A Weighted Bi-objective Strategy for Executing Scientific Workflows in Containerized Environments*

Wesley Ferreira[1], Liliane Kunstmann[2], Yuri Frota[1], Luan Teylo[3], Daniel de Oliveira[1]

[1]Institute of Computing – Universidade Federal Fluminense (IC/UFF)

[2]Instituto de Matemática Pura e Aplicada (IMPA)

[3]Centre INRIA de l'université de Bordeaux

wesleyferreira@id.uff.br, liliane.kunstmann@impa.br, luan.teylo@inria.fr,

{yuri,danielcmo}@ic.uff.br

Abstract. *Scientific workflows support the execution of complex simulation-based experiments across heterogeneous computing environments. Containerization technologies, such as Docker, improve portability by encapsulating tasks together with their dependencies. However, they also introduce challenges in resource management, as containers incur additional memory and CPU overhead and may execute concurrently on the same virtual or physical machine. These challenges are particularly critical in memory-constrained environments, where inefficient scheduling can lead to performance degradation or even task failures. To address this issue, we propose a weighted bi-objective scheduling strategy that considers memory consumption and execution time, allowing users to prioritize one objective or achieve a balance between the two. Experimental evaluations with both synthetic and real-world workflows demonstrate that our approach enhances performance and resource utilization.*

1. Introduction

Scientific workflows (hereafter referred to as workflows) are abstractions used to specify simulation-based experiments [de Oliveira et al. 2019]. They consist of sequences of computational tasks with well-defined data dependencies and execution constraints. The scale of an experiment defines the workflow's characteristics, including the volume of data processed and the complexity of its specification. This complexity arises from both the number of tasks and the structure of their control flow. To complete execution within a feasible time, such workflows often rely on distributed computing environments using workflow systems [Suter et al. 2026].

Well-known workflow systems, such as Pegasus [Deelman et al. 2021], SciCumulus [de Oliveira et al. 2012], and Parsl [Babuji et al. 2019], have been used to execute workflows across a broad range of environments, including HPC clusters, supercomputers, and clouds. Despite representing a step forward, many of these systems present portability limitations. In several cases, they were developed with a particular type of infrastructure in mind, which hinders their adaptability and usability in other contexts. For example, SciCumulus was designed for clouds, making its deployment in other environments less straightforward. Similarly, Pegasus relies on HTCondor, which may not be available or easily supported in many environments. Even systems claiming to support multiple execution platforms still impose constraints.

One way to mitigate the impact of infrastructure-specific configurations on workflow execution and enable the same system to execute seamlessly across multiple environments is

*This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001. The authors would also like to thank CNPq and FAPERJ.

the use of containerization [Struhár et al. 2020]. Containers, such as Docker and Singularity, encapsulate applications, which may include entire workflows, along with all their required dependencies, configuration files, and other necessary binaries. As a result, a containerized workflow can be migrated and executed across heterogeneous environments with minimal configuration overhead or manual intervention from the user. In fact, containerization has been adopted by some workflow systems, such as Nextflow [Di Tommaso et al. 2017] and `AkôFlow` [Ferreira et al. 2024], both of which natively support the execution of workflows in containerized environments.

Despite the clear advantage of enabling seamless execution of workflows across heterogeneous environments, container-based workflow execution also introduces challenges. One challenge is that, in addition to the application executed by each workflow task, the container itself consumes computational resources, *e.g.*, memory and CPU, from the host machine and can share these resources with other containers. If containers executing workflow tasks are not scheduled carefully, memory may become either underutilized or overutilized. Underutilization can lead to inefficient, slower, and potentially more costly executions (in the case of clouds, where the pay-as-you-go model is applied). At the same time, overutilization may cause performance degradation and even task failures, particularly in resource-constrained environments.

Therefore, scheduling containerized workflow tasks presents a challenge, particularly because the memory consumption of tasks executing the same activity can vary substantially depending on the characteristics of their input data. This variability makes it difficult to define in advance which host machine will execute the containerized task. Scheduling containerized tasks to the appropriate machines, based on their memory and CPU demands, can reduce resource contention and lead to more efficient use of the available hardware. Achieving a good balance between memory usage and workflow execution time is a priority, as it can enhance the overall throughput and performance of the containerized workflow.

In this paper, we propose a bi-objective weighted workflow execution strategy designed to improve the scheduling of containerized workflows. The proposed strategy allows users to balance memory usage and task execution time by employing a weighted function that considers both criteria simultaneously. This function takes into account the resource requirements of containerized tasks, specifically memory and execution time, as well as the current load on the host machine. We evaluate the proposed strategy using a set of synthetic workflows with varying structures and task profiles, along with the Montage workflow [Sakellariou et al. 2009] executed on the `AkôFlow` system. Experimental results demonstrate improvements in both performance and resource usage for synthetic and real-world workflows.

2. Problem Formulation

The problem of task scheduling has long been acknowledged as a challenging issue. Its resolution requires not only considering the availability and capacity of computing resources but also an analysis of the requirements associated with each task [Muntz and Coffman 1969]. The goal of task scheduling is to produce a mapping of tasks onto the available computational resources such that all task-specific constraints are satisfied, while optimizing one or more objective functions, *e.g.*, minimizing the makespan or maximize resource utilization.

The task scheduling problem becomes particularly complex in the context of workflows, where non-trivial data dependencies impose hard constraints on the execution order of tasks. Ensuring that these dependencies are respected is critical for maintaining both the correctness and the reliability of workflow execution. Typically, tasks, associated data, and their data dependencies are represented using a Directed Acyclic Graph (DAG), denoted as

$G = (V, A, a, \omega)$, following the formalism described by [Teylo et al. 2017]. Within this formalism, the set $V = N \cup D$ comprises tasks $i \in N$ and data files $d \in D$, while A denotes the set of directed edges that encode the precedence relationships between tasks and data files. Each task $i \in N$ is associated with a workload a_i, and ω_{id} represents the cost associated with the edge $(i, d) \in A$. Notably, under this formalism, every task in G is always preceded and succeeded by a data file $d \in D$, reflecting the structured dataflow.

The execution environment of a workflow consists of the set of all resources $j \in M$ available for task execution, which can be a physical machine or a Virtual Machine (VM). Each $j \in M$ is characterized by a computational slowdown index cs_j, which quantifies the ratio between the computing capacity of j and that of a baseline reference resource. In addition, each j is associated with an available memory capacity, denoted as M_j^{free}, and a number of available processing cores, denoted $\text{CPU}_j^{\text{free}}$. Consequently, the execution time of a workflow task $i \in N$ on a given resource $j \in M$ can be expressed as $T_{ij} = a_i \times cs_j$. Furthermore, each task i requires a specific amount of memory, M_i^{req}, as well as a certain number of processing cores, denoted as CPU_i, which must be satisfied for the task to be executed on the chosen resource.

The objective function, which we henceforth refer to as `AkôScore`, is computed for each task that is ready for scheduling, *i.e.*, whose data dependencies are satisfied. The goal of this score is twofold: (i) to promote the efficient utilization of memory and CPU across the available resources (since containerized tasks can execute concurrently in the same resource), and (ii) to minimize the makespan of the workflow. To achieve these objectives, we propose a scoring strategy that balances the trade-off between memory consumption and workflow execution time. Specifically, for a given task i in the scheduling queue, the `AkôScore` is evaluated for each $j \in M$. The resulting score, denoted as $S_{i,j}$, is formally defined in Equation 1:

$$S_{i,j} = A_{i,j} \cdot \left(\alpha \cdot \frac{1}{T_{i,j}} + (1 - \alpha) \cdot \left(\frac{M_j^{\text{free}} - M_i^{\text{req}}}{M^{\text{max}}} \right) \right) \tag{1}$$

The parameter α defines the relative importance assigned to the makespan and memory usage objectives, respectively. By adjusting this weight, the proposed strategy can be fine-tuned according to user preferences, *i.e.*, users may prioritize either a "fast" execution (without necessary using all resources all the time) or a "memory-optimized" execution (that aims at occupying the resources as much as possible), depending on the desired trade-off, where $\alpha \in [0, 1]$. It is worth noting that the memory consumption objective is normalized by M^{max}, which corresponds to the maximum memory capacity among all available resources (in the context of this paper, resources refer to VMs in AWS cloud). Furthermore, we define $A_{i,j} \in \{0, 1\}$ as a feasibility indicator of the scheduling decision, which is formally expressed in Equation 2.

$$A_{i,j} = \begin{cases} 1, & \text{if } \text{CPU}_j^{\text{free}} \geq \text{CPU}_i \text{ and } M_j^{\text{free}} \geq M_i^{\text{req}} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

Thus, j selected for the execution of containerized task i is the one that yields the highest value of `AkôScore`, as defined in Equation 3. In cases where $S_{i,j} = 0$ for all $j \in M$, task i cannot be scheduled because of insufficient available resources; consequently, it remains in the scheduling queue and will be reconsidered in the subsequent scheduling round. It is important to emphasize that, although this paper is primarily motivated by containerized environments, the proposed score strategy does not rely on any container-specific features yet (*e.g.*, image

size, shared caches), which is planned as future work.

$$S_i^* = \arg\max_{j \in M} S_{i,j} \tag{3}$$

3. The Proposed Bi-Objective Workflow Execution Strategy

This section presents the proposed scheduling strategy designed for containerized workflows. The discussion is structured as follows: first, we describe the underlying workflow system. Then, we detail the scheduling algorithm.

3.1. A Brief Tour to `AkôFlow`

`AkôFlow` [Ferreira et al. 2024] is a workflow system designed to support the execution of workflows in heterogeneous containerized infrastructures. Unlike several workflow systems that are typically bound to specific schedulers or infrastructures, `AkôFlow` builds upon Kubernetes to enable portable and provenance-aware workflow execution across multiple environments, *e.g.*, clusters and clouds. `AkôFlow` follows a layered architecture composed of five modules (Figure 1): (i) Client, (ii) Server, (iii) Proxy, (iv) Worker, and (v) Provenance Manager.

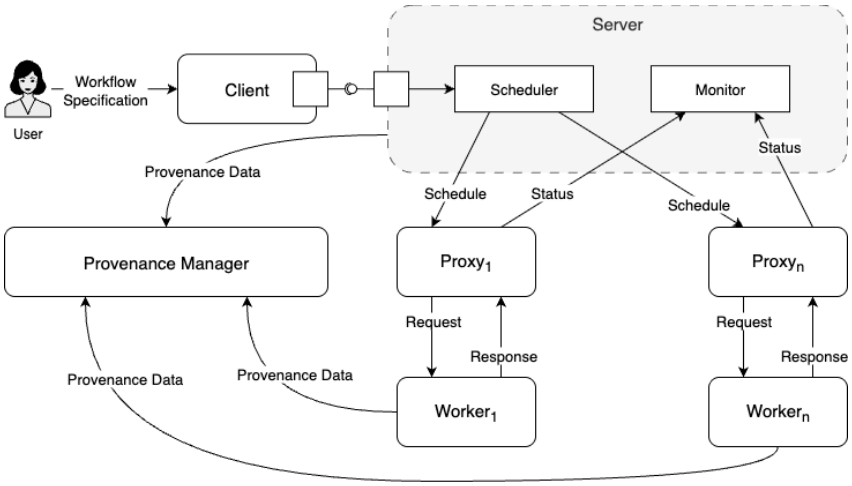

Figure 1. The Architecture of `AkôFlow`.

Users specify workflows through a YAML-based description that defines tasks, their data dependencies, and resource constraints (*e.g.*, CPU, memory, storage). The *Client* serializes this specification and submits it to the *Server*, which orchestrates workflow execution by scheduling multiple tasks to the *Workers*. The *Scheduler* component, at the core of the *Server*, schedules workflow tasks as containers, deployed as Kubernetes Pods. Each host machine can execute several Pods. `AkôFlow` schedules tasks based on two possible execution strategies: First-Data-First (FDF), which enables pipelined execution of tasks as soon as input data becomes available, and First-Activity-First (FAF), which enforces synchronization at each level of the workflow [Ogasawara et al. 2011]. It is important to note that the execution strategy proposed in this paper must be implemented within the *Scheduler*, which is designed to support changes in the algorithm without requiring recompilation. For details, refer to `https://github.com/UFFeScience/akoflow`.

Task execution takes place in the *Worker*, which instantiates containers, attaches storage volumes, and monitors resource usage. The *Proxy* facilitates efficient communication between the *Server* and *Workers*, particularly for tracking execution and identifying errors through the *Monitor* component. Finally, the *Provenance Manager* collects provenance data from both

Workers and the *Server* at runtime via the Kubernetes Metrics API, recording CPU, memory, and storage usage, as well as logs and execution states, in an SQLite database. This provenance model connects workflows, tasks, and performance metrics, thereby enabling traceability and reproducibility. In `AkôFlow`, the execution of a task progresses through a well-defined set of states. These states are: (i) *Pending*, (ii) *Ready*, (iii) *In Execution*, (iv) *Finished*, and (v) *Failed*. A task initially is in the *Pending* state until all of its data dependencies are satisfied. Once every predecessor task has completed, the task transitions to the *Ready* state and is inserted into the *Ready* queue of `AkôFlow`. This queue, therefore, represents the set of tasks that are eligible for scheduling by the *Scheduler*.

The *Scheduler* monitors the *Ready* queue and applies the selected scheduling strategy (described in Subsection 3.2) as soon as new tasks become available for execution. This design choice ensures that scheduling remains adaptive to the current state of workflow execution, in contrast to static, pre-computed scheduling plans. Once a task is assigned to a computational resource, it transitions to the *In Execution* state. During this phase, the task consumes resources, processes its input data, and generates the corresponding outputs. If execution completes successfully, the task enters the *Finished* state, informing that its outcomes are now available for dependent tasks. On the other hand, if errors occur during execution, the task transitions to the *Failed* state, where appropriate fault-handling or recovery strategies may be applied.

3.2. Scheduling Algorithm

This subsection describes the proposed scheduling algorithm (Algorithm 1) for containerized workflow tasks, which uses the `AkôScore` to schedule tasks to resources (in this context, VMs). The algorithm aims to maximize overall execution performance while respecting both memory and CPU constraints. The input to the algorithm is a list of tasks N and a list of available resources M. Each task $i \in N$ has associated resource requirements, including memory (M^{req_i}) and CPU cores ($\text{CPU}^{\text{req}_i}$), and a status that can be *Ready*, *In Execution*, *Finished*, or *Failed*. Each $j \in M$ maintains its available resources (M_j^{free} and $\text{CPU}_j^{\text{free}}$).

Algorithm 1 operates iteratively. The main loop (lines 1–25) continuously checks for the existence of tasks in the list N that are marked with the status *Ready*. For each of these tasks (line 2), the algorithm iterates over all available resources in the set M (line 5) and evaluates whether each VM meets the task's CPU requirement (line 6). Only if the resource has sufficient CPUs, the algorithm then verifies if the resource also has available memory to execute the containerized task (line 7). When both constraints are satisfied, the `AkôScore` for the task–resource pair is computed (line 8). If this score exceeds the current best score, the algorithm updates the best score and records the corresponding resource as the candidate for allocation (lines 10–11).

Once all resources have been evaluated for the current task, the algorithm checks whether a suitable resource was found (line 16). If so, the task is scheduled on the selected resource (line 17), and the resource's available memory and CPU are updated to reflect the schedule (lines 18–19). The task status is then updated to *In Execution* (line 20), indicating that it is currently executing. If no resource satisfies the CPU and memory requirements for the task, the task is skipped in this iteration (line 22) but remains in the list of ready tasks to be reconsidered in subsequent iterations. Through this process, Algorithm 1 ensures that tasks are scheduled while respecting the resource limitations and prioritizing assignments that maximize the expected execution performance as measured by the `AkôScore`. The iterative nature of the algorithm guarantees that all ready tasks are continuously evaluated until no further schedules are possible.

Algorithm 1 Scheduling Algorithm using `AkôScore`

Require: List of workflow tasks N, List of available resources M

while $\exists i \in N$ such as $status(i) =$ "$Ready$" **do**
  for all $i \in N$ such as $status(i) =$ "$Ready$" **do**
    $S_i^* \leftarrow -\infty$
    $j^* \leftarrow \emptyset$
    for all $j \in M$ **do**
      if $\text{CPU}_j^{\text{free}} \geq \text{CPU}_i^{\text{req}}$ **then** ▷ Check CPU availability
        if $M_j^{\text{free}} \geq M_i^{\text{req}}$ **then** ▷ Check memory availability
          $S_{i,j} \leftarrow AkôScore(i, j)$ ▷ Compute AkôScore for task i on resource j
          if $S_{i,j} > S_i^*$ **then**
            $S_i^* \leftarrow S_{i,j}$
            $j^* \leftarrow j$
          end if
        end if
      end if
    end for
    if $j^* \neq \emptyset$ **then**
      $Schedule(i \rightarrow j^*)$
      $M_{j^*}^{\text{free}} \leftarrow M_{j^*}^{\text{free}} - M_i^{\text{req}}$
      $\text{CPU}_{j^*}^{\text{free}} \leftarrow \text{CPU}_{j^*}^{\text{free}} - \text{CPU}_i^{\text{req}}$
      $status(i) =$ "$In\ Execution$"
    else
      Skip task i ▷ No resource has sufficient CPU or memory at this moment
    end if
  end for
end while

4. Experimental Evaluation

This section presents the experimental evaluation of the proposed scheduling strategy. The objective is to examine the performance of the execution strategy across different workflows, VM configurations (since we have chosen the AWS cloud as execution environment), and parameterizations of the α parameter.

4.1. Environment and Experiment Setup

All experiments reported in this section were executed in the AWS cloud. To ensure a representative coverage of different resource profiles, we selected three VM types that differ in terms of CPU architecture, number of vCPUs, and memory capacity: `c7i.large` (2 Intel vCPUs and 4GiB of memory), `c7a.xlarge` (4 AMD vCPUs and 8GiB of memory), and `c6i.xlarge` (4 Intel vCPUs and 8GiB of memory). These VM types were chosen to enable a comparison of scheduling behavior across processor generations, as well as to highlight architectural differences between AMD- and Intel-based VMs. In addition, these VM types are known to present variations in computational performance [De Lima et al. 2024], which is particularly relevant when evaluating the effectiveness of the proposed scheduling strategy under performance-oriented configurations, such as when the weighting parameter is set to $\alpha = 1.0$. All VMs were configured with Ubuntu 22.04 LTS as the operating system and had Kubernetes version 1.33 installed to provide the containerized execution environment. To reduce variability introduced by geographic distribution, all VMs were provisioned in the same AWS region (us-east-1). For each experimental run, the VMs were deployed at the start of the workflow execution and remained active until the execution was completed.

We evaluated the proposed scheduling strategy using two different categories of workflows: (i) a real-world scientific workflow, Montage [Sakellariou et al. 2009], and (ii) a set

of synthetic workflows specifically designed to isolate and control task characteristics systematically. Montage is a widely used astronomy workflow that constructs image mosaics by combining a collection of individual input tiles. The workflow comprises a set of tasks with heterogeneous computational requirements. Some tasks are I/O-bound, whereas others are CPU-intensive. Due to this diversity in computational behavior, Montage has become a *de facto* benchmark for evaluating workflow scheduling approaches. We also generated synthetic workflows derived from a CPU- and memory-intensive benchmark proposed by [Alves and Drummond 2017]. This benchmark performs a series of computationally intensive mathematical operations on a square matrix of size $N \times N$. The parameter N controls both execution time and memory consumption, as larger matrix sizes entail not only greater computational effort but also higher memory requirements. Based on this property, we derived synthetic applications by tuning N so that tasks would present varying resource demands. To be consistent with the environment setup previously described, we considered the maximum available memory of 8 GiB as the reference capacity and defined two different memory ranges. For each synthetic task, the value of N was randomly selected such that its memory consumption would fall within the specified range. Using this methodology, we established two synthetic task classes, each characterized by different memory usage profiles.

Class A. This class represents low-memory tasks. Each task is configured to use between 10–20% of the reference memory capacity, with an upper limit of 1.6 GiB. These tasks are intended to model lightweight applications that place minimal demands on memory while still requiring computational resources.
Class B. This class represents high-memory tasks. Each task consumes more than 60% of the reference memory capacity, corresponding to at least 4.8 GiB. These tasks are designed to emulate memory-intensive applications, where execution is constrained by memory availability.

Using the two aforementioned task classes, we constructed six workflows based on the same specification but varying task classes, as presented in Figure 2. Specifically, workflows $W1$ and $W2$ represent the two homogeneous cases, consisting exclusively of class A (white circles) and class B (grey circles) tasks, respectively. Workflows $W3a$ and $W3b$ represent workflows containing an equal proportion of class A and class B tasks. The distinction between $W3a$ and $W3b$ lies in the ordering of tasks: the first half of $W3a$ is composed of class A tasks, whereas $W3b$ is the opposite. This design enables us to investigate whether task ordering affects scheduling and performance. Finally, workflows $W4$ and $W5$ capture skewed distributions, incorporating 30/70 and 70/30 splits between class A and class B tasks, respectively. These configurations represent intermediate levels of heterogeneity, situated between the entirely homogeneous and balanced workflows.

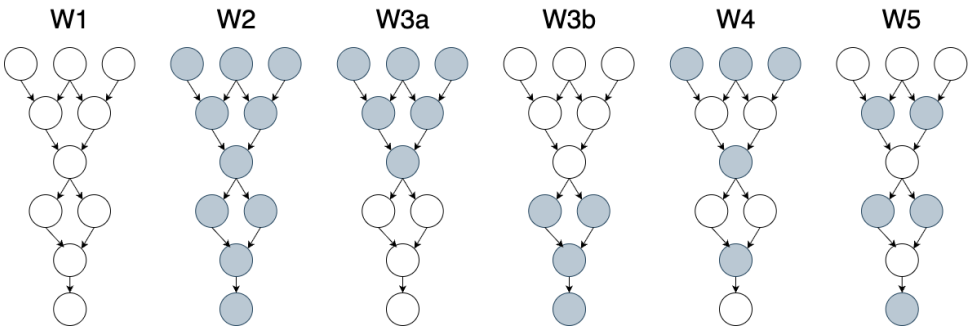

Figure 2. The six classes of synthetic workflows.

Each workflow was executed under three distinct settings of the parameter $\alpha = \{0.0, 0.5, 1.0\}$. A setting of $\alpha = 0.0$ corresponds to a memory-optimized strategy, which emphasizes maximizing memory consumption during workflow execution by avoiding the use of more VMs by concurrently executing multiple containers on the same VM. On the other hand, a setting of $\alpha = 1.0$ represents a performance-oriented strategy, prioritizing the reduction of makespan. An intermediate value of $\alpha = 0.5$ is used to achieve a balanced strategy that considers both memory usage and performance optimization simultaneously.

4.2. Results Discussion

The first evaluation was conducted by executing the six synthetic workflows described in Subsection 4.1. For each task within these workflows, `AkôFlow` provided an estimated execution time, which was determined based on the calculated slowdown index. In this study, the VM type `c6i.xlarge` was used as the baseline reference, while the slowdown indices for `c7a.xlarge` and `c7a.large` were determined to be 1.42 and 1.39, respectively. The resulting makespan for each workflow, considering the different values of the parameter α, is presented in Figure 3.

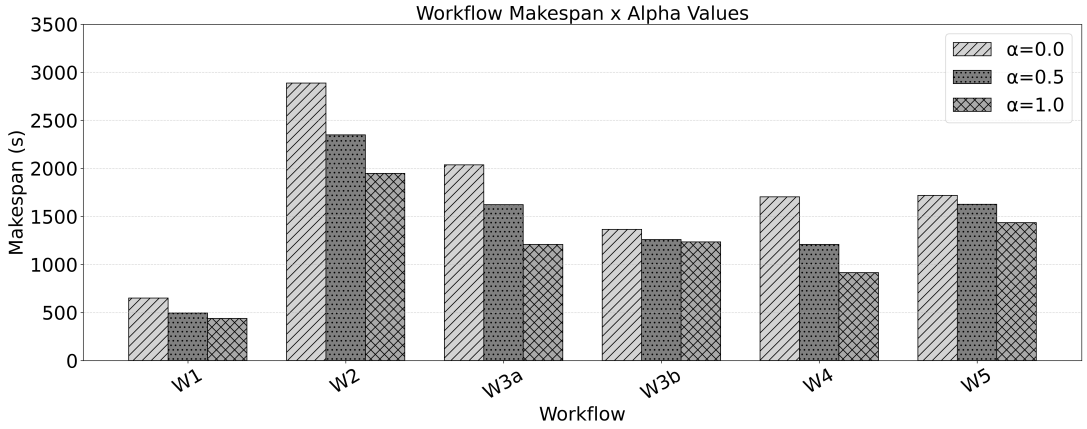

Figure 3. Makespan for the synthetic workflows and different values of parameter α.

By analyzing Figure 3, a clear relationship between the parameter α and the makespan for the six evaluated workflows can be observed. Across all workflows, a consistent trend emerges: higher values of α, which favor the use of higher-performance VMs, systematically lead to reduced makespan, as anticipated. This effect is evident for workflows dominated by tasks of the same class (A or B), such as $W1$ and $W2$. Workflow $W1$ achieves a minimum total time of 438 seconds when $\alpha = 1.0$, increasing to 651 seconds when $\alpha = 0.0$, corresponding to a 49% increase. Similarly, the makespan of $W2$ rises from 1,949 seconds to 2,889 seconds, a 48% increase, as α decreases from its maximum to minimum value. These findings indicate that workflows composed of homogeneous task types are highly sensitive to resource allocation and scheduling strategies.

Mixed workflows, including $W3a$, $W3b$, $W4$, and $W5$, display different behavior. $W3a$, with an even mix of task types, shows a significant increase in makespan from 1,208 s to 2,038 seconds as α decreases, whereas $W3b$ exhibits a smaller increase from 1,235 s to 1,365 s. This difference likely reflects the impact of activity ordering and dependencies, which can mitigate the cumulative effect of slower VMs. Similarly, $W4$ and $W5$, which contain uneven distributions of activity types, show intermediate trends: $W4$ increases from 915 s to 1,704 seconds, while $W5$ increases from 1435 s to 1720 seconds.

To show the task scheduling behavior, we selected two representative workflows (due to space limitations), $W1$ and $W4$, and present their scheduling plans in Figures 4 and 5. In these figures, heatmaps depict the assignment of tasks to specific VMs over time, providing a visual representation of resource usage. For $\alpha = 1.0$, both workflows present a straightforward scheduling strategy: all tasks are scheduled exclusively to the `c7a.xlarge` VM. This outcome is expected, as this VM type offers the highest computational performance and sufficient memory capacity to execute all tasks efficiently.

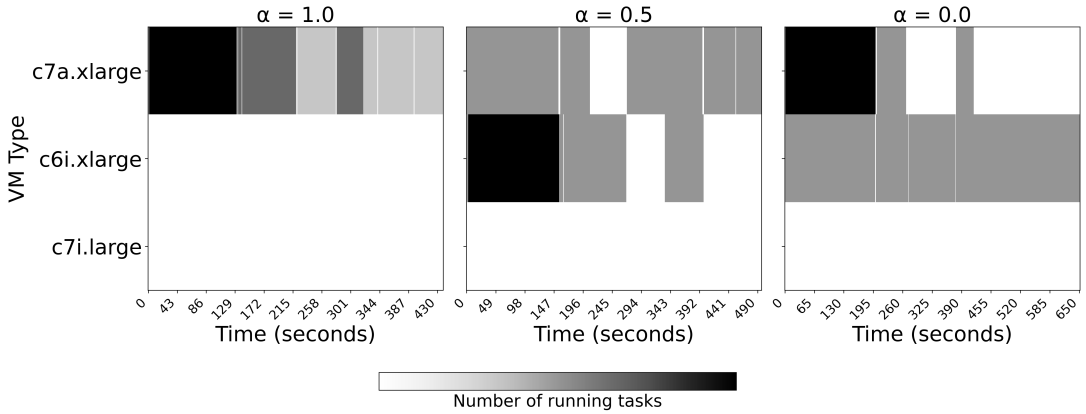

Figure 4. task scheduling plan for workflow $W1$.

However, when α is reduced to 0.5 and 0.0, the scheduling patterns diverge between the two workflows. In $W1$, tasks are distributed almost evenly across the available VMs, except `c7i.large`. This VM receives fewer tasks because it has only half the memory capacity of the other two VMs. Consequently, its `AkôScore` for $\alpha = \{0.0, 0.5\}$ — which is defined in terms of the amount of free memory — remains consistently lower than that of the other VMs. In contrast, $W4$ displays a more heterogeneous allocation: memory-intensive tasks are preferentially scheduled to `c7a.xlarge`, while the remaining tasks are scheduled on `c6i.xlarge`. This behavior highlights the influence of task characteristics and workflow composition on scheduling decisions, particularly when resource selection is constrained by the parameter α. These results show that both the selection of VM types (as controlled by α) and the composition of workflow tasks influence the makespan; however, `AkôFlow` was able to prioritize makespan reduction or maximize resource usage as defined by the user.

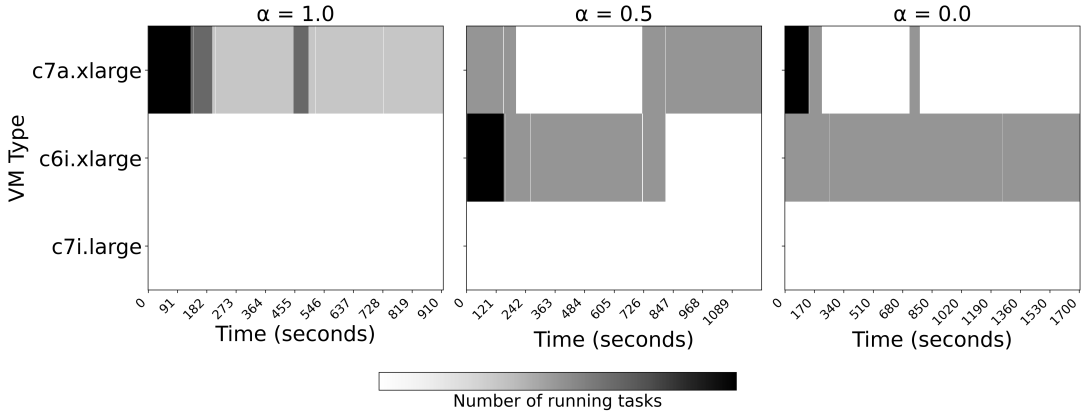

Figure 5. Task scheduling plan for workflow $W4$.

The second evaluation was conducted using the Montage workflow, with the makespan

presented in Figure 6 for different values of the parameter α. As shown, the configuration designed to prioritize makespan reduction ($\alpha = 1.0$) outperforms the balanced configuration, represented by $\alpha = 0.5$, achieving reduced execution time. However, the behavior of the $\alpha = 0.0$ configuration is particularly noteworthy, as it presents a shorter makespan than the $\alpha = 1.0$ configuration. At first glance, this may seem counterintuitive, since one might expect that fully prioritizing the execution time objective would always yield superior results in that dimension. Upon closer examination, this behavior can be understood in the context of the greedy nature of the proposed scheduling strategy. Specifically, for each task in the workflow, the strategy immediately assigns it to a VM with available resources. In doing so, the scheduler dynamically adapts to resource availability while attempting to optimize the chosen objective.

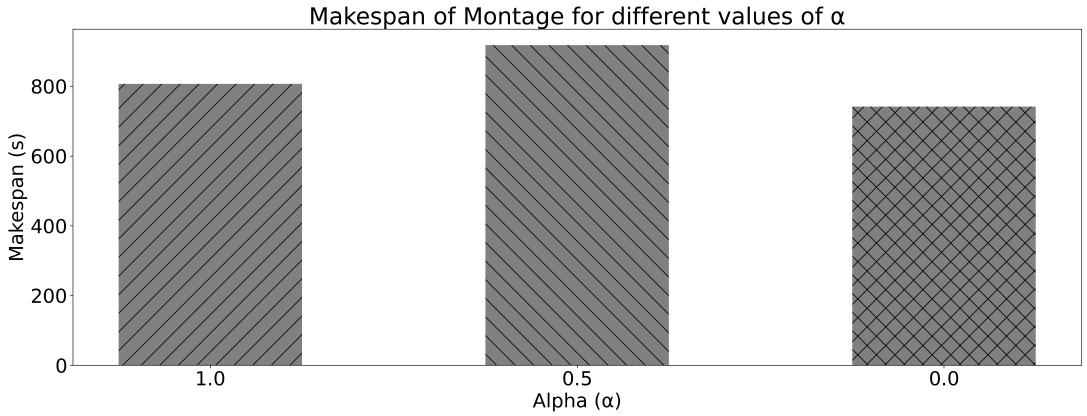

Figure 6. The makespan of Montage workflow for different α values.

For $\alpha = 0.0$ execution, this approach resulted in scheduling tasks to the most powerful VMs from the beginning. Subsequent tasks continue to be scheduled on these same VMs to maximize resource usage, effectively concentrating compute-intensive tasks on the powerful VMs. As a result, even though the configuration is not explicitly designed to minimize makespan, the execution time can be faster than that of $\alpha = 1.0$, where tasks may be distributed differently across VMs. Figure 7 illustrates this scheduling pattern, highlighting the impact of the greedy strategy on execution performance. These results suggest that greedy scheduling approaches, when combined with intelligent strategies, can leverage computational heterogeneity to reduce makespan, even in configurations that do not explicitly prioritize it.

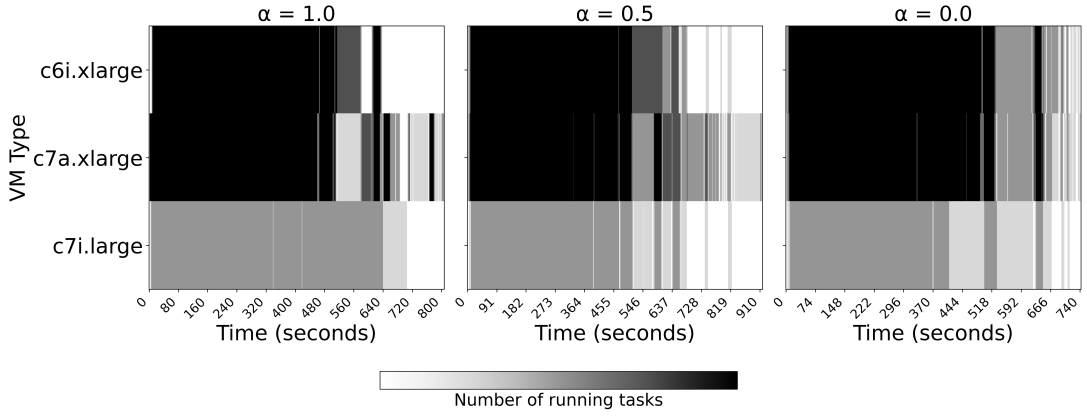

Figure 7. The task scheduling plan for Montage workflow.

5. Related Work

Previous studies have investigated the use of containers and virtualization to improve workflow execution in distributed and cloud environments. Early work explored the integration of containers into workflow systems [Zheng and Thain 2015, Zheng et al. 2017], emphasizing portability and isolation. More recent research on Workflow as a Service (WaaS) has focused on scheduling and cost efficiency [Rajasekar and Palanichamy 2021, Karmakar et al. 2024], proposing scheduling heuristics to reduce costs while satisfying Quality of Service.

Another research direction investigates how workflow systems integrate with orchestration infrastructures, *e.g.*, Kubernetes, in IaaS clouds. Early work explored workflow deployment on container schedulers such as Mesos [Zheng et al. 2017]. With Kubernetes emerging as a *de facto* standard, new approaches have been proposed. KubeAdaptor [Shan et al. 2023] bridges workflow scheduling algorithms with Kubernetes' scheduler to improve task ordering and resource utilization. Similarly, [Li et al. 2021] study scheduling of microservice-based workflows, emphasizing challenges in dynamic resource allocation and scalability. Several studies have also examined energy-efficient workflow scheduling in containerized environments. [Sun et al. 2025] propose RMES, a real-time multi-workflow scheduling algorithm that enhances parallelism and resource utilization, focusing on achieving energy savings that surpass those of prior methods. However, none of the approaches found explore the optimization of multiple factors, neither fully address the challenges of allocating multiple types of resources in heterogeneous environments.

6. Conclusions

In this paper, we presented a bi-objective weighted workflow execution strategy designed to improve the scheduling of containerized workflows. By considering both memory usage and task execution time, the proposed strategy allows users to balance resource efficiency and workflow performance according to their priorities. Our experiments, conducted with synthetic workflows and the real-world Montage workflow on heterogeneous AWS VMs, demonstrate the effectiveness of this approach under different computational and memory constraints.

The results highlight several findings. First, the parameter α effectively controls the trade-off between memory usage and makespan, allowing for memory-oriented, performance-oriented, or balanced scheduling strategies. Second, workflow composition and task ordering have a significant influence on scheduling outcomes, particularly for mixed workflows where the distribution of high- and low-memory tasks affects the makespan. Third, even in configurations that do not explicitly prioritize makespan, the use of greedy scheduling can exploit computational heterogeneity to achieve performance gains, as observed in the Montage workflow experiments. Future work will explore dynamic adjustment of the weighting parameter at runtime and the extension of the strategy to larger, more complex workflows and hybrid cloud-HPC environments.

References

Alves, M. M. and Drummond, L. (2017). A multivariate and quantitative model for predicting cross-application interference in virtual environments. *J. of Systems and Soft.*, 128:150–163.

Babuji, Y. N., Woodard, A., et al. (2019). Parsl: Pervasive parallel programming in python. In *HPDC 2019*, pages 25–36. ACM.

De Lima, M., Teylo, L., et al. (2024). An analysis of performance variability in aws virtual machines. In *SSCAD 2024*, pages 312–323. SBC.

de Oliveira, D. et al. (2012). A provenance-based adaptive scheduling heuristic for parallel scientific workflows in clouds. *J. Grid Comput.*, 10(3):521–552.

de Oliveira, D. C. M., Liu, J., and Pacitti, E. (2019). *Data-Intensive Workflow Management: For Clouds and Data-Intensive and Scalable Computing Environments*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers.

Deelman, E., da Silva, R. F., et al. (2021). The pegasus workflow management system: Translational computer science in practice. *J. Comput. Sci.*, 52:101200.

Di Tommaso, P., Chatzou, M., et al. (2017). Nextflow enables reproducible computational workflows. *Nature Biotechnology*, 35(4):316–319.

Ferreira, W. et al. (2024). Akôflow: um middleware para execução de workflows científicos em múltiplos ambientes conteinerizados. In *SBBD 2024*, pages 27–39, Florianópolis/SC. SBC.

Karmakar, K., Tarafdar, A., Das, R. K., and Khatua, S. (2024). Cost-efficient workflow as a service using containers. *Journal of Grid Computing*, 22(1):40.

Li, W., Li, X., and Ruiz, R. (2021). Scheduling microservice-based workflows to containers in on-demand cloud resources. In *2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 61–66.

Muntz, R. and Coffman, E. (1969). Optimal preemptive scheduling on two-processor systems. *IEEE Transactions on Computers*, C-18(11):1014–1020.

Ogasawara, E. S., de Oliveira, D., et al. (2011). An algebraic approach for data-centric scientific workflows. *Proc. VLDB Endow.*, 4(12):1328–1339.

Rajasekar, P. and Palanichamy, Y. (2021). Scheduling multiple scientific workflows using containers on iaas cloud. *Journal of Ambient Intelligence and Humanized Computing*, 12(7):7621–7636.

Sakellariou, R., Zhao, H., and Deelman, E. (2009). Mapping workflows on grid resources: Experiments with the montage workflow. In *Proc. of the CoreGRID ERCIM Working Group Workshop*, pages 119–132. Springer.

Shan, C., Xia, Y., Zhan, Y., and Zhang, J. (2023). Kubeadaptor: A docking framework for workflow containerization on kubernetes. *FGCS*, 148:584–599.

Struhár, V., Behnam, M., Ashjaei, M., and Papadopoulos, A. V. (2020). Real-time containers: A survey. In *Fog-IoT*, volume 80 of *OASIcs*, pages 7:1–7:9.

Sun, Z., Huang, H., Li, Z., and Gu, C. (2025). Energy-efficient real-time multi-workflow scheduling in container-based cloud. *Journal of Combinatorial Optimization*, 49(2):34.

Suter, F., Coleman, T., et al. (2026). A terminology for scientific workflow systems. *FGCS*, 174:107974.

Teylo, L., de Paula Junior, U., Frota, Y., de Oliveira, D., and Drummond, L. M. A. (2017). A hybrid evolutionary algorithm for task scheduling and data assignment of data-intensive scientific workflows on clouds. *FGCS*, 76:1–17.

Zheng, C. and Thain, D. (2015). Integrating containers into workflows: A case study using makeflow, work queue, and docker. VTDC '15, page 31–38, New York, NY, USA.

Zheng, C., Tovar, B., and Thain, D. (2017). Deploying high throughput scientific workflows on container schedulers with makeflow and mesos. In *CCGRID*, pages 130–139.