A novel cloud workflow scheduling algorithm based on stable matching game theory

Lorenzo Cappetti
lorenzo.cappetti@edu.unifi.it
Mat: 7165929

Chiara Peppicelli
chiara.peppicelli@edu.unifi.it
Mat: 7146573

Abstract

Cloud workflow scheduling is a fundamental and challenging problem in heterogeneous cloud environments, as the objectives of minimizing workflow makespan and ensuring fairness among tasks are often conflicting. Recently, Jia et al.[4] proposed SM-CPTD, a workflow scheduling algorithm based on Stable Matching Game Theory (SMGT), which combines stable matching with critical-path optimization and task duplication strategies to jointly address efficiency and fairness.

This report presents an implementation and experimental reproduction of the SM-CPTD algorithm, developed in Java by closely following the workflow model, scheduling procedure, and performance metrics defined in the original paper. The study is conducted on standard scientific workflow benchmarks, including Montage, CyberShake, LIGO, and Epigenomics, and analyzes the behavior of the algorithm under varying communication-to-computation ratios (CCR). The evaluation focuses on Scheduling Length Ratio (SLR), Average VM Utilization (AVU), and fairness-related metrics, providing insight into the effectiveness of the implemented solution. The report details the mathematical formulation of the problem, the implementation choices, and the experimental results obtained. The code and results are publicly available for reproducibility and further analysis (`https://github.com/Cappetti99/stable-matching-game-theory`*) [2].*

1. Introduction

Cloud computing provides scalable and on demand computational resources that enable the execution of complex scientific applications across distributed infrastructures. Scientific workflows, commonly modeled as directed acyclic graphs (DAGs), represent intricate computational pipelines spanning diverse domains including bioinformatics, astronomy, physics, and weather forecasting. Efficient scheduling of these workflows onto cloud virtual machines (VMs) remains a critical challenge, directly impacting execution time, resource utilization, and overall user satisfaction.

Traditional workflow scheduling algorithms predominantly focus on global optimization objectives, such as minimizing makespan or reducing execution costs. However, this perspective often overlooks an important dimension: task-level fairness. In real-world scenarios, individual tasks within a workflow may exhibit heterogeneous performance requirements or implicit priorities. Neglecting these aspects can result in systematic resource allocation imbalances, where certain tasks consistently receive suboptimal resources. Consequently, such tasks may fail to meet their performance expectations, even when the overall workflow completes within acceptable bounds, ultimately leading to degraded user satisfaction.

The work by Jia et al. [4] addresses this gap by proposing a novel scheduling algorithm based on Stable Matching Game Theory (SMGT), explicitly incorporating task fairness alongside traditional performance metrics. Their SM-CPTD (Stable Matching with Critical Path and Task Duplication) algorithm aims to balance workflow-level efficiency with equitable resource distribution among individual tasks, representing a significant departure from conventional scheduling approaches.

This project undertakes a comprehensive reimplementation and evaluation of the SM-CPTD algorithm. Our **primary objectives** are to:

1. Reproduce the core algorithmic components following the original paper's specifications;
2. Validate the performance improvements reported through independent experimentation;
3. Gain deeper insights into the design choices and their impact on scheduling outcomes;

We conduct our evaluation using standard scientific workflow benchmarks (*Montage*, *CyberShake*, *LIGO*, and *Epigenomics*[1]) under controlled experimental conditions, measuring performance across three key dimensions:

makespan minimization, **resource utilization**, and **task fairness**.

2. Related Work

Most workflow scheduling algorithms in cloud computing rely on heuristic or meta-heuristic techniques to optimize global objectives such as makespan or execution cost. While these approaches are effective in improving overall system performance, they generally overlook how resources are distributed among individual tasks, implicitly assuming uniform benefits across the workflow.

Only a limited number of studies address fairness explicitly, often through game-theoretic models. Fairness is typically considered at the system or VM level rather than directly at the task level. Jia et al. introduce a perspective using Stable Matching Game Theory (SMGT) for workflow scheduling, modeling task-to-VM allocation as a two-sided matching problem. Their SM-CPTD algorithm considers both makespan and task-level fairness, with fairness measured alongside other performance metrics.

3. System and Problem Formulation

3.1. Workflow Model

A scientific workflow is formally represented as a directed acyclic graph DAG $= (T, E)$, where:

- $T = \{t_0, t_1, \ldots, t_{n-1}\}$ denotes the set of n tasks.
- E represents the directed edges encoding precedence constraints and data dependencies.

Each task $t_i \in T$ is characterized by its computational size s_i, measured in million instructions (MI). The predecessors and successors of the task t_i are denoted by $pre(t_i)$ and $succ(t_i)$, respectively. In particular, tasks with no predecessors are designated as *entry tasks* (t_{entry}), while tasks with no successors are *exit tasks* (t_{exit}). A workflow may contain multiple entry and exit tasks.

A task t_i can **begin execution** only when:

- All predecessor tasks in $pre(t_i)$ have completed.
- All required data from predecessors has been transmitted to the assigned VM.

The **data transmission** requirements between tasks are captured by a matrix $TT \in \mathbb{R}^{n \times n}$, where element $TT_{i,j}$ represents the size of data that must be transmitted from task t_i to task t_j. By convention, $TT_{i,j} = 0$ if there is no dependency between tasks or if $i = j$.

3.2. Cloud Infrastructure Model

The cloud infrastructure consists of a heterogeneous set of virtual machines $V = \{\text{VM}_0, \text{VM}_1, \ldots, \text{VM}_{m-1}\}$, where each VM operates independently with distinct computational capabilities.

Each VM_k is characterized by its **processing capacity** p_k, measured in million instructions per second (MIPS). The heterogeneous nature of the infrastructure means that $p_i \neq p_j$ for $i \neq j$ in general.

Data transfer between VMs is governed by a **bandwidth** matrix $B \in \mathbb{R}^{m \times m}$, where $B(\text{VM}_k, \text{VM}_l)$ represents the network bandwidth from VM_k to VM_l. This matrix is not necessarily symmetric, reflecting realistic network asymmetries. By definition, $B(\text{VM}_k, \text{VM}_k) = 0$ since intra VM transfers incur no network overhead.

The **transmission time** for data from task t_i (executed on VM_k) to task t_j (executed on VM_l) is:

$$T_{\text{trans}}(t_i, t_j) = \begin{cases} \frac{TT_{i,j}}{B(\text{VM}_k, \text{VM}_l)} & \text{if } k \neq l \\ 0 & \text{if } k = l \end{cases} \tag{1}$$

The **execution time** of task t_i on VM_k is computed as:

$$\text{ET}(t_i, \text{VM}_k) = \frac{s_i}{p_k} \tag{2}$$

The **start time** of task t_i on VM_k is determined by:

$$\text{ST}(t_i, \text{VM}_k) = \begin{cases} 0 & t_i = t_{\text{entry}} \\ \max\limits_{t_j \in \text{pre}(t_i)} \{\text{FT}(t_j) + T_{\text{trans}}(t_i, t_j)\} & \text{otherwise} \end{cases} \tag{3}$$

Consequently, the **finish time** is:

$$\text{FT}(t_i, \text{VM}_k) = \text{ST}(t_i, \text{VM}_k) + \text{ET}(t_i, \text{VM}_k) \tag{4}$$

3.3. Problem Definition

The workflow scheduling problem consists of finding a mapping function

$$\sigma : T \to V$$

that assigns each task $t_i \in T$ to exactly one virtual machine $\text{VM}_k \in V$, such that all task dependencies are respected and the execution satisfies the resource constraints imposed by the cloud environment.

The assignment must satisfy the following constraints:

- **Precedence constraints:** A task t_i can start its execution only after all its predecessor tasks $t_j \in pre(t_i)$ have completed execution and the required data have been transferred.

- **VM availability constraint:** Each virtual machine can execute at most one task at a time. Tasks assigned to the same VM must be executed sequentially, and the start time of a task must not be earlier than the availability time of the assigned VM.
- **Task assignment constraint:** Each task must be assigned to exactly one virtual machine, and task migration during execution is not allowed.

Given these constraints, the objective of the workflow scheduling problem is to compute a feasible mapping σ that optimizes multiple performance criteria, including minimizing the workflow makespan, improving virtual machine utilization, and ensuring fairness among tasks.

3.4. Optimization Objectives and Evaluation Metrics

Our project adopts the same optimization objectives and evaluation metrics proposed in the paper in order to ensure a fair and consistent comparison. The considered metrics evaluate the performance of a scheduling solution from different perspectives, including execution efficiency, resource utilization, and fairness.

The **makespan** represents the total execution time of a workflow and is defined as the completion time of the last finished task. It is one of the most important performance indicators in workflow scheduling, as it reflects the overall efficiency of the scheduling strategy.

Formally, the makespan is defined as:

$$\text{makespan} = \max_{k \in \{0,\ldots,m-1\}} \{\text{MS}(\text{VM}_k)\} \tag{5}$$

where $\text{MS}(\text{VM}_k)$ denotes the finish time of the last task assigned to VM_k.

Minimizing the makespan is a primary optimization objective, as it directly reduces the total execution time of the workflow.

The **Scheduling length ratio (SLR)** is a normalized metric used to compare makespan values across workflows with different sizes and computational characteristics. It is defined as the ratio between the workflow makespan and a theoretical lower bound based on the critical path:

$$\text{SLR} = \frac{makespan}{\sum_{t_i \in CP} \min_{VM_j \in V} ET(t_i, VM_j)} \tag{6}$$

The SLR value is always greater than or equal to 1. Smaller SLR values indicate more efficient scheduling solutions, as they imply that the achieved makespan is closer to the theoretical minimum.

The **Average VM Utilization (AVU)** measures how effectively the available virtual machines are utilized during workflow execution. First, the **utilization** of a single virtual machine VM_k is defined as the ratio between the total execution time of tasks assigned to it and the workflow makespan:

$$VU(VM_k) = \frac{\sum_{t_i \in VM_k.\text{waiting}} ET(t_i, VM_k)}{\text{makespan}}. \tag{7}$$

The average utilization across all virtual machines is then computed as:

$$AVU = \frac{1}{m} \sum_{k=1}^{m} VU(VM_k), \tag{8}$$

where m is the number of virtual machines.

A higher AVU value indicates better utilization of computational resources and reduced idle time across the cloud infrastructure.

The **Variance of Fairness (VF)** is a metric designed to evaluate the fairness of a scheduling algorithm with respect to task execution. The metric is based on the concept of task satisfaction, which reflects how well the execution of each task meets its expected performance.

For each task (t_i), the **satisfaction** (S_i) is defined as the ratio between its actual execution time AET_i and its expected execution time EET_i (the execution time of the task on the fastest available virtual machine), formally:

$$S_i = \frac{AET_i}{EET_i} \tag{9}$$

The fairness among all tasks is then quantified by measuring the dispersion of task satisfaction values around their *average*. Let **M** be the **mean satisfaction** of all n tasks. The Variance of Fairness (VF) is defined as:

$$VF = \frac{1}{n} \sum_{i=1}^{n} (M - S_i)^2 \tag{10}$$

A smaller VF value indicates a more fair scheduling behavior, as it implies that task satisfactions are more evenly distributed across all tasks.

4. SM-CPTD Algorithm

The SM-CPTD (Stable Matching with Critical Path and Task Duplication) algorithm is a workflow scheduling strategy designed to minimize workflow makespan while ensuring fairness among tasks. It integrates stable matching game theory with two local optimization techniques based on critical path analysis and task duplication.

The algorithm operates in three main stages. First, the critical path of the workflow is identified to highlight tasks

that have a dominant impact on the overall makespan. Second, a stable matching based scheduling strategy is applied to assign tasks to virtual machines in a fair and efficient manner. Finally, a local optimization strategy based on task duplication is employed to further reduce communication delays and improve scheduling performance.

Algorithm 1 SM-CPTD

Input: DAG = (T, E), VM set V, CCR
Output: Task assignment $\sigma : T \to V$
1: $CP \leftarrow$ DCP(DAG)
2: $\sigma \leftarrow$ SMGT(DAG, V, CP)
3: $\sigma' \leftarrow$ LOTD(σ, DAG, V)
4: **return** σ'

4.1. Dynamic Critical Path (DCP)

In a directed acyclic graph (DAG), the critical path is defined as the *longest path* from the entry task to the exit task and determines the lower bound of the workflow makespan. Therefore, reducing the execution time of tasks located on the critical path is an effective way to improve overall scheduling performance.

To identify the critical path, each task is assigned a **rank value** that reflects the length of the longest path from that task to the exit node, following the strategy introduced in HEFT (Heterogeneous earliest finish time) [7]. The rank values are computed recursively in a backward manner, starting from the exit task and propagating toward the entry task, by combining the average computation time of each task with the maximum communication and execution cost of its successor tasks. Formally for a task t_i:

$$\text{rank}(t_i) = \overline{W}_i + \max_{t_j \in \text{succ}(t_i)} (\overline{c}_{i,j} + \text{rank}(t_j)) \tag{11}$$

For the exit task, the rank is defined as $\text{rank}(t_{\text{exit}}) = \overline{W}_{\text{exit}}$.

$\overline{W}_i$ denotes the average computation time of task t_i, and $\overline{c}_{i,j}$ represents the average communication time between tasks t_i and t_j, they are formally defined as follows:

$$\overline{W}_i = \frac{1}{m} \sum_{\text{VM}_k \in V} ET(t_i, \text{VM}_k) \tag{12}$$

$$\overline{c}_{i,j} = \frac{1}{m(m-1)} \sum_{\substack{\text{VM}_k, \text{VM}_l \in V \\ k \neq l}} \frac{\text{TT}_{i,j}}{B(\text{VM}_k, \text{VM}_l)} \tag{13}$$

where m is the number of virtual machines

These average cost estimates are used during the DCP rank computation, where tasks are not yet bound to specific virtual machines.

After computing the rank values of all tasks, the critical path is constructed by iteratively selecting, at each workflow level, the task with the maximum rank value. The resulting sequence of tasks represents the critical path of the DAG.

Tasks belonging to the critical path are then scheduled with higher priority and assigned to the virtual machines that minimize their completion time. Since reducing the execution time of the critical path directly contributes to minimizing the overall makespan, this scheduling strategy effectively improves workflow execution efficiency.

Algorithm 2 DCP Algorithm

1: Initialize $CP \leftarrow \emptyset$
2: Compute the rank of t_{exit}
3: **for** each task t_i in the DAG except exit tasks **do**
4: Compute the rank of t_i according to Eq.(11)
5: **end for**
6: **for** each level l in the DAG **do**
7: Select the task with the maximum rank at level l
8: Add the selected task to CP
9: **end for**

4.2. Stable Matching Game Theory (SMGT)

Stable Matching Game Theory (SMGT) provides an effective framework for achieving a balanced trade-off between fairness and efficiency in resource allocation problems. Owing to its ability to generate strict preference orders at low computational cost, SMGT has been successfully applied to a variety of matching problems, including the classical stable marriage problem and school choice systems. Recently, it has also been adopted for workflow scheduling in cloud environments.

In the context of workflow scheduling, SMGT is formulated as a two-sided matching problem involving two sets of participants: the set of tasks T and the set of virtual machines V. Each task and each VM maintains a strict preference list over the elements of the opposite set.

Preference Construction: The preference list of a task t_i is generated by ranking all VMs in ascending order of the task's estimated finish time on each VM. Similarly, the preference list of a Virtual Machine VM_k is generated by ranking tasks in ascending order of their finish time when executed on that VM. In this way, both tasks and VMs favor assignments that lead to *earlier completion*.

Hierarchical Matching Strategy: Due to the precedence constraints inherent in workflow DAGs, a hierarchical matching strategy is employed. Tasks are grouped by workflow levels, and matching is performed independently at each level. For a given level l, tasks in that level are iteratively assigned to their most preferred VMs according to their preference lists.

Algorithm 3 Algorithm SMGT

```
1: Call DCP to determine critical path.
2: for l = 0, 1, ..., level − 1 do
3:   vmPreference(l) {Generate the preference array for VMs.}
4:   task ← levelTask(l). {The tasks set of the l^th level.}
5:   Assign the critical task t in the l^th level to vm with the fastest processing speed.
6:   vm.waiting.add(t).
7:   Update(t). {Update t.ACT and t.AFT.}
8:   task.remove(t).
9:   while task.size > 0 do
10:    taskPreference(task). {Generate the preference matrix for task.}
11:    for j = 0; j < task.get(0).preference.length; j++ do
12:      u ← task.get(0).preference.get(j). {u represents a VM.}
13:      threshold(u, i). {Calculate the threshold of u.}
14:      if u.waiting.size < u.threshold then
15:        u.waiting.add(task.get(0)).
16:        Update(task.get(0)).
17:        task.remove(task.get(0)).
18:        Break.
19:      end if
20:      if u.waiting.size = u.threshold then
21:        p ← position(task.get(0), u). {Locate task.get(0) in u's preference matrix.}
22:        b ← largeTask(u). {The task's index with the largest preference value in waiting of u.}
23:        q ← position(task_b, u). {Locate task_b in preference matrix of u.}
24:        if p < q then
25:          Replace b by task.get(0).
26:          Update task.get(0).
27:          task.remove(task.get(0)).
28:          task.add(b).
29:          Break.
30:        else
31:          Continue.
32:        end if
33:      end if
34:    end for
35:  end while
36: end for
```

To prevent excessive assignment of tasks to high-performance virtual machines, a **capacity threshold** is imposed on each VM at every workflow level. The threshold limits the maximum number of tasks that can be simultaneously associated with a VM at a given level.

When a task is considered for assignment to a virtual machine VM_k at level l, two cases may occur. If the number of tasks currently waiting on VM_k is below its threshold, the task is directly accepted. Otherwise, when the waiting list of VM_k has reached its threshold, the incoming task is compared against the tasks already assigned to VM_k based on the VM's preference order.

If the new task is preferred over at least one task currently in the waiting list, it replaces the least preferred one; the displaced task is then returned to the pool of unassigned tasks and considered for assignment to its next preferred virtual machine. If the new task is not preferred over any of the tasks already assigned to VM_k, it is rejected and proceeds to its next preferred VM. This process continues until all tasks at level l are successfully assigned.

The threshold of VM_k at level l is defined as:

$$\text{threshold}(\text{VM}_k, l) = \left\lceil \frac{\sum_{v=0}^{l-1} n_v}{\sum_{j=0}^{m-1} p_j} \cdot p_k \right\rceil \tag{14}$$

where n_v denotes the number of tasks at level v, p_k represents the processing capacity of VM_k, and m is the total number of virtual machines. This formulation ensures that more powerful VMs are allocated proportionally more tasks, while maintaining balanced utilization across heterogeneous resources.

4.3. Local optimization based on task duplication (LOTD)

While global scheduling strategies aim to balance workload and optimize resource utilization, additional improvements in workflow performance can be achieved through local optimization techniques. Task duplication is a well-known approach that reduces the workflow makespan by exploiting idle time slots on virtual machines and by reducing communication delays between dependent tasks executed on different resources.

In this work, task duplication is applied as a *local optimization step*. By duplicating selected tasks, successor tasks can start earlier on their assigned virtual machines without incurring data transfer delays, thereby contributing to a reduction in the overall makespan.

To prevent excessive resource and memory consumption, task duplication is performed in a constrained manner. In particular, only entry tasks are considered for duplication. Since entry tasks have no predecessors, duplicating them does not introduce additional input communication overhead. Moreover, restricting duplication to entry tasks limits the memory footprint of the optimization process, avoiding unnecessary replication of intermediate data and ensuring that resource usage remains controlled.

To ensure that task duplication is applied safely and effectively, each candidate task must satisfy specific eligibility criteria before being duplicated on a virtual machine. These conditions guarantee that duplication reduces communication overhead without causing conflicts or excessive resource consumption.

Specifically, a task t_i can be considered for duplication on a virtual machine VM_k only if all the following conditions are satisfied:

1. **Successor-Based Constraint:** t_i can only be duplicated on VMs to which at least one of its successor tasks has been assigned. This ensures that duplication directly reduces inter VM communication cost.

2. **Idle Time Utilization:** The selected VM must have sufficient idle time to execute the duplicated instance of t_i without causing scheduling conflicts.

3. **Non-Interference Constraint:** The completion times of other tasks already assigned to VM_k must not be increased as a result of duplicating t_i.

By satisfying these constraints, LOTD leverages otherwise unused VM capacity to reduce communication delays while preserving the feasibility and stability of the existing schedule.

Algorithm 4 Algorithm LOTD

```
Call DCP and SMGT to generate pre-schedule S.
for k = 0, 1, ..., m − 1 do
    t ← the first task in the waiting list of VM_k.
    if t.AST ≠ 0 then
        minST ← ∞, where minST denotes the start time of t after the duplication of its predecessor.
        minPredecessor ← the index of the task with minST.
    end if
    for each predecessor p of t do
        if p is replicated then
            Calculate the start time of t, denoted by st.
        end if
        if st < minST then
            minPredecessor ← p
            minST ← st
        end if
    end for
    if minST < t.AST then
        Duplicate and assign the task with minPredecessor to VM_k.
        Update AST and AFT of each task in DAG.
    end if
end for
```

4.4. Computational Complexity

The SM-CPTD algorithm consists of three main phases: dynamic critical path identification, stable matching–based scheduling, and local optimization through task duplication. The computational complexity is analyzed in terms of the following parameters: $\boldsymbol{n}$ (number of tasks), $\boldsymbol{l}$ (number of workflow levels), and $\boldsymbol{m}$ (number of virtual machines).

The critical path identification phase computes a rank value for each task based on its successors. In the worst case, this process involves comparisons among multiple task pairs, resulting in a time complexity of $O(n^2)$.

The stable matching based scheduling phase represents the most computationally expensive component. Scheduling is performed level by level, and for each of the l workflow levels, tasks are matched to virtual machines using a stable matching process. While the best case complexity is $O(nm)$, repeated rejections and cascading reassignments may occur in the worst case, leading to a time complexity of $O(n^2)$ per level. As this phase dominates the overall cost, it determines the algorithm's asymptotic behavior.

The task duplication phase evaluates potential duplications by comparing tasks against available virtual machines, yielding a time complexity of $O(nm)$.

Therefore, the overall time complexity of SM-CPTD is:

$$O(n^2 \cdot l).$$

This result indicates that, in the worst case, SM-CPTD scales **quadratically** with the **number of tasks** and **linearly** with the **number of workflow levels**.

5. Implementation and Experimental Setup

Our implementation is developed in **Java**, chosen for its robust type system, mature ecosystem for enterprise-grade applications, and strong performance characteristics for compute intensive scheduling algorithms. Python is used for visualization and statistical analysis.

Table 1: Software and libraries used for implementation and analysis

	Technology / Library	Purpose
Core Implementation	Java 23.0.2 (OpenJDK Temurin)	Main scheduling engine
	Java Collections Framework	DAGs, task queues, VM management
	Standard Java I/O	XML parsing, JSON/CSV output
Visualization & Analysis	Python 3	Data analysis and visualization
	NumPy (2.3+)	Numerical operations
	Pandas (2.3+)	Data manipulation and aggregation
	Matplotlib (3.10+)	Plot generation
	Seaborn	Statistical visualizations
	NetworkX	DAG visualization

5.1. System Architecture

The implementation follows a modular architecture, designed to maintain a clear separation of concerns between

the different components of the workflow scheduling system. The main modules and their responsibilities are described below:

1. **SMCPTD.java**: Contains the main orchestrator implementing the three-phase SM-CPTD pipeline (DCP $\rightarrow$ SMGT $\rightarrow$ LOTD).
2. **DCP.java**: Implements the Dynamic Critical Path algorithm with recursive rank computation and critical path identification.
3. **SMGT.java**: Contains the Stable Matching Game Theory scheduler implementing level-by-level task assignment using VM and task preference lists while ensuring stability and threshold constraints.
4. **LOTD.java**: Implements the task duplication optimizer that reduces communication overhead through strategic entry task replication.
5. **Metrics.java**: Contains functions for computing performance metrics, including Scheduling Length Ratio (SLR), Average VM Utilization (AVU), Variance Factor (VF), and execution time, as well as utilities for validating and analyzing scheduling results.
6. **DataLoader.java**: Handles workflow and cloud environment data loading from CSV files and XML-based scientific workflow specifications (Pegasus format).
7. **ExperimentRunner.java:** Orchestrates experimental runs across different workflow configurations and communication-to-computation ratios (CCR), providing statistical analysis and performance evaluation capabilities.

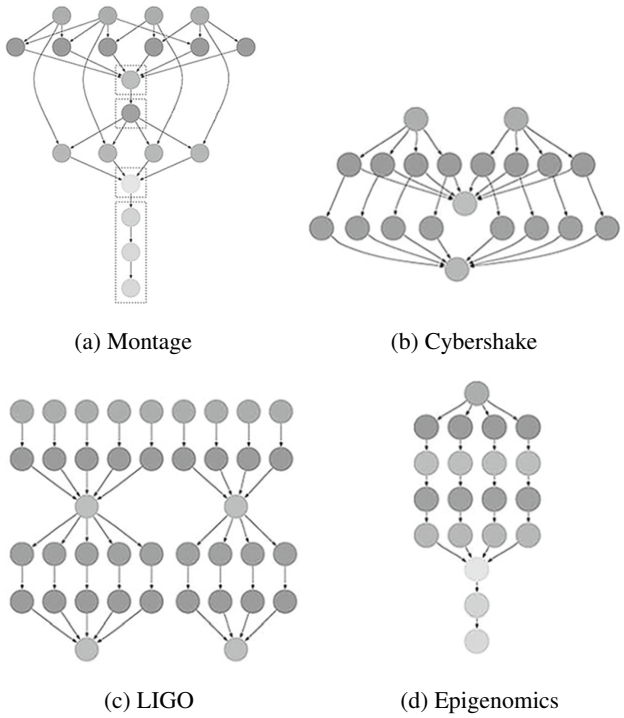
This modular design facilitates maintainability, allows independent development and testing of components, and supports extensibility for future enhancements or alternative scheduling strategies.

5.2. Workflow Generation

To ensure reproducibility and consistency with commonly used benchmarks in the literature, the workflow structures adopted in this study are generated from XML description files obtained from a public GitHub repository [1]. These XML files define the task graphs (DAGs), including task dependencies and structural patterns, and are widely used in cloud workflow scheduling research.

The following real-world scientific workflows are considered:

- **Montage**: An astronomy application for constructing image mosaics. Its structure consists of parallel pipelines with multiple join operations 1a.
- **CyberShake**: A seismic hazard characterization application. It exhibits a multi-level fork–join structure with complex task dependencies 1b.
- **LIGO**: A workflow for gravitational wave detection. Its structure follows a pipeline model with periodic synchronization points 1c.
- **Epigenomics**: A genomics data processing workflow. It is characterized by a mostly parallel structure with a final aggregation phase 1d.



(a) Montage (b) Cybershake

(c) LIGO (d) Epigenomics

Figure 1: Scientific workflows considered in this work[6]

.

5.3. Experiment Setup

The experimental configuration adopted in this study is summarized in Table 2.

Table 2: Simulation parameters and sampling distributions

Parameter	Range	Distribution
Task size (MI)	$500 \sim 700$	$\mathcal{U}(500, 700)$
VM processing capacity (MIPS)	$10 \sim 20$	$\mathcal{U}(10, 20)$
Bandwidth (Mbps)	$20 \sim 30$	$\mathcal{U}(20, 30)$
CCR	$0.4 \sim 2.0$	fixed per experiment
Number of VMs	5 / 10 / 50	fixed by workflow scale

All continuous parameters, except CCR, are sampled from uniform distributions within the reported ranges to

generate heterogeneous yet unbiased experimental scenarios. For CCR, a single fixed value is selected for each experiment within the specified range to simulate different levels of communication intensity.

Task computational sizes are drawn from $\mathcal{U}(500, 700)$ MI. Communication costs between two dependent tasks t_i and t_j are computed using the Communication-to-Computation Ratio (CCR) as:

$$TT_{i,j} = s_{t_i} \times \text{CCR} \tag{15}$$

where CCR is set to a fixed value for each experimental scenario, allowing the simulation of both computation intensive and communication intensive workflows.

VM processing capacities and network bandwidths are sampled from $\mathcal{U}(10, 20)$ MIPS and $\mathcal{U}(20, 30)$ Mbps, respectively. The number of virtual machines is fixed according to the workflow scale, with 5, 10, and 50 VMs used for small, medium, and large workflows.

To account for the stochastic nature of the sampling distributions and ensure statistical reliability, each experimental scenario is executed for **10 independent runs**. The simulation framework optionally allows the specification of a random seed. When provided, all random values are deterministically generated, ensuring full reproducibility of the experiments; otherwise, independent runs are performed with different random initializations. Additionally, to support visual inspection and analysis of the scheduling flow, the framework generates **Gantt chart** representations in JSON format for each execution, with optional visualization support.

All algorithms are implemented in Java (OpenJDK 21.0.9) using Visual Studio Code (version 1.107.1) on an AMD Ryzen 5 7600X 6-Core Processor with 32GB RAM, running on Ubuntu 24.04.2 LTS operating system.

6. Experimental Results

This section presents the experimental evaluation of the SM-CPTD algorithm. All experiments were conducted using the evaluation metrics defined in Section 3.4 and follow the experimental methodology proposed by Jia et al.

The goal of the evaluation is to assess scheduling performance in terms of makespan, Scheduling Length Ratio (SLR), Average VM Utilization (AVU), and fairness.

Unless otherwise stated, experiments are performed under three workflow scale configurations, which are used consistently throughout this section:

- **Small-scale**: 50 tasks executed on 5 virtual machines;
- **Medium-scale**: 100 tasks executed on 10 virtual machines;
- **Large-scale**: 1000 tasks executed on 50 virtual machines.

6.1. Makespan and SLR Analysis

To evaluate the sensitivity of scheduling performance to communication overhead, a **CCR** (Communication-to-Computation Ratio) **sensitivity analysis** was conducted on the four real-world scientific workflows: *Cybershake*, *Epigenomics*, *LIGO*, and *Montage*. For each workflow scale, the CCR value was varied within the range 0.4 to 2.0 with a 0.2 step.

Across all three plots (2, 3, and 4), SLR exhibits an increasing trend as CCR grows. This indicates that the achieved makespan moves further from the theoretical critical path lower bound as communication overhead increases, reflecting the scheduler's reduced ability to approach optimal performance when communication costs dominate.

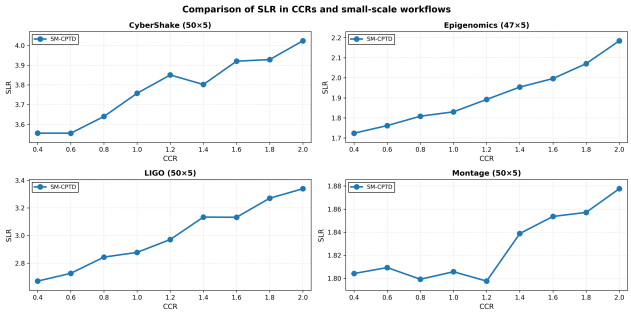


Figure 2: SLR vs CCR for small-scale workflows.

In **small-scale workflows** (figure 2), the SLR increase is more modest and occasionally irregular, particularly for Montage. With limited parallelism opportunities and fewer tasks to schedule, the impact of individual task placement decisions becomes more pronounced, leading to less predictable scheduling behavior as communication overhead varies.

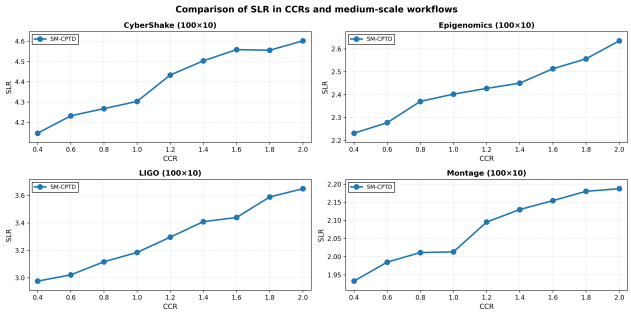


Figure 3: SLR vs CCR for medium-scale workflows.

Medium-scale workflows (figure 3) show a similar upward trend, but with slightly higher absolute SLR values, indicating that the gap between achieved makespan and the critical path lower bound widens as problem size increases and communication overhead becomes more significant.

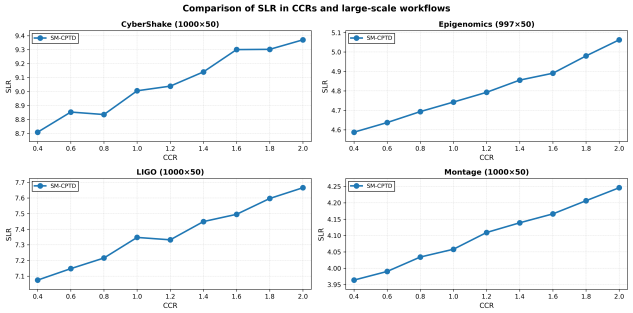


Figure 4: SLR vs CCR for large-scale workflows.

In **large-scale workflows** (figure 4), this growth is more pronounced and relatively smooth, with *Cybershake* and *LIGO* achieving the highest absolute SLR values. These results highlight that communication costs significantly affect performance at scale.

The observed differences among workflows can be further explained by their underlying DAG **structures**.

Cybershake, with its broad and shallow DAG, features multiple synchronization points where parallel branches converge. Higher CCR values amplify communication delays at these points, creating bottlenecks that elevate SLR.

LIGO, characterized by a wide, highly parallel structure, allows many tasks to execute concurrently but introduces numerous data dependencies across this broad execution front. As communication costs rise, these dependencies collectively slow execution relative to the critical path, resulting in high SLR.

Montage, in contrast, follows a sequential pipeline with a dominant critical path. In this case, communication overhead affects tasks more uniformly along the chain, and since increases in communication costs are partially captured in both the numerator (makespan) and the denominator (critical path bound) the resulting SLR shows a more modest growth.

Epigenomics presents a more balanced structure with moderate parallelism, allowing some overlap of computation and communication without the severe synchronization bottlenecks of wider workflows, leading to smooth and moderate SLR increases across CCR values.

Overall, these results demonstrate that workflows with wide parallel structures (*CyberShake*, *LIGO*) experience the largest gap between achieved makespan and theoretical minimum as communication overhead increases, while workflows with more sequential structures (*Montage*, *Epigenomics*) maintain performance closer to the critical path bound.

In addition to the CCR sensitivity analysis, we also investigated the effect of **varying the number of available virtual machines** (VMs) on scheduling performance (figure 5). For each workflow, the number of VMs was systematically increased from 30 to 70 while keeping other parameters, including CCR, fixed.

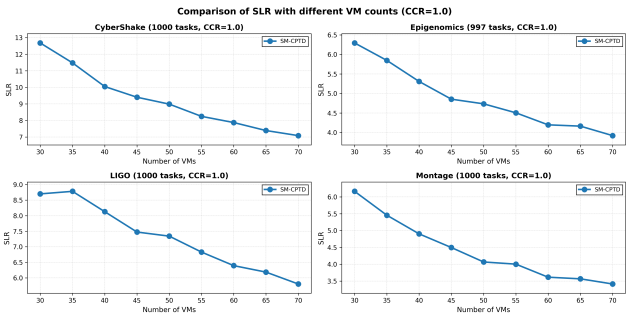


Figure 5: SLR vs number of virtual machines (large-scale workflows, CCR = 1.0).

Across all four scientific workflows, a clear inverse relationship is observed between the number of VMs and the Scheduling Length Ratio (SLR). As the number of VMs increases, the SLR steadily decreases, indicating that the achieved makespan approaches closer to the critical path lower bound. This trend demonstrates that the SM-CPTD algorithm effectively exploits additional parallelism provided by larger resource pools, achieving more efficient task scheduling and reducing idle time.

The **magnitude** of this improvement varies by workflow structure. *LIGO* and *CyberShake* benefit more from additional VMs, as their wide structures provide numerous opportunities for concurrent execution (value decrase of 4-6). In contrast, workflows with more sequential characteristics, *Montage* and *Epigenomics*, exhibit smaller SLR reductions (2-3), since their critical paths fundamentally limit the benefits of additional parallelism; nevertheless, these workflows still achieve the lowest absolute SLR values overall.

6.2. VM Utilization Analysis

The Average VM Utilization (AVU) metric is used to evaluate how effectively the available computational resources are exploited during workflow execution. As defined in Section 3.4, AVU measures the fraction of the workflow makespan during which virtual machines are actively executing tasks, with higher values indicating reduced idle time and better resource utilization.

Figures 6, 7, and 8 illustrate the impact of **increasing Communication-to-Computation Ratio** (CCR) values on AVU across small, medium, and large scale workflows (same range of CCR values used in the SLR vs CCR analysis).

Across all workflow scales, AVU exhibits a decreasing trend as CCR increases, reflecting the growing impact of

communication delays on overall resource utilization. As communication becomes more dominant, virtual machines spend a larger fraction of the makespan waiting for data transfers rather than executing tasks, resulting in increased idle time and lower AVU values.

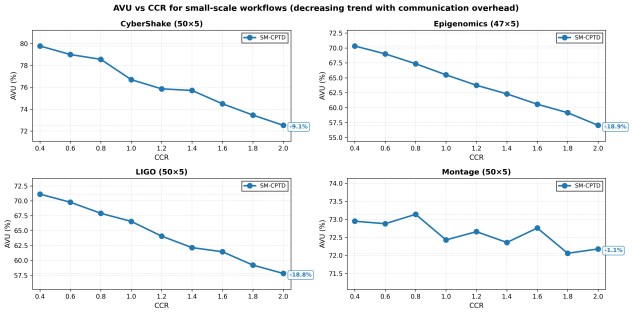


Figure 6: AVU vs CCR for small-scale workflows.

Small-scale workflows (figure 6) show the most pronounced sensitivity to CCR variations. The absolute AVU degradation is substantial across all workflows, with *Epigenomics* and *LIGO* experiencing degradation of up to approximately 19%.

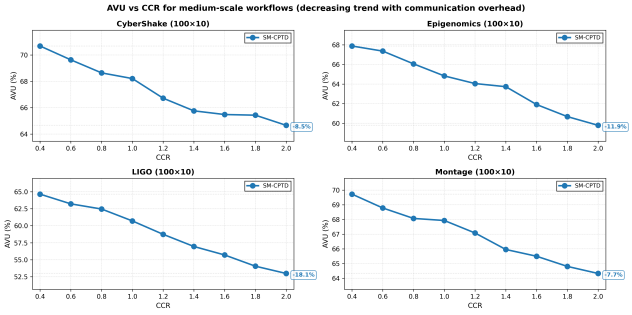


Figure 7: AVU vs CCR for medium-scale workflows.

Medium-scale workflows (figure 7) experience a less pronounced decline in AVU (*LIGO* and *Epigenomics* around 10–18%).

Large-scale workflows (figure 8), demonstrate more moderate AVU reduction (approximately 5-8%) across the CCR range.

The main observation is that the magnitude of AVU degradation varies significantly with workflow scale. Small-scale workflows exhibit the sharpest percentage decline in AVU as CCR increases, while large-scale workflows show more moderate degradation. This suggests that larger workflows with more tasks provide better opportunities to amortize communication costs and maintain VM utilization.

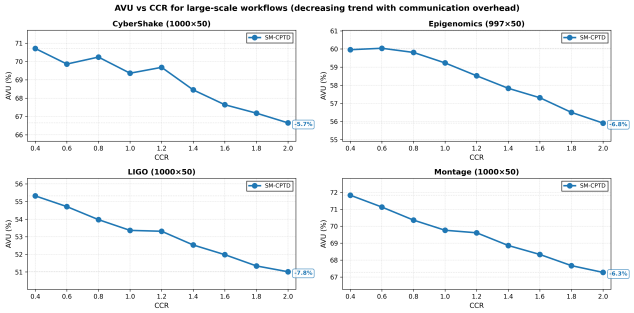


Figure 8: AVU vs CCR for large-scale workflows.

Overall, these results confirm that increasing CCR negatively impacts resource utilization across all workflows and scales, with the magnitude of this impact being most severe at smaller scales where limited parallelism opportunities amplify the effect of communication delays.

In addition to analyzing the impact of communication intensity, we investigated how the availability of computational resources influences virtual machine utilization **by varying the number of virtual machines** while keeping CCR fixed at 1.0.

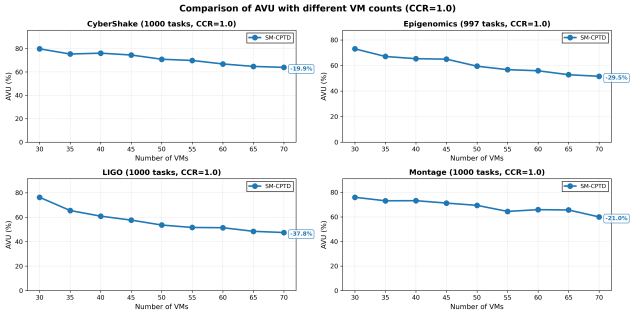


Figure 9: AVU vs number of virtual machines (large-scale workflows, CCR = 1.0).

Figure 9 shows that, across all workflows, the Average VM Utilization (AVU) consistently decreases as the number of available virtual machines increases from 30 to 70. This trend is a direct consequence of the AVU definition: when a fixed computational workload is distributed over a larger set of VMs, the execution time per VM decreases relative to the makespan, leading to increased idle periods and lower average utilization.

These results demonstrate that while additional computational resources can reduce makespan (as shown in the SLR analysis), the efficiency of resource utilization inevitably decreases as the number of VMs grows beyond the workflow's inherent parallelism capacity.

6.3. Fairness Analysis

In addition to makespan oriented metrics, fairness is evaluated using VF, which measures the dispersion of task satisfaction values across all scheduled tasks. The VF values are analyzed for **large-scale workflows**, 1000 tasks executed on 50 virtual machines with CCR equal to 1.0.

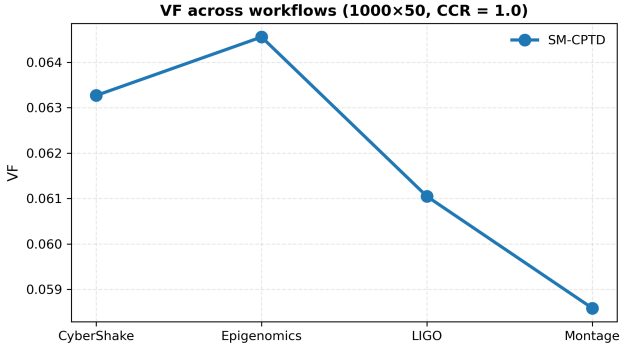


Figure 10: Variance of Fairness across workflows (large-scale, 50 VMs, CCR = 1.0).

As shown in Figure 10, all workflows exhibit VF values well below 1, which is consistent with the mathematical definition of the metric. In a system with homogeneous virtual machines and a large number of tasks, satisfaction values are expected to cluster tightly around the mean, resulting in a limited dispersion and therefore a low variance.

Minor differences among workflows can be observed; however, these variations remain small in absolute terms and may fluctuate across different runs. This behavior is expected, as VF is particularly sensitive when satisfaction values are highly concentrated: small changes in task to VM assignment or dependency resolution can slightly alter the dispersion without indicating a meaningful difference in overall fairness.

It is worth noting that the original SM-CPTD paper reports a figure labeled as "Variance of Fairness" for the same experimental configuration, with values ranging approximately from 1.25 to 1.47. Such values are mathematically inconsistent with the definition of VF, since by definition, the satisfaction $S_i \geq 1$ for all tasks, and a variance exceeding 1 would imply deviations from the mean greater than 100%, which is incompatible with the constraints of the scheduling model. We therefore hypothesize that the reported figure actually represents the average task satisfaction, and that the label may be incorrect.

To clarify this ambiguity and ensure full transparency, we explicitly report the average task satisfaction in Figure 11.

The figure shows that the magnitude of the average task satisfaction values is consistent with those reported in the original paper, reinforcing the interpretation that the previously labeled "VF" results correspond to average satisfaction.

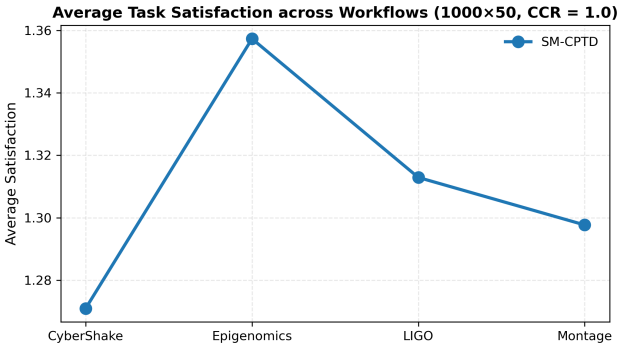


Figure 11: Average task satisfaction across workflows (large-scale, CCR = 1.0).

6.4. Ablation Study

Finally, an ablation study was conducted to evaluate the individual contribution of each component of the SM-CPTD pipeline. Four algorithmic variants were evaluated on large-scale workflows (approximately 1000 tasks, 50 virtual machines, CCR = 1.0):

1. SMGT only (baseline)
2. DCP + SMGT
3. SMGT + LOTD
4. DCP + SMGT + LOTD (full SM-CPTD)

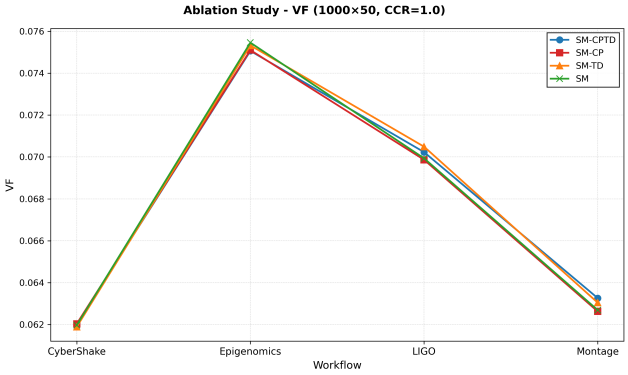


Figure 12: Ablation study results: VF.

Figure 12 reports the ablation study results for the Variance of Fairness (VF) across different workflows and scheduling variants. Overall, VF values remain consistently low for all configurations, confirming that fairness is largely preserved regardless of the specific algorithmic components enabled. The observed differences are small in absolute

magnitude and should therefore be interpreted as indicative trends rather than as statistically significant separations.

From the algorithmic perspective, the ablation study does not reveal a clear or stable separation among the different scheduling variants, this behavior is expected, as VF values are *consistently low* and therefore *highly sensitive* to small fluctuations. As a result, VF cannot be reliably used to favor one algorithmic variant over another in this setting. Instead, it should be interpreted primarily as a sanity check confirming that all variants preserve a high level of fairness.

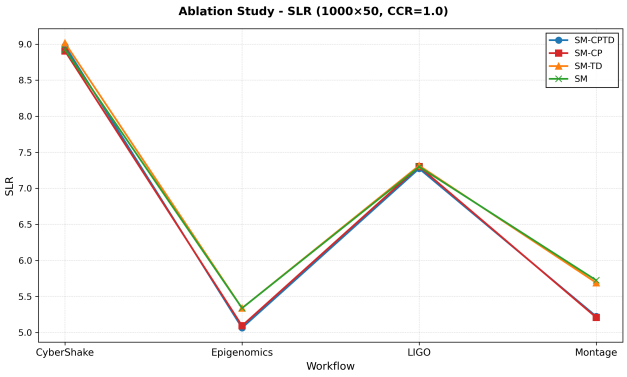


Figure 13: Ablation study results: SLR.

The ablation results above (figure 13) show that SM-CPTD consistently achieves the lowest SLR across all workflows, confirming the benefit of jointly exploiting critical-path awareness and task distribution. Removing the CP component (SM-TD) causes the largest degradation, especially for *Epigenomics* and *Montage*, indicating that these workflows are strongly CP dominated. Eliminating TD (SM-CP) leads to almost none performance loss, suggesting that TD mainly refines load balancing rather than driving scheduling decisions. Overall, CP is the primary contributor to performance, while TD provides complementary improvements that stabilize results across heterogeneous workflows.

The AVU ablation results (Figure 14) show that SM-CPTD consistently achieves the highest resource utilization across all evaluated workflows, highlighting the effectiveness of jointly integrating critical path awareness and task distribution mechanisms. When either component is removed, task duplication in SM-CP or critical path optimization in SM-TD, a noticeable degradation in AVU is observed across all workflows. The degradation becomes even more pronounced when both components are removed. These results confirm that the two components play complementary roles: critical path optimization primarily enhances global scheduling efficiency, while task distribution improves VM-level resource utilization.

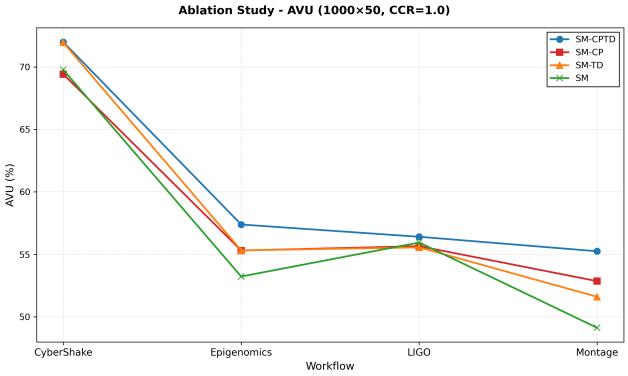


Figure 14: Ablation study results: AVU.

Overall, the ablation studies highlights the impact of algorithmic design choices on performance: SM-CPTD, which jointly incorporates critical-path awareness and task distribution, achieves the lowest Scheduling Length Ratio (SLR) and the highest average VM utilization across all workflows. Removing either component results in systematic performance degradation, confirming that these features are complementary and essential for robust task scheduling.

Taken together, these results demonstrate that SM-CPTD provides a highly effective and reliable scheduling strategy, simultaneously **maintaining fairness** and **maximizing execution efficiency**, and offering clear advantages over its reduced variants.

7. Conclusion

This work presented a complete reimplementation and experimental evaluation of the SM-CPTD workflow scheduling algorithm. The results confirm the effectiveness of the proposed approach in improving makespan related metrics, resource utilization, and fairness across different workflow scales and execution scenarios. The ablation study also conclusively shows that both the critical-path (CP) and task duplication (TD) components contribute essential and complementary improvements, with CP driving global efficiency and TD refining load balancing.

During the reimplementation process, several algorithmic and methodological aspects required careful interpretation, as some components of the original paper were not fully specified. In particular, ambiguities in the description of the scheduling procedure and inconsistencies in the reported fairness metric demanded additional analysis and validation.

Despite these challenges, the strong correlation between our experimental trends and those reported in the original paper validates the core contribution of SM-CPTD. This work underscores the importance of precise metric defini-

tions, comprehensive algorithmic documentation, and reproducible experimental frameworks in advancing workflow scheduling research. The complete source code and experimental data are publicly available to facilitate further investigation and extension of this work.

References

[1] adnanetalha. DAX-file: workflows dax format. https://github.com/adnanetalha/DAX-file. Accessed 2025/2026.

[2] Cappetti99, chiarapepp. stable-matching-game-theory. https://github.com/Cappetti99/stable-matching-game-theory. Accessed 2025/2026.

[3] E. Deelman, K. Vahi, M. Rynge, R. Mayani, R. Ferreira da Silva, G. Papadimitriou, and M. Livny. The evolution of the pegasus workflow management software. *Computing in Science & Engineering*, 21(4):22–36, 2019.

[4] Z. hong Jia, L. Pan, X. Liu, and X. jun Li. A novel cloud workflow scheduling algorithm based on stable matching game theory. *The Journal of Supercomputing*, 77(10):11597–11624, 2021.

[5] OpenAI. ChatGPT: Large Language Model. https://chat.openai.com/, 2025. Accessed 2025/2026.

[6] Pegasus WMS. Workflow gallery. https://pegasus.isi.edu/workflow_gallery/, 2025. Accessed: 2025-11-16.

[7] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.