UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors


UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC)
BARCELONATECH

COMPUTER ARCHITECTURE DEPARTMENT (DAC)


# Programming Model Abstractions For Optimizing I/O Intensive Applications

PH.D. THESIS

2021 | AUTUMN SEMESTER


**Author:**

Hatem ELSHAZLY
*hatem.elshazly@bsc.es*

**Advisor:**

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


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

# Dedication

This work would not have been possible without the love and support of the people around me. Thank you for helping me getting through this experience without waiting for anything in return.

To Marta, for her love and kindness, for sticking with me in bad and good moments, for making a new place feel like home.

To my mother, Aisha, and father, Mohamed, for always surrounding me with love, for being with me through hardships and encouraging me to never stop pursuing my dreams.

To my brother, Hosam, and sisters Radwa and Yasmin and her sweet little daughter Rofayda, for their unconditional love and support, for always standing by my side, their sense of humor and sarcasm is what keeps me going.

To Josep and Mayte, for showing me affection and care when they were not expected, being part of your family is something that will always be cherished.

To Faisal, Shady, Talaat and Mahmoud, for the best of times we spent together, for showing me what true friendship is all about.

From the bottom of my heart,

Hatem

# Declaration Of Authorship

I hereby declare that this thesis has been authored by myself and it is based on my own work, except where references are explicitly made to the work of others. None of the contents of this document has been previously published nor submitted, in part or in whole, to any other examination panels in this university or any other.

Signature: _____

Date: _____

# Acknowledgements

I gratefully thank my supervisor Rosa M. Badia Sala for giving my the opportunity to work on and finish this thesis. Her assistance since the start of the project will always be appreciated.

I also gratefully thank Jorge Ejarque Artigas. Without his assistance and positive attitude, finishing this thesis would not have been possible.

Also, I thank all former and current colleagues from the Barcelona Supercomputing Center (BSC) for the useful discussions and words of encouragement:

Pierlauro Sciarelli, Kevin Sala, Toni Muñoz, Kazi Asifuzzaman, Jimmy Aguilar Mena, Salvi Solà, Omar Shaaban, Francesc Lordan, Francisco Javier Conejero, Marc Dominguez, Sergio Rodríguez, Pol Alvarez, Nihad Mammadli, Cristian Ramon-Cortes and Daniele Lezzi.

This thesis is partially supported by the European Union through the Horizon 2020 research and innovation programme under contracts 721865 (EXPERTISE Project), the Spanish Government (PID2019-107255GB) and the Generalitat de Catalunya (contract 2014-SGR-1051).

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

# *Abstract*

**Programming Model Abstractions For Optimizing I/O Intensive Applications**

by Hatem ELSHAZLY

Recent advances in science and technology are being driven by computing systems that are able to deliver unprecedented levels of performance. Applications in many scientific areas and data analysis disciplines such as life sciences, earth sciences, artificial intelligence and big data analysis generate more and more amounts of data. However, storage and I/O performance has not kept a matching performance improvement trend. Hence, many applications have witnessed a paradigm shift from being computation-bound to be I/O-bound. I/O became the bottleneck that prevents achieving more performance. In addition to that, execution platforms design that is geared towards distribution and heterogeneity, has made it increasingly complex for non-expert users such as field experts and domain scientists to design, program and launch large-scale experiments. Thus, limiting programmability and productivity.

This thesis contributes to the efforts of optimizing nowadays I/O intensive applications by proposing and introducing programming model abstractions and mechanisms that have a twofold objective: On the one hand, improving applications I/O performance to achieve total overall performance enhancement. On the other hand, expose those techniques and mechanisms in such a manner that abstracts the complexities of modern distributed and heterogeneous execution platforms. We achieve this objective by extending the PyCOMPSs framework, a general purpose, task-based programming model for executing applications on large-scale distributed systems. The main contributions of this thesis are summarized in the following paragraphs.

First, we propose enabling ***I/O awareness*** in task-based programming models. I/O Awareness refers to the ability to separate the handling of I/O from computation. Such a separation allows the optimization of each workload in terms of scheduling and execution. We achieve I/O awareness by introducing the concept of *I/O tasks* that can overlap with compute tasks execution. In addition to that, we improve I/O performance by supporting programming annotations and mechanisms for mitigating application-level I/O congestion: manual constraining of tasks, and a mechanism for automatically setting and tuning task constraints based on execution time metrics.

Second, we target the transparent management and exploitation of the heterogeneity of modern storage systems to improve I/O performance. To this end, we propose a set of capabilities that can be referred to as ***Storage Heterogeneity Awareness***. For instance, we provide programming model support to transparently expose the underlying storage devices as a hierarchical pooled resource such that the top layer has the highest storage bandwidth whereas the bottom layer has the lowest storage bandwidth. Moreover, we propose dedicated I/O schedulers to take advantage of this storage devices organization to increase I/O task parallelism without causing I/O congestion. In addition to that, we present an automatic data movement mechanism to maximize the usage of higher storage layers.

Third, we increase applications parallelism by introducing a ***hybrid programming model*** that combines task-based programming models and MPI. Such a hybrid model allows the execution of tasks on distributed platforms, while using MPI to parallelize tasks execution. This goal is achieved by extending the task-based programming model to support *Native MPI Tasks*. The same application source code can have different Native MPI tasks (each

with its own number of MPI processes) and different sequential tasks. We extend the programming model to provide complete transparency with regard to the interaction between different types of tasks, tasks execution, data transfer, etc. This hybrid programming model enables the implementation of parallel-I/O tasks or using high-level parallel I/O libraries inside tasks.

Finally, we introduce a ***mechanism for eagerly releasing the data dependencies of tasks***. Using such a mechanism, successor tasks are launched for execution as soon as their data requirements are generated by the predecessor task(s). Unlike traditional programming models, task execution does not have to be delayed until the predecessor task(s) finishes execution. Our proposed behaviour enhances execution by allowing the overlapping execution of tasks. For instance, overlapping I/O with computation. To this end, we describe two necessary modifications to data dependencies definition and management: (i) parameter-aware dependencies, (ii) a mechanism for notifying the programming model that a task has generated data before reaching the `return` statement in the task code.

**Keywords:** Distributed Computing, High Performance Computing, I/O Bottleneck, Task-based Workflows, Heterogeneous Systems, COMPSs, PyCOMPSs, MPI, I/O Awareness, Storage-Heterogeneity Awareness, Eager-Realse Of Dependencies, Hybrid Programming Model, Programmability, Productivity

# Contents

# List of Figures

# List of Listings

# List of Tables

# Part I

# Introduction

# Chapter 1

# Introduction

## 1.1 Research Context

### 1.1.1 Towards the Exascale Computing Era

In recent years, the computing power of supercomputers and production systems in terms of execution rate of floating-point operations has increased to reach the Petaflops/Second landmark [107]. This exponential growth in computing power has benefited from multi-core architectures that can accommodate an increasing number of cores on a single chip.

In addition to processor technology scaling, it has been observed that reaching these new records in computing power was also made possible by two notable changes in infrastructure design: *Heterogeneous Design* and *Distributed Architecture*.

On the one hand, modern and future systems are increasingly incorporating hardware accelerators such as *Graphical Processing Units* (GPUs) and *Field Programmable Gate Arrays* (FPGAs) to leverage their computing capabilities and achieve more performance [13]. Such devices can be used along side multi-core CPUs to efficiently execute compute intensive applications [18].

On the other hand, platform architecture trends has have scaled out to include multiple computing nodes connected by a network in the same system [53]. Such architectural design takes advantage of the extra computing power added by including more nodes into the system to distribute the execution of computing workloads. Current production systems consist of thousands of computing nodes while future generations of supercomputers are expected to incorporate more [66].

Such astonishing increase in performance and processing capabilities is driving the computing scene towards what became to be known as the "*Exascale Computing Era*". Scientific and engineering breakthroughs have been made possible in numerous fields to solve the major problems that face us today. For instance, problems in disciplines as earth science, life science, climate modeling, astrophysics, etc [97].

The advances in modern infrastructure architecture and system design trends has been followed by similar advances in programming paradigms and software design. In order to get the best possible performance out of the underlying execution systems, it is necessary to use software frameworks and programming models that allow and support parallel and distributed execution.

However, one of the major drawbacks of such paradigm shift is the increased difficulty and complexity of developing parallel and distributed applications on modern infrastructures [54]. This is especially true for end-users and domain scientists with little or no background in computer science or parallel programming. To this end, several programming models, libraries and tools have emerged to ease the complexity of programming for performance on modern infrastructures [5].

### 1.1.2 I/O and Data Intensive Applications

The rapid increase of computing capabilities has made data processing and generation happen at a much faster rate. Nowadays, applications produce increasing amounts of data. The sizes of such data ranges from gigabytes to petabytes and beyond. For instance, LIGO (gravitational wave detection) generates 1500TB/year [56], climate modeling is projected to produce 100EB of data and Genomics analysis produces data that can reach up to 10PB [40].

The Big Data and Exascale Computing (BDEC) project [6] highlights that the management, analysis, mining and knowledge discovery from data sets of this scale are very challenging problems.

Previous research studies have observed patterns in the I/O behaviour of data intensive scientific applications [44], [15]. These applications alternate between computation and I/O phases in a periodic fashion [36]. The I/O phases occur in intense bursts of data that are produced by the application to be written to the storage device. Such pattern is recurrent in applications that do checkpointing [47] or post-mortem analysis where intermediate data are used to gain more scientific insights [14].

Figure 1.1 depicts the execution view of an I/O intensive application. Each compute phase is followed by an I/O phase (e.g., checkpointing the results of the previous compute phase). Once each I/O phase is over, it is followed by another compute phase except at the end of application's execution where there is no more computation. The time of each I/O phase varies depending on the size of I/O workload in each phase.

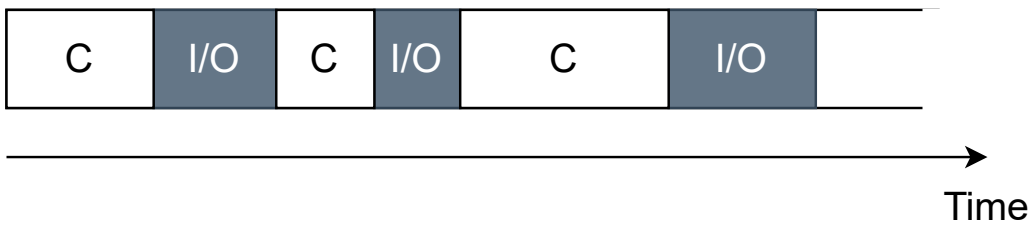



FIGURE 1.1: Execution View Of An I/O Intensive Application

### 1.1.3 The I/O Bottleneck

While the exponential growth of computing capabilities can be leveraged to achieve more parallelism and performance, the storage infrastructure performance is increasing at a much slower rate. Recent systems has delivered more than 10x peak computing performance with only around 3x improvement in Parallel File System (PFS) bandwidth. For example, the *Mira* IBM Blue Gene/Q supercomputer of the Argonne Leadership Facility (ALCF) has a peak performance of 10 PetaFlops [70], which is 20 times faster than its predecessor IBM Blue Gene/P system [83]. However, the I/O throughput on *Mira* has increased only 3 times compared to the previous system [84].

The increasing performance gap between compute rate and I/O rate has created what is now called *I/O Bottleneck*. That is, storage access has became the new bottleneck preventing improving applications performance. Consequently, scientific applications that are traditionally compute-intensive have undergone a paradigm shift in which I/O dominates execution time. In this situation, I/O operations become a major bottleneck for achieving any further scale up for critical applications [116].

Due to the under-provisioning of the bandwidth of storage systems, the I/O dominant phases of the I/O intensive applications overwhelm the bandwidth storage device creating the problem of *I/O Congestion*. I/O congestion was observed to cause significant slowdown

in the I/O performance of applications [36]. Indeed, I/O performance slowdown consequently degrades applications total performance.

### 1.1.4 Modern Storage and I/O Infrastructures

In order to bridge this performance gap, different studies has been performed at different perspectives. Research communities are studying and designing hardware and software solutions to optimize storage devices and I/O performance of I/O intensive applications.

Similar to heterogeneity infrastructure design trend adopted to achieve more computing performance, heterogeneous design has been adopted when designing storage subsystems. Newly emerging storage devices are incorporated in storage systems to optimize I/O performance. For example, Non-Volatile RAM (NVRAM) [78] and Solid-State Drivers (SSD) [69] are added to the underlying base storage subsystem of Parallel File System (e.g., Lustre [11], General Parallel File System (GPFS) [96]). These storage devices can help to reduce the gap between compute and I/O performance because of their high I/O bandwidth and low latency capabilities compared to the traditional cost-effective and low-bandwidth Hard Disk Drives (HDDs).

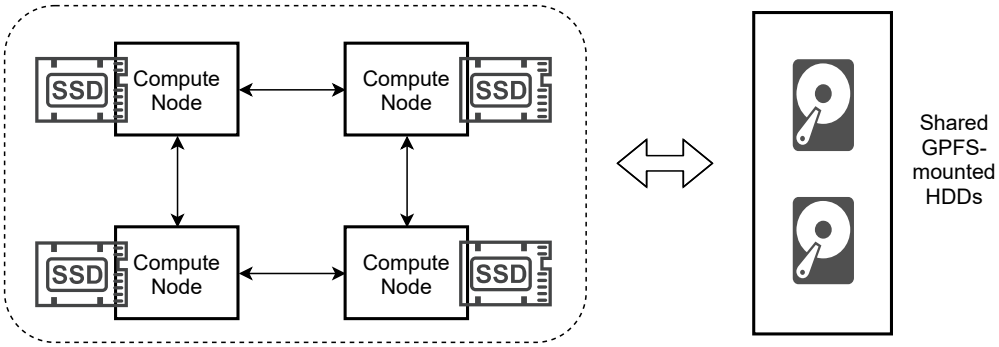
NVRAMs and SSDs are deployed in modern storage subsystems as *Burst Buffers* [63]. Due to their higher I/O bandwidth capabilities, they can absorb the data bursts produced by applications in their I/O-dominant phase. Consequently, prevent I/O congestion if huge amounts of data are written directly to the usually-backed HDD Parallel File System.

There are two approaches to incorporate Burst Buffers into a storage system: *Node-Local* and *Remote-Shared*. Figure 1.2 shows the two approaches of modern storage subsystems. Regardless of the design, compute nodes read input data from Parallel File System and Burst Buffers absorb intermediate data to later store it in the Parallel File System. For the sake of simplicity, SSDs are used as the Burst Buffer storage device. However, it is common to use a mixture of NVRAMs, SSDs and other new storage devices.

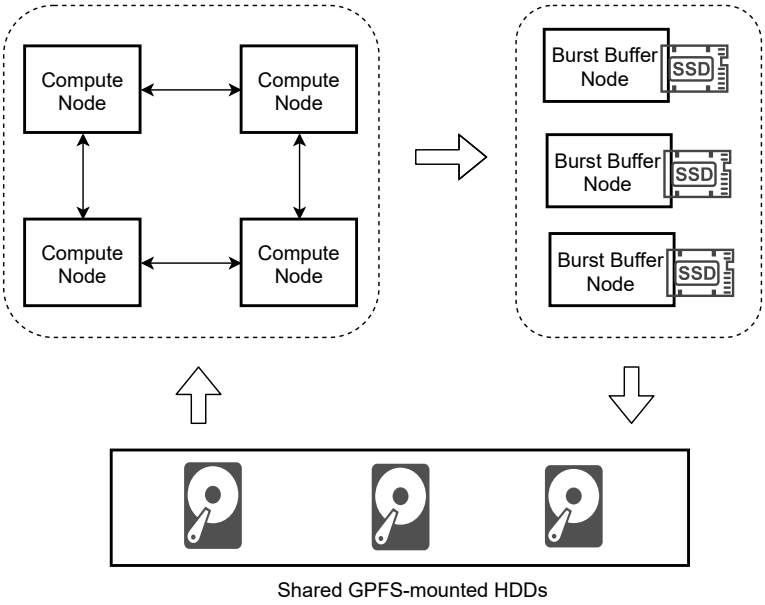
Figure 1.2(a) depicts a node-local Burst Buffer configuration where SSDs or NVRAMs are attached locally to each compute node. In this design data bursts are written to local storage devices directly.

Figure 1.2(b) depicts a remote-shared Burst Buffer design where the burst buffers are available for all compute nodes. This design offers more burst buffers to the compute nodes since any compute node can access any device in the Burst Buffer system, however, unlike the node-local design, remote-shared design adds overhead because large amounts of data need to travel the network to access the Burst Buffer system. Therefore, extra effort needs to be provided for provisioning the network and Burst Buffer system (i.e., managing metadata, coordinating access, etc..).

Besides the hardware advances in storage devices and heterogeneous design trend in storage subsystem design, it is of critical importance to provision the bandwidth of the Burst Buffer layers and manage it for performance by avoiding I/O congestion. To this end, several efforts have been done from the software perspective at two main levels of the software system stack: (i) global I/O scheduling to reduce I/O congestion between different I/O intensive applications when they make concurrent accesses to the Burst Buffers. (ii) parallelizing I/O and optimizing application's I/O access patterns by using I/O middleware such as MPI-IO [89] and HDF5 [105].




(a) Node-Local Burst Buffer Design




(b) Remote-Shared Burst Buffer Design

FIGURE 1.2: Modern Storage System Design

### 1.1.5 Task-based Programming Models

Task based parallel programming models [22] offer a high-level abstraction that facilitates extracting high performance from nowadays distributed and complex execution systems. They achieve such performance without exposing the infrastructure details of these systems to application developers. It is the responsibility of the task based system to manage the existing execution resources and schedule work to available resources dynamically at application execution time.

Following the execution model of task-based programming models, applications are decomposed into units of computations, called *tasks*. Tasks describe certain functions or pieces of code that will be executed in parallel. The specification of tasks and the order of their execution varies depending on the implementation of each programming model. A common task-based execution paradigm is by defining data dependencies between tasks. If a certain task $A$ produces a certain output, and another task $B$ requires the output of task $A$ as input, then a data dependency relationship is created between both tasks such that task $A$ is called the *Predecessor* task of $B$ and task $B$ is called the *Successor* task of $A$.

Indeed, a predecessor task can have one or many successors and a successor task can

have one or many predecessors. No matter how many predecessors a task may have, a successor task will be launched for execution if and only if all of its predecessors have successfully finished their execution.

Task-based models enable application developers to specify the code regions, intervals or functions that are to be treated and handled as tasks. Different task-based models offer different approaches such as a Graphical User Interface (GUI) or using Command Line Interfaces, configuration files or through a built-in capability in the programming language. Therefore, task-based models offer a more flexible manner to specify and exploit parallelism in applications.

Tasks do not have permanent state, i.e., all the data structure and variables created during the execution of a certain task are removed as soon as this task finishes execution. In addition to that, tasks do not share state among each other. A task is executed in a totally independent manner from all other tasks that are executed concurrently.

## 1.2 Challenges and Contributions

### 1.2.1 Problem Statement: A bird's-eye Overview

I/O performance became the bottleneck that is preventing to achieve higher levels of performance and limiting the scalability of applications. Due to the widening performance gap between compute performance and I/O throughput, the only way to achieve higher performance for data and I/O intensive applications is through optimizing the performance of their I/O-dominant phases.

This necessity to improve the I/O performance of applications goes side by side with an increasing complexity in applications or problem domain from one side and infrastructure and middleware or solution domain from the other side.

From the *problem domain perspective*, in fields of science and engineering (e.g., computational biology, molecular dynamics, mechanical turbines simulation, etc.), it is necessary to solve complex problems that exhibit irregular parallelism patterns. These patterns are characterized by their complex computation flows, access patterns and execution branches. Such problems are usually best represented by complex data structures such as trees and graphs.

From the *solution domain perspective*, nowadays infrastructures and architecture design is characterized by distribution and heterogeneity. In large-scale supercomputers and production systems, workloads are distributed to large number of Burst Buffer nodes to be stored. Moreover, such systems utilize heterogeneous hardware devices that offer different capabilities and higher performance. These systems have to be provisioned and their usage must be optimized to maximize I/O performance.

Furthermore, various I/O middlewares exist to parallelize I/O or perform fine-grained optimization to I/O access patterns. However, this fine-grained parallelism often depends on the architecture of the underlying storage system and the capabilities of the Parallel File System. Therefore, such solutions have to be tuned according to the configuration parameters of each storage system. Hence, increasing complexity and limiting portability.

As a consequence of this context, programming becomes a tedious task for application developers, let alone programming for improved I/O and total application performance. This problem can be traced back to a lack of suitable high-level abstractions that should be able to perform the following intertwined goals:

I improve I/O and total performance on modern heterogeneous storage subsystems.

II simplify the design and programming of applications and make it accessible to non-expert end-users such as field experts and domain scientists.

The next sections summarize the research questions that motivate this thesis, the main objectives of the thesis, its contributions and finally list the publications that are associated with this thesis.

### 1.2.2 Research Questions

Following the specification of the problem statement in the previous section (1.2.1), the main research question that is motivating this thesis can be summarized as follows:

- **Q**: How to improve applications I/O and total performance on distributed heterogeneous infrastructures without increasing programming complexity?

This main question can be decomposed into more specific questions to define the main points that this thesis will address:

- $\mathbf{Q}_1$: How to take advantage of the execution patterns of I/O intensive applications and address I/O performance problems to improve total performance?
- $\mathbf{Q}_2$: How to exploit heterogeneous storage devices in modern storage systems to improve I/O performance in a transparent manner?
- $\mathbf{Q}_3$: How to increase parallelism levels in applications while maintaining programming simplicity?
- $\mathbf{Q}_4$: How to overlap the execution of I/O and computation?

### 1.2.3 Objectives

The main objective of this thesis is to address the main research question **Q** defined in the previous section (1.2.2), that is, to address the lack of abstractions and techniques to improve I/O performance on modern systems without increasing programming complexity. To this end, this thesis presents high-level programming model techniques and abstractions to achieve two goals at the same time: First, abstract infrastructure details of modern heterogeneous storage systems from application programmers while exploiting it to optimize I/O performance. Second, exploit the execution patterns of I/O intensive applications to achieve not only I/O performance improvement, but total performance improvement.

Throughout this thesis, we use task-based programming models to propose our contributions. We argue that this type of programming models offers a suitable abstraction that can be leveraged to abstract the heterogeneity of underlying execution systems and exploit parallelism opportunities in I/O intensive applications to improve total performance. Hence, end users can focus on their domain science and the logic of experiments and simulations without dealing with the challenges of programming applications for I/O and total performance improvement on modern infrastructures.

More specifically, the objectives of this thesis can be listed as follows:

- $\mathbf{O}_1$: Improve I/O and total performance of applications by taking advantage of their overlapping compute-I/O phases and mitigating I/O congestion.
- $\mathbf{O}_2$: Abstract the heterogeneity of underlying storage systems and maximize its usage to optimize I/O performance in a transparent manner to application developers.
- $\mathbf{O}_3$: Increase applications parallelism by enabling fine-grained I/O parallelism inside high-level distributed execution models.

- **O**$_4$: Optimize applications performance by enabling a mechanism to overlap the execution of computation with I/O and accelerate applications execution.
- **O**$_5$: Validate these proposals with real-world applications and synthetic applications that mimic real execution patterns on large-scale heterogeneous systems.

### 1.2.4 Contributions to the field

The main contributions of this thesis are:

- **C**$_1$: I/O Aware task-based approach that is able to increase performance by overlapping I/O with computation and enhance I/O performance by mitigating I/O congestion.
- **C**$_2$: Storage-heterogeneity Aware task-based approach that is able to maximize the usage of the underlying storage system to improve I/O performance.
- **C**$_3$: A hybrid programming model that is able to support fine-grained I/O parallelism inside coarse-grained distributed tasks.
- **C**$_4$: A mechanism to enable the eager-release of data dependencies in task-based programming models to overlap the execution of computation and I/O.

Table 1.1 connects each of these contributions with the research questions and objectives that are mentioned in Section 1.2.2 and 1.2.3 respectively.

| Contribution | Research Questions | Objectives |
|---|---|---|
| C$_1$ | Q$_1$, Q$_4$ | O$_1$, O$_5$ |
| C$_2$ | Q$_2$ | O$_2$, O$_5$ |
| C$_3$ | Q$_3$ | O$_3$, O$_5$ |
| C$_4$ | Q$_4$ | O$_4$, O$_5$ |

TABLE 1.1: Relationship Between Contributions, Research Questions And Objectives.

We did a prototype implementation of each of the objectives by extending the PyCOMPSs programming model [100]. PyCOMPSs provides a unique high-level abstraction to enable the parallelization of Python applications that is based on sequential programming. Using PyCOMPSs, programmers just have to select methods to be considered as tasks, and the PyCOMPSs Runtime (COMPSs [8]) handles tasks execution asynchronously. Hence, it confirms with the thesis goal to facilitate application programming for end users. A more extensive overview of PyCOMPSs and its runtime system is presented in Chapter 3.

### 1.2.5 Publications

The following list presents the list of scientific publications that has resulted from the contributions of this thesis:

- $C_1$: I/O Awareness
  - *Towards Enabling I/O Awareness in Task-based Programming Models*
    **Hatem Elshazly**, Jorge Ejarque, Francesc Lordan, Rosa M Badia.
    Future Generation Computer Systems (FGCS), The International Journal of e-Science.
    March 2021.
    Impact Factor (JCR): 6.125 (Q1).
- $C_2$: Storage heterogeneity Awareness
  - *Storage Heterogeneity-Aware Task-based Programming Models to optimize I/O Intensive Applications*
    **Hatem Elshazly**, Jorge Ejarque, Rosa M Badia.
    Submitted.
- $C_3$: Hybrid Programming Models to exploit Parallelism
  - *Performance meets Programmability: Enabling Native Python MPI Tasks in PyCOMPSs*
    **Hatem Elshazly**, Francesc Lordan, Jorge Ejarque, Rosa M Badia.
    28th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP).
    November 2020.
    Core Rank: C.
- $C_4$: Eager-release of Task Dependencies
  - *Accelerated Execution via Eager-release of dependencies in Task-based Workflows*
    **Hatem Elshazly**, Francesc Lordan, Jorge Ejarque, Rosa M Badia.
    The International Journal of High Performance Computing Applications (IJHPCA).
    March 2021.
    Impact Factor (JCR): 1.956 (Q2).

## 1.3 Research Methodology

Throughout this thesis, a *Research and Development* method was followed to carry out the main objective of the thesis, which is to provide high-level techniques and abstractions to optimize I/O intensive applications executions on modern infrastructures. More specifically to achieve each of the objectives defined in the Objectives section (1.2.3). The following sections provide in-depth information about the scientific method design (Section 1.3.1), the development strategy (Section 1.3.2) and the validation strategy (Section 1.3.3).

### 1.3.1 Scientific Method Design

To achieve each of the objectives, we carried out a scientific approach that consists of four main phases:

1. *Literature Review Phase*: during this phase, we carried out an in-depth analysis and investigation about the problem, its critically, previous efforts and state of the art solutions that addressed this problem (more details in Part 2) and how can we contribute to solving it. At the end of this phase, we defined the problem statement (Section 1.2.1), the main objectives of the thesis (Section 1.2.3) and our target contributions (Section 1.2.4).

2. *Design and Implementation Phase*: We designed the proposal and its software architecture. Then, we implemented a prototype version ready for validation and evaluation.

3. *Experimentation Phase*: This phase included the activities of testing the prototype implementation to determine the advantages and disadvantages of our proposals. Our prototype was evaluated with real-world use cases, and sythethic use-cases that mimic real application patterns. We carried out our experiments on different large-scale production systems and supercomputers.

4. *Documentation and Dissemination Phase*: Finally, during this phase, we documented a description of the proposal along with a summary of the evaluation results to share it with the research community.

### 1.3.2 Development Strategy

We followed an Agile development strategy for achieving each objective. Following this approach, the objective is completed incrementally in short-to-medium cycles. At the end of each cycle, we investigated and measured the progress-so-far with respect to the final objectives and the main objectives of the thesis as a whole.

It should be noted that in our experience, identifying the main problem from the start along with the target applications has significantly helped us following this strategy and not losing track of the purpose of the objective nor how we are advancing towards solving the main problem statement as a whole.

Knowing the start point (i.e., the problem and the objective) and the end point (i.e., the estimated target and the target application(s)), we started the development in a top-down approach until a prototype implementation is ready for evaluation.

Finally, we go back and forth between development and initial evaluation to tune the prototype implementation and adjust it if necessary when the results of the initial evaluation has clearly diverged from our targets and expectations.

### 1.3.3 Validation Plan

After the initial prototype implementation has achieved the minimum expectations, we carried out an extensive evaluation and validation process with real-world workloads on various large-scale production systems. For example, our validation platforms include the MareNostrum 4 supercomputer [66] and the MareNostrum CTE-Power Cluster [33].

At this phase, the proposal is tested against different uses cases and the results are analyzed to study the advantages and disadvantages of the proposal and how they match the set expectations and advance our investigation towards solving the main problem of the thesis.

## 1.4 Thesis Structure

The rest of the thesis is structured as follows: with respect to the remainder of this Introduction (Part I), Chapter 2 describes previous efforts that have been performed with regard to I/O performance optimization. Followed by Chapter 3 which introduces necessary background information. Next, Part II details the contributions of the thesis. Each contribution is discussed in-detail in a separate chapter. Each chapter starts with a brief overview of the contribution and highlights the related work, and ends with a discussion about the conclusions regarding the contribution.

Part II is decomposed as follows:

- Chapter 4 presents the concepts of I/O awareness in task-based programming models, implementation of these concepts in the PyCOMPSs programming model and a prototype evaluation with different I/O intensive use cases.
- Chapter 5 proposes storage-heterogeneity awareness in task-based models by abstracting the heterogeneity of storage systems, presenting I/O dedicated schedulers with different policies. In addition to an automatic data movement technique for maximizing the utilization of faster storage devices.
- Chapter 6 introduces a hybrid programming model of tasks and MPI inside tasks. This chapter presents the programming abstraction for achieving such execution model along with its benefits in terms of enabling I/O parallelism in tasks while facilitating applications programmability.
- Chapter 7 describes a programming model technique to eagerly releasing tasks dependencies in order to overlap I/O with computation and accelerate applications executions. In addition to a detailed discussion about its performance on real and synthetic workloads.

Finally, Part III contains Chapter 8 to present the conclusions of the thesis and planned future work.

# Chapter 2

# State of the Art

This chapter gives a broad overview of recent research efforts and the state of the art related to this thesis. The specific related work with regard to each contribution is analyzed in more detail in separate sections in Part II (see Sections 4.2, 5.2, 6.2 and 7.2).

This chapter is structured as follows: First, Section 2.1 starts by presenting recent advances in the field of task-based programming models. Next, Section 2.2 describes recent efforts targeting the optimization of I/O performance by using I/O libraries and I/O middleware.

## 2.1 Task-based Programming Models

In recent years, task-based programming models have gained popularity in orchestrating and executing applications on large-scale distributed infrastructures [29]. Such programming models offer an almost-complete abstraction of the underlying large-scale complex infrastructure. Execution monitoring and management is transparently carried out by the programming model.

The main similarity between all of the task-based programming models is the working unit that is called *tasks*. Tasks describe pieces or regions of code that are going to be executed in parallel. The *Task* abstraction simplifies the process of application parallelization on large-scale distributed infrastructures because users only have to specify the tasks, and in some tasking frameworks the data dependencies. Data transfer and management between tasks executing on different nodes will be done transparently by the framework.

Different tasking models follow different approaches for tasks specification. These approaches mainly depend on the philosophy and execution model of the tasking framework. For example, some frameworks such as Aneka [108] and Jolie [71], have the concept of *bag of tasks*, where tasks wait for execution. At execution time, tasks are selected from the bag to be executed. In such scenario, tasks are assumed to be independent, i.e., no order or dependencies are enforced on tasks execution. Therefore, users have to explicitly manage the dependencies and plan the execution order.

Other tasking frameworks such as MapReduce [20] enforce a specific parallelism pattern where a map function is applied to different data in parallel, then the result is reduced according to a reduce function and returned to the user. Using such frameworks, it is the responsibility of programmers to specify the map and reduce functions whereas the execution order will be applied by the runtime system of the framework. Even though these frameworks take care of dependencies and execution order, its applicability to scientific applications that do not conform with the map-reduce parallelism pattern is limited.

Unlike frameworks that follow a rigid parallelism patterns, other frameworks enables application developers to specify the tasks workflow in a custom manner. Following this approach, the only responsibility of application programmers is to decide which regions of code should be considered as tasks, whereas the data dependencies between tasks and their

execution order is decided by the framework itself in the form of a Directed Acyclic Graphs (DAGs). For simplicity, we will call frameworks that follow this approach: *DAG Frameworks*.

Different DAG frameworks offer different ways to specify workflows of tasks. On the one hand, some frameworks require dependencies to be explicitly defined by means of Graphical User Interface (GUI) (such as Galaxy [1] and Taverna [45]), or a Command Line Interface (such as Copernicus [88]), or by an API provided by the programming language (such as Pegasus [21], Luigi [32] and Apache Airflow [4]). Such as approach can increase programming complexity as dependencies specifications need to be adjusted every time a modification needs to be made to the code. Furthermore, it is not practical when dealing with large and complex applications.

On the other hand, other DAG frameworks automatically infer the dependencies and execution workflow from users code. Such frameworks use input and output information between tasks to decide the order in which they should be executed. Examples of these frameworks include: Spark [119], Dask [93] and COMPSs [8]. This approach in workflow specification eases the programming of applications as users can develop applications in almost sequential manner with minimal modifications to code. In addition to that, it increases productivity as it offers a very flexible approach in specifying and changing task workflows.

Furthermore, the granularity of tasks differ between different task-models. Some tasking models and frameworks (such as Luigi) handle tasks to be of granularity of whole application where each task represents an application. This approach is useful to use in situations where applications have to be executed in specific order. However, in other frameworks (such as COMPSs), finer granularity task workflows are specified in which tasks represent specific function(s) in applications.

Due to their simple execution model, DAG frameworks offer the unique capability of providing the means to exploit the unstructured patterns of parallelism. These patterns arise because of the different control paths and execution flows that are increasingly common in nowadays applications.

Task-based programming models and frameworks are available in different programming languages (such as Java/C++/Python). Nevertheless, it is very common for a framework to support binding layers to allow the usage for applications in different programming languages (e.g., Spark supports Java and Python, COMPSs supports Java/C++/Python). However, Python-based tasking models and framework are known for their user friendly abstractions and also the ease of programmability of the Python programming language. Therefore, they are widely used by non-expert domain users and field scientists.

### 2.1.1 Discussion

Although task-based programming models and frameworks have a lot of advantages in terms of: ease of programmability, optimized execution, transparent data management on large-scale distributed infrastructures, all of these efforts are directed towards the optimization of applications computations. However, given that applications performance bottleneck in recent years shifted from the computing bottleneck to I/O bottleneck, current tasking framework offer no support to improve the I/O performance of applications.

Indeed, it is possible that users manually tune the configuration parameters of the execution environment of task-based frameworks to target I/O performance optimization. For instance, many frameworks (such as Dask) allow users to override or specify tasks scheduling parameters such as target nodes or number of tasks to be executed in parallel. Hence, this can be used to plan a scheduling policy where I/O workload is scheduled to specific nodes that have modern storage devices or to execute specific amounts of I/O requests at a time to prevent I/O congestion. However, such intrusive approach of programming and

execution planning results in several drawbacks that increase programming complexity and waste total performance improvement opportunities:

- First, users are required to have knowledge about I/O performance characteristic, I/O performance issues, I/O parameters tuning, etc. in order to plan the execution for improved I/O performance. In addition to the increased complexity, such requirement is not realistic for non I/O expert end users.
- Second, since there is no programming model distinction between I/O and compute workloads, tuning execution parameters for optimizing one type of execution will affect the other. Total application performance will be limited to either optimizing compute or I/O performance but not both. Thus, performance improvement opportunities are wasted.
- Third, details of heterogeneous storage systems are exposed to application programmers. Hence, programming complexity is increased. In addition to that, performance improvement opportunities are wasted because there is no automatic support to make execution time decisions and optimize the usage of modern storage subsystems given a certain I/O workload.
- Finally, related to the previous point, if I/O optimization is planned for a certain infrastructure, it may not yield similar or better performance on a different infrastructure. Thus, portability is not achieved.

Throughout the Contributions part of this thesis (Part II), we address the shortcomings and lack of support in current task-based programming models and frameworks towards addressing I/O performance issues.

## 2.2 I/O Performance Optimization

Extensive research has been done in the area of I/O performance optimization [10]. Those research efforts have been conducted from different perspectives that can be summarized as follows:

- First, new system design innovations to unburden the Parallel File System (PFS) and efficiently manage large number of concurrent I/O requests (i.e., mitigate I/O congestion) (Section 2.2.1).
- Second, optimizing I/O performance by enabling parallel I/O and improving I/O access patterns (Section 2.2.2).
- Third, coordinating the interference between I/O requests of multiple running applications (Section 2.2.3).
- Finally, transparently scheduling of I/O requests and data movement on heterogeneous storage systems (Section 2.2.4).

The next following sections highlight the most common ideas and state of the art in each of the previous perspectives.

### 2.2.1 Burst Buffers

The problem of I/O performance is caused because the increasing amounts of data produced by nowadays applications is outpacing the increase in Parallel File System (PFS) bandwidth [9]. As applications alternate between computationally dominant and I/O dominant execution phases, multiple jobs may access the PFS concurrently. Thus, overwhelming its bandwidth and creating the problem of I/O congestion.

Several works presented a solution to this problem [63], [16], [95]. The idea of this solution is to basically introduce a layer of faster and higher bandwidth storage devices called *Burst Buffers* (BBs). As presented in Section 1.2, Burst Buffers absorb the bursty I/O requests during applications I/O dominant execution phase then flush these requests at a later point of the execution asynchronously as a steady I/O stream to the PFS. Therefore, mitigating the I/O bottleneck at the PFS level.

Furthermore, previous studies have been performed the configuration and setup of the Burst Buffers within the production system [50], [39], [51]. Such efforts describe that Burst Buffers placement can be classified into two approaches: On the one hand, Burst Buffers are be locally attached to computing units. This configuration is called *Node-Local*, and it has the advantage that network congestion is avoided because data are stored locally. On the other hand, Burst Buffers are separated from compute nodes and attached to separate nodes on the network called *I/O Nodes*. Such Burst Buffer configuration is called *Remote-Shared*. In this configuration, the number of I/O nodes is typically less than the number of computing nodes and more than the number of servers in the PFS [109].

In remote-shared Burst Buffer configuration, I/O requests are absorbed in storage nodes which are responsible for handling them as efficiently as possible, then *forwarded* to the Parallel File System. In this configuration, I/O nodes are called *I/O Forwarding Layer*. In the I/O forwarding layer, optimization techniques are performed on the I/O request to improve its access patterns and reduce the load on the PFS. For instance, Vishwanath et al. [111] proposed aggregating requests from the computing nodes to the I/O forwarding layer to enhance the usage of the storage system. Another approach is to use caching and prefetching to hide the latency of remote accesses by caching data in I/O nodes and prefetching it whenever needed as proposed by Zhao et al. [120]. Moreover, several efforts presented adaptive methods to balance the workload to the file system according to the file system performance by monitoring the file system performance and schedule the workload accordingly [110], [118], [79].

In general, the innovation of Burst Buffers have greatly advanced I/O performance optimization research. The next sections give an overview about different I/O optimization research efforts at different perspectives. Each of those research efforts moves from the premise that the underlying storage system is equipped with Burst Buffers on top of a Parallel File System.

### 2.2.2 I/O Libraries

Applications access patterns include I/O requests sizes, number and access locations on disks. Each application has its own access pattern parameters. These parameters depend on how applications were designed and programmed and they have a direct impact on performance. Therefore, a lot of research efforts have been put into investigating how data accesses can be optimized.

One popular technique to boost I/O performance is to use I/O libraries. These libraries are responsible for applications I/O operations and have the power to perform optimizations to adapt to their access patterns. The most popular I/O library is MPI-IO [89] [102],

which enables the parallelization of applications I/O requests by providing a set of programming APIs.

MPI-IO has been extended by high-level I/O libraries such as HDF5 [105] and netCDF [59]. These libraries have two advantages over MPI-IO: On the one hand, they offer more optimizations by storing metadata for applications, such as the datatypes and dimensionality of an array, etc. On the other hand, they are easier to use because they abstract I/O operations by allowing the definition of complex data types and file formats that can be mapped to real files.

Several other I/O libraries have been developed to optimize applications access patterns. For instance, Seelam et al. [98] applies a library that traces and detects the application access pattern. Similar approaches that rely on access pattern detection to guide optimization decisions are proposed by Patrick et al. [86] and Lu et al. [64].

### 2.2.3 System-wide I/O Schedulers

Although I/O libraries can be used by some applications to locally optimize their access patterns, interference produced by multiple applications accessing the shared storage infrastructure might also compromise or decrease the efficiency of the optimizations and degrade the performance. Therefore, some research efforts focused on optimizing I/O performance by managing the I/O interference of different applications running at the same time in the system. In these scenarios, I/O scheduling techniques can be applied to improve access to storage layers by organizing and reordering requests. Such efforts claim that performance improvement is possible by taking into account multiple competing applications and important system-wide performance metrics such as system utilization.

Gainaru et al. [36] proposed a global I/O scheduler that has global view of the system and of the past behaviour of all applications running on it. These information can be used to optimize scheduler heuristics such as maximum efficiency or fairness.

Liang et al. [61] proposed a contention-aware resource scheduling strategy to improve the performance of burst buffers by minimizing I/O congestion caused by I/O of different applications. This strategy analyzes I/O load on the burst buffers nodes and assigns incoming I/O to burst buffer nodes with least I/O load. Herbein et al. [42] incorporated I/O workload scheduling into existing policies such as First Come First Served (FCFS) and EASY backfilling. The idea is to add I/O as additional constraint when determining if a job can be scheduled. Jobs are only scheduled for execution if and only if there are available resources to satisfy their I/O requirements.

Zhou et al. [121] presented an I/O batch scheduler with two policies: conservative and adaptive. The conservative policy avoids I/O congestion as much as possible targeting system-performance metrics. Whereas the adaptive policy allows I/O congestion to happen to increase jobs performance.

### 2.2.4 I/O Systems and Frameworks

The introduction of Burst Buffers into nowadays storage subsystems to absorb bursty I/O patterns helped improving I/O performance by mitigating I/O congestion. However, this heterogeneous design of the system has also increased the complexity of programming and execution planning. Hence, this scenario pushed research attention to innovate software solutions and frameworks to solve this issue.

Previous research efforts targeted abstracting storage and memory subsystem heterogeneity to maximize I/O performance. Systems such as BurstFS [112] and BurstMem [113] propose to redirect I/O calls from PFS to local NVMes and SSDs. Hermes [52] and UniviStor

[114] are frameworks that provide I/O buffering and optimize data movement across different storage devices. DataWarp [41] and Data Elevator [25] enhances applications writes by redirecting writes from PFS to Burst Buffers.

Other efforts proposed by Alturkestani et al. [3] presented a tool for optimizing oil exploration simulation application. This tool maximizes the usage of high bandwidth storage devices and handles data movement between faster storage layers from one side and the PFS from the other side.

### 2.2.5 Discussion

From the aforementioned description of the state of the art in I/O performance optimization research, one can notice that improving I/O performance is a complicated task. Different efforts target solving specific problems at different levels of the system and software stacks. These efforts can be classified into three main categories:

I Access patterns optimization.

II System-wide mitigation of I/O congestion.

III Taking advantage of storage system heterogeneity.

Upon analyizing the state of the art in the light of this categorization, we can identify the following observations:

- First, access pattern optimization techniques can improve applications I/O performance by improving its access patterns. However, such techniques are very application-centric, i.e., they do not provide any support tools to mitigate the problem of I/O congestion.
- Second, system-wide I/O mitigation approaches, unlike the first point, aims at solving I/O congestion using system-wide performance metrics. Therefore, they end up taking global decisions that may not be optimal for a specific application use-case.
- Third, mixing the two previous approach together to reach overall application performance improvement is a complex process, because it requires fine-tuning and previous experience with the application expected I/O workloads and infrastructure details.
- Finally, all these approaches specifically focus on optimizing I/O performance. Users have to deploy these different frameworks and techniques with other frameworks that target improving computation performance. Hence, such an approach increases the complexity and decreases portability and productivity.

Throughout the Contributions part of this thesis (Part II), we address the lack of a suitable user-level abstractions that targets improving I/O performance while maintaining ease of programmability and without sacrificing computation performance. Therefore, improving total application performance and increasing productivity.

# Chapter 3

# Background

This chapter provides an overview on the main frameworks and tools that are used throughout this thesis. First, Section 3.1 introduces the PyCOMPSs/COMPSs framework because this thesis extends its programming model and runtime functionalities as previously mentioned in Section 1.2.3. Next, Section 3.2 presents a general overview on the MPI library due to its predominant use in high performance computing and I/O parallelization. Also, MPI is used in different parts in this thesis.

## 3.1 PyCOMPSs

PyCOMPSs [100] is a task-based programming model used for executing Python applications on large-scale distributed infrastructures. is used to parallelize Python applications. The contributions of this thesis extends on the PyCOMPSs framework. PyCOMPSs relies on the COMPSs [8] framework and runtime system to carry out core functionalities such as tasks analysis, dependencies management, scheduling tasks and monitor their execution. PyCOMPSs allows parallelizing sequential applications by providing a programming interface which is used to specify which functions should be treated and executed as tasks. Due to its powerful programming model, PyCOMPSs makes it easy to exploit the inherent parallelism of applications at execution time by detecting tasks and data dependencies between them. In addition to transparently managing data transfers between tasks executing on different distributed nodes and a support to use Parallel File Systems.

Figure 3.1 illustrates an overview of the COMPSs framework. COMPSs enables parallelizing applications in different programming languages. It natively supports Java applications. In addition to that, it provides bindings for parallelizing and executing Python and C/C++ applications. It should be noted that the Python bindings and the COMPSs runtime together constitute the PyCOMPSs framework. Additionally, COMPSs allows transparent execution on various infrastructures (e.g., clusters, grids and clouds) without modifying the source code of the application. Furthermore, COMPSs is rich with features such as fault tolerance, job failure or infrastructure failure, and has a built-in checkpointing mechanism. Also, COMPSs supports ecosystem framework such as Extrae [34] that generates post-mortem execution traces that can be analysed with the Paraver tool [85].

The main advantages of the PyCOMPSs/COMPSs framework can be summarized in the following points:

- Ease of programmability: users do not have to deal with any parallelism aspects such as thread creation, synchronization or data distribution. Using PyCOMPSs, users should only select which application functions to be executed as task. It is the responsibility of the COMPSs runtime to handle all parallelism details and launch the selected tasks for execution in an asynchronous, parallel and distributed manner.

- Infrastructure Transparency: the PyCOMPSs framework abstracts all infrastructure details from users. Hence, PyCOMPSs applications do not include any configuration

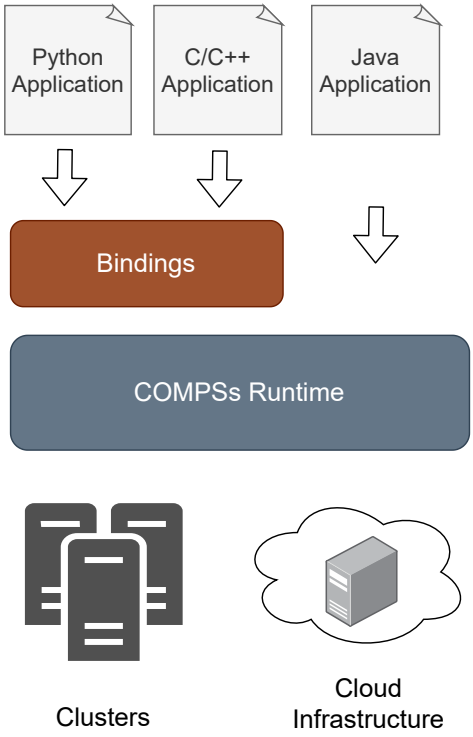



FIGURE 3.1: The COMPSs Framework Overview

of deployment details. Such capability makes PyCOMPSs applications portable across different infrastructures which increases portability and productivity.

### 3.1.1 Programming Model

The PyCOMPSs programming model is based on sequential programming, that is, applications are coded as if they are going to run sequentially. Users only have to select which functions in the application should be executed as tasks. Such selection is made possible by annotating the application code. These annotation can be classified into two type:

- Method Annotation: Annotations added to applications functions to mark such functions as tasks.
- Parameter Annotation: Annotations added to the parameters of annotated functions to handle data dependencies and transfers.

These annotations depend on the programming languages, i.e., COMPSs provides Java annotations for Java applications, Python annotations for Python applications and C/C++ annotations for C/C++ applications. Since the main contributions of this thesis extends the PyCOMPSs programming model, next sections will give an overview of the COMPSs annotations for the Python programming languages.

#### 3.1.1.1 Python

COMPSs uses a Python binding layer to parallelize Python applications. This binding layer exposes the Runtime functionalities to applications programmed in Python. This binding layer communicates with the COMPSs Runtime using the Java Native Interface (JNI) [60].

PyCOMPSs annotations are done inline, that is, in the same source file as the application code. The Method Annotations are in the form of Python decorators. Users can use the `@task` PyCOMPSs decorator to specify a function as a task. This decorator can be used to annotate a class or object methods in applications designed with object oriented paradigm. Once a function is annotated with `@task`, all the invocations of this function will become tasks at execution time.

Furthermore, the Parameter Annotations are specified inside the Method Annotation. For instance, users can specify if a certain parameter is to be read (IN), written (OUT) or both read and written (INOUT) in the method. Parameter annotations helps COMPSs runtime building data dependencies and deciding necessary data transfer during execution.

Listing 3.1 shows an example of a PyCOMPSs task annotation. The parameter $c$ is of type INOUT whereas parameters $a$ and $b$ are set to the default type IN. Even though no explicit return is specified in the example function of Listing 3.1, it will return the updated $c$ parameter because it is of type INOUT.

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

LISTING 3.1: PyCOMPSs Example Task Annotation

The IN, OUT, INOUT directionality parameters are used to specify the parameter annotation of objects. However, the parameters: FILE_IN, FILE_OUT, FILE_INOUT are used to annotate the parameters of file types.

A light-weight synchronisation API completes the PyCOMPSs syntax. As shown in Line 4 of Listing 3.2, the PyCOMPSs function `compss_wait_on` waits until all the `word_count` and `reduce_count` tasks finishes executions then retrieves the result to the master node that has launched the application. Once the value is transferred to the main node, the execution of the main program code is resumed. Notice that because PyCOMPSs is mostly used for executing applications in distributed environments, using the synchronization APIs may imply a data transfer from the remote node where the task calculating the value has been executed to the master node that is executing the main part of the application.

```
1 for block in data:
2     partial_result = word_count(block)
3     reduce_count(result, partial_result)
4     final_result = compss_wait_on(result)
```

LISTING 3.2: PyCOMPSs Example: Retrieving Task Result

Taking a closer look at Listing 3.1 and Listing 3.2, one can note that the code is almost the same as the code of a sequential application. However, few lines of codes should be added to mark tasks by annotating functions and retrieve the results to the master node if necessary by using the synchronization API.

In addition to `compss_wait_on`, the PyCOMPSs programming model provides other APIs to perform different functions. These APIs and a brief description of their functionalities are listed in Table 3.1.

| PyCOMPSs API Call | Use |
|---|---|
| `compss_wait_on(object)` | synchronises the given object and returns it to the master node. |
| `compss_barrier()` | synchronises the execution and waits the completion of all the previous tasks. |
| `compss_delete_object(object)` | requests the deletion of the given object. |
| `compss_open(file_name, mode='r')` | synchronises the given file and returns its associated file descriptor. |
| `compss_delete_file(file_name)` | requests the deletion of the given file. |

TABLE 3.1: List Of PyCOMPSs API Calls

In addition to the `@task` annotation for marking functions as tasks, the PyCOMPSs programming model provides additional annotations that extend the functionalities of the system. For instance, the `@constraint` to allow enforcing hardware of software requirements on tasks execution. The COMPSs runtime will not launch any executions of a constrained task unless all task requirements are satisfied. The `@constraint` allows specification of different parameters to specify the type of the constraints, for instance: `ComputingUnits` for specifying the number of required CPUs for task execution, `MemorySize` for specifying the required memory size. Listing 3.3 shows the same code in Listing 3.1 with added constraints of computing units of 8 CPUs. For a complete list of the annotations and parameters allows by the PyCOMPSs programming model, refer to the COMPSs User Guide: Application Development [90].

```
1 @constraint(ComputingUnits=8)
2 @task(c=INOUT)
3 def multiply(a, b, c):
4     c += a * b
```

LISTING 3.3: PyCOMPSs Example Task Annotation

### 3.1.2 COMPSs Runtime System

The COMPSs runtime system is deployed on execution infrastructures as a master-worker architecture. It has two main components:

- Master Component: Only one process running on the master node where the application has been launched. It is responsible for provisioning the whole execution environment (launching and terminating worker components), interacting with the user code, in addition to detecting data dependencies, schedule tasks and manage their executions.
- Worker Component: As many processes as available CPUs. It is responsible for receiving tasks execution request from the master and perform the actual task execution.

The master and worker component communicate with each other over the network using various adaptors and communication libraries to coordinate the distributed execution and data transfer of the application.

Figure 3.2 depicts the main components of the COMPSs master. These sub-components and their uses can be summarized as follows:

- Task Analyzer: responsible for detecting dependencies between tasks.
- Task Dispatcher: which consists of the Task Scheduler that is responsible for scheduling tasks to a suitable resource and a submission engine for submitting tasks to be executed. The COMPSs runtime allows transparent use of multiple schedulers, for instance: Load Balancing, Data Locality, etc. Desired scheduler is passed as a command-line argument at launch time, else the default Load Balancing scheduler will be used.

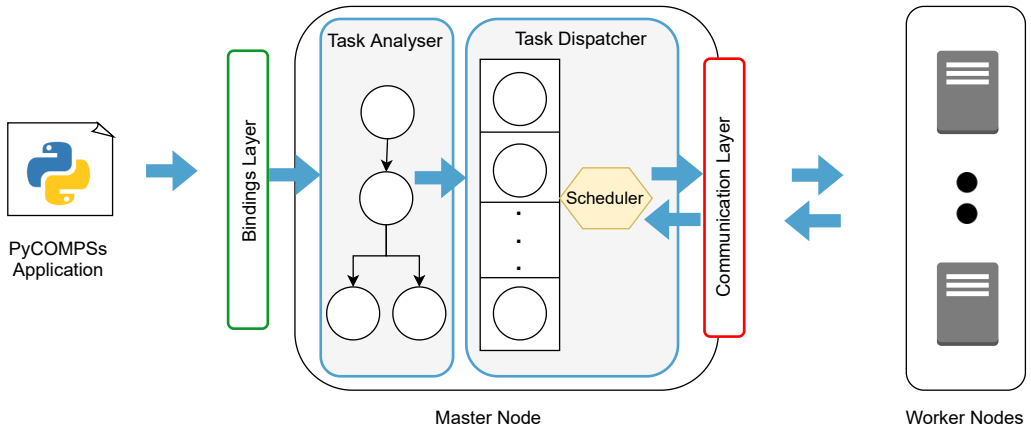



FIGURE 3.2: The COMPSs Runtime Overview

Once the master and worker components have been successfully launched, the Python binding interprets the used decorators in the application code. Then, it performs the necessary calls to the COMPSs runtime through the JNI. When the COMPSs runtime receives task execution requests, the Task Analyzer checks the task dependencies. If the task is dependency free, the Task Scheduler takes over to try to schedule the task to one of the available resources. Next, the Task Dispatcher of the master component sends the task execution request to the worker through a communication layer. Once the worker component on a certain worker node receives a task execution request, it assigns the task to one of the available processes to be executed. The worker monitors the task execution and reports the success or failure status to the master to take the necessary procedure.

In addition to the Task Analyser and the Task Dispatcher, the COMPSs master contains other components such as the Resource Optimizer. The Resource Optimizer is responsible for performing optimization decisions during applications execution. For instance, supporting infrastructure elasticity by adding or removing worker nodes at execution time according to the workload and requirements of applications. The Resource Optimizer runs on a separate process, hence, critical components of the system such as the Task Scheduler do not get blocked or burdened by doing extra operations.

Figure 3.3 depicts an overview of the architecture of the worker component of the COMPSs runtime. Every worker component consists of a Java process called the *Execution Manager* which is responsible for setting up and managing an execution platform at the launch time of the application. The execution platform consists of a Java thread pool with as many threads as the number of CPUs on the worker. These Java threads are called *Executors*. Each Java executor handles the execution of one task. Furthermore, each Java executor launches a Python process, called *Python Worker*, that will ultimately carry out the execution of the task. Java executors and Python workers communicate with each other using operating system pipes.

It should be noted that in order to facilitate the communication between the Java and Python components of the system and data transfer between nodes, tasks outputs are stored in a serialized format.

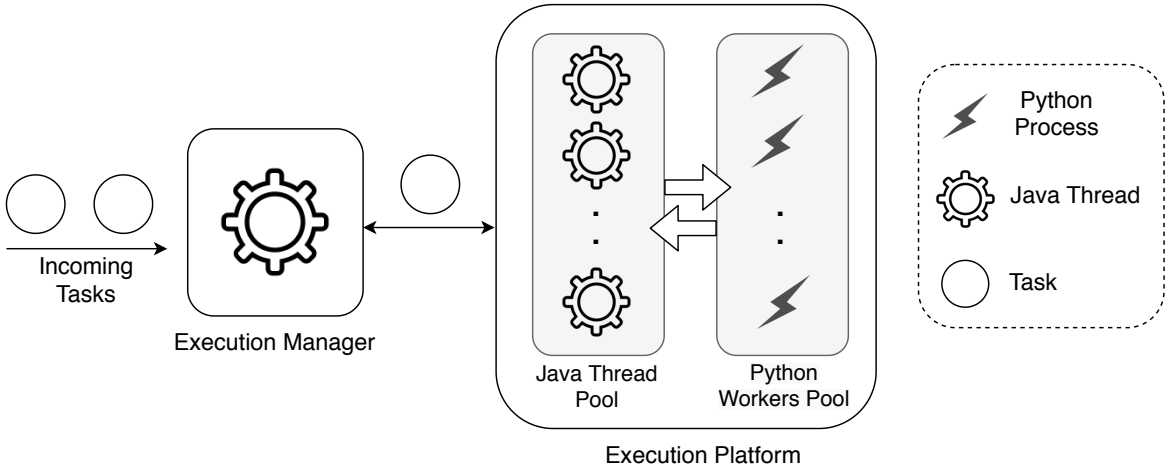



FIGURE 3.3: COMPSs Worker Component

## 3.2 Message Passing Interface (MPI)

The Message Passing Interface (MPI) is a standard for writing and executing message passing programs [72]. For a long time, it has been regarded as the de-facto standard for performing distributed memory parallelization and execution. The MPI standard provides two modes for inter-process communication: (i) *point-to-point communications* where only a sender process and receiver process are involved and (ii) *collective communications* where all the processes participate in messages the sends or receives calls. In addition to that, the standard describes APIs that allow wide range of functionalities such as process topologies, execution environment management, one-sided communications, profiling interface, among others [72]. Several implementations are available such as MPICH [73], OpenMPI [81], IMPI [48] and MVAPICH [75]. These implementations provides bindings for parallelizing and executing C and Fortran applications.

Parallelizing applications with MPI requires explicitly handling the spawning of processes and the communication between processes. All MPI processes execute the application, unless specified otherwise by conditional statements. MPI application must be compiled and executed by the MPI compiler and launch manager provided by the MPI implementation. This approach allow users full control of application flow and execution which, if done correctly and efficiently, can yield high performance improvement on the underlying infrastructure. However, achieving high performance MPI executions requires experience with different concepts of parallel computing, performance analysis and optimization. Such requirements are not suitable to inexperienced users with little background on these topics. In this scenario, application programming, debugging and optimization become a tedious process. Users become responsible for explicitly planning and distributing data between processes and collect the result to the desired process.

Listing 3.4 shows an example of MPI application written in C. The code shows a point-to-point communication where a number of processes is spawned at application launch time, one process is set as the m̈asterẅhile the rest of the processes as ẅorker.̈ In a typical MPI application, the master process distributes the data and receives the final results, whereas the worker processes receives the data from the master, do some computation, then send

back the result to the master. As shown in Listing 3.4, the send and receive communications between different processes are realized using the *MPI_Send* and *MPI_Receive* API calls.

```
#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 3.4: Simple MPI Application Written in C.
Source: Wikipedia, Message Passing Interface

Listing 3.5 shows the command-line used to launch the MPI application of Listing 3.4, and its result. Users specify the number of MPI processes at launch time, in this case 4 MPI processes are specified: process 0 is the master process and the rest are worker processes. Notice that exact order result is not guaranteed every time the program is launched, since MPI processes are executing in parallel.

In addition to providing a standard for optimizing computation, MPI also provides APIs to parallelize I/O operations through the MPI-IO library [89]. These parallel I/O APIs draw a lot of inspiration and similar to the original MPI APIs, for instance: writing is similar to sending data and reading is like receiving data, support for collective operations, user-defined data types to describe files layout, etc. In addition to that, MPI-IO is able to carry about access pattern guided optimizations provided by its popular implementation ROMIO

```
$ mpicc hello_mpi.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 3.5: Simple MPI Application Execution

[101]. MPI-IO APIs can be used to optimize I/O and n̈ormalM̈PI APIs can be used to optimize computations simultaneously in the same application.

MPI-IO exposes APIs that allow users to parallelize I/O in a similar flow to sequential POSIX I/O: open a file, read or write data to the file, then close the file. In addition to that, it provides different modes for doing parallel I/O: multiple processes access different files where a single process accesses a single file, or multiple processes access the same file.

Listing 3.6 shows a simple MPI-IO application written in C. The code simply writes a buffer to a file. Line 12 opens a file in writing mode only, if the file does not exist, it will be created. Indeed, similar to POSIX APIs, different access modes can be specified. File open and close operations are done collectively whereas file reads and writes are done independently, hence the process rank check on Line 16.

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char *argv[]){
5      MPI_File fh;
6      int buf[1000], my_rank;
7      /* Initialize the infrastructure necessary for communication */
8      MPI_Init(&argc, &argv);
9      /* Identify this process */
10     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
11     /* Create a file and open it in write mode */
12     MPI_File_open(MPI_COMM_WORLD, "example.out",
13                   MPI_MODE_CREATE|MPI_MODE_WRONLY,
14                   MPI_INFO_NULL, &fh);
15
16     if (my_rank == 1) {
17         /* Write the content of buf to the file */
18         MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
19     }
20     /* Close the file */
21     MPI_File_close(&fh);
22     /* Tear down the communication infrastructure */
23     MPI_Finalize();
24
25     return 0;
26 }
```

LISTING 3.6: Simple MPI-IO Application Written in C

It should be noted that MPI-IO applications are compiled and launched in the same manner as non MPI-IO applications similar to what is shown in Listing 3.5. The number of spawned MPI processes are specified at application launch time.

Indeed, in order for MPI-IO to be effective, it requires a system level support to parallel I/O operations, i.e., a parallel file system and hardware that support concurrent accesses.

# Part II

# Contributions

# Chapter 4

# Enabling I/O Awareness in Task-based Programming Models


## SUMMARY

Storage systems have not kept the same technology improvement rate as computing systems. As applications produce more and more data, I/O becomes the limiting factor for increasing applications performance. I/O congestion caused by concurrent access to storage devices is one of the main obstacles that cause I/O performance degradation and, consequently, total performance degradation.

Although task-based programming models made it possible to achieve higher levels of parallelism by enabling the execution of tasks in large-scale distributed platforms, this parallelism only benefits the compute workload of the application. Previous efforts addressing I/O performance bottlenecks either focused on optimizing fine-grained I/O access patterns using I/O libraries or avoiding system-wide I/O congestion by minimizing interference between multiple applications.

This chapter focuses on solving research questions: $\mathbf{Q}_1$ and $\mathbf{Q}_4$ by proposing to enable *I/O Awareness* in task-based programming models for improving the total performance of applications. An I/O aware programming model is able to create more parallelism and mitigate the causes of I/O performance degradation. On the one hand, more parallelism can be created by supporting special tasks for executing I/O workloads, called *I/O tasks*, that can overlap with the execution of compute tasks. On the other hand, I/O congestion can be mitigated by constraining I/O tasks scheduling. This chapter proposes two approaches for specifying such constraints: explicitly set by the users or automatically inferred and tuned during application's execution to optimize the execution of variable I/O workloads on a certain storage infrastructure.

In this chapter, we describe how to implement such proposals by extending the PyCOMPSs programming model and runtime system. Furthermore, in the evaluation section of this chapter, a performance evaluation of the implementation is performed with different use cases, each exhibiting different I/O workloads. The evaluation experiments on the MareNostrum 4 Supercomputer demonstrate that using the I/O aware implementation of PyCOMPSs can achieve significant performance improvement in the total execution time. This performance improvement can reach up to 43% of total application performance as compared to the traditional I/O non-aware implementation of PyCOMPSs.

## 4.1 Overview

The continuous growth in computing power has the ability to deliver increasing levels of parallelism to satisfy the computing demands of scientific applications. In order to harness this increasing computing power and turn it into performance improvements for applications, task-based programming models offer a flexible approach for parallelizing and executing applications in distributed platforms.

Along with the demand for computational power, scientific applications are also becoming I/O intensive where I/O performance dominates the total performance of applications. This paradigm shift has been already observed in critical scientific areas such as computational biology and climate science. Applications in these disciplines generate large amounts of data usually for checkpointing intermediate results and restarting after failures [47], or for performing post-processing operations such as visualization and post-mortem analysis [14]. The life cycle of these applications typically alternates between a computing phase followed by an I/O phase. During the I/O phase, large amounts of concurrent I/O requests overwhelm the I/O bandwidth of the storage system causing I/O congestion. I/O congestion causes significant slowdown in the I/O performance of applications [36]. Consequently, I/O performance degradation negatively affects applications total performance.

As discussed in Chapter 3 of Part I, improvements in storage devices such as solid state drives (SSDs) and non-volatile memories (such as NVRAM) have been introduced to supercomputers besides the traditional parallel file system to absorb the I/O of applications. These hybrid storage solutions have been gaining popularity because simply replacing all hard disks by higher bandwidth storage drives would have a high cost.

Nevertheless, as the amount of data generated by applications continues to grow, relying only on Burst Buffers is not enough to completely hide or mitigate I/O congestion. There must be a software-support for provisioning the use of the storage system.

However, current task-based programming models do not offer support targeting the I/O bottleneck of I/O intensive applications. Addressing I/O performance problems is conventionally done using low-level I/O libraries (e.g., MPI-IO, HDF5) that focus on parallelizing and optimizing I/O access patterns of applications. However, this approach is limited, as it cannot take advantage of coarse-grained performance improvements opportunities. Additionally, it does not take into account the problem of I/O congestion.

Other efforts addressed I/O congestion as a global scheduling problem using global I/O aware schedulers to optimize whole system utilization by minimizing I/O interference between different applications running on the system. However, this direction does not offer programming support to express opportunities of I/O performance improvements that are inherent in I/O intensive applications. Moreover, it focuses on optimizing system-wide performance metrics when handling the I/O of different running applications instead of optimizing the total performance of applications.

This chapter addresses the lack of support for mitigating the I/O performance bottleneck in task-based programming models. We argue that task-based programming models offer a suitable abstraction that can be leveraged to exploit parallelism opportunities in I/O intensive applications to improve total performance. On the one hand, I/O workloads can be wrapped by tasks whose execution overlap with the execution of compute tasks. Hence, application parallelism is increased. Furthermore, fine-grained I/O libraries can be still used for I/O optimization inside tasks (more details on this are discussed in Chapter 6). On the other hand, task-based programming models have application-level information such as the number of I/O tasks and the I/O bandwidth requirement of each task, that can be used to manage I/O congestion.

More specifically, this chapter proposes enabling *I/O Awareness* in task-based programming models. The main objective of an I/O aware task-based programming model is to

improve the performance of I/O intensive applications by exploiting their inherent performance improvement opportunities. To this end, I/O aware task-based systems should support the following capabilities:

- First, increasing task parallelism by defining *I/O Tasks* to handle I/O workloads execution. I/O tasks execution can be overlapped with compute tasks execution.
- Second, managing I/O congestion by controlling I/O tasks scheduling through constraining tasks execution.

The proposals of this chapter are realized by implementing the aforementioned I/O awareness capabilities in the PyCOMPSs task-based programming model [100]. I/O aware PyCOMPSs allows users to set constraints to I/O tasks to control their scheduling. Since the value of this constraint is fixed for the whole application execution, we call this a *Static* constraint.

However, identifying a suitable constraint at application development time may be complex due to the lack of information about the amount of I/O that the application will produce and I/O performance on a given infrastructure. Therefore, we propose an automatic and abstract constraining mechanism that is exposed by the execution manager to settle on and tune the constraints of I/O tasks during application's execution. Hence, offering greater flexibility and portability. This mechanism carries out a performance exploration process to identify the optimal manner to execute I/O tasks on a given system. We call this type of constraints: *Auto-Tunable* constraint. Using auto-tunable constraints, the burden of identifying the optimal constraint is removed by making the runtime system automatically infer and tune I/O tasks' bandwidth constraints with the goal of achieving total time performance benefit.

In the evaluation section of this chapter, we validate the implementation prototype by applying these capabilities in a set of applications with different I/O workloads on the MareNostrum 4 supercomputer. In addition to that, we compare their execution with a version that is not using the I/O awareness capabilities.

The rest of the chapter is structured as follows. Section 4.2 discusses related work. Section 4.3 presents the main concepts and capabilities of an I/O aware task-based system. Section 4.4 presents the design and implementation of the I/O awareness capabilities in PyCOMPSs: I/O tasks and storage bandwidth constraints. Section 4.5 analyzes the performance results of I/O aware PyCOMPSs on the MareNostrum 4 supercomputer using different I/O workloads. Finally, Section 4.6 offers a conclusion and a discussion about the achieved performance.

## 4.2 Related Work

As the design of large-scale execution systems headed towards distribution to provide more computing power by adding more computing nodes, task-based programming models has enabled optimized applications execution without exposing the infrastructure details to application developers. Therefore, they became the go-to programming models for executing domain-specific applications [29].

This section presents a brief description about some of the most popular task-based models and highlights the lack of support to improve I/O performance of applications. In addition to that, this section gives a brief description of I/O optimization efforts related to the contribution of this chapter.

Parsl [7] uses function decorators to compose workflows. Parsl provides a different set of extensible executors to address different parallelization requirements of applications and enable execution on different platforms.

Luigi [32] enables the explicit specification of dependency graphs. Using Luigi, users have to use the provided object oriented API to explicitly define dependencies in the code rather than annotating functions. At execution time, Luigi builds the execution graph by inspecting defined dependencies.

The Dask [93] Python library implements parallel versions of Python libraries such as Numpy [80] and Pandas [68]. Dask enables specifying constraints on tasks execution which is similar to one of our contributions. Additionally, it is possible to explicitly specify zero CPU requirements for tasks which can allow for overlapping I/O and computation. However, the Dask runtime does not provide an automatic mechanism for setting and tuning constraints such as the one proposed in this chapter.

Unlike the proposal of this chapter, the aforementioned tasking models do not have the notion of I/O tasks and there is no support for I/O-compute tasks overlap. Moreover, there is no programming model support for addressing I/O congestion.

As mentioned in Section 2.2 of Part I, numerous studies have been performed to address I/O performance problems at different levels. A traditional approach to improve applications I/O performance is to use I/O libraries, such as MPI-IO [89], HDF5 [105] and NetCDF [59]. These libraries provide a programming API to manipulate data access. Therefore, applications do not need to assume the POSIX interface. MPI-IO provides a low-level interface to enable parallel I/O. This interface can be used to define how to access a file system to perform parallel I/O operations. On the other hand, HDF5 and NetCDF provide file formats that optimize the storage of large amounts of data by stipulating their formats and performing low-level optimizations.

Using I/O libraries allows for fine-grained I/O optimization related to I/O access and storage. However, this approach does not address the problem of I/O congestion. Another limitation of this approach is portability; once these libraries are used in an application for a specific platform, it is not a straightforward task to use it on other platforms.

Targeting a solution for I/O congestion, efforts in this direction have addressed I/O congestion as a classical scheduling problem with the objective of optimizing for system-wide performance metrics. Gainaru et al. [36] proposed a global I/O scheduler that has a global view of the system and of the past behaviour of all applications running on it. These information can be used to optimize scheduling heuristics such as maximum efficiency or fairness.

Liang et al. [61] proposed a contention-aware resource scheduling strategy to improve the performance of the burst buffers by minimizing I/O congestion caused by I/O of different applications. This strategy analyzes I/O load on the burst buffers nodes and assigns incoming I/O to burst buffer nodes with least I/O load. Herbein et al. [42] incorporated I/O workload scheduling into existing policies such as First Come First Served (FCFS) and EASY backfilling. The idea is to add I/O as additional constraint when determining if a job can be scheduled. Jobs are only scheduled for execution if and only if there are available resources to satisfy their I/O requirements.

Zhou et al. [121] presented an I/O batch scheduler with two policies: conservative and adaptive. The conservative policy avoids I/O congestion as much as possible targeting system-performance metrics. Whereas the adaptive policy allows I/O congestion to happen to increase jobs performance.

Tillenius et al. [106] proposed a predictive model for task performance degradation and a resource aware scheduling policy by enabling users to set constraints for tasks execution using task annotation, similar in spirit to one of our contributions.

Unlike global I/O schedulers, our proposal does not target optimizing any system-wide performance metric, nor does it have any information about any other applications running on the system. I/O aware PyCOMPSs addresses I/O congestion from the view point of the application to increase its total performance. We consider that the previously mentioned

efforts targeting I/O performance improvement (i.e., I/O libraries and I/O schedulers) can jointly work along with our proposal to achieve optimal application performance while improving the system-wide performance targets.

It should be noted that these research efforts focused on addressing I/O congestion for write-intensive applications. In these applications, I/O congestion occurs when data are sent to be written to the storage device. This happens when the data are flushed from the system caches because they cannot fit in them or when caching is avoided altogether by doing direct I/O. Direct I/O is beneficial to avoid data loss if failure happened when the data are still in the system caches. However, in the case of reading, caching offers better performance than directly accessing the storage devices every time data are needed.

## 4.3 I/O Awareness in Task-based Programming Models

As previously mentioned in Section 1.1.2 in the Introduction (Part I), I/O intensive applications follow a periodic pattern of alternating compute-I/O phases. Figure 4.1 shows a high-level abstraction of the life cycle of such applications. Each compute phase is followed by an intense burst of I/O. It should be noted that this model assumes that the data consumed by the i-th I/O phase cannot be invalidated by the $i+1$ compute phase. This assumption is valid if the $i+1$ compute phase is independent from previous compute phases. Otherwise, if there is a data dependency between two compute phases (e.g., the $i-th$ and $i+1$ compute phase), then the $i+1$ compute phase should receive an independent copy of the data so as to not invalidate the data consumed by $i-th$ I/O phase.

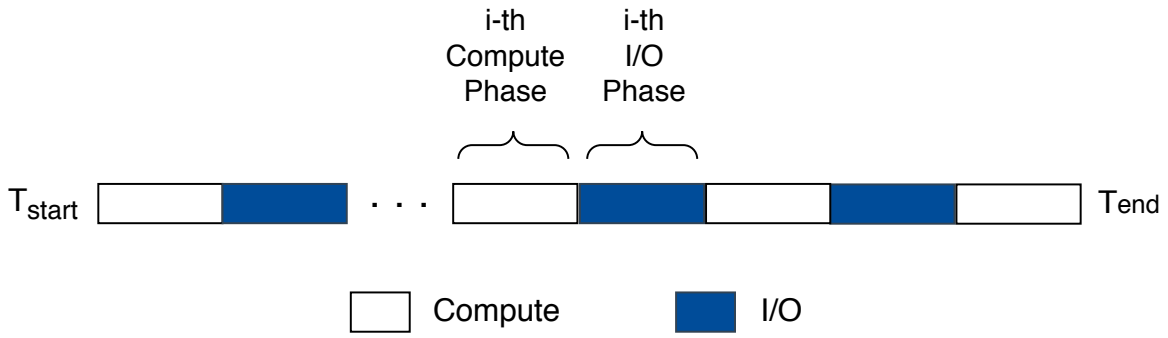



FIGURE 4.1: Life Cycle Of An I/O Intensive Application

Taking a closer look at Figure 4.1, one can observe optimization opportunities that are possible due to the compute-I/O workloads pattern. For instance, the start of each compute phase ($i+1$) is delayed until the previous I/O phase ($i$) has finished. However, since there is no dependency between each I/O phase and the following compute phase, their execution can be overlapped.

A traditional task-based system will not be able to exploit the parallelism opportunities of applications with such pattern. Figure 4.2 shows a high-level abstraction of how a traditional system can be used to execute an I/O intensive application. Using such models, computation and I/O are executed in one task without the ability to differentiate between each workload. Using such approach wastes a lot of performance improvement opportunities for both computation and I/O. This is due to two reasons:

- Application parallelism is decreased because computing resources cannot execute any compute workload while they are waiting for I/O completion.
- The scheduling of workloads is limited; tasks scheduling can be optimized for either compute performance or I/O performance but not for both. For instance, launching

more computing workloads in parallel usually results in more performance. However, in the case of I/O workloads it could result in increasing I/O congestion.

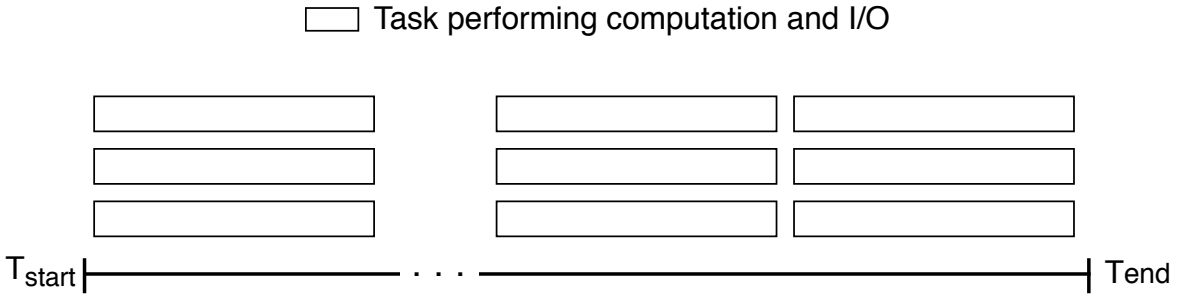



FIGURE 4.2: I/O Intensive Application Executed With A Traditional Task-based Programming Model

To take advantage of the performance improvement opportunities present in Figure 4.1, we propose enabling *I/O awareness*. I/O Awareness enables task-based models to separate compute and I/O workloads. Therefore, allowing for the optimization of each workload depending on its properties. Figure 4.3 shows the life cycle of an application executed with an I/O aware task-based model. This application has two types of tasks: tasks that execute compute workloads and tasks that execute I/O workloads. In an I/O aware execution, compute workloads execution can be overlapped with the execution of dependency free I/O workloads, i.e., the I/O workloads executed by tasks which their data dependencies are satisfied and can be released for execution. Thus, the level of parallelism is increased due to the overlapping execution of tasks. In addition to that, I/O workloads can be scheduled independently from compute workloads to improve I/O performance by minimizing I/O congestion.

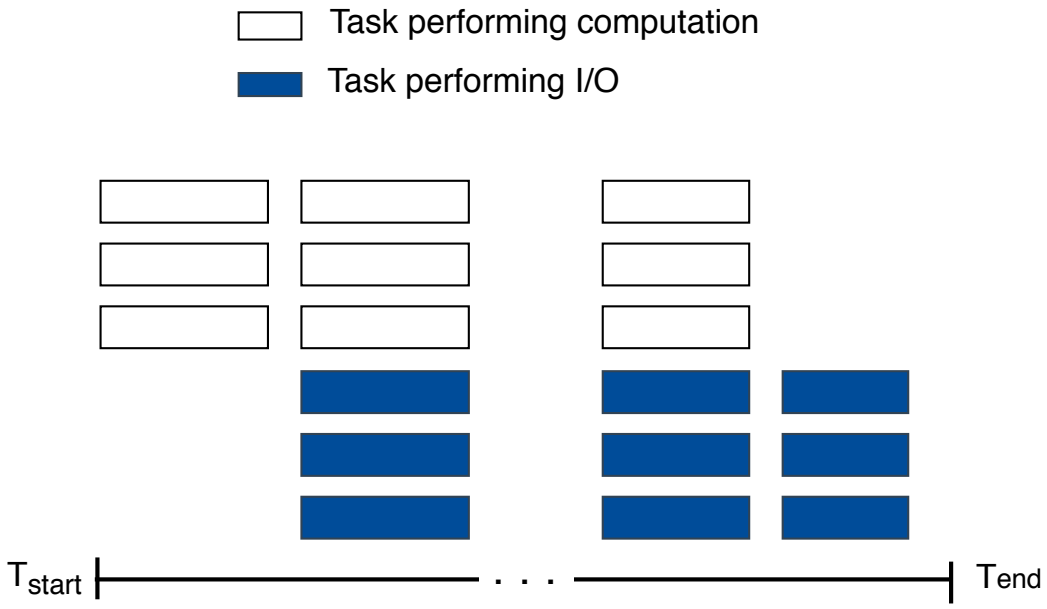



FIGURE 4.3: I/O Intensive Application Executed With An I/O Aware Task-based Programming Model

The following sections introduce in-depth the core concepts and features of an I/O aware system. Sections 4.3.1 to 4.3.3 present the I/O awareness capabilities that we propose to be supported by task-based models: Section 4.3.1 introduces the concept of *I/O Tasks*, which allow to take advantage of the proprieties of I/O workload to increase task parallelism by overlapping tasks execution. Next, Section 4.3.2 focuses on addressing I/O-specific performance problems such as I/O congestion by constraining tasks scheduling. Finally, Section 4.3.3 proposes the automatic inference of task constraints during application's execution.

### 4.3.1 I/O Tasks

We define *I/O tasks* as special tasks for exclusively executing I/O workloads in applications. Indeed, I/O tasks can execute a single I/O request or several consecutive requests (e.g., a loop of write accesses). Using I/O tasks, a programming model should be able to differentiate between I/O workloads and compute workloads and exploit their differences to create optimization opportunities.

Unlike current task-based systems that schedule all tasks based on computing constraints, I/O aware systems should be able to schedule I/O tasks according to their I/O bandwidth requirements instead of computing requirements. Hence, I/O tasks scheduling will not be bounded by the capabilities of the computing infrastructures nor by the compute workload in the application because their scheduling will not depend on the availability of computing units. This approach allows the scheduling of as many concurrent I/O tasks to reach the peak performance of the storage infrastructure, even if the number of running I/O tasks exceeded the available computing resources.

In addition to that, the execution of I/O tasks can be overlapped with the execution of compute tasks. Dependency-free I/O tasks can be executed along with compute tasks on the same CPU with negligible impact on the performance of compute tasks. This takes advantage of CPUs being idle during I/O execution. This capability increases task parallelism in applications because computing resources will not be occupied solely for executing I/O workloads and progress can be made in terms of executing compute tasks.

Moreover, Using I/O tasks to identify I/O workloads enables the scheduling of I/O tasks to specialized storage subsystems in distributed heterogeneous infrastructures. Because these heterogeneous storage subsystems can offer higher bandwidth, more I/O tasks can run concurrently without causing I/O congestion. The dedicated scheduling of I/O tasks to heterogeneous storage infrastructures is another contribution of this thesis that is presented and described in the next chapter (Chapter 5).

### 4.3.2 Storage Bandwidth Constraints

The second capability that we propose to enable I/O awareness in task-based systems is the ability to address the problem of I/O congestion. I/O congestion mainly occurs because the aggregate amount of data to be written by the concurrently running I/O tasks surpasses the maximum I/O bandwidth of the storage devices. Consequently, assuming that the I/O bandwidth is fairly allocated between concurrent I/O tasks, the more I/O tasks running concurrently, the less I/O bandwidth would be allocated to serve the requirements of each task. Therefore, the execution time of I/O tasks increases leading to not only degraded I/O performance but also total performance degradation. Therefore, I/O awareness which only provides increased parallelism by supporting I/O tasks may not be enough to achieve total performance improvement.

I/O congestion can be tackled by constraining the scheduling of I/O tasks. Using constraints to control I/O tasks scheduling guarantees that only a maximum number of tasks

can run concurrently at any given time of application's execution. Hence, I/O bandwidth can be managed and I/O congestion can be minimized or completely avoided.

To this end, we propose enabling the use of *Storage Bandwidth* constraints to specify an estimate for the storage bandwidth requirements of a task. Using the storage bandwidth constraint implies that if scheduling a task will over-allocate the storage bandwidth of the available storage resources, then this task should not be scheduled even though compute resources are idle. Tasks with storage bandwidth constraint will only be scheduled if there is available storage bandwidth to satisfy their constraints. Otherwise, they will wait for more storage bandwidth to become available.

One approach to specify the storage bandwidth constraints is to extend the programming model to allow users to set it at application development time. Hence, users can plan the execution of applications by controlling the level of I/O tasks parallelism that would benefit their applications on a given infrastructure.

Another approach is described by the following section which is to give programming model and runtime support to enable the automatic inference and tuning of constraints based on I/O performance given a certain I/O workload and a specific storage system state.

### 4.3.3 Automatic Inference of Storage Bandwidth Constraints

Identifying a suitable storage bandwidth constraint that minimizes I/O congestion and improves I/O and total application performance may be difficult at application development time. Indeed, a high storage bandwidth constraint leads to a lower number of concurrent tasks and more bandwidth allocated to each task. Hence, I/O congestion will be minimized and I/O performance will increase but task parallelism will decrease. Similarly, a low storage bandwidth constraint will allow more tasks to be executed concurrently but less I/O bandwidth will be allocated for each task. Hence, task parallelism will increase but increased I/O congestion will negatively affect application's performance. Therefore, a balanced constraint value is challenging to identify at application design time because it depends on execution time information. For instance, the amount and size of application's I/O workload, in addition to the I/O performance on a given storage system. Hence, a non-educated choice of the storage bandwidth constraint may lead to non-optimal performance and decreased portability.

To overcome the aforementioned difficulties and improve the programming experience and applications performance, we propose that I/O aware systems support the automatic inference of storage bandwidth constraints. The main objective of this mechanism is to allow the runtime system to automatically estimate a constraint that is not very high so it allows more I/O tasks to run concurrently in order to maximize task parallelism. At the same time, this constraint should not be very low so it minimizes I/O congestion as much as possible by avoiding the launch of a lot of I/O tasks concurrently.

More specifically, the automatic inference of a constraint is the process of finding a constraint value that maximizes task parallelism and minimizes I/O congestion. During application's execution, the runtime system should use a constraint that would minimize the execution time of the I/O tasks waiting to be scheduled.

In order to identify such a constraint, we propose a two-steps mechanism:

- First, the runtime system runs a *learning phase*, in which it collects information about the I/O tasks performance with different levels of I/O tasks parallelism (i.e., different constraint values).
- Second, the information collected during the learning phase is applied to a heuristic function with the objective of minimizing the execution time of the I/O tasks to be scheduled.

Section 4.3.3.1 describes in more detail the learning phase, whereas Section 4.3.3.2 presents the objective function for setting an optimal constraint given a number of I/O tasks.

#### 4.3.3.1 Learning Phase

During the learning phase, the system keeps track of the average I/O task time when running different number of concurrent I/O tasks. Indeed, the number of concurrent I/O tasks at any moment of the application execution is controlled by the value of the constraint that is used. Therefore, the system tries different constraint values to launch different number of concurrent I/O tasks.

The learning phase consists of several *Learning Epochs*. In each learning epoch, the system uses a different constraint value to launch different number of concurrent I/O tasks. The purpose of each learning epoch is to identify the average I/O task time when using a certain constraint value. Therefore, a learning epoch can be defined as the set of I/O tasks that are allowed to run concurrently when using a certain constraint.

It should be noted that the lifetime of an epoch is not defined by any time limits nor by any assumptions based on the task-graph properties. For example, if the maximum number of tasks allowed to run concurrently when using a certain constraint is 5, then the lifetime of the learning epoch of this constraint is the execution time of the 5 concurrent tasks. Once the average I/O task time of the maximum number of tasks allowed to run concurrently at any given time is obtained, a learning epoch is ended and the next epoch (where different constraint is used) is started.

Different approaches can be used to determine the details of the learning phase (i.e., the number of learning epochs and how to progress the learning phase). We propose two types of auto-tunable constraints: *bounded* and *unbounded*.

In the case of the bounded auto-tunable constraint, three values can be used to control the learning phase: *minimum* and *maximum* constraint values that set the boundaries of the constraint and a *delta* value which represents the step size that allows the progression from the *minimum* value to *maximum* value. The first learning epoch in the learning phase starts with the *minimum* constraint value, then the learning phase progresses until it reaches the *maximum* constraint value by multiplying the current constraint by the value of *delta*.

Whereas in the case of the unbounded automatic constraint, no values are used to bound or guide the learning phase, instead such values are estimated by the runtime system. An unbounded auto-tunable constraint starts with the lowest constraint value that would allow the maximum number of I/O tasks to run concurrently. After each learning epoch, the constraint value is doubled and used as the constraint of the next learning epoch. Doubling the constraint would progress the constraint value without risking skipping possible optimal constraint values.

For the unbounded constraint, after each learning epoch, the following condition is evaluated to decide whether to continue or end the learning phase:

$$t_{Epoch(i)} \leqslant t_{Epoch(i-1)}/2$$

where:

$t_{Epoch(i)}$: average execution time of I/O tasks in learning epoch $i$

This condition assumes that since the constraint is doubled each new learning epoch (i.e., the number of concurrent tasks is halved), then the average task time in a learning epoch

should decrease by at least half compared to the average task time recorded in the previous learning epoch.

Comparing the bounded and unbounded auto constraints, the bounded auto constraint has the ability to achieve more fine-grained results. This is because it has a longer learning phase where it tries as high constraint as the *maximum* constraint value set by the user. Whereas the unbounded auto constraint follows a stricter learning approach. It follows the assumption that not getting the expected I/O task time improvement in a learning epoch will lead to a divergence path where no more improvement should be expected. Indeed, these different behaviours lead to different application performances that are discussed in the evaluation section of this chapter (Section 4.5).

More details on how to acquire and calculate the hyper parameters of the learning phase for the bounded constraint (i.e., *minimum*, *maximum* and *delta*) and the starting constraint value for the unbounded constraint are described in the implementation section (Section 4.4.3.2).

#### 4.3.3.2 Objective Function

After the learning phase ends, the information that has been collected about the average task time when launching a certain number of concurrent tasks (i.e., using a certain constraint) are applied to minimize the following objective function:

$$\forall c \in C: \quad \min \ \mathrm{T}(numTasks, \mathrm{c}) \tag{4.1}$$

where:

$C$: is the set of constraints used during the learning phase.

$T(numTasks, c)$: is the time estimation for executing the given number of I/O auto-constrained tasks using the constraint $c$. This function can defined as:

$$\mathrm{T}(numTasks, c) =$$
$$(numTasks/maxNumTasks_c) * \mathrm{t_c}$$

where:

$maxNumTasks_c$: is the maximum number of concurrent I/O tasks allowed to run using the given constraint $c$.

$t_c$: is the average I/O task time when using the given constraint $c$.

The objective of this function is to choose a constraint that minimizes the execution time of the auto-constrained tasks waiting to be scheduled. In this function, the number of execution groups in which the tasks will be executed is calculated by dividing the given number of tasks by the maximum number of concurrent tasks using a given constraint $c$. Then this number is multiplied by the average task time obtained during the learning phase. For instance, if $numTasks$ is 20 and the current constraint $c$ only allows maximum amount of 10 concurrent I/O tasks, then these 20 tasks will be executed in 2 groups, each group contains a maximum of 10 concurrent tasks. The number of task groups is then multiplied by the average I/O tasks time to get the time estimation of executing the $numTasks$ using the constraint $c$.

After evaluating the objective function with all the constraints used during the learning phase, the constraint that results in the least execution time is assigned to the task.

## 4.4 Implementation

This section presents the implementation details of the I/O awareness capabilities in the PyCOMPSs framework. Section 4.4.1 introduces the special handling of the I/O workload through the use of *I/O Tasks* and how can their execution be overlapped with the execution of compute tasks. Section 4.4.2 describes how task constraints are used to control I/O tasks scheduling to minimize I/O congestion. Finally, Section 4.4.3 presents the details of the automatic inference mechanism of the storage bandwidth constraint.

### 4.4.1 I/O Tasks

Following task declaration conventions of the PyCOMPSs programming model described in Section 3.1.1, a task is declared as I/O task by the means of the `@IO` decorator. Listing 4.1 shows the I/O tasks annotation in PyCOMPSs. Besides using the `@task` decorator to define a PyCOMPSs task, the `@IO` decorator is used to declare that this task should be registered and handled as an I/O task.

```
1 @IO()
2 @task()
3 def io_task(data):
4     # perform I/O operations on data
```

LISTING 4.1: I/O Task Annotation

Figure 4.4 shows how using I/O tasks can affect the execution. In the main code snippet, a loop launches three tasks:

- a `generate_block` task which returns a block of a certain size.
- a `checkpoint` task which writes the block to the disk.
- a `scale` task which does some computation on the block, the output of this task is stored in the `results` list.

Notice that both the `checkpoint` and `scale` tasks are dependent on the `generate_block` task, however, they do not have dependencies between each other. Therefore, their execution can overlap.

As shown in Figure 4.4, the `checkpoint` task can be handled in two different ways during the execution of the application depending on how it is defined in the code: On the one hand, it can be defined as a normal **compute task** by only using the `@task` decorator. Consequently, when there are no available computing resources, the execution of the `scale` tasks will be delayed until the `checkpoint` tasks finish execution. On the other hand, the `checkpoint` task can be defined as an **I/O task** by using the `@IO` decorator. This way, the COMPSs runtime handles the `checkpoint` tasks as I/O tasks, hence, the `scale` compute tasks are launched and the execution of both tasks is overlapped.

In order to enable the overlapping execution between dependency free I/O tasks and compute tasks as illustrated in Figure 4.4, modifications were made to the master and worker components of the COMPSs runtime. In the master component, by default, the COMPSs runtime assigns one CPU to every task. Consequently, the runtime scheduler will not launch any new tasks for execution unless there is enough CPUs to execute them. However, unlike regular compute tasks, when the COMPSs runtime receives an I/O task registration request, it sets its computing requirements to zero. Consequently, incoming I/O tasks with no dependencies will be scheduled immediately even if all the CPUs in the infrastructure are consumed by compute tasks.

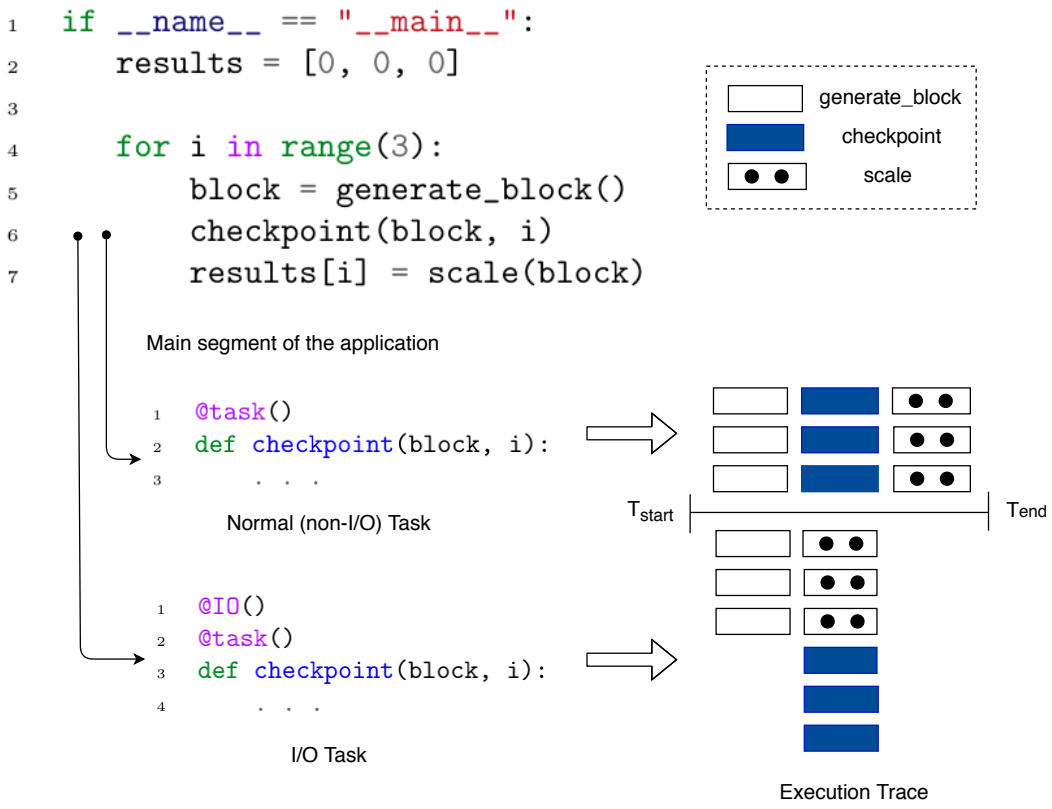



FIGURE 4.4: I/O Tasks Overlap With Compute Tasks

In case of using a shared working directory between tasks, it will be used to store tasks outputs in a serialized format. Therefore, no node-to-node data transfer is required and, as a consequence, I/O tasks are scheduled to the first candidate node (because I/O tasks always require 0 CPUs). However, if the working directory is not shared, then I/O tasks will be scheduled taking into consideration data locality.

Indeed, for the case when a shared working directory is used, alternative design approaches can be adopted to distribute the tasks to candidate workers (e.g., round-robin). However, our proposed behavior of scheduling tasks based on their requirements, specifically computing units and storage bandwidth, provides the most general approach and least intrusive implementation. In addition to that, our focus in this part of the thesis is to control the scheduling of I/O tasks based on their optimal bandwidth requirements; that minimizes I/O congestion and improves total application performance.

As for the worker component of the COMPSs runtime, it needs to support CPU oversubscription to enable the execution of I/O tasks side by side with compute tasks on the same CPU. Therefore, we extended the worker component architecture (described in Section 3.1.2) by adding an additional execution platform, called *I/O Execution Platform*, dedicated to handling I/O tasks execution. The other execution platform, called *Compute Execution Platform*, is dedicated to handling compute tasks execution.

Figure 4.5 illustrates a high-level overview of the architecture of the I/O aware worker component. Similar to the compute execution platform, the I/O execution platform handles the execution of I/O tasks by managing a number of executor threads. These executor threads are created at the launch time of the application and their number can be set in the PyCOMPSs launch command. By default, the I/O execution platform launches only one executor thread. However, the number of executor threads in the I/O execution platform can be set at application launch time by using the `io_executors` command-line argument

of the PyCOMPSs launch command.

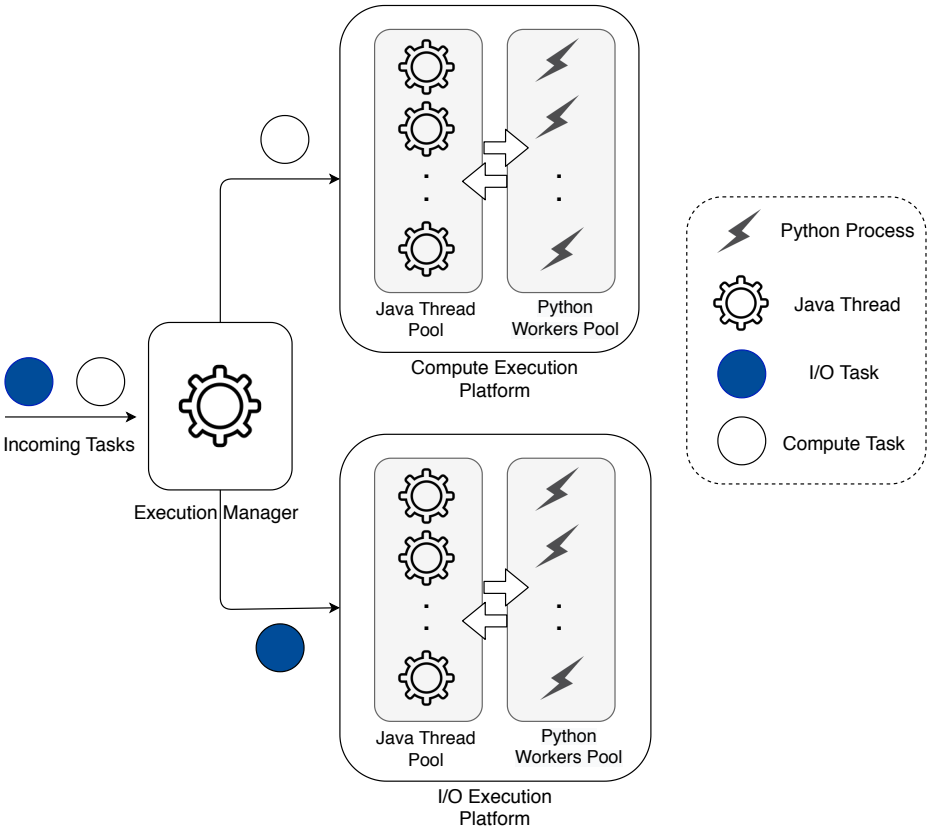



FIGURE 4.5: I/O Aware PyCOMPSs Worker; I/O Execution Platforms Handles The Execution Of I/O Tasks

It should be noted that at any given moment during application execution, the maximum number of tasks to be executed is equal to the number of threads in the execution platform. If there are more tasks to be executed on a worker than available I/O executors, then their execution will be blocked until any of the busy executors becomes available.

### 4.4.2 Static Storage Bandwidth Constraints

In order to constrain I/O scheduling to avoid I/O congestion, we extended the PyCOMPSs `@constraint` decorator (described in Section 3.1.1) to support a storage bandwidth constraint. During application execution, the COMPSs runtime keeps track of the state of the available I/O bandwidth of the system by updating its value based on the storage bandwidth requirement of each task.

Listing 4.2 shows a sample I/O task with the storage bandwidth constraint. Using the `storageBW` argument of the `@constraint` decorator, users can set an estimated bandwidth for I/O tasks. During the scheduling of the `constrained_write` task, the PyCOMPSs scheduler will use its storage bandwidth constraint value to determine at which point of application execution and to which node this task should be scheduled. If the storage bandwidth constraint of the `constrained_write` task is not satisfiable, then its execution will be blocked until its requirement becomes available.

The storage bandwidth constraint presented in Listing 4.2 is a *Static* constraint. This means that the value of the constraint is set before launching the application. Moreover, the value of this constraint does not change during the execution of the application.

```
@constraint(storageBW = 20)
@IO()
@task()
def constrained_write(data):
    ...
```

LISTING 4.2: Constrained I/O Task Using Storage Bandwidth Constraint

Static constraints can be used to optimize applications execution by enforcing execution constraints while offering programming simplicity. Such constraints are useful when execution time information are already known by the users from possibly previous runs. For instance, I/O workloads sizes, their optimal storage constraints, etc. When such information are not available at application design and coding time, then it is better to rely on the I/O awareness capabilities by using the auto-tunable constraints to identify these information and use them to manage the execution in order to improve the total performance.

It should be noted that the COMPSs runtime expects a *Resources Description File* to be provided at application launch time. This file is a XML format file that contains information about the available capabilities of the underlying infrastructure. It contains a list of all nodes participating in the execution and their properties. For instance, computing units of each node, memory size, storage capacity, etc. These system values are used at application execution time by the COMPSs runtime to enforce execution constraints.

In order to allow the COMPSs scheduler to reason about the available storage bandwidth in the system, we extended the resources description file to enable users to specify the maximum I/O bandwidth of the storage devices. This value will be used by the COMPSs runtime during applications execution to make better scheduling decisions. Listing 4.3 shows a part of a sample resources description file that specifies the PFS bandwidth for a computing node. It should be noted that the bandwidth units in the resources description file and the constraint value in the application code have to be the same. The COMPSs system does not require the values to be of any specific unit.

```
<ResourcesList>
    <ComputeNode Name="worker">
        ...
        <Storage Name="GPFS">
            <Type>PFS</Type>
            <Bandwidth>980</Bandwidth>
            ...
        </Storage>
        ...
    </ComputeNode>
</ResourcesList>
```

LISTING 4.3: Sample Resources Description File Of COMPSs

### 4.4.3 Auto-tunable Storage Bandwidth Constraints

This section describes how auto-tunable constraints can be specified programmatically in the PyCOMPSs framework, then it presents more details about the implementation of the learning phase and the objective function. For brevity, we will refer to *Auto-tunable Storage Bandwidth Constraints* as *Auto Constraints* in the rest of this chapter.

#### 4.4.3.1 Auto Constraints Syntax

As previously discussed in Section 4.3.3, we enabled two different types of auto constraints: *Bounded* and *Unbounded*. On the one hand, users can set bounded auto constraint as *auto(min, max, delta)*; where *min* represents the minimum starting constraint, *max* represents the maximum possible constraint and *delta* represents the value by which the runtime advances the constraint value from *min* to *max*. On the other hand, users can specify an unbounded auto constraint by setting the value of the storage bandwidth constraint to *auto*.

Listing 4.4 shows an example of bounded auto constraint. To set a bounded auto constraint, users have to estimate the hyper-parameters that will guide the learning phase (i.e. min, max and delta). Whereas Listing 4.5 shows an unbounded auto constraint, in which the runtime will estimate the min, max and delta hyper-parameters.

```
1 @constraint(storageBW = "auto(10,50,4)")
2 @IO()
3 @task()
4 def constrained_io_task(data):
5     ...
```

LISTING 4.4: Bounded Automatic Constraint With Syntax *auto(min, max, delta)*

```
1 @constraint(storageBW = "auto")
2 @IO()
3 @task()
4 def constrained_io_task(data):
5     ...
```

LISTING 4.5: Unbounded Automatic Constraint

#### 4.4.3.2 The Learning Phase

The COMPSs runtime automatically estimates the auto-constraints by carrying out the two-steps mechanism discussed in Section 4.3.3. After the completion of the learning phase, the runtime will have an *Auto Constraint Registry* for each auto-constrained task. This auto constraint registry contains pairs of: **constraint value** and the **average I/O task time** when using this constraint is used (*Constraint* $\rightarrow$ *Avg I/O Task Time*). Figure 4.6 illustrates the progress of a learning phase from the master and worker components of PyCOMPSs.

Looking closely at Figure 4.6, one can observe the progress of the learning phase for a given auto constraint. The figure shows two learning epochs, each learning epoch has three key moments:

- First, the learning epoch starts with assigning the storage bandwidth constraint ($C_1$ in the first learning epoch and $C_2$ in the second learning epoch) to all auto-constrained tasks. During each learning epoch, the master component schedules and launches tasks as long as the assigned task constraint does not bandwidth constraint of the system.
- Second, when the running I/O tasks that are in a learning phase finish executions successfully, the average time per I/O task is calculated and set in the auto-registry.

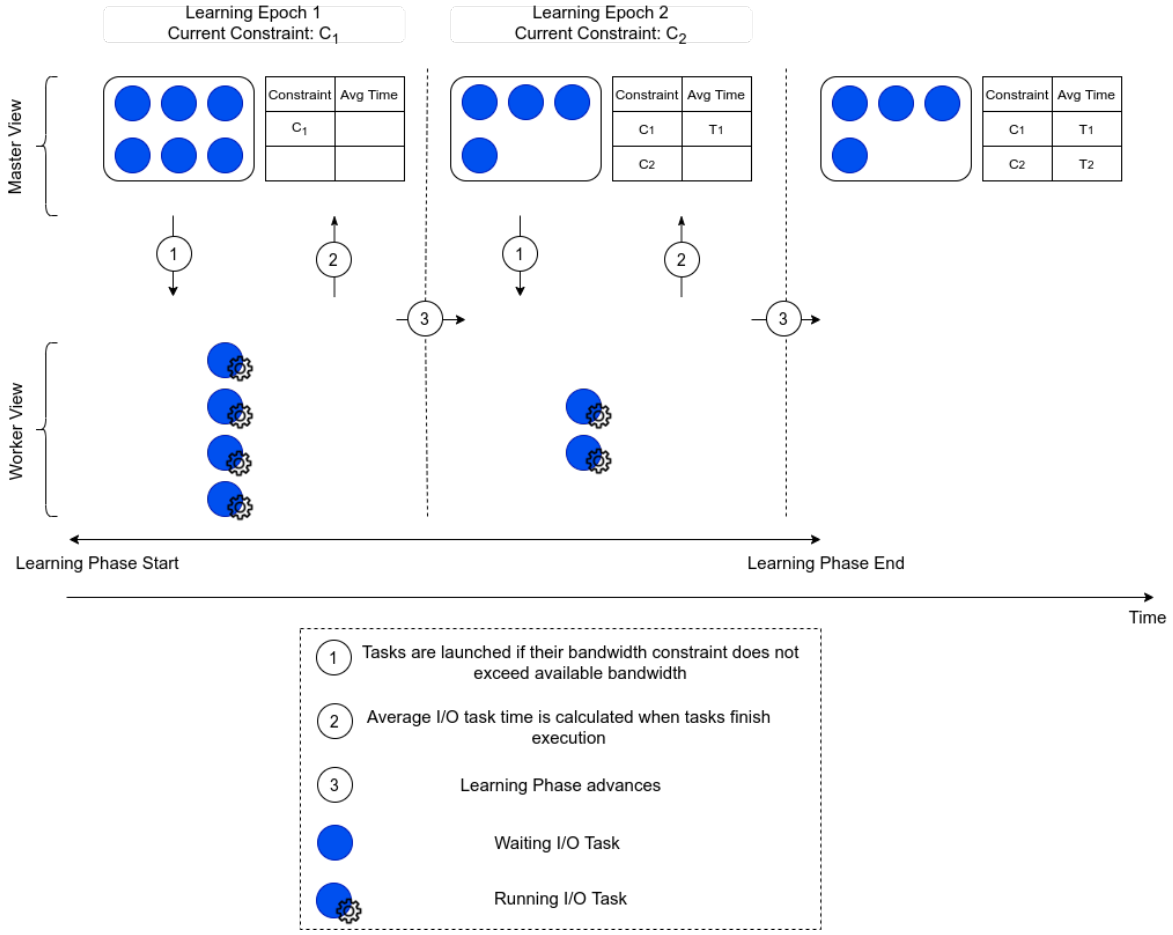



FIGURE 4.6: Learning Phase Progress

- Third, the master component decides whether to continue or end the learning phase depending on the criteria that was described in Section 4.3.3.1. In the case of continued learning phase, the scheduler sets the new constraint value and repeat steps 1 and 2 in Figure 4.6. However, in case the learning phase ends, the scheduler will use the auto constraint registry that was set during the learning phase to try to minimize the objective function described in Section 4.3.3.2 for the I/O auto-constrained tasks ready to be scheduled.

It should be noted that defining an auto constraint for a certain task will not affect how the COMPSs runtime handles the other tasks in the application. In addition to that, it is possible to have different auto-constrained tasks in the same application. The COMPSs runtime will run a separate learning phase for each auto-constrained task and will set a constraint suitable for the workload of each task. We assume that an I/O task will always produce the same I/O workload during the application lifetime.

In the case of an unbounded automatic constraint, the COMPSs runtime calculates the value of the starting constraint by dividing the maximum I/O bandwidth of the storage device by the number of I/O executors in each worker node. The number of I/O executors is a convenient choice for calculating the starting constraint because it represents the maximum number of I/O tasks that can run concurrently during any time of application execution.

In order to guarantee the integrity of the learning phase, the scheduler dedicates a worker node for each auto-constrained task in an active learning phase. These nodes are called *Active Learning Nodes*. Once a node is marked as an active learning node for a specific auto-constrained task, the scheduler avoids scheduling any other I/O tasks or auto-constrained

I/O tasks to that node. Therefore, it is guaranteed that the learning phase of an auto-constrained task will not be interfered by the other I/O tasks. As soon as the learning phase of an auto-constrained task ends, the scheduler un-marks the associated active learning node and use it for scheduling as normal. It should be noted that the compute tasks are scheduled normally on all available nodes because they do not consume any storage bandwidth resources.

We study the performance of both types of automatic constraint and the effect of changing their hyper-parameters (the values of *max, min, delta* in the bounded constraint and the number of I/O executors per worker node in the unbounded constraint) in the evaluation section (Section 4.5).

#### 4.4.3.3 The Objective Function

After finishing the learning phase for a certain auto-constrained task, the COMPSs runtime applies the *auto constraint registry* to objective function 4.1 to choose a constraint that will result in the execution of the pending auto-constrained tasks in the minimum possible time.

Some cases are considered when evaluating objective function 4.1 for a given number of scheduling-ready auto-constrained tasks:

- If the number of tasks is not divisible by the maximum number of concurrent tasks, then the time for executing any remainder is estimated. Then, it is added to the original time estimate $T(numTasks, c)$.
- If there is a tie and several constraints result in the same execution time for a given number of ready tasks, then the highest constraint is used because it result in the minimum I/O congestion.

It should be noted that the objective function is re-evaluated and a new constraint is set -if necessary- every time new execution requests of an auto-constrained task arrive to the scheduler. This approach allows the continuous tuning of the constraint value throughout application execution depending on the number of I/O auto-constrained tasks.

## 4.5 Evaluation

This section shows the improvement that I/O aware PyCOMPSs can achieve in the total performance of applications with different workloads.

We start this section with describing the infrastructure of the MareNostrum 4 supercomputer and its storage architecture (Section 4.5.1). Followed by a brief discussion about the impact of launching I/O tasks with compute tasks on the same node (Section 4.5.2). Next, we present a brief description of the applications used in the evaluation, their I/O workload characteristics and their performance results (Section 4.5.3). Finally, the section ends with presenting the experiments results that show the impact of the hyper-parameters of the auto constraints on applications performance (Section 4.5.4).

### 4.5.1 Infrastructure

The MareNostrum 4 supercomputer [66] of the Barcelona Supercomputeing Center (BSC) is composed of 3,456 nodes. Each node has two Intel Xeon Platinum chips, each with 24 processors for a total of 48 cores per node. The MareNostrum 4 supercomputer contains two

types of nodes: low memory and high memory. The low memory nodes contain 92 GB main memory whereas the high memory nodes contain 370 GB of main memory.

Figure 4.7 shows a high-level overview of the MareNostrum 4 supercomputer storage infrastructure. All nodes have access to shared Hard Disk Drives (HDD) with total capacity of 14 PetaBytes mounted with the IBM General Parallel File System (GPFS). The GPFS servers are shared and accessible to all the users of the system. Each computing node in the MareNostrum 4 supercomputer has a local Solid State Drive (SSD) with a capacity of 200 GB and bandwidth of 470 MB/s and 450 MB/s for reading and writing respectively.

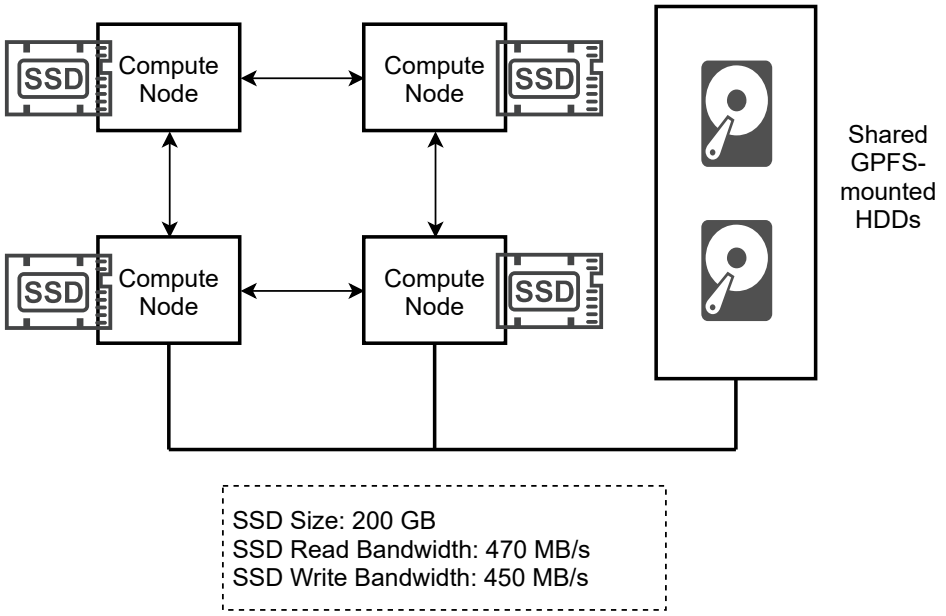



FIGURE 4.7: High-Level Overview Of The Storage Infrastructure On The MareNostrum 4 Supercomputer

In all the experiments, the GPFS is used to store the input data and final results -if any- of the applications. In addition to that, the GPFS is also used in our experiments as a shared working directory to store tasks logs and dependencies between tasks. Therefore, no node-to-node data transfer is required.

Node-local SSD disks are used as *Burst Buffers* to checkpoint the intermediate results of the applications. On the one hand, as they are used exclusively by the nodes running the experiments, they offer better performance than the globally shared HDD-backed GPFS. This is because the whole bandwidth of the SSDs are dedicated for our experiments and no interference occurs from the experiments of other MareNostrum 4 users. On the other hand, they offer a controlled environment in which performance benefit can be planned and expected. As previously discussed, using SSDs as a caching layer to absorb intensive I/O of applications has been discussed in previous I/O research.

### 4.5.2 I/O Tasks Impact on Compute Tasks

In order to measure the impact of oversubscribing the CPUs with I/O tasks on the performance of the compute tasks that already use these CPUs, we performed multiple experiments such that each experiment launches a fixed number of concurrent compute tasks and a variable number of concurrent I/O tasks. Each compute task performs a compute-intensive matrix multiplication algorithm, whereas each I/O task writes 100 MB of data to a file. In all experiments, the node is occupied with 48 concurrent compute tasks, which is

the maximum number of concurrent compute tasks that a node can host. However, in each experiment, a different number of concurrent I/O tasks is launched.

Each experiment was repeated 10 times and the average result is reported. Figure 4.8 depicts the average time of the compute and I/O tasks of the different experiments. It can be noted that I/O tasks have negligible effect on compute tasks or does not have any effect at all. Regardless of how many concurrent I/O tasks are being executed, the average time of compute tasks is not impacted. However, as the number of concurrent I/O tasks increases, the average time for an I/O task increases because of the increasing I/O congestion.

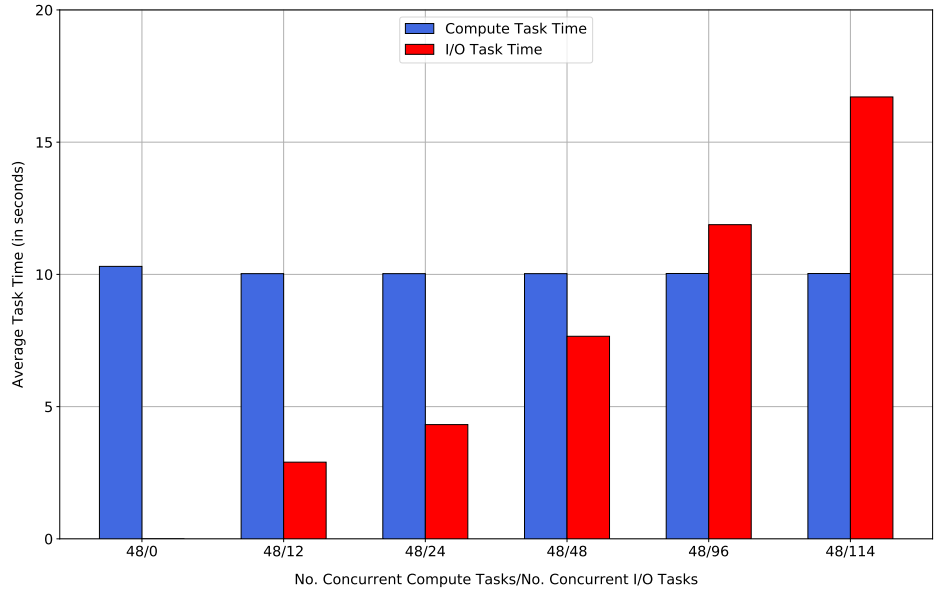



FIGURE 4.8: Average Time For Compute Tasks With Increasing Number of Concurrent I/O Tasks

### 4.5.3 Use Cases and Experiments

We implemented three different I/O intensive applications with PyCOMPSs. Each application exhibits a different I/O workload which allows for evaluating the impact of the I/O awareness capabilities in different scenarios. These three applications are:

- The *HMMER* application: an application that produces *Homogeneous I/O workload*; there is only one task that execute I/O in the application. In addition to that, for a given input size, the checkpointing task writes the same amount of I/O every time it is called during the lifetime of the application. This application is intended to show the impact of using I/O awareness capabilities on I/O throughput, I/O task time and total time. In addition to the behaviour of auto-tunable constraints with homogeneous I/O workloads.

- The *Variants Discovery Pipeline*: exhibits a *Heterogeneous I/O workload*, because it has more than one checkpointing task. Each checkpointing task writes different amount of data to the disk. This application is intended to show the behaviour of auto-tunable constraints with different I/O workloads.

- *Kmeans*: an iterative algorithm to test the effect of the number of available tasks on the total performance of the application when using auto-tunable constraints.

In all the experiments, the I/O non-aware PyCOMPSs implementation (i.e., no I/O tasks nor storage bandwidth constraints) is used as the baseline version. Moreover, for the *HMMER* and *Variants Discovery Pipeline*, we launched several runs of the I/O aware implementation. Each run has a different setting of the storage bandwidth constraint for the I/O tasks. These runs include:

- A non-constrained run where I/O tasks are used to execute I/O workload but no storage bandwidth constraints are used.
- Several runs with increasing values of static storage bandwidth constraint. The value of these constraints is MB/S.
- Two runs with the both types of the auto-tunable constraints. For each of the runs with an auto-tunable constraint, we show graphs of its learning phase progress during application execution.

It should be noted that for the *HMMER* application and the *Variants Discovery Pipeline*, reading I/O tasks have been used in order to read input data. However, they do not offer any performance benefit because they do not overlap with compute tasks.

In addition to that, in order to test the effectiveness of our proposals for solving the problem of I/O congestion, writing I/O tasks in all experiments has avoided using system buffers by flushing the data to storage devices. This is achieved by using the *fsync* call of the OS library of the Python programming language [82].

All the experiments were run on 12 high-memory MareNostrum 4 nodes plus one node dedicated as the master node. The master node runs the master component of the COMPSs runtime and manages the execution without taking part in any computation.

#### 4.5.3.1 HMMER Application

The *HMMER* Application is used for searching sequence databases for sequence homologous proteins or nucleotide sequences using a variant of Hidden Markov Models (HMM) called *profile-HMM*. It takes two inputs: a sequence database and a sequence file. Figure 4.9 depicts a sample PyCOMPSs skeleton dependency graph of the application.

Our implementation of the HMMER application first splits the sequence file and sequence database into multiple fragments. A *hmmpfam* task is called for each sequence and database fragment. The *hmmpfam* task calls the HMMER tool [27] on its sequence fragment and database fragment. Each *hmmpfam* task has as a *checkpointFrag* successor task that is responsible for checking the results of the HMMER tool. Later, the application calls a *gatherDB* task which gathers the results obtained of running a single sequence fragment against all database fragments. Finally, all the sequence fragments are gathered into one single file in the *gatherSeq* task.

For running the experiments of this application, we used as inputs a 64.5 GB HMM protein-families database (pfam) and a sequence file that contains 14,942,208 sequences with a total size of 3.2 GB. Databases and sequence files are available on the ftp servers of the European Bioinformatics Institute (EMBL-EBI) [30] which hosts up-to-date sequences, databases and software widely used by academics and life science researchers.

We set the number of database fragments and the sequence fragments to 48 each. This means that every run of the application will have 2,304 *hmmpfam* tasks followed by the same number of *checkpointFrag* tasks. Each *checkpointFrag* task writes 290 MB of data to a separate file on the node-local SSD disk. As each of the 12 worker nodes used in this experiment has 48 cores, then the maximum number of tasks that can run at the same time across the whole

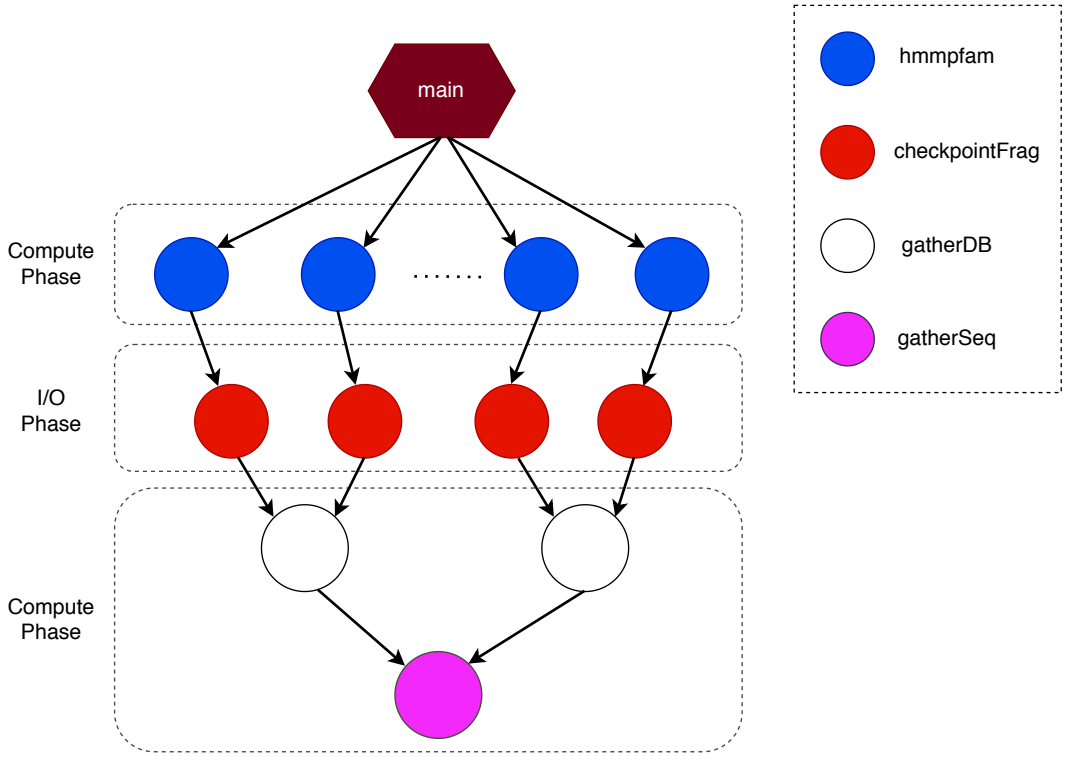



FIGURE 4.9: Task Skeleton Of The HMMER Application

system is 576 tasks. Consequently, the application will be executed in multiple compute-IO phases.

Figure 4.10 presents the performance results of the application. In Figure 4.10, the red bar represents the baseline run where non of the I/O capabilities (i.e., I/O tasks and storage bandwidth constraint) are used. Whereas the yellow bar represents a non-constrained run where only one capability of I/O aware PyCOMPSs is used: declaring the *checkpointFrag* as an I/O task but without using any storage bandwidth constraint for the I/O tasks. The blue bars represent runs with both I/O capabilities of PyCOMPSs, each run using a higher static storage bandwidth constraints. Finally, there are two bars: one shows the total time using the unbounded automatic storage bandwidth constraint and the other with a bounded auto storage bandwidth constraint of *auto(2,256,2)*. In order to show the negative impact of not controlling I/O task parallelism and I/O congestion, the Non-constrained experiment used 500 I/O executors. Using this number of I/O executors allows the execution of the maximum number of I/O tasks concurrently without causing any node failure for this application. On the other hand, the rest of the experiments used 225 I/O executors, which allows to host the concurrent execution of all I/O the tasks when using the least static storage constraint (i.e., 2).

As can be noted in Figure 4.10 using the I/O awareness capabilities of PyCOMPSs can achieve more than 38% performance improvement in the total time of the application compared to the baseline version. In the baseline run, the *checkpointFrag* tasks are treated as compute tasks, no overlap between I/O and computations occurs and only 48 *checkpointFrag* tasks can be executed at a time. However, using I/O tasks without setting any storage bandwidth constraint (as in the non-constrained run) results in a much worse total time than the baseline. Even though the execution of the *checkpointFrag* I/O tasks is overlapped with the execution of the *hmmpfam* compute tasks, the effect of the I/O congestion negatively affects the total time of the application.

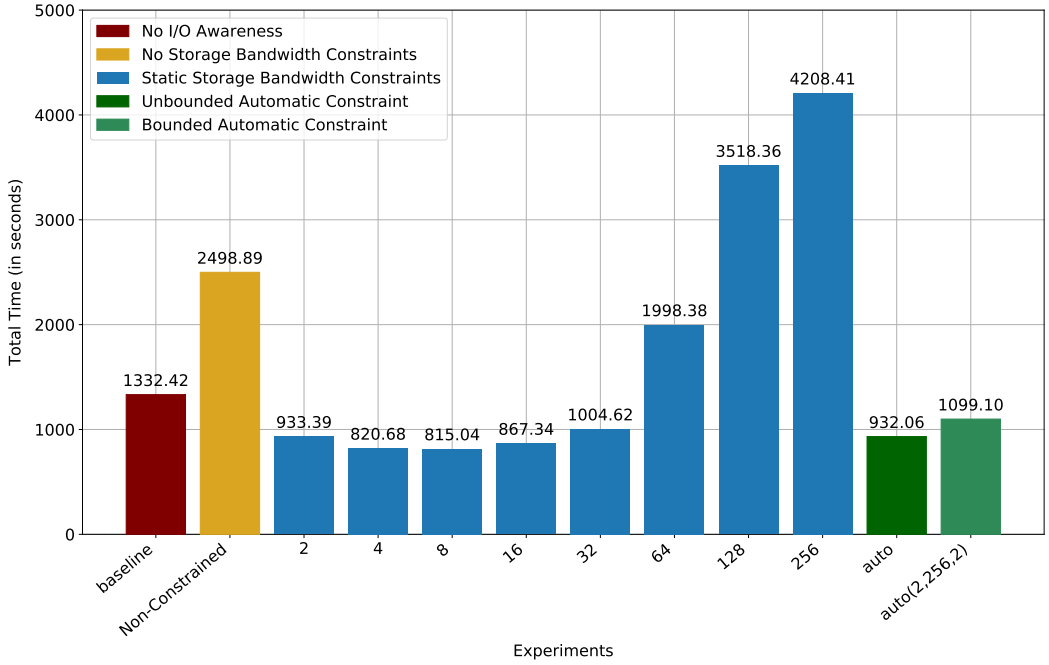



FIGURE 4.10: Experimental Results Of The HMMER Application

Nevertheless, continuing with Figure 4.10, as we start setting a storage bandwidth constraint for the I/O tasks, the total time of the application starts to decrease not only because I/O and compute tasks overlap execution but also I/O congestion is controlled. As the value of the storage bandwidth constraint increases, the total time of the application improves until a certain point where it starts to deteriorate again. Indeed, increasing the value of the constraint decreases the maximum number of concurrent I/O tasks. Even though executing less I/O tasks concurrently minimizes I/O congestion and improves I/O task time, this improvement in I/O task time does not compensate the decreased task parallelism. This is most apparent when using a storage bandwidth constraint of 256 where only one I/O task is allowed to run at a time. In this case, even though the whole I/O bandwidth is entirely dedicated for the currently running I/O task, the sequential execution of I/O tasks drastically harms the total time of the application.

Furthermore, it can be observed in Figure 4.10 that both runs with the auto storage bandwidth constraint achieve total time improvements compared to the baseline experiment. However, it can be noted that the total time when using a bounded auto constraint is worse than the total time when using the unbounded auto constraint.

Figure 4.11 presents the achieved I/O throughput for I/O tasks. The non-constrained experiment has the worst I/O throughput due to the increased and uncontrolled I/O congestion. As we start to control the number of I/O tasks running concurrently by using bandwidth constraints, I/O congestion decreases and the achieved I/O throughput begins to increase until it reaches the peak value when a constraint of 8 is used (which is the same value at which the application has the best total time in Figure 4.12(a)). As the constraint value keeps increasing, the number of I/O tasks running concurrently decreases, therefore I/O throughput slightly decreases because the local-SSDs of the nodes are not fully utilized. Furthermore, it can be observed that both of the auto constraints achieve peak I/O throughput similar to using the optimal constraint.

In order to understand the auto constraints behaviour, we refer to Figure 4.12 that shows the progress of the learning phases of both auto constraints during application's execution time.

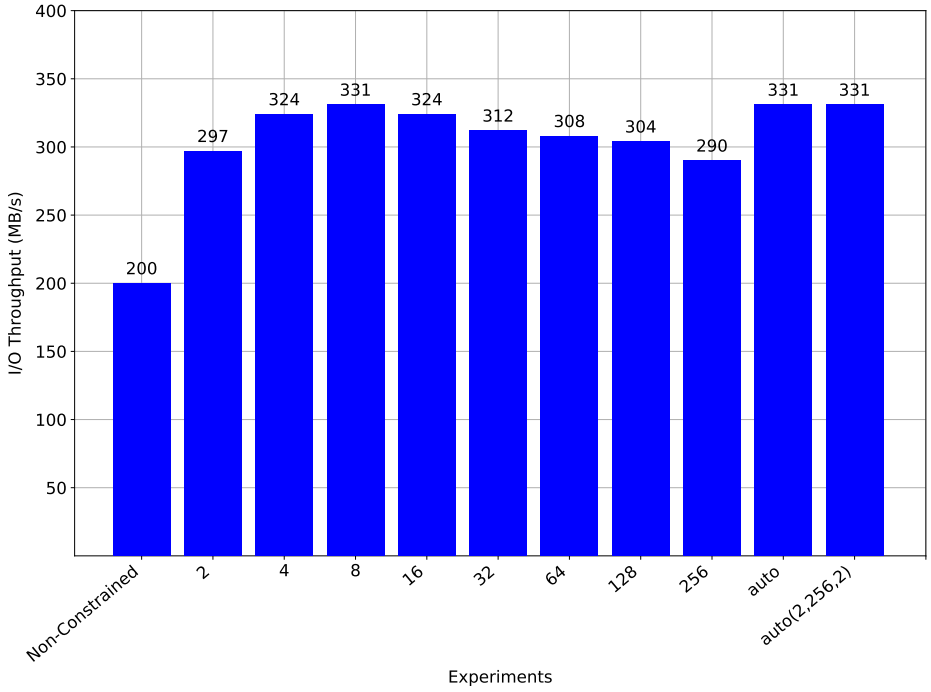
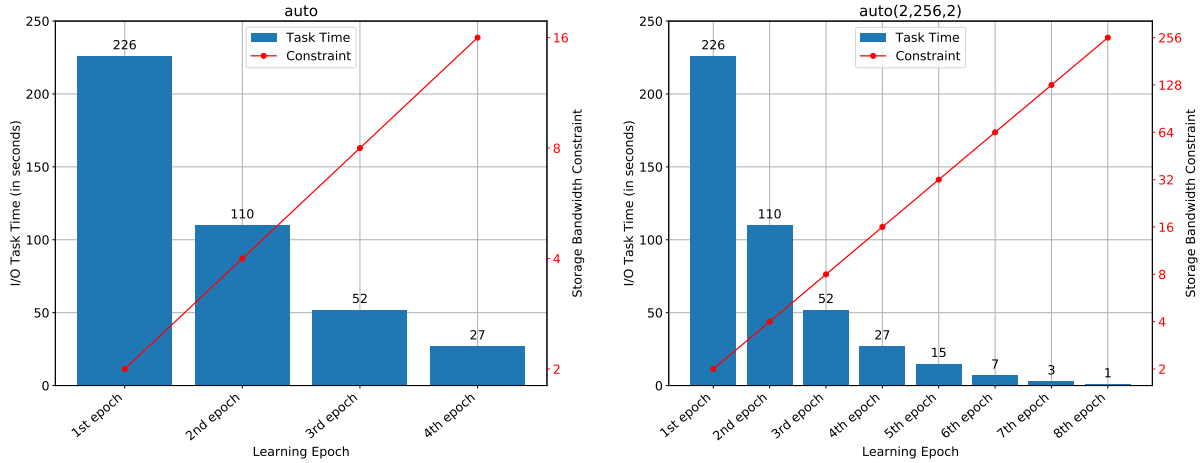



FIGURE 4.11: Achieved I/O Throughput In The HMMER Application




(a) Unbounded Auto-tunable Constraint

(b) Bounded Auto-tunable Constraint

FIGURE 4.12: Auto-tunable Constraints Learning Phase Progress In The HMMER Application

Figure 4.12(a) depicts the progress of the learning phase when using an unbounded auto constraint. First, the runtime sets the initial constraint to 2, because this run used 225 I/O executors on each worker node to handle the execution of I/O tasks. After the end of the first learning epoch the runtime registers the average I/O task time during this epoch and doubles the value of the constraint to progress the learning phase. When the second epoch ends, in order to decide whether to continue or abort the learning phase, the runtime checks whether the I/O task time in the second learning epoch is at least half of the I/O task time in the previous epoch. Since this condition is met, the runtime registers the average I/O task time during the second epoch. Next, the runtime progresses the learning phase until it stops after the fourth epoch because the continuation condition is violated; the task time in

the fourth epoch is not at least half of the task time in the third epoch. Upon the termination of the learning phase, the runtime applies the auto constraint registry, that now contains the average I/O task time in three learning epochs, to objective function 4.1. Finally, the runtime sets the constraint to 8 which is the value that minimizes the execution time of the auto constrained *checkpointFrag* I/O tasks ready for scheduling.

Similarly, the learning phase progress of the bounded auto constraint *auto(2,256,2)* is depicted in Figure 4.12(b). Using this type of auto constraints, the runtime starts the first epoch with the minimum constraint value provided in the `@constraint` decorator in the application code: 2. After the end of the first epoch, the runtime registers the average I/O task time and progress the learning phase by multiplying the current constraint value by the value of delta provided in the application code: 2. Therefore, the second learning epoch has a constraint of 4. The learning phase keeps progressing in this manner until the current value of the constraint becomes bigger than the maximum value provided by the user: 256. Therefore, the learning phase stops after the eighth learning epoch. Now that the runtime has filled the auto constraint registry, it uses the auto constraint registry to minimize the execution which in this case is 8.

As the bounded auto constraint spends more time in the learning phase, its application total time is worse than the unbounded auto constraint that follows a stricter and shorter learning process. In addition to that, the bounded auto constraint has a more fine-grained *auto constraint registry*, since it tries higher constraint values. During most of the execution time, the final constraint value of the bounded auto constraint and the unbounded auto constraint is the same (in this application, this constraint value is 8). However, for a certain number of scheduling-ready *checkpointFrag* I/O tasks, the fine-grained auto constraint registry of the bounded auto constraint may result in a different constraint value than the unbounded auto constraint for smaller number of tasks. However, the value of the constraint is re-adjusted and the minimization function is re-evaluated every time a new *checkpointFrag* I/O task arrives to the scheduler and the number of scheduling-ready *checkpointFrag* I/O tasks increases. Therefore, if the runtime sets a high constraint value for a certain number of scheduling-ready tasks, this constraint value will be adjusted in the next scheduling iteration.

#### 4.5.3.2 Variants Discovery Pipeline

The *Variants Discovery Pipeline* is popular pipeline of tools in the field of bioinformatics and computational genomics. The purpose of this pipeline is to discover genomic variants in sequence data.

Figure 4.13 illustrates the PyCOMPSs tasks dependency graph of this pipeline. Since the pipeline performs a lot of operations, we split it into three phases for visualization purposes: Data Preprocessing, Data Mapping and Variant Calling. We defined a checkpointing task for each major step in the pipeline to checkpoint the results of the pipeline so far. We define a major step in the pipeline as the last step in each processing phase (e.g., *convertSAMtoFASTQ* at the end of *Data Processing* phase) or any step that is not easily recomputed (e.g., after *bwa_map* in the *Data Mapping* phase). Decomposing and checkpointing the Variants Discovery Pipeline in the aforementioned manner is recommended by the widely followed *Broad Institute Best Practices Guide* [12]. This design of the pipeline has many advantages other than failure recovery; for example, the intermediate checkpointed data can be used for post-mortem analysis such as visualization, or to run separate pipelines on these data or simply to be stored in genomic databases for using them as references in future experiments.

When two compute tasks produce output of the same size, they call the same checkpointing task. For instance, the *bwa_map* that maps its input to the reference genome and

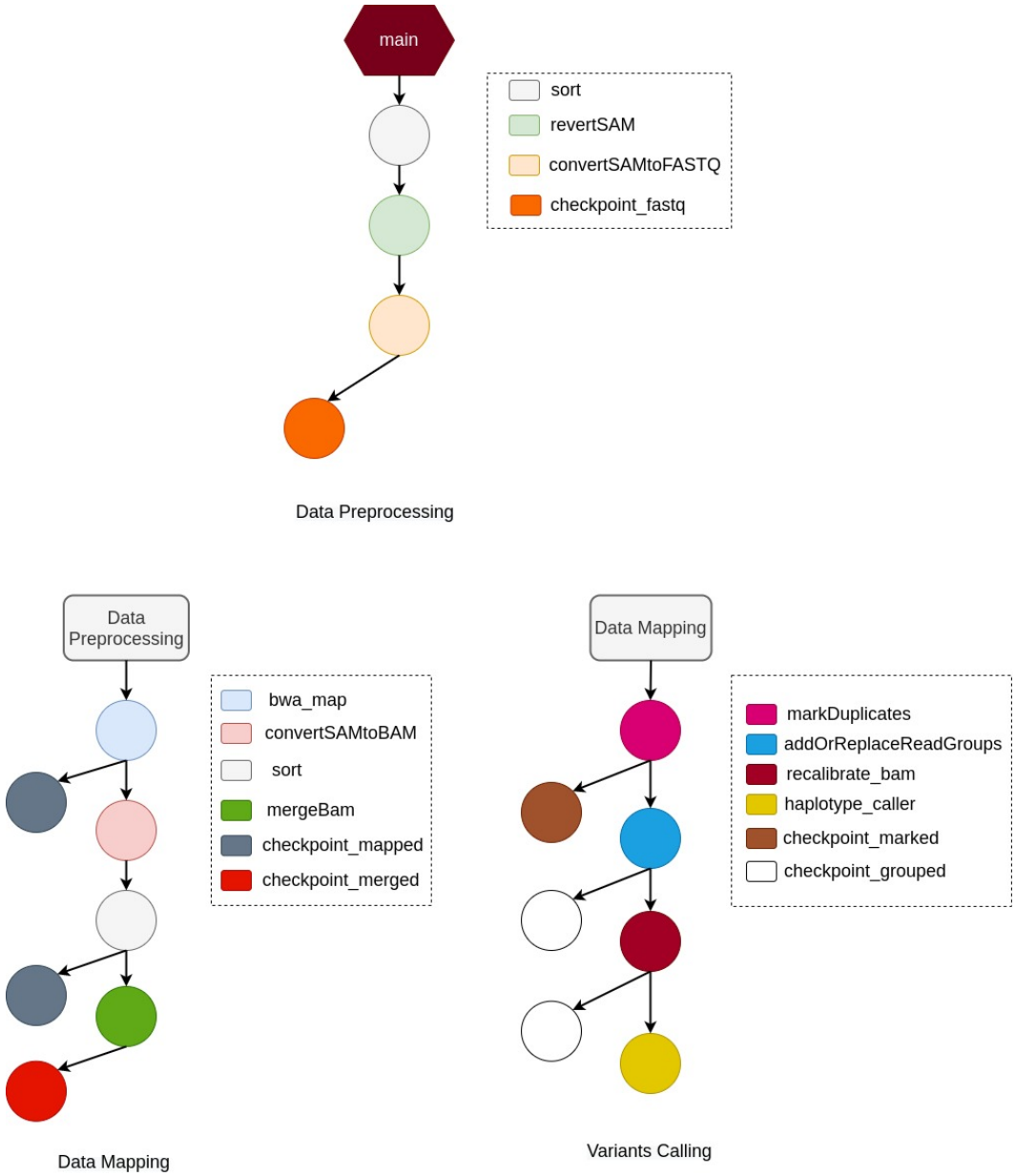



FIGURE 4.13: Task Skeleton Of The Variants Discovery Pipeline

the *sort* task that sorts the mapped sequence, use the *checkpoint_mapped* task because they produce almost the same size of output data.

We launched the experiments for this application with 1,728 sample sequence files. Each sequence file has a size of 72 MB in a compressed gzipped format. For each input, the application launches a separate pipeline to discover its variants. The input sequence files and the meta-data are publicly available on the GATK Broad Institute servers [37] which is a well-known resource for providing human sequencing data (e.g., sample sequences, genome references, variants databases, etc.). Table 4.1 lists the checkpointing tasks in the application and the data sizes that each task writes.

Figure 4.14 presents the performance results of different runs of the application. Using both capabilities of I/O awareness (i.e., I/O tasks and storage bandwidth constraints) can achieve up to 43% performance improvement in the total time of the application compared to the baseline run. The non-constrained run has the worst total time because of the I/O congestion that happens as all the I/O tasks concurrently access the node-local SSD disk of the

TABLE 4.1: Amount Of Data Written By Checkpointing Tasks

| Task | Amount of Data Produced |
|---|---|
| checkpoint_fastq | 162 MB |
| checkpoint_mapped | 290 MB |
| checkpoint_merged | 330 MB |
| checkpoint_marked | 596 MB |
| checkpoint_grouped | 615 MB |

same worker node. In this run, a maximum of 325 I/O tasks are allowed to run concurrently because 325 I/O executors are used. After several experimental runs, it was observed that this number of I/O executors allows the execution of the maximum number of I/O tasks to show the impact of I/O congestion without causing any node failures. Using the storage bandwidth constraint immediately mitigates the I/O congestion problem and the total time starts to improve. However, as the static storage bandwidth constraint increases, the total time starts to degrade due to the decreased level of task parallelism. Moreover, both auto constraints runs achieve performance improvement comparable to the optimal total time when using a static constraint of 4 with some overhead incurred due to the time spent in the learning phase.

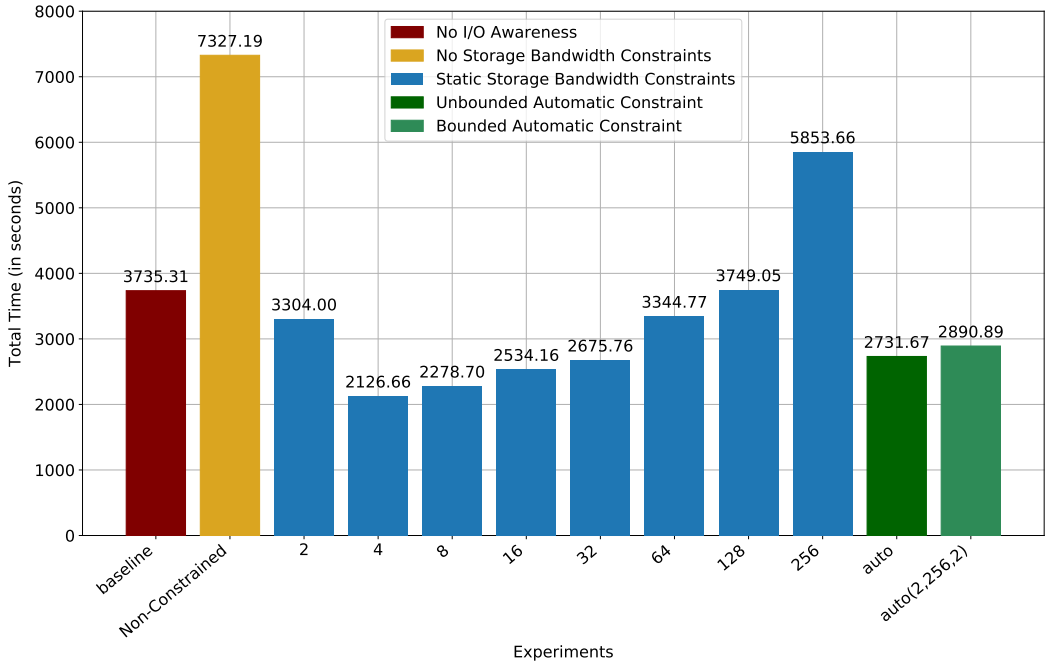



FIGURE 4.14: Experiment Results Of Variant Discovery Pipeline

It should be noted that in the static constraint runs, the same static constraint is used for all the checkpointing tasks mentioned in Table 4.1. However, in the auto constraint runs, the final value of the auto constraint is different for each checkpointing task. Each of these checkpointing tasks has its own learning phase, and the objective function is evaluated for each of them separately. Figures 4.15 to 4.19 show the learning phase progress for each checkpointing task with the unbounded and bounded auto constraint.

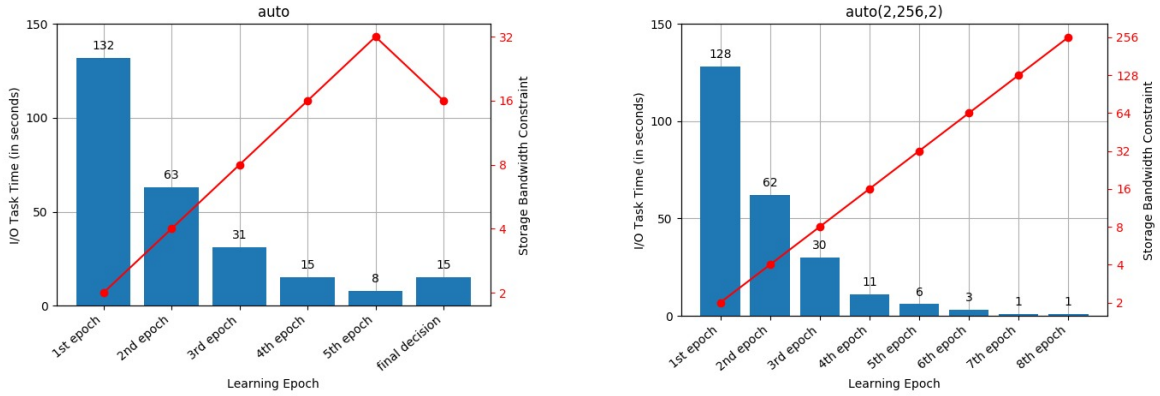



FIGURE 4.15: Learning Phase Of *checkpoint_fastq* Task

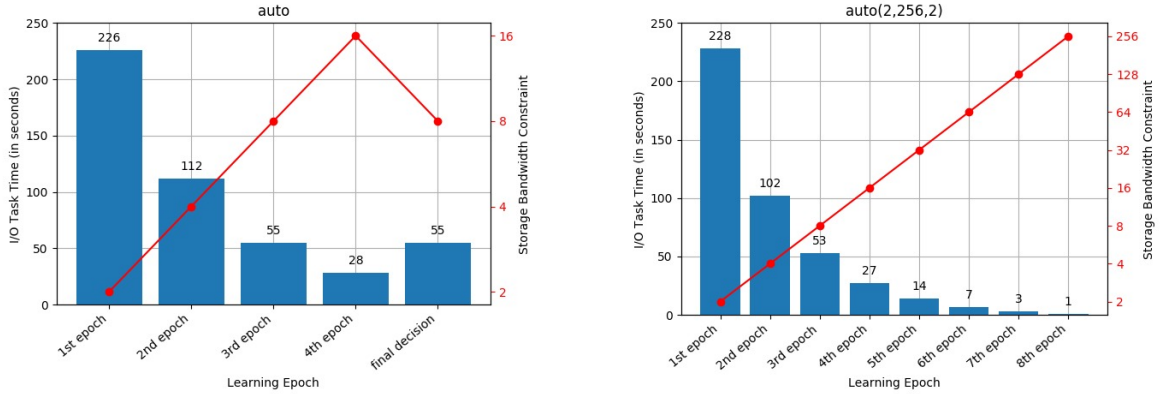



FIGURE 4.16: Learning Phase Of *checkpoint_mapped* Task

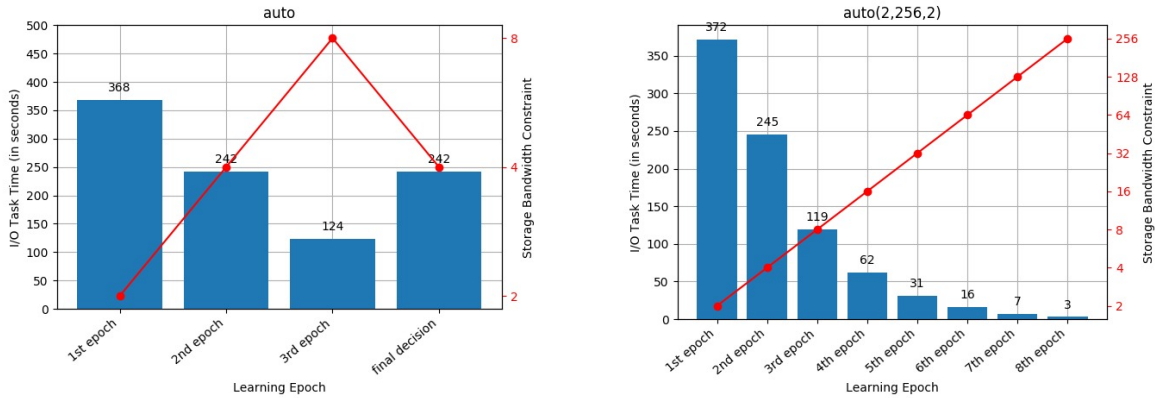



FIGURE 4.17: Learning Phase Of *checkpoint_marked* Task

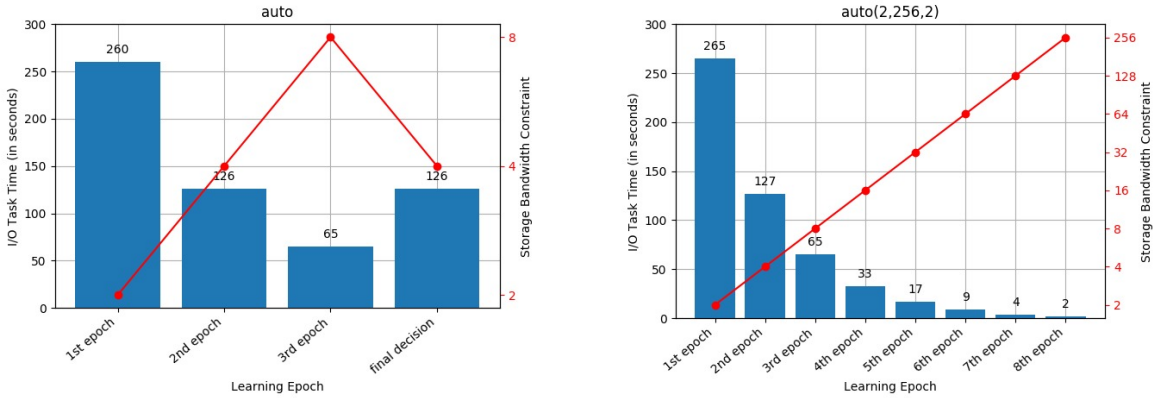



FIGURE 4.18: Learning Phase Of *checkpoint_merged* Task

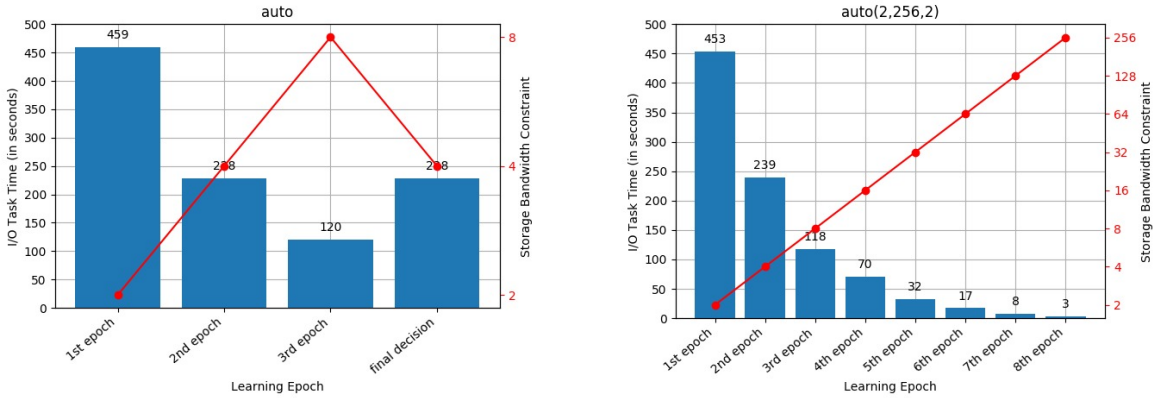



FIGURE 4.19: Learning Phase Of *checkpoint_grouped* Task

It can be noticed from Figures 4.15 to 4.19 that each checkpointing task goes through its own learning phase. The runtime uses the constraint value that will optimize the execution of the I/O workload of this task independently of the other checkpointing tasks. Table 4.2 lists the final auto constraints that were used for each checkpointing I/O task.

TABLE 4.2: Constraint Values For Checkpointing Tasks

| Task | Constraint |
|---|---|
| checkpoint_fastq | 16 |
| checkpoint_mapped | 8 |
| checkpoint_merged | 4 |
| checkpoint_marked | 4 |
| checkpoint_grouped | 4 |

Although different constraints are used for each checkpointing I/O task, using auto constraints can achieve a total time close to the optimal total time achieved when a static constraint of 4 is used. This is possible because the runtime sets the auto constraint for the *checkpoint_merged* and *checkpoint_grouped* tasks to 4, which is the constraint that leads to the best total time. Therefore, unlike static constraints where a certain constraint value maybe optimal for one checkpointing task but not optimal for the others, auto constraints will use the constraint that achieve best possible I/O task time and total execution time.

Note that the constraint choice of both: *checkpoint_merged* and *checkpoint_grouped* has bigger impact on the total time than the other checkpointing tasks because these two tasks are executed at the end of the pipeline where there are no compute tasks that can hide the effect of using a bad constraint.

#### 4.5.3.3 Kmeans Application

In order to evaluate the impact of the number of I/O tasks on the time of the learning phase of the auto constraints and consequently the application total time, we run multiple experiments with the Kmeans application as an example of iterative applications. The *Kmeans Algorithm* is a well-known machine learning algorithm that is widely used for different purposes such as cluster analysis in data mining fields. The Kmeans application follows an iterative process where it groups a set of multidimensional points into a number of clusters following a nearest mean distance rule. By changing the number of iterations, we change the number of tasks that will be executed in the application. Increasing the number of iterations will generate more tasks to be executed.

The dataset considered for evaluating the Kmeans application is composed of 10,000,000 points of 1000 dimensions, 3,000 centers and 500 fragments. Each of the checkpointing tasks writes 109 MB to the SSD storage disks.

Figure 4.20 shows the task dependency skeleton graph of the Kmeans application implemented with PyCOMPSs. A *generate_fragment* generates fragments of random data given a specific seed. In each iteration, the *partial_sum* task is called on each fragment to calculate the distance of each point to all cluster centers. The new centers are checkpointed using the *checkpointCenters* task.

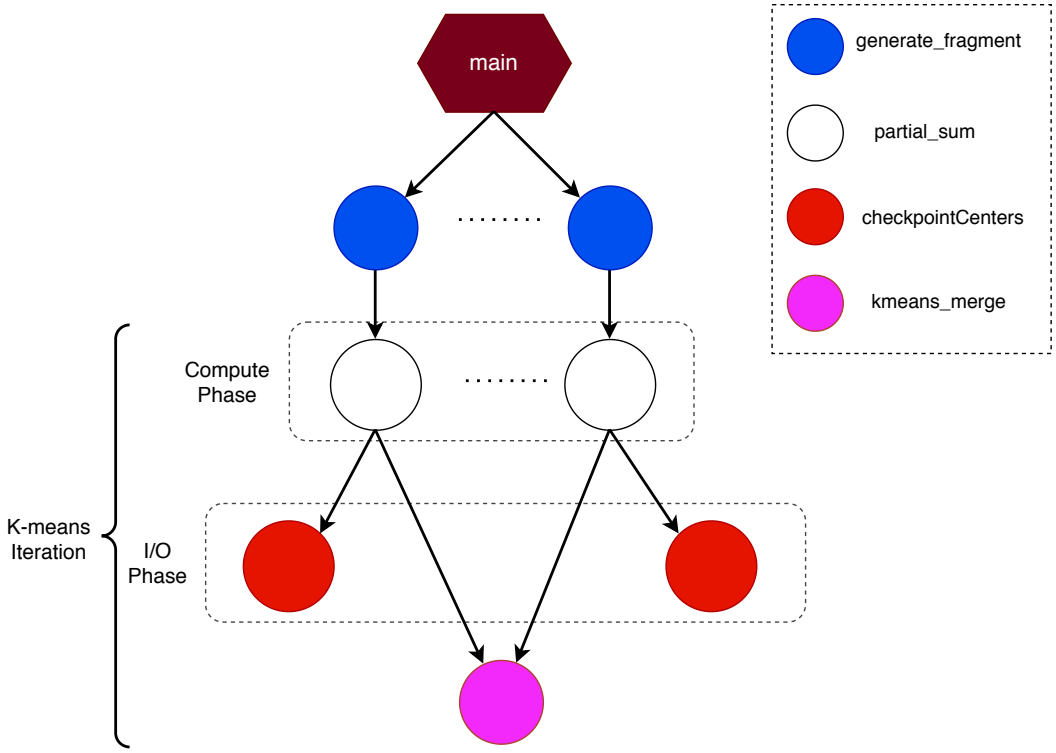



FIGURE 4.20: Task Skeleton Of The Kmeans Application

Figure 4.21 shows the experiments results of the Kmeans applications with different number of iterations. It can be noticed that with a single iteration the results of both of

the auto constraints experiments do not show a performance improvement. This can be explained due to the small number of auto-constrained checkpointing tasks in the application. Out of 500 checkpointing tasks to be executed, the unbounded auto constraint uses 435 checkpointing tasks for learning, whereas the bounded auto constraint uses 446 checkpointing tasks. Therefore, after the learning phase ends, a very small number of checkpointing tasks remains to take advantage of the results of the learning phase.

In order to validate this conclusion, we repeated the experiments but this time with higher numbers of iterations (3 and 6). In the case of 3 iterations, the total number of checkpointing task available for execution increases to 1500 tasks. Consequently, the number of auto-constrained tasks available for execution increases and we start getting performance improvement for both of the auto constraints. This gain increases with increasing the number of iterations because the application can make up the time spent in the learning phase and more I/O tasks overlap with the execution of compute tasks.

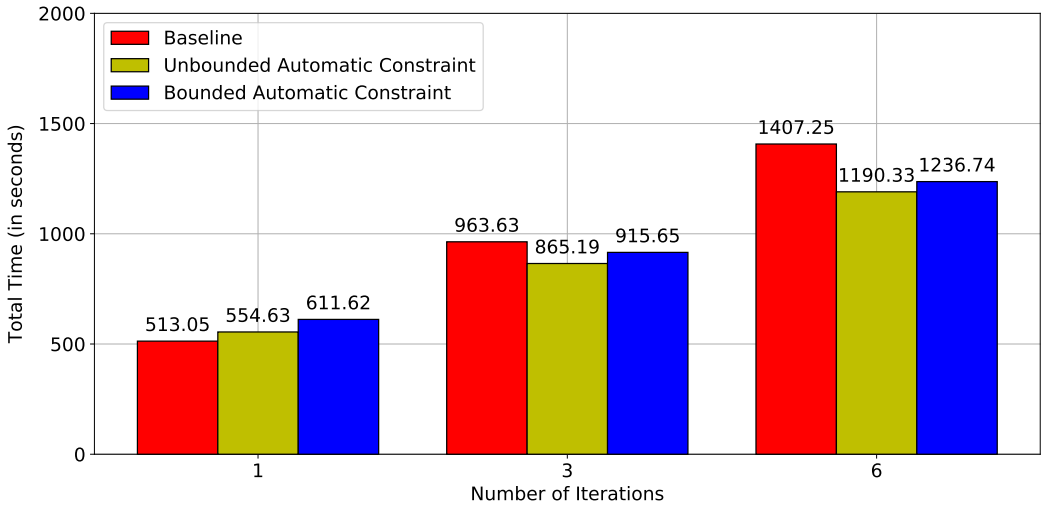


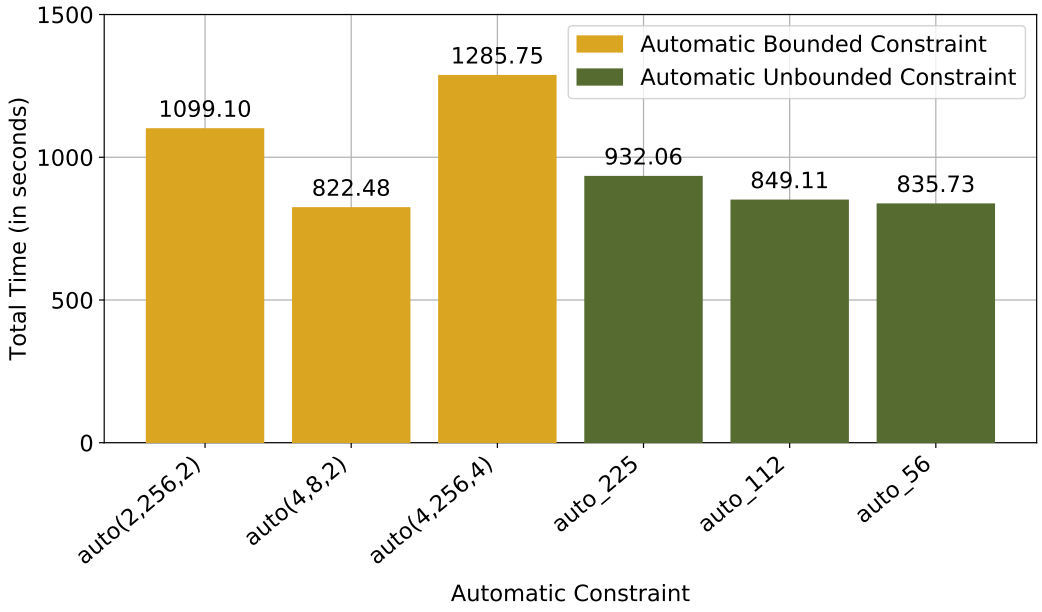

FIGURE 4.21: Kmeans Application With Different Number Of Iterations

### 4.5.4 Hyper-parameters Experiments

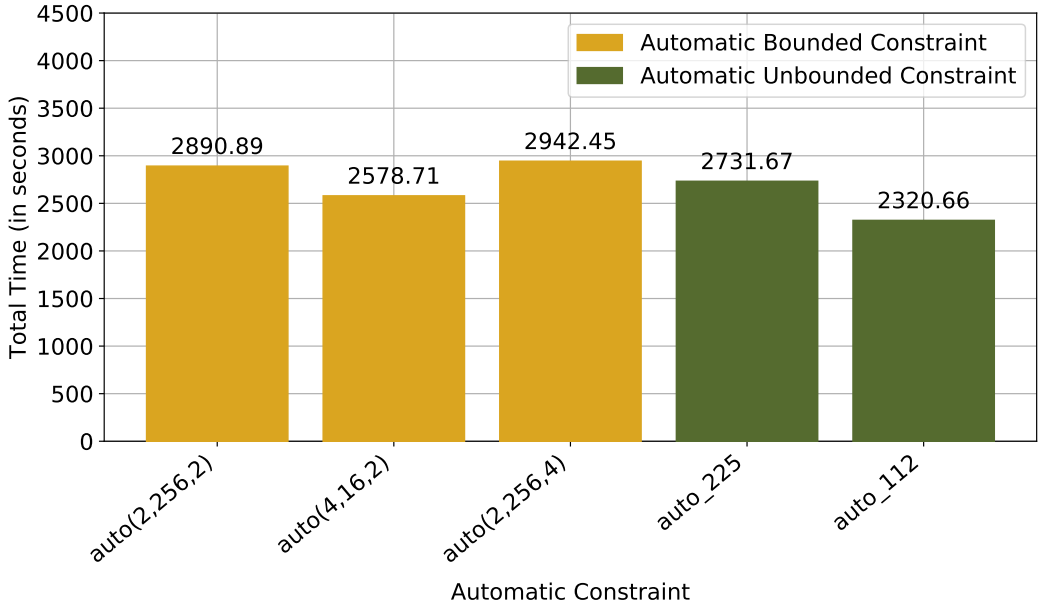
In this section, we evaluate the performance impact of changing the values of the hyper-parameters of both of the auto constraints. In the case of the bounded auto constraint, these parameters are *min*, *max* and *delta* set by the user in the `@constraint` decorator. Whereas in the case of the unbounded auto constraint, the hyper-parameter is the number of I/O executor threads per worker node.

To this end, we repeated the experiments of the HMMER application (homogeneous I/O workload) and the Variants Discovery Pipeline (heterogeneous I/O workload) using auto constraints with different hyper-parameter values. Figure 4.22 shows the experiment results for both applications. It can be noted that in both applications, changing the values of the hyper-parameters impacts the total performance. In the HMMER application (Figure 4.22(a)), the optimal constraint is 8 so setting the constraint to *auto(2,256,2)* incurs a long learning phase. Whereas adjusting the *min* and *max* values to *auto(4,16,2)* shortens the learning phase and results in a better total time. In another run, Setting a big value of *delta* like in the constraint *auto(4,256,4)* to speed up the learning phase resulted in a worse total time because a big value of *delta* skipped the optimal constraint, i.e., 8.

Furthermore, continuing with Figure 4.22(a), we can see that the unbounded auto constraint achieves better results. Using an unbounded auto constraint with 225 I/O executors




(a) Experiments Of The HMMER Application




(b) Experiments Of The Variants Discovery Pipeline

FIGURE 4.22: Hyper-parameters Experiments

incurs a longer learning time because the starting constraint is 2. However, the learning time in this case is not as long as *auto(2,256,2)* due to the strict learning conditions of the unbounded auto constraint. Decreasing the number of I/O executors results in a better total time since it approaches the optimal constraint: 8. With 112 I/O executors, the constraint of the first learning epoch will be set to 4 whereas it will be set to 8 with 56 I/O executors.

Likewise, a similar behaviour can be seen with the Variants Discovery Pipeline (Figure 4.22(b)). Adjusting the boundaries of the hyper-parameters like in *auto(4,16,2)* to decrease the learning phase improves the total time. Also, using a larger value of delta (*auto(2,256,4)*) may result in an increase in the total time because optimal constraints are skipped. On the other hand, using unbounded constraints achieve better total time: using 225 I/O executor achieves better total time than *auto(2,256,2)* because of the shorter learning phase. Moreover, using 112 I/O executors achieves better total time because the learning epoch starts with constraint value 4 which is the optimal constraint for the two checkpointing tasks at the end of the pipeline (i.e., *checkpoint_merged* and *checkpoint_grouped*) and the learning phase is shorter.

## 4.6 Discussion

This chapter of the thesis targets enabling I/O awareness in task-based programming models. I/O aware models are able to exploit I/O intensive applications and provide mechanisms for solving I/O performance bottlenecks such as I/O congestion. Such mechanisms are exposed to users with abstractions that try to express the inherent parallelism of applications and also maintain the programming complexity at minimum.

In order to take advantage of the I/O awareness capabilities, it is necessary to separate I/O from computation when programming applications. Hence, an I/O aware runtime system can take advantage of I/O properties to improve the performance of applications.

I/O awareness increases the amount of parallelism inherent in I/O intensive applications by taking advantage of the optimization opportunities possible due to I/O workloads and compute-I/O patterns. With an I/O aware task-based programming model, opportunities for compute-I/O execution overlap can be exploited. In addition to that, I/O performance bottlenecks such as I/O congestion can be mitigated, thus resulting in total time performance improvements.

Unlike global I/O system schedulers that target minimizing I/O congestion for different running applications, we presented in this chapter a mechanism for I/O mitigation that only takes into consideration application-specific parameters such as the number of I/O tasks and their performance at different levels of parallelism. These information are then used to calculate a convenient constraint value that targets both I/O performance and total application performance.

We implemented the I/O awareness capabilities in the PyCOMPSs tasking framework and evaluated it with different I/O workloads. The evaluation demonstrates that significant total performance improvement can be achieved compared to the default I/O non-aware PyCOMPSs implementation. Our experiments show that total performance will not be achieved only by overlapping the execution of compute and I/O tasks because the negative effects of increasing I/O congestion. Therefore, mitigating I/O congestion by using the storage bandwidth constraint is pivotal to get total performance improvement.

As future work, we plan to extend our proposals to address the case when shared resources are used to absorb the I/O of applications. In this scenario, certain assumptions and modified mechanisms have to be adopted to take into account the I/O performance variability on shared resources. Furthermore, we aim to extend the auto-tunable constraints to

support the inference of other constraints such as as memory size and number of processes in MPI tasks executions.

# Chapter 5

# Managing Storage Systems Heterogeneity


## SUMMARY

Task-based programming models have enabled the optimized execution of the computation workloads of applications. Such models can take advantage of large-scale distributed infrastructures by allowing the parallel execution of applications in high-level work components called *tasks*. Therefore, greatly improving the computing performance of applications.

Nevertheless, in the era of Big Data and Exascale, the amount of data produced by applications has already surpassed terabytes and is rapidly increasing. Hence, I/O became the bottleneck to overcome in order to improve applications performance.

New storage technologies offer higher bandwidth and faster solutions than traditional Parallel File Systems (PFS). Such storage devices are deployed in modern day infrastructures to boost I/O performance by absorbing the increasing amounts of data generated by applications. Hence, it is necessary for any programming model targeting more performance to manage this heterogeneity and take advantage of it to maximize performance.

This chapter focuses on solving the research question, $\mathbf{Q}_2$, concerned with providing suitable abstractions to address the heterogeneity of modern storage systems. More specifically, this chapter proposes enabling *Storage-heterogeneity awareness* in task-based systems, that is, transparently leveraging the underlying heterogeneous storage devices to improve I/O performance, consequently, optimizing whole application execution.

In this chapter, we describe a set of proposals that should be supported in any storage-heterogeneity aware programming model. These proposals include: First, abstracting the underlying heterogeneity of storage systems from application developers and organizing it in a hierarchy to improve applications performance. Second, supporting dedicated I/O schedulers with different scheduling policies to optimize the execution of different I/O workloads. Finally, an automatic data movement flushing technique to maximize the usage of faster storage devices.

In addition to that, this chapter describes the design and implementation details of the aforementioned proposals in the PyCOMPSs task-based programming model.

The evaluation section presents the performance results of different applications on the MareNostrum CTE-Power cluster. This cluster has heterogeneous storage infrastructure that contains multiple storage layers. Our experiments demonstrate that such proposals can achieve up to almost 5x I/O performance speedup and 48% total time improvement compared to the reference PFS-based usage of the execution infrastructure.

## 5.1 Overview

Task-based programming models offer suitable abstractions that allow the exploitation of large-scale distributed execution environments. Such models are able to take advantage of distributed heterogeneous computing infrastructures, therefore, delivering increased computational performance.

However, modern applications process and generate huge amounts of data [31]. As previously discussed in Section 1.1.2, scientific and big data applications produce increasing amounts of data and aim for resilience (e.g., checkpointing the applications intermediate data to enable restart after failure) [47]. In addition to that, storage systems that rely on Parallel File Systems (PFS) (e.g., Lustre [11] or GPFS [96]) face significant challenges in terms of limited performance [122]. Therefore, applications have gone through a paradigm shift where improving I/O performance becomes critical for enhancing the whole application performance [117].

As a response to the need for absorbing large amounts of data and optimizing I/O performance, large-scale systems have incorporated newly emerging storage technologies such as Non-Volatile RAM (NVRAM) [78] and Solid-State Drivers (SSD) [69] into their underlying base storage system of the PFS. These storage devices can help reducing the gap between compute and I/O performance because of their high I/O bandwidth and low latency capabilities. They act as *Burst Buffers* [63] that absorb the data produced by applications in their I/O-dominant phase. This approach can improve applications I/O performance by providing a fast solution to write data from memory and enhancing applications reliability.

Even though this heterogeneity in the storage systems design would benefit applications I/O performance, it comes with additional complexity that could prevent achieving enhanced I/O performance [17]. Each storage device needs to be provisioned according to its own capabilities in order to achieve maximum performance. For instance, because of the limited capacity of each device, I/O workload should be distributed in a manner to optimize overall I/O execution. In addition to that, each storage device has to be provisioned for bandwidth to avoid the problem of I/O congestion that negatively impacts the performance [36].

Due to the lack of sufficient mechanisms and techniques to optimize I/O performance in traditional task-based programming models (see Section 5.2), the aforementioned complexities are exposed to application programmers. It becomes their responsibility to carry the burden of planning and optimizing applications execution on heterogeneous storage systems. Leading not only to a complex development process, but also to possible underutilization of the storage system and wasted I/O performance improvement opportunities.

In this chapter, we address the need for seamlessly and transparently managing heterogeneous storage devices and taking advantage of them to optimize I/O performance. To this end, we propose enabling *Storage-Heterogeneity Awareness* in task-based programming models. The main idea herein is to abstract all the details of the storage infrastructure and expose it as a single storage unit to application programmers. Furthermore, heterogeneity-aware task-based models should provide execution time support to organize different storage devices into layers to optimize the use of each layer for performance.

Following this approach has twofold advantages:

- First, reducing infrastructure complexities and easing applications development. Application code becomes agnostic to the underlying storage system. Hence, no code modifications are required to adapt to changes in the storage infrastructure.

- Second, maximizing I/O performance given a set of storage layers with different capabilities. This performance improvement can be achieved because task-based programming models have execution time knowledge about the state of each storage layer (e.g.,

capacity, bandwidth). Thus, optimized scheduling decisions can be made. In addition to that, task-based models have knowledge about data dependencies between tasks. Such information can be used to free storage layers capacities. Hence, maximizing the utilization of storage layers that offer high bandwidth.

Supporting these capabilities at programming model level allows the optimization of execution based on the characteristic of applications such as their I/O workloads, data requirements of tasks, etc. In addition to performing data management in heterogeneous storage systems in a completely transparent manner to application developers.

The main contributions of this chapter can be summarized as follows:

- A proposal to enable task-based models to abstract the heterogeneity of storage systems and expose it as a single resource to optimize I/O performance.
- Dedicated I/O schedulers that provide different scheduling policies. Each policy targets the optimization of I/O execution for certain scenarios.
- An automatic data movement flushing mechanism to maximize the utilization of the storage system.

These proposals are implemented in the PyCOMPSs task-based programming model [100]. We demonstrate the benefits of these proposals by evaluating the implementation prototype with different applications that exhibit different I/O workloads. All the experiments were run on an execution platform with heterogeneous storage systems: The MareNostrum CTE-Power Cluster of the Barcelona Supercomputing Center. Our experiments show significant performance improvements that reached up to 5x I/O performance speedup and 48% total time improvement compared to the reference PFS-based implementation.

The rest of this chapter is organized as follows: Section 5.2 presents the related work. Section 5.3 introduces the contributions of the chapter: the I/O schedulers and the data movement mechanism. Section 5.4 describes the implementation details of the proposals in the PyCOMPSs framework. The performance results are presented and analyzed in Section 5.5. Finally, Section 5.6 discusses the main conclusions of this chapter.

## 5.2 Related Work

Task-based programming models offer different interfaces and schedulers that try to exploit large-scale infrastructures to maximize the performance of applications compute workloads. However, they do not provide similar runtime support that specifically targets I/O performance optimization. Therefore, performance improvement opportunities are wasted. Examples of such programming models were previously described in Section 2.1 and Section 4.2.

Previous research efforts targeted abstracting the heterogeneity of storage and memory systems to maximize I/O performance. MLBS [3] is a library for optimizing oil exploration simulation application by maximizing the usage of high bandwidth memories and data movement between the PFS and faster storage and memory layers. Hermes [52] and UniviStor [114] present a system for I/O buffering and optimizing data movement between different storage and memory layers. DataWarp [41] and Data Elevator [25] are burst buffer management software that enhance applications writes by redirecting them to remote-shared burst buffers from PFS. Systems such as BurstFS [112] and BurstMem [113] propose to redirect I/O write calls to node-local SSDs.

In contrast to previous work, we propose in this chapter an ***application-level programming model support*** to transparently optimize the usage of heterogeneous storage systems.

Unlike system-level solutions and middleware, our proposals do not require any system administration nor configuration knowledge or effort. Users can directly launch heterogeneity-aware PyCOMPSs applications without requiring to install or setup any filesystem or system middleware nor link any external libraries nor launch any external processes or daemons. As the proposals of this chapter are supported by the programming model, it increases applications portability because there are no prerequisite system installation nor external software configuration.

In addition to that, because the proposals of this chapter are programming model extensions, they are application-specific, i.e., they target the optimization of the running applications based on their execution patterns and performance on the underlying infrastructure, instead of relying on global system-wide metrics.

## 5.3 Storage Heterogeneity Awareness

Although the heterogeneous design of storage systems can offer drastic improvements in terms of I/O performance, programming for performance improvement on such systems is a major challenge. For instance, it is difficult to find out the optimal scheduling of the I/O tasks that would maximize the performance gain out of the storage infrastructure. Application developers have to manually determine which and how much data should be written to each storage device in the infrastructure. Such an approach can be possible with applications that exhibit a small number of I/O tasks. However, it is prohibitive in large applications that have a big number of I/O tasks and produce different amounts of data.

Consequently, the main purpose of our work in this chapter is twofold:

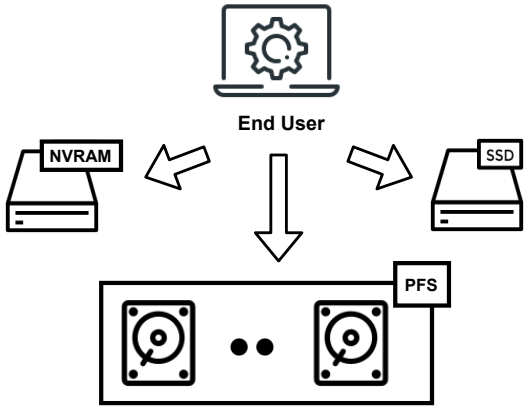
- On the one hand, to maximize the advantage of the heterogeneity of modern storage infrastructure to alleviate the performance gap between I/O and computation.
- On the other hand, achieve such performance maximization in a transparent manner without increasing programming difficulty.

We argue that this goal can be achieved by enabling task-based models to be *Storage-heterogeneity Aware*. Defining a task-based model as storage heterogeneity-aware means that it is able to exploit the capabilities of the underlying storage infrastructure to improve I/O performance in a transparent way to application developers. To achieve such transparency, programming models should abstract the heterogeneity of the underlying storage system and expose it as a single pooled resource. Different storage devices can be organized into layers according to the bandwidth of each device. Such hierarchical arrangement can be made so that the highest storage layer offers the highest bandwidth, while the bottom layer offers the least bandwidth. If two storage devices have the same bandwidth, then the device that has less capacity should precede the device that has more capacity.

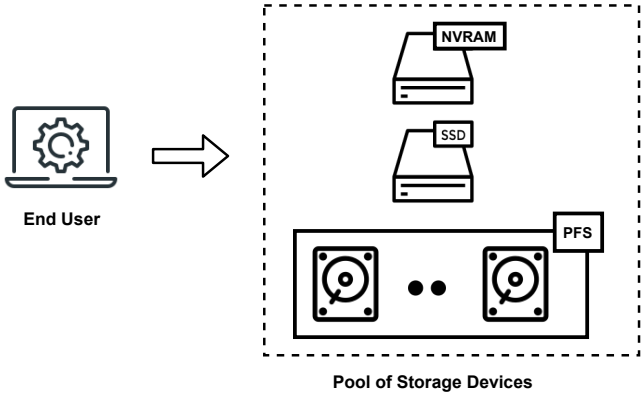
The objective of such an arrangement is to increase I/O task parallelism as much as possible by maximizing the usage of higher layers. As higher storage layers provide higher bandwidth, they can allow more I/O task parallelism (i.e., more I/O tasks running concurrently) without causing I/O congestion.

Figure 5.1 shows the two approaches of handling heterogeneous storage systems. The traditional heterogeneous non-aware approach (Figure 5.1(a)) exposes all the storage devices to application developers. With this approach, users manually specify the full path of the storage device to write the data to it. There is no runtime support to control the assignment of tasks to different storage devices. Whereas our proposed heterogeneous-aware approach (Figure 5.1(b)) organizes the storage devices into one pooled resource. In this case, the heterogeneity of the storage system is hidden from applications. Consequently, there is no need to modify the applications to adapt to infrastructure changes. In addition to that,

it is the responsibility of the runtime system to exploit the capabilities of the storage infrastructure to optimize applications execution.




(a) Heterogeneous Non-Aware




(b) Heterogeneous Aware

FIGURE 5.1: Different Views Of Storage Systems Heterogeneity

Figure 5.2 depicts a high-level view of the heterogeneous-aware organization of the storage system of the MareNostrum CTE-Power cluster of the Barcelona Supercomputing Center [33]. A storage-heterogeneity aware programming model would organize the storage infrastructure of the CTE-Power cluster into three hierarchical layers: the top layer contains two NVRAM devices, each with a maximum bandwidth of ~6 GB/s and capacity of 3 TBs. In the second layer, there are two SSD storage devices, each with a maximum bandwidth of ~3 GB/s and 1.9 TB capacity. Finally, the PFS resides at the bottom layer with a maximum bandwidth of ~900 MB/s and total capacity of 8 Petabytes.

In addition to heterogeneity abstraction, heterogeneous-aware task-based systems should optimize the execution of applications by scheduling I/O tasks in a manner to exploit the capabilities of the different storage devices. Since higher storage layers offer more bandwidth, they allow more task parallelism. The higher the layer, the more I/O tasks that can be launched concurrently on this layer without exceeding its maximum bandwidth. Hence, improving performance.

In this chapter, we propose dedicated I/O schedulers. Each scheduler offers a different policy to optimize I/O tasks execution taking advantage of the heterogeneous-aware view of storage infrastructures.

Furthermore, in order to maximize the usage of higher storage layers, we propose an automatic data movement mechanism to flush obsolete data from higher layers to lower levels

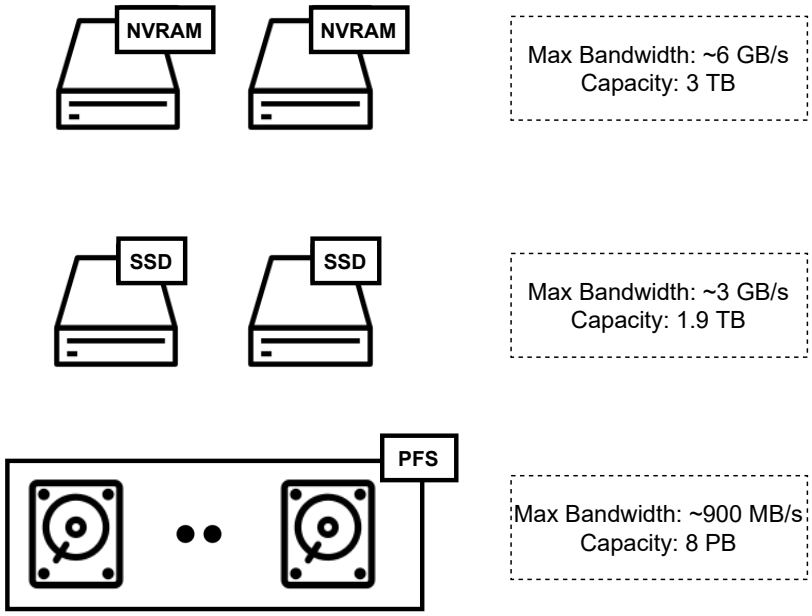



FIGURE 5.2: Heterogeneous-Aware Storage Layers Organization Of The MareNostrum CTE-Power Cluster

in a transparent manner to application developers. Consequently, continuously freeing the capacities of higher storage layers so that more tasks can be scheduled to them. Identifying such data is possible because the runtime system of the task-based programming model already has information about if any data would be required as inputs by any future successor tasks.

The remainder of this section is decomposed as follows: First, we describe the programming model abstractions to specify I/O tasks and their bandwidth requirements in Section 5.3.1. Then, Section 5.3.2 describes the scheduling model. Section 5.3.3 introduces two I/O dedicated schedulers with their different policies. Finally, Section 5.3.4 details the automatic approach for flushing data.

### 5.3.1 Programming Model Abstractions

The focus of this thesis, and consequently of this chapter, is to introduce techniques to optimize I/O performance without increasing the complexity of applications development. Therefore, we take advantage of the I/O awareness abstractions that were previously described in Chapter 4 to implement the proposals of this chapter.

In order to separate I/O from computations, we use the *I/O Task* abstraction that has been already discussed in Section 4.4.1. I/O tasks can be handled by the programming model runtime system to exploit parallelism opportunities to optimize applications execution. For instance, I/O tasks can be launched for execution even if there are no free computing resources. Since I/O tasks do not consume a lot of CPU time, their execution can be overlapped with compute tasks. An I/O task can be defined by annotating the target function with the PyCOMPSs-defined Python decorator: `@IO`.

In addition to that, in order to specify constraints for I/O tasks execution such as estimated storage bandwidth and capacity, we use the `@constraint` PyCOMPSs decorator as previously described in Section 4.4.2. The `@constraint` decorator enables the specification of the required bandwidth and capacity of a task by the use of `storageBW` and `storageSize` respectively.

Since the identification of the required bandwidth of tasks may not be simple at application design time, we provide an automatic mechanism for setting and auto-tuning tasks storage bandwidth constraints that is described in Section 4.4.3. Following this manner, users can rely on the PyCOMPSs runtime system to set a suitable storage bandwidth constraint that will not optimize I/O performance, but also total application performance.

In both approaches (i.e., static or auto-tunable constraints), the model assumes that a certain I/O task will produce the same I/O workload (e.g., write the same amount of data) over application lifetime. This assumption is useful as it allows to take educated decisions at scheduling time

Finally, in order to completely hide the storage infrastructure details from the application code, the type of the target output files of an I/O task has to be specified as $FILE\_OUT$ in the `@task` decorator. Such a type indicates to the COMPSs runtime that the associated parameter is going to be an output file. Consequently, the COMPSs runtime redirects all the I/O that will be done to this file to the device where the task has been scheduled. Hence, in the task code, users can open a file for writing as if the file is in the current working directory, i.e., without having to specify the complete path of the storage device. Moreover, the COMPSs runtime handles data transfers between different storage layers or different working nodes in a transparent manner.

Figure 5.3 illustrates the use of our proposed programming model abstractions to abstract storage heterogeneity from task implementation and to transparently separate the handling I/O and computations. The main part of the application code (Line 14) contains multiple calls to two tasks: (i) A `calculate` task which is computing a certain value. (ii) A `checkpoint` I/O task, which is writing the data that has been produced by the `calculate` task. The required storage bandwidth (`storageBW`) and size (`storageSize`) are specified using the `@constraint` decorator. The storage bandwidth can be explicitly specified or delegated to the COMPSs runtime by the use of the auto-tunable constraints. When both task calls arrive to the COMPSs runtime, the task dependency graph is built according to the data dependencies, in this case every execution of the `checkpoint` task has a dependency with a corresponding execution of a `calculate` compute task. However, at scheduling time, the `calculate` task is assigned to the compute scheduler whereas the `checkpoint` task is scheduled by the I/O scheduler. Moreover, it should be noted that in Line 7, the parameter `filename` is defined as a `FILE_OUT` parameter. The details of to which storage device the task has been scheduled and to which storage device the file will been written does not affect the task implementation (Line 9).

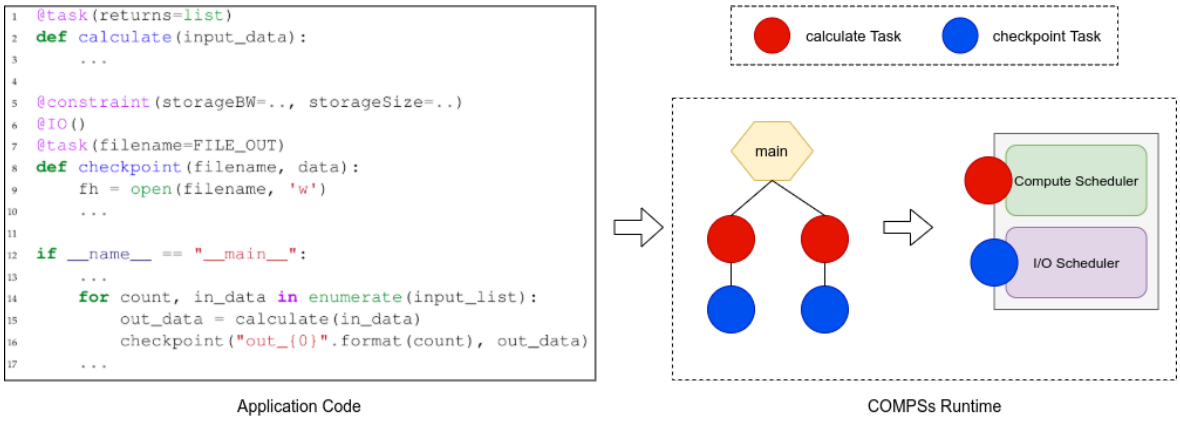



FIGURE 5.3: Programming Model Support For Storage-Heterogeneity Awareness

It should be noted that the available storage devices and their capabilities (e.g., capacity,

bandwidth) are described in a configuration file and passed to PyCOMPSs at application launch time. By using the maximum bandwidth that a storage device provides and the bandwidth required by I/O tasks, the scheduler can control the scheduling of tasks.

### 5.3.2 Scheduling Model

I/O schedulers can take optimized scheduling decisions based on execution time objectives. Such objectives can be specified as avoiding I/O congestion and maximizing the utilization of higher layers in the storage hierarchical organization.

All I/O tasks should be scheduled with the I/O scheduler in a transparent manner. Hence, application developers do not have to be concerned about how the heterogeneous storage resources should be used to get the maximum benefit out of the storage infrastructure.

Before introducing the scheduling model, it is important to arrange the available storage resources of each worker into the hierarchical view discussed in Section 5.3. After launching the application, the runtime system loads all available information about the storage layers and creates a list ranked from the highest bandwidth to the lowest bandwidth in each worker resource. The scheduling of I/O tasks to the storage layers should be done from top to bottom across all storage layers, starting from the storage layer that offers the highest bandwidth until it reaches the bottom storage layer that has the lowest bandwidth.

The scheduling routine is presented in Algorithm 1. It tries to schedule an I/O task $t$ to the highest possible layer of one of the workers in the workers set $W$. If the task is scheduled, it returns $True$, otherwise, $False$. Line 2 defines $W_{candidates}$ which is the set of workers that can currently host task $t$ execution. Line 3 retrieves task $t$ storage requirements, i.e., storage bandwidth $BW_{Req}$ and capacity $C_{Req}$. In Lines 4 through 12, every storage layer on each worker is considered. For each layer $l$ on worker $w$, the storage parameters of the layer are retrieved, i.e., current available bandwidth $BW_{Avail}$ and capacity $C_{Req}$ (Line 6). Line 7 describes how the decision of whether to schedule the I/O task $t$ to a certain storage layer $l$ on worker $w$ is made. Such decision depends on two execution time variables:

i Whether the bandwidth requested by the task ($BW_{Req}$) exceeds the current available bandwidth of the storage layer ($BW_{Avail}$).

ii Whether there is enough capacity ($C_{Avail}$) on this layer to satisfy the task required capacity ($C_{Req}$).

If the storage layer $l$ does not have enough capacity, or if launching the task will cause I/O congestion, then the scheduler considers the next storage layer of worker $w$. Otherwise, if the storage and bandwidth conditions are satisfied, then worker $w$ is added to the set of candidates that can host the task execution $W_{candidates}$. Then, Line 9 skips the rest of the layers on the current worker because it is guaranteed that the next layers provide less bandwidth.

After all the workers $W$ are considered, Line 13 checks if the set of candidate workers $W_{candidates}$ contains any worker. On the one hand, if it does not contain any worker then this means that the requirements of task $t$ cannot be satisfied by any of the storage layers of any worker. Hence, the scheduler decides not to launch the task and waits to the next scheduling iteration, so that enough bandwidth becomes available when some of the currently running tasks finish execution.

On the other hand, if $W_{candidates}$ contains candidate workers, then Line 16 retrieves the best candidate worker $w_{target}$ and its storage layer $l_{target}$ out of all candidate workers set. The best worker is the one that has the highest storage layer that can host the task execution.

Finally, the bandwidth and capacity of the target layer $l_{target}$ are updated (Lines 18, 19) and the target worker $w_{target}$ and layer $l_{target}$ is set for task $t$ (Lines 20, 21).

The scheduling algorithm in Algorithm 1 is simple but it reflects the motivation of this work, that is, to improve I/O performance by prioritizing the usage of higher/faster storage layers while also preventing I/O congestion. This is achieved by monitoring the current state of the storage resources and controlling I/O tasks scheduling.

**Algorithm 1:** Scheduling Algorithm

**Input :** $t$ as I/O task, $W$ as the set of available workers
**Output:** $True$ if the task can be scheduled, $False$ otherwise

```
1  Function Schedule(t, W):
2      W_candidates ⟵ ∅
3      BW_Req, C_Req ⟵ getStorageReqs(t)
4      foreach w ∈ W do
5          foreach l ∈ Layers_w do
6              BW_Avail, C_Avail ⟵ getAvailStorageParams(l)
7              if BW_Req ≤ BW_Avail and C_Req ≤ C_Avail then
8                  W_candidates.insert(w)
9                  Continue
10             end
11         end
12     end
13     if W_candidates == ∅ then
14         return False
15     else
16         w_target, l_target ⟵ getBestCandidate(W_candidates)
17         BW_Avail, C_Avail ⟵ getAvailStorageParams(l_target)
18         BW_Avail ⟵ BW_Avail − BW_Req
19         C_Avail ⟵ C_Avail − C_Req
20         t.setTargetWorker(w_target)
21         t.setTargetStorageLayer(l_target)
22         return True
23     end
24 End Function
```

Since each storage layer is a scarce resource with limited capacities, the decision of which I/O task to be considered for scheduling on a certain storage layer is crucial to our objective, i.e., maximizing the utilization of higher layers to increase I/O task parallelism and improve I/O performance. To this goal, in the next sections, we introduce two classes of I/O schedulers with different scheduling policies. Each scheduler is responsible for taking

an execution time decision about which tasks should be considered for scheduling on which storage layer to optimize I/O execution in certain scenarios.

### 5.3.3 I/O Schedulers

This section introduces two different classes of I/O schedulers:

1. Homogeneous I/O Scheduler (Section 5.3.3.1).
2. Heterogeneous I/O Scheduler (Section 5.3.3.2).

Both schedulers follow the same core behaviour: suppose that an application executes a number of $T$ I/O tasks, each scheduler has a dependency-free subset of the $T$ tasks to be launched for execution on a number of worker nodes with heterogeneous storage layers. Both schedulers use the algorithm described in Section 5.3.2 to check if a task can be scheduled to a certain layer on a worker. However, each scheduler differs in the strategy of choosing which I/O task to be considered first for scheduling.

#### 5.3.3.1 Homogeneous I/O Scheduler

We propose this class of schedulers to be used when the I/O workload of applications is *homogeneous*, i.e., the amount of data to be written by I/O tasks does not change throughout the lifetime of the application. Hence, I/O tasks can be scheduled to a suitable storage layer without worrying about scheduling fairness or not achieving optimal performance. This is because this class of I/O schedulers assumes that it will not receive a more *critical* I/O workload at a later point of the execution, i.e., I/O tasks that write more data and require more bandwidth.

Under this class of schedulers, we propose a *First Come First Served* (FCFS) I/O scheduler. The next section describes this scheduling scheduler in more detail.

##### 5.3.3.1.1 First Come First Served

This scheduler works similarly to a traditional *First Come First Served* (FCFS) job scheduler. It schedules I/O tasks depending on the order in which they arrive. Figure 5.4 illustrates the behaviour of this scheduler. Each storage layer is considered in terms of current available capacity and storage bandwidth, if either properties does not satisfy the task requirements, then the next layer is considered.

The pseudocode in Algorithm 2 describes the details of how the FCFS I/O scheduler works. For all the tasks in the task set $T$, try to schedule current task $t_i$ on all the workers $W$ by calling the `Schedule` routine that is described in Algorithm 1. If the current task $t_i$ can be scheduled, then it is added to the set of scheduled tasks $T_{Sched}$ to be launched for execution. Otherwise, if task $t_i$ cannot be scheduled, it waits in the queue until the next scheduling iteration. Then, the next task is considered.

As previously presented in Algorithm 1, an I/O task will only be scheduled on the target layer if there is enough capacity in the target layer as well as the execution of this task will not create a bandwidth contention on the layer.

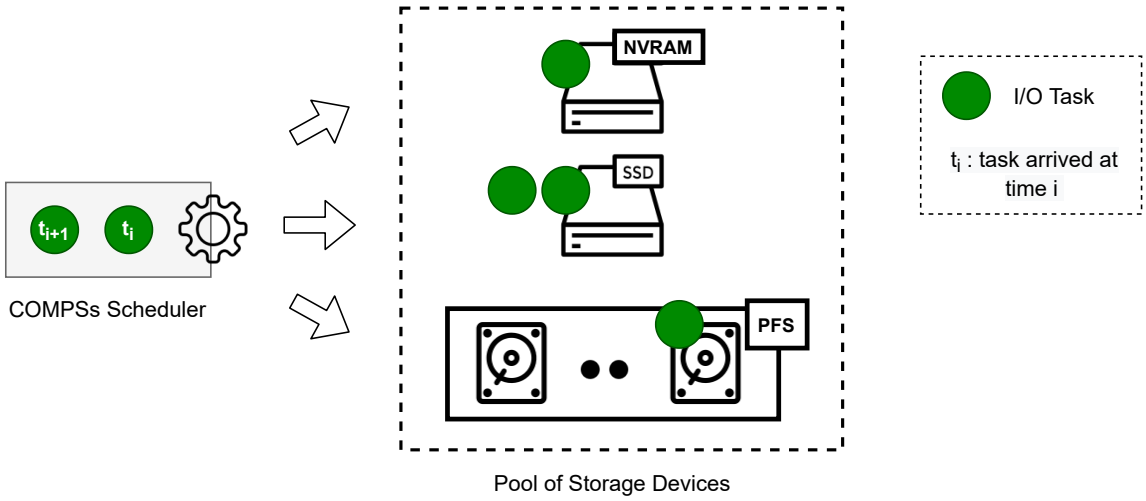



FIGURE 5.4: Overview Of The Homogeneous FCFS I/O Scheduler; Tasks are Scheduled In A Top-Down Fashion In A First Come First Served Manner

**Algorithm 2:** First Come First Served I/O Scheduler

**Input :** $T$ as the set of I/O Tasks ready for scheduling, $W$ as the set of available workers

**Output:** $T_{Sched}$ set of scheduled I/O tasks

```
1 Function FCFS (T, W) :
2     foreach t_i ∈ T do
3         if Schedule(t_i, W) then
4             add t_i to T_Sched
5         end
6     end
7     return T_Sched
8 End Function
```

#### 5.3.3.2 Heterogeneous I/O Schedulers

While the FCFS scheduler tries to take advantage of storage layers that offer higher bandwidth, it may not be suitable for applications that have multiple I/O tasks, each task producing a different amount of I/O (i.e., have different storage bandwidth and capacity requirements). In such scenario, using a FCFS scheduler may lead to unfair scheduling that results in sub-optimal performance. This can be explained because the FCFS scheduler processes the tasks in the order in which they arrive, and it may occur that the first batch of tasks to arrive to the scheduler requires less bandwidth compared to tasks that arrive afterwards. Hence, tasks that require higher bandwidth will be scheduled on lower storage layers because the first batch of tasks have consumed the bandwidth and/or capacity of the higher storage layers. Therefore, less parallelism will be achieved. In order to overcome this issue, we propose the *heterogeneous* I/O schedulers that can be used for optimizing applications that exhibit heterogeneous I/O workload.

A heterogeneous I/O scheduler takes into consideration the bandwidth requirements of tasks, such that tasks that have higher bandwidth requirements are given precedence to be scheduled on higher storage layers. In addition to that, it uses execution time information about tasks performance on the storage layers to optimize the scheduling decisions (as will be explained in Sections 5.3.3.2.1 and 5.3.3.2.2).

The rationale behind the heterogeneous I/O scheduler is to maximize the utilization of storage layers that offer higher bandwidth because they allow more task parallelism, i.e., allow a bigger number of higher bandwidth tasks to run concurrently than bottom layers that offer less bandwidth. Hence, task parallelism is increased and improved I/O performance can be achieved.

Such behaviour is achieved by sorting the incoming I/O tasks according to their bandwidth requirements in a descending order, so that the tasks that require higher bandwidth would be scheduled before the tasks that require less bandwidth.

Figure 5.5 illustrates an overview of how the heterogeneous I/O schedulers work. Tasks with higher bandwidth requirements (red tasks then yellow tasks) will be scheduled before tasks with lower bandwidth requirements (green tasks) even though they have arrived later than the green tasks. Hence, more critical tasks will run concurrently on the higher storage layers because such layers provide more bandwidth.

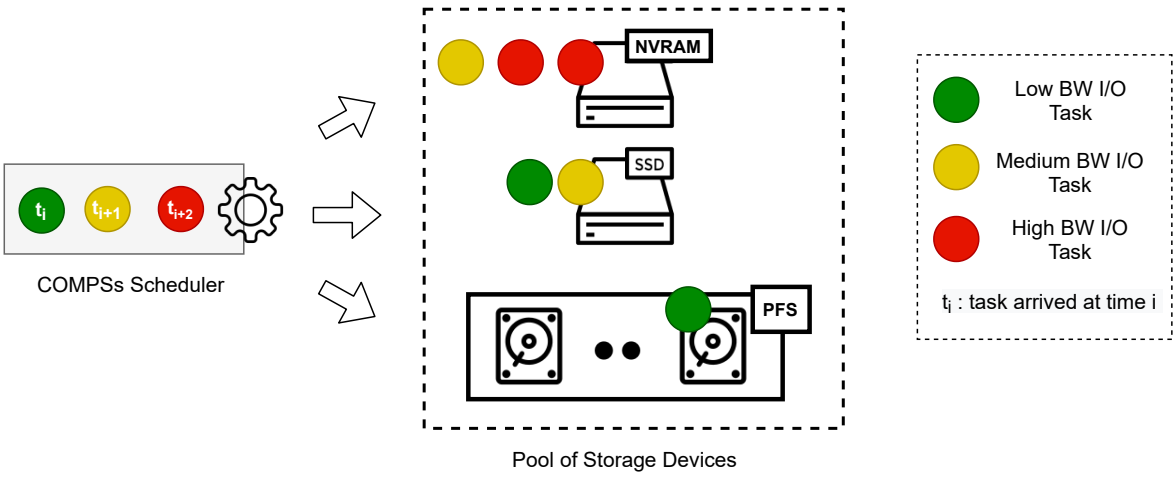



FIGURE 5.5: Overview Of Heterogeneous I/O Schedulers; Tasks are Scheduled In A Top-Down Fashion According To Their Bandwidth Requirements

We enable two scheduling policies for this scheduler: *Modified Priority* (Section 5.3.3.2.1) and *Backfilling* (Section 5.3.3.2.2).

#### 5.3.3.2.1 Modified Priority

This scheduler is similar to a traditional priority scheduler. Tasks are scheduled according to a certain priority that is determined by a task requirement. In our case, a task has a higher priority if it has a higher storage bandwidth requirement. Tasks with higher priority are scheduled before tasks with lower priority, whereas tasks with equal priorities are scheduled in a first come first served manner.

Figure 5.6 shows the execution trace of a sample application that uses the modified priority I/O scheduler. This application has 10 I/O tasks launched at the same time: 4 tasks have a certain bandwidth requirement (in red), while the other tasks have a lower bandwidth requirement (in blue). The Y-axis of this figure represents the threads that execute the tasks, while the X-axis represents time. The storage system can host the execution of a maximum of 2 concurrent red tasks or 4 concurrent blue tasks. Using the Modified Priority scheduler, blue tasks have to wait for the scheduling and execution of red tasks to be completely finished. Tasks are strictly scheduled in the order of their bandwidth requirements. If any red tasks cannot be currently scheduled, the whole scheduling iteration is aborted until there are enough resources to schedule the waiting task.

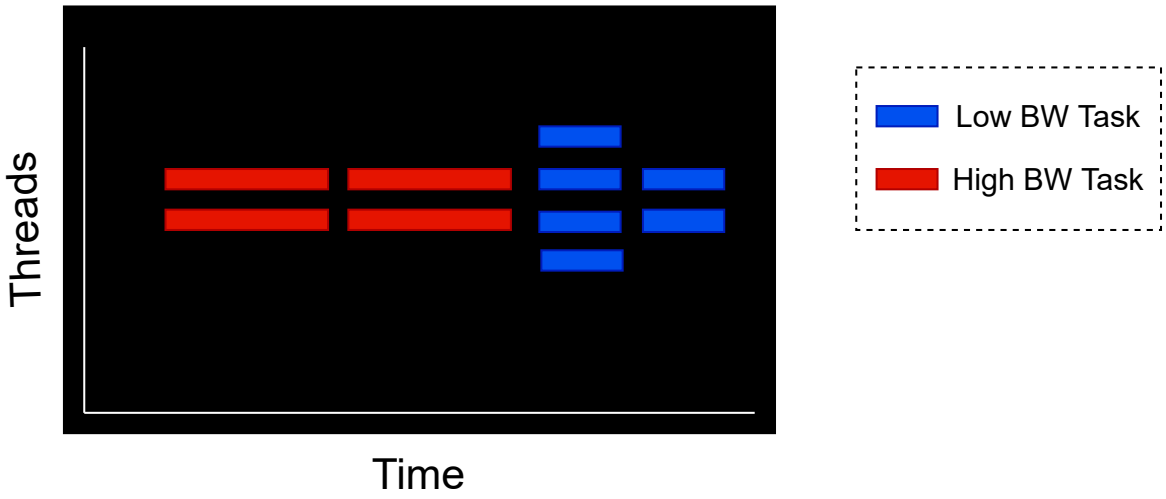



FIGURE 5.6: Execution Trace of An Application That Uses Modified Priority Scheduler

We modified the behaviour of the priority scheduler to maximize the utilization of higher storage layers. This modification allows the scheduler to not launch a certain task immediately and wait until a higher storage layer becomes available on any worker. If a task has been already scheduled to be executed and has been an assigned storage layer of a certain worker, this task will not be immediately launched to the assigned layer, instead, it will wait until its requirements become available on a higher storage layer. The decision of whether to immediately launch the task or not is based on whether there is an execution time benefit from waiting until a higher storage layer becomes available. Such benefit is estimated by checking if the combined time of waiting until a layer becomes ready plus the average task execution time on that layer is less than the average task execution time on the assigned layer.

Algorithm 3 depicts the pseudocode for the priority scheduler . The first difference between this scheduler and the FCFS scheduler is that I/O tasks have to be sorted according

to their bandwidth before the start of the scheduling process. Hence, while going through the storage layers of the workers from top to bottom, the scheduler first considers tasks with higher bandwidth for each layer. Using the priority scheduler , if a task cannot be scheduled to a certain layer then the scheduler stops and does not consider the next task in the queue until the next scheduling iteration. This approach tries to avoid that the capacities of the storage layers get totally consumed by tasks with less bandwidth.

It should be noted that the scheduler starts another scheduling iteration once any task returns from execution, freeing storage resources that can be re-used such as bandwidth.

**Algorithm 3:** Priority Scheduler

**Input :** $T$ as the set of I/O Tasks ready for scheduling, $W$ as the set of available workers

**Output:** $T_{Sched}$ as the set of scheduled I/O tasks

```
1  Function PRIORITY(T, W):
2      T_Sorted ⟵ sortBWDescending(T)
3      foreach t_i ∈ T_Sorted do
4          if Schedule(t_i, W) then
5              if canMaximize(t_i, W) then
6                  Break
7              else
8                  add t_i to T_Sched
9              end
10         else
11             Break
12         end
13     end
14     return T_Sched
15 End Function
```

Another notable difference in Algorithm 3 is the *canMaximize* routine (Line 5). The purpose of this routine is to maximize the utilization of higher storage layers. If a task can benefit from waiting until a higher layer can host it, then it should not be immediately launched for execution to the assigned worker and storage layer.

Algorithm 4 shows the pseudocode of the *canMaximize* routine. For all available resources, it goes through all the storage layers with higher bandwidth than the task's assigned storage layer $l_{assigned}$. For each layer, the benefit of making the task wait until a higher layer becomes available is checked (Line 9). This is done by comparing the time a task has to wait to be executed on the candidate layer ($w_l$) added to the average execution time on the candidate layer ($e_l$) against the average execution time on the assigned layer ($e_{assigned}$). If there is a time benefit from not immediately scheduling the task, then the routine returns to the main scheduling routine in Algorithm 3. Otherwise, the same check is repeated to the remaining storage layers on the remaining workers.

The waiting time for a task, i.e., the time a task has to wait to be executed on a certain storage layer, is calculated by calling the `estimateWaitingTime` routine (Line 8). It can

be found by calculating the minimum remaining execution time of all the tasks currently running on that storage layer. The remaining execution time of a running task on a certain storage layer $l$ can be calculated using the following formula:

$$remainingTime_t = averageTime_t - startTime_t$$

where:
$remainingTime_t$: remaining execution time of task $t$ on the storage layer.
$averageTime_t$: average execution time of task $t$ on the storage layer.
$startTime_t$: start execution time of task $t$ on the storage layer.

**Algorithm 4:** Storage Layer Maximization

**Input :** $t$ as I/O task, $W$ as set of available workers
**Output:** $True$ If it is better to wait for higher layer, $False$ otherwise

```
1  Function canMaximize(t, W):
2      l_assigned ⟵ getAssignedStorageLayer(t)
3      foreach w ∈ W do
4          foreach l ∈ Layers_w do
5              if l ≥ l_assigned then
6                  Continue
7              else
8                  waitingTime ⟵ estimateWaitingTime(l)
9                  if waitingTime + e_l < e_assigned then
10                     return True
11                 end
12             end
13         end
14     end
15     return False
16 End Function
```

It should be noted that most of the time it is guaranteed that the running task with minimum remaining execution time will free enough storage bandwidth so that the next task can be scheduled. This happens because tasks are sorted in descending bandwidth order, therefore, the task that is about to finish has equal or more bandwidth than the next task to be scheduled.

#### 5.3.3.2.2 Backfilling

Using the priority scheduler, even though there may be some available storage resources, the scheduling process stops if these available resources do not satisfy the requirements of the task in the head of the scheduling queue. Such a strict scheduling approach can lead to idle resources and increase the waiting time of the other tasks. Hence, wasting performance improvement opportunities.

To address the shortcomings of the priority scheduler, we propose a backfilling scheduler. Such a scheduler allows other tasks to be scheduled and launched as long as they will not delay the start of the tasks in the head of the scheduling queue. Unlike the priority scheduler, if a task cannot run because its storage requirement is not satisfied, then the next task in the scheduling queue is considered for scheduling. Tasks with less bandwidth are scheduled and launched if and only if they will not delay the start of the waiting task.

Figure 5.7 illustrates the execution trace of the sample application that was described at the beginning of Section 5.3.3.2.1, but in this case it uses the backfilling I/O scheduler. Unlike the execution trace shown in Figure 5.6, if any of the red tasks cannot be scheduled, blue tasks can be backfilled with the execution of the red tasks if they will not delay the start of any waiting red task.

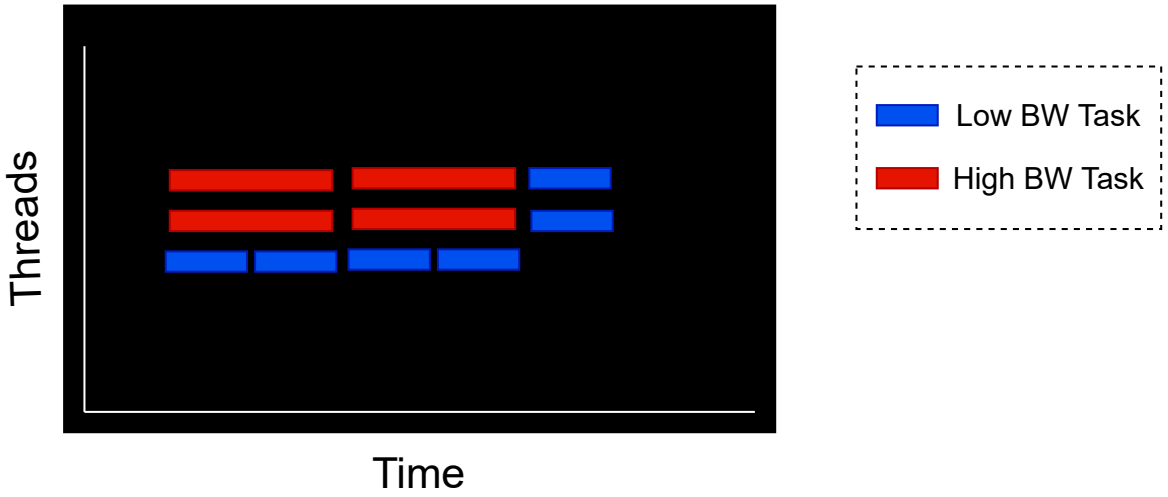



FIGURE 5.7: Execution Trace of An Application That Uses Backfilling Scheduler

Algorithm 5 shows the pseudocode of the backfilling scheduling scheduler . Similar to the priority scheduler , all tasks are sorted according to their bandwidth requirements in a descending order. However, unlike the priority scheduler , the backfilling scheduler tries to schedule the next task in the queue if the current task cannot be currently scheduled because there are no enough resources. The variable $waitingTask$ (Line 3) holds the waiting task and the variable $waitingTime$ (Line 4) indicates the minimum time the $waitingTask$ will have to wait to be launched for execution, i.e., its earliest possible start time.

The pseudocode in Algorithm 5 describes three situations that should be considered when a task $t_i$ is being considered for scheduling:

- If the current task $t_i$ can be scheduled and $t_i$ is the waiting task from the previous scheduling iterations, this means that this iteration is a new scheduling iteration and $t_i$ is not a waiting task anymore (Lines 7-11). Therefore, task $t_i$ can be added to the scheduled tasks $T_{Sched}$ to be launched for execution and the algorithm proceeds to schedule the following remaining tasks.

- If there is no $waitingTask$ (Line 13), tasks are scheduled directly. However, if there is a $waitingTask$ (Line 14), a task $t_i$ will only be scheduled if and only if its execution on the target layer ($e_{estimated}$) will not delay the waiting task, i.e., the execution time $e_{estimated}$ is less than the $waitingTime$ of the waiting task.

- Finally, if a task cannot be scheduled (Lines 18-24), then $t_i$ will be marked as the waiting task if there is no previously assigned waiting task at the head of the tasks set. The

$waitingTime$ of a task is estimated by the system by a call to the `estimateWaitingTime` routine that is similar to the one described in the priority scheduling scheduler .

It should be noted that when calculating the waiting time of a task, this algorithm does not check whether the task with the minimum remaining time will free enough resources to enable the execution of the waiting task. Such an approach may not be very exact compared to another approach that calculates the exact minimum remaining time of tasks that free enough resources after their execution finishes. Nevertheless, our proposed approach avoids spending a lot of time doing computation in a critical system component such as the scheduler.

The backfilling scheduler performs more work than the priority scheduler because it considers the scheduling of less critical tasks while more critical ones are waiting for storage bandwidth resources to become available. Therefore, the backfilling scheduler may achieve more performance compared to the priority scheduler in some situations. However, it has a disadvantage that less critical tasks will consume the capacities of higher storage layers. Hence, it may not become possible to schedule more critical tasks to higher storage layers when storage bandwidth becomes available. The evaluation section discusses this point (Section 5.5).

**Algorithm 5:** Backfilling Scheduler

**Input :** $T$ as the set of I/O Tasks ready for scheduling, $W$ as the set of available workers

**Output:** $T_{Sched}$ as the set of scheduled I/O tasks

```
1  Function BACKFILLING(T, W):
2      T_Sorted ⟵ sortBWDescending(T)
3      waitingTask ⟵ Null
4      waitingTime ⟵ 0
5      foreach t_i ∈ T_Sorted do
6          if Schedule(t_i, W) then
7              if waitingTask != Null and waitingTask == t_i then
8                  waitingTask ⟵ Null
9                  add t_i to T_Sched
10                 Continue
11             end
12             e_estimated ⟵ getEstimatedExecutionTime(t_i)
13             if waitingTask == Null or
14             waitingTask != Null and e_estimated < waitingTime then
15                 add t_i to T_Sched
16                 Continue
17             end
18         else
19             if waitingTask == Null then
20                 waitingTask ⟵ t_i
21                 waitingTime ⟵ estimateWaitingTime()
22                 Continue
23             end
24         end
25     end
26     return T_Sched
27 End Function
```

### 5.3.4 Automatic Data Flushing

All the I/O schedulers introduced in the previous sections schedule I/O tasks to the storage layers in a top-down manner from the storage layer with the highest bandwidth to the

storage layer with the lowest bandwidth. Every time an I/O task is scheduled to a storage layer, it consumes a certain amount of the storage layer capacity because it is writing data. During application execution, the capacities of higher storage layers get consumed until no more tasks can be scheduled to these layers anymore. As a result, in the next scheduling iterations, tasks will be scheduled to the bottom layers that have more free capacity but offer less bandwidth. Hence, less task parallelism can be achieved.

In order to overcome this issue and maximize the utilization of higher storage layers, we propose an automatic data flushing mechanism. The main idea is to transparently and periodically flush obsolete data from higher storage layers to lower storage layers. Consequently, the system continuously frees up capacity in higher storage layers. Hence, in the next I/O scheduling iterations, higher storage layers would have enough capacity to host the execution of more critical I/O tasks.

Looking back at Figure 5.4 that illustrates how a FCFS scheduler works, since both the NVRAM and SSD layers are full, incoming tasks $t_i$ and $t_{i+1}$ will be scheduled to the PFS layer where there is enough capacity. Therefore, leading to less task parallelism and degraded performance because the PFS has lower bandwidth than the NVRAM and SSD. This problem becomes more apparent when tasks of high bandwidth requirements arrive to the scheduler. Adopting the automatic flushing mechanism will help mitigate this problem as depicted in Figure 5.8. Since data on higher layers are flushed to bottom layers (in this case, the PFS), when future tasks arrive to the scheduler, they can be scheduled to higher storage layers. Thus, increasing task parallelism and improving I/O performance.

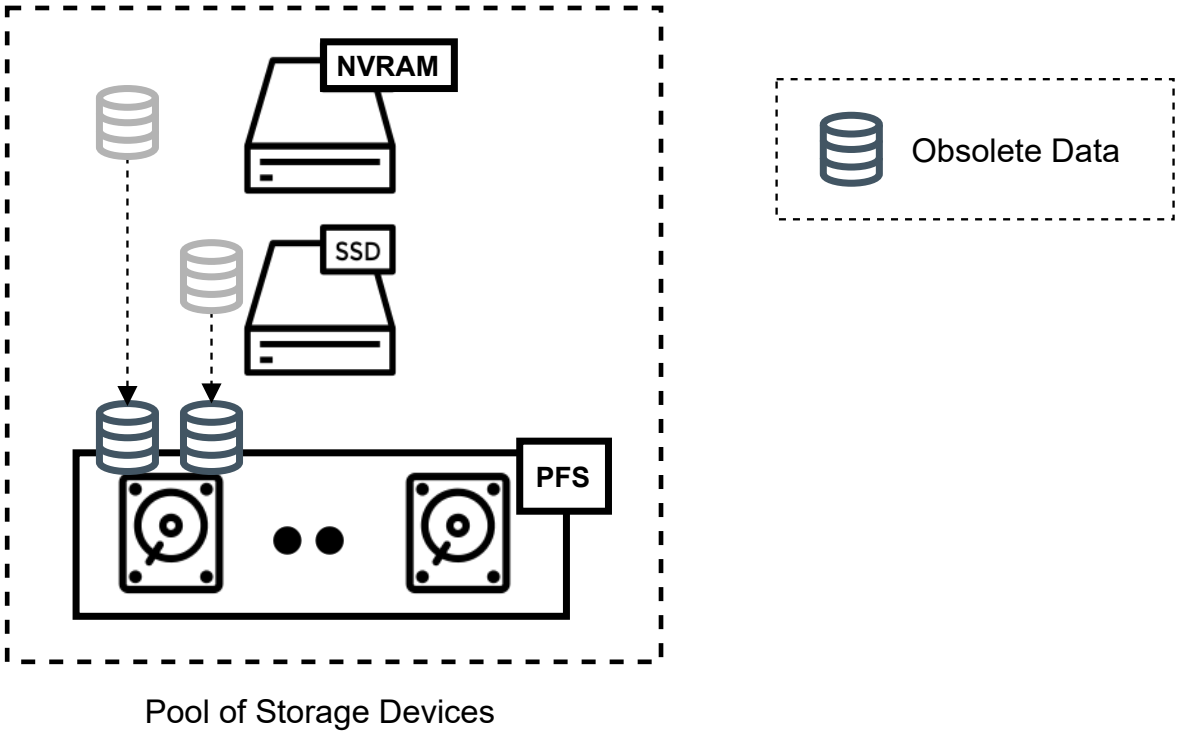



FIGURE 5.8: Flushing Mechanism Maximizes Storage Devices Utilization by Freeing Up Their Capacities

The flushing mechanism hides from the users all the details about deciding which data should be flushed, to which storage layer and at which point of the execution the flushing should take place. Such mechanism takes advantage of execution time information that is difficult to obtain at the development time of complex applications. For instance, data

dependencies, i.e., which data are not going to be required by any successor tasks in future executions. In addition to data locations, which will depend on the used I/O scheduler.

Algorithm 6 presents the pseudocode of the flushing data movement mechanism. $D$ is the set of output data of all executed tasks so far. For each data $d$, the pseudocode is checking if there are any tasks that will require $d$ by a call to the `hasReaders` routine (Line 3). If there are future task executions that have $d$ as input, then the rest of the routine is skipped and the next data is considered. Otherwise, if no future tasks require $d$, then the pseudocode retrieves the current storage layer on which $d$ is stored (Line 6). Then, each storage layer gets considered, starting from the bottom/lowest layer ($BottomLayerIndex$) to the layer directly below where the data currently resides ($currentLayer + 1$). Data $d$ will be flushed if and only if there is enough capacity on the layer $i$ to store it (Lines 9-14).

**Algorithm 6:** Flushing Mechanism

**Input :** $D$ as the set of data considered for flushing
**Output:** $D_{flush}$ as the set of data that is going to be flushed

```
1  Function FLUSH(D):
2      foreach d ∈ D do
3          if hasReaders(d) == True then
4              Continue
5          end
6          currentLayer ⟵ getCurrentLayer(d)
7          foreach i ∈ {BottomLayerIndex ··· currentLayer + 1} do
8              d_size ⟵ getSize(d)
9              if C_i >= d_size then
10                 add d to D_flush
11                 C_i ⟵ C_i − d_size
12                 currentLayer ⟵ currentLayer + d_size
13                 Continue
14             end
15         end
16     end
17     return D_flush
18 End Function
```

Because the flushing mechanism may involve moving large amounts of data, it should only be started when the applications are in their compute-dominant phase. This can be detected when there is no I/O activity at any point of application execution, i.e., no I/O tasks are running nor scheduled to run on any of the storage layers. Therefore, not negatively impacting applications performance by transparently overlapping computation with data movement.

Although the flushing mechanism can help achieve more I/O performance by increasing the available capacity of higher storage layers, it consumes time relative to the amount of the

data to be flushed. Therefore, using this mechanism should be done only with applications that alternate between compute and I/O phases to carry out the flushing operation during the compute phase and hide its cost. In addition to that, the application should spend enough time doing computations to hide the cost of flushing the data.

## 5.4 System Implementation

The runtime of PyCOMPSs, i.e., COMPSs is capable of abstracting the underlying infrastructures from applications code by accepting two resource description files at launch time: *resources.xml* which describes all the resources of the infrastructure, and a *project.xml* which describes the resources that will take part in applications execution. The COMPSs runtime uses these files to register the resources and builds a runtime resource object that represents each resource and its state.

During applications' execution, different components of the COMPSs runtime such as the Task Scheduler use the resource objects to make scheduling decisions and optimize applications execution. The next sections describe implementation details in COMPSs for supporting the storage heterogeneity awareness capabilities that were presented in the previous section.

### 5.4.1 Storage Devices Management

When PyCOMPSs applications are launched, the COMPSs master process is started. As shown in Figure 5.9, the master process is responsible, among other things, for loading the worker machines information from the resource description files and launch the COMPSs worker component.

Each storage devices described in the resource description files is represented by a *Storage* object. Storage objects contain all the details about the storage device such as type, capacity, storage bandwidth, mount directory, etc.

The COMPSs master and worker components take part in the management of the storage devices. The master component is responsible for taking global system-wide scheduling decisions, whereas the worker is responsible for managing local storage devices and ensuring correct tasks assignment to the requested storage device.

#### 5.4.1.1 COMPSs Master

Once the COMPSs master process has loaded and registered all the storage devices, each to a corresponding Storage objects, it pools them into a *Storage Devices* object. As shown in Figure 5.10, the Storage Devices object is a list in which all the storage devices are sorted according the manner described in Section 5.3.2; in a descending order starting from the storage device with the highest bandwidth to the storage device with the lowest bandwidth. If there is a tie in bandwidth, then the storage device with less capacity is placed on top of the storage device with more capacity. Therefore, it is guaranteed that the top storage layer has the highest bandwidth and least capacity, whereas the bottom layer has the least bandwidth and highest capacity.

Furthermore, the Storage Devices object is used whenever any information is required about the storage devices and their states, i.e., remaining capacities, current bandwidth, etc. Such information are required by different components of the COMPSs master such as the Task Scheduler and the Resource Optimizer.

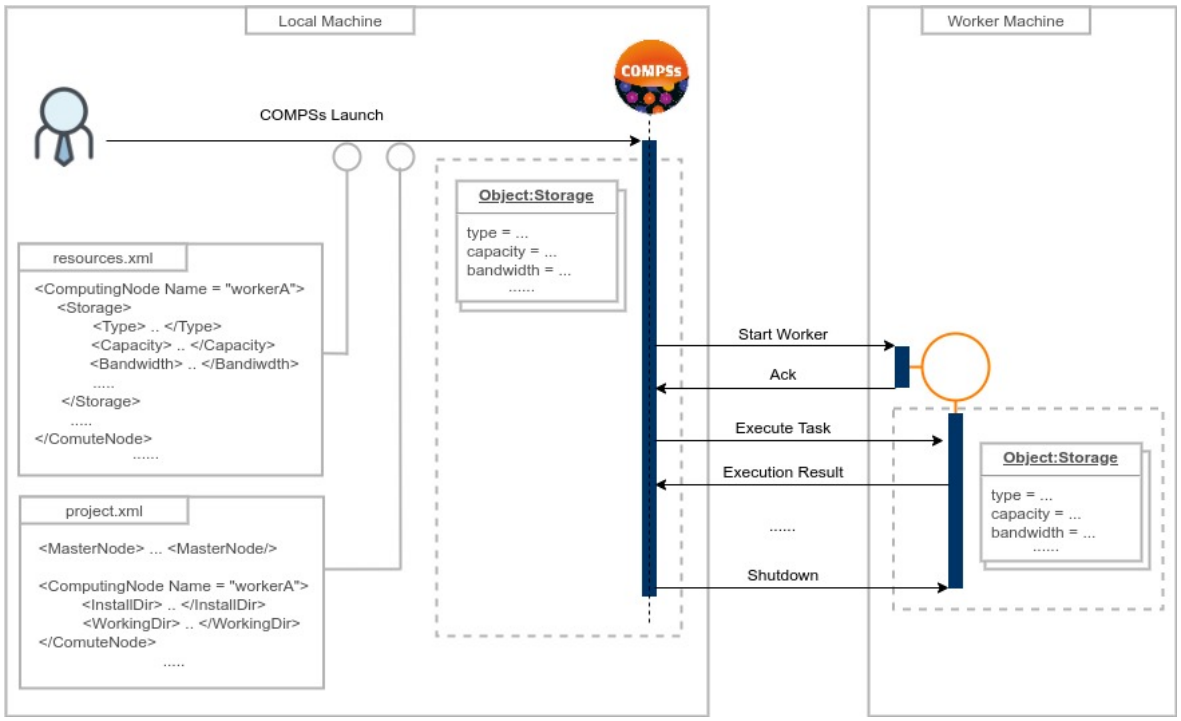



FIGURE 5.9: COMPSs Master And Worker Storage Management

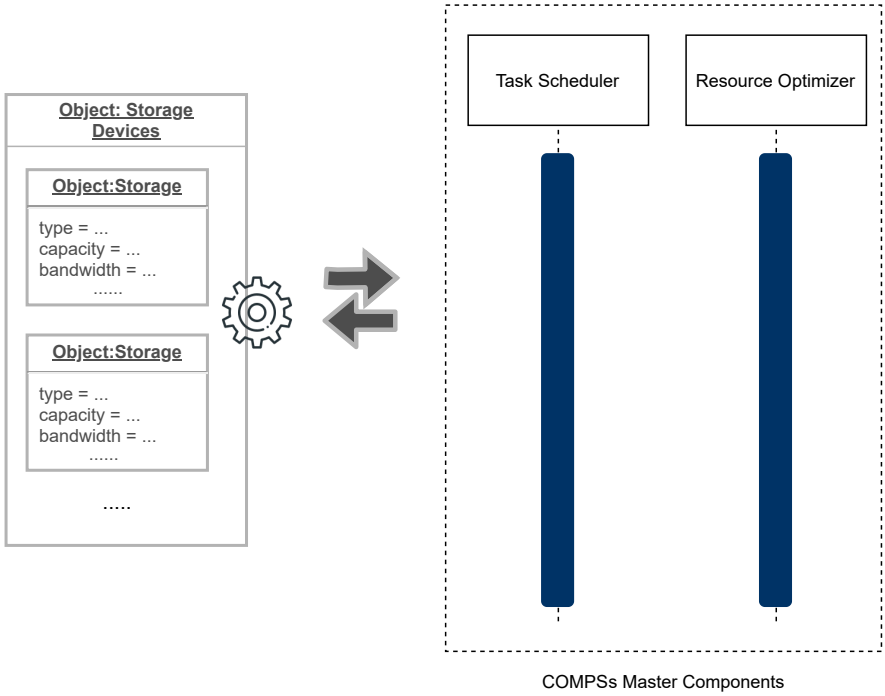



FIGURE 5.10: Storage Devices Management In COMPSs Master

#### 5.4.1.2 COMPSs Worker

Storage devices can be bound to different CPUs in the worker machines. In order to achieve the expected performance out of a certain device, it is important for the task that requests a device to be bounded to the correct CPU. Otherwise, I/O performance on that device can greatly deteriorate.

As illustrated in Figure 5.11, once the COMPSs worker has received an I/O task execution request from the master, it checks to which CPUs the requested storage device is bound, then it binds the process that is executing the task to the target CPU. When the task finishes execution, the COMPSs worker unbinds the process that has executed the task.

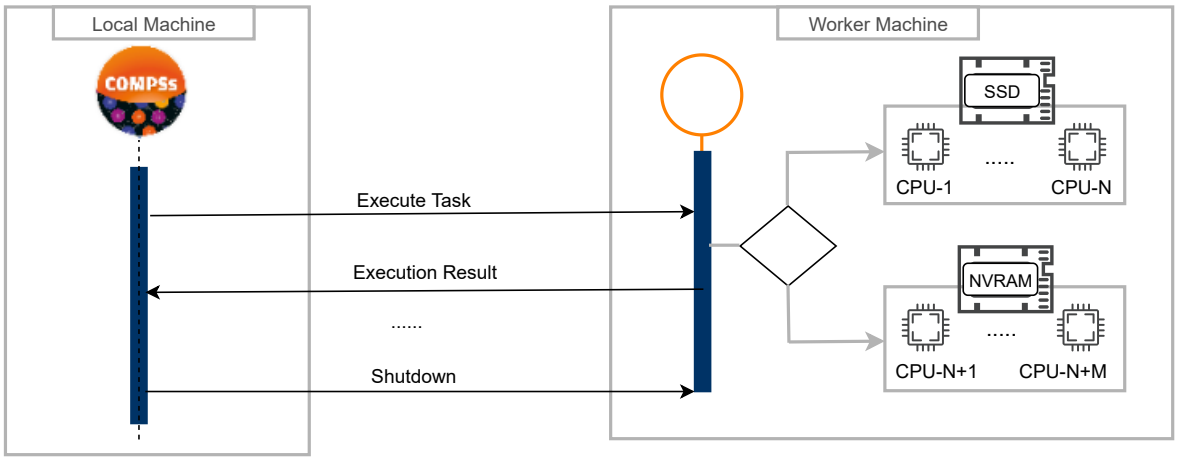



FIGURE 5.11: Storage Devices Management In COMPSs Worker

### 5.4.2 I/O Schedulers

PyCOMPSs allows users to choose the preferred I/O scheduler at application launch time. This is done through the use of the `io_scheduler` command-line argument of the PyCOMPSs launch command. Listing 5.1 shows a snippet of the PyCOMPSs launch command for executing a Python application on distributed infrastructures. Changing the scheduler does not require modifying or changing applications code. I/O tasks will be scheduled according to the policy of the desired scheduler.

```
1 # IOBackfillingScheduler -> Backfilling Scheduler
2 # IOPriorityScheduler -> Priority Scheduler
3 # FcfsScheduler -> FCFS scheduler
4 enqueue_compss
5     --io_scheduler=IOBackfillingScheduler
6     --num_nodes=10
7     my_app.py
```

LISTING 5.1: Example PyCOMPSs Launch Command With The Backfilling I/O Scheduler

All COMPSs schedulers extend the main scheduler component which is the Task Scheduler. The Task Scheduler is an abstract class that includes the functions that are common to all schedulers such as a `schedule` function responsible for scheduling the tasks and a `calculateScore` function which gives a score to all the available workers that can execute the task. Tasks are scheduled to workers with the highest score. Each worker score has

many attributes that determine whether a task should be executed on this worker or not. For instance, data locality score (i.e., whether a data transfer is needed) and storage layer score. A worker `X` has a higher storage layer than worker `Y` if the layer that the task is going to use in worker `X` is higher than worker `Y`. The scheduler determines the storage layer of the worker by using a scheduling function that implements Algorithm 1 described in Section 5.3.2.

When a task finishes execution, the scheduler gets notified and a new scheduling iterations starts if necessary. The compute scheduler is called to schedule the compute tasks and the I/O scheduler is called to schedule the I/O tasks. The type of the task that has just finished execution (i.e., compute or I/O) is not important when starting a new scheduling iteration. A new scheduling iteration starts for both the compute and I/O tasks whenever any task finishes execution.

### 5.4.3 Flushing Mechanism

As previously mentioned in Section 5.3.4, due to the big amounts of data involved in the flushing operation, it may take a lot of time. Therefore, the flushing mechanism should be started when no I/O tasks are running so that delays in I/O tasks execution would be avoided.

Therefore, we implemented the flushing mechanism in COMPSs by the use of helper processes in the master and worker components. In the master part of COMPSs, the Resource Optimizer component carries out, among other things, executing the flushing mechanism. Resource Optimizer is an independent process within the COMPSs framework, therefore, it is guaranteed that no other component of the COMPSs runtime would be burdened or blocked when the flushing mechanism is taking place.

Figure 5.12 illustrates the flushing mechanism. Once a task returns from execution, the scheduler notifies the Resource Optimizer that a task has finished execution. The Resource Optimizer implements the algorithm that is discussed in Section 5.3.4. For all the outputs of the returned task, the Resource Optimizer calls the `hasReaders()` function of the Task Analyser component. If there are no future tasks that require the data, and there are a bottom layer that can store the data, then it is added to a *flushCandidates* list. Then the Resource Optimizer checks with the scheduler if there are any running I/O tasks. If no I/O activity is taking place, then the Resource Optimizer sends the `flushCommand` to the appropriate worker. This command contains all the data that should be flushed and to which storage layer.

Similar to the COMPSs master implementation, in order to not burden the worker part with the extra work of the flushing mechanism and block other critical functionalities, the flushing mechanism is done by a helper process. When the worker receives a `flushCommand` from the master, it assigns this command to the helper process to handle it. After flushing all the data, the worker returns an acknowledgement message to the master.

It should be noted that the flushing mechanism is executed per worker. Whenever a task has finished execution on a certain worker, the flushing mechanism is carried out for that worker.

From users point of view, the flushing mechanism is enabled by setting the `data_movement` argument of the PyCOMPSs command-line launch command to `true` as shown in Listing 5.2 without modifying any line in the application code.

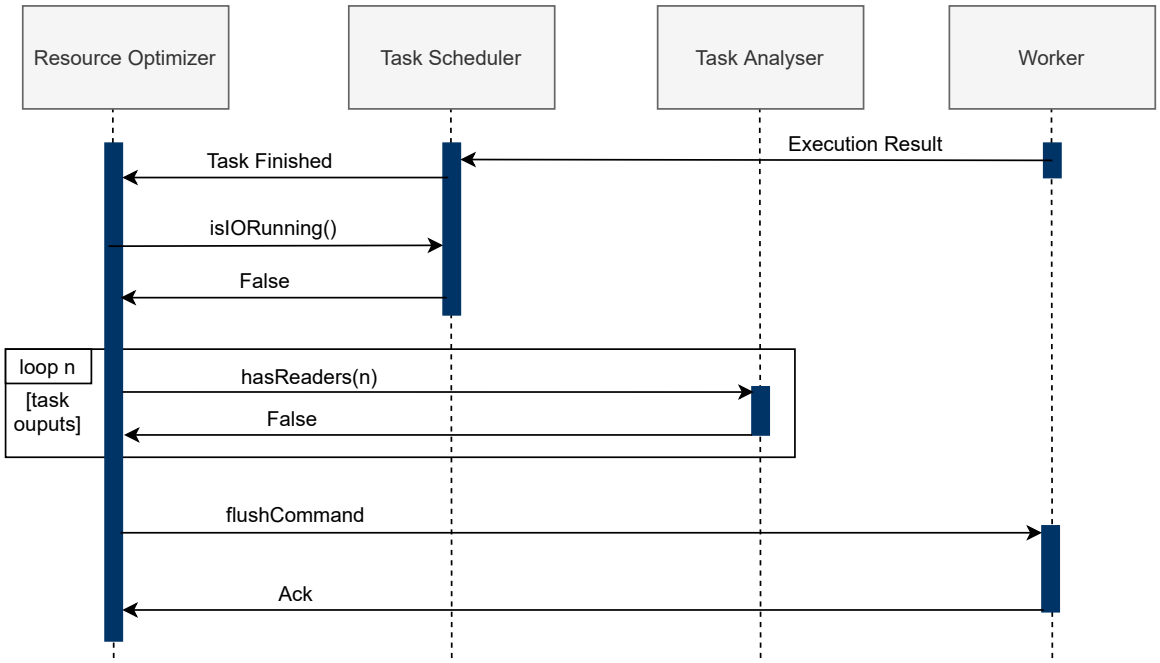



FIGURE 5.12: Flushing Mechanism In COMPSs

```
# default -> data_movement=false
enqueue_compss
    --io_scheduler=IOBackfillingScheduler
    --data_movement=true
    --num_nodes=10
    my_app.py
```

LISTING 5.2: Enabling The Flushing Mechanism In The PyCOMPSs Launch Command

## 5.5 Evaluation

In this section we show the performance improvements that can be achieved by evaluating the storage heterogeneity-aware PyCOMPSs prototype on different applications that exhibit different I/O workloads.

Section 5.5.1 starts with describing the infrastructure of the execution platform and its storage infrastructure. Then, we present a brief description of the applications used in the evaluation, their I/O workload characteristics and their performance results: Section 5.5.2.1 discusses the results with an application that exhibits homogeneous I/O workload. Whereas Section 5.5.2.2 presents the results of a real application that exhibits heterogeneous workload, in addition to a synthetic application to demonstrate the differences between the heterogeneous schedulers.

### 5.5.1 Infrastructure

We have used the MareNostrum CTE-Power of the Barcelona Supercomputer Center [33] as an execution platform to run our experiments. The hardware architecture of this system can be described as follows:

***MareNostrum CTE-Power*** is composed of 54 nodes that are incorporated into the bigger MareNostrum 4 supercomputer [66]. Each node has two Witherspoons processors, each with 20 cores. The MareNostrum CTE-Power cluster has 1 Terabyates of SSD scratch storage and two local NVMe devices with a total capacity of 6 TeraBytes. All nodes have access to a shared Hard Disk Drive (HDD) with a total capacity of almost 8 PetaBytes mounted with the IBM General Parallel File System (GPFS). We used the IOR benchmark [43] to measure the write bandwidth of the storage devices. At the time of running the experiments, the tool measured 6026 MB/s and 2743 MB/s of write bandwidth for the NVRAM and SSD devices respectively. Whereas the PFS measured 900 MB/s of write bandwidth.

It should be noted that we only used one SSD device, and also limited the capacity of the NVRAM layers to 1 Terabytes to suit the amount of data produced by our uses cases and to simulate a storage infrastructure where the scheduling decisions impact applications performance.

Table 5.1 summarizes the storage system capabilities that we used in the experiments.

| System | NVRAM | SSD | PFS |
|---|---|---|---|
| **MareNostrum CTE-Power** | 6026 MB/S | 2743 MB/S | 900 MB/S |

TABLE 5.1: MareNostrum CTE-Power Storage Layers Measured Bandwidth

### 5.5.2 Use Cases And Experiments

We implemented two different I/O intensive real applications with PyCOMPSs. Each application exhibits a different I/O workload which allows for evaluating the impact of the storage heterogeneity-aware capabilities in different scenarios. These applications are:

- *Checkpointing HMMER Application*: an application that produces *homogeneous I/O workload*. For a given input size, the checkpointing task writes the same amount of I/O every time it is called during the lifetime of the application.
- *Multi-References Sequence Alignment*: an application that produces *heterogeneous I/O workload*. There are different I/O tasks, each task produces different amount of I/O.

In all the experiments, the baseline experiment is the PyCOMPSs implementation that does not use any of the heterogeneity-awareness capabilities. This implementation uses only PFS to write applications data. We compare the baseline version against multiple experiments that used the heterogeneity-aware PyCOMPSs prototype. Each experiment is intended to show how the proposed I/O schedulers behave under certain I/O workloads. These experiments include:

1. Homogeneous I/O scheduler with FCFS policy.
2. Heterogeneous I/O scheduler with priority policy.
3. Heterogeneous I/O scheduler with backfilling policy.
4. Using the flushing mechanism with the scheduler that achieved highest performance.

At application launch time of the heterogeneity-aware PyCOMPSs experiments, the COMPSs runtime loaded the information about available storage devices and their capabilities in order to organize them in a hierarchical view from the device with highest bandwidth to the

device with lowest bandwidth. The top storage layers cluster are used as first-choice fast buffering layers before scheduling any task to the lowest layer (i.e., the PFS).

In all experiments, constraints are used to specify the required bandwidth and capacity of the I/O tasks. For the purpose of measuring the impact of our proposals on the I/O and total performance, we set the required bandwidth to be equal to the required capacity for all I/O tasks. Indeed, other techniques can be used to determine the required bandwidth and/or capacity similar to the techniques that are described in Section 4.4.3 of Chapter 4.

All the experiments were run on 12 worker nodes plus one node dedicated as master node. The master node runs the master component of the PyCOMPSs runtime and manages the execution without taking part in any computation. In addition to that, all experiments were run 10 times and the average results are reported.

#### 5.5.2.1 Checkpointing HMMER Application

We use the HMMER application that was previously described in Section 4.5.3.1. The HMMER application takes as input a sequence database and a sequence file. It splits the sequence and database to fragments, then calls the HMMER tool [27] for each combination of sequence and database fragment, finally it gathers the resulting sequence fragments into one file. The PyCOMPSs implementation of the application has three phases of computation-I/O-computation. In the first computation phase, a *hmmpfam* task is called for each input of database fragment and sequence fragment. Every *hmmpfam* task is followed by a *checkpointFrag* task which checkpoints the resulting fragment. Finally, database fragments are gathered by *gatherDB* tasks and resulting sequence fragments are gathered by *gatherSeq* tasks.

The experiments of this application used as inputs a 64.5 GB HMM protein-families database (pfam) and a sequence file that contains 140,942,208 sequences with a total size of 50 GB. Databases and sequence files are available for public use in the European Bioinformatics Institute (EMBL-EBI) servers [30] which hosts up-to-date sequences, databases and software widely used by academics and life science researchers.

Each of the 12 worker nodes used in this experiment processed 48 sequences. The application split each sequence file to 96 fragments. Each *checkpointFrag* writes 400 MB.

The performance results of the application on the CTE-Power cluster is presented in Figure 5.13. All the experiments that use the heterogeneity-aware capabilities of PyCOMPSs achieve drastic performance improvement over the baseline experiment in both I/O time and total time. As the system is able to take advantage of the heterogeneity of the storage infrastructure, improvements can reach up to almost 3x I/O performance speedup and 44% total time improvement with the FCFS scheduler and flushing experiment. This performance benefit is possible because I/O tasks are first scheduled to storage layers that provide high bandwidth, i.e., NVRAM and SSD. Thus, task parallelism increases because more I/O tasks can run concurrently. Hence, performance improvement is achieved. Once the faster storage layers does not have enough bandwidth or capacity, tasks are scheduled to the PFS.

Taking a closer look at Figure 5.13, although the priority and backfilling schedulers achieve I/O time and total time improvement over the baseline experiment, they produce similar results compared to the FCFS scheduler. As the I/O tasks of this application always write the same amount of data, the benefit of using the backfilling and priority is not apparent. Moreover, these schedulers produce a slight overhead in the I/O time and total time due to the extra operations that they perform such as sorting tasks and carrying out the execution time checks to optimize the usage of the faster storage layers.

Furthermore, continuing with Figure 5.13, it can be noted that enabling the flushing mechanism with the FCFS scheduler achieves the best I/O performance and total time. This can be explained because the flushing mechanism continuously frees up capacity in higher

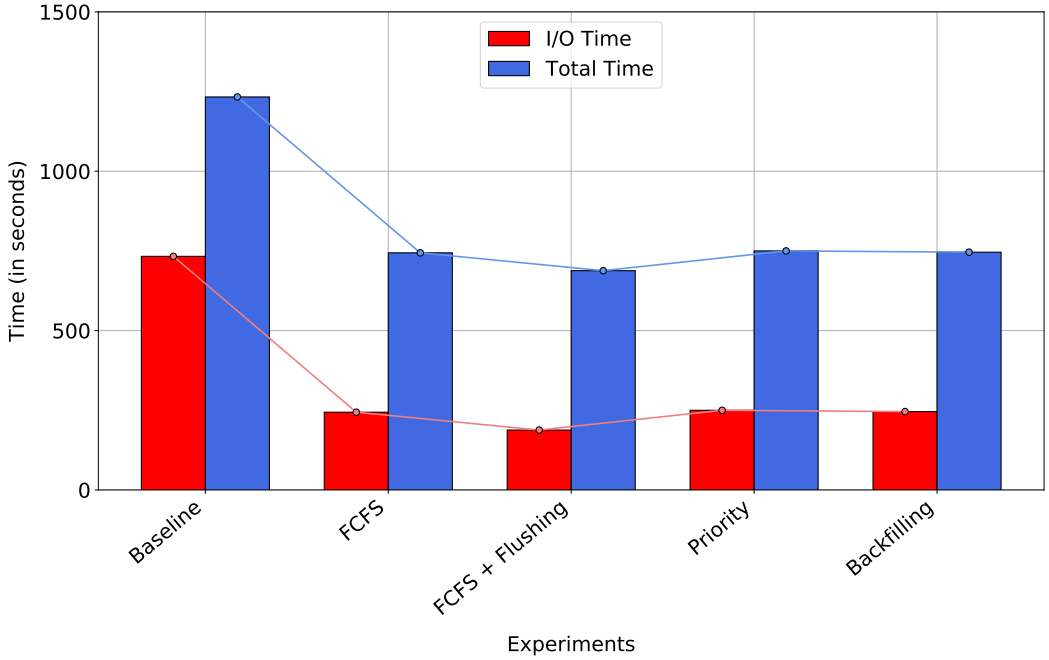



FIGURE 5.13: Performance Results Of The HMMER Application On The CTE-Power Cluster

storage layers by flushing the obsolete data to bottom layers. Due to the compute-I/O pattern of this application, the associated overhead is hidden and does not negatively affect the performance. The flushing process is launched during the compute intensive phases when no I/O tasks are running, hence, avoiding degrading I/O and total performance.

#### 5.5.2.2 Multi-References Sequence Alignment

In the fields of life sciences and bioinformatics, short sequences (called reads) are often aligned to multiple reference genomes to identify how much does a reference represent a species or to find regions of similarities which helps identifying to which species the sample under investigation belongs. The output of this operation varies in size depending on how much the sequence file matches a certain reference or chromosome. We developed a PyCOMPSs application that aligns the input sequences to two genomes and writes the alignment results to separate files. Figure 5.14 shows the skeleton of the task dependency graph of the application. It consists of the following tasks: two alignment tasks (*align_ref1*, *align_ref2*); each task aligns the input sequence to a different genome. Each alignment task is then followed by an I/O task that writes the alignment output to a separate file (*write_res1*, *write_res2*).

Both alignment tasks used the BWA tool for short reads alignment [58]. The used input files and reference genomes are publicly available on the National Center for Biotechnology Information (NCBI) servers [76] which is a well-known resource for downloading genomic sequences. We replicated sequence files to make each worker node process 2400 sequence files, each file has a size of 79 MB in compressed gzip format. All input sequence files are aligned against two different builds of the human genome reference: HG19 and HG38 [46], each of 38 GB of size. The inputs are processed in iterations, an alignment phase followed by a writing phase. From previous experiments of this application, it is expected that the amount of data to be written by *write_res1* and *write_res2* are 200 MB and 800 MB respectively. The total size of data produced by this application is almost 3 Terabytes.

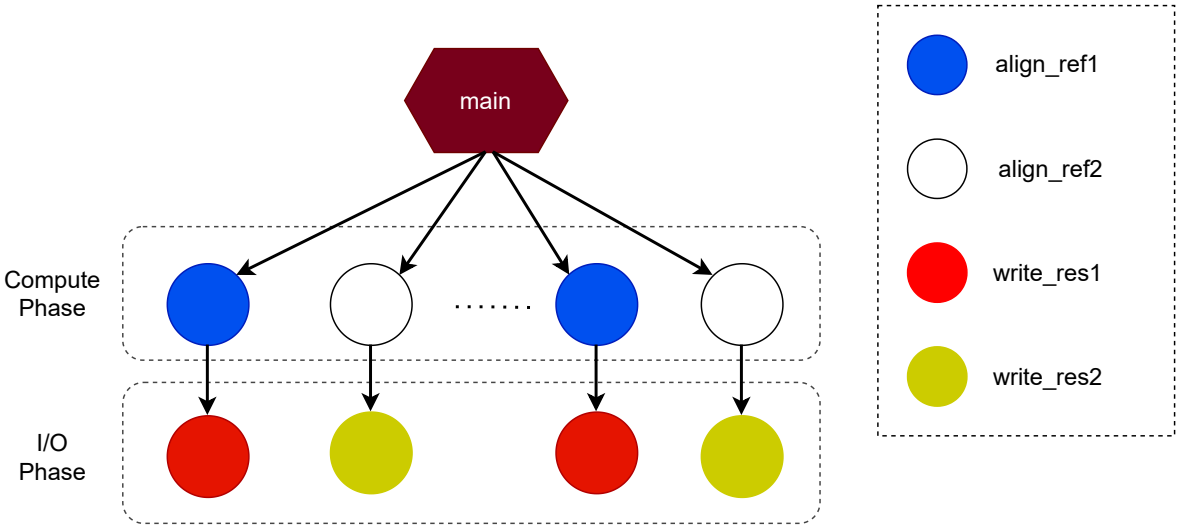



FIGURE 5.14: Task Graph Skeleton Of The Multi-References Sequence Alignment PyCOMPSs Application

Figure 5.15 shows the performance results of the application on the CTE-Power cluster. Similar to the results of the previous application, the heterogeneity-aware experiments achieve a significant performance improvement that reaches up to 5x I/O performance speedup and up to 48% total time improvement when using the backfilling scheduler with flushing. The heterogeneity-Aware PyCOMPSs implementation can take advantage of the higher bandwidth storage layers of the underlying storage infrastructure to increase I/O task parallelism. Therefore, increasing the I/O performance and total performance of the application.

However, unlike the results of the previous application, it can be noted that the priority and backfilling schedulers achieve better I/O performance speedup compared to the FCFS scheduler. This can be explained because this application has heterogeneous I/O workloads. Therefore, the priority and backfilling schedulers are able to exploit the underlying storage layers to maximize I/O performance by prioritizing the scheduling of critical I/O tasks with higher bandwidth demands to higher storage layers. On the contrary, the FCFS scheduler schedules the tasks in order of their arrival to the scheduler. Therefore, less critical I/O tasks (i.e., with lower bandwidth demands) fill the bandwidth and consume the capacity of the higher storage layers.

A closer look at Figure 5.15 shows that the backfilling scheduler is achieving better performance than the priority scheduler. This can be explained by the behaviour of each scheduler. The priority scheduler stops scheduling when there are no enough resources to schedule the task under consideration. Whereas using the backfilling scheduler, if a task cannot be scheduled because there are no enough resources, subsequent tasks are scheduled if they do not delay launching the waiting task. Hence, the backfilling scheduler advances tasks execution and maximizes resource utilization.

Furthermore, continuing with Figure 5.15, it can be noted that using the flushing mechanism with the priority and backfilling scheduler achieves better I/O performance and total performance. The flushing mechanism continually frees up the capacity of the storage layers. Hence, critical I/O tasks can be scheduled to higher storage layers and task parallelism is increased.

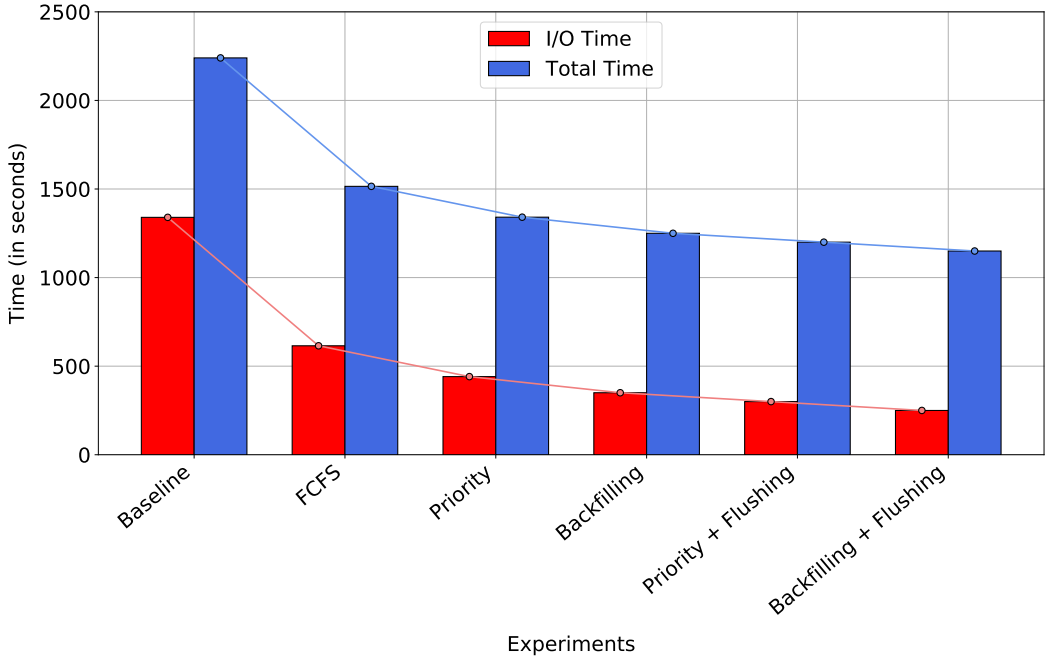



FIGURE 5.15: Performance Results Of The Multi-References Sequence Alignment Application On The CTE-Power Cluster

#### 5.5.2.3 Synthetic Heterogeneous I/O Workload

In order to understand the difference in performance between the priority and backfilling schedulers, we launched multiple experiments with a synthetic application. This synthetic application mimics the pattern of the Multi-References Sequence Alignment Application as illustrated in Figure 5.14, i.e., interchanging iterations of computations and I/O. The application launches 2400 compute tasks followed by the same number of I/O tasks. The compute tasks generate a certain amount of data, whereas the I/O tasks write the data that has been generated to a separate file. Half of the I/O tasks writes 800 MB of data (the same amount of data written by *write_res2* in use case 5.5.2.2), while the other half of I/O task writes 500 MB (bigger size than the data written by task *write_res1* in use case 5.5.2.2).

Figure 5.16 shows the I/O time and total time of the heterogeneous I/O schedulers (i.e., priority and backfilling) on the CTE-Power cluster. Unlike the previous use case (5.5.2.2), the backfilling scheduler has a worse I/O and total performance than the priority scheduler. Such behaviour is the result of increased data sizes. On the one hand, the backfilled I/O tasks consume more capacity of the higher storage devices. Therefore, the capacities of higher storage layers become full and critical tasks are scheduled to lower storage layers. Therefore, task parallelism decreases and performance degrades. On the other hand, because the priority scheduler schedules tasks strictly according to their bandwidth, the capacities of higher storage layers are persevered for the execution of more critical tasks.

Also in Figure 5.16, it can be noted that using the flushing mechanism, the backfilling scheduler returns to outperform the priority scheduler. As data are periodically flushed and the capacities of higher layers are continuously freed, the application can take advantage of their high bandwidth to schedule more critical tasks while backfilling less critical tasks.

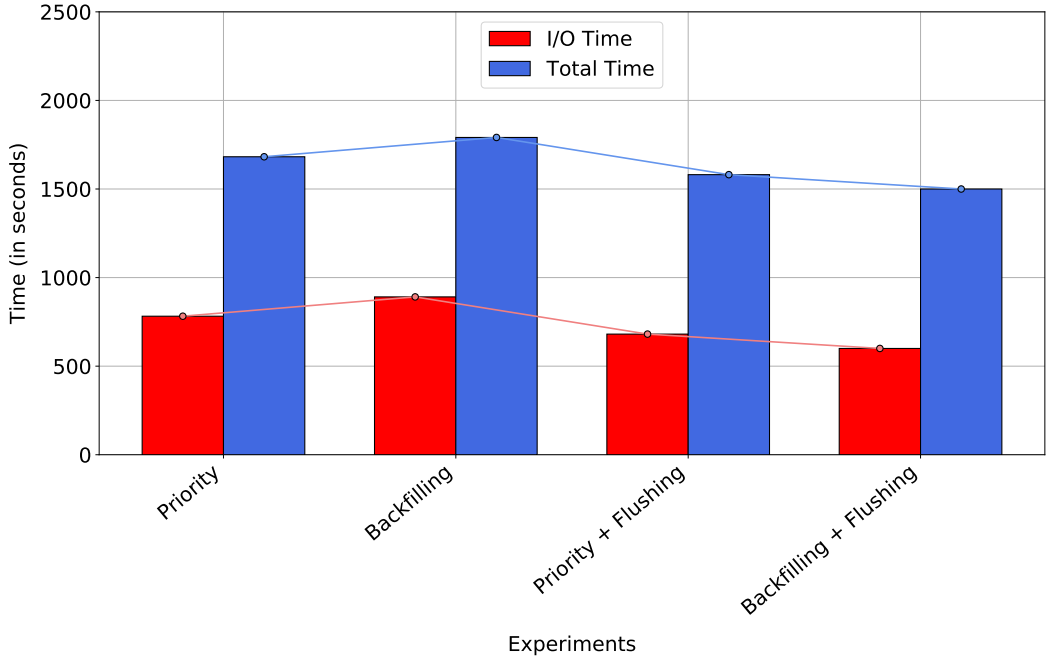



FIGURE 5.16: Performance Results Of The Heterogeneous I/O Schedulers On The CTE-Power Cluster

## 5.6 Discussion

This chapter presented proposals to take advantage of the heterogeneous design trend of modern storage systems with storage-heterogeneity aware task-based programming models. Thus, improving applications I/O performance and total performance.

Storage heterogeneity-aware task-based programming models are capable of optimizing I/O performance for applications that follow an I/O intensive compute-I/O patterns. A storage heterogeneity-aware system is defined by the following capabilities:

- First, abstracting the storage system heterogeneity and exposing it as a resource pool where priority is given to storage devices that provide higher bandwidth.
- Second, supporting dedicated I/O schedulers that offer different policies to optimize executions of various I/O workloads.
- Third, a data movement flushing mechanism that periodically frees the capacity of higher storage layers, hence, maximizing their utilization.

We implemented a prototype of these capabilities in the PyCOMPSs programming model. Our experiments on the MareNostrum CTE-Power cluster showed that the prototype is able to achieve significant performance improvement for two real-world applications. The First Come First Served (FCFS) scheduler is able to optimize I/O performance when the I/O workload is constant throughout applications lifetime. Whereas the priority and backfilling schedulers achieve better I/O performance in applications that have variable I/O workloads.

Applications performance differs when using the priority and backfilling schedulers depending on the input sizes. As input sizes increase, the priority scheduler tend to give more performance than the backfilling scheduler. This is due to the strict behaviour of the priority scheduler that helps to preserve the capacity of higher storage layers to schedule a higher

number of critical I/O tasks. On the other hand, the backfilling scheduler backfills less critical I/O tasks while more critical tasks are pending resources. Therefore, it outperforms the priority scheduler when the data sizes of less critical I/O tasks is relatively small. As the data sizes of these tasks increase, the performance diminishes until it underperforms the priority scheduler.

With all schedulers, the flushing mechanism has proved to maximize I/O performance. Such mechanism is launched only when no I/O activity is detected to hide the overhead of moving the data between storage layers. The time spent during the flushing mechanism is relative to the sizes of the data to be flushed and how many storage layers exist in the storage infrastructure.

# Chapter 6

# Hybrid Programming Models for Programmability and Performance

## SUMMARY

For many years, the Message Passing Interface library (MPI) has been used to improve applications performance by taking advantage of fine-grained performance improvement opportunities. These opportunities have been made possible by many-core architectures of nowadays computing machines. However, the increasing demand for performance alongside with the increased complexity in modern computing systems has given rise to hybrid programming models that aim for maximum performance. Hybrid parallel programming models combine one or more programming models to try to exploit more performance out of the underlying execution infrastructures. Nevertheless, the performance gain is accompanied by increased programming complexity.

Task-based parallel programming models offer high-level abstractions necessary to develop efficient applications to be executed on distributed infrastructures without sacrificing programmability nor portability. However, a lot of opportunities for fine-grained parallelism is not addressed. Therefore, wasting performance improvement opportunities.

This chapter focuses on solving and answering research question $\mathbf{Q}_3$ that is concerned with a twofold objective: exploiting different levels of parallelism to achieve more performance without increasing the complexity of applications programming. More specifically, this chapter proposes a hybrid programming model of tasks and MPI executed inside these tasks. Such programming model can use tasks to achieve coarse-grained parallelism and MPI to exploit a finer-grained level of parallelism inside the tasks. Therefore, improvement can be reached at task performance that would be reflected in the total performance. Such a proposal can be used to enable parallel I/O inside distributed executed tasks. Moreover, it can be used to parallelize the execution of tasks in compute-intensive applications.

In this chapter, we realize this proposal by introducing a hybrid programming model of the PyCOMPSs task-based model and MPI. This hybrid programming model enables application developers to parallelize the execution of PyCOMPSs tasks using MPI. Throughout this chapter, we refer to this type of tasks as *Native MPI Tasks*. These tasks execute part of the application code (i.e., task code) in parallel with MPI and they are handled in a similar manner to sequential tasks in terms of dependency detection, scheduling and inputs/outputs handling.

The evaluation section of this chapter shows that without compromising applications programmability, using *Native MPI* tasks in PyCOMPSs offers significant performance improvement when compared to the sequential implementation of the tasks. This performance improvement can reach up to 1.9x improvement in total performance for an I/O intensive application and up to 3x improvement in total performance for a compute intensive application.

## 6.1 Overview

Current parallel platforms are able to offer unprecedented levels of performance, with future platforms are projected to offer even more performance. These systems are characterized by their many-core architectures that can accommodate an increasing number of cores on a single chip. This increasing on-chip scaling is accompanied by an increase in the number of computing nodes. Nowadays systems consist of thousands of processing nodes, with future systems are expected to have more of them.

For many years, MPI [72] has been the de-facto standard for parallel programming. It is widely used to parallelize applications to get higher performance out of many-core architectures. Both compute intensive and I/O intensive applications use MPI to parallelize critical parts of their execution. For instance, MPI-I/O [89] is used to improve I/O performance by enabling parallelization and access patterns optimization. In addition to that, many I/O optimization libraries such as HDF5 [105] and NetCDF [59] leverage MPI-IO to provide high-level I/O optimization abstractions.

Due to the increased distribution and architectural heterogeneity of modern systems, hybrid programming models of **MPI + X** (where X is another parallel programming model) has been proposed and used by the community to achieve more parallelism (See Section 6.2 for more details). However, due to the low-level APIs provided by these programming models, the details of the underlying infrastructure is exposed to application developers. Additionally, in order to achieve the desired performance, experience in performance optimization, parallel programming and distributed computing is required, which may not be common for inexperienced users such as field experts and domain specific scientists.

The growing complexity and heterogeneity of these programming models and computing systems hinder the development of parallel applications from fully exploiting the capabilities of the underlying systems. Application developers have to handle the complexity of performance scalability and the details of the underlying hardware being exposed to applications.

In recent years, task-based programming models offered developers a high-level abstractions to execute applications on distributed systems by decomposition of the application into work units, called *tasks*. Encapsulating applications work into tasks is of great importance to cope with the heterogeneity of the underlying infrastructures. Therefore, task-based programming models have gained popularity as means of extracting high performance from complex infrastructures without exposing low-level details to application developers [29]. All the details of resource management, data transfers, monitoring execution, and work distribution is handled by the task-based programming model.

Even though task-based programming models offered a users-friendly high-level approach for applications execution on large-scale distributed infrastructures, a lot of finer-grained performance improvement opportunities are possible through the use of programming libraries such as MPI.

This chapter introduces the novel hybrid programming model of **Task-based programming model + MPI**. Such programming model offers the necessary abstraction to simplify applications development for large-scale distributed execution, in addition to exploiting the performance of fine-grained processing. This hybrid model enables the execution of MPI code inside tasks as opposed to executing external binaries or scripts with MPI. We refer to such tasks as: *Native* MPI tasks.

In this chapter, we describe the implementation details of our proposal in the PyCOMPSs programming model. Enabling MPI code execution in PyCOMPSs tasks allows the deployment of large workflows that use MPI code in tasks to exploit different levels of parallelism in applications. Therefore, improving applications performance.

The evaluation section of this chapter presents the performance results of our proposal on I/O and compute intensive applications. Using our proposed programming model enables up to 5x speedup in I/O performance and up to 1.9x improvement in the overall execution time in an I/O intensive application. Furthermore, it offers up to 3x improvement in overall execution time in a compute intensive application.

The rest of this chapter is organized as follows: Section 6.2 presents the related work. Section 6.3 describes the details of the hybrid programming models of tasks and MPI. The architectural design of this programming model is presented in Section 6.4. Section 6.5 discusses the evaluation results. Finally, Section 6.6 summarizes our conclusions.

## 6.2 Related Work

Task-based parallel programming models can be classified according to the level of granularity they are trying to achieve. Programming models such as OpenMP [19], Cilk [57], and the Intel TBB framework [92] are characterized by fine-grained parallelism. For instance, a parallel loop can be implemented as a set of tasks each corresponding to one or more loop iterations. OmpSs [26] is another fine-grained programming model that allows the specification of tasks by annotating the code with data directionality clauses that specify the accessed data and how it will be used (read, write or read and write).

Other task-based parallel programming models target orchestrating workflows at a higher level of granularity. Ruffus [38] offers an explicit syntax to define computational pipelines. COSMOS [55] offers an alternative for implementing workflows using the Map-Reduce paradigm [20]. Unlike these frameworks and systems, PyCOMPSs allows more flexibility in expressing parallelism details since it does not enforce a specific paradigm.

In recent years, hybrid parallel programming models have gained popularity for achieving more performance on modern computing infrastructures. Different combinations were discussed in previous work [23, 92]. For instance, a widely-used hybrid model is the one combining MPI and OpenMP. In this model, OpenMP threads perform compute-intensive work on local data on each node whereas MPI is responsible for the communication between processes. This model is particularly popular because it takes advantage of the hardware hierarchy of many-core systems. Indeed, OpenMP can be used to parallelize an application into tasks. However, OpenMP creates fine-grained tasks whereas PyCOMPSs creates coarse-grained tasks that are executed in distributed systems.

Spark-MPI [65] is an attempt to enable the integration of MPI with the Spark framework [119] by using a middle-ware to interface between Spark and existing MPI libraries. It uses a process management interface to execute MPI operations among Spark workers. This platform can execute MPI applications *or* Spark applications. However, the proposal of this chapter is to support the specification, interaction and execution of sequential tasks and tasks executed by MPI, both types of tasks can be in the same application.

Rabenseifner et al. [91] discussed the different ways of using the hybrid parallel programming model combining MPI and OpenMP on a hierarchical hardware structure and their potentials and challenges. Dinan et al. [24] explored a hybrid programming model that combines MPI and Unified Parallel C (UPC) and demonstrated its improvements on the scalability of locality-constrained UPC codes. Jacobsen et al. [49] presented MPI-CUDA implementation to exploit parallelism in multi-GPU clusters and investigated strategies to improve the efficiency and scalability of the execution on these infrastructures.

Previous attempts were made to improve the interoperability between MPI and Task-based parallel programming models. Sala et al. [94] presented an API to improve the interoperability between MPI communication primitives with OmpSs. This attempt targeted fine-grained parallelism in shared memory systems. Also, Marjanović et al. [67] presented

an approach for overlapping communication and computing by using a hybrid approach of MPI and a task-based shared memory programming model. However, the proposal of this chapter targets increasing parallelism levels of coarse-grained tasks in distributed infrastructures.

## 6.3 Native MPI Tasks

A hybrid programming model that is able to combine two different execution contexts, a coarse-grained task-based execution context and fine-grained MPI execution is possible through providing the following main abstraction: Native MPI Tasks.

Using Native MPI tasks, tasks can be designed and coded similar to any MPI program. For instance, MPI semantics and APIs can be used inside tasks such as `MPI_Init`, `MPI_Send` and `MPI_Receive`, etc. In addition to that, Native MPI tasks are executed by multiple MPI parallel processes without affecting any the execution of any other task in the application.

The main idea of this programming model is to encapsulate MPI execution inside Native MPI tasks. Such that the same application source files can have tasks that are executed sequentially and tasks that use the MPI API and are executed by multiple MPI parallel processes.

Figure 6.1 illustrates the concept of Native MPI tasks in a task-based execution. The *nativity* aspect of Native MPI tasks is due to a twofold reason:

i Native MPI tasks execute task code, instead of calling and executing an external binary file that is not part of the application code base. Therefore, Native MPI tasks can use the libraries that have been imported in the source file, and they have access to shared and global variables that were created outside the task.

ii Native MPI tasks interact with other tasks (sequential tasks or other Native MPI tasks) in a transparent manner. In terms of dependencies, Native MPI tasks can have multiple successors or predecessors of sequential tasks or other Native MPI tasks. All data interactions (i.e., receiving inputs, producing outputs) between different types of tasks are done transparently similar to sequential-sequential tasks data management.

All inputs to Native MPI tasks are accessible to all the MPI processes launched by this task. In addition to that, similar to MPI code execution, the output of Native MPI tasks is a list that contains the return value of all the MPI processes that have participated in the execution of the tasks.

The next sections describe the details of Native MPI tasks in the PyCOMPSs programming model: defining Native MPI tasks, specifying the number of MPI processes, etc. In addition to describing the behaviour of Native MPI tasks during application execution.

### 6.3.1 Programming Model Annotations

As previously discussed in Section 3.1.1 of Chapter 3, the PyCOMPSs programming model enables the parallelization of sequential code by the means of Python annotations or decorators. Therefore, following task declaration conventions of the PyCOMPSs programming model, a method is declared as a Native MPI task by the means of the `@mpi` annotation. This annotation must contain certain parameters to configure the MPI runtime environment, such as:

- MPI Runner: Path to the MPI executor to use. For instance, `mpirun` or `mpiexec`, specified by the `runner` argument of the decorator.

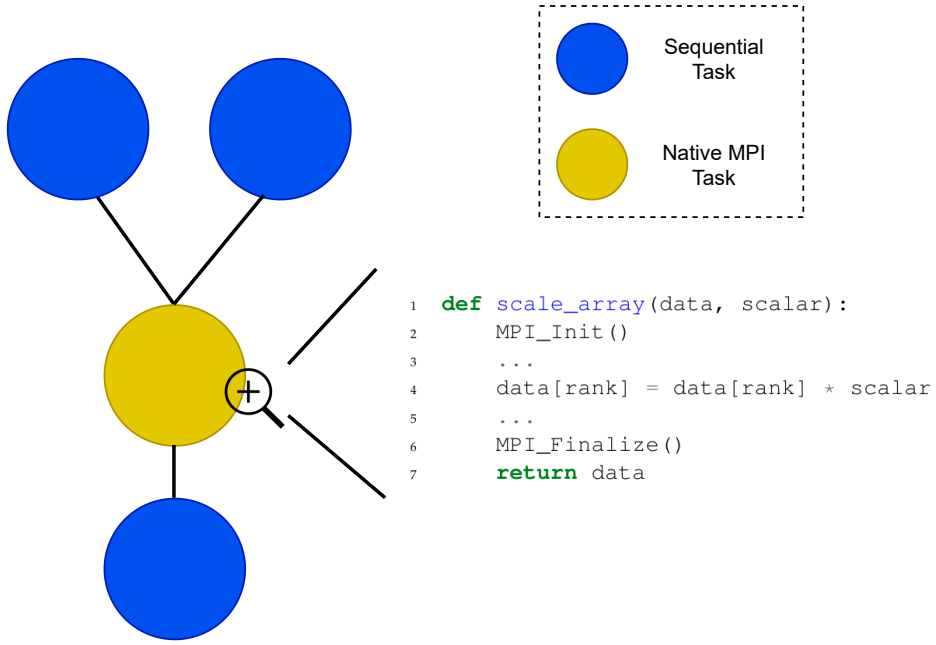



FIGURE 6.1: A Sample Task Execution Graph With Native MPI Task

- The number of MPI processes that are going to execute the task. This number is specified using the `processes` argument of the MPI decorator.

Notice that the `@mpi` annotation should be placed on top of the task annotation (i.e., `@task`) that annotates the target object or class method. Listing 6.1 shows an example Native MPI task defined in PyCOMPSs.

```
@mpi(runner="mpirun", processes=20)
@task()
def native_mpi_task(data):
    # perform parallel operations on data
    ...
```

LISTING 6.1: Native MPI Task Annotation

The `@mpi` annotation also provides other arguments such as `scale_by_cpu`, which can be used together with the `computing_units` argument of the `@constraint` decorator to scale the number of MPI processes by the number of computing units. Listing 6.2 shows an example of using the `scale_by_cpu` argument. The task in Listing 6.2 will be executed by 6 MPI processes.

```
@constraint(computing_units=2)
@mpi(runner="mpirun", processes=3, scale_by_cpu=true)
@task()
def native_mpi_task(data):
    # perform parallel operations on data
    ...
```

LISTING 6.2: Scaled Native MPI Task Annotation

The default MPI process placement behaviour is to fill the computing nodes one by one such that each MPI process uses a single CPU in the computing node. Other process placement policies can be specified by the use of environment variables supported by the MPI environment.

Listing 6.3 depicts an example PyCOMPSs application. This application contains the following tasks:

i A `generate_seed` sequential task that returns a number (Lines 1-4).

ii A `return_scaled_rank` Native MPI task that launches 4 MPI processes. Each MPI process returns its rank multiplied by the seed generated by the `generate_seed` task (Lines 6-13).

Looking closely at Listing 6.3, it is the same as the sequential version of the application except for a few additional lines of code that are the PyCOMPSs annotations and the synchronization API (Line 21). The `return_scaled_rank` task code is the same as a vanilla Python MPI application. Line 9 loads the Python MPI library necessary for using MPI API. Line 11 identifies the rank of each MPI process and Line 13 returns each rank multiplied by the input seed.

```
1  @task(returns=int)
2  def generate_seed():
3      # return a certain number
4      return 10
5
6  @mpi(runner="mpirun", processes=4)
7  @task(returns=list)
8  def return_scaled_rank(seed):
9      from mpi4py import MPI
10
11     rank = MPI.COMM_WORLD.rank
12
13     return rank * seed
14
15 if __name__ == "__main__":
16
17     arr = generate_seed()
18     scaled_ranks = return_scaled_rank(arr)
19
20     # wait for computation and retrieve scaled_ranks result
21     scaled_ranks = compss_wait_on(scaled_ranks)
22
23     print scaled_ranks
```

LISTING 6.3: Example PyCOMPSs Application With Native MPI Task

Figure 6.2 shows the execution output of the application in Listing 6.3. The output of the application is a list that contains the return value of each MPI process. PyCOMPSs applications are launched using the scripts: `runcompss` for local executions and `enqueue_compss` for large-scale executions.

```
[  INFO] Inferred PYTHON language
[  INFO] Using default location for project file: /opt/COMPSs//Runtime/configuration/xml/projects/default_project.xml
[  INFO] Using default location for resources file: /opt/COMPSs//Runtime/configuration/xml/resources/default_resources.xml
[  INFO] Using default execution type: compss

----------------- Executing native_mpi_exp.py -------------------------

Picked up _JAVA_OPTIONS: -Djdk.net.URLClassPath.disableClassPathURLCheck=true
WARNING: COMPSs Properties file is null. Setting default values
[(481)    API]  -  Starting COMPSs Runtime v2
[0, 10, 20, 30]
[(4667)    API]  -  Execution Finished

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

FIGURE 6.2: Example PyCOMPSs Application With Native MPI Task: Execution Log

### 6.3.2 Execution Time Behaviour

As previously mentioned, Native MPI tasks use the `processes` argument of the `@mpi` PyCOMPSs decorator to specify the number of MPI processes required for task execution. The scheduler does not launch Native MPI tasks unless there is an enough number of CPUs to satisfy their requirements. The effect of this behaviour is discussed in more details in the evaluation section (Section 6.5).

When PyCOMPSs applications are launched, the COMPSs runtime is responsible for performing many operations such as detecting tasks dependencies, tasks scheduling, execution monitoring, data transfer between tasks and retrieving tasks results to the master node. However, tasks encapsulate their code execution, i.e., once the task is being executed, the COMPSs runtime is not involved in the task logic that is being executed. The logic of the task is carried out by a Python process if it is a sequential task or parallel MPI processes if it is a Native MPI task.

Native MPI tasks of PyCOMPSs allow all MPI operations to be performed inside the task as if users were designing and programming a vanilla MPI application. For instance, creating communicators, splitting communicators, message passing between different processes, collective operations, etc. Nevertheless, Several running Native MPI tasks in a PyCOMPSs application cannot exchange messages between their MPI processes.

Different Native MPI tasks are completely independent of each other in terms of scheduling, MPI environment configuration and execution. Each Native MPI task has its own MPI configuration and execution environment. For instance, number of MPI processes, MPI runner type, etc.

Listing 6.4 shows an example PyCOMPSs application that has two Native MPI tasks. Both tasks perform MPI parallel operations on the input data that they receive from a predecessor sequential task (i.e., `generate_data`). Each Native MPI task has its own `processes` argument, hence, number of MPI processes. The task graph of this application is illustrated by Figure 6.3. At execution time, each Native MPI task has its own MPI execution environment and communication is not possible between the different MPI processes of different Native MPI tasks.

```python
@task(returns=list)
def generate_data():
    ...

@mpi(runner="mpirun", processes=4)
@task(returns=list)
def calculate_mean(data):
    # Calculate the mean in parallel
    ...

@mpi(runner="mpirun", processes=2)
@task(returns=list)
def calculate_std_dev(data):
    # Calculate the standard deviation in parallel
    ...

if __name__ == "__main__":
    ...
    data = generate_data()
    mean = calculate_mean(data)
    std_dev = calculate_std_dev(data)
    ...
```

LISTING 6.4: Example PyCOMPSs Application With Multiple Native MPI Tasks

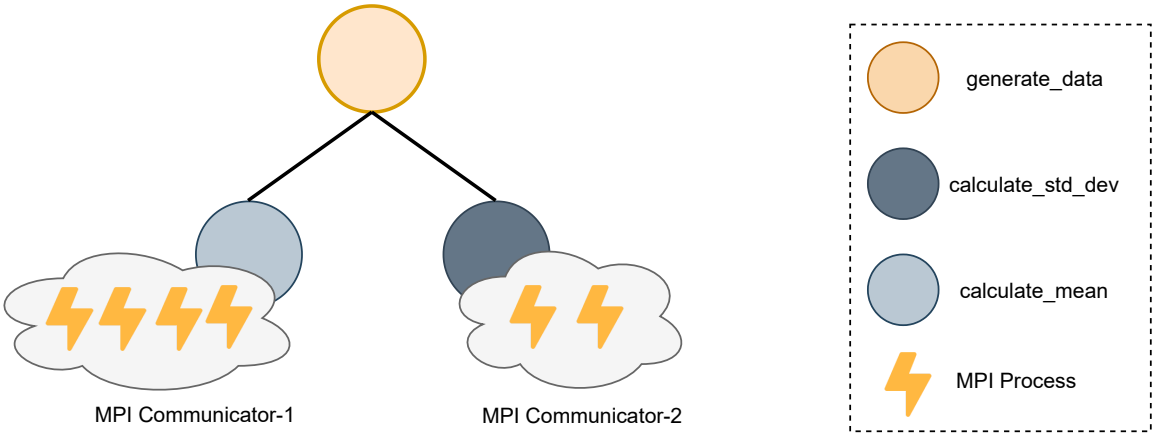



FIGURE 6.3: Sample PyCOMPSs Task Graph With Two Native MPI Tasks. Each Native MPI Task Has Its Own MPI Communicator

The design approach of isolating the execution environment of each Native MPI task enables flexibility in application design. Each Native MPI task has its own number of MPI processes and its own execution design in terms of number of communicators, rank distribution, etc. Hence, providing users more liberty in designing each Native MPI task according to its purpose, workload, criticality and also the heterogeneity of the underlying infrastructure.

### 6.3.3 Native MPI Tasks Execution

Tasks execution in PyCOMPSs is carried out by persistent Python threads called *Persistent Workers*. These persistent workers are launched after starting the COMPSs worker component. They exist throughout the lifetime of the application, i.e., only terminated at the end of application execution. By default, the number of persistent workers on each worker node is equal to the number of CPUs on that node, such that, each Python worker uses one CPU. Indeed, the number of persistent workers or their CPU affinity can be changed by users. If any of these persistent workers is not executing a task, it is set to a sleep or idle mode.

In order to make the MPI execution of tasks and the separation between Native MPI tasks execution possible, they are executed by a special temporary workers called *MPI Workers*. Unlike the default persistent workers of PyCOMPSs, MPI workers are temporary. Once a worker receives a Native MPI task for execution, it launches a MPI worker to execute that task. The MPI worker carries out the necessary pre- and post- execution operations such as launching the MPI environment with the required number of MPI processes, preparing the inputs of the tasks, gathering the output of the task into a list, and reporting the execution status.

Figure 6.4 shows a high-level overview of the execution behavior of different PyCOMPSs tasks on two nodes, each with 3 CPUs. At application launch time, as many persistent Python processes as CPUs are launched on each worker. Once a worker receives sequential task execution, it wakes up one of the idle Python processes to execute the task. However, Native MPI tasks are executed by MPI temporary workers. MPI workers do not use the persistent workers, and launch the requested MPI processes to execute the task. As soon as the Native MPI task execution finishes, the MPI worker is terminated. More details on the implementation of the MPI worker in the next section.

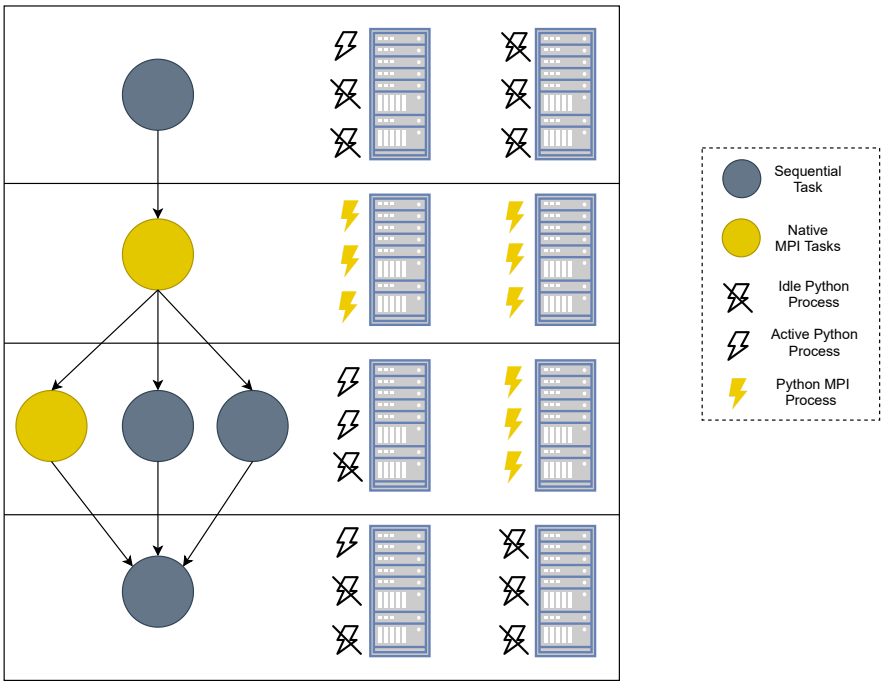



FIGURE 6.4: PyCOMPSs Tasks Execution Behaviour. Sequential Tasks Use PyCOMPSs Persistent Workers. Whereas Native MPI Tasks Use MPI Workers For Launching The Required Number Of MPI Processes (The First Native MPI Task Requires 6 MPI Processes, The Second Native MPI Task Requires 3 MPI processes)

## 6.4 Architectural Design

As previously mentioned, executing MPI code in tasks by the use of Native MPI tasks is completely transparent to application developers. Users only have to add a few lines of code, that are task decorators and task constraints without worrying about any execution time management operations such as data transfers, preparing inputs and outputs, etc. Such chores of task management and execution details are carried out by the PyCOMPSs runtime system (i.e., COMPSs).

In order to enable such transparency, we have made enhancements and extensions to both the master and worker components of the COMPSs runtime. The following sections describe the details of these extensions. Section 6.4.1 describes the enhancements added to the COMPSs runtime master. Whereas Section 6.4.2 describes the extensions made to the COMPSs runtime worker.

### 6.4.1 COMPSs Master

At the master side of COMPSs, the master components work with the *Task* abstraction which contains information about task properties such as type, dependencies. The Task abstraction allows other components such as the Task Analyser and the Task scheduler to be independent of the actual type of task (i.e., sequential, Native MPI, etc.). Nevertheless, we enhanced the task detection to support new programming model annotations introduced by the Native MPI task and also describe the scheduling of Native MPI tasks by the Task Scheduler.

#### 6.4.1.1 Task Detection and Scheduling

The Python bindings layer registers the annotated tasks to the COMPSs runtime with no further handling. Tasks are detected by the means of Python decorators on top of the object or class method or usual method. Hence, our prototype only extends the previous PyCOMPSs version by adding the new decorator for the Native MPI task (i.e., `@mpi`). Table 6.1 lists the supported arguments for the Native MPI decorator.

| Argument | Value Type | Use | Mandatory? |
|---|---|---|---|
| `runner` | String | specifies the MPI runner, e.g. `mpirun`, `mpiexec` | Yes |
| `processes` | Integer | specifies the number of MPI processes | Yes |
| `scale_by_cpu` | Boolean | specifies if the number of MPI processes should be scaled by the number of CPUs of the `@constraint` decorator | No |
| `priority` | Boolean | indicates whether the task has priority | No |

TABLE 6.1: List Of Supported Arguments In The *@mpi* Decorator

Figure 6.5 illustrates the detection and analysis of Native MPI Tasks. The Python binding organizes the task information listed in Table 6.1 and other information such as the task signature, expected number of outputs, their types, etc. The binding sends task registry requests to the COMPSs runtime master. As soon as the task registry request reaches the COMPSs runtime master, it encapsulates the task information in a Task object. Then, this

Task object gets processed by the runtime components such as the Task Analyser and the Task scheduler. The Task Analyser detects the dependencies between the given Task objects and previously received Task objects without taking into consideration the task type.

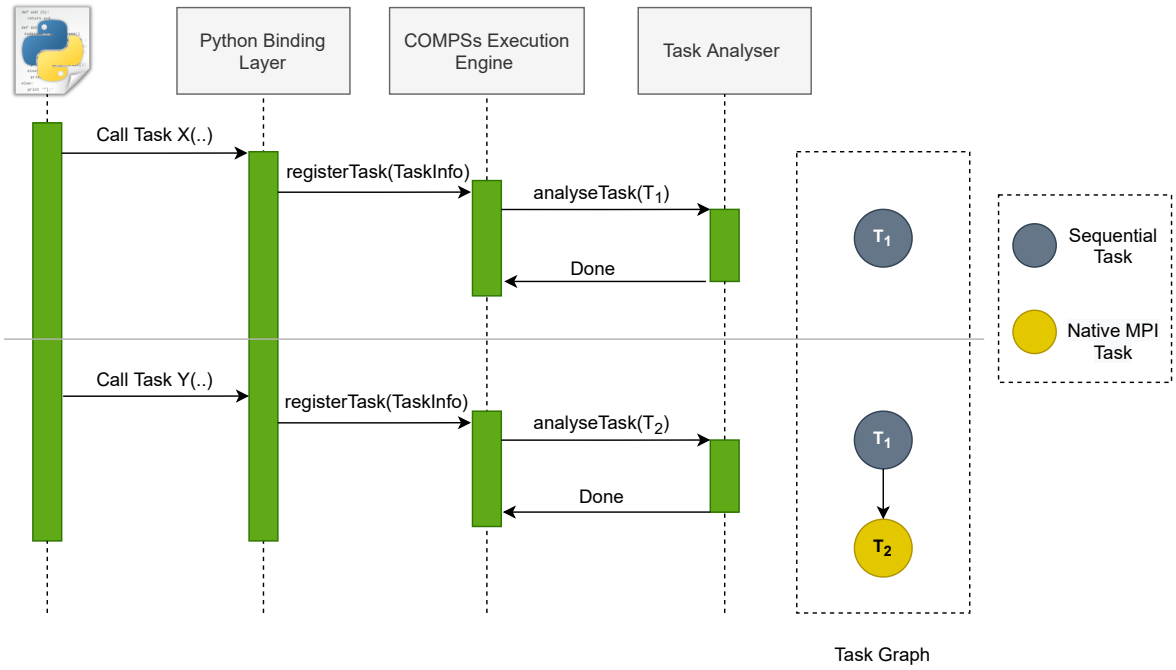



FIGURE 6.5: Native MPI Task Detection And Analysis

With regard to Native MPI tasks scheduling, they are launched for execution only if their number of processes can be satisfied by an equivalent number of free computing units.

### 6.4.2 COMPSs Worker

The COMPSs master component is able to communicate job execution requests from and to the COMPSs workers on the worker nodes regardless of the type of the task to be executed. In addition to that, COMPSs can communicate these messages with different kinds of infrastructures such as grids or cloud machines. This communications transparency is possible because the communication layer abstracts the infrastructure details from the master. Nevertheless, the worker processes that are spawned on each worker depends on the underlying communications adaptor implementation.

#### 6.4.2.1 Invokers

When a task execution request arrives to the COMPSs worker, it is assigned to one of the persistent workers. This worker starts the process of invocation the actual task code. Before executing a Native MPI task, the worker sets a number of environment variables that are accessible in the task code. For instance, information about the assigned resources. Table 6.2 lists these environment variables.

The persistent worker uses the *GenericInvoker* class that provides an abstract API for executing different types of tasks. For instance, sequential tasks, Native MPI tasks, binary tasks, etc. The GenericInvoker class is extended to different types of invokers, each for invoking a specific type of tasks. These child classes can be listed as follows:

| Environment Variable | Description |
|---|---|
| COMPSS_HOSTNAMES | List of computing nodes names or IPs |
| COMPSS_NUM_NODES | Number of computing nodes |

TABLE 6.2: List Of Environment Variables For Native MPI Tasks In The Worker

- The *PipedInvoker* class: used for invoking sequential Python tasks. This class uses Linux Pipes in order to communicate with the persistent Python process that is going to actually call the task code.
- The *BinaryInvoker* class: used for invoking external binaries. This class forks, executes and monitors the execution of the binary command provided in the task. The BinaryInvoker class forks a *ProcessBuilder* process with the binary command of the task and returns the exit value of the executed binary.

In order to execute Native MPI tasks and launch the Python code of the task with multiple MPI processes, we extended the GenericInvoker class to a new sub-class which is called: *PythonMPIInvoker* class. This class is similar to both the PipedInvoker class and the BinaryInvoker class. On the one hand, similar to the PipedInvoker, because it is used to call Python code, that is the task code of the application. However, it does not use Linux pipes because it requires the Python code to be executed by $N$ MPI processes requested by the task. On the other hand, it is similar to the BinaryInvoker class because it follows a similar approach to calling the Python worker that eventually calls the Python code.

The main idea of the PythonMPIInvoker class is to fork a *ProcessBuilder* process that launches the Python worker with the MPI runner provided by the user and the number of MPI processes. This Python worker is going to call the actual Python task code. Such worker is called: *MPI Worker*. Since the MPI worker is launched with $N$ MPI processes, all the functions that it is going to call will also be executed by the same number of MPI processes. The next section presents more details about the MPI worker and its characteristics.

#### 6.4.2.2 MPI Worker

The MPI worker is a special worker that is executed by $N$ MPI processes such that $N$ is the number of MPI processes requested by the task. It performs the same operations as the sequential persistent Python workers such as deserializing tasks inputs, serializing tasks outputs, calling the actual task code, monitoring the execution and returning the exist status.

MPI workers are called per Native MPI execution. This allows for flexibility, as the *ProcessBuilder* process is launching the MPI runner with the number of MPI processes requested by the tasks.

Figure 6.6 illustrates the process of executing Native MPI tasks in the worker side. The COMPSs worker receives a task execution request for task $t_1$, the COMPSs worker detects the type of the task and decides to call the PythonMPIInvoker invoker because task $t_1$ is a Native MPI task. The PythonMPIInovker invoker forks a new ProcessBuilder process that launches the MPI worker `mpi_worker.py` using the MPI runner and number of MPI processes specified requested in task $t_1$. The MPI worker performs the necessary pre-execution operations then call the actual task. Notice that since this MPI worker is being executed with $N$ MPI processes, the task code will also be executed by $N$ MPI processes. After the task execution ends, the MPI worker performs the necessary post-execution operation then

terminates after sending the exit value to the calling ProcessBuilder which in turn returns it to the invoker that notifies the COMPSs worker of task completion.

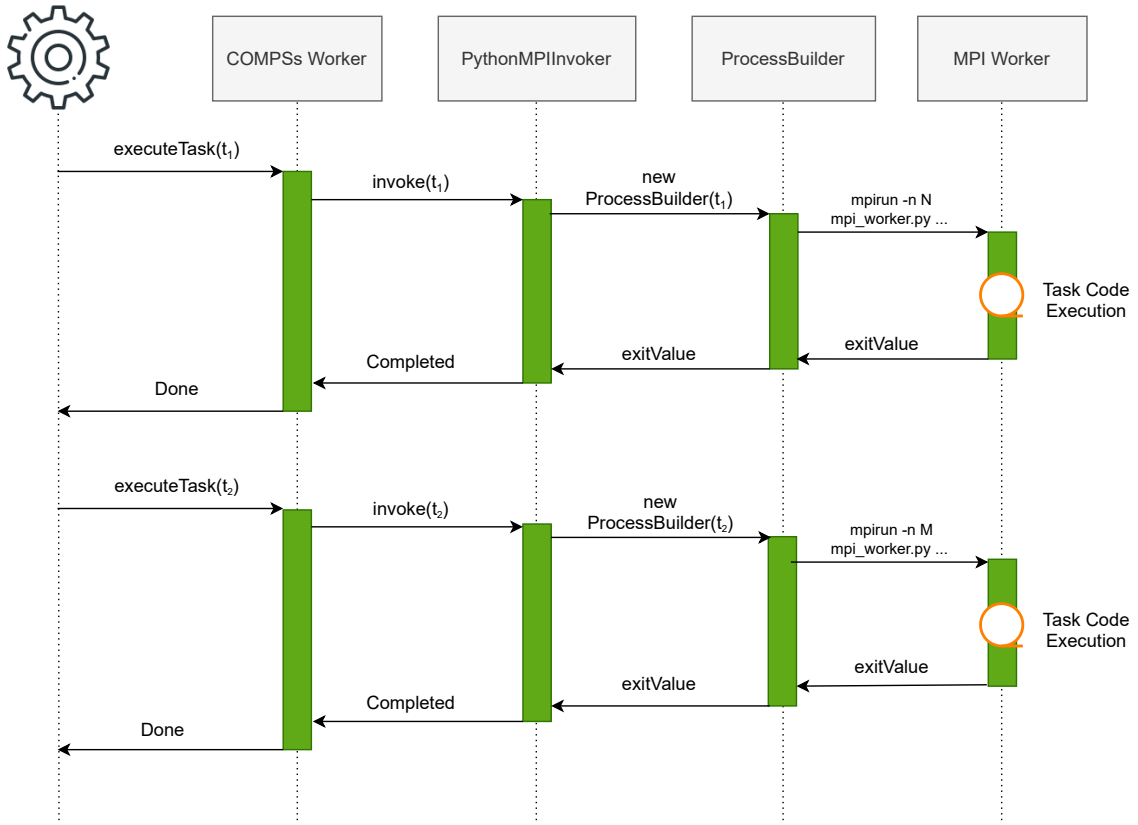



FIGURE 6.6: Native MPI Tasks Execution By MPI Workers

Furthermore, continuing with Figure 6.6, at a later point of the execution, another task $t_2$ arrives at the COMPSs worker for execution. The COMPSs worker uses the PythonMPIInvoker to invoke the task because it is a Native MPI task. The PythonMPIInvoker forks a ProcessBuilder process but this time with $M$ MPI processes that are requested by $t_2$. Hence, the $M$ MPI processes that executed the MPI worker will propagate and execute the actual task code in parallel.

It should be noted that all the MPI processes that participate in the task execution deserialize all the inputs of the tasks. This approach may seem redundant, however, it has the advantage that all the task inputs will be available to all MPI processes. Hence, not limiting the user code by mapping a group of inputs to a group of MPI processes. Therefore, enabling greater flexibility in task design.

Moreover, Figure 6.7 illustrates the exit value check process in the MPI worker. After Native MPI task execution end, the MPI process with rank 0 collects the return value of all MPI processes by using a `MPI_Reduce` call and performs the necessary checksum operations to make sure that the execution of each MPI process has finished successfully.

```
1       local_exit_value = 0
2       ...
3       # call task code and get exit value
4       ...
```

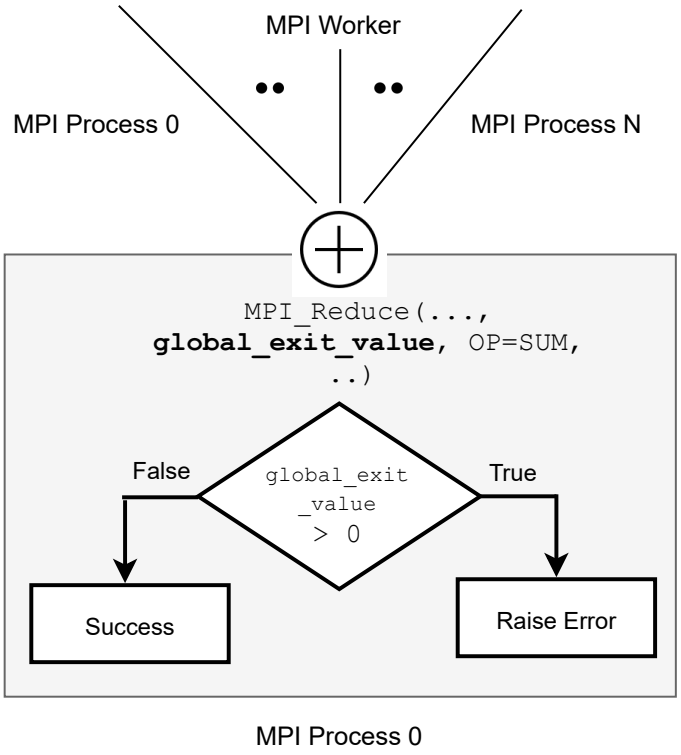



FIGURE 6.7: Exit Value Check For Native MPI Tasks

Figure 6.8 shows the process of collecting the return value of the task. Following the same approach to checking the exit value, MPI process 0 gathers all the return values of the other processes via a `MPI_Gather` API call. Therefore all the return values of all the MPI processes that have participated in the execution are stored in a list owned by MPI process 0. After preparing the return value of the task, process 0 serializes it to disk in the appropriate directory specified by the COMPSs worker and received in the MPI launch command. Finally, MPI process 0 returns the appropriate exit value to the caller process and all the processes finalize the MPI environment for this execution. It should be noted that Line 4 in the code snippet of Figure 6.8 shows that calling the task code returns a tuple of two values: the task returns and the exit value of the task for the calling MPI process. This is not the actual returns of the task code, instead it is the returns of the post-processing that is performed by the implementation of the `@task` decorator, which is carried out after the task execution has finished. The actual call of the task code only returns the values specified by the `return` keyword in the task code.

```
1       local_task_returns = None
2       ...
3       # call task code and get exit value
4       local_task_returns, local_exit_value = ...
5       ...
```

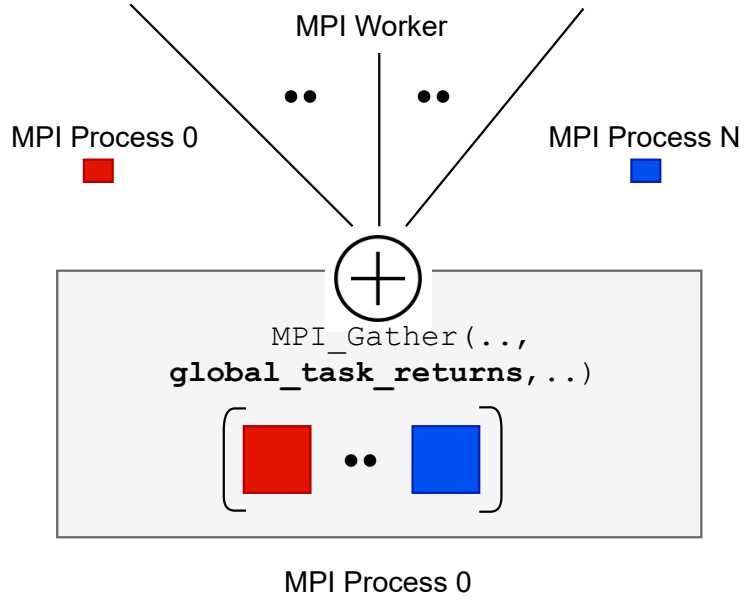



FIGURE 6.8: Exit Value Check For Native MPI Tasks

## 6.5 Evaluation

This section presents an evaluation of the proposed hybrid programming model of PyCOMPSs and MPI inside tasks through the use of Native MPI tasks. First, Section 6.5.1 evaluates this hybrid programming model based on applications programmability. Then, Section 6.5.2 presents the impact of using Native MPI tasks on the performance of two real-world applications and a discussion about the impacts of using Native MPI tasks in PyCOMPSs.

### 6.5.1 Programmability Evaluation

Developers productivity can be measured in terms of a function that relates the following factors: From the one hand, the effort to write application code expressed by the number of lines of code. From the other hand, the performance obtained by such code. High productivity can be characterized by a low effort in writing or modifying application code that results in a high performant application. Whereas low productivity can be expressed by a high effort in writing or changing application code without getting sufficient performance when executing the application. Indeed, other factors can affect the productivity such as the working environment, development frameworks, deploying prerequisite software on the target system, etc. However, we only consider those factors that are related to application programming and performance when using Native MPI tasks and without using them.

This section demonstrates that Native MPI tasks improves productivity by easing the coding of the application. Hence, the application that is discussed in this section only serves to highlight the benefits of using Native MPI tasks at the programming level without focusing on the performance. Indeed, we discuss only a single application. However, the

conclusions of this section can be generalized on any application with similar characteristics.

Comparing our approach of enabling Native MPI tasks in the PyCOMPSs task-based parallel programming models to other approaches that aim at achieving more performance by exploiting multi-core architectures and distributed computing, our proposal has the potential to achieve better performance without compromising applications programmability and productivity. Table 6.3 summarizes the main differences between our proposed approach and other solutions such as MPI programmed pure Python and MPI programmed pure C.

| Approach | Parallelism | Programming Language |
|---|---|---|
| C with MPI | MPI Processes | C |
| Pure Python with MPI | MPI Processes | Python |
| Vanilla PyCOMPSs | Tasks | Python |
| PyCOMPSs with Native MPI support | Tasks + MPI Processes | Python |

TABLE 6.3: Comparison Between Different Approaches For Distributed Computing

Table 6.3 shows different approaches that promise more performance by parallelizing applications. For instance, a pure Python or C applications in which MPI is used for parallel execution. Using the Python programming language for developing applications has became popular in disciplines such as artificial intelligence, web analysis and life sciences due to its simplicity and ease of use. On the contrary, The C/C++ programming language has low-level data structures that can achieve more performance, however it is more complex for users with no programming expertise. In both cases, MPI can be used to parallelize applications execution. Nevertheless, using only MPI without using any framework that supports transparent distributed execution pose challenges to non-expert developers:

- First, this approach only relies on MPI parallel processes in order to achieve more applications performance. Hence, it does not reach the same parallelism level as using the hybrid model of PyCOMPSs and MPI.
- Second, application developers have to take care of fine parallelism details such as how many parallel process should execute each part of the application; which parts of the code will be executed in parallel and which parts will be executed sequentially. This is usually done by using if-conditionals to check the ranks of the MPI processes or by spawning child processes from the MPI master process. In addition to that, managing the data between processes in such a context is a challenging task that can increase programming complexity.

Continuing with Table 6.3, The PyCOMPSs programming model enables the distributed execution of Python applications while the maintaining ease of programming. Using PyCOMPSs, users can taskify their applications simply by adding the PyCOMPSs task decorators before the functions declaration. However, the vanilla PyCOMPSs supports only task parallelism, leaving a lot of performance improvement opportunities inside tasks unaddressed. Therefore, the vanilla PyCOMPSs enables ease of development and performance, even though this performance can be improved by parallelizing tasks execution.

Finally, unlike the aforementioned approaches, the hybrid programming model of PyCOMPSs and MPI promises ease of development and high performance:

- First, it extends the PyCOMPSs programming model, therefore, maintaining the ease of development. Such programming model allows the same source file to contain different types of tasks (e.g., Native MPI tasks, sequential tasks, etc.).
- Second, it combines two levels of parallelism, i.e., task parallelism and MPI parallelism by the use of Native MPI tasks. All PyCOMPSs tasks have their own independent configuration and execution environment. This execution transparency is possible because the PyCOMPSs runtime handles tasks execution and manages data transfers between different tasks in an abstract manner.

Therefore, the hybrid programming model of PyCOMPSs and MPI (i.e., Native MPI tasks) increases productivity by enabling a higher degree of parallelism with less programming effort and less complexity on the application side.

Listing 6.5 provides a sample sequential application. The main for loop (Lines 16-19) processes a set of files that contain a list of numbers. Each file is read and its numbers are returned (Lines 6-9), then the mean is calculated for each numbers list (Lines 1-4). Listings 6.6 and 6.7 shows different parallel implementations with MPI, and PyCOMPSs with Native MPI tasks respectively.

```
1  def calculate_mean(data):
2      # Calculate the mean
3      ...
4      return mean
5
6  def read_file(path):
7      # Read the file
8      ...
9      return data
10
11 if __name__ == "__main__":
12     files_paths = sys.argv[1]
13
14     mean_list = []
15
16     for path in files_paths:
17         data = read_file(path)
18         mean = calculate_mean(data)
19         mean_list.append(mean)
20
21     print mean_list
```

LISTING 6.5: Sample Application For Calculating The Mean Number In Input Files

```
def calculate_mean(data):
    # Calculate the mean in parallel
    ...
    return mean

def read_file(path):
    # Read the file in parallel
    ...
    return data

if __name__ == "__main__":
    rank = MPI.COMM_WORLD.rank

    files_paths = sys.argv[1]
    local_mean_list = []

    For path in files_path:
        data = None
        # All the MPI processes, e.g., 40 MPI process \\
        # execute the read_file function
        data = read_file(path)

        # Wait until all MPI processes have finished
        MPI_Barrier()

        local_mean = None
        # Only 20 MPI processes should execute \\
        # the calculate_mean function
        if rank < 21:
           local_mean = calculate_mean(data)
        local_mean_list.append(local_mean)

        MPI_Barrier()

    if rank == 0:
        global_mean_list = []

    MPI.COMM_WORLD.Gather(local_mean_list,
                          global_mean_list,
                          root=0)

    if rank == 0:
        print global_mean_list
```

LISTING 6.6: Sample MPI Application For Calculating The Mean Number In Input Files

```
1  @mpi(runner="mpirun", processes=20)
2  @task(returns=list)
3  def calculate_mean(data):
4      # Calculate the mean in parallel
5      ...
6      return mean
7
8  @mpi(runner="mpirun", processes=40)
9  @task(returns=list)
10 def read_file(path):
11     # Read the file in parallel
12     ...
13     return data
14
15 if __name__ == "__main__":
16     files_paths = sys.argv[1]
17
18     mean_list = []
19
20     for path in files_paths:
21         data = read_file(path)
22         mean = calculate_mean(data)
23         mean_list.append(mean)
24
25     mean_list = compss_wait_on(mean_list)
26
27     print mean_list
```

LISTING 6.7: Sample PyCOMPSs Application With Native MPI Tasks For Calculating The Mean Number In Input Files

The code in Listing 6.6 shows one possible MPI implementation for parallelizing the application. The MPI processes read the input files in parallel and do the mean calculation in parallel. After calculating the mean, all the results are gathered to MPI process 0 to be displayed. Whereas the code in Listing 6.7 shows a hybrid PyCOMPSs and MPI implementation. The functions `read_file` and `calculate_mean` are defined as Native MPI tasks by the use of PyCOMPSs annotations. Indeed, the implementation of the `read_file` and `calculate_mean` functions/tasks in the MPI implementation and the PyCOMPSs implementation can be similar. Therefore, they are not taken into consideration when comparing both implementations.

Comparing the three implementations: the sequential implementation (Listing 6.5), the MPI implementation (Listing 6.6), and the PyCOMPSs with Native MPI tasks implementation (Listing 6.7), one can notice the simplicity of using PyCOMPSs with Native MPI tasks to parallelize the application as compared to the MPI implementation. On the one hand, with PyCOMPSs, the code is task-parallelized with minimal additions of 5 lines of code: Lines 1-2, 8-9 which are functions annotations to declare the tasks and their required number of MPI processes, in addition to Line 25 which is the PyCOMPSs synchronization API used to retrieve the results to the master node. On the other hand, the MPI implementation introduced obvious changes to the code to the logic of the code to confirm with the MPI parallelization paradigm. For instance, adding if-conditionals to make different ranks perform

certain operations. Line 38 in Listing 6.6 is a `MPI_Gather` call to collect the result to MPI process 0.

Furthermore, application programming and design flexibility is reduced in the MPI implementation because the entire application is launched with a specific number of MPI processes. Therefore, the number of MPI processes that execute the `read_file` function is the same as the number of MPI processes that execute the `calculate_mean` function. Separating the MPI processes that execute both functions requires tedious design and coding effort such as controlling the number of MPI processes to enter the task by using if-conditionals on the ranks then making sure which MPI process has which result, or, spawning a different MPI communicator and making the necessary pre- and post- operations such as configuring the new communicator and redistributing the results. In the PyCOMPSs implementation, controlling the number of MPI processes to execute each function/task is as straight-forward as changing the values of the `processes` argument in the `@mpi` decorator in Listing 6.7. In addition to that, with fewer lines of code, PyCOMPSs with Native MPI tasks enable task parallelism and MPI parallelism. The quantitative evaluation of such levels of parallelism is presented in the next section.

### 6.5.2 Performance Evaluation

To measure the performance impact of enabling MPI parallelization in Native MPI tasks of PyCOMPSs, we ran experiments on the following types of applications an I/O intensive application and a compute intensive application. We studied the performance impact on task performance and also total application performance. In both applications, we targeted the MPI parallelization of critical tasks in the application by declaring such tasks as Native MPI tasks. The baseline is a PyCOMPSs implementation of the applications where these critical tasks are implemented sequentially.

Section 6.5.2.1 presents the used infrastructure. Section 6.5.2.2 describes the performance impacts of using MPI in Native MPI tasks to parallelize I/O tasks in a *Blocked Matrix Multiplication* application. Section 6.5.2.3 presents the performance impacts of using Native MPI tasks to parallelize compute tasks in a *Web Archives Analysis* application. Finally, Section 6.5.2.4 discusses the performance trade-offs between task parallelism and MPI parallelism.

#### 6.5.2.1 Infrastructure

All experiments of the following sections were run on the MareNostrum 4 supercomputer of the Barcelona Supercomputer Center (BSC). This supercomputer was previously described in Section 4.5.1.

Due to the master-worker deployment architecture of PyCOMPSs, each submission to the supercomputer queuing system was done with the number of worker nodes plus one that is dedicated as the master node. In all the experiments of this section, we mentioned only the number of worker nodes used for each experiment. In this configuration, the master node only launches and manages the execution on worker nodes and does not perform any computations.

All the experiments in Sections 6.5.2.2 and 6.5.2.3 were launched on 8 high-memory nodes of the MareNostrum4 supercomputer. Each experiment was run 10 times and the average results are reported.

#### 6.5.2.2 Write-Intensive Blocked Matrix Multiplication

For testing the performance impact of parallelizing I/O tasks with MPI using *Native MPI* tasks in PyCOMPSs, we implemented a task-parallel PyCOMPSs version of the Blocked

Matrix Multiplication Algorithm. Matrix multiplication is one of the most important and recurrent matrix operations that has many uses such as network theory and population modeling. Our application decomposes the matrices into blocks and multiply the blocks to each other, then it writes the multiplication result to a file on the General Parallel File System (GPFS) of the Marenostrum 4 supercomputer. The application wrote the results to HDF5 formatted-files. HDF5 [105] is a binary data storage format that has a directory-like structure called HDF5 groups. We used the h5py Python library [104] which provides a high-level Python API for performing parallel I/O.

Figure 6.9 shows a snippet of the task graph of one of the experiments. It consists of three main tasks:

- `create_block`: generates `N` number of blocks of random floating point numbers. Each block has a total of `M` numbers.
- `multiply`: performs the multiplication of matrix blocks.
- `write_result`: writes matrix blocks to a HDF5 file, each block is written to a separate HDF5 group. There are two implementations of this task: (i) in the baseline experiment, the writes are done sequentially. (ii) in the Native MPI implemention of the task, each block is written in parallel by `X` MPI processes.

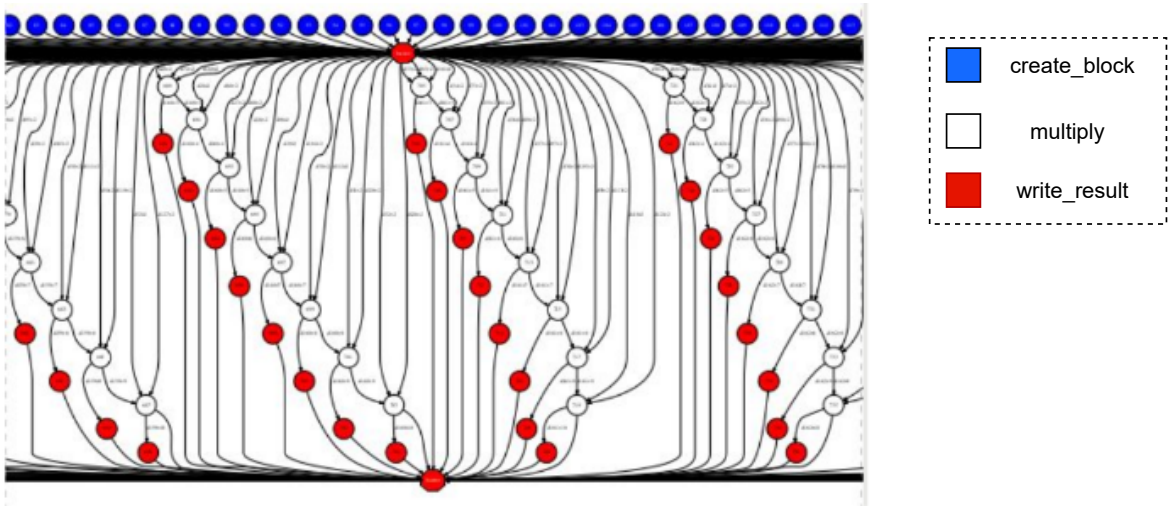
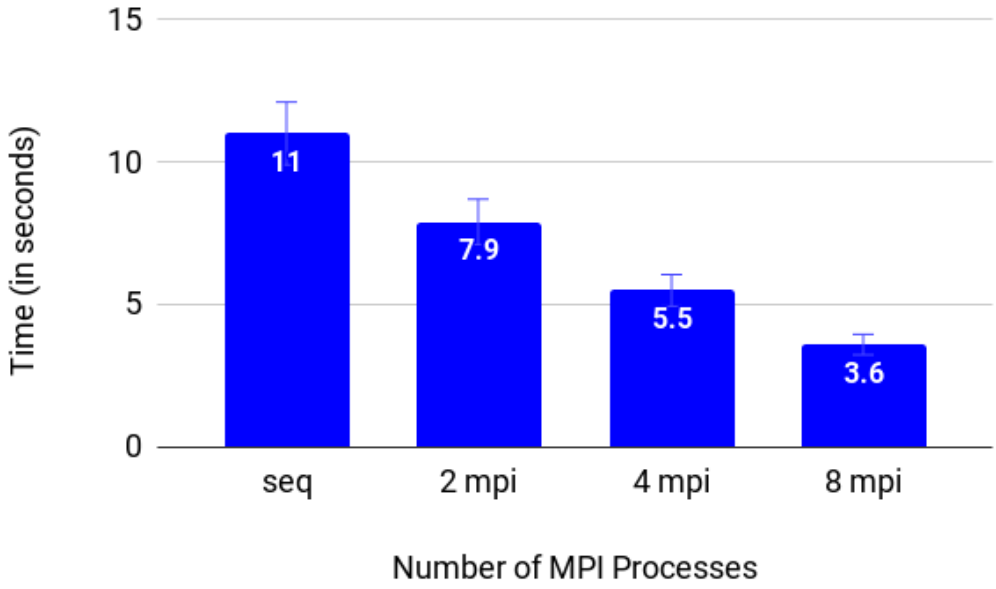



FIGURE 6.9: Blocked Matrix Multiplication Task Graph Snippet

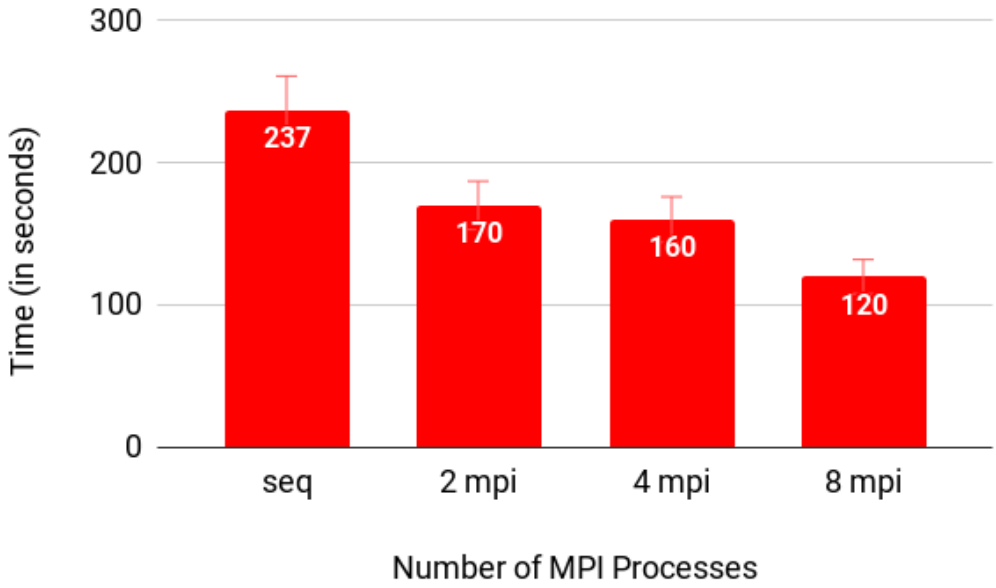
We used as input a two generated 2D matrices that are decomposed into blocks of elements. The number of blocks in each matrix is 16 blocks. Each block contains 8192x8192 floating point random numbers. All runs of this application produced 674 GB of data. This application was run several times, each time with a different implementation or configuration of the `write_result` task: One time using the sequential implementation for the baseline experiments, and several times with an increasing number of MPI processes per Native MPI task. The application has a total number of 1216 tasks: 192 task that generates a block, 512 multiply block tasks that carries out the multiplication process and corresponding 512 write tasks that writes the result of each multiplication.

Figure 6.10 presents the performance results of the application. Figure 6.10(a) shows the performance benefits on the `write_result` task level, whereas Figure 6.10(b) shows the total performance of the application. The MPI implementation of the task enabled by the use of Native MPI tasks resulted in a significant performance improvement that reaches up to about 3x I/O speedup when using 8 MPI processes per task compared to the sequential

implementation of the task (Figure 6.10(a)). This performance improvement on task-level is reflected as a performance improvement in the total time of the application that reaches up to 1.9x total performance speedup (Figure 6.10(b)).




(a) Average Time Per Write Task




(b) Total Execution Time

FIGURE 6.10: Performance Results Of The Blocked Matrix Multiplication Application

The performance benefit shown in Figure 6.10 in the task time and total application time can be explained by the performance benefit gained from increasing the number of MPI processes per Native MPI tasks. Increasing the number of MPI processes enables the `write_result` task to execute faster which advances the execution of tasks in the whole

application.

Nevertheless, one can notice that the performance results in Figures 6.10(a) and 6.10(b) is diminishing as the number of MPI processes per Native MPI task increases. This can be explained due to that the more MPI processes are used, the smaller the size of the data to write gets and the overhead of the PFS components become more noticeable. Besides, as the number of MPI processes per *Native MPI* task increases, the total number of MPI processes performing I/O operations increase. Therefore, overloading the PFS.

#### 6.5.2.3 Web Archives Analysis

In order to evaluate the performance of Native MPI tasks in a compute-intensive applications, we developed a PyCOMPSs application that calculates the term frequency (TF-IDF) of web archives. Web archives are stored in a special file format called WARC (Web ARChive) [115], which is a file format used for web pages archiving and it is widely used for web analysis. A WARC file is a container of WARC records that consist of URIs, Dates, lengths and web page content. Each record in the WARC file can be of different length. Figure 6.11 shows a snippet of the task graph of one of the experiments. The application consists of the following tasks:

- A `read_record` reading task that reads a record from the file.
- A `calculate_tf_idf` compute task that calculates TF-IDF.

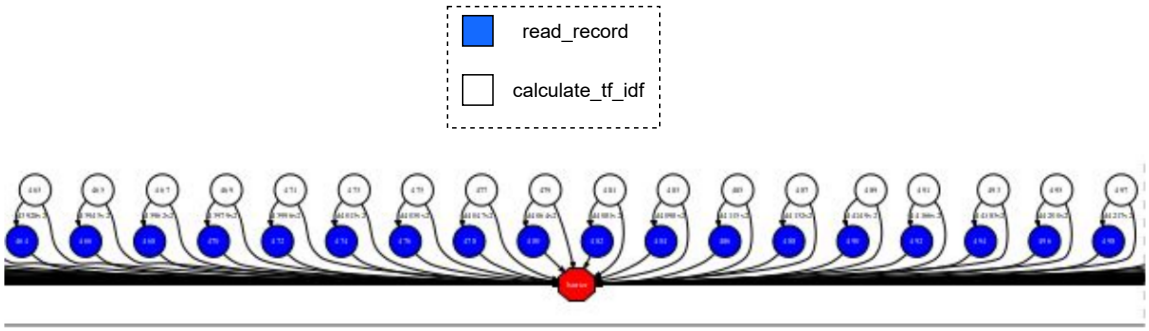
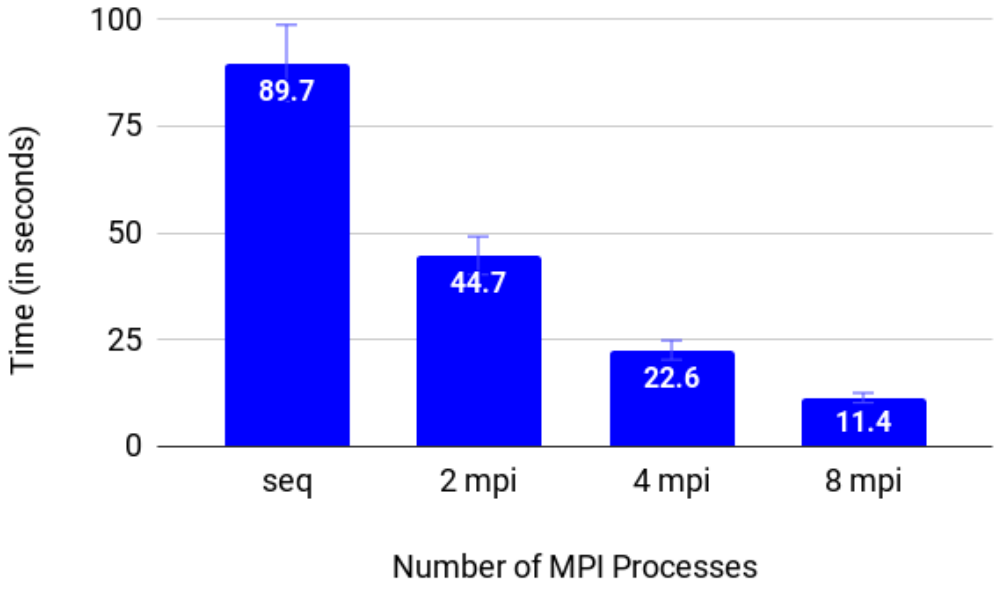


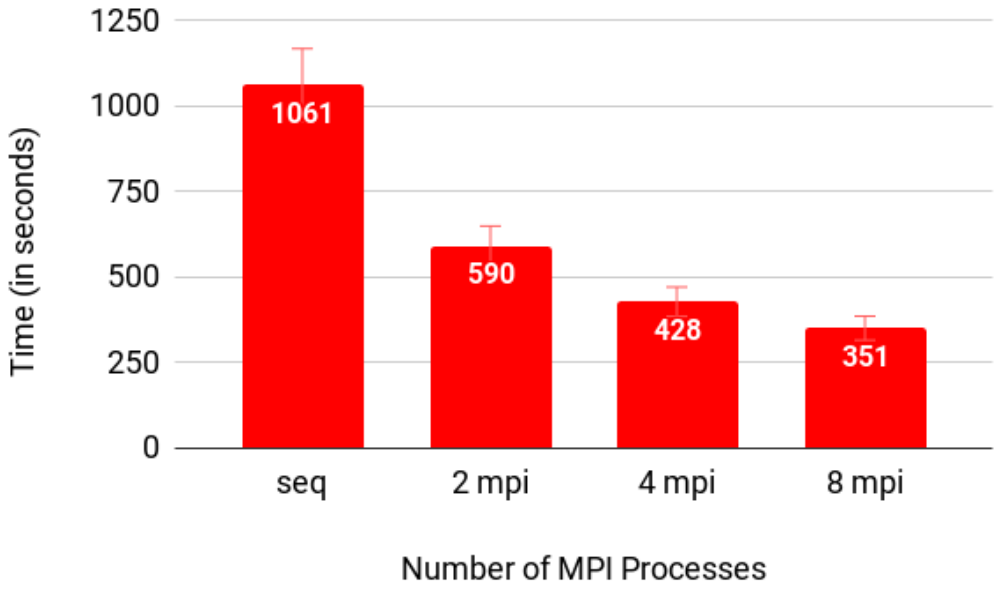

FIGURE 6.11: Web Analysis Task Graph Snippet

We identified the `calculate_tf_idf` tasks as the target tasks to be identified as Native MPI tasks. Several experiments were launched: the baseline experiment with a sequential implementation of the `calculate_tf_idf` task, and several other runs with a parallel MPI implementation of the `calculate_tf_idf` task with increasing number of MPI processes. The experiments used a WARC file of a total size of 186 GB from Common Crawl [28]: an open repository of web crawl data. The total number of tasks for this application is 1440 tasks: 720 read tasks and 720 corresponding compute tasks.

Figure 6.12 shows the performance results of the application. Figure 6.12(a) presents the performance results at the `calculate_tf_idf` task level, where as Figure 6.12(b) shows the total performance results. Similar to the results of the previous application, using Native MPI tasks to parallelize the `calculate_tf_idf` task execution results in noticeable performance improvement when increasing the number of MPI processes. Using 8 MPI processes per compute task results in a more than 7x performance speedup at task time compared to the sequential implementation of the task (Figure 6.12(a)). This performance improvement at task level is reflected in the total performance of the application that reached 3x performance speedup compared to the sequential impelemntation of the compute task (Figure 6.12(b)).




(a) Average Time Per Compute Task




(b) Total Execution Time

FIGURE 6.12: Performance Results Of The Web Analysis Application

The total time improvement when increasing the number of MPI processes per compute task can be explained by the benefit gained of parallel task execution. Hence, increasing the number of MPI processes per decreases the time of dominant compute tasks and thus shortening the critical path of the application which contributes to the betterment of the total time performance.

Nevertheless, similar to the performance pattern of the matrix multiplication application, the performance improvement in the total time of the application is diminishing when increasing the number of MPI processes per compute task. The following section discusses

the causes of such behaviour.

#### 6.5.2.4 Parallelism Trade-off

To further understand the performance and behaviour of Native MPI tasks in our hybrid programming model prototype, several experiments were conducted on the Blocked Matrix Multiplication and the Web Analysis applications. Every experiment is launched multiple times with increasing number of MPI processes per Native MPI tasks (2, 4, 8, 16 and 48). All All experiments were repeated on increasing number of nodes (4, 8 and 12).

Figure 6.13(a) presents the results of the Blocked Matrix Multiplication application and Figure 6.13(b) presents the results of the Web Analysis application. It can be noted that in both figures, as the number of nodes increases, the total execution time of both applications improves. As the number of executing resources increases, PyCOMPSs is able to launch more tasks to be executed in parallel, hence, task parallelism increases and applications total time improves.

Nevertheless, it can be noted in both charts of Figure 6.13 that for a specific number of nodes, as the number of MPI processes per Native MPI task increases the total execution time decreases until it reaches a point after which it starts to increase. For the Matrix Multiplication application (Figure 6.13(a)), this point is 4 MPI processes for 4 nodes and 8 MPI processes for 8 and 12 node. Whereas, for the Web Archive Analysis application (Figure 6.13(b)) this turning point is 8 MPI processes for 4, 8 nodes and 16 MPI processes for 12 nodes.

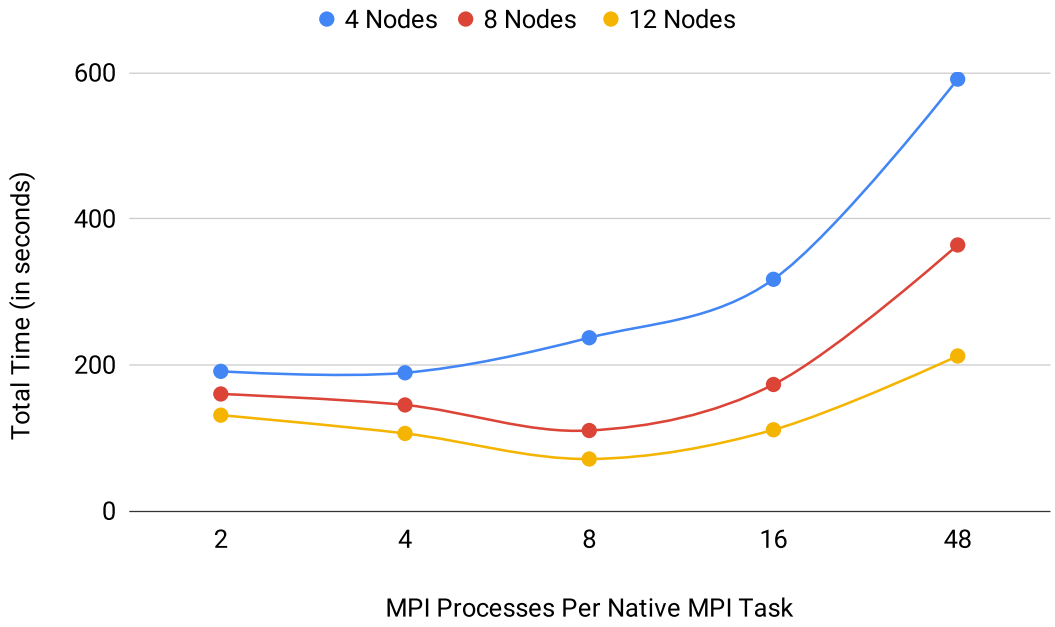
As already mentioned in the previous sections (Sections 6.5.2.2 and 6.5.2.3), although both applications achieve better performance on task level with increased number of MPI processes per task, this improvement is not reflected in applications total time after a certain number of MPI processes. Increasing the number MPI processes per Native MPI task decreases task parallelism. As the number of MPI processes per Native MPI task increases, the scheduling requirements of the task increases. During the execution time of Native MPI tasks, all the CPUs that are reserved for the task cannot be reused until the task has finished its execution. Hence, executing Native MPI tasks occupies more resources and less tasks can be concurrently launched for execution. Such an effect can result in deteriorating total time performance at the application level, even though the performance improvement increases at the task level.

This effect is mitigated as the number of resources increases because there are enough resources to maintain the same level or allow for more task parallelism. This can be noted in Figure 6.13(a), for 4 nodes the performance degrades when more than 4 MPI processes per task are requested. However, when the number of execution worker nodes increase to 8 and 12 nodes, the total execution time starts degrading at a later point when more than 8 MPI processes per task are requested. The same can be noted in Figure 6.13(b) where for 4 and 8 nodes the total execution time degrades at 8 MPI processes but when the number of nodes is increased to 12, this point shifts to 16 MPI processes.

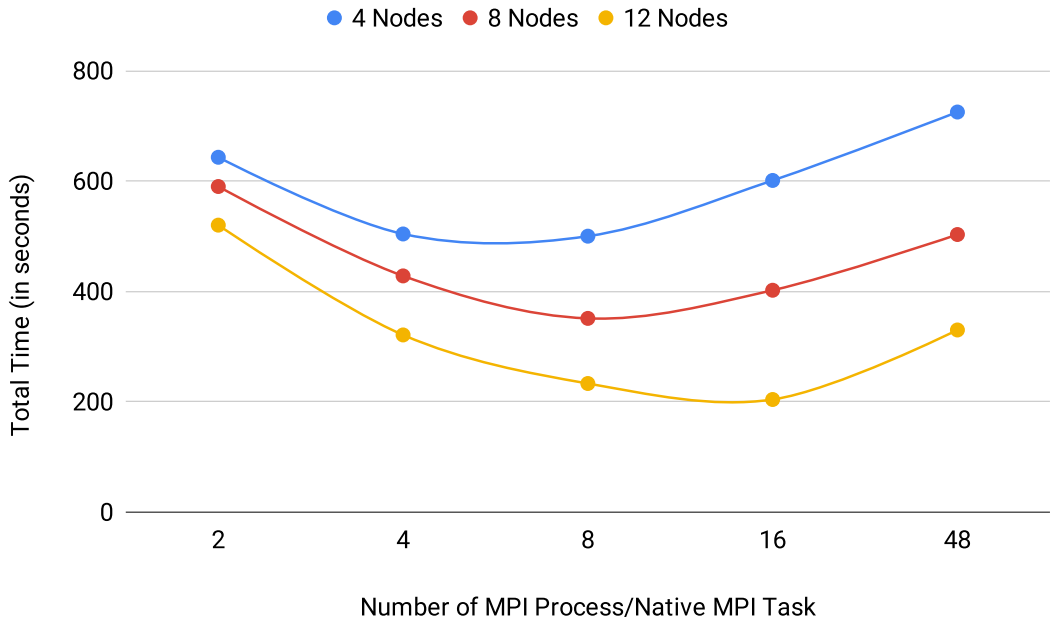
It should be noted that both applications have different performance turning points, because performance improvements and degradation depends on application characteristics, e.g., number of Native MPI tasks compared to sequential tasks, performance gain from increasing the number of MPI processes per Native MPI task, whether Native MPI tasks exist on the critical path of the application, etc.

## 6.6 Discussion

This chapter presents a hybrid programming model of high-level task-based programming models and MPI for tasks parallelization. Enabling the execution of MPI code natively




(a) Blocked Matrix Multiplication Application




(b) Web Analysis Application

FIGURE 6.13: Scalability Results

for parallelizing coarse-grained tasks in task-based models offers great benefits in terms of both programmability and performance.

Using such hybrid programming models, applications can be designed and programmed to better exploit the capabilities of modern systems while abstracting their complexity from applications side. In this chapter, we presented the implementation details of a prototype version of the PyCOMPSs task-based programming model and Native MPI tasks to execute

MPI code. Our prototype can be used to parallelize sequential code using minimal additions to applications code. Tasks implementation can be parallelized with MPI using the `@mpi` decorator. By enabling Native MPI tasks, the same application source file can contain different Native MPI tasks, each has different number of MPI processes, together with different sequential tasks. Our implementation prototype handles all the necessary operations for transparent managing of tasks execution and data transfer between different types of tasks.

Our experiments showed that using such a model gives significant improvements over sequential executions in both I/O intensive applications and compute intensive applications. This performance improvement is achieved in both Native MPI task level and total application time. However, a trade-off arises between per task MPI parallelism and total application task parallelism when increasing the number of MPI processes per Native MPI tasks. The number of tasks that can run concurrently deceases when increasing the number of MPI processes per task. Therefore task parallelism decreases, which may negatively affect the total time of the application.

As future work, we plan to improve the scheduling of tasks taking into account the programming model knowledge of the criticality of tasks, also the number of pending tasks and their computing requirements to better utilize the underlying infrastructure. Furthermore, we plan to support inter-task MPI processes communication.

# Chapter 7

# Optimizing Execution with the Eager-Release of Dependencies


## SUMMARY

Task-based programming models offer a flexible way to express the unstructured parallelism patterns of nowadays complex applications by providing high-level abstractions. This expressive capability is necessary to achieve maximum possible performance for applications that are executed in distributed large-scale execution infrastructures.

In task-based workflows, tasks are launched for execution when their data dependencies are satisfied. However, even though the data dependencies of a certain task might have already been produced, the execution of this task will be delayed until its predecessor tasks completely finish their execution. As a consequence of this traditional approach of releasing dependencies, the amount of parallelism inherent in applications is limited and performance improvement opportunities are wasted.

This chapter tries to solve research question $\mathbf{Q}_4$ by proposing an eager approach for releasing data dependencies. Hence, solving the limitations of traditional task-based programming models and optimizing applications execution. Following this approach, the execution of tasks will not be delayed until their predecessor tasks completely finish their execution, instead, tasks will be launched for execution as soon as their data requirements are available. Such approach can be used to overlap I/O and computation. Hence, more parallelism is exposed and applications can achieve higher levels of performance by overlapping the execution of tasks.

Towards achieving this goal, in this chapter we propose applying two conceptual and design changes to task-based programming models and systems. First, modifying the dependency relationships of tasks to be specified not only in terms of predecessor and successor tasks but also in terms of the data that caused these dependencies. Second, triggering the release of dependencies as soon as a predecessor task generates the output data instead of having to wait until the end of the predecessor execution to release all its dependencies.

As will be shown in this chapter, the proposal of eager-release of dependencies will not only be useful for I/O intensive applications, but also, compute intensive applications.

Additionally, this chapter presents the implementation details of the proposed changes in the PyCOMPSs framework and evaluates the performance with different use cases. The evaluation experiments show that using an eager approach for releasing dependencies can achieve more than 50% performance improvement in the total execution time as compared to the default approach of releasing dependencies.

## 7.1 Overview

As discussed in Section 1.1.5, task-based programming model decomposes applications into tasks. These tasks are organized in the form of a Directed Acyclic Graph (DAG) by detecting data dependencies between them so that each task has predecessor(s) and successor(s). Data dependencies between tasks control the scheduling of tasks and their execution. Tasks are launched for execution if they are dependency-free, i.e., all their predecessors have finished their execution successfully.

A data dependency relationship can occur between two tasks such that a task depends only on some of the outputs of its predecessor not the whole output set. Throughout this chapter, such type of dependency will be referred to as *Partial Dependency*. In this scenario, regardless of whether the outputs that constitute the dependency relationship are ready, a task requiring these outputs will not be executed until the predecessor task completely finishes its execution. In this chapter, such type of releasing dependencies will be referred to as *Lazy Release* of dependencies.

A task that periodically produces output data has the potential of creating more parallelism by allowing the overlapped execution of tasks. For instance, if a task produced some data that are required by one of its successors, this successor task can be released for execution while the predecessor task is still producing the remaining outputs.

Using a task-based programming model with a lazy approach of releasing dependencies has a drawback that limits the maximum amount of achievable parallelism in applications. Successor tasks execution will be blocked until their predecessor tasks completely finish execution even if the data required by the successor tasks have been already produced.

Such scenario where parallelism opportunities are wasted because of applications execution pattern and lazy release of dependencies is common in parallel I/O and distributed systems domains [99], [62]. In these domains, a task generates output data by reading from a file in a parallel manner but the workload of the parallel processes is imbalanced or they experience performance variability. Therefore, all successor tasks cannot start execution until the time-consuming processes have returned.

Furthermore, applications can be designed inefficiently because there is no programming model support for the timely release of tasks nor data deletion. All task's data has to be kept in memory until the task execution finishes. However, this is not feasible in applications where tasks read big data that cannot fit as a whole in memory. Thus, avoiding this problem leads to inefficient application design.

Additionally, this limitation can be also observed in compute-intensive applications in domains such as physical or geometrical systems simulations or data parallel applications like text analysis or bioinformatics applications. In these applications, tasks generate multiple output data where each has independent execution pipelines.

The contributions described in this chapter target the problem of the limited parallelism because of the lazy approach of releasing data dependencies. In this chapter, we propose an eager approach for releasing data dependencies. Using this approach, a task starts execution as soon as its data dependencies are ready, even if the predecessor task that produced these data has not yet finished execution. Adopting this approach accelerates the rate in which tasks are launched for execution, thus, more parallelism is exploited and higher performance can be achieved.

We use the PyCOMPSs [100] task-based programming model to demonstrate our contribution and its impact on the performance of applications. However, it should be noted that our proposal can be adopted by any system that uses data dependencies to manage executions.

The contribution in this chapter can be summarized in the following points:

1. Introducing an eager approach for releasing data dependencies in task-based systems. This approach is achieved by the following proposals:
    - A proposal for modifying the management of tasks dependencies. Tasks should be released for execution if their data requirements have been generated, instead of according to the execution status of their predecessor tasks.
    - A proposal for notifying the runtime system that a task has produced output data before the executing process reaches the `return` statement in the task code. Hence, dependencies can be released and successor tasks can start their execution as soon as their data requirements are ready even if the predecessor task has not finished execution.
2. An implementation of the eager approach for releasing dependencies in the PyCOMPSs framework and the evaluation of its performance with different use cases.

To the best of our knowledge, the aforementioned proposals are not supported by any of the current task-based programming models (see Section 7.2 for more details).

The rest of this chapter is organized as follows: Section 7.2 lists the related work to this contribution. Section 7.3 describes the motivation behind this contribution and formally states the problem. Next, Section 7.4 formally introduces our proposals for achieving the eager-release of data dependencies. Section 7.5 presents the implementation details of the eager approach for releasing dependencies in the PyCOMPSs programming model and runtime. Section 7.6 starts by evaluating the overhead of implementing the eager-release mechanism in the PyCOMPSs framework. Also, we present the performance evaluation of this contribution with different use cases that exhibit real patterns. Two of the use cases have a different rate of returning data and releasing dependencies which offers an evaluation of different performance aspects of our proposal. Whereas the other use case shows how our proposal enables a more efficient design for I/O intensive applications that have larger-than-memory inputs or exhibit high memory requirements. Finally Section 7.7 concludes the main points of this chapter.

## 7.2 Related Work

Examples of Python-based task-based programming models were highlighted in both Section 2.1 of the state of the art chapter and the related work sections of the previous contributions: Sections 4.2, and 6.2. Those task-based programming model follow a traditional approach for releasing data dependencies and start successor tasks execution. Successor tasks are only executed for execution if their predecessor task has completely finished execution.

In non Python-based task-based programming models, Perez et al. [87] proposed some techniques to improve task nesting and dependencies in OpenMP. One of the proposals, similar in spirit to one of our contributions, includes an API for releasing fine-tuned dependencies in a nested tasks scenario. In this work, an API call enables the release of sub-tasks without waiting for a super-task to finish its execution.

## 7.3 Problem Statement and Motivation

We define the problem in general Directed Acyclic Graphs (DAGs) with vertices and edges that model the applications. In this model, a DAG $G = (V, E)$ has vertices $v \in V$ representing tasks and directed edges $e \in E$ representing dependencies. These dependencies

are input and output data of the tasks. We denote $I(v)$ as the inputs of task $v$ and $O(v)$ as its outputs.

Two vertices (tasks) $p, s \in V$ are said to have a dependency relationship if the task corresponding to the vertex $s$ needs a data produced by the task corresponding to vertex $p$. More formally, a directed edge $e = (p, s)$ exists if $I(s) \cap O(p) \neq \emptyset$.

In the dependency $e = (p, s)$, task $p$ is a predecessor of $s$ because at least one of its outputs $o \in O(p)$ satisfies $o \in I(s)$. Also, task $s$ is a successor task of $p$ because at least one of its inputs $i \in I(s)$ satisfies $i \in O(p)$.

In task-based programming models, dependency relationships control when a task will be executed. Tasks are launched for execution if their data dependencies are met. Current task-based models specify dependency relationships only in terms of tasks and not the data that caused these dependencies. Consequently, successor tasks are launched for execution when all their predecessor tasks produce their entire output set(s) and completely finish execution.

It is possible that there is a dependency relationship $e = (p, s)$, where the successor task $s$ does not require the whole set of $O(p)$, instead, it depends only on a partial set of $O(p)$, such that $I(s) \subset O(p)$. We call this type of dependencies *Partial Dependency*. However, even though task $s$ is partially dependent on task $p$, it will not be executed unless task $p$ has produced all its outputs and finished execution.

Figure 7.1 highlights this point. Task $p$ produces output set: $O(p) = \{out_1, out_2\}$ and tasks $s_1$ and $s_2$ have input sets: $I(s_1) = \{out_1\}$ and $I(s_2) = \{out_2\}$. The dependency relationships are defined as $e_1 = (p, s_1)$ and $e_2 = (p, s_2)$ where $I(s1) \cap O(p) = \{out_1\}$ and $I(s_2) \cap O(p) = \{out_2\}$. Although $s_1$ and $s_2$ require two different outputs of $p$: $out_1$ and $out_2$ respectively, both successors have to wait until all outputs of $p$ are produced and $p$ completely finishes execution.

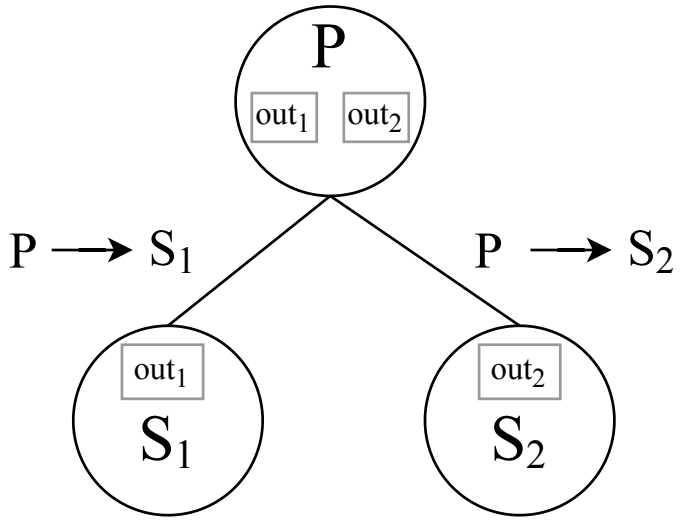



FIGURE 7.1: Dependency Relationships Identified Only As Task:Task Dependency

We call this default manner of releasing data dependencies: *Lazy Release*. The lazy release of data dependencies can be characterized by the inability to release a task for execution as soon as its data dependencies are satisfied. Therefore, a task is blocked and its execution is delayed until its predecessor task completely finishes execution, although the data constituting the dependency might have been ready before this point.

The lazy release of dependencies can be traced back to two reasons:

1. Data dependency relationships between tasks are identified only as task:task relationships. No information is stored about the data/parameters that resulted in the dependency relationship.

2. No mechanism exists to notify the runtime system that a dependency parameter has been generated. All the output values/dependency parameters of a task are returned when the task execution ends and the executing process of the task reaches the return statement in the application code.

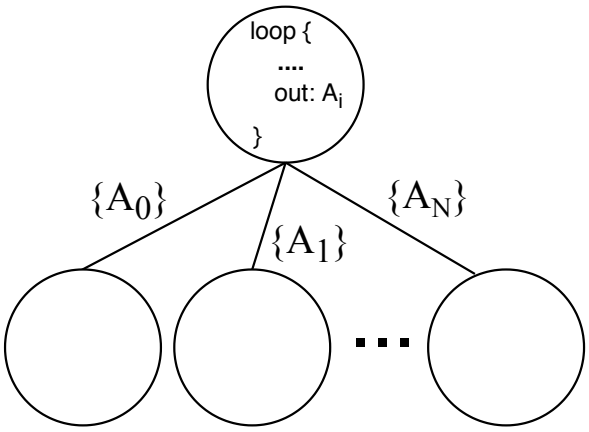
The two reasons mentioned above are intertwined. If data (dependency parameters) are generated at an early point of task execution, it will be returned after task execution ends. Similarly, even if the runtime was made aware of the availability of a dependency parameter, it will not be able to the release the tasks dependent on that parameter.

This approach of releasing dependencies will work fine if a successor task depends on the whole output set of its predecessor or if the predecessor generates the dependency values in a fast rate. However, the limitations of this approach will start to appear and affect applications performance if a successor task has a partial dependency with its predecessor and this predecessor spends time between the generation of each output/dependency parameter. This time spent between calculating different values creates a window for more parallelism opportunities by overlapping the execution of tasks.

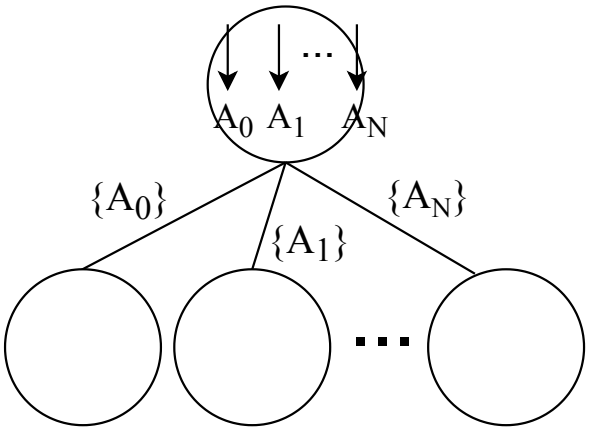
An example where the lazy approach of releasing dependencies can hinder the performance of applications: a 1:N task dependency graph where successors have a partial dependency relationship with the predecessor (the predecessor task generates N output parameters, each one feeding one of the successor tasks). In this scenario, the predecessor task (called *generator*) generates data and its N successors (called *consumers*) each partially depends on some of these data.

Figure 7.2 shows two possible 1:N task graphs. Figure 7.2(a) shows a sequential generator task which uses a loop to generate a value at each iteration. This generator task has N successors where successor $i$ depends on a value generated at iteration $i$. A consumer depending on a data generated at iteration 0 will not be released for execution until the generator task completes its N iterations and finishes its execution.

Similarly, Figure 7.2(b) illustrates a MPI parallel task with multiple consumers where each consumer depends on the output of one of the MPI processes. Regardless of how fast one MPI process generates its data, the execution of consumer tasks is blocked until the data generator task completely finishes execution.




(a) N Consumers Partially Dependent On A Sequential Task




(b) N Consumers Partially Dependent On A Parallel MPI Task

FIGURE 7.2: Examples Of 1:N Partial Data Dependency Relationships

It should be noted that in some cases the example graph in Figure 7.2(a) can be modified to remove the 1:N dependency pattern and replace it with task parallel N:N pattern. However, such modification may introduce overheads due to $N$ tasks creation and management. In addition to that, workflow modification may not be possible in certain scenarios where the producer task cannot be split. For instance, if the predecessor task is executing a legacy code or calling an external binary that cannot be modified. Another case would be if the predecessor task is a streaming task that receives data from a stream and then distributes these data to its successors.

## 7.4 Eager-Release of Dependencies

In this section, we present our proposals for eagerly releasing data dependencies in task-based models. An eager-release of data dependencies will exploit the parallelism possibilities inherent in applications. Following this approach, tasks are launched for execution as soon as their data dependencies are produced without having to wait the predecessor task to completely finish its execution. Hence, the release of tasks for execution is accelerated and more performance can be achieved.

The mechanism for eagerly releasing data dependencies can be achieved by applying two conceptual and design modifications to task-based systems. Each modification offers a

solution to one of the shortcomings in the task-based system design that were addressed in the previous section. These two modifications are:

- Modifying the specification of dependency relationships to include the parameters that caused the dependencies (Section 7.4.1).
- Triggering the release of dependencies without having to wait the predecessor task to finish its execution (Section 7.4.2).

Indeed, these two modifications are intertwined. A task-based system that is aware of the data that caused a dependency between two tasks, will not be very useful if a successor task has to wait until its predecessor completely finishes execution. Likewise, notifying a system that a task has generated an output will not be useful if the system cannot identify which are the successors that require these data.

### 7.4.1 Parameter-Aware Dependencies

The first necessary step to achieve the eager-release of dependencies is changing the dependency specification. It is not sufficient that a tasking system only uses the data (i.e., inputs and outputs of tasks) to establish dependency relationships between tasks. The system needs to be aware of which are the data/parameters that have resulted in dependency relationships between tasks by defining the dependency relationship in terms of the task and its required data from the predecessors. This way, successor tasks will depend on whether the required data has been produced by predecessor tasks instead of depending on whether the predecessor tasks has finished execution.

The approach of identifying dependency relationships in terms of the tasks and their input parameters (outputs of predecessors) instead of the tasks themselves will be referred to as: *Parameter-Aware Dependency*.

Formally, given a DAG $G = (V, E)$ where $i, j \in V$ are vertices representing tasks and $e \in E$ is a directed edge representing a data dependency between the predecessor task $i$ and the successor task $j$. A parameter-aware specification of the dependency relationship $e$ would be expressed as:

$$e = (i, j, d_1, d_2, .., d_n)$$

where $I(s) \cap O(p) = \{d_1, d_2, .., d_n\}$ and $n \leq |O(p)|$.

By adopting a parameter-aware approach for specifying dependencies between tasks, data are not only used by task-models to detect dependencies between tasks, but also, these data are explicitly included as part of the dependencies specification.

Figure 7.3 depicts the parameter-aware specification of the dependency relationships in Figure 7.1. In a parameter-aware model, the dependency relationship between the predecessor task $p$ and the successor tasks $s_1$ and $s_2$ can be expressed as: $e_1 = (p, s_1, out_1)$ and $e_2 = (p, s_2, out_2)$ respectively, as opposed to their previous tasks-only specification $e_1 = (p, s1)$ and $e_2 = (p, s2)$. Now that each dependency relationship is identified by the data that caused it, the system should release tasks $s_1$ and $s_2$ as soon as their data requirement is ready (i.e., produced by task $p$) regardless of the execution state of task $p$.

### 7.4.2 Triggering The Release of Dependencies

In traditional task-based systems, the termination of task execution and releasing the data dependencies are considered as two dependent steps:

- First, a task has to finish its execution, usually identified when the executing process reaches the `return` statement in the task code.

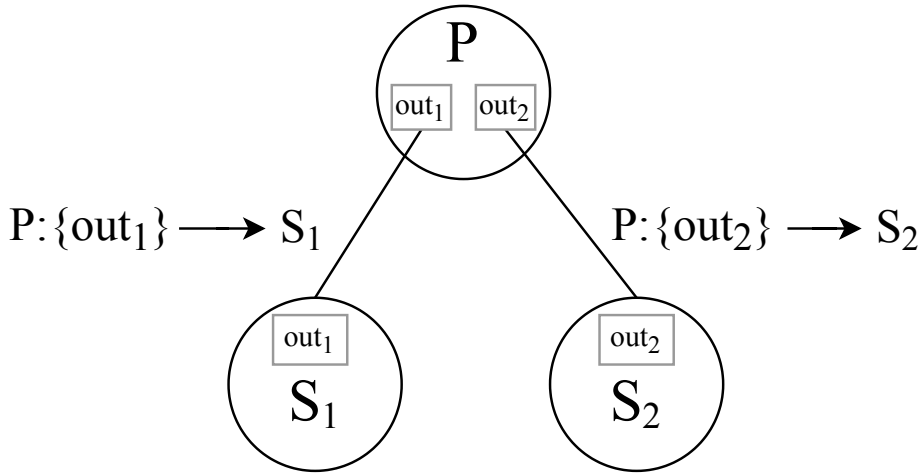



FIGURE 7.3: Parameter-Aware Dependency Relationships

- Then, once the task execution ends, the runtime system receives all the output values included in the `return` statement and starts to release the successor tasks that require them, if these successor tasks do not require more output values from other tasks.

In order to take advantage of modifying the specification of data dependency relationships to be parameter-aware, it is necessary to make task-based systems aware that output data are ready as soon as they are produced instead of waiting until the end of task execution. Otherwise, a parameter-aware dependency will have the same behaviour as a traditional tasks-only dependency specification because all the dependencies will be released after the task completely finishes the execution.

To this end, we propose extending the programming model of task-based systems to enable the notification of the runtime system whenever a task produces output data without having to wait until the `return` statement is reached in the task code. Thus, triggering the release of data dependencies and launching successor tasks that require these data without waiting to the end of the predecessor task execution.

The proposed extension is to include an API that can be used in the task code to allow users to notify the runtime system that output data are ready. Indeed, other approaches can be used such as automatically checking the memory addresses of data. However, our proposed approach will allow for more flexibility in designing applications, because it enables users to plan and optimize the execution of applications by choosing and prioritizing when the dependencies should be released during tasks execution.

Indeed, the runtime-notification API can be used at any point in the task code before the `return` statement. Every time such API call is encountered by the process executing the task code, the runtime system is made aware that data was produced. Once the runtime system is notified that a task has produced output data, it releases all successor tasks that require these data if the rest of their data requirements is satisfied. Meanwhile, the process that is executing the task will resume task code execution as normal until it encounters the `return` statement.

Figure 7.4 shows the changes in the task graph state when releasing output data during task execution. Given a task graph $G = (p, s_1, s_2)$ that has a predecessor task $p$ and two successor tasks $s_1$ and $s_2$. The parameter-aware dependency relationships of this graph can be specified as: $e_1 = (p, s_1, d_1)$ and $e_2 = (p, s_2, d_2)$ where tasks $s_1$ and $s_2$ require data $d_1$ and $d_2$ from task $p$ respectively. Using parameter-aware dependencies and triggering the release of data dependencies, $s_1$ and $s_2$ can be released for execution as soon as their data requirement is produced instead of unnecessarily waiting until the end of task $p$ execution.

At time $T_1$ of task $p$ execution, data $d_1$ is produced and the runtime is notified that $d_1$ is ready so it releases task $s_1$. The same behaviour is repeated at time $T_2$, the runtime releases task $s_2$ because it was notified that task $p$ has produced data $d_2$. The execution of task $p$ continues until it reaches the `return` statement in task $p$ code.

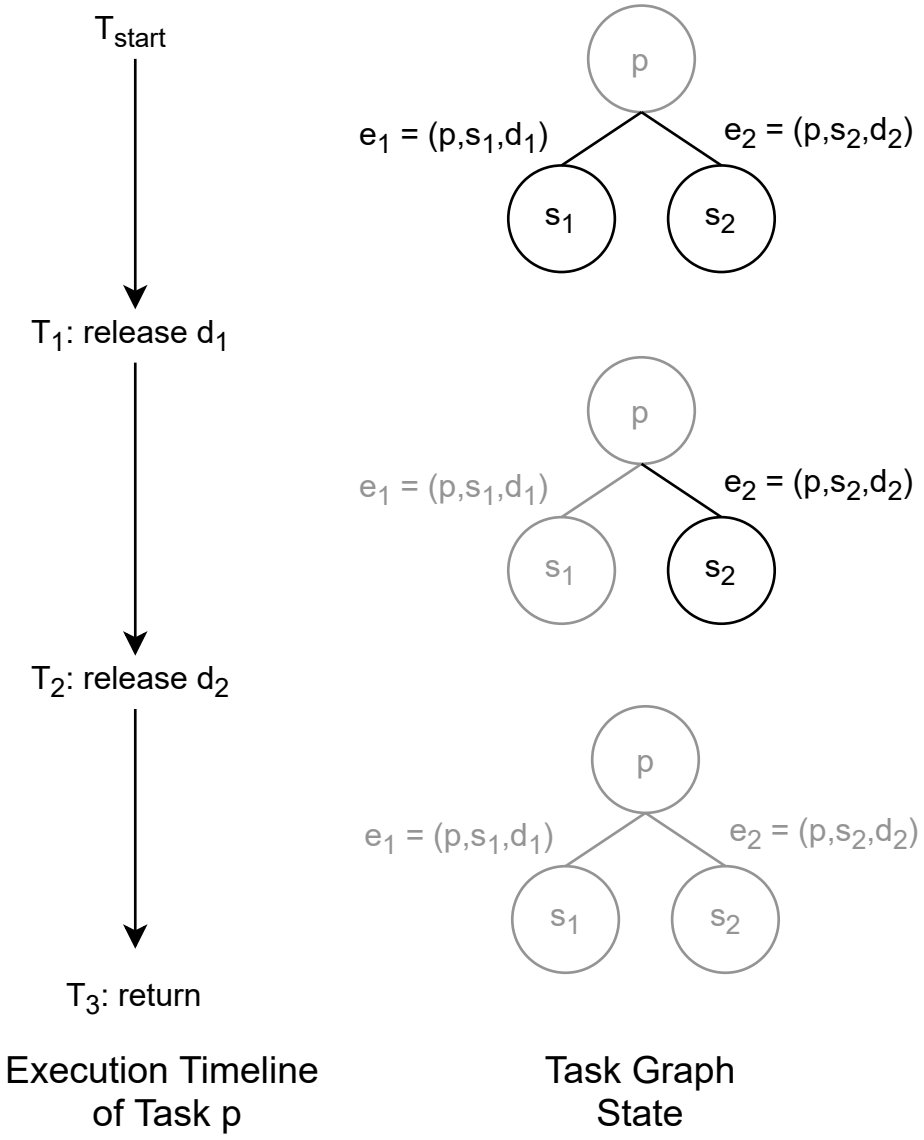



FIGURE 7.4: Releasing Data Dependencies Once Data Are Produced

## 7.5 System Design

Enabling the eager mechanism for releasing dependencies in the PyCOMPSs framework involved implementation efforts in both components of the PyCOMPSs runtime: the master and worker components. Some parts were re-implemented in the master component where the dependencies are managed and released. Also, a new functionality was implemented in the worker component to enable the triggering of data releases in the user code and notifying the master that an output value is ready.

However, it should be noted that it was not necessary to re-implement the complete PyCOMPSs runtime. For instance, the task scheduler was not re-implemented.

This section is divided into two parts: Section 7.5.1 describes the modifications made on dependencies management in the master component to enable parameter-aware dependencies. Section 7.5.2 presents an extension to the PyCOMPSs programming model to enable the notification of ready output values and trigger the release of data dependencies before the end of task execution.

### 7.5.1 Parameter-aware Dependencies

For analysing the dependencies and scheduling a task, the PyCOMPSs runtime uses two main abstractions:

1. *Task*: the main abstraction representing a task. It contains task-related information such as its predecessors and successors, parameters, description and task status.
2. *Parameter*: represents task parameters; their IDs, types and their direction (i.e., whether they are inputs or outputs).
3. *Execution Action*: represents a task instance that is ready for scheduling and execution.

Whenever the COMPSs master receives a task execution request, the *Task Analyser* first identifies if this task has any data dependency relationships with the previously received tasks. The Task Analyser identifies a data dependency relationship between two tasks if an IN or INOUT parameter of a task has the same Parameter id of an OUT parameter of another task.

Once the task analysis has been done, the *Task Dispatcher* instantiates and uses *Execution Action* objects for scheduling. In addition to that, *Execution Action* objects are used to identify which successors should be released after predecessors finish execution. A task may have several *Execution Actions*, for instance, if a task is being called multiple times in a for loop, the PyCOMPSs runtime will instantiate one *Task* object representing the task and as many *Execution Actions* as the number of times that task is called in the for loop.

Every *Execution Action* of a task has a *Predecessor Set* that contains the *Execution Action* IDs of all its predecessors and a *Successor Set* that contains the *Execution Action* IDs of all its successors. Listing 7 shows a pseudo code of the process of building the predecessor set and the successor set of a given *Execution Action*. Predecessor tasks are fetched using a *Task* object. Next, for all the *Execution Actions* of all the predecessor, the predecessor and successor sets are updated.

When the predecessor set of a certain successor task becomes empty, this successor is marked as free of dependencies and released for execution. As noted in Listing 7, data dependency parameters are not included in managing the dependency relationship thus making them tasks-only specified relationships.

In order to enable the parameter-aware specification of dependencies, we introduced a new abstraction: *Task Dependency Parameter*. This abstraction is an extension of the *Parameter* abstraction. It creates a relationship between task parameters and the *Execution Action* objects of the predecessor task. Hence, the runtime can identify which task (i.e., *Execution Action*) should be released for execution when it gets notified that certain parameter(s) have been generated by a running task/*Execution Action*.

Listing 8 depicts the pseudo code for creating parameter-aware dependency relationships. The ID of the *Parameter* object that has resulted in the dependency is stored in all of the *Execution Action* objects of the predecessor and successor tasks. Given a certain task, all of its input parameters are retrieved. The producer of each parameter is identified, then the *Execution Actions* of this producer are retrieved and the corresponding predecessor and successor sets are updated.

**Listing 7:** Parameter-unaware Dependency Management

```
Input: t as the task object, e as its execution action
1 for Task p: t.getPredecessors() do
2     for ExecutionAction e_Predecessor: p.getExecutionActions() do
3         e.PredecessorSet.add(ePredecessor);
4         e_Predecessor.SuccessorSet.add(e);
5     end
6 end
```

As can be noted in Listing 8, instead of managing dependencies only using the *Execution Action* objects that represent running instances of the tasks, the parameter-aware implementation additionally uses parameters IDs. Predecessor and successor sets are updated with a pair of the form: <*Parameter ID*, *Execution Action*>.

**Listing 8:** Parameter-Aware Dependency Management

```
Input: t as the task object, e as its execution action
1 for TaskDependencyParameter param: t.getParameters() do
2     producer = tdp.getProducers();
3     if producer != Null then
4         for ExecutionAction e_Predecessor: producer.getExecutionActions() do
5             e.PredecessorSet.add(param.id, e_Predecessor);
6             e_Predecessor.SuccessorSet.add(param.id, e);
7         end
8     end
9 end
```

Once the *Task Dispatcher* gets notified that a task has generated a data/parameter, the corresponding ID of that parameter will be used to remove the associated predecessor *Execution Action* from all successor *Execution Actions*. *Execution Actions* are launched for execution if they have an empty predecessors set.

### 7.5.2 Triggering Dependencies Release

We extended the programming model of PyCOMPSs to include a new API to indicate that data has been generated, hence, triggering the dependency release mechanism in the master component of COMPSs without having to wait for the task execution to end. This API has the following form:

***compss_ready_value(Data Object, Index)***

This notification API requires two parameters as input: Index which is a unique identifier for the generated data that will be mapped to a parameter ID, and Data Object which is the actual data to be returned.

Figure 7.1 illustrates a sample task code that uses `compss_ready_value()` to return two outputs at two different times of a task execution.

```
1  @task(returns=2)
2  def sample_task():
3      # performs lengthy computation on variable a
4      compss_ready_value(a, 0)
5              .....
6      # performs more computation on variable b
7      compss_ready_value(b, 1)
8
9  if __name__ == "__main__":
10     ...
11     outs = sample_task()
12     for out in outs:
13         calculate(out)
14     ...
```

LISTING 7.1: Sample Task Using *compss_ready_value()* To Release Output Values

It should be noted that in Figure 7.1, the number of output values of the task is specified in the `@task` decorator. Using this information, the programming model and runtime know exactly how many outputs to expect from a task. On the one hand, the runtime will be able to carry out necessary management operations such as instantiating *Parameter* and *Task Dependency Parameter* objects and creating the corresponding dependency relationships. On the other hand, the programming model will allow iterating and indexing the task outputs in the application code. Otherwise, if the number of returns is not specified, the returns of the tasks will be treated as one single object in the runtime and the programming model.

In order to differentiate between different runtime events, we added a *readyValue* message to indicate the event of data value generation. This message is of the following form:

***readyValue (JobID, ParamID)***

Whereas to indicate the event of task termination, the runtime uses the *endTask* message in the following form:

***endTask (JobID)***

Figure 7.5 presents a high level sequence diagram of the different trigger messages and how they are handled by the worker several components. Once the Python worker that executes the task receives a compss_ready_value call, it first maps the index passed in the call to a parameter ID and then signals its corresponding Java executor thread with a *readyValue* message. As soon as the Java executor thread receives a *readyValue* message it notifies the *Execution Manager* that a value was received.

Once the execution of the task code finishes, the control flow goes back the Python worker that triggers an *endTask* message to the Java executor thread to indicate that a task execution has finished. When the Java executor receives an *endTask* message, it notifies the *Execution Manager* that a task execution has finished. Both messages eventually are handled by the *Execution Manager* that notifies the COMPSs master that a certain event has occurred.

On one hand, when the *Task Dispatcher* receives a *readyValue* message, it fetches the *Execution Action* object corresponding to the task ID provided in the *readyValue* message. For a given parameter ID, it releases all the execution actions associated with that parameter

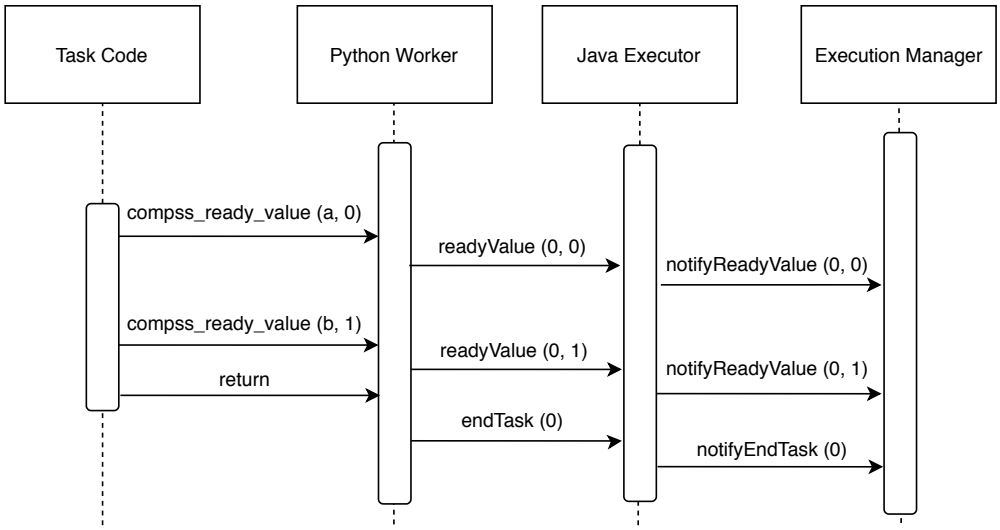



FIGURE 7.5: Worker Execution Workflow Using *compss_ready_value()*.

ID. On the other hand, when the *Task Dispatcher* receives *endTask* message, it releases the remaining successors -if any- and carries out postmortem operations related to the task that has just finished execution.

The serialization of the ready values is done as soon as the Python worker receives the `compss_ready_value()` call and not deferred until the task finishes execution. Figure 7.6 shows a timeline for the different operations carried out in a task releasing two values. Before executing the application code in the task, the python worker process that handles the execution of the task deserializes the inputs of that task so that they can be used inside it. In the lazy approach of releasing dependencies, when the task returns (i.e., the execution of the user code ends), the executing process returns to the Python worker to serialize all the returns of the task. On the other hand, using `compss_ready_value()` to trigger the release of dependencies, the value is serialized as soon as the call is made. Once the value is serialized, the Python worker process starts the workflow of notifying the runtime that a return value is available.

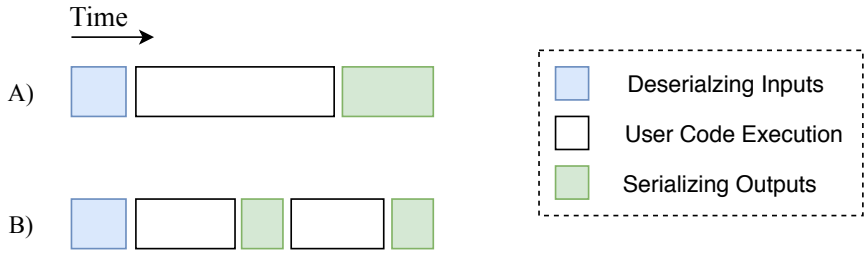



FIGURE 7.6: Operations Carried Out During Task Execution. In A Lazy-release Of Dependencies (A), All The Returns Are Serialized When The Task Execution Ends. Whereas Using *compss_ready_value()* (B), Return Values Are Serialized Once The Call Is Made

## 7.6 Evaluation

In this section, we evaluate our proposal against the default lazy-release of dependencies. In all the experiments, we use the default dependencies release approach of PyCOMPSs as the baseline. We start by describing the infrastructure in Section 7.6.1. Section 7.6.2

presents an evaluation of the overhead associated with using `compss_ready_value` to eagerly release data dependencies in PyCOMPSs. Finally, Section 7.6.3 presents the impact of using eager-release of dependencies on the performance of use cases that exhibit real patterns.

### 7.6.1 Infrastructure Setup

All experiments were run on the MareNostrum 4 supercomputer, located in the Barcelona Supercomputing Center (BSC). A description of the MareNostrum 4 supercomputer can be found in Section 4.5.1. The number of used nodes is specified in the following sections for the overhead experiments and each of the use cases.

### 7.6.2 Overhead Evaluation

This section examines three overhead aspects of using an eager-release approach in PyCOMPSs:

- First, it shows the impact of making `compss_ready_value` calls to return increasing the number of objects.
- Then, it presents the impact of increasing the sizes of returned objects.
- Finally, it examines the network overhead caused by using `compss_ready_value` calls and its effect on the total time.

We developed a benchmark that has a task graph $G = (p, s_1, s_2, .., s_n)$ where $p$ is the predecessor task and $s_i$ where $i \in n$ are successor tasks. The number of successor tasks is equal to the number of objects returned by the predecessor task such that each successor requires only one output of the predecessor task (i.e., $I(s_i) \cap O(p) = \{d_i\}$). The predecessor/generator task generates and returns objects whereas successor/consumer tasks do not perform any computations. Figure 7.7 illustrates the task dependency graph skeleton of this benchmark. In order to be able to test different aspects of the eager-release mechanism, the generator task can be tuned to return different number of objects or different sizes of objects. All the experiments in this section were run two times: one time where the generator eagerly releases dependencies (i.e., eager generator). The second run has a default lazy generator that releases all the dependencies when task execution ends (i.e., lazy generator).

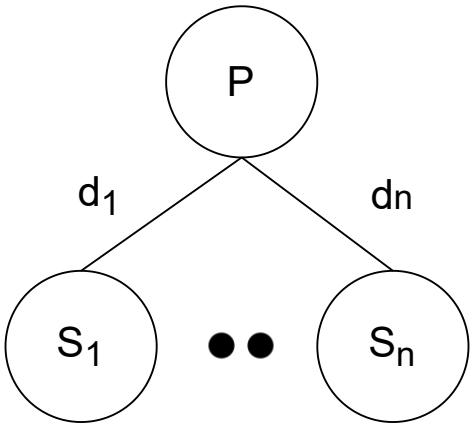



FIGURE 7.7: Benchmark Task Dependency Graph

All results were obtained running on a set of 6 nodes of MareNostrum 4, using one node as master node and five nodes as worker nodes.

#### 7.6.2.1 Impact of Increasing The Number of Returned Objects

For measuring the impact of returning different number of objects using `compss_ready_value`, we ran the benchmark multiple times, each time with different number of returns. In all the runs, all returns are of the same size. The eager generator uses `compss_ready_value` to return an integer each time it is called, whereas the lazy generator returns all values at the end of task execution.

Figure 7.8 shows the impact of increasing the number of returned objects on task time and total time with both dependency release approaches. For fewer number of returns (10 and 100), the task time and total time in both approaches are almost the same. As the number of returns increases in the eager-release approach, the overhead of making the `compss_ready_value` call starts to appear and the task time increases. However, as task time increases in the eager-release case, the total time decreases for larger number of returns in comparison to the lazy-release case.

The time increase in the task due to the overhead of making `compss_ready_value` calls is compensated by the fact that using an eager-release approach, tasks are released earlier for execution. The execution of consumer tasks is overlapped with the execution of the generator task so the total time decreases. Whereas in the lazy-release approach, as the number of returns increases, the amount of tasks to be executed after the generator task increases and total time also increases.

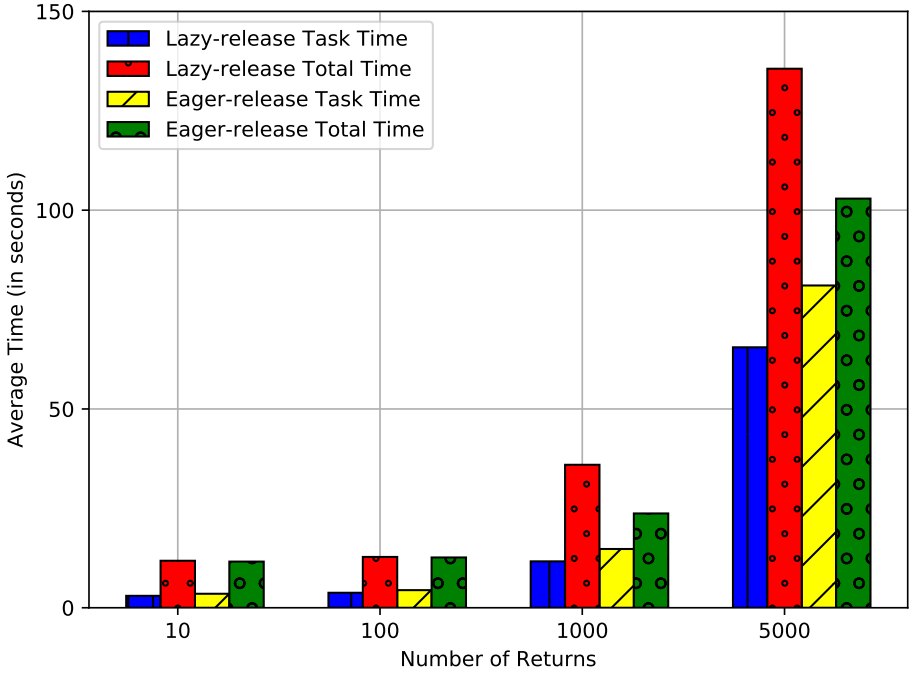



FIGURE 7.8: Impact Of Increasing Number Of Returns On Task And Total Time

#### 7.6.2.2 Impact of Increasing The Sizes of Returned Objects

Next, we measure the impact of using `compss_ready_value` to return multiple Python objects (of type list) with different sizes. This experiment aims to specifically measure the effect of data serialization on task time and total time. To this end, we fixed the number of returns in both the eager generator and lazy generator to 200 returns. We launched five runs, the size of the returned lists increases by one million integer in each run.

Figure 7.9 shows that as the lists sizes increase, no significant difference can be observed on the task time and the total time between the lazy-release approach and the eager-release approach. In addition to that, the gain achieved from using eager-release approach is almost the same for different return sizes. As the sizes of the lists increase, the average time of the generator task increases. However, since the successor tasks are dummy tasks that does not perform any computation, their execution time for different return sizes is almost the same. Thus, the total time increases but the gain remains the same.

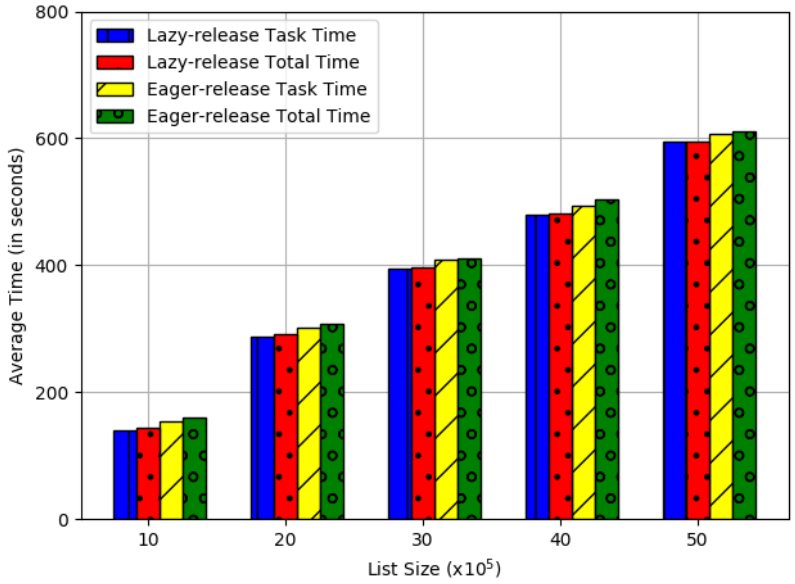
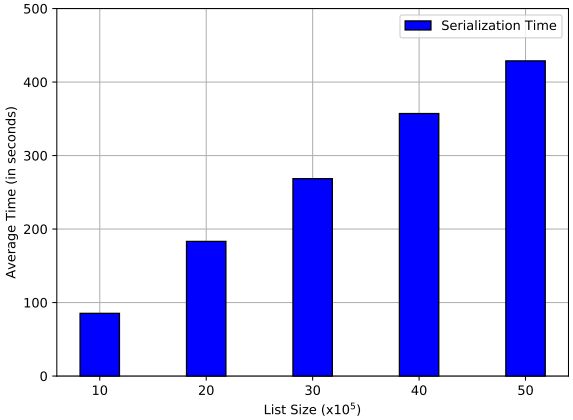
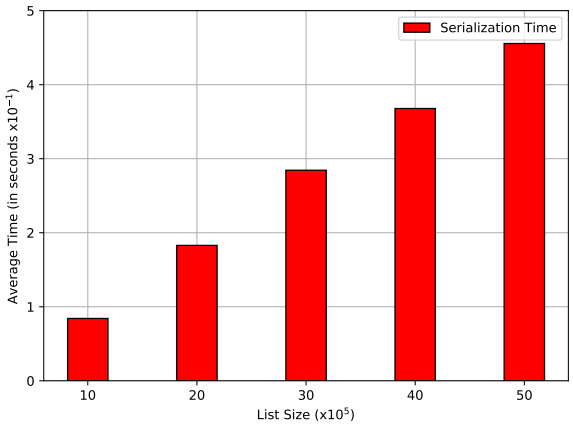



FIGURE 7.9: Impact Of Increasing Sizes Of Returns On Task Time And Total Time

Figure 7.10 offers a closer look at the time spent in serialization in both release approaches. In both cases, as the size of return increases, the serialization time increases. However, in the lazy-release approach (Figure 7.10(a)), the serialization of all the returns is carried out after the task code ends. After the Python worker serializes all the returns, the *endTask* message is triggered. Whereas in the eager-release approach using `compss_ready_value` (Figure 7.10(b)), the serialization is done only for the return value that is passed in the `compss_ready_value` call. As soon as this return value is serialized, a *readyValue* message is triggered for that value.




(a) Serialization Of All Returns In Python Worker After Task Execution




(b) Serialization Of The Return Value In *compss_ready_value* Call

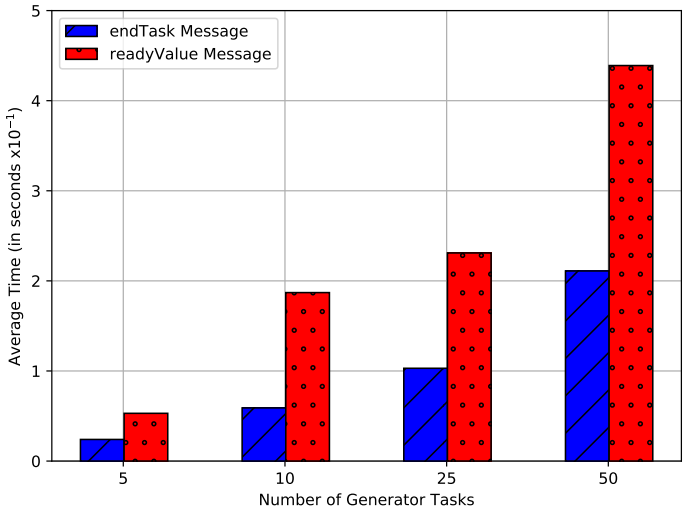
FIGURE 7.10: Serialization Time With Increasing Sizes Of Returns In Lazy-Release Approach And Eager-Release Approach

#### 7.6.2.3 Impact of Network Overhead

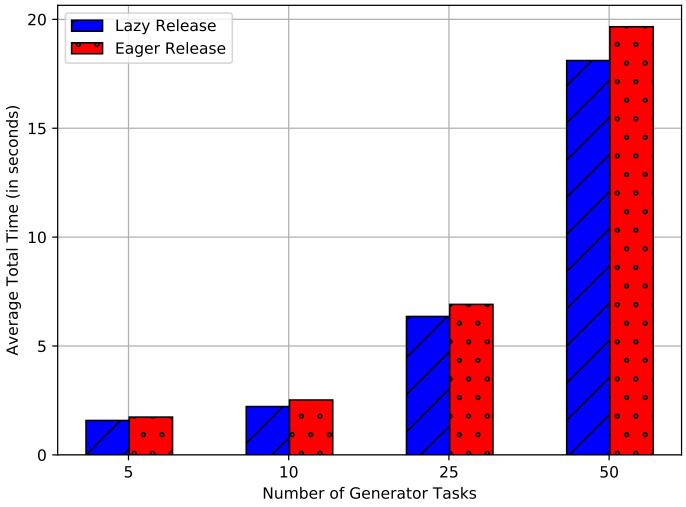
Finally, the experiments in this part measure the network overhead of using an eager-release approach to release dependencies and its impact on the total time. As explained in Section 7.5.2, every time a `compss_ready_value` is called, it uses a *readyValue* message to notify the runtime that the return value passed in the call has been produced. Whereas the lazy-release approach uses only one *endTask* message after the task ends to notify the availability of all the return values of the task.

Several runs were launched, each run has a different number of generator tasks where each generator returns 100 integers. Figure 7.11(a) shows that the average time for receiving a *readyValue* message in the eager-release case and an *endTask* message in the lazy-release case increases as the number of generators increase. It can be noticed that the average time of receiving a *readyValue* message is higher than receiving a *endTask* message. As the number of generators increases, the number of returns increases and *readyValue* messages start flooding the network. This network overhead affects the total time as shown in Figure 7.11(b).

Nevertheless, the effect of network overhead on the total time could be mitigated in real applications where successor tasks spend time performing computation as will be demonstrated in the use cases in the following sections.




(a) Average Time To Receive Release Messages With Increasing Numbers Of Generators




(b) Total Time With Increasing Number Of Generators

FIGURE 7.11: Impact On Network With Increasing Number Of Generators

### 7.6.3 Use Cases

In this section, we present performance results that shows the impact of our proposal on the performance of three different use cases. Each use case exhibits a different rate of releasing dependencies, thus, showing different performance aspects of the eager-release mechanism:

- Section 7.6.3.1 discusses a use case that returns data and releases dependencies at a high rate using a parallel generator task.
- Then, Section 7.6.3.2 presents a use case in which using the eager mechanism for releasing dependencies enables a more efficient application design.

- Finally, Section 7.6.3.3 presents a use case that features a sequential generator task where data are returned and dependencies are released in a time interleaving manner. This use case shows that the eager-release mechanism can be used to optimize the performance of compute-intensive applications.

#### 7.6.3.1 Web Archives Analysis

Web archives are stored in a special format called WARC (Web ARChive) [115]: a file format used for archiving web pages that it is widely used for web crawling and text analysis applications. Each WARC stores several web archives with different sizes.

We developed an application that reads in parallel the records of WARC files and calculates the term frequency (TF-IDF) of each record. Figure 7.12 shows the PyCOMPSs skeleton graph of the application. A generator Native MPI task (more details on Native MPI tasks are discussed in Chapter 6) reads N records in parallel using MPI. For each record, there are two consuming tasks that will carry out a TF-IDF operation. Since WARC records are of different sizes, MPI processes will spend variable times reading them, hence their successors are released at different times. To run the experiments of this use case, we used datasets from CommonCrawl [28], which is a nonprofit website that freely provides web archives and web crawl data.

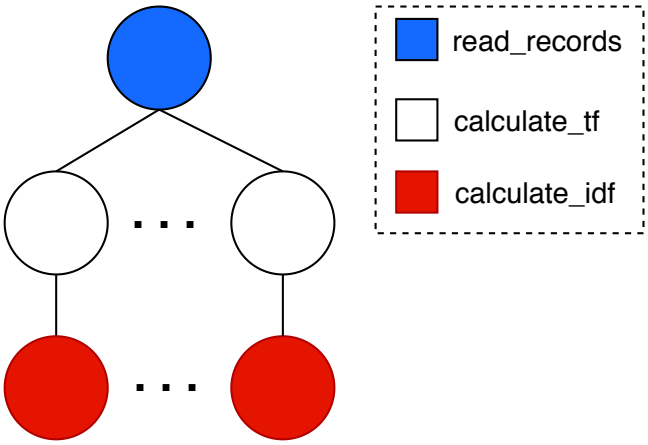



FIGURE 7.12: Skeleton Graph of WARC Analysis

For running the experiments of this use case, we used dedicated nodes for doing I/O (called I/O nodes) and dedicated nodes for doing computation (called compute nodes). By using the `ProcessorName` propriety of the `@constraint` decorator in the application code, PyCOMPSs schedules each task to matching resources. This application is annotated in such a way that Native MPI I/O tasks will be scheduled to I/O nodes and compute tasks will be scheduled to compute nodes. This way of organizing the infrastructure mitigates the interference levels between I/O and compute tasks which leads to better overall performance as proven in previous I/O research [2], [63].

To show the impact of using eager-release of dependencies when using a parallel MPI task, we used 6 I/O nodes and 8 computing nodes. One I/O task launches 288 MPI processes across the I/O nodes. Each MPI process reads a chunk of records from a WARC file that contains 6000 records. The number of records to be read is divided equally across the MPI processes with the MPI process of last rank reading any remainders. For each record, TF-IDF computation tasks are scheduled to the computing nodes. Figure 7.13 shows the distribution of record sizes of the sample WARC file used in this experiment. Records are of

different sizes so MPI processes will take variable amounts of time reading them and their dependencies will be released at different times.

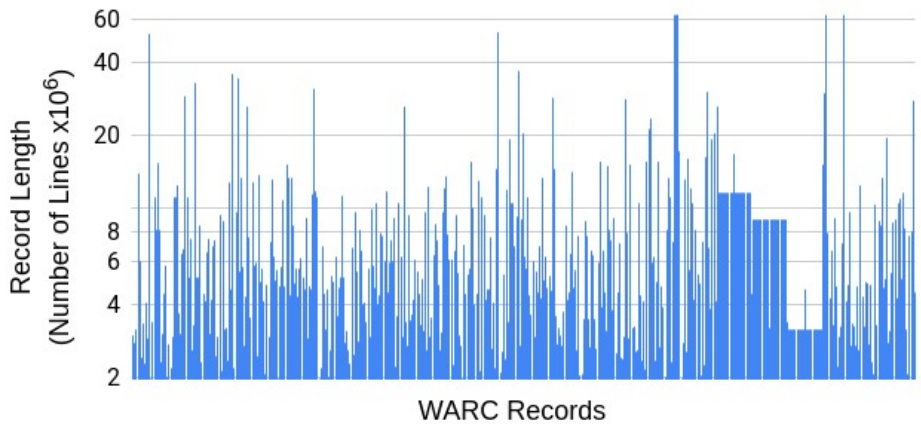



FIGURE 7.13: Distribution Of Records Lengths In A WARC File

Figure 7.14 presents the performance results of the application when reading different number of records. Using the eager dependency release achieves performance improvement that can reach up to 35% in total time compared to the lazy release run. In the lazy dependency release, all the consumer tasks wait until the generator task ends to be scheduled and executed. Once the read task ends, consumer tasks overwhelm the computing resources and hence, most of them spend more time waiting until there is an available execution resource. On the other hand, with eager release of dependencies, consumer computation tasks are released as soon as their data dependency is read and computation overlaps with I/O. Therefore, better performance is achieved as consumer tasks start execution while the read task is still running. Hence, avoiding resource and system contention that happens in the lazy dependency release case when the reading task ends.

Moreover, looking closely at Figure 7.14, the performance gain of the eager-release approach varies between the read time (which is the maximum achievable performance improvement) and the lazy-release time (which is the minimum achievable performance improvement). In both scenarios, the performance of an eager-release will never be better than the reading time and will not be worst than the lazy-release performance.

To better understand the results of Figure 7.14, we refer to Figure 7.15 which shows two screenshots of the execution traces of one of the experiments using the lazy dependency release and eager dependency release. To generate execution traces, the runtime of PyCOMPSs uses the Extrae tool [34] to instrument the start and end of each task. At the end of the execution, the generated trace file can be viewed using the Paraver tool [85]. Both screenshots in Figure 7.15 has X-axis that represents time and Y-axis that represents execution threads; each thread executes one task at a time. The execution traces in Figure 7.15 show a blue MPI read task that spans MPI processes across I/O nodes. Each record is processed by a `calculate_tf` white task then a `calculate_idf` red task on the compute nodes.

As it appears in Figure 7.15, in the case of eager-release, records of small and medium sizes can be executed as soon as they are read without having to wait the `read_records` task to read all records. While the MPI processes of the `read_records` task spend more time reading larger records, the eager-release execution releases the dependencies of small

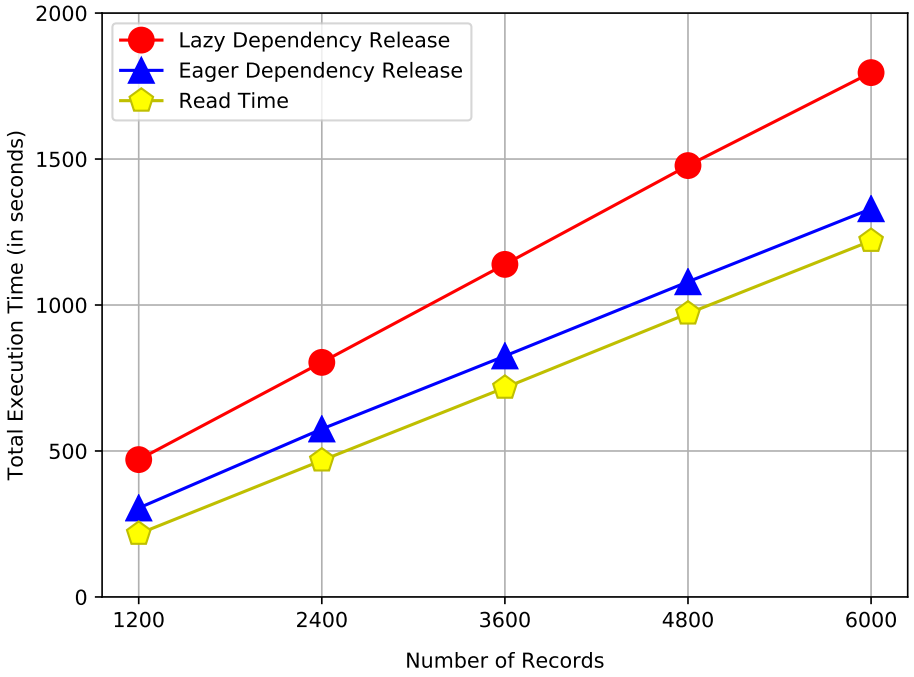



FIGURE 7.14: Performance Results With Increasing Number Of Records

and medium size records and starts their processing. On the contrary, in the case of the lazy-release execution, the processing of the small and medium size records is blocked until the `read_records` task reads all the records. After the `read_records` task finishes execution, all dependencies are released and `calculate_tf` and `calculate_tf` tasks overwhelm the execution resources, hence, tasks have to wait for available resources to be start execution.

Since data returns are done in parallel, the amount of performance improvement achieved in the eager-release execution is proportional to how fast the parallel processes will return a value and release its dependency. In the WARC analysis application, the performance improvement depends on the number of small and medium sizes records compared to the number of larger records. As this number increases, more performance improvement will be achieved using eager-release of dependencies. Nevertheless, if all records have the same size, the MPI processes will spend almost equal time reading them and their dependencies will be released at the same time. Consequently, their execution will not overlap with the reading task and the eager-release approach will behave like a lazy-release one.

For testing the scalability of using eagerly releasing dependencies in a parallel task, we tested this application with different number of nodes and different workloads. For the scalability experiments we used a WARC file which contains 24000 record of different sizes. For running the strong scalability experiments, the workload is maintained fixed by using the same number of I/O nodes and increasing the number of compute nodes. Each I/O node runs one I/O native MPI read task that launches 48 MPI process per node for a total number of 240 MPI process across all the I/O nodes. Each MPI process reads 100 record from the WARC file. For each record, TF-IDF compute tasks are launched on any of the compute nodes.

As for the weak scalability tests, the workload was changed by changing the number of I/O nodes in each experiment. By changing the number of I/O nodes, the number of I/O tasks changes and the workload changes because the amount of records to be read changes. In these experiments, each I/O node hosts a parallel MPI read task that launches

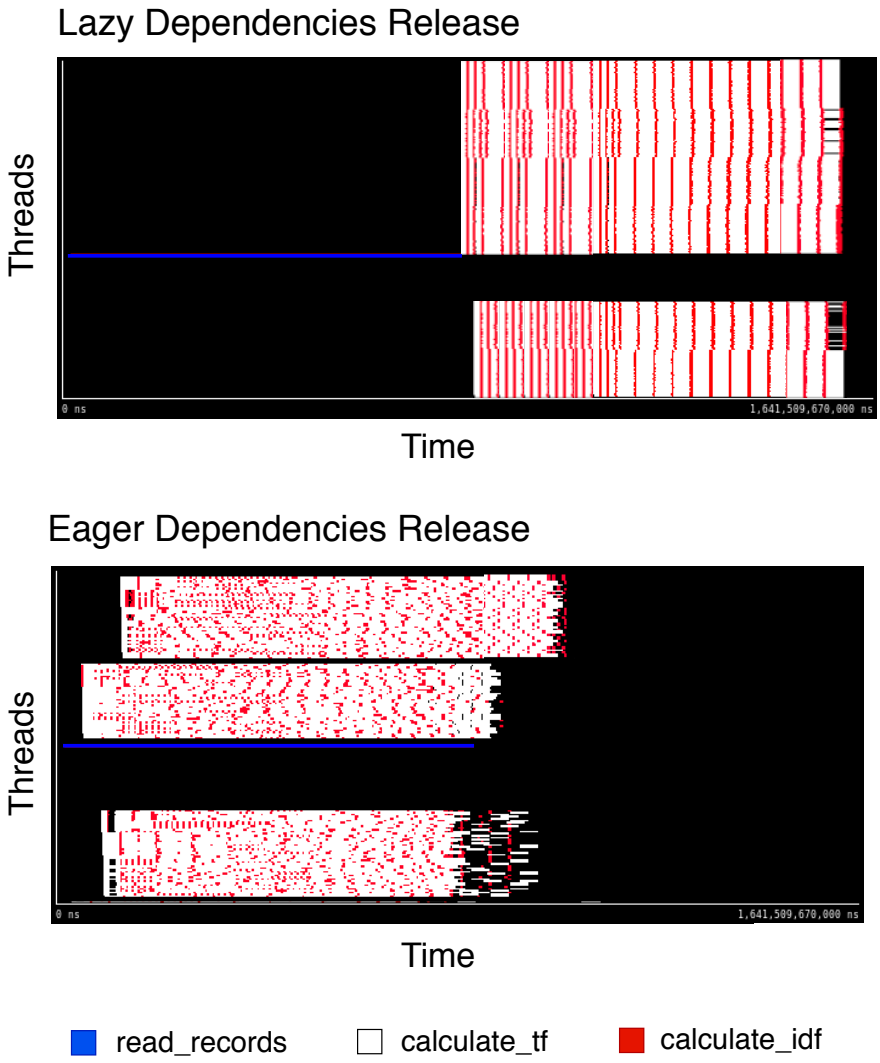
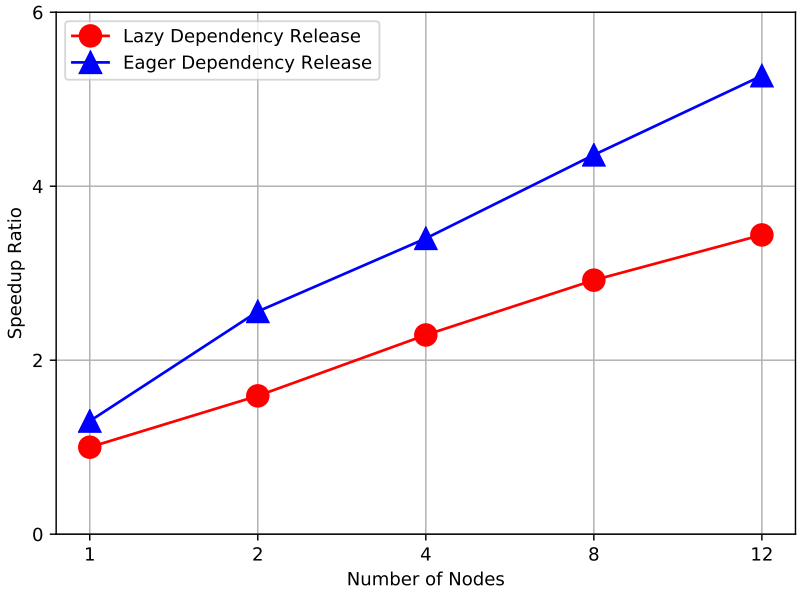



FIGURE 7.15: Execution Traces Of WARC Analysis Application (Top: Trace Of The Lazy Dependency Release; Bottom: Trace Of The Eager Dependency Release)

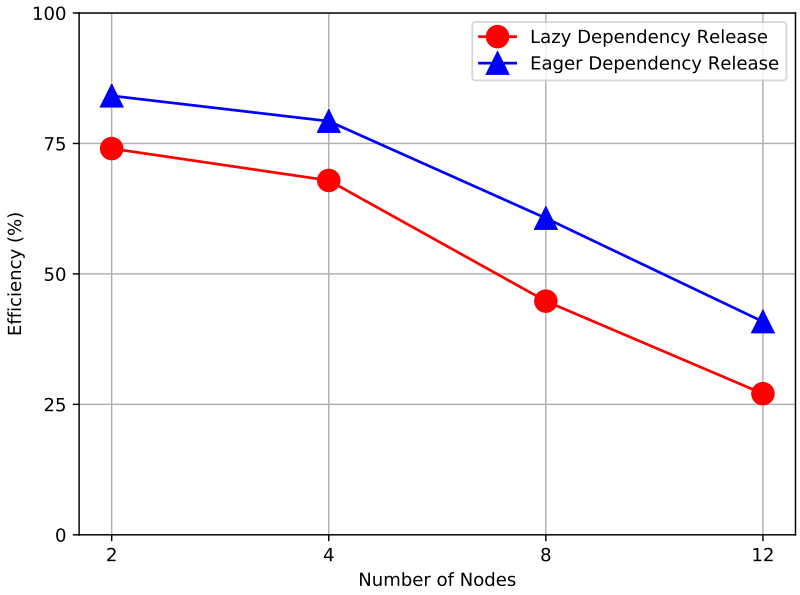
48 MPI process per I/O node, each MPI process reads 100 record from the file. In the eager-release case, once MPI process reads a record, it uses `compss_ready_value` to release the successor compute task.

Figure 7.16 presents the scalability results of the application. In Figure 7.16(a), an eager-release approach achieves better strong scalability than the lazy-release approach. With eager-release of dependencies, as the number of computing nodes increases, more resources are available to satisfy the consumer tasks that are eagerly released. Given that the amount of work for each MPI process is different so that dependencies will be released at different times allowing overlapping computation, and thus, performance improvement.

Furthermore, Figure 7.16(b) illustrates the weak scalability of the application. An eager-release approach also achieves better weak scalability than the lazy-release approach. The more data returned and released in parallel, the more workload the components of the system have to handle. It should be noted that as the data are returned in parallel, dependencies are released at a higher rate and the overhead of the system increases resulting in a similar trend but better performance compared to the lazy-release case.




(a) Strong Scalability




(b) Weak Scalability

FIGURE 7.16: Scalability Results of WARC Analysis Application

#### 7.6.3.2 Pairwise Sequence Alignment

In the fields of life sciences and bioinformatics, pairwise sequence alignment (or mapping) is used to identify regions of similarity between two DNA or RNA sequences. Sequence alignment is carried out by mapping tools that take two inputs: an unknown (or target) sequence and a known sequence (or a reference) with the goal of identifying whether both sequences have a structural, functional or evolutionary relationship.

Sequence files are usually of big sizes as they contain a large number of DNA or RNA

fragments, called *Reads*. Therefore, an out-of-core approach is used to align the target sequence files to the reference file since the whole target sequence file is too big to fit in memory. In this approach, the target file is processed in parts: the application reads a part that can fit in memory then processes it before reading the next part.

A different solution to this problem is to stream the target sequence file such that the mapper tool processes streams of reads. This way, the cost associated to keeping large number of reads in memory is eliminated. Our proposed approach of eagerly releasing data dependencies makes this solution possible in a task-based execution context: the processing of reads can start as soon as a read or group of reads are received from the I/O stream. Otherwise, using a streaming solution with lazy dependencies release would not be effective as the reads would still need to be kept in memory. Indeed, this solution approach can be generalized to solve any problem that has inputs of sizes bigger than the available memory or any problem that has a workload that exhibits high memory requirements.

We developed a sequence alignment PyCOMPSs application that has the task skeleton depicted by Figure 7.17: a `get_reads` task loads reads from a sequence file and distributes reads to `align` successor tasks, each calling a mapper tool on its input reads. In order to show the capabilities of the eager approach for releasing dependencies, we compared two versions of the application in the experiments:

- An out-of-core version in which the application follows the pattern: `get_reads` task reads part of the target file that can fit in memory, then after it finishes loading that part, `align` tasks are released in a default lazy dependencies release fashion. This pattern gets repeated until the whole target file is processed.
- A streaming version in which the `get_reads` task opens an I/O stream to load the reads from the target sequence input file, then eagerly releases an `align` task when a read or group of reads has been received from the stream.

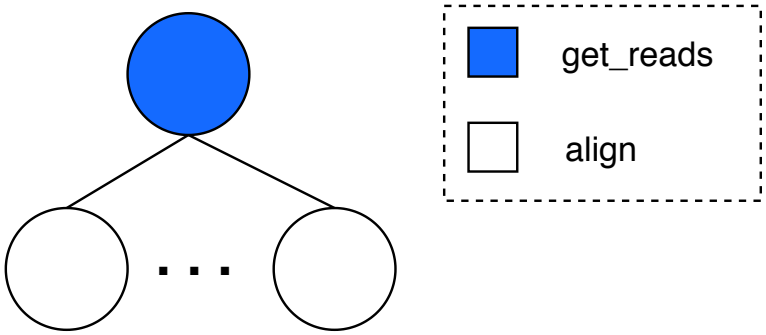
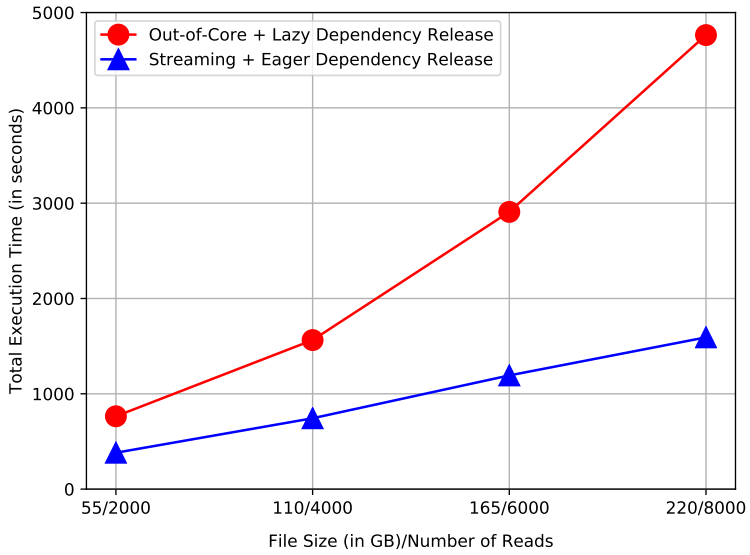



FIGURE 7.17: Skeleton Graph Of The Pairwise Sequence Alignment Application

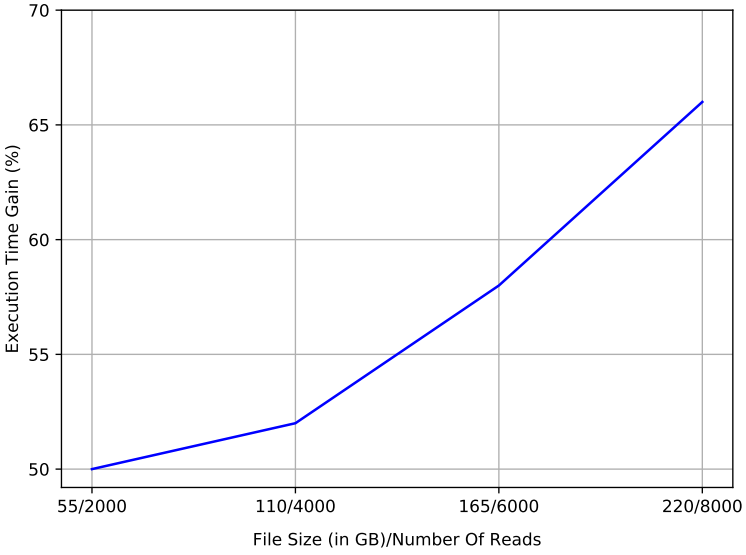
For running the experiments, we used different target sequence input files of increasing sizes (55 GB-220 GB). A target sequence file of 55 GB has a total of 685,388,928 reads. No matter what is the size of the target input file, each `align` task processes almost 342,694 reads. The `align` tasks use the popular Burrows-Wheeler Aligner (BWA) [58] to align their input reads to a subset of the human genome reference (HG38). The input files and the human genome reference are publicly available on the FTP servers of the European Bioinformatics Institute [30] (EMBL-EBI) and the Broad Institute [37] which are well-known resources for providing sequencing data (e.g., sample sequences, genome references, etc.). All experiments used 8 nodes of the MareNostrum 4 supercomputer for tasks execution.

Figure 7.18 shows the significant improvement that can be achieved when streaming the input and eagerly releasing aligning tasks compared to reading parts of the file then

releasing all the aligning tasks. As the number of reads is doubled every experiment (file size is doubled), the number of reads and the number of the `align` tasks that process each read also doubles. In the case of the out-of-core version, the number of `get_reads` tasks doubles as the file size doubles. For instance, in the case of using an input of 55 GB, one `get_reads` tasks is used to load the whole file in memory, whereas with an input of 110 GB, two `get_reads` tasks are used to read two parts of the file. In order to guarantee the memory requirements of the `get_reads` tasks, we constrained their execution using the `memorySize` argument in the `@constraint` decorator of PyCOMPSs. On the contrary, in the streaming case, only one `read_tasks` is used regardless of the input file size, also without specifying any memory constraints for the task execution.




(a) Total Execution Time With Increasing File Sizes/Number Of Reads




(b) Execution Time Gain With Increasing Number Of File Sizes/Number Of Reads

FIGURE 7.18: Performance Results With Increasing File Sizes/Number Of Reads

As shown in Figure 7.18(a), as the target file size and the number of reads increases,

the performance improvement gained by in the case of input streaming and eagerly releasing successor tasks increases. The performance improvement achieved in the case of input streaming and eagerly releasing the `align` tasks is possible because the cost of repeatedly reading parts of the input file in memory is eliminated. The memory requirements of the `get_reads` tasks in the out-of-core version also prevents more `align` tasks to run in parallel. Moreover, similar to the previous use case, eagerly releasing the `align` tasks accelerates the execution as tasks execution is overlapped and less tasks overwhelm the infrastructure at the end of `get_reads` task execution. Whereas in the out-of-core and lazy release implementation, all `align` tasks are released after the end of `get_reads` execution. Hence, they overwhelm the infrastructure and each task has to wait more time for computing resources to be available.

Figure 7.18(b) shows the performance gain in the application. As the file size (i.e., the number of reads) increases, the gain achieved by using input streaming and eager release of dependencies increases.

#### 7.6.3.3 Domain Decomposition of Geometrical Shapes

Due to the time and memory required for solving problems in mechanical and engineering disciplines, domain decomposition techniques are applied to split a global domain into sub-domains. Later, a Partial Differential Equation (PDE) is performed on each sub-domain in a parallel manner. In these problems, sub-domains are not of equal dimensions because of the irregular geometry of the domain.

Therefore, this problem is a good candidate to study the impact of eager dependencies release as implemented in PyCOMPSs. We developed an application that decomposes a 2D mesh using an iterative solver into several sub-domains and then it applies a partial differential equation on each sub-domain. It should be noted that this application and its workload pattern corresponds to a real workflow used in the field of mechanical engineering modeling. The iterative solver used for creating sub-domains is called the *FETI Method* [35], which is a well-known domain decomposition method in the field of numerical analysis. In addition to that, the pattern that this application mimics is used in the modeling experiments of the EXPERTISE project [103], which is a European multidisciplinary project that aims at the modeling of turbine components.

Figure 7.19 presents the PyCOMPSs task graph skeleton of the application. A domain decomposition generator task returns N sub-domains. Each sub-domain will be processed by one consumer successor task, where each consumer will apply a PDE on its input sub-domain.

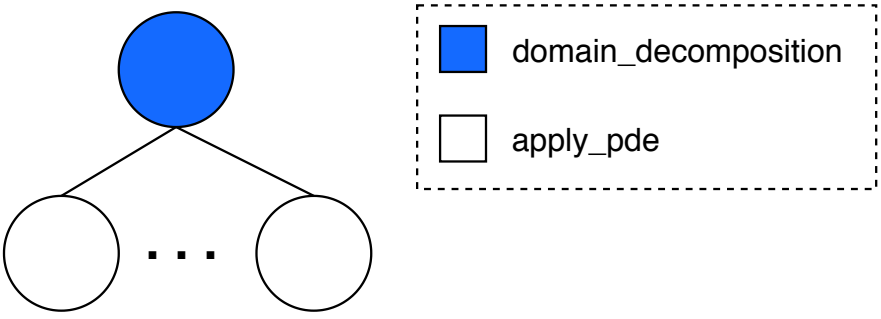



FIGURE 7.19: Skeleton Graph Of Domain Decomposition Application

We have measured the performance of the application with eager dependency release and lazy dependency release implementations on 8 nodes of MareNostrum 4 for executing

the tasks of the application. Since the number of sub-domains returned by the generator task has an important effect on the availability of work (and hence performance), we run the experiments with a variable number of data returns per generator task. In each experiment, we used 4 domain decomposition tasks. Each task decomposes one domain and returns several sub-domains. In addition to that, we maintained the number of consumer tasks equal to the number of sub-domain releases (i.e., in all experiments, one consumer task is used to process one sub-domain).

Figure 7.20 shows the execution time of running the application with eager dependency release and lazy dependency release and the performance gain achieved with increased number of sub-domains. As shown in Figure 7.20(a), as the number of sub-domains returned per generator task increases, the performance improvement obtained by the eager dependency release mechanism increases. This is because in the lazy-release case, as the number of sub-domains increases, the number of consumer tasks to be executed after the generator task ends increases. In this case, all consumer tasks require execution resources at the same time, thus tasks spend more time waiting for available resources.

Whereas in the eager-release case, as soon as the runtime is made aware of a generated sub-domain, it releases the consumer task that has a data dependency with that sub-domain. Hence, consumer tasks that are ready for execution do not spend as much time waiting for execution resources as they do in the lazy-release case when the generator finishes execution.

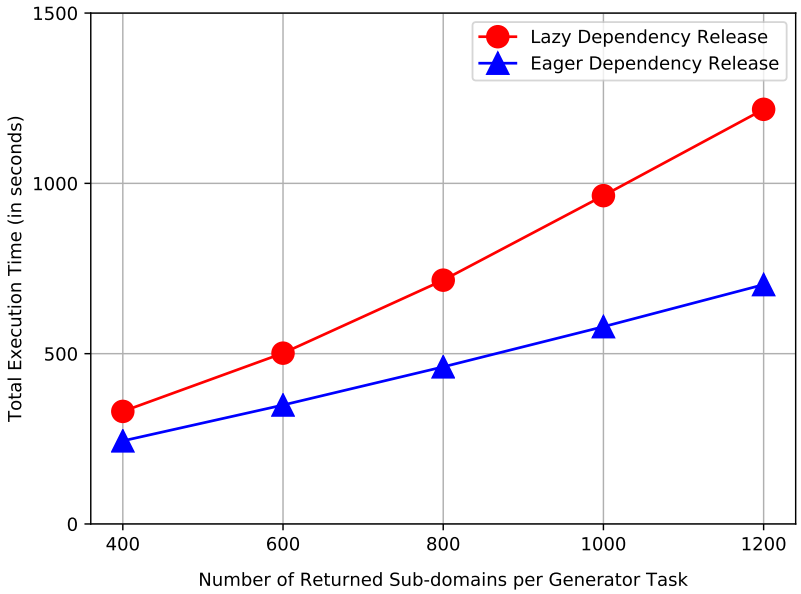
Figure 7.20(b) presents the performance gain achieved using eager-release as the number of returned sub-domains increases over using the default lazy-release of PyCOMPSs. As the number of returned sub-domains increases, the execution of the eagerly-released consumer tasks starts earlier and overlaps with the execution of generator task. Therefore, unlike lazy-release case, less consumer tasks are left to be executed after the generator task ends. Hence, tasks spend less time waiting for a free resource.

It should be noted that the number of consumer tasks with respect to the number of available resources have an impact on the performance gain. On one hand, as the number of returned sub-domain increases, consumer tasks overwhelm the available resources after the generator task end in the lazy-release case. Consequently, consumer tasks spend more time waiting for free resource and the execution time increases. On the other hand, when the number of returned sub-domains decreases with respect to the number available resources, performance gain of using eager-release decreases over using a lazy-release approach. This is because there are enough resources to execute tasks so consumer tasks will wait less time to be executed.

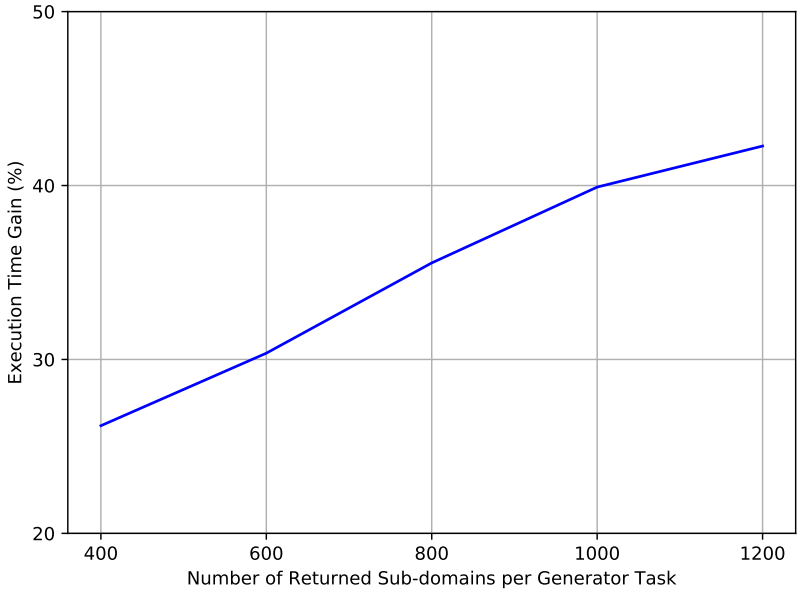
Figure 7.21 presents the scalability results of the application with both dependency release approaches. For the strong scalability experiments the number of returns is kept fixed while the amount of worker nodes is increased. On the other hand, for the weak scalability experiments, the data set is increased in the same proportion as the amount of worker nodes.

For the strong scalability experiments, we used 10 domain decomposition generator tasks, each task returning 800 sub-domains. This workload is fixed for all the number of nodes. As for the weak scalability experiment, the number of sub-domains increases from 800 sub-domain to 9600 sub-domain as the number of workers increases. For example, for one worker node, we launched one domain decomposition generator task that returns 800 sub-domains. For two workers, we launched two domain decomposition tasks each returns 800 sub-domains and as the number of nodes increased the workload is increased in the same pattern so that each worker would perform the same amount of work.

Figure 7.21(a) shows that eager-release approach achieves better strong scalability. With fewer number of nodes, the performance improvement of both approaches is somehow close

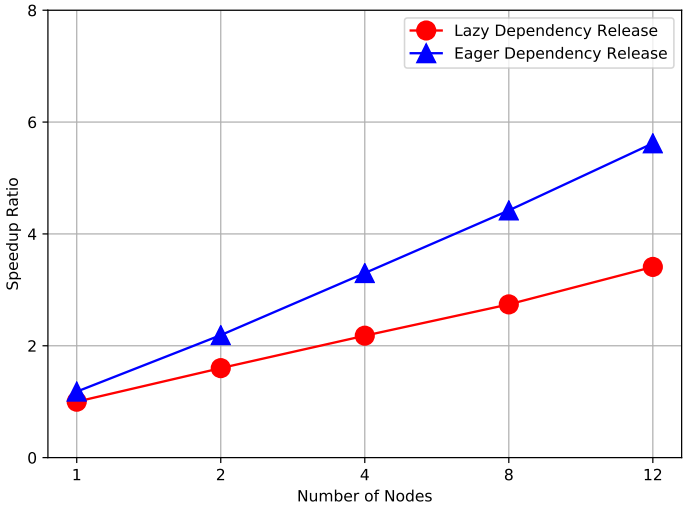



(a) Total Execution Time With Increasing Number Of Sub-domains

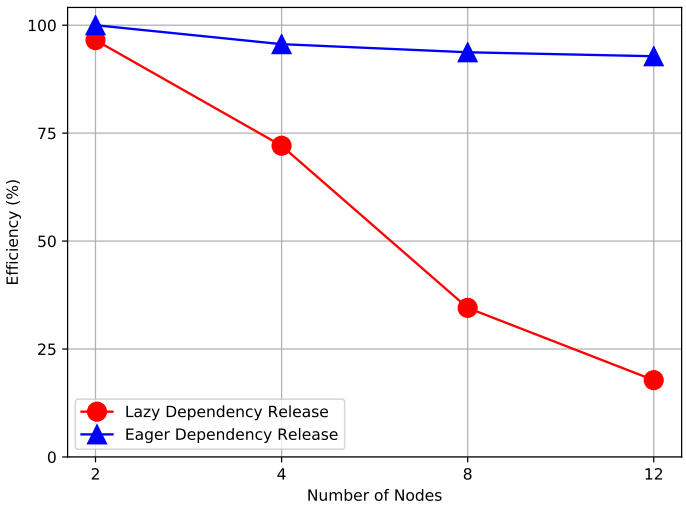



(b) Execution Time Gain With Increasing Number Of Sub-domains

FIGURE 7.20: Performance Results With Several Number Of Sub-domains Per Generator Task




(a) Strong Scalability




(b) Weak Scalability

FIGURE 7.21: Scalability Results Of Domain Decomposition Application

because there is no enough resources to execute ready consumers. However, as the number of nodes increases, the eager-release approach achieves more and more performance improvement since there are more resources to execute ready tasks.

Moreover, in Figure 7.21(b), unlike the lazy-release approach, an eager-release approach shows a more steady efficiency trend with increasing workloads. As the workload increases in the lazy-release approach, the load on the runtime components increases since all the tasks need to be processed at the same time. However, using an eager-release approach, the workload does not create a bottleneck at any of the runtime components since tasks are handled (and their successors are dispatched) at interleaving time intervals. It should be noted that the gain achieved with increasing number of returned sub-domains hides the overhead of `compss_ready_value` and the network overhead.

## 7.7 Discussion

This chapter presented a proposal to improve the performance of task-based executions by modifying how data dependencies between tasks are perceived and managed in task-based systems. The proposal is to eagerly releasing data dependencies instead of being delayed until the predecessor tasks finish execution. Using this approach for releasing dependencies enables higher levels of performance by exploiting the inherent parallelism in the applications and overlapping tasks execution.

In this chapter, we implemented an eager mechanism for releasing dependencies in the PyCOMPSs framework. This is achieved by introducing two related modifications to the design of the system:

1. Changing the management of dependencies to be identified not only in terms of tasks, but also in terms of the dependency parameters that caused the dependencies. This approach enables the release of tasks as soon as their data dependencies by making the dependency parameters part of the dependency relation.

2. Through the use of `compss_ready_value` API call, dependencies release can be triggered from the application code as soon as a return value is ready instead of waiting to the end of task execution when the executing process reaches the `return` statement.

This proposal was evaluated against real and artificial workloads and it has demonstrated that eagerly releasing dependencies can achieve better performance than using the default lazy approach of releasing dependencies. In addition to that, under certain conditions, the performance gain increases as the workload increases. Using an eager-release mechanism, the execution of tasks overlaps as they are released earlier for execution as opposed to delaying their execution until their predecessor tasks finish execution.

Moreover, the performance gain achieved in an eager-release approach depends on the frequency of output returns. Performance gains are guaranteed when data are returned and dependencies are released in a time interleaving manner allowing overlapping execution. However, when the rate of returns increases, like in the case of a parallel task that returns multiple data dependencies, the performance gain diminishes. As the time between returning values decreases, an eager-release approach follows a similar trend to a lazy-release approach and the overhead of the system starts to affect the performance.

Enabling the eager-release of dependencies allow performance improvement opportunities through I/O-compute overlap. Successor compute tasks in read-intensive applications can be released for execution as soon as their data requirement has been read, without having to wait for the whole dataset to be read. Moreover, the eager mechanism for releasing dependencies makes it possible to design I/O streaming applications, for instance reading

data that cannot fit-in-memory as a stream (e.g., record after a record) and constantly releasing the compute tasks and freeing the memory once a record has been read.

In addition to the performance benefits for I/O intensive use cases, the eager-mechanism for releasing dependencies has shown performance improvements for compute intensive applications. For example, applications where a task is constantly generating or distributing data to successors.

As future work, we plan to investigate approaches to automatically detect the generation of data instead of manually using an API call in the task code without affecting the flexibility of applications design. In addition to that, we plan to extend the eager-release mechanism to support releasing dependencies of nested tasks.

# Part III

# Conclusions And Future work

# Chapter 8

# Conclusions And Future Work

## 8.1 Conclusions

This thesis contributes to the optimization of I/O intensive applications. It proposes programming model abstractions and techniques that help to mitigate the I/O bottleneck which prevents achieving more performance in nowadays applications. Throughout this thesis, we introduced various concepts to task-based programming models in order to achieve more I/O and total performance on modern heterogeneous and distributed large-scale infrastructures. Our prototype is capable of significantly improving I/O and total performance of applications by providing the following capabilities: (i) improving I/O performance by mitigating I/O performance problems such as I/O congestion. (ii) exploiting the heterogeneity of modern storage systems, (iii) increasing parallelism by supporting hybrid executions (a combination between tasks and MPI inside tasks), (iv) overlapping I/O and computations. These capabilities are exposed in a transparent and abstract manner to application developers by providing simple annotations that enable high performance execution without increasing programming complexity. In the next following paragraphs, we provide an overview of the conclusions that are discussed at the end of each chapter in the Contributions part of this thesis (Part II).

**Chapter 4** proposes the separation of handling I/O and computations at programming model level. This separation allows increased levels of parallelism by exploiting applications compute-I/O patterns to overlap the execution of I/O with computation. In addition to that, it enables the optimized scheduling and execution of I/O and compute workload. We refer to such concepts as *I/O Awareness*. In order to achieve I/O awareness, we introduce the I/O tasks abstraction. I/O tasks are tasks that only perform I/O operations as opposed to compute tasks that perform computations. I/O tasks are declared by the use of the `@IO` annotation. Moreover, we propose different approaches for solving the problem of I/O congestion that negatively affects I/O performance. Such goal is possible by constraining tasks execution, hence, the following mechanisms are introduced: (i) using static storage bandwidth constraints, where users estimate and set storage bandwidth constraints at application development time. (ii) using auto-tunable constraints. Auto-tunable constraints are automatically set and tuned storage bandwidth constraints. To identify a suitable constraint, the programming model launches a learning phase where it collects performance metrics about I/O tasks performance when different numbers of concurrent I/O tasks are running. Then, it applies these performance metrics to an objective function to estimate the optimal constraint that results in improved I/O and total performance. The evaluation section shows that adopting the aforementioned techniques resulted in a significant performance improvement of different applications with different I/O workloads. In addition to that, the evaluation section studies the factors that impact the duration of the learning phase and its effects on applications total performance.

**Chapter 5** builds on the concepts and abstractions that are introduced in Chapter 4 to take advantage of the heterogeneity of modern storage systems to improve I/O performance. In order to maximize I/O execution in a heterogeneous storage infrastructure, we describe in this chapter some capabilities that we refer to as *Storage heterogeneity Awareness*. A storage-heterogeneity aware programming model supports the following features: (i) abstracting the heterogeneity of the storage devices and exposing them as one hierarchical storage resource, such that the highest storage device would provide the highest bandwidth and the lowest storage device in the hierarchy is the one providing the lowest bandwidth. Therefore I/O tasks would be scheduled first to higher layers, therefore, I/O task parallelism is increased without causing I/O congestion. (ii) supporting dedicated I/O scheduling. We propose two classes of I/O schedulers: *homogeneous* schedulers that offer a First Come First Serve scheduling policy and *heterogeneous* schedulers that offer a modified priority and backfilling scheduling policies. The homogeneous scheduler can be used when applications have one kind of I/O tasks, whereas the heterogeneous scheduler is more suitable for applications that have different kinds of I/O tasks, each producing a different I/O workload. (iii) Finally, a storage-heterogeneity aware system should support an automatic mechanism for maximizing the utilization of faster storage layers. Such mechanism works by periodically flushing obsolete data from higher storage layers to lower storage layers. These capabilities are supported in a completely transparent manner to applications. The evaluation of our prototype implementation of storage-heterogeneity awareness programming model showed the performance improvement that can be gained on applications with different I/O patterns on an execution platform that has different heterogeneous storage infrastructure.

**Chapter 6** targets performance maximization on distributed many-core architectures by proposing a hybrid programming model that combines task-based programming models and MPI. This hybrid programming model uses tasks to achieve coarse-grained task parallelism on large-scale distributed infrastructures, whereas MPI is used to gain fine-grained parallelism by parallelizing tasks execution. For instance, enabling parallel I/O and high-level I/O libraries in tasks. We enable such hybrid programming model by supporting *Native MPI Tasks*. Native MPI tasks allows application developers to use MPI code to parallelize tasks executions. Unlike executing external MPI binaries or scripts, Native MPI tasks are native to the programming model, such that the same source file can contain several Native MPI tasks that are executed by different parallel MPI processes, and sequential tasks that are executed by a single process. Therefore, easing the design and programming of applications because Native MPI tasks can use global data structures, imported modules, call other functions in the main code, etc. Native MPI tasks can be declared by using the `@mpi` annotation, and the number of MPI processes per task can be specified by adding the `@constraint` annotation on top of the `@mpi` annotation. The evaluation section of this chapter demonstrates the benefits that can be reached by using this hybrid programming model in terms of programmability and performance. These benefits are shown not only in an I/O intensive application, but also a compute intensive application. Parallelizing tasks execution by using Native MPI tasks can lead to a notable performance improvement at the task-level. Moreover, this task-level performance enhancement is reflected as an improvement in the total application performance. The evaluation section concludes with discussing a performance trade-off that arises between tasks parallelism and MPI parallelism.

Finally, **Chapter 7** aims at accelerating applications execution by suggesting modifications in the definition and management of data dependencies of tasks in task-based programming models. We propose an *Eager approach for releasing data dependencies*, unlike the traditional approach for releasing them. Eagerly releasing data dependencies allows successor tasks to be released for execution as soon as their data dependencies are ready, without having to wait for predecessor task(s) to completely finish execution. Thus, a programming

model that supports the eager-release of data dependencies can accelerate the execution of tasks and create opportunities for overlapping computation with I/O. In order to support the eager-release of data dependencies, we describe the following core requirements: (i) parameter-aware dependencies, in which the data dependencies between tasks are described not only in terms of predecessor and successor tasks, but also in terms of the data that resulted in the dependency relationships. (ii) a mechanism for notifying the programming model that the data dependencies of a successor task are ready, so that the successor task can be launched for execution. We realized this mechanism by enabling the use of a simple API call `compss_ready_value` that notifies the programming model that a data has been generated in a task, without waiting until the task reaches the `return` statement. Therefore, tasks are released for execution as soon as possible without any delays. Our prototype implementation is evaluated on different applications that exhibit real-world execution patterns. The results show that using the eager-approach for releasing dependencies can achieve significant performance improvements compared to the baseline version that uses the traditional manner for releasing data dependencies. In addition to that, the eager-release of dependencies makes it possible to support more optimal application designs for I/O intensive applications that read large amounts of data that cannot fit in memory.

## 8.2 Future Work

As the data produced by applications are continuously increasing, I/O performance will continue to be the issue-to-solve in order to gain more performance on modern day infrastructures. In this context, the contributions of this thesis provide a basis on which more efforts can be built to adapt to the needs and requirements of scientific and data-intensive applications. In this section, we provide an overview of the future work that can be done with regard to each of the contributions described in Part II of this thesis.

With regard to **Chapter 4**, since separating I/O from computations may be difficult in some applications, we plan to provide a new abstraction that identifies a task that performs both computation and I/O. Moreover, the programming models should support mechanisms to optimize the execution of such *hybrid* tasks. These mechanisms should monitor tasks computation phase and I/O phase, then perform optimization decisions with regard to the scheduling and resource consumption accordingly. In addition to that, we plan to develop mechanisms that are able to mitigate I/O congestion when the I/O is done to a shared storage resource. In this scenario, the programming model needs to be aware that I/O performance is not deterministic due to the many applications that use the resource.

As for **Chapter 5**, heterogeneous storage systems design is becoming the de-facto design option, future systems will include more heterogeneous storage and memory solutions. Therefore, we plan to extend our work in this part to take into consideration not only the storage devices, but also the memory systems. Data can be kept in memory and retrieved whenever they are required by applications. In addition to that, we plan to implement the automatic data movement mechanism for read-intensive applications, where data should be pulled/moved to faster storage hierarchies according to certain execution time metrics, e.g., the frequency of using the data.

In the case of **Chapter 6**, new programming models emerge every day for getting more performance out of the hierarchical and heterogeneous architectures of current execution platforms. Therefore, we plan to extend our hybrid programming model to include other programming models. Such programming models can be expressed as *Tasks + MPI + X* where $X$ is a third programming model that provide a different layer of performance, e.g., OpenMP [19] or programming frameworks targeting GPUs such as CUDA [77] or OpenCL [74]. Moreover, we plan to support scheduling algorithms that take into consideration tasks

requirements and their criticality, the used programming model(s) inside tasks and the available resources in the infrastructure. Also, support inter-task MPI communication.

Finally, for **Chapter 7**, we plan to support automatic mechanisms to automatically detect data generation inside tasks without making the users do any code modifications. Such methods can continuously probe memory addresses to check if they have been recently used or add hook functions to trigger a certain function call when a memory address is accessed. In addition to that, we plan to extend the eager-release of dependencies to support releasing nested tasks. Nested tasks can be implemented as regular functions or can be a completely separate applications or pipelines.

# Part IV

# Bibliography

# Bibliography


[1] 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.

[2] N. Ali et al. "*Scalable I/O forwarding framework for high-performance computing systems*". In: *2009 IEEE International Conference on Cluster Computing and Workshops*. 2009, pp. 1–10. DOI: 10.1109/CLUSTR.2009.5289188.

[3] T. Alturkestani, H. Ltaief, and D. Keyes. "Maximizing I/O Bandwidth for Reverse Time Migration on Heterogeneous Large-Scale Systems". In: *Euro-Par 2020: Parallel Processing*. Cham: Springer International Publishing, 2020, pp. 263–278.

[4] *Apache Airflow (The Apache Software Foundation, 2017)*. Retrieved from http://airflow.apache.org. Accessed 15 December, 2017.

[5] K. Asanovic and et al. "A view of the Parallel Computing Landscape". In: *Communications of the ACM* 52.10 (2009), pp. 56–67. DOI: 10.1145/1562764.1562783.

[6] M Asch et al. "Big data and extreme-scale computing: Pathways to Convergence-Toward a shaping strategy for a future software and data ecosystem for scientific inquiry". In: *The International Journal of High Performance Computing Applications* 32.4 (2018), pp. 435–479. DOI: 10.1177/1094342018778123. eprint: https://doi.org/10.1177/1094342018778123. URL: https://doi.org/10.1177/1094342018778123.

[7] Y. Babuji et al. "Parsl: Pervasive Parallel Programming in Python". In: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '19. Phoenix, AZ, USA: Association for Computing Machinery, 2019, 25–36. ISBN: 9781450366700. DOI: 10.1145/3307681.3325400. URL: https://doi.org/10.1145/3307681.3325400.

[8] R. M. Badia and et al. "COMP superscalar, an interoperable programming framework". In: *SoftwareX* 3 (2015), pp. 32–36. DOI: 10.1016/j.softx.2015.10.004.

[9] John Bent et al. "Jitter-free co-processing on a prototype exascale storage stack". In: *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. 2012, pp. 1–5. DOI: 10.1109/MSST.2012.6232382.

[10] Francieli Zanon Boito et al. "A Checkpoint of Research on Parallel I/O for High-Performance Computing". In: *ACM Comput. Surv.* 51.2 (Mar. 2018). ISSN: 0360-0300. DOI: 10.1145/3152891. URL: https://doi.org/10.1145/3152891.

[11] Peter Braam. *The Lustre Storage Architecture*. 2019. arXiv: 1903.01955 [cs.OS].

[12] *Broad Institute Best Practices*. https://gatk.broadinstitute.org/hc/en-us/sections/360007226651-Best-Practices-Workflows. Date of Last Access: 26th August, 2021.

[13] Andre R. Brodtkorb et al. "State-of-the-Art in Heterogeneous Computing". In: *Sci. Program.* 18.1 (2010), 1–33. DOI: 10.1155/2010/540159. URL: https://doi.org/10.1155/2010/540159.

[14] S. Byna et al. “Parallel I/O, analysis, and visualization of a trillion particle simulation”. In: *SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–12.

[15] Philip Carns et al. “24/7 Characterization of Petascale I/O Workloads”. In: *2009 IEEE International Conference on Cluster Computing and Workshops*. 2009, pp. 1–10. DOI: 10.1109/CLUSTR.2009.5289150.

[16] Philip Carns et al. “Understanding and Improving Computational Science Storage Access through Continuous Characterization”. In: *ACM Trans. Storage* 7.3 (2011). ISSN: 1553-3077. DOI: 10.1145/2027066.2027068. URL: https://doi.org/10.1145/2027066.2027068.

[17] Adrian M. Caulfield et al. “Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing”. In: *SC ’10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 2010, pp. 1–11.

[18] C. Cullinan, Timothy Richard Frattesi, and C. Wyant. “Computing Performance Benchmarks among CPU, GPU, and FPGA”. In: 2012.

[19] Leonardo Dagum and Ramesh Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[20] 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). DOI: 10.1145/1327452.1327492.

[21] 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.

[22] 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.

[23] J. Diaz, C. Muñoz-Caro, and A. Niño. “A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era”. In: *IEEE Transactions on Parallel and Distributed Systems* 23.8 (2012), pp. 1369–1386. ISSN: 1045-9219. DOI: 10.1109/TPDS.2011.308.

[24] James Dinan et al. “Hybrid Parallel Programming with MPI and Unified Parallel C”. In: *Proceedings of the 7th ACM International Conference on Computing Frontiers*. CF ’10. New York, NY, USA: ACM, 2010, pp. 177–186. ISBN: 978-1-4503-0044-5. DOI: 10.1145/1787275.1787323. URL: http://doi.acm.org/10.1145/1787275.1787323.

[25] Bin Dong et al. “Data Elevator: Low-Contention Data Movement in Hierarchical Storage System”. In: *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. 2016, pp. 152–161. DOI: 10.1109/HiPC.2016.026.

[26] Alejandro Duran et al. “Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures.” In: *Parallel Processing Letters* 21 (2011), pp. 173–193. DOI: 10.1142/S0129626411000151.

[27] Biosequence Analysis using Profile Hidden Markov Models. Web Page at http://hmmer.org/. Date of Last Access: 26th August, 2021. 2019.

[28] Common Crawl Wesbite. at https://commoncrawl.org/. Date of Last Access: 10th August, 2021. 2007.

[29] Computational Data Analysis Workflow Systems. https://s.apache.org/existing-workflow-systems/. Date of Last Access: 20th August, 2021. 2015.

[30] EMBL-EBI FTP Server. Web Page at http:ftp://ftp.ebi.ac.uk/pub/. Date of Last Access: 26th August, 2021. 2019.

[31] High-Performance Data Analysis: HPC Meets Big Data. http://www.hpcuserforum.com/presentations/tuscon2013/IDCHPDABigDataHPC.pdf. Date of Last Access: 6th August, 2021. 2015.

[32] Luigi. Github at https://github.com/spotify/luigi. Date of Last Access: 6th July, 2021. 2015.

[33] MareNostrum CTE-Power Architecture. Web Page at https://www.bsc.es/support/POWER_CTE-ug.pdf. Date of Last Access: 13th August, 2021. 2021.

[34] *Extrae Tool (Barcelona Supercomputing Center, 2017)*. Retrieved from https://tools.bsc.es/extrae. Last Accessed 20 August, 2021.

[35] C Farhat et al. "FETI-DP: a dual-primal unified FETI method - part 1: a faster alternative to the two-level FETI method". In: *International Journal for Numerical Methods in Engineering* 50 (2001), pp. 1523–1544. ISSN: 0029-5981.

[36] A. Gainaru et al. "Scheduling the I/O of HPC Applications Under Congestion". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. 2015, pp. 1013–1022.

[37] *GATK-Broad Institute*. https://gatk.broadinstitute.org/hc/en-us/articles/360035890811-Resource-bundle. Date of Last Access: 26th August, 2021.

[38] L. Goodstadt. "Ruffus: a lightweight Python library for computational pipelines". In: *Bioinformatics* 26.21 (Sept. 2010), pp. 2778–2779. ISSN: 1367-4803. URL: https://doi.org/10.1093/bioinformatics/btq524.

[39] K. Harms et al. *Impact of burst buffer architectures on application portability*. Tech. rep. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States): Oak Ridge Leadership Computing Facility (OLCF), 2016.

[40] Bill Harrod. *Big Data and Scientific Discovery*. https://www.exascale.org/bdec/sites/www.exascale.org.bdec/files/talk4-Harrod.pdf. 2014.

[41] Dave Henseler et al. "Architecture and Design of Cray Datawarp". In: Cray User Group CUG, 2016.

[42] S. Herbein et al. "Scalable I/O-Aware Job Scheduling for Burst Buffer Enabled HPC Clusters". In: *HPDC '16*. 2016.

[43] *HPC IO Benchmark Repository.* https://github.com/hpc/ior. (Date of last access: 10th April, 2021).

[44] Wei Hu et al. "Storage Wall for Exascale Supercomputing". In: *Frontiers of Information Technology and Electronic Engineering* 17 (). DOI: https://doi.org/10.1631/FITEE.1601336.

[45] 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.

[46] *Human Genome Builds*. URL: https://gatk.broadinstitute.org/hc/en-us/articles/360035890951-Human-genome-reference-builds-GRCh38-or-hg38-b37-hg19.

[47] D. Ibtesham et al. "On the Viability of Compression for Reducing the Overheads of Checkpoint/Restart-Based Fault Tolerance". In: *2012 41st International Conference on Parallel Processing*. 2012, pp. 148–157.

[48] *Intel MPI implementation (Intel Corporation, 2017)*. Retrieved from https://software.intel.com/en-us/intel-mpi-library. Last Accessed: 30 July, 2021.

[49] Dana Jacobsen. "An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters". In: 2010.

[50] Harsh Khetawat et al. "Evaluating Burst Buffer Placement in HPC Systems". In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 2019, pp. 1–11. DOI: 10.1109/CLUSTER.2019.8891051.

[51] Dries Kimpe et al. "Integrated In-System Storage Architecture for High Performance Computing". In: *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*. ROSS '12. New York, NY, USA: Association for Computing Machinery, 2012. ISBN: 9781450314602. DOI: 10.1145/2318916.2318921. URL: https://doi.org/10.1145/2318916.2318921.

[52] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. "Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Buffering System". In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '18. Association for Computing Machinery, 2018, 219–230. ISBN: 9781450357852.

[53] 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.

[54] 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.

[55] Alex K. Lancaster et al. "COSMOS: Python library for massively parallel workflows". In: *Bioinformatics* 30.20 (2014). ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btu385. URL: https://dx.doi.org/10.1093/bioinformatics/btu385.

[56] Albert Lazzarini. *Advanced LIGO Data and Computing*. https://labcit.ligo.caltech.edu/~dhs/Adv-LIGO/review-june03/breakouts/Astrophysics,%20Data%20Analysis%20Hardware,%20Data%20Acquisition/lazz-ldas-newer.pdf. 2003.

[57] Charles E. Leiserson. "The Cilk++ concurrency platform". In: *DAC*. 2009.

[58] Heng Li and Richard Durbin. "Fast and accurate long-read alignment with Burrows–Wheeler transform". In: *Bioinformatics* 26.5 (2010), pp. 589–595. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btp698. URL: https://doi.org/10.1093/bioinformatics/btp698.

[59] Jianwei Li et al. "Parallel netCDF: A High-Performance Scientific I/O Interface". In: *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. 2003, pp. 39–39. DOI: 10.1109/SC.2003.10053.

[60] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc., 1999. ISBN: 0201325772.

[61] W. Liang et al. "CARS: A contention-aware scheduler for efficient resource management of HPC storage systems". In: *Parallel Comput.* 87 (2019), pp. 25–34.

[62] Lin-Wen Lee, P. Scheuermann, and R. Vingralek. "*File assignment in parallel I/O systems with minimal variance of service time*". In: *IEEE Transactions on Computers* 49.2 (2000), pp. 127–140. ISSN: 2326-3814. DOI: 10.1109/12.833109.

[63] N. Liu et al. "On the role of burst buffers in leadership-class storage systems". In: *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. 2012, pp. 1–11.

[64] Yin Lu et al. "Revealing Applications' Access Pattern in Collective I/O for Cache Management". In: *Proceedings of the 28th ACM International Conference on Supercomputing*. ICS '14. New York, NY, USA: Association for Computing Machinery, 2014, 181–190. ISBN: 9781450326421. DOI: 10.1145/2597652.2597686. URL: https://doi.org/10.1145/2597652.2597686.

[65] N. Malitsky et al. "Building near-real-time processing pipelines with the spark-MPI platform". In: *2017 New York Scientific Data Summit (NYSDS)*. 2017, pp. 1–8. DOI: 10.1109/NYSDS.2017.8085039.

[66] *MareNostrum 4 Supercomputer*. Web Page at https://www.bsc.es/marenostrum/marenostrum. Date of Last Access: 26th July, 2021. 2021.

[67] Vladimir Marjanović et al. "Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach". In: *Proceedings of the 24th ACM International Conference on Supercomputing*. ICS '10. New York, NY, USA: Association for Computing Machinery, 2010, 5–16. ISBN: 9781450300186. DOI: 10.1145/1810085.1810091. URL: https://doi.org/10.1145/1810085.1810091.

[68] W. McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 51 –56.

[69] Rino Micheloni. "Solid-State Drive (SSD): A Nonvolatile Storage System". In: *Proceedings of the IEEE* 105.4 (2017), pp. 583–588. DOI: 10.1109/JPROC.2017.2678018.

[70] *Mira Supercomputer of Argonne Leadership Facility*. https://www.alcf.anl.gov/alcf-resources/mira. 2021.

[71] 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.

[72] *MPI: A Message-Passing Interface Standard*. Tech. rep. 3.1. June 2015. URL: http://mpi-forum.org/docs/.

[73] *MPICH: High-Performance Portable MPI (MPICH Collaborators, 2017)*. Retrieved from https://www.mpich.org. Last Accessed: 30 July, 2021.

[74] A. Munshi et al. *OpenCL programming guide*. Pearson Education, 2011. DOI: 10.5555/2049883.

[75] *MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE (NBCL, 2017)*. Retrieved from http://mvapich.cse.ohio-state.edu. Last Accessed: 30 July, 2021.

[76] *National Center For Biotechnology Information*. URL: https://www.ncbi.nlm.nih.gov/.

[77] J. Nickolls et al. "Scalable parallel programming with CUDA". In: *Queue* 6.2 (2008), pp. 40–53. DOI: 10.1145/1365490.1365500.

[78] *Non-volatile Random Access Memory (NVRAM) As A Replacement For Traditional Mass Storage*. https://patents.google.com/patent/US20140297938A1/en. 2016.

[79] Kazuki Ohta et al. "Optimization Techniques at the I/O Forwarding Layer". In: *2010 IEEE International Conference on Cluster Computing*. 2010, pp. 312–321. DOI: 10.1109/CLUSTER.2010.36.

[80] T. E. Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[81] *Open MPI: Open Source High Performance Computing (The Open MPI Project, 2017)*. Retrieved from https://www.open-mpi.org. Last Accessed: 30 July, 2021.

[82] *OS Python Library*. Web Page at https://docs.python.org/3/library/os.html. Date of Last Access: 15th August, 2021. 2020.

[83] "Overview of the IBM Blue Gene/P project". In: *IBM Journal of Research and Development* 52.1.2 (2008), pp. 199–220. DOI: 10.1147/rd.521.0199.

[84] *Parallel I/O on Mira Supercomputer*. https://www.alcf.anl.gov/files/Parallel_IO_on_Mira_0.pdf. 2021.

[85] *Paraver Tool (Barcelona Supercomputing Center, 2017)*. Retrieved from https://tools.bsc.es/paraver. Last Accessed 20 August, 2021.

[86] Christina M. Patrick et al. "Cashing in on Hints for Better Prefetching and Caching in PVFS and MPI-IO". In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. HPDC '10. New York, NY, USA: Association for Computing Machinery, 2010, 191–202. ISBN: 9781605589428. DOI: 10.1145/1851476.1851499. URL: https://doi.org/10.1145/1851476.1851499.

[87] J. Perez et al. "*Improving the Integration of Task Nesting and Dependencies in OpenMP*". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pp. 809–818. DOI: 10.1109/IPDPS.2017.69.

[88] 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.

[89] Jean-Pierre Prost. "MPI-IO". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1191–1199. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_297. URL: https://doi.org/10.1007/978-0-387-09766-4_297.

[90] *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 20 August, 2021.

[91] R. Rabenseifner, G. Hager, and G. Jost. "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes". In: *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. 2009, pp. 427–436. DOI: 10.1109/PDP.2009.43.

[92] James Reinders. "Intel threading building blocks - outfitting C++ for multi-core processor parallelism". In: 2007.

[93] M. Rocklin. "Dask: Parallel Computation with Blocked algorithms and Task Scheduling". In: 2015, pp. 130 –136. DOI: 10.25080/Majora-95ae3ab6-01e.

[94] K. Sala et al. "Integrating Blocking and Non-Blocking MPI Primitives with Task-Based Programming Models". In: 2010.

[95] Kento Sato et al. "A User-Level InfiniBand-Based File System and Checkpoint Strategy for Burst Buffers". In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2014, pp. 21–30. DOI: 10.1109/CCGrid.2014.24.

[96] Frank Schmuck and Roger Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters". In: *Proceedings of the 1st USENIX Conference on File and Storage Technologies*. FAST '02. USA: USENIX Association, 2002, 19–es.

[97] Committee on the Potential Impact of High-End Computing on Illustrative Fields of Science and National Research Council Engineering. *On the Potential Impact of High-End Computing on Illustrative Fields of Science and National Research Council Engineering*. National Academies Press, 2008.

[98] Seetharami Seelam et al. "Masking I/O latency using application level I/O caching and prefetching on Blue Gene systems". In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470438.

[99] Z. Tan et al. "*ALDM: Adaptive Loading Data Migration in Distributed File Systems*". In: *IEEE Transactions on Magnetics* 49.6 (2013), pp. 2645–2652. ISSN: 1941-0069. DOI: 10.1109/TMAG.2013.2251616.

[100] 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.

[101] Rajeev Thakur, William Gropp, and Ewing Lusk. "A Case for Using MPI's Derived Datatypes to Improve I/O Performance". In: *Proceedings of SC98: High Performance Networking and Computing*. ACM Press, 1998. URL: http://www.mcs.anl.gov/~thakur/dtype/.

[102] Rajeev Thakur, William Gropp, and Ewing Lusk. *Achieving High Performance with MPI-IO*.

[103] *The EXPERTISE Project*. Web Page at http://www.msca-expertise.eu/. Date of Last Access: 26th August, 2021. 2021.

[104] *The h5py package: a Pythonic interface to HDF5*. URL: http://www.h5py.org/.

[105] The HDF Group. *Hierarchical data format version 5*. 2000-2010. URL: http://www.hdfgroup.org/HDF5.

[106] M. Tillenius et al. "Resource-Aware Task Scheduling". In: *ACM Trans. Embed. Comput. Syst.* 14.1 (Jan. 2015). ISSN: 1539-9087. DOI: 10.1145/2638554. URL: https://doi.org/10.1145/2638554.

[107] *TOP 500 Supercomputer List*. Web page at https://www.top500.org/lists/top500/. (Date of last access: 10th September, 2020).

[108] 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 (2009), pp. 267–295. DOI: 10.3233/978-1-60750-073-5-267.

[109] Venkatram Vishwanath et al. "Accelerating I/O Forwarding in IBM Blue Gene/P Systems". In: *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 2010, pp. 1–10. DOI: 10.1109/SC.2010.8.

[110] Venkatram Vishwanath et al. "Accelerating I/O Forwarding in IBM Blue Gene/P Systems". In: *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 2010, pp. 1–10. DOI: 10.1109/SC.2010.8.

[111] Venkatram Vishwanath et al. "Topology-Aware Data Movement and Staging for I/O Acceleration on Blue Gene/P Supercomputing Systems". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. New York, NY, USA: Association for Computing Machinery, 2011. ISBN: 9781450307710. DOI: 10.1145/2063384.2063409. URL: https://doi.org/10.1145/2063384.2063409.

[112] Teng Wang et al. "An Ephemeral Burst-Buffer File System for Scientific Applications". In: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, pp. 807–818. DOI: 10.1109/SC.2016.68.

[113] Teng Wang et al. "BurstMem: A high-performance burst buffer system for scientific applications". In: *2014 IEEE International Conference on Big Data (Big Data)*. 2014, pp. 71–79. DOI: 10.1109/BigData.2014.7004215.

[114] Teng Wang et al. "UniviStor: Integrated Hierarchical and Distributed Storage for HPC". In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 2018, pp. 134–144. DOI: 10.1109/CLUSTER.2018.00025.

[115] *Web Archive Format*. Web page at http://bibnum.bnf.fr/WARC/. Date of last access: 10th August, 2021. 2021.

[116] B. Xie et al. "Characterizing output bottlenecks in a supercomputer". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–11.

[117] Bing Xie et al. "Predicting Output Performance of a Petascale Supercomputer". In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '17. New York, NY, USA: Association for Computing Machinery, 2017, 181–192. ISBN: 9781450346993. DOI: 10.1145/3078597.3078614. URL: https://doi.org/10.1145/3078597.3078614.

[118] Yiqi Xu et al. "vPFS: Bandwidth virtualization of parallel storage systems". In: *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. 2012, pp. 1–12. DOI: 10.1109/MSST.2012.6232370.

[119] 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.

[120] Dongfang Zhao, Kan Qiao, and Ioan Raicu. "HyCache+: Towards Scalable High-Performance Caching Middleware for Parallel File Systems". In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2014, pp. 267–276. DOI: 10.1109/CCGrid.2014.11.

[121] Z. Zhou et al. "I/O-Aware Batch Scheduling for Petascale Computing Systems". In: *2015 IEEE International Conference on Cluster Computing*. 2015, pp. 254–263.

[122] *Scientific Grand Challenges: Cross-Cutting Technologies for Computing at the Exascale Workshop*. 2010-02. URL: http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Crosscutting_grand_challenges.pdf.