Online Self-Healing Control Loop to Prevent and Mitigate Faults in Scientific Workflows

Online Self-Healing Control Loop to Prevent and Mitigate Faults in Scientific Workflows

Rafael Ferreira da Silva[a,*], Rosa Filgueira[b], Ewa Deelman[a], Erola Pairo-Castineira[c], Ian Overton[c], Malcolm Atkinson[b]

[a]*University of Southern California, Information Sciences Institute, Marina del Rey, CA, USA*
[b]*School of Informatics, University of Edinburgh, Edinburgh EH8 9LE, UK*
[c]*MRC Human Genetics Unit, MRC IGMM, University of Edinburgh, Edinburgh, 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. In spite of many success stories, a key challenge for running workflow in distributed systems is failure prediction, detection, and recovery. In this paper, we present a novel online self-healing framework, where failures are predicted before they happen, and are mitigated when possible. The proposed approach is to use control theory developed as part of autonomic computing, and in particular apply the *proportional-integral-derivative* controller (PID controller) control loop mechanism, which is widely used in industrial control systems, to mitigate faults by adjusting the inputs of the mechanism. The PID controller aims at detecting the possibility of a fault 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 the Big Data era—data footprint and memory usage. We define, implement, and evaluate PID controllers to autonomously manage data and memory usage of a bioinformatics workflow that consumes/produces over 4.4TB of data, and requires over 24TB of memory to run all tasks concurrently. Experimental results indicate that workflow executions may significantly benefit from PID controllers, 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 control loop, and faults are detected and mitigated far in advance.

Keywords: Scientific workflows, Self-healing, Fault detection and handling

1. Introduction

Modern science often requires the processing and analysis of vast amounts of data in search of postulated phenomena, and the validation of core principles through the simulation of complex system behaviors and interactions. This is the case in fields such as astronomy, bioinformatics, physics, climate and ocean modeling [1]. In order to support the computational and data needs of today's science, new knowledge must be gained on how to deliver the growing capabilities of the national cyberinfrastructure and more recently commercial cloud to the scientist's desktop in an accessible, reliable, and scalable way. Scientific workflows have emerged as a flexible representation to declaratively express complex such applications with data and control dependencies, and have become mainstream to support computational sciences. One of the challenges in workload management in distributed system is failure prediction, detection, and recovery, both at the application and resource-level. Failures affect the turnaround time of the applications, and that of the greater analysis and therefore the productivity of the scientists that depend on the power of distributed computing to do their work.

In this work, we investigate how the *proportional-integral-derivative* controller (PID controller) control loop mechanism, which is widely used in industrial system, can be applied to predict and prevent failures on end-to-end workflow executions across a distributed, heterogeneous computational environment. The basic idea behind a PID controller is to 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 current rate of change. The main advantage of using a PID controller is that the control loop mechanism progressively monitors the evolution of the workflow execution, detecting possible faults far in advance, and when needed performs actions to lead the execution to a steady-state.

The main contributions of this paper include:

1. An online self-healing control loop using PID controllers to prevent and mitigate faults on workflow executions under unknown conditions;

2. The evaluation of the proposed control loop to tackle two major problems of the Big Data era: data footprint and

*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.filgueira@ed.ac.uk` (Rosa Filgueira), `deelman@isi.edu` (Ewa Deelman), `Erola.Pairo-Castineira@igmm.ed.ac.uk` (Erola Pairo-Castineira), `ian.overton@ed.ac.uk` (Ian Overton), `mpa@staffmail.ed.ac.uk` (Malcolm Atkinson)

memory usage;

3. The characterization of a bioinformatics workflow, which consumes/produces over 4.4TB of data, and requires over 24TB of memory;

4. An experimental evaluation via simulation to demonstrate the feasibility of the proposed self-healing process using simple PID controllers; and

5. 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.

The remainder of this paper is organized as follows. Section 2 gives an overview of related work. Section 3 describes the general self-healing process and presents the concepts of PID, 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 controllers to improve error detection and handling. Section 7 summarizes our results and identifies future work.

2. Related Work

Several strategies and techniques were developed to detect and handle failures during scientific workflow executions [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. Nevertheless, they are either offline, i.e. cannot be computed during the execution of the application, and/or make strong assumptions about resource and application characteristics. Task resubmission [13, 14] and task replication [10, 15] are two of the most common online techniques for handling failures. They are used to increase the probability of having a successful execution in another computing resource, however they should be used sparingly to avoid overloading the execution platform [16]. In previous works, we proposed a pioneer autonomic method described as a MAPE-K loop (Monitoring, Analysis, Planning, Execution, and Knowledge) [17] to cope with online non-clairvoyant workflow executions faults on grids [18, 19, 20], where unpredictability is addressed by using a-priori knowledge extracted from execution traces, detailed monitoring, and analysis of the execution behavior. Online problems are addressed by periodic monitoring updates. Instances are modeled as Fuzzy Finite State Machines (FuSM), where an external curative process determines state degrees of membership. Degrees of membership are computed from metrics assuming that faults have outlier performance. Based on fault degrees, the healing process identifies severity levels of faults using thresholds determined from execution traces. A specific set of actions is then selected from association rules among fault levels. Although this is the first work on self-healing of workflow executions in online and unknown conditions, 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).

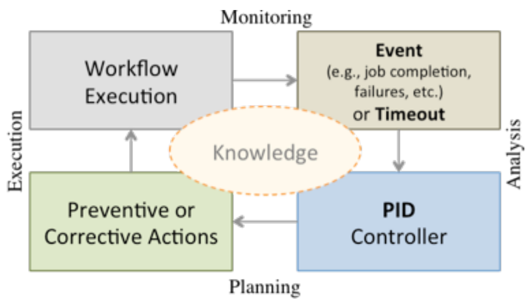


Figure 1: Overview of the MAPE-K loop.

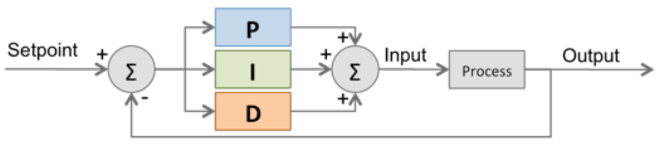


Figure 2: General PID control system loop.

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

3. General Healing Process

In this work, we present a novel self-healing learning 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 preventive 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 on the PID controller. If the controller detects that the system is moving towards an unstable state, the controller will trigger actions to prevent or mitigate faults.

3.1. PID Controllers

The keystone component of the healing process is the *proportional-integral-derivative* controller (PID controller) [21] control loop mechanism, which is widely used in industrial control systems, to mitigate faults by adjusting the process control inputs. Examples of such systems are the ones where the temperature, pressure, or the flow rate, need to be controlled. In such scenarios, the PID controller aims at detecting the possibility of a fault far enough in advance so that an action can be performed to prevent it from happening.

Figure 2 shows the general PID control system loop. The *setpoint* 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.

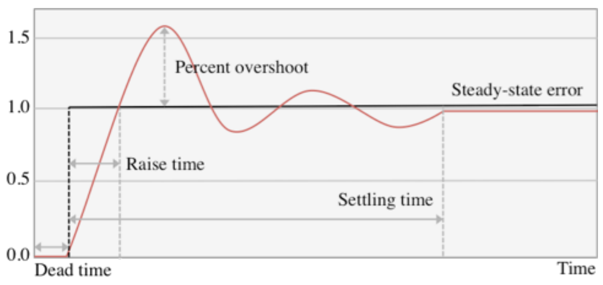


Figure 3: Response of a typical PID closed loop system.

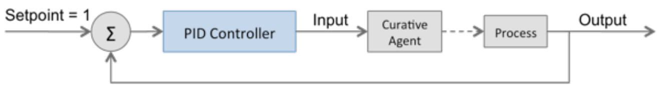


Figure 4: General PID control system loop of our process.

The control system performance is measured through a step function as a *setpoint* command variable, and the response of the process variable. The response is quantified by measuring defined waveform characteristics as shown in Figure 3. Raise 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.

Process variables (output) are determined by fault-specific metrics quantified online. The *setpoint* is constant and defined as 1. The PID controller for the self-healing process proposed in this work provides an input value for a *Curative Agent*, which determines whether an action should be performed (Figure 4). Negative input values mean the control system is raising too fast and may tend to the overshoot state (i.e., a faulty state), therefore preventive or corrective actions should be performed. Actions may include task pre-emption, task resubmission, task clustering, task cleanup, storage management, etc. In contrast, positive input values mean that the control system is smoothly rising to the steady state. The control signal $u(t)$ (output) is defined as follows:

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

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 by 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 [22] for tuning the PID controllers.

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 1: Ziegler-Nichols tuning, using the oscillation method [23]. Note that 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.

Ziegler-Nichols closed loop method. 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: (1) the period T_u of the oscillation frequency at the stability limit, and (2) the gain margin K_u for loop stability. In this method, the K_i and K_d gains are first set to zero. Then, the proportional gain K_p is increased until it reaches the ultimate gain K_u, at which the output of the loop starts to oscillate. K_u and the oscillation period T_u are then used to set the gains according to the values described in Table 1 [23]. In this paper, we will use the Ziegler-Nichols method to determine tuned gain values for each PID controller used to prevent and mitigate faults in workflow executions.

3.2. Metrics

Fault-specific metrics used to set process variables in the PID controller rely on historical information (mostly workload traces) to define fault degrees. More specifically, they rely on workflow structure metrics as shown in [5, 12]. The metrics target the optimization of quality of service (QoS) factors of a workflow execution such as the workflow makespan (time between the start and finish of all workflow tasks), the number of predicted faults, the number of detected faults, and the number of unhandled faults during a workflow runs.

In our previous work [18, 19, 20], faults are classified into degrees of membership computed from metrics assuming that faults have outlier performance. Although a fault could fall into several degrees, the mechanism is limited to the use of corrective actions once a fault is detected, i.e., no preventive actions are performed. This approach can be represented as a PI controller, where the derivative response (prediction of future errors) is not considered. In this work, we pay particular attention to the definition of metrics to compute the derivative response (D), and their associated preventive actions. The main advantage of using a PID controller is that the healing mechanism progressively monitors the evolution of the workflow execution, detecting possible faults far in advance, and when needed performs actions to lead the execution to a steady-state.

4. Defining Control Loops

In the proposed self-healing process, a PID controller is defined and used for each possible-future fault identified from workload traces (historical data). Therefore, our healing process is composed of a set of independent PID controllers. In some cases, particular type of faults cannot not be modeled as a full PID controller, i.e., 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 (*proportional-integral*) controller, for instance, can be defined and deployed. In this paper, we demonstrate the feasibility of the use of PID controllers as part of a self-healing control loop by tackling two major issues of workflow executions: data and memory footprint.

4.1. Workflow Data Footprint and Management

In the era of Big Data Science, applications are producing and consuming ever-growing data sets in domains such as astronomy, physics, climate science, earthquake science, biology and others. A run of scientific workflows that manipulate 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 fault, data cleanup tasks are often automatically inserted into the workflow by the workflow management system [2], or the number of concurrent executions is limited to prevent data usage overflow. Cleanup tasks remove data sets that are no longer needed by downstream tasks but nevertheless they add an important overhead to the workflow execution. Thus, these tasks should be strategically positioned in a workflow in order to minimize the disk space usage while adding the minimum overhead to the execution.

PID Controller. The controller process variable (output) is defined as the ratio of the estimated disk space required by current tasks in execution, and the actual available disk space. The system is in a *non-steady* state if the total amount of disk space consumed is above (overshoot) a predefine threshold (*setpoint*), or the amount of used disk space is below the defined optimal capacity. Figure 5 shows an example of a response curve of a simple PID controller for the management of disk usage during a workflow execution. The raise time is the amount of time required by the system to gather information about the user's disk quota and actual disk usage. The proportional (P) response is computed as the error between the desired amount of disk space usage, and the actual used disk space; the integral (I) response is computed from the sum of the disk usage errors for all task executions; and the derivative (D) response is computed as the difference between the current and the previous disk overflow (or underutilization) error values.

Corrective Actions. The output of the PID controller (control signal $u(t)$, Equation 1) indicates whether the system is in a non-steady state. Negative values indicate that the current disk usage trespass the threshold of the minimum required available disk space. In contrast, positive values indicate that the current running tasks do not use the desired disk capacity. For values of $u(t) < 0$, (1) data cleanup tasks can be added to the current workflow execution to remove unused intermediate data (adding cleanup tasks may imply rearranging the priority of all tasks in the queue), or (2) tasks can be preempted due to the inability to remove data–the inability of cleaning up data may

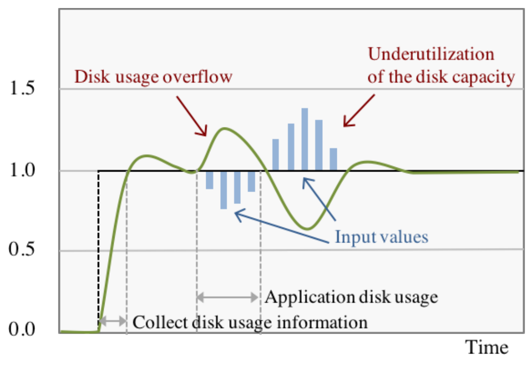


Figure 5: Example of a response curve from a PID controller for the management of the workflow data footprint.

lead the execution to an unrecoverable state, and thereby to a failed execution. Otherwise (for $u(t) > 0$), the number of concurrent task executions may be increased.

4.2. Workflow Memory Usage and Management

Large scientific computing applications rely on complex workflows to analyze large volume of data. These jobs are often running in HPC resources (clusters/cloud) over thousands of CPU cores and simultaneously performing data accesses, data movements, and computation, dominated by memory-intensive operations, which are not designed to take full advantage of these HPC capabilities. The performance of those memory-intensive operations (e.g., reading a large volume of data from disk, decompressing in memory massive amount of data or performing a complex calculation which generates large datasets, etc.) are quite often limited to the memory capacity of the resource where the application is being executed. Therefore, if those operations overflow the physical memory limit it can result to application's performance degradation or application's failure. Typically, the application end-user is responsible for optimizing the application, modifying the code if it is needed for complying the amount of memory that can be used on that resource, which implies that scientists get distracted by details of the computing resource they use instead of being focus on their research.

This work addresses the memory challenge proposing an *in-situ* analysis of memory, to adapt the number of concurrent tasks executions according to the memory usage required by an application at runtime.

PID Controller. The controller process variable (output) is defined as the ratio of the estimated total peak memory usage required by current tasks in execution, and the actual available memory. The system is in a *non-steady* state if the amount of available memory is below the *setpoint*, or if the current available memory is above the minimum required. Figure 6 shows an example of a response curve of a simple PID controller for the management of memory usage during a workflow execution. The raise time is the amount of time required by the system to gather information about the current memory usage, and isolate the amount of memory consumed by the workflow tasks,

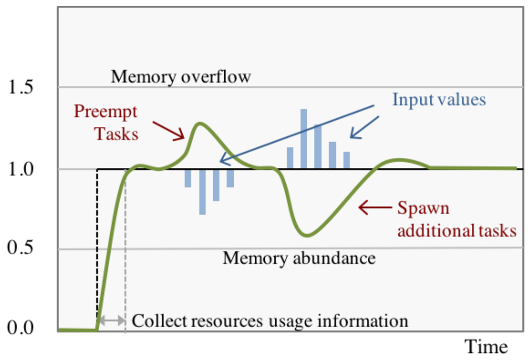


Figure 6: Example of a response curve from a PID controller for the management of the workflow memory footprint.

and the memory consumed by system-level applications. The proportional (P) response is computed as the error between the memory consumption *setpoint* for running tasks, and the actual memory usage; the integral (I) response is computed from the sum of the memory usage errors for all task executions; and the derivative (D) response is computed as the difference between the current and the previous memory overflow (or underutilization) error values.

Corrective Actions. Negative values for the control signal $u(t)$ indicate that the ensemble of running tasks are leading the system to an overflow state, thus some tasks should be preempted to prevent the system to run out of memory. For positive $u(t)$ values, the memory consumption of current running tasks is below a predefined memory consumption *setpoint*. Therefore, the workflow management system may spawn additional tasks for concurrent execution.

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. In this paper, the metrics (and thereby the PID controller) defined to tackle data and memory issues perform independent actions that do not (directly) affect the workflow execution performance. The analysis of simultaneous multiples control loops is out of the scope of this paper, although we plan to evaluate such conditions in the future.

5. Experimental Evaluation

5.1. Scientific Workflow Application

Patterns of mutation in human populations can provide a cipher to interpret the human genome's 3 billion letters in the context of health and disease. For example, knowledge about the prevalence and co-occurrence of mutations can illuminate new biology and advance the design of more effective therapies [24, 25]. The 1000 genomes project provides a reference for human variation, having reconstructed the genomes of 2,504 individuals across 26 different populations to energize these approaches [24]. 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 has been designed and implemented as a scientific workflow (Figure 7) using the Pegasus workflow management system [26]. The purpose of this workflow is to fetch, parse, and analyze data from the 1000 genomes project, and it is composed of five different tasks[1] described as follows:

Individuals. This task fetches and parses the Phase 3 data[2] [24] from the 1000 genomes project by chromosome, downloading and decompressing the corresponding chromosome file. These files list all of Single nucleotide polymorphisms (*SNPs*) variants in that chromosome and which individuals have each one. *SNPs* are the most common type of genetic variation among people, and are the ones we consider in this work. An `individual` task creates output files for each individual of *rs numbers* [3], where individuals have mutations on both alleles.

Populations. The 1000 genome project has 26 different populations, which are part of our study, from many different locations around the globe. Detailed information of the 26 populations can be found at [27]. A `population` task downloads a file per population selected. For this work, we have used five super populations: African (*AFR*), Mixed American (*AMR*), East Asian (*EAS*), European (*EUR*), and South Asian (*SAS*). Furthermore, we have also selected *ALL* population, which means that all individuals from the latest release are considered.

Sifting. A `sifting` 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. In other words, *SIFT* classifies the substitutions as tolerated or deleterious. Therefore, *VEP* determines the effect of an individual variants on genes, transcripts, and protein sequence, as well as regulatory regions [28]. For each chromosome, the `population` task downloads and decompresses the corresponding *VEP*, and selects only the *SNPs* variants that has a *SIFT* score, recording in a file (per chromosome) the *SIFT* score and the *SNPs* variants ids, which are: (1) *rs number*, (2) *ENSEMBL GEN ID* (a gene annotation provided by Ensembl project [29]), and (3) *HGNC ID* (the HUGO Gene Nomenclature Committee (HGNC) [30] is the only worldwide authority that assigns standardized nomenclature to human gene).

Pair_Overlap_Mutations. This task measures the overlap in mutations (also called *SNPs* variants) among pairs of individuals by population and by chromosome. Considering two individuals, if both individuals have a mutation in a given *SNPs* then they are considered to have a mutation overlap. A

[1]The workflow's tasks source code are available online at https://github.com/rosafilgueira/Mutation_Sets.

[2]The sequencing of the 1000 genome project was carried out in phases one and three of the main project. In this work, we have used only the data from the third phase.

[3] Reference *SNP* cluster ID (*rs numbers*) is an identifier used by researchers and databases to refer to specific SNPs variant.

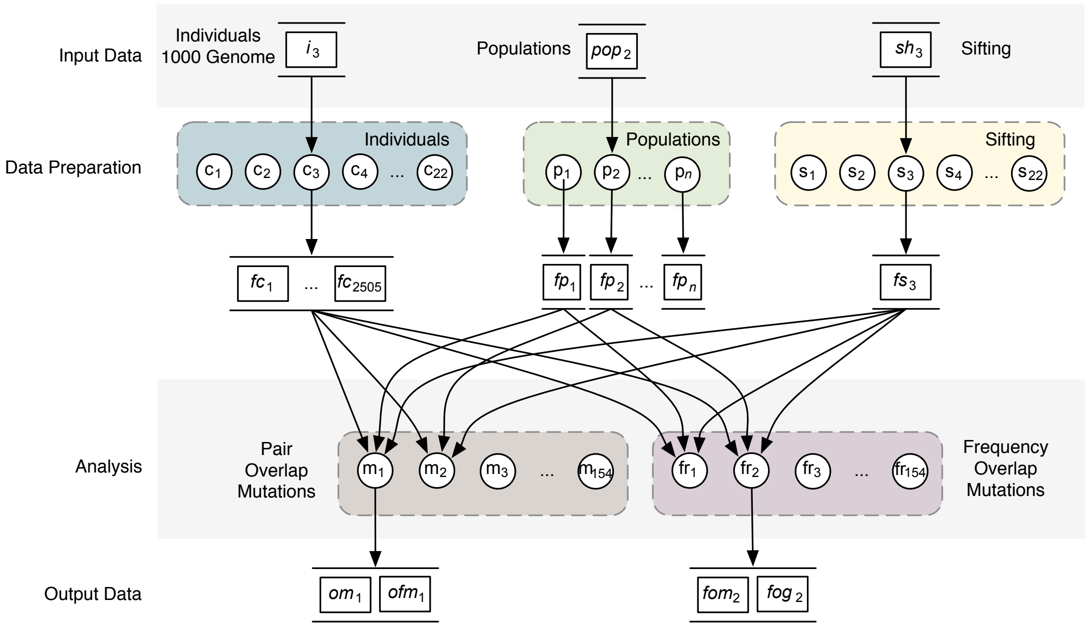


Figure 7: Overview of the 1000 genome sequencing analysis workflow.

`pair_overlap_mutation` task performs several correlations by using different configurations: (1) modifying the number of pair of individuals (all individuals, half of individuals, and 100 random individuals), and (2) modifying the number of *SNPs* variants (only the *SNPs* variants with a score less than 0.05 and all the *SNPs* variants). As an input, this task requires (1) all the individuals files generated by the `Individual` task for the particular chromosome being analyzed; (2) the file generated by the `sifting` task for that particular chromosome; and (3) the population file generated by the `population` task according with the selected population. For performing those analyses, the task computes an array (per chromosome, population, and *SIFT* level selected) which has as many entries as individuals, and each entry contains the list of *SNPs* variants that each individual has according with the *SIFT* score select. These analyses have been repeated 100 times. The results of these correlations are stored as text files and plot figures, producing a total of 12 files and figures per population, per chromosome, and per run. Furthermore, two colour distribution maps are computed for visualizing the correlation matrix of all the individuals among themselves (with and without taking account the *SIFT* scores), by population, chromosome, and number of run.

Frequency_Overlap_Mutations. This tasks measures the frequency of overlapping in mutations by selecting a number of random individuals (in this work we have selected 26 random individuals), and selecting all *SNPs* variants without taking into account their *SIFT* scores. This analysis has been repeated 100 times. This task requires the same input files as for `pair_overlap_mutation`, and it also computes an array (per chromosome and population), which has as many entries as individuals, and each entry contains the list of *SNPs* variants belonging to each individual. For each run, the task randomly selects a group of 26 individuals from this array and computes the number of overlapping in mutations among that group (given a *SNPs* variant the task counts the total of individuals that has that variant in that group). Then, the task computes the frequency of mutations that has the same number of overlapping in mutations (e.g., if *variant 1*, *variant 12*, *variant 31*, and *variant 35* are the only variants which presents 3 overlapping among individuals, we could say that the frequency of 3 overlapping among that group of individuals is 4 mutations). The results are stored as a text file and as a histogram plot per population, per chromosome, and per run.

5.2. Workflow Characterization

We profiled the 1000 genome sequencing analysis workflow using the Kickstart [31] profiling tool. Kickstart monitors and records task execution in scientific workflows. It captures fine-grained profiling data such as 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 some 4,000 cores with up to 2 TB of memory. Tasks are scheduled using the Open Grid Scheduler batch system onto Scientific Linux 7. 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 2 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 of disk (174GB in average per task) and memory (411GB in average per task). The total workflow data 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 allow us to barely run up to 3 concurrent `individual` tasks in the large node, and some of the remaining tasks in smaller nodes. Therefore, data and memory management is crucial to perform a successful run of the workflow execution, while increasing user satisfaction.

In online environments, one simple (and typically used) approach for estimating the runtime, data footprint, and memory of future workflow tasks is to use the mean values of previous executions of similar tasks. Most of the memory peak tasks shown in Table 2 have small standard deviation values compared to the mean, thus the estimation of these requirements based on the mean value would yield reasonable accuracy. On the other hand, runtime and data footprint standard deviation values are too high to have reasonable accuracy using the mean. In the experiments, we compare the efficiency between an online method using the mean to estimate task requirements, and an online method where task scheduling is driven by the PID controllers.

5.3. *Experiment Conditions*

The experiments use trace-based simulation. We developed an activity-based simulator that implements the self-healing loop described in this paper. 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. Additionally, we assume a shared network file system among the nodes with total capacity of 500GB.

We use a random based policy with task preemption for task scheduling that traverses the workflow in a breadth-first search manner. To avoid unrecoverable faults (workflow failure) due to run out of disk space, we implemented a data cleanup mechanism to remove data that are no longer required by downstream tasks [2]. Nevertheless, data cleanup tasks are only triggered if the maximum storage capacity is reached. In this case, all running tasks are preempted, the data cleanup task is executed, and the workflow resumes its execution. Note that this mechanism may add an important overhead to the workflow execution.

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 used). Therefore, we do not attempt to optimize task preemption (which criteria should be used to select tasks for removal) since our goal is to demonstrate the feasibility of the approach with simple use case scenarios.

PID Controllers. 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 overshoots or underutilization of the system. In order to accommodate overshoots, we define our settling point as 80% of the maximum total capacity (for both storage and memory usage), and a steady-state error of 5%. As aforementioned, for this experiment we assume $K_p = K_i = K_d = 1$ to demonstrate the feasibility of the approach regardless the use of tuning methods. Note that a single PID controller is used to manage disk usage (shared network file system), while an independent memory controller is deployed for each computing node. The controller input value indicates the amount of disk space or memory that should be consumed by tasks. If the input value is positive, more tasks are scheduled (resp. tasks are preempted). The control loop process uses then the mean values presented in Table 2 to estimate the number of tasks to be scheduled/preempted. Note that due to the high values of standard deviation, estimations are not accurate. Task characteristics estimation is beyond the scope of this work, and sophisticated methods to provide accurate estimates can be found in [32, 33]. However, this work intend to demonstrate that even using inaccurate estimation methods, PID controllers can cope with the estimation error.

Reference Workflow Execution. In order to measure the efficiency of our online method under unknown conditions, we compare the workflow execution performance (in terms of the turnaround time to execute all tasks, a.k.a. *makespan*) to a reference workflow—computed offline under known conditions, i.e., all requirements (e.g., runtime, disk, memory) are known in advance. Due to the randomness inherent to the scheduling algorithm used in this work, we performed several runs for the reference workflow, which yields an averaged makespan of 382,887.7s (~106h), with a confidence level of 95%.

5.4. *Experimental Results and Discussion*

This subsection shows the experimental results of using PID controllers to prevent and circumvent faults in the 1000 genome sequencing analysis workflow. We have conducted workflow runs with three different types of controllers: (`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` (Figure 8). We have performed several runs of each configuration to produce results with a confidence level of 95%. The simulator code, as well as the workflow description are available as part of the research object of this paper [34].

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 2: Execution profile of the 1000 genome sequencing analysis workflow.

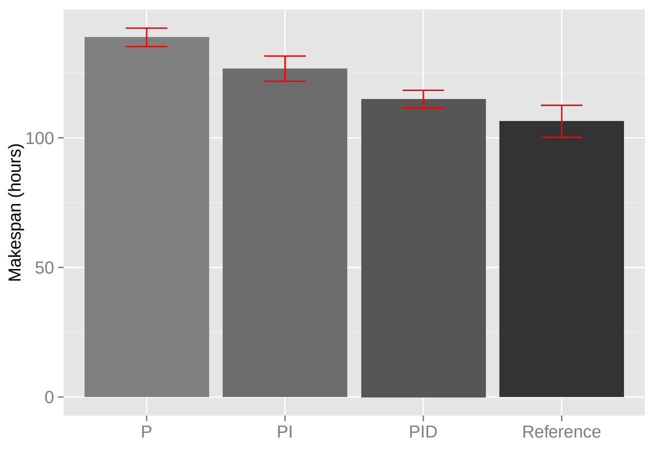


Figure 8: Average workflow makespan for different configurations of the controllers: (`P`) proportional, (`PI`) proportional-integral, and (`PID`) proportional-integral-derivative. Under online and unknown conditions, the `PID` controller produces the lowest slowdown (1.08), while the slowdown for the `PI` and `P` controllers are 1.19 and 1.30, respectively. `Reference` denotes the makespan of a reference workflow execution computed offline and under known conditions.

Overall makespan evaluation. Figure 8 shows the average makespan (in hours) for the three configurations of the controller and the reference workflow execution. The degradation of the makespan when using controllers is expected due to the online and unknown conditions (no information about the tasks is know in advance). In spite of the mean does not provide accurate estimates, the use of a control loop mechanism diminishes this effect. Moreover, 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 `PID` controllers outperforms executions using only the proportional (`P`) or the `PI` controller. The `PID` controller slows down the application of 1.08 (average makespan 413,850s), while the application slowdown is 1.19 and 1.30 for the `PI` (average makespan 456,083s) and `P` (average makespan 499,538s) controllers, respectively. This result suggests that the derivative component (prediction of future errors) has significant impact on the workflow executions, and that the accumulation of past errors (integral component) also has important contribution 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.

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

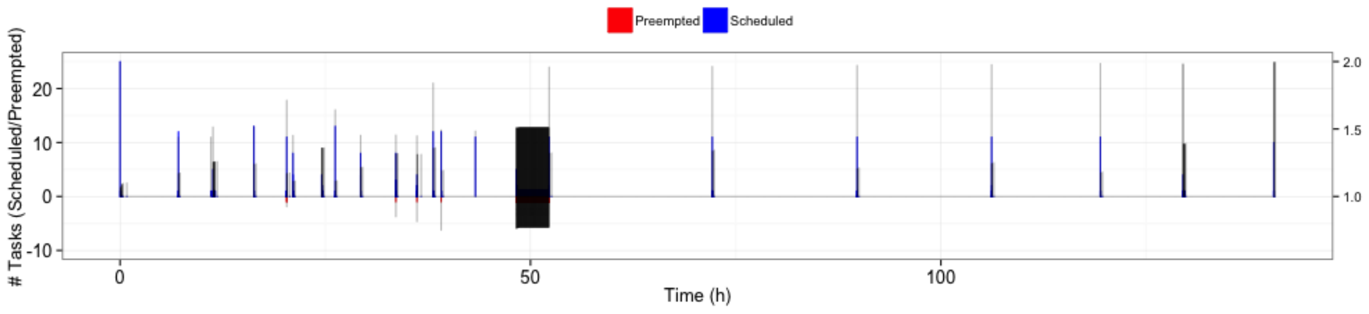
Table 3: Average number of tasks preempted and cleanup tasks executed per workflow run for the `P`, `PI`, and `PID` controllers.

Due to the randomness inherent to the scheduling algorithm, we did not perform runs where mixed `PID`, `PI`, and `P` controllers were part of the same simulation, since it would be very difficult to determine the influence of each controller.

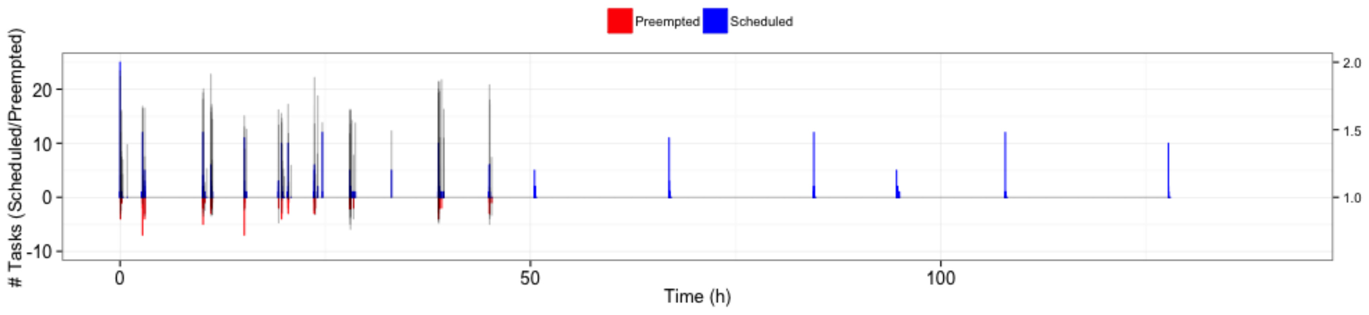
Data footprint. Figure 9 shows the time series of the number of tasks scheduled or preempted during the workflow executions. For each controller configuration, we present a single execution which the makespan is the closest to the average makespan value shown in Figure 8. Task preemption is represented 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 of the controller input value (black line) for disk usage during the workflow execution. Recall that *positive* input values ($u(t) > 0$, Equation 1) trigger task scheduling, while *negative* input values ($u(t) < 0$) trigger task preemption.

The proportional controller (`P`, Figure 9a) is limited to the current error, i.e., the amount of disk space that is over/underutilized. Since the controller input value is strictly proportional to the error, there is a burst on the number of tasks to be schedule at the beginning of the execution. This bursty pattern and the nearly constant variation of the input value lead the system to an inconsistent state, where the remaining tasks to be scheduled cannot lead the controller within the steady-state. Consequently, tasks are constantly scheduled and then preempted. In the example scenario shown in Figure 9a, this process occurs during about 4h, and performs more than 6,000 preemptions. Table 3 shows the average number of preemptions and cleanup tasks occurrences per workflow execution. In 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 (mostly influenced by the number of task preemptions), which is observed from the low average application speedup of 1.18.

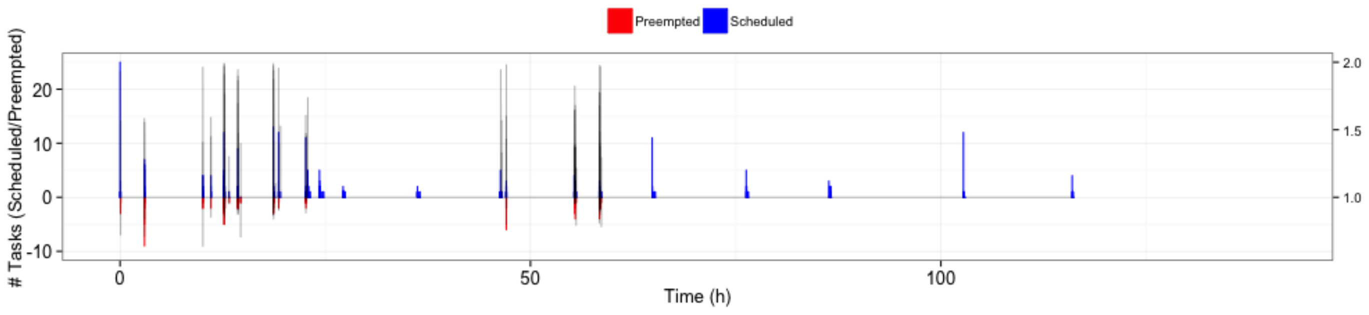
The proportional-integral controller (`PI`, Figure 9b) aggregates the cumulative error when computing the response of the controller. As a result, the bursty pattern is smoothed along the



(a) Proportional Controller (P)



(b) Proportional-integral Controller (PI)



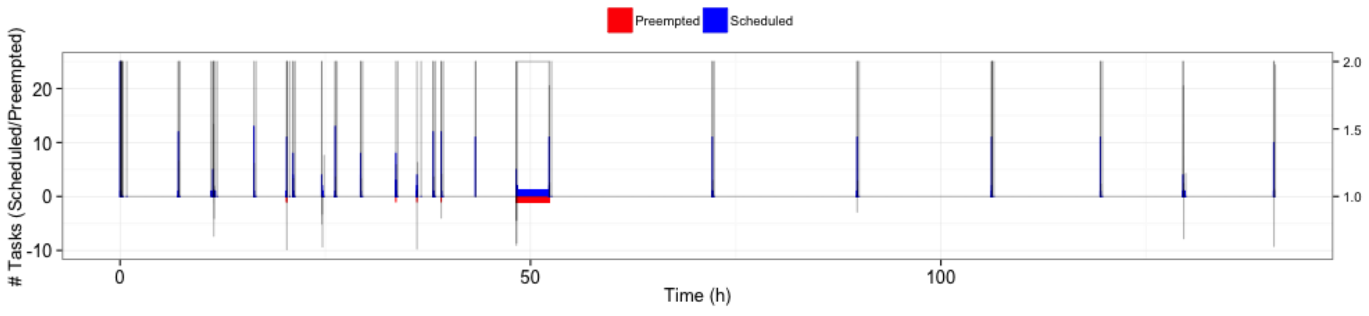
(c) Proportional-integral-derivative Controller (PID)

Figure 9: *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 (black line) during the workflow execution. Note that 1.0 is the setpoint, i.e., no action is required.

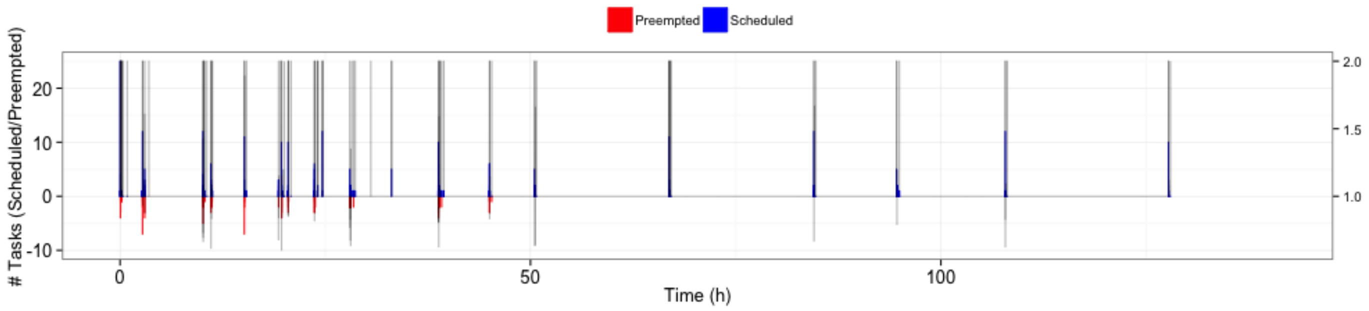
execution, and task concurrency is increased. The cumulative error tend to increase the response of the PI controller at each iteration (both positively or negatively). Thus, task preemption occurs earlier during execution. On the other hand, this behavior mitigates the *vicious circle* present in the P controllers, and 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, which is due to the higher level of concurrency among task executions.

The proportional-integral-derivative controller (PID, Figure 9c) gives significant importance to 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 increase) the previous error trend. In this case, 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 9c is much less than the number triggered by the PI controller: the average number of preempted tasks is 73, and only 4 cleanup tasks in average are spawned (Table 3).

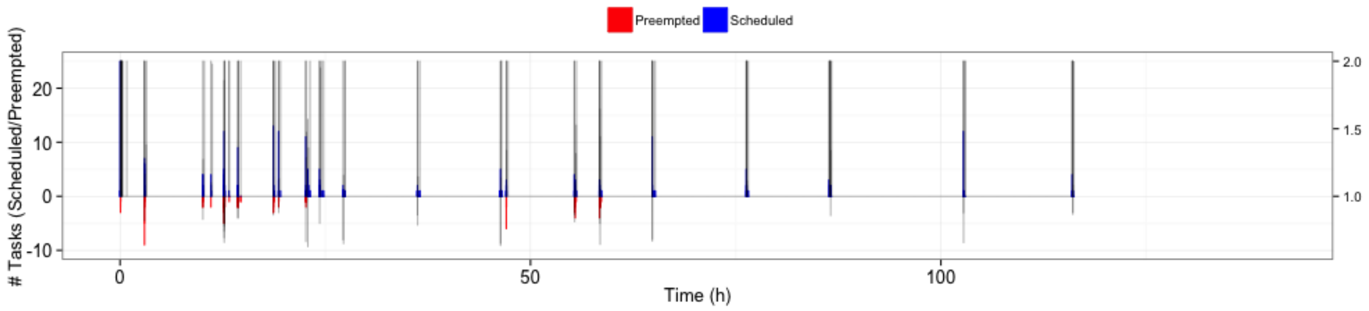
Memory Usage. Figure 10 shows the time series of the number of tasks scheduled or preempted during the workflow executions. Similarly, for each controller configuration, we present a single execution representing the average makespan value shown in Figure 8. The right y-axis shows the step response of the controller input value (black 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 (total of 315 tasks).



(a) Proportional Controller (P)



(b) Proportional-integral Controller (PI)



(c) 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 lifespan of a workflow execution (left *y*-axis). The right *y*-axis represents the step response of the controller input value (black line) during the workflow execution. Note that 1.0 is the setpoint, i.e., no action is required. 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.

The total memory allocation required to run all these tasks is over 4TB, which might lead the system to memory overflow states.

When using the proportional controller (P, Figure 10a), most of the actions are triggered by the data footprint controller (Figure 9a). As aforementioned, memory does not become an issue when only the proportional error is accounted, since task execution is nearly sequential (low level of concurrency). As a result, only a few tasks (in average less than 5) are preempted due to memory overflow. Note that the process of constant task scheduling (around 50h of execution) is strongly influenced by the memory controller. Also, the step response shown in Figure 10a 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. 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 triggered by controllers should be performed by priority or according to the controller hierarchical level.

The proportional-integral controller (PI, Figure 10b) mitigates this effect, since the cumulative error prevents the controller to trigger repeatedly actions. Observing the step response of the PI memory controller and the PI data footprint controller (Figure 9b), we notice that most of the task preemptions are triggered by the memory controller, particularly in the first quarter of the execution. 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. If there were no memory controllers, the workflow execution would tend to memory overflow, and thereby into a failed state.

The derivative component of the `PID` controller (Figure 10c) acts as a catalyzer to improve memory usage: it decreases the overshoot and the settling time without affecting the steady-state error. As a result, the number of actions triggered by the `PID` memory controller is significantly reduced when compared to the `PI` or `P` controllers.

Although the experiments conducted in this feasibility study considered equal weights for each of the components in a PID 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, and 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 three PID gain parameters (K_p, K_i, and K_d).

6. Tuning PID Controllers

The goal of tuning a PID loop is to make it stable, responsive and to minimize overshooting. However, there is no optimal way to achieve responsiveness without compromising overshooting, 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. The Ziegler-Nichols closed loop method was briefly introduced in Section 3.1, and a detailed explanation of the method can be found in [22]. Thus, in the section we will present how we determine the period T_u, and 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 [35]:

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 1, and use these parameter values in the controller.

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

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

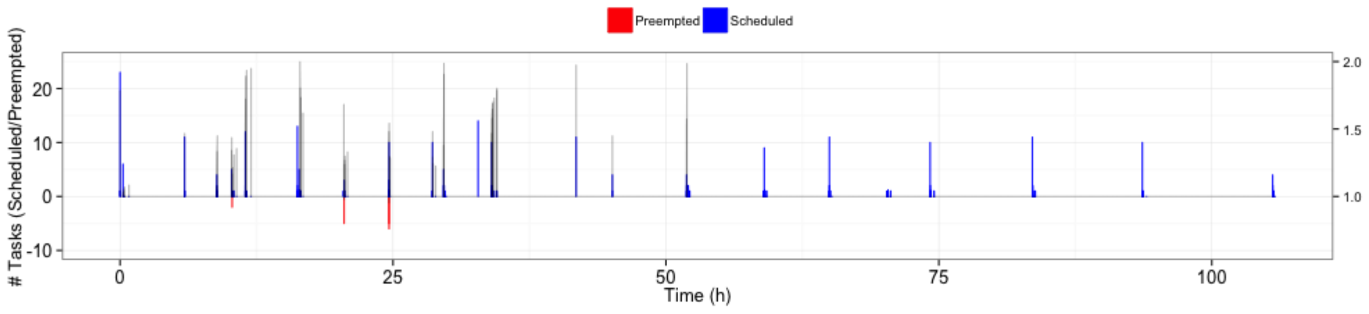
Table 4: Tuned gain parameters (K_p, K_i, and K_d) for both the data footprint and memory usage PID controllers. K_u and T_u are computed using the Ziegler-Nichols method, and represent the ultimate period and critical gain, respectively. The gain parameters are computed using the formulas for the controller parameters in the Ziegler-Nichols' closed loop method (Table 1).

(Table 2). We conducted several runs (O(100)) with the proportional (`P`) controller to compute the period T_u and the gain margin K_u. Table 4 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` controller.

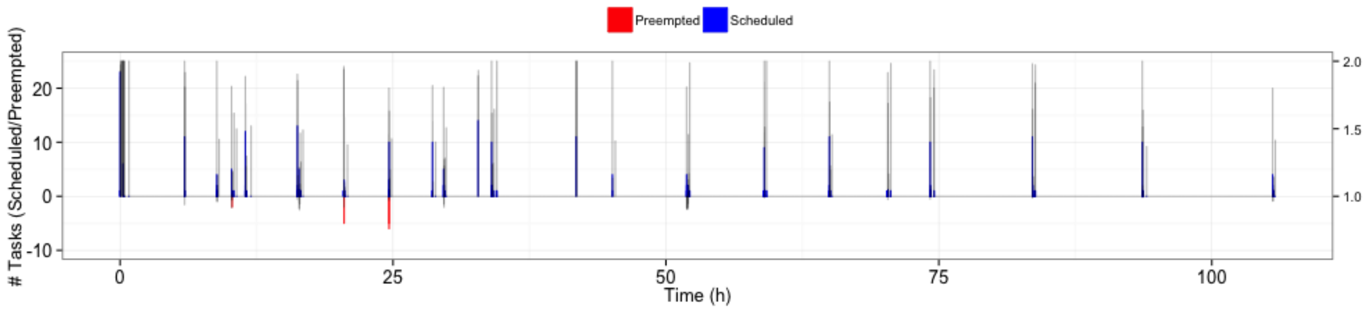
6.2. Experimental Evaluation and Discussion

We have conducted runs with the tuned `PID` controllers for both the data footprint and memory usage. Figure 11 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. Additionally, 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 11a) and the memory usage (Figure 11b), show lower peaks and troughs values during the workflow execution when compared to the `PID` controllers using equal weights for each gain parameter (Figures 9c and 10c, respectively). More specifically, the controller input value is reduced of about 30% for the memory controller attached to a standard cluster. This behavior is attained through the ponderations provided 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), since few preemptions are still triggered.

Although the Ziegler-Nichols 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 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 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 on the user's space (or per workflow engine) to manage single (or a few) workflow executions, ensuring that correctly defined workflow complete with acceptable performance. In addition, the time required to process the current state of the system and decide whether to trigger an action is nearly instantaneous, what favors the use of PID controllers on online and real-time workflow systems. More sophisticated methods (e.g., using machine learning) may provide better approaches to tune the gain parameters. However, they may also add an important overhead to the system.



(a) PID Data Footprint Controller



(b) PID Memory Controller

Figure 11: *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 line) during the workflow execution. Note that 1.0 is the setpoint, i.e., no action is required. 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.

7. Conclusion

In this paper, we have introduced, evaluated, and discussed the feasibility of using simple PID (proportional-integral-derivative) controllers to prevent and mitigate faults in workflow executions. We have presented a general self-healing control loop to constantly monitor the workflow execution, and trigger preventive or corrective actions to avoid leading workflow executions to unrecoverable states. We have then addressed two common faults of today's science applications, data footprint and memory usage (main issues in data-intensive workflows, i.e., Big Data), as use cases to demonstrate the feasibility of the proposed approach. Additionally, our self-healing mechanism works online and under unknown conditions—task requirements are not known in advance.

Experimental results using simple defined control loops (no tuning) show that faults are detected and prevented far in advance, 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 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.

Results of this work open a new venue of research in workflow management systems. Although in this paper PID controllers have been used to prevent and mitigate faults in workflow executions at runtime, this approach could be extended beyond the application level, for example resource provisioning algorithms may use PID controllers to balance the tradeoff between performance and cost in cloud computing.

We acknowledge that PID 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 application and infrastructure levels, to determine to which extent this approach may negatively impact the system. We also plan to share execution traces of the workflow application used in this work, and extend our synthetic workflow generator [36] (that can produce realistic synthetic workflows based on profiles extracted from execution traces) to generate estimates of data and memory usages based on the gathered measurements.

Acknowledgments

This work was funded by DOE contract number #DESC0012636, “Panorama—Predictive Modeling and Diagnostic Monitoring of Extreme Science Workflows”, and the Postdoctoral and Early Career Researcher Exchanges (PECE) fellowship funded by the Scottish Informatics and Computer Science Allience (SICSA) in 2016.

References

[1] I. J. Taylor, E. Deelman, D. B. Gannon, M. Shields, Workflows for e-Science: scientific workflows for grids, Springer Publishing Company, Incorporated, 2014.

[2] 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.

[3] A. Bala, I. Chana, Intelligent failure prediction models for scientific workflows, Expert Systems with Applications 42 (3) (2015) 980–989.

[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) (2013) 170–181.

[5] W. Chen, R. Ferreira da Silva, E. Deelman, R. Sakellariou, Balanced task clustering in scientific workflows, in: IEEE 9th International Conference on eScience, eScience'13, 2013, pp. 188–195. doi:10.1109/eScience.2013.40.

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

[7] Y. Zhang, A. Mandal, C. Koelbel, K. Cooper, Combined fault tolerance and scheduling techniques for workflow applications on computational grids, in: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE Computer Society, 2009, pp. 244–251.

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

[9] W. Cirne, F. Brasileiro, D. Paranhos, L. F. W. Góes, W. Voorsluys, On the efficacy, efficiency and emergent behavior of task replication in large distributed systems, Parallel Computing 33 (3) (2007) 213–234.

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

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

[12] W. Chen, R. Ferreira da Silva, E. Deelman, R. Sakellariou, Using imbalance metrics to optimize task clustering in scientific workflow executions, Future Generation Computer Systems 46 (0) (2015) 69–84. doi:10.1016/j.future.2014.09.014.

[13] 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) (2016) 30.

[14] 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.

[15] 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.

[16] H. Casanova, On the harmfulness of redundant batch requests., in: hpdc, 2006, pp. 255–266.

[17] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50.

[18] 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.

[19] 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.

[20] R. Ferreira da Silva, T. Glatard, F. Desprez, Self-managing of operational issues for grid computing: The case of the virtual imaging platform, in: S. Bagchi (Ed.), Emerging Research in Cloud Distributed Computing Systems, IGI Global, 2015, pp. 187–221. doi:10.4018/978-1-4666-8213-9.ch006.

[21] S. W. Sung, J. Lee, I.-B. Lee, Proportional–integral–derivative control, Process Identification and PID Control (2010) 111–149.

[22] J. G. Ziegler, N. B. Nichols, Optimum settings for automatic controllers, trans. ASME 64 (11).

[23] A. S. McCormack, K. R. Godfrey, Rule-based autotuning based on frequency domain identification, Control Systems Technology, IEEE Transactions on 6 (1) (1998) 43–61.

[24] . G. P. Consortium, et al., A global reference for human genetic variation, Nature 526 (7571) (2015) 68–74.

[25] C. J. Ryan, C. J. Lord, A. Ashworth, Daisy: picking synthetic lethals from cancer genomes, Cancer cell 26 (3) (2014) 306–308.

[26] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, K. Wenger, 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.

[27] Populations - 1000 genome, http://1000genomes.org/category/population.

[28] Variant effect predictor, www.ensembl.org/info/docs/tools/vep.

[29] Ensembl genome browser 84, http://uswest.ensembl.org.

[30] Hugo gene nomenclature committee, http://www.genenames.org/help/symbol-report.

[31] G. Juve, B. Tovar, R. Ferreira da Silva, D. Król, D. Thain, E. Deelman, W. Allcock, M. Livny, Practical resource monitoring for robust high throughput computing, in: 2nd Workshop on Monitoring and Analysis for High Performance Computing Systems Plus Applications, HPCMASPA'15, 2015, pp. 650–657. doi:10.1109/CLUSTER.2015.115.

[32] 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.

[33] R. Ferreira da Silva, M. Rynge, G. Juve, I. Sfiligoi, E. Deelman, J. Letts, F. Würthwein, M. Livny, Characterizing a high throughput computing workload: The compact muon solenoid (CMS) experiment at LHC, Procedia Computer Science 51 (2015) 39–48, international Conference On Computational Science, {ICCS} 2015 Computational Science at the Gates of Nature. doi:10.1016/j.procs.2015.05.190.

[34] Research object: Workflows and pid controllers, http://scitech.isi.edu/ro/pid.

[35] F. Haugen, Ziegler-nichols' closed-loop method, Tech. rep., TechTeach (2010).

[36] R. Ferreira da Silva, W. Chen, G. Juve, K. Vahi, E. Deelman, Community resources for enabling and evaluating research on scientific workflows, in: 10th IEEE International Conference on e-Science, eScience'14, 2014, pp. 177–184. doi:10.1109/eScience.2014.44.