Self-Management of Operational Issues for Grid Computing: The Case of The Virtual Imaging Platform

Rafael Ferreira da Silva
University of Southern California, Information Sciences Institute, USA, rafsilva@isi.edu

Tristan Glatard
CNRS, University of Lyon, CREATIS, INSERM, France, glatard@creatis.insa-lyon.fr
McConnell Brain Imaging Centre, Montreal Neurological Institute, McGill University, Canada

Frédéric Desprez
INRIA, University of Lyon, LIP, UMR CNRS 5668, ENS Lyon, France,
Frederic.Desprez@inria.fr

ABSTRACT

Science gateways, such as the Virtual Imaging Platform (VIP), enable transparent access to distributed computing and storage resources for scientific computations. However, their large scale and the number of middleware systems involved in these gateways lead to many errors and faults. This chapter addresses the autonomic management of workflow executions on science gateways in an online and non-clairvoyant environment, where the platform workload, task costs, and resource characteristics are unknown and not stationary. The chapter describes a general self-management process, based on the MAPE-K loop (Monitoring, Analysis, Planning, Execution, and Knowledge), to cope with operational incidents of workflow executions. Then, this process is applied to handle late task executions, task granularities, and unfairness among workflow executions. Experimental results show how our approach achieves a fair quality of service by using control loops that constantly perform online monitoring, analysis, and execution of a set of curative actions.

Key words: Self-Management, Grid Computing, Scientific Workflow, Science-Gateway, Virtual Imaging Platform, Rafael Ferreira da Silva, Tristan Glatard, Frédéric Desprez, Information Sciences Institute, CREATIS, McConnel Brain Imaging Centre, ENS Lyon

INTRODUCTION

Distributed computing infrastructures such as campus clusters, Grids, and now Clouds have become daily instruments of scientific research. As collections of independent computers linked by a network presented to the users as a single coherent system (e.g. Open Science Grid, XSEDE, EGI, and Amazon EC2), they enable easy collaboration among researchers, enhanced reliability and availability, and high-performance computing (Romanus et al., 2012). Increasingly, these systems are becoming more complex, heterogeneous, and prone to failures that can affect the productivity of their users.

In the meantime, science gateways, such as the Virtual Imaging Platform (VIP) (Ferreira da Silva et al., 2011; Glatard et al., 2013), are emerging as user-level platforms to facilitate the access to distributed computing and storage resources for scientific computations. Their high-level interface allows scientists to transparently run their analyses on large sets of computing resources. However, their large scale and the number of middleware systems involved in these gateways lead to many errors and faults. Applications running on these infrastructures are also growing in complexity and volume. Moreover, many scientists now formulate their computational problems as scientific workflows (Taylor, 2007). Workflows allow researchers to easily express multi-step computational tasks, for example: retrieve data from an instrument or a database, reformat the data, and run an analysis. Scientists expect such gateways to deliver high quality of service (QoS), where the workload and resources are efficiently and automatically handled, and the system is fault-tolerant. In order to provide fair QoS, scientific workflow executions are often backed by substantial human intervention that requires constantly monitoring of the running experiments and infrastructure to prevent or handle faults and ensure successful application completion.

Automating fault prevention, detection, and handling is challenging in such platforms. Science gateways have no *a-priori* model of the execution time of their applications because 1) task costs depend on input data with no explicit model, and 2) characteristics of the available resources depend on background load (Ferreira da Silva, Juve, et al., 2013). Modeling application execution time in these conditions requires cumbersome experiments, which cannot be conducted for every new application in the platform. As a consequence, such platforms operate in *non-clairvoyant* conditions, where little is known about executions before they actually happen. Such platforms also run in *online* conditions, i.e. users may launch or cancel applications at any time and resources may leave at any time too.

In this chapter, we propose a general self-management process for autonomous detection and handling of operational incidents in scientific workflow executions on grids (Ferreira da Silva, Glatard, & Desprez, 2012, 2013b). Our process is described as a MAPE-K loop (Kephart & Chess, 2003), which consists of monitoring (M), analysis (A), planning (P), execution (E), and knowledge (K). Self-management techniques, generally implemented as MAPE-K loops, provide an interesting framework to cope with online non-clairvoyant problems. They address non-clairvoyance by using a-priori knowledge about the platform (e.g. extracted from traces), detailed monitoring, and analysis of its current behavior. They can also cope with online problems by periodical monitoring updates. Our ultimate goal is to reach a general model of such a scientific gateway that could autonomously detect and handle operational incidents, and control the behavior of non-clairvoyant, online platforms to limit human intervention required for their operation. Performance optimization is a target but the main point is to ensure that correctly-defined executions are completed, that performance is acceptable, and that misbehaving runs (e.g. failures coming from user errors or unrecoverable infrastructure downtimes) are quickly detected and handled before they consume too many resources.

BACKGROUND

In this section, we introduce definitions necessary to understand the rest of this chapter, and present the relevant work regarding strategies to address the operational incidents proposed in this chapter: task replication, task grouping, and fairness among workflow executions.

Scientific Gateways. Some Software-as-a-Service platforms, commonly called scientific gateways, integrate application software with access to computing and storage resources via web portals or desktop applications, where users can process their own data with predefined applications. Science-gateways are used in different scientific domains such as multi-disciplinary, climate, and medical imaging.

Scientific Workflows. Scientific workflows allow users to easily express multi-step computational tasks, for example retrieve data from an instrument or a database, reformat the data, and run an analysis. A scientific workflow describes the dependencies between the tasks. In most cases the workflow is described as a directed acyclic graph (DAG), where the nodes are tasks (or group of tasks) and the edges denote the task (or group of tasks) dependencies. Sometimes control structures (e.g. loops, ifs) are also used within workflows. Scientific workflows are described as high-level abstraction languages that conceal the complexity of execution infrastructures to the user. Workflow language formalism is a formalism expressing the causal/temporal dependencies among a number of tasks to execute. Workflow interpretation and execution are handled by a workflow engine that manages the execution of the application on the distributed computing infrastructure. In addition, scientific workflows facilitate application management by assembling dependencies on deployment, and by enabling automatic interface generation in a scientific gateway.

Grid Computing. Computational grids emerged in the middle of the past decade as a paradigm for high-throughput computing for scientific research and engineering, through the federation of heterogeneous resources distributed geographically in different administrative domains (Foster, 2001). Resource sharing is governed by virtual organizations (VO), which are a set of individuals or institutions defined around a set of resource-sharing rules and conditions. Grid computing infrastructures are federations of cooperating resource infrastructure providers, working together to provide computing and storage services for research communities. These infrastructures can be characterized into research and production infrastructures. Research infrastructures are designed to support computer-science experiments related to parallel, large-scale or distributed computing, and networking. Production infrastructures, on the other hand, are designed to support large scientific experiments. They can be classified into HPC (High-Performance Computing) and HTC (High throughput computing). HPC systems focuses on tightly coupled parallel tasks, while HTC focuses on the efficient execution of a large number of loosely coupled tasks. The grid middleware enables users to submit tasks to store data and execute computation on grid infrastructures. Task scheduling, resources management, data storage, replication, and transfers are handled by the middleware. It also enables security functions, such as authentication and authorization.

Task replication. Task replication, a.k.a. redundant requests, is commonly used to address non-clairvoyant problems (Cirne, Brasileiro, Paranhos, Góes, & Voorsluys, 2007), but it should be used sparingly, to avoid overloading the middleware and degrading fairness among users (Casanova, Desprez, & Suter, 2010). For instance, (Litke, Skoutas, Tserpes, & Varvarigou, 2007) propose a task replication strategy to handle failures in mobile grid environments. Their approach is based on the Weibull distribution to estimate the number of replicas to guarantee a specific fault-tolerance level. In (Ramakrishnan et al., 2009), task replication is enforced as a fault-tolerant mechanism to increase the probability to complete a task successfully. Recently, (Ben-Yehuda, Schuster, Sharov, Silberstein, & Iosup, 2012) proposed a framework for dynamic selection of Pareto-efficient scheduling strategy, where tasks are replicated only in the tail phase when task completion rate is low. All the proposed approaches make strong assumptions on task and resource characteristics, such as the expected duration and resource performance. An important aspect to be evaluated when replicating task is the resource waste, a.k.a. the cost of task replication. (Cirne et al., 2007) evaluate the waste of resources by measuring the percentage of wasted cycles among all the cycles required to execute the application.

Task grouping. The low performance of *fine-grained* tasks is a common problem in widely distributed platforms where the scheduling overhead and queuing times are high, such as grid and cloud systems. Several works have addressed the control of task granularity of bag of tasks. For instance, (Muthuvelu et al., 2005) proposed an algorithm to group bag of tasks based on their granularity size--defined as the processing time of the task on the resource. Resources are ordered by their decreasing values of capacity (in MIPS) and tasks are grouped up to the resource capacity. This process continues until all tasks are grouped and assigned to resources. Then, (Ng Wai Keat, 2006) and (Ang, Ng, Ling, Por, & Liew, 2009) extended the previous work by introducing bandwidth in the scheduling framework to enhance the performance of task scheduling. Resources are sorted in decreasing order of bandwidth, then assigned to grouped tasks downward ordered by processing requirement length. Later, (Muthuvelu, Chai, & Eswaran, 2008) extended (Muthuvelu et al., 2005) to determine task granularity based on QoS requirements, task file size, estimated task CPU time, and resource constraints. Meanwhile, (Liu & Liao, 2009) proposed an adaptive fine-grained job scheduling algorithm (AFJS) to group lightweight tasks according to processing capacity (in MIPS) and bandwidth (in Mb/s) of the current available resources. To accommodate with resource dynamicity, the grouping algorithm integrates monitoring information about the current availability and capability of resources. (Zomaya & Chan, 2004) studied limitations and ideal control parameters of task clustering by using genetic algorithms. Their algorithm performs task selection based on the earliest task start time and task communication costs; it converges to an optimal solution of the number of clusters and tasks per cluster. Although the reviewed works significantly reduce communication and processing time, neither of them is non-clairvoyant and online at the same time. Recently, (Muthuvelu, Chai, Chikkannan, & Buyya, 2010) proposed an online scheduling algorithm to determine the task granularity of compute-intensive bag-of-tasks applications. The granularity optimization is based on task processing requirements, resource-network utilization constraint, and users QoS requirements (user's budget and application deadline). Submitted tasks are categorized according to their file sizes, estimated CPU times, and estimated output file sizes, and arranged in a tree structure. The scheduler selects a few tasks from these categories to perform resource benchmarking. In a collaborative work (Chen, Ferreira da Silva, Deelman, & Sakellariou, 2013), we presented three balancing methods to address the load balancing problem when clustering scientific workflow tasks. We defined three imbalance metrics to quantitative measure workflow characteristics based on task runtime variation (HRV), task impact factor (HIFV), and task distance variance (HDV). Although these are online approaches, the solutions are still clairvoyant.

Fairness. Fairness among scientific workflow executions has been addressed in several studies considering the scheduling of multiple scientific workflows. For instance, (Henan Zhao & Sakellariou, 2006) address fairness based on the slowdown of DAGs; they consider a clairvoyant problem where the execution time and the amount of data transfers are known. Similarly, (N'Takpe & Suter, 2009) propose a mapping procedure to increase fairness among parallel tasks on multi-cluster platforms; they address an offline and clairvoyant problem where tasks are scheduled according to the critical path length, maximal exploitable task parallelism, or amount of work to execute. (Casanova et al., 2010) evaluate several scheduling online algorithms of multiple parallel task graphs (PTGs) on a single, homogeneous cluster. Fairness is measured through the maximum stretch (a.k.a. slowdown) defined by the ratio between the PTG execution time on a dedicated cluster, and the PTG execution time in the presence of competition with other PTGs. (C.-C. Hsu, Huang, & Wang, 2011) and (Sommerfeld & Richter, 2011) propose an online HEFT-based algorithm to schedule multiple workflows; they address a clairvoyant problem where tasks are ranked based on the length of their critical path, and tasks are mapped to the resources with the earliest finish time. (Hirales-Carbajal et al., 2012) schedule multiple parallel workflows on a Grid in a non-clairvoyant but offline context, assuming dedicated resources. Their multi-stage scheduling strategies consist of task labeling and adaptive allocation, local queue prioritization and site scheduling algorithm. Fairness among workflow tasks is achieved by task labeling based on task run time estimation. Recently, (Arabnejad & Barbosa, 2012) proposed an algorithm addressing an online but clairvoyant problem where tasks are assigned to resources based on their rank values; task rank is determined from the smallest remaining time among all remaining tasks of the workflow, and from the percentage of remaining tasks. Finally, in their evaluation of non-preemptive task scheduling, (Sabin, Kochhar, & Sadayappan, 2004) assess fairness by assigning a fair start time to each task, defined by the start time of the task on a complete simulation of all tasks whose queue time is lower than that one. If a task has started its execution after its fair start time, it is considered unfairly treated. Results are trace-based simulations over a period of one month, but the study is performed in a clairvoyant context. (Skowron & Rzadca, 2013) proposed an online and non-clairvoyant algorithm to schedule sequential jobs on distributed systems. They consider a non-clairvoyant model where job's processing time is unknown until the job completes. However, they assume that resources are homogeneous (what is not the case on Grid computing). In contrast, our method considers resource performance, the execution of concurrent activities, and task dependency in scientific workflow executions.

THE VIRTUAL IMAGING PLATFORM

The Virtual Imaging Platform (VIP) (Ferreira da Silva et al., 2011; Glatard et al., 2013) is an openly-accessible platform for scientific workflow executions on a production grid. Figure 1 show the overall VIP architecture for workflow execution. It is composed of 1) a web portal which interfaces users to applications described as workflows, 2) a data management tool to handle transfer operations between users machines and the Grid storage, 3) a workflow engine to process user inputs and spawn computational tasks, 4) a workload management system for resource provisioning and task scheduling, and 5) an execution infrastructure. In VIP, users authenticate to a web portal with login and password, and they are then mapped to X.509 robot credentials. From the portal, users transfer data and launch applications workflows to be executed on the Grid. Workflows are compositions of activities defined independently from the processed data and that only consist of a program description and requirements. At runtime, activities receive data and spawn invocations from their input parameter sets. Invocations are independent from each other (bag of tasks) and executed on the computing resource as single-core tasks, which can be resubmitted in case of failures. VIP applications are executed on the biomed virtual organization (VO) of the European Grid Infrastructure (EGI). EGI is a federation of over 350 resources centers (sites) across more than 50 countries, which has access to more than 320,000 logical CPUs and 152 PB of disk space. The biomed VO has access to some 90 computing sites of 22 countries, offering 190 batch queues and approximately 4 PB of disk space.

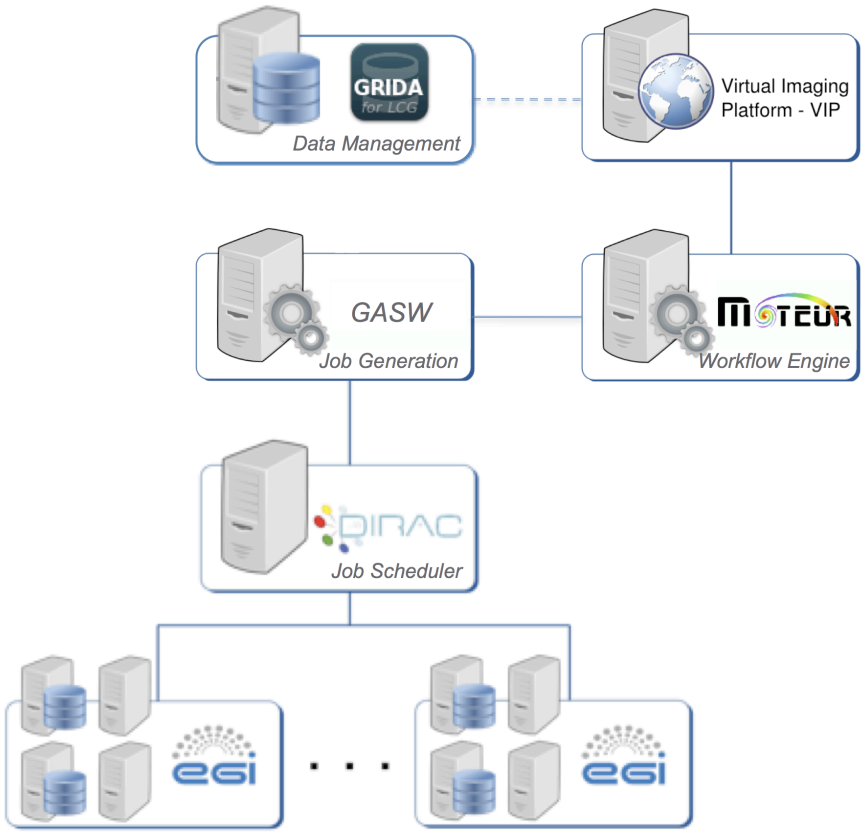


Figure 1: VIP architecture for workflow execution.

For a user, a typical application execution consists of the following steps: 1) select an application, 2) upload input data, 3) launch a workflow, and 4) download results. These steps are shown in steps 1, 2 and 11 from Figure 2. For the platform, it consists of performing a workflow execution. A workflow description and a set of input parameters is received and processed by the workflow engine, which produces invocations. In VIP, workflows are interpreted and executed using the MOTEUR workflow engine (Glatard, Montagnat, Lingrand, & Pennec, 2008), which provides an asynchronous grid-aware enactor. From invocations the workflow engine generates Grid tasks, and submits to the DIRAC (Tsaregorodtsev et al., 2010) workload management system, which implements a late binding between tasks and resources. DIRAC deploys pilot jobs on computing resources; pilot jobs run special agents that fetch user tasks from the task queue, set up their environment and steer their execution; task execution consists of downloading input data, executing the application, and uploading results. Figure 2 summarizes this process.

SELF-MANAGEMENT OF WORKFLOW EXECUTIONS ON GRIDS

The resource heterogeneity of production Grids, such as EGI, raises workflow execution issues, for instance, input and output data transfers may fail because of network glitches or limited site inter-communication; application executions may fail because of corrupted executable files, missing dependencies, or incompatibility; application executions may slowdown because of resources with poorer performance. Furthermore, the high communication overhead and queuing time intrinsic to such infrastructures may delay the workflow execution.

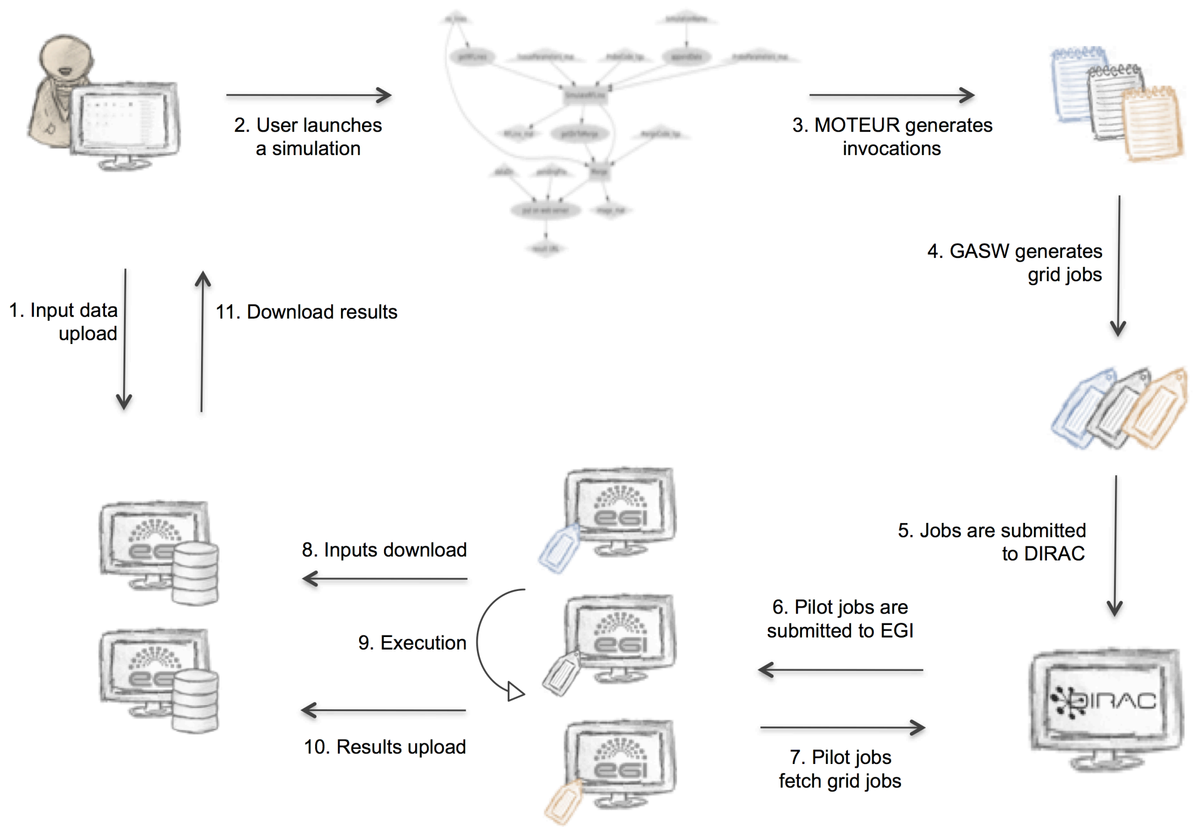
In this section, we propose a general self-management process (Ferreira da Silva et al., 2012; Ferreira da Silva, Glatard, et al., 2013b) to autonomously handle operational incidents on workflow executions. Instances involved in a workflow execution are modeled as Fuzzy Finite State Machines (FuSM) (Malik, Mordeson, & Sen, 1994) where state degrees of membership are determined by an external process. Degrees of membership are computed from metrics assuming that incidents have outlier performance, e.g. a site or a particular invocation behaves differently than the others. These metrics make little assumptions on the application or resource characteristics. Based on incident degrees, the process identifies incident levels using thresholds determined from the platform history. A specific set of actions is then selected from association rules among incident levels. The process is described formally in the next paragraphs.

Let $I = \{x_i,\ i = 1,..., n\}$ be the set of possible incidents and $\eta = (\eta_1,..., \eta_n) \in [0,1]^n$ their degrees in the FuSM. Incident x_i can occur at m_i different levels $\{x_{i,j}\ , j = 1,\ldots, m_i\}$ delimited by thresholds values $\tau_i = \{\tau_{i,j}, 1,..., m_i\}$. The level of incident i is determined by j such that $\tau_{i,j} \leq \eta_i < \tau_{i,j+1}$. A set of actions $a_i(j)$ is available to address $x_{i,j}$:

$$\begin{aligned} a_i : [1,m_i] &\rightarrow \wp(A) \\ j &\rightarrow a_i(j) \end{aligned} \qquad (1)$$

where A is the set of possible actions taken by the self-management process and $\wp(A)$ is the power set of A.

In addition to the incidents themselves, incident co-occurrences are taken into account. Association rules (Agrawal, Imieliński, & Swami, 1993) are used to identify relations between levels of different incidents. Association rules to $x_{i,j}$ are defined as $R_{i,j} = \{r^{u,v}_{i,j} = (x_{u,v}, x_{i,j}, \rho^{u,v}_{i,j})\}$. Rule $r^{u,v}_{i,j}$ means that when $x_{u,v}$ happens then $x_{i,j}$ also happens with confidence $\rho^{u,v}_{i,j} \in [0,1]$. The confidence of a rule is an estimate of probability $P(x_{i,j} \mid x_{u,v})$. For the sake of completeness, $r^{i,j}_{i,j} \in R_{i,j}$ and $\rho^{i,j}_{i,j} = 1$. We also define $R = \cup_{i\in[1,n], j\in[1,m]} R_{i,j}$. The inference made by an association rule does not necessarily imply causality. Instead, it quantifies co-occurrence between the rule's terms (Tan, 2006).



2. User launches a simulation
3. MOTEUR generates invocations
4. GASW generates grid jobs
1. Input data upload
11. Download results
5. Jobs are submitted to DIRAC
8. Inputs download
9. Execution
6. Pilot jobs are submitted to EGI
10. Results upload
7. Pilot jobs fetch grid jobs
egi
DIRAC

Figure 2: Workflow execution flow in VIP.

Algorithm 1 presents the algorithm used at each iteration of the self-management process. Incident degrees are determined based on metrics and incident levels j are obtained from historical data. A roulette wheel selection (De Jong, 1975) based on η is performed to select $x_{i,j}$ the incident level of interest for the iteration. In a roulette wheel selection, incident x_i is selected with a probability p_i proportional to its degree: $p(x_i) = \eta_i \int_{j=1}^{n} \eta_j$. A potential cause $x_{u,v}$ for incident $x_{i,j}$ is then selected from another roulette wheel selection on the association rules $r^{u,v}_{i,j}$, where x_u is at level v. Rule $r^{u,v}_{i,j}$ is weighted $\eta_u \times \rho^{u,v}_{i,j}$ in this second roulette selection. Only first-order causes are considered here but the approach could be extended to include more recursion levels. Note that $r^{i,j}_{i,j}$ participates in this selection so that a first-order cause is not systematically chosen. Finally, actions in $a_u(v)$ are performed.

Algorithm 1 One iteration of the self-managing process.

```
1:  **input:** history of $\eta$
2:  **output:** set of actions $a$
3:  wait for event or timeout
4:  determine incident degrees $\eta \in [0,1]$ based on metrics
5:  determine incident levels $j$ such that $\tau_{i,j} \leq \eta_i < \tau_{i,j+1}$
6:  select incident $x_i$ by roulette wheel selection based on $\eta$
7:  select rule $r^{u,v}_{i,j} = (x_{u,v}, x_{i,j}, \rho^{u,v}_{i,j}) \in R_{i,j}$ by roulette wheel selection based on $\eta_u \times \rho^{u,v}_{i,j}$, where $x_u$ is at level $v$
8:  $a=a_u(v)$
9:  perform actions in $a$
```

Incident degrees are quantified in discrete incident levels so that different sets of actions can be used to address different levels of the incident. Thresholding consists in clustering platform configurations into groups. We determine τ_i, the threshold value of an incident degree x_i, from execution traces, for which different thresholding approached can be used. For instance, we could consider that *x%* of the platform configurations are inappropriate while the rest are acceptable. The choice of *x*, however, would be arbitrary. Instead, we inspect the modes of the distribution of η_i to determine a threshold. Thresholds τ_i are determined from visual mode clustering. The number m_i of incident levels associated to incident *i* is set as the number of modes in the observed distribution of η_i. Incidents levels and thresholds are determined offline; thus they do not create any overhead on the workflow execution. The process is parameterized on real application traces acquired in production on the European Grid Infrastructure (EGI) (Ferreira da Silva & Glatard, 2013).

In the rest of this chapter, we show the instantiation of our self-management process to address two workflow activity-level incidents: the long tail effect issue, and the task granularity problem; and an incident at platform level: unfairness among workflow executions.

HANDLING BLOCKED ACTIVITIES

The long-tail effect is a common frustration for users who have to wait to retrieve the last pieces of their computation. This issue happens due to execution on slow machines, poor network connection, or communication issues, and leads to substantial speed-up reductions. In this section, we propose an algorithm to handle the long-tail effect and to control task replication (Ferreira da Silva et al., 2012; Ferreira da Silva, Glatard, et al., 2013b). Our method identifies blocked activities as the ones whose tasks are performing worse than the median of already completed tasks. Tasks are assumed of identical costs. This assumption considers that the variation of task durations of correct executions due to resource heterogeneity is negligible compared to the variation when an incident happens. Algorithm 2 describes our activity blocked control process.

Algorithm 2 Main loop for activity blocked control.

```
1: input: m workflow executions
2: while there is an active workflow do
3:    wait for timeout or task status change in any workflow
4:    determine blocked degree $\eta_b$
5:    if $\eta_b > \tau_b$ then
6:       replicate late tasks
7:    end if
8: end while
```

Incident Degree and Levels

Activity blocked degree $\boldsymbol{\eta_b}$. We define the incident degree η_b of an activity from the maximum of the performance coefficients p_i of its *n* tasks, which relate the task phase durations (`setup`, `inputs download`, `application execution`, and `outputs upload`) to their medians:

$$\eta_b = 2 \cdot \max\left\{ p_i = p(t_i, \tilde{t}) = \frac{t_i}{\tilde{t} + t_i}, i \in [1,n] \right\} - 1 \qquad (2)$$

where $\mathrm{t_i = t_{i\_setup} + t_{i\_input} + t_{i\_exec} + t_{i\_output}}$ is the estimated duration of task *i* and $\tilde{\mathrm{t}}_\mathrm{i} = \tilde{\mathrm{t}}_{i\_setup} + \tilde{\mathrm{t}}_{\mathrm{i\_input}} + \tilde{\mathrm{t}}_{\mathrm{i\_exec}} + \tilde{\mathrm{t}}_{\mathrm{i\_output}}$ is the sum of the median durations of tasks 1 to *n*. Note that $\max\{p_i, i \in [1,n]\} \in [0.5,1]$ so that $\eta_b \in [0,1]$. Moreover, $\lim_{t_i \to +\infty} p_i = 1$ and $\max\{p_i, i \in [1,n]\} = 0.5$ when all the tasks behave like the median. When less than 2 tasks are completed, medians remain undefined and the control process is inactive.

The estimated duration t_i of a task is computed phase by phase, as follows: 1) for completed task phases, the actual consumed resource time is used; 2) for ongoing task phases, the maximum value between the current consumed resource time and the median consumed time is taken; and 3) for unstarted task phases, the time slot is filled by the median value. Figure 3 illustrates the task estimation process where the actual durations are used for the two first completed phases (42s for `setup` and 300s for `inputs download`), the `application execution` phase uses the maximum value between the current value of 20s and the median value of 400s, and the last phase (`outputs upload`) is filled by the median value of 5s, as it is not started yet. Table 1 shows a summary of the symbols used in this section.

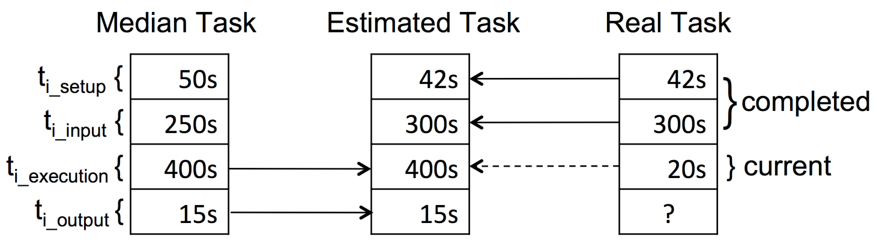


Figure 3: Task estimation based on median values.

Parameter	Description
η_b	Activity blocked incident degree
p	Performance coefficient of a task
$t,\ \tilde{t}$	Estimated duration of a task and sum of the median estimated durations of tasks
τ_b	Threshold value for η_b
R	Set of replicas
w	Waste coefficient

Table 1: Explanation of the symbols used in this section.

Threshold value τ_b. The threshold value for η_b separates configurations where the activity has acceptable performance ($\eta_b \le \tau_b$) from configurations where the activity is blocked ($\eta_b > \tau_b$). We determine τ_b from observed distributions of η_b. The blocked degree η_b was computed after each event found in the platform historical data (Ferreira da Silva & Glatard, 2013), and as shown in Figure 4. Since the modes are not clearly separable visually, we used K-Means to determine the threshold value $\tau_b = 0.35$. We assume that values in the lowest mode correspond to acceptable performance, and values in the highest mode correspond to low performance. Thus, for $\eta_b > 0.35$ task replication will be triggered.

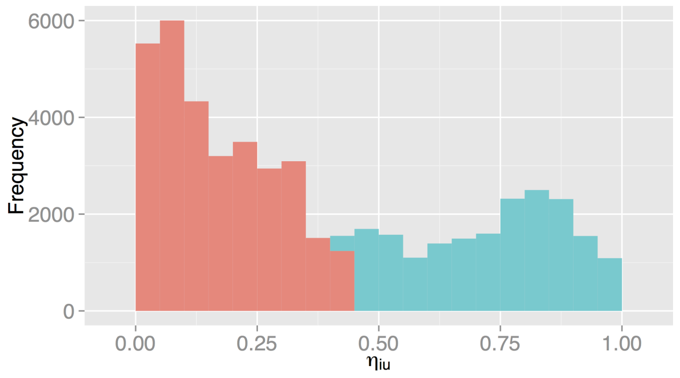


Figure 4: Histogram of activity-blocked degree sampled in bins of 0.05.

Task replication. Blocked activities are addressed by task replication. To limit resource waste, the replication process for a particular task is controlled by two mechanisms. First, a task is not replicated if a replica is already queued. Second, if replica *j* has better performance than replica *r* (i.e. $p(t_r, t_j) > \tau_b$, see Equation 2) and replica *j* is in a more advanced phase than replica *r*, then replica *r* is aborted. Algorithm 3 presents the algorithm of the replication process. It is applied to all tasks with $p_i > \tau_b$, as defined on Equation2.

Algorithm 3 Replication process for one task.

```
1:  input: set of replicas $R$ of a task $i$
2:  rep = true
3:  for $r \in R$ do
4:      for $j \in R, j \neq r$ do
5:          if $p(t_r, t_j) > \tau_b$ and $j$ is a step further than $r$ then
6:              abort $r$
7:          end if
8:      end for
9:      if ($r$ is started and $p(t_r, t_j) \leq \tau_b$) or $r$ is queued then
10:         rep = false
11:     end if
12: end for
13: if rep == true then
14:     replicate $r$
15: end if
```

Experiments and Results

The experiment presented hereafter evaluate the ability of the activity blocked control process to improve workflow makespan without wasting resources in case of tasks are late.

Experiment conditions. The self-management control process was implemented as a plug-in of the MOTEUR workflow engine, receiving notifications about task status changes and task phase durations. Task replication is performed by resubmitting running tasks to DIRAC. To avoid concurrency issues in the writing of output files, a simple mechanism based on file renaming is implemented. To limit infrastructure overload, running tasks are replicated up to 5 times. MOTEUR is configured to resubmit failed tasks up to 5 times in all runs.

This experiment uses a correct execution where the application is supposed to run properly and produce the expected results. Five repetitions are performed for each workflow activity. Two workflow activities

Workflow activity	#Tasks	CPU time	Input	Output
FIELD-II (data-intensive)	122	few seconds to 15 minutes	~208 MB	~40 KB
Mean-Shift (CPU-intensive)	250	few minutes to 1 hour	~182 MB	~1KB

Table 2: Workflow activity characteristics.

are considered for the experiment: `FIELD-II/pasa` and `Mean-Shift/hs3`. FIELD is a program to simulate ultrasound transducer fields and ultrasound imaging using linear acoustics. Mean-Shift is an image processing technique used to implement filtering, clustering, and segmentation in a *d*-dimensional space. Table 2 summarizes their main characteristics. A workflow execution using our method (`Self-Management`) is compared to a control execution (`No-Management`). Executions are launched on the biomed VO of the EGI, in production conditions. `Self-Management` and `No-Management` are both launched simultaneously to ensure similar grid conditions. The DIRAC scheduler is configured to equally distribute resources among executions.

Task replication may waste resources, i.e., resources are consumed by a set of tasks that compute the same operations. Here, resource waste is measured by the amount of resource time consumed by `Self-Management` executions related to the amount of resource time consumed by control executions. We use the waste coefficient (w), defined as follows:

$$w = \frac{\sum_{i=1}^{n} h_i + \sum_{j=1}^{m} rj}{\sum_{i=1}^{n} c_i} - 1 \qquad (3)$$

where h_i and c_i are the resource time consumed (CPU time + data transfers time) by n completed tasks for `Self-Management` and `No-Management` executions respectively, and r_i is the resource time consumed by m unused replicas. Note that task replication usually leads to $h_i \leq c_i$. If $w > 0$, `Self-Management` wastes resources compared to the control execution. Otherwise, `Self-Management` consumes fewer resources than `No-Management`, which can happen when faster resources are selected.

Results and discussion. Figures 5 and 6 and Tables 3 and 4 show the makespan and waste coefficient values of `FIELD-II/pasa` (left) and `Mean-Shift/hs3` (right) for the 5 repetitions, respectively. The makespan was considerably reduced in all repetitions of both activities. Speed-up values yielded by `Self-Management` ranged from 1.7 to 4.5 for `FIELD-II/pasa` and from 1.5 to 3.2 for `Mean-Shift/hs3`. The `Self-Management` process also reduces resource consumption up to 35% when compared to the control execution. This happens because replication increases the probability to select a faster resource. The total number of replicated tasks for all repetitions is 292 for `FIELD-II/pasa` (i.e. 0.48 task replication per task in average) and 712 for `Mean-Shift/hs3` (i.e. 0.57 task replication per task in average).

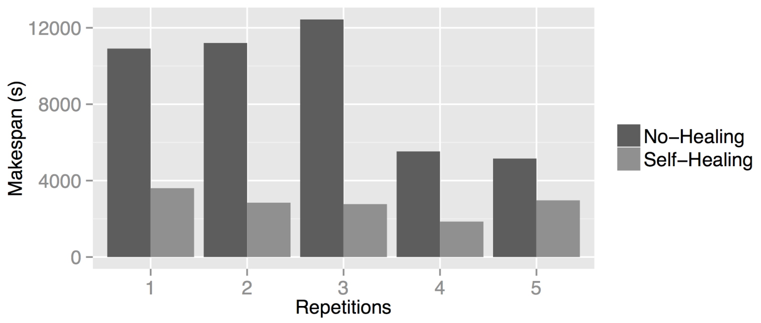


Figure 5: Execution makespan values for `FIELD-II/pasa`.

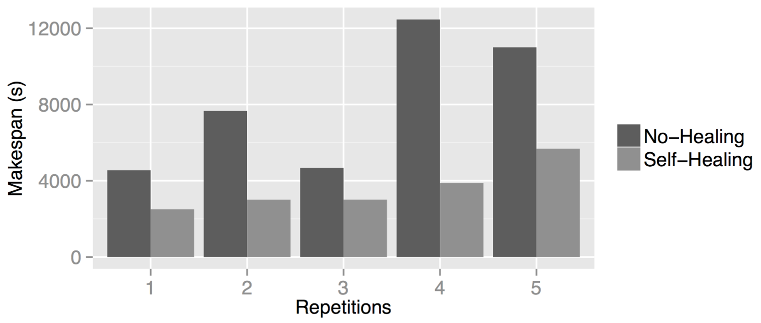


Figure 6: Execution makespan values for `Mean-Shift/hs3`.

Repetition	h	r	c	w
1	41,338s	23,823s	71,853s	-0.09
2	37,190s	28,251s	66,435s	-0.01
3	40,209s	25,068s	68,792s	-0.05
4	39,009s	32,973s	78,723s	-0.08
5	38,847s	37,393s	78,988s	-0.03

Table 3: Waste coefficient values (w) for `FIELD-II/pasa`.

Repetition	h	r	c	w
1	97,875s	17,709s	116,853s	-0.01
2	85,100s	19,086s	161,801s	-0.35
3	98,736s	25,162s	125,615s	-0.01
4	107,071s	62,746s	204,456s	-0.17
5	126,344s	2,195s	131,446s	-0.02

Table 4: Waste coefficient values (w) for `Mean-Shift/hs3`.

OPTIMIZING TASK GRANULARITY

Controlling the granularity of workflow activities executed on grids is required to reduce the impact of task queuing and data transfer time overheads. Most existing granularity control approaches assume extensive knowledge about the applications and resources (e.g. task duration on each resource), and that both the workload and available resources do not change over time (Ang et al., 2009; Muthuvelu et al., 2005, 2010; Ng Wai Keat, 2006). However such estimates are hard to obtain in production conditions (Ferreira da Silva, Juve, et al., 2013). Therefore, we propose a granularity control algorithm (Ferreira da Silva, Glatard, & Desprez, 2014, 2013a) for platforms where such clairvoyant and offline conditions are not realistic. Our method groups tasks when the fineness degree of the application, which takes into account the ratio of shared data and the queuing/round-trip time ratio, becomes higher than a threshold determined from execution traces. The algorithm also ungroups task groups when new resources arrive. Algorithm 4 describes our task granularity control composed of two processes: 1) fineness control groups too fine task groups for which the fineness degree η_f is greater than threshold τ_f, and 2) coarseness control ungroups too coarse task groups for which the coarseness degree η_c is greater than threshold τ_c. Table 5 shows a summary of the symbols used in this section.

Algorithm 4 Main loop for granularity control.

```
1:  input: m waiting tasks
2:  create n 1-task groups $T_i$
3:  while there is an active task group do
4:      wait for timeout or task status change
5:      determine fineness degree $\eta_f$
6:      if $\eta_f > \tau_f$ then
7:          group task groups using Algorithm 5
8:      end if
9:      determine coarseness degree $\eta_c$
10:     if $\eta_c > \tau_c$ then
11:         ungroup coarsest task groups
12:     end if
13: end while
```

Parameter	Description
η_f, η_c	Fineness and coarseness incident degrees
T	Set of tasks within a grouped task
d	Ratio between transfer time of input shared data and execution time
r	Ratio between task queuing times and task turnaround time
$\tilde{t}_{\_shared}$	Median transfer time of the input data shared among all tasks of an activity
τ_f, τ_c	Threshold values for η_f and η_c

Table 5: Explanation of the symbols used in this section.

Incident Degree and Levels

Fineness control

Fineness degree η_f. Let n be the number of waiting tasks in a workflow activity, and m the number of task groups. Tasks of an activity are assumed independent, but with similar costs (bag of tasks). Initially, 1 group is created for each task ($n = m$). T_i is the set of tasks in group i, and n_i is the number of tasks in T_i. Groups are a partition of the set of waiting tasks: $T_i \bigcap_{i \neq j} T_j = \varnothing$ and $\sum_{i=1}^{m} n_i = n$. The activity fineness degree η_f is the maximum of all group fineness degrees f_i:

$$n_f = \max_{i \in [1,m]}(f_i) \tag{4}$$

All η_f are in [0,1], and high fineness degrees indicate fine granularities. We use a *max* operator in this equation to ensure that *any* task group with a too fine granularity will be detected. The fineness degree f_i of group i is defined as:

$$f_i = d_i \cdot r_i \tag{5}$$

where d_i is the ratio between the transfer time of the input data shared among all tasks in the activity, and the total execution time of the group:

$$d_i = \frac{\tilde{t}_{\_shared}}{\tilde{t}_{\_shared} + n_i(\tilde{t} - \tilde{t}_{\_shared})} \tag{6}$$

where $\tilde{t}_{\_shared}$ is the median transfer time of the input data shared among all tasks in the activity, and $\tilde{t}$ is the sum of its median task phase durations corresponding to application setup, input data transfer, application execution and output data transfer: $\tilde{t} = \tilde{t}_{\_setup} + \tilde{t}_{\_\mathrm{input}} + \tilde{t}_{\_\mathrm{exec}} + \tilde{t}_{\_\mathrm{output}}$. Median values $\tilde{t}_{\_shared}$ and $\tilde{t}$ are computed from values measured on completed tasks. When less than 2 tasks are completed, medians remain undefined and the control process is inactive. This online estimation makes our process non-clairvoyant with respect to the task duration, which is progressively estimated as the workflow activity runs. Yet, it assumes that all tasks in an activity have similar costs.

In Equation 5, r_i is the ratio between the maximum of the task queuing times q_i in the group, and the total round-trip time (queuing + execution) of the group:

$$r_i = \frac{\max_{j \in [1,n_i]} q_j}{\max_{j \in [1,n_i]} q_j + \tilde{t}_{\_shared} + n_i(\tilde{t} - \tilde{t}_{\_shared})} \tag{7}$$

Group queuing time is the max of all task queuing times in the group; group execution time is the time to transfer shared input data and the time to execute all task phases in the group except for the transfers of shared input data. Note that d_i, r_i, and therefore f_i and η_f are in [0,1]. η_f tends to 0 when there is little shared input data among the activity tasks or when the task queuing times are low compared to the execution times; in both cases, grouping tasks is indeed useless. Conversely, η_f tends to 1 when the transfer time of shared input data becomes high, and the queuing time is high compared to the execution time; grouping is needed in this case.

Threshold value τ_f. The threshold value for η_f separates configurations where the activity's fineness is acceptable ($\eta_f \leq \tau_f$) from configurations where the activity is too fine ($\eta_f > \tau_f$). We determine τ_f from execution traces (Ferreira da Silva & Glatard, 2013), inspecting the distribution modes of η_f. Values of η_f in the highest mode of the distribution, i.e. which are clearly separated from the others, will be considered too fine. Figure 7 shows the histogram of these values. The histogram appears bimodal, which indicates that η_f separates platform configurations in two distinct groups. We assume that these groups correspond to *acceptable fineness* (lowest mode) and *too fine granularity* (highest mode), and thus we choose $\tau_f = 0.55$. For $\eta_f \geq 0.55$, task grouping will therefore be triggered.

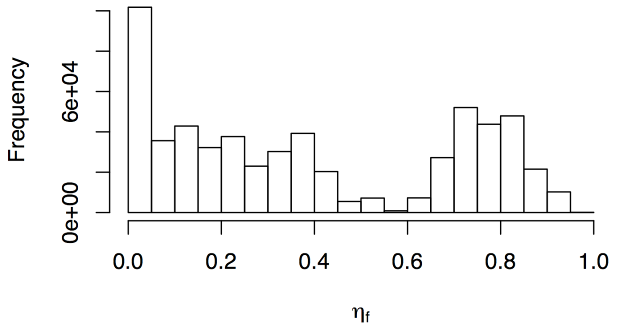


Figure 7: Histogram of fineness incident degree sampled in bins of 0.05.

Task grouping. We assume that running tasks cannot be pre-empted, i.e. only waiting tasks can be grouped. Algorithm 5 describes our task grouping algorithm. Groups where $f_i > \tau_f$ are grouped pairwise until $\eta_f \leq \tau_f$ or until the amount of waiting groups Q is smaller or equal to the amount of running groups R. Although η_f ignores scattering (Equation 4 uses a *max* operator), the algorithm considers it by grouping tasks in all groups where $f_i > \tau_f$. Ordering groups by decreasing f_i values tends to equally distribute tasks among groups. The grouping process stops when $Q \leq R$ to avoid parallelism loss. This condition also avoids conflicts with the ungrouping process described in the next sub-section.

Algorithm 5 Task grouping.

```
1:  input: $f_1$ to $f_m$     // group fineness degrees, sorted in decreasing order
2:  input: $Q$, $R$     // number of queued and running task groups
3:  for $i = 1$ to $m - 1$ do
4:    $j = i + 1$
5:    while $f_i > \tau_f$ and $Q > R$ and $j \leq m$ do
6:      if $f_i > \tau_f$ then
7:        group all tasks of $T_j$ into $T_i$
8:        recalculate $f_i$ using Equation 5
9:        $Q = Q - 1$
10:     end if
11:     $j = j + 1$
12:   end while
13:   $i = j$
```

```
14: end for
15: delete all empty task groups
```

Coarseness control

Condition $Q > R$ used in Algorithm 5 ensures that all resources will be exploited *if the number of available resources is stationary* (i.e., constant). In case the number of available resources decreases, the fineness control process may further reduce the number of groups. However, if the number of available resources increases, task groups may need to be ungrouped to maximize resource exploitation. This ungrouping is implemented by our coarseness control process. The process monitors the value of η_c defined as:

$$n_c = \frac{R}{Q+R} \tag{8}$$

The threshold value τ_c is set to 0.5 so that $\eta_c > \tau_c \Leftrightarrow Q < R$.

When an activity is considered too coarse, its groups are ordered by increasing values of η_f and the first groups (i.e. the coarsest ones) are split until $\eta_c < \tau_c$. Note that ungrouping increases the number of queued tasks, therefore tends to reduce η_c.

Experiments and Results

The experiments presented hereafter evaluate, in a production environment, the fineness control process under stationary load, and the interest of controlling coarseness under non-stationary load.

Experiment Conditions. The granularity control process was implemented as a plugin of the MOTEUR workflow manager, receiving notifications about task status changes and task phase durations. The plugin then uses this data to group and ungroup tasks according to Algorithm 4, where the timeout value is set to 2 minutes. To ensure resource limitation without overloading the production system with test tasks, experiment executions are limited to 3 sites of different countries. As no online task modification is possible in the DIRAC workload management system, we implemented task grouping by canceling queued tasks and submitting grouped tasks as a new task.

Three workflow activities (summarized in Table 6), implementing different types of medical image simulation, are used in the experiments: `SimuBloch`, `FIELD-II`, and `PET-Sorteo/emission`. SimuBloch is a simulator made for fast simulation of MRIs based on Bloch equation. Two sets of experiments are conducted under different load patterns. The first experiment evaluates the fineness control process only under stationary load. It consists of separated executions of `SimuBloch`, `FIELD-II`, and `PET-Sorteo/emission`. A workflow activity using our task grouping mechanism (`Fineness`) is compared to a control activity (`No-Granularity`). Resource contention on the 3 execution sites is maintained high and constant so that no ungrouping is required. The second experiment evaluates the interest of using the ungrouping control process under non-stationary load. It uses activity `FIELD-II`. An execution using both fineness and coarseness control (`Fineness-Coarseness`) is compared to an execution without coarseness control (`Fineness`) and to a control execution (`No-Granularity`). Executions are started under resource contention, but the contention is progressively reduced during the experiment. This is done by submitting a heavy workflow before the experiment starts, and canceling it when half of the control tasks are completed.

Workflow activity	#Tasks	CPU time	Input	Output	$\tilde{t}_{\_shared} / \tilde{t}$
SimuBloch (data-intensive)	25	few seconds	~15 MB	< 5 MB	~0.9
FIELD-II (data-intensive)	122	few seconds to 15 minutes	~208 MB	~40 KB	[0.4,0.6]
Mean-Shift (CPU-intensive)	250	few minutes to 1 hour	~182 MB	~1KB	[0.5,0.8]

Table 6: Workflow activity characteristics.

Results and Discussion.

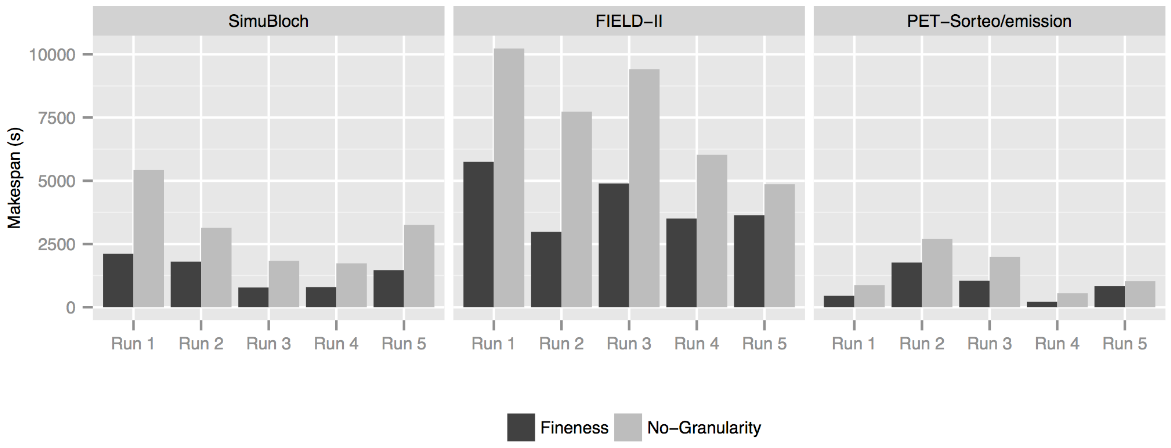


Figure 8 shows the makespan of `SimuBloch`, `FIELD-II`, and `PET-Sorteo/emission` executions. `Fineness` yields a significant makespan reduction for all repetitions. Table 7 shows the makespan (*M*) values and the final number of task groups. The task grouping mechanism is not able to group all `SimuBloch` tasks in a single group because 2 tasks must be completed for the process to have enough information about the application (i.e. $\tilde{t}_{\_shared}$ and $\tilde{t}$ can be computed). This is a constraint of our non-clairvoyant conditions, where task durations cannot be determined in advance. `FIELD-II` tasks are initially not grouped, but as the queuing time becomes important, tasks are considered too fine, thus they are grouped. `PET-Sorteo/emission` is an intermediary case where only a few tasks are grouped. Results show that the task grouping mechanism speeds up `SimuBloch` and `FIELD-II` executions up to a factor of 2.6, and `PET-Sorteo/emission` executions up to a factor of 2.5.

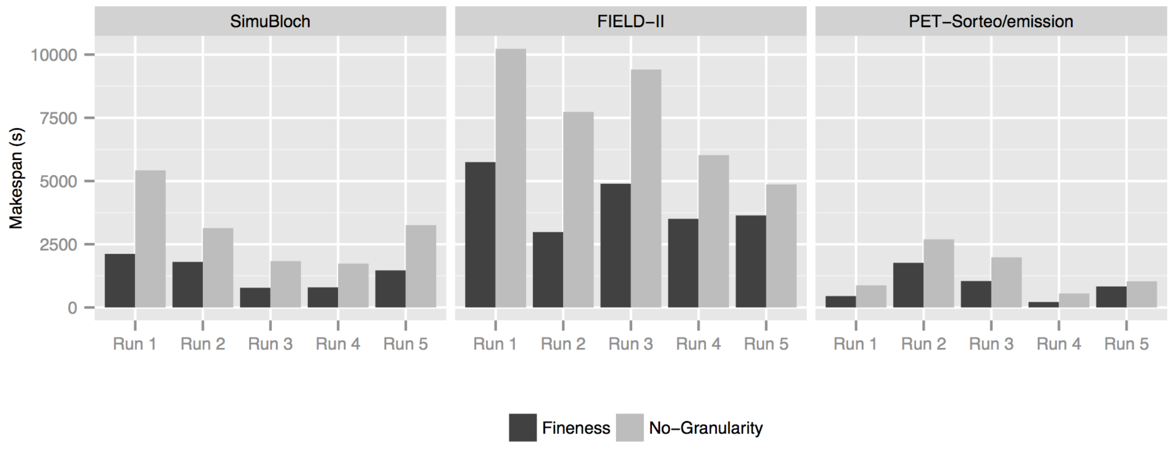


Figure 8: Makespan for `Fineness` and `No-Granularity` executions under stationary load.

		SimuBloch		FIELD-II		PET-Sorteo	
		M (s)	Groups	*M* (s)	Groups	*M* (s)	Groups
1	No-Granularity	5421	25	10230	122	873	80
	Fineness	2118	3	5749	80	451	57
2	No-Granularity	3138	25	7734	122	2695	80
	Fineness	1803	3	2982	75	1766	40
3	No-Granularity	1831	25	9407	122	1983	80
	Fineness	780	4	4894	73	1047	53
4	No-Granularity	1737	25	6026	122	552	80
	Fineness	797	6	3507	61	218	64
5	No-Granularity	3257	25	4865	122	1033	80
	Fineness	1468	4	3641	91	831	71

Table 7: Makespan (*M*) and number of task groups for `SimuBloch`, `FIELD-II`, and `PET-Sorteo/emission` executions for the 5 repetitions.

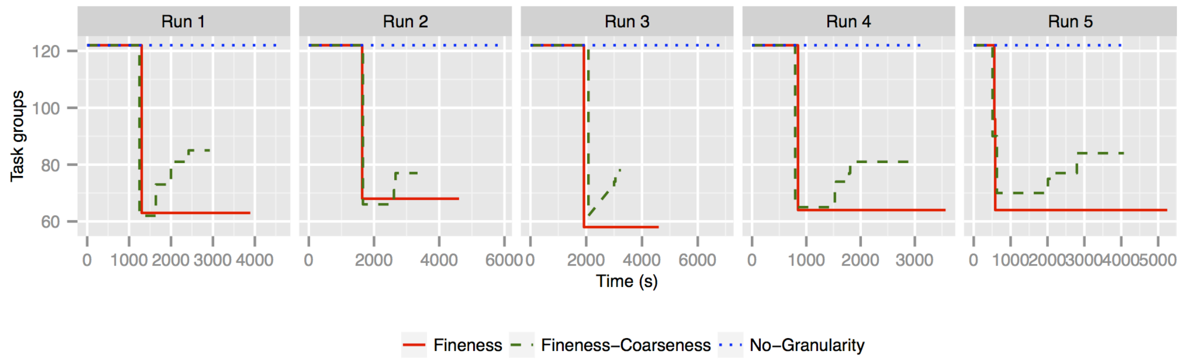


Figure 9 shows the evolution of task groups for `FIELD-II` executions under non-stationary load (resources arrive during the experiment). Makespan values are reported in Table 8. In the first three repetitions, resources emerge progressively during workflow executions. `Fineness` and `Fineness-Coarseness` speed up executions up to a factor of 1.5 and 2.1. Since `Fineness` does not benefit from newly arrived resources, it has a lower speed up compared to `No-Granularity` due to parallelism loss. In the two last repetitions (where resources appear suddenly), the ungrouping process in `Fineness-`

Coarseness has similar performance than No-Granularity since the execution maximizes the parallelism, while Fineness is penalized by its lack of adaptation: a slowdown of 20% is observed compared to No-Granularity.

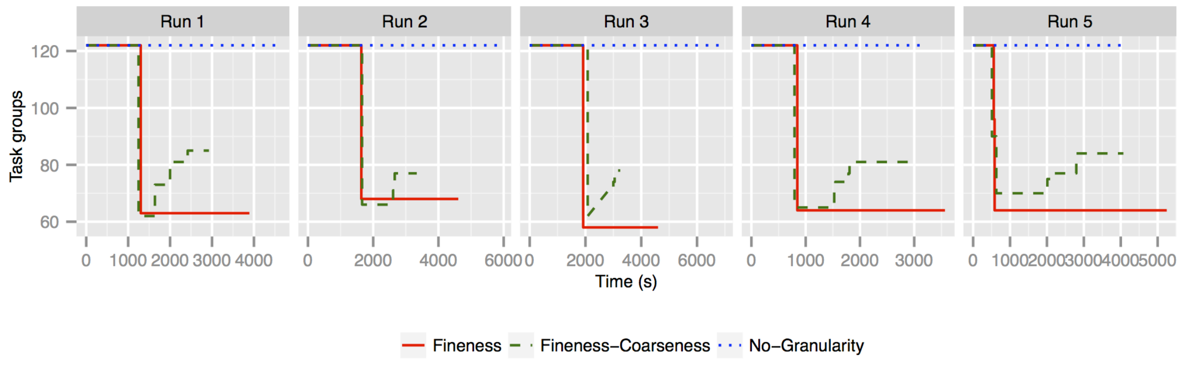


Figure 9: Evolution of task groups for FIELD-II executions under non-stationary load (resources arrive during the experiment).

	Run 1		Run 2		Run 3		Run 4		Run 5	
Workflow activity	M (s)	$\bar{e}$ (s)	M (s)	$\bar{e}$ (s)	M (s)	$\bar{e}$ (s)	M (s)	$\bar{e}$ (s)	M (s)	$\bar{e}$ (s)
No-Granularity	4617	2011	5934	2765	6940	3855	3199	1863	4147	2295
Fineness	3892	2036	4607	2090	4602	2631	3567	1928	5247	2326
Fineness-Coarseness	2927	1708	3335	1829	3247	2091	2952	1586	4073	2197

Table 8: Makespan (M) and average queuing time ($\bar{e}$) for FIELD-II workflow execution for the 5 repetitions.

Our task granularity control process works best under high resource contention, when the amount of available resources is stable or decreases over time. Coarseness control can cope with soft increases in the number of available resources, but fast variations remain difficult to handle. In the worst-case scenario, tasks are first grouped due to resource limitation, and resources suddenly appear once all task groups are already running. In this case the ungrouping algorithm has no group to handle, and granularity control penalizes the execution. Task pre-emption should be added to the method to address this scenario.

CONTROLLING FAIRNESS AMONG WORKFLOW EXECUTIONS

Fairly allocating distributed computing resources among workflow executions is critical to multi-user platforms such as VIP. However, this problem remains mostly studied in clairvoyant and offline conditions, where task durations on resources are known, or the workload and available resources do not vary along time. We consider a non-clairvoyant, online fairness problem where the platform workload, task costs and resource characteristics are unknown and not stationary. We propose a fairness control loop which assigns task priorities based on the fraction of pending work in the workflows (Ferreira da Silva et al., 2014; Ferreira da Silva, Glatard, & Desprez, 2013c). Workflow characteristics and performance on the target resources are estimated progressively, as information becomes available during the execution. Workflows consist of linked activities spawning tasks for which the executable and input data are known, but the computational cost and produced data volume are not. Algorithm 6 summarizes our fairness control process. Fairness is controlled by allocating resources to workflows according to their fraction of pending work. It is done by re-prioritizing tasks in workflows where the unfairness degree η_u is greater than a threshold τ_u. Table 9 shows a summary of the symbols used in this section.

Algorithm 6 Main loop for fairness control.

```
1:  input: $m$ workflow executions
2:  while there is an active workflow do
3:     wait for timeout or task status change in any workflow
4:     determine unfairness degree $\eta_u$
5:     if $\eta_u > \tau_u$ then
6:        re-prioritize tasks using Algorithm 7
7:     end if
8:  end while
```

Parameter	Description
η_u	Unfairness incident degree
W, w	Fraction of pending work
Q, R	Number of queued and running tasks
P	Performance of the activity
T	Relative observed duration
$\tilde{t}$	Sum of the median task execution times
τ_u	Threshold value for η_u
Δ	Number of waiting tasks of an activity
μ	Area under the curve η_u during the execution

Table 9: Explanation of the symbols used in this section.

Incident Degree and Levels

Unfairness degree η_u. Let *m* be the number of workflows with an active activity; a workflow activity is active if it has at least one waiting (queued) or running task. The unfairness degree η_u is the maximum difference between the fractions of pending work:

$$\eta_u = W_{\max} - W_{\min} \tag{9}$$

with $W_{min} = min\{W_i,\ i \in [1,m]\}$ and $W_{max} = max\{W_i,\ i \in [1,m]\}$. All W_i are in [0,1]. For $\eta_u = 0$, we consider that resources are fairly distributed among all workflows; otherwise, some workflows consume more resources than they should. The fraction of pending work W_i of a workflow $i \in [1,m]$ is defined from the fraction of pending work $w_{i,j}$ of its n_i active activities:

$$W_i = \max_{j \in [1,n_i]}(w_{i,j}) \tag{10}$$

All $w_{i,j}$ are between 0 and 1. A high $w_{i,j}$ value indicates that the activity has a lot of pending work compared to the others. We define $w_{i,j}$ as:

$$w_{i,j} = \frac{Q_{i,j}}{Q_{i,j} + R_{i,j}P_{i,j}} \cdot T_{i,j} \tag{11}$$

where $Q_{i,j}$ is the number of waiting tasks in the activity, $R_{i,j}$ is the number of running tasks in the activity, $P_{i,j}$ is the performance of the activity, and $T_{i,j}$ is its relative observed duration. $T_{i,j}$ is defined as the ratio between the median duration $\tilde{t}_{i,j}$ of the completed tasks in activity *j* and the maximum median task duration among all active activities of all running workflows:

$$T_{i,j} = \frac{\tilde{t}_{i,j}}{\max_{v\in[1,m],w\in[1,n_i^*]}(\tilde{t}_{u,w})} \quad (12)$$

Tasks of an activity all consist of the following successive phases: `setup`, `inputs download`, `application execution`, and `outputs upload`; $\tilde{t}_{i,j}$ is computed as $\tilde{t}_{i,j} = \tilde{t}_{i,j}^{setup} + \tilde{t}_{i,j}^{input} + \tilde{t}_{i,j}^{exec} + \tilde{t}_{i,j}^{output}$. Medians are progressively estimated as tasks complete. At the beginning of the execution, $T_{i,j}$ is initialized to 1 and all medians are undefined; when two tasks of activity j complete, $\tilde{t}_{i,j}$ is updated and $T_{i,j}$ is computed with Equation 12. In this equation, the *max* operator is computed only on $n_i^* \le n_i$ activities with at least 2 completed tasks, i.e. for which $\tilde{t}_{i,j}$ can be determined. We are aware that using the median may be inaccurate. However, without a model of the applications' execution time, we have to rely on observed task durations. Using the whole time distribution (or at least its few first moments) may be more accurate but it would make the method more complex.

In Equation 11, the performance $P_{i,j}$ of an activity varies between 0 and 1. A low $P_{i,j}$ indicates that resources allocated to the activity have bad performance for the activity; in this case, the contribution of running tasks is reduced and $w_{i,j}$ increases. Conversely, a high $P_{i,j}$ increases the contribution of running tasks, therefore decreases $w_{i,j}$. For an activity j with k_j active tasks, we define $P_{i,j}$ as:

$$P_{i,j} = 2 \cdot \left(1 - \max_{u\in[1,k_j]}\left\{\frac{t_u}{\tilde{t}_{i,j} + t_u}\right\}\right) \quad (13)$$

where $t_u = t_u^{setup} + t_u^{input} + t_u^{exec} + t_u^{output}$ is the sum of the estimated durations of task u's phases. Estimated task phase durations are computed as the max between the current elapsed time in the task phase (0 if the task phase has not started) and the median duration of the task phase. $P_{i,j}$ is initialized to 1, and updated using Equation 13 only when at least 2 tasks of activity j are completed. Note that computing $P_{i,j}$ is equivalent to computing the complement of the activity blocked degree $1 - \eta_b$, for activity j of workflow i.

If all tasks perform as the median, i.e. $t_u = \tilde{t}_{i,j}$, then $\max_{u\in[1,k_j]}\left\{t_u / (\tilde{t}_{i,j} + t_u)\right\} = 0.5$ and $P_{i,j} = 1$. Conversely, if a task in the activity is much longer than the median, i.e. $t_u >> \tilde{t}_{i,j}$, then $\max_{u\in[1,k_j]}\left\{t_u / (\tilde{t}_{i,j} + t_u)\right\} \approx 1$ and $P_{i,j} \approx 0$.

This definition of $P_{i,j}$, considers that bad performance results in a few tasks blocking the activity. Indeed, we assume that the scheduler does not deliberately favor any activity and that performance discrepancies are manifested by a few *unlucky* tasks slowed down by bad resources. Performance, in this case, has a relative definition: depending on the activity profile, it can correspond to CPU, RAM, network bandwidth, latency, or a combination of those. We admit that this definition of $P_{i,j}$ is a bit rough. However, under our non-clairvoyance assumption, estimating resource performance for the activity more accurately is hardly possible because 1) we have no model of the application, therefore task durations cannot be predicted from CPU, RAM or network characteristics, and 2) network characteristics and even available RAM are shared among concurrent tasks running on the infrastructure, which makes them hardly measurable.

Thresholding unfairness τ_u. Task prioritization is triggered when the unfairness degree is considered critical, i.e $\eta_u > \tau_u$. Inspecting the modes of the distribution of η_u we determine that values of η_u in the highest mode of the distribution, i.e. which are clearly separated from the others, will be considered unfair. Figure 10 shows the histogram of these values, where only $\eta_u \neq 0$ values are represented. This histogram is clearly bi-modal, which is a good property since it reduces the influence of τ_u. From this histogram, we choose $\tau_u = 0.2$. For $\eta_u > 0.2$, task prioritization is triggered.

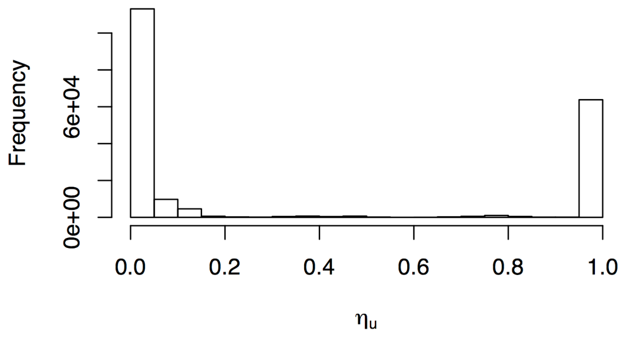


Figure 10: Histogram of the unfairness degree η_u sampled in bins of 0.05.

Task prioritization. Task priority is an integer initialized to 1. The action taken to cope with unfairness is to increase the priority of $\Delta_{i,j}$ waiting tasks for all activities j of workflow i where $w_{i,j}$ - $W_{\min} > \tau_u$. Running tasks cannot be pre-empted. $\Delta_{i,j}$ is determined so that $\tilde{w}_{i,j} = W_{\min} + \tau_u$, where $\tilde{w}_{i,j}$ is the estimated value of $w_{i,j}$ after $\Delta_{i,j}$ tasks are prioritized. We approximate $\tilde{w}_{i,j}$ as:

$$\tilde{w}_{i,j} = \frac{Q_{i,j} - \Delta_{i,j}}{Q_{i,j} + R_{i,j}P_{i,j}} \cdot \hat{T}_{i,j} \tag{14}$$

which assumes that $\Delta_{i,j}$ tasks will move from status queued to running, and that the performance of new resources will be maximal. It gives:

$$\Delta_{i,j} = Q_{i,j} - \left\lfloor \frac{(\tau_u + W_{\min})(Q_{i,j} + R_{i,j}P_{i,j})}{\hat{T}_{i,j}} \right\rfloor \tag{15}$$

where $\lfloor \ \rfloor$ rounds a decimal down to the nearest integer value.

Algorithm 7 describes our task re-prioritization algorithm. *maxPriority* is the maximal priority value in all workflows. The priority of $\Delta_{i,j}$ waiting tasks is set to *maxPriority* + 1 in all activities j of workflows i where $w_{i,j}$ - $W_{\min} > \tau_u$. Note that this algorithm takes into account scatter among W_i although η_u ignores it (see Equation 9). Indeed, tasks are re-prioritized in *any* workflow i for which $w_{i,j}$ - $W_{\min} > \tau_u$.

Algorithm 7 Task re-prioritization.

```
1:  input: $W_1$ to $W_m$    // fractions of pending works
2:  maxPriority = max task priority in all workflows
3:  for $i$ = 1 to $m$ do
4:    if $W_i - W_{\min} > \tau_u$ then
5:      for $j$ = 1 to $a_i$ do
6:        // $a_i$ is the number of active activities in workflow $i$
7:        if $w_{i,j} - W_{\min} > \tau_u$ then
8:          compute $\Delta_{i,j}$ from Equation 15
9:          for $p$ = 1 to $\Delta_{i,j}$ do
```

```
10:                 if $\exists$ waiting task $q$ in activity $j$ with priority $\leq$ maxPriority then
11:                     $q$.priority = maxPriority + 1
12:                 end if
13:             end for
14:         end if
15:     end for
16:   end if
17: end for
```

The method also accommodates online conditions. If a new workflow i is submitted, then $R_{i,j} = 0$ for all its activities and $\hat{T}_{i,j}$ is initialized to 1. This leads to $W_{\max} = W_i = 1$, which increases η_u. If η_u goes beyond τ_u, then $\Delta_{i,j}$ tasks of activity j of workflow i have their priorities increased to restore fairness. Similarly, if new resources arrive, then $R_{i,j}$ increase and η_u is updated accordingly.

Experiments and Results

The experiments presented hereafter evaluate our method on a set of identical workflows, where the variability of the measured makespan can be used as a fairness metric. In addition, we add a very short workflow to this set of identical workflow, which was one of the configurations motivating this study.

Experiment conditions. Fairness control was implemented as a MOTEUR plug-in receiving notifications about task and workflow status changes. Each workflow plug-in forwards task status changes and $\tilde{t}_{i,j}$ values to a service centralizing information about all the active workflows. This service then re-prioritizes tasks according to Algorithms 6 and 7. The timeout value used in Algorithm 6 is set to 3 minutes. As no online task modification is possible in DIRAC, we implemented task prioritization by canceling and resubmitting queued tasks to DIRAC with new priorities. Two real medical simulation workflows are considered: `GATE` and `SimuBloch`. GATE is a Geant4-based open-source software to perform nuclear medicine simulations, especially for TEP and SPECT imaging, as well for radiation therapy. Table 10 summarizes their main characteristics.

Workflow activity	#Tasks	CPU time	Input	Output
GATE (CPU-intensive)	100	few minutes to 1 hour	~115 MB	~40 MB
SimuBloch (data-intensive)	25	few seconds	~15 MB	< 5 MB

Table 10: Workflow characteristics.

Three different fairness metrics are used in the experiments. First, the standard deviation of the makespan, written σ_m, is a straightforward metric that can be used when identical workflows are executed. Second, we define the unfairness μ_u as the area under the curve η_u (see Equation 9) during the execution:

$$\mu = \sum_{i=2}^{M} \eta_u(t_i) \cdot (t_i - t_{i-1}) \tag{16}$$

where M is the number of time samples until the makespan. This metric measures if the fairness process can indeed minimize its own criterion η_u. In addition, the slowdown s of a completed workflow execution is measured as:

$$s = \frac{M_{multi}}{M_{own}} \tag{17}$$

where M_{multi} is the makespan observed on the shared platform, and M_{own} is the estimated makespan if it was executed alone on the platform. In our conditions, M_{own} is estimated as:

$$M_{own} = \max_{p \in \Omega} \sum_{u \in p} t_u \tag{18}$$

where Ω is the set of task paths in the workflow, and t_u is the measured duration of task u. This assumes that concurrent executions only impact task waiting time. For instance, network congestion or changes in performance distribution resulting from concurrent executions are ignored. We use σ_s, the standard deviation of the slowdown to quantify unfairness. The standard deviation of the makespan (σ_m) is also used.

Results and discussion. Figure 11 shows the makespan for the set of identical workflows. The unfairness degree η_u is shown in Figure 12, while the makespan standard deviation σ_m, slowdown standard deviation σ_s and unfairness μ_u for the 4 repetitions using the set of identical workflows is shown in **Table 11**. The difference among makespans and unfairness degree values are significantly reduced in all repetitions of `Fairness`. Both `Fairness` and `No-Fairness` behave similarly until η_u reaches the threshold value $\tau_u = 0.2$. Unfairness is then detected and the mechanism triggers task prioritization. Paradoxically, the first effect of task prioritization is a slight increase of η_u. Indeed, $P_{i,j}$ and $\hat{T}_{i,j}$, that are initialized to 1, start changing earlier in `Fairness` than in `No-Fairness` due to the availability of task duration values to compute $\tilde{t}_{i,j}$. Note that η_u reaches similar maximal values in both cases, but reaches them faster in `Fairness`. The fairness mechanism then manages to decrease η_u back under 0.2 much faster than it happens in `No-Fairness` when tasks progressively complete. Quantitatively, the fairness mechanism reduces σ_m up to a factor of 15, σ_s up to a factor of 7, and μ_u by about 2.

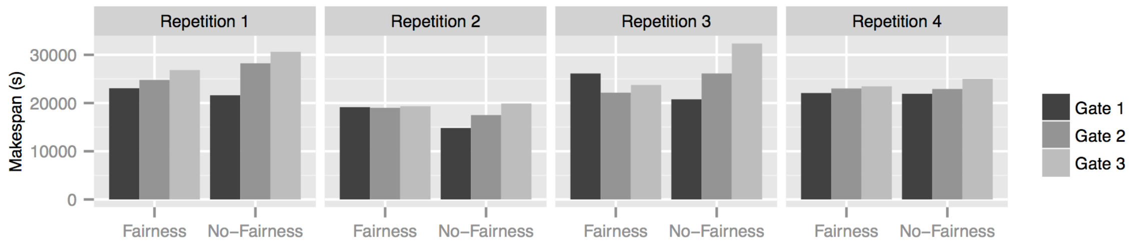


Figure 11: Comparison of the makespan for the 3 identical workflows.

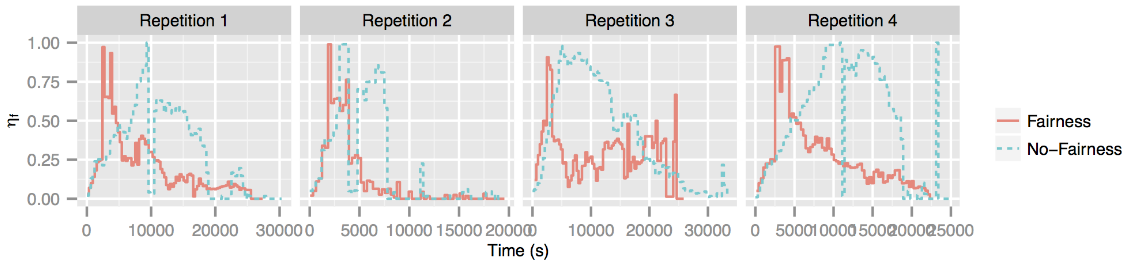


Figure 12: Unfairness degree η_u for the set of identical workflows.

	Repetition 1			Repetition 2			Repetition 3			Repetition 4		
	$\sigma_m(s)$	σ_s	$\mu(s)$	$\sigma_m(s)$	σ_s	$\mu(s)$	$\sigma_m(s)$	σ_s	$\mu(s)$	$\sigma_m(s)$	σ_s	$\mu(s)$
NF	4666	1.03	8758	2541	0.50	4154	5791	2.10	13392	1567	0.87	12283
F	1884	0.40	5292	167	0.84	2367	2007	0.84	7243	706	0.24	6070

Table 11. Makespan standard deviation σ_m, slowdown standard deviation σ_s and unfairness μ_u.

Figure 13 shows the makespan for the case where a very short workflow is introduced. Unfairness degree η_u is shown in Figure 14. **Table 12** shows unfairness μ_u and slowdown standard deviation σ_s. In all cases, the makespan of the very short `SimuBloch` executions is significantly reduced for `Fairness`. The evolution of η_u is coherent with the first experiment: a common initialization phase followed by an anticipated growth and decrease for `Fairness`. `Fairness` reduces σ_s up to a factor of 5.9 and unfairness up to a factor of 1.9. Table 13 shows the execution makespan (m), average wait time ($\bar{w}$) and slowdown (s) values for the `SimuBloch` execution launched after the 3 `GATE`. As it is a non-clairvoyant scenario where no information about task execution time and future task submission is known, the fairness mechanism is not able to give higher priorities to `SimuBloch` tasks in advance. Despite that, the fairness mechanism speeds up `SimuBloch` executions up to a factor of 2.9, reduces task average wait time up to factor of 4.4 and reduces slowdown up to a factor of 5.9.

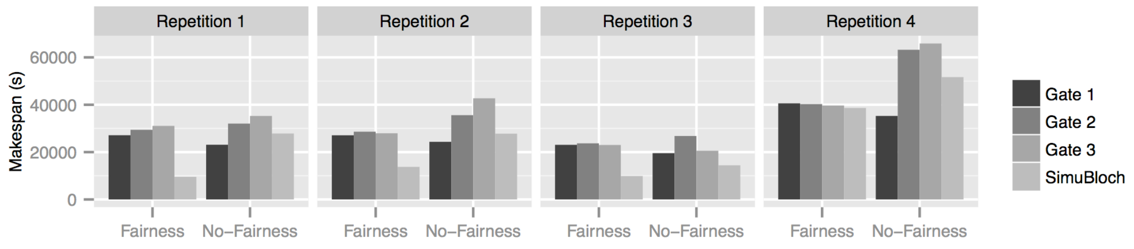


Figure 13: Comparison of the makespan for 3 identical workflows and a very short workflow.

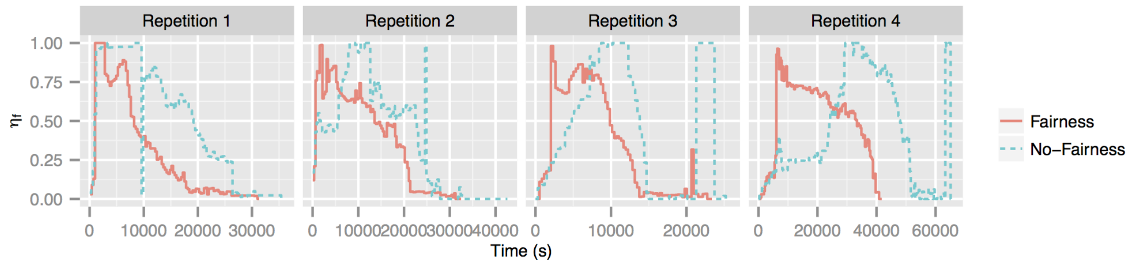


Figure 14: Unfairness degree η_u for a set of identical workflows and a very short workflow.

	Repetition 1		Repetition 2		Repetition 3		Repetition 4	
	σ_s	$\mu(s)$	σ_s	$\mu(s)$	σ_s	$\mu(s)$	σ_s	$\mu(s)$
NF	94.88	7269	100.05	16048	87.93	11331	213.60	28190
F	15.95	9085	42.94	12543	57.62	7721	76.69	21355

Table 12. Slowdown standard deviation σ_s and unfairness μ_u.

Run	Type	m (secs)	$\bar{w}$ (secs)	s
1	No-Fairness	27854	18983	196.15
	Fairness	9531	4313	38.43
2	No-Fairness	27784	19105	210.48
	Fairness	13761	10538	94.25
3	No-Fairness	14432	13579	182.48
	Fairness	9902	8145	122.25
4	No-Fairness	51664	47591	445.38
	Fairness	38630	27795	165.79

Table 13: `SimuBloch`'s makespan, average wait time and slowdown.

In all experiments, fairness optimization takes time to begin because the method needs to acquire information about the applications, which are totally unknown when a workflow is launched. We could think of reducing the time of this information-collecting phase, e.g. by designing initialization strategies maximizing information discovery, but it could not be totally removed. Currently, the method works best for applications with a lot of short tasks because the first few tasks can be used for initialization, and optimization can be exploited for the remaining tasks. The worst-case scenario is a configuration where the number of available resources stays constant and equal to the number of tasks in the first submitted workflow: in this case, no action could be taken until the first workflow completes, and the method would not do better than first-come-first-served. Pre-emption of running tasks should be considered to address that.

INTERACTIONS BETWEEN TASK GRANULARITY AND FAIRNESS CONTROL

Adjusting task granularity obviously impacts resource allocation, therefore fairness among executions. We approach this issue from an experimental angle, testing the following hypotheses: 1) the granularity control loop reduces fairness among executions; and 2) the fairness control loop avoids this reduction (Ferreira da Silva et al., 2014).

Two experiments are conducted. The first experiment tests whether the task granularity control process penalizes fairness among workflow executions; and the second tests whether the fairness control process mitigates the unfairness created by the granularity control process. For each experiment, a workflow set where one workflow uses the granularity control process (`Granularity`, `G`) is compared to a control workflow set (`No-Granularity`, `NG`). A workflow set consists of three `SimuBloch` workflows submitted sequentially. In the `Granularity` set, the first workflow has the granularity control process enabled, and the others do not. The first experiment has a fairness service, which only measures the unfairness among workflow executions, but no action is triggered. In the second experiment, task prioritization is triggered once unfairness is detected.

Both experiments are launched simultaneously to ensure similar grid conditions. For each grouped task resubmitted in the `Granularity` execution, a task in the `No-Granularity` is resubmitted too in each experiment to ensure equal race conditions for resource allocation. Similarly, for each task prioritized in the first experiment, a task in the second is also prioritized to ensure equal race conditions. Again, experiment results are not influenced by the submission process overhead since both `Granularity` and `No-Granularity` of both experiments experience the same overhead. Therefore, performance results obtained in both experiments can be compared to each other.

Results and discussion. Figure 15 shows the comparison of the slowdown for the first experiment. Unfairness degree η_u is shown in Figure 16. **Table 14** shows the makespan standard deviation σ_m, slowdown standard deviation σ_s, and unfairness μ_u. Both `Granularity` and `No-Granularity` executions behave similarly until η_f reaches the threshold value $\tau_f = 0.55$. Tasks are considered too fine and the mechanism triggers task grouping. In all cases, the slowdown of the workflow executed with granularity (the first one on the `Granularity` set) is the lowest, i.e. its execution benefits of the grouping mechanism by saving waiting times and data transfers of shared input data. In the workflow set where the granularity control process is enabled, the unfairness value is up to a factor of 2 higher when compared to the workflow set where the granularity is disabled (`No-Granularity`).

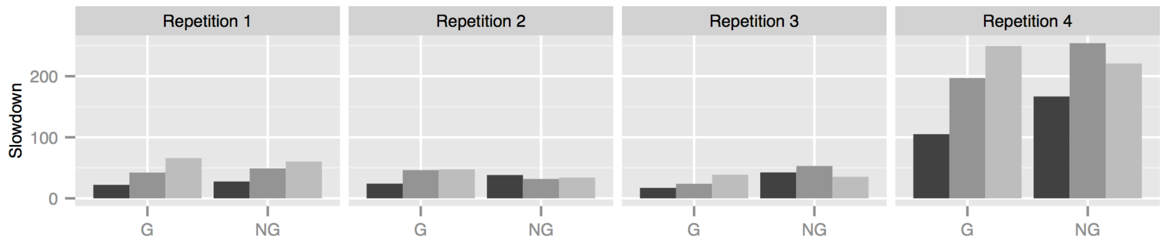


Figure 15: Granularity without fairness: comparison of the slowdowns.

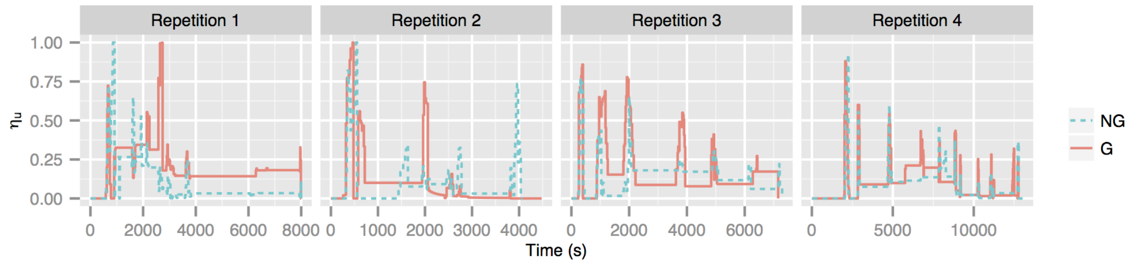


Figure 16: Granularity without fairness: unfairness degree η_u.

	Repetition 1			Repetition 2			Repetition 3			Repetition 4		
	$\sigma_m(s)$	σ_s	$\mu(s)$	$\sigma_m(s)$	σ_s	$\mu(s)$	$\sigma_m(s)$	σ_s	$\mu(s)$	$\sigma_m(s)$	σ_s	$\mu(s)$
NF	2446	16.55	823	15	3.24	503	1273	8.91	998	1364	38.70	1040
F	2962	21.94	1638	1015	13.15	527	2566	11.01	1212	2017	73.90	1250

Table 14. Granularity without fairness: unfairness and standard deviation of the makespan and slowdown.

Figure 17 shows the comparison of slowdown for the second experiment. Unfairness degree η_u is shown in Figure 18. **Table 15** shows makespan standard deviation σ_m, slowdown standard deviation σ_s, and unfairness μ_u. Both `Granularity` and `No-Granularity` executions have similar unfairness values. The same behavior is observed in σ_m and σ_s for repetitions 1, 3, and 4. This is not the case of repetition 2, in which resources suddenly appeared while tasks were being grouped. This resulted in parallelism loss for some workflow executions while the others were less impacted.

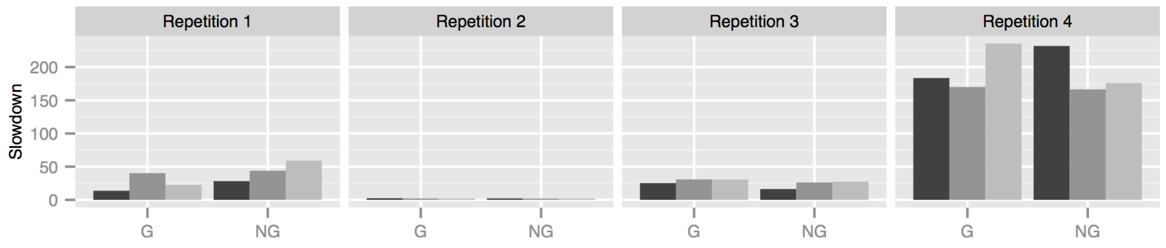


Figure 17: Granularity with fairness: comparison of the slowdowns.

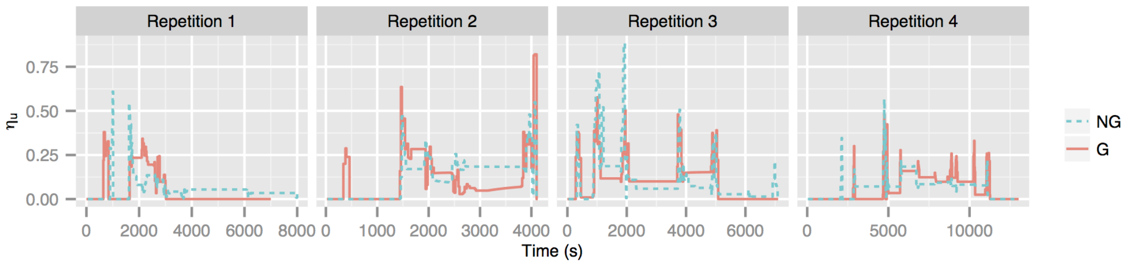


Figure 18: Granularity with fairness: unfairness degree η_u.

	Repetition 1			Repetition 2			Repetition 3			Repetition 4		
	$\sigma_m(s)$	σ_s	$\mu(s)$	$\sigma_m(s)$	σ_s	$\mu(s)$	$\sigma_m(s)$	σ_s	$\mu(s)$	$\sigma_m(s)$	σ_s	$\mu(s)$
NF	2548	15.41	445	78	0.11	501	2768	6.11	718	1044	35.37	826
F	2384	13.47	335	456	0.23	455	1674	3.11	712	1028	34.53	754

Table 15. Granularity with fairness: unfairness and standard deviation of the makespan and slowdown.

FUTURE RESEARCH DIRECTIONS

The self-management method introduced in this chapter demonstrated its effectiveness to handle operational incidents on workflow executions. The use of a MAPE-K loop is fundamental to achieve a fair quality of service by using control loops that constantly perform online monitoring, analysis, and execution of a set of curative actions. However, some limitations are also identified. For instance, the method needs to acquire information about the applications, which are completely unknown when a workflow is launched. When handling blocked activities, this limitation delays the decision to replicate a task; at least two tasks should be finished to estimate the median durations of each phase. The same delay is observed when handling the granularity of tasks, where tasks are grouped once an estimation of the duration is available. For the fairness process, the relative observed duration parameter also depends on task duration estimations, thus the metric does not consider this parameter while the estimations are not available. One approach to circumvent this issue could be to initialize such estimations according to observed distributions of these values, adjusting the estimations along the workflow execution.

CONCLUSION

In this chapter, we introduced the Virtual Imaging Platform (VIP), an openly accessible online science-gateway for medical imaging simulation, which provides access to distributed computing and storage resources. We then addressed the autonomic management of workflow executions on VIP in an online and non-clairvoyant environment. We introduced our general self-management mechanism, based on the MAPE-K loop, to cope with operational incidents of workflow executions. Then, we showed the application of our method to handle late task executions, task granularities, and unfairness among workflow executions.

The self-mechanism method proposed in this chapter demonstrated its effectiveness to handle operational incidents on workflow executions. The use of a MAPE-K loop is fundamental to achieve a fair quality of service by using control loops that constantly perform online monitoring, analysis, and execution of a set of curative actions. Although we showed the application of the self-management method in a medical imaging science-gateway using a grid infrastructure, the method is general enough to be used by other platforms and infrastructures.

REFERENCES

Agrawal, R., Imieliński, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, *22*(2), 207–216. doi:10.1145/170036.170072

Ang, T. F., Ng, W. K., Ling, T. C., Por, L. Y., & Liew, C. S. (2009). A Bandwidth-Aware Job Grouping-Based Scheduling on Grid Environment. *Information Technology Journal*, *8*(3), 372–377. doi:10.3923/itj.2009.372.377

Arabnejad, H., & Barbosa, J. (2012). Fairness Resource Sharing for Dynamic Workflow Scheduling on Heterogeneous Systems (pp. 633–639). IEEE. doi:10.1109/ISPA.2012.94

Ben-Yehuda, O. A., Schuster, A., Sharov, A., Silberstein, M., & Iosup, A. (2012). ExPERT: Pareto-Efficient Task Replication on Grids and a Cloud (pp. 167–178). IEEE. doi:10.1109/IPDPS.2012.25

Casanova, H., Desprez, F., & Suter, F. (2010). On cluster resource allocation for multiple parallel task graphs. *Journal of Parallel and Distributed Computing*, *70*(12), 1193–1203. doi:10.1016/j.jpdc.2010.08.017

Chen, W., Ferreira da Silva, R., Deelman, E., & Sakellariou, R. (2013). Balanced Task Clustering in Scientific Workflows (pp. 188–195). IEEE. doi:10.1109/eScience.2013.40

Cirne, W., Brasileiro, F., Paranhos, D., Góes, L. F. W., & Voorsluys, W. (2007). On the efficacy, efficiency and emergent behavior of task replication in large distributed systems. *Parallel Computing*, *33*(3), 213–234. doi:10.1016/j.parco.2007.01.002

De Jong, K. A. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. University of Michigan.

Ferreira da Silva, R., Camarasu-Pop, S., Grenier, B., Hamar, V., Manset, D., Montagnat, J., … Glatard, T. (2011). Multi-infrastructure workflow execution for medical simulation in the Virtual Imaging Platform. In *2011 HealthGrid Conference*.

Ferreira da Silva, R., & Glatard, T. (2013). A Science-Gateway Workload Archive to Study Pilot Jobs, User Activity, Bag of Tasks, Task Sub-steps, and Workflow Executions. In I. Caragiannis, M. Alexander, R. M. Badia, M. Cannataro, A. Costan, M. Danelutto, … J. Weidendorfer (Eds.), *Euro-Par 2012: Parallel Processing Workshops* (pp. 79–88). Springer Berlin Heidelberg. Retrieved from http://link.springer.com/chapter/10.1007/978-3-642-36949-0_10

Ferreira da Silva, R., Glatard, T., & Desprez, F. (2012). Self-Healing of Operational Workflow Incidents on Distributed Computing Infrastructures (pp. 318–325). IEEE. doi:10.1109/CCGrid.2012.24

Ferreira da Silva, R., Glatard, T., & Desprez, F. (2013a). On-Line, Non-clairvoyant Optimization of Workflow Activity Granularity on Grids. In F. Wolf, B. Mohr, & D. an Mey (Eds.), *Euro-Par 2013 Parallel Processing* (pp. 255–266). Springer Berlin Heidelberg. Retrieved from http://link.springer.com/chapter/10.1007/978-3-642-40047-6_28

Ferreira da Silva, R., Glatard, T., & Desprez, F. (2013b). Self-healing of workflow activity incidents on distributed computing infrastructures. *Future Generation Computer Systems*, *29*(8), 2284–2294. doi:10.1016/j.future.2013.06.012

Ferreira da Silva, R., Glatard, T., & Desprez, F. (2013c). Workflow Fairness Control on Online and Non-clairvoyant Distributed Computing Platforms. In F. Wolf, B. Mohr, & D. an Mey (Eds.), *Euro-Par 2013 Parallel Processing* (pp. 102–113). Springer Berlin Heidelberg. Retrieved from http://link.springer.com/chapter/10.1007/978-3-642-40047-6_13

Ferreira da Silva, R., Glatard, T., & Desprez, F. (2014). Controlling fairness and task granularity in distributed, online, non-clairvoyant workflow executions: CONTROLLING FAIRNESS AND TASK GRANULARITY IN WORKFLOWS. *Concurrency and Computation: Practice and Experience*, n/a–n/a. doi:10.1002/cpe.3303

Ferreira da Silva, R., Juve, G., Deelman, E., Glatard, T., Desprez, F., Thain, D., … Livny, M. (2013). Toward fine-grained online task characteristics estimation in scientific workflows (pp. 58–67). ACM Press. doi:10.1145/2534248.2534254

Foster, I. (2001). The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, *15*(3), 200–222. doi:10.1177/109434200101500302

Glatard, T., Lartizien, C., Gibaud, B., Ferreira da Silva, R., Forestier, G., Cervenansky, F., … Friboulet, D. (2013). A Virtual Imaging Platform for Multi-Modality Medical Image Simulation. *IEEE Transactions on Medical Imaging*, *32*(1), 110–118. doi:10.1109/TMI.2012.2220154

Glatard, T., Montagnat, J., Lingrand, D., & Pennec, X. (2008). Flexible and Efficient Workflow Deployment of Data-Intensive Applications On Grids With MOTEUR. *International Journal of High Performance Computing Applications*, *22*(3), 347–360. doi:10.1177/1094342008096067

Henan Zhao, & Sakellariou, R. (2006). Scheduling multiple DAGs onto heterogeneous systems (p. 14 pp.). IEEE. doi:10.1109/IPDPS.2006.1639387

Hirales-Carbajal, A., Tchernykh, A., Yahyapour, R., González-García, J. L., Röblitz, T., & Ramírez-Alcaraz, J. M. (2012). Multiple Workflow Scheduling Strategies with User Run Time Estimates on a Grid. *Journal of Grid Computing*, *10*(2), 325–346. doi:10.1007/s10723-012-9215-6

Hsu, C.-C., Huang, K.-C., & Wang, F.-J. (2011). Online scheduling of workflow applications in grid environments. *Future Generation Computer Systems*, *27*(6), 860–870. doi:10.1016/j.future.2010.10.015

Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. *Computer*, *36*(1), 41–50. doi:10.1109/MC.2003.1160055

Litke, A., Skoutas, D., Tserpes, K., & Varvarigou, T. (2007). Efficient task replication and management for adaptive fault tolerance in Mobile Grid environments. *Future Generation Computer Systems*, *23*(2), 163–178. doi:10.1016/j.future.2006.04.014

Liu, Q., & Liao, Y. (2009). Grouping-Based Fine-Grained Job Scheduling in Grid Computing (pp. 556–559). IEEE. doi:10.1109/ETCS.2009.132

Malik, D. S., Mordeson, J. N., & Sen, M. K. (1994). On subsystems of a fuzzy finite state machine. *Fuzzy Sets and Systems*, *68*(1), 83–92. doi:10.1016/0165-0114(94)90274-7

Muthuvelu, N., Chai, I., Chikkannan, E., & Buyya, R. (2010). On-Line Task Granularity Adaptation for Dynamic Grid Applications. In C.-H. Hsu, L. T. Yang, J. H. Park, & S.-S. Yeo (Eds.), *Algorithms and Architectures for Parallel Processing* (Vol. 6081, pp. 266–277). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://www.springerlink.com/index/10.1007/978-3-642-13119-6_24

Muthuvelu, N., Chai, I., & Eswaran, C. (2008). An Adaptive And Parameterized Job Grouping Algorithm For Scheduling Grid Jobs (pp. 975–980). IEEE. doi:10.1109/ICACT.2008.4493929

Muthuvelu, N., Liu, J., Soe, N. L., Venugopal, S., Sulistio, A., & Buyya, R. (2005). A Dynamic Job Grouping-based Scheduling for Deploying Applications with Fine-grained Tasks on Global Grids. In *Proceedings of the 2005 Australasian Workshop on Grid Computing and e-Research - Volume 44* (pp. 41–48). Darlinghurst, Australia, Australia: Australian Computer Society, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=1082290.1082297

N'Takpe, T., & Suter, F. (2009). Concurrent scheduling of parallel task graphs on multi-clusters using constrained resource allocations (pp. 1–8). IEEE. doi:10.1109/IPDPS.2009.5161161

Ng Wai Keat, T. A. (2006). Scheduling framework for bandwidth-aware job grouping-based scheduling in grid computing. *Malaysian Journal of Computer Science*, *19*.

Open Science Grid. (2014). Retrieved from http://www.opensciencegrid.org

Ramakrishnan, L., Huang, T. M., Thyagaraja, K., Zagorodnov, D., Koelbel, C., Kee, Y.-S., … Mandal, A. (2009). VGrADS: enabling e-Science workflows on grids and clouds with fault tolerance (p. 1). ACM Press. doi:10.1145/1654059.1654107

Romanus, M., Mantha, P. K., McKenzie, M., Bishop, T. C., Gallichio, E., Merzky, A., … Jha, S. (2012). The Anatomy of Successful ECSS Projects: Lessons of Supporting High-throughput High-performance Ensembles on XSEDE. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond* (pp. 46:1–46:9). New York, NY, USA: ACM. doi:10.1145/2335755.2335843

Sabin, G., Kochhar, G., & Sadayappan, P. (2004). Job fairness in non-preemptive job scheduling (pp. 186–194 vol.1). IEEE. doi:10.1109/ICPP.2004.1327920

Skowron, P., & Rzadca, K. (2013). Non-monetary fair scheduling: a cooperative game theory approach (p. 288). ACM Press. doi:10.1145/2486159.2486169

Sommerfeld, D., & Richter, H. (2011). Efficient Grid Workflow Scheduling Using a Two-Tier Approach. Presented at the HealthGrid 2011.

Tan, P.-N. (2006). *Introduction to data mining* (1st ed.). Boston: Pearson Addison Wesley.

Taylor, I. J. (2007). *Workflows for e-science scientific workflows for grids*. London: Springer. Retrieved from http://public.eblib.com/EBLPublic/PublicView.do?ptiID=337445

Tsaregorodtsev, A., Brook, N., Ramo, A. C., Charpentier, P., Closier, J., Cowan, G., … Zhelezov, A. (2010). DIRAC3 – the new generation of the LHCb grid software. *Journal of Physics: Conference Series*, *219*(6), 062029. doi:10.1088/1742-6596/219/6/062029

XSEDE. (2014). Retrieved from http://www.xsede.org

Zomaya, A. Y., & Chan, G. (2004). Efficient clustering for parallel tasks execution in distributed systems (pp. 167–174). IEEE. doi:10.1109/IPDPS.2004.1303164

ADDITIONAL READING

Callaghan, S., Deelman, E., Gunter, D., Juve, G., Maechling, P., Brooks, C., ... & Jordan, T. (2010). Scaling up workflow-based applications. Journal of Computer and System Sciences, 76(6), 428-446.

Callaghan, S., Maechling, P., Small, P., Milner, K., Juve, G., Jordan, T. H., ... & Brooks, C. (2011). Metrics for heterogeneous scientific workflows: A case study of an earthquake science application. International Journal of High Performance Computing Applications, 1094342011414743.

Camarasu-Pop, S., Glatard, T., Da Silva, R. F., Gueth, P., Sarrut, D., & Benoit-Cattin, H. (2013). Monte Carlo simulation on heterogeneous distributed systems: A computing framework with parallel merging and checkpointing strategies. Future Generation Computer Systems, 29(3), 728-738.

Camarasu-Pop, S., Glatard, T., Mościcki, J. T., Benoit-Cattin, H., & Sarrut, D. (2010). Dynamic partitioning of GATE Monte-Carlo simulations on EGEE. Journal of Grid Computing, 8(2), 241-259.

Chen, W., & Deelman, E. (2011). Workflow overhead analysis and optimizations. In Proceedings of the 6th workshop on Workflows in support of large-scale science (pp. 11-20). ACM.

Deelman, E., Gannon, D., Shields, M., & Taylor, I. (2009). Workflows and e-Science: An overview of workflow system features and capabilities. Future Generation Computer Systems, 25(5), 528-540.

Deelman, E., Juve, G., Malawski, M., & Nabrzyski, J. (2013). Hosted science: Managing computational workflows in the cloud. Parallel Processing Letters, 23(02).

Ferreira da Silva, R., Chen, W., Juve, G., Vahi, K., & Deelman, E. (2014) Community Resources for Enabling Research in Distributed Scientific Workflows. 10th IEEE International Conference on e-Science.

Gil, Y., González-Calero, P. A., & Deelman, E. (2007). On the black art of designing computational workflows. In Proceedings of the 2nd workshop on Workflows in support of large-scale science (pp. 53-62). ACM.

Glatard, T., Rousseau, M. E., Camarasu-Pop, S., Rioux, P., Sherif, T., Beck, N., ... & Evans, A. C. (2014) Interoperability between the CBRAIN and VIP web platforms for neuroimage analysis.

Hoffa, C., Mehta, G., Freeman, T., Deelman, E., Keahey, K., Berriman, B., & Good, J. (2008). On the use of cloud computing for scientific workflows. In eScience, 2008. eScience'08. IEEE Fourth International Conference on (pp. 640-645). IEEE.

Juve, G., Chervenak, A., Deelman, E., Bharathi, S., Mehta, G., & Vahi, K. (2013). Characterizing and profiling scientific workflows. Future Generation Computer Systems, 29(3), 682-692.

Kandaswamy, G., Mandal, A., & Reed, D. A. (2008). Fault tolerance and recovery of scientific workflows on computational grids. In Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on (pp. 777-782). IEEE.

Kumar, V. S., Sadayappan, P., Mehta, G., Vahi, K., Deelman, E., Ratnakar, V., ... & Saltz, J. (2009). An integrated framework for performance-based optimization of scientific workflows. In Proceedings of the 18th ACM international symposium on High performance distributed computing (pp. 177-186). ACM.

Malawski, M., Juve, G., Deelman, E., & Nabrzyski, J. (2012). Cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (p. 22). IEEE Computer Society Press.

Montagnat, J., Isnard, B., Glatard, T., Maheshwari, K., & Fornarino, M. B. (2009). A data-driven workflow language for grids based on array programming principles. In Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science (p. 7). ACM.

Olabarriaga, S. D., Jaghoori, M. M., Korkhov, V., van Schaik, B., & van Kampen, A. (2013, November). Understanding workflows for distributed computing: nitty-gritty details. In Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science (pp. 68-76). ACM.

Plankensteiner, K., Prodan, R., & Fahringer, T. (2009). A new fault tolerance heuristic for scientific workflows in highly distributed environments based on resubmission impact. In e-Science, 2009. e-Science'09. Fifth IEEE International Conference on (pp. 313-320). IEEE.

Russell, N., van der Aalst, W., & ter Hofstede, A. (2006, January). Workflow exception patterns. In Advanced Information Systems Engineering (pp. 288-302). Springer Berlin Heidelberg.

Samak, T., Gunter, D., Goode, M., Deelman, E., Mehta, G., Silva, F., & Vahi, K. (2011). Failure prediction and localization in large scientific workflows. In Proceedings of the 6th workshop on Workflows in support of large-scale science (pp. 107-116). ACM.

Shahand, S., Benabdelkader, A., Jaghoori, M. M., Mourabit, M. A., Huguet, J., Caan, M. W., ... & Olabarriaga, S. D. (2014). A data-centric neuroscience gateway: design, implementation, and experiences. Concurrency and Computation: Practice and Experience.

Singh, G., Vahi, K., Ramakrishnan, A., Mehta, G., Deelman, E., Zhao, H., ... & Katz, D. S. (2007). Optimizing workflow data footprint. Scientific Programming, 15(4), 249-268.

Vöckler, J. S., Juve, G., Deelman, E., Rynge, M., & Berriman, B. (2011). Experiences using cloud computing for a scientific workflow application. In Proceedings of the 2nd international workshop on Scientific cloud computing (pp. 15-24). ACM.

Wieczorek, M., Hoheisel, A., & Prodan, R. (2008). Taxonomies of the multi-criteria grid workflow scheduling problem. In Grid Middleware and Services (pp. 237-264). Springer US.

Zhang, Y., Mandal, A., Koelbel, C., & Cooper, K. (2009). Combined fault tolerance and scheduling techniques for workflow applications on computational grids. In Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on (pp. 244-251). IEEE.

KEY TERMS AND DEFINITIONS

Grid Computing: Federation of heterogeneous resources distributed geographically in different administrative domains to provide computing and storage services for research communities.

Scientific Gateway: Integrates application software with access to computing and storage resources via web portals or desktop applications.

Scientific Workflow: Allows users to easily express multi-step computational tasks, for example retrieve data from an instrument or a database, reformat the data, and run an analysis

Task Grouping: Groups fine-grained tasks into coarse-grained tasks to reduce the scheduling and queuing time overheads inherent to distributed computing platforms.

Task Replication: Common technique to increase the probability of successfully complete task executions in distributed computing platforms.

Task Resubmission: Most common technique to address failures on task executions in distributed computing platforms.

Unfairness Among Workflow Executions: Computing resources are not fairly (i.e. proportionally) allocated to workflow applications. It occurs when the demand is higher than the offer, that is, when some workflows are slowed down by concurrent executions.

BIOGRAPHY

Rafael Ferreira da Silva is a Computer Scientist in the Collaborative Computing Group at the USC Information Sciences Institute. He received his PhD in Computer Science from INSA-Lyon, France, in 2013. In 2010, he received his Master's degree in Computer Science from Universidade Federal de Campina Grande, Brazil, and his BS degree in Computer Science from Universidade Federal da Paraiba, in 2007. His research focuses on the execution of scientific workflows on heterogeneous distributed systems such as Clouds and Grids. See http://www.rafaelsilva.com for further information.

Tristan Glatard obtained a PhD in grid computing applied to medical image analysis from the University of Nice Sophia-Antipolis in 2007. He was a post-doc at the University of Amsterdam in 2008. He is now a researcher at CNRS Creatis in Lyon, working on distributed systems for medical imaging applications.

Frédéric Desprez is a Chief Senior Research Scientist at Inria and holds a position at the LIP laboratory (ENS Lyon, France). He co-founded the SysFera company where he holds a position as scientific advisor. He received his PhD in C.S. from Institut National Polytechnique de Grenoble, France, in 1994 and his MS in C.S. from ENS Lyon in 1990. His research interests include parallel algorithms, scheduling for large scale distributed platforms, data management, and grid and cloud computing. He leads the Grid'5000 project, which offers a platform to evaluate large-scale algorithms, applications, and middleware systems. See http://graal.ens-lyon.fr/~desprez/ for further information.