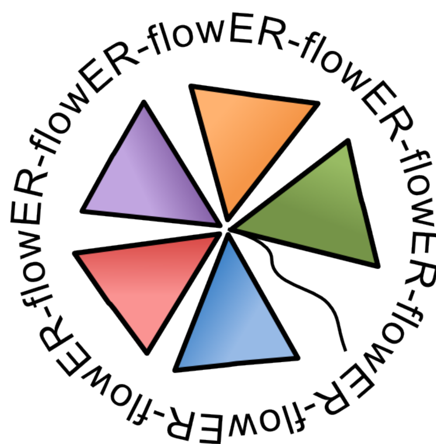


Project number: **RI 312579**

Project acronym: **ER-flow**

Project full title:
Building an European Research Community through Interoperable Workflows and Data

Theme: **Research Infrastructures**
Call Identifier: **FP7-Infrastructures-2012-1**

Funding Scheme: **Coordination and Support Action**

D4.2: Study of Virtual Data Objects generation and error recovery in a CGI

CNRS

Due date of milestone: 01/03/2013 Start date of project: 01/09/2012	Actual submission date: 26/02/2014 Duration: 24 months
Dissemination Level: PU	

Contents

1 Introduction **5**
1.1 Context and goal of the document 5

2 A self-healing process for workflow execution on grids **6**

3 Application to error detection and handling **9**
3.1 Incident Degrees 9
3.2 Incident Levels and Association Rules 11
3.2.1 Training Dataset 11
3.2.2 Incident Levels 12
3.2.3 Association Rules 12
3.3 Actions 15
3.3.1 Task replication 15
3.3.2 File Replication 15
3.3.3 Site Blacklisting 15
3.4 Experiments 16
3.4.1 Experiment conditions and metrics 16
3.4.2 Results and Discussion 17

4 Application to granularity control **21**
4.1 Task Granularity Control Process 21
4.1.1 Fineness control 21
4.1.2 Coarseness control 23
4.2 Experiments and Results 24
4.2.1 Experiment Conditions 24
4.2.2 Results and Discussion 25
4.3 Conclusion 27

5 Application to fairness control **29**
5.1 Fairness control process 29
5.1.1 Measuring unfairness: η_u. 29
5.1.2 Thresholding unfairness: τ_u. 31
5.1.3 Task prioritization. 31
5.2 Experiments and results 32
5.2.1 Experiment conditions 32
5.2.2 Results and discussion 34
5.3 Conclusion 36

6 General conclusion **37**

References **39**

List of Figures

1 Fuzzy Finite State Machine (FuSM) representing an activity. 7
2 One iteration of the healing process. 7
3 Example case showed as a MAPE-K loop. 8
4 Task estimation based on median values. 9
5 Cumulative amount of running activities from April to August 2011. 12
6 Histograms of incident degrees sampled in bins of 5%. 13
7 Replication process for one task. 16
8 Site misconfigured: replication process for one file. 16
9 Execution makespan for `FIELD-II/pasa` (top) and `Mean-Shift/hs3` (bottom). . . . 18
10 Experiment 1: CDF of the number of completed tasks for `FIELD-II/pasa` repetitions. 19
11 Experiment 1: CDF of the number of completed tasks for `Mean-Shift/hs3` repetitions. 20
12 Experiment 2: makespan of `FIELD-II/pasa` and `Mean-Shift/hs3` for 3 different runs. 20
13 Distribution of sites and batch queues per country in the biomed VO (January 2013) (*left*) and histogram of fineness incident degree sampled in bins of 0.05 (*right*). . . . 23
14 Experiment 1: makespan for `Fineness` and `No-Granularity` executions for the 3 workflow activities under stationary load. 26
15 Experiment 2: makespan (top) and evolution of task groups (bottom) for `FIELD-II` executions under non-stationary load (resources arrive during the experiment). . . . 27
16 Distribution of sites and batch queues per country in the biomed VO (January 2013) (*left*) and histogram of the unfairness degree η_u sampled in bins of 0.05 (*right*). . . . 31
17 Experiment 1 (identical workflows). Top: comparison of the makespans; middle: unfairness degree η_u; bottom: makespan standard deviation σ_m, slowdown standard deviation σ_s and unfairness μ. 35
18 Experiment 2 (very short execution). Top: comparison of the makespans; middle: unfairness degree η_u; bottom: unfairness μ and slowdown standard deviation. 36
19 Experiment 3 (different workflows). Top: comparison of the slowdown; middle: unfairness degree η_u; bottom: unfairness μ and slowdown standard deviation. 37

List of Tables

1 Status of deliverable 4
2 Change History 4
3 Example case. 8
4 Distribution of sites and batch queues per country in the biomed VO (January 2013). 11
5 Incident levels and actions. 14
6 Confidence of rules between incident levels. 15
7 Waste coefficient values for `FIELD-II/pasa`. 18
8 Waste coefficient values for `Mean-Shift/hs3`. 19
9 Number of submitted faulty tasks. 19
10 Example 24
11 Experiment 1: makespan (M) and number of task groups for `SimuBloch`, `FIELD-II` and `PET-Sorteo/emission` executions for the 5 repetitions. 26
12 Experiment 2: makespan (M) and average queuing time ($\bar{q}$) for `FIELD-II` workflow execution for the 5 repetitions. 27
13 Example 33
14 Workflow characteristics ($\rightarrow$ indicate task dependencies). 33
15 Experiment 2: `SimuBloch`'s makespan, average wait time and slowdown. 35

Status and Change History

Status	Name	Date	Signature
Draft	T. Glatard, R. Ferreira da Silva, F. Desprez, J. Montagnat	26/02/2014	n.n electronically
Reviewed	P. Kacsuk	25/02/2014	n.n electronically
Approved	G. Terstyanszky	26/02/2014	n.n electronically

Table 1: Status of deliverable

Version	Date	Pages	Author	Modification
0.1	11/02/2014	all	T. Glatard, R. Ferreira da Silva, F. Desprez	first draft
0.2	16/02/2014	all	J. Montagnat	corrections

Table 2: Change History

Contents

1 Introduction

1.1 Context and goal of the document

Science-gateways, such as the SHIWA Simulation Platform (SSP), provide access to important amounts of resources transparently. However, their large scale and the number of middleware systems involved lead to many system errors and faults. Easy-to-use interfaces provided by these gateways exacerbate the need for properly solving operational incidents encountered on grid computing infrastructures since end users expect high reliability and performance with no extra monitoring or parametrization from their side. In practice, such services are often backed by substantial support staff who monitors running experiments by performing simple yet crucial actions such as rescheduling tasks, restarting services, killing misbehaving runs or replicating data files to reliable storage facilities. Fair QoS can then be delivered, yet with numerous human intervention. For instance, SSP administration is time consuming and accurate expertise on workflow execution and grid computing. Failures may arise from the task execution level, e.g. application execution failures, or unavailability of files, up to the web portal level, e.g as unscheduled downtimes.

Automating such operations is challenging for two reasons. First, no reliable user activity prediction can be assumed, and new workloads may arrive at any time. Therefore, the considered metrics, decisions and actions have to remain simple and to yield results while the application is still executing. Second, there is a lack of information on applications and resources in production conditions. Computing resources are usually dynamically provisioned from heterogeneous clusters, clouds or desktop grids without any reliable estimate of their availability and characteristics. Models of application execution times are hardly available either, in particular on heterogeneous computing resources. To the best of our knowledge, there is no operation management framework making no assumption on workload variations (*online operation*) nor computing resources availability (*non-clairvoyant*).

This deliverable summarizes our findings on the third objective of WP4:

> to study how errors can be recovered efficiently by minimizing the amount of re-computations needed. (Reliability)

At the time of writing the ER-flow proposal, the plan was to tackle the reliability problem in meta-workflows by leveraging the Virtual Data Object (VDO) concept developed in WP4 and enabling partial re-execution of faulty sub-workflows, so that the meta-workflow is not impacted by sub-components failure. However, we chose to focus on task-level reliability instead in view of the challenges highlighted in ER-flow. ER-flow users reported several reliability issues encountered with the SSP, which resulted in a (on-going) discussion to improve resource monitoring at the infrastructure level:

- Setting up of a Nagios instance for resource monitoring in the SHIWA VO;
- Support of the SHIWA VO in the VAPOR portal[1].

In this deliverable, we propose and evaluate a set of error detection and recovery algorithms for workflows integrated in a general self-healing process that operates at the science-gateway level. We show how the methods presented here efficiently complement infrastructure-level solutions, and therefore could be useful for the SSP operated by ER-flow.

The methods presented in this document are all implemented and tested in the MOTEUR workflow system, officially supported by the SHIWA Simulation Platform. As this work is prospective

[1] https://operations-portal.egi.eu/vapor

and required platform adaptations and testing, the experiments were not conducted on the SSP production platform but through the Virtual Imaging Platform (VIP), a science-gateway with a workflow-based architecture similar to the SSP. There is no fundamental barrier to implementing these solutions in the SSP though.

The error recovery techniques studied in this deliverable are based on a general healing process for workflow executions. Workflows are compositions of *activities* that consist only of a program description. At runtime, activities receive data and spawn tasks for which the executable name and input data are known, but the computational cost and produced data volume are not. Workflow activities are modeled as Fuzzy Finite State Machines (FuSM) [19] where state degrees of membership are determined by an external healing 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. Based on incident degrees, the healing process identifies incident levels using thresholds determined from platform history. A specific set of actions is then selected from association rules among incident levels.

Section 2 describes our general self-healing process for workflow execution. We then instantiate it for error detection (section 3), granularity control (section 4), and finally fairness control among users (section 5).

2 A self-healing process for workflow execution on grids

A workflow activity is modeled as an FuSM with 13 states shown on Figure 1. The activity is initialized in `Submitting Invocations` where all the tasks are generated and submitted. Tasks consist of 4 successive phases: initialization, inputs download, application execution and output upload. They are all assumed independent, but with similar execution times (bag of tasks). `Running` is a state where no particular issue is detected; no action is taken and the activity is assumed to behave normally. `Completed` (resp. `Failed`) is a terminal state used when all the invocations are successfully completed (resp. at least one invocation failed). These 4 states are crisp (not fuzzy) and exclusive. Their degree can only be 0 or 1 and if 1 then all the other states have a degree of 0. The 9 other states are fuzzy states corresponding to detected incidents.

The healing process sets the degree of FuSM states from incident detection metrics and invocation statuses. Then, it determines actions to address the incidents. If no action is required then the process waits until an event occurs (task status change) or a timeout is reached.

Let $I = \{x_i, i = 1, \dots, n\}$ be the set of possible incidents (9 in this work) and $\eta = (\eta_1, \dots, \eta_n) \in [0, 1]^n$ their degrees in the FuSM. Incident x_i can occur at m_i different levels $\{x_{i,j}, j = 1, \dots, m_i\}$ delimited by threshold values $\tau_i = \{\tau_{i,j}, j = 1, \dots, m_i\}$. 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 $x_{i,j}$:

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

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

In addition to the incidents themselves, incident causes are taken into account. Association rules [1] 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 $P(x_{i,j}|x_{u,v})$. Note that $r^{i,j}_{i,j} \in R_{i,j}$ and $\rho^{i,j}_{i,j} = 1$. We also define $R = \bigcup_{i\in[\![1,n]\!],j\in[\![1,m_i]\!]} R_{i,j}$.

Figure 2 presents the algorithm used at each iteration of the healing process. Incident degrees are determined based on problem-specific metrics (see next sections) and incident levels j are obtained from historical data. A roulette wheel selection [6] 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

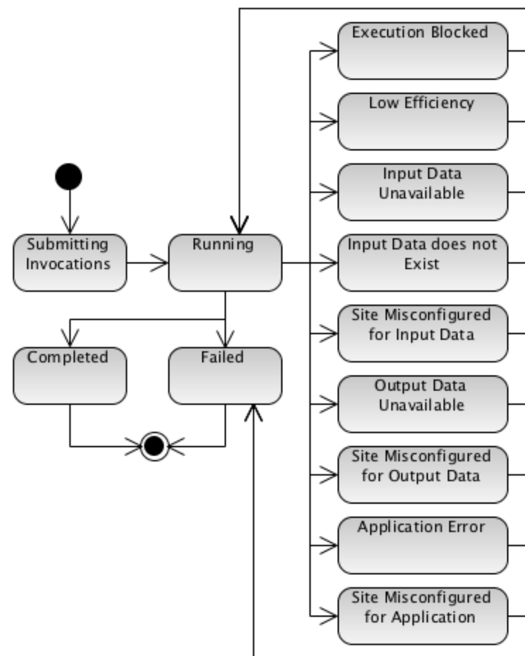


Figure 1: Fuzzy Finite State Machine (FuSM) representing an activity.

with a probability p_i proportional to its degree: $p(x_i) = \eta_i / \sum_{j=1}^{n} \eta_j$. A potential cause $x_{u,v}$ for incident $x_{i,j}$ is then selected from another roulette wheel selection on the association rules $r_{i,j}^{u,v}$, where x_u is at level v. Rule $r_{i,j}^{u,v}$ is weighted $\eta_u \times \rho_{i,j}^{u,v}$ in the 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.

Input: invocation statuses and history of η
Output: set of actions a

01. wait for event or timeout
02. determine incident degrees η based on metrics
03. determine incident levels j such that $\tau_{i,j} \leq \eta_i < \tau_{i,j+1}$
04. select incident x_i by roulette wheel selection based on η
05. select rule $r_{u,v} = (x_{u,v}, x_{i,j}, \rho_{i,j}^{u,v}) \in R_{i,j}$ by roulette wheel selection based on $\eta_u \times \rho_{i,j}^{u,v}$, where x_u is at level v
06. $a = a_u(v)$
07. perform actions in a

Figure 2: One iteration of the healing process.

Table 3 illustrates this mechanism on an example case where only 3 incidents are considered, and Figure 3 shows it as a MAPE-K loop.

Step 02 and 03: incident degrees and levels are determined:

x_i: incident name	Degree η_i	Level j
x_1: activity blocked	0.8	2
x_2: low efficiency	0.1	1
x_3: input data does not exist	0.4	1

Step 04: $x_{1,2}$ is selected with probability $\frac{0.8}{0.8+0.4+0.1}$.

Step 05: association rules $r^{2,1}_{1,2}$, $r^{3,1}_{1,2}$ and $r^{1,2}_{1,2}$ are considered:

Rule	Confidence
$r^{2,1}_{1,2}$: $x_{2,1} \rightarrow x_{1,2}$	0.8
$r^{3,1}_{1,2}$: $x_{3,1} \rightarrow x_{1,2}$	0.2
$r^{1,2}_{1,2}$: $x_{1,2} \rightarrow x_{1,2}$	1

$r^{2,1}_{1,2}$ is chosen with probability $\frac{0.8\times0.4}{0.8\times0.4+0.2\times0.1+0.8\times1}$.

Step 06: actions in $a_2(1)$ are performed.

Table 3: Example case.

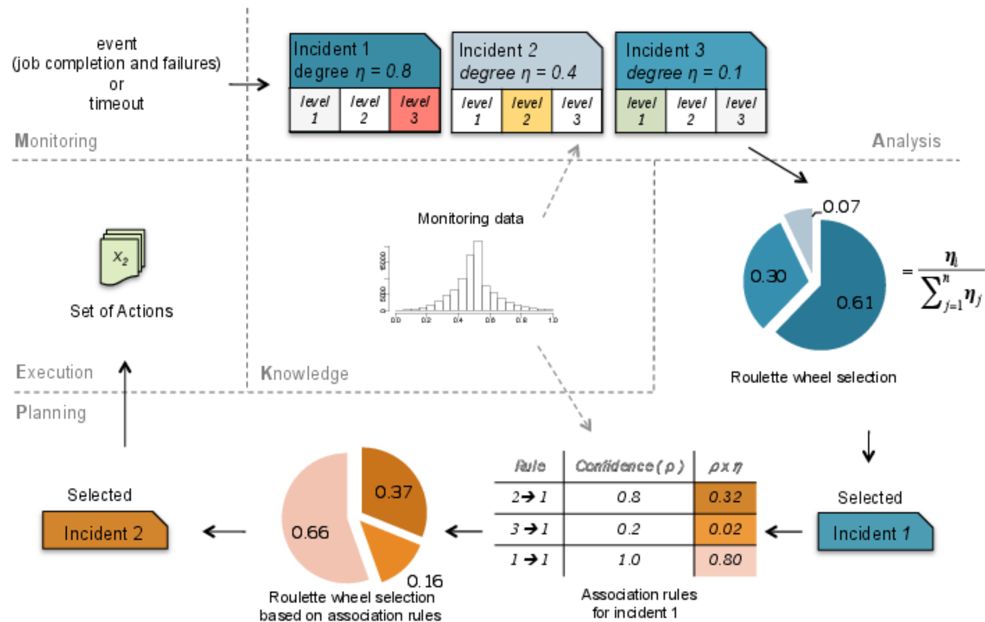


Figure 3: Example case showed as a MAPE-K loop.

3 Application to error detection and handling

This section presents a first example of the healing process on workflow activity incidents related to input and output data transfer errors, and application execution errors. First, we define metrics used to determine each incident degree; then, incident levels and association rules are defined from historical information and associated to action sets. Finally, experiments are conducted to evaluate the healing process in production conditions. We consider nine incidents: activity blocked, low efficiency, input data unavailability and non-existence, output data unavailability, application execution error, and site misconfiguration for input and output data, and application execution.

3.1 Incident Degrees

Activity Blocked. This incident happens when an invocation is considered late compared to the others. It is responsible for many operational issues, leading to substantial speed-up reductions. For instance, it occurs when one invocation of the activity requires more CPU cycles or when the invocation faces longer waiting times, lost tasks or executes on resources with poorer performance. We define the incident degree η_b of an activity from the max 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:

$$\eta_{\mathrm{b}} = 2.\max\left\{ p_i = p(t_i, \tilde{t}) = \frac{t_i}{\tilde{t} + t_i}, i \in [1, n] \right\} - 1 \tag{2}$$

where $t_i = t_{i_setup} + t_{i_input} + t_{i_exec} + t_{i_output}$ is the estimated duration of task i and $\tilde{t} = \tilde{t}_{setup} + \tilde{t}_{input} + \tilde{t}_{exec} + \tilde{t}_{output}$ 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.

The estimated duration t_i of a task is computed phase by phase, as follows: (i) for completed task phases, the actual consumed resource time is used; (ii) for ongoing task phases, the maximum value between the current consumed resource time and the median consumed time is taken; and (iii) for unstarted task phases, the time slot is filled by the median value. Figure 4 illustrates the estimation process of a task 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 15s, as it is not started yet.

t_{i_setup}

t_{i_input}

t_{i_exec}

t_{i_output}

Figure 4: Task estimation based on median values.

Low Efficiency. This happens when the time spent by all the activity invocations in data transfers dominates CPU time. It may be due to sites with poor network connectivity or be intrinsic to the application. The incident degree is defined from the ratio between the cumulative CPU

time C_i consumed by all completed invocations and the cumulative execution time of all completed invocations:

$$\eta_{\mathrm{e}} = 1 - \frac{\sum_{i=1}^{n(t)} C_i}{\sum_{i=1}^{n(t)}(C_i + D_i)}$$

where D_i is the time spent by invocation i in data transfers.

Input Data Unavailable. This happens when a file is registered in the file catalog but the storage resource(s) is(are) unavailable or unreachable. The incident degree η_{iu} in this state is determined from the input transfer failure rate due to data unavailability. Transfers of completed, failed, and running invocations are considered.

Input Data does not Exist. This happens when an incorrect data path was specified, the file was removed by mistake or the file catalog is unavailable or unreachable. Again, the incident degree η_{ie} is directly determined by the input transfer failure rate due to non-existent data. Non-existent file is distinguished from file unavailability using ad-hoc parsing of standard error files. Transfers of completed, failed, and running invocations are considered.

Site Misconfigured for Input Data. This incident happens when sites have utmost input data transfer failure rate. The incident degree η_{is} is measured as follows:

$$\eta_{\mathrm{is}} = \max(\phi_1, \phi_2, \ldots, \phi_k) - \mathrm{median}(\phi_1, \phi_2, \ldots, \phi_k)$$

where ϕ_i denotes the input transfer failure ratio (including both input data unavailable and input data does not exist) on site i and k is the number of white-listed sites used by the activity. The difference between the maximum rate and the median ensures that the incident degree has high values only when some sites are misconfigured. This metric is correlated but not redundant with the two previous ones. If some input data file is not available due to site-independent issues with the storage system, then η_{iu} will grow but η_{is} will remain low because all sites fail identically. On the contrary, η_{is} may grow while η_{iu} and η_{ie} remain low.

Output Data Unavailable. Output data can also be unavailable. Unavailability happens due to three main reasons: the user did not specify the output path correctly, the application did not produce the expected data, or the file catalog or storage resource are unavailable or unreachable. The incident degree η_{ou} is determined by the output transfer failure rate. Transfers of completed, failed and running invocations are considered.

Site Misconfigured for Output Data. The incident degree η_{os} in this incident is determined as follows:

$$\eta_{\mathrm{os}} = \max(\psi_1, \psi_2, \ldots, \psi_k) - \mathrm{median}(\psi_1, \psi_2, \ldots, \psi_k)$$

where ψ_i denotes the output transfer failure ratio on site i and k is the number of white-listed sites used by the activity.

Application Error Applications can fail due to a variety of reasons among which: the application executable is corrupted, dependencies are missing, or the executable is not compatible with the execution host. The incident degree η_{a} in this state is measured by the task failure rate due to application errors. Completed, failed, and running tasks are considered.

Site Misconfigured for Application The incident degree η_{as} in this state is measured as follows:

$$\eta_{\text{as}} = \max(\alpha_1, \alpha_2, \ldots, \alpha_k) - \text{median}(\alpha_1, \alpha_2, \ldots, \alpha_k)$$

where α_i denotes the task failure rate due to application errors on site i and k is the number of white-listed sites used by the activity.

3.2 Incident Levels and Association Rules

Incident degrees η_i are quantified in discrete incident levels so that different sets of actions can be used to address different levels of the incident. The number and values of the thresholds are determined from observed distributions of η_i. 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. Thresholds $\tau_{i,j}$ are determined from mode clustering. Incidents levels and thresholds are determined offline; thus they do not create any overhead on the workflow execution.

3.2.1 Training Dataset

Distributions of incident degrees were determined from the science-gateway workload archive [8] available in the grid observatory[2]. These traces were collected from the Virtual Imaging Platform [7] between April and August 2011. Applications deployed on this platform are described as workflows executed using the MOTEUR workflow engine [15]. Resource provisioning and task scheduling is provided by DIRAC [26] using so-called "pilot jobs". Resources are provisioned online with no advance reservations. Tasks are executed on the biomed virtual organization (VO) of the European Grid Infrastructure (EGI)[3] which, as of January 2013, has access to some 90 computing sites of 22 countries, offering 190 batch queues and approximately 4 PB of disk space. Table 4 shows the distribution of sites per country supporting the biomed VO. This dataset contains 1,082

Country	Number of sites	Number of batch queues
UK	13	50
Italy	12	30
France	12	31
Greece	9	11
Spain	5	7
Germany	5	14
Portugal	4	7
Turkey	3	3
Poland	3	4
Netherlands	3	12
Croatia	3	6
Bulgaria	3	3
FYROM	2	2
Brazil	2	3
Vietnam	1	1
Slovakia	1	1
Russia	1	2
Other (.org)	1	1
Moldova	1	1
Mexico	1	1
Cyprus	1	1
China	1	1

Table 4: Distribution of sites and batch queues per country in the biomed VO (January 2013).

[2] http://www.grid-observatory.org
[3] http://www.egi.eu

executions of 36 different workflows executed by 26 users. Workflow executions contain 1,838 activity instances, corresponding to 92,309 invocations and 123,025 tasks (including resubmissions). Figure 5 shows the cumulative amount of running activities along this period. It shows that the workload is quite uniformly distributed although a slight increase is observed in June.

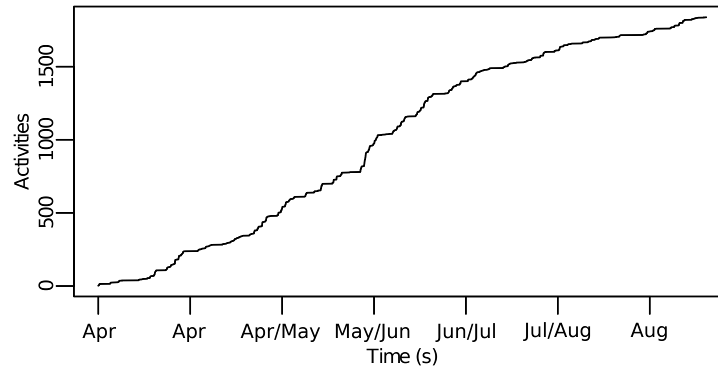


Figure 5: Cumulative amount of running activities from April to August 2011.

3.2.2 Incident Levels

We replayed the events found in this dataset to compute incident degree values after each event (total of 641,297 events). Figure 6 displays histograms of computed incident degrees. For readability purposes, only $\eta_i \neq 0$ values are represented. Most of the histograms appear multi-modal, which confirms that incident degrees are quantified. Level numbers and threshold values τ are set from visual mode detection in these histograms and reported on Table 5 with associated actions.

Incidents at level 1 are considered painless for the execution and they do not trigger any action. Other levels can lead to radical (completely stop the activity or blacklist a site) or intermediate actions (task or file replication).

The use of historical information to determine the threshold value put on η_b reduces the impact of the assumption that all tasks of a given workflow activity will have the same duration. Indeed, the threshold value quantifies what is an acceptable deviation of the task duration from its median value.

3.2.3 Association Rules

Association rules are computed based on the frequency of occurrences of two incident levels in the training dataset. The confidence $\rho_{i,j}^{u,v}$ of a rule $x_{u,v} \Rightarrow x_{i,j}$ measures the probability that an incident level $x_{i,j}$ happens when $x_{u,v}$ occurs. Table 6 shows rule samples extracted from the training dataset and ordered by decreasing confidence. The set of rules leading to activity blocked ($x_{1,2}$) and low efficiency ($x_{2,2}$) incidents shows that they are partially dependent on other "cause" incidents, which is considered by the self-healing process.

At the bottom of the table we find rules with null confidence. These are consistent with common-sense interpretation of the incident dependencies (e.g. no site-specific issue when input data is unavailable).

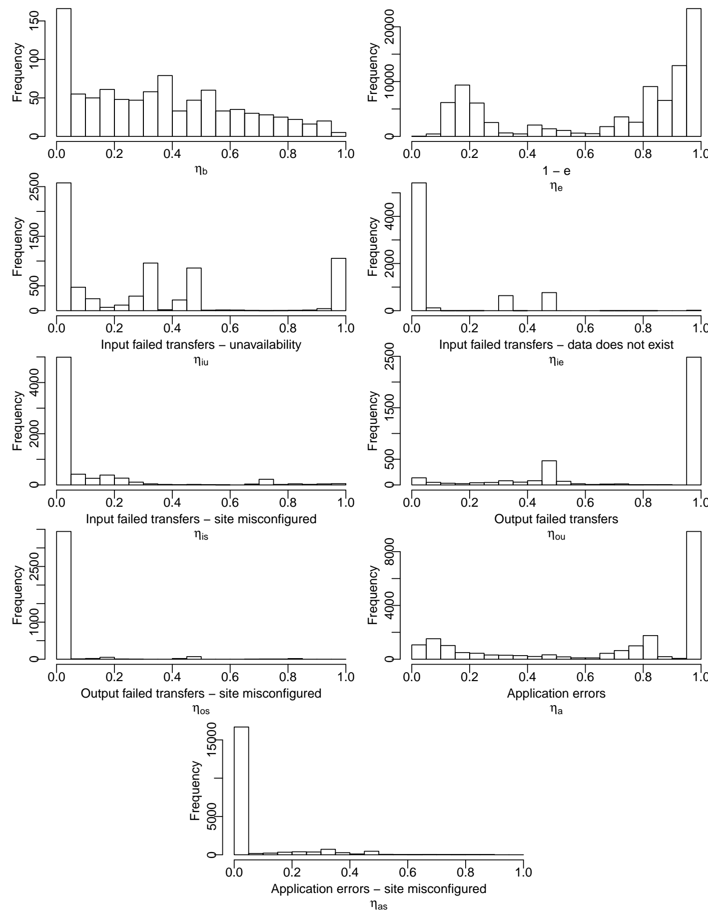


Figure 6: Histograms of incident degrees sampled in bins of 5%.

Incident (x_i)	Number of incident levels (m_i)	Level 1		Level 2		Level 3	
		$\tau_{i,1}$	actions	$\tau_{i,2}$	actions	$\tau_{i,3}$	actions
x_1: activity blocked	2	0	$\emptyset$	0.7	replicate tasks with $p_i > \tau$		
x_2: low efficiency	2	0	$\emptyset$	0.6	replicate input files replicate tasks with $p_i > \tau$		
x_3: input data unavailable	3	0	$\emptyset$	0.2	replicate input files	0.8	stop activity
x_4: input data does not exist	2	0	$\emptyset$	0.8	stop activity		
x_5: site misconfigured for input data	3	0	$\emptyset$	0.3	replicate files on sites reachable from problematic site	0.65	blacklist site
x_6: output data unavailable	2	0	$\emptyset$	0.8	stop activity		
x_7: site misconfigured for output data	2	0	$\emptyset$	0.1	blacklist site		
x_8: application error	2	0	$\emptyset$	0.5	stop activity		
x_9: site misconfigured for application	2	0	$\emptyset$	0.1	blacklist site		

Table 5: Incident levels and actions.

Association rule	$\rho_{i,j}^{u,v}$
$x_{5,2} \Rightarrow x_{2,2}$	0.3809
$x_{7,2} \Rightarrow x_{1,2}$	0.3529
$x_{5,3} \Rightarrow x_{1,2}$	0.3333
$x_{1,2} \Rightarrow x_{2,2}$	0.3059
$x_{3,2} \Rightarrow x_{1,2}$	0.2975
$x_{7,2} \Rightarrow x_{2,2}$	0.2941
$x_{5,2} \Rightarrow x_{1,2}$	0.2608
$x_{9,2} \Rightarrow x_{1,2}$	0.2435
$x_{2,2} \Rightarrow x_{1,2}$	0.2383
...	...
$x_{3,2} \Rightarrow x_{2,2}$	0.1276
$x_{7,2} \Rightarrow x_{3,3}$	0.1250
$x_{3,3} \Rightarrow x_{9,2}$	0.1228
$x_{7,2} \Rightarrow x_{3,2}$	0.0625
...	...
$x_{3,3} \Rightarrow x_{5,2}$	0.0000
$x_{3,3} \Rightarrow x_{5,3}$	0.0000
$x_{4,2} \Rightarrow x_{5,2}$	0.0000
$x_{4,2} \Rightarrow x_{5,3}$	0.0000
$x_{5,2} \Rightarrow x_{3,3}$	0.0000
$x_{5,2} \Rightarrow x_{4,2}$	0.0000
$x_{5,3} \Rightarrow x_{3,3}$	0.0000
$x_{5,3} \Rightarrow x_{4,2}$	0.0000

Table 6: Confidence of rules between incident levels.

3.3 Actions

Four actions are performed by the self-healing process: task replication, file replication, site blacklisting and activity stop. The first three are described below.

3.3.1 Task replication

Blocked activities and activities of low efficiency 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. $p(t_r, t_j) > \tau$, see equation 2) and j is in a more advanced phase than r, then replica r is aborted. Figure 7 presents the algorithm of the replication process. It is applied to all tasks with $p_i > \tau$, as defined on equation 2.

3.3.2 File Replication

File replication is implemented differently depending on the incident. In case of input data unavailability, a file is replicated to a storage resource selected randomly. The maximal allowed number of file replicas is set to 5. In case a site is misconfigured, replication to the site local storage resource is first attempted. This aims at circumventing inter-domain connectivity issues. If there is no local storage available or the replication process fails, then a second attempt is performed to a storage resource successfully accessed by other tasks executed on the same site. Otherwise, a storage resource is randomly selected. Fig. 8 depicts this process.

3.3.3 Site Blacklisting

Problematic sites are only temporarily blacklisted during a time interval set from exponential backoff. The site is first blacklisted for 1 minute only and then put back on the white list. In case it

```
Input: Set of replicas R of a task i

01. rep = true
02. for r ∈ R do
03.   for j ∈ R, j ≠ r do
04.     if p(t_r, t_j) > τ and j is a step further than r then
05.       abort r
06.   done
07.   if r is started and p(t_r, t̃) ≤ τ then
08.     rep = false
09.   else if r is queued then
10.     rep = false
11. done
12. if rep == true then
13.   replicate r
```

Figure 7: Replication process for one task.

```
Inputs: File f, set of storage resources S, set of completed tasks on the same site T

01. replicate f to local storage resource
02. if replication not successful then
03.   select storage s_i ∈ S where t ∈ T could access s
04.   replicate f to s_i
05.   if replication not successful then
06.     select randomly s_r ∈ S
07.     replicate f to s_r
08.   done
09. done
```

Figure 8: Site misconfigured: replication process for one file.

is detected misconfigured again, then the blacklist duration is increased to 2 minutes, then to 4 minutes, 16 minutes, etc.

3.4 Experiments

The healing process is implemented in the Virtual Imaging Platform (see description in section 3.2.1) and deployed in production. The experiments presented hereafter, conducted for two real workflow activities, evaluate the ability of the healing process to (i) improve workflow makespan by replicating tasks of blocked activities (*Experiment 1*) and (ii) quickly identify and report critical issues (*Experiment 2*). Another experiment, evaluating the handling of low efficiency, site misconfiguration, and input data unavailability, was reported in [9].

3.4.1 Experiment conditions and metrics

Experiment 1 aims at testing that blocked activities are properly detected and handled; the other incidents are ignored. This experiment uses a correct execution where all the input files exist and the application is supposed to run properly and produce the expected results. Five repetitions are performed for each workflow activity.

Experiment 2 aims at testing that unrecoverable errors are quickly identified and the execution is stopped. Unrecoverable errors are intentionally injected in 3 different runs: in run `non-existent inputs`, non-existent file paths are used for all the invocations; in `application-error`, all the file paths exist but input files are corrupted; and in `non-existent output`, input files are correct but

the application does not produce the expected results.

Two workflow activities are considered for each experiment. `FIELD-II/pasa` consists of 122 invocations of an ultrasonic simulator on an echocardiography 2D dataset. It is a data-intensive activity where invocations use from a few seconds to some 15 minutes of CPU time; it transfers 208 MB of input data and outputs about 40 KB of data. `Mean-Shift/hs3` has 250 CPU-intensive invocations of an image filtering application. Invocation CPU time ranges from a few minutes up to one hour; input data size is 182 MB and output is less than 1 KB. Files are replicated on two storage sites for both activities.

For each experiment, a workflow execution using our method (`Self-Healing`) is compared to a control execution (`No-Healing`). Executions are launched on the biomed VO of the EGI, in production conditions, i.e., without any control of the number of available resources and reliability. `Self-Healing` and `No-Healing` are both launched simultaneously to ensure similar grid conditions. The DIRAC scheduler is configured to equally distribute resources among executions.

The FuSM and healing process are implemented in the MOTEUR workflow engine. The timeout value in the healing process is computed dynamically as the median of the task inter-completion delays in the current execution. 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 only. MOTEUR is configured to resubmit failed tasks up to 5 times in all runs of both experiments. We use DIRAC v5r12p9 and MOTEUR 0.9.19.

The waste metric used by Cirne et al. [5] does not fit our context because it cannot provide an effective estimation of the amount of resource wasted by self-healing simulations when compared to the control ones. Here, resource waste is assessed by the amount of resource time consumed by the simulations performing the healing process related to the amount of resource time consumed by control simulations. We use the `waste coefficient` (w), defined as follows:

$$w = \frac{\sum_{i=1}^{n} h_i + \sum_{j=1}^{m} r_j}{\sum_{i=1}^{n} c_i} - 1$$

where h_i and c_i are the resource time consumed (CPU time + data transfers time) by n completed tasks for `Self-Healing` and `No-Healing` simulations 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 $w > 0$, the healing approach wastes resources compared to the control. If $w < 0$, then the healing approach consumes less resources than the control, which can happen when faster resources are selected.

3.4.2 Results and Discussion

Experiment 1 Figure 9 shows the makespan of `FIELD-II/pasa` and `Mean-Shift/hs3` for the 5 repetitions. The makespan is considerably reduced in all repetitions of both activities. Speed-up values yielded by `Self-Healing` range from 2.6 to 4 for `FIELD-II/pasa` and from 1.3 to 2.6 for `Mean-Shift/hs3`.

Figures 10 and 11 present a cumulative density function (CDF) of the number of completed tasks for `FIELD-II/pasa` and `Mean-Shift/hs3`, respectively. In most cases completion curves of both `Self-Healing` and `No-Healing` executions are similar up to 95%. This confirms that both executions are executed in similar grid conditions. In some cases (e.g. `Repetition 2` in Figure 11) `Self-Healing` execution even presents lower performance than `No-Healing` execution but it is compensated by the long-tail effect produced by the latter.

Tables 7 and 8 show the waste coefficient values for the 5 repetitions for `FIELD-II/pasa` and `Mean-Shift/hs3` respectively. The `Self-Healing` process reduces resource consumption up to 26% 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 172 for

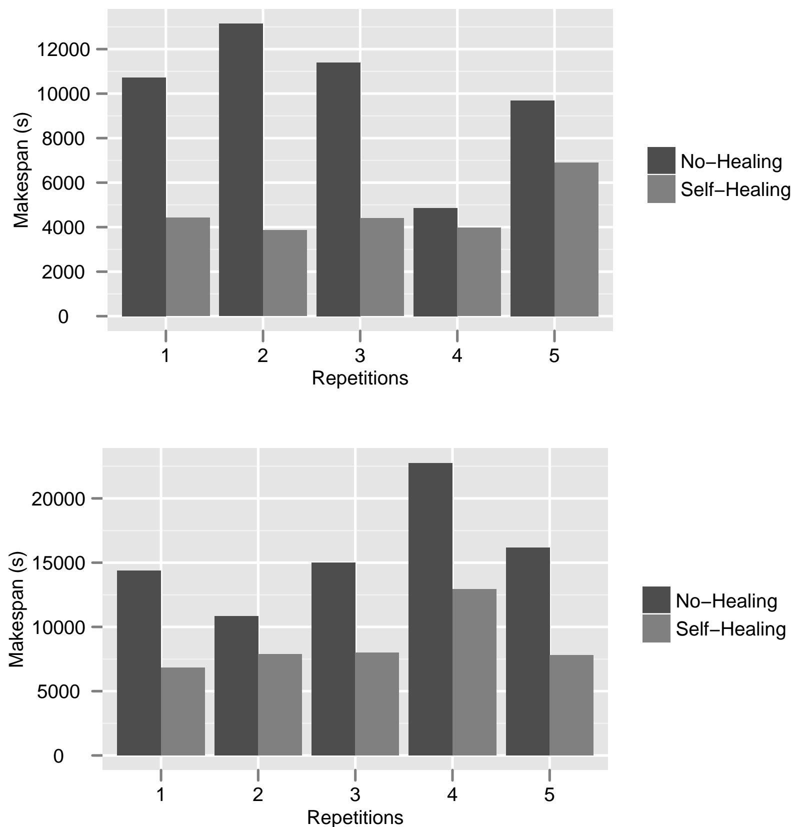


Figure 9: Execution makespan for `FIELD-II/pasa` (top) and `Mean-Shift/hs3` (bottom).

`FIELD-II/pasa` (i.e. 0.28 task replication per invocation in average) and 308 for `Mean-Shift/hs3` (i.e. 0.24 task replication per invocation in average).

Repetition	h	r	c	w
1	$56,159$s	$2,203$s	$64,163$s	-0.10
2	$60,991$s	$6,383$s	$79,031$s	-0.15
3	$60,473$s	$10,818$s	$77,851$s	-0.09
4	$42,475$s	$1,420$s	$41,528$s	0.05
5	$56,726$s	$4,527$s	$82,555$s	-0.26

Table 7: Waste coefficient values for `FIELD-II/pasa`.

Experiment 2 Figure 12 shows the makespan of `FIELD-II/pasa` and `Mean-Shift/hs3` for the 3 runs where unrecoverable errors are introduced. `No-Healing` was manually stopped after 7 hours to avoid flooding the infrastructure with faulty tasks. In all cases, `Self-Healing` is able to detect the issue and stop the execution far before `No-Healing`. It confirms that the healing process is able to identify unrecoverable errors and stop the execution accordingly. As shown on Table 9, the number of submitted fault tasks is significantly reduced, which has benefits both to the infrastructure and to the gateway itself.

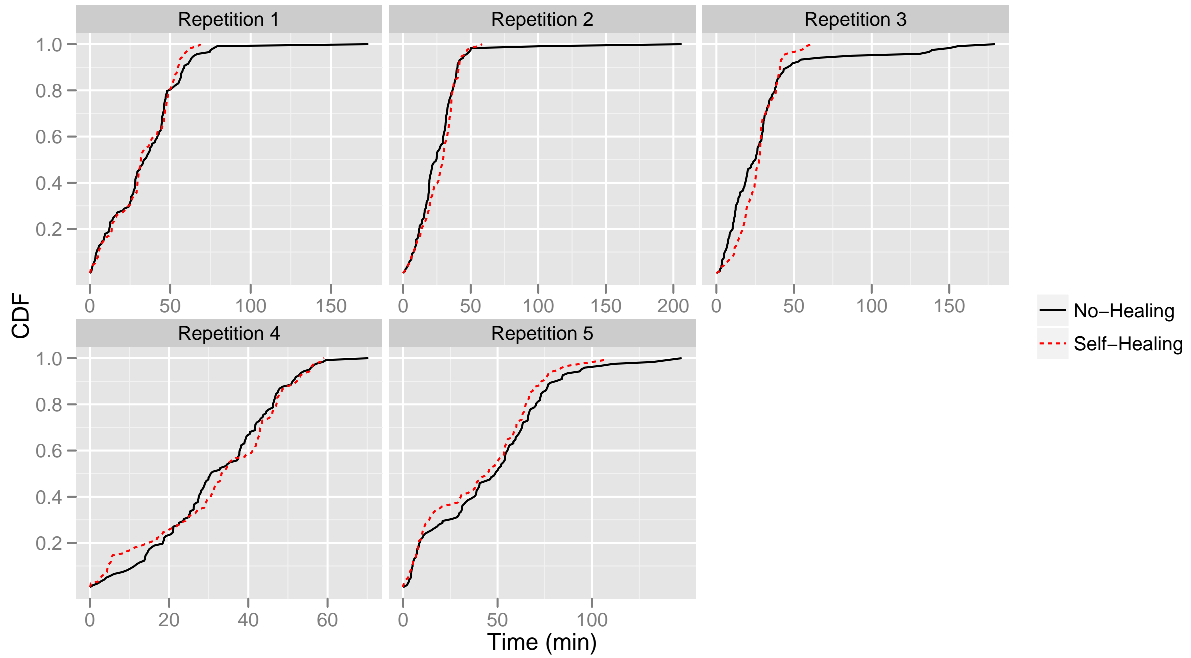


Figure 10: Experiment 1: CDF of the number of completed tasks for `FIELD-II/pasa` repetitions.

Repetition	h	r	c	w
1	$119,597$s	$5,778$s	$126,714$s	-0.02
2	$125,959$s	$4,792$s	$161,493$s	-0.20
3	$133,935$s	$14,352$s	$151,091$s	-0.02
4	$147,077$s	$2,898$s	$152,282$s	-0.02
5	$141,494$s	$17,514$s	$159,152$s	-0.01

Table 8: Waste coefficient values for `Mean-Shift/hs3`.

Run		Number of tasks	
		Self-Healing	No-Healing
application-error	FIELD-II/pasa	196	732
	Mean-Shift/hs3	249	1500
non-existent input	FIELD-II/pasa	293	732
	Mean-Shift/hs3	417	1500
non-existent output	FIELD-II/pasa	287	732
	Mean-Shift/hs3	364	1500

Table 9: Number of submitted faulty tasks.

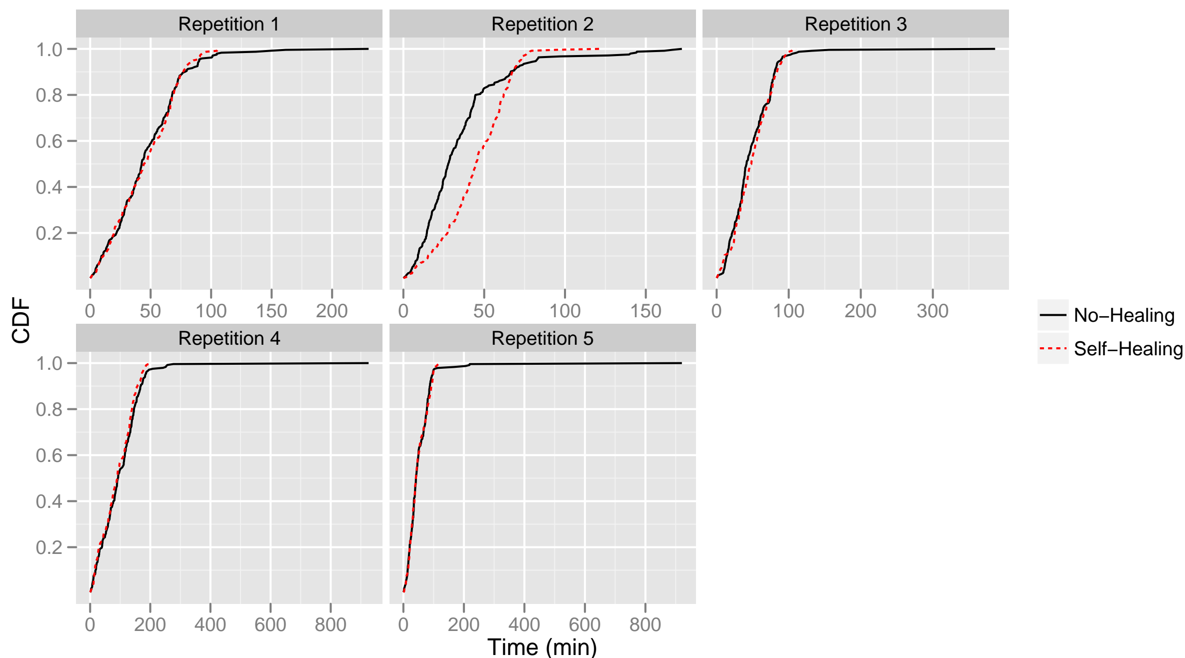


Figure 11: Experiment 1: CDF of the number of completed tasks for `Mean-Shift/hs3` repetitions.

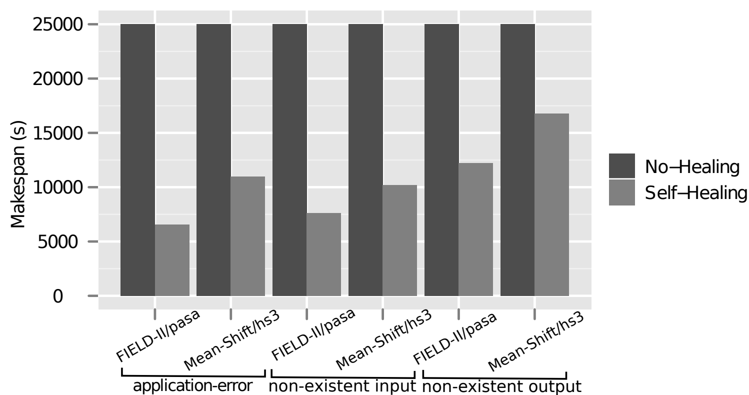


Figure 12: Experiment 2: makespan of `FIELD-II/pasa` and `Mean-Shift/hs3` for 3 different runs.

4 Application to granularity control

The low performance of *lightweight* (a.k.a. *fine-grained*) tasks is a common problem on widely distributed platforms where the communication overhead and queuing time are high, such as grid systems. To address this issue, fine-grained tasks are commonly grouped into *coarse-grained* tasks [22, 25, 21, 18, 2], which reduces the cost of data transfers when grouped tasks share input data [22] and saves queuing time when resources are limited [25]. However, task grouping also limits parallelism and therefore should be used sparingly.

We consider such a granularity problem in a platform executing workflows on a grid, for instance the SHIWA Simulation Platform. Workflows are compositions of *activities* that consist only of a program description. At runtime, activities receive data and spawn tasks for which the executable name and input data are known, but the computational cost and produced data volume are not. We propose an algorithm to optimize the granularity of workflow activities on non-clairvoyant online grid platforms. Our algorithm progressively discovers the characteristics of the running applications to compute a metric quantifying the fineness degree of a task group. This fineness metric includes measured task queuing times, and median-based estimations of task running times and transfer time of shared input data. Tasks are grouped when the fineness metric goes beyond a threshold learned from platform traces. In addition, a de-grouping mechanism is triggered when parallelism losses are detected, i.e. when the number of queued tasks is lower than the number of running tasks. The method is implemented in VIP, and evaluated with different applications, in production conditions, on the European Grid Infrastructure (EGI[4]).

4.1 Task Granularity Control Process

Algorithm 1 describes our task granularity control composed of two processes: (i) the fineness control process groups too fine task groups for which the fineness degree η_f is greater than threshold τ_f, and (ii) the coarseness control process de-groups too coarse task groups for which the coarseness degree η_c is greater than threshold τ_c. This section describes how η_f, η_c, τ_f and τ_c are computed, and details the grouping and de-grouping algorithms.

Algorithm 1 Main loop for granularity control

```
1: input: n 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 η_f
6:     if η_f > τ_f then
7:         group task groups using Algorithm 2
8:     end if
9:     determine coarseness degree η_c
10:    if η_c > τ_c then
11:        degroup coarsest task groups
12:    end if
13: end while
```

4.1.1 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 related to an activity are assumed independent, but with similar execution times (bag of tasks). This hypothesis is critical for our method. 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: $T_i \bigcap_{i \neq j} T_j = \emptyset$ and $\sum_{i=1}^{m} n_i = n$. The activity

[4] http://www.egi.eu

fineness degree η_f is the maximum of all group fineness degrees f_i:

$$\eta_f = \max_{i \in [1,m]} (f_i). \tag{3}$$

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:

$$f_i = d_i \cdot r_i, \tag{4}$$

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:

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

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: $\tilde{t} = \tilde{t}_{_setup} + \tilde{t}_{_input} + \tilde{t}_{_exec} + \tilde{t}_{_output}$. 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 durations.

In equation 4, r_i is the ratio between the max of the task current queuing times q_i in the group (measured for each task individually), and the total round-trip time (queuing+execution) of the group:

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

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 plus 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, inspecting the modes of the distribution 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.

We use traces collected from VIP [14] between January 2011 and April 2012, made available through the science-gateway workload archive [8]. The data set contains 680,988 tasks (including resubmissions and replications) linked to activities of 2,941 workflows executed by 112 users; task average waiting time is about 36 min. Applications deployed in VIP are described as workflows executed using the MOTEUR workflow engine [15]. Resource provisioning and task scheduling are provided by DIRAC [26] using so-called "pilot jobs". Resources are provisioned online with no advance reservations. Tasks are executed on the biomed virtual organization (VO) of the European Grid Infrastructure (EGI)[5] which has access to some 150 computing sites world-wide and to 120

[5]http://www.egi.eu

storage sites providing approximately 4 PB of storage. Fig. 13 (left) shows the distribution of sites per country supporting the biomed VO.

The fineness degree η_f was computed after each event found in the data set. Fig. 13 (right) 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.

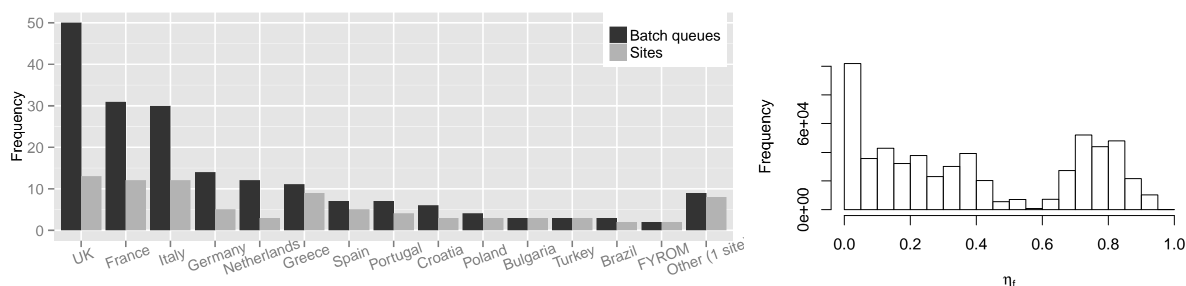


Figure 13: Distribution of sites and batch queues per country in the biomed VO (January 2013) (*left*) and histogram of fineness incident degree sampled in bins of 0.05 (*right*).

Task grouping. We assume that running tasks cannot be pre-empted, i.e. only waiting tasks can be grouped. Algorithm 2 describes our task grouping. Groups with $f_i > \tau_f$ 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 (Eq. 3 uses a max), 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 de-grouping process described in the next sub-section.

Algorithm 2 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 > τ_f and Q > R and j ≤ m do
6:         if f_j > τ_f then
7:             Group all tasks of T_j into T_i
8:             Recalculate f_i using Equation 4
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
```

4.1.2 Coarseness control

Condition $Q > R$ used in Algorithm 2 ensures that all resources will be exploited *if the number of available resources is stationary.* 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

Let's consider a workflow composed of one activity with 10 tasks initially split in 10 groups, and assume that task input data are shared among all tasks (i.e. $\tilde{t}_{_shared} = \tilde{t}_{_input}$).
Let $\tilde{t} = 10$ and $\tilde{t}_{_shared} = 7$ (in arbitrary time units) obtained from two completed task groups.
At time t, we assume $R = 2$ and $Q = 6$ with the following values for waiting task groups:

i	$\max_{j\in[1,n_i]} q_j$	d_i	r_i	f_i
5	50	0.70	0.83	0.58
6	48	0.70	0.82	0.58
7	45	0.70	0.81	0.57
8	43	0.70	0.81	0.57
9	41	0.70	0.80	0.56
10	40	0.70	0.80	0.56

Eq. 3 gives $\eta_f = 0.58$. As $\eta_f > \tau_f = 0.55$ and $Q > R$, the activity is considered too fine and task grouping is triggered. Groups with $f_i > \tau_f$ are grouped pairwise until $\eta_f \le \tau_f$ or $Q \le R$:

i	$\max_{j\in[1,n_i]} q_j$	d_i	r_i	f_i
11 [5,6]	50	0.53	0.79	0.42
12 [7,8]	45	0.53	0.77	0.41
13 [9,10]	41	0.53	0.76	0.40

Groups 5 and 6, 7 and 8, and 9 and 10 are grouped into groups 11, 12, and 13.

Let's consider that at time $t' > t$, group 11 starts running, thus $Q = 2 < R = 3$.
Eq. 5 gives $\eta_c = 0.6$. As $\eta_c > \tau_c = 0.5$, the activity is consider too coarse and task de-grouping is triggered. Then, group 13 is de-grouped to balance η_c.

Table 10: Example

resources increases, task groups may need to be de-grouped to maximize resource exploitation. This de-grouping is implemented by our coarseness control process.

The coarseness control process monitors the value of η_c defined as:

$$\eta_c = \frac{R}{Q+R}. \tag{5}$$

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 de-grouping increases the number of queued tasks, therefore tends to reduce η_c. Table 10 illustrates the method on a simple example.

4.2 Experiments and Results

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

4.2.1 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 de-group tasks according to Algorithm 1, where the timeout value is set to 2 minutes.

The target computing platform for these experiments is the biomed VO where the traces used to determine τ_f were acquired (see Section 4.1.1). To ensure resource limitation without flooding the production system, experiments are performed only on 3 sites of different countries. Tasks generated by MOTEUR are submitted to the biomed VO of EGI using the DIRAC scheduler.

Three workflow activities, implementing different kinds of medical image simulation, are used in the experiments. `SimuBloch` [3] is a very short activity made of 25 concurrent tasks; task CPU time

is of a few seconds; input data size is about 15 MB and output is less than 5 MB; $\tilde{t}_{_shared}$ is about 90% of the execution time. `FIELD-II` [17] consists of 122 data-intensive concurrent tasks ranging from a few seconds to 15 minutes of CPU time (tasks have the same cost, but their duration is resource-dependent); it transfers 208 MB of input data and outputs about 40 KB of data; $\tilde{t}_{_shared}$ ranges from 40% to 60% of the execution time. `PET-Sorteo/emission` [24] has 80 tasks of 2 CPU minutes; input data size is about 20 MB and output is about 50 MB; $\tilde{t}_{_shared}$ ranges from 50% to 80% of the execution time.

Two sets of experiments are conducted, under different load patterns. Experiment 1 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 de-grouping is required.

Experiment 2 evaluates the interest of using the de-grouping 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 experiment tasks are completed.

For both experiments, control and tested executions are launched simultaneously to ensure similar grid conditions. As no online task modification is possible in DIRAC, we implemented task grouping by canceling queued tasks and submitting grouped tasks as a new task. For each grouped task resubmitted in the `Fineness` or `Fineness-Coarseness` executions, a task in the `No-Granularity` is resubmitted too to ensure equal race conditions for resource allocation, and that each execution faces the same re-submission overhead. Five repetitions of each experiment are performed, along a time period of 4 weeks to cover different grid conditions. We use MOTEUR 0.9.21, configured to resubmit failed tasks up to 5 times, and with the task replication mechanism described in [13] activated. We use the DIRAC v6r6p2 instance provided by France-Grilles[6]. Results could not be compared to other grouping/de-grouping methods due to the lack of non-clairvoyant, online method available in the literature.

4.2.2 Results and Discussion

Experiment 1: Fig. 14 shows the makespan of `SimuBloch`, `FIELD-II`, and `PET-Sorteo/emission` executions. `Fineness` yields a significant makespan reduction for all repetitions. Table 11 shows the makespan (M) values and the 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 and 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.

Experiment 2: Fig. 15 shows the makespan (top) and evolution of task groups (bottom). Makespan values are reported in Table 12. In the first three repetitions, resources appear 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 of newly arrived resources, it has a lower speed up compared to `No-Granularity` due to parallelism loss. In the two last repetitions, the de-grouping process in `Fineness-Coarseness` allows to reach similar performance than

[6]https://dirac.france-grilles.fr

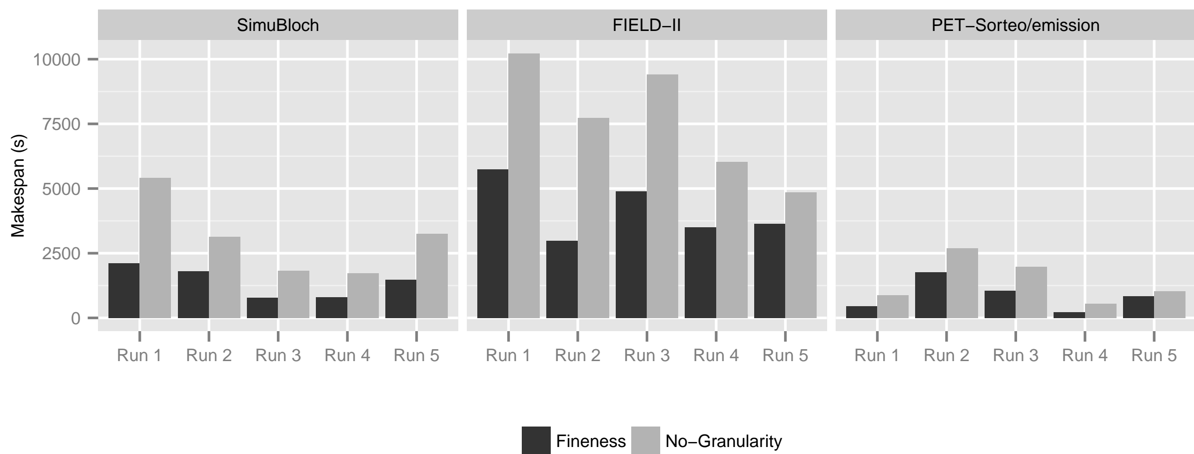


Figure 14: Experiment 1: makespan for `Fineness` and `No-Granularity` executions for the 3 workflow activities under stationary load.

`No-Granularity`, while `Fineness` is penalized by its lack of adaptation: a slowdown of 20% is observed compared to `No-Granularity`.

		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 11: Experiment 1: makespan (M) and number of task groups for `SimuBloch`, `FIELD-II` and `PET-Sorteo/emission` executions for the 5 repetitions.

Table 12 also shows the average queuing time values for Experiment 2. The linear correlation coefficient between the makespan and the average queuing time is 0.91, which indicates that the makespan evolution is indeed correlated to the evolution of the queuing time induced by the granularity control process.

Our task granularity control process works best under high resource contention, when the amount of available resources is stable or decreases over time (Experiment 1). Coarseness control can cope with soft increases in the number of available resources (Experiment 2), 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 de-grouping 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.

In addition, our method is dependent on the capability to extract enough accurate information from completed tasks to handle active tasks using median estimates. This may not be the case for activities which execute only a few tasks.

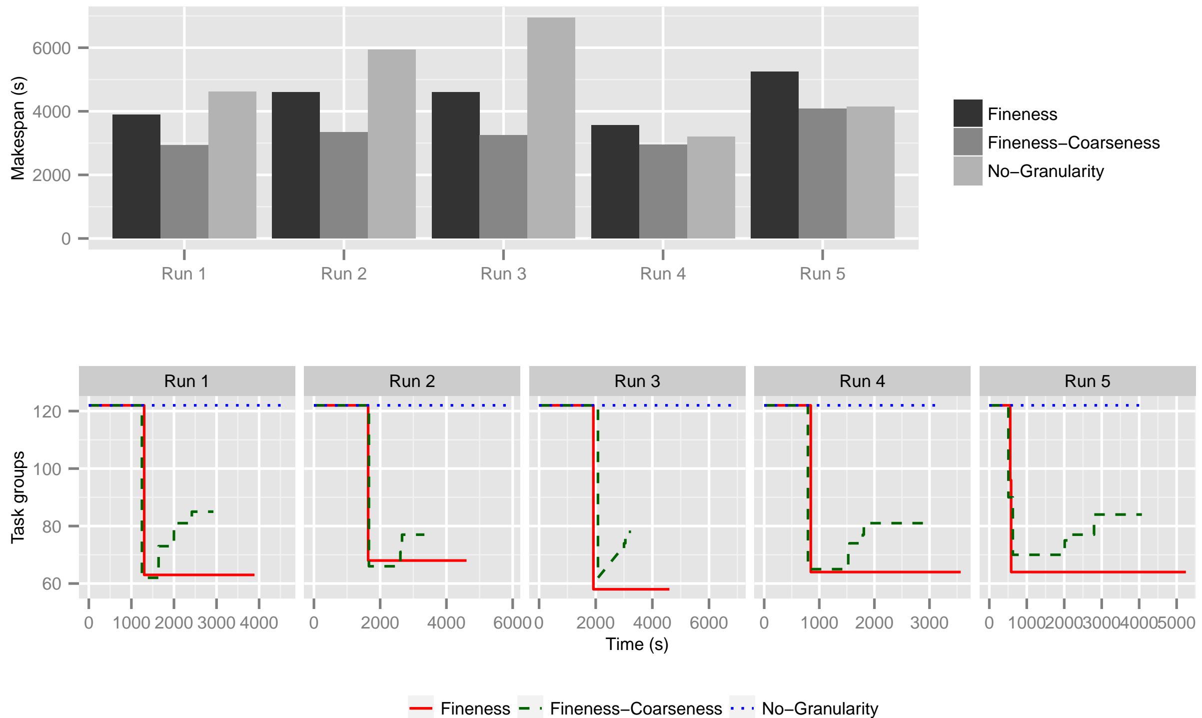


Figure 15: Experiment 2: makespan (top) and evolution of task groups (bottom) for `FIELD-II` executions under non-stationary load (resources arrive during the experiment).

4.3 Conclusion

We presented a method to address task granularity in distributed workflows in an online and non-clairvoyant environment. We defined a metric η_f for online determination of task fineness based on queue waiting time and estimated data transfer time of shared input data. For high η_f values, tasks are considered too fine and task grouping is triggered. Queued tasks are grouped pairwise as long as the number of queued tasks is greater than the number of running tasks and η_f is considered too fine. We also define a metric η_c for online determination of task coarseness based on the ratio of the number of queued tasks related to the number of running tasks. This metric aims at maximizing resource exploitation by de-grouping tasks groups when the number of available resources increases.

The task granularity control strategy was implemented in the MOTEUR workflow engine and deployed on EGI with the DIRAC resource manager. We tested it on three applications extracted from the Virtual Imaging Platform, a science gateway for medical simulation. Two experiments

	Run 1		Run 2		Run 3		Run 4		Run 5	
	M (s)	$\bar{q}$ (s)	M (s)	$\bar{q}$ (s)	M (s)	$\bar{q}$ (s)	M (s)	$\bar{q}$ (s)	M (s)	$\bar{q}$ (s)
No-Granularity	4617	2111	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 12: Experiment 2: makespan (M) and average queuing time ($\bar{q}$) for `FIELD-II` workflow execution for the 5 repetitions.

were conducted, to evaluate the fineness control process only under stationary load and the fineness and coarseness control process under non-stationary load. Results showed that under stationary load, our fineness control process significantly reduces the makespan of all applications. Under non-stationary load, task grouping is penalized by its lack of adaptation, but our de-grouping algorithm corrects it in case variations in the number of available resources are not too fast.

5 Application to fairness control

The problem of fairly allocating computing resources to application workflows rapidly arises on shared computing platforms such as grids or clouds. It must be addressed whenever the demand for resources is higher than the offer, that is, when some workflows are slowed down by concurrent executions. In some cases, unfairness makes the platform totally unusable, for instance when very short executions are launched concurrently with longer ones. We define fairness as in [23, 27, 4], i.e. as the variability in a set of workflows of the *slowdown* $\frac{M_{multi}}{M_{own}}$, where M_{multi} is the makespan when concurrent executions are present, and M_{own} is the makespan without concurrent executions.

In this section, we propose an algorithm to control fairness on non-clairvoyant online platforms. Based on a progressive discovery of applications' characteristics on the infrastructure, our method dynamically estimates the fraction of pending work for each workflow. Task priorities are then adjusted to harmonize this fraction among active workflows. This way, resources are allocated to application workflows relatively to their amount of work to compute. The method is implemented in VIP, and evaluated with different workflows, in production conditions, on the EGI. We use the slowdown as a *post-mortem* metric, to evaluate our method once execution times are known.

The next section details our fairness control process, and we then present experiments and results.

5.1 Fairness control process

Workflows are directed graphs of activities spawning sequential tasks for which the executable and input data are known, but the computational cost and produced data volume are not. Workflow graphs may include conditional and loop operators. Algorithm 3 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-prioritising tasks in workflows where the unfairness degree η_u is greater than a threshold τ_u. This section describes how η_u and τ_u are computed, and details the re-prioritization algorithm.

Algorithm 3 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 η_u
5: **if** $\eta_u > \tau_u$ **then**
6: re-prioritize tasks using Algorithm 4
7: **end if**
8: **end while**

5.1.1 Measuring unfairness: 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:

$$\eta_u = W_{\max} - W_{\min}, \tag{6}$$

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:

$$W_i = \max_{j \in [1, n_i]} (w_{i,j}) \tag{7}$$

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:

$$w_{i,j} = \frac{Q_{i,j}}{Q_{i,j} + R_{i,j}P_{i,j}} \cdot \hat{T}_{i,j}, \tag{8}$$

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 $\hat{T}_{i,j}$ is its relative observed duration. $\hat{T}_{i,j}$ 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:

$$\hat{T}_{i,j} = \frac{\tilde{t}_{i,j}}{\max_{v\in[1,m],w\in[1,n_i^*]}(\tilde{t}_{v,w})} \tag{9}$$

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, $\hat{T}_{i,j}$ is initialized to 1 and all medians are undefined; when two tasks of activity j complete, $\tilde{t}_{i,j}$ is updated and $\hat{T}_{i,j}$ is computed with equation 9. In this equation, the max operator is computed only on $n_i^* \leq n_i$ 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 must rely on observed task durations. Using the whole time distribution (or at least its few first moments) may be more accurate but it would complexify the method.

In Eq. 8, 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:

$$P_{i,j} = 2\left(1 - \max_{u\in[1,k_j]}\left\{\frac{t_u}{\tilde{t}_{i,j} + t_u}\right\}\right), \tag{10}$$

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 Eq. 10 only when at least 2 tasks of activity j are completed.

If all tasks perform as the median, i.e. $t_u = \tilde{t}_{i,j}$, then $\max_{u\in[1,k_j]}\left\{\frac{t_u}{\tilde{t}_{i,j}+t_u}\right\} = 0.5$ and $P_{i,j} = 1$. Conversely, if a task in the activity is much longer than the median, i.e. $t_u \gg \tilde{t}_{i,j}$, then $\max_{u\in[1,k_j]}\left\{\frac{t_u}{\tilde{t}_{i,j}+t_u}\right\} \approx 1$ 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 doesn't 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 (i) we have no model of the application, therefore task durations cannot be predicted from CPU, RAM or network characteristics, and (ii) network characteristics and even available RAM are shared among concurrent tasks running on the infrastructure, which makes them hardly measurable.

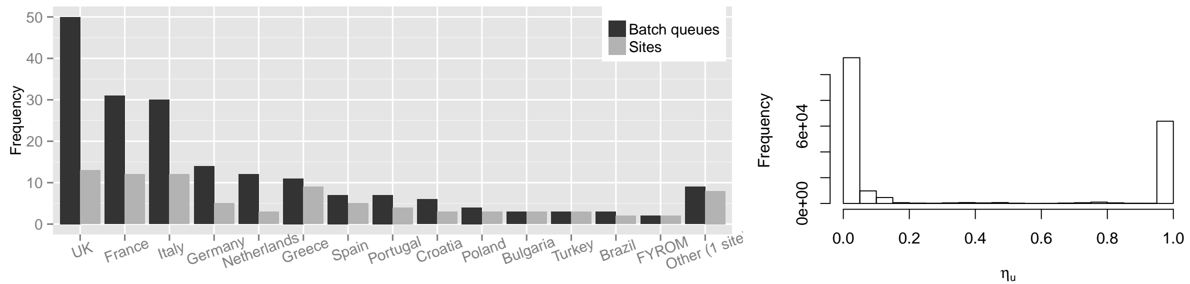


Figure 16: Distribution of sites and batch queues per country in the biomed VO (January 2013) (*left*) and histogram of the unfairness degree η_u sampled in bins of 0.05 (*right*).

5.1.2 Thresholding unfairness: τ_u.

Task prioritisation is triggered when the unfairness degree is considered critical, i.e $\eta_u > \tau_u$. Thresholding consists in clustering platform configurations in two groups: one for which unfairness is considered acceptable, and one for which task re-prioritization is needed. We determine τ_u from execution traces, for which different thresholding approaches can be used. For instance, we could consider that x% of the platform configurations are unfair while the rest are acceptable. The choice of x, however, would be arbitrary. Instead, we inspect the modes of the distribution of η_u to determine a threshold with a practical justification: values of η_u in the highest mode of the distribution, i.e. which are clearly separated from the others, will be considered unfair.

In this work, the distribution of η_u is measured from traces collected in VIP between January 2011 and April 2012 [8]. The data set contains 680,988 tasks (including resubmissions and replications) of 2,941 workflow executions executed by 112 users; task average queueing time is about 36 min. Applications deployed in VIP are described as GWENDIA workflows [20] executed using the MOTEUR workflow engine [15]. Resource provisioning and task scheduling are provided by DIRAC [26]. Resources are provisioned online with no advance reservations. Tasks are executed on the biomed virtual organization (VO) of the European Grid Infrastructure (EGI) which has access to some 150 computing sites world-wide and to 120 storage sites providing approximately 4 PB of storage. Fig. 16 (left) shows the distribution of sites per country supporting the biomed VO.

The unfairness degree η_u was computed after each event found in the data set. Fig. 16 (right) 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.

5.1.3 Task prioritization.

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. Task priority is an integer initialized to 1. $\Delta_{i,j}$ 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:

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

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:

$$\Delta_{i,j} = Q_{i,j} - \left\lfloor \frac{(\tau_u + W_{\min})(Q_{i,j} + R_{i,j}P_{i,j})}{\hat{T}_{i,j}} \right\rfloor, \tag{11}$$

Algorithm 4 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 > τ_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 > τ_u then
8:         Compute Δ_{i,j} from equation 11
9:         for p=1 to Δ_{i,j} do
10:          if ∃ waiting task q in activity j with priority ≤ 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
```

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

Algorithm 4 describes our task re-prioritization. *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 Eq. 6). Indeed, tasks are re-prioritized in *any* workflow i for which $W_i - W_{\min} > \tau_u$.

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. Table 13 illustrates the method on a simple example.

5.2 Experiments and results

Experiments are performed on a production grid platform to ensure realistic conditions. Evaluating fairness in production by measuring the slowdown is not straightforward because M_{own} (see definition in the introduction) cannot be directly measured. As described in Section 5.2.1, we estimate the slowdown from task durations, but this estimation may be challenged. Thus, Experiment 1 evaluates our method on a set of identical workflows, where the variability of the measured makespan can be used as a fairness metric. In Experiment 2, we add a very short workflow to this set of identical workflow, which was one of the configurations motivating this study. Finally, Experiment 3 considers the more general case of 4 different workflows with heterogeneous durations.

5.2.1 Experiment conditions

Fairness control was implemented as a MOTEUR plugin receiving notifications about task and workflow status changes. Each workflow plugin 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 3 and 4. As no online task modification is possible in DIRAC, we implemented task prioritization by canceling and resubmitting queued tasks to DIRAC with new priorities. This implementation decision adds an overhead to task executions. Therefore, the timeout value used in Algorithm 3 is set to 3 minutes.

The computing platform for these experiments is the biomed VO used to determine τ_u in Section 5.1.2. To ensure resource limitation without flooding the production system, experiments are performed only on 3 sites of different countries (France, Spain and Netherlands). Four real medical

Let's consider two identical workflows composed of one activity with 6 tasks,
and assume the following values at time t:

i	$Q_{i,1}$	$R_{i,1}$	$\tilde{t}_{i,1}$	$P_{i,1}$	$\hat{T}_{i,1}$	$W_i = w_{i,1}$
1	1	3	10	0.9	1.0	0.27
2	6	0	-	1.0	1.0	1.00

Values unknown at time t are noted '-'. Workflow 1 has 2 completed and 3 running tasks
with the following phase durations (in arbitrary time units):

u	t_u^{setup}	t_u^{input}	t_u^{exec}	t_u^{output}	t_u
1	2	2	4	1	9
2	1	2	3	2	8
3	2	3	5	-	-
4	2	2	-	-	-
5	1	-	-	-	-

We have $\tilde{t}_{1,1}^{setup} = 2$, $\tilde{t}_{1,1}^{input} = 2$, $\tilde{t}_{1,1}^{exec} = 4$ and $\tilde{t}_{1,1}^{output} = 2$. Therefore, $\tilde{t}_{1,1} = 10$.

The configuration is clearly unfair since workflow 2 has not started tasks.
Eq. 6 gives $\eta_u = 0.73$. As $\eta_u > \tau_u = 0.2$, the platform is considered unfair and task re-prioritization is triggered.

$\Delta_{2,1}$ tasks from workflow 2 should be prioritized. According to Eq. 11:
$\Delta_{2,1} = Q_{2,1} - \left\lfloor \frac{(\tau_u + W_1)(Q_{2,1}+R_{2,1}P_{2,1})}{\hat{T}_{2,1}} \right\rfloor = 6 - \left\lfloor \frac{(0.2+0.27)(6+0\cdot 1.0)}{1.0} \right\rfloor = 4$

At time $t' > t$:

i	$Q_{i,1}$	$R_{i,1}$	$\tilde{t}_{i,1}$	$P_{i,1}$	$\hat{T}_{i,1}$	$W_i = w_{i,1}$
1	1	3	10	0.8	1.0	0.29
2	2	4	-	1.0	1.0	0.33

Now, $\eta_u = 0.04 < \tau_u$. The platform is considered fair and no action is performed.

Table 13: Example

simulation workflows are considered: `GATE` [16], `SimuBloch`, `FIELD-II` [17], and `PET-Sorteo` [24]; their main characteristics are summarized on Table 14.

Three experiments are conducted. Experiment 1 tests whether unfairness among *identical workflows* is properly addressed. It consists of three `GATE` workflows sequentially submitted, as users usually do in the platform. Experiment 2 tests if the performance of *very short workflow executions* is improved by the fairness mechanism. Its workflow set has three `GATE` workflows launched sequentially, followed by a `SimuBloch` workflow. Experiment 3 tests whether unfairness among *different workflows* is detected and properly handled. Its workflow set consists of a `GATE`, a `FIELD-II`, a `PET-Sorteo` and a `SimuBloch` workflow launched sequentially.

For each experiment, a workflow set using our fairness mechanism (`Fairness – F`) is compared to a control workflow set (`No-Fairness – NF`). No method from the literature could be included in the comparison because, as mentioned in the introduction, they are either non-clairvoyant or offline. `Fairness` and `No-Fairness` are launched simultaneously to ensure similar grid conditions. For each task priority increase in the `Fairness` workflow set, a task in the `No-Fairness` workflow set task queue is also prioritized to ensure equal race conditions for resource allocation. Experiment results are not influenced by the re-submission process overhead since both `Fairness` and `No-Fairness` experience the same overhead. Four repetitions of each experiment are done, along a time period of four weeks to cover different grid conditions. Grid conditions vary among repetitions because

Workflow		#Tasks	CPU time	Input	Output
GATE	(CPU-intensive)	100	few minutes to one hour	∼115 MB	∼40 MB
SimuBloch	(data-intensive)	25	few seconds	∼15 MB	< 5 MB
FIELD-II	(data-intensive)	122	few seconds to 15 minutes	∼208 MB	∼40 KB
PET-Sorteo	(CPU-intensive)	1→80→1→80→1→1	∼10 minutes	∼20 MB	∼50 MB

Table 14: Workflow characteristics (→ indicate task dependencies).

computing, storage and network resources are shared with other users. We use MOTEUR 0.9.21, configured to resubmit failed tasks up to 5 times, and with the task replication mechanism described in [13] activated. We use the DIRAC v6r5p1 instance provided by France-Grilles[7], with a first-come, first-served policy imposed by submitting workflows with decreasing priority values.

Two different fairness metrics are used. The unfairness μ is the area under the curve η_u during the execution:

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

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 defined by:

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

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:

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

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, not performance. 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 the unfairness. In Experiment 1, the standard deviation of the makespan (σ_m) is also used.

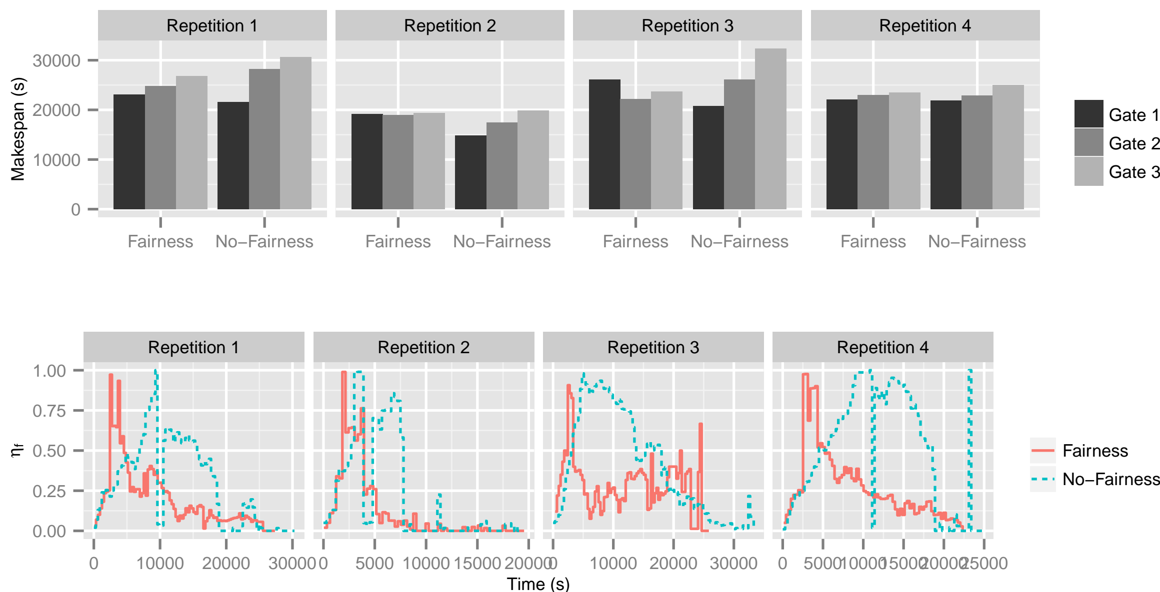
5.2.2 Results and discussion

Experiment 1 (*identical workflows*): Fig. 17 shows the makespan, unfairness degree η_u, makespan standard deviation σ_m, slowdown standard deviation σ_s and unfairness μ for the 4 repetitions. 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. Finally, slight increases of η_u are sporadically observed towards the end of the execution. This is due to task replications performed by MOTEUR: when new tasks are created, the fraction of pending work W increases, which has an effect on η_u. Quantitatively, the fairness mechanism reduces σ_m up to a factor of 15, σ_s up to a factor of 7, and μ by about 2.

Experiment 2 (*very short execution*): Fig. 18 shows the makespan, unfairness degree η_u, unfairness μ and slowdown standard deviation. In all cases, the makespan of the very short `SimuBloch` executions is significantly reduced for `Fairness`. The evolution of η_u is coherent with Experiment 1: 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 15 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

[7] https://dirac.france-grilles.fr



	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	0.07	2367	2007	0.84	7243	706	0.24	6070

Figure 17: Experiment 1 (identical workflows). Top: comparison of the makespans; middle: unfairness degree η_u; bottom: makespan standard deviation σ_m, slowdown standard deviation σ_s and unfairness μ.

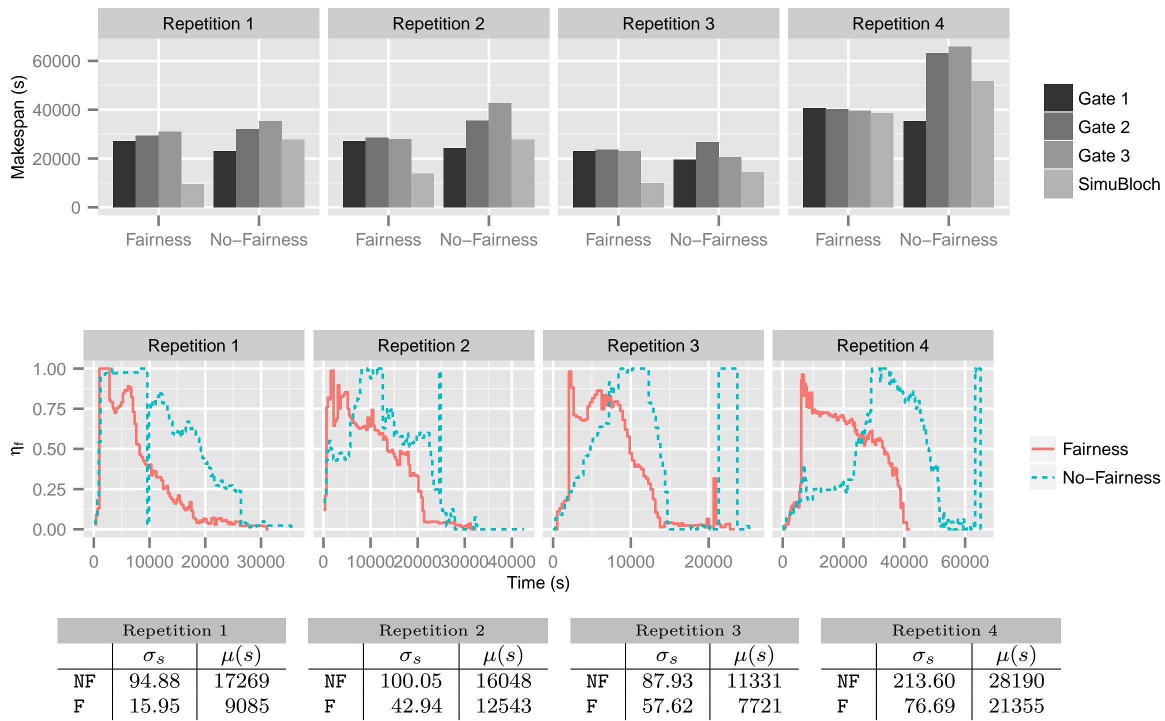
is not able to give higher priorities to `SimuBloch` tasks in advance. Despite that, the fairness mechanism speeds up `SimuBloch` executions up to a factor of 2.9, reduces task average wait time up to factor of 4.4 and reduces slowdown up to a factor of 5.9.

Experiment 3 (*different workflows*): Fig. 19 shows slowdown, unfairness degree, unfairness μ and slowdown standard deviation σ_s for the 4 repetitions. `Fairness` slows down `GATE` while it speeds up all other workflows. This is because `GATE` is the longest and the first to be submitted; in `No-Fairness`, it is favored by resource allocation to the detriment of other workflows. The evolution of η_u is similar to Experiments 1 and 2. σ_s is reduced up to a factor of 3.8 and unfairness up to a factor of 1.9.

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	182.68
	Fairness	9902	8145	122.25
4	No-Fairness	51664	47591	445.38
	Fairness	38630	27795	165.79

Table 15: Experiment 2: `SimuBloch`'s makespan, average wait time and slowdown.

In all 3 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 couldn't be totally removed. Currently, the method works best for applications with a lot of short tasks because the first few tasks can be



Repetition 1		
	σ_s	$\mu(s)$
NF	94.88	17269
F	15.95	9085

Repetition 2		
	σ_s	$\mu(s)$
NF	100.05	16048
F	42.94	12543

Repetition 3		
	σ_s	$\mu(s)$
NF	87.93	11331
F	57.62	7721

Repetition 4		
	σ_s	$\mu(s)$
NF	213.60	28190
F	76.69	21355

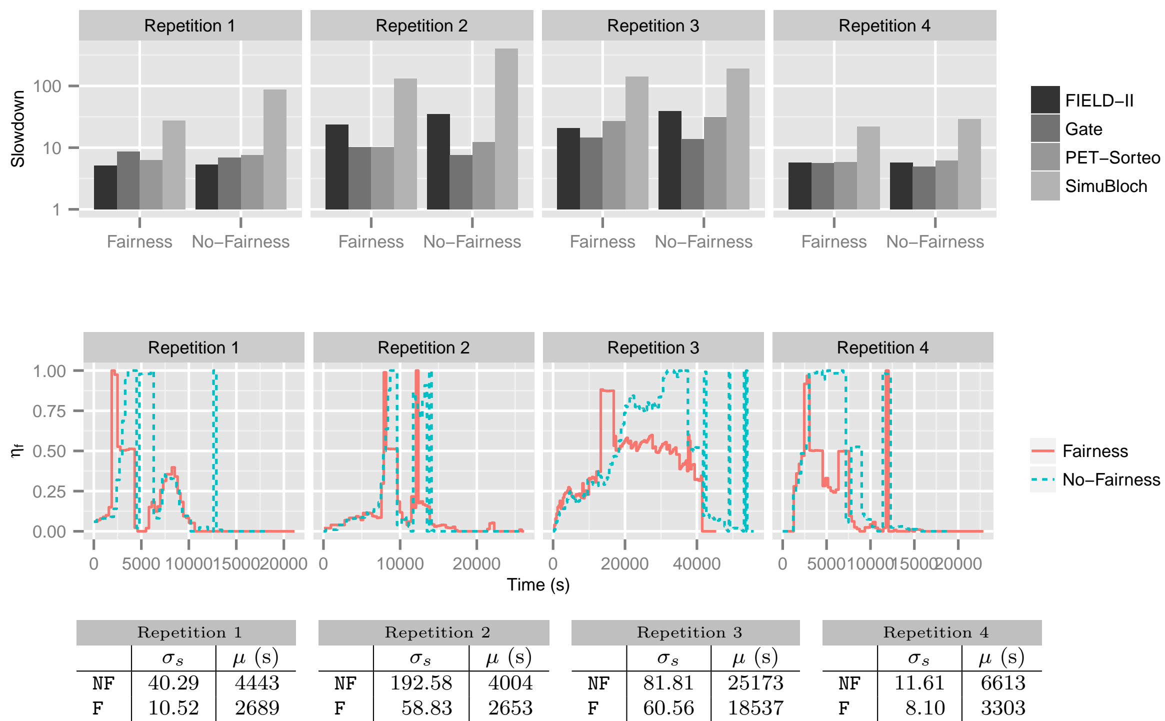
Figure 18: Experiment 2 (very short execution). Top: comparison of the makespans; middle: unfairness degree η_u; bottom: unfairness μ and slowdown standard deviation.

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.

5.3 Conclusion

We presented a method to address unfairness among workflow executions in an online and non-clairvoyant environment. We defined a novel metric η_u quantifying unfairness based on the fraction of pending work in a workflow. It compares workflow activities based on their ratio of queuing tasks, their relative durations, and the performance of resources where tasks are running. Performance is defined from the variability of task duration in the activity: good performance is assumed to lead to homogeneous task durations. To separate fair configurations from unfair ones, a threshold on η_u was determined from platform traces. Unfair configurations are handled by increasing the priority of pending tasks in the least performing workflows. This is done by estimating the number of running tasks that these workflows should have to bring η_u under the threshold value.

The method was implemented in the MOTEUR workflow engine and deployed on EGI with the DIRAC resource manager. We tested it on four applications extracted from VIP, a science gateway for medical simulation used in production. Three experiments were conducted, to evaluate the capability of the method to improve fairness (i) on identical workflows, (ii) on workflow sets containing a very short execution and (iii) on different workflows. In all cases, results showed that our method can very significantly reduce the standard deviation of the slowdown, and the average value of our metric η_u.



	Repetition 1		Repetition 2		Repetition 3		Repetition 4	
	σ_s	μ (s)	σ_s	μ (s)	σ_s	μ (s)	σ_s	μ (s)
NF	40.29	4443	192.58	4004	81.81	25173	11.61	6613
F	10.52	2689	58.83	2653	60.56	18537	8.10	3303

Figure 19: Experiment 3 (different workflows). Top: comparison of the slowdown; middle: unfairness degree η_u; bottom: unfairness μ and slowdown standard deviation.

6 General conclusion

To address reliability issues faced by SSP users, we proposed a simple, yet practical method for autonomous detection and handling of operational incidents in workflow activities. No strong assumption is made on the task duration or resource characteristics and incident degrees are measured with metrics that can be computed online, which is particularly relevant in the context of a general-purpose and multi-DCIs platform such as the SSP. We made the hypothesis that incident degrees were quantified into distinct levels, which we verified using real traces. Incident levels are associated offline to action sets ranging from light execution tuning (file/task replication) to radical site blacklisting or activity interruption. Action sets are selected based on the degree of their associated incident level and on confidence of association rules determined from execution history. We also demonstrated the usefulness of our method on workflow granularity and fairness control.

Our strategies were implemented in the MOTEUR workflow engine and deployed on the European Grid Infrastructure (EGI) with the DIRAC resource manager. We took special care to design strategies that could be implemented in practice in science-gateways deployed on EGI. We believe that the results presented in this deliverable could provide useful input to the handling of operational issues in the SHIWA Simulation Platform.

The results presented in this document were published in high-level peer-reviewed conferences and journals: [11], [10] and [12].

References

[1] Rakesh Agrawal, Tomasz Imieliski, and Arun Swami. Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, 22(2):207–216, June 1993.

[2] T.F. Ang and et al. A bandwidth-aware job grouping-based scheduling on grid environment. *Information Technology Journal*, 8:372–377, 2009.

[3] Fang Cao and et al. MRI estimation of T1 relaxation time using a constrained optimization algorithm. In *Multimodal Brain Image Analysis*, volume 7509 of *Lecture Notes in Computer Science*, pages 203–214. 2012.

[4] Henri Casanova, Frdric Desprez, and Frdric Suter. On cluster resource allocation for multiple parallel task graphs. *J. of Par. and Dist. Computing*, 70(12):1193 – 1203, 2010.

[5] W Cirne, F. Brasileiro, D. Paranhos, L.F.W. Goes, and W. Voorsluys. On the Efficacy, Efficiency and Emergent Behavior of Task Replication in Large Distributed Systems. *Parallel Computing*, 33:213–234, 2007.

[6] Kenneth Alan De Jong. Analysis of the behavior of a class of genetic adaptive systems. 1975.

[7] R. Ferreira da Silva, S. Camarasu-Pop, Baptiste Grenier, Vanessa Hamar, David Manset, Johan Montagnat, Jérôme Revillard, Javier Rojas Balderrama, Andrei Tsaregorodtsev, and T. Glatard. Multi-Infrastructure Workflow Execution for Medical Simulation in the Virtual Imaging Platform. In *HealthGrid 2011*, Bristol, UK, 2011.

[8] R. Ferreira da Silva and T. Glatard. A science-gateway workload archive to study pilot jobs, user activity, bag of tasks, task sub-steps, and workflow executions. In *CoreGRID/ERCIM Workshop on Grids, Clouds and P2P Computing*, Rhodes, GR, 2012.

[9] R. Ferreira da Silva, T. Glatard, and Frédéric Desprez. Self-healing of operational workflow incidents on distributed computing infrastructures. In *IEEE/ACM CCGrid 2012*, pages 318–325, Ottawa, Canada, 2012.

[10] R. Ferreira da Silva, T. Glatard, and Frédéric Desprez. On-line, non-clairvoyant optimization of workflow activity granularity on grids. In *Euro-Par*, pages 255–266, Aachen, Germany, 2013.

[11] R. Ferreira da Silva, T. Glatard, and Frédéric Desprez. Self-healing of workflow activity incidents on distributed computing infrastructures. *Future Generation Computer Systems*, 2013.

[12] R. Ferreira da Silva, T. Glatard, and Frédéric Desprez. Workflow fairness control on online and non-clairvoyant distributed computing platforms. In *Euro-Par*, pages 102–113, Aachen, Germany, 2013.

[13] R Ferreira da Silva, Tristan Glatard, and Frederic Desprez. Self-healing of operational workflow incidents on distributed computing infrastructures. *CCGrid'12*, pages 318–325, 2012.

[14] T. Glatard and et al. A virtual imaging platform for multi-modality medical image simulation. *IEEE Transactions on Medical Imaging*, 32:110–118, 2013.

[15] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and Efficient Workflow Deployment of Data-Intensive Applications on Grids with MOTEUR. *International Journal of High Performance Computing Applications (IJHPCA)*, 22(3):347–360, August 2008.

[16] S Jan, D Benoit, E Becheva, T Carlier, F Cassol, P Descourt, T. Frisson, L. Grevillot, L. Guigues, L. Maigne, C. Morel, Y Perrot, N Rehfeld, D. Sarrut, D R Schaart, S Stute, U. Pietrzyk, D. Visvikis, N. Zahra, and I. Buvat. Gate v6: a major enhancement of the gate simulation platform enabling modelling of ct and radiotherapy. *Physics in medicine and biology*, 56(4):881–901, 2011.

[17] J. Jensen and N. Svendsen. Calculation of pressure fields from arbitrarily shaped, apodized and excited ultrasound transducers. *IEEE T-UFFC*, 39(2):262–267, 1992.

[18] Ng Wai Keat and et al. Scheduling framework for bandwidth-aware job grouping-based scheduling in grid computing. *Malaysian Journal of Computer Science*, 19, 2006.

[19] D.S. Malik, John N. Mordeson, and M.K. Sen. On subsystems of a fuzzy finite state machine. *Fuzzy Sets and Systems*, 68(1):83–92, November 1994.

[20] Johan Montagnat and et al. A data-driven workflow language for grids based on array programming principles. In *WORKS'09*, , pages 1–10, Portland, USA, 2009. ACM.

[21] Nithiapidary Muthuvelu and et al. 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*, ACSW Frontiers '05, pages 41–48. Australian Computer Society, Inc., 2005.

[22] Nithiapidary Muthuvelu and et al. Task granularity policies for deploying bag-of-task applications on global grids. *FGCS*, 29(1):170 – 181, 2012.

[23] Tchimou N'Takpe and Frederic Suter. Concurrent scheduling of parallel task graphs on multi-clusters using constrained resource allocations. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, 2009.

[24] A. Reilhac and et al. PET-SORTEO: Validation and Development of Database of Simulated PET Volumes. *IEEE Trans. on Nuclear Science*, 52:1321–1328, 2005.

[25] Gurmeet Singh and et al. Workflow task clustering for best effort systems with pegasus. In *Mardi Gras'08*, pages 9:1–9:8, New York, NY, USA, 2008. ACM.

[26] A Tsaregorodtsev, N Brook, A Casajus Ramo, Ph Charpentier, J Closier, G Cowan, R Graciani Diaz, E Lanciotti, Z Mathe, R Nandakumar, S Paterson, V Romanovsky, R Santinelli, M Sapunov, A C Smith, M Seco Miguelez, and A Zhelezov. DIRAC3. The New Generation of the LHCb Grid Software. *Journal of Physics: Conference Series*, 219(6):062029, 2009.

[27] Henan Zhao and Rizos Sakellariou. Scheduling multiple dags onto heterogeneous systems. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 159–159, 2006.