Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

A Framework for Automated Parallel Execution of Scientific Multi-Workflow Applications in the Cloud with Work Stealing

Helena Schubert I. L. Silva

Dissertação do Mestrado em Informática

Orientador
Prof. Dr. Alba C. M. A Melo

Brasília
2024

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

A Framework for Automated Parallel Execution of Scientific Multi-Workflow Applications in the Cloud with Work Stealing

Helena Schubert I. L. Silva

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Brasília, 04 de outubro de 2024

Dedicatória

Primeiramente, dedico este longo trabalho e todos os frutos que dele vierem a meus amados pais, Jurema Schubert e Antonio Elesbão. Obrigada, mãe, por todo seu apoio durante esse período! Obrigada, mãe, por também ser meu ombro amigo nas dificuldades e por ser uma das minhas principais inspirações na vida, principalmente sobre coragem! Obrigada, pai, por sempre ter apoiado meus esforços e por ter falado que era melhor cursar computação que física na graduação! Foi uma sábia escolha.

Dedico também este trabalho aos pais dos meus pais, cujas trajetórias estão sempre ecoando na minha mente, provocando-me orgulho, servindo-me de exemplo e trazendo-me carinho. Em particular, dedico este trabalho à minha avó Manuelina de Jesus Lima da Silva, a Dona Santa. Que saudade, minha vó querida!

Por último, dedico também ao meu amado bem, Danilo Bispo, por me ouvir e me dar forças, e à minha querida amiga Luana Signorelli, por me entender tão bem. Como é bom ter companhia nas batalhas!

Acknowledgments

I would like to thank the many professors and colleagues whose ideas and inspirations are present in this work. In particular, I would like to mention Professor Dr. Maria Clícia Castro, Dr. Fabrício A. B. Silva, and especially my advisor, Professor Dr. Alba Cristina M. A. Melo, who has been a great inspiration to me and to many of us who graduated from my beloved University of Brasília.

Resumo

Um Framework para Execução Paralela Automática de Aplicações de Múltiplos Workflows Científicos na Nuvem com Roubo de Trabalho

Workflows científicos são executados em diversos laboratórios de pesquisa todos os dias em vários continentes, contribuindo significativamente para os avanços da Ciência. Na maioria das vezes, os workflows são executados por *scripts* desenvolvidos de maneira *ad hoc* em infra-estruturas computacionais defasadas. Na presente Dissertação de Mestrado, visamos propor e avaliar um *framework* para execução automática de aplicações compostas por múltiplos *workflows* científicos na nuvem AWS. Para tirar proveito do paralelismo, o *framework* proposto executa-se em plataforma com diversos *nodes* computacionais e várias *threads* em cada nodo. Adicionalmente, como existe um grande número de operações de E/S nestes *workflows*, dois tipos de sistema de arquivos serão usados (compartilhado e local). Finalmente, propomos uma estratégia multi-nível de roubo de trabalho para reduzir o desbalanceamento de carga. Nossa resultados mostram que a estratégia paralela combinada ao roubo de trabalho contribuem para a redução significativa de tempo de execução.

Palavras-chave: Workflow científico, Roubo de trabalho, Computação em nuvem.

Abstract

Scientific Workflows are executed in several research laboratories every day on many continents, contributing significantly to advances in Science. Most of the time, workflows are executed by scripts developed in an *ad hoc* manner on outdated computing infrastructures. In this MSc Dissertation, we propose and evaluate a framework for automatic execution of scientific applications composed of multi-workflows in the AWS cloud. To take advantage of parallelism, the proposed framework runs on two levels on a platform with several computational nodes and several threads in each node. Furthermore, as there is a large number of I/O operations in these workflows, two types of file systems are used (shared and local). Finally, we propose a multi-level work stealing strategy to reduce load imbalance. Our results show that the parallel strategy combined with work stealing contributes to a significant reduction of the execution time.

Keywords: Scientific workflows, Work stealing, Cloud computing.

Contents

List of Figures

List of Tables

Acronyms

ASCI-Red Accelerated Strategic Computing Initiative Red.

AWS Amazon Web Service.

BoT Bag-of-Tasks.

DAG Directed Acyclic Graph.

EC2 Elastic Computing Cloud.

ERES Energy and Resource Efficient workflow Scheduling.

EUSF Energy and Uncertain task ET aware workflow Scheduling Framework.

FaaS Function as a Service.

FIFO First-In-First-Out.

Flops Floating Point Operations per Second.

GPU Graphic Processing Unit.

HEFT Heterogeneous Earliest Finish Time.

HPC High Performance Computing.

HPCaaS High Performance Computing as a Service.

IaaS Infrastructure as a Service.

IaC Infrastructure as Code.

LIFO Last-In-First-Out.

MIMD Multiple Instructions Multiple Data.

MOHEFT Multi-Objective Heterogeneous Earliest Finish Time.

MPI Message Passing Interface.

MPICH MPI over CHameleon.

NCBI National Center for Biotechnology Information.

NFS Network File System.

NOSF onliNe multi-workflOw Scheduling Framework.

OpenMP Open Multi-Processing.

PaaS Platform as a Service.

REST API Representational State Transfer Application Programming Interface.

ROSA unceRtainty-aware Online Scheduling Algorithm.

RPC Remote Procedure Call.

S3 Simple Storage Service.

SaaS Software as a Service.

Slurm Simple Linux Utility for Resource Management.

SRA Sequence Read Archive.

TCP/IP Transmission Control Protocol/Internet Protocol.

TWA Total Workflow Array.

VM Virtual Machine.

VPC Virtual Private Cloud.

WS Work Stealing.

WSA Workflow Status Array.

XaaS Everything as a Service.

Chapter 1

Introduction

In this chapter, we firstly introduce the motivation for this dissertation. Then, we present the problem assumptions, followed by the objectives of this work. Afterwards, we present the contributions which were achieved with this work and finally we appoint the organization of this document.

1.1 Motivation

Scientific workflows are executed in a regular basis in research laboratories all over the world, aiming to solve complex problems from several domains such as Biology, Medicine, Geology, among others. Usually, scientific workflows use real-world data as input. These inputs are typically stored as files after being read by specialized machines. In Bioinformatics workflows, for example, it is common for each experiment to generate one or more files containing their raw biological data, such as mRNA or DNA [11][12][13][14]. These files are then analyzed alongside other files of the same type using Bioinformatics tools. Typically, each file needs to be preprocessed and pass through quality control phases before analysis. Moreover, it is often beneficial to store the preprocessed files, in addition to the original raw data, since multiple workflows may use the same preprocessed inputs. This approach avoids repeating the preprocessing phase for each workflow, when analyzing a previously processed input with different analyses' tools. The preprocessing and quality control stages typically take the form of a linear workflow and are applied to each file within a given dataset.

Processing files involves numerous I/O operations, which can be time-consuming. Consequently, the preprocessing and quality control phases are often the slowest part of a Bioinformatics workflow. For instance, in the workflow proposed in [11], the preprocessing and quality control phases took 60 hours, whereas the analysis' phases took less than 5 minutes.

In most Bioinformatics laboratories, researchers are not experts in Computer Science nor have experimented technicians at their disposal. They often use ad hoc execution scripts which are error-prone and difficult to modify and update. This also may lead to underuse of the available infrastructure. In addition, most Bioinformatics laboratories cannot manage properly the computing infrastructure, which requires frequent updates of the computing hardware (computers, network, disks, etc) and numerous substitutions of faulty components. For this reason, in the last years, many laboratories have moved their infrastructure to the cloud [15]. Executing workflows in the cloud can drastically reduce the infrastructure costs but needs special attention, particularly when parallel executions are targeted. Using scalable frameworks, instead of simple scripts, may also allow better use of the cloud infrastructure.

In order to achieve better performance regarding execution time, cost and/or other parameters, scientific workflows should be scheduled. Workflows are defined as sets of tasks with temporal dependencies, meaning the output of one task serves as the input for the next task, and they are expressed as a Directed Acyclic Graph (DAG) $W = (T, E)$, where T are the tasks and E are the dependencies among the tasks. The general problem of scheduling tasks of a workflow is proven NP-Complete [16]. In addition, the problem of scheduling a workflow composed of a sequence of tasks (linear workflow), in its general formulation, is proven NP-Hard [17]. For this reason, many heuristic scheduling strategies have been proposed in the literature in the last decades [18] [19] [20] [21] [22].

Workflow scheduling strategies may be classified as static, when they allocate tasks to computing resources before the execution begins and do not change the allocation during runtime [1]. Static scheduling strategies assume that a large amount of information about the tasks and the computing environment is known a priori, including the tasks execution time, the performance of I/O operations, among others. However, in most realistic scenarios, these data are not known previously and, thus, dynamic scheduling strategies should be applied. Dynamic scheduling allocates and/or re-allocates tasks during runtime. Usually, it assumes that very few information about the tasks and/or the execution environment is known in the moment of the allocation [1]. In this case, a common goal of dynamic schedulers is to achieve load balancing, since this contributes to reduce the overall execution time.

Even though most schedulers deal with the problem of scheduling a single workflow [23] [24], there are in the literature some proposals that tackle the problem of scheduling multiple workflows. In [25], multiple linear workflows scheduled in a simulated distributed system are taken into consideration. Applications are modelled as single workflows, which may arrive at different moments and tasks that compose the workflows may be added or suppressed during execution. The mean execution time of the tasks is known in advance and, upon arrival, the workflow is assigned to the smallest queue of two randomly chosen nodes. A strategy for scheduling multiple workflows with different arrival times is also proposed in [26]. The workflows execute in a simulated cloud environment, with multiple Virtual Machines (VMs), each one with a different cost. As in the previous case [25], execution times of the workflows' tasks in each VM are known in advance.

Even though many workflow managers have been proposed for cloud computing [27], such as Galaxy [28] and Pegasus [29], they consider single workflow applications and use static approaches to assign tasks to computing nodes. As far as we know, there is no proposal in the literature that dynamically schedules multiple linear workflows with unknown task execution times, aiming to reduce the overall execution time in a real cloud environment.

1.2 Problem assumptions

In this work, we are targeting a typical class of linear scientific Bioinformatics workflows, where the same workflow is executed multiple times for different inputs. The workflows we are targeting have the following characteristics:

1. the size of the input may be significant (from tens of Megabytes, up to hundreds of Gigabytes) and vary for each workload;
2. several temporary files of considerable size are produced;
3. the execution time of each task is not previously known since it depends on the size and the contents of the input;
4. the same chain of programs is executed for many input files, resulting in a multi-workflow application;
5. executing the multi-workflow application can take hours.

To summarize, we are dealing with applications composed of long-lived multiple workflows, where the tasks of each workflow have unpredictable execution times and high disk activity. In such scenario, parallel computing and sophisticated scheduling approaches are required. In addition, we assume that the end-users do not have computational expertise.

1.3 Objective

The main goal of this dissertation is to investigate workflow scheduling strategies which are appropriate for applications composed of multiple workflows. We also intend to propose a framework that executes efficiently applications composed of scientific linear multi-workflows with unknown task execution times in the cloud. As a secondary objective, we aim to provide an easily configurable interface to the end-user.

1.4 Contributions

The main contributions of this work are:

1. An MPI/OpenMP framework to schedule linear multi-workflow applications in the cloud or in a High Performance Computing (HPC) architecture (*standard* mode). The framework receives two files as input: (a) a workflow skeleton file (b) an input file with n file names. Using the skeleton, we generate n workflows, which will process different input files, sorted by their lengths. The workflows are distributed among the threads of each node, using a striped approach. The execution terminates when all workflows are processed. The proposal was executed in the SDumont supercomputer (*sdumont.lncc.br*) using 100 I/O intensive synthetic workflows. The framework was tested in 1, 2, 4, 8 and 16 Virtual Machines (VMs), each one of them with 24 CPUs. The storage used was the Lustre parallel file system (*www.lustre.org*). We measured the total execution time and the execution time of each node.

2. An improved MPI/OpenMP version of the framework that incorporates dynamic scheduling with work stealing, named *work stealing* mode, which executes jointly with the *standard* mode. After having the workflows distributed among the nodes with the striped approach, the *work stealing* mode starts its execution at the moment when a thread becomes idle at the first time. In this case, the thread steals a workflow from another thread within the same node at first. If all threads in the same node are idle, work is stolen from another node. The execution finishes when all workflows are processed. An evaluation of this proposal was made in the AWS EC2, using the ParallelCluster (*aws.amazon.com/hpc/parallelcluster*) to deploy an HPC like architecture. We tested in 1, 2, 4, 8 and 16 instances, each one of them with 4 vCPU. We used the Amazon FSx for Lustre (*aws.amazon.com/pt/fsx/lustre*) to save the input and output files, and the local file system (Scratch) of each EC2 instance to save the temporary files. The framework was tested with two applications. The first one is composed of 74 I/O intensive synthetic workflows and the second one has 400 real Bioinformatics workflows which process RNA samples from real organisms.

A paper describing Contribution 2 and its results was published in [30].

1.5 Organization of the Dissertation

The remainder of this document is organized as follows. Chapter 2 presents the concepts of workflow and workflow scheduling, with emphasis to work stealing. In chapter 3 we present the concepts related to large scale parallel and distributed systems, focusing on HPC, cloud computing and HPC in cloud computing. Chapter 4 discusses related work. Chapter 5 presents the design of our framework. In chapter 6, we present our experimental results. Chapter 7 presents the conclusion and future works. Finally, Annex I presents the first page of our paper, published in the International Conference Euro-Par 2024.

Chapter 2

Overview of Workflows

In this chapter, we first present the basic concepts of workflows. Then we provide an overview of the problem of workflows scheduling, focusing on the work stealing approach.

2.1 Directed Acyclic Graph (DAG)

Workflows are a traditional way to represent an application composed of tasks. Usually, a Directed Acyclic Graph (DAG) is used to express temporal dependencies between two tasks t_i and t_j [31]. Tasks are executable units and, along with edges, compose a workflow.

Figure 2.1 presents a workflow with 7 tasks (t_0 to t_6). In this workflow, task t_2 must be executed before task t_4, since t_2's output is used as t_4's input. On the other hand, tasks t_3 and t_4 may be executed in parallel. With the workflow model, weights may be assigned to (a) the tasks, expressing the execution time, and (b) the edges, expressing the communication time. If no weights are assigned, it often means that tasks execute in the unity (1) and communication time is negligible.

Figure 2.1 illustrates an arbitrary DAG, but often DAGs have a predefined form such as linear, tree, fork-join (Figure 2.2). In the linear graphs, each task is preceded of only one task (except for the first one) and succeed of just one (except for the last one). Graphs modeled as trees have tasks which can have multiple children, but each child task has only one parent task. However, usually, the tree DAG graphs have a shared final task which represents the end of all leaf tasks. Fork-join DAGs are graph in which the first task represents the DAG's beginning and has multiple children tasks that can be executed in parallel, which in turn, share the same final child task.

A DAG can be expressed as $W = (T, E)$ where W is workflow, T is the set of n tasks of W, where $T = \{t_0, t_1, ..., t_{n-1}\}$ and E is the set of edges among the tasks, given by $E = \{e_{(i,j)} | (t_i, t_j \in T)\}$ where an edge of a DAG indicates a direct dependency where, given $e_{i,j} \in E_s$, and $t_i, t_j \in T$, t_i must be executed before t_j. Assuming that $pred(t_j)$

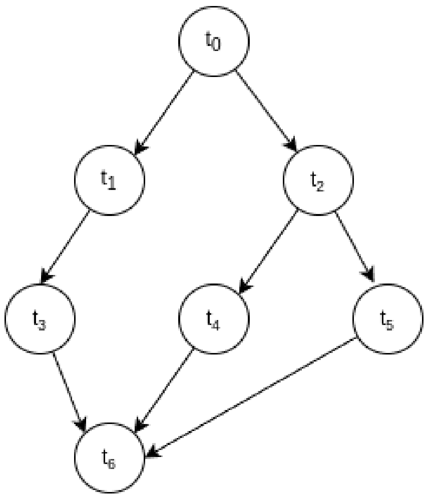


Figure 2.1: Example of an arbitrary DAG.

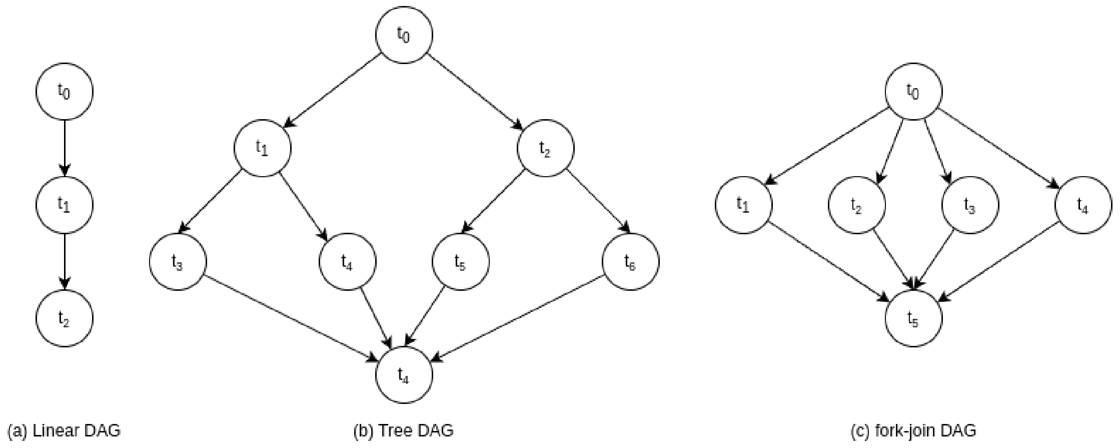


Figure 2.2: Some types of DAGs.

is the set of tasks that precede t_j and $succ(t_i)$ is the set of tasks that succeed t_i, then $t_i \subset pred(t_j)$ and $t_j \subset succ(t_i)$.

One or more workflows may compose a workflow application, resulting in a single workflow or multi-workflow application, respectively. Multi-workflow applications are also called workflow ensembles, since they are related to each other and the output of the application is in fact the output of all workflows in the ensemble. The structure of each workflow is very similar in the workflow ensemble and the main difference is the input data of each workflow [2]. Figure 2.3 illustrates (a) single workflow and (b) multi-workflow applications.

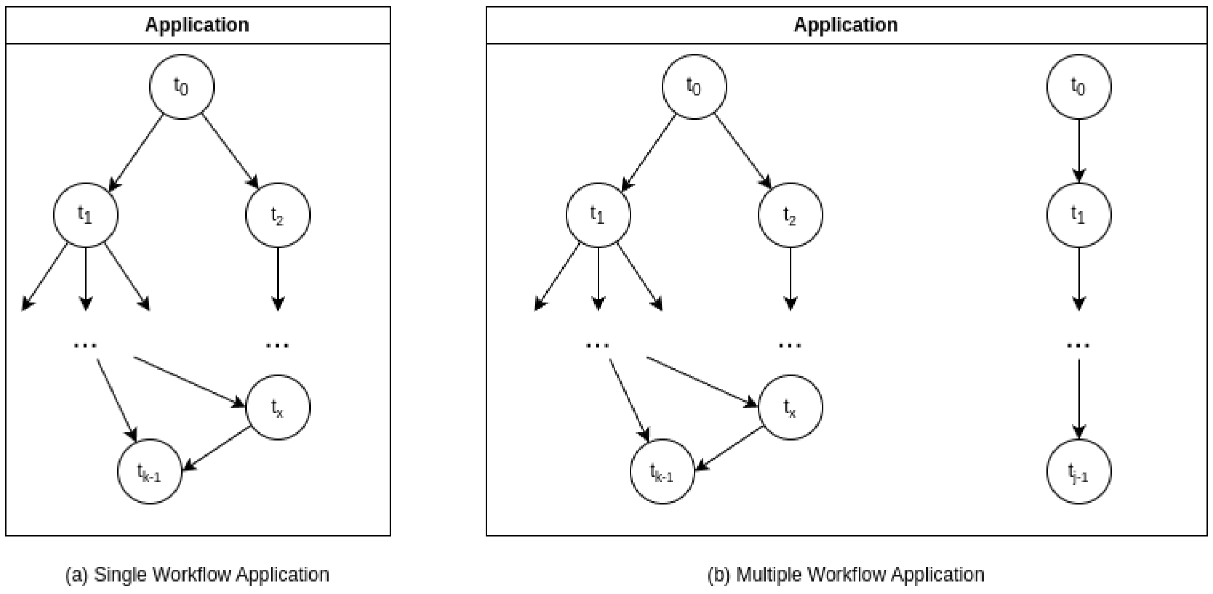

Application
t0
t1
t2
...
tx
tk-1
(a) Single Workflow Application
Application
t0
t1
t2
...
tx
tk-1
t0
t1
...
tj-1
(b) Multiple Workflow Application

Figure 2.3: Applications composed of single and multiple workflows.

2.2 Workflow Scheduling

Workflow scheduling defines how to map a DAG that expresses an application onto a set of resources, preserving the precedence conditions of the workflow. Typically, the goal of workflow scheduling is to optimize objective functions that aim at costs, energy and the overall execution time (makespan), among others [32]. The makespan of a workflow is computed from the beginning of the execution of the first task to the end of the execution of the last task (t_0 and t_6 in Figure 2.1, respectively).

The generic problem of scheduling a workflow is NP-complete [16], whereas scheduling a linear workflow in a generic way is an NP-Hard problem [17]. For this reason, there are several proposed algorithms that employ techniques to search for a good scheduling solution within a reasonable time.

One of the first taxonomies for workflow scheduling was proposed by Casavant and Kuhl [1], whose first level is in the Figure 2.4. In this taxonomy, static scheduling assumes that a lot information about tasks is available beforehand and tasks-to-processor mapping is done before the execution of the application begins. In dynamic scheduling, it is assumed that very few information about the tasks and resources is available beforehand. Thus, scheduling decisions are made when tasks of the workflow are already executing. The same paper highlights the importance of load balancing, stating that assigning the same load to all processors contributes to reduce the overall execution time.

The taxonomy proposed by Rodriguez and Buyya [2] is based on [1], applied to scientific workflows executing in the cloud. Concerning the task-VM mapping dynamicity, schedulers may be static, dynamic or hybrid, as shown in Figure 2.5. As in [1], in the

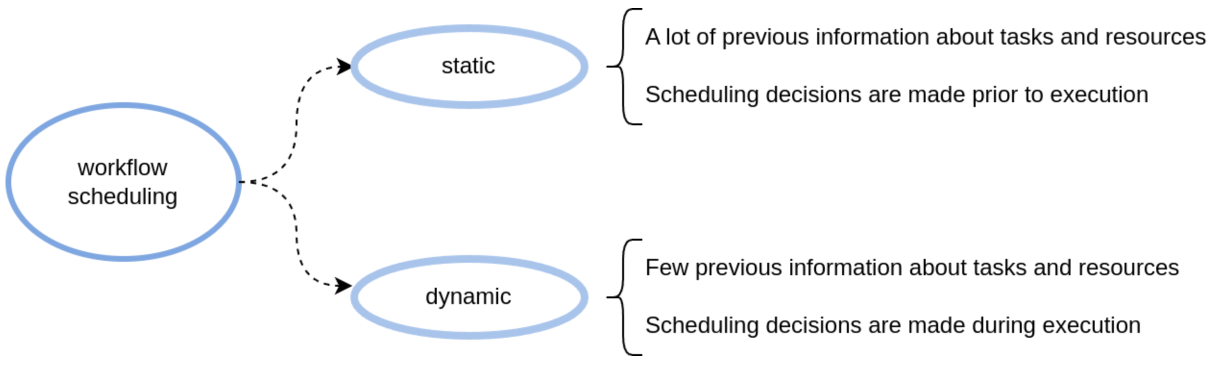

workflow scheduling
static
dynamic
A lot of previous information about tasks and resources
Scheduling decisions are made prior to execution
Few previous information about tasks and resources
Scheduling decisions are made during execution

Figure 2.4: First level of scheduling taxonomy of Casavant and Kuhl [1].

static approaches, the task-VM mapping is produced at once before the execution of the workflow begins. Dynamic schedulers take mapping decisions during runtime. The hybrid approach may be "runtime refinement" or "subworkflow static". Runtime refinement hybrid approaches make mapping decisions before execution begins. The system is monitored during execution and scheduling decisions may be revisited, i.e., the mapping may change at runtime. The subworkflow approach statically schedules each subworkflow when its first task is ready.

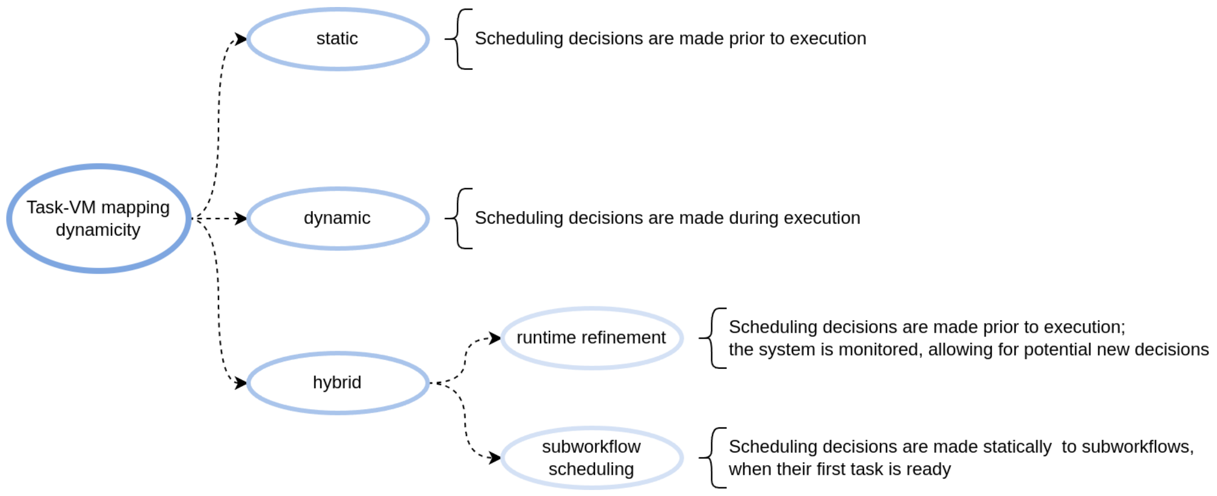

Task-VM mapping dynamicity
static
Scheduling decisions are made prior to execution
dynamic
Scheduling decisions are made during execution
hybrid
runtime refinement
Scheduling decisions are made prior to execution; the system is monitored, allowing for potential new decisions
subworkflow scheduling
Scheduling decisions are made statically to subworkflows, when their first task is ready

Figure 2.5: Task-VM mapping dynamicity of workflow scheduling in the cloud taxonomy of Rodriguez and Buyya [2].

2.3 Work Stealing (WS)

Work Stealing (WS) is a scheduling strategy that aims to achieve load balancing. It is classified as dynamic according to [1] or as hybrid runtime refinement according to [2] (Section 2.2). In WS, task-to-processor mapping is made before execution begins. When a processor finishes to process its tasks, it becomes idle. Idle resources have the capability to transfer and execute tasks from other busy resources.

There are three basic components in WS [33]:

- The task *queue* (stack), that contains the next tasks to be processed by a given resource.
- The *stealer* (thief), is a manager of a resource that takes tasks from another resource for itself to execute them. A stealer emerges when its resource becomes idle. This occurs when the tasks originally assigned to it are completed.
- The *stolen* (victim), which is the one that has a busy resource, still having tasks in its execution queue. When the resource realizes it has been stolen from, it does not execute the stolen task.

In [33], Blumofe et al. proposed work stealing for a multi-threaded runtime system called Cilk. Tasks are dynamically created, i. e., tasks can create (spawn) other tasks, and tasks are non-blocking. The system is composed of several processors which in their turn, may execute several tasks. The tasks are put in a ready pool, composed of priority queues, which are called levels. Each processor has one ready pool. The processor executes the tasks in the head of its pool (highest priority) until the pool is empty. In this case, it becomes a stealer and steals tasks from a randomly selected processor, stealing the highest priority tasks. Communication occurs with message passing.

The work [34], conducted by Blumofe and C. Leiserson, examined the complexity of WS schedulers, addressing both temporal and spatial dimensions. Their work focused on Multiple Instructions Multiple Data (MIMD) architectures, where processors execute tasks in parallel. Threads execute tasks with dependencies according to a DAG. The authors also contemplated that tasks could be dynamically invoked, where a parent task is able to invoke a new task already with a list of tasks to be performed. This new task is placed at the head of the processing stack of its processor, ready to be the next one processed. To balance the workload among processors, an idle processor can steal a task from a processor with tasks in waiting at the stack. However, unlike [33], when a processor steals, it takes from the tail of the stolen processor's stack.

The work [35], by Michael et al., proposes the idempotent work stealing approach for shared memory multi-threaded (multi-processor) execution. With idempotence, the authors mean that a task can be executed more than once, but at least once, without compromising the correctness of the application. This application-dependent feature allows for the removal synchronization operations, thus increasing the performance.

In their work, a thread has a stack with tasks that it will execute. If a thread exhausts its task stack, i.e., becomes idle, it is allowed to steal tasks from another thread. Additionally, a thread can receive new tasks in the queue that it owns.

The authors proposed and evaluated three algorithms: Last-In-First-Out (LIFO) - tasks are taken from the tail of the task queue; First-In-First-Out (FIFO) - tasks are taken from the head of the queue; and double-ended - where the owner of the queue extracts from the tail, and stealer threads extract from the head. In all these scenarios, the owner thread inserts tasks at the tail of the queue. Furthermore, it is ensured that all tasks will eventually be executed, and the stealing operation always returns a valid outcome.

In all algorithms proposed in [35], when a thread steals a task, there is no need to create a memory barrier, that would ensure no other thread accesses the queue at this moment. Therefore, a stolen task may also be executed by its original thread.

With the advent of distributed architectures, more recent works employ WS across multiple nodes. In contrast to earlier studies, there is now the utilization of distributed memory. The purpose of Dinan et al.'s work [36], is to explore how dynamic load balancing implemented with WS behaves in a scalable system with shared memory - more specifically, with the Partitioned Global Address Space (PGAS) model.

In the architecture of the model, the authors considered the use of a global Task Pool, provided in PGAS, although the authors defined that each process should maintain its own task queue with local access. Thus, even though each processor has its own task queue, and there is only one thread running per processor, tasks are visible and made available to other processors by PGAS, which provides asynchronous and non-blocking access to tasks. In this work, all tasks in the pool are considered independent, meaning their execution does not require any blocking for another task to complete. The task execution order is LIFO, with each processor maintaining a queue from which tasks are taken for processing from the head, while tasks are stolen from the end of the queue.

In the WS algorithm, an idle processor starts stealing from another chosen randomly as soon as the tasks of the stealing node are concluded. The stealing processor checks the metadata of the tasks from the stolen processor to see if there is still work to be done. If so, the stealer locks the stolen queue, checks once again if there is still work to be done, and then transfers one or more tasks from the end of the stolen queue to its own queue, unlocking the stolen queue at the end. If there are no more tasks, the stealer selects another processor randomly. This search continues until another processor with available tasks is found or the global end condition is attained.

To find and inform all processors of the global end condition, there is a mapping of all nodes in the form of a binary tree. When a process becomes idle, it gathers its vote with the votes of its children nodes and sends it to its parent node. However, if a node is being stolen from, it passes a negative vote upwards, requesting a revote. If the root node receives a vote, it notifies all nodes of the result. If no further re-vote is required, the root node sends one more message with the termination of work.

Chapter 3

Large Scale Parallel and Distributed Systems

3.1 Overview

A distributed system is defined as "*a collection of independent computers that appears to its users as a single coherent system*"[37]. In other words, a distributed system is built upon a network of computers creating the illusion of a unique system for both users and programmers [38]. A large scale distributed system is a distributed system composed of numerous interconnected computers (nodes), which can be either heterogeneous or homogeneous. The large amount of connected nodes provides a great computational power, which has been increasing quickly over the years. As examples of large scale distributed systems there are supercomputers, grid computing, clusters and cloud computing. In the following paragraphs, we provide a historical view of large scale distributed computing.

The concept of providing computational power as a service dates back to the 1960s. In 1961, McCarthy introduced the idea of utility computing, where computational resources could be shared and rented by multiple users, similar to how telephone lines are used as a service [39]. Right after that, in 1964, one of the first supercomputers was released: the CDC 6600, focused on scientific applications performance and on the use of instruction-level parallelism [3]. After that, in 1968, Licklider and Taylor proposed the concept of a network with multiple interconnected computers, possible to be used through interactive time-sharing multi-access points [40].

The evolution of supercomputers was benefited from technological advances. ARPANET adopted the Transmission Control Protocol/Internet Protocol (TCP/IP) in 1983. Around the same time, the concept of Remote Procedure Call (RPC) was proposed to operate over such protocols, and it became typically used in distributed systems for communication [41]. Similar to what was foreseen by McCarthy, Licklider and Taylor, in

1995 the term grid computing was coined, denoting a large-scale distributed system with geographically distant computers and storage systems. Its first purpose was to compute data-intensive scientific applications [42]. In the 1990s, the concept of cluster also became very popular. In cluster computing, many off-the-shelf computers were connected in order to increase performance, as a supercomputer. In 1997, a new paradigm called cloud computing [43] was first mentioned by Chellappa, which is a large scale distributed system on-demand plataform with a more centralized physical infrastructure than grid computing. Unlike grid computing, that provisions using whole time periods, the cloud on-demand system charges the user only for the time the system was actually used, diminishing costs and avoiding over-provisioning [44].

3.2 High Performance Computing (HPC)

3.2.1 History

High Performance Computing (HPC), also known as Supercomputing, refers to the high performance computers and their applications. Its objective is to run compute-intensive applications within a feasible time. One of the strategies for increasing performance of an application is to apply parallelism on it and HPC offers many levels of parallelism to execute parallelized applications [45].

The development of HPC has supported the development of various scientific fields [3]. For instance, Linear Algebra problems benefit significantly from HPC, as these problems can be easily decomposed into parallel computations. Nowadays, many fields execute computationally intensive workflows in HPC, such as Financial Services, Oil and Gas and Life Sciences [3], that take advantage on the HPC's performance to produce results much faster. Due the importance of these applications that run in HPC, its development keeps going since the 1960 decade.

In 1976, Seymour Cray founded the Cray Research and released Cray I, that could operate up to 136 MegaFlops (Floating Point Operations per Second) and had only one CPU. In 1982, Cray X-MP was released with support for four CPUs and could operate up to 200 MegaFlops per CPU, or 800 at total. Three years latter, the Cray 2 was released and overcame the memory bottleneck by using a local memory, which allowed the Cray 2 to reach up to 1.9 GigaFlops.

In the 1990s, the "supercomputer race" was at full pace and it was clear that there should be an objective way to rank supercomputers. The Linpack benchmark, created by Jack Dongarra et al. in the 1970s, measures how fast a computer solves a dense system of linear equations and it was chosen to biannually rank the supercomputers [46]. Linpack is being used to rank the 500 fastest supercomputers from 1993 to nowadays and the results are available in the Top500's site, *https://www.top500.org*.

Supercomputers' performance kept increasing over the years and was continuously compared using the Linpack benchmark. In 1996, Intel released the Accelerated Strategic Computing Initiative Red (ASCI-Red), which was the first supercomputer built with general-purpose devices, i.e. Pentium Pros and Pentium II Xeon processors, and it was the first supercomputer to reach 1 TeraFlop [47]. The possibility of using generic processors allowed that many organizations to build their own supercomputers and computer clusters.

In 2008, when IBM released the Roadrunner supercomputer, which had nodes composed of 9 processors: one PowerPC and 8 simple vector processors called Synergistic Processing Units (SPU), and it was the first supercomputer to reach the petascale, using more than 20,000 CPUs [47]. In June 2024, the fastest supercomputer is Frontier, from USA. It broke the exascale barrier on June 2022, reaching 1.1 ExaFlops with over 8 million AMDs CPUs [48] [49].

3.2.2 Architecture

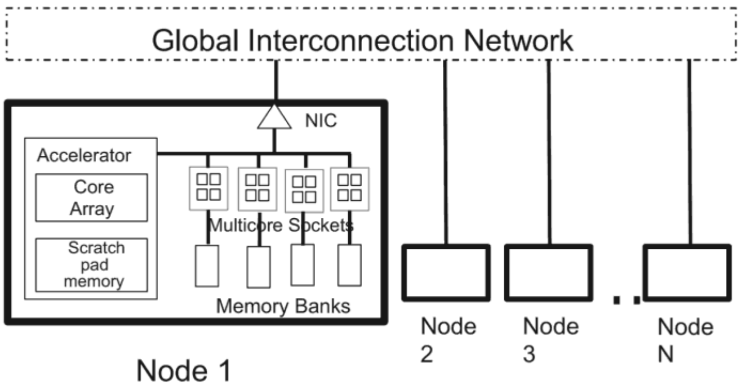


Figure 3.1: Representation of a generic HPC system. Adapted from [3].

A modern HPC system is represented on Figure 3.1. This system has a number of nodes ranging from tens of thousands to millions. The nodes are connected by a global interconnection network. Each node is composed of a set of CPUs (multicores), a set of accelerators, which are mostly Graphic Processing Units (GPUs), and a set of memory banks. Communication among nodes is made through message passing.

An ideal HPC application should be developed to utilize all available parallelism to maximize its performance. This includes parallelism within nodes (using shared memory) and among nodes (using distributed memory). Parallelism reduces the total execution time of an application by breaking it into smaller tasks that can be computed simultaneously.

3.2.3 HPC Programming

Programming HPC applications is complex and many aspects should be taken into consideration. In order to take advantage of the HPC environment, it is necessary to consider the multiple nodes; the network among the nodes and their communication topology; the possibility to use multi-core and shared memory programming; the storage systems, etc. In this section, we explore some tools and concepts about running HPC applications.

Submitting jobs

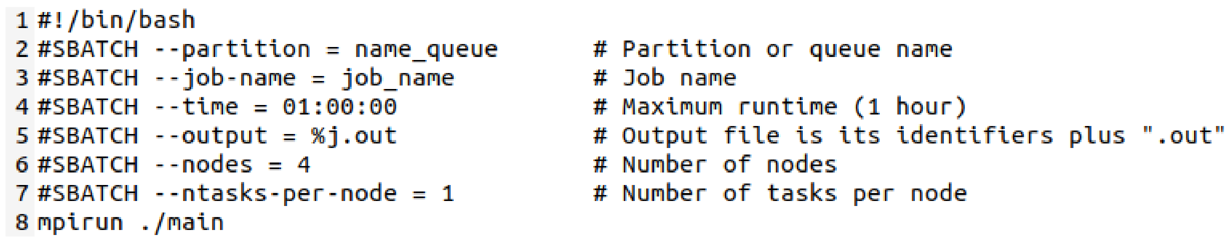
Usually, HPC applications are executed in batch and are called jobs. Many users may submit jobs to the same supercomputer, and they compete for the available resources, which can be exclusive to a user or shared. Usually, each user has a fraction of the total number of nodes available to use.

In an HPC environment, there can be more than one type of node. Logically, there are login nodes and worker nodes. The first type is reserved for user connections, usually via SSH, while the second type is responsible for executing the jobs. Physically, a supercomputer may have various types of nodes. Typically, applications are submitted to the same kind of nodes, which are in the same queue, that consists of a pool of identical nodes. Since many jobs can run simultaneously, a newly submitted job, requesting the use of the nodes in one given queue, might have to wait in the queue to be executed.

Slurm

Before a job is executed, resources must be allocated to it. This is done by a resource management and job scheduler, such as Simple Linux Utility for Resource Management (Slurm), available at *www.schedmd.com/download-slurm/*. Slurm is a scalable and flexible open-source resource manager and job scheduler used in HPC. It manages and allocates computational resources, distributing them to users and applications. It offers a framework for the user to start, to execute and to monitor her/his job. Moreover, it also schedules the jobs and it is able to make contention of resources to allocate jobs that request multiple nodes.

The Slurm's framework uses a script to submit the job to execution. The job script contains the reference to the actual application that will be executed; the reference to input and outputs; the number of processes that will be used; the number of required nodes; the required queue; the time limit, among others. The job script is very similar to any bash script. Figure 3.2 shows an example of a Slurm script.



```
1 #!/bin/bash
2 #SBATCH --partition = name_queue      # Partition or queue name
3 #SBATCH --job-name = job_name         # Job name
4 #SBATCH --time = 01:00:00             # Maximum runtime (1 hour)
5 #SBATCH --output = %j.out             # Output file is its identifiers plus ".out"
6 #SBATCH --nodes = 4                   # Number of nodes
7 #SBATCH --ntasks-per-node = 1         # Number of tasks per node
8 mpirun ./main
```

Figure 3.2: Example of a Slurm script

In Figure 3.2, at line 1, the script's interpreter is set in line 1 (in this case, it is the bash interpreter). In line 2, the name of the execution queue is provided and line 3 provides the name of the job. The maximum job runtime is set in line 4. In line 5, it is set which is the output file, where "%j" is the job's id. In line 6 it is set how many nodes the job will use and in line 7 it is provided the maximum number of processes per node. Finally, in line 8 the execution command is provided.

After configuring the Slurm script, it is ready to be submitted to its queue. To do so, one should type the command:

```
sbatch script_name
```

To monitor the jobs, it is used:

```
squeue -u your_user_name
```

which will show the list of all submitted jobs for the given user. This list shows the job ID, its status, how much time it has already used, how many nodes it is using, and which specific nodes they are.

3.2.4 Message Passing Interface (MPI)

Message Passing Interface (MPI) [50] is a community-driven standard interface for message passing developed for distributed memory parallel computers, allowing the use of multiple nodes in the same application. It consists in a standard API that does message-passing calls, but its implementation may vary. At first, it was developed as a language binding to Fortran 77 and to C, and the first implementation was called MPI over CHameleon (MPICH), released in 1995 by Argonne National Laboratory. Nowadays, many enterprises maintain their own MPI internal implementation, as does IBM and Intel, but using the same standard interface.

Whereas the Slurm allocates resources and can set how many nodes and how many processes per node an application will have, MPI does the communication among the processes. The processes run concurrently, and each MPI process, that is called "rank", has a number from 0 to $n-1$, where n is the number of processes. MPI provides an object named "communicator" that manages the processes' address space and other proprieties, in order to provide the communication among them. An MPI application can contain one or more communicators.

Table 3.1 presents some of the most used MPI's functions. The initialization functions set all the parameters for MPI environment. The standard one is the `MPI_Init`, that does not have thread support. The initialization function `MPI_Init_thread` allows multiple threads to run into the MPI process. There are multiple options for using threads with an MPI process, and each one represents one level of thread support. The simplest thread support is to allow only one thread per process, using `MPI_THREAD_SINGLE`. It is possible to have multiple threads per MPI process using `MPI_FUNNELED`, but then only the main thread is able to make MPI calls. With `MPI_THREAD_SERIALIZED` option, multiple threads can make MPI calls per process, but only one at a time. Finally, the fourth level of thread support is the `MPI_THREAD_MULTIPLE`, in which multiple threads can make MPI calls, with no restrictions. The function `MPI_Finalize` clears the MPI environment.

The remainder functions of Table 3.1 are MPI functionalities. MPI allows to obtain how many nodes are available with the `MPI_Comm_size` function. The function `MPI_Comm_rank` gets the process' rank. Nodes can communicate through messages using blocking (`MPI_Send` and `MPI_Recv`) or non-blocking (`MPI_Isend` and `MPI_Irecv`) functions. The first type is synchronous and the functions block the process which made the call until the corresponding process has executed the MPI call properly. The second type is the non-blocking or asynchronous, and its functions return right after being called, allowing the process to keep working, regardless if the confirmation is received from the corresponding part. To ensure that the messages are properly received, the `MPI_Wait` may be used. It is a barrier that blocks the process until the message is received. As alternative, the `MPI_Test` function may be executed, that checks if the message has already been received or not, verifying the message's flags and status, without blocking the process.

3.2.5 Open Multi-Processing (OpenMP)

In the HPC context, besides executing MPI in multiples nodes, it may be interesting to run multiple threads on a single node, with two levels of parallelism. This concept is called hybrid computing, mixing threads and processes. One of the options to program a hybrid application is mixing MPI with OpenMP.

`MPI_Init`	Initializes MPI environment
`MPI_Init_thread`	Initializes MPI environment for multiple threads
`MPI_Finalize`	Finalizes and restores state set by MPI initialization
`MPI_Comm_size`	Returns number of MPI processes
`MPI_Comm_rank`	Returns process' id (rank)
`MPI_Send`	Sends blocking message
`MPI_Recv`	Receives blocking message
`MPI_Isend`	Sends non-blocking message
`MPI_Irecv`	Receives non-blocking message
`MPI_Test`	Updates flags and status
`MPI_Wait`	Blocks until confirmation

Table 3.1: MPI functions

Open Multi-Processing (OpenMP) [51] is an application programming interface for the shared memory paradigm. It is available for C/C++ and Fortran that allows multiple threads to run with shared memory. OpemMP contains the compiler directives, environment variables and runtime library routines.

Table 3.2 shows some of the OpenMP directives. Directives are used to specify regions of the code that will be processed by threads and in which way. The directive `#pragma omp parallel` specifies that a certain code will be executed by multiple threads. It is also possible to set the visibility of the variables for the parallel code using clauses. The clause `private` sets the variables that will not be shared by the threads; the clause `shared` sets the variables that will be shared among all the threads; the clause `firstprivate` defines variables that will not be shared, their values existed prior the thread invocation and remain the same. Apart from variables clauses, there is also the `num_threads` one, which defines how many threads will be executed.

The directive `#pragma omp single` defines a code region that must be executed by only one thread. This directive must be nested inside a parallel region code. By default, threads not executing the single directive will wait at an implicit barrier at the end of the single region, meaning the thread executing the single directive is not blocked, but rather, all other threads wait until the single directive completes. To avoid this implicit barrier and allow threads to continue executing, the clause `nowait` can be specified alongside the single directive.

The directive `#pragma omp critical` specifies a code region that can be only executed by one thread at a time. This functionality is particularly useful to avoid race conditions when multiple threads are updating the same variable or shared resource. The directive `#pragma omp flush` ensures that all threads have a consistent view of memory for all shared objects. It is also possible to specify particular variables to be synchronized among the threads.

`#pragma omp parallel`	Defines the code to be executed by multiple threads
`#pragma omp single`	Defines the code to be executed by a single thread
`#pragma omp critical`	Defines the code to be executed by each thread at a time
`#pragma omp flush`	Unifies all threads to share the same memory object

Table 3.2: OpenMP Directives

Apart from directives, there are also OpenMP functions. For instance, there is `omp_get_thread_num`, which return the thread's id. The threads id is in integer ranging from 0 to the number of threads less 1. It is also possible to obtain the total number of threads with the function `omp_get_num_threads`.

3.2.6 Lustre

Lustre (*https://www.lustre.org*) is a parallel distributed file system first released in 2003. The name "Lustre" comes from combining "Linux" and "clusters". Its development was initially part of the Department of Energy's Accelerated Strategic Computing Initiative (ASCI) Path Forward program. Lustre was designed to be highly scalable, making it an usual choice for HPC. Not only storage, but also I/O throughput can be increased dynamically. It can support tens of thousands of clients, store petabytes of data, and handle I/O bandwidths reaching hundreds of gigabytes per second. It utilizes high-performance networking infrastructure, including low-latency communication and Remote Direct Memory Access (RDMA) over InfiniBand. Lustre has high interoperability with a dedicated MPI-IO interface, enhancing MPI applications. Moreover, it supports file exportation through widely adopted distributed file system interfaces like Network File System (NFS).

Lustre ensures data and metadata consistency through its POSIX-compliant file system interface, which supports atomic operations for most tasks. Lustre maintains high availability through multiple failover modes that use shared storage partitions and integrate with various high-availability managers. These capabilities ensure that Lustre remains operational and that data remains accessible even in the event of hardware failures.

Due to the I/O performance that Lustre enables in parallel and distributed architectures, it was chosen as the parallel file system for our experiments, which involve multiple nodes. Lustre provides scalability, enabling a single job to execute across multiple nodes while accessing the same file system without becoming a bottleneck, which is beneficial to our framework's goal.

Figure 3.3 presents a typical Lustre structure. Apart from the clients and the Lustre Networking (LNET), every Lustre aspect has two components: server and target. The first one refers to an interface with other components, whereas the second one stores space for the server. Thus, the Management Server (MGS) manages configuration information to all Lustre components; and the Management Target (MGT) stores the MGS, which may be redundant for security. The Metadata Server (MDS) manages the namespace, file names, permissions, etc., whereas the Metadata Target (MDT) stores the same metadata. The Object Storage Server (OSS) answers I/O requests and the Object Storage Target (OST) stores and manages the physical files of content that are sent to user by OSS. Each file is stored in one or more OST. The LNET connects the whole system, providing communication, and it supports many protocols, such as TCP/IP.

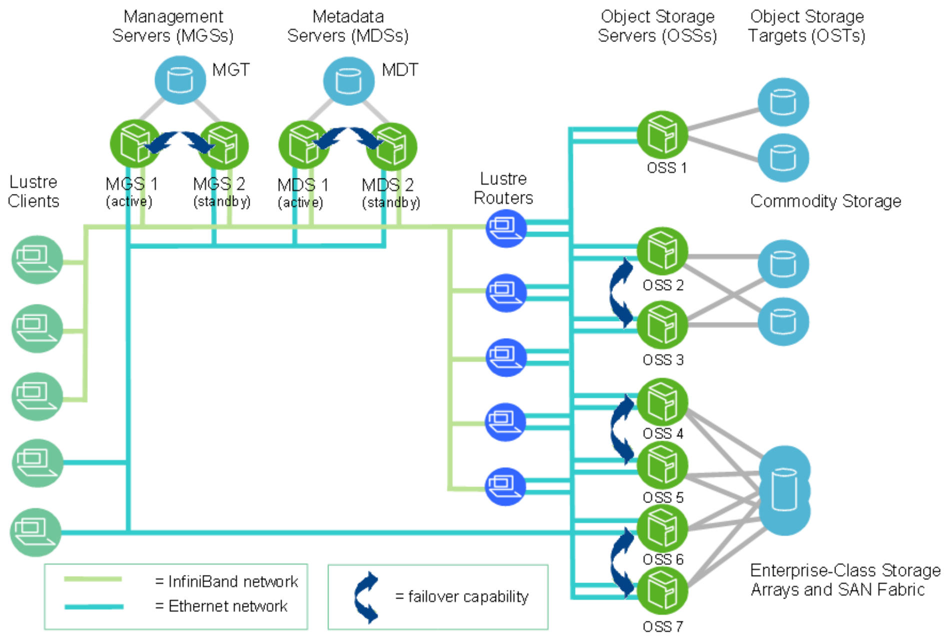


Figure 3.3: Lustre cluster at scale example. From [4].

3.3 Cloud Computing

In late 1990s, Salesforce became one of the first cloud computing providers, offering a cloud solution to manage enterprise sales [52]. In 2006, Amazon launched the Elastic Computing Cloud (EC2) and Simple Storage Service (S3) services, which are *cloud computing* services that allow users to use Amazon's computational resources through the Internet [53]. Afterwards, cloud computing has become increasingly popular. Foster et al. in [44] stated that the constant growth of data and scientific applications had contributed to the rise of the cloud, along with the advent of multi-core architectures. Nowadays, many companies offer cloud services, such as Microsoft with Azure [54] and Google with Google Cloud [55], besides Amazon with Amazon Web Service (AWS).

Regarding the concept of cloud, it is a distributed computing paradigm that, according to Foster et al. [44], "*is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on-demand to external customers over the Internet*".

Foster et al. in [44], aside with Mel et al. in [6] raised some important concepts applied to cloud computing, namely:

- *Virtualization*: is the abstraction through software of a computational resource. The virtual component, such as CPU or network, can be separated from its physical resource [56]. A Virtual Machine (VM), for example, is a virtualization of an operating system over a hardware component. This isolates one VM environment from another in the same host machine, and allows multiple users to share the same hardware.
- *Resource pooling*: is the cloud provider set of computational resources that is available to the cloud users [6].
- *On-demand*: refers to resources availability to a cloud user only when they are in fact demanded to be used. Cloud providers usually offer a *on-demand self-service*, that dispenses previous negotiations between the parties [6]. In this case, the user only pays for the resource he/she has effectively used (pay-as-you-go model).
- *Elasticity* refers to the possibility to the cloud user to require and release resources on-demand from the resource pooling with dynamic-provisioning [6].

The entity that owns the infrastructure resources and offers the cloud as services to cloud users is known as the cloud provider. The cloud user, or cloud costumer, is the entity that uses the cloud. Figure 3.4 presents the main components of a cloud computing system.

3.3.1 Cloud Computing Service Models

The cloud computing services have more than one model option available to cloud users. Each one of them varies in flexibility and user responsibility over the cloud infrastructure, as shown in Figure 3.5. The model of choice depends also on the cloud user purpose and profile, i.e., if it is a end-user or a software developer. This section is based on [6][57][5] and presents the classic cloud computing service models.

IaaS

In Infrastructure as a Service (IaaS), the user deploys her/his own VM in the cloud. The user is capable of installing, managing and executing software on the VMs. IaaS provides

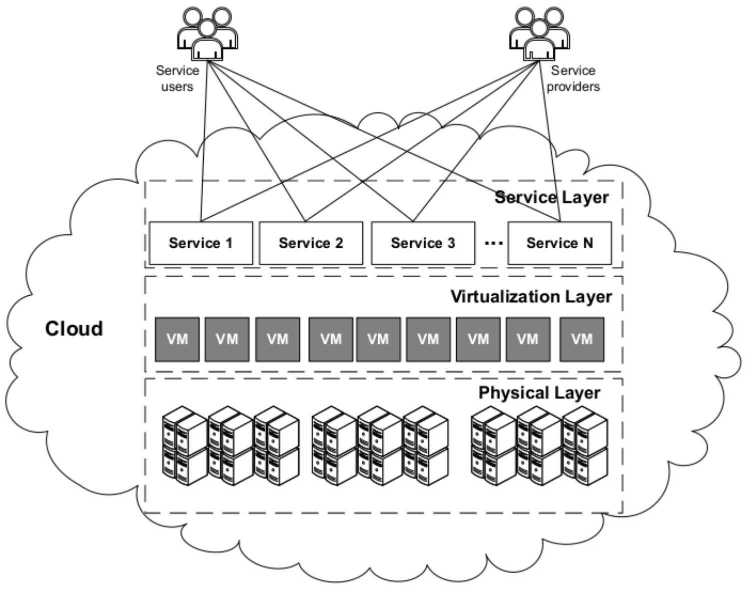


Figure 3.4: Cloud computing system model. Figure from [5].

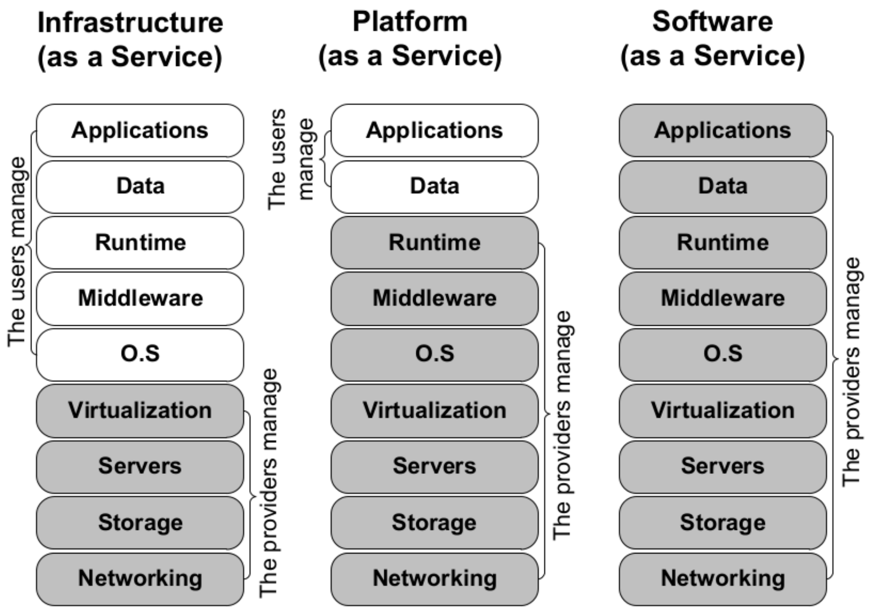


Figure 3.5: Cloud computing service models. Adapted from [6].

high flexibility. The user can also chooses the storage devices, network and computational resources. A well known example of IaaS service is the AWS EC2, that offers VMs running over a myriad of available processors [58].

The flexibility and the infrastructure's control provided by IaaS offers advantages as the possibility to use any necessary tool, such as software and libraries. The flexibility allows quick tools updates and changes in the resources, ensuring that the application can adapt rapidly to new demands and technologies. On the other hand, security remains the responsibility of the end user, which may represent risks if safe practices are not followed.

PaaS

The Platform as a Service (PaaS) is less flexible than IaaS. It offers an environment where the users are able to code, build and deploy their own applications. Usually, each PaaS service offers languages, libraries and tools within a specific domain. Underlying resources, as network, operating system or storage, are out of the user's management capability. One example of PaaS service is the Google Colab, a platform where the user is able to program and run a Python application, using available libraries [59].

PaaS offers less control to the end user, which can prevent potentially dangerous downloads and tool updates, making it ideal for organizations that need to retrain infrastructure access for their developers while still providing a development environment. However, this may lead to difficulties such as tools compatibility issues. Some security features may be difficult or impossible to configure due to provider lock-in, which may also represent some level of risk.

SaaS

Software as a Service (SaaS) is the least flexible model, since it only offers a specific software service. In the SaaS, the cloud runs an application, and only its user interface is accessible to the users. All computing and deployment resources are unavailable. One example of SaaS service is movie and series stream platforms, such as Netflix [60], which offers a service that allows the user to watch a myriad of media ready to be consumed.

SaaS enables the distribution of services regardless of geographic location, allowing end users to access the service from anywhere with an internet connection. It can also provide quick service update to the service's end users. One of the SaaS concerns is the privacy, since the provider may access and store the users' data. It also allows only limited customization.

Other Cloud Service Models

Classic cloud computing services include IaaS, PaaS, and SaaS. However, some authors also consider emerging service models, such as Function as a Service (FaaS), High Performance Computing as a Service (HPCaaS), and the more generic Everything as a Service (XaaS) [61]. FaaS is the common service model for serverless architectures, which are designed to execute applications that respond to specific events with corresponding actions (or functions). These architectures are called serverless because applications only run when triggered, typically operating in containers rather than on VMs [62].

HPCaaS offers HPC architecture and functionalities through a cloud provider (service model explained further in Section 3.4). XaaS is a general term for any service model available in the cloud, providing users with access to various resources and services over the Internet to meet their specific needs [61].

3.4 HPC in Cloud Computing

Apart of the classic cloud models, there are other models, such as High Performance Computing as a Service (HPCaaS), although it can be also classified within the classical models (SaaS, PaaS and IaaS). The model consists of a service providing proper environment to execute HPC applications over a cloud infrastructure [63]. Using HPC in cloud offers to the users the cloud's benefits, as on-demand provisioning, that allows the user to pay for only what she/he has used. Other benefits are the on-demand network access, which provides HPC access to the user regardless of her/his geographical localization; the possibility to choose and configure the VMs in a diverse resource pool; the fast resource provisioning among others [64]. Since the HPC in the cloud usually allows the user to configure the operating system and to install other softwares, the HPCaaS is typically classified as a subset of IaaS.

An HPC cluster in the cloud must be deployed using several basic components. Processors with one or more cores are required, along with a storage system, and possibly accelerators such as GPUs [65]. Additionally, a head node is necessary, through which the user will access the other components via the Internet. Ideally, the network connecting all the components, the Virtual Private Cloud (VPC) network, should be high-performance. Some software tools are also commonly required for HPC in the cloud, such as job scheduler and MPI implementation, and they should be available in the image used to deploy the virtual machine on the nodes.

In order to deploy and to configure an HPC cluster in the cloud, the cloud providers usually offer Infrastructure as Code (IaC) tools to the users. IaC enables provisioning, configuring and managing the cloud infrastructure using source code, in addition to simplifying the setup and the start-up [65]. These tools also allow the user to monitor and to manage the life cyle of an HPC cluster in the cloud. Some of cloud providers and their IaC options to deploy HPC are presented in the next sections.

3.4.1 Microsoft Azure

Microsoft Azure provides two options to manage a HPC in its infrastructure. One of them is the IoC Azure Batch [7], which can be classified as IaaS model. It provides multiple options of IaC languages and frameworks, which include python, .NET and

Terraform. It creates and manages a pool of nodes in the Azure cloud, in addition to install applications. It also has its own job scheduler and resource manager, which are able to calculate automatically the amount of required resources depending on the application necessity. It supports Intel MPI and Microsoft MPI. Besides, Azure offers three options of parallel file systems: Lustre, GlusterFS and BeeFs.

Figure 3.6 presents how usually an application is structured when it is deployed with Azure Batch. Typically, the application server is consumed by final users as a SaaS. On the application workflow, firstly a head node uploads data to a storage service, which will be consumed by multiple worker nodes that execute a parallel job. The final results are uploaded to the storage system and then consumed by the head node.

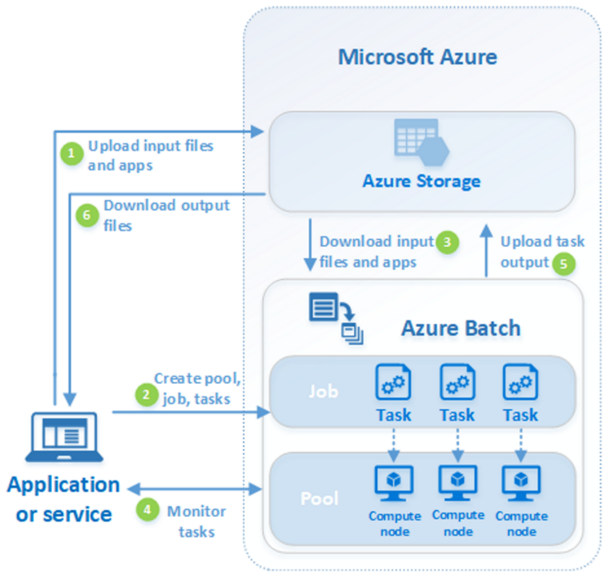


Figure 3.6: Usual workflow of an application deployed with Azure Batch. From [7].

The other Azure HPC manager is the Azure CycleCloud [66]. Unlike Azure Batch, Azure CycleCloud provides multiple third party job schedulers and resources managers, as Slurm, Grid Engine and HPC Pack among others. It provides a command line interface and a web application interface to setup, execute and monitors the HPC infrastructure. The CycleCloud interface does the role of middle-man between the user and the head node. It is installed as an application server and the user communicates to it through Representational State Transfer Application Programming Interface (REST API), thus it can be classified as a PaaS. It also offers an orchestrator that is able to automatically dimension the number of worker nodes.

3.4.2 Google Cloud

Google cloud offers the IoC Cloud HPC Toolkit [67]. It uses the HPC blueprint: a YAML file composed of Terraform or Packer configuration files, called HPC modules, to setup the HPC cluster. Then, another tool, namely "ghpc engine", generates a deployment folder, which is used to deploy the cluster onto Google Cloud. The default job scheduler is the Slurm and it also provides Intel MPI. In addition, Google Cloud offers DDN EXAscaler Lustre as a parallel file system. It offers plenty flexibility to the users, being classified as IaaS.

The HPC modules define many architectures settings, such as compute resources, networking, job schedulers, file systems, and monitoring applications. In addition to DDN EXAscaler Lustre, there are also NFS, Filestore (a high performance network file system) and other types of storage systems. There are also multiple options of job schedulers, such as Google Cloud's Batch and GKE.

Figure 3.7 presents a typical HPC architecture deployed with the Cloud HPC Toolkit. In this case, Slurm is used. Apart from the worker nodes in the compute partition (Figure 3.7), there also debug nodes in the debug partition, the login node and the controller node. Cloud HPC Toolkit allows that different partitions have different instances types.

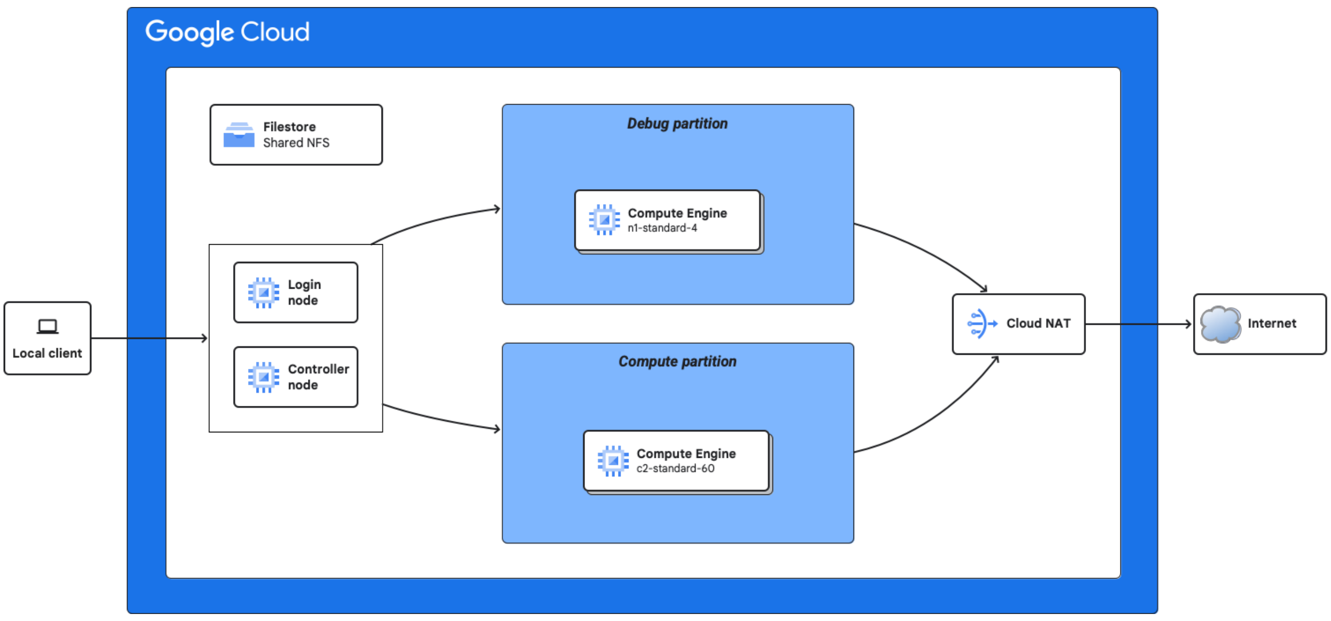


Figure 3.7: Google Cloud HPC architecture diagram. From [8].

3.4.3 AWS ParallelCluster

AWS ParallelCluster [10] is an IoC tool that assists users in deploying, configuring and managing a High Performance Computing (HPC) architecture cluster hosted in the AWS infrastructure, using instances of Elastic Computing Cloud (EC2) [10] and it was this work IoC of choice, along with AWS cloud provider. With ParallelCluster, the users are able to choose the type and number of VMs, alongside with storage systems, virtual network, operating system, etc., to run their applications on, being classified a IaaS. Figure 3.8 shows how a HPC system can be set using ParallelCluster. In this case, the SSH connection is made to the head node, which mounts a file system on AWS FSx for Lustre [4] with connection to the S3 storage. The processes that belong to the application are mapped to the worker nodes using the Slurm scheduler [68]. In this example, GPU instances (p3) are used.

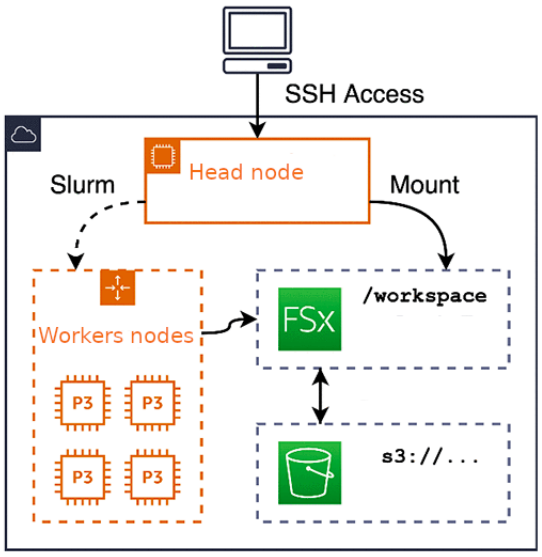


Figure 3.8: Possible design of ParallelCluster system utilization. Adapted from [9].

Figure 3.9 presents the steps to design and to use the ParallelCluster. With the ParallelCluster manager Python wizard, the user configures the ParallelCluster's basic setup. In this step, the user can set how many nodes she/he wants to have on her/his cluster and if she/he wants the ParallelCluster to create a Virtual Private Cloud (VPC) automatically or not. Then, it generates a configuration YAML file, which allows the user to further customize the ParallelCluster's design. For example, it is possible to set the chosen storage system on the configuration file. Afterwards, the wizard assists the user to deploy the cluster based on the YAML file. Then, the wizard also offers the option to connect to the cluster via a SSH client. Once connected, the user finally is able to submit jobs through a job scheduler (eg: Slurm). The complete documentation presenting the configuration options and rules is available on [69].

The storage systems can be classified into shared and local. The local storage is the one attached to the AWS instance. It is ephemeral, that is, once the instance is deallocated, the data is deleted. Since the local storage device is locally attached to the

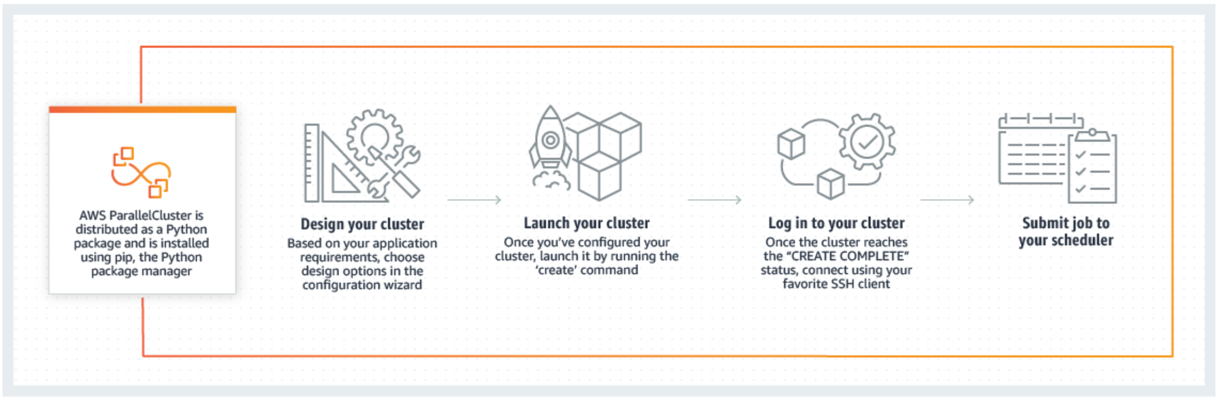

AWS ParallelCluster is distributed as a Python package and is installed using pip, the Python package manager
Design your cluster
Based on your application requirements, choose design options in the configuration wizard
Launch your cluster
Once you've configured your cluster, launch it by running the 'create' command
Log in to your cluster
Once the cluster reaches the "CREATE COMPLETE" status, connect using your favorite SSH client
Submit job to your scheduler

Figure 3.9: AWS ParallelCluster functioning from [10]

hardware component, it is supposed to have a lower latency than other more distant storage devices. Due to those characteristics, the local storage system is commonly used to store temporary files.

Chapter 4

Related Work

In this chapter, we present some related works about workflow scheduling in the cloud. We discuss the objectives, the technique and the results of each one. We also classify them according to the taxonomy discussed on the Section 2.2 (references [1] and [2]). At the end of the chapter, we present and discuss a comparative table.

4.1 Methodology

To select the works explored in this chapter, we followed a sequence of steps. First, we used a set of search strings in Google Schoolar (*https://scholar.google.com/*), namely:

- "scientific workflow", "cloud" and "scheduling";
- "workflow", "cloud" and "scheduling";
- "linear workflow" and "scheduling";
- "multi workflow" and "scheduling".

We then filtered for works published after 2014, focusing on recent research. We prioritized articles from reputable journals and conferences.

For an initial analysis, we reviewed each paper's abstract, introduction, and conclusion. At this step, we selected papers focused on workflow scheduling in the cloud, giving preference to those centered on scientific workflows. We also aimed to include a range of different solutions, avoiding repetition of similar solutions. Finally, we read each article in its entirety and carried out a detailed analysis.

4.2 Durillo and Prodan, 2014

The work proposed by Durillo and Prodan in [70] aims to schedule a single scientific workflow in the cloud with multiple objectives. It uses the algorithm Multi-Objective Heterogeneous Earliest Finish Time (MOHEFT), a multi-objective version of the algorithm Heterogeneous Earliest Finish Time (HEFT) [71], to achieve the Pareto front, considering the objectives of minimizing the makespan and minimizing the cost. The MOHEFT algorithm presents a list of solutions, which enables the user to choose the one that suits him/her more.

The original algorithm, HEFT, uses a heuristic to rank the tasks. The longest path between each task and the final task of its workflow is this heuristic. Each task is then mapped to the resource that will finish earlier to compute it, thus finding the optimal scheduling of a workflow in a heterogeneous environment. MOHEFT also starts ranking the tasks in the same way as HEFT, but instead of mapping them onto the resource that will finish them earlier, it will create a list of possible solutions. The solutions must be valid, non-dominated and diverse, exploring the entire solution space uniformly, avoiding local optimal solutions [70].

The authors made the experiments with simulated and real workflows in the AWS EC2 cloud, with makespan and economic costs as objectives. In the results, MOHEFT had more diverse solutions than the multi-objective workflow scheduler SPEA2* [72], which means that the users have more possibilities to choose among the tradeoff. Both MOHEFT and SPEA2* found the lowest cost. MOHEFT is classified as static according to both taxonomies, [1] and [2].

4.3 Sadooghi et al., 2016

In [73], I. Sadooghi et al. proposed Albatross, a task scheduler and execution framework designed for multiple workflows of data analytics applications with Map-reduce workloads. It is designed to run in the cloud or in other types of distributed systems, similar to Spark [74] and Hadoop [75]. Albatross has distributed schedulers over the nodes, which pull tasks to themselves, instead of receiving from a centralized scheduler. In the Albatross framework, all nodes have their own distributed hash table and their own distributed message queue. The first one manages the metadata, whereas the second one contains a copy from the main queue, i.e. all the tasks and their dependencies, providing parallel access to all the tasks while also guaranteeing that no task is read by more than one node.

The Albatross framework starts distributing the input dataset to all the nodes; then, it distributes tasks among all the workers' local queue using a hash function in a random manner, pursuing load balance. In order to achieve data locality, when a node verifies that it does not have a given task's data, it tries to send the task to the node who has it. If the node is overloaded, the task goes to the end of the first node local queue, that will try to send the task once more, later. If the other node is still overloaded, the first node imports the task's data, improving load balance.

The absence of a centralized scheduler has similarities with the work stealing scheduling, in which the nodes pull tasks or workflows to themselves. However, Albatross' target application has a large number of short-lived tasks, typical from data analytics applications and different from many scientific workflows, whose coarse-grain tasks may have long duration.

Albatross was tested on AWS EC2 with benchmarks and real applications and its throughput outperformed Spark's and Hadoop's when the applications had a high level of task granularity. Since no information is used to schedule the task at first, Albatross' scheduler can be classified as dynamic in [1], but as runtime refinement in [2], due the task movement to adjust locality.

4.4 Rodriguez and Buyya, 2017

Rodriguez and Buyya proposed "BAGS" in [76], a scheduler and resource provisioner that minimizes the makespan of a single scientific workflow in the cloud under a budget constraint. The proposal is to firstly divide a DAG into Bag-of-Tasks (BoT), so that every task of the same level can be executed concurrently, which can create homogenous BoTs (all the tasks are the same), heterogeneous BoTs (with different tasks) and single-task BoTs. Then, assuming that the faster a VM is, the more expensive it is, the BAGS' system increases the VMs' price of each task iteratively, until the most expensive VMs are chosen when they still are under the budget.

To provision VMs for homogenous tasks in a BoT, the Mixed Integer Linear Programming (MILP) model was applied, which was designed to estimate the number and types of VMs that can be afforded within a given budget, ensuring tasks are processed with a minimum makespan. A provisioning plan is executed for each BoT, allocating the type and quantity of VMs according to the budget. During the scheduling, the tasks are sorted based on their estimated execution time, and the longer tasks are executed first, using a Max-Min algorithm. However, if there is one or more idle VMs from another BoT provisioning plan, a task can be allocated on it, if it is at least as fast as the task's original scheduled VM. This feature allows the reuse of VMs and diminishes over-provisioning.

BAG was simulated with CloudSim and its performance was compared to GreedyTime-CD and Critical-Greedy, and overall, BAG had better makespans than the other algorithms and yet could meet the budget. It can be classified as static according to [1], since it uses a lot of previous information of tasks and resources to schedule before the execution, and as runtime refinement according to [2], since it can reschedule tasks to reuse idle VMs.

4.5 Stavrinides and Karatza, 2021

In terms of objective, the most similar work to ours is [25], as its goal is to schedule multiple linear workflows in the cloud, aiming to reduce the makespan of workflow applications. However, the work considers an online system that receives workflows at various points in time and allows changes on the linear structure, i.e., the task number increases or decreases. To address this, the authors propose a dynamic and heuristic scheduler that assumes the workflows' tasks execution time is known in advance.

The work utilizes the Periodic-Shortest-Cumulative-Time-First algorithm to prioritize workflows in the queue for each computational resource, placing the workflow with the smallest cumulative service time at the front. At the end of each time period, the cumulative time is recalculated. It considers that each workflow will be processed on the same computational resource to reduce data transfer time between tasks. This scheduler periodically reevaluates the priority queue, due the possible change in the number of tasks. Additionally, new workflows may arrive. The Two Random Choices technique is used to map workflows to computational resources, that is, the resource with the shortest queue between two randomly selected nodes, will be chosen to receive a new workflow.

When compared to the baseline First-Come-First-Served scheduler, the proposed scheduler exhibited advantages ranging from 14.6% to 25.3% in makespan reduction. Since this approach only monitors and reschedules prior to the execution and using information about the workflow metrics, it can be classified as a static scheduler according to [1] and to [2].

4.6 Krämer et al., 2021

Krämer et al.'s work [77] differs from most of other works (except for [73] and ours) by not requiring any task data other than an initial workflow structure. The authors' goal is to develop a scientific workflow management system to be run in a distributed environment, such as cloud, that supports workflows that are not DAGs, i.e., those with inner loops and dynamic structures that are revealed during runtime. The authors also aim to develop a simpler system to manage cyclic workflows than other existing solutions. Thus, those goals are different than our work's goal, which is primarily focused on makespan.

In order to achieve their goals, Krämer et al. proposed an algorithm that receives a YAML file that contains the structure of a workflow. Their algorithm identifies *process chains*, which are linear subworkflows that take part of a greater original workflow, and are generated from workflow inner loops. A scheduler allocates resources dynamically to compute each linear workflow.

Compared to the workflow management system Pegasus [78], the work of Krämer et al. enables cyclic workflows, which Pegasus does not support. Pegasus requires prior knowledge of the workflow structure, unlike [77]. This scheduler can be classified as dynamic according to [1], and as subworkflow scheduling according to [2], since it schedules each one of the linear subworkflows only once.

4.7 Chen et al., 2021

The scheduler proposed on [79], namely unceRtainty-aware Online Scheduling Algorithm (ROSA), also schedules online multiple workflow applications, but its objective is to meet the workflows deadline, while trying to minimize the cost. Based on an estimated execution time, the algorithm computes the Predicted Latest Start Time (PLST) of each task, which is the latest time that a task can start, so that it does not exceed the workflow predicted finish time threshold, namely the deadline. This prediction is made with uncertainty, thus a value is added to the PLST, which is based on probability.

The uncertain value is propagated through the workflows' PLST tasks, however, once a task is finished, its uncertainty disappears, and the PLSTs of its successor tasks are calculated. The ready tasks are sorted in a non-descending order based on their PLST. An estimated instance cost is also computed for each task. At the end of the algorithm, the task is mapped to the computational resource with the smallest cost that does not exceeds the deadline.

The results show that this work outperformed other algorithms, being more cost-effective than EPSM [80], which exceeded the cost limit by 10.07%. The ROSA algorithm must receive tasks duration beforehand, but it recalculates their expected finish time and it can do allocations changes if it perceives a delay that would exceed the deadline, thus it can be seen as a dynamic scheduling according to [1] and to [2].

4.8 Taghinezhad-Niar et al., 2022

Similarly to Section 4.7, the work of Taghinezhad-Niar et al. [81] employs heuristics considering uncertainties in task duration and dynamically adjusts workflow applications as needed to schedule multiple workflows in an online platform. The work in [81] aims to maintain the execution of workflows within cost, time, and energy constraints. Two schedulers are proposed: (a) Energy and Uncertain task ET aware workflow Scheduling Framework (EUSF) and (b) CUSF, its Cost-aware extension, to reduce costs for the end user. Both leverage stochastic task execution times to calculate the resource to which each task is allocated.

Tasks are prioritized based on the Earliest Deadline First and executed when all their preceeding tasks have been completed. Furthermore, energy-aware and cost-aware algorithms provision resources based on energy and cost constraints, respectively, for task allocation. Parameters such as Earliest Start Time and Earliest Finish Time are adjusted at the end of the execution of each task, potentially influencing the resources allocated to subsequent tasks in a cascading manner.

When compared to the Energy and Resource Efficient workflow Scheduling (ERES) algorithm [82], EUSF achieved a 15% energy savings and a 34% cost reduction. Both EUSF and CUSF increased the deadline meeting success rate by almost 15% and 55%, respectively, compared to onliNe multi-workflOw Scheduling Framework (NOSF) [83] and ERES. The scheduling strategy can be classified as dynamic according to both [1] and [2].

4.9 Xia et al., 2023

The work of Xia et al. [26] employs heuristic algorithms to minimize the makespan and the cost of online scheduling workflow applications, each one of them with a single workflow. Firstly, the algorithm maintains a ready tasks pool, with all the workflows' tasks that are ready to be executed. Then, it calculates a priority rank of each task from the ready task pool based on the longest path between each task and the final task of its workflow, alongside with the sum of the estimated duration of the tasks in the path and their data transmission time between tasks.

The task with the highest priority rank is selected and goes to a resource allocation phase based on Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS) [84] to find the minimum makespan and cost. TOPSIS is a method to make decisions based on multiple objectives, which, in this case, are: minimizing workflow's makespan and minimizing the cost. It calculates the possible choices and chooses the option which is the closest to the best choice of both objectives simultaneously. With the best resource chosen, the task is sent to a ready queue of that specific resource.

When compared to other similar algorithms, as [85], the work [26] achieved the lowest makespan for all cases with different workflow arrival intervals. It uses known information about tasks, resources and communication, but, since it recalculates tasks scheduling after they arrive on the ready pool tasks, it can be classified as an dynamic scheduling according to [1] and [2].

4.10 Comparative Table

Table 4.1 presents the main characteristics of the works discussed in this chapter. The terms "mult" means multiple; "sing" means single; "r ref" means runtime refinement; "s sch" means subworkflow scheduling; "dyn" means dynamic. In addition, "info" refers to the information required to do the scheduling, "wf" means workflow, "app" means application and "sched class" means scheduling classification.

Table 4.1: Comparative table of works on workflow scheduling in clouds

Paper	**Year**	**Info**	**Goal**	**Wf type**	**Wf in app**	**Platform**	**Technique**	**Sched Class**
Durillo [70]	2014	exec time & transf time	makespan & cost	generic	sing	AWS EC2	sched all tasks & MOHEFT	static
Sadooghi [73]	2016	none	min. data movement & fairness	generic	sing	AWS EC2	pulling & locality	r ref dyn
Rodriguez [76]	2017	exec time & transf time	makespan & cost	generic	sing	simulated	sched ready tasks & MILP	r ref static
Stavrinides [25]	2021	exec time	cost & resrc util & fairness	linear	sing	simulated	sched ready tasks & queue system	static
Krämer [77]	2021	none	fairness	generic	sing	AWS EC2&	sched ready tasks & FCFS	s sch dyn
Chen [79]	2021	exec time	cost	generic	sing	simulated	sched ready tasks & queue system	dyn
Taghinezhad [81]	2022	exec time	makespan & cost& energy	generic	sing	simulated	sched ready tasks & EDF	dyn
Xia [26]	2023	exec time & transf time	makespan & cost	generic	sing	simulated	sched ready tasks & decis matrices	dyn
Our work	**2024**	**none**	**makespan**	**linear**	**mult**	**AWS EC2& Parallel Cluster**	**sched all tasks & re-sched with work stealing**	**r ref dyn**

It can be seen in Table 4.1 that the only work that considers an application composed of multiple workflows is ours. Moreover, most of the works assume that the execution time of the tasks and/or the resources' data are known previously. Most of the works were evaluated in simulated cloud environments (e.g. CloudSim). The works [77] and [73] do not use information about tasks, and the work [77] was evaluated in AWS EC2 and Terraform, and since [77] is a workflow management system, its goal is to provide fairness. In [73] it is actually assumed that workflows' tasks are many and executed very quickly, typical from data analytics workflows, and it also pursues fairness.

In the column 'Wf in app,' we present the number of workflows for each application. All of the works we found, except ours, schedule each workflow application individually, i.e. they schedule a single workflow application. Our work, however, manages a multi-workflow application, scheduling multiple workflows within the same application to reduce its overall makespan.

Apart from our work, two works used runtime refinement: [73] and [76], which means that, after a previous scheduling, during runtime tasks are rescheduled. While [73] uses it to maintain the fairness, [76] uses the runtime refinement to avoid overprovision of VMs and idle time. To our knowledge, this is the first work which schedules an application composed of multiple workflows in a real cloud, with no a priori information on execution or transfer time, using work stealing, aiming to reduce makespan.

Chapter 5

Design of the Framework

The framework proposed in the MsC dissertation was developed in two moments. First, we designed an MPI/OpenMP framework to execute applications composed of multi-workflows in the cloud. This is called version 1 in this document. Then, we included the work stealing policy into our framework, generating version 2.

In Section 5.1, we discuss the linear workflows scheduled in our framework. The version 1 of our Framework is presented in the Section 5.2, whereas Section 5.3 presents version 2.

5.1 Linear Workflows Considered in the Framework

Figure 5.1 presents the linear workflows scheduled in our framework. The multi-workflow application is composed of m workflows which execute the same n tasks, and each task constitutes one processing step. Each workflow receives a different input file (or files) and produces an output file (or files). In the processing steps, intermediary files may be produced.

It is assumed that each workflow has at least two tasks. The concept of workflow usually means that the output of a task is the input of the next one, however, in our approach, we do not restrict to this concept and the tasks may have independent inputs/outputs or even no input at all. Nonetheless, the workflows still have temporal dependency. That means that, for example, that each task may generate an independent output that will not be processed by the next task. Nevertheless, in the linear structure, t_i will only be executed when t_{i-1} finishes its execution.

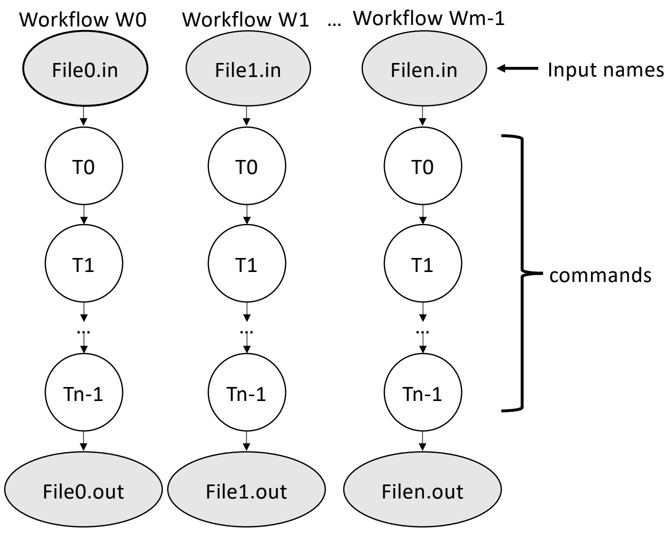


Figure 5.1: Application composed of linear workflows scheduled in our framework.

5.2 Framework Version 1

The version 1 of the proposed framework executes a multi-workflow application in a multi-node environment inside the cloud or in an HPC architecture, where each node has one process composed of multiple threads, embodying a hybrid parallel architecture. Nodes communicate through message passing (e.g. MPI) and threads inside each node communicate through shared memory (e.g. OpenMP).

Figure 5.2 shows an overview of Framework version 1. Each node has at least one working thread and only the $Node_0$ has a control thread. In the beginning, the $Node_0$'s control thread reads two files: workflow skeleton and input files names. The workflow skeleton contains an ordered list of n commands, one command per line, and the input files names contains m names of the input files. The $Node_0$'s control thread instantiates m workflows, with n tasks each, which will process different input files and stores the instantiated workflows in the Total Workflow Array (TWA), which has m entries and contains the information about all workflows, each one in a specific position. After that, $Node_0$'s control thread sends the TWA to all nodes. Upon reception, the TWA is put on shared memory and, in a self-assignment and striped way, it is used by the working threads in each node (including $Node_0$'s working threads) to determine which workflow to execute. The framework ends when all workflows are executed.

5.2.1 Initialization

To initiate the application, $Node_0$'s controller thread reads two files: (a) workflow skeleton, which contains an ordered list of commands, with their input marked as \$, and (b)

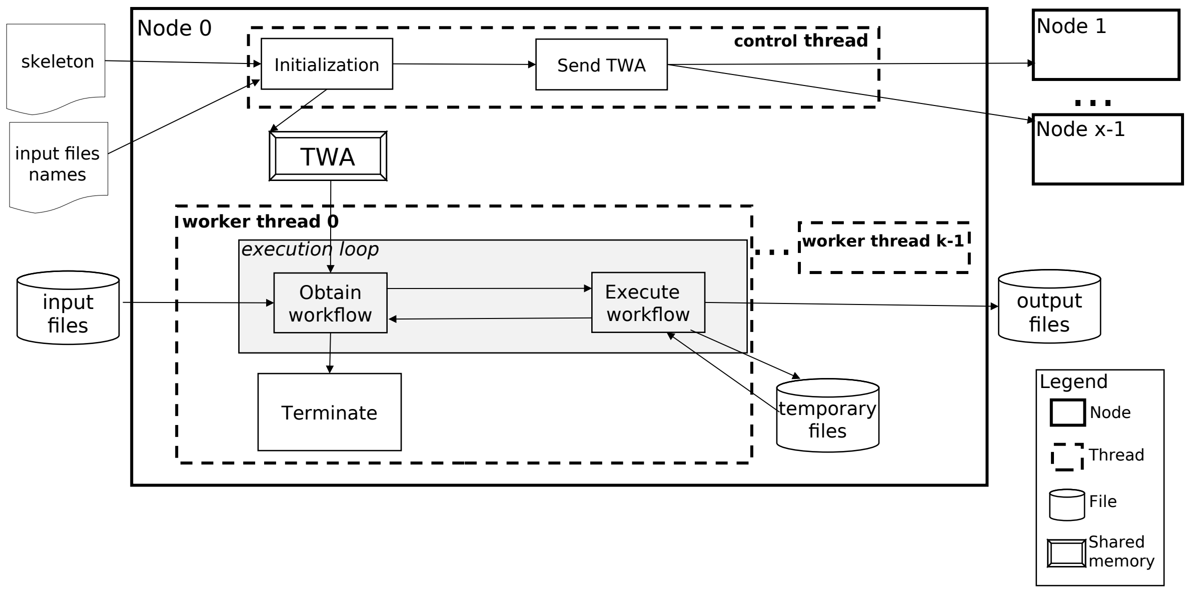


Figure 5.2: Overview of the architecture of the Framework version 1.

workflow input files names, with the actual workflow input file names. Using (b), a list sorted by the sizes of the input files is created.

Subsequently, on $Node_0$, the skeleton and input name files are processed, generating the Total Workflow Array (TWA), with instantiated workflows. In this step, every \$ is replaced by the actual file name (Figure 5.3). Following this step, the TWA is sent as a message to all other nodes (Figure 5.2), i.e., the TWA is fully replicated.

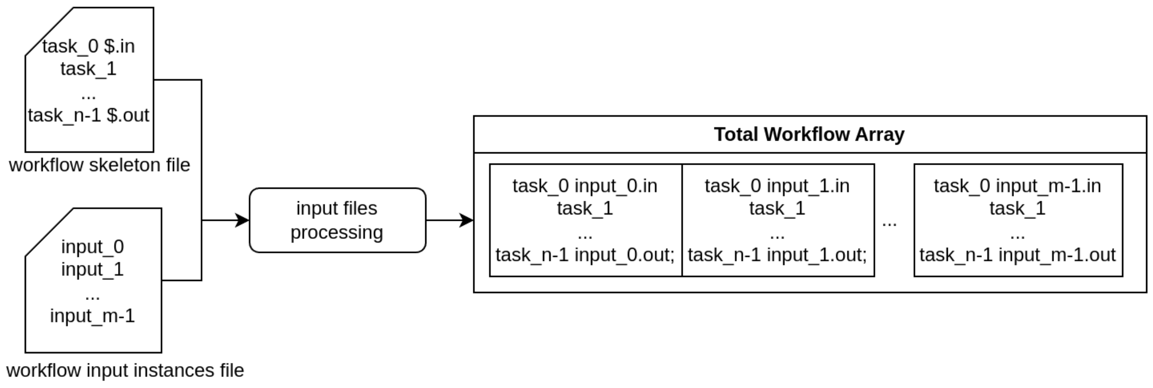


Figure 5.3: Skeleton and input file processing, for m workflows, each one with n tasks, generating the Total Workflow Array (TWA).

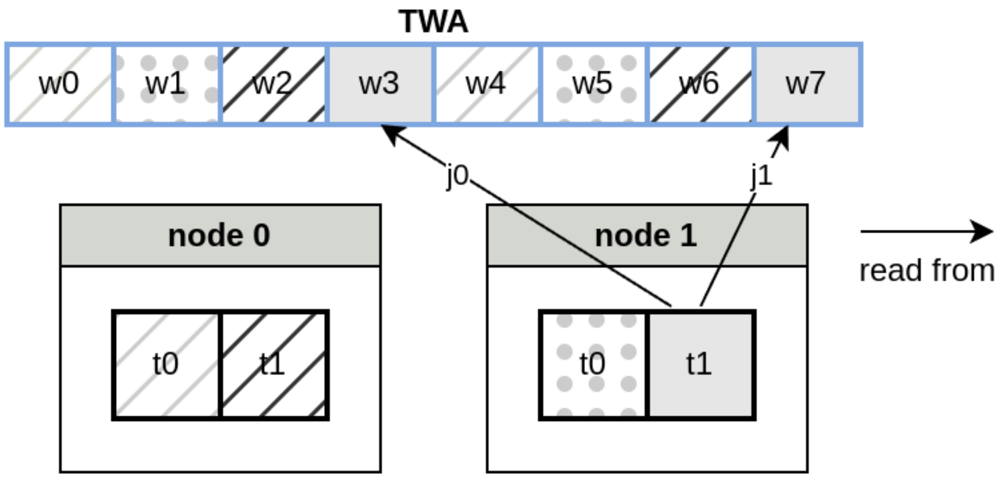


Figure 5.4: Striped assignment of workflows to threads in Version 1.

5.2.2 Workflow Execution

In order to allocate workflows to threads, we opted to use a function, in order to avoid communication at allocation time. The TWA indexes are locally accessed by the threads inside the nodes in a striped way. The working threads use the Equation 5.1 to obtain the index ($index_j$) of the workflow they will compute.

$$index_j = (n * t * j) + (n * Thread_i) + Node_k \tag{5.1}$$

In this equation, $index_j$ is the TWA index, n is the number of nodes, t is the number of working threads per node, $Node_k$ is the node identifier, $Thread_i$ is the thread identifier and j is the iteration number.

For example, in an application with 2 nodes, 2 threads per node and 8 instantiated workflows, $Thread_1$ of $Node_1$ will execute the workflow in TWA[3], where $index_0 = 2 * 2 * 0 + 2 * 1 + 1 = 3$ at the first iteration ($j = 0$) and in TWA[7], where $index_1 = 2 * 2 * 1 + 2 * 1 + 1 = 7$ at the second one ($j = 1$), as shown in Figure 5.4. This way, the workflows executions are distributed in a striped way among the nodes and among the threads within the nodes iteratively.

The application execution finalizes when all its workflows are concluded. If a node finishes its assigned workflows before other ones, it will await all of them to be released.

5.3 Framework Version 2

The version 2 of our framework is an improvement of version 1, thus it also executes the same type of multi-workflow application in the cloud or in an HPC architecture. Version

2 has of two modes: (a) *standard* mode and (b) *work stealing* mode. The *standard* mode functioning is similar to the one presented in Section 5.2: the $Node_0$'s control thread receives the skeleton file and the input files names and generates the TWA, which will be sent to every other node (Figure 5.5). Just as in version 1, in the *standard* mode, the threads use self-assignment striped functions to select the workflows to be executed, with no synchronization.

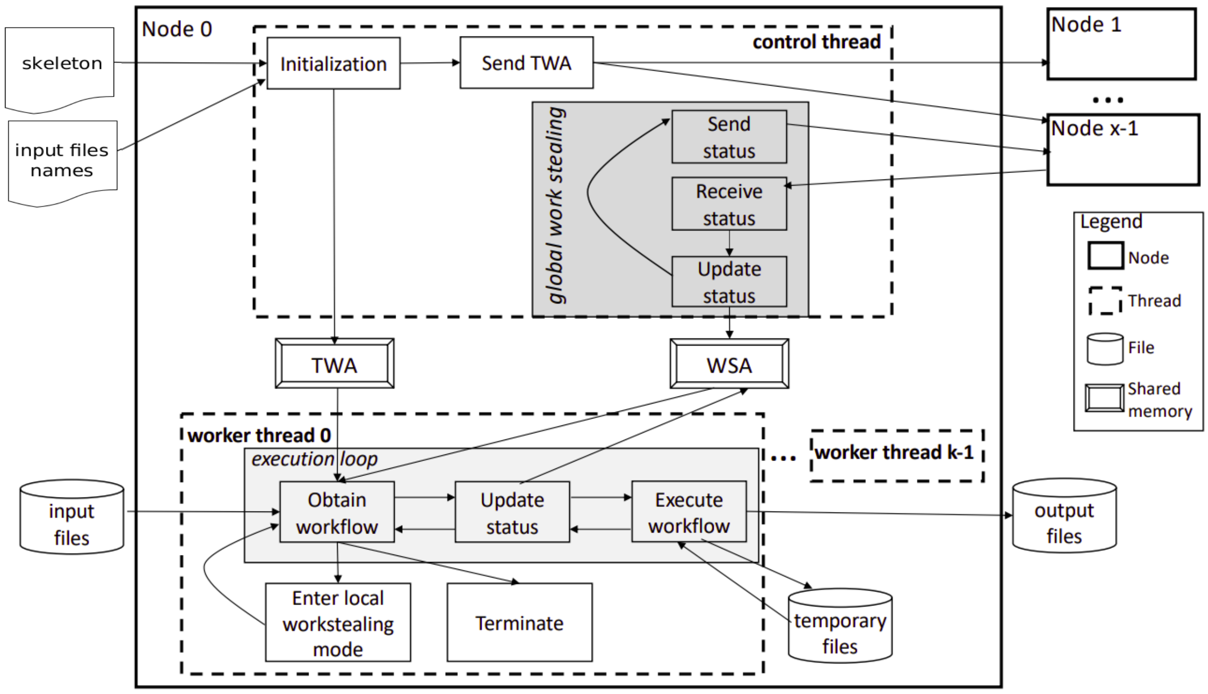


Figure 5.5: Overview of the architecture of the Framwork version 2, with work stealing.

The main difference between versions 1 and 2 is the addition of the *work stealing* mode in version 2. When a thread of a given node becomes idle for the first time, it enters in the local work stealing mode (Figure 5.5). Another difference between the versions is the management of the workflows' statuses. In version 2, each node has a control thread that initiates and, during the global work stealing, updates the Workflow Status Array (WSA), which contains the statuses of each one of the workflows assigned to it: *waiting*; *started*; *done*. Besides, the worker threads also updates the WSA when they begin and conclude a workflow. At the beginning, all statuses are set to *waiting*.

At the execution's beginning, the working threads enter in the *standard* mode and execute their assigned workflows by locally accessing the TWA and updating the workflow status at the WSA, both at the beginning (*started*) and at the end (*done*) of each execution (execution loop in Figure 5.5). The temporary files are written into a local file system and the output is written to a shared file system.

When a thread of a given node first becomes idle, its node enters in the local work stealing mode. In local work stealing, the idle threads first steal workflows from threads that belong to their node. If all workflows are at the *done* state that node, the threads start to steal workflows from threads in other nodes (global work stealing). In both cases, local or global, the data structure (WSA or TWA) is accessed by the stealer from the end to the beginning, in search of a workflow in the *waiting* state, i.e., its execution has not started yet. The application terminates when all threads of all nodes finish to process all workflows.

5.3.1 ***Standard*** mode

Just as in Framework version 1, in version 2 *standard* mode, $Node_0$ firstly receives the skeleton file and the input file names file. $Node_0$ uses these files to create the TWA (Figure 5.3) and sends a copy of it to every other node (Figure 5.5). As soon as a node receives the TWA, its control thread initiates the Workflow Status Array (WSA), which contains all instantiated workflows statuses assigned to that node, and sets all elements of this array as *waiting*. The *standard* mode avoids communication between the nodes and also uses a striped function to access the TWA.

As in version 1, in version 2 the workflows assignment to each thread is done in a striped manner, using Equation 5.1, in order to calculate the TWA index to be accessed. In Figure 5.6, the same example used in Section 5.2.1 is shown, in which there are two nodes with two threads each, thus the calculated TWA's indexes are the same. However, in the version 2 of our framework, there is an additional structure in each node: the WSA. When a thread reads a workflow from the TWA, it updates its status in its node's WSA, from *waiting* to *started*. After the workflow is finished, the thread updates its status to *done*. When a thread becomes idle, it may initiate the *work stealing* mode or, if there is no workflow steal, the thread terminates itself.

5.3.2 ***Work Stealing***

When a working thread of $Node_s$ concludes its assigned workflows and becomes idle, it starts to steal workflows inside the node entering in the local work stealing mode (Figure 5.5). The stealer thread accesses the WSA, from back to front, looking for workflows in the *waiting* state. If such a workflow is found, its state is changed to *started* and the stealer thread executes the workflow.

If there are no workflows in the local WSA in the waiting state, global work stealing starts, and the state of the node ($Node_s$) is changed to *stealer*. The control thread of

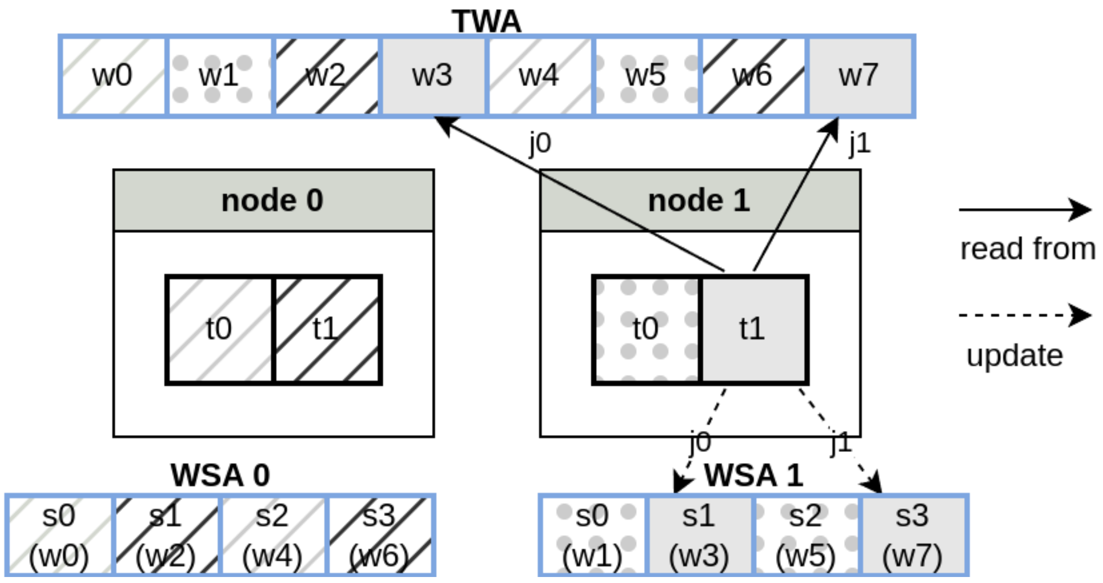


Figure 5.6: Striped function assignment with TWA and WAS.

$Node_s$ computes which node will be *stolen* ($Node_t$), using Equation 5.2

$$Node_t = |Node_s - n + 1| \tag{5.2}$$

where n is the number of nodes, $Node_t$ is the node to be stolen and $Node_s$ is the stealer. The Figure 5.7 shows which node is selected either to steal or to be stolen by its pair. The first node to conclude its assigned work, will steal from its pair, which can vary at each job execution.

The $Node_s$ sends the *ws* message to $Node_t$, requesting its WSA (WSA_t). Upon reception of the *ws* message, $Node_t$ becomes *stolen* and sends its WSA (WSA_t) to $Node_s$, which updates its own copy of WSA (WSA_s) (Figure 5.6). $Node_s$ accesses then its WSA_s from the end to the beginning, until it finds a workflow that is marked as *waiting*. The *stealer* thread then marks this workflow as *started* in the local WSA_s and starts the execution of the *stolen* workflow. When the workflow execution terminates, its status is set to *done*.

The WSA exchange occurs regularly between $Node_s$ and $Node_t$, and reaches its end when any of those nodes finish to search for *waiting* workflows. Since WSAs are exchanged periodically between $Node_s$ and $Node_t$, a situation may arise where $Node_s$ (*stealer*) chooses a workflow in the *waiting* status but, meanwhile, $Node_t$ (stolen) has started its execution. In this case, $Node_t$ will notice this in the next communication period and will abort the workflow's execution. Since we are dealing with long-lived workflows and temporary files are written into the local node's file system, we guarantee that the correct output will be produced. Therefore, our work stealing approach is idempotent (Section

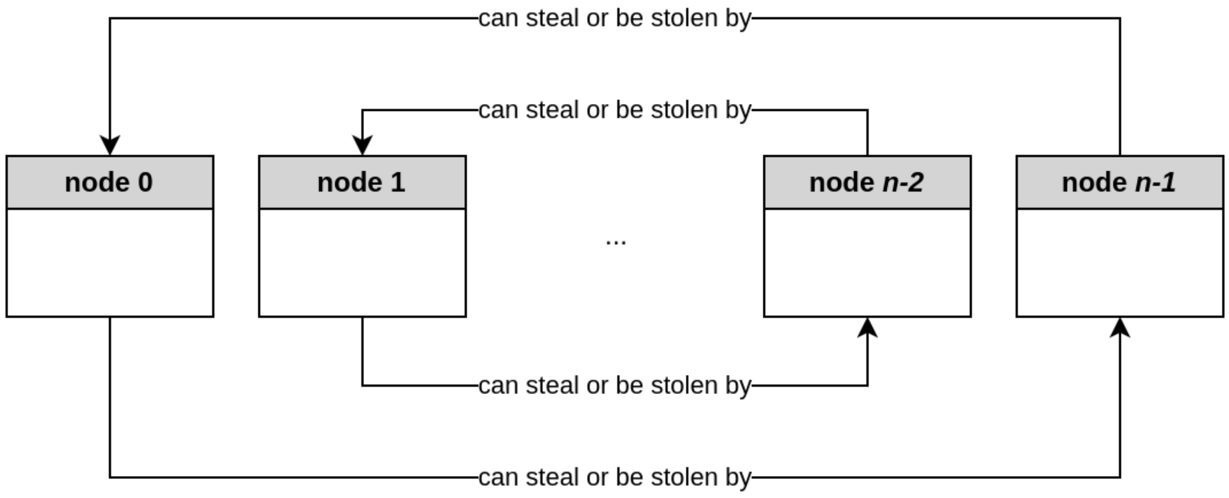

can steal or be stolen by
can steal or be stolen by
node 0
node 1
...
node n-2
node n-1
can steal or be stolen by
can steal or be stolen by

Figure 5.7: Global work stealing node selection.

2.3), but the duplicate processing will only occur within one communication period (e.g. 1s) between nodes $Node_t$ and $Node_s$.

Regarding the *stolen* node, the another difference between the *standard* mode and the *WS* mode is the periodic exchange of WSA (Figure 5.5). $Node_s$ sends a message to $Node_t$ with WSA_s. When $Node_t$ receives WSA_s, it updates its local copy. If the same workflow is in the *started* status in WSA_t and WSA_s, it means that both nodes started its execution. In this case, $Node_t$ terminates the execution of the workflow, marking it in WSA_t as *done*. So, in the point of view of the stolen node, this workflow's execution is responsibility of $Node_s$. For the other cases, WSA_t is updated with the same statuses of WSA_s (*done* and *waiting*). When all assigned workflows of $Node_t$ or $Node_s$ are set as *done* or *started*, the node's control thread sends the last message, warning that the WSA exchange will end.

Similarly, when the *stealer* node $Node_s$ asks and receives WSA_t from $Node_t$, it marks every *started* or *done* element from WSA_t, to *done* in WSA_s. When a working thread becomes idle, it accesses the WSA_s, from end to beginning, and looking for the first workflow in the *waiting* state (i). When it is found, the working thread uses the index i of WSA_s in Equation 5.3 to access the workflow in $TWA[inverted\_index_i]$.

$$inverted\_index_i = i * n + Node_t \tag{5.3}$$

The working thread sets $WSA[i]$ as *started* and proceeds to execute its workflow. At the end of the execution, the working thread sets $WSA[i]$ as *done*. When a working thread of $Node_s$ does not find a *waiting* element in WSA_s, $Node_s$'s control thread sends a final message to $Node_t$, ending the WSA exchange.

Chapter 6

Experiments and Results

In our experiments, we used two versions of our framework (Sections 5.2 and 5.3). Framework version 1 refers to "without WS" experiments and Framework version 2 refers to "with WS" experiments.

Each experiment used a different multi-workflow application. An I/O intensive multi-workflow application was executed in the supercomputer SDumont with the Framework version 1 and two multi-workflow applications, a synthetic and a real one, were executed in AWS ParallelCluster with the Framework version 2. We had to use two different environments because the access to SDumont was terminated during our work, which led us to switch to the AWS environment to finish it.

In this chapter, we first present the experiments made with Framework version 1 in the supercomputer (Section 6.2). Then, we present the Framework version 2 results in the AWS Cloud with a synthetic (Section 6.3) and a real application (Section 6.4).

6.1 Test Environment

The development of our Framework version 1 and its tests were done in the SDumont supercomputer. Afterwards, our Framework version 2 was developed and tested in AWS EC2 ParallelCluster.

6.1.1 Santos Dumont supercomputer

SDumont [86] is a Brazilian supercomputer for scientific researches. It is composed of multiple types of processors and accelerators. In SDumont, jobs may be submitted to different queues using the Slurm job scheduler. The queue used in our experiments was comprised of nodes with Intel Xeon Cascade Lake Gold 6252 processors with 24 cores. Each pair of processors composes a resource node, and each resource node has 768GB of RAM. The network that connects all SDumont devices is the Infiniband EDR, with 100Gb/s.

We used 1, 2, 4, 8 and 16 nodes, and each one of them has one Intel processor. Our framework was compiled with gcc and run with a version of openMPI 4 developed for SDumont. The file system used was Lustre v2.12.

6.1.2 AWS EC2 ParallelCluster

The second environment of our experiments was the AWS cloud and we opted to create a cluster environment inside the cloud with the AWS ParallelCluster tool (Section 2.4.1). We used the m6idn.xlarge EC2 (Elastic Computing Cloud) instance, which has an Intel Xeon of third generation processor with 4 vCPUs, 16 GiB of RAM, 237 GB SSD and up to 30 Gbps of network bandwidth.

The AWS Parallel Cluster was configured with 1, 2, 4, 8 or 16 VM instances inside the AWS region us-east-2 (Ohio). Amazon FSx for Lustre (*https://aws.amazon.com/fsx/lustre/*) was the file system used for the input and output files whereas the temporary files were placed in the local storage of each instance, in a volume called scratch. Our framework was compiled with gcc/9.4.0, using the MPI version openmpi40-aws-4.1.5-1.

6.2 Results for Version 1 - Synthetic I/O

In this experiment, we executed in the SDumont supercomputer a multi-workflow application composed of two simple I/O intensive tasks that generates two files and computes the difference (diff) between these files. The job receives a list of files sorted by their sizes, that will compose the workflows. The first task creates a file of size s, where s is the number of GB specified as a parameter. The second task reads the file written by the previous task, and copies its content to another file. The diff is used to see if both files are actually equal. Figure 6.1 illustrates this synthetic I/O intensive workflow.

All the files were written on Lustre. The experiment had 100 workflows and the files' size were stratified into 10 groups, from 10GB to 1GB, decrementing 1GB in each group. Each node used up to 24 threads, which is the total number of cores per node. Jobs with 1, 2 and 4 nodes used exactly 24 threads, whereas the remaining jobs divided the number of workflows evenly, that is, in jobs with 8 nodes, there were 12 or 13 threads; in jobs with 16 nodes, there were 6 or 7 threads per node.

Figure 6.2 presents the average makespan and the minimum makespan of the jobs running the I/O intensive multi-workflow application. For each number of nodes, the experiment was repeated three times, and the standard deviation was between 0.13 (16

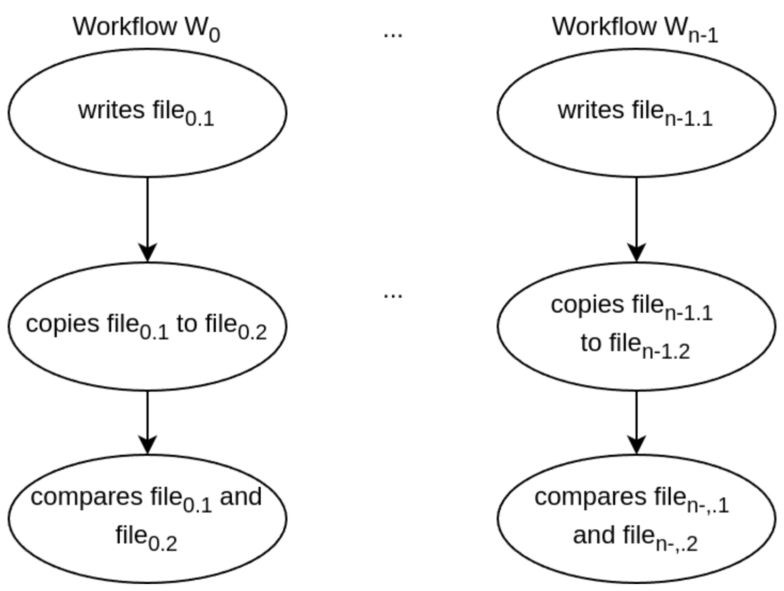

Workflow W_0
...
Workflow W_{n-1}
writes $file_{0.1}$
writes $file_{n-1.1}$
copies $file_{0.1}$ to $file_{0.2}$
...
copies $file_{n-1.1}$ to $file_{n-1.2}$
compares $file_{0.1}$ and $file_{0.2}$
compares $file_{n-,.1}$ and $file_{n-,.2}$

Figure 6.1: Synthetic I/O intensive workflow of version 1

nodes) and 3.48 (8 nodes). The best makespan was achieved with 4 nodes (minimum of 18.35 minutes), whereas the worst makespan happened with 16 nodes (minimum of 36.85 minutes).

In order to further understand the behaviour of Framework version 1, we used the metric GB/s, which was calculated by taking the sum of the GB assigned to each node and dividing it by the total execution time (in seconds) that each node took to complete all its workflows, as shown in Figures 6.3 (2 nodes) to 6.6 (16 nodes).

In Figure 6.3 (2 nodes), we can see that the nodes' performances are balanced. On the other hand, in the Figure 6.4 (4 nodes), the fastest node had over twice the performance of the slowest one, indicating an imbalance. Such performance difference is greater in the job with 8 nodes: the fastest node's performance was over 11 times the slowest one (Figure 6.5). We observed a similar behaviour with 16 nodes, where the fastest node's performance was over 16 times the slowest one (Figure 6.6).

According to the Figures 6.4 to 6.6, there are notable performance discrepancies among identical nodes, even when workloads are fairly distributed with the striped function. External factors affecting the final makespan were not anticipated before the workflows' execution. This highlights the need for a dynamic scheduling approach, such as work stealing.

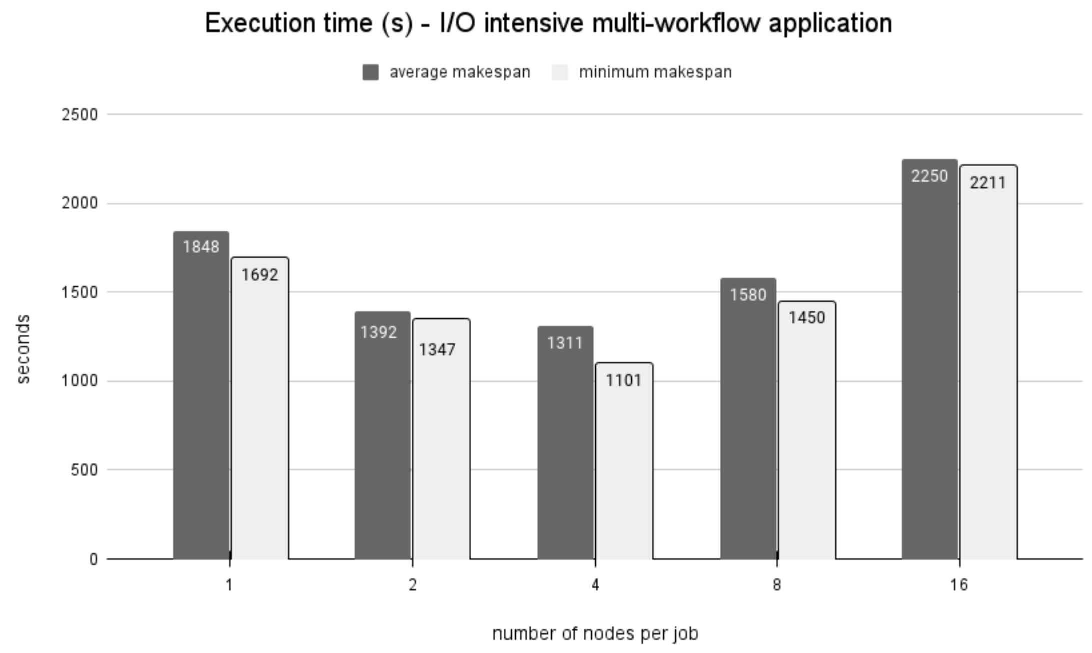


Figure 6.2: Execution time of 100 I/O intensive workflows. Framework Version 1

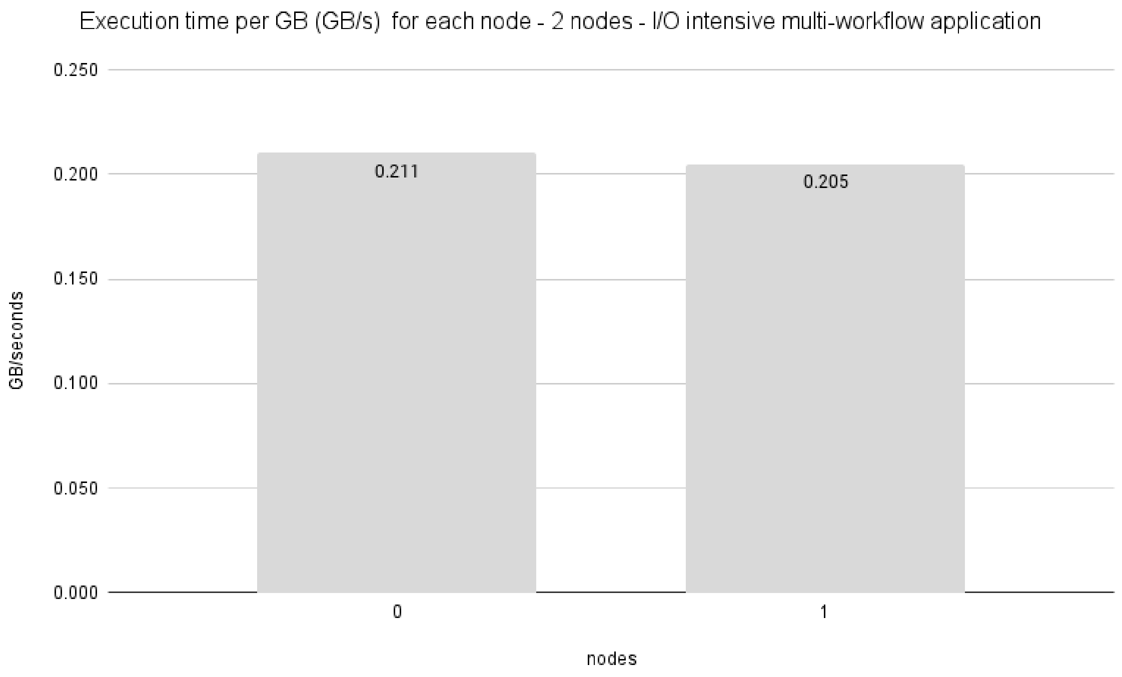


Figure 6.3: Execution time per GB of each node in job with 2 nodes

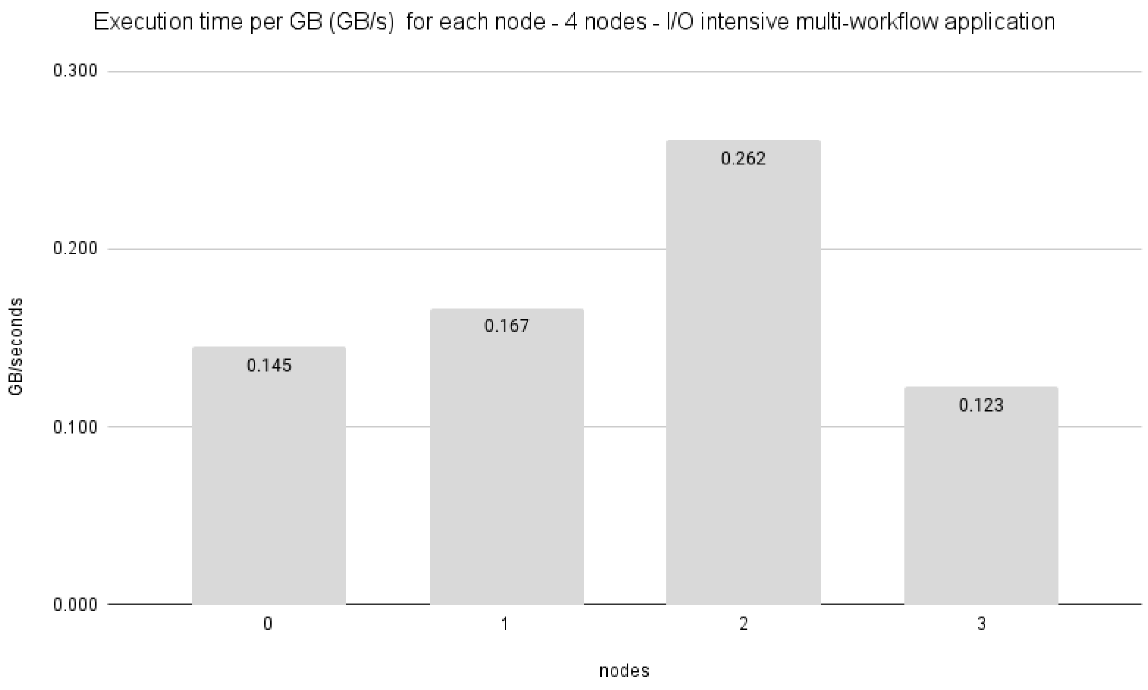


Figure 6.4: Execution time per GB of each node in job with 4 nodes

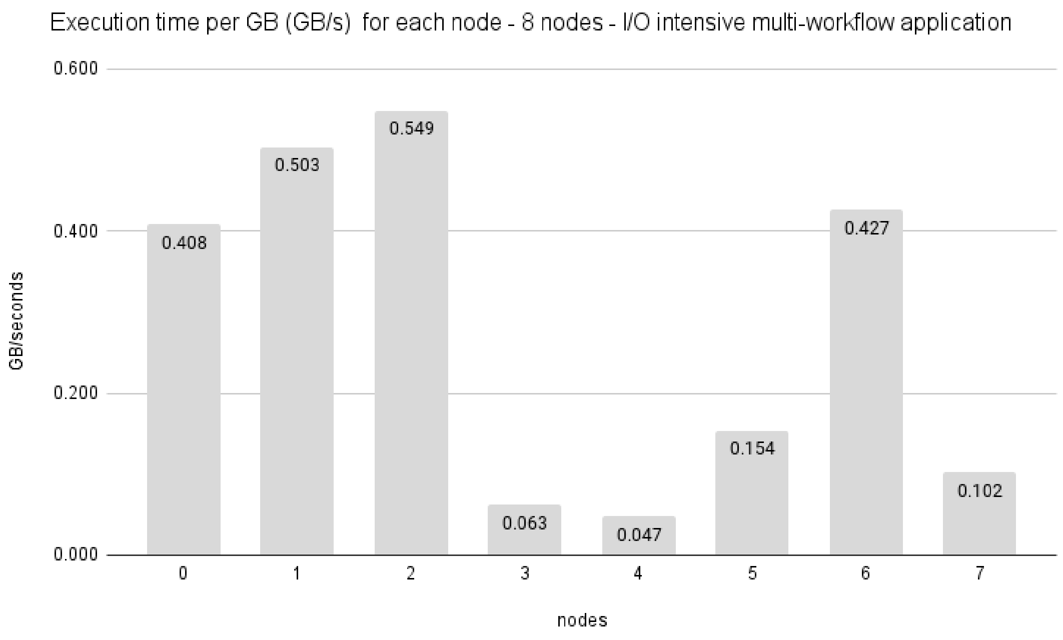


Figure 6.5: Execution time per GB of each node in job with 8 nodes

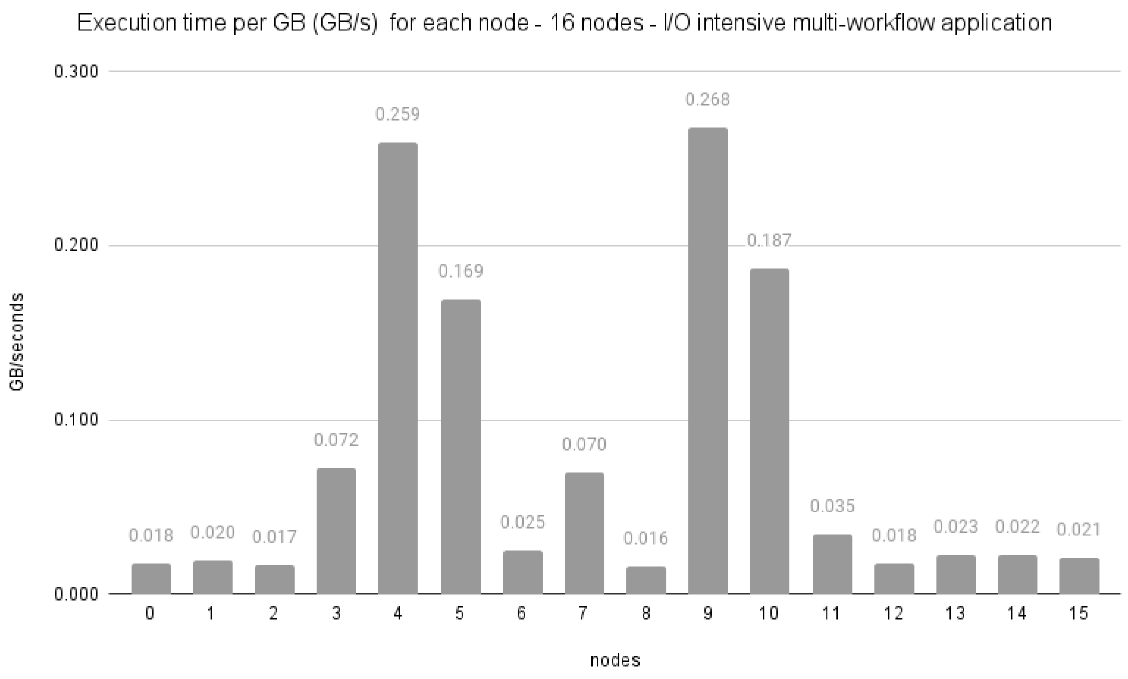


Figure 6.6: Execution time per GB of each node in job with 16 nodes

6.3 Framework Version 2 - Synthetic I/O fio application

To evaluate the performance of the Framework version 2 in the AWS ParallelCluster with an I/O bound scenario, the fio IO benchmark tool (*https://linux.die.net/man/1/fio*) was employed. The following metrics were collected: average bandwidth and execution time. The I/O bound multi-workflow fio application had 74 workflows, where files were written to and read from AWS Lustre and AWS local store volumes (scratch), using fio calls, as shown in Figure 6.7.

The local store volumes (scratch) were 237GB SSD volumes with individual usage to each node, whereas Lustre is the parallel file system. The sizes of the input files ranged from 1M to 946MB, following a random stratified approach, distributed as follows: [946MB,400MB]: 11 files (14.8%); [399MB,100MB]:20 files (27.1%); [99MB,1MB]:43 files (58.1%). We considered 2 scenarios: 1 process per task (fio's parameter numjob set to 1) and 4 processes per task (fio's parameter numjob set to 4), with 1, 2, 4, 8 and 16 nodes.

The execution times for each workflow application are presented in Figures 6.8 and 6.9. The best speedup compared to sequential time for numjob=1 was 3.56x, using 8 nodes and with WS. As for numjob=4, the best speedup was 3.44x, using 4 nodes and with WS.

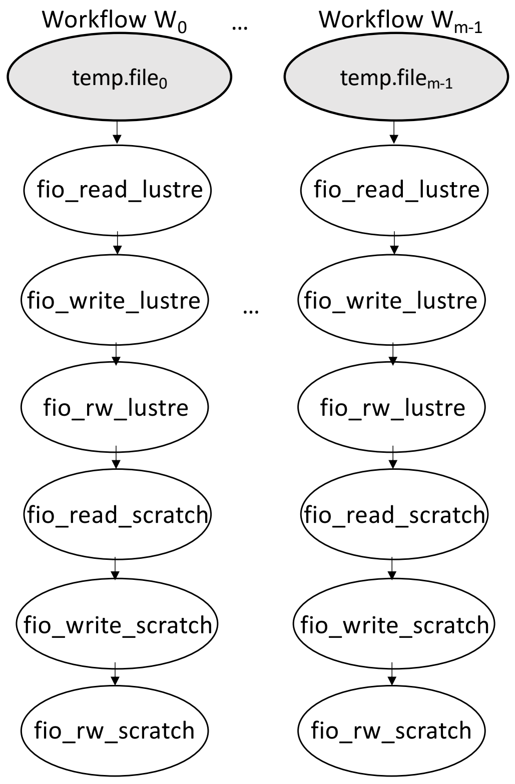


Figure 6.7: Synthetic I/O intensive workflows

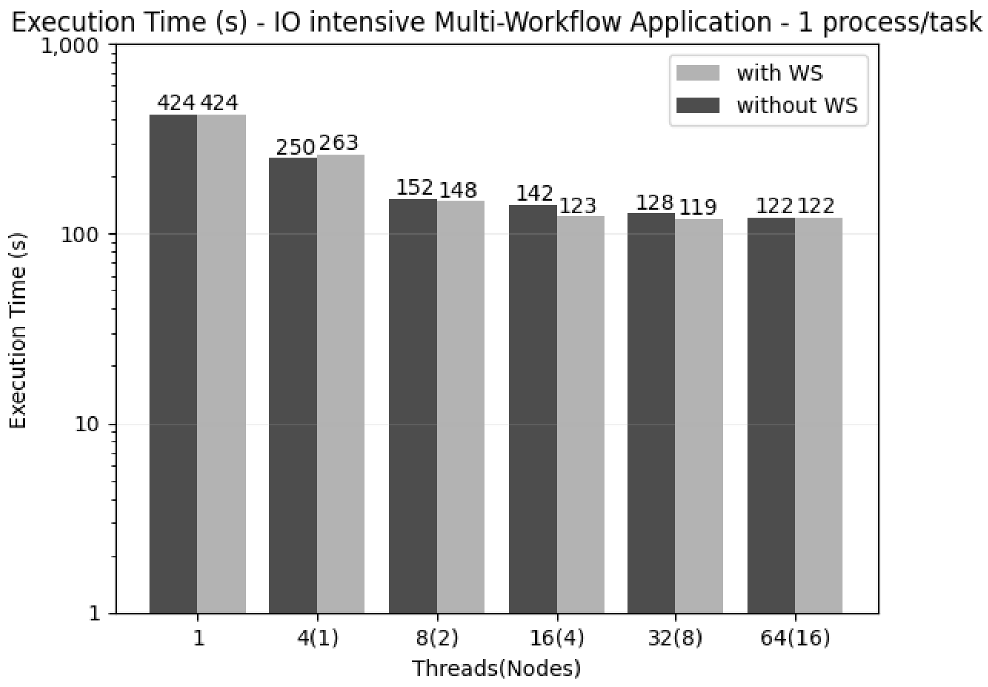


Figure 6.8: Execution time (1 processes/task)

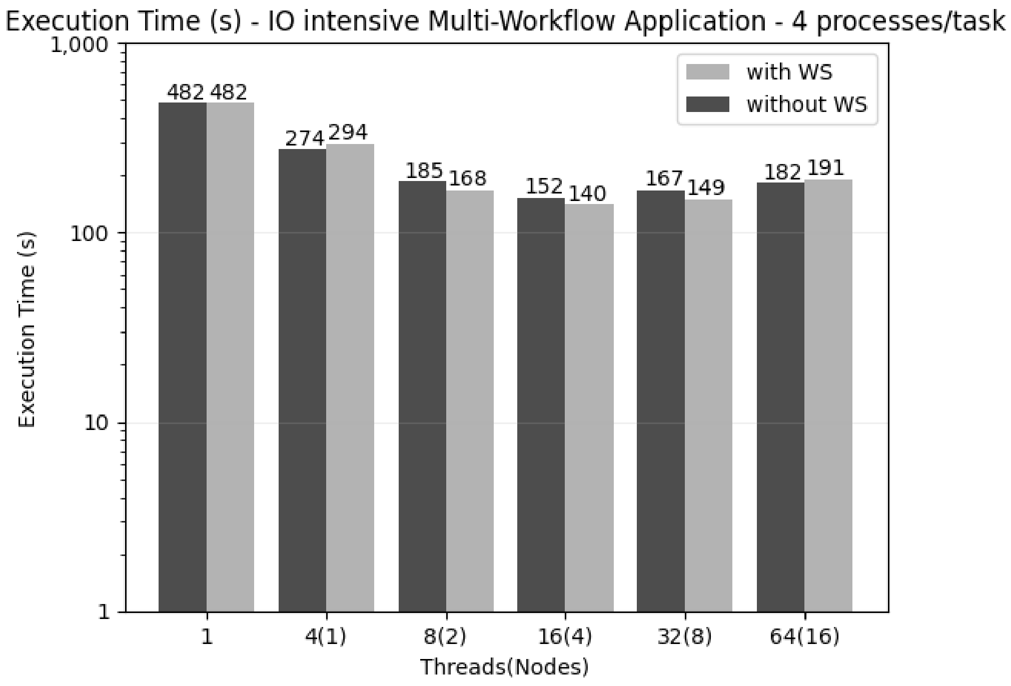


Figure 6.9: Execution time (4 processes/task)

When observing the bandwidth results generated by fio (Figures 6.10 and 6.11) on the framework version 2, it is noted that Lustre does not exhibit an overall clear pattern of behavior regarding bandwidth, specially when the average bandwidth is considered. However, we can see that the bandwidth is smaller when 4, 8 and 16 nodes are used during the reading. The bandwidth showed unpredictable behavior, with a minimum average of up to 12.5MB/s in the case of reading with 1 process per task in the framework with 8 nodes, up to a maximum average of 2924.3MB/s in the case of 4 simultaneous processes per task also in 8 nodes.

On the other hand, the bandwidth observed in the scratch volume had a clearer behaviour pattern as seen in Figure 6.11. It was noted that the average bandwidths were higher with 4 or more nodes than with 1 or 2 nodes per framework across all cases of reading and writing, with both 1 or 4 processes per task. Furthermore, the average bandwidth in scratch was also higher, ranging from 1031.2MB/s when reading with one process per node and using 2 nodes, to 4072.2MB/s in writing with 4 processes per node and using 16 nodes. So, based on these results, we concluded that it is better to use the local scratch file system, whenever possible.

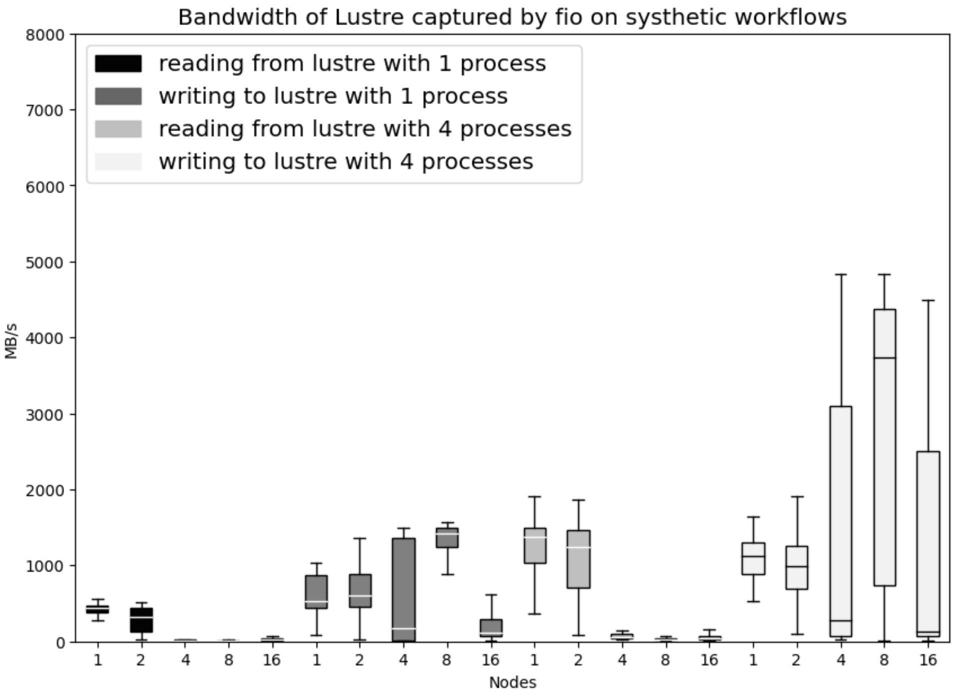


Figure 6.10: Lustre bandwidth for the fio application

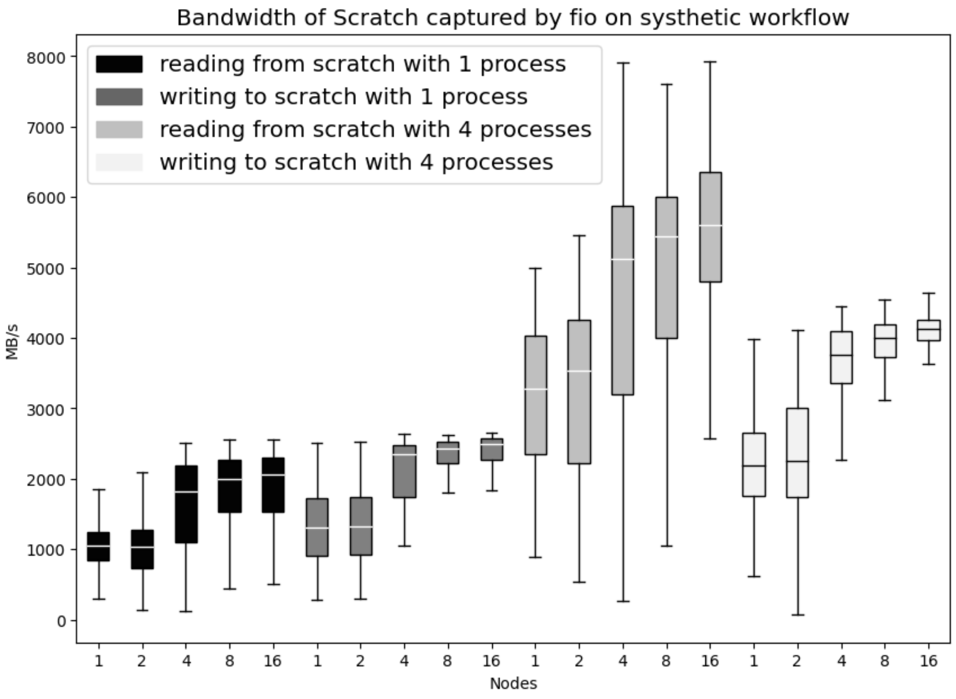


Figure 6.11: Scratch bandwidth for the fio application

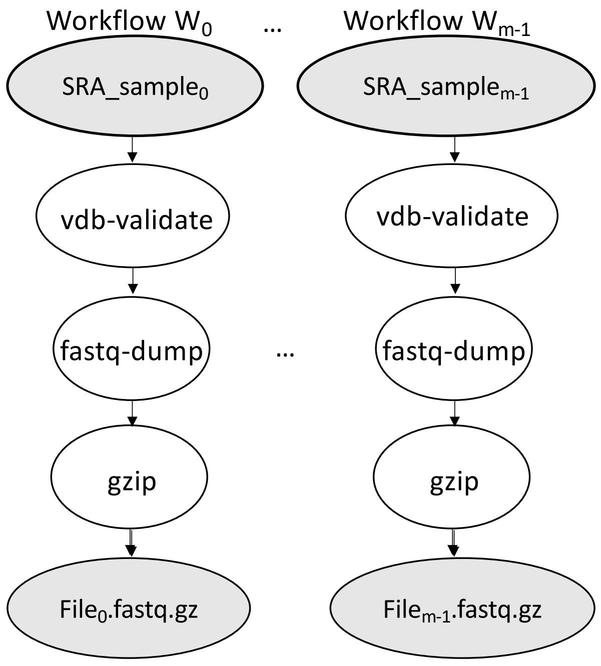


Figure 6.12: Real multi-workflow Bioinformatics application executed in our framework

6.4 Framework version 2 - Real Bioinformatics application

6.4.1 Description of the Application

In the real application experiment, we executed the data-preprocessing phase of the Cellheap bioinformatics workflow [11]. Our multi-workflow application (Figure 6.12) receives as input RNA-seq Sequence Read Archive (SRA) samples and executes three programs.

First, *vdb-validate* is executed to validate the integrity of the SRA input sample. Then, *fastq-dump* is executed to convert SRA files into fastq files. These two tools are available at *https://github.com/ncbi/sra-tools/*. Finally *gzip* is executed to compress the files. For each workflow, the output file is a fastq.gz file which contains the sequence reads that correspond to the sample and will be used in further analyses.

The samples used in the experiment are a part of real dataset project PRJNA743046, available at National Center for Biotechnology Information (NCBI) [87], which has 30587 samples from covid-19 genome sequencing and it is part of the Covid-19 Outbreak umbrella project (PRJNA615625), which compiles various subprojects of Covid-19 sequencing. We decided to use 400 of those samples, generating 400 workflows, i. e., m in Figure 6.12 is equal to 400. The sample sizes ranged from 10MB to 200MB, and the samples were selected with the stratified random sampling technique, considering 3 strata, depending

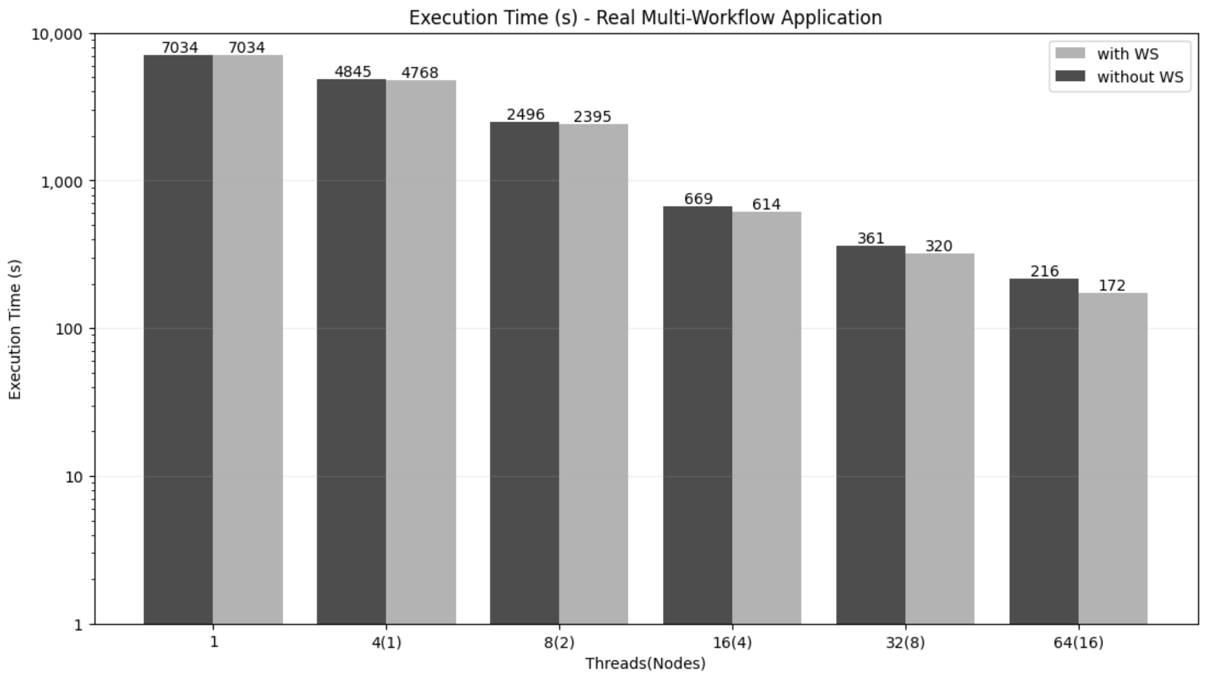


Figure 6.13: Execution time of real multi-workflow application (400 workflows)

on their size: (a) 10MB to 100MB; (b) 100MB to 110MB; and (c) 190MB to 200MB. In strata (a), (b) and (c), 300, 95 and 5 samples were selected.The configuration of the three strata was done based on the characteristics of the dataset. The overall size of the input data for the whole application was 19.7GB, considering the 400 samples. We downloaded the samples from the public repository found in National Center for Biotechnology Information (NCBI) and placed them in AWS S3 buckets. Then, the first task (*vdb-validate*) accesses S3 and writes its output temporary file to the local file system and so on. The final output ($file_x.fastq.gz$ in Figure 6.12) is written to AWS Lustre, a total of 20.9GB for output files. Temporary files of roughly 180GB were produced during the execution of the real multi-workflow application.

6.4.2 Execution times

The execution times were collected in the AWS ParallelCluster (Section 6.1.2) with 1, 2, 4, 8 and 16 VMs. The results are shown in Figure 6.13 and 6.14, where label `1` corresponds to the sequential solution and labels `4(1)`, `8(2)`, etc, correspond to the parallel execution in the format `Threads(Nodes)`.

In Figure 6.13, we can see that our framework was able to attain a substantial reduction in the execution time. For instance, the execution time was reduced from 1 hour and 57 min (sequential) to 2 min and 52 sec (16 nodes with WS). If we consider 8 nodes with WS, the execution time was 5 min and 20 sec. Similar to the I/O intensive synthetic

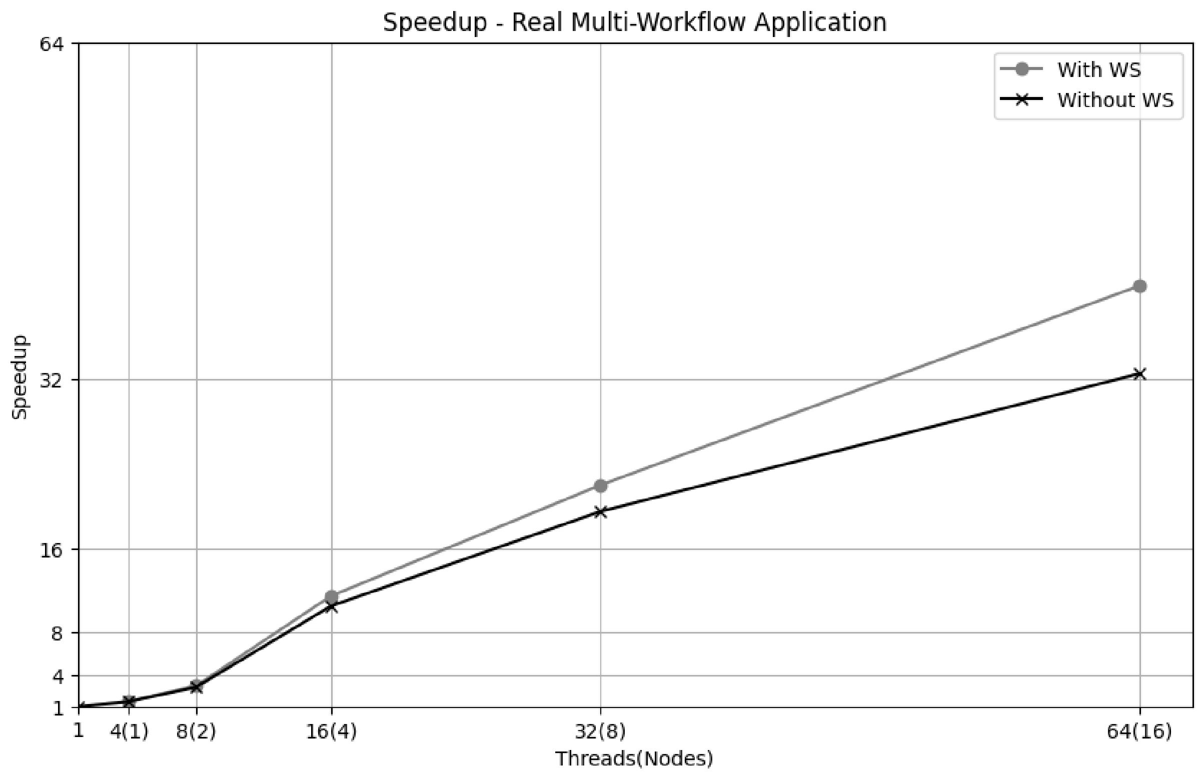


Figure 6.14: Speedup time of real multi-workflow application (400 workflows)

application, we observed a great reduction in the execution time when 2 and 4 nodes are considered. In this case, the reduction was from 39 min and 55 sec to 10 min and 14 sec.

We can see in Figure 6.14 that our framework provides a solution with very good scalability. Also, adding work stealing improves the speedup, becoming better as long as more nodes are considered, if we compare to the no work stealing mode.

Figure 6.15 presents the work stealing gain, showing how much the framework with WS reduces the execution time when compared to the framework without WS for each number of threads in the real Bioinformatics multi-workflow application with 400 workflows. The gain is calculated as $(et_{noWS} - et_{WS})/et_{noWS}$, where et_{noWS} is the execution time of the job without work stealing and et_{WS} is the execution time with work stealing for each given number of threads. As it is shown, when the number of threads (and nodes) increases, so does the gain. The greatest work stealing gain was a little over 20%, in the job with 16 nodes and 64 threads.

6.4.3 Idle thread Analysis

As stated in Section 6.2, the execution time of each node can vary, even when the striped function is applied to achieve load balance. As an attempt to improve the load balance, the version with work stealing (version 2) was developed and executed. Besides the

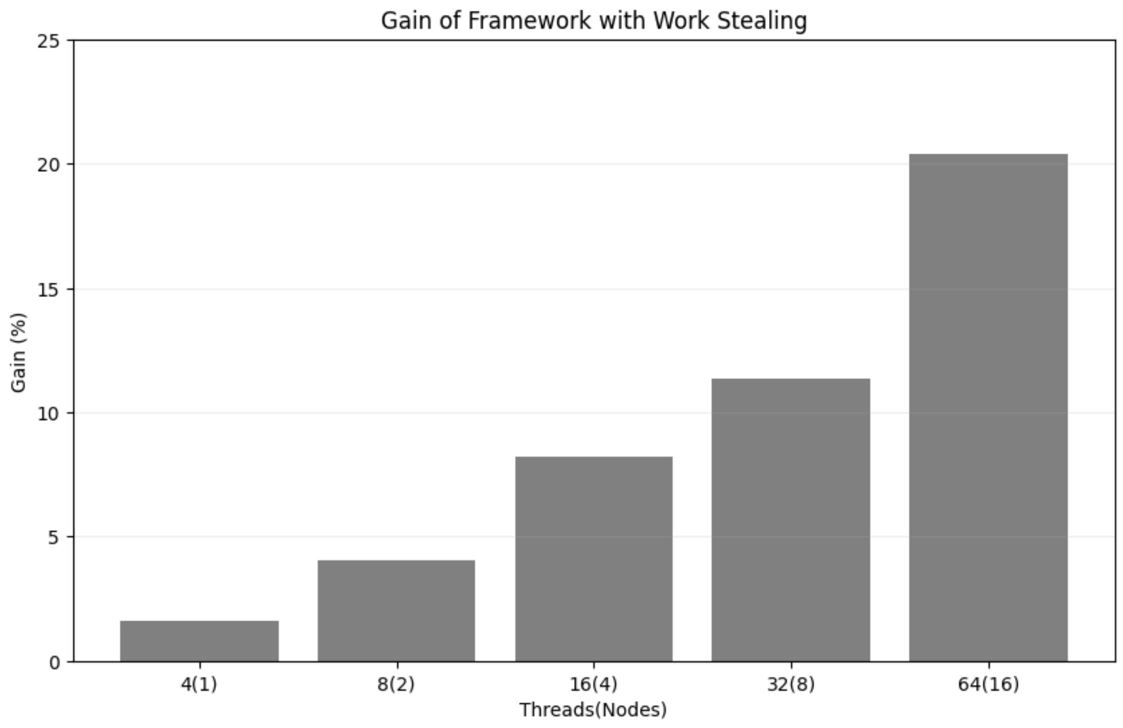


Figure 6.15: Gain of framework with work stealing compared with the framework without work stealing in real Bioinformatics application

	first idle moment std. dev.	**nodes' execution time std. dev.**
2 nodes	70.71	2.83
4 nodes	10.03	6.38
8 nodes	17.96	6.23
16 nodes	13.57	7.33

Table 6.1: Standard Deviation of each node's first idle time and execution time

clear improvement reached with the jobs' execution time, we also analyzed the individual performance of each node.

In order to identify the influence of the work stealing, we measured, for each job of the real Bioinformatics multi-workflow application, the standard deviation of the moment that the first thread became idle in each node and the standard deviation of the total execution time of each node. In Table 6.1, we present these results. In the second column, there is the standard deviation of the moment that the first thread of each job's node became idle. In the third column, it is shown the standard deviation of the execution time of each node. When we compared both metrics for each number of nodes, we see that the standard deviation of the nodes' execution time is smaller than the one of the first thread to become idle moment. That is an indication that the work stealing actually increased the work balance.

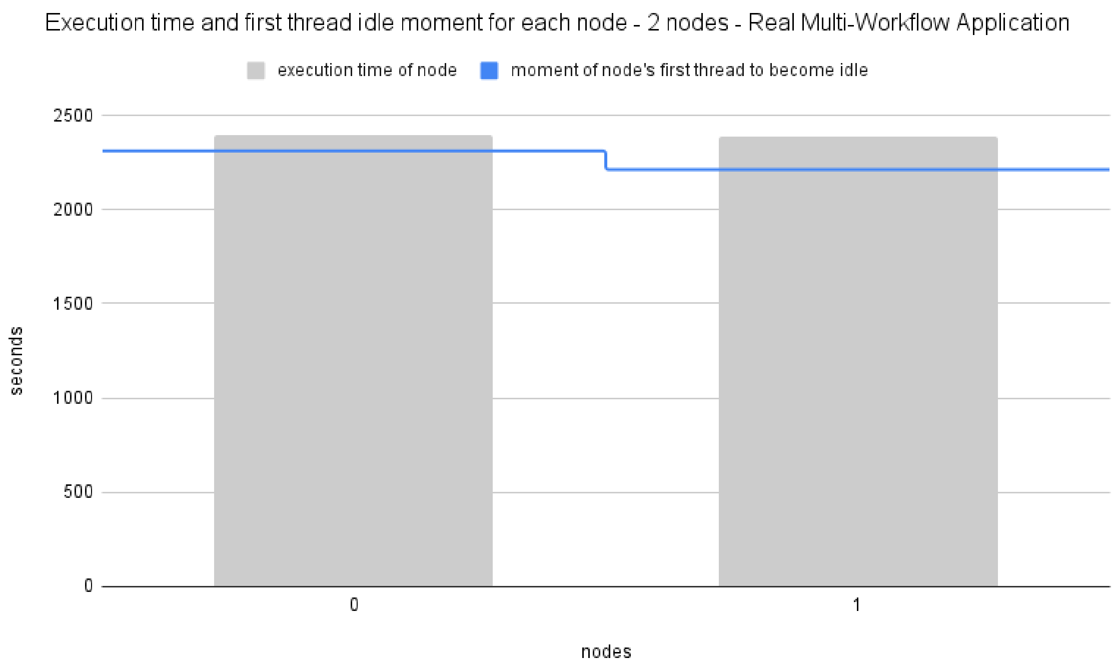


Figure 6.16: Execution time of each node with the moment that the first thread of each node becomes idle, version 2 with 2 nodes, real multi-workflow application

Considering the real Bioinformatics multi-workflow application experiment, we compared the total execution time of each individual node on the real workflow case with the moment that the first thread of each node became idle. Figures 6.16 to 6.19 show the execution time that each node of each job (2, 4, 8 and 16 nodes) took. The lines over the execution time bars indicate when was the moment that a thread became idle and started to steal work within the node or from another node.

The Figure 6.16 presents the execution time of each node of the the job with two nodes. Both nodes concluded their workload almost at same time, but $Node_1$ started to steal slightly earlier than $Node_0$.

In Figure 6.17 there is the execution time of the job with 4 nodes. In this case, the execution time of each node is close to each other, and $Node_3$ begun to steal earlier than the others, although there is not a visible difference among the other nodes' idle moment.

In Figure 6.18, we can see a greater difference between the moment that each of the eight nodes first had an idle thread, however each node execution time remained almost the same. With eight nodes, $Node_7$ was the first to have an idle thread. The difference of the first thread idle moment is the greatest with 16 nodes (Figure 6.19). In this case, the first node to have an idle thread was node 14.

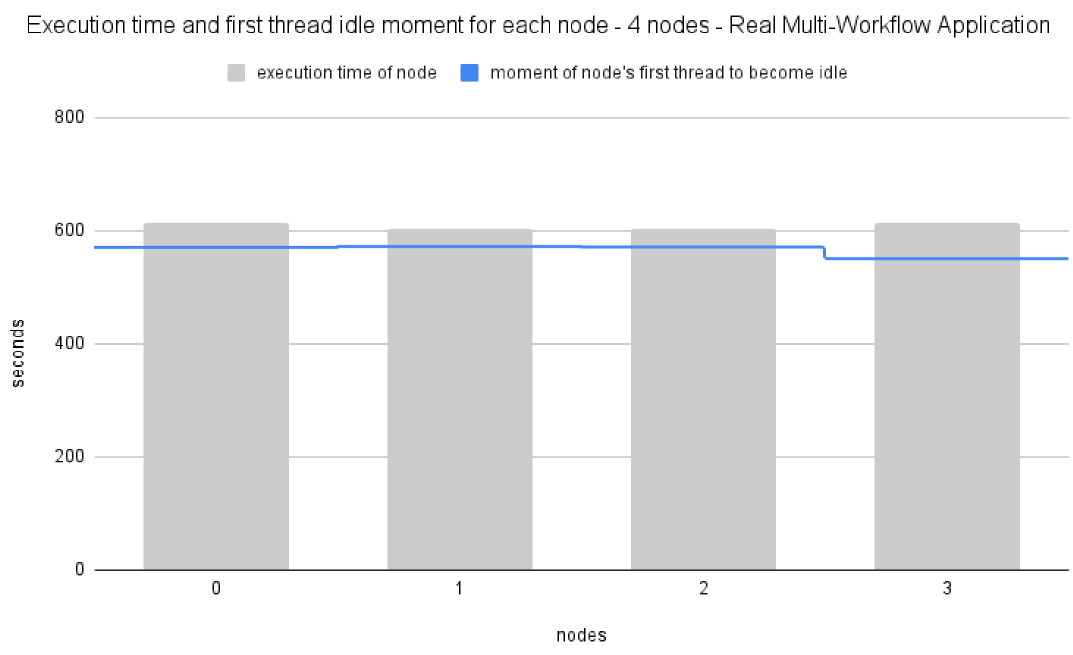


Figure 6.17: Execution time of each node with the moment that the first thread of each node becomes idle, version 2 with 4 nodes, real multi-workflow application

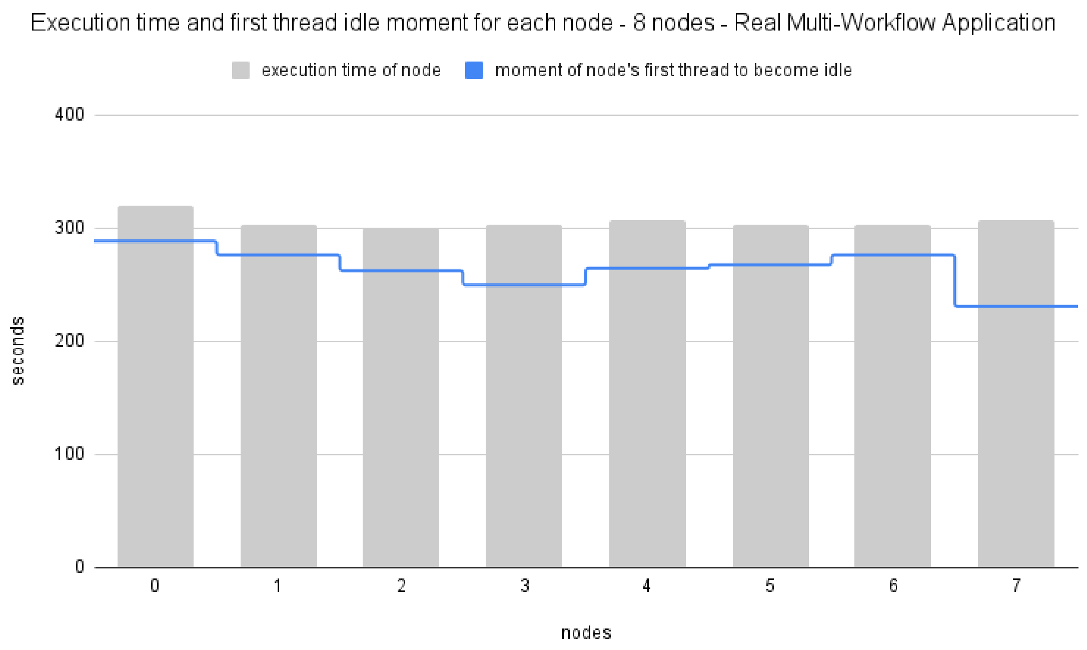


Figure 6.18: Execution time of each node with the moment that the first thread of each node becomes idle, version 2 with 8 nodes, real multi-workflow application

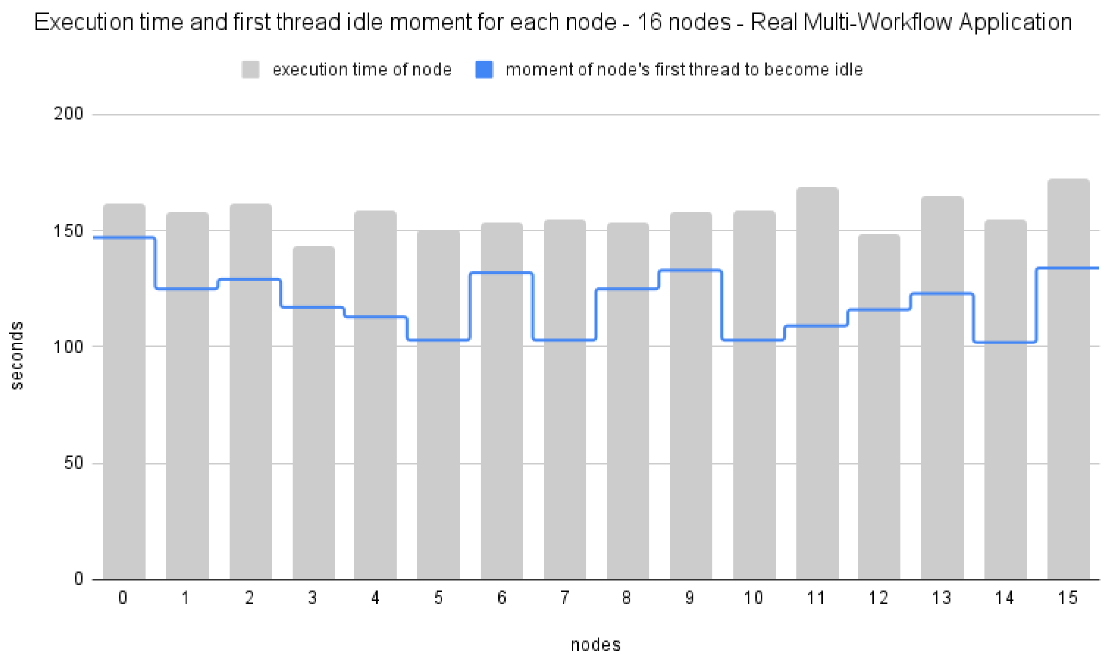


Figure 6.19: Execution time of each node with the moment that the first thread of each node becomes idle, version 2 with 16 nodes, real multi-workflow application

Chapter 7

Conclusion and Future Works

In this chapter, we present the conclusion of our work, comparing the initial objectives with the obtained results. Then, we discuss some future works.

7.1 Conclusion

Our main goal was to investigate workflow scheduling strategies to efficiently execute linear multi-workflow applications in a distributed architecture. In order to achieve this goal, we firstly focused on developing a framework to be executed in a distributed architecture, then we opted for developing a work stealing version of our framework. Besides, we maintained our framework configuration simple to the end-user.

First of all, we proposed and evaluated two versions of an MPI/OpenMP framework for execute a linear multi-workflow application in large scale distributed architectures, focusing cloud computing and HPC. The linear multi-workflow application is a usual class of Bioinformatics workflows, and we assumed that (a) there is considerable I/O activity, (b) there is production of many temporary files, (c) very few information about the workflows' tasks is known before execution, (d) the size of the input may have impact on the overall execution time. We developed our framework versions using and MPI/OpenMP approach with multiple processes along multiple computer nodes and multiple threads within the nodes. This allowed the scalability of our framework versions and the distribution of the multi-workflows through the nodes and threads.

Another objective was to reduce the multi-workflow application makespan, so that the application could be executed efficiently. Apart from the possibility to run the workflows in a parallel manner, which already diminishes makespan, we added a work stealing scheduling policy to our framework version 2 in order to avoid idle resources and to increase load balancing, thus further reducing the total makespan. The work stealing scheduler does not need previous information about the tasks, which makes it computationally inexpensive and independent from task data.

Our work stealing scheduler has two phases: inter-node and intra-node. Firstly, the idle thread steals a linear workflow from another thread within its node. Secondly, if there is no more workflow in that node, the idle thread can steal from another node. Stealing from another thread within the same node reduces the need for message passing between nodes, which is advantageous since message passing involves costly I/O operations.

Finally, our last objective was to facilitate the configuration of the framework by the end-user. Both framework versions require two files: one with the workflow skeleton, containing a linear list of tasks, and one for the input list, ordered by descending size. Both files can be created using a simple text editor and do not require prior programming knowledge. To configure the framework, users only need knowledge of the tasks and their inputs, which is suitable for biologists and other researchers who are our target audience.

In the experimental results, firstly we tested our framework version 1 (without work stealing) in the supercomputer SDumont, using up to 16 nodes and an I/O intensive synthetic multi-workflow application (100 workflows). Our results showed that the framework job running on 4 nodes was the fastest, achieving a 1.71x speedup compared to the slowest job running on 16 nodes, and a 1.41x speedup compared to the job running on 1 node on average.

When we analysed the individual performance of each node in the SDumont experiment, we realized that there was a great difference among them, specially when more nodes were added. This performance difference could be relieved with a dynamic scheduling strategy and that is why we included work stealing.

After developing the framework version 2 (with work stealing), we set up a cluster of up to 16 VMs instances inside the cloud AWS, using the ParallelCluster tool, with different file systems: one for input/output files and other for temporary files. Our results with an I/O intensive benchmark multi-workflows application and a real Bioinformatics multi-workflow application show that our framework is able to reduce considerably the makespan, taking advantage of parallelism and work stealing.

The results for the I/O intensive benchmark showed that the fastest job had 16 threads (4 nodes) and work stealing, achieving 3.44x speedup when compared to the sequential job. The best result for the real Bioinformatics application was met in the job with 64 threads (16 nodes) and with work stealing: it took less than 3 minutes, while the sequential execution of the multi-workflow application too almost 2 hours, with a 40.90x speedup compared to the sequential processing job. It is also notable that the influence of the work stealing scheduler increases with the number of nodes used per job. The time savings of framework version 2 were over 20% with 16 nodes per job, whereas with only

2 nodes per job, the time savings were 4%.

When we analyzed the moment when the first thread of each node became idle and compared it with the execution time per node, we observed the load balance brought by work stealing. In the real Bioinformatics multi-workflow application, we analyzed both metrics and their standard deviation. The standard deviation of individual node's execution times for each number of nodes per job was significantly lower than the standard deviation for the first idle thread moment. This indicates the impact of work stealing on load balance and total execution time.

In conclusion, the work stealing approach allowed the workflows distribution to achieve a lower makespan. Although the work stealing makes a significantly makespan reduction, the parallel workflows processing has a greater influence on the reduction, when compared to the sequential execution. The amount of I/O operations also has an impact on the makespan. In both environments, SDumont and AWS ParallelCluster, during the I/O intensive multi-workflow application, the lowest makespan was achieved with 4 nodes.

7.2 Future works

As future work, we suggest:

- to investigate a different policy for choosing the node to be stolen. Since each thread currently has only one node option to steal from, it may not always steal from the node with the greatest need. A different stealing choice strategy can further benefit load balancing. However, any change must maintain the message passing as low as possible;
- to investigate tuning configuration, such as the most appropriate number of threads per node according to the input and to the performance and the optimal frequency to exchange the Workflow Status Array (WSA) between stealer and stolen nodes;
- to design a scheduler that considers both the makespan and the cost, since the cost of a cloud service can be very high when too much time and/or too many VMs are used;
- to adapt Framework version 2 to transient instances cloud service (e.g. AWS spot), which may reduce the cost. This allows to allocate a new spot instance during the execution and to re-instantiate a suspended task on it. However, it needs to be continuously monitored and adjusted accordingly, thus it is necessary to develop a fault tolerance policy to use in case of spot instance revocation;

- to add the framework to a Bioinformatics portal. This would be even more friendly to the end-users as it would free them to manage an IaaS model cloud, replacing it for a SaaS model. This approach could also reach a greater audience;
- to add our Framework version 2 to a scientific workflow scheduler system as an extension, so that other users may see our Framework as an available option for executing applications composed of multiple linear workflows;
- to investigate the use of containers and adapt our work for execution on them.

References

[1] Casavant, Thomas L. and Jon G. Kuhl: *A taxonomy of scheduling in general-purpose distributed computing systems.* IEEE Transactions on software engineering, 14(2):141–154, 1988. ix, 2, 8, 9, 30, 31, 32, 33, 34, 35, 36

[2] Rodriguez, Maria Alejandra and Rajkumar Buyya: *A taxonomy and survey on scheduling algorithms for scientific workflows in iaas cloud computing environments.* Concurrency and Computation: Practice and Experience, 29(8):e4041, 2017. ix, 7, 8, 9, 30, 31, 32, 33, 34, 35, 36

[3] Sterling, Thomas, Maciej Brodowicz, and Matthew Anderson: *High performance computing: modern systems and practices.* Morgan Kaufmann, 2018. ix, 13, 14, 15

[4] *Lustre.* `https://www.lustre.org/`, Accessed on Mars 28, 2024. ix, 21, 28

[5] Leite, Alessandro Ferreira: *A user-centered and autonomic multi-cloud architecture for high performance computing applications.* 2015. ix, 22, 23

[6] Mell, Peter, Tim Grance, *et al.*: *The nist definition of cloud computing.* 2011. ix, 22, 23

[7] *What is azure batch.* `https://learn.microsoft.com/pt-br/azure/batch/batch-technical-overview#additional-batch-capabilities`, Accessed on June 20, 2024. ix, 25, 26

[8] *Cloud hpc toolkit.* `https://cloud.google.com/hpc-toolkit/docs/quickstarts/slurm-cluster`, Accessed on June 20, 2024. ix, 27

[9] Ford, Alex: *Building an interactive and scalable ml research environment using aws parallelcluster.* `https://aws.amazon.com/pt/blogs/machine-learning/building-an-interactive-and-scalable-ml-research-environment-using\-aws-parallelcluster/`, visited on 2019-11, Accessed on Mars 28, 2024. ix, 28

[10] *Aws parallelcluster.* `https://aws.amazon.com/pt/hpc/parallelcluster/`, visited on 2024-25-03. ix, 27, 28, 29

[11] Silva, Vanessa S, Maiana OC Costa, Maria Clicia S Castro, Helena S Silva, Maria Emilia MT Walter, Alba CMA Melo, Kary AC Ocaña, Marcelo T dos Santos, Marisa F Nicolas, Anna Cristina C Carvalho, *et al.*: *Cellheap: A workflow for optimizing covid-19 single-cell rna-seq data processing in the santos dumont supercomputer.* In

Advances in Bioinformatics and Computational Biology: 14th Brazilian Symposium on Bioinformatics, BSB 2021, Virtual Event, November 22–26, 2021, Proceedings 14, pages 41–52. Springer, 2021. 1, 55

[12] Chen, Yang, En Min Li, and Li Yan Xu: *Guide to metabolomics analysis: a bioinformatics workflow.* Metabolites, 12(4):357, 2022. 1

[13] Cadzow, Murray, James Boocock, Hoang T Nguyen, Phillip Wilcox, Tony R Merriman, and Michael A Black: *A bioinformatics workflow for detecting signatures of selection in genomic data.* Frontiers in Genetics, 5:293, 2014. 1

[14] Ko, GunHwan, Pan Gyu Kim, Jongcheol Yoon, Gukhee Han, Seong Jin Park, Wangho Song, and Byungwook Lee: *Closha: bioinformatics workflow system for the analysis of massive sequencing data.* BMC bioinformatics, 19:97–104, 2018. 1

[15] Banimfreg, Bayan H: *A comprehensive review and conceptual framework for cloud computing adoption in bioinformatics.* Healthcare Analytics, page 100190, 2023. 2

[16] Ullman, Jeffrey D.: *Np-complete scheduling problems.* Journal of Computer and System sciences, 10(3):384–393, 1975. 2, 8

[17] Agrawal, Kunal, Anne Benoit, Loic Magnan, and Yves Robert: *Scheduling algorithms for linear workflow optimization.* In *2010 IEEE International symposium on parallel & distributed processing (IPDPS)*, pages 1–12. IEEE, 2010. 2, 8

[18] Teylo, Luan, Alan L Nunes, Alba CMA Melo, Cristina Boeres, Lúcia Maria de A Drummond, and Natalia F Martins: *Comparing sars-cov-2 sequences using a commercial cloud with a spot instance based dynamic scheduler.* In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 247–256. IEEE, 2021. 2

[19] Konjaang, J Kok and Lina Xu: *Cost optimised heuristic algorithm (coha) for scientific workflow scheduling in iaas cloud environment.* In *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing,(HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, pages 162–168. IEEE, 2020. 2

[20] Khaleel, Mustafa Ibrahim: *Multi-objective optimization for scientific workflow scheduling based on performance-to-power ratio in fog–cloud environments.* Simulation Modelling Practice and Theory, 119:102589, 2022. 2

[21] Gao, Yongqiang, Shuyun Zhang, and Jiantao Zhou: *A hybrid algorithm for multi-objective scientific workflow scheduling in iaas cloud.* IEEE Access, 7:125783–125795, 2019. 2

[22] Iranmanesh, Amir and Hamid Reza Naji: *Dchg-ts: a deadline-constrained and cost-effective hybrid genetic algorithm for scientific workflow scheduling in cloud computing.* Cluster Computing, 24:667–681, 2021. 2

[23] Durillo, Juan J and Radu Prodan: *Multi-objective workflow scheduling in amazon ec2.* Cluster computing, 17:169–189, 2014. 2

[24] Rodriguez, Maria A and Rajkumar Buyya: *Budget-driven scheduling of scientific workflows in iaas clouds with fine-grained billing periods.* ACM Transactions on Autonomous and Adaptive Systems (TAAS), 12(2):1–22, 2017. 2

[25] Stavrinides, Georgios L and Helen D Karatza: *Multicriteria scheduling of linear workflows with dynamically varying structure on distributed platforms.* Simulation Modelling Practice and Theory, 112:102369, 2021. 2, 3, 33, 36

[26] Xia, Yuanqing, Yufeng Zhan, Li Dai, and Yuehong Chen: *A cost and makespan aware scheduling algorithm for dynamic multi-workflow in cloud environment.* The Journal of Supercomputing, 79(2):1814–1833, 2023. 3, 35, 36

[27] Badia Sala, Rosa Maria, Eduard Ayguadé Parra, and Jesús José Labarta Mancho: *Workflows for science: A challenge when facing the convergence of hpc and big data.* Supercomputing frontiers and innovations, 4(1):27–47, 2017. 3

[28] Jalili, Vahid, Enis Afgan, Qiang Gu, Dave Clements, Daniel Blankenberg, Jeremy Goecks, James Taylor, and Anton Nekrutenko: *The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2020 update.* Nucleic acids research, 48(W1):W395–W402, 2020. 3

[29] Deelman, Ewa, Rafael Ferreira da Silva, Karan Vahi, Mats Rynge, Rajiv Mayani, Ryan Tanaka, Wendy Whitcup, and Miron Livny: *The pegasus workflow management system: translational computer science in practice.* Journal of Computational Science, 52:101200, 2021. 3

[30] Silva, Helena SIL, Maria CS Castro, Fabricio AB Silva, and Alba CMA Melo: *A framework for automated parallel execution of scientific multi-workflow applications in the cloud with work stealing.* In *European Conference on Parallel Processing*, pages 298–311. Springer, 2024. 4

[31] Benoit, Anne, Ümit V Çatalyürek, Yves Robert, and Erik Saule: *A survey of pipelined workflow scheduling: Models and algorithms.* ACM Computing Surveys (CSUR), 45(4):1–36, 2013. 6

[32] Singh, Lovejit and Sarbjeet Singh: *A survey of workflow scheduling algorithms and research issues.* International Journal of Computer Applications, 74(15), 2013. 8

[33] Blumofe, Robert D, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou: *Cilk: An efficient multithreaded runtime system.* ACM SigPlan Notices, 30(8):207–216, 1995. 10

[34] Blumofe, Robert D., Charles E. Leiserson, R. D. Blumofe, and C. E. Leiserson: *Scheduling multithreaded computations by work stealing.* Journal of the ACM, 46(5):720–748, 1999, ISSN 00045411. 10

[35] Michael, Maged M., Martin T. Vechev, and Vijay A. Saraswat: *Idempotent work stealing.* ACM SIGPLAN Notices, 44(4):45–53, 2009, ISSN 15232867. 10, 11

[36] Dinan, James, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha: *Scalable work stealing.* Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, 2009. 11

[37] Van Steen, Maarten and A Tanenbaum: *Distributed systems principles and paradigms.* Network, 2(28):1, 2002. 13

[38] Coulouris, George F, Jean Dollimore, and Tim Kindberg: *Distributed systems: concepts and design.* pearson education, 2005. 13

[39] Garfinkel, Simson: *Architects of the information society: 35 years of the Laboratory for Computer Science at MIT.* MIT press, 1999. 13

[40] Licklider, Joseph CR and Robert W Taylor: *The computer as a communication device.* Science and technology, 76(2):30–32, 1968. 13

[41] *A survey of remote procedure calls.* ACM SIGOPS Operating Systems Review, 24(3):68–79, 1990. 13

[42] Abramson, H Ñimrod D and R Giddy Sosic: *The grid: Blueprint for a new computing infrastructure.* IEEE, USA, 1995. 14

[43] Chellappa, Ramnath: *Intermediaries in cloud-computing: A new computing paradigm.* In *INFORMS Annual Meeting, Dallas*, pages 26–29, 1997. 14

[44] Foster, Ian, Yong Zhao, Ioan Raicu, and Shiyong Lu: *Cloud computing and grid computing 360-degree compared.* In *2008 grid computing environments workshop*, pages 1–10. Ieee, 2008. 14, 21, 22

[45] Borin, Edson, Lúcia Maria A Drummond, Jean Luc Gaudiot, Alba Melo, Maicon Melo, and Philippe OA Navaux: *Why move hpc applications to the cloud?* In *High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment*, pages 1–5. Springer, 2023. 14

[46] *The linpack benchmark.* `https://www.top500.org/project/linpack/`, Accessed on May 30, 2024. 14

[47] Anthony, Sebastian: *The history of supercomputers.* `https://www.extremetech.com/extreme/125271-the-history-of-supercomputers`, visited on 2012-04, Accessed on May 31, 2024. 15

[48] *Frontier - hpe cray.* `https://top500.org/system/180047/`, Accessed on May 30, 2024. 15

[49] *No. 1 since june 2022.* `https://www.top500.org/resources/top-systems/frontier-doescoak-ridge-national-laboratory/`, Accessed on June 9, 2024. 15

[50] *Mpi documents.* `https://www.mpi-forum.org/docs/`, Accessed on July 13, 2024. 17

[51] *The openmp api specification for parallel programming.* `https://www.openmp.org/`, Accessed on July 13, 2024. 19

[52] Surbiryala, Jayachander and Chunming Rong: *Cloud computing: History and overview.* In *2019 IEEE Cloud Summit*, pages 1–7. IEEE, 2019. 21

[53] Kaufman, Lori M: *Data security in the world of cloud computing.* IEEE Security & Privacy, 7(4):61–64, 2009. 21

[54] *Microsoft azure.* `https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/`, Accessed on April 07, 2024. 21

[55] *Google cloud.* `https://cloud.google.com/`, Accessed on April 07, 2024. 21

[56] Goldberg, Robert P: *Survey of virtual machine research.* Computer, 7(6):34–45, 1974. 22

[57] Hoefer, Christina N and Georgios Karagiannis: *Taxonomy of cloud computing services.* In *2010 IEEE Globecom Workshops*, pages 1345–1350. IEEE, 2010. 22

[58] *Aws ec2.* `https://aws.amazon.com/pt/ec2/`, Accessed on June 19, 2024. 23

[59] *Google colab.* `https://colab.google/`, Accessed on June 19, 2024. 24

[60] *What is netflix?* `https://help.netflix.com/en/node/412`, Accessed on Setember 9, 2024. 24

[61] Sharma, Sugam: *Evolution of as-a-service era in cloud.* arXiv preprint arXiv:1507.00939, 2015. 24, 25

[62] Lynn, Theo, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha: *A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms.* In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 162–169. IEEE, 2017. 24

[63] Deniziak, Stanislaw and Slawomir Bąk: *Scheduling of distributed applications in hhpcaas clouds for internet of things.* In *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 1–4. IEEE, 2020. 25

[64] Alves, Maicon Melo: *What is cloud computing?* In *High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment*, pages 9–25. Springer, 2023. 25

[65] Borin, Edson and Otávio O Napoli: *Deploying and configuring infrastructure.* In *High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment*, pages 55–74. Springer, 2023. 25

[66] *Azure cyclecloud documentation.* `https://learn.microsoft.com/en/azure/cyclecloud/?view=cyclecloud-8`, Accessed on June 20, 2024. 26

[67] *Cloud hpc toolkit.* `https://cloud.google.com/hpc-toolkit/docs/overview`, Accessed on June 20, 2024. 27

[68] *Slurm - workfload manager.* `https://slurm.schedmd.com/`, visited on 2015-25-03. 28

[69] *Aws parallelcluster user guide (v3).* `https://docs.aws.amazon.com/pdfs/parallelcluster/latest/ug/aws-parallelcluster-ug.pdf`, Accessed on Mars 28, 2024. 28

[70] Durillo, Juan J and Radu Prodan: *Multi-objective workflow scheduling in amazon ec2.* Cluster computing, 17:169–189, 2014. 31, 36

[71] Topcuoglu, Haluk, Salim Hariri, and Min You Wu: *Performance-effective and low-complexity task scheduling for heterogeneous computing.* IEEE transactions on parallel and distributed systems, 13(3):260–274, 2002. 31

[72] Yu, Jia, Michael Kirley, and Rajkumar Buyya: *Multi-objective planning for workflow execution on grids.* In *2007 8th IEEE/ACM International Conference on Grid Computing*, pages 10–17. IEEE, 2007. 31

[73] Sadooghi, Iman, Geet Kumar, Ke Wang, Dongfang Zhao, Tonglin Li, and Ioan Raicu: *Albatross: An efficient cloud-enabled task scheduling and execution framework using distributed message queues.* In *2016 IEEE 12th International Conference on e-Science (e-Science)*, pages 11–20. IEEE, 2016. 31, 33, 36, 37

[74] Zaharia, Matei, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica: *Spark: Cluster computing with working sets.* In *2nd USENIX workshop on hot topics in cloud computing (HotCloud 10)*, 2010. 31

[75] White, Tom: *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012. 31

[76] Rodriguez, Maria A and Rajkumar Buyya: *Budget-driven scheduling of scientific workflows in iaas clouds with fine-grained billing periods.* ACM Transactions on Autonomous and Adaptive Systems (TAAS), 12(2):1–22, 2017. 32, 36, 37

[77] Krämer, Michel, Hendrik M Würz, and Christian Altenhofen: *Executing cyclic scientific workflows in the cloud.* Journal of Cloud Computing, 10(1):1–26, 2021. 33, 34, 36

[78] Deelman, Ewa, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira Da Silva, Miron Livny, *et al.*: *Pegasus, a workflow management system for science automation.* Future Generation Computer Systems, 46:17–35, 2015. 34

[79] Chen, Huangke, Xiaomin Zhu, Guipeng Liu, and Witold Pedrycz: *Uncertainty-aware online scheduling for real-time workflows in cloud service environment.* IEEE Transactions on Services Computing, 14(4):1167–1178, 2018. 34, 36

[80] Rodriguez, Maria A and Rajkumar Buyya: *Scheduling dynamic workloads in multi-tenant scientific workflow as a service platforms.* Future Generation Computer Systems, 79:739–750, 2018. 34

[81] Taghinezhad-Niar, Ahmad, Saeid Pashazadeh, and Javid Taheri: *Qos-aware online scheduling of multiple workflows under task execution time uncertainty in clouds.* Cluster Computing, 25(6):3767–3784, 2022. 34, 36

[82] Garg, Neha, Damanpreet Singh, and Major Singh Goraya: *Energy and resource efficient workflow scheduling in a virtualized cloud environment.* Cluster Computing, 24:767–797, 2021. 35

[83] Liu, Jiagang, Ju Ren, Wei Dai, Deyu Zhang, Pude Zhou, Yaoxue Zhang, Geyong Min, and Noushin Najjari: *Online multi-workflow scheduling under uncertain task execution time in iaas clouds.* IEEE Transactions on Cloud Computing, 9(3):1180–1194, 2019. 35

[84] Uzun, Berna, Mustapha Taiwo, Aizhan Syidanova, and Dilber Uzun Ozsahin: *The technique for order of preference by similarity to ideal solution (topsis).* Application of multi-criteria decision analysis in environmental and civil engineering, pages 25–30, 2021. 35

[85] Hsu, Chih Chiang, Kuo Chan Huang, and Feng Jian Wang: *Online scheduling of workflow applications in grid environments.* Future Generation Computer Systems, 27(6):860–870, 2011. 35

[86] *Sdumont.* `https://sdumont.lncc.br/`, Accessed on July 10, 2024. 46

[87] *National center for biotechnology information (ncbi), project prjna743046.* `https://www.ncbi.nlm.nih.gov/Traces/study/?acc=PRJNA743046&o=acc_s%3Aa`, Accessed on Mars 01, 2024. 55

Annex I

Paper published in the International Conference Euro-Par 2024

A Framework for Automated Parallel Execution of Scientific Multi-workflow Applications in the Cloud with Work Stealing

Helena S. I. L. Silva[1], Maria C. S. Castro[2], Fabricio A. B. Silva[3], and Alba C. M. A. Melo[1](✉)

[1] University of Brasilia (UnB), Brasilia 70910-900, Brazil
200051873@aluno.unb.br, alves@unb.br
[2] Rio de Janeiro State University (UERJ), Rio de Janeiro, Brazil
clicia@ime.uerj.br
[3] Fundacao Oswaldo Cruz (Fiocruz), Rio de Janeiro, Brazil
fabricio.silva@fiocruz.br

Abstract. In this paper, we propose and evaluate an MPI/OpenMP framework to execute cloud applications composed of scientific linear multi-workflows with unknown task execution times and substantial I/O activity. In order to achieve load balancing, our framework incorporates a two-level work stealing strategy, with intra-node and inter-node stealing. The framework was evaluated in a cluster of 16 virtual machine (VM) instances (4 vCPUs), deployed on AWS Parallel Cluster. The results show that, for a real Bioinformatics application composed of 400 workflows, we are able to reduce the execution time from 1 h and 57 min (sequential) to 2 min and 52 s (16 instances), achieving a speedup of 40.89x, with 64 threads.

Keywords: Scientific workflows · Work stealing · Cloud computing

1 Introduction

Scientific workflows are a powerful tool for executing complex scientific applications composed of tasks with temporal dependencies, where the output of one task is the input of another. Workflows are expressed as a Directed Acyclic Graph (DAG). The general problem of scheduling tasks of an arbitrary workflow is proven NP-Complete [20], and the problems of mininizing the period and minimizing the latency for scheduling workflows composed of a sequence of tasks (linear workflow) is proven NP-Hard [3] for most formulations. For this reason, many heuristic strategies have been proposed [13,14].

Workflow scheduling strategies [3,11] may be classified as static, when they use a large amount of information about the tasks and computing environment

Supported by CNPq/Brazil and Fiocruz/Brazil.