this management to users, the constraint has been defined using an environment variable.

Since the NMMB-MONARCH produces a vast amount of outputs (41 different variables), we have included a final step to ease the results interpretation. Hence, the last step retrieves the outputs from the entire simulation and produces a set of animated plots for each of the analysed variables. More in detail, this last step (known as *Figures and animations creation*) is composed by two different tasks: the first one aimed at producing the images of a particular variable per day, and a second one to gather all images that belong to the same variable and build the corresponding animation.

```java
@MPI(mpiRunner = "mpirun",
    binary = "/path/to/NEMS.x",
    workingDir = "/path/to/nems/out",
    computingNodes = "${NEMS_NODES}")
@Constraints(computingUnits = "${NEMS_CUS_PER_NODE}")
Integer nems(
    @Parameter(type = Type.FILE, direction = Direction.OUT,
        stream = StdIOStream.STDOUT) String stdOutFile,
    @Parameter(type = Type.FILE, direction = Direction.OUT,
        stream = StdIOStream.STDERR) String stdErrFile
);
```

LISTING 4.15: NMMB-MONARCH: Annotation of the `nems` MPI binary in Java.

```python
@constraint(computingUnits='$NEMS_CUS_PER_NODE')
@mpi(runner='mpirun',
     binary='/path/to/NEMS.x',
     workingDir='/path/to/nems/out',
     computingNodes='$NEMS_NODES')
@task(returns=int,
      stdOutFile=FILE_OUT_STDOUT,
      stdErrFile=FILE_OUT_STDERR)
def nems():
    pass
```

LISTING 4.16: NMMB-MONARCH: Annotation of the `nems` MPI binary in Python.

To conclude, the final workflow structure for both the Java and the Python versions is depicted in Figure 4.4. At the end of the execution, the scientists can check the simulation results from the images and animations. Additionally, they can also check the raw results from the simulation or use them as input for other specific analysis.

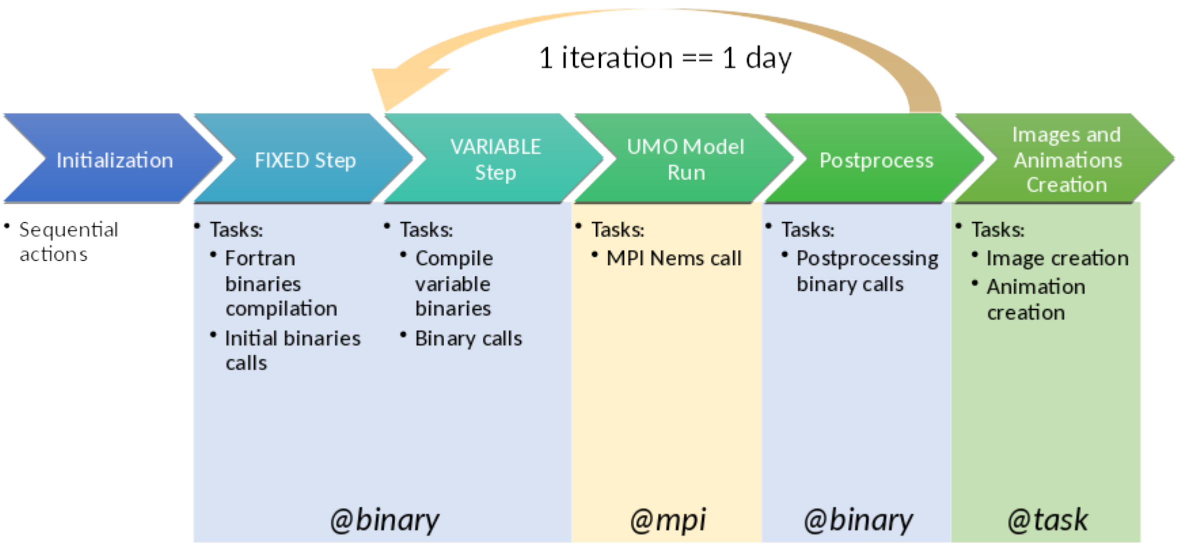


FIGURE 4.4: NMMB-MONARCH: Structure diagram with tasks definition.

4.6.3 Evaluation

4.6.3.1 Parallelisation analysis

Regarding the code, as shown in Table 4.6, the NMMB-MONARCH application was originally composed by a set of BASH scripts to orchestrate the workflow and Fortran 77/90 binaries.

Language	Files	Blank	Comment	Code
Fortran 90	23	394	2806	7581
Fortran 77	8	182	3568	6518
Bash	16	185	134	776

TABLE 4.6: NMMB-MONARCH: Original number of lines of code.

In contrast, Table 4.7 shows the result of porting the NMMB-MONARCH application into our prototype defining the workflow either in Java or Python. On the one hand, the Fortran binaries are kept the same because the binary and mpi decorators maintain its compatibility in both Java and Python languages.

Language	Files	Blank	Comment	Code
Fortran 90	23	394	2806	7581
Fortran 77	8	182	3568	6518
Java	18	560	890	2721
Python	18	515	710	2399

TABLE 4.7: NMMB-MONARCH: Final number of lines of code.

On the other hand, the lines of code of the main workflow significantly increase in both languages, raising from 776 up to 2721 in Java and 2399 in Python. The main reason for the larger amount of code lines is that the new implementations have better configuration management and object-oriented structure, which adds lines of code but improves the debugging, maintenance and extension of the code. Moreover, thanks to Java and Python's wide adoption, scientists can also use Integrated Development Environments (IDEs) and benefit from existing third-party libraries.

Regarding the task-level parallelism, Figure 4.5 shows the data dependency graph of the execution of three simulation days for the Python version. The DAG for the Java version is similar so the same conclusions can be applied. Readers can identify the *Fixed*, *Variable*, *UMO Model*, *Postprocess* and *Figures and animations creation* steps, where the *Variable*, *UMO Model* and *Postprocess* are executed thrice.

4.6.3.2 Computing infrastructure

The infrastructure used in this comparison is the Nord 3 cluster (a subset of the MareNostrum 3 supercomputer [148]), located at the Barcelona Supercomputing Center (BSC). This supercomputer is composed of 84 nodes, each with two Intel SandyBridge-EP E5-2670 (8 cores at 2.6 GHz with 20 MB cache each), a main memory of 128 GB, FDR-10 Infiniband and Gigabit Ethernet network interconnections, and 1.9 PB of disk storage. It provides a service for researchers from a wide range of different areas, such as life sciences, earth sciences, and engineering.

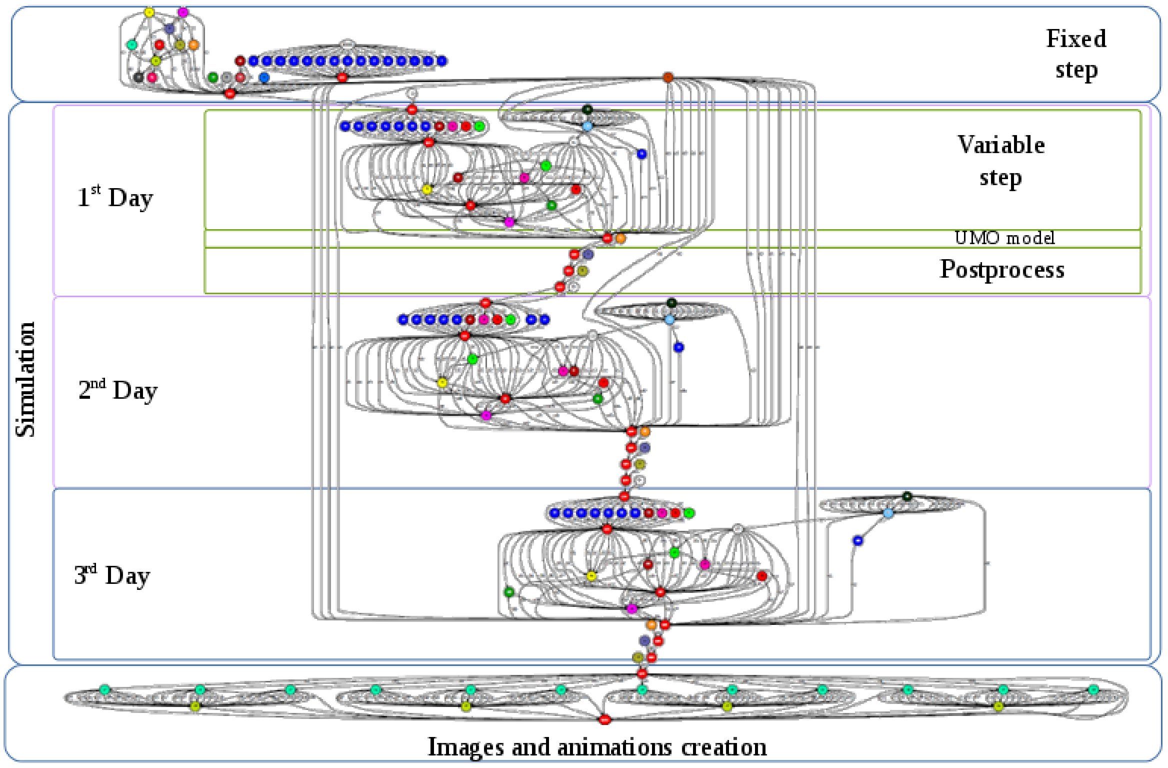


FIGURE 4.5: NMMB-MONARCH: Dependency graph of three days simulation.

The Java version used is 1.8.0_u112 64 bits. Regarding Python, the version used is 2.7.13.

Since the NEMS binary uses MPI, and there is no other task that can be performed in parallel with it, the task responsible of it will have a constraint defining that all workers will be dedicated to it. Consequently, the NEMS execution can benefit from all available resources during its operation. The MPI used is OpenMPI 1.8.1. All the other tasks are sequential and use a single core for their computation and have no further resource constraints.

4.6.3.3 Simulation dataset

In this experiment, we have performed three simulations, each one running for 24 hours and producing a 3-hourly output, which represents eight partial results per day. This is a reduced version of a production experiment as a proof-of-concept. Production simulations in NMMB-MONARCH require many years of simulation to cover a significant period and produce relevant results.

Since there are not yet satisfactory three-dimensional dust concentration observations, the initial state of dust concentration in the model is defined by the 24-hour forecast from the previous-day model run. The model at the starting day is run using "cold start" conditions, i.e., the zero-concentration initial state.

For the domain configuration, we replicated a usual operational configuration for a global simulation at BSC ($1.4° \times 1°$ horizontal resolution and 40 vertical sigman-hybrid layers), producing a combination of meteorological and chemistry output variables. The meteorological initial conditions are generated from NCEP global analysis ($0.5° \times 0.5°$).

In addition, the results shown in this article consider the generation of visualisation results for 41 different variables (e.g., dust deposition, wet dust deposition, dust concentration, etc.).

4.6.3.4 Global performance

Figure 4.6 shows the performance from a global perspective for the original, Java, and Python implementations of the NMMB-MONARCH application. For both plots shown in the figure, the horizontal axis shows the number of workers (with 16 cores each) used for each execution, the orange colour is the original implementation of the workflow (BASH), the green colour is the Java version, and the blue colour is the Python version. Notice that the Java and Python versions of the workflow are internally parallelised by our prototype. Moreover, the top plot represents the mean execution time over 10 runs, and the bottom plot represents the speed-up of each version with respect to the original workflow running with a single worker.

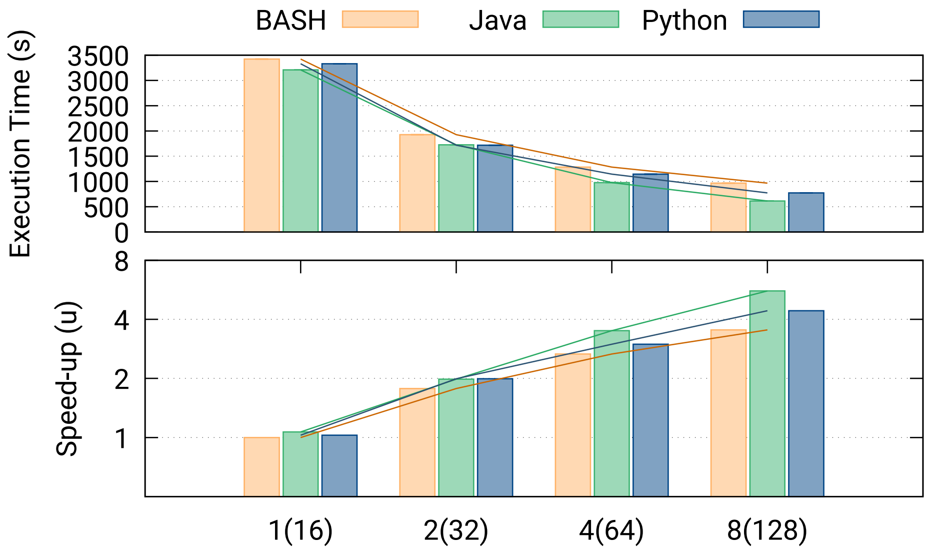


FIGURE 4.6: NMMB-MONARCH: Execution time and speed-up.

The results show that the original implementation of the workflow using BASH was already scaling pretty well, achieving a 3.5 speed-up when using 8 workers. However, both new implementations outscale the original one, achieving 5.59 speed-up (java) and 4.43 speed-up (python) when using 8 workers. This is due to the fact that our new implementations are able to parallelize the fixed, variable, and post-process steps while the original workflow cannot.

4.6.3.5 Performance per step

Considering the execution with 4 worker processes (64 cores), we can focus on the performance results for each step. Table 4.8 compares the previous version (BASH) against the Java and Python ones.

The first conclusion that can be obtained is that the parallelisation with both Java and Python improves the performance in the *Fixed* and *Variable* steps due to the possibility of performing multiple tasks at the same time during these steps. In opposition, the *Model Simulation* and *Post Process* do not improve because they are composed, respectively, by a single task, and two tasks with a sequential dependency. However, the performance of these steps does not degrade either.

Notice that the best speed-up is achieved on the *Fixed* step for both the Java (2.48) and Python (2.43) versions. However, since this step is only executed once per execution and the rest of the phases are executed iteratively per simulation day, the overall speed-up will vary

Step	**Execution Time (s)**			**Speed-up (u)**	
	BASH	**Java**	**Python**	**Java**	**Python**
Fixed	290	117	119	2.48	2.43
Variable	26	19	22	1.37	1.18
Model Simulation	244	242	233	1.01	1.04
Post process	38	34	33	1.12	1.15
Total	**601**	**413**	**415**	**1.45**	**1.45**

TABLE 4.8: NMMB-MONARCH: Performance per step with 4 workers (64 cores).

depending on the number of simulated days. Hence, considering the speed-up of each step, Equations 4.1 and 4.2 compute the expected speed-up in terms of the number of simulated days (N) for Java and Python respectively.

$$\text{Java : Expected SpeedUp(N)} = \frac{117 \cdot 2.48 + 295 \cdot 1.04 \cdot N}{117 + 295 \cdot N} \tag{4.1}$$

$$\text{Python : Expected SpeedUp(N)} = \frac{119 \cdot 2.43 + 288 \cdot 1.06 \cdot N}{119 + 288 \cdot N} \tag{4.2}$$

For large simulations, the previous equations tend to 1. When N is large enough, the computational load of the application is basically the *Model Simulation* step because the *Fixed* step is only executed once and the rest are significantly smaller. Hence, the equations basically measure the speed-up of the *Model Simulation* when using the Java or Python implementations against the previous implementation. Since all the implementations are executing this step using MPI, there is no performance difference between them. We must highlight that this result does **not** mean that the application cannot scale when running large simulations. This result means that all the versions behave similarly for large simulations, that is, that all the versions will provide the same speed-up with respect to the sequential execution when scaling to larger simulations and number of cores.

4.6.3.6 Behaviour analysis

For a more in-depth analysis of the workflow, we have generated a Paraver [183] trace of a three-day simulation with 4 workers (64 cores) of the Python workflow. As previously demonstrated, the Java and Python implementations behave similarly and thus, the conclusions derived from this analysis can also be applied to the Java version.

Figure 4.7 shows three timelines where each row corresponds to a thread in the worker. The top view shows a task view, where different coloured segments represent tasks, and each colour identifies a different task type. The middle and bottom views are the internals of the MPI NEMS task, where the yellow lines represent communications between MPI ranks.

In the top view, on the left, it is possible to identify the first step (namely *Fixed*), which is performed only once when the application starts. The maximum parallelism in this step is 16, so the first worker is being used completely while the other workers stay idle.

In the same way, it is possible to identify the last step at the right (*Image and Animation Creation*). As previously explained, this step contains two types of tasks (creation of the images and creation of the animation) that have a dependency between them. Hence, the step executes first all the tasks to create the images in parallel and, next, the tasks to create the animation. Consequently, all four workers are being used.

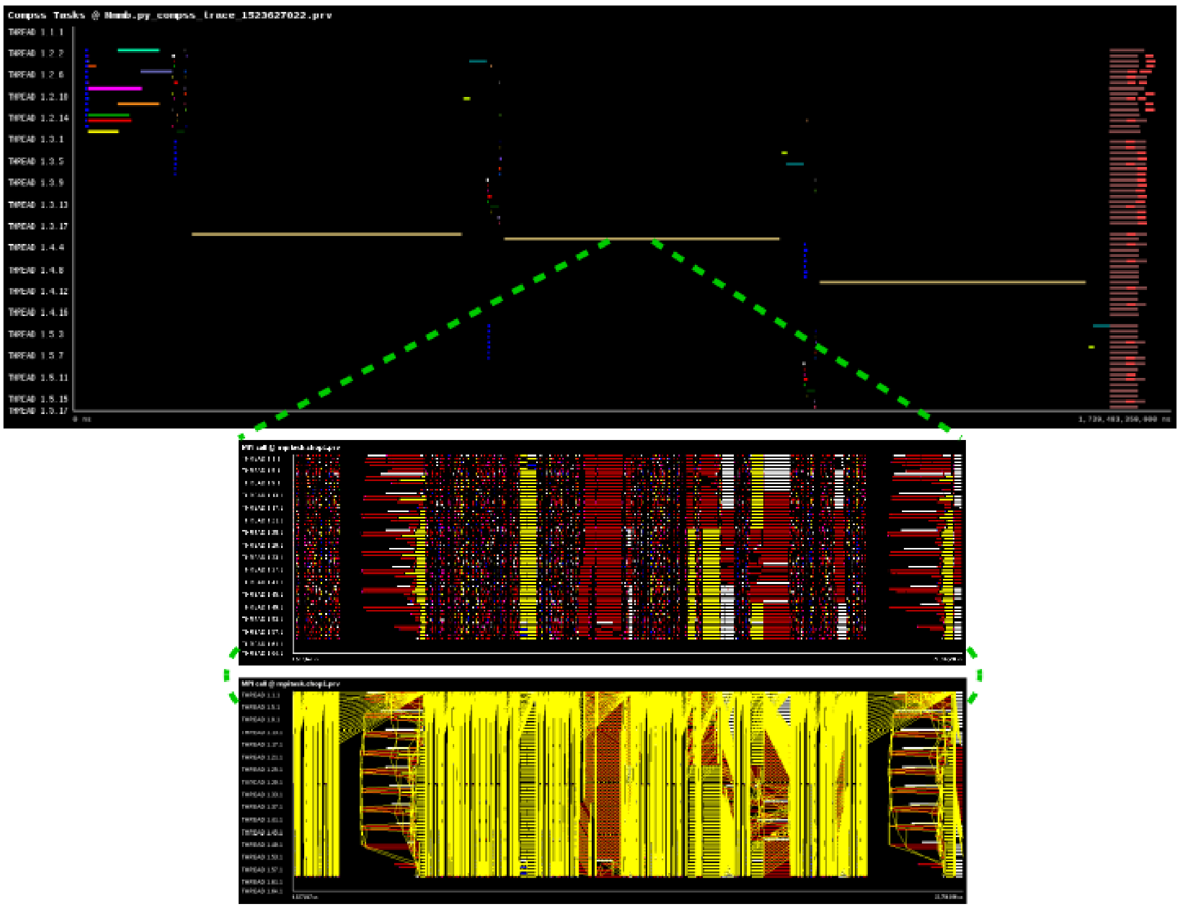


FIGURE 4.7: NMMB-MONARCH: Paraver trace of the Python version using 4 workers (64 cores).

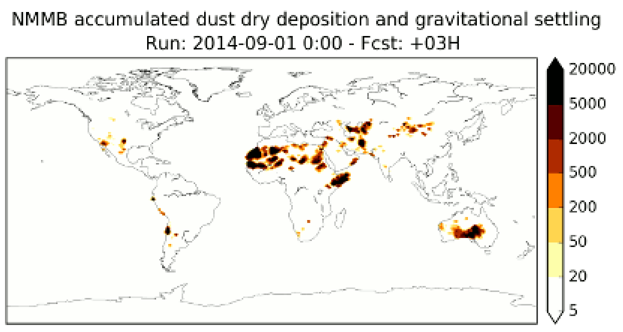
In between, the iterative process of the *Variable*, *Model Simulation* and *Post Process* steps happens three times (one per simulation day). The *Variable* and *Post Process* are not capable of filling completely the four workers because the dependencies between tasks limit the maximum parallelism of both steps. However, the step that requires most of the time is the *Model simulation* which, although it is performing a single task, it is exploiting all four nodes underneath thanks to MPI.

The trace detail in the middle and bottom views show, respectively, the MPI events and the MPI communications of the *Model simulation* task. Notice that, during this phase, both frameworks, PyCOMPSs and MPI, are working together and sharing the computing nodes. Although in this case MPI is using the four nodes, other applications may reserve some nodes for MPI and use the others to compute remaining sequential PyCOMPSs tasks.

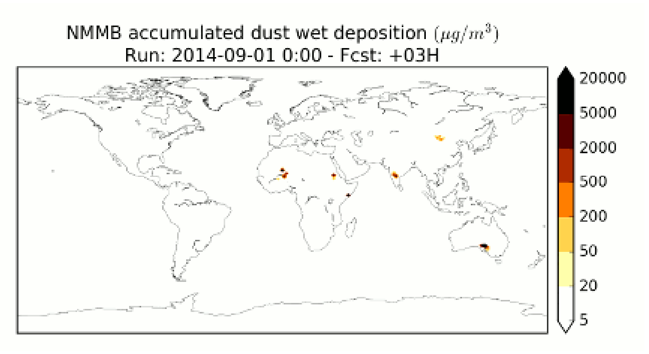
4.6.3.7 Scientific analysis

A scientific production run will consist of a simulation of a few years while in the results presented here we only simulated a three-days period. However, we have followed the usual production workflow to generate valid scientific results. Hence, our prototype has proven to be a useful tool to perform these complex simulations both in Java and Python.

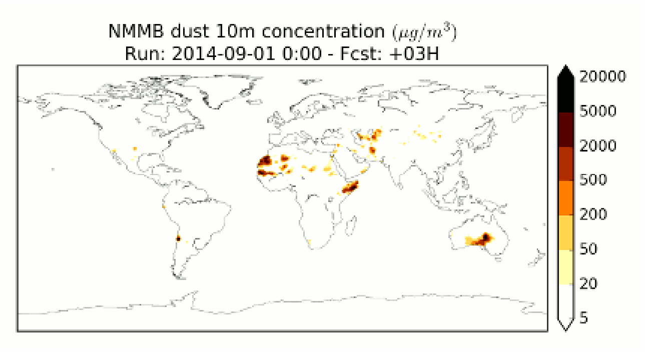
This same situation applies to the tasks that produce the animated plots. In a real application, this task would be replaced by an analysis task, gathering all the outputs and computing diagnostics or usual statistics operations (e.g., mean, average, deviation). However, we considered more illustrative to generate visual outputs as shown in Figure 4.8.


NMMB accumulated dust dry deposition and gravitational settling
Run: 2014-09-01 0:00 - Fcst: +03H
20000
5000
2000
500
200
50
20
5

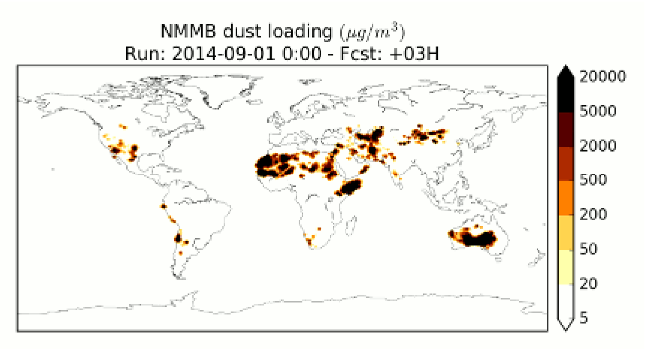
(a)


NMMB accumulated dust wet deposition ($\mu g/m^3$)
Run: 2014-09-01 0:00 - Fcst: +03H
20000
5000
2000
500
200
50
20
5

(b)


NMMB dust 10m concentration ($\mu g/m^3$)
Run: 2014-09-01 0:00 - Fcst: +03H
20000
5000
2000
500
200
50
20
5

(c)


NMMB dust loading ($\mu g/m^3$)
Run: 2014-09-01 0:00 - Fcst: +03H
20000
5000
2000
500
200
50
20
5

(d)

FIGURE 4.8: NMMB-MONARCH: Visual outputs. (a) Dust dry deposition, (b) Dust wet deposition, (c) Dust concentration at 10m, and (d) Dust load.

Chapter 5

Computational resources using container techonologies

Summary

Data Science workflows are particularly sensitive to load imbalance; varying the computational load significantly and quickly. This chapter focuses on solving the research question **Q2**. Our proposal contributes by supporting container technologies to adapt the computational resources much faster. Furthermore, container technologies increase our prototype's programmability since users can implement parallel distributed applications inside a controlled environment (container) and pack, deploy, and execute them in a one-click fashion.

Internally, our prototype distinguishes between (i) static, (ii) HPC, and (iii) dynamic resource management. We integrate container platforms at the three levels, providing representative use cases of all the scenarios using Docker, Singularity, and Mesos. The evaluation demonstrates that all the container technologies keep the application's execution scalability while significantly reducing the deployment overhead; thus enabling our prototype to adapt better the resources to the computational load. More in detail, Docker performs similarly than bare-metal and KVM with applications with small data dependencies and shows a significant overhead when using intensive multi-host networking. Mesos shows a bigger overhead than Docker but makes completely transparent the deployment phase, saving the user to deal with the intricacies of interacting directly with Docker. Finally, the execution with Singularity in queue-managed clusters shows an extremely low execution overhead.

5.1 General overview

Cloud Computing [27] has emerged as a paradigm where a large amount of capacity is offered on-demand and only paying for what is consumed. This paradigm relies on virtualisation technologies which offer isolated and portable computing environments called Virtual Machines (VMs). These VMs are managed by hypervisors, such as Xen [33], KVM [127] or VMWare [229], which are in charge of managing and coordinating the execution of the computations performed in the different VMs on top of the bare-metal.

Beyond the multiple advantages offered by these technologies and the overhead introduced during operation, the main drawback of these technologies in terms of usability is the management of the VM images. To build an image for an application, users have to deploy a VM with a base image that includes the operating system kernel and libraries, to install the application-specific software, and to create an image snapshot which will be used for further deployments. This process can take from several minutes to hours even for experienced developers and turns to be a complicated and tedious work for scientist or developers without a strong technological background.

To deal with these issues, a new trend in Cloud Computing research has recently appeared [186]. It proposes to substitute VMs managed by hypervisors with containers managed by container engines, also called *containerisers*, such as Docker [154, 72]. They provide a more efficient layer-based image mechanism, such as AUFS [4], that simplifies the image creation, reduces the disk usage, and accelerates the container deployment. The main difference between VM and containers relies on the fact that VM images include the whole OS and software stack, and the hypervisor has to be loaded every time a VM is deployed. As opposed, containers running on the same machine share the OS kernel and common layers, which reduces the amount of work to be done by container engines.

In any case, either container and VM images are very convenient for packaging applications because the user only needs to deploy a VM or container with the customised image. The complexity mainly falls back onto the application developer when creating the customised image. In the case of distributed applications, the process is more complicated since the user has to deploy multiple containers and configure them to coordinate the execution of the application properly. Therefore, to facilitate this process, there is a need for an extra component that must coordinate the deployment, configuration, and execution of the application in the different computing nodes. We propose that the programming model runtime can manage these tasks.

This chapter focuses on solving the research question **Q2**; proposing a methodology and an implementation to integrate container technologies to ease the installation of software dependencies and better adapt the computing resources to the workload of the application. At its starting point, COMPSs was already capable of dynamically adapting the computational resources to the remaining workload of the application by creating and destroying VMs. However, since data science workflows vary quicker than traditional HPC ones, we have extended our prototype with container technologies to benefit from its light-weight deployment to adapt the computational resources rapidly. Furthermore, our design smooths the integration of container engines with parallel programming models and runtimes by proposing a methodology to facilitate the development, execution, and portability of parallel distributed applications.

The combination of container engines with our prototype brings several benefits for developers. On the one hand, the programming model provides a straightforward methodology to parallelise applications from sequential codes and decoupling the application from the underlying computing infrastructure. On the other hand, containers provide efficient image management and application deployment tools which facilitate the packaging and distribution of applications. Hence, the integration of both frameworks enables developers to easily port, distribute, and scale their applications to parallel distributed computing platforms.

5.2 Related Work

Some previous work has been performed to facilitate the portability of applications to different distributed platforms. For what relates to clouds, software stacks as OpenNebula [223, 177] or OpenStack [207, 178] provide basic services for image management and contextualisation of VMs. Contextualisation services typically include the networking and security configuration. Concerning image management, these platforms expose APIs that provide methods to import customised images as well as to create snapshots once the user has manually modified the base image. However, these manual image modifications can be tedious work for complex applications.

Some recent work has focused on automating this process by adding new tools or services on top of the basic services offered by providers. CloudInit [52] is one of the most used tools to automate the VM image creation. It consists of a package installed in the base image which can be configured with a set of scripts that will be executed during the VM boot time. Another extended way to configure and customise VM images is based on DevOps tools development like Puppet [191] or Chef [49] where a *Puppet manifest* or a *Chef receipt* is deployed, instead of executing a set of configuration scripts. Some examples of these solutions can be found in [28], [41] or [125]. However, these solutions have a drawback: customising the image at deployment time (installing a set of packages downloading files, etc.) can take some minutes. It can be assumable in the first deployment but not for adaptation where new VMs must be deployed in seconds. To solve this issue, some services like [79] have been proposed to perform offline image modifications, in order to reduce the installation and configurations performed at deployment time.

In the case of containers, most of the container platforms already include similar features to customise container images easily. In the case of Docker [154, 72], we can write a *Dockerfile* to describe an application container image. In this file, we have to indicate a parent image and the necessary customisation commands to install and run the application. Due to the layered-based image system, parent and customised images can be reused and extended by applications, achieving better deployment times. This is one of the main reasons why several users are porting their application frameworks to support Docker containers. Cloud Providers have started to provide services for deploying containers such as the Google Container Engine [131] and cloud management stacks have implemented drivers to support Docker containers as another type of VM, such as the Nova-Docker-driver [171] for OpenStack or OneDoc [175], a Docker driver for Open Nebula. Apart from Docker, different container platforms have appeared recently. Sections 5.3.2 and 5.4 provide more details about the different available container engines.

Also, some work has been produced to integrate application frameworks, such as workflow management systems, with container engines. Skyport [92] is an extension to an existing framework to support containers for the execution of scientific workflows. This framework differs from our proposal in the definition of the application which requires to explicitly write the workflows' tasks in the form of JSON documents where the input/output of each block has to be specified along with the executable. In COMPSs applications, the programming model is pure sequential with annotations that identify the parts of the code (tasks) to be executed by the runtime. Skyport uses Docker but delegates to the users the responsibility to create the application images and to establish scalability procedures using the available infrastructure tools, while in our work we propose a way to manage the image creation and resource scalability transparently. Moreover, we also provide extensions for Docker, Singularity, and Mesos clusters. In [238], authors describe the integration of another workflow system, Makeflow, with Docker. As in Skyport, the main difference with our proposal is in the programming model, which in Makeflow is represented by chained calls to executables in the form of Makefiles and is tailored to bioinformatics applications; it does not provide any tool to build container images from the workflow code, and the supported elasticity is done per task. For each task in the workflow, Makeflow creates a container; thus not being capable to reuse the containers for different tasks. Finally, Nextflow [70] is another option which proposes a DSL language that extends the Unix pipes model for the composition of workflows. Both Docker and Singularity engines are supported in Nextflow, but it has similar limitations than other frameworks, such as the manual image creation and a limited elasticity supported only for Amazon EC2.

Framework	Supported Container Engines	Image Creation	Elasticity
Skyport	Docker	Manual	Manual (provider API)
Makeflow	Docker, Singularity, Umbrella [153]	Manual	Limited ⋆
Nextflow	Docker, Singularity	Manual	Limited ⋆⋆
Our prototype	Docker, Singularity, Mesos	Automatic	Full support ⋆ ⋆ ⋆

⋆ Always container per task.
⋆⋆ Only in Amazon EC2, no transparent scale-down.
⋆ ⋆ ⋆ Transparent resource scale-up/down in containers and VM platforms.

TABLE 5.1: Comparison of container support in state of the art frameworks

Table 5.1 summarises the comparison between our proposal with previously proposed container integration. We propose to integrate container engines with parallel programming model runtimes, such as COMPSs, to provide a framework where users can easily migrate from a sequential application to a parallel distributed application. The proposed extensions to COMPSs provide the features to create the application container images automatically and to deploy and execute the application in container-based distributed platforms transparently.

Our prototype distinguishes between static, HPC, and dynamic resource management. We integrate container platforms at the three scenarios, providing representative use cases using Docker, Singularity, and Mesos. Notice that generalising the support to different container platforms is important because, although they provide similar behaviours, each solution has different features and needs, and targets different types of organisations. Also, we believe that dynamic container management is a must to take full profit of the container platforms and thus, container platforms must be integrated in such a way that frameworks can dynamically create and destroy containers during the application's execution.

5.3 Resource Orchestration Platforms

Resource Orchestration Platforms (ROP) are capable of managing clusters, clouds, virtual machines or containers. As shown in Figure 5.1, there is a clear division between software developed to administrate and manage batch ROP (such as supercomputers), and software developed for interactive ROP (such as clouds or containers). Next subsections provide further information about both approaches by defining, comparing, and categorising the latest software.

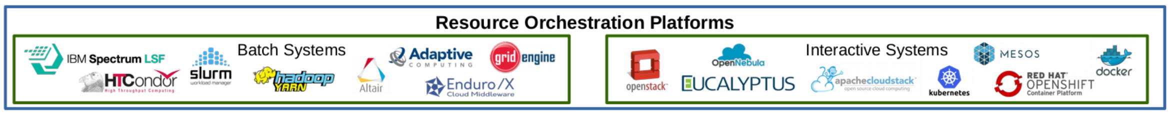


FIGURE 5.1: Classification of Resource Orchestration Platforms (ROP)

5.3.1 Batch ROP

Batch ROP are in charge of scheduling jobs that can run without user interaction, require a fixed amount of resources, and have a hard timeout among the resources they manage. Since these systems are typically for HPC facilities (i.e., supercomputers), they are designed to administrate a fixed number of computational resources with homogeneous capabilities. To support modern HPC facilities with heterogeneous resources, batch ROP can include constraints or queue configurations. Also, in an attempt to make the HPC infrastructure more flexible, some alternatives include job elasticity to vary the number of resources assigned to a job at execution time.

5.3.1.1 Software discussion and examples

Although every system has its own set of user commands, they all provide more or less the same functionalities to the end-user; the only exception being some advanced features such as environment copy, project allocations, or generic resources. However, system administrators will find significant differences between systems. Thus, in general terms, choosing long-term well-supported software and finding the appropriate system administrator are the key points when looking for a batch system.

For instance, SLURM [234, 211] is a free and open-source job scheduler used on about the 60% of the TOP500 supercomputers. IBM Spectrum LSF [114], Torque [224], and PBS [106, 185] are also widely used options with constant updates and support. There is also more specialised alternatives, like HT Condor [217, 112] or Hadoop Yarn [226, 18], with less use but with active user communities, that are often chosen because of their specialised features (e.g., scavenging resources from unused nodes or managing Hadoop clusters).

5.3.1.2 Taxonomy

Table 5.2 presents our taxonomy for the batch ROP. First, we describe their architecture, detailing whether the implementation language is Java (J in the table), Python (P), C++, or C. We also differentiate between the supported underlying operating systems - Windows (W), Linux (L), or Unix-Like (U) - and file systems - NFS (N), Posix (P), HDFS (H), or Any (A) -.

Next, we point out configurable features that system administrators might require; such as support for heterogeneous resources, job priority, and group priority. Also, we provide information about the system limits; stating the maximum number of nodes and jobs that each system has proven to work with.

Regarding fault tolerance, we indicate the support for job checkpointing. Also, regarding security, we categorise the user authentication between operating system (OS) and many (M), and the stored data encryption between many (M), Yes ($\star$) or None (-).

Finally, we have included the maintainer since batch ROP are critical for the software stack. Thus, reliability and support from the maintainer might be critical when choosing between the different options. Moreover, the framework's availability is also a high-priority issue. For this purpose, we consider the license of each framework following the next nomenclature: (1 in the table) Apache 2.0 [21], (2) GNU GPL2 [96], (3) GNU GPL3 [97], (4) GNU LGPL2 [98], (5) BSD [42], (6) MIT License [156], (7) other public open-source software licenses (e.g., Academic Free License v3 [3], Mozilla Public License 2 [158], Eclipse Public License v1.0 [78]), and (8) custom private license or patent.

Software		Features												
		Architecture			Configuration			Limits		F.T.	Security			
		Language	OS	File System	Heter. Resources	Job priority	Group priority	Max. nodes	Max. jobs	Job Checkpointing	Authentication	Encryption	Maintainer	License
Batch ROP	Enduro/X [80]	C,C++	U	P	⋆	-	-			⋆	OS	M	Maximax Ltd	2
	GridEngine [91]	C	W,U	A	⋆	⋆	⋆	10k	300k	⋆	M	⋆	Univa	8
	Hadoop Yarn [226, 18]	J	W,U	H	⋆	⋆	-	10k	500k		M	⋆	Apache SF	1
	HT Condor [217, 112]	C++	W,U	N	⋆	⋆	⋆	10k	100k	⋆	M	M	UW-Madison	1
	IBM Spectrum LSF [114]	C,C++	W,U	A	⋆	⋆	⋆	9k	4M	⋆	M	⋆	IBM	8
	OpenLava [123]	C,C++	L	N	⋆	⋆	⋆			⋆	OS	-	Teraproc	2
	PBS Pro [106, 185]	C,P	W,L	P	⋆	⋆	⋆	50k	1M	⋆	OS	-	Altair	2
	SLURM [234, 211]	C	U	A	⋆	⋆	⋆	120k	100k	⋆	M	-	SchedMD	2
	Torque [224]	C	U	A	⋆	⋆	⋆			⋆	OS	-	Adaptive Computing	8

Legend: ⋆ Available - Not available / Not Applicable

TABLE 5.2: Classification of *Batch ROP*.

5.3.1.3 Analysis

First, all the alternatives are available for Unix-Like systems; GirdEngine, Hadoop Yarn, HT Condor, IBM Spectrum LSF, and PBS Pro can also work with Windows nodes. As expected, notice that the only system working with HDFS is Hadoop Yarn; while the rest work with NFS or POSIX file systems.

Second, regarding the configuration options, all the systems support heterogeneous resources, job priority (except Enduro/X), and group priority (except Enduro/X and Hadoop Yarn). Also, job checkpointing is available in all the alternatives except for Hadoop Yarn.

Third, regarding the tested limits, SLURM is the only system that has been proven to work with more than 100k nodes. PBS Pro has been tested with 50k nodes, and GridEngine, Hadoop Yarn, HT Condor and IBM Spectrum LSF are far behind with around 10k nodes. On the other hand, IBM Spectrum LSF and PBS Pro are the only systems that have been proven to handle more than 1 million jobs. Next, Hadoop Yarn has been tested with 500k jobs, Grid Engine with 300k jobs, and HT Condor and SLURM with 100k jobs. Unluckily, we have not been able to find reliable information regarding the limit of nodes nor the limit of jobs for Enduro/X, OpenLava, or Torque.

Fourth, all the systems support user authentication through different methods. However, only Enduro/X, GridEngine, Hadoop Yarn, HT Condor, and IBM Spectrum LSF provide data encryption. This fact might not be an issue since batch ROP are usually installed in secure clusters.

Finally, all the alternatives except GridEngine and Torque are available through different public open licenses. However, the maintainers offer paid plans for installation and support that are highly recommended for large clusters.

5.3.2 Interactive ROP

In contrast to batch ROP, interactive ROP are designed for on-demand availability of computing and storage resources; freeing the user from directly managing them. Since these systems are designed for clouds and containers, they are required to (1) integrate and free resources from the system, and (2) dynamically adapt the resources assigned to a running job to fulfil its requirements. Typically, these systems handle heterogeneous resources in one or many data centres from one or many organisations.

5.3.2.1 Software discussion and examples

OpenStack [207, 178] and OpenNebula [223, 177] are the reference software for managing cloud computing infrastructures based on virtual machines. Both solutions are deployed as Infrastructure as a Service (IaaS), supporting multiple, heterogeneous, and distributed data centres and offering private, public, and hybrid clouds. Also, both are free and open-source.

On the other hand, containerisation has become increasingly popular thanks to its light-weight deployment. Docker [154] has become the most popular container technology by offering Platform as a Service (PaaS) products that rely on the OS-level virtualisation to deliver software in packages (also known as containers). Each container is isolated from the rest and contains its own software, libraries, and configuration files. Docker Swarm [165, 214] is the native mode to manage clusters of Docker Engines. Similarly, Kubernetes [107, 134] is an open-source container-orchestration system to provide automatic deployment and scaling of applications running with containers in a cluster.

Previous work has been published around interactive ROP that already analyses many of the features of our taxonomy and discusses some of the alternatives. For instance, [231] provides an in-depth comparison about OpenStack and OpenNebula. Also, the principal investigator and the chief architect discuss the OpenNebula project in [155]. Furthermore, there are online comparisons about Kubernetes and Docker Swarm [132], or Kubernetes and Mesos [133] that were useful when retrieving information for our taxonomy.

5.3.2.2 Taxonomy

Table 5.3 presents our taxonomy of the surveyed interactive ROP. The first two columns classify whether the different technologies work with virtual-machines or containers. Next column defines the supported virtualisation formats; distinguishing between raw (R in the table), compressed (C), Docker (D) or Appc (A). Also, we indicate the number of available hypervisors (e.g., KVM, Xen, Qemu, vSphere, Hyper-V, bare-metal) and the implementation language, differentiating between Java (J), Go (G), Python (P), C (C), and C++ (C++).

Regarding the user interaction, we define the different interfaces for each system: web (W), client (C), REST (R), EC2 (E), and HTTP (H). Moreover, we also include the language of the available libraries distinguishing between Java (J), Python (P), Ruby (R), Go (G), and C++.

Regarding the elasticity, we indicate whether the systems support automatic scaling and bursting. Also, we indicate the maximum number of nodes or cores that the system has proven to manage. Furthermore, we indicate whether the systems provide accounting and user quotas since these might be interesting features for system administrators. Similarly, we indicate whether the systems support load balancing, object storage, live migration, rolling updates, self-healing techniques, and rollbacks. Finally, we detail the license considering the same options than in the Batch ROP classification(see Section 5.3.1.2 for further details).

5.3.2.3 Analysis

First, only OpenStack can handle virtual machines and containers simultaneously. Also, Eucalyptus is the only system using virtual machines that has not support for compressed

Software		Features																				
		Virtualisation				Code	Interaction		Elasticity			Mgmt.		Application								
		VM	Container	Format	Hypervisor	Language	Interface	Libraries	Automatic Scaling	Bursting	Scalability	Accounting	User quotas	Load Balancing	Object Storage	Live Migration	Rolling Update	Self-healing	Rollbacks	License
Interactive systems	CloudStack [136, 14]	⋆	-	R,C	7	J	W,R,E	-	⋆	-		⋆	⋆	⋆	-	⋆	-	-	-	1
	Docker Swarm [165, 214]	-	⋆	D	-	G	C,H	G,P	-	-	1kn	-	-	⋆	-	-	⋆	⋆	-	1
	Eucalyptus [174, 81]	⋆	-	R	1	J,C	W,C,E	-	⋆	-		-	⋆	⋆	⋆	⋆	-	-	-	3
	Kubernetes [107, 134]	-	⋆	D	-	G	W,C,R	G,P	⋆	-	5kn	-	⋆	⋆	-	⋆	⋆	⋆	⋆-	1
	Mesos [109, 23]	-	⋆	D,A	-	C++	W,H	J,C++	⋆	-	50kn	-	⋆	-	-	-	⋆	⋆	-	1
	OpenNebula [223, 177]	⋆	-	R,C	3	C++	W,C,R	R,J	⋆	⋆		⋆	⋆	-	-	⋆	-	-	-	1
	OpenStack [207, 178]	⋆	⋆	R,C	6	P	W,C,R	P	⋆	-	120kc	⋆	⋆	⋆	⋆	⋆	-	-	-	1
	RedHat OpenShift [199]	-	⋆	D	-	G	R,C,W	-	⋆	⋆	1kn	-	⋆	⋆	-	⋆	⋆	⋆	⋆	1

Legend: ⋆ Available - Not available / Not Applicable

TABLE 5.3: Classification of *Interactive ROP*.

formats. The rest (i.e., Apache CloudStack, OpenNebula, and OpenStack) can handle raw and compressed formats through different hypervisors.

Second, regardless of the virtualisation, the systems offer many interfaces; the most common ones being client, REST, or web interfaces. Also, regarding elasticity, all the alternatives except Docker Swarm provide automatic scaling. However, only OpenNebula and RedHat OpenShift provide cloud bursting.

Regarding user management, all the surveyed systems (except Docker Swarm) provide user quotas, but only Apache CloudStack, OpenNebula, and OpenStack provide user accounting. Also, regarding application features, most of the systems provide load balancing and live migration. On the other hand, rolling updates, self-healing, and rollbacks are more common in container platforms than virtual machine platforms. Also, Kubernetes and RedHat OpenShift are the richest options; providing all the application management features except Object storage.

Finally, we must highlight that all the alternatives are available through different public open licenses. However, for large clusters and continuous support, it is recommended to check the paid plans offered by the maintainers.

5.4 Description of reference ROP

5.4.1 OpenStack

OpenStack [207, 178] was born in July 2010 as a collaboration between Rackspace and NASA to manage massive amounts of computing, storage, and networking resources. Nowadays, OpenStack is a widely-used open-source software for deploying and managing private and public clouds. It provides a dashboard and an API that give administrators the control of large pools of computing, storage and networking resources while empowering the users to provision resources through web and command-line interfaces.

Furthermore, OpenStack is built as a combination of different modules; providing a flexible adaptation to different administration requirements. Three modules define the core software project: (i) Nova, the compute infrastructure, (ii) Swift, the storage infrastructure, and (iii) Glance, the image service. On the one hand, Nova hosts the cloud computing system and manages all the activities required to support the life cycle of the instances. On the other hand, Swift offers a large and consistent object store distributed across nodes. Finally, Glance is a lookup and retrieval system to monitor virtual machine images.

5.4.2 OpenNebula

OpenNebula [223, 177] was born as a research project in 2005 and first released in March 2008 as an open-source project. Its primary focus is private clouds, providing a simple solution to orchestrate and configure virtualised data centres or clusters. However, it also supports Hybrid and Public clouds by providing cloud interfaces to expose its functionalities for virtual machines, storages, and network management. Moreover, OpenNebula provides users with an elastic platform for fast delivery and scalability of services. These services are hosted as Virtual Machines (VMs) and submitted, managed, and monitored in the cloud using virtual interfaces.

In contrast to OpenStack, OpenNebula has a classical cluster-like architecture with a front end and a set of cluster nodes running the VMs; requiring at least one physical network to connect all the cluster nodes with the front end.

5.4.3 Docker framework

Docker [154] is an open platform for developing, shipping, and running applications. It provides a way to run applications securely isolated in a container. The difference between Docker and usual VMs is that Docker does not need the extra load of a hypervisor to run the containers, and it uses an efficient read-only layered image system achieving lighter deployments. To improve the Docker experience, several services and tools have been created. For our case study, the relevant ones are Docker-Swarm, Docker-Compose, and DockerHub.

First, Docker Swarm [165, 214] is a cluster management tool for Docker. It merges a pool of Docker hosts enabling the deployment of containers in the different hosts with the same Docker API and giving to the user the impression that it has a single, virtual Docker host. Also, Docker Swarm is in charge of transparently managing the inter-host networking and storage. Moreover, it allows defining scheduling policies to manage where the containers must be placed in the cluster.

Next, Docker Compose is a tool to define complex applications easily which require deploying multiple Docker containers. It provides a simple schema to allow users to define the different containers required by their application. Once the user has defined the application, Docker Compose is in charge of automatically deploying and configuring the different containers.

Finally, DockerHub is a public cloud-based image registry service which enables the users to store and share their applications' docker images. The Docker framework also offers the Docker Registry which is an open-source service with the same API as DockerHub that can be installed on the provider premises in order to store and share users' images in a local, private and controlled way. This Docker Registry can also be used as a cache for DockerHub to minimise the effect of performance degradations and downtimes of the DockerHub service.

5.4.4 Kubernetes

Kubernetes [107, 134] is a portable, extensible, open-source orchestration software for managing containerised workloads and services that was open-sourced by Google in 2014. It provides a framework to run distributed systems resiliently; allowing to orchestrate a cluster of virtual machines and schedule containers to run on those virtual machines based on their available compute resources and the resource requirements of each container. Among its large ecosystem, we highlight that Kubernetes provides service discovery, load balancing, failover, dynamic adaptation (i.e., creates, removes or adapts containers to achieve a desired state in terms of CPU and memory resources), constrained scheduling (i.e. schedules containers across the cluster nodes considering CPU and memory boundaries), storage orchestration, and self-healing (i.e. restarts or replaces failed containers).

Kubernetes has been widely adopted by both the community and the industry; becoming the most popular Container Orchestration Engine (COE). However, there are many alternatives for the coordination of containers across clusters of nodes. For instance, Docker Swarm is tightly integrated into the Docker ecosystem; using the Docker CLI to manage all container services. In contrast to Kubernetes, Docker Swarm is easy to set up for any operating system and requires little effort to learn since it only needs a few commands to get started. However, Docker Swarm is bounded to the limitations of the Docker API. Another option is the Marathon framework on Apache Mesos that provides an unmatchable fault-tolerance and scalability. However, its complexity to set up and manage clusters makes it impractical for many teams.

5.4.5 Singularity

Singularity [138, 210] in a container engine that focuses on providing users with a customisable, portable, and reproducible environment to execute their applications. As other container engines, Singularity is based on images where the users have full control to install the required software stack to run their applications (OS, library versions, etc.). Although Singularity defines its own container description and format, for compatibility purposes, it is capable of importing Docker images.

Concerning the version used in this study (Singularity 2.4.2), there are two main differences in comparison with other container engines such as Docker or Kubernetes. First, Singularity is capable of running containers in non-privileged user space and accessing special host resources such as GPUs, and high-speed networks like Infiniband. Second, Singularity does not provide shared virtual networking and multi-container orchestration since it focuses on HPC environments. Although this lack of isolation and orchestration avoids network virtualisation overheads, it forces services running in containers hosted on the same node to share the network interface, hostname, IP, etc. Hence, Singularity is often combined with queue managers (such as SLURM or LSF) to provide these features at host level. Also, we highlight that more recent versions of Singularity (3.0 and later) include network virtualisation and full integration with Container Network Virtualisation (CNI) [57].

5.4.6 Mesos

Mesos [109, 23] is a resource manager designed to provide efficient resource isolation and sharing across distributed applications. It consists of a master daemon that manages agent daemons (slaves) running on each cluster node, and frameworks that run tasks on these agents. The slaves register with the master and offer resources, i.e., capacity to run tasks. Mesos uses the concept of frameworks to encapsulate processing engines whose creation is triggered by the schedulers registered in the system. Frameworks reserve resources for the execution of tasks while the master can reallocate resources to frameworks dynamically.

Mesos supports two types of containerisers, the Mesos native containeriser, and the Docker containeriser. Mesos containeriser uses native OS features directly to provide isolation between containers, while Docker containeriser delegates container management to the Docker engine. Mesos native containeriser provides interoperability with Docker images, thus making possible to reuse the same application image transparently with regards to the specific Mesos deployment.

5.5 Architecture

COMPSs is capable of abstracting the applications from the underlying infrastructure by deploying a worker process in each computational resource. This methodology guarantees that the communication between the master and the worker processes during the application's execution is the same regardless of the underlying infrastructure. However, the configuration and remote deployment of the worker process varies depending on the target computational resource.

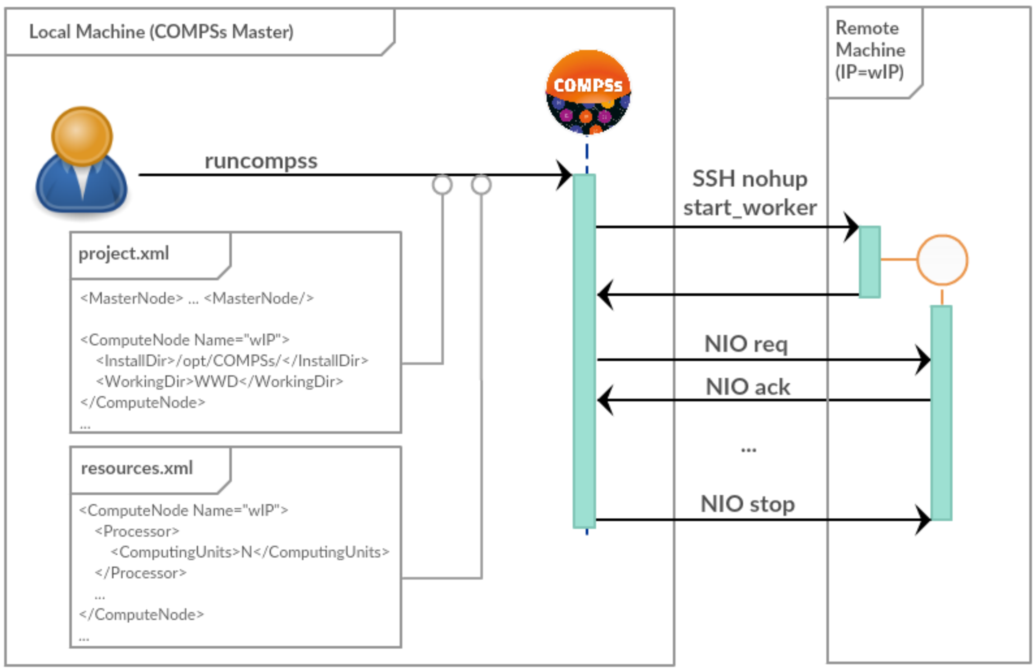
Internally, COMPSs differentiates three scenarios to configure and deploy the worker processes: (i) static, (ii) HPC, and (iii) dynamic computational resources. Next subsections describe each scenario, detail the adaptations performed in our prototype to handle containers, and provide use case examples using some of the available container platforms.

5.5.1 Static computational resources

We consider static computational resources those computing resources that are available during the whole execution of the application without further requests (e.g., grids). This scenario is also useful for debugging when configuring the developer's laptop as a computing resource.

5.5.1.1 COMPSs static resources management

COMPSs applications are executed running the `runcompss` command. As shown in Figure 5.2, this command starts the COMPSs master process that, among many other things, loads the remote machines' information from the *project.xml* and *resources.xml* files to deploy and configure the worker processes. The worker processes are deployed using SSH at the beginning of the application's execution. However, the SSH connection is terminated once the worker process has started so that the master and the worker can communicate through



```
<MasterNode> ... <MasterNode/>

<ComputeNode Name="wIP">
   <InstallDir>/opt/COMPSs/</InstallDir>
   <WorkingDir>WWD</WorkingDir>
</ComputeNode>
...
```

```
<ComputeNode Name="wIP">
   <Processor>
      <ComputingUnits>N</ComputingUnits>
   </Processor>
   ...
</ComputeNode>
...
```

FIGURE 5.2: COMPSs static resources management

Java NIO during the application's execution. Finally, at the end of the application's execution, a stop message is sent from the master to the worker to stop the worker process and clean the remote machine.

5.5.1.2 New static container management

Similarly to regular applications, our prototype provides a `runcompss_container` submission command to encapsulate all the required steps to run a containerised application. As shown in Figure 5.3, the first step is the creation of the container image containing the application, its dependencies, and our prototype. The second step is the execution of the application using the container engines. We must highlight that the first step is done only once per application, while the second step runs every time an application is executed. Also, although the operations in each step may vary when using different container engines, the `runcompss_container` abstracts the final user from the underlying container engine.

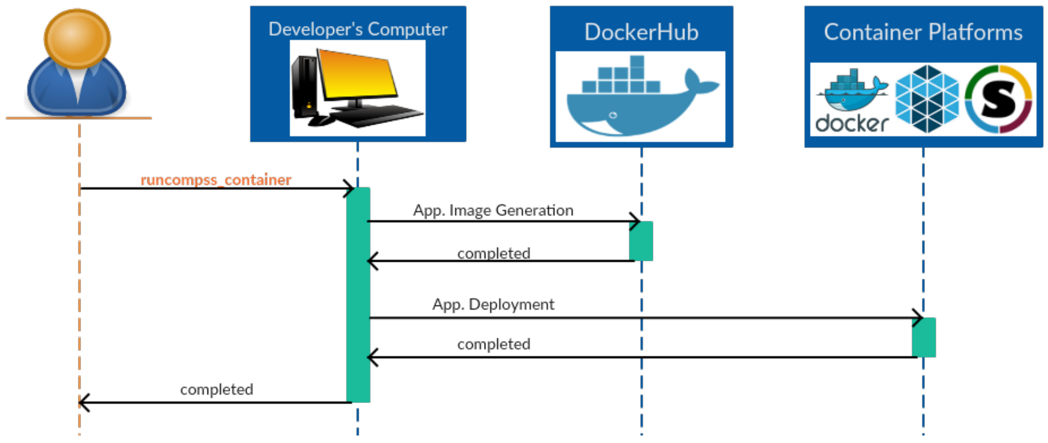


FIGURE 5.3: Integration with containers platforms.

5.5.1.2.1 Submission command

From the users' point of view, the only difference when running containerised applications is the submission command. Listing 5.1 compares the submission command for regular and containerised applications.

```
# Normal execution
runcompss
   --classpath=/home/john/matmul/matmul.jar
   matmul.objects.Matmul 16 4

# Docker execution
runcompss_container
   --engine=docker
   --engine-manager='129.114.108.8:4000'
   --initial-worker-containers=5
   --container_image='john123/matmul-example'
   --classpath=/home/john/matmul/matmul.jar
   matmul.objects.Matmul 16 4
```

LISTING 5.1: Comparison between normal and container execution.

On the one hand, regular applications are executed by invoking the `runcompss` command followed by the application main class and arguments. This command transparently loads the Runtime and starts the application execution.

On the other hand, containerised applications are executed by invoking the `runcompss_container` command. This command supports the same options than the `runcompss` command but requires some extra arguments to specify the application container image, the container engine, the engine manager IP address and port, and how many containers must be initially deployed as computing resources. Notice that we assume that the container platform is available to deploy the application containers and that the developer's computer has installed the container platform client to execute the application as well as to create and share the application images across the cluster.

5.5.1.2.2 Container image creation

The majority of container engines are capable of importing, converting or directly running containers with Docker image format. However, every container uses its own API and deployment model to execute the containers. For this reason, the container image creation is a common process regardless of the container engine, while the application's deployment and execution depend on the container engine.

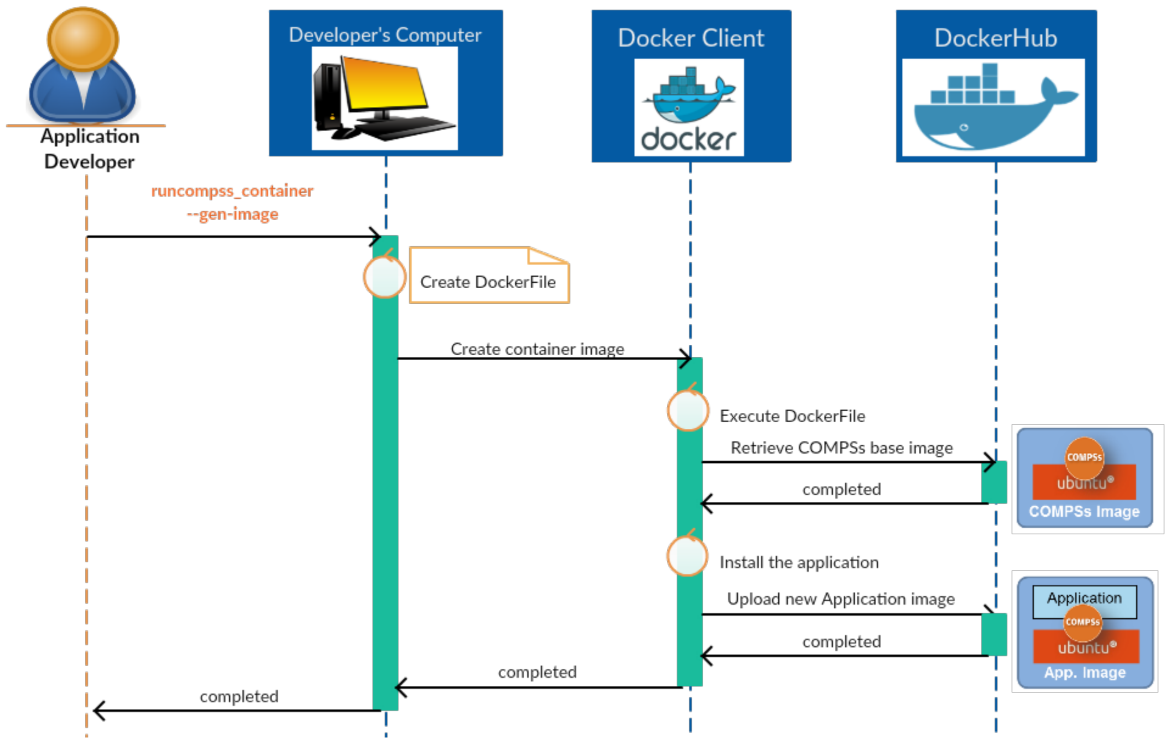


FIGURE 5.4: Image generation phase.

Figure 5.4 describes the overall workflow to generate a Docker image for an application. Notice that this is a generic process that any other framework could use to create an application container image transparently. As an overview, our prototype creates the application container image needed by the Docker containers and uploads it to DockerHub in order to make it available to any container platform. To do so, we have included a utility that creates a *DockerFile* describing how to create the application Docker image. More in detail, it describes how to install the application context directory (the directory where application files are located) and the required dependencies on top of our prototype's base image as a separate layer. This base image is a public Docker image located at DockerHub which already contains a ready to use Runtime and its required dependencies. The image creation is performed by executing the DockerFile with the Docker client which automatically pulls the base image, installs the application on the base image as a new layer, and uploads it to the DockerHub.

In this way, different applications deployed in Docker share the same base layer, and thus, the deployment of a new application only requires to download the new application layer. Moreover, the deployment of several instances of the same application or new worker nodes does not need any new installation. Hence, taking advantage of the Docker layer system, our prototype can increase the deployment speed and can perform better adaptations.

5.5.1.2.3 Container execution

Before starting the COMPSs master process, the `runcompss_container` starts the required containers and assigns them a valid IP address. This step is done by interacting with the ROP and is completely dependent on the underlying container engine. However, from this step on, containers are treated like any other resource. Hence, the `runcompss_container` can build the *project.xml* and *resources.xml* files accordingly and start the COMPSs master normally.

5.5.1.3 Use Case 1: Docker

Figure 5.5 depicts how our prototype orchestrates the deployment and execution of an application using the Docker container engine. In this phase, we define a Docker-Compose application by creating a *docker-compose.yml* file which describes a master container (where to execute the main application) and a set of worker containers (where to execute the tasks).

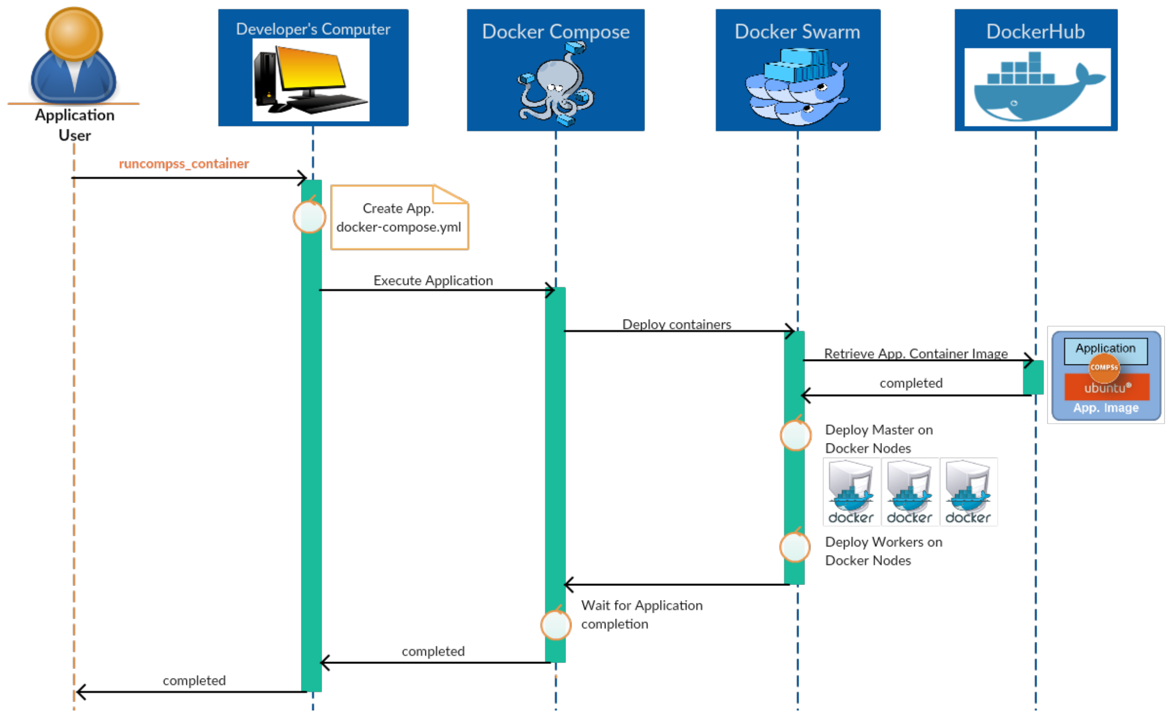


FIGURE 5.5: Deployment phase.

Even though the containers execute different parts of the application, both types of containers boot the same application image; the only difference being the command executed once the container is deployed. On the one hand, the master container executes the master process and the application. On the other hand, the worker containers start the worker daemon and wait for the master messages to execute tasks. Furthermore, as in any regular application, the results are copied back to the user's machine at the end of the execution and the running containers are shut down and removed.

Once the application is defined, we use Docker-Compose to deploy the containers in the Docker cluster managed by Docker-Swarm. In this phase, as depicted in Figure 5.6, an application network is also created across the containers on top of the overlay network which interconnects the Docker hosts.

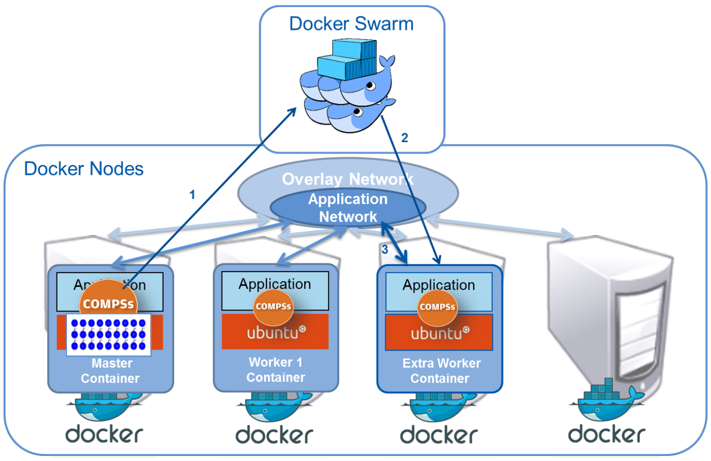


FIGURE 5.6: Dynamic integration with Docker.

We highlight that the separation of the image creation and application execution enables developers with an easy way to distribute their applications. Moreover, it allows other scientists to reproduce the results produced by an application. Our prototype provides a unique `runcompss_container` command to do both, the image creation and application execution. As shown in Listing 5.2, when providing the `--gen_image` flag, the command creates the application image and uploads it to the DockerHub repository. On the other hand, to simply execute the application the users can provide the engine, the Docker-Swarm endpoint, the Docker image identifier, the number of containers used as workers, and the application arguments.

```
# Image generation
runcompss_container \
   --gen-image \
   --context_dir=/home/john/matmul/

# Execution
runcompss_container \
   --engine=docker \
   --engine-manager='129.114.108.8:4000' \
   --initial-worker-containers=5 \
   --container_image='john123/matmul-example' \
   --classpath=/home/john/matmul/matmul.jar \
   matmul.objects.Matmul 16 4
```

LISTING 5.2: Image generation and execution of a sample application in Docker using the *runcompss_container* command.

Finally, we have demonstrated the static container management using Docker, but our design can be extended to other container engines. For instance, the same functionality can be achieved using Kubernetes as Docker-Swarm and Kompose [128] as Docker-Compose. Since the application's deployment is already defined using a `docker-compose.xml` file, Kompose will be used to translate this description to the Kubernetes' format, and the different services will be deployed using the Kubernetes API. Hence, Kubernetes can be integrated with our prototype using the same design shown in Figure 5.5 but changing Docker-Swarm and Docker-Compose per Kubernetes and Kompose, respectively.

5.5.2 HPC computational resources

We consider HPC computational resources those resources that are requested to ROP and reserved for the whole execution of the application. This is often used in supercomputers where queue managers grant a fixed set of resources for a certain period of time.

5.5.2.1 COMPSs HPC resources management

As shown in Figure 5.7, COMPSs is integrated with the ROP by providing a high-level submission command (`enqueue_compss`) that requests the reservation to the ROP, configures and deploys the master and worker processes in each node, and launches the application. At its current state, COMPSs supports SLURM, LSF, PBS, and SGE; although system administrators must fill a configuration template with the specificities of each supercomputer.

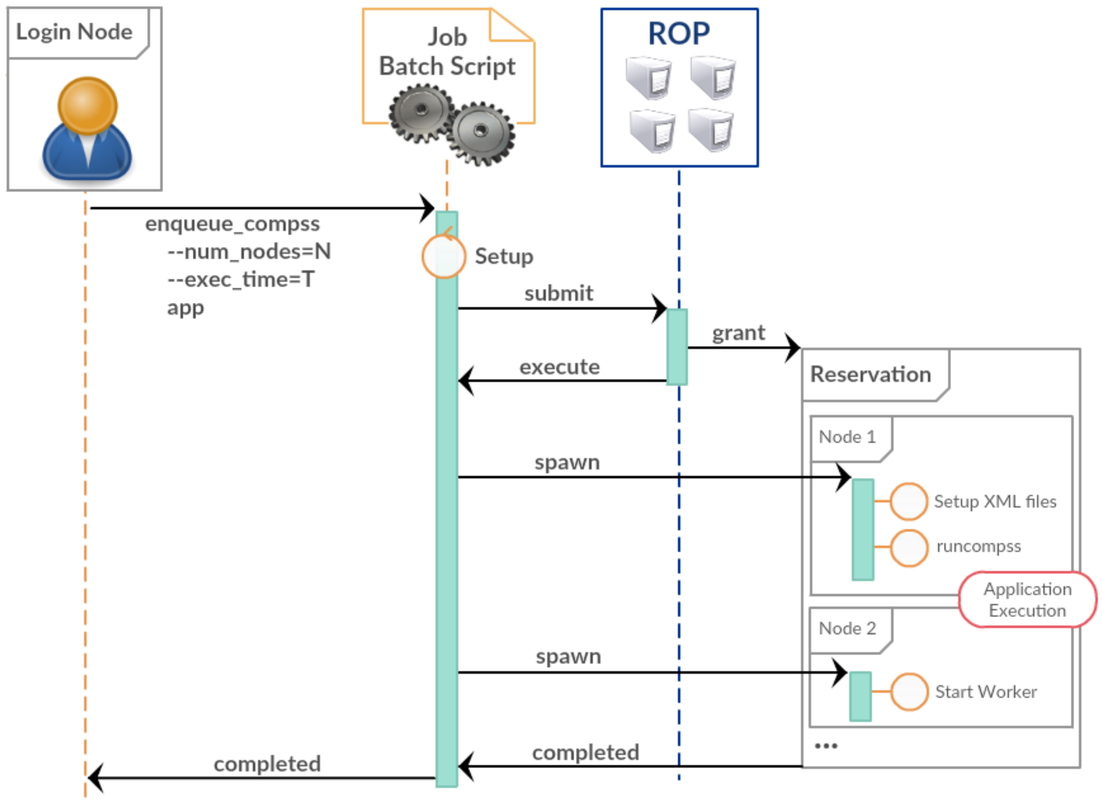


FIGURE 5.7: COMPSs HPC resources management

5.5.2.2 New HPC container management

The process to run containerised applications in HPC infrastructures is very similar to the one used to run regular HPC applications with COMPSs. From the users' point of view, the only required modification is to provide the `container_image` flag to the `enqueue_compss` command to specify the id of the image that contains the application. The image must be previously generated following the steps explained in Section 5.5.1.2.2.

Internally, as shown in Figure 5.8, our prototype retrieves the container image from DockerHub and converts it to the required format during the submission phase. Moreover, our prototype starts a container in each of the available resources during the deployment phase so that, later, the application can be run normally as any regular application.

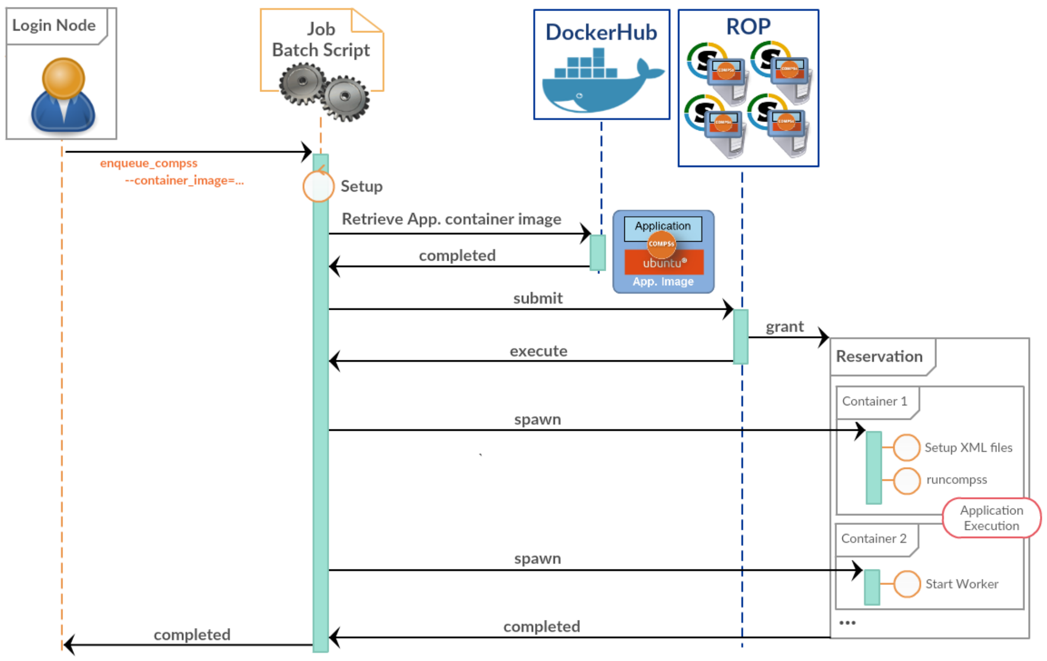


FIGURE 5.8: Containers HPC resources management

5.5.2.3 Use Case 2: Singularity

This use case shows the execution of containerised applications in HPC using Singularity. First, Listing 5.3 shows the submission command for regular and containerised applications. As previously stated, we highlight that the only difference between both commands is the addition of the `container_image` flag.

Second, Figure 5.9 shows how our prototype interacts with the HPC system with Singularity to deploy and execute containerised applications. When the `--container_image` flag is activated, the `enqueue_compss` command imports the image from DockerHub and creates a Singularity image by invoking the `singularity import <container-image>`. Then, as with regular applications, the command generates a submission script which generates the queue system directives to perform the reservation of the nodes. However, instead of starting the master and worker processes directly, it starts a container in each node that runs the master or worker process according to the node configuration.

```
# Normal cluster execution
enqueue_compss \
  --exec_time=30 \
  --num_nodes=5 \
  --classpath=/cluster/home/john/matmul/matmul.jar \
  matmul.objects.Matmul 16 4

# Singularity cluster execution
enqueue_compss \
  --exec_time=30 \
  --num_nodes=5 \
  --container_image='john123/matmul-example' \
  --classpath=/home/john/matmul/matmul.jar \
  matmul.objects.Matmul 16 4
```

LISTING 5.3: Submission comparison between a normal and a Singularity cluster.

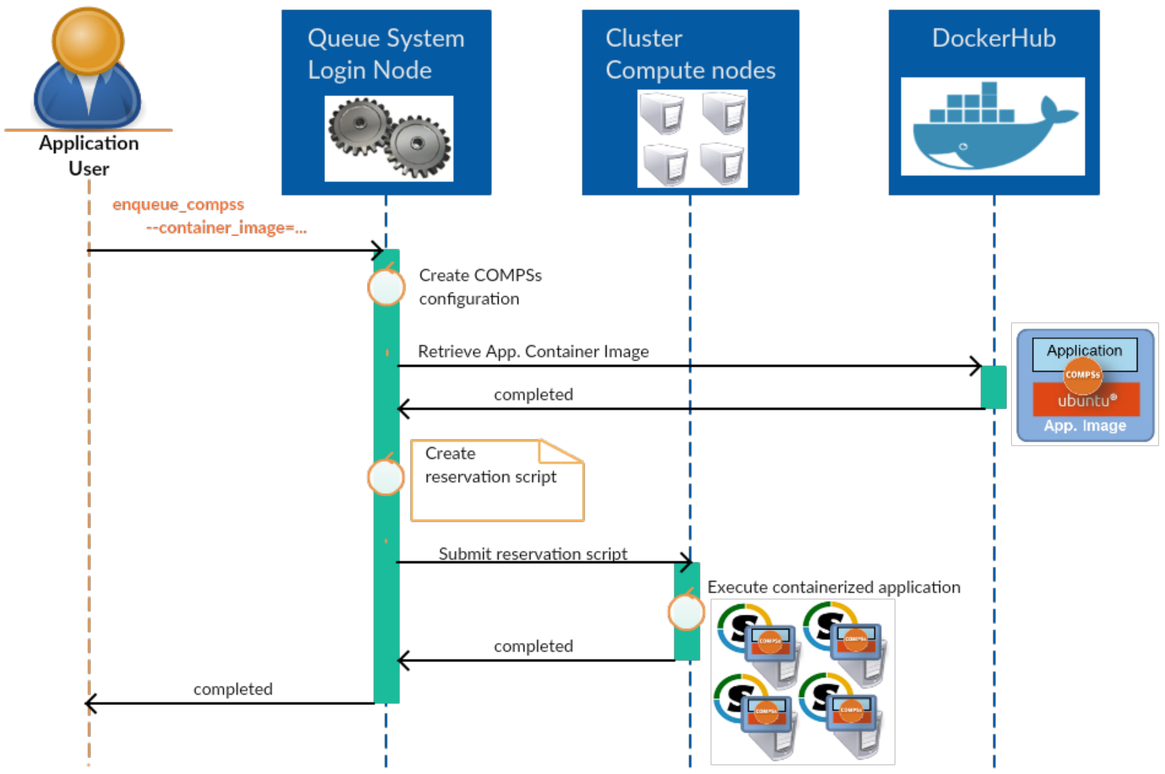

Application User
Queue System Login Node
Cluster Compute nodes
DockerHub
enqueue_compss --container_image=...
Create COMPSs configuration
Retrieve App. Container Image
completed
Application
COMPSs
ubuntu
App. Image
Create reservation script
Submit reservation script
Execute containerized application
completed
completed

FIGURE 5.9: Application deployment with Singularity.

5.5.3 Dynamic computational resources

We consider dynamic computational resources those resources that can be requested and freed during the application's execution. Although this scenario was originally designed for clouds, nowadays, queue managers and container platforms can also provide elasticity mechanisms for on-demand resource reservation.

5.5.3.1 COMPSs dynamic resources management

One of the main benefits of Cloud computing platforms is elasticity [89, 145]. Users can request more or fewer resources according to their needs. At its starting point, COMPSs already has built-in adaptation mechanisms to dynamically increase or decrease the number of cloud VMs during the application's execution depending on the remaining workload. To do so, it estimates the workload by profiling the previous executions of each task, measures the resource creation time, and compares both values to order the creation or the destruction of a resource. In its current state, COMPSs has connectors for ROCCI [180, 201], jClouds [19], SLURM, and VMM [87, 228].

COMPSs defines a *Connector* interface to interact with the ROP. This interface provides two methods to create and destroy COMPSs workers; abstracting the logic of requesting and freeing computational resources from the ROP specificities to actually request or free computational resources. On the one hand, the `create_resource` method expects the connector implementation to request the allocation of a new resource to the ROP, configure the granted resource, and start the worker process. On the other hand, the `destroy_resource` method expects the connector implementation to simply de-allocate the resource since the worker process is already stopped and all the required data is already transferred elsewhere.

As shown in Figure 5.10, at the beginning of the application's execution, the COMPSs master loads by reflection the *Connector* implementation to interact with the ROP. During the

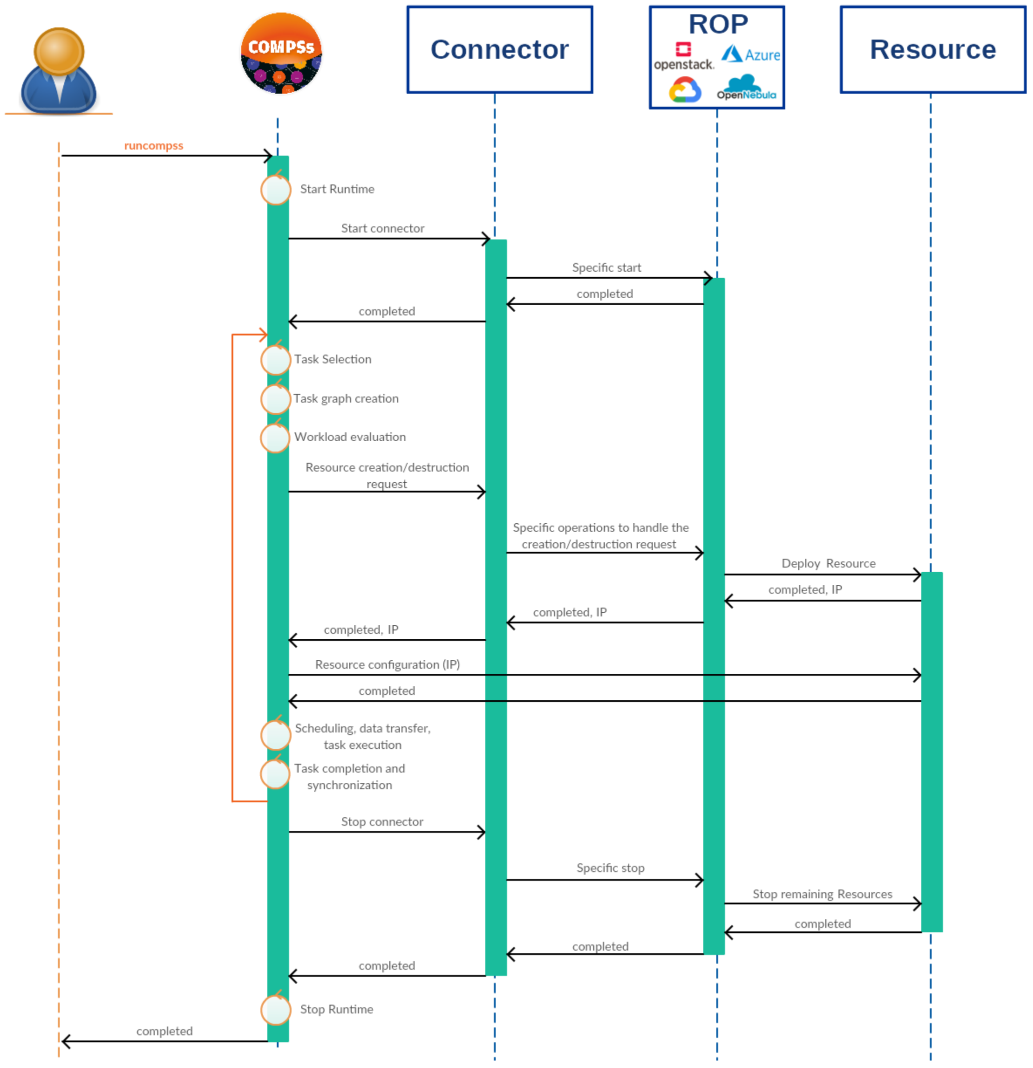


FIGURE 5.10: COMPSs dynamic resources management

application's execution, the COMPSs master evaluates its policies and, eventually, asks the Connector to allocate or de-allocate resources. This logic is performed in a separated thread (*ResourceOptimizer*) and, as previously stated, is independent of the ROP. Finally, at the end of the application's execution, the COMPSs master instructs the *Connector* implementation to de-allocate any remaining resource and perform any shutdown operation.

5.5.3.2 New dynamic container management

Since many container engines can request and free containers, we have extended the *Connector* interface to support containers. Furthermore, we have implemented two new connectors to support Docker and Mesos. Notice that the *Connector* interface abstracts the logic of creating and destroying resources from the ROP. Thus, we have not modified the Runtime logic to extend the adaptation mechanisms to support container engines. However, the *Connector* interface is general enough to fit any other framework with adaptation mechanisms, which makes our Docker and Mesos connector implementations portable and extensible for other high-level abstraction frameworks.

From the users' point of view, the applications can run distributedly on top of a container platform without modifying a single line of code. It is sufficient to modify our prototype's configuration files by adding the *Connector* path (provided in the installation), the platform manager endpoint, the container image, and the initial, the minimum and the maximum number of containers.

5.5.3.3 Use Case 3: Docker

The integration with Docker includes a pluggable *Connector* implementation which connects the resource manager with Docker-Swarm and allows it to deploy or destroy containers according to the decisions taken by the Runtime. If the Runtime decides that an additional container is needed, it contacts the Docker-Swarm manager to request the creation of a container using the application image and the application network. Then, the Docker-Swarm manager deploys the extra worker container, starting the worker daemons and connecting the new container to the application network that exists across the containers. This plug-in is included in the base image and is automatically configured by the `runcompss_container` script.

5.5.3.4 Use Case 4: Mesos

A Framework running on top of Mesos consists of two components: a scheduler and an executor. On the one hand, the scheduler registers with the Mesos master, receives resource offerings from it, and decides what to do with the offered resources. On the other hand, the executor is launched on the slave nodes to run framework tasks.

Figure 5.11 depicts the integration with Mesos. In our case, the scheduler is integrated with the Runtime, and the negotiation of resources is performed through a specific *Connector* (Mesos Framework in the Figure) that registers a new framework in Mesos. Notice that both the Runtime and the workers are executed as Mesos slaves within Docker containers and that we create an overlay network on the Mesos cluster to make direct connections.

Once the resources are offered to our prototype, it deploys the workers on the nodes creating a direct connection between the master and the workers (blue arrows in the Figure). Our implementation uses the default Mesos executor to spawn the containers.

It is worth highlighting again that the integration of Mesos is completely transparent to the application developers who are not requested to provide any information related to the resources in the definition of the tasks. Moreover, the adoption of containers allows easy

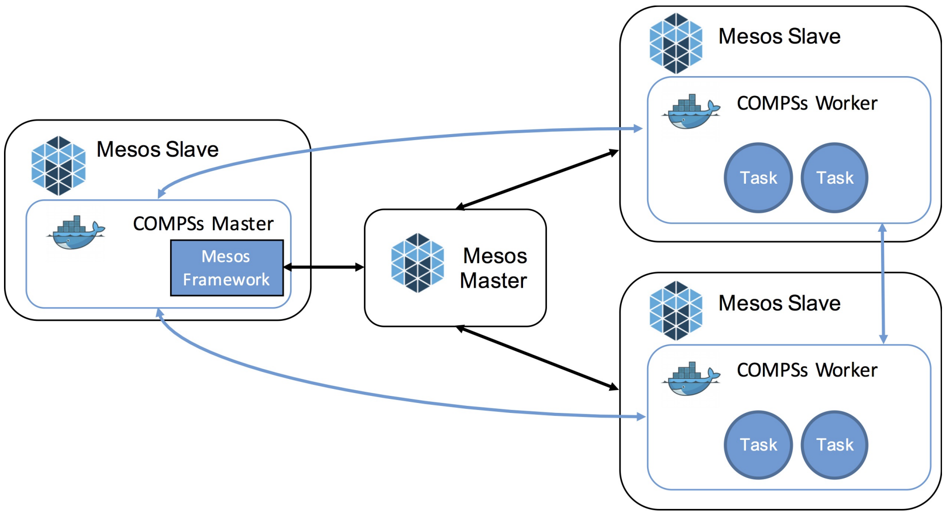


FIGURE 5.11: Dynamic integration with Mesos.

and transparent deployments of applications, without the need of installing our prototype and the developed applications in the cluster.

To execute an application into the Mesos cluster, our prototype uses Chronos [51] to pass a JSON file with the description of the container to deploy and the command to be executed. Listing 5.4 contains the description of a Simple application with the definition of the Docker image to be deployed and the URIs of the files to be copied in the sandbox of each worker.

```
{
     "name": "COMPSs",
     "command": "/opt/COMPSs/Runtime/scripts/user/runcompss
 --project=/mnt/mesos/sandbox/project_mesosFramework.xml
 --resources=/mnt/mesos/sandbox/resources_mesosFramework.xml
 --classpath=/mnt/mesos/sandbox/Simple.jar
 simple.Simple 1 25 1 3 60",
     "shell": true,
     "epsilon": "PT30M",
     "executor": "",
     "executorFlags": "",
     "retries": 2,
     "owner": "john@bsc.es",
     "async": false,
     "successCount": 190,
     "errorCount": 3,
     "cpus": 0.5,
     "disk": 5120,
     "mem": 512,
     "disabled": false,
     "container": {
         "type": "DOCKER",
         "image": "compss/compss:2.0-mesos-0.28.2",
         "network": "USER"
     },
     "uris": [
         "http://bscgrid05.bsc.es/~john/Matmul.tar.gz",
         "http://bscgrid05.bsc.es/~john/conf.tar.gz",
         "http://bscgrid05.bsc.es/~john/DockerKeys.tar.gz"
     ],
     "schedule": "R1//PT24H"
}
```

LISTING 5.4: Definition of an application execution with Chronos.

The previous example shows an interesting feature of our prototype to deploy the application at execution time dynamically. The user provides the package of the application in a compressed file and lists it in the URIs section of the JSON document so that Chronos will copy it in the sandbox of the master container. Then, the Runtime transfers it to the worker containers at execution time. This mechanism is particularly useful for testing purposes, allowing to use the base Docker image without creating a new layer and uploading it to the DockerHub; leaving this step only for the final version of the application.

5.6 Evaluation

We have defined two sets of experiments to evaluate the integration of our prototype with the different container platforms. On the one hand, the first set evaluates the deploying time and adaptation capabilities. On the other hand, the second set evaluates the performance of running two benchmark applications on top of the different container platforms in comparison to normal executions in bare-metal, Cloud Infrastructure, or HPC cluster. This second set evaluates if there is any performance degradation caused by the use of containers.

5.6.1 Computing infrastructure

The experiments have been done using two different infrastructures. On the one hand, static and dynamic container management has been tested using the Chameleon Cloud [46]. Chameleon Cloud is an NSF funded project which provides a configurable experimental environment for large-scale cloud research. The Chameleon Cloud provides two types of infrastructure services: a traditional cloud managed by OpenStack over KVM and a bare-metal reconfiguration, where the user can reserve a set of bare-metal nodes flavoured with the image that the user selects. On the other hand, HPC container management has been tested using the MareNostrum 3 supercomputer [148].

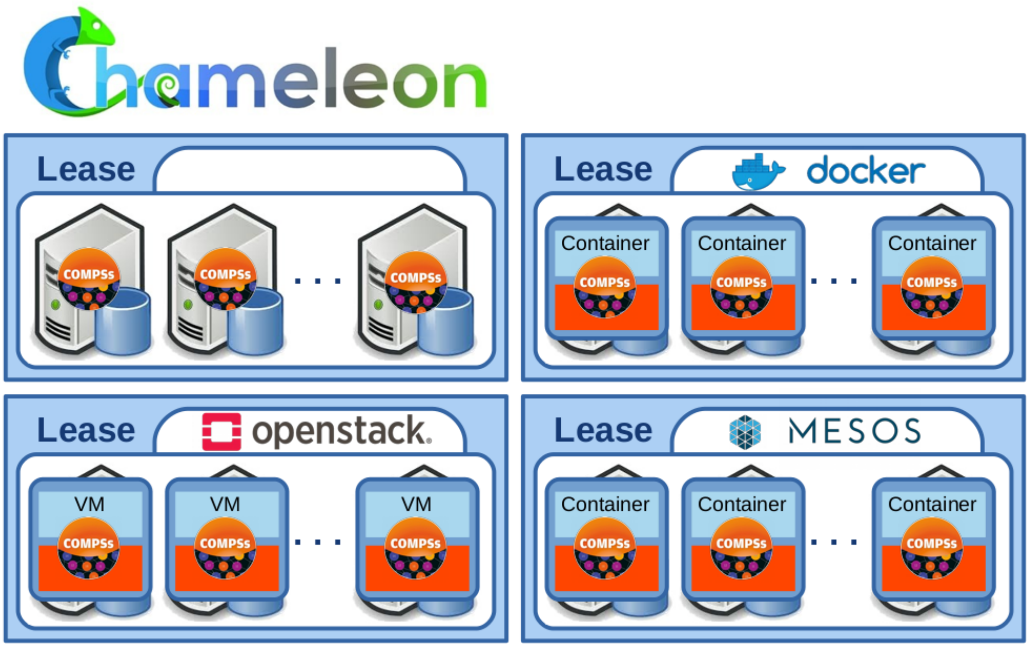


FIGURE 5.12: Chameleon Cloud testbed environment configurations.

Figure 5.12 describes the different Chameleon Cloud scenarios that we have evaluated. We have set up four different environments: Bare-metal, KVM-OpenStack, Docker, and Mesos. The first scenario consists of a set of bare-metal flavoured nodes where we directly run the applications. The second scenario consists of a Cloud where OpenStack manages a set of nodes virtualised by KVM. The third scenario consists of a Docker-Swarm cluster built on top of a set of bare-metal instances. In this case, each bare-metal node hosts a Docker Engine, which deploys the applications' containers. Finally, the fourth scenario consists of a Mesos cluster on top of a set of bare-metal instances. The Mesos cluster uses the DC/OS platform that includes a virtual network feature that provides an IP-per-Container for Mesos and Docker containers. These virtual networks allow containers launched through the Docker Containeriser to co-exist on the same IP network, allocating each container its own unique IP address. This is a requirement for our prototype because each worker is deployed in a container and the Runtime needs to connect to each worker directly.

Each scenario uses up to 9 bare-metal nodes provided by Chameleon with exactly the same configuration (2 x Intel Xeon CPU E5-2670 v3 with 12 cores each and 128GB of RAM). When running an application, one node, container or VM runs as master (which manages the application execution), and the rest run as workers (which execute the application tasks). In all the environments, nodes, containers, and VMs are defined to use the whole compute node (24 VCPUs and 128 GB of RAM) and deploy the same software stack (Ubuntu-14.04 with our prototype installed). However, depending on the environment, the image size varies. The qcow2 image size for the cloud environment is about 1GB, and the compressed docker image size is about 800MB. Regrading the OpenStack services to manage images and create instances, we have used the installation provided by Chameleon described in [47].

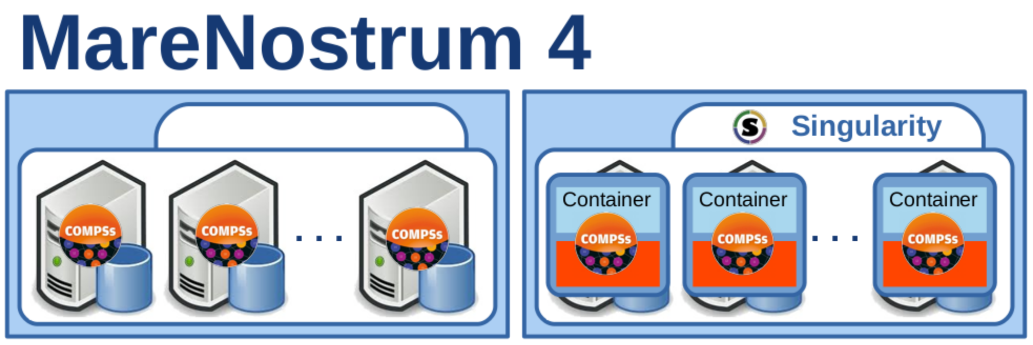


FIGURE 5.13: MareNostrum 3 testbed environment configurations.

Figure 5.13 describes the different MareNostrum 3 scenarios that we have evaluated. We compare the execution in a set of bare-metal nodes against a second scenario using Singularity containers. Each scenario uses up to 9 bare-metal nodes with exactly the same configuration (2x Intel SandyBridge-EP E5-2670/1600 20M 8-core at 2.6 GHz with 8x16 GB DDR3-1600 DIMMs of RAM). When running an application, one node, or container runs as master (which manages the application execution), and the rest run as workers (which execute the application tasks).

5.6.2 Benchmark applications

The experimentation performed consists of the deployment and execution of two benchmark applications in the different environments. The first application consists of a blocked multiplication of two big matrices (*Matmul*). This application presents a large number of data dependencies between tasks. Specifically, each matrix of the experiment contains 2^{28}

floating-point elements (2^{14} x 2^{14}). In order to share the workload, they have been divided into 256 square blocks (16 x 16), each of them containing 2^{20} elements. It is quite I/O intensive because the data dependencies between tasks require transferring the matrix blocks through the network as well as some disk utilisation.

In contrast, the second experiment is an embarrassingly parallel application without data dependencies. This benchmark simply performs a series of trigonometric computations in parallel without exchanging any data. In this case, the I/O utilisation is mainly used by the messages exchanged by our prototype to run the parallel computations.

5.6.3 Docker

5.6.3.1 Deployment evaluation

In the case of the deployment evaluation, we have measured the time to perform the deployment of the applications into the considered environments in different scenarios (when the image is already in the infrastructure, or not, etc.). The measurements for the KVM-OpenStack and Docker scenarios are summarised in Table 5.4. Since both benchmark applications have the same size and the deployment times of the Embarrassingly Parallel benchmark are very similar, we only present the Matrix Multiplication times. Moreover, for this experiment, we have not considered the bare-metal scenario since the computing nodes are already set-up and the deployment of our framework and the applications must be performed manually copying and installing the required files on all the nodes.

	Cloud (KVM/OpenStack)			**Docker**				
Phase	**Action**	**Time (s)**		**Action**	**Time (s)**			
		w/o Image	**Image Cached**		**w/o Images**	**w Ubuntu**	**w COMPSs**	**w App.**
Build	Base VM deployment	33.58	N/A	Image Creation	73.87	68.88	15.48	N/A
	App. Installation	15.45	N/A	Image upload	8.66	8.66	8.66	N/A
	Image Snapshot	60.36	N/A					
Total Construction		109.39	N/A		82.53	77.54	24.14	N/A
Deployment	VM deployment	83.68	18.18	Image download	12.39	12.39	12.39	N/A
	VM boot	15.09	15.09	Container deployment	4.75	4.75	4.75	4.75
Total Deployment		98.77	33.27		17.28	17.28	17.28	4.75
Total Construction & Deployment		208.16	33.27		99.67	94.68	41.28	4.75

TABLE 5.4: Docker deployment evaluation.

In the construction phase, the creation of a customised image includes the deployment of a VM with the base image, the time to install the application and to create the snapshot which is the most expensive phase and whose duration depends on the image size. In this case, the creation time takes 109 seconds. In contrast, the creation of application's Docker container depends on which layer we already have in the Docker infrastructure because the new image is a layer on top of previous ones. In this case, the Docker image creation takes from 24 seconds, when the base image is cached in the Docker engines, up to 82 seconds when no images are available.

At deployment phase, both cases (Cloud and Docker) are quite similar if the node has the image locally cached or must be downloaded from a central image store. In the best case for the cloud environment, the deployment and boot are completed in around 30 seconds. However, if the image must be downloaded, the deployment can take up to around 98 seconds. Hence, the total creation and deployment time in the case of a Cloud can take from 33 seconds up to 208 seconds. In contrast, the container deployment in the best case takes around 5 seconds, when all the layers are cached in the Docker engines, while in the worst case it takes 17 seconds. Thus, the total creation and deployment time in Docker can take from 5 seconds up to 99 seconds, significantly improving the deployment.

Notice that the faster deployment time of Docker is significantly relevant when adapting the computational resources to the workload of data science applications. The faster the resource deployment is, the finer the adjustment of the resources can be, which implies a faster application execution and reducing the underutilisation of computational resources.

5.6.3.2 Performance evaluation

To evaluate the performance, we have measured the execution time of 5 runs of both applications using a different number of nodes in the different environments. Regarding the Matrix Multiplication, Figure 5.14 shows the average execution time, and Figure 5.15 depicts the overhead with respect to the bare-metal execution. Similarly, Figures 5.16 and 5.17 show the average execution time and the overhead with respect to the bare-metal execution for the Embarrassingly Parallel benchmark.

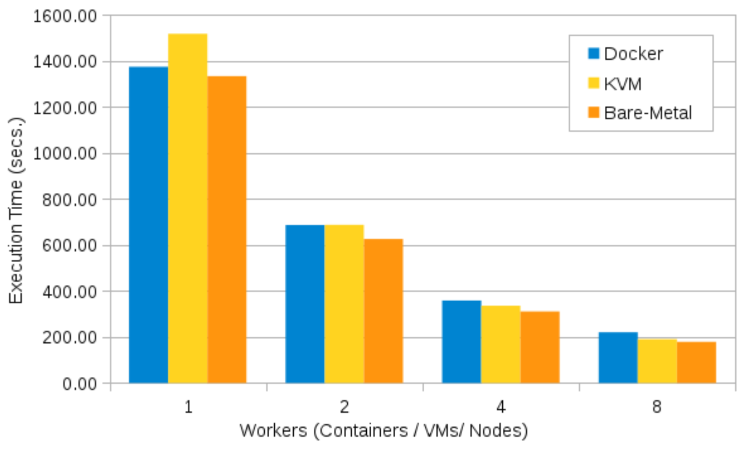


FIGURE 5.14: Scalability evaluation of the Matrix Multiplication (with data dependencies).

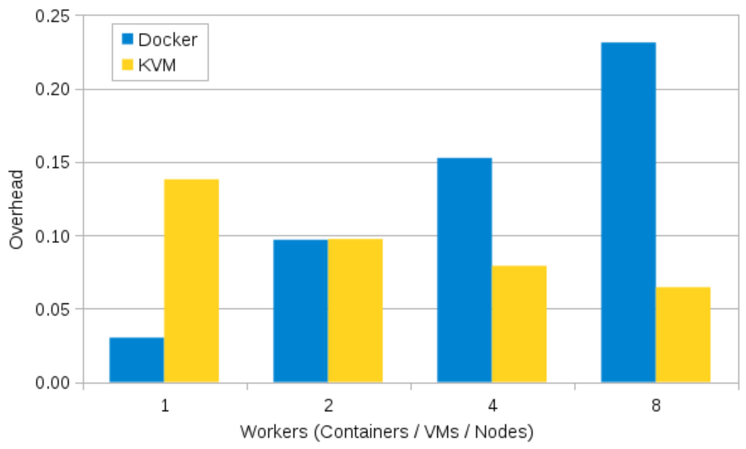


FIGURE 5.15: Overhead evaluation of the Matrix Multiplication (with data dependencies) with respect to the bare-metal.

The Matrix Multiplication case uses disk and network I/O in order to transfer and load the matrix blocks required to compute the partial multiplications. Figures 5.14 and 5.15 show that Docker and bare-metal are performing similarly when using a single node, and KVM is performing a bit slower than bare-metal (around 14%). This is because KVM has more overhead when managing disk I/O than Docker, as observed in previous comparisons [85].

However, when increasing the distribution of the computation (2, 4, and 8 nodes), the computation and the disk I/O overhead are also distributed across the nodes, and the networking usage is increased because the more resources we have, the more matrix blocks transfers are required. Hence, the multi-host networking overhead increases and becomes the most important source of overhead. When using two nodes, the overhead is almost the same in KVM and Docker cases, while when four and eight nodes KVM performing better than Docker.

To verify this assumption, we have performed a small network performance experiment. In the same infrastructure than the previous tests, we have transferred a file of 1.2GB. First, between two bare-metal nodes, then between two VMs deployed with OpenStack/KVM, and, finally, between two Docker containers using the overlay network. The results of this experiment are summarised in Table 5.5, where we can see that the networking overhead in the Docker overlay network is significantly bigger than other approaches.

Scenario	**Tranfer Time (s)**
Baremetal	6.54
KVM/OpenStack	6.97
Docker Overlay	8.50

TABLE 5.5: Docker networking evaluation.

In the case of the Embarrassingly Parallel benchmark, as shown in Figures 5.16 and 5.17, all the cases are performing very similarly (overheads are between 1 and 10%). This is because the overhead introduced by Docker and KVM in terms of CPU and memory management is relatively small. The difference is basically due to the multi-host networking used by the Runtime to send the messages to execute tasks in the workers remotely. The default *overlay* network of Docker is performing worse than the *bridge* network of KVM. The relative

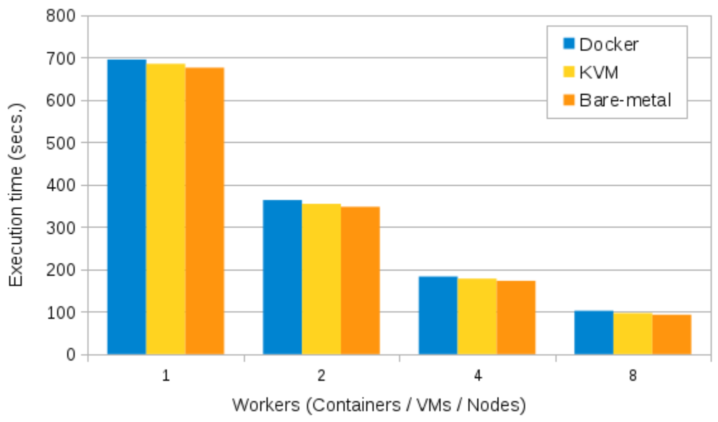


FIGURE 5.16: Scalability evaluation of the Embarrassingly Parallel application (without data dependencies).

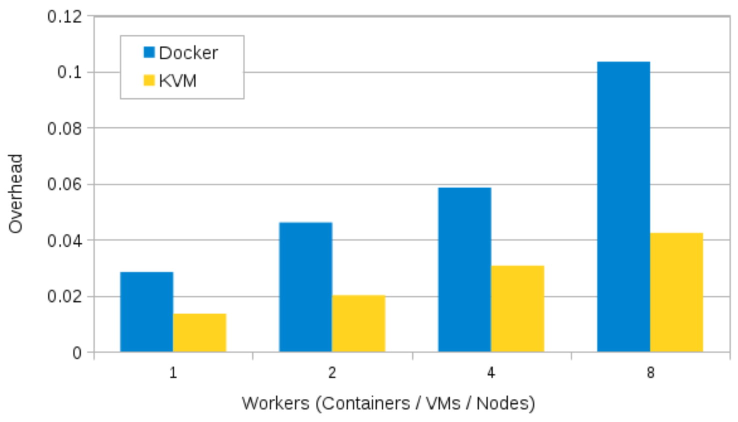


FIGURE 5.17: Overhead evaluation of the Embarrassingly Parallel application (without data dependencies) with respect to the bare-metal.

overhead increases with the number of nodes used, mainly because the computation time is reduced due to the increased parallelism available, but the number of transfers required to run the tasks is still the same because the number of tasks is the same.

5.6.3.3 Adaptation evaluation

To validate how the deployment time influences the adaptation of the application execution, we have executed the same Matrix Multiplication without any initial worker container

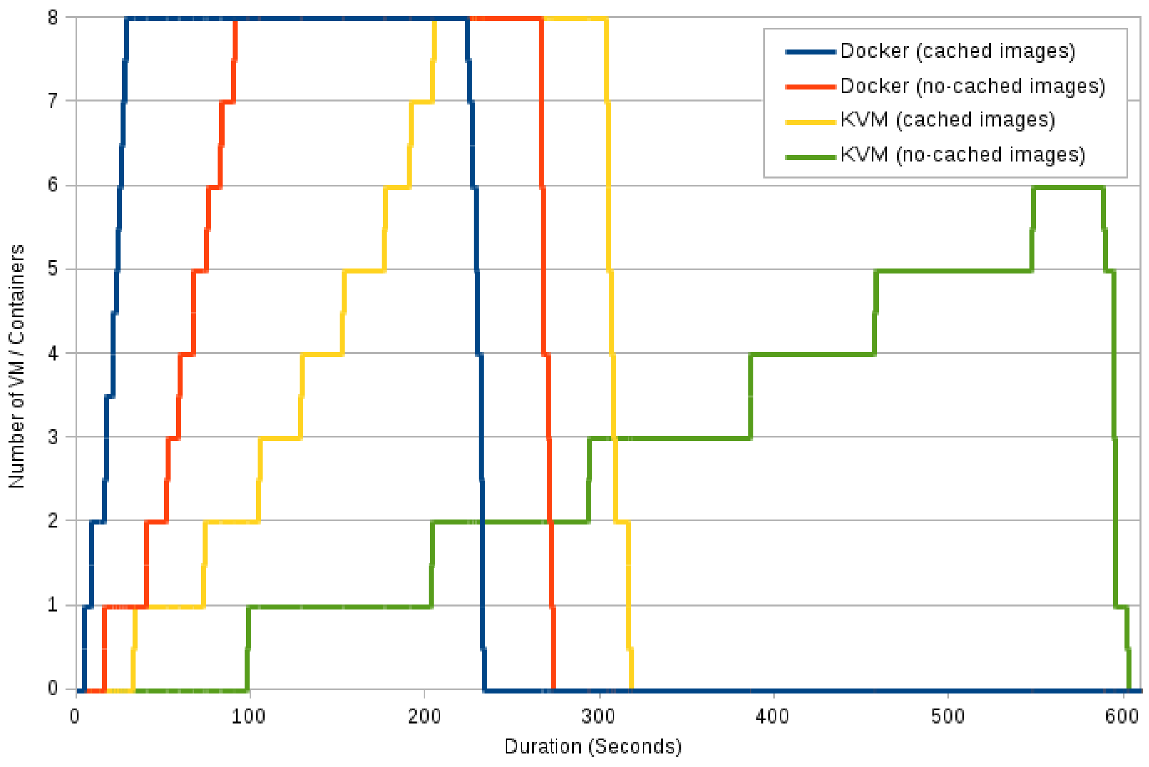


FIGURE 5.18: Deployment time effect in the Matrix Multiplication resource adaptation.

in order to see how the Runtime adapts the number of resources to the application load with different deployment times.

Figure 5.18 shows the execution time and the number of VM/Containers created during the application execution in the Docker cluster and the OpenStack/KVM cloud environments. In both environments, we have run the same application twice. In the first run, the images were not stored in the computing nodes so, in both environments, the images had to be downloaded from the DockerHub or the OpenStack image repository respectively. In the second run, images were already cached in the computing nodes, so the total deployment time only considers the deployment and boot times of VMs or containers. In both cases, the Docker scenario exhibits faster adaptation because the Runtime detects earlier that having extra resources speeds up the execution since the resources are available earlier for execution.

5.6.4 Singularity

Similar experiments have been performed to evaluate the integration with Singularity. In this case, we have not evaluated the adaptation since Singularity is usually combined with other resource managers or queue systems like SLURM.

5.6.4.1 Deployment evaluation

In the case of container deployment, Table 5.6 shows the time to deploy a Singularity container in different scenarios, compared with the Docker case.

	Singularity			**Docker**				
Phase	**Action**	**Time (s)**		**Action**	**Time (s)**			
		w/o Image	**Image Cached**		**w/o Images**	**w Ubuntu**	**w COMPSs**	**w App.**
Deployment	Image import	79.80	N/A	Image import	75.68	63.47	17.35	N/A
	Container deployment	0.45		Container deployment	4.75	4.75	4.75	4.75
Total Deployment		80.25	0.45		80.43	68.22	22.10	4.75

TABLE 5.6: Singularity deployment evaluation.

The application image construction phase is the same than the Docker scenarios because Singularity can import Docker images. However, to run the application in Singularity containers, the Docker image of the application must be imported and converted to Singularity as explained in Section 5.5.2.3. This process includes the download of the application image and bootstraps the Docker image in a Singularity image. The main drawback of this conversion is that Singularity does not cache the previously downloaded layers. Hence, it can not take advantage of the layered-based feature of the Docker images to reuse the already cached layers and, every time we need to convert an application image because it is not in the compute cluster, Singularity has to download all the application image layers. In contrast, the deployment of a container once the image has been converted is considerably faster than Docker.

5.6.4.2 Performance evaluation

In this case, we have executed 5 runs of the Matrix Multiplication benchmark in the MareNostrum 3 supercomputer with Singularity and without it. Figure 5.19 shows the comparison of the average execution time in both configurations. Notice that both runs perform similarly and the overhead at execution time is very low.

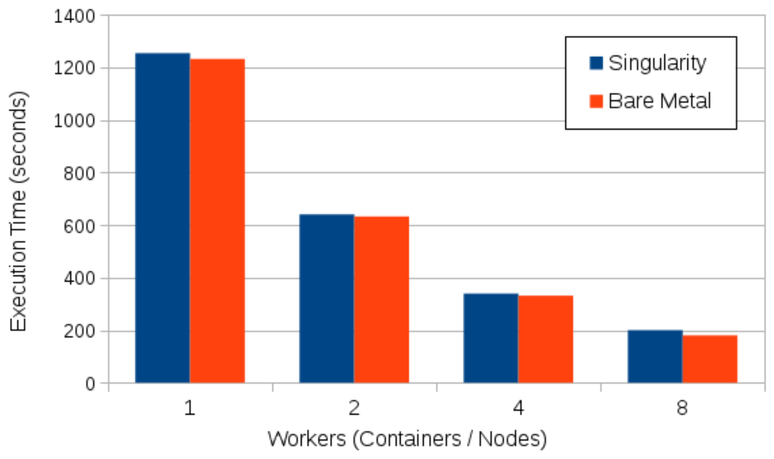


FIGURE 5.19: Matrix Multiplication application execution with Singularity.

5.6.4.3 Porting of a real-world application: GUIDANCE

We have ported GUIDANCE [102, 101] to use our prototype in combination with Singularity. GUIDANCE implements a Genome-Wide Association Studies (GWAS) workflow in Java. In genetics, a Genome-Wide Association Study tries to unveil the correlation between genomic variants and phenotypes (observable characteristics such as diseases). The complete workflow defines five steps that require 10 state of the art binaries (i.e., SHAPEIT [68], Eagle [143], Impute [111], Minimac3 [60], snptest [147], PLINK [192], QCTool [147], BCFTools [166], SAMTools [140], and R [198]).

Until now, scientists usually execute this type of workflows step by step. For each step, they submit a set of jobs which execute the binary that performs an independent partial analysis of a given data set. Once a step is finished, they execute another step which performs another analysis based on previous results. Moreover, depending on the input data (e.g., phenotypes, subjects, reference panels, and chromosomes), the workflow performs different computations that have different computational and memory requirements. For instance, the requirements can vary more than one order of magnitude depending on the case; adding a high degree of heterogeneity to the execution. Also, the of the association phase differs significantly depending on the number of variants present in each of the chunks; causing a severe load imbalance when using static scheduling.

The porting of GUIDANCE to our prototype has several benefits. First, the whole execution can be orchestrated in a single run; preventing the users from manually defining several jobs. Second, the resource manager and the scheduler are capable of defining different computational and memory requirements per task while exploiting all the available resources and mitigating the load imbalance. Third, the use of containers provides a homogeneous environment for all the required state of the art tools regardless of the underlying infrastructure.

Although we cannot provide in-depth details due to confidentiality, the experimentation has been performed using the MareNostrum 4 supercomputer [149], and following the same steps than the concept applications described in the previous sections. First, we have built a Docker image containing GUIDANCE, its dependencies, and our prototype. Next, we have executed the application using the `enqueue_compss` command with the `--container_image` flag. Our experimentation uses up to 4800 cores (100 nodes), spawning 93,858 tasks, generating 120,018 files (217.68 Gb), and analyzing 2,860 subjects.

5.6.5 Mesos

5.6.5.1 Deployment evaluation

Since we are using the same image as in the Docker experiments, we have not evaluated the image construction phase. Moreover, for the deployment phase, we have used the Mesos default Docker executor to spawn the containers in the slave nodes obtaining the same times, with no significant overhead compared to the Docker experiments listed in Table 5.4.

5.6.5.2 Performance evaluation

To evaluate the performance of the extensions to support Mesos, we have measured the execution time of 5 runs of the Matrix Multiplication application using different number of nodes.

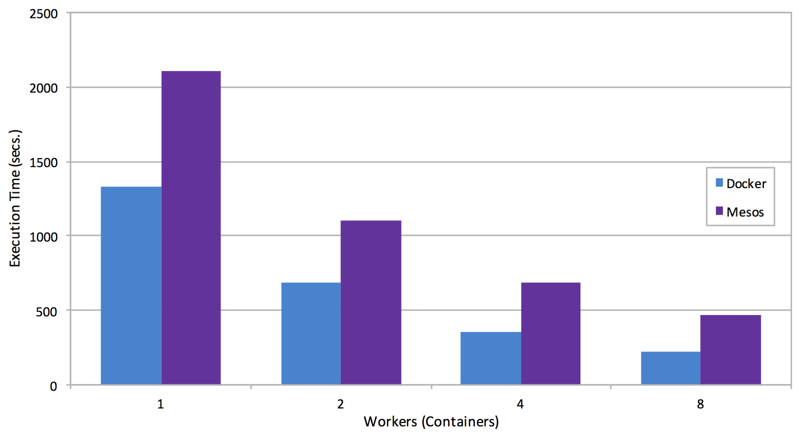


FIGURE 5.20: Mesos scalability evaluation of the Matrix Multiplication (with data dependencies).

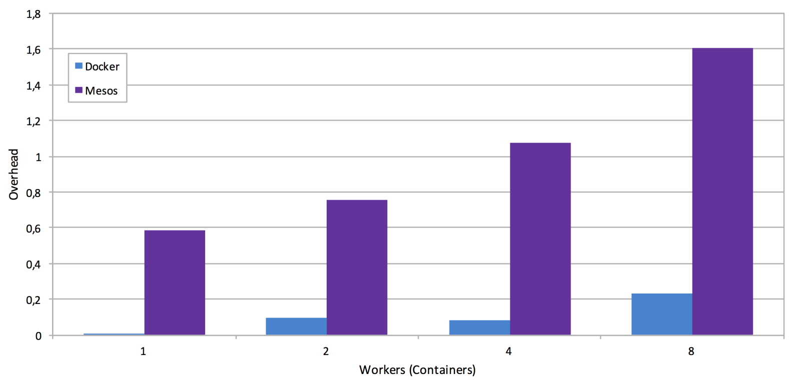


FIGURE 5.21: Mesos overhead evaluation of the Matrix Multiplication (with data dependencies) with respect to the bare-metal.

Figures 5.20 and 5.21 depict the average execution times and the overhead compared to the average values of Docker that were presented in the previous section. Notice that the computation times are higher than in all the previous experiments still providing a good scalability. Looking at the overhead figure, it can be argued that the overhead is caused by the heavy usage of the overlay network and of the disk I/O to transfer the blocks of the matrix. In particular, we had to deploy a DC/OS virtual network for Mesosphere that adds network agents in each node to enable the connections across the containers. Anyway, the results demonstrate that the Runtime properly adapts the tasks distribution to the availability of resources and benefits from the resources abstraction provided by Mesos.

5.7 Discussion

Our methodology integrates the different capabilities of the container platforms with task-based parallel programming models. The combination of programming models with container platforms brings several benefits for developers. On the one hand, the COMPSs programming model provides a straightforward methodology to parallelise applications from sequential codes and decoupling the application from the underlying computing infrastructure. On the other hand, containers provide an efficient image management and application deployment tools that facilitate the packaging and distribution of applications. Hence, the integration of both frameworks enables developers to easily port, distribute, and scale their applications to parallel distributed computing platforms.

The proposed application-containers integration is done considering three scenarios. First, the static container management focuses on (i) the creation of a Docker image that includes the application software and the programming model runtime, and (ii) the orchestration of the deployment and execution of the application using container resources. Also, after the creation, the application image is uploaded to the Docker-Hub repository to make it available to other users. Second, the HPC container management extends the use of containers to supercomputers with minor modifications from the users' point of view. Third, the dynamic container management fully exploits container engines by enabling adaptation mechanisms. These mechanisms adapt the available container resources during the application's execution to the remaining workload.

We have implemented a prototype of the proposed application-container integration on top of COMPSs and validated it with four use cases using Docker, Singularity, and Mesos. Furthermore, for each implementation, we have evaluated how the system behaves in the application building and deployment phases, as well as the overhead introduced at execution time in comparison to other alternatives such as bare-metal and KVM/OpenStack cloud.

The evaluation demonstrates that the application execution with Docker performs similarly to bare-metal and KVM for applications with small data dependencies. However, when using intensive multi-host networking, Docker has a bigger overhead than KVM. On the other hand, regarding the deployment, we have seen that the time to deploy containers is reduced significantly compared with VM deployment, thus enabling our prototype to adapt better the resources to the computational load, creating more containers when a large parallel region is reached, and destroying containers when a sequential or small parallel region is reached.

In the Mesos integration case, the experiments show that our prototype keeps the scalability in the execution of the applications but exhibits a bigger overhead than the Docker-Swarm implementation. Nevertheless, the adoption of Mesos is very convenient because it makes completely transparent the deployment phase; saving the user to deal with the intricacies of interacting directly with Docker.

In the case of container platforms for HPC (e.g. Singularity or Shifter), we have extended the integration of COMPSs with Cluster's Resource and Queue Managers to support the deployment and execution of containerised applications. The experimentation shows that the execution overhead is extremely low. Moreover, there are no scability issues in the container networking because Singularity does not virtualise I/O and uses the host resources directly.

Finally, it is worth pointing that our prototype brings several benefits for application developers. On the one hand, the COMPSs programming model provides a straightforward methodology to parallelise applications from sequential codes and decoupling the application from the underlying computing infrastructure. On the other hand, containers provide efficient image management and application deployment tools which facilitate the packaging and distribution of applications. Thus, the integration of both frameworks enables developers to easily, port, distribute and scale their applications to parallel distributed computing platforms. Furthermore, the application can be shared with other users by facilitating the identifier of the created application image and the application execution parameters.

As future work, we are going to evaluate experimental alternatives for Docker multi-host networking [73] to test if our prototype with Docker can perform better than KVM in all situations. For what relates to the Mesos support, we plan to perform bigger tests to evaluate the scalability, to test the adaptation capabilities with dynamically added slave nodes, and to analyse the networking issues to understand the source of overhead.

Furthermore, we plan to support Kubernetes and to enable the orchestration of applications where each task requires a separated container. This feature will allow the users to prepare before-hand a different image with only the required software and dependencies for each task; potentially achieving better deployment times.

Chapter 6

Automatic parallelisation

SUMMARY

This chapter of the thesis focuses on solving the research question **Q3**; enhancing our prototype to automatically parallelise some sequential code structures and execute them in distributed environments. Our approach has been designed for non-expert users; meaning that it provides a simple annotation so that users do not need to define tasks nor data dependencies while still achieving acceptable performances.

Hence, this part of the thesis introduces and evaluates AutoParallel, a Python module to automatically find an appropriate task-based parallelisation of affine loop nests and execute them in parallel in a distributed computing infrastructure. It is based on sequential programming and contains one single annotation (in the form of a Python decorator) so that anyone with intermediate-level programming skills can scale up an application to hundreds of cores.

The evaluation demonstrates that AutoParallel goes one step further in easing the development of distributed applications. On the one hand, the programmability evaluation highlights the benefits of using a single Python decorator instead of manually annotating each task and its parameters or, even worse, having to develop the parallel code explicitly (e.g., using OpenMP, MPI). On the other hand, the performance evaluation demonstrates that AutoParallel is capable of automatically generating task-based workflows from sequential Python code while achieving the same performances than manually taskified versions of established state-of-the-art algorithms (i.e., Cholesky, LU, and QR decompositions). Finally, AutoParallel is also capable of automatically building data blocks to increase the tasks' granularity; freeing the user from creating the data chunks, and re-designing the algorithm. For advanced users, we believe that this feature can be useful as a baseline to design blocked algorithms.

6.1 General overview

The last improvements in programming languages and models have focused on simplicity and abstraction; leading Python [203] to the top of the list of the programming languages for non-experts [44]. However, there is still room for improvement when preventing users from dealing directly with distributed and parallel computing issues. This chapter of the thesis focuses on solving the research question **Q3**; describing a methodology and an implementation to distributedly execute automatically parallelised sequential code. Even if the COMPSs programming model already provides an automatic taskification of the user code (via an interface or a method annotation), we go one step further with AutoParallel: a Python module to automatically parallelise applications and execute them in distributed environments. Our philosophy is to ease the development of parallel and distributed applications so that anyone with intermediate-level programming skills can scale up an application to hundreds of cores. In this sense, AutoParallel is based on sequential programming and only requires a single Python decorator that frees the user from manually taskifying the original code. Internally, it relies on PLUTO (see Section 6.3) to parallelise affine loop nests and to taskify the obtained code so that PyCOMPSs can distributedly execute it using any underlying infrastructure (clusters, clouds, and containers). Moreover, to avoid single instruction tasks, AutoParallel can also increase the tasks' granularity by automatically building data blocks (chunks).

Notice that the automatic parallelisation focuses intermediate-level users and thus, it has been designed to achieve a best-effort parallelisation. Readers must consider that, although experts could achieve higher performances, we are looking for a general purpose manner to parallelise the user code automatically.

6.2 Related work

Nowadays, simulations are run in distributed environments and, although Python has become a reference programming language, there is still much work to do to ease parallel and distributed computing issues. In this concern, Python can provide parallelism at three levels. First, parallelism can be achieved internally through many libraries such as NumPy [230] and SciPy [122], which offer vectorised data structures and numerical routines that automatically map operations on vectors and matrices to the BLAS [38] and LAPACK [9] functions; executing the multi-threaded BLAS version (using OpenMP [58] or TBB [222]) when present in the system. Notice that, although parallelism is completely transparent for the application user, parallel libraries only benefit from intra-node parallelism, while our solution aims for distributed computing. Moreover, NumPy offers vectorised data structures and operations in a transparent way to prevent users from defining loops to handle NumPy values directly. In contrast, our solution requires a Python decorator on top of a method containing affine loop nests. Thus, to parallelise a vector operation, the users must explicitly define the loop nests to apply an operation to all the vector elements. However, we demonstrate the benefits of combining inter- and intra-node parallelism using PyCOMPSs [215] and NumPy in [7] and [8]. Similarly, some NumPy extensions can be integrated with PyCOMPSs to boost the intra-node performance. For instance, NumExpr [173] can be used to optimise the computation of numerical expressions and, as detailed in [194], the Numba [139, 172] compiler annotations can be combined with the PyCOMPSs programming model.

Secondly, many modules can explicitly provide parallelism. The multiprocessing module [181] provides support for the spawning of processes in SMP machines using an API similar to the threading module, with explicit calls for creating processes. In addition, the

Parallel Python (PP) module [182] provides mechanisms for parallel execution of Python codes, with an API that includes specific functions for specifying the number of workers to be used, submitting the jobs for execution, getting the results from the workers, etc. Also, the mpi4py [59] library provides a binding of MPI for Python which allows the programmer to handle parallelism both inter-node and intra-node. However, in all cases, the burden of parallelism specific issues is assigned to the programmer.

Third, other libraries and frameworks enable Python distributed and multi-threaded computations such as PyCOMPSs [215], Dask [202], PySpark [195], and Pydron [161]. PyCOMPSs is a task-based programming model that targets sequential programming and provides a set of decorators to enable the programmer to identify methods as tasks and a small synchronisation API. Its runtime exploits the inherent parallelism of the applications by building, at execution time, a data dependency graph of the tasks and executing them using a distributed parallel platform (clusters, clouds, and containers). On the other hand, Dask is a native Python library that allows the creation and distributed execution of Directed Acyclic Graphs (DAG) of a set of operations on NumPy and pandas [152] objects. Also, PySpark is a binding to the widely extended framework Spark [237]. Finally, Pydron [161] is a semi-automatic parallelisation Python module that transparently translates the application into a data-flow graph which can be distributed across clusters or clouds. It offers a `@schedule` decorator to automatically parallelise calls to methods that are annotated with the `@functional` decorator. In comparison to PyCOMPSs, Pydron's `@functional` decorator is equivalent to PyCOMPSs' `@task` decorator. However, Pydron analyses the abstract syntax tree of the function's code to build the data dependency graph while PyCOMPSs requires extra parameter annotations inside the `@task` decorator. Moreover, the `@schedule` decorator enables users to activate and deactivate the parallelisation of `@functional` methods, while PyCOMPSs parallelises any call to a `@task` method.

6.3 PLUTO

This section introduces PLUTO [187, 39] since it is the key software to extend our prototype with automatic parallelisation. We have chosen PLUTO because (i) it provides an abstract representation of loop nests available from any language (which allows our prototype to handle loop nests from Java, Python and C/C++), and (ii) because it is capable of proposing a parallelisation without actually executing the generated code. Moreover, we had the opportunity to collaborate with the PLUTO designers and developers for a few months during the development of this thesis. However, as stated in the future work, AutoParallel is designed so that PLUTO can be easily substituted by other tools (such as Apollo [213, 151]).

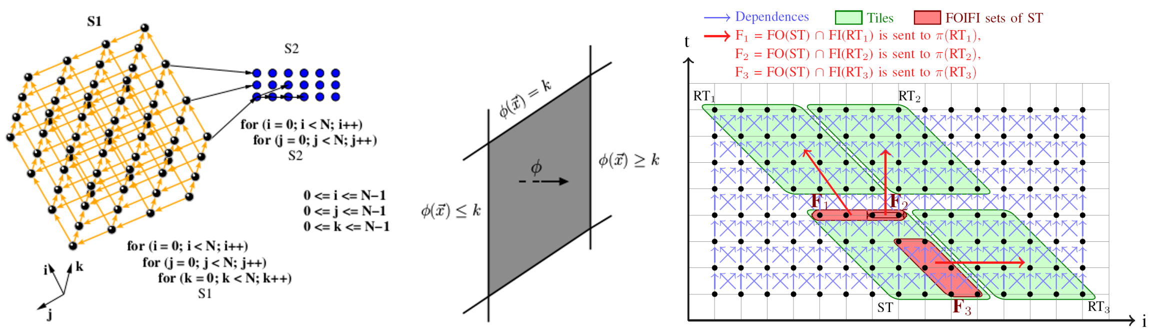


FIGURE 6.1: PLUTO overview.
Source: PLUTO's official website [187].

Many compute-intensive scientific applications spend most of their execution time running nested loops. The Polyhedral Model [53] provides a powerful mathematical abstraction to analyse and transform loop nests in which the data access functions and loop bounds are affine combinations (linear combinations with a constant) of the enclosing loop iterators and parameters. As shown in Figure 6.1, this model represents the instances of the loop nests' statements as integer points inside a polyhedron, where inter and intra-statement dependencies are characterised as a dependency polyhedron. Combining this representation with Linear Algebra and Integer Linear Programming, it is possible to reason about the correctness of a sequence of complex optimising and parallelising loop transformations.

PLUTO [187, 39] is an automatic parallelisation tool based on the Polyhedral Model to optimise arbitrarily nested loop sequences with affine dependencies. At compile time, it analyses C source code to optimise and parallelise affine loop-nests and automatically generate OpenMP C parallel code for multi-cores. Although the tool is fully automatic, many options are available to tune tile sizes, unroll factors, and outer loop fusion structure.

As shown in Figure 6.2, PLUTO internally translates the source code to an intermediate OpenScop [35] representation using CLAN [36]. Next, it relies on the Polyhedral Model to find affine transformations for coarse-grained parallelism, data locality, and efficient tiling. Finally, PLUTO generates the OpenMP C code from the OpenScop representation using CLooG [34]. We must highlight that the generated code is also optimised for data locality and made amenable to auto-vectorisation.

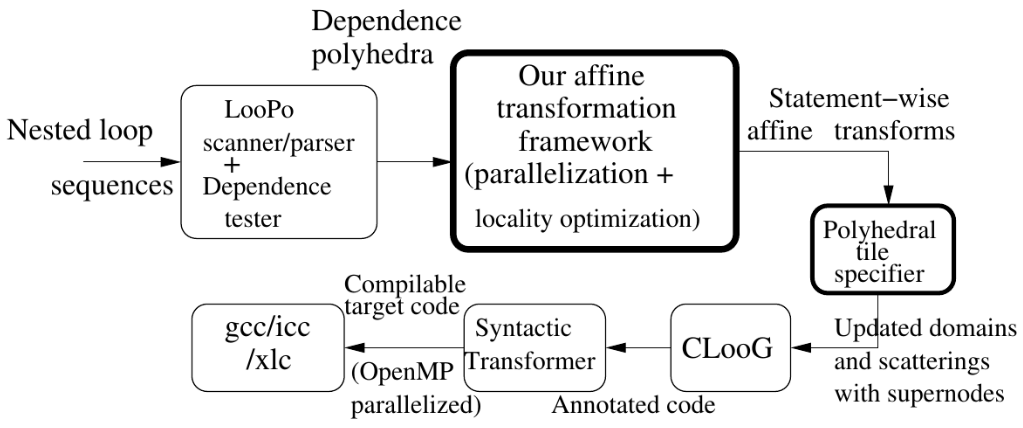

Dependence polyhedra
Nested loop sequences
LooPo scanner/parser + Dependence tester
Our affine transformation framework (parallelization + locality optimization)
Statement−wise affine transforms
Polyhedral tile specifier
Updated domains and scatterings with supernodes
CLooG
Syntactic Transformer
Annotated code
Compilable target code (OpenMP parallelized)
gcc/icc /xlc

FIGURE 6.2: PLUTO source-to-source transformation.
Source: [40].

6.3.1 Loop tiling

Among many other options, PLUTO can tile code by specifying the `--tile` option. In general terms, as shown in Listing 6.1, tiling a loop of given size `N` results in a division of the loop in `N/T` repeatable parts of size `T`. For instance, this is suitable when fitting loops into the L1 or L2 caches or, in the context of this thesis, when building the data blocks to increase the tasks' granularity.

Along with this option, users can let PLUTO set the tile sizes automatically using a rough heuristic, or manually define them in a `tile.sizes` file. This file must contain one tile size on each line and as many tile sizes as the loop nest depth.

In the context of parallel applications, tile sizes must be fine-tuned for each application so that they maximise locality while making sure there are enough tiles to keep all cores busy.

```
    # Original loop             # Tiled loop
    for i in range(N):          for i in range(N/T):
        print(i)                    for t in range(T):
                                        print(i*T + t)
```

LISTING 6.1: Tiling example.

6.4 Architecture

Our proposal eases eases the development of distributed applications by letting users program their application in a standard sequential fashion. It is developed on top of PyCOMPSs and PLUTO. When automatically parallelising sequential applications, users must only insert an annotation on top of the potentially parallel functions to activate the AutoParallel module. Next, the application can be launched using PyCOMPSs.

Following a similar approach than PyCOMPSs, we have included a new annotation (the Python decorator `@parallel`) to specify which methods should be automatically parallelised at runtime. Notice that, since PLUTO and the Polyhedral Model can only be applied to affine loops, the functions using this decorator must contain loop nests in which the data access functions and loop bounds are affine combinations (linear combinations with a constant) of the enclosing loop iterators and parameters. Otherwise, the source code will remain intact. Table 6.1 shows the valid flags for the decorator.

Flag	Default Value	Description
`pluto_extra_flags`	None	List of flags for the internal PLUTO command
`taskify_loop_level`	0	Taskification loop depth (see Section 6.4.2)
`force_autogen`	True	When set to False, loads a previously generated code
`generate_only`	False	When set to True, only generates the parallel version of the code

TABLE 6.1: List of flags for the `@parallel` decorator.

As shown in Figure 6.3, the AutoParallel Module analyses the user code searching for `@parallel` annotations. Essentially, when found, the module calls PLUTO to generate its parallelisation and substitutes the user code by a newly generated code. Once all annotations have been processed, the new tasks are registered into PyCOMPSs, and the execution continues as a regular PyCOMPSs application (as described in Section 3.1). Finally, when the application has ended, the generated code is stored in a (`_autogen.py`) file and the user code is restored.

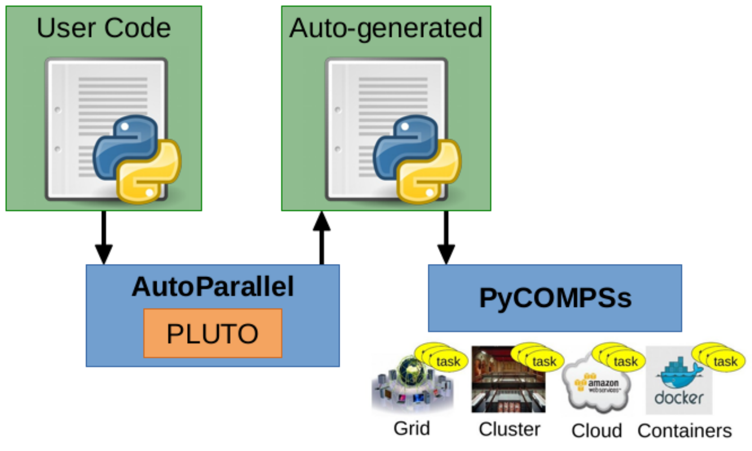


FIGURE 6.3: Overview of the AutoParallel Module.

6.4.1 AutoParallel module

The internals of the AutoParallel module are quite complex (more than 5.000 lines of code) because it automatically re-writes the python's AST representation of the user code at execution time, while interacting with C/C++ libraries in an intermediate format (OpenScop). This section only provides a first approach to AutoParallel by detailing its five main components. Complete code and documentation can be found in [193].

- **Decorator** Implements the `@parallel` decorator to detect functions that the user has marked as potentially parallel.
- **Python To OpenScop Translator** For each affine loop nest detected in the user function, builds a Python Scop object representing it that can be bulked into an OpenScop format file.
- **Paralleliser** Returns the Python code resulting from parallelising an OpenScop file. Since Python does not have any standard regarding parallel annotations, the parallel loops are annotated using comments with OpenMP syntax.
- **Python to PyCOMPSs Translator** Converts an annotated Python code into an application using PyCOMPSs by inserting the necessary task annotations and data synchronisations. This component can also build data blocks from loop tiles and taskify them if enabled by the user (see Section 6.4.2 for more details).
- **Code Replacer** Replaces each loop nest in the initial user code by the auto-generated code so that PyCOMPSs can execute the code in a distributed computing platform. When the application has finished, it restores the user code and saves the auto-generated code in a separated file.

Also, for the sake of clarity, Figure 6.4 shows the relationship between the different components and their expected inputs and outputs.

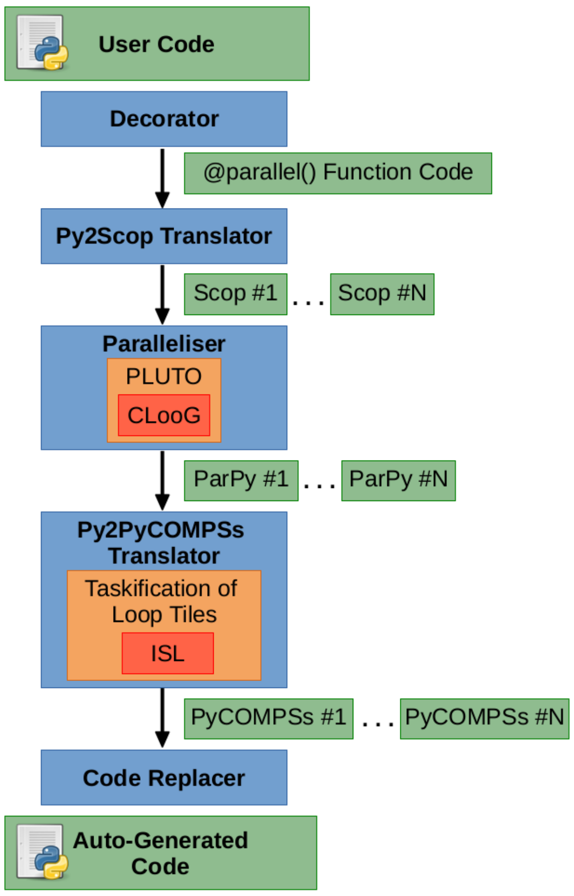


FIGURE 6.4: AutoParallel Module Internals.

For instance, Listing 6.2 shows the relevant parts of an Embarrassingly Parallel application with the main function annotated with the `@parallel` decorator that contains two nested loops.

```
# Main Function
from pycompss.api.parallel import parallel
@parallel()
def ep(mat, n_size, m_size, c1, c2):
    for i in range(n_size):
        for j in range(m_size):
            mat[i][j] = compute(mat[i][j], c1, c2)
```

LISTING 6.2: EP example: user code.

In addition, Listing 6.3 shows the parallelisation proposed by the AutoParallel module. On the one hand, the automatically generated source code contains the definition of a new task (*S1*) that includes the task decorator, annotations for its data dependencies (line 2), and the function code (line 4). Notice that the function code is automatically generated from the inner statement in the original loop (line 7 in Listing 6.2).

On the other hand, the generated source code contains the *ep* function with some modifications. First, AutoParallel introduces a new set of variables (e.g., *lbp*, *ubp*, *lbv*, *ubv*) to control the bounds of each loop. Also, the loop nest is modified following the PLUTO output to call the automatically generated tasks (line 14) and exploit the inherent parallelism available in the original code. For instance, in the example, the loop bounds have been interchanged (*n_size* and *m_size*). Finally, AutoParallel includes a barrier (line 15) used as synchronisation point at the end of the function code.

```
1  # [COMPSs Autoparallel] Begin Autogenerated code
2  @task(var2=IN, c1=IN, c2=IN, returns=1)
3  def S1(var2, c1, c2):
4      return compute(var2, c1, c2)
5
6  def ep(mat, n_size, m_size, c1, c2):
7      if m_size >= 1 and n_size >= 1:
8          lbp = 0
9          ubp = m_size - 1
10         for t1 in range(lbp, ubp + 1):
11             lbv = 0
12             ubv = n_size - 1
13             for t2 in range(lbv, ubv + 1):
14                 mat[t2][t1]=S1(mat[t2][t1],c1,c2)
15     compss_barrier()
16 # [COMPSs Autoparallel] End Autogenerated code
```

LISTING 6.3: EP example: auto-generated code.

6.4.2 Taskification of loop tiles

Many compute-intensive scientific applications are not designed as block computations, and thus, the tasks proposed by the AutoParallel module are single statements. Although this can be harmless in tiny parallel environments, it leads to poor performance when executed using large distributed environments since the tasks' granularity is not large enough to surpass the overhead of transferring the task definition, and the input and output data. To face this issue, we have extended the *Python to PyCOMPSs Translator* to automatically build data blocks from loop tiles and taskify them. As shown in Listing 6.4, the users can enable this behaviour by providing the `tile=True` option to the `@parallel` decorator. The tile sizes are computed automatically by PLUTO but the users can provide an extra file named `tile.sizes` in the application's root directory to fine-tune them.

```
from pycompss.api.parallel import parallel
@parallel(tile=True)
def ep(mat, n_size, m_size, c1, c2):
    for i in range(n_size):
        for j in range(m_size):
            mat[i][j] = compute(mat[i][j], c1, c2)
```

LISTING 6.4: EP example: user code with taskification of loop tiles.

Essentially, the taskification of loop tiles means letting PLUTO process the parallel code by generating tiles, and extract the loop tiles into tasks. As explained in Section 6.4.2, the tiled loops generated by PLUTO duplicate the depth of the original loop nest while decreasing the number of iterations per loop. Hence, extracting all the tiles and converting them into tasks maintains the original loop depth, decreases the number of iterations per loop, and increases the task granularity.

Since tasks may use N-dimensional arrays, the taskification of loop tiles also implies to create the necessary data blocks (chunks) for each parameter before the task callee. The automatically generated code builds the data blocks on the main code and passes them to the tasks using the PyCOMPSs Collection parameter annotation. This annotation ensures that all the internal objects of the chunks are registered so that all the parameter dependencies are respected. Following with the previous example, Listing 6.5 shows the automatically generated code with taskification of loop tiles.

```python
# [COMPSs Autoparallel] Begin Autogenerated code
@task(t2=IN, m_size=IN, t1=IN, n_size=IN, coef1=IN, coef2=IN,
      mat={Type: COLLECTION_INOUT, Depth: 2})
def LT2(t2, m_size, t1, n_size, coef1, coef2, mat):
  for t3 in range(32*t2, min(m_size, 32*t2 + 32)):
    lbv = 2*t1
    ubv = min(n_size - 1, 2*t1 + 1)
    for t4 in range(lbv, ubv + 1):
      mat[t4 - 2*t1][t3 - 32*t2] = S1_no_task(mat[t4 - 2*t1][t3 - 32*t2], coef1, coef2)

def S1_no_task(var2, coef1, coef2):
  return compute(var2, coef1, coef2)

def ep(mat, n_size, m_size, coef1, coef2):
  if m_size >= 1 and n_size >= 1:
    lbp = 0
    ubp = int(math.floor(float(n_size - 1)/float(2)))
    for t1 in range(lbp, ubp + 1):
      lbp = 0
      ubp = int(math.floor(float(m_size - 1)/float(32)))
      for t2 in range(lbp, ubp + 1):
        lbp = 32 * t2
        ubp = min(m_size - 1, 32*t2 + 31)
        # Chunk creation
        LT2_aux_0 = [[mat[gv0][gv1] for gv1 in ...] for gv0 in ...]
        # Task call
        LT2(t2, m_size, t1, n_size, coef1, coef2, LT2_aux_0)
  compss_barrier()
# [COMPSs Autoparallel] End Autogenerated code
```

LISTING 6.5: EP example: auto-generated code with taskification of loop tiles.

Notice that the generated code with taskification of loop tiles is significantly more complex. In the example, the original loop nest has depth 2, the tile size for the `t1` loop is set to 2, and the tile size for the `t2` loop is set to 32. Also, readers may identify the `t3` loop as the tile of `t2` loop, and the `t4` loop as the tile of the `t1` loop (indexes have been swapped automatically due to data locality).

Regarding the tasks, `LT2` contains a loop nest of depth 2 with 2 iterations for the `t4` loop, and 32 iterations for the `t3` loop. Furthermore, each N-dimensional array used as a parameter is annotated as a Collection with its direction (IN or INOUT) and depth (number of dimensions). In the example, `mat` is annotated as a `COLLECTION_INOUT` of depth 2. Moreover, inside the task code, the array accesses are modified accordingly to the received data chunks.

Regarding the main code, the original loops are modified considering the tiles' decoupling. In the example, the number of iterations of the `t1` loop is divided by 2, and the number of iterations of the `t2` loop is divided by 32. Furthermore, the data chunks are built before the task callee by only copying the original object references. In the example, we build the `LT2_aux_0` chunk from `mat` and update the callee parameters accordingly. Finally, similarly to the previous cases, the end of the main code also includes a barrier as a synchronisation point.

6.4.3 Python extension for CLooG

As described in Section 6.3, PLUTO operates internally with the OpenScop format. It relies on Clan to translate input code from C/C++ or Fortran to OpenScop, and on CLooG to translate output code from OpenScop to C/C++ or Fortran.

Since we are targeting Python code, a translation from Python to OpenScop is required before calling PLUTO, and another translation from OpenScop to Python is required at the end. Regarding the input, we have developed the *Python To OpenScop Translator* component inside AutoParallel to manually translate the code because Clan is not adapted for supporting additional languages and PLUTO accepts OpenScop codes as input. Regarding the output, we have extended CLooG so that the written code is directly in Python. Hence, we have extended the language options and modified the Pretty Printer in order to translate every OpenScop statement into its equivalent Python format. Since Python does not have any standard regarding parallel annotations, the parallel loops are annotated with comments in OpenMP syntax.

6.5 Programmability evaluation

Considering an idealistic environment, the developer's productivity can be expressed as the relation between the effort to write the code of the application and the performance obtained by such code. We are aware that many other factors such as the physical working environment, adequate development frameworks and tools, meeting times, code reviews, burndowns, etc. can affect the developer's productivity, but we only consider elements directly related with the code. Furthermore, these terms can be considered constants or eventualities when comparing the productivity difference for the same developer when using (or not) the AutoParallel module.

In this section, we demonstrate that our approach improves the developer's productivity significantly by easing the coding of the application. On the other hand, next sections (Section 6.6 and 6.7) focus on evaluating and comparing the performance of the automatically generated codes. Hence, the application presented in this section highlights the benefits at the programming model level of using AutoParallel without focusing on performance. Also, we consider an application that calculates the centre of mass of a given system but the outcomes can be applied to any application containing affine loop nests.

6.5.1 Centre of Mass

The following application calculates the centre of mass of a given system. The *system* itself is composed of *objects* that are composed of *parts* in a certain *position* of the space. Also, each part has a predefined *mass*. Equations 6.1 describe how to compute the centre of mass of the system (CM) by first calculating the centre of mass and the aggregated mass of each object and, next, computing the centre of mass of the whole system.

$$\begin{gathered} mass_{obj} = \sum_{j=0}^{num\_parts} mass_j \ , \ cm_{obj} = \frac{\sum_{j=0}^{num\_parts} mass_{obj,j} \cdot position_j}{\sum_{j=0}^{num\_parts} mass_j} \\ CM = \frac{\sum_{i=0}^{num\_objs} mass_i \cdot cm_i}{\sum_{i=0}^{num\_objs} mass_i} \end{gathered} \tag{6.1}$$

Listing 6.6 provides the sequential code to calculate the centre of mass following Equation 6.1. The first loop nest (lines 9 to 21 in the figure) calculates the numerator and the denominator of cm_{obj} so that the second loop nest (lines 28 to 33) can compute the centre of mass of each object in the system. Then, the third loop nest (lines 36 to 37) computes the centre of mass of the whole system.

Given the sequential code, the users can add the `@parallel` decorator to automatically taskify and run the application in a distributed environment. Listing 6.7 shows that the

```
 1  def calculate_cm(num_objs, num_parts, num_dims, objs, masses):
 2      import numpy as np
 3
 4      # Initialize object results
 5      objs_cms = [[np.float(0) for _ in range(num_dims)] for _ in range(num_objs)]
 6      objs_mass = [np.float(0) for _ in range(num_objs)]
 7
 8      # Calculate CM and mass of every object
 9      for obj_i in range(num_objs):
10          # Calculate object mass and cm position
11          for part_i in range(num_parts):
12              # Update total object mass
13              objs_mass[obj_i] += masses[objs[obj_i][part_i][0]]
14              # Update total object mass position
15              for dim in range(num_dims):
16                  objs_cms[obj_i][dim] += masses[objs[obj_i][part_i][0]]
17                      * objs[obj_i][part_i][1][dim]
18          # Store final object CM and mass
19          for dim in range(num_dims):
20              objs_cms[obj_i][dim] /= objs_mass[obj_i] if objs_mass[obj_i] != np.float(0)
21                  else np.float(0)
22
23      # Initialize system results
24      system_mass = np.float(0)
25      system_cm = [np.float(0) for _ in range(num_dims)]
26
27      # Calculate system CM for every object
28      for obj_i in range(num_objs):
29          # Update total mass
30          system_mass += objs_mass[obj_i]
31          # Update system mass position
32          for dim in range(num_dims):
33              system_cm[dim] += objs_mass[obj_i] * objs_cms[obj_i][dim]
34
35      # Calculate system CM
36      for dim in range(num_dims):
37          system_cm[dim] /= system_mass if system_mass != np.float(0) else np.float(0)
38
39      return system_cm
```

LISTING 6.6: Centre of mass: sequential code.

```
from pycompss.api.parallel import parallel

@parallel()
def calculate_cm(num_objs, num_parts, num_dims, objs, masses):
    # Exactly the same code than the original
    ...

    return system_cm
```

LISTING 6.7: Centre of mass: AutoParallel annotations.

AutoParallel module can be enabled by just adding 2 lines (the import and the decorator) on top of the method declaration.

Next, Table 6.2 compares the user code and the automatically generated code in terms of annotations and loop configuration.

On the one hand, although PyCOMPSs' annotations are quite simple compared to other programming models (such as MPI), the automatically generated code contains 6 different task definitions with 15 parameter annotations. More in-depth, Listing 6.8 details each task definition, where *S1* corresponds to the statement in line 13 of the sequential code (Listing 6.6), *S2* to line 16, *S3* to line 20, *S4* to line 30, *S5* to line 33, and *S6* to line 37. Furthermore, notice that AutoParallel creates a new task per statement in the original loop nest, even if

Version	Code Analysis			Loops Analysis		
	Annotations		API Calls	Main	Total	Max Depth
	Method	Param.				
autoparallel (user code)	1	0	0	3	7	3
autoparallel (generated)	6	15	3	14	29	3

TABLE 6.2: Centre of mass: code and loop analysis.

```python
from pycompss.api.task import task
from pycompss.api.parameter import *

@task(var2=IN, var1=INOUT)
def S1(var2, var1):
    var1 += var2

@task(var2=IN, var3=IN, var1=INOUT)
def S2(var2, var3, var1):
    var1 += var2 * var3

@task(var2=IN, var3=IN, var1=INOUT)
def S3(var2, var3, var1):
    var1 /= var3 if var2 != np.float(0)  else np.float(0)

@task(var1=IN, system_mass=INOUT)
def S4(var1, system_mass):
    system_mass += var1

@task(var2=IN, var3=IN, var1=INOUT)
def S5(var2, var3, var1):
    var1 += var2 * var3

@task(system_mass=IN, var1=INOUT)
def S6(system_mass, var1):
    var1 /= system_mass if system_mass != np.float(0) else np.float(0)
```

LISTING 6.8: Centre of mass: Automatically generated tasks.

the internal operation is the same at the end. Hence, a user manually parallelising and taskifying the previous sequential code will obtain a similar solution but using 3 tasks instead of 6 since *S1* and *S4*, *S2* and *S5*, and *S3* and *S6* can be unified.

On the other hand, AutoParallel re-writes the loop nests in order to exploit data locality and perform some optimisations depending on the input values. Hence, the 3 original loop nests have been split into 14 loop nests containing a total of 29 for loops. However, notice that the maximum depth (3) is preserved to ensure that the complexity remains the same. More in-depth, Listing 6.9 shows the automatically generated code, including the task and data synchronisation calls. Due to space constraints, we only include the generated code for the second and third loop nests (lines 28 to 33 and 36 to 37 in the sequential code on Listing 6.6) since the first one is more complex and has been divided into 10 loop nests.

Regarding the second loop nest, *S5* computes the nominator of the system's centre of mass and *S4* the denominator (total mass of the system). Also, AutoParallel has split the main loop into 3 loops (lines 21, 33, and 37 in Listing 6.9) considering different input values and iterating in a different way over the loop space to exploit data locality.

Regarding the third loop nest, although AutoParallel has automatically added the *lbp* and *ubp* variables to control the loop bounds, the loop structure is kept the same. Furthermore, the original statement is substituted by a task call to *S6*.

AutoParallel automatically adds a `compss_barrier()` to avoid possible data collisions (lines 42 and 50 in Listing 6.9) after the code corresponding to each original loop nest. Also,

```
1   from pycompss.api.api import compss_wait_on
2
3   def calculate_cm(num_objs, num_parts, num_dims, objs, masses):
4       import numpy as np
5
6       # Initialize object results
7       objs_cms = [[np.float(0) for _ in range(num_dims)] for _ in range(num_objs)]
8       objs_mass = [np.float(0) for _ in range(num_objs)]
9
10      # Calculate CM and mass of every object
11      ...
12
13      # Calculate system CM for every object
14      system_mass = np.float(0)
15      system_cm = [np.float(0) for _ in range(num_dims)]
16
17      if num_objects >= 1:
18          if num_objects >= 2:
19              lbp = 0
20              ubp = min(num_dims - 1, num_objects - 1)
21              for t1 in range(lbp, ubp + 1):
22                  S4(objs_mass[t1], system_mass)
23                  S5(objs_mass[0], objs_cms[0][t1], system_cm[t1])
24                  lbp = 1
25                  ubp = num_objects - 1
26                  for t2 in range(1, num_objects - 1 + 1):
27                      S5(objs_mass[t2], objs_cms[t2][t1],  system_cm[t1])
28          if num_dims >= 1 and num_objects == 1:
29              S4(objs_mass[0], system_mass)
30              S5(objs_mass[0], objs_cms[0][0], system_cm[0])
31          lbp = max(0, num_dims)
32          ubp = num_objects - 1
33          for t1 in range(lbp, ubp + 1):
34              S4(objs_mass[t1], system_mass)
35          lbp = num_objects
36          ubp = num_dims - 1
37          for t1 in range(lbp, ubp + 1):
38              lbp = 0
39              ubp = num_objects - 1
40              for t2 in range(0, num_objects - 1 + 1):
41                  S5(objs_mass[t2], objs_cms[t2][t1],system_cm[t1])
42      compss_barrier()
43
44      # Calculate system CM
45      if num_dims >= 1:
46          lbp = 0
47          ubp = num_dims - 1
48          for t1 in range(lbp, ubp + 1):
49              S6(system_mass, system_cm[t1])
50      compss_barrier()
51
52      system_cm = compss_wait_on(system_cm)
53
54      return system_cm
```

LISTING 6.9: Centre of mass: Automatically generated loop nest.

there is a synchronisation of the `system_cm` variable (line 52) before the return of the function so that PyCOMPSs synchronises and transfers its final value.

To conclude, AutoParallel is capable of automatically taskifying a sequential code and re-order the loop nests to exploit data locality by just adding one single annotation. For the centre of mass application, AutoParallel frees the users from defining 6 tasks, annotating 15 parameters, building each task call and re-ordering the *objects*, *parts*, and *dimensions* loops to exploit data locality. Also, notice that PyCOMPSs already provides a simple programming model compared to many other frameworks such as MPI. In those cases, AutoParallel frees the users from explicitly defining the code for each process, handling data transfers, and synchronising the different processes. Finally, we must highlight that the centre of mass application is just an example and that the results shown in this section can be extrapolated to any other application containing affine loop nests.

6.6 Performance evaluation

This section demonstrates that our approach eases the coding of the application using one single Python decorator on top of sequential code, while obtaining similar performances than manually parallelised codes. Next subsections evaluate the code complexity and the performance for established algorithms when using AutoParallel or PyCOMPSs. We evaluate the Cholesky, LU, and QR decompositions in order to demonstrate that AutoParallel is capable of automatically generating code as efficient as solutions that have been a reference in state of the art for more than ten years, but requiring much less effort to write them.

It is worth highlighting that AutoParallel is an additional module to generate PyCOMPSs annotated applications automatically but, at execution time, the generated code acts as a regular PyCOMPSs application. Hence, the execution performance is strongly related to the PyCOMPSs performance. This section reports the performance evaluation of several applications in comparison with their implementations using only PyCOMPSs to evaluate only the overhead introduced by AutoParallel. Therefore, the performance evaluation of PyCOMPSs and its Runtime is beyond the scope of this section. For further details, in our previous work [7] and [8], we analysed in-depth the performance obtained when executing linear algebra applications when combining PyCOMPSs for inter-node parallelism and NumPy for intra-node parallelism. Also, in [56], we compared the PyCOMPSs Runtime against Apache Spark.

6.6.1 Computing infrastructure

The results presented in this section have been obtained using the MareNostrum 4 supercomputer [149] located at the Barcelona Supercomputing Center (BSC). This supercomputer begun operating at the end of June 2017. Its current peak performance is 11.15 Petaflops, ten times more than its previous version, MareNostrum 3. The supercomputer is composed by 3456 nodes, each of them with two Intel® Xeon Platinum 8160 (24 cores at 2,1 GHz each). It has 384.75 TB of main memory, 100Gb Intel®Omni-Path Full-Fat Tree Interconnection, and 14 PB of disk storage.

We have used PyCOMPSs version 2.3.rc1807 (available at [55]), PLUTO version 0.11.4, CLooG version 0.19.0, and AutoParallel version 0.2 (available at [193]). We have also used Intel®Python 2.7.13, Intel®MKL 2017, Java OpenJDK 8 131, GCC 7.2.0, and Boost 1.64.0.

All the benchmark codes used for this experimentation are also available at [82].

6.6.2 General description of the applications

In general terms, the matrices are chunked in smaller square matrices (known as *blocks*) to distribute the data easily among the available resources so that the square blocks are the minimum entity to work with [103]. Furthermore, the initialisation is performed in a distributed way, defining tasks to initialise the matrix blocks. These tasks do not take into account the nature of the algorithm, and they are scheduled in a round robin manner. For all the evaluation applications, the execution time measures the application's computations and the data transfers required during the execution by the framework, but does not include the initial transfers of the input data.

Given a fixed matrix size, increasing the number of blocks increases the maximum parallelism of the application since blocks are the tasks' minimum work entities. On the other hand, increasing the block size increases the tasks' computational load, which, at some point, will surpass the serialisation and transfer overheads. Hence, the number of blocks and the block size for each application are a trade-off to fill all the available cores while maintaining acceptable performance.

For all the evaluated applications, we compare the code written by a PyCOMPSs expert user (*userparallel* version) against the sequential code with the `@parallel` annotation (*autoparallel* user code) and the automatically generated code by the AutoParallel module (*autoparallel* generated). Furthermore, we provide a figure showing the execution results for each application. The figure contains two plots where the horizontal axis shows the number of worker nodes (with 48 cores each) used for each execution, the blue colour is the *userparallel* version, and the green colour is the *autoparallel*. The top plot represents the mean, maximum, and minimum execution times over 10 runs and the bottom plot represents the speed-up of each version with respect to the *userparallel* version running with a single worker (48 cores).

6.6.3 Cholesky

The Cholesky factorisation can be applied to Hermitian positive-defined matrices. This decomposition is a particular case of the LU factorisation, obtaining two matrices of the form $U = L^t$. Our version of this application applies the right-looking algorithm [37] because it is more aggressive, meaning that in an early stage of the computation there are blocks of the solution that are already computed and all the potential parallelism is released as soon as possible.

Version	**Code Analysis**			**Loops Analysis**		
	Annotations		**API Calls**	**Main**	**Total**	**Max Depth**
	Method	**Param.**				
userparallel	3	14	0	1	4	3
autoparallel (user code)	1	0	0	1	4	3
autoparallel (generated)	4	11	1	3	9	3

TABLE 6.3: Cholesky: code and loop analysis.

Table 6.3 analyses the *userparallel*, the *autoparallel*'s original user code, and the *autoparallel*'s automatically generated code in terms of code and loop configuration. While the *userparallel* version requires the definition of three tasks (`potrf`, `solve_triangular`, and `gemm`) using 14 parameter annotations, the *autoparallel*'s original code only requires a single `@parallel` decorator. On the other hand, the *autoparallel*'s automatically generated code includes four tasks; 3 equivalent to the *userparallel* tasks and an additional one to generate blocks initialised to zero.

Regarding the loop configuration, the *userparallel* and the *autoparallel*'s original code have the same structure. However, the *autoparallel*'s automatically generated code has divided the original loop into three main loops maintaining the maximum loop depth (three).

Figure 6.5 shows the execution results of the Cholesky decomposition over a dense matrix of $65,536 \times 65,536$ elements decomposed in 32×32 blocks with $2,048 \times 2,048$ elements each. As stated in the general description of this section, the top plot represents the execution time while the bottom plot represents the speed-up with respect to the *userparallel* version running with a single worker (48 cores). Also, we have chosen 32 blocks because it is the minimum amount providing enough parallelism for 192 cores, and bigger block sizes (e.g., $4,096 \times 4,096$) were impossible due to memory constraints. The speed-up of both versions is limited by the block-size due to the small task granularity, reaching 2 when using 4 workers. Although the *userparallel* version spawns 6,512 tasks and the *autoparallel* version spawns 7,008 tasks, the execution times and the overall performance of both versions are

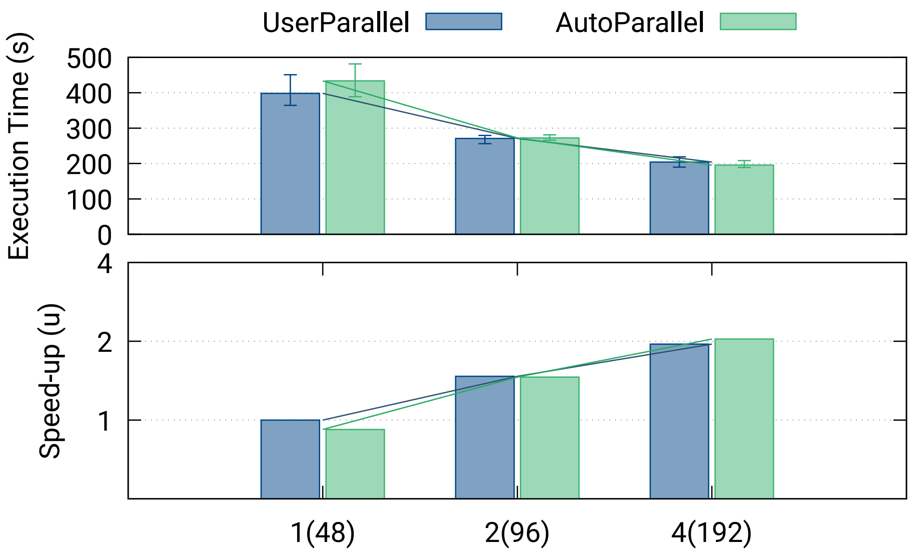


FIGURE 6.5: Cholesky: Execution times and speed-up with respect to the *user-parallel* version using a single worker (48 cores).

almost the same (the difference is less than 5%). This is due to the fact that the *autoparallel* version spawns an extra task per iteration to initialise blocks to zero on the matrix's lower triangle that has no impact on the overall computation time.

6.6.4 LU

For the LU decomposition, an approach without pivoting [100] has been the starting point. However, since this approach might be unstable in general [69], some modifications have been included to increase the stability of the algorithm while keeping the block division and avoiding bringing an entire column into a single node.

Version	**Code Analysis**			**Loops Analysis**		
	Annotations		**API**	**Main**	**Total**	**Max**
	Method	**Param.**	**Calls**			**Depth**
userparallel	4	13	0	2	6	3
autoparallel (user code)	1	0	0	2	6	3
autoparallel (generated)	12	33	3	4	9	3

TABLE 6.4: LU: code and loop analysis.

Table 6.4 analyses the *userparallel*, the *autoparallel*'s original user code, and the *autoparallel*'s automatically generated code in terms of code and loop configuration. Regarding the code, the *userparallel* version requires the definition of 4 tasks (namely `multiply`, `invert_triangular`, `dgemm`, and `custom_lu`) along with 13 annotated parameters. In contrast, the *autoparallel*'s original user code only requires the `@parallel` annotation. Also, the *autoparallel*'s automatically generated code generates 12 different task types (along with 33 annotated parameters) because it generates one task type per statement in the original loop, even if the statement contains the same task call. For instance, the original LU contains four calls to the `invert_triangular` function that are detected as different statements and converted to different task types.

Regarding the loop configuration, both the *userparallel* and the *autoparallel*'s original user code have the same structure: 2 loop nests of depth 3. However, the *autoparallel*'s automatically generated code splits them into 4 main loops of the same depth because it has different optimisation codes for different variable values.

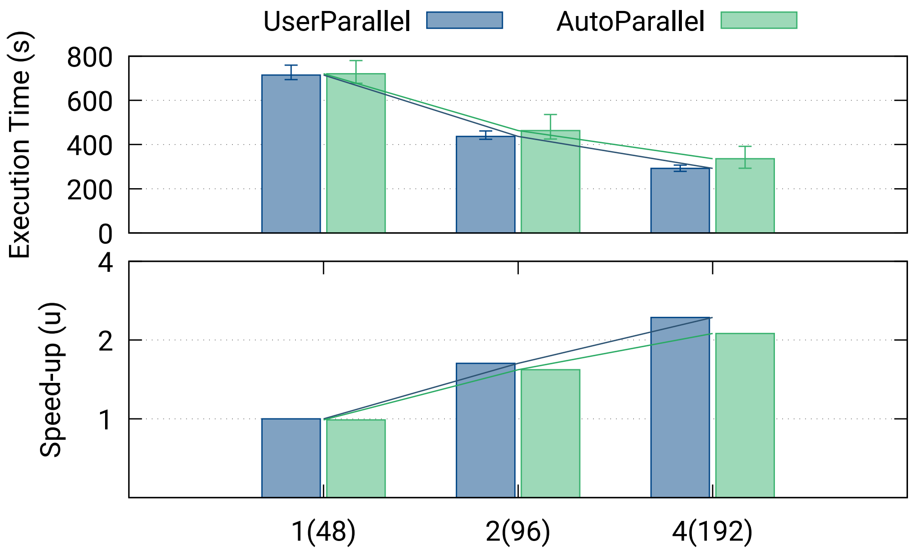


FIGURE 6.6: LU: Execution times and speed-up with respect to the *userparallel* version using a single worker (48 cores).

Figure 6.6 shows the execution results of the LU decomposition with a $49,152 \times 49,152$ dense matrix of 24×24 blocks with $2,048 \times 2,048$ elements each. As stated in the general description of this section, the top plot represents the execution time while the bottom plot represents the speed-up with respect to the *userparallel* version running with a single worker (48 cores). As in the previous example, the overall performance is limited by the block size. This time the *userparallel* version slightly outperforms the *autoparallel* version; achieving, respectively, a 2.45 and 2.13 speed-up with 4 workers (192 cores).

Regarding the number of tasks, the *userparallel* version spawns 14,676 tasks while the *autoparallel* version spawns 15,227 tasks. This difference is due to the fact that the *autoparallel* version initialises distributedly an intermediate zero matrix, while the *userparallel* initialises it in the master memory.

Figure 6.7 shows a detailed Paraver trace of both versions running with 4 workers (192 cores) using the same time scale. The *userparallel* version is shown on top while the *autoparallel* version is shown at the bottom. The *autoparallel* version (bottom) is more coloured because it has more tasks, although, as previously explained, they execute the same function in the end. Notice that the performance degradation of the *autoparallel* version is due to the fact that the maximum parallelism is lost before the end of the execution. On the contrary, the *userparallel* version maintains the maximum parallelism until the end of the execution.

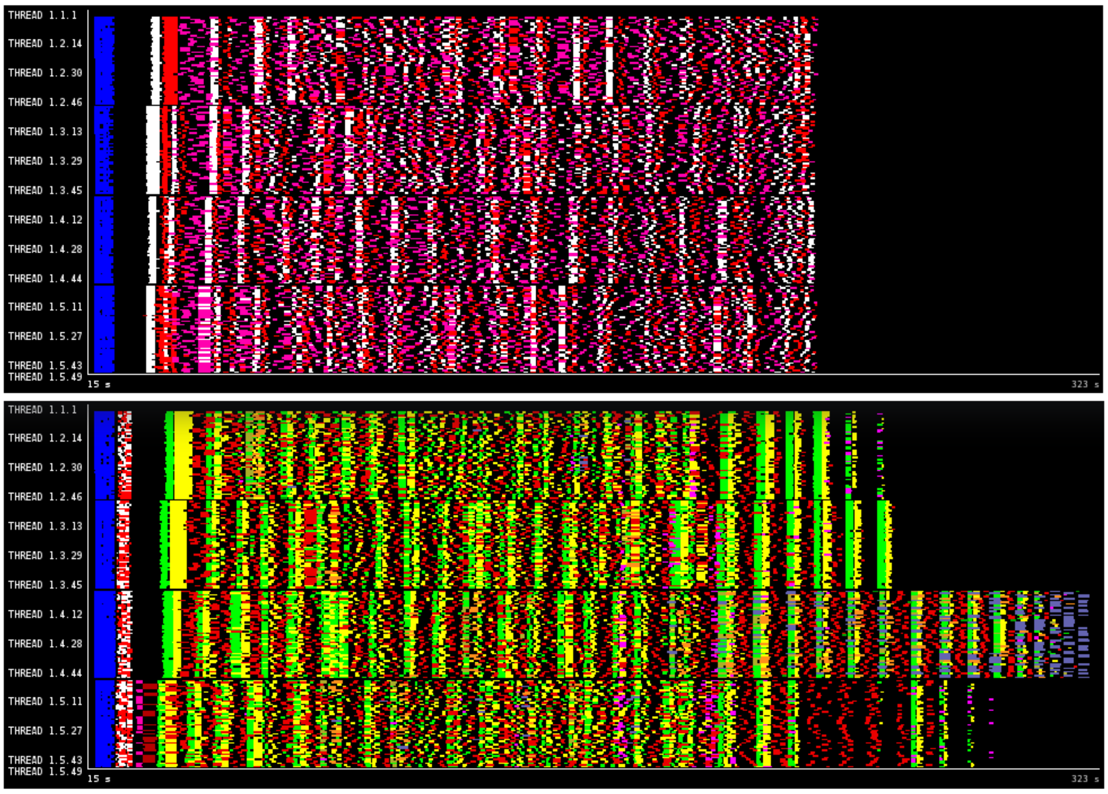


FIGURE 6.7: LU: Paraver trace. At the top, the *userparallel* and, at the bottom, the *autoparallel* version.

6.6.5 QR

Unlike traditional QR algorithms that use the Householder transformation, our implementation uses a method based on Givens rotations [197]. This way, data can be accessed by blocks instead of columns.

Table 6.5 analyses the *userparallel*, the *autoparallel*'s original user code, and the *autoparallel*'s automatically generated code in terms of code and loop configuration. The QR decomposition represents one of the most complex use cases in terms of data dependencies; thus, having more tasks and parameter annotations than the previous applications. While the *userparallel* requires 4 tasks (namely `qr`, `dot`, `little_qr`, and `multiply_single_block`) along with 19 parameter annotations, the *autoparallel*'s original user code only requires, as always, one single `@parallel` annotation. In contrast, the *autoparallel*'s automatically generated code defines 20 tasks along with 60 parameter annotations. As in the LU decomposition, many of these tasks are generated from different statements that perform the same task call at the end.

Regarding the loop configuration, both the *userparallel* and the*autoparallel*'s original user

Version	Code Analysis			Loops Analysis		
	Annotations		API	Main	Total	Max
	Method	Param.	Calls			Depth
userparallel	4	19	0	1	6	3
autoparallel (user code)	1	0	0	1	6	3
autoparallel (generated)	20	60	1	2	7	3

TABLE 6.5: QR: code and loop analysis.

code have the same structure: 1 main loop nest with a total of 6 loops and a maximum depth of 3. However, the *autoparallel*'s automatically generated code splits the main loop in two in order to increase the data locality. Notice that no additional complexity is added to the algorithm since the maximum depth remains 3.

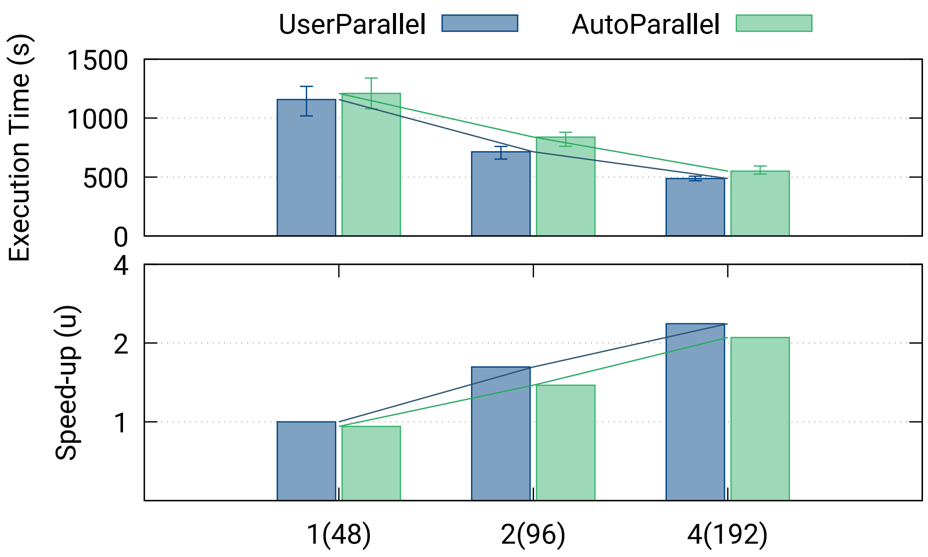


FIGURE 6.8: QR: Execution times and speed-up with respect to the *userparallel* version using a single worker (48 cores).

Figure 6.8 shows the execution results of the QR decomposition with a $32,768 \times 32,768$ matrix of 16×16 blocks with $2,048 \times 2,048$ elements each. As stated in the general description of this section, the top plot represents the execution time while the bottom plot represents the speed-up with respect to the *userparallel* version running with a single worker (48 cores). The *autoparallel* version spawns 26,304 tasks, and the *userparallel* version spawns 19,984 tasks. As in the previous examples, the overall performance is limited by the block size. However, the *userparallel* version slightly outperforms the *autoparallel* version; achieving a 2.37 speed-up with 4 workers instead of 2.10. The difference is mainly because the *autoparallel* version spawns four copy tasks per iteration (`copy_reference`), while the *userparallel* version executes this code in the master side copying only the reference of a future object.

6.7 Evaluation of the automatic data blocking

This section evaluates the capability of automatically generating data blocks (chunks) from pure sequential code and executing them in a distributed infrastructure. As discussed next, this approach provides several advantages in terms of code re-organisation and data blocking; increasing the tasks' granularity and, thus, the performance of fine-grain applications.

6.7.1 GEMM

We have implemented a Python version of the General Matrix-Matrix product (GEMM) from the Polyhedral Benchmark suite [188]. The implementation considers general rectangular matrices with float complex elements and performs $C = \alpha \cdot A \cdot B + \beta \cdot C$. In general terms, the arrays and matrices are implemented as plain NumPy arrays or matrices. This means that there are no *blocks*, and thus, the minimum work entity is a single element (a float). As in the previous set of experiments, the initialisation is performed in a distributed way; defining tasks to initialise the matrix elements. Also, the execution time measures the application's computations and the data transfers required during the execution by the framework, but does not include the initial transfers of the input data.

Version	**Code Analysis**			**Loops Analysis**		
	Annotations		**API Calls**	**Main**	**Total**	**Max Depth**
	Method	**Param.**				
userparallel	2	8	0	1	4	3
autoparallel (user code)	1	0	0	1	4	3
autoparallel FG (generated)	2	8	1	2	5	3
autoparallel LT (generated)	4	21	1	2	5	3

TABLE 6.6: GEMM: code and loop analysis.

Table 6.6 analyses the *userparallel*, the *autoparallel*'s original user code, the *autoparallel*'s automatically generated code using fine-grain (*autoparallel FG*), and the *autoparallel*'s automatically generated code with taskification of loop tiles (*autoparallel LT*) in terms of code and loop configuration. As expected, the *autoparallel*'s original user code only requires the `@parallel` annotation. Although the *autoparallel FG* works with single elements and *userparallel* with blocks, both versions include 2 tasks (namely `scale`, and `multiply`) with 8 parameter annotations. On the contrary, the *autoparallel LT* version defines 4 tasks (the two original ones and their two loop-tasked versions) with 21 parameter annotations. The original tasks are kept because, in configurations that do not use PLUTO's tiles, it is possible to find function calls that cannot be loop-taskified. However, in this case, only the loop-tasked versions are called during the execution.

Regarding the loop structure, both the *userparallel* and the *autoparallel*'s original user code have 1 main loop of maximum depth 3. Also, the two automatically generated codes are capable of splitting the main loop into two loops for better parallelism: one for the scaling operations and the other for the multiplications. However, the *autoparallel LT* code is significantly more complex (in terms of lines of code, cyclomatic complexity, and n-path) due to the tiling and chunk creation.

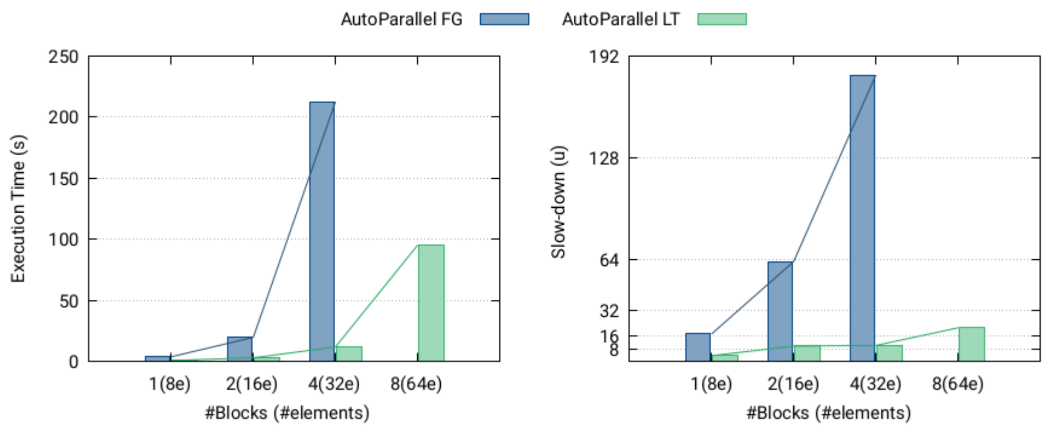


FIGURE 6.9: GEMM: Execution time and slow-down with respect to the blocked *userparallel* version using 1 worker (48 cores) and same matrix size.

Figure 6.9 shows the execution results of the GEMM application with one single worker (48 cores) and with matrices of 8, 16, 32, and 64 elements. To have equivalent executions, the tile sizes are set to 8 for the *autoparallel LT*, and the block size is set to 8 for the *userparallel*. The left plot shows the execution time of the *autoparallel FG* (blue) and the *autoparallel LT* (green). The right plot shows the slow-down of both versions with respect to the blocked *userparallel* version using a single worker (48 cores) and the same matrix size.

The automatic parallelisation without taskification of loop tiles (*autoparallel FG*) behaves 17.30 and 179.94 times slower than the blocked version (*userparallel B*) using 8 and 32 elements, respectively. In contrast, the *autoparallel LT* behaves 10.15 times slower than the *userparallel B* when using 32 elements; which improves by 17.73 the performance of the fine-grain version. This experiment highlights the importance of blocking fine-grain applications since defining single elements as the minimum task entity leads to tasks with too little computation that cause a massive overhead of task management, object serialisation and, data transfer inside PyCOMPSs.

Due to this same reason and as shown in the previous Section 6.6, the appropriate block sizes to obtain reasonable performances should be between $2,048 \times 2,048$ and $8,192 \times 8,192$ elements per block. Although AutoParallel can generate codes using bigger tile sizes, PyCOMPSs suffers from serialisation issues since each element inside the collection is treated as a separated task parameter. In contrast, the hand-made blocked version (*userparallel*) serialises all the elements of the block into a single object parameter and thus, can run with bigger block sizes. We are confident that PyCOMPSs could lower the serialisation overhead by serialising each element of the collection in parallel or serialising the whole collection into a single object.

Nonetheless, we believe that the automatic taskification of loop tiles is a good baseline to obtain blocked algorithms since the only difference between the automatically generated code with taskification of loop tiles and the hand-made blocked version is the treatment of data chunks. More specifically, the *userparallel* version uses single objects per chunk instead of collections of objects. Notice that AutoParallel cannot systematically annotate data chunks as objects since this is only possible when the data chunks are disjoint because, otherwise, the dependencies of each element inside the blocks need to be treated separately. However, advanced users can analyse the automatically generated data chunks, determine if they are disjoint, and use the automatically generated code as a baseline to change the annotations of the data chunks from `COLLECTION` to `OBJECT`. In this last scenario, advanced users only require to modify annotation of the data chunks; keeping the main code of the algorithm and the code of the tasks.

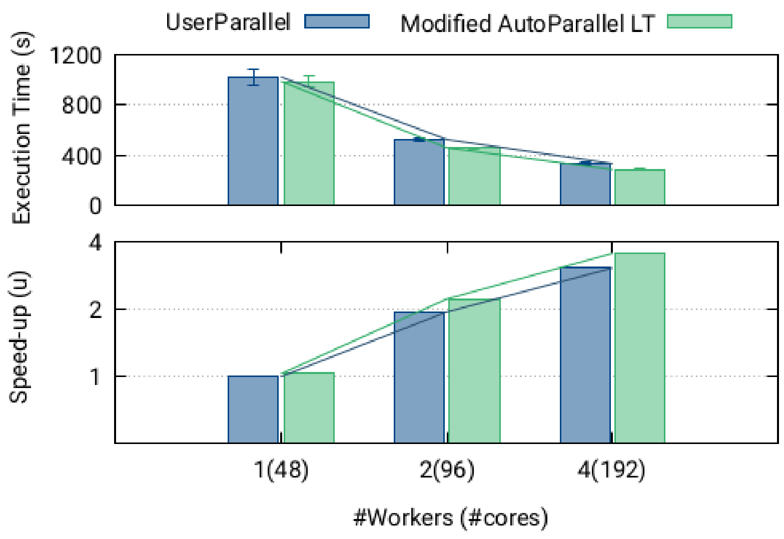


FIGURE 6.10: GEMM with object annotations: Execution time and speed-up with respect to the blocked *userparallel* version using 1 worker (48 cores).

For instance, Figure 6.10 compares the execution results of the *userparallel* blocked version and the *autoparallel LT* version changing the collection annotations per object annotations when running the GEMM application over a dense matrix of $65,536 \times 65,536$ elements decomposed in 32×32 blocks with $2,048 \times 2,048$ elements each. The top plot represents the execution time while the bottom plot represents the speed-up with respect to the *userparallel* blocked version running with a single worker (48 cores). Also, we have chosen 32 blocks because it is the minimum amount providing enough parallelism for 192 cores, and bigger block sizes (e.g., $4,096 \times 4,096$) were impossible due to memory constraints.

We must highlight that the modified *autoparallel LT* version outperforms the *userparallel* version because it is capable of better exploiting the parallelisation of the scaling operation. With little modifications regarding the parameter annotations, advanced users can obtain better blocked algorithms than manually parallelised codes. Hence, we believe that the automatic taskification of loop tiles is a good baseline to design complex blocked algorithms.

6.8 Discussion

AutoParallel is developed as a core part inside our final prototype. This part of the thesis has presented and evaluated AutoParallel, a Python module to automatically parallelise affine loop nests and execute them on distributed infrastructures. Built on top of PyCOMPSs and PLUTO, it is based on sequential programming so that anyone can scale up an application to hundreds of cores. Instead of manually taskifying a sequential python code, the users only need to add a `@parallel` annotation to the methods containing affine loop nests.

The evaluation shows that the codes automatically generated by the AutoParallel module for the Cholesky, LU, and QR applications can achieve similar performance than manually parallelised versions without requiring any effort from the programmer. Thus, AutoParallel goes one step further in easing the development of distributed applications.

Furthermore, the taskification of loop tiles can be enabled by providing a single decorator argument (`tile=True`) to automatically build data blocks from loop tiles and increase the tasks' granularity. Although the overhead with respect to the hand-made blocked version is still far from acceptable, the taskification of loop tiles provides an automatic way to build data blocks from any application; freeing the users from dealing directly with the complexity of block algorithms and allowing them to stick to basic sequential programming. Also, for advanced users, the generated code can be used as a baseline to modify the annotations of the data chunks and obtain the same performance than the hand-made blocked version when the data chunks are disjoint.

As future work, we believe that the taskification of loop tiles is a good approach for fine-grain applications provided that the serialisation performance is improved. For instance, PyCOMPSs could lower the serialisation overhead by serialising each element of the collection in parallel or by serialising the whole collection into a single object.

Finally, AutoParallel could be integrated with different tools similar to PLUTO to support a broader scope of loop nests. For instance, Apollo [213, 151] provides automatic, dynamic and speculative parallelisation and optimisation of programs' loop nests of any kind (for, while or do-while loops). However, its integration would require PyCOMPSs to be extended with some speculative mechanisms.

Chapter 7

Transparent execution of Hybrid Workflows

SUMMARY

In the past years, e-Science applications have evolved from large-scale simulations executed in a single cluster to more complex workflows where these simulations are combined with High-Performance Data Analytics (HPDA). To implement these workflows, developers are currently using different patterns; mainly task-based and dataflow (see Chapter 2 for further details). However, since these patterns are usually managed by separated frameworks, the implementation of these applications requires to combine them; considerably increasing the effort for learning, deploying, and integrating applications in the different frameworks.

This chapter of the thesis focuses solving the research question **Q4** by proposing a way to extend task-based management systems to support continuous input and output data to enable the combination of task-based workflows and dataflows (Hybrid Workflows from now on) using a single programming model. Hence, developers can build complex Data Science workflows with different approaches depending on the requirements. To illustrate the capabilities of Hybrid Workflows, we have built a Distributed Stream Library and a fully functional prototype extending COMPSs. The library can be easily integrated with existing task-based frameworks to provide support for dataflows. Also, it provides a homogeneous, generic, and simple representation of object and file streams in both Java and Python; enabling complex workflows to handle any data type without dealing directly with the streaming back-end.

During the evaluation, we introduce four use cases to illustrate the new capabilities of Hybrid Workflows; measuring the performance benefits when processing data continuously as it is generated, when removing synchronisation points, when processing external real-time data, and when combining task-based workflows and dataflows at different levels. The users identifying these patterns in their workflows may use the presented uses cases (and their performance improvements) as a reference to update their code and benefit of the capabilities of Hybrid Workflows. Furthermore, we analyse the scalability in terms of the number of writers and readers and measure the task analysis, task scheduling, and task execution times when using objects or streams.

7.1 General overview

For many years, large-scale simulations, High-Performance Data Analytics (HPDA), and simulation workflows have become a must to progress in many scientific areas such as life, health, and earth sciences. In such a context, there is a need to adapt the High-Performance infrastructure and frameworks to support the needs and challenges of workflows combining these technologies [218].

Traditionally, developers have tackled the parallelisation and distributed execution of these applications following two different strategies. On the one hand, task-based workflows orchestrate the execution of several pieces of code (*tasks*) that process and generate data values. These tasks have no state and, during its execution, they are isolated from other tasks; thus, task-based workflows consist of defining the data dependencies among tasks. On the other hand, dataflows assume that tasks are persistent executions with a state that continuously receive/produce data values (streams). Through dataflows, developers describe how the tasks communicate to each other.

Regardless of the workflow type, directed graphs are a useful visualisation and management tool. Figure 7.1 shows the graph representation of a task-based workflow (left) and its equivalent dataflow (right). The task dependency graph consists of a producer task (coloured in pink) and five consumer tasks (coloured in blue) that can run in parallel after the producer completes. The dataflow graph also has a producer task (coloured in pink), but one single stateful consumer task (coloured in blue) which processes all the input data sequentially (unless the developer internally parallelises it). Rather than waiting for the completion of the producer task to process all its outputs, the consumer task can process the data as it is generated.

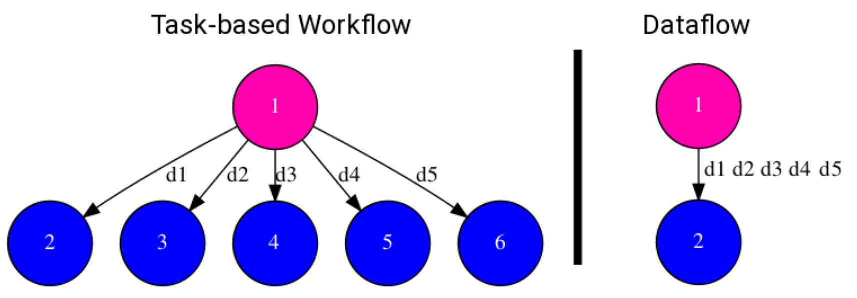


FIGURE 7.1: Equivalent Task-based Worflow and Dataflow.

This chapter of the thesis focuses solving the research question **Q4**; describing a methodology and an implementation to distributedly execute a combination of Task-based Workflows and Dataflows. Next sections detail and evaluate the extensions of our prototype required to support Hybrid Workflows. On the one hand, we extend task-based frameworks to support continuous input and output data using the same programming model. This extension enables developers to build complex Data Science pipelines with different approaches depending on the requirements. The evaluation demonstrates that the use of Hybrid Workflows has significant performance benefits when identifying some patterns in task-based workflows; e.g., when processing data continuously as it is generated, when removing synchronisation points, when processing external real-time data, and when combining task-based workflows and dataflows at different levels. Also, notice that using a single programming model frees the developers from the burden of deploying, using, and communicating different frameworks inside the same workflow.

On the other hand, we present the Distributed Streaming Library that can be easily integrated with existing task-based frameworks to provide support for dataflows. The library provides a homogeneous, generic, and simple representation of a stream for enabling complex workflows to handle any kind of data without dealing directly with the streaming backend. At its current state, the library supports file streams through a custom implementation, and object streams through Kafka [130].

7.2 Related work

Nowadays, state-of-the-art frameworks typically focus on the execution of either task-based workflows or dataflows. Thus, next subsections provide a general overview of the most relevant frameworks for both task-based workflows and dataflows. Furthermore, since our prototype combines both approaches into a single programming model and allows developers to build Hybrid Workflows without deploying and managing two different frameworks, the last subsection details other solutions and compares them with our proposal.

7.2.1 Task-based frameworks

Although all the frameworks handle the tasks and data transfers transparently, there are two main approaches to define task-based workflows. On the one hand, many frameworks force developers to explicitly define the application workflow through a recipe file or a graphical interface. FireWorks [10, 118] defines complex workflows using recipe files in Python, JSON, or YAML. It focuses on high-throughput applications, such as computational chemistry and materials science calculations, and provides support arbitrary computing resources (including queue systems), monitoring through a built-in web interface, failure detection, and dynamic workflow management. Taverna [113, 25] is a suite of tools to design, monitor, and execute scientific workflows. It provides a graphical user interface for the composition of workflows that are written in a Simple Conceptual Unified Flow Language (Scufl) and executed remotely by the Taverna Server to any underlying infrastructure (such as supercomputers, Grids or cloud environments). Similarly, Kepler [146, 219] also provides a graphical user interface to compose workflows by selecting and connecting analytical components and data sources. Furthermore, workflows can be easily stored, reused, and shared across the community. Internally, Kepler's architecture is actor-oriented to allow different execution models into the same workflow. Also, Galaxy [5, 90] is a web-based platform for data analysis focused on accessibility and reproducibility of workflows across the scientific community. The users define workflows through the web portal and submit their executions to a Galaxy server containing a full repertoire of tools and reference data. In an attempt to increase the interoperability between the different systems and to avoid the duplication of development efforts, Tavaxy [2] integrates Taverna and Galaxy workflows in a single environment; defining an extensible set of re-usable workflow patterns and supporting cloud capabilities. Although Tavaxy allows the composition of workflows using Taverna and Galaxy sub-workflows, the resulting workflow does not support streams nor any dataflow pattern.

On the other hand, other frameworks implicitly build the task dependency graph from the user code. Some opt for defining a new scripting language to manage the workflow. These solutions force the users to learn a new language but make a clear differentiation between the workflow's management (the script) and the processes or programs to be executed. Nextflow [71, 169] enables scalable and reproducible workflows using software containers. It provides a fluent DSL to implement and deploy workflows but allows the adaptation of pipelines written in the most common scripting languages. Swift [232, 221] is a parallel scripting language developed in Java and designed to express and coordinate parallel invocations of application programs on distributed and parallel computing platforms. Users only define the main application and the input and output parameters of each program, so that Swift can execute the application in any distributed infrastructure by automatically building the data dependencies.

Other frameworks opt for defining some annotations on top of an already existing language. These solutions avoid the users from learning a new language but merge the workflow annotations and its execution in the same files. Parsl [184] evolves from Swift and provides an intuitive way to build implicit workflows by annotating "apps" in Python codes. In Parsl, the developers annotate Python functions (apps) and Parsl constructs a dynamic, parallel execution graph derived from the implicit linkage between apps based on shared input/output data objects. Parsl then executes apps when dependencies are met. Parsl is resource-independent, that is, the same Parsl script can be executed on a laptop, cluster, cloud, or supercomputer. Dask [202] is a library for parallel computing in Python. Dask follows a task-based approach being able to take into account the data-dependencies between the tasks and exploiting the inherent concurrency. Dask has been designed for computation and interactive data science and integration with Jupyter notebooks. It is built on the dataframe data-structure that offers interfaces to NumPy, Pandas, and Python iterators. Dask supports implicit, simple, task-graphs previously defined by the system (Dask Array or Dask Bag) and, for more complex graphs, the programmer can rely in the `delayed` annotation that supports the asynchronous executions of tasks by building the corresponding task-graph. COMPSs [32, 144, 54] is a task-based programming model for the development of workflows/applications to be executed in distributed programming platforms. The task-dependency graph (or workflow) is generated at execution time and depends on the input data and the dynamic execution of the application. Thus, compared with other workflow systems that are based on the static drawing of the workflow, COMPSs offers a tool for building dynamic workflows, with all the flexibility and expressivity of the programming language.

7.2.2 Dataflow frameworks

Stream processing has become an increasingly prevalent solution to process social media and sensor devices data. On the one hand, many frameworks have been created explicitly to face this problem. Apache Flink [16] is streaming dataflow engine to perform stateful computations over data streams (i.e., event-driven applications, streaming pipelines or stream analytics). It provides exactly-once processing, high throughput, automated memory management, and advanced streaming capabilities (such as windowing). Flink users build dataflows that start with one or more input streams (sources), perform arbitrary transformations, and end in one or more outputs (sinks). Apache Samza [24] allows building stateful applications for event processing or real-time analytics. Its differential point is to offer built-in support to process and transform data from many sources, including Apache Kafka, AWS Kinesis, Azure EventHubs, ElasticSearch, and HDFS. Samza users define a stream application that processes messages from a set of input streams, transforms them by chaining multiple operators, and emits the results to output streams or stores. Also, Samza supports at-least-once processing, guaranteeing no data-loss even in case of failures. Apache Storm [225] a is distributed real-time computation system based on the master-worker architecture and used in real-time analytics, online machine learning, continuous computation, and distributed RPC, between others. Storm users define topologies that consume streams of data (spouts) and process those streams in arbitrarily complex ways (bolts), re-partitioning the streams between each stage of the computation however needed. Although Storm natively provides at-least-once processing, it also supports exactly-once processing via its high-level API called Trident. Twitter Heron [135] is a real-time, fault-tolerant stream processing engine. Heron was built as the Storm's successor, meaning that the topology concepts (spouts and bolts) are the same and has a compatible API with Storm. However, Heron provides better resource isolation, new scheduler features (such as on-demand resources), better throughput, and lower latency.

7.2.3 Hybrid frameworks

Apache Spark [237] is a general framework for big data processing that was originally designed to overcome the limitations of MapReduce [64]. Among the many built-in modules, Spark Streaming [236] is an extension of the Spark core to evolve from batch processing to continuous processing by emulating streaming via micro-batching. It ingests input data streams from many sources (e.g., Kafka, Flume, Kinesis, ZeroMQ) and divides them into batches that are then processed by the Spark engine; allowing to combine streaming with batch queries. Internally, the continuous stream of data is represented as a sequence of RDDs in a high-level abstraction called Discretized Stream (DStream).

Notice that Spark is based on high-level operators (operators on RDDs) that are internally represented as a DAG; limiting the patterns of the applications. In contrast, our approach is based on sequential programming, which allows the developer to build any kind of application. Furthermore, micro-batching requires a predefined threshold or frequency before any processing occurs; which can be "real-time" enough for many applications, but may lead to failures when micro-batching is simply not fast enough. In contrast, our solution uses a dedicated streaming engine to handle dataflows; relying on streaming technologies rather than micro-batching and ensuring that the data is processed as soon as it is available.

On the other hand, other solutions combine existing frameworks to support Hybrid Workflows. Asterism [86] is a hybrid framework combining `dispel4py` and Pegasus at different levels to run data-intensive stream-based applications across platforms on heterogeneous systems. The main idea is to represent the different parts of a complex application as `dispel4py` workflows which are, then, orchestrated by Pegasus as tasks. While the stream-based execution is managed by `dispel4py`, the data movement between the different execution platforms and the workflow engine (submit host) is managed by Pegasus. Notice that Asterism can only handle dataflows inside task-based workflows (`dispel4py` workflows represented as Pegasus' tasks), while our proposal is capable of orchestrating nested task-flows, nested dataflows, dataflows inside task-based workflows, and task-based workflows inside dataflows.

7.3 Kafka

This section reviews the essential Kafka [130][20] features because it illustrates the streaming models, and it is the starting point of our implementation for object streams. However, as explained in the next sections, we highlight that our interface is designed for any backend and that Kafka is only used as a streaming model example for the experimentation.

Kafka [130][20] is a streaming platform that runs as a cluster to store streams of records in topics. Built on top of ZooKeeper [26], it is used to publish and subscribe streams of records (similarly to message queueing), store streams of records in a fault-tolerant manner, and process streams of records as they occur. Furthermore, its main focus is real-time streaming applications and data pipelines.

Figure 7.2 illustrates the basic concepts in Kafka and how they relate to each other. *Records* – each blue box in the figure – are key-value pairs containing application-level information registered along with its publication time.

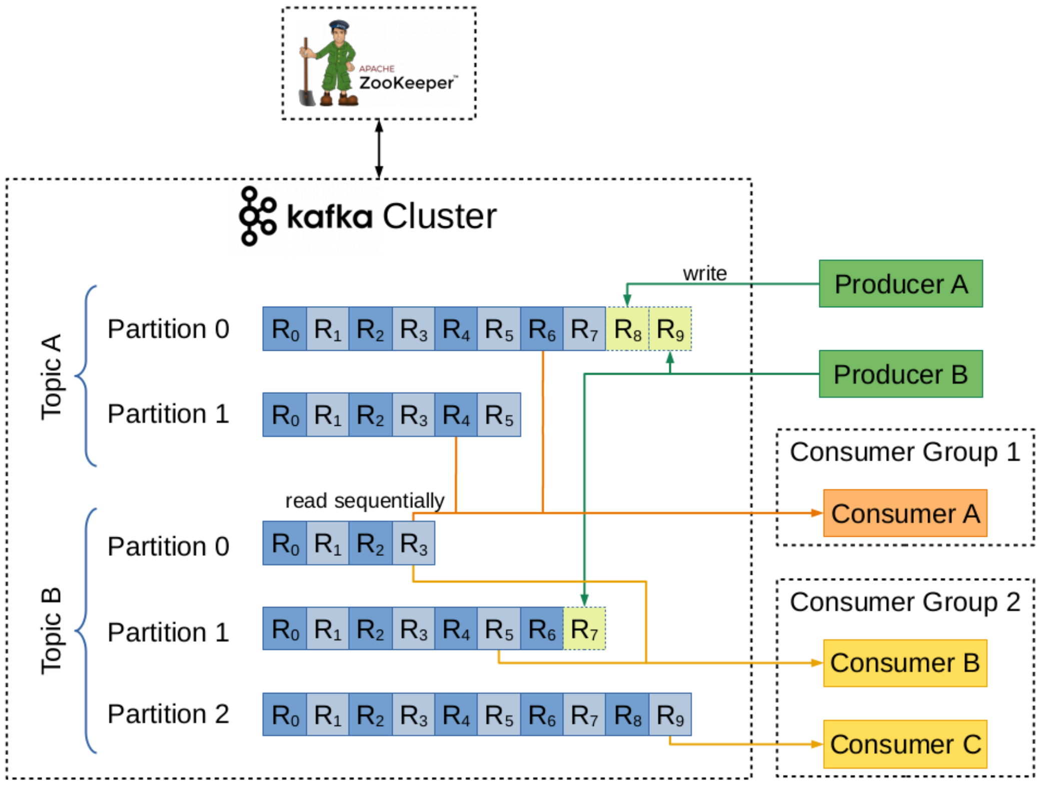


FIGURE 7.2: Description of Kafka's basic concepts.

Kafka users define several categories or *topics* to which records belong. Kafka stores each topic as a partitioned log with an arbitrary number of partitions and maintains a configurable number of partition replicas across the cluster to provide fault tolerance and record-access parallelism. Each partition contains an immutable, publication-time-ordered sequence of records each uniquely identified by a sequential id number known as the *offset* of the record. The example in the figure defines two topics (*Topic A* and *Topic B*) with 2 and 3 partitions, respectively.

Finally, *Producers* and *Consumers* are third-party application components that interact with Kafka to publish and retrieve data. The former add new records to the topics of their choice, while the latter subscribe to one or more topics for receiving records related to them. Consumers can join in *Consumer groups*. Kafka ensures that each record published to a topic is delivered to at least one consumer instance within each subscribing group; thus, multiple processes on remote machines can share the processing of the records of that topic. Although most often delivered exactly once, records might duplicate when one consumer crashes without a clean shutdown and another consumer within the same group takes over its partitions.

Back to the example in the figure, *Producer A* publishes one record to *Topic A*, and *Producer B* publishes two records, one to *Topic A* and one to *Topic B*. *Consumer A*, with a group of its own, processes all the records in *Topic A* and *Topic B*. Since *Consumer B* and *Consumer C* belong to the same consumer group, they share the processing of all the records from *Topic B*.

Besides the Consumer and Producer API, Kafka also provides the Stream Processor and Connector APIs. The former, usually used in the intermediate steps of the fluent stream processing, allows application components to consume an input stream from one or more topics and produce an output stream to one or more topics. The latter is used for connecting producers and consumers to already existing applications or data systems. For instance, a connector to a database might capture every change to a table.

7.4 Architecture

Figure 7.3 depicts a general overview of the proposed solution. When executing regular task-based workflows, the application written following the programming model interacts with the runtime to spawn the remote execution of tasks and retrieve the desired results. Our proposal includes a representation of a stream (*DistroStream* Interface) that provides applications with homogeneous stream accesses regardless of the stream backend supporting them. Moreover, we extend the programming model and runtime to provide task annotations and scheduling capabilities for streams.

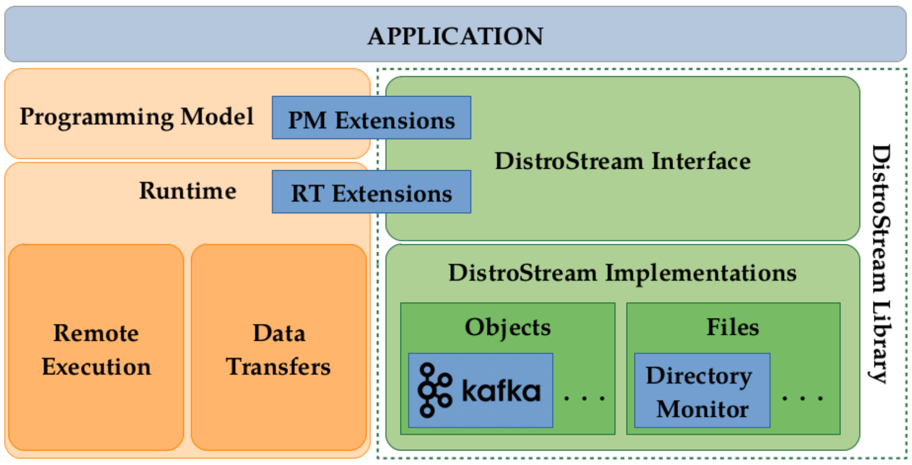


FIGURE 7.3: General architecture.

The following subsections discuss the architecture of the proposed solution in a bottom-up approach, starting from the representation of a stream (*DistroStream* API) and its implementations. Next, we describe the *Distributed Stream Library* and its internal components. Finally, we detail the integration of this library with the programming model (COMPSs) and the necessary extensions of its runtime system.

7.4.1 Distributed Stream interface

The Distributed Stream is a representation of a stream used by applications to publish and receive data values. Its interface provides a common API to guarantee homogeneity on all interactions with streams.

As shown in Listing 7.1, the `DistroStream` interface provides a `publish` method for submitting a single message or a list of messages (lines 5 and 6) and a `poll` method to retrieve all the currently available unread messages (lines 9 and 10). Notice that the latter has an optional `timeout` parameter (in milliseconds) to wait until an element becomes available or the specified time expires. Moreover, the streams can be created with an optional `alias` parameter (line 2) to allow different applications to communicate through them. Also, the interface provides other methods to query stream metadata; such as the stream type (line 13), id (line 14), or alias (line 15). Finally, the interface includes methods to check the status of a stream (line 18), and to close it (line 21).

```java
// INSTANTIATION
public DistroStream(String alias) throws RegistrationException;

// PUBLISH METHODS
public abstract void publish(T message) throws BackendException;
public abstract void publish(List<T> messages) throws BackendException;

// POLL METHODS
public abstract List<T> poll() throws BackendException;
public abstract List<T> poll(long timeout) throws BackendException;

// METADATA METHODS
public StreamType getStreamType();
public String getAlias();
public String getId();

// STREAM STATUS
public boolean isClosed();

// CLOSE STREAM
public final void close();
```

LISTING 7.1: Distributed Stream Interface in Java.

Due to space constraints, the figure only shows the Java interface, but our prototype also provides the equivalent interface in Python.

7.4.2 Distributed Stream implementations

As shown in Figure 7.4, two different implementations of the `DistroStream` API provide the specific logic to support object and file streams. Object streams are suitable when sharing data within the same language or framework. On the other hand, file streams allow different frameworks and languages to share data. For instance, the files generated by an MPI simulation in C or Fortran can be received through an stream and processed in a Python or Java application.

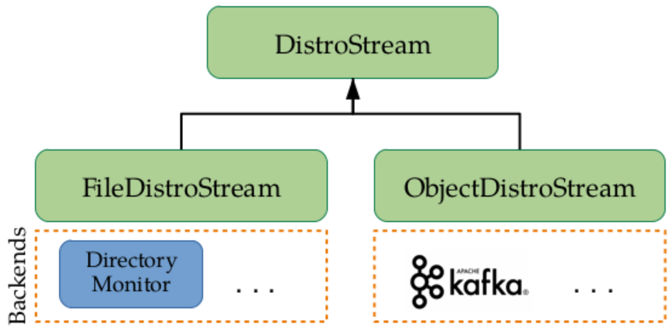


FIGURE 7.4: DistroStream class relationship.

7.4.2.1 Object streams

`ObjectDistroStream` (ODS) implements the generic `DistroStream` interface to support object streams. Each ODS has an associated `ODSPublisher` and `ODSConsumer` that interact appropriately with the software handling the message transmission (streaming backend). The ODS instantiates them upon the first invocation of a publish or a poll method respectively. This behaviour guarantees that the same object stream has different publisher and consumer instances when accessed from different processes, and that the producer and consumer instances are only registered when required, avoiding unneeded registrations on the streaming backend.

At its current state, the available implementation is backed by Kafka, but the design is ready to support many backends. Notice that the ODS, `ODSPublisher`, and `ODSConsumer` are just abstractions to hide the interaction with the underlying backend. Hence, any other backend (such as an MQTT broker) can be supported without any modification at the workflow level by implementing the functionalities defined in these abstractions.

Considering the Kafka concepts introduced in Section 7.3, each ODS becomes a Kafka topic named after the stream id. When created, the `ODSPublisher` instantiates a `Kafka-Producer` whose `publish` method builds a new `ProducerRecord` and submits it to the corresponding topic via the `KafkaProducer.send` method. If the `publish` invocation sends several messages, the `ODSPublisher` iteratively performs the publishing process for each message so that Kafka registers it as separated records.

Likewise, a new `KafkaConsumer` is instantiated along with an `ODSConsumer`. Then, the `KafkaConsumer` is registered to a consumer group shared by all the consumers of the same application to avoid replicated messages, and subscribed to the topic named after the id of the stream. Hence, the `poll` method retrieves a list of `ConsumerRecords` and deserialises their values. To ensure that records are processed exactly-once, consumers also interact with Kafka's `AdminClient` to delete all the processed records from the database.

```
 1  // PRODUCER
 2  void produce(List<T> objs) {
 3    // Create stream (alias is not mandatory)
 4    String alias = "myStream";
 5    ObjectDistroStream<T> ods = new ObjectDistroStream<>(alias);
 6    // Metadata getters
 7    System.out.println("Stream Id: " + ods.getId());
 8    System.out.println("Stream Alias: " + ods.getAlias());
 9    System.out.println("Stream Type: " + ods.getStreamType());
10    // Publish (single element or list)
11    for (T obj : objs) {
12      ods.publish(obj);
13    }
14    ods.publish(objs);
15    // Close stream
16    ods.close()
17  }
18
19  // CONSUMER
20  void consume(ObjectDistroStream<T> ods) {
21    // Poll current elements (without timeout)
22    if (!ods.isClosed()) {
23      List<T> newElems = ods.poll();
24    }
25    // Poll until stream is closed (with timeout)
26    while (!ods.isClosed()) {
27      List<T> newElems = ods.poll(5)
28    }
29  }
```

LISTING 7.2: Object Streams (ODS) example in Java.

Listing 7.2 shows an example using object streams in Java. Notice that the stream creation (line 5) forces all the stream objects to be of the same type `T`. Internally, the stream serialises and deserialises the objects so that the application can publish and poll elements of type `T` directly to/from the stream. As previously explained, the example also shows the usage of the `publish` method for a single element (line 12) or a list of elements (line 14), the `poll` method with the optional `timeout` parameter (lines 23 and 27, respectively), and the common API calls to close the stream (line 16), check its status (line 22), and retrieve metadata information (lines 7 to 9).

Due to space constraints, the example only shows the ODS usage in Java, but our prototype provides an equivalent implementation in Python.

7.4.2.2 File streams

The `FileDistroStream` implementation (FDS) backs the `DistroStream` up to support the streaming of files. Like ODS, its design allows using different backends; however, at its current state, it uses a custom implementation that monitors the creation of files inside a given directory. The *Directory Monitor* backend sends the file locations through the stream and relies on a distributed file system to share the file content. Thus, the monitored directory must be available to every client on the same path.

```
1   // PRODUCER
2   void produce(String baseDir, List<String> fileNames) throws IOException {
3     // Create stream (alias is not mandatory)
4     String alias = "myStream";
5     FileDistroStream<T> fds = new FileDistroStream<>(alias, baseDir);
6     // Publish files (no need to explicitly call the publish
7     // method, the baseDir directory is automatically monitored)
8     for (String fileName : fileNames) {
9       String filePath = baseDir + fileName;
10      try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {
11        writer.write(...);
12      }
13    }
14    // Close stream
15    fds.close()
16  }
17
18  // CONSUMER
19  void consume(FileDistroStream<T> fds) {
20    // Poll current elements (without timeout)
21    if (!fds.isClosed()) {
22      List<String> newFiles = fds.poll();
23    }
24    // Poll until stream is closed (with timeout)
25    while (!fds.isClosed()) {
26      List<String> newFiles = fds.poll(5)
27    }
28  }
```

LISTING 7.3: File Streams (FDS) example in Java.

Listing 7.3 shows an example using file streams in Java. Notice that the FDS instantiation (line 5 in the listing) requires a base directory to monitor the creation of files and that it optionally accepts an alias argument to retrieve the content of an already existing stream. Also, files are not explicitly published on the stream since the base directory is automatically monitored (lines 8 to 13). Instead, regular methods to write files are used. However, the consumer must explicitly call the `poll` method to retrieve a list of the newly available file paths in the stream (lines 22 and 26). As with ODS, applications can also use the common API calls to close the stream (line 15), check its status (lines 21 and 25), and retrieve metadata information.

Due to space constraints, the example only shows the FDS usage in Java, but our prototype provides an equivalent implementation in Python.

7.4.3 Distributed Stream Library

The Distributed Stream Library (`DistroStreamLib`) handles the stream objects and provides three major components. First, the `DistroStream` API and implementations described in the previous sections.

Second, the library provides the DistroStream Client that must be available for each application process. The client is used to forward any stream metadata request to the DistroStream Server or any stream data access to the suitable stream backend (i.e., *Directory Monitor*, or *Kafka*). To avoid repeated queries to the server, the client stores the retrieved metadata in a cache-like fashion. Either the Server or the backend can invalidate the cached values.

Third, the library provides the *DistroStream Server* process that is unique for all the applications sharing the stream set. The server maintains a registry of active streams, consumers, and producers with the purpose of coordinating any stream data or metadata access. Among other responsibilities, it is in charge of assigning unique ids to new streams, checking the access permissions of producers and consumers when requesting `publish` and `poll` operations, and notifying all registered consumers when the stream has been completely closed and there are no producers remaining.

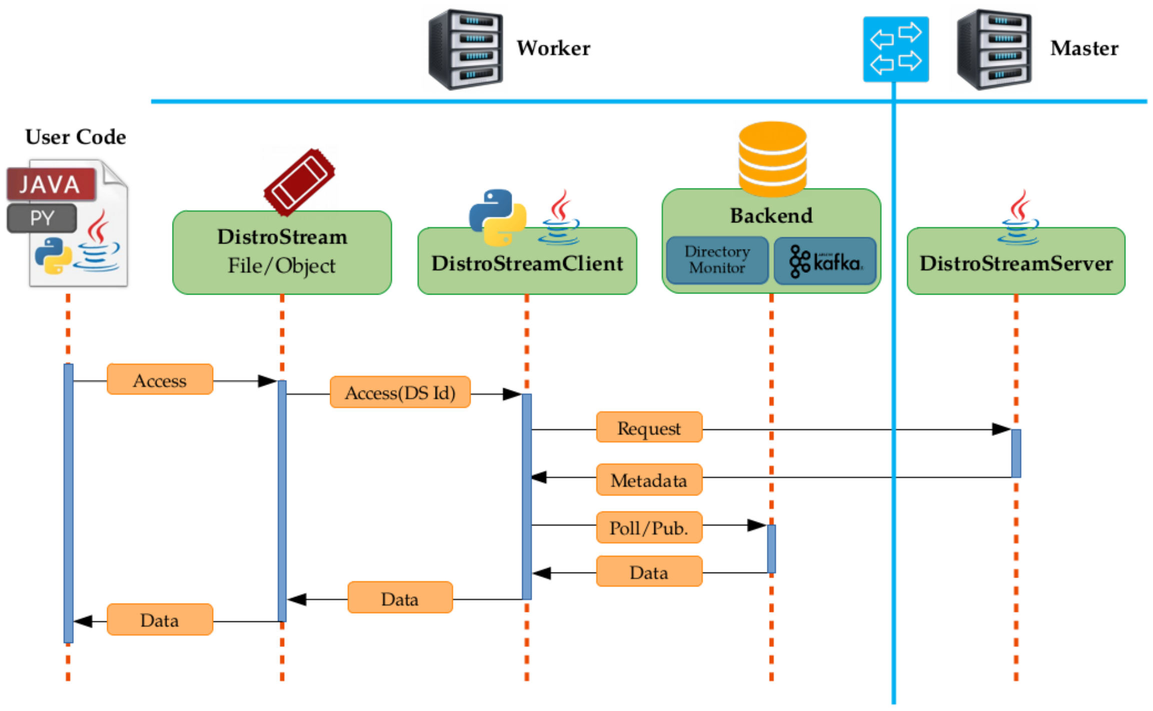


FIGURE 7.5: Sequence diagram of the Distributed Stream Library components.

Figure 7.5 contains a sequence diagram that illustrates the interaction of the different Distributed Stream Library components when serving a user petition. The *DistroStream* implementation used by the applications always forwards the requests to the *DistroStream Client* available on the process. The client communicates with the *DistroStream Server* for control purposes, and retrieves from the backend the real data.

7.4.4 Programming model extensions

As already mentioned in Section 3.1, the prototype to evaluate Hybrid Workflows is based on the COMPSs workflow manager. At programming-model level, we have extended the COMPSs Parameter Annotation to include a new *STREAM* type.

As shown in Listing 7.4, on the one hand, the users declare producer tasks (methods that write data into a stream) by adding a parameter of type STREAM and direction OUT (lines 3 to 7 in the listing). On the other hand, the users declare consumer tasks (methods that read data from a stream) by adding a parameter of type STREAM and direction IN (lines 9 to 13 in the listing). In the current design, we have not considered INOUT streams because we do not imagine a use case where the same method writes data into its own stream. However, it can be easily extended to support such behaviour when required.

```
1  public interface Itf {
2
3      @Method(declaringClass = "Producer")
4      Integer sendMessages(
5          @Parameter(type = Type.STREAM, direction = Direction.OUT)
6          DistroStream stream
7      );
8
9      @Method(declaringClass = "Consumer")
10     Result receiveMessages(
11         @Parameter(type = Type.STREAM, direction = Direction.IN)
12         DistroStream stream
13     );
14 }
```

LISTING 7.4: Stream parameter annotation example in Java.

Furthermore, we want to highlight that this new annotation allows integrating streams smoothly with any other previous annotation. For instance, Listing 7.5 shows a single producer task that uses two parameters: a stream parameter typical of dataflows (lines 5 and 6) and a file parameter typical of task-based workflows (lines 7 and 8).

```
1  public interface Itf {
2
3      @Method(declaringClass = "Producer")
4      Integer sendMessages(
5          @Parameter(type = Type.STREAM, direction = Direction.OUT)
6          DistroStream stream,
7          @Parameter(type = Type.FILE, direction = Direction.IN)
8          String file
9      );
10 }
```

LISTING 7.5: Example combining stream and file parameters in Java.

7.4.5 Runtime extensions

As depicted in Figure 7.6, COMPSs registers the different tasks from the application's main code through the Task Analyser component. Then, it builds a task graph based on the data dependencies and submits it to the *Task Dispatcher*. The Task Dispatcher interacts with the Task Scheduler to schedule the data-free tasks when possible and, eventually, submit them to execution. The execution step includes the job creation, the transfer of the input data, the job transfer to the selected resource, the real task execution on the worker, and the output retrieval from the worker back to the master. If any of these steps fail, COMPSs

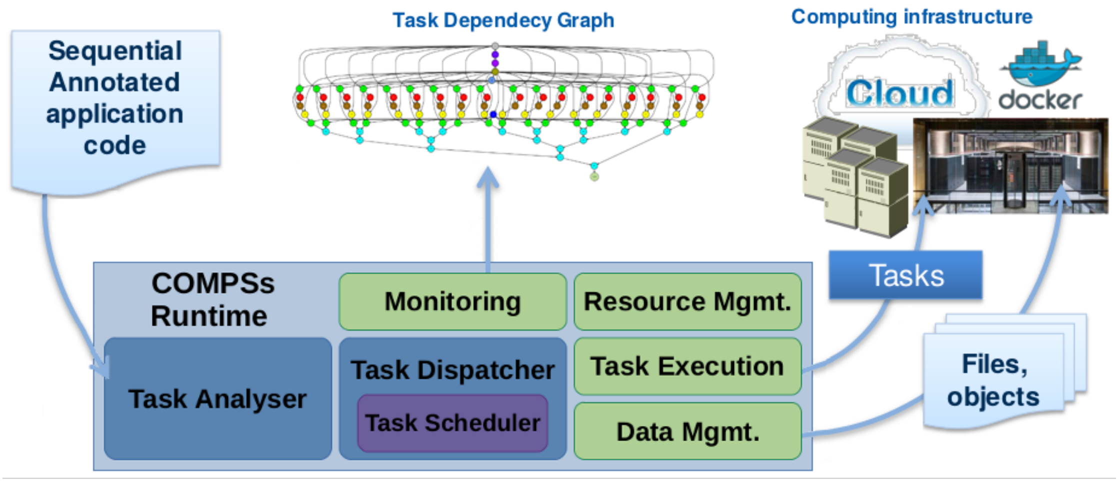


FIGURE 7.6: Structure of the internal COMPSs components.

provides fault-tolerant mechanisms for partial failures. Also, once the task has finished, COMPSs stores the monitoring data of the task, synchronises any data required by the application, releases the data-dependent tasks so that they can be scheduled, and deletes the task.

Therefore, the new Stream annotation has forced modifications in the Task Analyser and Task Scheduler components. More specifically, notice that a stream parameter does not define a traditional data dependency between a producer and a consumer task since both tasks can run at the same time. However, there is some information that must be stored so that the Task Scheduler can correctly handle the available resources and the data locality. In this sense, when using the same stream object, our prototype prioritises producer tasks over consumer tasks to avoid wasting resources when a consumer task is waiting for data to be produced by a non-running producer task. Moreover, the Task Scheduler assumes that the resources that are running (or have run) producer tasks are the data locations for the stream. This information is used to schedule the consumer tasks accordingly and minimise as much as possible the data transfers between nodes.

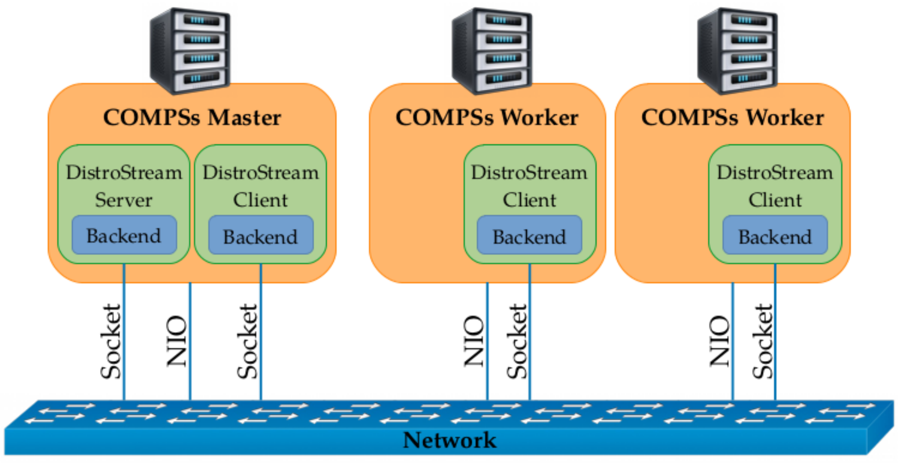


FIGURE 7.7: COMPSs and Distributed Stream Library deployment.

Regarding the components' deployment, as shown in Figure 7.7, the COMPSs master spawns the *DistroStream Server* and the required backend. Furthermore, it includes a *DistroStream Client* to handle the stream accesses and requests performed on the application's main code. On the other hand, the COMPSs workers only spawn a *DistroStream Client* to handle the stream accesses and requests performed on the tasks. Notice that the COMPSs master-worker communication is done through NIO [120], while the *DistroStream Server-Client* communication is done through Sockets.

7.5 Use cases

Enabling Hybrid task-based workflows and dataflows into a single programming model allows users to define new types of complex workflows. We introduce four patterns that appear in real-world applications so that the users identifying these patterns in their workflows can benefit from the new capabilities and performance improvements of Hybrid Workflows. Next subsections provide in-depth analysis of each use case.

7.5.1 Use case 1: Continuous data generation

One of the main drawbacks of task-based workflows is waiting for task completion to process its results. Often, the output data is generated continuously during the task execution rather than at the end. Hence, enabling data streams allows users to process the data as it is generated. The following use case is a simplification of a collaboration with the Argonne National Laboratory for the development of the Decaf application [233]. Decaf is a hierarchical heterogeneous workflow composed of subworkflows where each level of the hierarchy uses different programming, execution, and data models.

```python
@constraint(computing_units=CORES_SIMULATION)
@task(varargs_type=FILE_OUT)
def simulation(num_files, *args):
    ...

@constraint(computing_units=CORES_PROCESS)
@task(input_file=FILE_IN, output_image=FILE_OUT)
def process_sim_file(input_file, output_image):
    ...

@constraint(computing_units=CORES_MERGE)
@task(output_gif=FILE_OUT, varargs_type=FILE_IN)
def merge_reduce(output_gif, *args):
    ...

def main():
    # Parse arguments
    num_sims, num_files, sim_files, output_files, output_gifs = ...
    # Launch simulations
    for i in range(num_sims):
        simulation(num_files, *sim_files[i])
    # Process generated files
    for i in range(num_sims):
        for j in range(num_files):
            process_sim_file(sim_files[i][j], output_images[i][j])
    # Launch merge phase
    for i in range(num_sims):
        merge_reduce(output_gifs[i], *output_images[i])
    # Synchronise files
    for i in range(num_sims):
        output_gifs[i] = compss_wait_on_file(output_gifs[i])
```

LISTING 7.6: Simulations' application in Python without streams.

For instance, Listing 7.6 shows the code of a pure task-based application that launches `num_sims` simulations (line 21). Each simulation produces output files at different time steps of the simulation (i.e., an output file every iteration of the simulation). The results of these simulations are processed separately by the `process_sim_file` task (line 25) and merged to a single GIF per simulation (line 28). The example code also includes the task definitions (lines 1 to 13) and the synchronisation API calls to retrieve the results (line 31).

Figure 7.8 shows the task graph generated by the previous code when running with 2 simulations (`num_sims`) and 5 files per simulation (`num_files`). The `simulation` tasks are shown in blue, the `process_sim_file` in white and red, and the `merge_reduce` in pink. Notice that the simulations and the processing of the files cannot run in parallel since the task-based workflow forces the completion of the simulation tasks to begin any of the processing tasks.

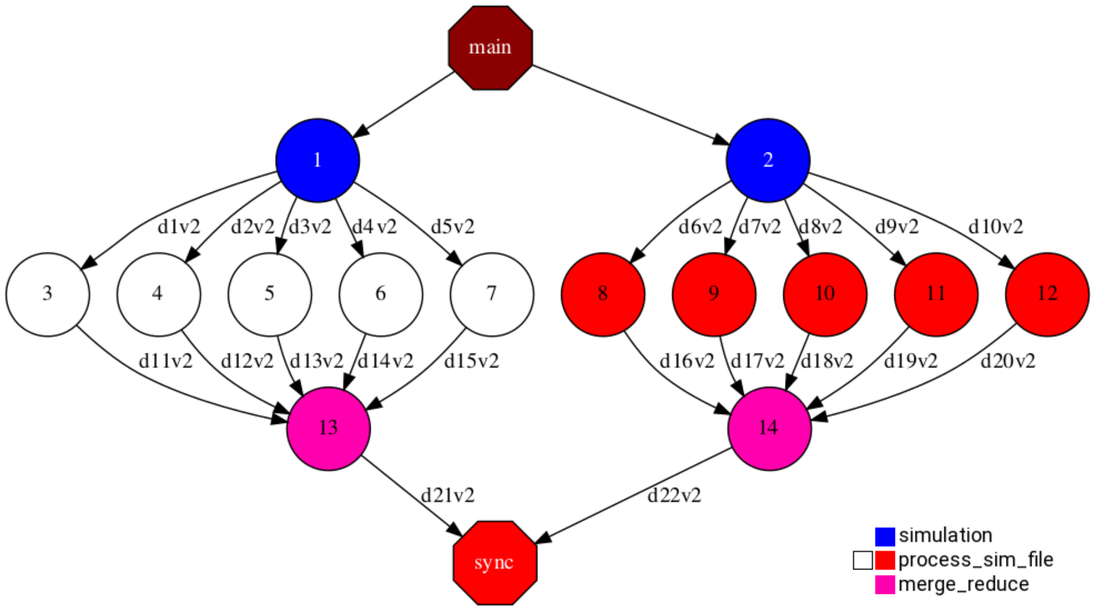


FIGURE 7.8: Task graph of the simulation application without streaming.

On the other hand, Listing 7.7 shows the code of the same application using streams to retrieve the data from the simulations as it is generated and forwarding it to its processing tasks. The application initialises the streams (lines 20 to 22), launches `num_sims` simulations (line 25), spawns a process task for each received element in each stream (line 34), merges all the output files into a single GIF per simulation (line 37), and synchronises the final results. The `process_sim_file` and `merge_reduce` task definitions are identical to the previous example. Conversely, the `simulation` task definition uses the `STREAM_OUT` annotation to indicate that one of the parameters is a stream where the task is going to publish data. Also, although the simulation, merge, and synchronisation phases are very similar to the pure task-based workflow, the processing phase is completely different (lines 27 to 34). When using streams, the main code needs to check the stream status, retrieve its published elements, and spawn a `process_sim_task` per element. However, the complexity of the code does not increase significantly when adding streams to an existing application.

```
@constraint(computing_units=CORES_SIMULATION)
@task(fds=STREAM_OUT)
def simulation(fds, num_files):
    ...

@constraint(computing_units=CORES_PROCESS)
@task(input_file=FILE_IN, output_image=FILE_OUT)
def process_sim_file(input_file, output_image):
    ...

@constraint(computing_units=CORES_MERGE)
@task(output_gif=FILE_OUT, varargs_type=FILE_IN)
def merge_reduce(output_gif, *args):
    ...

def main():
    # Parse arguments
    num_sims, num_files, output_images, output_gifs = ...
    # Initialise streams
    input_streams = [None for _ in range(num_sims)]
    for i in range(num_sims):
        input_streams[i] = FileDistroStream(base_dir=stream_dir)
    # Launch simulations
    for i in range(num_sims):
        simulation(input_streams[i], num_files)
    # Process generated files
    for i in range(app_args.num_simulations):
        while not input_streams[i].is_closed():
            # Process new files
            new_files = input_streams[i].poll()
            for input_file in new_files:
                output_image = input_file + ".out"
                output_images[i].append(output_image)
                process_sim_file(input_file, output_image)
    # Launch merge phase
    for i in range(app_args.num_simulations):
        merge_reduce(output_gifs[i], *output_images[i])
    # Synchronise files
    for i in range(app_args.num_simulations):
        output_gifs[i] = compss_wait_on_file(output_gifs[i])
```

LISTING 7.7: Simulations' application in Python with streams.

Figure 7.9 shows the task graph generated by the previous code when running with the same parameters than the pure task-based example (2 simulations and 5 files per simulation). The colour code is also the same than the previous example: the `simulation` tasks are shown in blue, the `process_sim_file` in white and red, and the `merge_reduce` in pink. Notice that streams enable the execution of the processing tasks while the simulations are still running; potentially reducing the total execution time and increasing the resources utilisation (see Section 7.6.2 for further details).

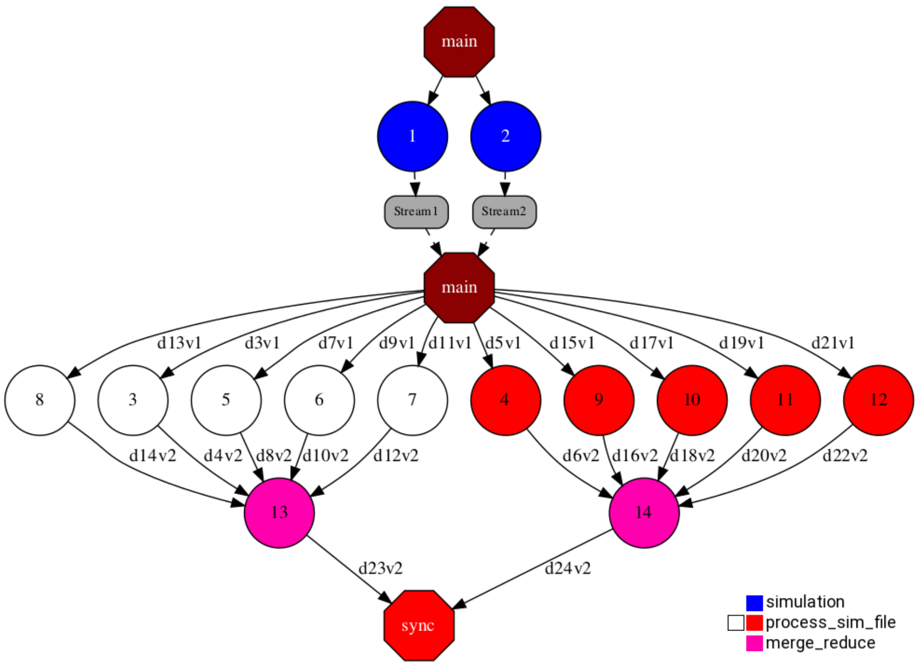


FIGURE 7.9: Task graph of the simulation application with streaming.

7.5.2 Use case 2: Asynchronous data exchange

Streams can also be used to communicate data between tasks without waiting for the tasks' completion. This technique can be useful when performing parameter sweep, cross-validation, or running the same algorithm with different initial points.

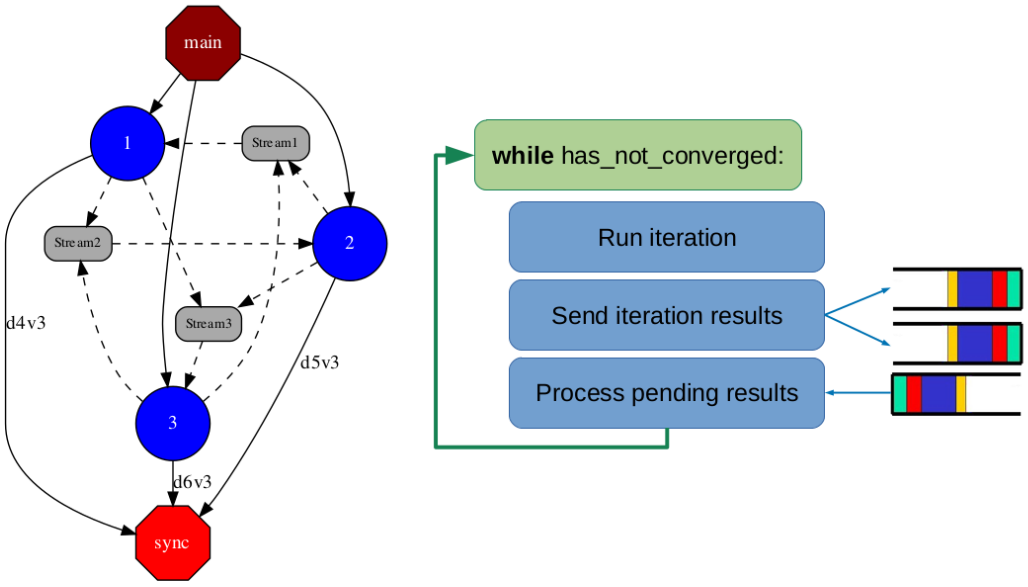


FIGURE 7.10: Task graph of the multi-simulations application.

For instance, Figure 7.10 shows three algorithms running simultaneously that exchange control data at the end of every iteration. Notice that the data exchange at the end of each iteration can be done synchronously by stopping all the simulations, or asynchronously by sending the updated results and processing the pending messages in the stream (even though some messages of the actual iteration might be received in the next iteration). Furthermore, each algorithm can run a complete task-based workflow to perform the iteration calculus obtaining a nested task-based workflow inside a pure dataflow.

7.5.3 Use case 3: External streams

Many applications receive its data continuously from external streams (i.e., IoT sensors) that are not part of the application itself. Moreover, depending on the workload, the stream data can be produced by a single task and consumed by many tasks (one to many), produced by many tasks and consumed by a single task (many to one), or produced by many tasks and consumed by many tasks (many to many). The *Distributed Stream Library* supports all three scenarios transparently, and allows to configure the consumer mode to process the data at least once, at most once, or exactly once when using many consumers.

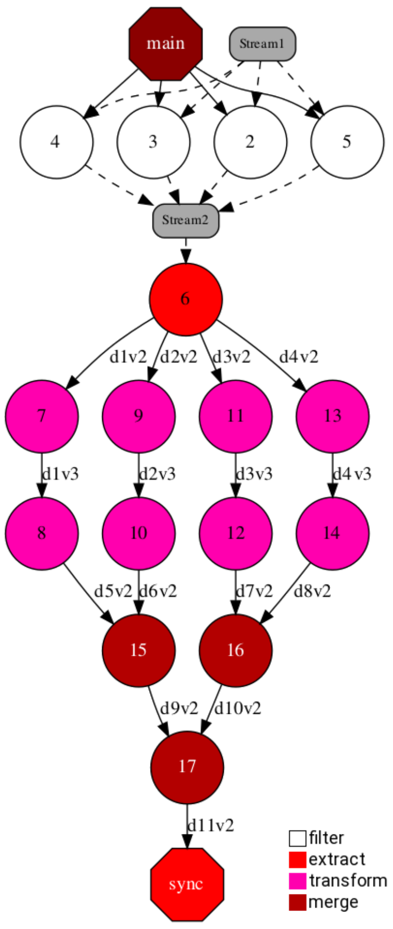


FIGURE 7.11: Task graph of the sensor application.

Figure 7.11 shows an external sensor (*Stream 1* in the figure) producing data that is filtered simultaneously by 4 tasks (coloured in white). The relevant data is then extracted from an internal stream (*Stream 2*) by an intermediate task (task 6, coloured in red), and used to run a task-based algorithm. The result is a hybrid task-based workflow and dataflow. Also, the sensor uses a one-to-many stream configured to process the data exactly once, and the *filter* (coloured in white) tasks use a many-to-one stream to publish data to the *extract* task (coloured in red).

7.5.4 Use case 4: Dataflows with nested task-based workflows

Our proposal also allows to combine task-based workflows and dataflows at different levels; having nested task-based workflows inside a dataflow task or vice-versa. This feature enables the internal parallelisation of tasks, allowing workflows to scale up and down resources depending on the workload.

For instance, Figure 7.12 shows a dataflow with two nested task-based workflows. The application is similar to the previous use case: the task 1 (coloured in pink) produces the data, task 2 (in white) filters it, task 3 (in blue) extracts and collects the data, and task 4 (in red) runs a big computation.

Notice that, in the previous use case, the application always has 4 filter tasks. However, in this scenario, the filter task has a nested task-based workflow that accumulates the received data into batches and spawns a new filter task per batch. This technique dynamically adapts the resource usage to the amount of data received by the input stream. Likewise, the big computation task also contains a nested task-based workflow. This shows that users can parallelise some computations internally without modifying the original dataflow.

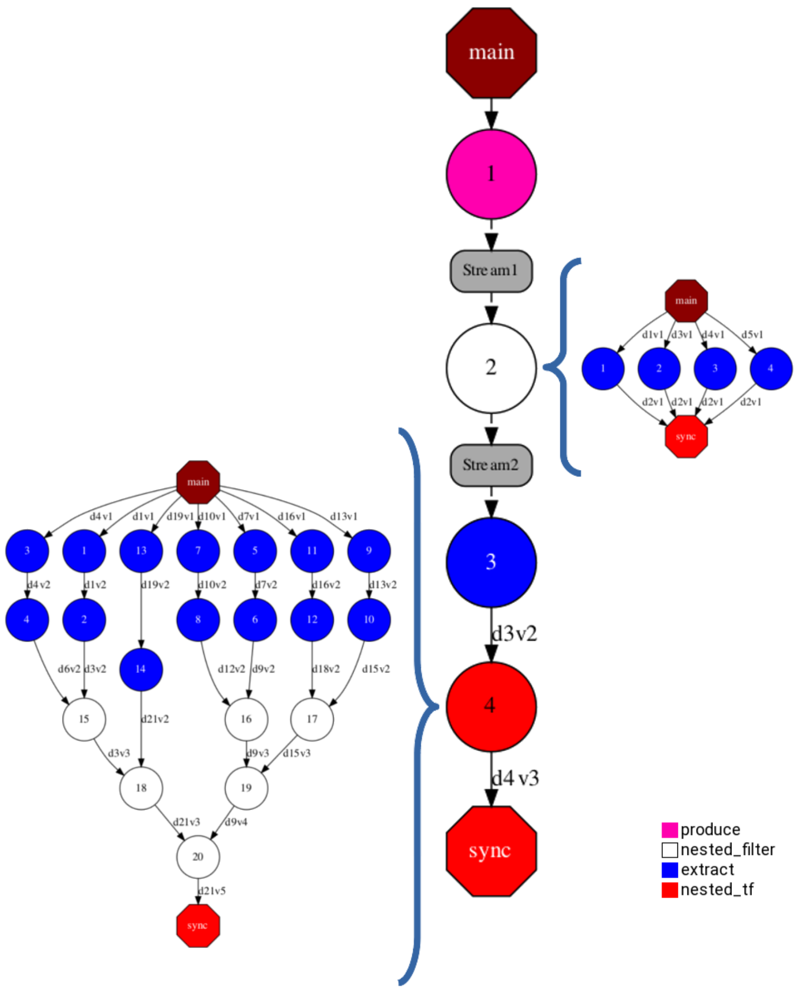


FIGURE 7.12: Task graph of the hybrid nested application.

7.6 Evaluation

This section evaluates the performance of the new features enabled by our prototype when using data streams against their equivalent implementations using task-based workflows. Furthermore, we analyse the stream writer and reader processes' scalability and load balancing. Finally, we provide an in-depth analysis of the COMPSs runtime performance by comparing the task analysis, task scheduling, and task execution times when using pure task-based workflows or streams.

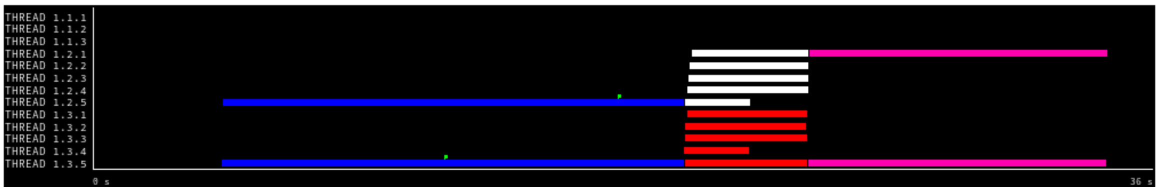
7.6.1 Experimental setup

The results presented in this section have been obtained using the MareNostrum 4 supercomputer [149] located at the Barcelona Supercomputing Center (BSC). Its current peak performance is 11.15 Petaflops. The supercomputer is composed by 3456 nodes, each of them with two Intel®Xeon Platinum 8160 (24 cores at 2,1 GHz each). It has 384.75 TB of main memory, 100Gb Intel®Omni-Path Full-Fat Tree Interconnection, and 14 PB of shared disk storage managed by the Global Parallel File System.

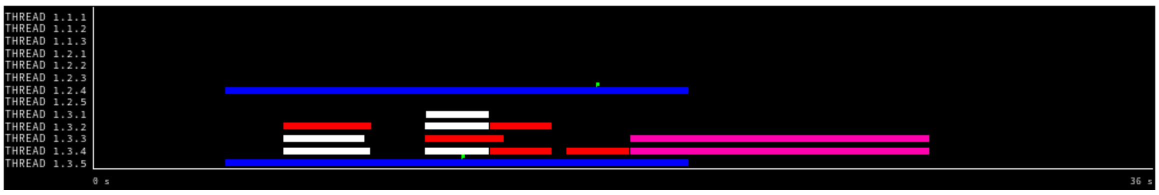
Regarding the software, we have used DistroStream Library (available at [94]), COMPSs version 2.5.rc1909 (available at [55]), and Kafka version 2.3.0 (available at [93]). We have also used Java OpenJDK 8 131, Python 2.7.13, GCC 7.2.0, and Boost 1.64.0.

7.6.2 Gain of processing data continuously

As explained in the first use case in Section 7.5.1, one of the significant advantages when using data streams is to process data continuously as it is generated. For that purpose, Figure 7.13 compares the Paraver [183] traces of the original COMPSs execution (pure task-based workflow) and the execution with streams. Each trace shows the available threads in the vertical axis and the execution time in the horizontal axis - $36s$ in both traces. Also, each colour represents the execution of a task type; corresponding to the colours shown in the task graphs of the first use case (see Section 7.5.1). The green flags indicate when a simulation has generated all its output files and has closed its associated writing stream. Both implementations are written in Python, and the *Directory Monitor* is set as stream backend.



(a)



(b)

FIGURE 7.13: Paraver traces to illustrate the gain of processing data continuously: (a) Original COMPSs execution, (b) Execution with streams.

In contrast to the original COMPSs execution (a), the one with streams (b) executes the processing tasks (white and red) while the simulations (blue) are still running; significantly reducing the total execution time and increasing the resources utilisation. Moreover, the `merge_reduce` tasks (pink) are able to begin its execution even before the simulation tasks are finished, since all the streams have been closed and the `process_sim_file` tasks have already finished.

In general terms, the gain of the implementation with streams with respect to the original COMPSs implementation is proportional to the number of tasks that can be executed in parallel while the simulation is active. Therefore, we perform an in-depth analysis of the trade-off between the generation and process times. It is worth mentioning that we define the *generation time* as the time elapsed between the generation of two elements of the simulation. Hence, the total duration of the simulation is the generation time multiplied by the number of generated elements. Also, the *process time* is defined as the time to process a single element (that is, the duration of the `process_sim_file` task).

The experiment uses 2 nodes of 48 cores each. Since the COMPSs master reserves 12 cores, there are two available workers with 36 and 48 cores respectively. The simulation is configured to use 48 cores, leaving for the other tasks 36 cores while it is active and 84 cores when it is over. Also, the process tasks are configured to use one single core.

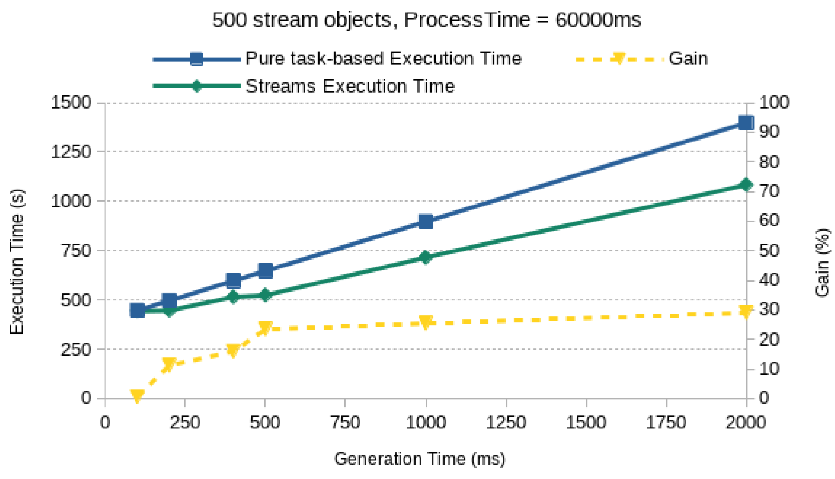


FIGURE 7.14: Average execution time and gain of a simulation with increasing generation time.

Figure 7.14 depicts the average execution time of 5 runs where each simulation generates 500 elements. The process time is fixed to 60,000 ms, while the generation time between stream elements varies from 100 ms to 2,000 ms. For short generation times, almost all the processing tasks are executed when the generation task has already finished, obtaining no gain with respect to the implementation with objects. For instance, when generating elements every 100 ms, the simulation takes 50,000 ms in total ($500 elements \cdot 100 ms/element$). Since the process tasks last 60,000 ms, none of them will have finished before the simulation ends; leading to almost no gain.

When increasing the generation time, more and more tasks can be executed while the generation is still active; achieving a 23% gain when generating stream elements every 500 ms. However, the gain is limited because the last generated elements are always processed when the simulation is over. Therefore, increasing the generation time from 500 ms to 2,000 ms only raises the gain from 23% to 29%.

On the other hand, Figure 7.15 illustrates the average execution time of 5 runs that generate 500 process tasks with a fixed generation time of 100 ms and a process time varying from 5,000 ms up to 60,000 ms. Notice that the total simulation time is 50,000 ms. When

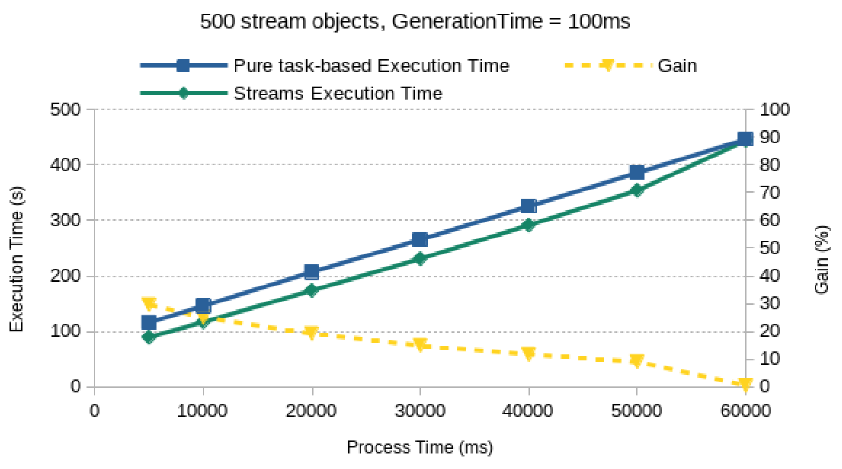


FIGURE 7.15: Average execution time and gain of a simulation with increasing process time.

the processing time is short, many tasks can be executed while the generation is still active; achieving a maximum 30% gain when the processing time is 5,000ms.

As with the previous case, when the processing time is increased, the number of tasks that can be executed while the generation is active also decreases and, thus, the gain. Also, the gain is almost zero when the processing time is big enough (60,000ms) so that none of the process tasks will have finished before the generation ends.

7.6.3 Gain of removing synchronisations

Many workflows are composed of several iterative computations running simultaneously until a certain convergence criterion is met. As described in Section 7.5.2, this technique is useful when performing parameter sweep, cross-validation, or running the same algorithm with different initial points.

To this end, each computation requires a phase at the end of each iteration to exchange information with the rest. When using pure task-based workflows, this phase requires to stop all the computations at the end of each iteration, retrieve all the states, create a task to exchange and update all the states, transfer back all the new states, and resume all the computations to the next iteration. The left task graph of Figure 7.16 shows an example of such workflows with two iterations of two parallel computations. The first two red tasks initialise the state of each computation, the pink tasks perform the computation of each iteration, and the blue tasks retrieve and update the state of each computation.

Conversely, when using Hybrid Workflows, each computation can exchange the information at the end of each iteration asynchronously by writing and reading the states to/from streams. This technique avoids splitting each computation into tasks, stopping and resuming each computation at every iteration, and synchronising all the computations to exchange data. The right task graph of Figure 7.16 depicts the equivalent Hybrid Workflow of the previous example. Each computation is run in a single task (white) that performs the state initialisation, all the iterations, and all the update phases at the end of each iteration.

Using the previous examples, Figure 7.17 evaluates the performance gain of avoiding the synchronisation and exchange process at the end of each iteration. Hence, the benchmark executes the pure task-based workflow (blue) and the Hybrid Workflow (green) versions of the same workflow written in Java and using Kafka as streaming backend. Also, it is

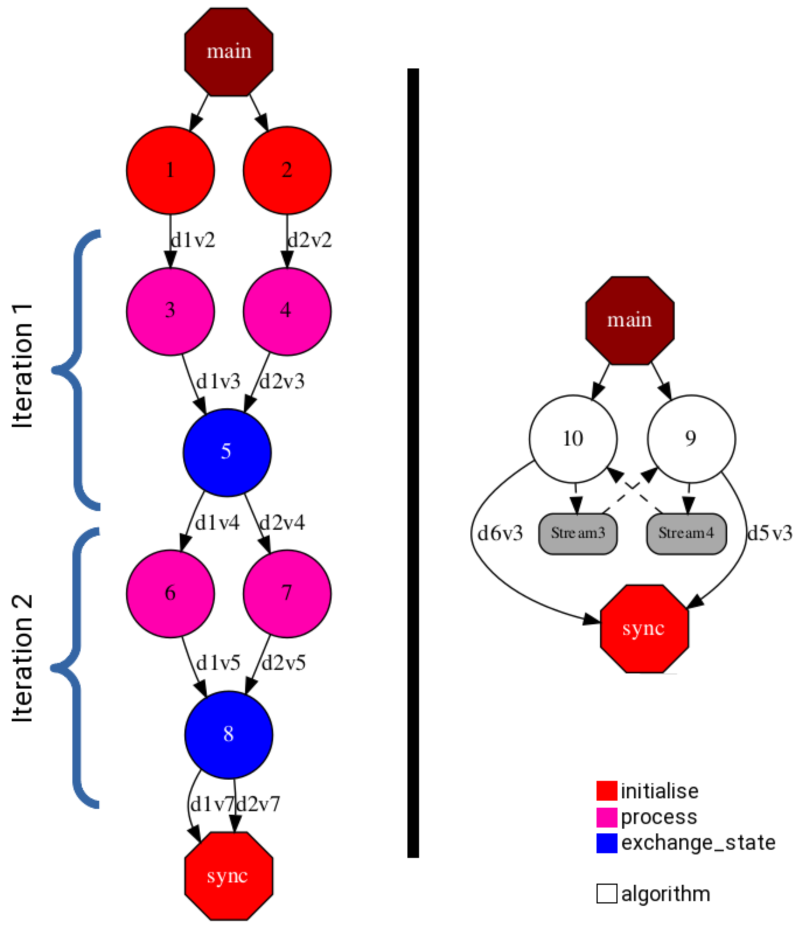


FIGURE 7.16: Parallel iterative computations. Pure task-based workflow and Hybrid Workflow shown at left and right, respectively.

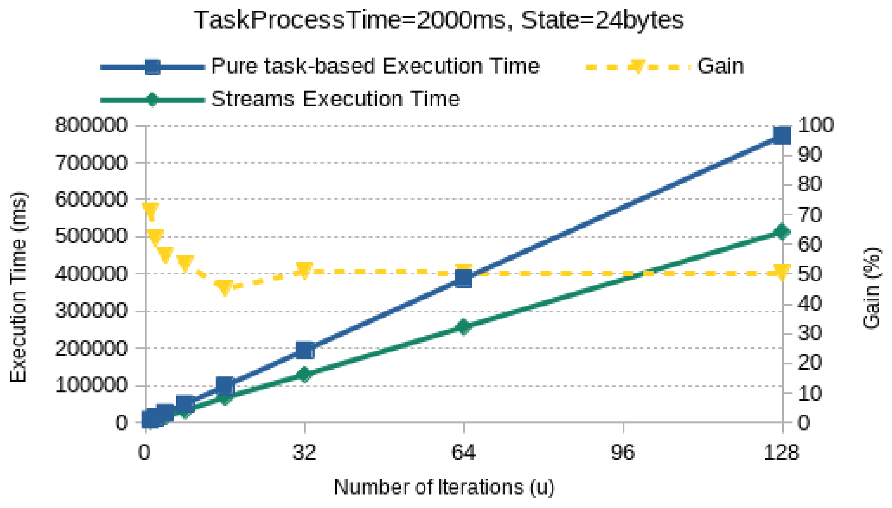


FIGURE 7.17: Average execution time and gain of a simulation with an increasing number of iterations.

composed of two independent computations with a fixed computation per iteration (2,000 ms) and an increasing number of iterations. The results shown are the mean execution times of 5 runs of each configuration.

Notice that the total gain is influenced by three factors: the removal of the synchronisation task at the end of each iteration, the cost of transferring the state between the process and the synchronisation tasks, and, the division of the state's initialisation and process. Although we have reduced the state of each computation to 24 bytes and used a single worker machine to minimise the impact of the transfer overhead, the second and third factors become important when running a small number of iterations (below 32), reaching a maximum gain of 71% when running a single iteration. For a larger number of iterations (over 32), the removal of the synchronisation becomes the main factor, and the total gain reaches a steady state with a gain around 50%.

7.6.4 Stream writers and readers scalability and load balance

Our prototype supports N-M streams, meaning that any stream can have an arbitrary amount of writers and readers. To evaluate the performance and load balance, we have implemented a Java application that uses a single stream and creates N writer tasks and M reader tasks. Although our writer and reader tasks use a single core, we spawn each of them in separated nodes so that the data must be transferred. In more sophisticated use cases, each task could benefit from an intra-node technology (such as OpenMP) to parallelise the processing of the stream data.

Figures 7.18 and 7.19 depict the average execution time and the efficiency of 5 runs with an increasing number of readers, respectively. Each series uses a different number of writers, also going from 1 to 8. Also, the writers publish 100 elements in total, the size of the published objects is 24 bytes, and the time to process an element is set to 1,000ms. The efficiency is calculated using the ideal execution time as reference; i.e. the number of elements multiplied by the time to process an element and divided by the number of readers.

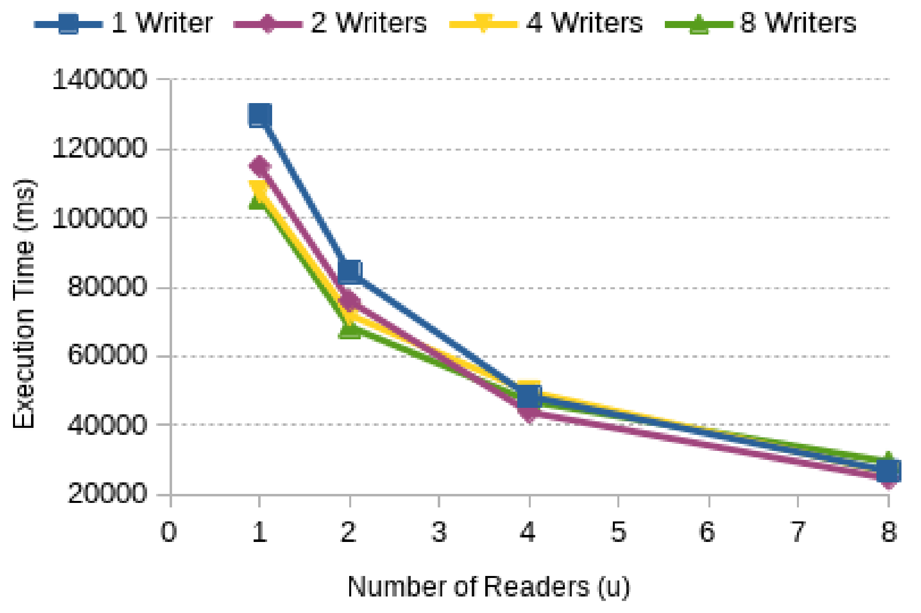


FIGURE 7.18: Average execution time with increasing number of readers and different number of writers.

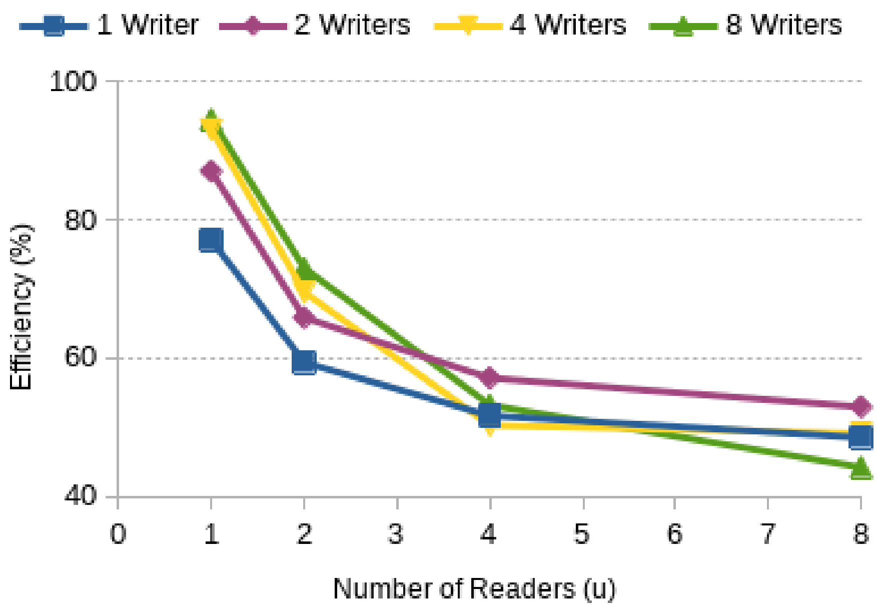


FIGURE 7.19: Efficiency with increasing number of readers and different number of writers.

Since the execution time is mainly due to the processing of the elements in the reader tasks, when increasing the number of writers, there are no significant differences. However, for all the cases, increasing the number of readers significantly impacts the execution time, achieving a 4.84 speed-up with 8 readers. Furthermore, the efficiencies using 1 reader are close to the ideal (87% on average) because the only overheads are the creation of the elements, the task spawning, and the data transfers. However, when increasing the number of readers, the load imbalance significantly affects efficiency; achieving around 50% efficiency with 8 readers.

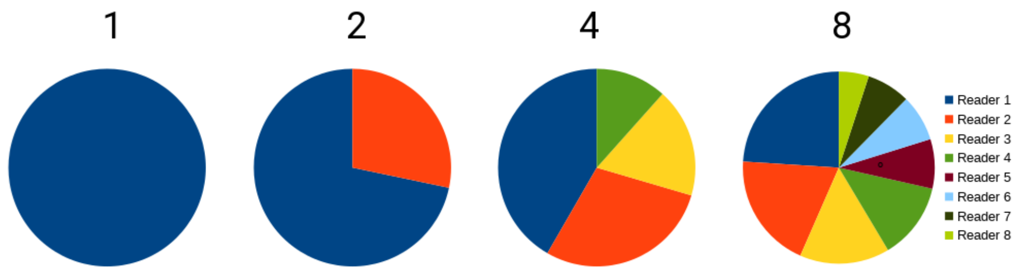


FIGURE 7.20: Number of stream elements processed per reader.

It is worth mentioning that the achieved speed-up is lower than the ideal (8) due to load imbalance. Thus, the elements processed by each reader task are not balanced since elements are assigned to the first process that requests them. Figure 7.20 illustrates an in-depth study of the load imbalance when running 1, 2, 4, or 8 readers. Notice that when running with 2 readers, the first reader gets almost 75% of the elements while the second one only processes 25% of the total load. The same pattern is shown when increasing the number of readers; where half of the tasks perform 70% of the total load. For instance, when running with 4 readers, 2 tasks perform 69% of the work (34.5% each), while the rest only performs 31% of the total load (15.5% each). Similarly, when running with 8 readers, 4 tasks perform 70% of the total load (17.5% each), while the other four only process 30% (7.5% each).

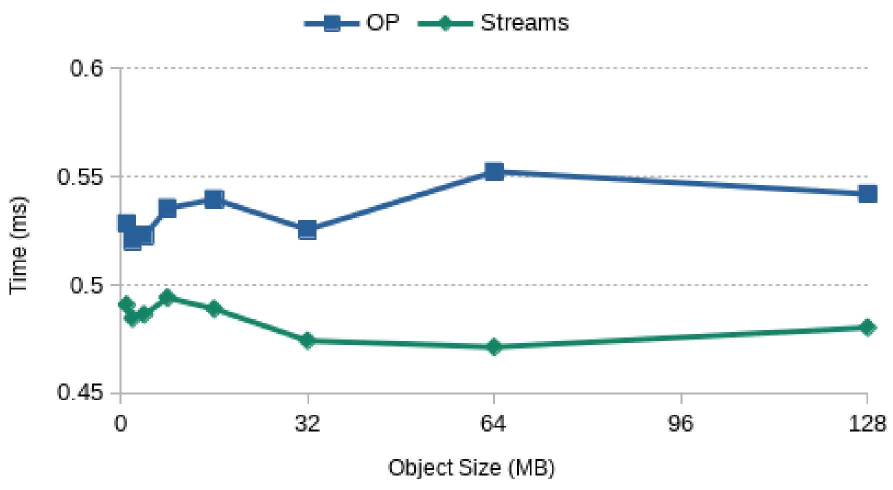
At its current state, the *Distributed Stream Library* does not implement any load balance technique, nor limit the number of elements retrieved by each `poll` call. As future work, since the library already stores the processes registered to each stream, it could implement some policy and send only a subset of the available elements to the requesting process rather than all of them.

7.6.5 Runtime overhead

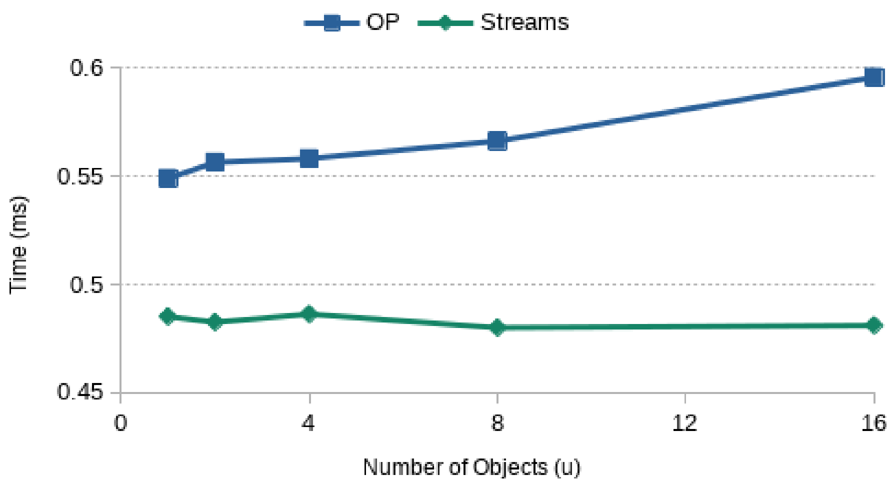
To provide a more in-depth analysis of the performance of our prototype, we have compared each step of the task life-cycle when using `ObjectParameters` (OP from now on) or `ObjectDistroStreams` (streams from now on). The following figures evaluate the task analysis, task scheduling, and task execution average times of 100 tasks using (a) a single object of increasing size (from 1 MB to 128 MB) or (b) an increasing number of objects (from 1 to 16 objects) of fixed size (8 MB). Both implementations are written in Java and Kafka is used as the stream backend. Regarding the task definition, notice that the OP implementation requires an `ObjectParameter` for each object sent to the task. In contrast, the streams implementation only requires a single `StreamParameter` since all the objects are sent through the stream itself.

Figure 7.21 compares the task analysis results. The task analysis is the time spent by the runtime to register the task and its parameters into the system. It is worth mentioning that increasing the object's size does not affect the analysis time in either the OP nor the streams implementations. There is, however, a difference around 0.05 ms due to the creation of internal structures to represent object parameters or stream parameters.

On the other hand, increasing the number of objects directly affects the task analysis time because the runtime needs to register each task parameter individually. For the OP implementation, each object maps to an extra task parameter, and thus, the task analysis time slightly increases when increasing the number of objects. Conversely, for the streams implementation, the stream parameter itself is not modified since we only increase the number of published objects. Hence, the task analysis time remains constant when increasing the number of objects.



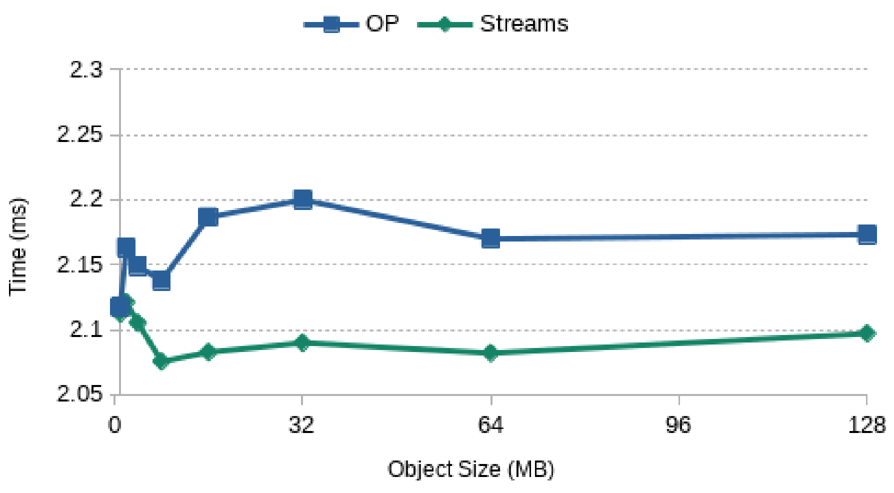
(a)



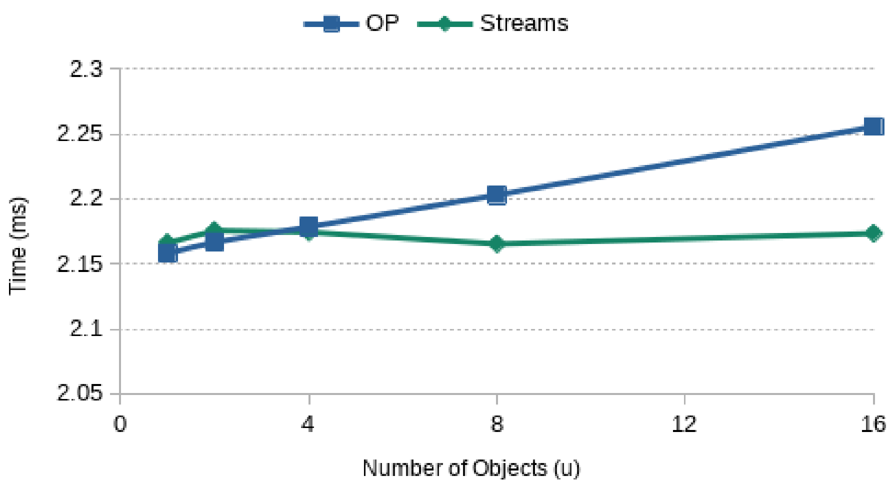
(b)

FIGURE 7.21: Task analysis average time for one single parameter with increasing sizes (a) or increasing number of parameters (b).

Figure 7.22 compares the task scheduling results. On the one hand, the scheduling time for both implementations varies from 2.05 ms to 2.20 ms but does not show any clear tendency to increase when increasing the object's size. On the other hand, when increasing the number of objects, the scheduling time increases for the OP implementation and remains constant for the streams implementation. This behaviour is due to the fact that the default COMPSs scheduler implements data locality and, thus, the scheduling time is proportional to the number of parameters. Similarly to the previous case, increasing the number of objects increases the number of task parameters for the OP implementation (increasing its scheduling time), but keeps a single parameter for the streams implementation (maintaining its scheduling time).



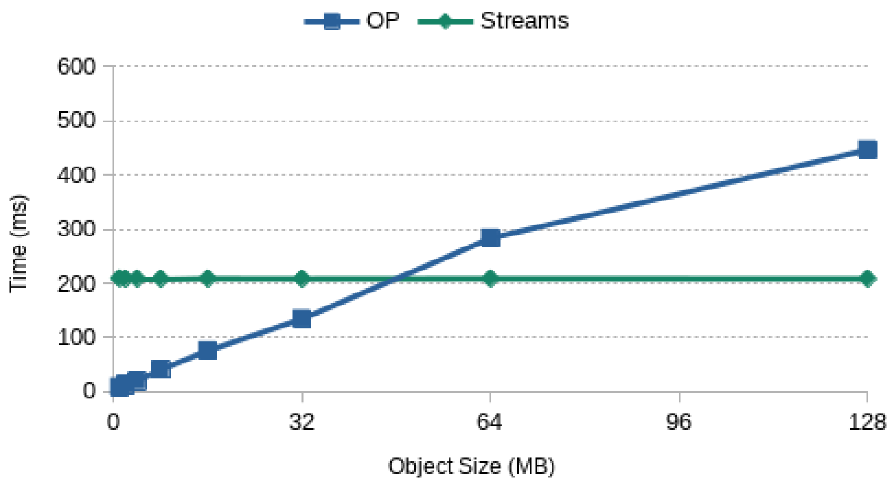
(a)



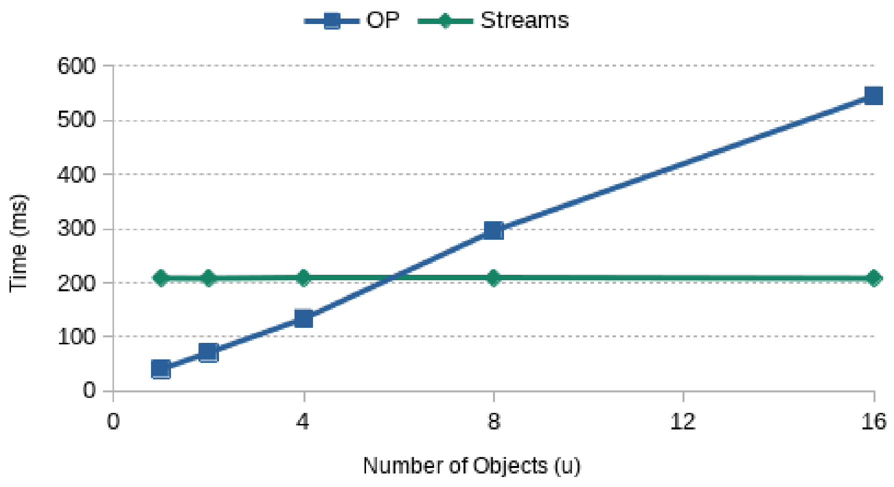
(b)

FIGURE 7.22: Task scheduling average time for one single parameter with increasing sizes (a) or increasing number of parameters (b).

Figure 7.23 compares the task execution results. The task execution time covers the transfer of all the task parameters and the task execution itself. Regarding the streams implementation, the time remains constant around 208 ms regardless of the object's size and the number of objects because the measurement only considers the transfer of the stream object itself and the execution time of the `poll` method. It is worth mentioning that the actual transfers of the objects are done by Kafka when invoking the `publish` method on the main code, and thus, they are executed in parallel while COMPSs spawns the task in the worker machine.



(a)



(b)

FIGURE 7.23: Task execution average time for one single parameter with increasing sizes (a) or increasing number of parameters (b).

Conversely, the execution time for the OP implementation increases with both the object's size and the number of objects, since the serialisation and transfer times also increase. However, the task execution does not need to fetch the objects (the `poll` method) since all of them have already been transferred. This trade-off can be observed in the figure, where the OP implementation performs better than the streams implementation when using task parameters smaller than 48 MB and performs worse for bigger cases. Notice that only the total objects' size is relevant since the same behaviour is shown when using a single 48 MB object or 6 objects of 8 MB each.

Since the real object transfers when using streams are executed during the `publish` method and cannot be observed measuring the task execution time, we have also measured the total execution time of the benchmark for both implementations. Figure 7.24 shows the total execution time with an increasing number of objects of 8 MB. In contrast to the previous plot, both implementations have an increasing execution time proportional to the objects' size. Also, the streams implementation only outperforms the OP implementation when using more than 12 objects.

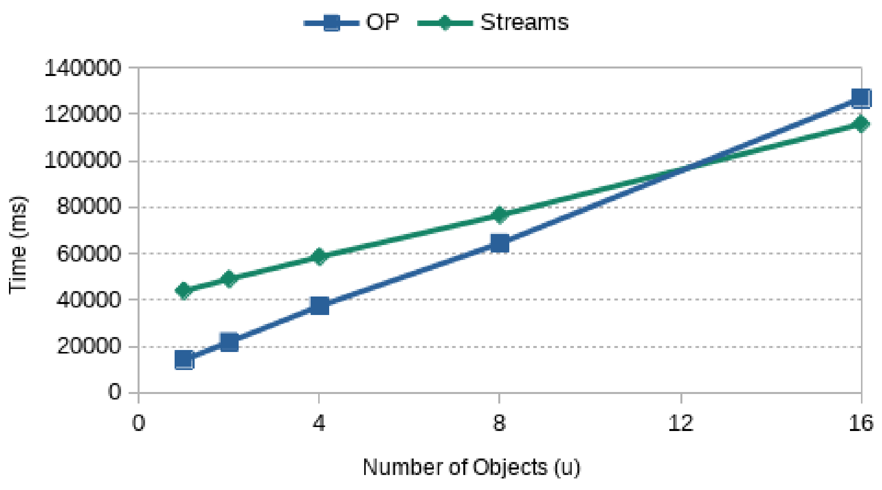


FIGURE 7.24: Total execution time with increasing number of parameters.

To conclude, since there are no major differences regarding the task analysis time nor the task scheduling time, we can safely assume that the use of streams instead of regular objects is recommended when the total size of the task parameters exceeds 48 MB and there are more than 12 objects published to the stream.

7.7 Discussion

This part of the thesis demonstrates that task-based workflows and dataflows can be integrated into a single programming model to better-cover the needs of the new Data Science workflows. Using Hybrid Workflows, developers can build complex pipelines with different approaches at many levels using a single framework.

The proposed solution relies on the *DistroStream* concept: a generic API used by applications to handle stream accesses homogeneously regardless of the software backing it. Two implementations provide the specific logic to support object and file streams. The first one, *ObjectDistroStream*, is built on top of Kafka to enable object streams. The second one, *FileDistroStream*, monitors the creation of files inside a directory, sends the file locations through the stream, and relies on a distributed file system to share the file content.

The *DistroStream* API and both implementations are part of the *DistroStreamLib*, which also provides the *DistroStream Client* and the *DistroStream Server*. While the client acts as a broker on behalf of the application and interacts with the corresponding backend, the server manages the streams' metadata and coordinates the accesses.

By integrating the *DistroStreamLib* into a task-based Workflow Manager, its programming model can easily support Hybrid Workflows. Our described prototype extends the

COMPSs programming model to enable tasks with continuous input and output data by providing a new annotation for stream parameters. Implementing the handling of such stream-type values lead to some modifications on the Task Analyser and Task Scheduler components of the runtime. Using the *DistroStreamLib* also implied changes at deployment time since its components need to be spawned along with COMPSs. On the one hand, the COMPSs master hosts the *DistroStream Server*, the required stream backend, and a *DistroStream Client* to handle stream accesses on the application's main code. On the other hand, each COMPSs worker contains a *DistroStream Client* that performs the stream accesses on tasks. Although the described prototype only builds on COMPSs, it can be used as an implementation reference for any other existing task-based framework.

This part of the thesis also presents four use cases illustrating the new capabilities that the users may identify in their workflows to benefit from the use of Hybrid Workflows. On the one hand, streams can be internal or external to the application and can be used to communicate continuous data or control data. On the other hand, streams can be accessed inside the main code, native tasks (i.e., Java or Python), or non-native tasks (i.e., MPI, binaries, and nested COMPSs workflows). Furthermore, the *Distributed Stream Library* supports the one to many, many to one, and many to many scenarios transparently, and allows to configure the consumer mode to process the data at least once, at most once, or exactly once when using many consumers.

The evaluation demonstrates the benefit of processing data continuously as it is generated; achieving a 30% gain with the right generation and process times and resources. Also, using streams as control mechanism enabled the removal of synchronisation points when running several parallel algorithms, leading to a 50% gain when running more than 32 iterations. Finally, an in-depth analysis of the runtime's performance shows that there are no major differences regarding the task analysis time nor the task scheduling time when using streams or object tasks, and that the use of streams is recommended when the total size of the task parameters exceeds 48 MB using more than 12 objects.

Although the solution is fully functional, some improvements can be made. Regarding the *DistroStream* implementations, we plan to extend the *FileDistroStream* to support shared disks with different mount-points. On the other hand, we will add new *ObjectDistroStream*'s backend implementations (apart from Kafka), so that users can choose between them without changing the application's code. Similarly, we plan to extend our implementation to use Persistent Self-Contained Object Storages (such as dataClay [150, 62] or Hecuba [6, 216, 105]). Envisaging that a single *DistroStream Server* could become a bottleneck when managing several applications involving a large number of cores, we consider replacing the client-server architecture by a peer-to-peer approach. Finally, by highlighting the benefits from hybrid flows, we expect to attract real-world applications to elevate our evaluation to more complex use cases.

Part III

Conclusions and future work

Chapter 8

Conclusions and future work

8.1 Conclusions

In general terms, this thesis contributes to the adaptation of High-Performance frameworks to support Data Science workflows. Our prototype is capable of (i) orchestrating different frameworks inside a single programming model, (ii) integrating container platforms at static and dynamic level, (iii) enabling the automatic task-based parallelisation of affine loop nests, and (iv) executing Hybrid Workflows (a combination of task-based workflows and dataflows). Moreover, this thesis eases the development of distributed applications for intermediate-level users by providing simple annotations and interfaces that abstract the developers from the parallel and distributed challenges.

Next, we provide a general overview of the conclusions discussed at the end of each part of the thesis. First, regarding Chapter 4, the `@binary`, `@OmpSs`, `@MPI`, `@COMPSs`, and `@MultiNode` annotations allow to easily orchestrate different frameworks inside a single programming model. Hence, the users can build complex workflows where some steps require highly optimised state of the art frameworks. Moreover, these annotations are designed to be easily extended to include new frameworks in the future. This contribution makes advances in the state of the art since it is the first Task-based workflows with implicit task definition to support the transparent execution of external binaries, tools, and libraries. Also, during the evaluation, we have ported the NMMB-MONARCH application in both Java and Python; demonstrating (in both cases) a huge increase in programmability while maintaining the same performance.

Second, regarding Chapter 5, we integrate container engines considering three different scenarios. On the one hand, the static container management focuses on the creation of a Docker image (that includes the application software and the programming model runtime), and the orchestration of the deployment and execution of the application using container resources. On the other hand, the HPC container management extends the use of containers to supercomputers with minor modifications from the users' point of view. Finally, the dynamic container management fully exploits container engines by enabling adaptation mechanisms to quickly adapt the available resources to the remaining workload during the application's execution. For each scenario, we have implemented a use case using Docker, Singularity, or Mesos. Overall, the evaluation demonstrates that all the container engines keep the application's execution scalability while significantly reducing the deployment overhead; thus enabling our prototype to adapt better the resources to the computational load.

Third, regarding Chapter 6, we have developed AutoParallel as a core part of our prototype to automatically parallelise affine loop nests and execute them in distributed infrastructures. Since it is based on sequential programming and only requires a single annotation (the `@parallel` Python decorator), anyone with intermediate-level programming skills can scale up an application to hundreds of cores. More specifically, the evaluation shows that the codes automatically generated by the AutoParallel module for the Cholesky, LU, and QR applications can achieve similar performance than manually parallelised versions without requiring any effort from the programmer. Moreover, the loop taskification automatically builds data blocks from loop tiles and increase the tasks' granularity. This is of particular interest when applied to fine-grain applications (e.g., where the tasks' granularity is a single float operation) since it frees the users from dealing directly with the complexity of block algorithms and allows them to stick to basic sequential programming. Also, for advanced users, the generated code can be used as a baseline to design complex blocked algorithms.

Finally, regarding Chapter 7, we extend our prototype to support the combination of task-based workflows and dataflows (Hybrid Workflows) using a single programming model so that developers can build complex pipelines with different approaches at many levels. This contribution makes advances in the state of the art since Hybrid Workflows are not supported in similar environments. To this end, we have built the Distributed Stream Library (*DistroStreamLib*) that includes: (i) a generic *DistroStream* API to handle stream accesses homogeneously regardless of the software backing it, (ii) two implementations to provide the specific logic to support object and file streams (in both Java and Python), (iii) the *DistroStream Client* that acts as a broker on behalf of the application and interacts with the corresponding backend, and (iv) the *DistroStream Server* that manages the streams' metadata and coordinates the accesses. During the evaluation, we present four use cases illustrating the new capabilities that the users may identify in their workflows to benefit from the use of Hybrid Workflows. More specifically, we demonstrate the benefit of processing data continuously as it is generated, the benefit of using streams as a control mechanism to remove synchronisation points when running several parallel algorithms, and analyse the runtime's performance to recommend the use of streams in terms of the number and size of the task parameters.

8.2 Future work

Even if this thesis provides a set of mechanisms to adapt High-Performance frameworks to support Data Science workflows, there is still room for improvement when preventing users from dealing directly with distributed and parallel computing issues. In this chapter, we provide a general overview of the future work discussed at the end of each part of the thesis.

First, regarding Chapter 4, we plan to extend the annotations to support other state-of-the-art frameworks. For instance, we consider integrating other task-based frameworks (such as Apache Spark [237]), and frameworks targeting GPUs (such as CUDA [170] and OpenCL [162]). Also, while working with real-world use cases, we have noticed that many binaries require specific names for input and output parameters, and to pack or unpack input and output files. Thus, we plan to provide more tools and options for binary parameters.

Second, regarding Chapter 5, although containers provide efficient image management and application deployment tools that facilitate the packaging and distribution of applications, the performance on some scenarios can be improved. Thus, we plan to evaluate experimental alternatives for Docker multi-host networking [73] to test if our prototype with Docker can perform better than KVM in all situations. Also, regarding Mesos, we plan to perform bigger tests to evaluate the scalability, the adaptation capabilities with dynamically added slave nodes, and the networking issues to understand the source of overhead. On the other hand, we plan to support Kubernetes [134] and to enable the orchestration of applications where each task requires a separated container. This feature will allow the users to prepare before-hand a different image with only the required software and dependencies for each task; potentially achieving better deployment times.

Third, regarding Chapter 6, although the Loop Taskification provides an automatic way to create blocks from sequential applications, its performance can be improved. For instance, PyCOMPSs could lower the serialisation overhead by serialising each element of the collection in parallel or by serialising the whole collection into a single object. Also, AutoParallel could be integrated with different tools similar to Pluto to support a broader scope of loop nests (such as APOLLO [213, 151]).

Finally, regarding Chapter 7, we plan to extend the *FileDistroStream* to support shared disks with different mount-points. Also, we will add new *ObjectDistroStream*'s backend implementations (apart from Kafka), so that users can choose between them without changing the application's code. Similarly, we plan to extend our implementation to use Persistent Self-Contained Object Storages (such as dataClay [150, 62] or Hecuba [6, 216, 105]). On the other hand, envisaging that a single *DistroStream Server* could become a bottleneck when managing several applications involving a large number of cores, we consider replacing the client-server architecture by a peer-to-peer approach.

Part IV

Bibliography

Bibliography

[1] M. Abadi and et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[2] Mohamed Abouelhoda, Shadi Alaa Issa, and Moustafa Ghanem. "Tavaxy: Integrating Taverna and Galaxy workflows with cloud computing support". In: *BMC bioinformatics* 13.1 (2012), p. 77.

[3] *Academic Free License version 3.0 (Opensource.org, 2005)*. Retrieved from https://opensource.org/licenses/AFL-3.0. Accessed 2 October, 2019.

[4] *Advanced Multi-layered unification filesystem*. Retrieved from https://aufs.sourceforge.net. Accessed 11 April, 2017.

[5] E. Afgan and et al. "The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update". In: *Nucleic acids research* (2016), gkw343. DOI: 10.1093/nar/gkw343.

[6] G. Alomar, Y. Becerra, and J. Torres. "Hecuba: Nosql made easy". In: *BSC Doctoral Symposium (2nd: 2015: Barcelona)*. Barcelona Supercomputing Center, 2015, pp. 136–137.

[7] R. Amela and et al. "Enabling Python to Execute Efficiently in Heterogeneous Distributed Infrastructures with PyCOMPSs". In: *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing*. Denver, CO, USA: ACM, 2017, 1:1–1:10. ISBN: 978-1-4503-5124-9. DOI: 10.1145/3149869.3149870.

[8] R. Amela and et al. "Executing linear algebra kernels in heterogeneous distributed infrastructures with PyCOMPSs". In: *Oil |& Gas Science and Technology - Revue d'IFP Energies Nouvelles (OGST)* (2018). DOI: 10.2516/ogst/2018047.

[9] E. Anderson and et al. *LAPACK Users' guide*. SIAM, 1999.

[10] J. Anubhav and et al. "FireWorks: a dynamic workflow system designed for high-through-put applications". In: *Concurrency and computation: practice and experience* 27 (May 2015). DOI: 10.1002/cpe.3505.

[11] *Apache Airflow (The Apache Software Foundation, 2017)*. Retrieved from http://airflow.apache.org. Accessed 15 December, 2017.

[12] *Apache Apex (The Apache Software Foundation, 2017)*. Retrieved from https://apex.apache.org. Accessed 15 December, 2017.

[13] *Apache Beam (The Apache Software Foundation, 2017)*. Retrieved from https://beam.apache.org. Accessed 18 September, 2017.

[14] *Apache CloudStack - Open Source Cloud Computing (The Apache Software Foundation, 2017)*. Retrieved from https://cloudstack.apache.org. Accessed 3 December, 2019.

[15] *Apache Crunch (The Apache Software Foundation, 2017)*. Retrieved from https://crunch.apache.org. Accessed 15 December, 2017.

[16] *Apache Flink (Apache Flink Contributors, 2019)*. Retrieved from https://flink.apache.org. Accessed 9 August, 2019.

[17] *Apache Gearpump (The Apache Software Foundation, 2017)*. Retrieved from https://gearpump.apache.org. Accessed 15 December, 2017.

[18] *Apache Hadoop YARN (The Apache Software Foundation, 2019)*. Retrieved from https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html. Accessed 3 December, 2019.

[19] *Apache jClouds (The Apache Software Foundation, 2014)*. Retrieved from https://jclouds.apache.org/. Accessed 4 May, 2020.

[20] *Apache Kafka: A distributed streaming platform (The Apache Software Foundation, 2017)*. Retrieved from https://kafka.apache.org. Accessed 30 November, 2017.

[21] *Apache License, version 2.0 (The Apache Software Foundation, 2019)*. Retrieved from https://www.apache.org/licenses/LICENSE-2.0. Accessed 2 October, 2019.

[22] *Apache Maven (The Apache Software Foundation, 2017)*. Retrieved from https://maven.apache.org. Accessed 20 July, 2017.

[23] *Apache Mesos (The Apache Software Foundation, 2018)*. Retrieved from http://mesos.apache.org. Accessed 3 December, 2019.

[24] *Apache Samza (The Apache Software Foundation, 2017)*. Retrieved from http://samza.apache.org. Accessed 9 August, 2019.

[25] *Apache Taverna (Taverna Committers, 2019)*. Retrieved from https://taverna.incubator.apache.org. Accessed 1 April, 2019.

[26] *Apache ZooKeeper (The Apache Software Foundation, 2017)*. Retrieved from https://zookeeper.apache.org. Accessed 1 December, 2017.

[27] M. Armbrust and et al. "Above the clouds: A berkeley view of cloud computing". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28* (2009).

[28] D. Armstrong and et al. "Contextualization: dynamic configuration of virtual machines". In: *Journal of Cloud Computing* 4.1 (2015), p. 1.

[29] K. Asanovic and et al. "A view of the Parallel Computing Landscape". In: *Communications of the ACM* 52.10 (Oct. 2009), pp. 56–67. DOI: 10.1145/1562764.1562783.

[30] K. Asanovic and et al. "The landscape of parallel computing research: A view from berkeley". In: *Technical Report UCB/EECS-2006-183* 2 (2006).

[31] *Autosubmit (Barcelona Supercomputing Center, 2020)*. Retrieved from https://www.bsc.es/research-and-development/software-and-apps/software-list/autosubmit. Accessed 19 February, 2020.

[32] R. M. Badia and et al. "COMP superscalar, an interoperable programming framework". In: *SoftwareX* 3 (Dec. 2015), pp. 32–36. DOI: 10.1016/j.softx.2015.10.004.

[33] P. Barham and et al. "Xen and the art of virtualization". In: *ACM SIGOPS Operating Systems Review*. Vol. 37. 5. ACM. 2003, pp. 164–177. DOI: 10.1145/1165389.945462.

[34] C. Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think". In: IEEE Computer Society, Sept. 2004, pp. 7–16. DOI: 10.1109/PACT.2004.1342537.

[35] C. Bastoul. *OpenScop: A Specification and a Library for Data Exchange in Polyhedral Compilation Tools*. Tech. rep. Paris-Sud University, France, Sept. 2011. URL: http://icps.u-strasbg.fr/people/bastoul/public_html/development/openscop/docs/openscop.html.

[36] C. Bastoul and et al. "Putting Polyhedral Loop Transformations to Work". In: Springer, 2004, pp. 209–225. DOI: 10.1007/978-3-540-24644-2_14.

[37] P. Bientinesi, B. Gunter, and R. A. van de Geijn. "Families of Algorithms Related to the Inversion of a Symmetric Positive Definite Matrix". In: *ACM Trans. Math. Softw.* 35.1 (July 2008), 3:1–3:22. DOI: 10.1145/1377603.1377606.

[38] *BLAS, Basic Linear Algebra Subprograms (University of Tennesse. Oak Ridge National Laboratory. Numerical Algorithms Group Ltd., 2017)*. Retrieved from http://www.netlib.org/blas. Accessed 8 October, 2019.

[39] U. Bondhugula and et al. "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model". In: Springer, Apr. 2008, pp. 132–146. DOI: 10.1007/978-3-540-78791-4_9.

[40] U. Bondhugula et al. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *SIGPLAN Not.* 43.6 (June 2008), pp. 101–113. DOI: 10.1145/1375581.1375595.

[41] D. Bruneo and et al. "CloudWave: Where adaptive cloud management meets DevOps". In: *2014 IEEE Symposium on Computers and Communications (ISCC)*. IEEE. 2014, pp. 1–6.

[42] *BSD License (The Linux Information Project, 2005)*. Retrieved from http://www.linfo.org/bsdlicense. Accessed 2 October, 2019.

[43] *Cascading (Cascading Maintainers, 2017)*. Retrieved from http://www.cascading.org. Accessed 15 December, 2017.

[44] S. Cass. *The Top Programming Languages 2019: Python remains the big kahuna, but specialist languages hold their own*. Cited 19 December 2019. 2019. URL: https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019.

[45] *Celery (Ask Solem, 2017)*. Retrieved from http://www.celeryproject.org. Accessed 15 December, 2017.

[46] *Chameleon Cloud Project (TACC, 2017)*. Retrieved fromhttps://www.chameleoncloud.org. Accessed 11 April, 2017.

[47] *Chameleon Cloud Project (TACC, 2017)*. Retrieved from https://www.chameleoncloud.org/about/hardware-description. Accessed 11 April, 2017.

[48] R. Chandra and et al. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[49] *Chef (Chef Software Inc., 2020)*. Retrieved from https://www.chef.io. Accessed 5 May, 2020.

[50] S. Chiba. "Load-time Structural Reflection in Java". In: *ECOOP 2000 - Object-Oriented Programming* 1850 (May 2000), pp. 313–336. DOI: 10.1007/3-540-45102-1_16.

[51] *Chronos Scheduler for Mesos (Mesos Contributors, 2017)*. Retrieved from https://mesos.github.io/chronos. Accessed 11 April, 2017.

[52] *Cloud-init (Canonical Ltd., 2020)*. Retrieved from https://launchpad.net/cloud-init. Accessed 5 May, 2020.

[53] A. Cohen and et al. "Facilitating the Search for Compositions of Program Transformations". In: ACM, 2005, pp. 151–160. DOI: 10.1145/1088149.1088169.

[54] *COMP Superscalar, COMPSs (Barcelona Supercomputing Center, 2017)*. Retrieved from https://compss.bsc.es. Accessed 15 December, 2017.

[55] *COMPSs GitHub (Barcelona Supercomputing Center, 2018)*. Retrieved from https://github.com/bsc-wdc/compss. Accessed 4 June, 2018.

[56] J. Conejero et al. “Task-based programming in COMPSs to converge from HPC to big data”. In: *The International Journal of High Performance Computing Applications* 32.1 (2018), pp. 45–60. DOI: 10.1177/1094342017701278.

[57] *Container Network Interface - Networking for Linux containers (Cloud Native Computing Foundation, 2020)*. Retrieved from https://github.com/containernetworking/cni. Accessed 4 May, 2020.

[58] L. Dagum and R. Menon. “OpenMP: An Industry-Standard API for Shared-Memory Programming”. In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), pp. 46–55. DOI: 10.1109/99.660313.

[59] L. Dalcín, R. Paz, and M. Storti. “MPI for Python”. In: *Journal of Parallel and Distributed Computing* (2005). DOI: https://doi.org/10.1016/j.jpdc.2005.03.010.

[60] S. Das and et al. “Next-generation genotype imputation service and methods”. In: *Nature genetics* 48.10 (2016), p. 1284.

[61] Dask Development Team. *Dask: Library for dynamic task scheduling*. 2016. URL: http://dask.pydata.org.

[62] *dataClay (Barcelona Supercomputing Center, 2019)*. Retrieved from https://www.bsc.es/research-and-development/software-and-apps/software-list/dataclay. Accessed 4 December, 2019.

[63] J. Dean and S. Ghemawat. “Mapreduce: Simplified data processing on large clusters”. In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation* 6 (2004), pp. 10–10. DOI: 10.1145/1327452.1327492.

[64] J. Dean and S. Ghemawat. “Mapreduce: Simplified data processing on large clusters”. In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation* 6 (2004), pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[65] E. Deelman. “Big Data Analytics and High Performance Computing Convergence Through Workflows and Virtualization”. In: *Big Data and Extreme-Scale Computing* (2016).

[66] E. Deelman and et al. “Pegasus, a workflow management system for science automation”. In: *Future Generation Computer Systems* 46 (2015), pp. 17–35. DOI: 10.1016/j.future.2014.10.008.

[67] E. Deelman et al. “Workflows and e-Science: An over-view of workflow system features and capabilities”. In: *Future Generation Computer Systems* 25.5 (2009), pp. 528 – 540. DOI: 10.1016/j.future.2008.06.012.

[68] O. Delaneau, J. Marchini, and J. Zagury. “A linear complexity phasing method for thousands of genomes”. In: *Nature methods* 9.2 (2012), p. 179.

[69] J. W. Demmel and N. J. Higham. “Stability of Block Algorithms with Fast Level-3 BLAS”. In: *ACM Trans. Math. Softw.* 18.3 (Sept. 1992), pp. 274–291. DOI: 10.1145/131766.131769.

[70] P. Di Tommaso and et al. “The impact of Docker containers on the performance of genomic pipelines”. In: *PeerJ* 3 (July 2015). Ed. by Fabien Campagne, e1273. DOI: 10.7717/peerj.1273.

[71] Paolo Di Tommaso et al. “Nextflow enables reproducible computational workflows”. In: *Nature biotechnology* 35.4 (2017), p. 316.

[72] *Docker (Docker Inc., 2017)*. Retrieved from `https://www.docker.com`. Accessed 11 April, 2017.

[73] *Docker Plugins (Docker Inc., 2017)*. Retrieved from `https://docs.docker.com/engine/extend/legacy_plugins`. Accessed 11 April, 2017.

[74] J. Dongarra and et al. "The international Exascale Software Project roadmap". In: *International Journal of High Performance Computing Applications* 25.1 (Feb. 2011), pp. 3–60. DOI: `10.1177/1094342010391989`.

[75] A. Duran and et al. "Ompss: a proposal for programming heterogeneous multi-core architectures". In: *Parallel processing letters* 21.02 (2011), pp. 173–193. DOI: `10.1142/S0129626411000151`.

[76] *EcFlow (Atlassian, 2017)*. Retrieved from `https://software.ecmwf.int/wiki/display/ECFLOW/ecflow+home`. Accessed 2 August, 2017.

[77] *Eclipse IDE (Eclipse Foundation Inc., 2017)*. Retrieved from `https://eclipse.org`. Accessed 20 July, 2017.

[78] *Eclipse Public License v1.0 (Eclipse Foundation Inc., 2019)*. Retrieved from `https://www.eclipse.org/legal/epl-v10.html`. Accessed 2 October, 2019.

[79] Jorge Ejarque and et al. "Service Construction Tools for Easy Cloud Deployment". In: *7th IBERIAN Grid Infrastructure Conference Proceedings*, p. 119.

[80] *Enduro/X (Mavimax, 2015)*. Retrieved from `https://www.endurox.org`. Accessed 3 December, 2019.

[81] *Eucalyptus (Appscale Systems, 2018)*. Retrieved from `https://www.eucalyptus.cloud`. Accessed 3 December, 2019.

[82] *Experimentation GitHub (Barcelona Supercomputing Center, 2018)*. Retrieved from `https://github.com/cristianrcv/pycompss-autoparallel/tree/master/examples`. Accessed 4 June, 2018.

[83] *Extrae Tool (Barcelona Supercomputing Center, 2017)*. Retrieved from `https://tools.bsc.es/extrae`. Accessed 20 July, 2017.

[84] T. Fahringer and et al. "Askalon: A grid application development and computing environment". In: *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing* (2005), pp. 122–131. DOI: `10.1109/GRID.2005.1542733`.

[85] W. Felter et al. "An updated performance comparison of virtual machines and linux containers". In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE. 2015, pp. 171–172. DOI: `10.1109/ISPASS.2015.7095802`.

[86] R. Filgueira et al. "Asterism: Pegasus and Dispel4py Hybrid Workflows for Data-Intensive Science". In: *2016 Seventh International Workshop on Data-Intensive Computing in the Clouds (DataCloud)*. Nov. 2016, pp. 1–8. DOI: `10.1109/DataCloud.2016.004`.

[87] A. Finn et al. *Microsoft private cloud computing*. John Wiley & Sons, 2012.

[88] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[89] G. Galante and et al. "An Analysis of Public Clouds Elasticity in the Execution of Scientific Applications: a Survey". In: *Journal of Grid Computing* 14.2 (2016), pp. 193–216. ISSN: 1572-9184. DOI: `10.1007/s10723-016-9361-3`.

[90] *Galaxy (Galaxy Team, 2019)*. Retrieved from `https://usegalaxy.org`. Accessed 26 March, 2019.

[91] W. Gentzsch. “Sun grid engine: Towards creating a compute power grid”. In: *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*. IEEE, 2001, pp. 35–36.

[92] W. Gerlach and et al. “Skyport: container-based execution environment management for multi-cloud scientific workflows”. In: *Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds*. IEEE Press. 2014, pp. 25–32.

[93] *GitHub: Apache Kafka (The Apache Software Foundation, 2019)*. Retrieved from `https://github.com/apache/kafka`. Accessed 12 July, 2019.

[94] *GitHub: DistroStream Library (Barcelona Supercomputing Center, 2019)*. Retrieved from `https://github.com/bsc-wdc/distro-stream-lib`. Accessed 3 September, 2019.

[95] *GNU Bash (Free Software Foundation, 2017)*. Retrieved from `https://www.gnu.org/software/bash`. Accessed 20 July, 2017.

[96] *GNU General Public License, version 2 (Free Software Foundation, 2017)*. Retrieved from `https://www.gnu.org/licenses/old-licenses/gpl-2.0.html`. Accessed 2 October, 2019.

[97] *GNU General Public License, version 3 (Free Software Foundation, 2016)*. Retrieved from `https://www.gnu.org/licenses/gpl-3.0.en.html`. Accessed 2 October, 2019.

[98] *GNU Lesser General Public License, version 2.1 (Free Software Foundation, 2018)*. Retrieved from `https://www.gnu.org/licenses/old-licenses/lgpl-2.1.html`. Accessed 2 October, 2019.

[99] *GNU Plot*. Retrieved from `http://www.gnuplot.info`. Accessed 20 July, 2017.

[100] G. H. Golub and C. F. Van Loan. *Matrix Computations (3rd Ed.)* Baltimore, MD, USA: Johns Hopkins University Press, 1996. ISBN: 0-8018-5414-8.

[101] *GUIDANCE: An easy-to-use platform for comprehensive GWAS and PheWAS (Computational Genomics Group at the Barcelona Supercomputing Center, 2019)*. Retrieved from `http://cg.bsc.es/guidance`. Accessed 13 May 2020.

[102] M. Guindo-Martínez, R. Amela, and et al. “The impact of non-additive genetic associations on age-related complex diseases”. In: *bioRxiv* (2020). DOI: `10.1101/2020.05.12.084608`.

[103] J. A. Gunnels et al. “FLAME: Formal Linear Algebra Methods Environment”. In: *ACM Trans. Math. Softw.* 27.4 (Dec. 2001), pp. 422–455. DOI: `10.1145/504210.504213`.

[104] *Hazelcast Jet (Hazelcast Inc., 2017)*. Retrieved from `https://jet.hazelcast.org`. Accessed 15 December, 2017.

[105] *Hecuba (Barcelona Supercomputing Center, 2019)*. Retrieved from `https://github.com/bsc-dd/hecuba`. Accessed 2 October, 2019.

[106] R. L. Henderson. “Job scheduling under the portable batch system”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1995, pp. 279–294.

[107] K. Hightower, B. Burns, and J. Beda. *Kubernetes: up and running: dive into the future of infrastructure*. O’Reilly Media Inc., 2017.

[108] C. Hill et al. “The architecture of the Earth System Modeling Framework”. In: *Computing in Science Engineering* 6.1 (Jan. 2004), pp. 18–28. ISSN: 1521-9615. DOI: `10.1109/MCISE.2004.1255817`.

[109] B. Hindman and et al. “Mesos: A platform for fine-grained resource sharing in the data center”. In: *NSDI*. Vol. 11. 2011, pp. 22–22.

[110] M. Hirzel and et al. “IBM streams processing language: Analyzing big data in motion”. In: *IBM Journal of Research and Development* 57.3/4 (2013), pp. 7–11. DOI: `10.1147/JRD.2013.2243535`.

[111] B. N. Howie, P. Donnelly, and J. Marchini. “A flexible and accurate genotype imputation method for the next generation of genome-wide association studies”. In: *PLoS genetics* 5.6 (2009), e1000529.

[112] *HTCondor - High Troughput Computing (University of Wisconsin-Madison - Computer Sciences Department, 2019)*. Retrieved from `https://research.cs.wisc.edu/htcondor`. Accessed 3 December, 2019.

[113] D. Hull and et al. “Taverna: a tool for building and running workflows of services”. In: *Nucleic Acids Research* 34(Web Server issue) (2006), W729–W732. DOI: `10.1093/nar/gkl320`.

[114] *IBM LSF (IBM, 2016)*. Retrieved from `https://www.ibm.com/support/knowledgecenter/en/SSETD4/product_welcome_platform_lsf.html`. Accessed 3 December, 2019.

[115] *IBM Streams (IBM, 2017)*. Retrieved from `https://www.ibm.com/cloud/streaming-analytics`. Accessed 15 December, 2017.

[116] *Intel MPI implementation (Intel Corporation, 2017)*. Retrieved from `https://software.intel.com/en-us/intel-mpi-library`. Accessed 30 November, 2017.

[117] *INTERTWinE addresses the problem of programming-model design and implementation for the Exascale (European Union’s Horizon 2020 Research and Innovation programme under Grant Agreement no. 671602, 2020)*. Retrieved from `http://www.intertwine-project.eu/`. Accessed 2 July, 2020.

[118] *Introduction to FireWorks (workflow software) - FireWorks 1.8.7 documentation (A. Jain, 2019)*. Retrieved from `https://materialsproject.github.io/fireworks`. Accessed 26 March, 2019.

[119] Z. Janjic and R. Gall. *Scientific Documentation of the NCEP Nonhydrostatic Multiscale Model on the B Grid (NMMB). Part 1 Dynamics, Technical Report*. Tech. rep. 80307-3000. BOULDER, COLORADO: NCEP, Apr. 2012.

[120] *Java NIO (Oracle, 2010)*. Retrieved from `https://docs.oracle.com/javase/1.5.0/docs/guide/nio/index.html`. Accessed 20 January, 2020.

[121] *Java Programming Language (Oracle, 2017)*. Retrieved from `https://www.oracle.com/es/java/index.html`. Accessed 20 July, 2017.

[122] E. Jones, T. Oliphant, and P. Peterson. *SciPy: Open source scientific tools for Python*. 2001–. URL: `http://www.scipy.org/`.

[123] P. Joshi and M. R. Babu. “Openlava: An open source scheduler for high performance computing”. In: *2016 International Conference on Research Advances in Integrated Navigation Systems (RAINS)*. 2016, pp. 1–3.

[124] S. Kaisler et al. “Big Data: Issues and Challenges Moving Forward”. In: *46th Hawaii International Conference on System Sciences* (Jan. 2013), pp. 995–1004. ISSN: 1530-1605. DOI: `10.1109/HICSS.2013.645`.

[125] G. Katsaros and et al. “Cloud application portability with tosca, chef and openstack”. In: *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE. 2014, pp. 295–302.

[126] B. W. Kernighan and D. M. Ritchie. *The C programming language*. 2006.

[127] A. Kivity and et al. "KVM: the Linux virtual machine monitor". In: *Proceedings of the Linux symposium*. Vol. 1. 2007, pp. 225–230.

[128] *Kompose (The Kubernetes Authors, 2019)*. Retrieved from https://kompose.io. Accessed 13 May, 2020.

[129] K. Krauter, R. Buyya, and M. Maheswaran. "A taxonomy and survey of grid resource management systems for distributed computing". In: *Software: Practice and Experience* 32.2 (2002), pp. 135–164. DOI: 10.1002/spe.432.

[130] J. Kreps, N. Narkhede, J. Rao, et al. "Kafka: A distributed messaging system for log processing". In: 2011, pp. 1–7.

[131] SPT Krishnan and J. L. U. Gonzalez. "Google compute engine". In: *Building Your Next Big Thing with Google Cloud Platform*. Springer, 2015, pp. 53–81.

[132] *Kubernetes and Docker Swarm Compared (Platform 9, 2017)*. Retrieved from https://platform9.com/blog/kubernetes-docker-swarm-compared. Accessed 3 December, 2019.

[133] *Kubernetes and Mesos Compared (Platform 9, 2016)*. Retrieved from https://platform9.com/blog/compare-kubernetes-vs-mesos. Accessed 3 December, 2019.

[134] *Kubernetes (The Linux Foundation, 2019)*. Retrieved from https://kubernetes.io. Accessed 3 December, 2019.

[135] S. Kulkarni and et al. "Twitter heron: Stream processing at scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 239–250. DOI: 10.1145/2723372.2742788.

[136] R. Kumar and et al. *Apache cloudstack: Open source infrastructure as a service cloud computing platform*. Tech. rep. 2014, pp. 111–116.

[137] V. Kumar et al. *Introduction to parallel computing: design and analysis of algorithms*. Vol. 400. Benjamin/Cummings Redwood City, 1994. DOI: 10.1109/MCC.1994.10011.

[138] V. Kurtzer G. M.and Sochat and M. W. Bauer. "Singularity: Scientific containers for mobility of compute". In: *PloS one* 12.5 (2017).

[139] S. K. Lam, A. Pitrou, and S. Seibert. "Numba: A llvm-based python jit compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–6.

[140] H. Li and et al. "The sequence alignment/map format and SAMtools". In: *Bioinformatics* 25.16 (2009), pp. 2078–2079.

[141] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc., 1999. ISBN: 0201325772.

[142] X. Liu, N. Iftikhar, and X. Xie. "Survey of Real-time Processing Systems for Big Data". In: *Proceedings of the 18th International Database Engineering &#38; Applications Symposium (IDEAS)* 25 (2014), pp. 356 –361. DOI: 10.1145/2628194.2628251.

[143] R. Loh and et al. "Reference-based phasing using the Haplotype Reference Consortium panel". In: *Nature genetics* 48.11 (2016), p. 1443.

[144] F. Lordan and et al. "ServiceSs: an interoperable programming framework for the Cloud". In: *Journal of Grid Computing* 12.1 (Mar. 2014), pp. 67–91. DOI: 10.1007/s10723-013-9272-5.

[145] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. "A review of auto-scaling techniques for elastic applications in cloud environments". In: *Journal of Grid Computing* 12.4 (2014), pp. 559–592. DOI: 10.1007/s10723-014-9314-7.

[146] B. Ludäscher and et al. "Scientific workflow management and the Kepler system". In: *Concurrency and Computation: Practice and Experience* 18.10 (2006), pp. 1039–1065.

[147] J. Marchini and B. Howie. "Genotype imputation for genome-wide association studies". In: *Nature Reviews Genetics* 11.7 (2010), p. 499.

[148] *MareNostrum 3 (Barcelona Supercomputing Center, 2018)*. Retrieved from https://www.bsc.es/marenostrum/marenostrum/mn3. Accessed 20 May, 2018.

[149] *MareNostrum 4 (Barcelona Supercomputing Center, 2017)*. Retrieved from https://www.bsc.es/innovation-and-services/supercomputers-and-facilities/marenostrum. Accessed 20 July, 2017.

[150] J. Martí and et al. "Dataclay: A distributed data store for effective inter-player data sharing". In: *Journal of Systems and Software* 131 (2017), pp. 129–145.

[151] J. M. Martinez Caamaño and et al. "Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones". In: *Concurrency and Computation: Practice and Experience* 29.15 (2017), e4192. DOI: 10.1002/cpe.4192.

[152] W. McKinney. "Pandas: a Foundational Python Library for Data Analysis and Statistics". In: *Python for High Performance and Scientific Computing* (2011), pp. 1–9.

[153] H. Meng and D. Thain. "Umbrella: A Portable Environment Creator for Reproducible Computing on Clusters, Clouds, and Grids". In: *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*. VTDC '15. ACM, 2015, pp. 23–30. DOI: 10.1145/2755979.2755982.

[154] D. Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux Journal* 2014.239 (2014), p. 2.

[155] D. Milojičić, I. M. Llorente, and R. S. Montero. "Opennebula: A cloud management tool". In: *IEEE Internet Computing* 15.2 (2011), pp. 11–14.

[156] *MIT License (Opensource.org, 2019)*. Retrieved from https://opensource.org/licenses/MIT. Accessed 2 October, 2019.

[157] F. Montesi et al. "Jolie: a Java orchestration language interpreter engine". In: *Electronic Notes in Theoretical Computer Science* 181 (2007), 19–33. DOI: 10.1016/j.entcs.2007.01.051.

[158] *Mozilla Public License Version 2.0 (Mozilla Foundation, 2019)*. Retrieved from https://www.mozilla.org/en-US/MPL/2.0. Accessed 2 October, 2019.

[159] *MPI: A Message-Passing Interface Standard*. Tech. rep. 3.1. June 2015. URL: http://mpi-forum.org/docs/.

[160] *MPICH: High-Performance Portable MPI (MPICH Collaborators, 2017)*. Retrieved from https://www.mpich.org. Accessed 30 November, 2017.

[161] S. C. Müller et al. "Pydron: Semi-automatic parallelization for multi-core and the cloud". In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 645–659.

[162] A. Munshi et al. *OpenCL programming guide*. Pearson Education, 2011. DOI: 10.5555/2049883.

[163] D. G. Murray and et al. "CIEL: a universal execution engine for distributed data-flow computing". In: *Proceedings of the 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*. ACM, 2011, pp. 113–126. DOI: 10.5555/1972457.1972470.

[164] *MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE (NBCL, 2017)*. Retrieved from http://mvapich.cse.ohio-state.edu. Accessed 30 November, 2017.

[165] N. Naik. "Building a virtual system of systems using docker swarm in multiple clouds". In: *2016 IEEE International Symposium on Systems Engineering (ISSE)*. 2016, pp. 1–3.

[166] V. Narasimhan and et al. "BCFtools/RoH: a hidden Markov model approach for detecting autozygosity from next-generation sequencing data". In: *Bioinformatics* 32.11 (2016), pp. 1749–1751.

[167] *Netflix Blog: Stream processing with Mantis (B. Schmaus, et al., 2016)*. Retrieved from https://medium.com/netflix-techblog/stream-processing-with-mantis-78af913f51a6. Accessed 15 December, 2017.

[168] *Netflix Conductor (2017)*. Retrieved from https://netflix.github.io/conductor. Accessed 15 December, 2017.

[169] *Nextflow: A DSL for parallel and scalable computational pipelines (Barcelona Centre for Genomic Regulation, 2019)*. Retrieved from https://www.nextflow.io. Accessed 26 March, 2019.

[170] J. Nickolls et al. "Scalable parallel programming with CUDA". In: *Queue* 6.2 (2008), pp. 40–53. DOI: 10.1145/1365490.1365500.

[171] *Nova-Docker driver for OpenStack (OpenStack, 2017)*. Retrieved from https://github.com/openstack/nova-docker. Accessed 15 November, 2016.

[172] *Numba: A High Performance Python Compiler (Anaconda, 2020)*. Retrieved from http://numba.pydata.org. Accessed 8 April, 2020.

[173] *NumExpr: Fast numerical expression evaluator for NumPy (D. M. Cooke, F. Alted, et al., 2020)*. Retrieved from https://github.com/pydata/numexpr. Accessed 8 April, 2020.

[174] D. Nurmi and et al. "The eucalyptus open-source cloud-computing system". In: *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society. 2009, pp. 124–131.

[175] *OneDock: Docker driver for OpenNebula (Indigo-dc, 2017)*. Retrieved from https://github.com/indigo-dc/onedock. Accessed 5 May, 2020.

[176] *Open MPI: Open Source High Performance Computing (The Open MPI Project, 2017)*. Retrieved from https://www.open-mpi.org. Accessed 30 November, 2017.

[177] *OpenNebula (OpenNebula Project - OpenNebula.org, 2019)*. Retrieved from https://opennebula.org. Accessed 3 December, 2019.

[178] *OpenStack (OpenStack Foundation, 2019)*. Retrieved from https://www.openstack.org. Accessed 3 December, 2019.

[179] Oracle. *JavaNIO*. URL: http://www.oracle.com/technetwork/articles/javase/nio-139333.html.

[180] B. Parák and et al. "The rOCCI project: providing cloud interoperability with OCCI 1.1". In: *International Symposium on Grids and Clouds (ISGC) 2014*. Vol. 210. SISSA Medialab. 2014, p. 014.

[181] *Parallel Processing and Multiprocessing in Python (Python Software Fundation, 2019)*. Retrieved from https://wiki.python.org/moin/ParallelProcessing. Accessed 8 October, 2019.

[182] *Parallel Python Software (V. Vanovschi, 2019)*. Retrieved from http://www.parallelpython.com. Accessed 8 October, 2019.

[183] *Paraver Tool (Barcelona Supercomputing Center, 2017)*. Retrieved from https://tools.bsc.es/paraver. Accessed 20 July, 2017.

[184] *Parsl: Parallel Scripting in Python (University of Chicago, 2019)*. Retrieved from http://parsl-project.org. Accessed 26 March, 2019.

[185] *PBS Professional - Open Source Project (Altair Engineering Inc., 2019)*. Retrieved from https://www.pbspro.org. Accessed 3 December, 2019.

[186] R. Peinl, F. Holzschuher, and F. Pfitzer. "Docker Cluster Management for the Cloud - Survey Results and Own Solution". In: *Journal of Grid Computing* 14.2 (June 2016), pp. 265–282. ISSN: 1572-9184. DOI: 10.1007/s10723-016-9366-y.

[187] *Pluto (Pluto Contributors, 2017)*. Retrieved from http://pluto-compiler.sourceforge.net. Accessed 28 November, 2017.

[188] *PolyBench/C: The Polyhedral Benchmark suite (Ohio State University, 2018)*. Retrieved from http://web.cse.ohio-state.edu/~pouchet.2/software/polybench. Accessed 18 June, 2018.

[189] IBIS Project. *JavaGAT*. URL: http://www.cs.vu.nl/ibis/javagat.html.

[190] S. Pronk and et al. "Copernicus: A new paradigm for parallel adaptive molecular dynamics". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), 60:1–60:10. DOI: 10.1145/2063384.2063465.

[191] *Puppet (Puppet Inc., 2020)*. Retrieved from https://puppet.com. Accessed 5 May, 2020.

[192] S. Purcell and et al. "PLINK: a tool set for whole-genome association and population-based linkage analyses". In: *The American Journal of Human Genetics* 81.3 (2007), pp. 559–575.

[193] *PyCOMPSs AutoParallel Module GitHub (Barcelona Supercomputing Center, 2018)*. Retrieved from https://github.com/cristianrcv/pycompss-autoparallel. Accessed 4 June, 2018.

[194] *PyCOMPSs User Manual (Barcelona Supercomputing Center, 2020)*. Retrieved from https://compss-doc.readthedocs.io/en/2.6/Sections/02_User_Manual_App_Development.html. Accessed 8 April, 2020.

[195] *PySpark (Apache Software Fundation, 2019)*. Retrieved from https://spark.apache.org/docs/latest/api/python/index.html. Accessed 8 October, 2019.

[196] *Python Org (Python Software Foundation, 2017)*. Retrieved from https://www.python.org. Accessed 20 July, 2017.

[197] G. Quintana-Orti and et al. "Scheduling of QR Factorization Algorithms on SMP and Multi-Core Architectures". In: *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*. PDP '08. IEEE Computer Society, 2008, pp. 301–310. DOI: 10.1109/PDP.2008.37.

[198] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2015. URL: https://www.R-project.org/.

[199] *RedHat OpenShift (Red Hat Inc., 2019)*. Retrieved from https://www.openshift.com. Accessed 3 December, 2019.

[200] D. A. Reed and J. Dongarra. "Exascale Computing and Big Data". In: *Communications of the ACM* 58.7 (July 2015), pp. 56–68. DOI: 10.1145/2699414.

[201] *rOCCI - A Ruby OCCI Framework (F. Feldhaus, 2012)*. Retrieved from http://occi-wg.org/2012/04/02/rocci-a-ruby-occi-framework/index.html. Accessed 4 May, 2020.

[202] M. Rocklin. "Dask: Parallel Computation with Blocked algorithms and Task Scheduling". In: 2015, pp. 130 –136. DOI: 10.25080/Majora-95ae3ab6-01e.

[203] G. van Rossum and F. L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011. ISBN: 1906966141, 9781906966140. DOI: 10.5555/2011965.

[204] P. Russom and et al. "Big data analytics". In: *TDWI best practices report, fourth quarter* 19 (2011).

[205] S. Sagiroglu and D. Sinanc. "Big data: A review". In: *International Conference on Collaboration Technologies and Systems (CTS)* (2013), pp. 42–47. DOI: 10.1109/CTS.2013.6567202.

[206] K. Sala and et al. "Improving the interoperability between MPI and task-based programming models". In: *Proceedings of the 25th European MPI Users' Group Meeting*. 2018, pp. 1–11.

[207] O. Sefraoui, M. Aissaoui, and M. Eleuldj. "OpenStack: toward an open-source solution for cloud computing". In: *International Journal of Computer Applications* 55.3 (2012), pp. 38–42.

[208] S. Shahrivari. "Beyond batch processing: towards real-time and streaming big data". In: *Computers* 3.4 (2014), pp. 117–129. DOI: 10.3390/computers3040117.

[209] C. Simmendinger and et al. "Interoperability strategies for GASPI and MPI in large-scale scientific applications". In: *The International Journal of High Performance Computing Applications* 33.3 (2019), pp. 554–568. DOI: 10.1177/1094342018808359.

[210] *Singularity (Sylabs.io, 2020)*. Retrieved from https://sylabs.io/docs. Accessed 19 February, 2020.

[211] *Slurm Workload Manager (Slurm Team, 2019)*. Retrieved from https://slurm.schedmd.com. Accessed 3 December, 2019.

[212] B. Stroustrup. *The C++ programming language. Pearson Education*. 2013.

[213] A. Sukumaran-Rajam and P. Clauss. "The Polyhedral Model of Nonlinear Loops". In: *ACM Trans. Archit. Code Optim.* 12.4 (Dec. 2015), 48:1–48:27. ISSN: 1544-3566. DOI: 10.1145/2838734.

[214] *Swarm Mode Overview (Docker Inc., 2019)*. Retrieved from https://docs.docker.com/engine/swarm. Accessed 3 December, 2019.

[215] E. Tejedor and et al. "PyCOMPSs: Parallel computational workflows in Python". In: *The International Journal of High Performance Computing Applications (IJHPCA)* 31 (2017), pp. 66–82. DOI: 10.1177/1094342015594678.

[216] E. Tejedor and et al. "PyCOMPSs: Parallel computational workflows in Python". In: *The International Journal of High Performance Computing Applications (IJHPCA)* 31.1 (2017), pp. 66–82.

[217] D. Thain, T. Tannenbaum, and M. Livny. "Distributed computing in practice: the Condor experience". In: *Concurrency and computation: practice and experience* 17.2-4 (2005), pp. 323–356.

[218] *The Intersection of AI, HPC and HPDA: How Next-Generation Workflows Will Drive Tomorrow's Breakthroughs (Damkroger, P. A., 2018)*. Retrieved from `https://www.top500.org`. Accessed 7 August, 2019.

[219] *The Kepler Project (Kepler Contributors, 2017)*. Retrieved from `https://kepler-project.org`. Accessed 1 April, 2019.

[220] *The OmpSs Programming Model (Barcelona Supercomputing Center, 2020)*. Retrieved from `https://pm.bsc.es/ompss`. Accessed 27 February, 2020.

[221] *The Swift Parallel Scripting Language (Swift Project Team, 2019)*. Retrieved from `http://swift-lang.org/main`. Accessed 26 March, 2019.

[222] *Threading Building Blocks (Intel, 2019)*. Retrieved from `https://software.intel.com/en-us/tbb`. Accessed 8 October, 2019.

[223] G. Toraldo. *Opennebula 3 cloud computing*. Packt Publishing Ltd, 2012.

[224] *TORQUE Resource Manager (Adaptive Computing Inc., 2019)*. Retrieved from `https://www.adaptivecomputing.com/products/torque`. Accessed 3 December, 2019.

[225] A. Toshniwal and et al. "Storm@ twitter". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 147–156. DOI: `10.1145/2588555.2595641`.

[226] V. K. Vavilapalli and et al. "Apache hadoop yarn: Yet another resource negotiator". In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[227] C. Vecchiola, X. Chu, and R. Buyya. "Aneka: A software platform for .NET-based cloud computing". In: *High Speed and Large Scale Scientific Computing* 18 (July 2009), pp. 267–295. DOI: `10.3233/978-1-60750-073-5-267`.

[228] *Virtual Machine Manager Documentation (Microsoft, 2020)*. Retrieved from `https://docs.microsoft.com/en-gb/system-center/vmm/?view=sc-vmm-2019`. Accessed 4 May, 2020.

[229] *VM Ware (VMware Inc., 2017)*. Retrieved from `http://www.vmware.com`. Accessed 11 April, 2017.

[230] S. var der Walt, S. C. Colbert, and G. Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science and Engg.* 13.2 (Mar. 2011), pp. 22–30. DOI: `10.1109/MCSE.2011.37`.

[231] X. Wen and et al. "Comparison of open-source cloud management platforms: OpenStack and OpenNebula". In: *2012 9th International Conference on Fuzzy Systems and Knowledge Discovery*. 2012, pp. 2457–2461.

[232] M. Wilde and et al. "Swift: A language for distributed parallel scripting". In: *Parallel Computing* 37(9) (2011), pp. 633–652. DOI: `10.1016/j.parco.2011.05.005`.

[233] O. Yildiz et al. "Heterogeneous hierarchical workflow composition". In: *Computing in Science & Engineering* 21.4 (2019), pp. 76–86.

[234] A. B. Yoo, M. A. Jette, and M. Grondona. "Slurm: Simple linux utility for resource management". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[235] M. Zaharia and et al. "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters". In: *HotCloud* 12 (2012), pp. 10–16. DOI: `10.5555/2342763.2342773`.

[236] M. Zaharia and et al. "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters". In: *HotCloud* 12 (2012), pp. 10–16.

[237] M. Zaharia and et al. “Spark: Cluster Computing with Working Set”. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (2010). DOI: 10.5555/1863103.1863113.

[238] C. Zheng and D. Thain. “Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker”. In: *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*. ACM. 2015, pp. 31–38.