Using Simple PID-inspired Controllers for Online Resilient Resource Management of Distributed Scientific Workflows

Using Simple PID-inspired Controllers for Online Resilient Resource Management of Distributed Scientific Workflows

Rafael Ferreira da Silva[a,*], Rosa Filgueira[b,c], Ewa Deelman[a], Erola Pairo-Castineira[d,e], Ian M. Overton[d,e,f], Malcolm P. Atkinson[c]

[a] *University of Southern California, Information Sciences Institute, Marina del Rey, CA, USA*
[b] *British Geological Survey, Lyell Centre, Edinburgh EH14 4AP, UK*
[c] *School of Informatics, University of Edinburgh, Edinburgh EH8 9LE, UK*
[d] *MRC Institute of Genetics and Molecular Medicine, University of Edinburgh, Edinburgh, UK*
[e] *Usher Institute of Population Health Sciences and Informatics, University of Edinburgh, Edinburgh, UK*
[f] *Current address: Centre For Cancer Research and Cell Biology, Queen's University Belfast, Belfast, UK*

Abstract

Scientific workflows have become mainstream for conducting large-scale scientific research. As a result, many workflow applications and Workflow Management Systems (WMSs) have been developed as part of the cyberinfrastructure to allow scientists to execute their applications seamlessly on a range of distributed platforms. Although the scientific community has addressed this challenge from both theoretical and practical approaches, failure prediction, detection, and recovery still raise many research questions. In this paper, we propose an approach inspired by the control theory developed as part of autonomic computing to predict failures before they happen, and mitigated them when possible. The proposed approach is inspired on the *proportional-integral-derivative* controller (PID controller) control loop mechanism, which is widely used in industrial control systems, where the controller will react to adjust its output to mitigate faults. PID controllers aim to detect the possibility of a non-steady state far enough in advance so that an action can be performed to prevent it from happening. To demonstrate the feasibility of the approach, we tackle two common execution faults of large scale data-intensive workflows—data storage overload and memory overflow. We developed a simulator, which implements and evaluates simple standalone PID-inspired controllers to autonomously manage data and memory usage of a data-intensive bioinformatics workflow that consumes/produces over 4.4TB of data, and requires over 24TB of memory to run all tasks concurrently. Experimental results obtained via simulation indicate that workflow executions may significantly benefit from the controller-inspired approach, in particular under online and unknown conditions. Simulation results show that nearly-optimal executions (slowdown of 1.01) can be attained when using our proposed method, and faults are detected and mitigated far in advance of their occurrence.

Keywords: Scientific workflows, Fault detection and handling, Resilient Big Data workflows, Autonomic computing

1. Introduction

Scientists want to extract the maximum information out of their data—which are often obtained from scientific instruments and processed in large-scale distributed systems. Today's computational and data science applications may comprise thousands of computational tasks and process large datasets (from remote sensors, instruments, etc.), which are often distributed and stored on heterogeneous resources. Scientific workflows are a mainstream solution to process large-scale scientific computations in distributed systems, and have supported traditional and breakthrough research across several domains [1]. As a result, many workflow applications and Workflow Management Systems (WMSs) have been developed as part of the cyberinfrastructure to allow scientists to execute their applications seamlessly on a range of distributed platforms [2, 3].

In spite of impressive achievements today, failure prediction, detection, and recovery remain a major challenge in workload management in distributed systems, both at the application and resource levels. Failures affect the makespan of the applications, and therefore the productivity of the scientists that depend on the power of distributed computing to do their work. Throughout the remainder of this paper, a failure may represent an inconsistent state of the system, which may be an actual fault, poor performance, or a constraint violation.

Unsurprisingly, failure detection and handling for distributed scientific applications has been the subject of significant effort, both from practitioners and from researchers [4, 5, 6, 7, 8, 9, 10, 11, 12]. However, most of these approaches do not aim to prevent faults, but to mitigate their impact. They also make strong assumptions about resource and application characteristics, so that resource management problems are rendered tractable. But the resulting solutions may perform poorly in practice when un-

*Corresponding address: USC Information Sciences Institute, 4676 Admiralty Way Suite 1001, Marina del Rey, CA, USA, 90292
Email addresses: rafsilva@isi.edu (Rafael Ferreira da Silva), rosa@bgs.ac.uk (Rosa Filgueira), deelman@isi.edu (Ewa Deelman), Erola.Pairo-Castineira@igmm.ed.ac.uk (Erola Pairo-Castineira), ian.overton@qub.ac.uk (Ian M. Overton), Malcolm.Atkinson@ed.ac.uk (Malcolm P. Atkinson)

expected events (e.g., network glitches, external load, etc.) occur. Therefore, there is a lack of realistic solutions, which is the major cause for the discrepancy between proposed theoretical techniques and methods, and their practical application [13].

In this work, we target the resource management challenge to attain "resilience" in executions of large-scale scientific workflows on distributed infrastructures. More specifically, we investigate how the principles of the *proportional-integral-derivative* controller (PID controller) control loop mechanism, which is widely used in industrial systems, can be applied to predict and prevent failures in end-to-end workflow executions across distributed, heterogeneous computational environments. The basic idea behind a PID controller is as follows: read data from a sensor, then compute the desired actuator output by calculating proportional (P), integral (I), and derivative (D) responses and summing those three components to compute the output. Each of the components can often be interpreted as the present error (P), the accumulation of past errors (I), and a prediction of future errors (D), based on the current rate of change. The main advantage of using the principles of a PID controller is that the control loop mechanism progressively monitors the evolution of the workflow execution, detecting possible faults before they occur, and when needed performs actions that lead the execution to a steady-state.

The main contributions of this paper include:

1. A process for resilient management of computing resources, which uses the concepts of PID controllers to prevent and mitigate two major problems of the Big Data era: data storage overload and memory overflow;
2. The characterization of a bioinformatics workflow, which consumes/produces over 4.4TB of data, and requires over 24TB of memory;
3. An evaluation via simulation to demonstrate the feasibility of the proposed approach using simple PID-inspired controllers; and
4. A performance optimization study to tune the parameters of the control loop to provide nearly-optimal workflow executions, where faults are detected and handled far in advance of their occurrence.

Although PID controllers are commonly used in closed environments where a steady-state can be reached and maintained, the preliminary evaluation study conducted in this work demonstrates their ability to tackle inconsistent states of a dynamic distributed system by limiting the oscillation analysis to short intervals. In [14], we have presented a first evaluation of the use of a control loop approach, inspired by PID controllers, to prevent faults in online distributed systems. In this work, we detail the model of our resilient resource management process, and extend the previous analysis by further evaluating the behaviors of livelocks and the characteristics of the controller response input value.

This paper is structured as follows. Section 2 gives an overview of related work. Section 3 presents the general resilient resource management process, which is inspired by the principles of PID controllers, while Section 4 describes the two types of faults evaluated in this paper. The experimental evaluation is presented in Section 5, and Section 6 presents a study to tune the gain parameters of the PID-inspired controllers to improve error detection and handling. Section 7 summarizes our results and identifies future work.

2. Related Work

Several offline strategies and techniques were developed to detect and handle failures during scientific workflow executions [4, 5, 6, 7, 8, 9, 15]. Autonomic online methods were also proposed to cope with workflow failures at runtime, for example by providing checkpointing [16, 17, 18], provenance [17, 19], task resubmission [10, 11], and task replication [8, 12], among others. However, these systems do not aim to prevent faults, but mitigate their impact, and although task replication may increase the probability of having a successful execution on another computing resource, it should be used sparingly to avoid overloading the execution platform [20]. The above systems also make strong assumptions about resource and application characteristics. A recent survey on fault-tolerance mechanisms for task clustering [21], highlights approaches to cope with tasks exhibiting low performance, however most of the techniques also assume that accurate estimates of task requirements are available.

Although several works address task requirement estimations based on provenance data [22, 23, 24, 25], accurate estimations are still challenging, and may be specific to a certain type of application. In [26], a prediction algorithm based on machine learning (Naïve Bayes classifier) is proposed to identify faults before they occur, and to apply preventive actions to mitigate the faults. Experimental results show that faults can be predicted with up to 94% accuracy; however, that approach is tied to a small set of applications, and it is assumed that the application requirements do not change over time. In previous work, we proposed an autonomic method described as a MAPE-K loop to cope with online non-clairvoyant workflow execution faults on grids [27, 28], where unpredictability is addressed by using a-priori knowledge extracted from execution traces to identify severity levels of faults, and apply a specific set of actions. Although this is the first work on self-healing of workflow executions under online and unknown conditions (e.g., workload unawareness, external load, etc.), experimental results on a real platform show an important improvement of the QoS delivered by the system. However, the method does not prevent faults from happening (actions are performed once faults are detected). In this paper, we revisit the MAPE-K loop concept to enable fault detection and handling; however, without depending on reliable estimates obtained from a-priori knowledge. In [29], a machine learning approach based on inductive logic programming is proposed for fault prediction and diagnosis in grids. This approach is limited to small-scale applications with a few parameters—the number of rules may exponentially increase as the number of tasks in a workflow or the handled parameters increases. Feedback loops have also been proposed to tackle failures in workflow systems [30], however,

no mechanisms are available to prevent an unrecoverable fault from happening.

To the best of our knowledge, this is the first work that uses the concepts of PID controllers to mitigate faults in scientific workflow executions under online and unknown conditions.

3. General Resilient Resource Management Process

Automating fault prevention, detection, and handling is challenging for two reasons. First, the problem is *online* by nature because no reliable user activity prediction can be assumed (e.g., task runtime estimates are not accurate). New workloads may arrive at any time, and resources may leave at any time. Therefore, the decisions, actions, and considered metrics have to remain simple and to yield good results while the application is still executing. Second, it is *unpredictable* due to the lack of reliable application and platform models [22], and due to the lack of information about the performance of computing and network resources in production environments. Hence, platform and application models also have to remain simple, and adapt to the dynamic behavior of the system. In this work, we present a novel resilient resource management process for autonomous detection and handling of possible-future faults in scientific workflow executions, under online and unpredictable conditions. The process uses the MAPE-K loop principle as a basis for constantly performing online monitoring, analysis, planning, and execution of a set of preventative and/or corrective actions (Figure 1). In this process, when an event occurs during the workflow execution (e.g., job completion, failures, or timeouts), an analysis event is triggered in the controller (inspired by PID controllers). If the controller detects that the system is moving towards an unstable state, the controller will notify a decision agent process that may trigger actions to prevent or mitigate faults.

3.1. Overview of PID Controllers

The *proportional-integral-derivative* controller (PID controller) [31, 32] control loop mechanism is key to address faults under online and unknown conditions. In industrial systems, a PID controller is typically used to assess the state of a single fine-grained measurement (e.g., temperature, pressure, acceleration, etc.) to improve the efficiency of the control loop. Although "one-fit-all" metrics could be handled by PID controllers, they usually represent complex mathematical models

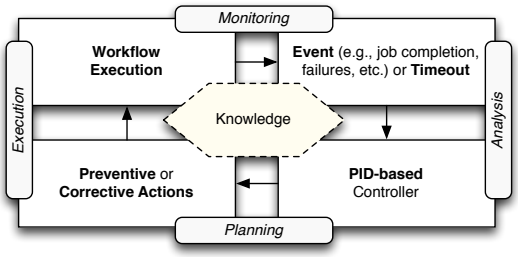


Figure 1: Overview of the Resilient Resource Management Process based on the MAPE-K loop.

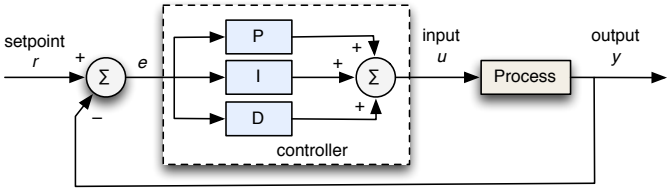


Figure 2: General diagram of closed loop systems with an ideal PID controller (based on error feedback).

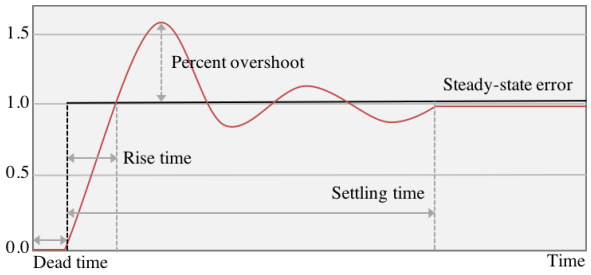


Figure 3: Response of a typical PID closed loop system.

in which the output signal would be hard to interpret, and the decision problem of what preventive or corrective action to perform becomes more difficult. In this work, we follow the same fine-grained approach, where we design and implement an autonomous process for resilient resource management for WMSs, which is inspired by PID controllers (named in this paper PID-inspired controllers). We then define controllers for different metrics at different levels (e.g., memory or disk usage per node, shared file system usage per platform, etc.). In such scenarios, the PID-inspired controller aims at detecting the possibility of a fault far enough in advance that an action can be performed to prevent it from happening. Figure 2 shows a general ideal PID control system loop. The *setpoint* (or reference signal) is the desired or command value for the process variable. The control system algorithm uses the difference between the output (process variable) and the *setpoint* to determine the desired actuator input to drive the system.

The control system performance is measured through a step function as a *setpoint* command variable, and the response of the process variable. The response (output) is quantified by measuring defined waveform characteristics as shown in Figure 3. Rise time is the amount of time the system takes to go from about 10% to 90% of the *steady-state*, or final, value. Percent overshoot is the amount that the process variable surpasses the final value, expressed as a percentage of the final value. Settling time is the time required for the process variable to settle to within a certain percentage (commonly 5%) of the final value. Steady-state error is the final difference between the process variable and the *setpoint*. Dead time is a delay between when a process variable changes, and when that change can be observed.

The input/output relation for an ideal PID controller with error feedback is defined as follows:

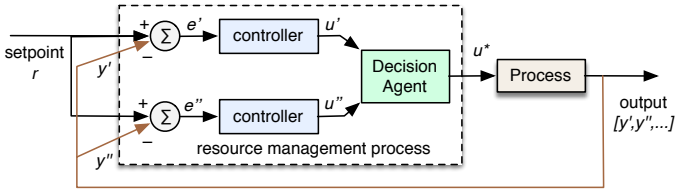


Figure 4: Overview of the resilient resource management process.

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt}, \quad (1)$$

where K_p is the proportional gain constant, K_i is the integral gain constant, K_d is the derivative gain constant, and e is the error defined as the difference between the *setpoint* and the process variable value.

Tuning the proportional (K_p), integral (K_i), and derivative (K_d) gain constants is challenging and a research topic in itself. Therefore, in this paper we initially assume $K_p = K_i = K_d = 1$ for the sake of simplicity and to demonstrate the feasibility of the process, and then we use the *Ziegler-Nichols* closed loop method [33] for tuning the PID controllers (see Section 6).

3.2. Model and Design

Although the PID controller shown in Figure 2 represents an idealized controller and several modifications are often required to obtain a controller that is practically useful [32]. We argue that the concepts provided by this abstraction suffice to derive a controller-inspired model for our resilient resource management process. Figure 4 shows an overview of the proposed process. In our model, process variables (output) are determined by fault-specific metrics quantified online (see Section 4). In contrast to a typical PID closed loop, the *Process* output is a set of fault degree measurements as previously referred (y', y'', etc.)—each corresponding output measurement feeds its respective controller. Fault degrees are computed from metrics assuming that faults have outlier performance, e.g. low network bandwidth, data packet losses, low CPU utilization, etc. [27]. The *setpoint* is constant and defined as 1. Our process may be composed by a set of standalone PID-inspired controllers, in which each control signal u is an input value for a *Decision Agent*. The Decision Agent collects all control signal values (formed entirely from the error e), and determines whether a preventive or curative set of actions u^* should be performed. Note that an action is not represented by a value that will be fed to the process, instead it indicates actual operations that will be performed. Negative error e values mean the control system is rising too fast and may tend to an overshoot state (i.e., reach a faulty state), therefore preventive or corrective actions should be performed. Actions may include task preemption, task resubmission, task clustering, task cleanup, storage management, etc. In contrast, positive values indicate the system is in an undershot state. Low e values indicate the control system is smoothly moving towards the steady state.

4. Modeling Simple Controller-inspired Processes

In our proposed approach, a standalone PID-inspired controller is defined and used for each possible-future fault identified from workload traces (historical data). In some cases, a particular type of fault cannot be modeled using the principles of a three-term controller. For example, there are faults that cannot be predicted far in advance (e.g., unavailability of resources due to a power cut). In this case, a PI-inspired (*proportional-integral*) controller could be defined and deployed. In production computing systems, a large number of controllers may be defined and used to control, for example, CPU utilization, network bandwidth, etc. In this paper, we demonstrate the feasibility of our proposed process by tackling two common, yet practical, issues of workflow executions: data and memory overflow.

4.1. Workflow Data Footprint and Management

In the era of Big Data Science, applications are producing and consuming ever-growing data sets. A run of scientific workflows that manipulates these data sets may lead the system to an out of disk space fault if no mechanisms are in place to control how the available storage is used. To prevent this, data cleanup tasks are often automatically inserted into the workflow by the workflow management system [34], or the number of concurrent task executions is limited to prevent data usage overflow. Cleanup tasks remove data sets that are no longer needed by downstream tasks or temporarily move current unlocked data into permanent storage devices (to free local storage space for running tasks), but nevertheless they may add an important overhead to the workflow execution [35]—a cleanup task may involve staging data out to an external storage device and registering the data into a data catalog. As a result, additional operations may be required to stage in these data for future task executions.

PID-inspired Controller. The process variable for the data management process (output y_d) is defined as the ratio between the actual used disk space ω' including current tasks in execution, and the total disk space ω, i.e. $y_d = \frac{\omega'}{\omega}$. In an ideal scenario, $y_d \to 1$ (i.e., the *setpoint*) maximizes utilization, however no overflow is allowed. Thus, a lower threshold is typically used to accommodate overflow and prevent non-recoverable failures. Therefore, the system is in a *non-steady* state if the total amount of available disk space is below or above a predefined threshold τ_d (i.e., $y_d = \frac{\omega'}{\omega \cdot \tau_d}$). The proportional (P) response is computed as the error between the *setpoint* r, and the process variable y_d; the integral (I) response is computed from the sum of the disk usage errors (cumulative value of the proportional responses, i.e. $e = r - y_d$); and the derivative (D) response is computed as the difference between the current and the previous disk overflow (or underutilization) error values. Therefore, the control signal u_d is defined as follows:

$$u_d(t) = K_p \cdot (r - y_d(t)) + K_i \sum_{n=1}^{t} e(n) + K_d \cdot (e(t) - e(t-1)). \quad (2)$$

Corrective Actions. The output of the PID-inspired controller (control signal u_d, Equation 2) indicates whether the controller identified an anomaly behavior w.r.t. data management. Negative values indicate that the current disk usage is above the threshold of the minimum required available disk space (a safety measure to avoid an unrecoverable faulty state). In contrast, positive values indicate that the current running tasks do not maximize disk usage. For values of $u_d < 0$, (*i*) data cleanup tasks can be triggered to remove unused intermediate data (adding cleanup tasks may imply rearranging the priority of all tasks in the queue), or (*ii*) tasks can be preempted due to the inability to remove data—the inability of cleaning up data may lead the execution to an unrecoverable state, and thereby to a failed execution. Otherwise (for $u_d > 0$), the number of concurrent task executions may be increased. The control signal value is then used as input for the Decision Agent (Section 4.3), which accounts for all control signal values from all controllers to perform preventive/corrective actions when necessary.

4.2. Workflow Memory Usage and Management

Large scientific computing applications rely on complex workflows to analyze large volumes of data. These tasks are often running in HPC resources over thousands of CPU cores and simultaneously performing data accesses, data movements, and computation, dominated by memory-intensive operations (e.g., reading a large volume of data from disk, decompressing in memory massive amounts of data or performing a complex calculation which generates large datasets, etc.). The performance of those memory-intensive operations are quite often limited by the memory capacity of the resource where the application is being executed. Therefore, if those operations overflow the physical memory limit the result may be application performance degradation or application failure. Typically, the end-user is responsible for optimizing the application, modifying the code if necessary to comply with the amount of memory that can be used on that resource. This work addresses the memory challenge by proposing an in-situ analysis of memory usage, to adapt the number of concurrent tasks executions according to the memory usage required by an application at runtime.

PID-inspired Controller. The process variable for the memory management process (output y_m) is defined as the ratio between the actual peak memory usage σ' by current tasks in execution, and the total memory capacity of the computing node σ. Similarly to the data management controller, a threshold τ_m is also used to accommodate overflows, thus $y_m = \frac{\sigma'}{\sigma \cdot \tau_m}$. The system is in a *non-steady* state if the amount of memory available is below or above τ_m. The proportional (P) response is computed as the error between the memory consumption *setpoint* value, and the output y_m; the integral (I) response is computed from cumulative proportional responses (previous memory usage errors); and the derivative (D) response is computed as the difference between the current and the previous memory overflow (or underutilization) error values. The control signal u_m is then defined as follows:

$$u_m(t) = K_p \cdot (r - y_m(t)) + K_i \sum_{n=1}^{t} e(n) + K_d \cdot (e(t) - e(t-1)). \quad (3)$$

Corrective Actions. Negative values for the control signal u_m indicate that the collection of running tasks are leading the system to an overflow state (i.e., anomalous behavior), thus some tasks should be preempted to prevent the system from running out of memory. For positive u_m values, the memory consumption of current running tasks is below a predefined memory consumption *setpoint* (i.e., underutilization). Therefore, the workflow management system may spawn additional tasks for concurrent execution.

4.3. Decision Agent

In a typical PID control loop, the response variable of the control loop that leads the system to a *setpoint* (or within a steady-state error) is defined as waveforms, which can be composed of overflows or underutilization of the system. As aforementioned, in order to accommodate overflows, we arbitrarily define the *setpoint* of our resource management process as 80% of the maximum total capacity (for both storage and memory usage), and a steady-state error of 5%. The evaluated process is composed of a single PID-inspired controller u_d, used to manage disk usage (shared network file system); while an independent memory controller u_m^n is deployed for each computing node n. As discussed in the previous subsections, the control signal values indicate whether the system is leading to an overflow or underutilization state, and thus actions may be triggered. These values are the input for the Decision Agent, which weights them to decide the appropriate set of actions u^* to be performed (Figure 4). The Decision Agent may also be based on an Intelligent System, where decisions do also account, for example, for historical data, system performance metrics (e.g., I/O or network throughput, etc.), workflow structure and look-ahead planning [11, 36], and the use of statistical and machine learning methods [22, 37].

In order to demonstrate the feasibility of our proposed approach, we consider that values of $u > 0$ indicate that the amount of disk space or memory consumed by the current running tasks fits the system resources and additional tasks may be spawned (resp. tasks are preempted). When managing a set of controllers, it is important to ensure that an action performed by a controller does not counteract an action performed by another one. Therefore, the decision about the number of tasks to be scheduled/preempted is driven by $\lfloor u \rfloor$, which represents the *min* between the response value of the unique disk usage controller, and the memory controller per resource n:

$$\lfloor u_n(t) \rfloor = \min(u_d(t), u_m^n(t)). \quad (4)$$

The Decision Agent process uses the mean values of disk and memory requirements (as the ones used in this work shown in Table 1, Section 5.2) to estimate the number of tasks to be scheduled/preempted. The Decision Agent seeks then for a set of tasks, in which the sum of their disk $\bar{\omega}$ and memory $\bar{\sigma}$ requirements are less than or equal to the thresholds. For the task

scheduling operation, a task k will be scheduled to a resource n at instant t iff:

$$\begin{cases} \bar{\omega}_k \leq \lfloor u_n(t) \rfloor \times \omega, \\ \bar{\sigma}_k \leq \lfloor u_n(t) \rfloor \times \sigma_n. \end{cases} \quad (5)$$

For task preemption, current running tasks are added to the set of tasks to be preempted P while the sum of disk $\bar{\omega}$ and memory $\bar{\sigma}$ requirements for all tasks $p \in P$ do not satisfy the following conditions:

$$\begin{cases} \lfloor u_n(t) \rfloor \times \omega > \sum_{p \in P} \bar{\omega}_p, \\ \lfloor u_n(t) \rfloor \times \sigma_n > \sum_{p \in P} \bar{\sigma}_p. \end{cases} \quad (6)$$

In the first condition, the disk usage requirement ($\lfloor u_n(t) \rfloor \times \omega$) may be reduced if data cleanup tasks can be executed. Thus, the condition is scaled down by the magnitude of the amount of data that can be removed (i.e., data files that are not used by the current running tasks). Strategies to define the optimal number of data cleanup tasks and their positioning in the workflow graph are out of the scope of this work, and can be found in [34].

Typically, mean values yield high values of standard deviation (due to variations inherent to the application itself, or the system including external load), thus estimations may not be accurate. Task characteristics estimation is beyond the scope of this work, and sophisticated methods to provide accurate estimates can be found in [22, 23, 24, 25]. However, this work intends to demonstrate that even using inaccurate estimation methods, our proposed process can cope with the poor estimates and still yield good results.

5. Experimental Evaluation

5.1. Scientific Workflow Application

The 1000 genomes project provides a reference for human variation, having reconstructed the genomes of 2,504 individuals across 26 different populations [38]. The test case used in this work identifies mutational overlaps using data from the 1000 genomes project in order to provide a null distribution for rigorous statistical evaluation of potential disease-related mutations. This test case (Figure 5) has been implemented as a Pegasus [39, 40] workflow, and is composed of five different tasks:

Individuals. This task fetches and parses the Phase 3 data [38] from the 1000 genomes project per chromosome. These files list all of the Single nucleotide polymorphisms (*SNPs*) variants in that chromosome and which individuals have each one. An `individual` task creates output files for each individual of *rs numbers*, where individuals have mutations in at least one of the two alleles.

Populations. The 1000 genome project has 26 different populations from many different locations worldwide [41]. The `populations` task fetches and parses five super populations (African, Mixed American, East Asian, European, and South Asian), and a set of all individuals.

Sifting. This task computes the *SIFT* scores of all of the *SNPs* variants, as computed by the Variant Effect Predictor (*VEP*). *SIFT* is a sequence homology-based tool that Sorts Intolerant From Tolerant amino acid substitutions, and predicts whether an amino acid substitution in a protein will have a phenotypic effect. *VEP* determines the effect of individual variants on genes, transcripts, and protein sequences, as well as regulatory regions. For each chromosome, the `sifting` task processes the corresponding *VEP*, and selects only the *SNPs* variants that have a *SIFT* score.

Pair_Overlap_Mutations. This task measures the overlap in mutations (*SNPs*) among pairs of individuals. Considering two individuals, if both individuals have a given SNP then they have a mutation overlap. It performs several correlations including different numbers of pairs of individuals, and different numbers of *SNPs* variants (only the *SNPs* variants with a score less than 0.05, and all the *SNPs* variants); and computes an array (per chromosome, population, and *SIFT* level selected), which has as many entries as individuals—each entry contains the list of *SNPs* variants per individual according to the *SIFT* score.

Frequency_Overlap_Mutations. This task calculates the frequency of overlapping mutations across n subsamples of j individuals. For each run, the task randomly selects a group of 26 individuals from this array and computes the number of overlapping mutations among the group. Then, the `individuals` task computes the frequency of mutations that have the same number of overlapping mutations.

5.2. Workflow Characterization

We profiled the 1000 genome sequencing analysis workflow using the Kickstart [42] profiling tool. Kickstart monitors and records task execution in scientific workflows (e.g., process I/O, runtime, memory usage, and CPU utilization). Runs were conducted on the *Eddie Mark 3*, which is the third iteration of the University of Edinburgh's compute cluster. The cluster is composed of 4,000+ cores with up to 2 TB of memory. For running the characterization experiments, we have used three types of nodes, depending of the size of memory required for each task:

1. 1 Large node with 2 TB RAM, 32 cores, Intel® Xeon® Processor E5-2630 v3 (2.4 GHz), for running the `individual` tasks;
2. 1 Intermediate node with 192GB RAM, 16 cores, Intel® Xeon® Processor E5-2630 v3 (2.4 GHz), for running the `sifting` tasks;
3. 2 Standards nodes with 64 GB RAM, 32 cores, Intel® Xeon® Processor E5-2630 v3 (2.4 GHz), for running the remaining tasks.

Table 1 shows the execution profile of the workflow. Most of the workflow execution time is allocated to the `individual` tasks. These tasks are in the critical path of the workflow due to their high demand for disk (174GB on average per task) and memory (411GB on average per task). The total workflow data

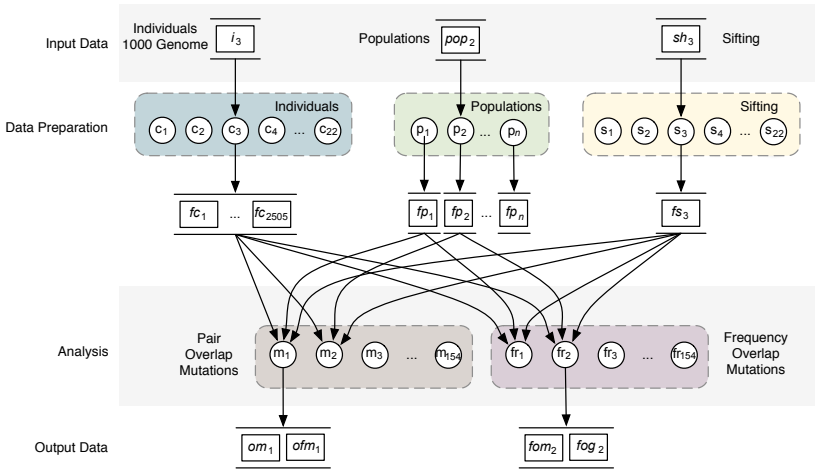


Figure 5: Overview of the 1000 genome sequencing analysis workflow.

footprint is about 4.4TB. Although the large node provides 2 TB of RAM and 32 cores, we would only be able to run up to 4 concurrent tasks per node. In *Eddie Mark 3*, the standard disk quota is 2GB per user, and 200GB per group. Since this quota would not suffice to run all tasks of the 1000 genome sequencing analysis workflow (even if all tasks run sequentially), we had a special arrangement to increase our quota to 500GB. Note that this increased quota allows us to barely run 3 concurrent `individual` tasks in the large node, and some of the remaining tasks in smaller nodes. Therefore, data and memory management are crucial to perform a successful run of the workflow, while meeting the life scientists' expectations.

5.3. Experiment Conditions

Scientific workflows and workflow systems must be evaluated on large-scale platforms, since scalability is a major concern for next-generation applications. However, large-scale platforms are typically non-dedicated with shared network infrastructures and shared compute resources (e.g., space-shared via batch queues). Furthermore, real-world platforms are known to exhibit transient behaviors due to load spikes, maintenance, software upgrade, and (mis)configurations. As a result, evaluation experiments are not inherently repeatable. Additionally, executing large-scale workflows merely to compare the performance of workflow executions consumes resources and energy—it is typical to run series of back-to-back experiments to address concerns for the validity of the drawn conclusions. Therefore, the experiments use cycle-based simulation. Since most workflow simulators are event-based [43, 44], we developed an activity-based simulator to simulate every time slice (or cycle) of the controllers' behavior (which is available online [45]), while in an event-based simulation, each event occurs at a particular instant in time and marks a change of state in the system. The simulator provides support for task scheduling and resource provisioning at the workflow level. The simulated computing environment represents the three nodes from the *Eddie Mark 3* cluster described in Section 5.2 (total 80 CPU cores). Additionally, we assume a shared network file system among the nodes with total capacity of 500GB.

We use an FCFS policy with task preemption and backfill for task scheduling—tasks submitted at the same time are randomly chosen (may introduce variability in the execution), and preempted tasks return to the top of the queue. To avoid unrecoverable faults due to running out of disk space, we implemented a *fault-tolerance* data cleanup mechanism to remove data that are no longer required by downstream tasks [34]. In this case, data cleanup tasks are only triggered if the maximum storage capacity is reached: all running tasks are preempted, the data cleanup task is executed, and the workflow resumes its execution. Recall that this mechanism may add a significant overhead to the workflow execution (see Section 4.1). For this set of experiments, we initially assume $K_p = K_i = K_d = 1$ to demonstrate the feasibility of the approach regardless the use of tuning methods.

The goal of this experiment is to ensure that correctly defined executions complete, that performance is acceptable, and that possible-future faults are quickly detected and automatically handled before they lead the workflow execution to an unrecoverable state (measured by the number of data cleanup tasks dispatched by the *fault-tolerance* mechanism described above). Therefore, we do not attempt to optimize task preemption (which criteria should be used to select tasks for removal, or perform checkpointing) since our goal is to demonstrate the feasibility of the approach with simple use case scenarios.

Reference Workflow Execution. In order to measure the efficiency of our proposed method under online and unknown conditions, we compare the workflow execution performance (in terms of workflow makespan) to a reference workflow execution. The reference workflow is computed offline under known conditions, i.e., all requirements (e.g., runtime, disk, memory) are accurate and known in advance. We performed several runs for the reference workflow using the FCFS policy with backfill, which yielded an averaged makespan of 382,887.7s (~106h, standard deviation $\leq 5\%$).

Task	Count	Runtime		Data Footprint		Memory Peak	
		Mean (s)	Std. Dev.	Mean (GB)	Std. Dev.	Mean (GB)	Std. Dev.
Individual	22	31593.7	17642.3	173.79	82.34	411.08	17.91
Population	7	1.14	0.01	0.02	0.01	0.01	0.01
Sifting	22	519.9	612.4	0.94	0.43	7.95	2.47
Pair_Overlap_Mutations	154	160.3	318.7	1.85	0.85	17.81	20.47
Frequency_Overlap_Mutations	154	98.8	47.1	1.83	0.86	8.18	1.42
Total (cumulative)	359	590993.8	–	4410.21	–	24921.58	–

Table 1: Execution profile of the 1000 genome sequencing analysis workflow.

Configuration	Avg. Makespan (h)	Slowdown
Reference	106.36	–
P	138.76	1.30
PI	126.69	1.19
PID	114.96	1.08

Table 2: Average workflow makespan for different configurations of the controllers: (P) proportional, (PI) proportional-integral, and (PID) proportional-integral-derivative. `Reference` denotes the makespan of a reference workflow execution computed offline and under known conditions.

5.4. Experimental Results and Discussion

We have conducted workflow runs with three different types of controller: (P) only the proportional component is evaluated: $K_p = 1$, and $K_i = K_d = 0$; (PI) the proportional and integral components are enabled: $K_p = K_i = 1$, and $K_d = 0$; and (PID) all components are activated: $K_p = K_i = K_d = 1$. The reference workflow execution is reported as `Reference`. We have performed several runs of each configuration to produce results with statistical significance (errors below 5%).

5.4.1. Overall makespan evaluation

Table 2 shows the average makespan (in hours) for the three configurations of the controller and the reference workflow execution. The degradation of the makespan is expected due to the online and unknown conditions (no information about the tasks is available in advance). In spite of the fact that the mean does not provide accurate estimates, the use of a control loop mechanism diminishes this effect. The use of controllers may also degrade the makespan due to task preemption. However, if tasks were scheduled only using the estimates from the mean, the workflow would not complete its execution due to lack of disk space or memory overflows.

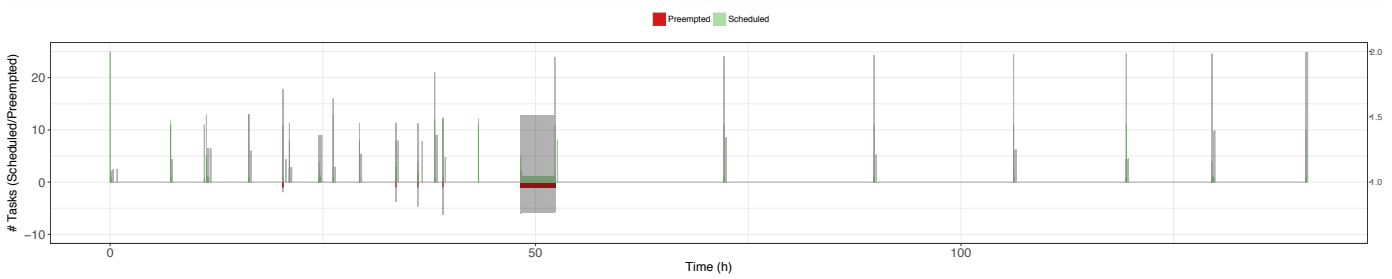
Executions using our resilient resource management process (enabled by PID-inspired controllers) outperform executions using only the proportional (P) or the PI components. The PID-inspired controller slows down the application by 1.08, while the application slowdown is 1.19 and 1.30 for the PI and P controllers, respectively. This result suggests that the derivative component (prediction of future errors) has a significant impact on the workflow executions, and that the accumulation of past errors (integral component) is also important to prevent and mitigate faults. Therefore, below we analyze how each of these components influence the number of tasks scheduled, and the peaks and troughs of the controller response function. We did not perform runs where mixed PID, PI, and P controllers were part of the same simulation (i.e., all controllers that compose the process shown in Figure 4 have the same components), since it would be very difficult to determine the influence of each controller.

5.4.2. Data footprint

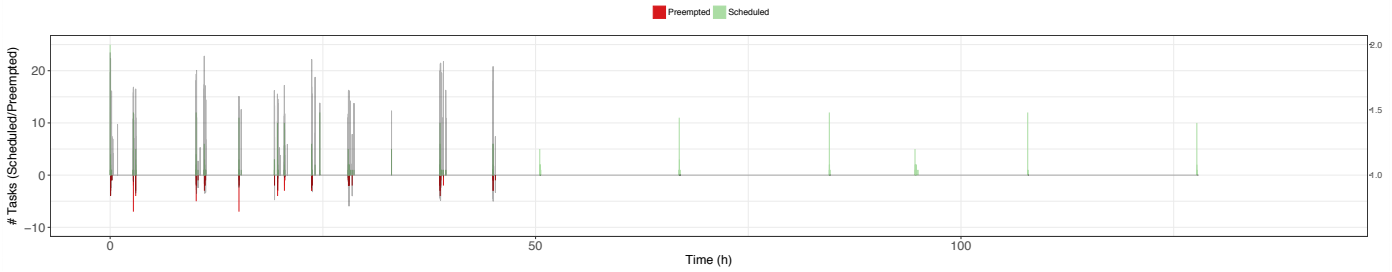
Figure 6 shows the time series of the number of tasks scheduled or preempted during workflow executions. For each controller configuration, we present a single execution, where the makespan is the closest to the average makespan value shown in Table 2. Task preemptions are represented as negative values (red bars), while positive values (blue bars) indicate the number of tasks scheduled at an instant of time. Additionally, the right y-axis shows the step response u_d of the controller input value (black/gray line) for disk usage during the workflow execution. Recall that *positive* input values ($u_d(t) > 0$, Equation 2) trigger task scheduling, while *negative* input values ($u_d(t) < 0$) trigger task preemption and/or data cleanup tasks.

The proportional controller (P, Figure 6a) is limited to the current error, i.e., the Decision Agent is driven by the amount of disk space that is over/underutilized. Since the controller input value is strictly proportional to the error, there is a burst in the number of tasks to be scheduled during the workflow execution. This bursty pattern and the nearly constant variation of the input value lead the system to an inconsistent state (*livelock*), where the remaining tasks to be scheduled cannot let the controller reach the steady-state (appears as a black opaque rectangle in the figure). Consequently, tasks are constantly scheduled and then preempted. In the example scenario shown in Figure 6a, this process occurs for approximately 4h (between 48-52h), and performs more than 6,000 preemptions. Figure 7a shows a 1-hour snippet of this behavior (between 49h and 50h), that characterizes the livelock—the act of scheduling a task is followed by that task's preemption. Note that the response of the controller input value (black line) oscillates with a similar magnitude. Since the proportional controller has no mechanism to attenuate the proportional component (current error), the system remains in an inconsistent state until some external disturbance change the current state of the system. In this particular example, the livelock is only resolved upon task completion, so that other tasks can start to run.

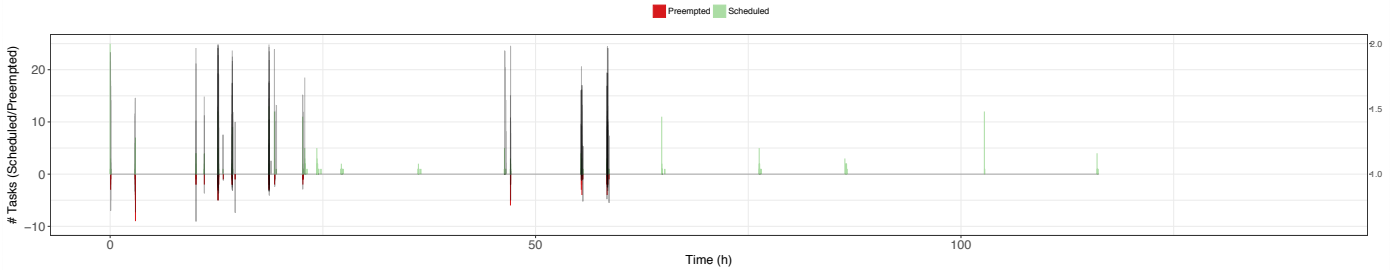
Table 3 shows the average number of preemptions and cleanup tasks occurrences per workflow execution. On average, proportional controllers produced more than 7,000 preemptions, but no cleanup tasks. The lack of cleanup tasks indicate that the number of concurrent executions is very low



(a) Proportional Controller (`P`)

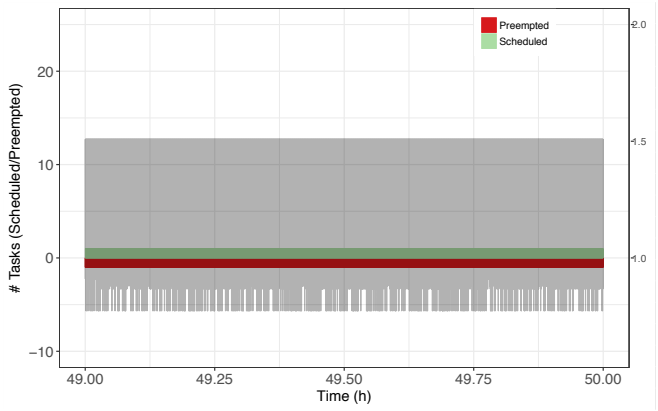


(b) Proportional-integral Controller (`PI`)

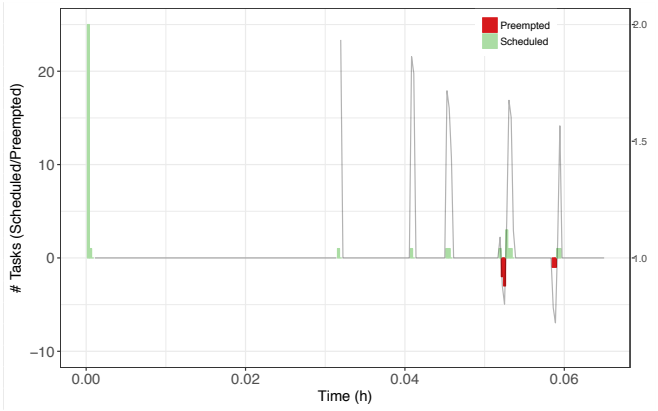


(c) Proportional-integral-derivative Controller (`PID`)

Figure 6: *Data Footprint*: Number of tasks scheduled (blue bars for positive values) and preempted (red bars for negative values) during the lifespan of a workflow execution (left y-axis). The right y-axis represents the step response of the controller input value u_d (black/gray line) during the workflow execution.



(a) Proportional Controller (`P`)



(b) Proportional-integral-derivative Controller (`PID`)

Figure 7: *Data Footprint*: Number of tasks scheduled (blue bars for positive values) and preempted (red bars for negative values) during the (a) *livelock* phase for the proportional controller and the (b) initial phase of the workflow execution. Black/gray lines indicate the controller control signal u_d value.

(mostly influenced by the number of task preemptions), which is observed from the high average application slowdown of 1.30.

The proportional-integral controller (`PI`, Figure 6b) aggregates the cumulative error when computing the response of the controller. As a result, the bursty pattern is smoothed along the execution, task concurrency is increased, and the livelock is prevented. However, the cumulative error tends to increase the response of the `PI` controller at each iteration (both positively or negatively). As a result, task preemption occurs earlier during execution. On the other hand, this behavior mitigates the *vicious cycle* (livelock) present in the `P` controllers, and

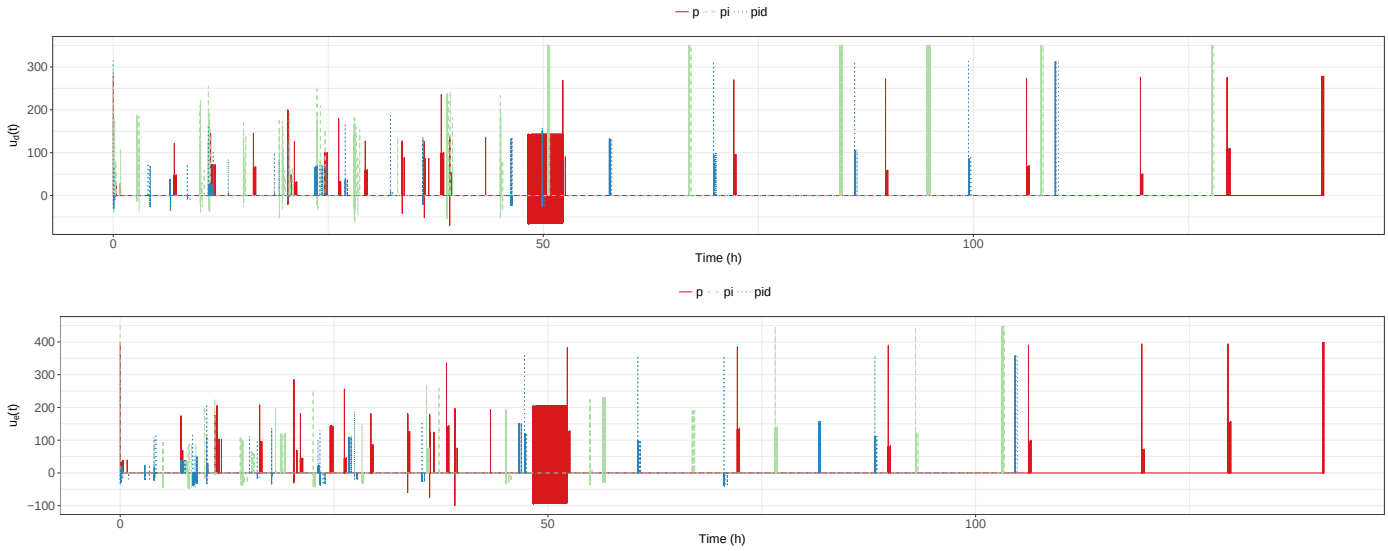


Figure 8: *Data Footprint*: Values for input $u_n(t)$ (top) and error $u_e(t)$ (bottom) for `P`, `PI`, and `PID` controllers shown in Figure 6.

Controller	# Tasks Preempted	# Cleanup Tasks
`P`	7225	0
`PI`	168	48
`PID`	73	4

Table 3: Average actual number of tasks preempted and cleanup tasks executed per workflow run when using `P`, `PI`, and `PID` controllers.

consequently the average number of preempted tasks is substantially reduced to 168 (Table 3). A drawback of using a `PI` controller, is the presence of cleanup tasks (48 tasks on average), which is due to the higher level of concurrency among task executions. In contrast to the proportional controller, tasks are gradually scheduled, mostly via backfilling, then disk space becomes the bottleneck. The excessive use of cleanup tasks to free space substantially slowdowns the workflow execution (1.19, Table 2).

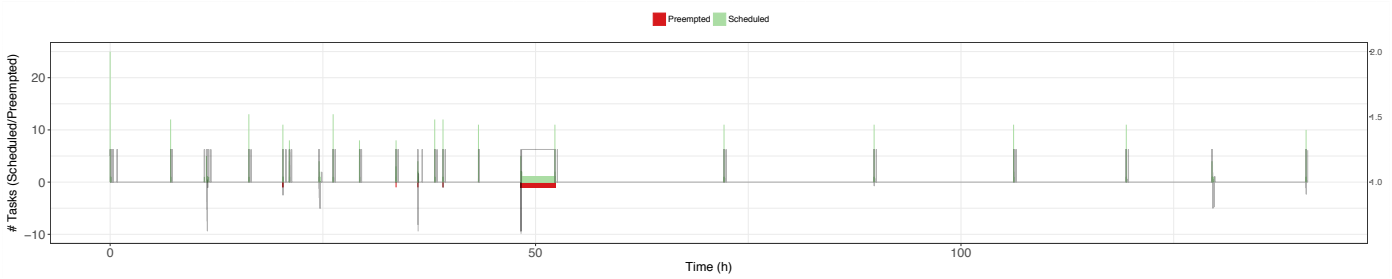
The proportional-integral-derivative controller (`PID`, Figure 6c) emphasizes the previous response produced by the controller (the last computed error). The derivative component drives the controller to trigger actions once the current error follows (or increases) the previous error trend. This behavior is shown in Figure 7b (it shows a snippet for the first 6 minutes of the workflow execution). Note that the task scheduling burst at the beginning of the execution represents the *rise time* (Figure 3) from the typical response of a PID controller. The control loop only performs actions when disk usage is moving towards an overflow or underutilization state. Note that the number of actions (scheduling/preemption) triggered in Figure 6c is much less than the number triggered by the `PI` controller: the average number of preempted tasks is 73, and only 4 cleanup tasks on average are spawned (Table 3).

5.4.3. Memory Usage

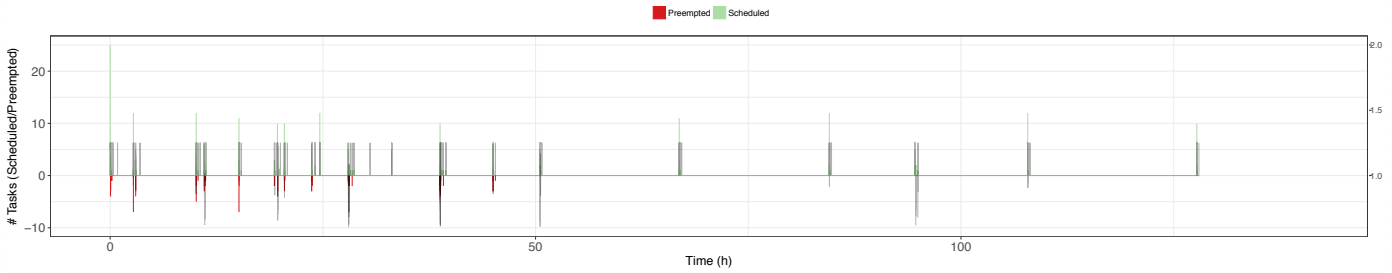
Figure 9 shows the time series of the number of tasks scheduled or preempted during the workflow executions for the memory controllers. The right y-axis shows the step response of the controller input value u_m (black/gray line) for memory usage during the workflow execution. We present the response function of a controller attached to a standard cluster (32 cores, 64GB RAM, Section 5.2), which runs the `population`, `pair_overlap_mutations`, and `frequency_overlap_mutations` tasks. The total memory allocations required to run all these tasks is over 4TB, which might lead the system to memory overflow states (in case there is enough disk space available to run multiple concurrent tasks).

When using the proportional controller (`P`, Figure 9a), most of the actions are triggered by the data footprint controller (Figure 6a). As aforementioned, memory does not become an issue when only the proportional error is taken into account, since task execution is nearly sequential (low level of concurrency). As a result, only a few tasks (on average less than 5) are preempted due to memory overflow. Note that the process of constant task scheduling (~50h of execution) is strongly influenced by the memory controller. Additionally, the step response shown in Figure 9a highlights that most of the task preemptions occur in the standard cluster. This result suggests that actions performed by the global data footprint controller is affected by actions triggered by the local memory controller. From Figure 10a, we observe that the memory controller does not enter an inconsistent state (livelock), instead it always prompts positive response values. The approach to mitigate conflicted actions from multiple controllers enforced by the Decision Agent (Equation 4, Section 4.3) nullifies this response, since task preemption has higher priority (we do not intend to lead the system to an unrecoverable state). The analysis of the influence of multiple concurrent controllers is out of the scope of this paper, however this result demonstrates that controllers should be used sparingly, and actions induced by controllers should be performed by priority or the controller hierarchical level.

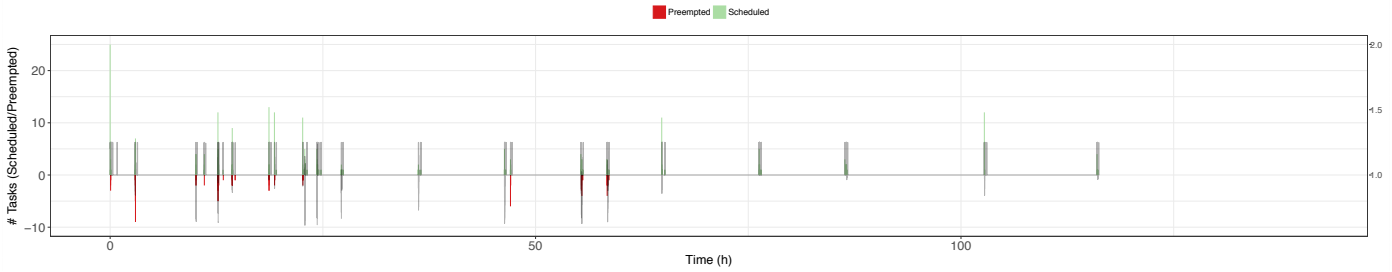
Similar to the `PI` controller for data footprint, the memory usage `PI` controller (Figure 9b) mitigates this effect—the cu-



(a) Proportional Controller (P)

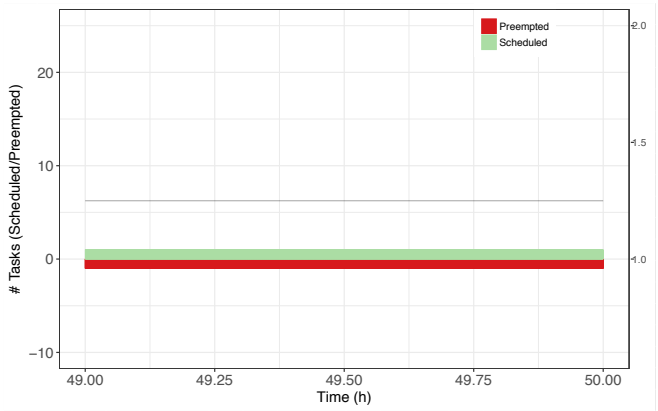


(b) Proportional-integral Controller (PI)

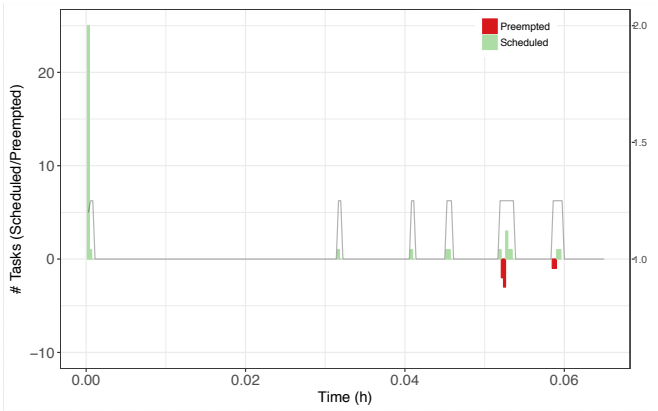


(c) Proportional-integral-derivative Controller (PID)

Figure 9: *Memory Usage*: Number of tasks scheduled (blue bars for positive values) and preempted (red bars for negative values) during the lifespan of a workflow execution (left *y*-axis). The right *y*-axis represents the step response of the controller input value (black/gray line) during the workflow execution. This figure shows the step response function of a controller attached to a standard cluster (32 cores, 64GB RAM), which has more potential to arise memory overflows.



(a) Proportional Controller (P)



(b) Proportional-integral-derivative Controller (PID)

Figure 10: *Memory Usage*: Number of tasks scheduled (blue bars for positive values) and preempted (red bars for negative values) during the (a) *livelock* phase for the proportional controller and the (b) initial phase of the workflow execution. Black/gray lines indicate the control signal u_m value.

mulative error prevents the controller from triggering repeated actions. Observing the step response u_m of the PI memory controller and the PI data footprint controller (Figure 6b), we notice that most of the task preemptions are triggered by the memory controller, particularly in the first quarter of the execution (see slopes of the black/gray lines). The average data footprint per task of the `population`, `pair_overlap_mutations`, and `frequency_overlap_mutations` tasks is 0.02GB, 1.85GB, and 1.83GB (Table 3), respectively. Thus, the data footprint controller tends to increase the number of concurrent tasks. In the absence of memory controllers, the workflow execution would tend to memory overflow, and thus lead to a failed, unre-

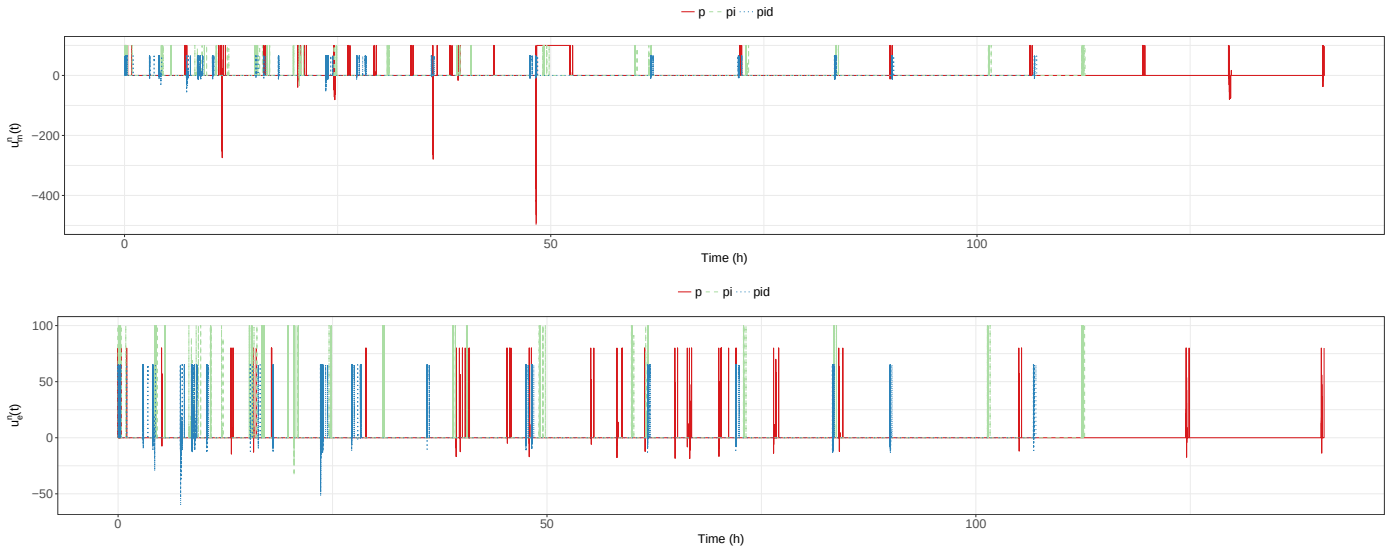


Figure 11: *Memory Usage*: Values for input $u_n(t)$ (top) and error $u_e(t)$ (bottom) for `P`, `PI`, and `PID` controllers shown in Figure 9.

Control Type	K_p	K_i	K_d
P	$0.50 \cdot K_u$	–	–
PI	$0.45 \cdot K_u$	$1.2 \cdot K_p/T_u$	–
PID	$0.60 \cdot K_u$	$2 \cdot K_p/T_u$	$K_p \cdot T_u/8$

Table 4: Ziegler-Nichols tuning, using the oscillation method. These gain values are applied to the parallel form of the PID controller, which is the object of study in this paper. When applied to a standard PID form, the integral and derivative parameters are only dependent on the oscillation period T_u.

coverable state.

The derivative component of the `PID` controller (Figure 9c) acts as a catalyst to improve memory usage: it decreases the overflow and the settling time without affecting the steady-state error. As a result, the number of actions triggered by the `PID`-inspired memory controller is significantly reduced when compared to the `PI` or `P` controllers. Figure 10b emphasizes the gradual increase on the number of tasks scheduled at the beginning of the workflow execution, and the first task preemption triggered by the data footprint controller.

Although the experiments conducted in this feasibility study considered equal weights for each of the components in a PID-inspired controller (i.e., $K_p = K_i = K_d = 1$), we have demonstrated that correctly defined executions complete with acceptable performance, and that faults were detected far in advance of their occurrence, and were automatically handled before they lead the workflow execution to an unrecoverable state. In the next section, we explore the use of a simple and commonly used tuning method to calibrate the gain parameters from our process' controllers inspired by the PID principles.

6. Tuning PID Controllers

The goal of tuning a PID control loop is to make it stable, responsive, and to minimize overflow. However, there is no optimal way to achieve responsiveness without compromising overflow, or vice-versa. Therefore, a plethora of methods have been developed for tuning PID control loops. In this paper, we use the Ziegler-Nichols method to tune the gain parameters of the data footprint and memory controllers. This is one of the most common heuristics that attempts to produce tuned values for the three PID gain parameters (K_p, K_i, and K_d) given two measured feedback loop parameters derived from the following measurements: (*i*) the period T_u of the oscillation frequency at the stability limit, and (*ii*) the gain margin K_u for loop stability.

6.1. Determining T_u and K_u

The Ziegler-Nichols oscillation method is based on experiments executed on an established closed loop. The overview of the tuning procedure is as follows [46]:

1. Turn the PID controller into a P controller by setting $K_i = K_d = 0$. Initially, K_p is also set to zero;
2. Increase K_p until there are sustained oscillations in the signal in the control system. This K_p value is denoted the ultimate (or critical) gain, K_u;
3. Measure the ultimate (or critical) period T_u of the sustained oscillations; and
4. Calculate the controller parameter values according to Table 4, and use these parameter values in the controller.

A detailed explanation of the method can be found in [33]. In this section, we present how we determine the period T_u, and the gain margin K_u for loop stability.

Since workflow executions are intrinsically dynamic (due to the arrival of new tasks at runtime), it is difficult to establish a sustained oscillation in the signal. Therefore, in this paper we measured sustained oscillation in the signal within the execution of long running tasks—in this case the `individual` tasks (Table 1). We conducted runs (O(100)) with the proportional

Controller	K_u	T_u	K_p	K_i	K_d
Data Footprint	0.58	3.18	0.35	0.22	0.14
Memory Usage	0.53	12.8	0.32	0.05	0.51

Table 5: Tuned gain parameters (K_p, K_i, and K_d) for both the data footprint and memory usage PID controllers. K_u and T_u are computed using the Ziegler-Nichols method, and represent the ultimate period and critical gain, respectively.

(P) controller to compute the period T_u and the gain margin K_u. Table 5 shows the values for K_u and T_u for each controller used in the paper, as well as the tuned gain values for K_p, K_i, and K_d for the PID-inspired controller.

6.2. Experimental Evaluation and Discussion

We have conducted runs with the tuned PID-inspired controllers for both the data footprint and memory usage. Figure 12 shows the time series of the number of tasks scheduled or preempted during the workflow executions, and the step response of the controller input value (right *y*-axis). The average workflow execution makespan is 386,561s, which yields a slowdown of 1.01. The average number of preempted tasks is around 18, and only a single cleanup task was used in each workflow execution. The controller step responses, for both the data footprint Figure 12a) and the memory usage (Figure 12b), show lower peaks and troughs during the workflow execution when compared to the PID controllers using equal weights for the gain parameters (Figures 6c and 9c, respectively). More specifically, the controller input value is reduced by 30% for the memory controller attached to a standard cluster. This behavior is attained through the weighting yielded by the tuned parameters. However, tuning the gain parameters cannot ensure that an optimal scheduling will be produced for workflow runs (mostly due to the dynamism inherent to workflow executions) as few preemptions are still triggered.

Although the Ziegler-Nichols method provides quasi-optimal workflow executions (for the workflow studied in this paper), the key factor of its success is due to the specialization of the controllers, and thereby the process, to a single application. In production systems, such methodology may not be realistic because of the variety of applications running by different users—deploying a PID-inspired controller per application and per component (e.g., disk, memory, network, etc.) may significantly increase the complexity of the system and the system's requirements. On the other hand, controllers may be deployed in the user's space (or per workflow engine) to manage a smaller number of workflow executions.

A key advantage of using a process inspired by the principles of PID controllers is the ability to identify faults far in advance before they occur, and to perform pondered corrective or preventive actions to smoothly mitigate the undesired state. Additionally, the time required to process the current state of the system and decide whether to trigger an action is nearly instantaneous—it simply needs to solve a linear equation. This strongly favors the use of controller-inspired solutions on the execution of online distributed workflow applications. More sophisticated methods (e.g., using machine learning) may provide better approaches to tune the gain parameters. However, they may also add a significant overhead.

7. Conclusion

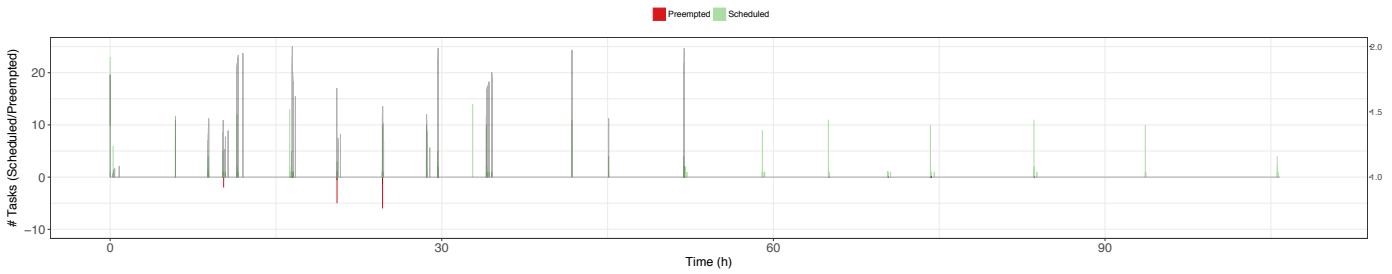
In this paper, we have described, evaluated, and discussed the feasibility of using the principles of PID controller to develop an online resilient resource management process to prevent and mitigate faults and under unknown conditions in workflow executions. We have addressed two common faults of today's science applications, data storage overload and memory overflow (main issues in data-intensive workflows), as use cases to demonstrate the feasibility of the proposed approach.

Simulation results using simple defined standalone control loops (no tuning) show that faults are detected and prevented before their occur, leading workflow execution to its completion with acceptable performance (slowdown of 1.08). The experiments also demonstrated the importance of each component in a PID-inspired controller. We then used the Ziegler-Nichols method to tune the gain parameters of the controllers (both data footprint and memory usage). Experimental results show that the control loop system produced nearly optimal scheduling—slowdown of 1.01. Therefore, we claim that the preliminary results of this work open a new avenue of research in workflow management systems.

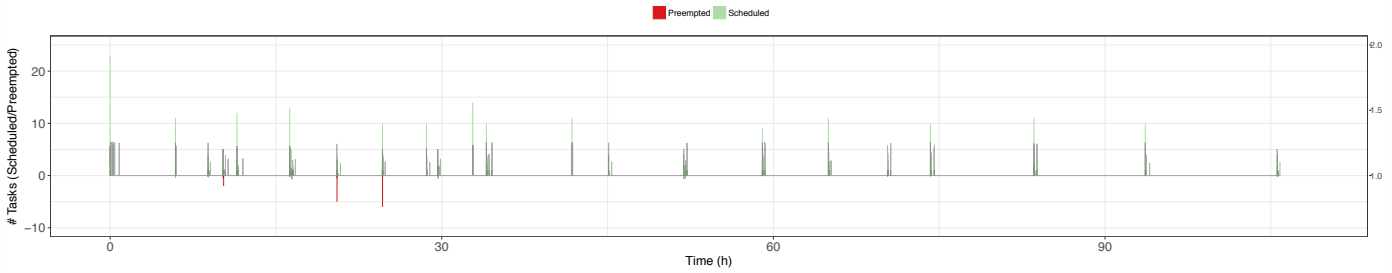
Although the process proposed in this paper is inspired by the principles of PID controllers, actual controllers may yield better accuracy (a thoroughly study on defining, implementing, and evaluating PID controllers would be required). We do also acknowledge that controllers should be used sparingly, and metrics (and actions) should be defined in a way that they do not lead the system to an inconsistent state—as observed in this paper when only the proportional component was used. Therefore, we plan to investigate the simultaneous use of multiple control loops at the workflow and infrastructure levels, to determine to which extent this approach may negatively impact the system. The analysis of the influence of using multiple concurrent controllers at different levels is challenging because of (1) the large number of controllers, which can be deployed per workflow task and related performance metrics, or per execution node or core; (2) the "chain effect" of distinguished, and possibly conflicted, actions resulted from the actuator output of the controllers; and (3) the dynamic behavior inherent to distributed systems. Future work include the design and implementation of a solution for analyzing the challenges aforementioned using an accurate, scalable simulation framework [47], which will allow us to design a realistic system for usage in production systems.

Acknowledgments

This work was funded by DOE contract number #DESC0012636, "Panorama—Predictive Modeling and Diagnostic Monitoring of Extreme Science Workflows". This work was carried out when Rosa Filgueira worked for the



(a) PID Data Footprint Controller



(b) PID Memory Controller

Figure 12: *Tuning PID Controllers*: Number of tasks scheduled (blue bars for positive values) and preempted (red bars for negative values) during the lifespan of a workflow execution (left y-axis). The right y-axis represents the step response of the controller input value (black/gray line) during the workflow execution. The bottom of the figure shows the step response function of a memory controller attached to a standard cluster (32 cores, 64GB RAM), which has more potential to arise memory overflows. The average workflow makespan is 386,561s, i.e. an average application slowdown of 1.01.

University of Edinburgh, and was funded by the Postdoctoral and Early Career Researcher Exchanges (PECE) fellowship funded by the Scottish Informatics and Computer Science Allience (SICSA) in 2016, and Erola Pairo-Castineira was supported by the Wellcome Trust-University of Edinburgh Institutional Strategic Support Fund (to Ian M. Overton).

References

[1] I. J. Taylor, et al., Workflows for e-Science: scientific workflows for grids, 2014.

[2] C. S. Liew, M. P. Atkinson, M. Galea, T. F. Ang, P. Martin, J. I. V. Hemert, Scientific workflows: moving across paradigms, ACM Computing Surveys (CSUR) 49 (4) (2016) 66. doi:10.1145/3012429.

[3] R. Ferreira da Silva, R. Filgueira, I. Pietri, M. Jiang, R. Sakellariou, E. Deelman, A characterization of workflow management systems for extreme-scale applications, Future Generation Computer Systems 75 (2017) 228–238. doi:10.1016/j.future.2017.02.026.

[4] N. Muthuvelu, C. Vecchiola, I. Chai, E. Chikkannan, R. Buyya, Task granularity policies for deploying bag-of-task applications on global grids, Future Generation Computer Systems 29 (1) 170–181. doi:10.1016/j.future.2012.03.022.

[5] G. Kandaswamy, A. Mandal, D. A. Reed, Fault tolerance and recovery of scientific workflows on computational grids, in: 2008. CCGRID'08. 8th IEEE International Symposium on Cluster Computing and the Grid, IEEE, 2013, pp. 777–782. doi:10.1109/CCGRID.2008.79.

[6] Y. Zhang, A. Mandal, C. Koelbel, K. Cooper, Combined fault tolerance and scheduling techniques for workflow applications on computational grids, in: 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE Computer Society, 2008, pp. 244–251. doi:10.1109/CCGRID.2009.59.

[7] J. Montagnat, T. Glatard, D. Reimert, K. Maheshwari, E. Caron, F. Desprez, Workflow-based comparison of two distributed computing infrastructures, in: 2010 5th Workshop on Workflows in Support of Large-Scale Science (WORKS), IEEE, 2009, pp. 1–10. doi:10.1109/WORKS.2010.5671856.

[8] O. A. Ben-Yehuda, A. Schuster, A. Sharov, M. Silberstein, A. Iosup, Expert: Pareto-efficient task replication on grids and a cloud, in: 2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS), IEEE, 2007, pp. 167–178. doi:10.1109/IPDPS.2012.25.

[9] H. Arabnejad, J. Barbosa, Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems, in: 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA), IEEE, 2012, pp. 633–639. doi:10.1109/ISPA.2012.94.

[10] D. Poola, K. Ramamohanarao, R. Buyya, Enhancing reliability of workflow execution using task replication and spot instances, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 10 (4) (2015) 30. doi:10.1145/2815624.

[11] W. Chen, R. Ferreira da Silva, E. Deelman, T. Fahringer, Dynamic and fault-tolerant clustering for scientific workflows, IEEE Transactions on Cloud Computing 4 (1) (2016) 49–62. doi:10.1109/TCC.2015.2427200.

[12] I. Casas, J. Taheri, R. Ranjan, L. Wang, A. Y. Zomaya, A balanced scheduler with data reuse and replication for scientific workflows in cloud computing systems, Future Generation Computer Systems-doi:10.1016/j.future.2015.12.005.

[13] U. Schwiegelshohn, How to design a job scheduling algorithm, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 2014, pp. 147–167. doi:10.1007/978-3-319-15789-4_9.

[14] R. Ferreira da Silva, R. Filgueira, E. Deelman, E. Pairo-Castineira, I. M. Overton, M. Atkinson, Using simple pid controllers to prevent and mitigate faults in scientific workflows, in: 11th Workflows in Support of Large-Scale Science (WORKS'16), 2016.

[15] M. Tanaka, O. Tatebe, Design of fault tolerant pwrake workflow system supported by gfarm file system, in: Proceedings of the 9th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers, IEEE Press, 2016, pp. 7–12. doi:10.1109/MTAGS.2016.7.

[16] A. Hary, A. Akoglu, Y. AlNashif, S. Hariri, D. Jenerette, Design and evaluation of a self-healing kepler for scientific workflows, in: 19th ACM International Symposium on High Performance Distributed Computing (HPDC), 2010, pp. 340–343. doi:10.1145/1851476.1851525.

[17] S. Köhler, S. Riddle, D. Zinn, T. McPhillips, B. Ludäscher, Improving workflow fault tolerance through provenance-based recovery, in: International Conference on Scientific and Statistical Database Management, 2011, pp. 207–224. doi:10.1007/978-3-642-22351-8_12.

[18] D. Poola, K. Ramamohanarao, R. Buyya, Fault-tolerant workflow scheduling using spot instances on clouds, Procedia Computer Science 29 (2014) 523–533. doi:10.1016/j.procs.2014.05.047.

[19] F. Costa, et al., Handling failures in parallel scientific workflows us-

ing clouds, in: High Performance Computing, Networking, Storage and Analysis (SCC), 2012, pp. 129–139.
[20] H. Casanova, On the harmfulness of redundant batch requests, in: 15th IEEE International Conference on High Performance Distributed Computing, IEEE, 2006, pp. 255–266. doi:10.1109/HPDC.2006.1652157.
[21] S. Sindhuja, S. Kayalvili, A survey on dynamic and fault-tolerant clustering for scientific workflows, International Journal of Engineering Science 3030.
[22] R. Ferreira da Silva, G. Juve, M. Rynge, E. Deelman, M. Livny, Online task resource consumption prediction for scientific workflows, Parallel Processing Letters 25 (3). doi:10.1142/S0129626415410030.
[23] I. Pietri, G. Juve, E. Deelman, R. Sakellariou, A performance model to estimate execution time of scientific workflows on the cloud, in: Workflows in Support of Large-Scale Science (WORKS), 2014 9th Workshop on, IEEE, 2014, pp. 11–19.
[24] H. Hiden, S. Woodman, P. Watson, A framework for dynamically generating predictive models of workflow execution, in: 8th Workshop on Workflows in Support of Large-Scale Science (WORKS), 2013, pp. 77–87. doi:10.1145/2534248.2534256.
[25] A. M. Chirkin, et al., Execution time estimation for workflow scheduling, in: 9th Workshop on Workflows in Support of Large-Scale Science (WORKS), 2014, pp. 1–10.
[26] A. Bala, et al., Intelligent failure prediction models for scientific workflows, Expert Systems with Applications 42 (3) (2012) 980–989.
[27] R. Ferreira da Silva, T. Glatard, F. Desprez, Self-healing of workflow activity incidents on distributed computing infrastructures, Future Generation Computer Systems 29 (8) (2013) 2284–2294. doi:10.1016/j.future.2013.06.012.
[28] R. Ferreira da Silva, T. Glatard, F. Desprez, Controlling fairness and task granularity in distributed, online, non-clairvoyant workflow executions, Concurrency and Computation: Practice and Experience 26 (14) (2014) 2347–2366. doi:10.1002/cpe.3303.
[29] M. Ferro, et al., A proposal to apply inductive logic programming to self-healing problem in grid computing: How will it work?, Concurrency and Computation: Practice and Experience 23 (17) (2011) 2118–2135.
[30] R. Seiger, S. Huber, P. Heisig, U. Assmann, Enabling self-adaptive workflows for cyber-physical systems, in: International Workshop on Business Process Modeling, Development and Support, Springer, 2016, pp. 3–17.
[31] S. W. Sung, J. Lee, I.-B. Lee, Process identification and PID control, John Wiley & Sons, 2009.
[32] K. J. Aström, R. M. Murray, Feedback systems: an introduction for scientists and engineers, Princeton university press, 2010.
[33] J. G. Ziegler, et al., Optimum settings for automatic controllers, trans. ASME 64 (11).
[34] S. Srinivasan, G. Juve, R. Ferreira da Silva, K. Vahi, E. Deelman, A cleanup algorithm for implementing storage constraints in scientific workflow executions, in: 9th Workshop on Workflows in Support of Large-Scale Science, WORKS'14, 2014, pp. 41–49. doi:10.1109/WORKS.2014.8.
[35] W. Chen, E. Deelman, Workflow overhead analysis and optimizations, in: Proceedings of the 6th workshop on Workflows in support of large-scale science, ACM, 2011, pp. 11–20. doi:10.1145/2110497.2110500.
[36] A. Mandal, P. Ruth, I. Baldin, R. Ferreira da Silva, E. Deelman, Toward prioritization of data flows for scientific workflows using virtual software defined exchanges, in: First International Workshop on Workflow Science (WoWS 2017), 2017, pp. 566–575. doi:10.1109/eScience.2017.92.
[37] R. Ferreira da Silva, G. Juve, E. Deelman, T. Glatard, F. Desprez, D. Thain, B. Tovar, M. Livny, Toward fine-grained online task characteristics estimation in scientific workflows, in: 8th Workshop on Workflows in Support of Large-Scale Science, WORKS '13, 2013, pp. 58–67. doi:10.1145/2534248.2534254.
[38] . G. P. Consortium, et al., A global reference for human genetic variation, Nature 526 (7571) (2012) 68–74.
[39] 1000genome workflow, https://github.com/pegasus-isi/1000genome-workflow.
[40] E. Deelman, et al., Pegasus, a workflow management system for science automation, Future Generation Computer Systems 46 (0) (2015) 17–35. doi:10.1016/j.future.2014.10.008.
[41] Populations - 1000 genome, http://1000genomes.org/category/population.
[42] G. Juve, B. Tovar, R. Ferreira da Silva, D. Krol, D. Thain, E. Deelman, W. Allcock, M. Livny, Practical resource monitoring for robust high throughput computing, in: Workshop on Monitoring and Analysis for High Performance Computing Systems Plus Applications, 2015. doi:10.1109/CLUSTER.2015.115.
[43] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, R. Buyya, Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Software: Practice and Experience 41 (1) (2011) 23–50. doi:10.1002/spe.995.
[44] R. Ferreira da Silva, W. Chen, G. Juve, K. Vahi, E. Deelman, Community resources for enabling research in distributed scientific workflows, in: 10th IEEE International Conference on e-Science (eScience 2014), 2014. doi:10.1109/eScience.2014.44.
[45] Pid simulator, https://github.com/rafaelfsilva/pid-simulator.
[46] F. Haugen, Ziegler-nichols' closed-loop method, Tech. rep., TechTeach (2010).
[47] H. Casanova, S. Pandey, J. Oeth, R. Tanaka, F. Suter, R. Ferreira da Silva, WRENCH: A Framework for Simulating Workflow Management Systems, in: 13th Workshop on Workflows in Support of Large-Scale Science (WORKS'18), 2018, pp. 74–85. doi:10.1109/WORKS.2018.00013.