A Dynamic Data-throttling Approach to Minimize Workflow Imbalance

RICARDO J. RODRÍGUEZ, Centro Universitario de la Defensa, General Military Academy, Spain
RAFAEL TOLOSANA-CALASANZ, Dept. of Comput. Sci. and Syst. Eng., University of Zaragoza, Spain
OMER F. RANA, School of Computer Science & Informatics, Cardiff University, UK

Scientific workflows enable scientists to undertake analysis on large datasets and perform complex scientific simulations. These workflows are often mapped onto distributed and parallel computational infrastructures to speed up their executions. Prior to its execution, a workflow structure may suffer transformations to accommodate the computing infrastructures, normally involving task clustering and partitioning. However, these transformations may cause workflow imbalance because of the difference between execution task times (runtime imbalance) or because of unconsidered data dependencies that lead to data locality issues (data imbalance). In this article, to mitigate these imbalances, we enhance the workflow lifecycle process in use by introducing a workflow imbalance phase that quantifies workflow imbalance after the transformations. Our technique is based on structural analysis of Petri nets, obtained by model transformation of a data-intensive workflow, and Linear Programming techniques. Our analysis can be used to assist workflow practitioners in finding more efficient ways of transforming and scheduling their workflows. Moreover, based on our analysis, we also propose a technique to mitigate workflow imbalance by data throttling. Our approach is based on autonomic computing principles that determine how data transmission must be throttled throughout workflow jobs. Our autonomic data-throttling approach mainly monitors the execution of the workflow and recompute data-throttling values when certain watchpoints are reached and time derivation is observed. We validate our approach by a formal proof and by simulations along with the Montage workflow. Our findings show that a dynamic data-throttling approach is feasible, does not introduce a significant overhead, and minimizes the usage of input buffers and network bandwidth.

CCS Concepts: • **Theory of computation → Network optimization**; **Linear programming**; • **Software and its engineering → Petri nets**;

Additional Key Words and Phrases: Scientific workflows, optimization, Petri nets, linear programming

The research of Ricardo J. Rodríguez was supported in part by the EU H2020 research and innovation programme under grant agreement no. 644869 (DICE), by the Spanish MINECO project CyCriSec (TIN2014-58457-R), and by the University, Industry and Innovation Department of the Aragonese Government under *Programa de Proyectos Estratégicos de Grupos de Investigación* (DisCo research group, ref. T21-17R). The research of Rafael Tolosana-Calasanz was supported in part by the University, Industry and Innovation Department of the Aragonese Government under *Programa de Proyectos Estratégicos de Grupos de Investigación* (COSMOS research group, ref. T35-17D).
Authors' addresses: R. J. Rodríguez, Centro Universitario de la Defensa, General Military Academy, Carr. de Huesca s/n, Zaragoza, 50090, Spain; email: rjrodriguez@unizar.es; R. Tolosana-Calasanz, Dept. of Comput. Sci. and Syst. Eng., University of Zaragoza, Calle María de Luna 1, Zaragoza, 50018, Spain; email: rafaelt@unizar.es; O. F. Rana, School of Computer Science & Informatics, Cardiff University, Queen's Buildings, 5 The Parade, Roath, Cardiff, CF24 3AA, UK; email: o.f.rana@cs.cardiff.ac.uk.

1 INTRODUCTION

Over the past few years, workflow technologies have assisted scientists in perfoming computational experiments and simulations on large datasets. A workflow can be seen as a high-level specification of a scientific experiment, often represented as a *Directed Acyclic Graph* (DAG), consisting of a set of tasks and their data- and control-flow dependencies. Workflow technologies typically map workflow specifications onto distributed and parallel computational infrastructures.

Prior to that mapping and scheduling process, however, a workflow is subject to a number of structural transformations to speed up the overall execution time, aiming at reducing the significant overheads that exist at the computing infrastructures. These transformations typically involve the aggregation of fine-grained tasks, creating clusters (jobs) of tasks eventually mapped onto computational resources. In the transformed graph, nodes represent jobs and edges represent dependencies. In such a computational model, jobs can executed in parallel when no dependencies exist. Otherwise, jobs with dependencies are blocked until their dependencies are solved: (i) in the case of control dependencies, when the parent jobs finish (for instance, in Taverna's SCULF workflow language (Sroka and Hidders 2009), one can specify that a task t_1 needs to be executed completely before another task t_2 starts by means of a control-dependency); and (ii) in the case of data dependencies, once all data inputs are transferred to the node location.

Nevertheless, many of the existing task clustering techniques generate *workflow imbalance* (Chen et al. 2015). In the transformed workflow, imbalance arises in jobs with various input data when these data arrive at different times: In other words, some paths of the graph are faster than others. As a result, data transfers are accomplished earlier than required, thus involving an unnecessary use of buffer space (storage), since these input data are buffered locally at the job node waiting for others to complete. Even when this buffer space is a shared, limited capacity resource, it remains blocked by the job until the remaining input datasets arrive. Hence, the greater the difference between arrival times of input data sets (in the same job), the greater the inefficiency. Furthermore, if the network links are shared, then the *transmit as-fast-as possible* policy also leads to an unnecessary usage of network bandwidth.

Workflow imbalance is mainly caused by *runtime* and *data* imbalance. Runtime workflow imbalance (Chen et al. 2015) can arise when a task clustering does not consider execution time differences between tasks and, thus, these tasks are aggregated forming jobs whose eventual execution generates imbalance. This type of imbalance can also arise at runtime due to unexpected failures or deviations from estimated execution time. However, data imbalance can appear as a result of a clustering that does not consider data dependencies, and, therefore, the resulting workflow structure contains data dependencies among jobs that may lead to data locality issues. Hence, data are poorly distributed, resulting into unsatisfactory data transfer times and increased data transfer-related failures.

One of the earliest studies in investigating the workflow imbalance problem was accomplished by Park and Humphrey (2008). Rather than transforming the workflow structure, they proposed to correct the impact of workflow imbalance on network resources using a data-throttling framework that enforces data transfers at different rates, correcting the imbalanced branches of DAGs. However, a workflow programmer/engineer needs to *manually* describe and specify the requirements on the delay of data transfers. However, the novel approach in Chen et al. (2015) proposed a clustering technique that acts on runtime and data imbalance. The technique merges tasks considering two criteria: (i) the runtime dispersion of tasks (essentially based on the standard deviation), which reduces the runtime dispersion of the resulting jobs by evenly distributing the task runtimes among jobs, and (ii) task dependencies (workflow structure), based on grouping together tasks with a significant number of dependencies. However, in such a work (Chen et al. 2015), workflow imbalance cannot be quantified, and as a result, different outputs obtained from alternative workflow transformations cannot be compared from an imbalance perspective.

In this article, we propose a formal model-based technique that (i) measures the degree of workflow imbalance and (ii) from it we compute data-throttling rates to correct it. We extend our previous work (Rodríguez et al. 2012; Rodríguez et al. 2012) by (i) enriching the formal details and proofs of our technique and by (ii) adding autonomic principles (Kephart and Chess 2003) to the computation of the data-throttling rates, so that it can adapt itself to unexpected performance variations at runtime. From a workflow methodological perspective, we propose to introduce an additional phase into the workflow lifecycle process, thereby after workflow transformation, workflow imbalance can be quantified and corrected when the effect may affect workflow performance significantly.

In particular, to measure workflow imbalance, we model the semantics of a workflow DAG with a subclass of Petri nets, namely, *Marked Graph* (MG). An MG is a Petri net in which every place has exactly one incoming arc and exactly one outgoing arc. We make use of the analytical power of Petri nets to *exactly* measure the degree of imbalance of DAG branches across all its nodes. In our model, each pair *input data–job* (represented by an input place–transition in the MG) can be seen as a queuing system, and then we assume that the whole workflow is executed continuously *ad infinitum*. Hence, in such conditions, Little's Law (Little 1961) ($L = \lambda \cdot W$) applies: The average number of data elements buffered (L) is equal to the product of the firing rate of a job (λ), and W is equal to the waiting time before a job starts execution. When multiple input data arrive to a node, the waiting time for the slowest one is equal to 0. While for the rest ones, a waiting time > 0 exists, which we can be measured into a metric called *slack*. Hence, a perfectly balanced workflow has slack values equal to zero for any input data, meaning that all input data arrive simultaneously when transmitted to a node (i.e., there is no waiting time for every job). We show that the computation of slack values is solvable in polynomial time by using linear programming techniques.

Our technique is therefore complementary to the existing *clustering/scheduling* techniques, since it provides a formal way of analyzing and quantifying imbalance: Any scheduling technique that transforms a workflow could make use of our slack computation technique to measure the imbalance of the resulting workflow transformation.

Finally, in this article we propose to balance branches by throttling data transfers when clustering techniques do not achieve a perfectly balanced workflow. As recognized by Park and Humphrey (2008), current practice of moving data as early as possible is either (i) unnecessary when viewed in isolation or (ii) harmful when viewed in the large, due to finite capacity and competing transfers on the same link and buffer. Our autonomic, slack-based data throttling strategy monitors execution time of jobs, feeds the Petri net model, and subsequently optimizes data transfers accordingly so that slack values tend to 0. Our strategy works as follows: After a computational job is completed, the estimated time (from the model) is compared with the elapsed time (from the real execution), and in case of deviation, our performance model is fed with this information, recomputing slacks and the optimal throttling values. Our approach is validated with synthetic Montage workflows through simulation.

This article is organized as follows. Section 2 reviews the related work. Section 3 provides background on Petri nets and workflows. Section 4 introduces our workflow modeling. Section 5 describes the theoretical foundations in which our approach relies. The autonomic data-throttling algorithm of our approach is presented in Section 6. The validation is shown in Section 7. Finally, Section 8 concludes the article and states future work.

2 RELATED WORK

Ordinary Petri nets and their extensions have been widely used for the specification, analysis and implementation of workflows (van der Aalst and van Hee 2004). In the scientific workflow community, they were also utilized. For instance, GWorkflowDL (Pellegrini et al. 2008; Vossberg et al. 2008), Grid-Flow (Guan et al. 2006), and FlowManager (Aversano et al. 2002) are representative examples, to name a few. Petri nets were also used for the specification of hierarchical scientific workflows, incorporating of exception handling and check-pointing mechanisms (Hoheisel 2006; Tolosana-Calasanz et al. 2010a, 2010b). In this article, we use Petri nets for deriving performance models of pure graph-based workflows.

An overview of the most significant systems was carried out in Yu and Buyya (2005), where existing automated data transfer strategies utilized among tasks in workflows were classified (namely centralized, mediated, and peer-to-peer). A centralized approach utilizes a central point for data transmission. This solution is not scalable and occurs in systems where the data transfer times are much smaller than computation times. Taverna (Oinn et al. 2006) typically utilizes a centralized data transfer, due to the characteristics of the problems it tackles. In a mediated strategy, the locations of the intermediate data are managed by a distributed data management system. Finally, a peer-to-peer approach transfers data directly between processing nodes. The direct transmissions of peer-to-peer approaches reduce both transmission time and the bottleneck problem caused by centralized and mediated approaches. Thus, they are suitable for large-scale intermediate data transfer. The technique proposed in our article is focused on data-intensive workflows using a peer-to-peer–based data movement strategy.

Pegasus workflow system (Deelman et al. 2007) also dealt with data-intensive workflows and incorporated both mediated and peer-to-peer transfers. In the mediated approach, Pegasus utilizes a data replica catalog that stores the intermediate generated data, so that data can be subsequently retrieved rather than recomputed again. Workflow performance speedup was also a matter of study in Pegasus, and workflow specifications go through a process of clustering (grouping of small tasks) and partitioning that helps the meta-scheduler to optimize the execution time. A similar approach was followed in Duan et al. (2006) with further optimization at runtime by adapting to the dynamically changing state of underlying resources. However, none of these approaches made an effective usage of both network bandwidth and buffer/storage of files.

An analysis of the overhead for scientific workflows in Grid environments was given in Nerieri et al. (2006). The analysis included both load imbalance and data movement, which were identified as main sources for overheard. In Park and Humphrey (2008), the problem of load imbalance and data throttling for scientific workflows was also analyzed. The authors proposed a process envelope-based framework for throttling data transfers. Nonetheless, they do not provide any analysis method to automatically obtain those data-throttling values. In this article, we describe a method that can automatically derive (sub-optimal) values for data throttling.

Performance analysis of scientific workflows was also studied in Duan et al. (2009) and Tolosana-Calasanz et al. (2008). The performance method in Duan et al. (2009) was based on a hybrid Bayesian-neural network to predict the execution time of workflow tasks. Bayesian network was used as a high-level representation of the probability distribution of activity performance affected by different factors. The important attributes were dynamically selected by the Bayesian network and fed into a radial basis function neural network to make further predictions. The performance analysis approach used in this article is similar to Tolosana-Calasanz et al. (2008), where a method that provided a parameterized Petri net-based graphical model of the overall workflow structure was proposed.

Finally, a Petri net-based structural analysis for business workflows was proposed in Aalst et al. (2002). The authors utilized a specific class of Petri nets, WF-nets, tailored toward workflow analysis. WF-nets are suitable to model workflows with different kind of control operations, such as sequence, choice, synchronization, fork, or merge. The structural analysis that can be undertaken on these nets includes correctness, deadlock analysis, and liveness.

3 BACKGROUND ON PETRI NETS

This section details basic definitions needed to understand the rest of this article. In particular, we introduce concepts regarding Petri nets (PN). We assume the reader is familiar with the fundamentals of Petri nets, such as its structure and firing rules. An introduction to PN is given in Murata (1989).

Definition 3.1. A *Petri net (Murata 1989)* is a 4-tuple $\mathcal{N} = \langle P, T, \mathbf{Pre}, \mathbf{Post}\rangle$, where:

- P and T are disjoint non-empty sets of *places* and *transitions* ($|P| = n$, $|T| = m$) and
- **Pre** (**Post**) are the pre- (post-)incidence non-negative integer matrices of size $|P| \times |T|$.

The *pre-* and *post-set* of a node $v \in P \cup T$ are respectively defined as $^\bullet v = \{u \in P \cup T | (u, v) \in F\}$ and $v^\bullet = \{u \in P \cup T | (v, u) \in F\}$, where $F \subseteq (P \times T) \cup (T \times P)$ is the set of directed arcs. A Petri net is said to be *self-loop free* if $\forall p \in P, t \in T$ $t \in {}^\bullet p$ implies $t \notin p^\bullet$. *Ordinary* nets are Petri nets whose arcs have weight 1. The *incidence matrix* of a Petri net is defined as $\mathbf{C} = \mathbf{Post} - \mathbf{Pre}$.

A distribution of tokens over the places is called a *marking*$\mathbf{m} \in \mathbb{Z}_{\geq 0}^{|P|}$, and it represents the state of the Petri net. An *initial marking* of a Petri net P is denoted as $\mathbf{m}_0$. A *marking* of a place $p \in P$, denoted as $\mathbf{m}(p) \in \mathbb{Z}_{\geq 0}$, is the number of tokens of a place p. Formally:

Definition 3.2. A *Petri net system*, or a *marked Petri net*$\mathcal{S} = \langle \mathcal{N}, \mathbf{m_0}\rangle$, is a Petri net $\mathcal{N}$ with an *initial marking*$\mathbf{m_0}$.

A transition $t \in T$ is *enabled* at marking $\mathbf{m}$ if $\mathbf{m} \geq \mathbf{Pre}(\cdot, t)$, where $\mathbf{Pre}(\cdot, t)$ is the column of **Pre** corresponding to transition t. A transition t enabled at $\mathbf{m}$ can *fire* yielding a new marking $\mathbf{m}' = \mathbf{m} + \mathbf{C}(\cdot, t)$ (*reached* marking). This is denoted by $\mathbf{m}\xrightarrow{t}\mathbf{m}'$. A sequence of transitions $\sigma = \{t_i\}_{i=1}^n$ is a *firing sequence* in $\mathcal{S}$ if there exists a sequence of markings such that $\mathbf{m_0}\xrightarrow{t_1}\mathbf{m}_1\xrightarrow{t_2}\mathbf{m}_2 \ldots \xrightarrow{t_n}\mathbf{m}_n$. In this case, marking $\mathbf{m}_n$ is said to be *reachable* from $\mathbf{m_0}$ by firing σ, and it is denoted by $\mathbf{m}_0\xrightarrow{\sigma}\mathbf{m}_n$. The *firing count vector* $\sigma \in \mathbb{Z}_{\geq 0}^{|T|}$ of the fireable sequence σ is a vector such that $\sigma(t)$ represents the number of occurrences of $t \in T$ in σ. If $\mathbf{m_0}\xrightarrow{\sigma}\mathbf{m}$, then we can write $\mathbf{m} = \mathbf{m_0} + \mathbf{C}\sigma$ in vector form, which is referred to as the *linear* (or *fundamental*) *state equation* of the net.

The set of markings reachable from $\mathbf{m_0}$ in $\mathcal{N}$ is denoted as $RS(\mathcal{N}, \mathbf{m_0})$ and it is called the *reachability set.* A Petri net system $\langle \mathcal{N}, \mathbf{m_0}\rangle$ is *reversible* if for each marking $\mathbf{m} \in RS(\mathcal{N}, \mathbf{m_0})$, $\mathbf{m_0}$ is reachable from $\mathbf{m}$.

A transition t is *live* if it can be fired from every reachable marking. A system is *live* when every transition is live. A net is *structurally live* if there exists an initial marking making it live. A system is *bounded* if and only if its reachability set is finite. A net is structurally bounded if and only if it is bounded, regardless of the initial marking.

Ordinary nets are Petri nets whose arcs have weight 1, i.e., $\forall p \in P, t \in T, \mathbf{Pre}(p, t), \mathbf{Post}(p, t) \in [0, 1]$. Marked graphs (MGs) are a subclass of ordinary Petri nets that are characterized by the fact that each place has exactly one input and exactly one output arc. More formally:

Definition 3.3 (Murata 1989). A *marked graph* (MG) is an ordinary Petri net such that $\forall p \in P, |{}^{\bullet}p| = |p^{\bullet}| = 1$.

A *P-semiflow* (*T-semiflow*) is a non-negative integer vector $\mathbf{y} \in \mathbb{Z}_{>0}^{|P|}$ ($\mathbf{x} \in \mathbb{Z}_{>0}^{|T|}$) such that it is a left (right) anuller of the net's incidence matrix, i.e., $\mathbf{y}^{\intercal}\mathbf{C} = \mathbf{0}$ ($\mathbf{C}\mathbf{x} = \mathbf{0}$). A P-semiflow implies a token conservation law independent of any firing of transitions. A P- (or T-)semiflow $\mathbf{v}$ is minimal when its support, $\|\mathbf{v}\| = \{i, \mathbf{v}(i) \neq 0\}$, is not a proper superset of the support of any other P- (or T-)semiflow, and the greatest common divisor of its elements is one.

Petri nets are used for system performance modeling and evaluation by considering the inclusion of time. There exist two main approaches: to introduce the notion of time in places or to introduce it in transitions. In this article, we assume that transitions represent actions that have associated a duration. Therefore, we associate a duration to the firing delay of transitions (Ramchandani 1974). Furthermore, we consider that the firing delay of transitions follow an exponential distribution function.

A Petri net model in which each transition has an exponential rate is called a *Stochastic Petri net* (SPN) (Ajmone Marsan et al. 1995a; Florin and Natkin 1985). These rates are considered to be marking-independent, i.e., its values remain constant. In this article, we make use of SPNs, whose underlying Petri net model is a Marked Graph. More formally:

Definition 3.4 (Florin and Natkin 1985). A *Stochastic Marked Graph* (SMG) system is a pair $\langle \mathcal{N}, \mathbf{m_0}, \delta \rangle$ where $\mathcal{N}$ is a Marked Graph, and $\delta \in \mathbb{R}_{\geq 0}^{|T|}$ is a positive real function, such that $\delta(t)$ is the mean of the exponential firing time distribution associated to a transition $t \in T$.

Let us also note that every SMG net is structurally live and structurally bounded. Furthermore, if an SMG net is live for an acceptable initial marking, then the system is reversible. As stated in Ajmone Marsan et al. (1995b), the underlying Continuous Time Markov Chain associated to the SMG is ergodic. Therefore, the steady-state distribution serves as a basis for the quantitative evaluation of an SMG in terms of performance indices, particularly the expected value of the number of tokens in a given place (*average marking*) is denoted as $\overline{\mathbf{m}}$, and the mean number of firings of a transition per unit time (throughput) is denoted as χ.

4 MODEL TRANSFORMATION: FROM DATA-INTENSIVE WORKFLOWS TO PETRI NETS

In this section, we first introduce the formal definition of a data-intensive workflow and its lifecycle. The reader can refer to van der Aalst and van Hee (2004) and Taylor et al. (2007) as an introduction to data-intensive workflows. Then, we describe the transformation rules to obtain a Petri net model from a given workflow specification, as described by the previous formal definition.

4.1 Data-intensive Workflow Definition and Lifecycle

A data-intensive workflow is typically modeled as a DAG, where nodes represent workflow tasks (actions) and edges represent task dependencies (dataflow). Each task is defined by a program and a set of input parameters.

To speed up the overall workflow execution time (also known as makespan), data-intensive workflows are mapped onto distributed and parallel computational infrastructures. In general terms, a workflow's lifecycle undergoes a number of stages. First, at the *workflow mapping stage*, a workflow specification is often transformed from a high-level specification into a structure that is more suitable for the target computing infrastructure: When workflow tasks are relatively short running (from a few seconds to a few minutes), workflow tasks can be merged forming job clusters to minimize the usage overheads of these resources. In the transformed DAG, nodes

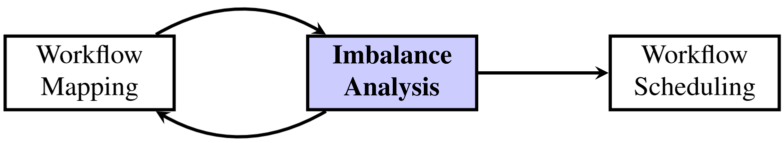


Fig. 1. Lifecycle of a data-intensive workflow, including workflow imbalance analysis (highlighted).

represent jobs, whereas edges typically represent dependencies that involve data transfers; that is, output files produced by one task are input files of other task. Second, at the *workflow scheduling stage*, the workflow engine can schedule workflow jobs onto computational resources. It should be noted that only jobs that have all their input dependencies solved can be executed in parallel. This model of computation is followed by several workflow systems, as Pegasus (Deelman et al. 2007), Askalon (Duan et al. 2005), or Taverna (Oinn et al. 2006), among others.

This general workflow lifecycle can be enhanced with a new stage, proposed in this article, as depicted in Figure 1. After the workflow mapping stage, a *workflow imbalance analysis stage* can be added, where the transformed workflow specification is analysed and its imbalance quantified. At that point, in case of having obtained an imbalanced transformation, the process could go back to the mapping stage to improve the output, i.e., by trying an alternative mapping technique. In case the workflow imbalance analysis is satisfactory, the workflow imbalance analysis stage can end. Moreover, if required, marginal workflow imbalance can be corrected, for instance, by data throttling (as it will be also shown in the rest of this article).

It should be noted that the input specification of our technique is a transformed DAG, whose size can be measured in terms of both the number of nodes and arcs that it contains, which can be considerably smaller than in the original workflow specification. We formally define a transformed data-intensive workflow as follows.

Definition 4.1. A transformed data-intensive workflow is a DAG represented by a 4-tuple $\mathcal{W} = \langle \mathcal{J}, \mathcal{D}, \xi, \psi \rangle$, where:

- $\mathcal{J}, |\mathcal{J}| \geq 1$ is a nonempty set of execution jobs;
- $\mathcal{D} \subseteq \mathcal{J} \times \mathcal{J}$ is a subset of the Cartesian product $\mathcal{J} \times \mathcal{J}$ that indicates the execution jobs $j_i, j_k \in \mathcal{J}$, that are dependent, i.e., it indicates whether a job j_k needs data generated by the job j_i to execute.
- $\xi : \mathcal{J} \rightarrow \mathbb{R}_+$ is a function that assigns for each $j \in \mathcal{J}$ a value $c \in \mathbb{R}_+$ that indicates the involved overhead induced by the parallel or distributed infrastructure in which the job is being executed plus the computational time required to execute the job.
- $\psi : \mathcal{D} \rightarrow \mathbb{Z}_{\geq 0}$ is a function that indicates, for each $(j_i, j_k) \in \mathcal{D}$, the size $s \in \mathbb{Z}_{\geq 0}$ of transmitted data (in bytes) between the job j_i and the job j_k.

Note that the set of execution jobs $\mathcal{J}$ can be divided into three nonempty disjoint sets $\mathcal{J}_I, \mathcal{J}_O$, and $\mathcal{J}_{IO}$, i.e., $\mathcal{J} = \mathcal{J}_I \bigcup \mathcal{J}_O \bigcup \mathcal{J}_{IO}, \mathcal{J}_I \bigcap \mathcal{J}_O = \emptyset, \mathcal{J}_I \bigcap \mathcal{J}_{IO} = \emptyset$, and $\mathcal{J}_O \bigcap \mathcal{J}_{IO} = \emptyset$, where $\mathcal{J}_I$ is the set of jobs that only have input data dependencies, i.e., $\forall j \in \mathcal{J}_I, j' \neq j \in \mathcal{J}, (j, j') \notin \mathcal{D}$; $\mathcal{J}_O$ is the set of jobs that only have output data dependencies, i.e., $\forall j \in \mathcal{J}_O, j' \neq j \in \mathcal{J}, (j', j) \notin \mathcal{D}$; and $\mathcal{J}_{IO}$ is the set of jobs that have input and output data dependencies, i.e., $\forall j \in \mathcal{J}_{IO}, j' \neq j, j'' \neq j' \in \mathcal{J}, (j, j'), (j'', j) \in \mathcal{D}$.

4.2 Workflow Modeling Using Petri Nets

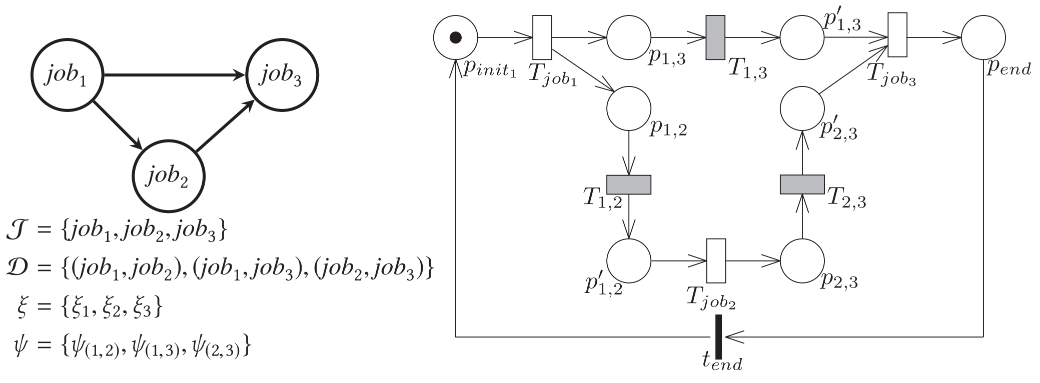
Our imbalance analysis approach is based on Petri net theory. Thus, first, we need to derive a Petri net model from a DAG-based workflow specification, before any imbalance analysis is performed. In this section, we introduce the transformation rules to transform a data-intensive workflow

$\mathcal{W} = \langle \mathcal{J}, \mathcal{D}, \xi, \psi \rangle$ into an SMG system $\langle \mathcal{N}, \mathbf{m_0}, \delta \rangle$, where $\mathcal{N}$ is indeed an MG. The set of places P of the obtained PN is divided into two nonempty disjoint sets P_D and P_A, i.e., $P = P_D \bigcup P_A$, where P_D contains the places that hold tokens representing data being transmitted between workflow jobs and P_A contains auxiliary places added in the model during the transformation process. Similarly, the set of transitions is divided into three nonempty disjoint sets, i.e., $T = T_J \bigcup T_D \bigcup \{t_{aux}\}$, where T_J contains the transitions representing workflow jobs, T_D contains the transitions representing a transmission between jobs in the workflow, and transition t_{end} is a special transition added to make the obtained net model cyclic, which is a requirement for analytical purposes. In particular, the model transformation steps are as follows.

(1) First, each job $j \in \mathcal{J}$ is transformed into a transition $t_j \in T_J$. The transition t_j represents the execution of job j. The delay of transition t_j is set to $\xi(j)$, i.e., $\delta(t_j) = \xi(j)$.
(2) Then, for each $d = (j_i, j_k) \in \mathcal{D}$, we create two auxiliary places $p_d, p'_d \in P_D \subset P$ and a transition $t_d \in T_D$. Such a transition t_d is connected to both p_d, p'_d (as an input place and an output place, respectively), that is, ${}^\bullet t_d = p_d$ and $t_d^\bullet = p'_d$. Last, place p_d is connected to t_{j_i} as an output place, i.e., $t_{j_i}^\bullet = p_d$, and place p'_d is connected to t_{j_k} as an input place, i.e., $t_{j_k}^\bullet = p'_d$. Note that places p_d, p'_d, represent the output buffer and input buffer of jobs j_i and j_k, respectively. The initial markings of both places are set to zero. The transition t_d represents the time needed to transmit data from j_i to j_k. Hence, $\delta(t_d)$ is set to a value that indicates how long it takes to transfer data of size $\psi(j_i, j_k)$. For the sake of simplicity, we assume that the transmission time of $\psi(j_i, j_k)$ depends only on the bandwidth BW of the data link between j_i and j_k and a latency ε, and we consider a *transmit-as-fast-as-possible* data transfer policy, which is common practice in scientific workflows. Thus, $\delta(t_d) = \frac{\psi(j_i, j_k)}{BW} + \varepsilon$.
(3) As the last step, we *close the net*. This step is needed to have a model that can be analysed with our theoretical framework. In this regard, we create an auxiliary place $p_j \in P_A$ for every $j \in \mathcal{J}_I$ and we connect it, as an output place, to each $t_j \in T_J$, i.e., ${}^\bullet p_j = t_j$. Then, we create a transition t_{aux} and connect it, as an output transition, to each of the those places p_j, i.e., $p_j^\bullet = t$. We set the delay of t to zero, i.e., $\delta(t) = 0$ (it is said to be an *immediate transition*, since its firing consumes no time). Finally, we create an auxiliary place $p_{j'} \in P_A$ for every $j' \in \mathcal{J}_O$ to connect the transition t to each $t_{j'} \in T_J$, i.e., ${}^\bullet p_{j'} = t, p_{j'}^\bullet = t_{j'}$. We set the initial marking of those places $p_{j'}$ to one, i.e., $\mathbf{m_0}(p_{j'}) = 1$.

Note that the transformation steps do not create any place $p \in P$ with more than an output transition, i.e., $\forall p \in P, |p^\bullet| = |{}^\bullet p| = 1$. Hence, the Petri net obtained after transformation is a Marked Graph. Furthermore, since we consider transitions are timed, the obtained model is an SMG (see Definition 3.4).

Let us illustrate the model transformation by means of an example. Figure 2(a) shows a workflow $\mathcal{W}$ composed of three computational jobs (tasks) $\mathcal{J} = \{job_1, job_2, job_3\}$ and their data-link dependencies $\mathcal{D} = \{(job_1, job_2), (job_1, job_3), (job_2, job_3)\}$, while Figure 2(b) shows the Petri net obtained after transformation. Let us perform the first transformation step. For each job $j_i \in \mathcal{J}, 1 \leq i \leq 3$, we create a transition T_{job_i} and set its delay $\delta(T_{job_i}) = \xi_i$. Then, let us perform the second step. Now, we iterate in each $(job_i, job_j) \in \mathcal{D}$ creating places $p_{i,j}, p'_{i,j}$, and a transition $T_{i,j}$ (we highlighted these transitions in grey color to distinguish them). We set the delay of these transitions to $\delta(T_{i,j}) = \psi_{(i,j)}$, assuming a bandwidth of 1 and a latency of 0. We connect $p_{i,j}$ and $p'_{i,j}$ to $T_{i,j}$ as input and output places, respectively (i.e., $p_{i,j}^\bullet = {}^\bullet p'_{i,j} = T_{i,j}$). Finally, the last step closes the net. In this case, $job_1 \in \mathcal{J}_O$ and $job_3 \in \mathcal{J}_I$. Thus, we fi rst create a place p_{end} and connect it as the output place of T_{job_3}. Then, we create a transition t_{end} and we connect it as the output transition of p_{end}.



$$\mathcal{J} = \{job_1, job_2, job_3\}$$
$$\mathcal{D} = \{(job_1, job_2), (job_1, job_3), (job_2, job_3)\}$$
$$\xi = \{\xi_1, \xi_2, \xi_3\}$$
$$\psi = \{\psi_{(1,2)}, \psi_{(1,3)}, \psi_{(2,3)}\}$$

(a) (b)

Fig. 2. (a) A workflow DAG specification and (b) its transformation to a PN.

We end the transformation process by creating a place p_{init_1} that it connects as an input place to T_{job_1} and as an output place to t_{end}. Furthermore, the initial marking of p_{init_1} is set to 1.

5 WORKFLOW IMBALANCE ANALYSIS

As discussed previously, workflow imbalance typically appears when a workflow specification is transformed prior to its execution to improve its performance. In this section, we propose the application of formal models to analyze and quantify workflow imbalance. Our approach relies on the concept of *slack*. We consider slack as a non-negative real number associated with each input link of a synchronization point. Roughly speaking, slack values measure how probable is the imbalance degree for each input in such a synchronization point. That is, the higher the slack, the greater the probability of having imbalance in such an input. These slack values can be computed directly from the SMG model, obtained after transformation of a given workflow DAG. Let us first explain how these values are computed.

The rationale is based on Little's Law, given that SMG can be seen as a queuing system (Campos and Silva 1992). Let us consider a workflow task w_t that requires an input i to execute. Let the place p_i and the transition $t = p^\bullet$ be the pair of elements in the SMG that represent i and w_t, respectively. Then, the pair (p, t) can be seen as a simple queueing system to which, if the limits of average marking and steady-state throughput exist, Little's formula (Little 1961) can be directly applied (Campos and Silva 1992) as

$$\overline{\mathbf{m}}(p) = \mathbf{Pre}(p,t)\chi(t)\mathbf{r}(p), \tag{1}$$

where $\mathbf{Pre}(p,t)\chi(t)$ is the output rate of tokens from place p (which is indeed equal to the input rate in steady state) and $\mathbf{r}(p)$ is the average residence time at place p, i.e., the average time spent by a token in place p.

In fact, $\mathbf{r}(p)$ is the sum of the average waiting time due to a possible synchronization in transition t plus the average service time of t, denoted by $\delta(t)$. Therefore, $\delta(t)$ becomes a lower bound for the average residence time:

$$\overline{\mathbf{m}}(p) = \mathbf{Pre}(p,t)\chi(t)\mathbf{r}(p) \geq \mathbf{Pre}(p,t)\chi(t)\delta(t). \tag{2}$$

Since we connect the output transitions to the input places in the Petri net obtained as a transformation of a workflow, it becomes a strongly connected SMG. Therefore, it has a single minimal t-semiflow that is equal to 1, which implies that *the steady-state throughput is the same for every transition*. Therefore, a single scalar variable Θ suffices to express the throughput bound to be computed for all transitions. This steady-state throughput Θ for every transition of a SMG can be computed by solving the following Linear Programming Problem (LPP) (Chiola et al. 1993):

$$\begin{aligned} \max\ \Theta \text{ subject to} & \\ & \overline{\mathbf{m}}(p) \geq \delta(p^\bullet)\Theta, \forall p \in P \\ & \overline{\mathbf{m}} = \mathbf{m_0} + \mathbf{C}\ \sigma \\ & \overline{\mathbf{m}},\ \sigma \geq \mathbf{0}. \end{aligned} \tag{3}$$

Let us remark that the dual problem of LPP 3 enables us to compute a lower bound for the average interfiring time of transitions Γ, which is equal to the inverse of Θ, i.e., $\Gamma = \frac{1}{\Theta}$. Besides, this dual problem of LPP 3 (as stated in Campos and Silva (1992)) also provides the bottleneck of the net, that is, the slowest P-semiflow in the net. In fact, the computed value of Γ is the cycle time of such a P-semiflow.

Note that *for the case of an SMG obtained after the transformations of a workflow mapping stage, this value of* Γ *matches with the makespan of the workflow*: Since we close the obtained net, every place in the net will be contained in some P-semiflow. Furthermore, by the PN model transformation, Step 3 ensures that every minimal P-semiflow of the net will contain two places $p, p \neq p' \in P_A : p \in {}^\bullet t_{aux}, p' \in t^\bullet_{aux}$. Recall that every P-semiflow determines a cycle in the net. Hence, the slowest P-semiflow of the net model will represent the slowest path from the beginning of the workflow to the end.

The inequality in Equation (3) becomes an equality for those places that belong to the critical cycle (Rodríguez and Júlvez 2010). Note that this inequality can be rewritten as

$$\overline{\mathbf{m}}(p) \geq \delta(p^\bullet)\Theta \rightarrow \overline{\mathbf{m}}(p) = \delta(p^\bullet)\Theta + \mu(p), \tag{4}$$

where $\mu(p)$ *is the slack of place* p. As commented on previously, for every place p in the critical cycle, it necessarily holds that $\mu(p) = 0$.

This degree of freedom of slacks enables us to compute a reachable marking, named *tight marking* (Carmona et al. 2009) and denoted as $\tilde{\mathbf{m}} \in \mathbb{R}^{|P|}$, where each transition with various input places has at least one of them with null slack, i.e., $\forall t \in T, |{}^\bullet t| > 1 \rightarrow \exists p \in {}^\bullet t, \tilde{\mathbf{m}}(p) = \Theta\delta(t)$, and $\forall p' \in {}^\bullet t \setminus \{p\}, \tilde{\mathbf{m}}(p') = \Theta\delta(t) + \mu(p'), \mu(p') \geq 0$. Therefore, the tight marking can be computed by solving the following LPP (Carmona et al. 2009):

$$\begin{aligned} \max \sum \sigma \text{ subject to} & \\ & \tilde{\mathbf{m}}(p) \geq \delta(p^\bullet)\Theta \quad \text{for every } p \in P \\ & \tilde{\mathbf{m}} = \mathbf{m_0} + \mathbf{C}\ \sigma \\ & \sigma(t_p) = k, \end{aligned} \tag{5}$$

where t_p is a transition that belongs to the critical path, $\mathbf{C}$ is the Petri net incidence matrix, σ is the vector of firing counts of transitions, $\delta(p^\bullet)$ is the average service time of output transition $p^\bullet$, and $k \in \mathbb{R}$ is a constant number.

The tight marking $\tilde{\mathbf{m}}$ as a result of Equation (5) provides both the tight places of the net, i.e., the set of places $P' \subset P$ that fulfill $\tilde{\mathbf{m}}(p) = \delta(p^\bullet)\Theta, \forall p \in P'$, and the set of places $P^\mu = P \setminus P', P^\mu \bigcap P' = \emptyset, P^\mu \bigcup P' = P$, that have some slack, i.e., $\tilde{\mathbf{m}}(p) = \delta(p^\bullet)\Theta + \mu(p), \mu(p) > 0, \forall p \in P^\mu$.

Since Θ and $\delta(p^\bullet)$ are known, the slack of every place $p \in P^\mu$ can be straightforwardly computed as

$$\mu(p) = \tilde{\mathbf{m}}(p) - \delta(p^\bullet)\Theta. \tag{6}$$

Once the slack values for every place are known, for each slack, we are able to compute a *delay* value α that minimizes it as much as possible (Rodríguez et al. 2012). For every place p with non-zero slack, the maximum value of α is equal to $\frac{\mu(p)}{\Theta}$.

Theorem 5.1. *Let $\mathcal{N}_{\mathcal{W}}$ be the Petri net obtained after transformation of a workflow $\mathcal{W} = \langle \mathcal{J}, \mathcal{D}, \xi, \psi \rangle$. Let $p \in P$ be a place such that $\mu(p) > 0$ and let $p^\bullet = t \in T$ be the transition that represents a data transmission $d \in \mathcal{D}$ that motivates such a slack in p. Let $\delta(t)$ be the transmission time consumed by d and let Θ be the steady-state throughput of $\mathcal{N}$. Then, the addition of a delay $\alpha \leq \frac{\mu(p)}{\Theta}$ to $\delta(t)$ minimizes the slack in place p as much as possible.*

Proof. From Equation (5), the tight marking of p is computed as $\tilde{\mathbf{m}}(p) = \delta(p^\bullet)\Theta + \mu(p)$. Recall that $p \in P^\mu$, i.e., $\mu(p) > 0$. Assume that we increment in $\alpha > 0$ units of time the average service time of t, that is, $\delta'(t) = \delta(t) + \alpha$. Now, if we recompute the tight marking, then we shall obtain $\tilde{\mathbf{m}}'(p) = \delta'(p^\bullet)\Theta + \mu'(p), \mu(p) > \mu'(p) \geq 0$. Consider that Θ remains constant; otherwise, we changed the steady-state throughput of the net. The new vector σ' is also equal to the previous σ, since the change in the delay of the transition does not affect to the count of transition firings. Hence, $\tilde{\mathbf{m}}'(p) = \tilde{\mathbf{m}}(p)$. Let $\beta \in \mathbb{R}_{>0}$ be the value in which $\mu(p)$ is reduced, i.e., $\mu'(p) = \mu(p) - \beta$. Hence, $\tilde{\mathbf{m}}'(p) = \tilde{\mathbf{m}}(p) \Rightarrow (\delta(p^\bullet) + \alpha)\Theta + \mu(p) - \beta = \delta(p^\bullet)\Theta + \mu(p) \Rightarrow \alpha = \frac{\beta}{\Theta}$. Since $\mu'(p) \geq 0, 0 < \beta \leq \mu(p)$. Therefore, $\alpha \leq \frac{\mu(p)}{\Theta}$. □

6 Toward an Autonomic Data Throttling to Minimize Workflow Imbalance

In this section, we first recall the automated data-throttling algorithm previously introduced in Rodríguez et al. (2012). Then, we show the problems that may arise when this algorithm was used, for instance, upon the occurrence of unexpected failures. To solve those problems, we finally propose an improved algorithm that uses autonomic principles to detect unexpected failures and correct and adapt data-throttling rates, if needed.

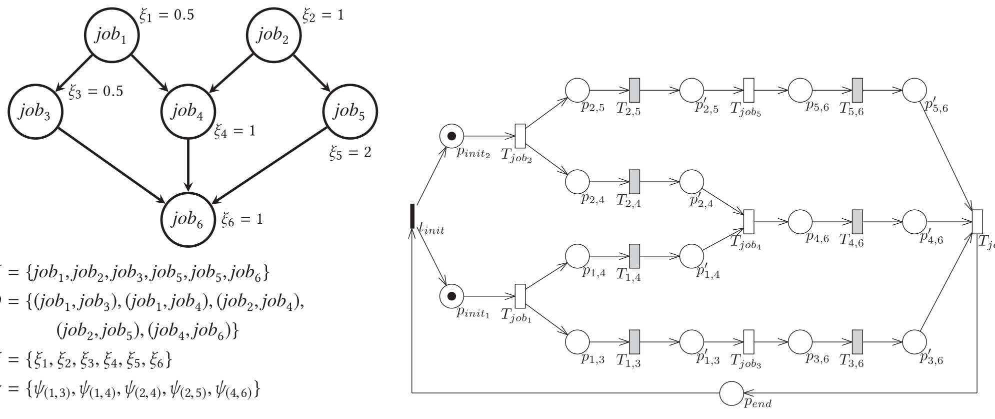
6.1 An Automated Data-throttling Analysis Approach

Algorithm 1 shows the pseudo-code to conduct an automated workflow imbalance analysis. This algorithm was previously introduced in Rodríguez et al. (2012). Here, it has been adapted to the terminology that we use in this article. As an input, it receives a transformed (e.g. clusterized) DAG workflow model $\mathcal{W} = \langle \mathcal{J}, \mathcal{D}, \xi, \psi \rangle$. As an output, it generates throttling values for (sub-)optimal network and buffer usage.

ALGORITHM 1: Automated data-throttling analysis (adapted from Rodríguez et al. (2012)).

```
Input: Clusterized DAG workflow $\mathcal{W} = \langle \mathcal{J}, \mathcal{D}, \xi, \psi \rangle$
Output: Sub-optimal throttling values for intermediate data transfers
1 Transform $\mathcal{W}$ into an SMG system $\mathcal{S}_{\mathcal{W}} = \langle \mathcal{N}, \mathbf{m_0}, \delta \rangle$
2 Compute the slack-based performance analysis of $\mathcal{S}_{\mathcal{W}}$
3 Build the set of slack clustering $C$
4 foreach cluster $c \in C$ do
5     Compute data-throttling values
6     Recompute slacks using current computed data-throttling values
7 end
```

Step 1 transforms $\mathcal{W}$ into an SMG system $\mathcal{S}_{\mathcal{W}} = \langle \mathcal{N}, \mathbf{m_0}, \delta \rangle$, as explained in Section 4.2. Then, Step 2 performs the slack-based performance analysis as described in Section 5. In this step, the


job_1
$\xi_1 = 0.5$
job_2
$\xi_2 = 1$
job_3
$\xi_3 = 0.5$
job_4
$\xi_4 = 1$
job_5
$\xi_5 = 2$
job_6
$\xi_6 = 1$
p_{init_2}
T_{job_2}
t_{init}
p_{init_1}
T_{job_1}
$p_{2,5}$ $T_{2,5}$ $p'_{2,5}$ T_{job_5} $p_{5,6}$ $T_{5,6}$ $p'_{5,6}$
$p_{2,4}$ $T_{2,4}$ $p'_{2,4}$
T_{job_4} $p_{4,6}$ $T_{4,6}$ $p'_{4,6}$ T_{job_6}
$p_{1,4}$ $T_{1,4}$ $p'_{1,4}$
$p_{1,3}$ $T_{1,3}$ $p'_{1,3}$ T_{job_3} $p_{3,6}$ $T_{3,6}$ $p'_{3,6}$
p_{end}

$$\mathcal{J} = \{job_1, job_2, job_3, job_5, job_5, job_6\}$$
$$\mathcal{D} = \{(job_1, job_3), (job_1, job_4), (job_2, job_4), (job_2, job_5), (job_4, job_6)\}$$
$$\xi = \{\xi_1, \xi_2, \xi_3, \xi_4, \xi_5, \xi_6\}$$
$$\psi = \{\psi_{(1,3)}, \psi_{(1,4)}, \psi_{(2,4)}, \psi_{(2,5)}, \psi_{(4,6)}\}$$

(a) (b)

Fig. 3. (a) A workflow with six task and multiple inter-tasks dependencies and (b) its PN-based representation.

critical path (i.e., the path with longest delay) is obtained and slack values are computed for every execution job in the workflow.

Step 3 deals with *slack clustering*. Note that the data-throttling values computed for a given data transmission may impact other data-throttling computations. Let us illustrate this issue with an example. Consider the DAG workflow depicted in Figure 3, used as running example in the sequel. The slack computation in this example is $\mu(p'_{1,4}) = 0.088063$, $\mu(p'_{3,6} = 0.352254$, and $\mu(p'_{5,6}) = 0.176127)$. That is, data coming in paths from job_1 to job_4, from job_3 to job_6, and from job_5 to job_6, respectively, arrive sooner than data coming from the other paths. Computing data-throttling values for inputs at job_6 in first place and then at job_4 implies that when inputs at job_4 are delayed, then $job_4 \rightarrow job_6$ are also implicitly delayed, and, hence, the previous adjustment done at job_6 to minimize the slack may be invalid. Thus, to minimize the complexity of this process, we classify data transfers into groups of elements that are mutually independent.

Recall that slacks provide a measure of the expected delay in synchronization points. We define a *synchronization point* as a job with multiple inputs, i.e., a job j is a synchronization point if $j \in \mathcal{J}_{IO} \bigcup \mathcal{J}_I$, and $(j', j) \in \mathcal{D}' \subset \mathcal{D}, j' \neq j \in \mathcal{J}, |\mathcal{D}'| > 1$.

In steps 4–7, the computation of data throttling to minimize delays is performed. This computation is as follows. As proved by Theorem 5.1, for each place p with a slack $\mu(p) > 0$, we compute an addition delay as $\alpha = \frac{\mu(p)}{\Theta}$. Let us consider the transmission $d = (j_i, j_k) \in \mathcal{D}$ that produces $\mu(p) > 0$. As we commented previously, for the sake of simplicity, we assume that the transmission time of $\psi(j_i, j_k)$ depends only on the bandwidth BW of the data link between j_i and j_k and a latency ε. Thus, $\delta(t_d) = \frac{\psi(j_i, j_k)}{BW} + \varepsilon$. To throttle a data transmission between j_i and j_k, and assuming a constant latency, we compute the throttled bandwidth BW' as

$$BW' = \left(\frac{1}{BW} + \frac{\alpha}{\psi(j_i, j_k)}\right)^{-1}. \tag{7}$$

Furthermore, when j_i belongs to the slowest workflow path and assuming only one link for each job, the same quantity of reduction of bandwidth in one data transmission can be increased in the slowest path, hence accelerating the workflow makespan. Then, for each slack cluster, we compute the throttling values for the slack values as indicated above. Finally, these data-throttling values are considered as an output, while slacks are recomputed considering these throttling values to have the correct values needed for the next cluster.

Let us illustrate how Algorithm 1 works by means of an example. Consider the workflow $\mathcal{W}$ depicted in Figure 3(a) as an input, which is composed of six jobs taking each job_i a total of ξ_i to complete its execution. We assume that each job_i is executed at a different host machine and that the infrastructure has a dedicated network topology, i.e., every host interconnects to the others. The available bandwidth is 100Mbps with a latency of $\varepsilon = 1e^{-4}$s. For the sake of simplicity, we assume that $\psi_{i,k} = 10MB, \forall(job_i, job_k) \in \mathcal{D}$ (and, therefore, $\delta(T_{i,j}) = 0.8001$ for every $T_{i,j}$).

The Petri net obtained after the transformation of $\mathcal{W}$ is shown in Figure 3(b) (step 1). The slowest path of the workflow is, in this case, $job_2 \rightarrow job_5 \rightarrow job_6$. Note that $\mathcal{W}$ has two synchronization points, where a slack may appear in their inputs, i.e. in the input data paths of job_4 and input the data of job_6. The slack computations in this example are $\mu(p'_{1,4}) = 0.088063$, $\mu(p'_{3,6}) = 0.352254$, and $\mu(p'_{4,6}) = 0.176127$ (step2). That is, data coming from paths (i) from job_1 to job_4, (ii) from job_3 to job_6, and (iii) from job_5 to job_6, respectively, arrive sooner than the data coming from the other paths.

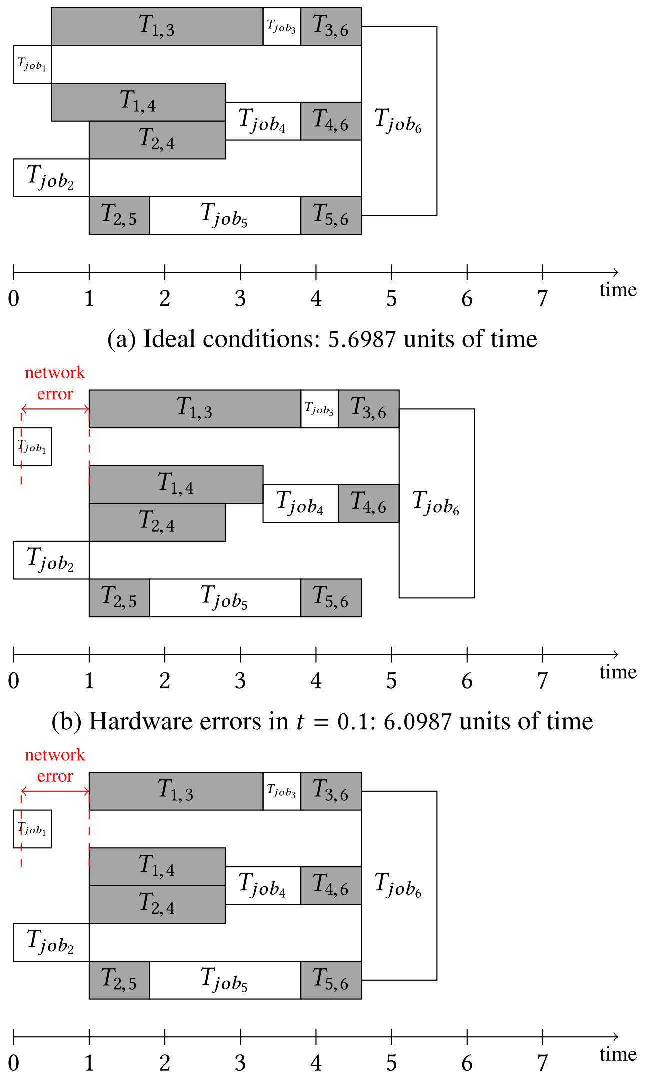
Step 3 clusters inputs with slack whose data-throttling does not affect others. We use the concept of *slack levels* introduced in Rodríguez et al. (2012) to cluster slacks. In this case, the paths $job_3 \rightarrow job_6$ and $job_4 \rightarrow job_6$ are at slack level 1, while $job_1 \rightarrow job_4$ is at slack level 2 (since its throttling may affect to the path $job_4 \rightarrow job_6$, which has some slack). Hence, $p'_{3,6}$ and $p'_{4,6}$ are grouped in slack level 1, while $p'_{1,4}$ is grouped in slack level 2. Note that slack clustering determines the number of places with some slack in each workflow path.

Steps 4–7 compute data-throttling values and recompute slack values after each cluster adjustment. The adjustment is performed in increasing order of slack level, starting from level 1. Thus, in the example, we first decrease the bandwidth for transmission $job_2 \rightarrow job_4$. Note that under a different network topology such a single link per job, the reduction of bandwidth in those transitions can be used to increase the one for the transmission $job_2 \rightarrow job_5$. Data-throttling values obtained after computation are as follows: transmission $job_1 \rightarrow job_3$ is throttled to a 29.55% of the total bandwidth, transmission $job_1 \rightarrow job_4$ to 35.86%, and transmission $job_2 \rightarrow job_4$ to 45.62%. A computation of the makespan of the running example under different scenarios is given in Figure 4.

6.2 Toward an Autonomic Data-throttling Analysis Approach

The automated data-throttling analysis of Algorithm 1 relies on performance information from historical executions, and it assumes that execution environmental conditions do not change, while the workflow is being executed. However, in real distributed environments this assumption does not hold, since execution conditions may change at runtime due to several issues, such as spurious errors, intermittent network failures, performance interference, or even hardware failures that makes computational resources unusable. In those cases, data-throttling rates computed by Algorithm 1 become invalid, since the conditions assumed for its computation are different.

Let us show the effect of these unexpected situations by means of the running example. Figure 4(a) plots the makespan of the workflow assuming no errors and after applying data-throttling values computed by Algorithm 1. Suppose that at time $t = 0.1$, a hardware error occurs in the node there job_1 is being executed, and its network card becomes irresponsive for 0.9s. Hence, although job_1 finishes its execution successfully, it cannot send any data to other jobs until the network card recovers its full functionality. Hence, this unexpected failure implicitly delays the arrival of input data to job_3 and job_4. This effect is illustrated in red, dashed lines in Figure 4(b). Thus, the overall workflow execution suffers a penalty, since the computed data-throttling rates, which were computed before starting the workflow execution, were unaware of the unexpected failure that occurs upon the real execution.



(a) Ideal conditions: 5.6987 units of time

(b) Hardware errors in $t = 0.1$: 6.0987 units of time

(c) Hardware errors in $t = 0.1$: 5.6987 units of time

Fig. 4. Makespan of the running example considering the use of Algorithm 1 in a scenario of (a) ideal conditions (no errors) and (b) in a scenario where a network hardware error occurs at $t = 0.1$. The scenario of (c) considers the use of Algorithm 2 and the same occurrence of the error.

To overcome the *static nature* of execution conditions implicitly assumed in Algorithm 1, we propose an autonomic approach that controls and monitors the execution of jobs in real time to update the data-throttling rates, when unexpected performance issues occur.

Our autonomic approach is presented in Algorithm 2. Basically, it improves the previous Algorithm 1 by recomputing data-throttling rates when some deviation from expected timing is detected. As an input, it receives a clusterized DAG workflow model $\mathcal{W} = \langle \mathcal{J}, \mathcal{D}, \xi, \psi \rangle$. As an output, it ensures that the workflow $\mathcal{W}$ was executed and the usage of input buffers and network bandwidth were optimized as much as possible.

ALGORITHM 2: Automatic data-throttling analysis.

```
Input: Clusterized DAG workflow $\mathcal{W} = \langle \mathcal{J}, \mathcal{D}, \xi, \psi \rangle$
Output: Workflow $\mathcal{W}$ execution with an optimal usage of input buffers andnetwork bandwidth
1  Compute Algorithm 1 to obtain data-throttling rates under ideal conditions
2  Set watchpoints after the execution of jobs $j \in \mathcal{J}$ whose transmissions are throttled
3  Start workflow $\mathcal{W}$ enactment
4  repeat
5      if a watchpoint in job j is reached then
6          if execution time of j exceeds expected execution time then
7              Update data-throttling rates by computing Algorithm 1
8              Reset watchpoints after the execution of jobs $j \in \mathcal{J}$ whose transmissions are throttled
9          end
10     end
11 until $\mathcal{W}$ has been executed
```

Step 1 computes data-throttling rates using Algorithm 1. Then, step 2 sets a watchpoint for each job $j \in \mathcal{J}$ whose transmissions must be throttled. In step 3, the workflow enactment is launched. Steps 4–10 ensure that $\mathcal{W}$ is executed in an optimized way. To this aim, the expected execution time of j is compared to current elapsed time when a watchpoint is reached. When a time deviation is observed, Algorithm 1 is executed again considering the new conditions; that is, we update the $\mathcal{W}$ with the real execution time of ended job and of completed transmissions (step 7). Then, new watchpoints are set (if any) in step 8. The iteration loop ends when $\mathcal{W}$ is totally executed.

Recall the running example and the aforementioned unexpected situation. The workflow makespan under this unexpected condition is depicted in Figure 4(b). Consider now that we apply Algorithm 2 rather than Algorithm 1. As commented on, we obtain initial data-throttling rates of 29.55% for transmission $job_1 \rightarrow job_3$, 35.86% for $job_1 \rightarrow job_4$, and 45.62% for $job_2 \rightarrow job_4$. These values are computed in step 1 of Algorithm 2. Then, watchpoints are set to job_1 and job_2, and workflow enactment (steps 2 and 3) starts. The watchpoint of job_1 is first reached, as job_1 ends before job_2. Since the expected end of time of $task_1$ (0.5 units of time) differs from current end of time (0.9 u.t.), an update of data-throttling rates is required. The new data-throttling rates are 34.39% for $job_1 \rightarrow job_3$, 43.27% to $job_1 \rightarrow job_4$, while $job_2 \rightarrow job_4$ suffers no alteration. When the next watchpoint in job_2 is reached, no update is required, since there is no mismatch between timestamps. Finally, workflow ends its execution.

The makespan of the workflow $\mathcal{W}$ when executed under the control of Algorithm 2 is illustrated in Figure 4(c). Note that the error effect was overriden by the dynamic data throttling by means of increasing transmission rates. In this case, the workflow makespan is equal to 5.6987, i.e., the same as in the scenario of Figure 4(a). A question may arise regarding the computational overhead caused by execution of Algorithm 2. In the next section, we show the empirical results that verify that such overhead is insignificant. Furthermore, we use the Montage (Berriman et al. 2007) workflow to compute how much the usage of input buffers improves when Algorithm 1 and Algorithm 2 are applied.

7 EXPERIMENTS AND VALIDATION

In this section, we first analyse the influence of data throttling on workflow makespan: We provide a formal proof, so that we can guarantee, for any case, that the workflow makespan with data throttling is always lower than or equal to the makespan of the same workflow when no data throttling is performed. The validation of Algorithm 1 was already given in Rodríguez et al. (2012). The experiments were conducted by simulation using the SimGrid tool (Casanova et al. 2008), which enables us to simulate a scientific workflow in a given execution platform, both defined following a well-defined XML syntax. SimGrid allows us to specify the network bandwidth rates to be used between different hosts. However, it is unsuitable for the validation of Algorithm 2, since it does not support the modification of a data transmission rate of a transmission task, once the simulation of the task was started. To overcome that problem, we developed our own workflow simulator so that we can better show evidence of our autonomic approach. We describe the experiments performed and provide a discussion of results. Last, we (briefly) comment the main limitations of our approach.

7.1 Data-throttling Influence on Workflow Makespan

As discussed in the previous section, considering a negligible computational cost of Algorithm 2, the makespan of a workflow where Algorithm 2 is being applied can never be greater than the makespan of the same workflow without the effect of Algorithm 2.

Theorem 7.1. *The makespan of a workflow in which data are throttled is always lower than or equal to the makespan of the same workflow when no data throttling is performed, being the conditions of both workflow executions equal.*

Proof. Let $\mathcal{W} = \langle \mathcal{J}, \mathcal{D}, \xi, \psi \rangle$ be a workflow and $\mathcal{N}$ its corresponding SMG, obtained after model transformation as indicated in Section 4. Let Θ the makespan of the workflow $\mathcal{W}$ and let $\mathbf{y}$ be the slowest P-semiflow of $\mathcal{N}$, whose cycle time is equal to $\Gamma = 1/\Theta$ as computed by LPP 3 (or its dual). Assume that there exists some place p with a positive slack value, i.e., $\exists p \in P : \mu(p) > 0$. By definition of slack values, we know that p does not belong to the support of $\mathbf{y}$, i.e., $p \notin \|\mathbf{y}\|$.

Now, we perform data throttling in the workflow; in the following, we use the symbol $'$ as a super-index to indicate the values after having performed data throttling. Hence, we apply Theorem 5.1, and we increase the delay of transition ${}^\bullet p$ by $0 < \alpha \leq \frac{\mu(p)}{\Theta}$ units of time, i.e., $\delta'(p) = \delta(p) + \alpha$.

Let $\mathbf{y}_p$ be the P-semiflow that contains the place p in its support. Let Γ' be the cycle time associated to $\mathbf{y}_p$, which is affected by the addition of α. Suppose that the makespan of the workflow has increased, i.e., $\Gamma' > \Gamma$ (that is, $\Theta' < \Theta$). Since the steady-state throughput has changed, we do not know how $\tilde{\mathbf{m}}'(p)$ and $\tilde{\mathbf{m}}(p)$ are related.

First, let us suppose that $\tilde{\mathbf{m}}'(p) > \tilde{\mathbf{m}}(p)$. Hence, $\delta'(p)\Theta' > \delta(p)\Theta + \mu(p)$. Since $0 < \alpha \leq \frac{\mu(p)}{\Theta}$, then:

$$(\delta(p) + \alpha)\Theta' > \delta(p)\Theta + \mu(p) \Rightarrow \left(\delta(p) + \frac{\mu(p)}{\Theta}\right)\Theta' > \delta(p)\Theta + \mu(p).$$

Rearranging terms: $\delta(p)(\Theta' - \Theta) > \mu(p)(1 - \frac{\Theta'}{\Theta})$. Since $\Theta' < \Theta$, then $0 > \delta(p)(\Theta' - \Theta)$. Therefore, $0 > \mu(p)(1 - \frac{\Theta'}{\Theta})$. By definition of slack, $\mu(p) > 0$, and, thus, $0 > 1 - \frac{\Theta'}{\Theta} \Rightarrow \Theta' > \Theta$, which is a contradiction.

Let us suppose now that $\tilde{\mathbf{m}}'(p) \leq \tilde{\mathbf{m}}(p)$. Hence, $\delta'(p)\Theta' \leq \delta(p)\Theta + \mu(p)$. As before,

$$(\delta(p) + \alpha)\Theta' \leq \delta(p)\Theta + \mu(p) \Rightarrow \left(\delta(p) + \frac{\mu(p)}{\Theta}\right)\Theta' \leq \delta(p)\Theta + \mu(p).$$

Rearranging terms: $\delta(p)(\Theta' - \Theta) \leq \mu(p)(1 - \frac{\Theta'}{\Theta})$. Note that $\Theta' < \Theta$, then $0 > \delta(p)(\Theta' - \Theta)$. Similarly, $\frac{\Theta'}{\Theta} < 1$ and thus, $\mu(p)(1 - \frac{\Theta'}{\Theta}) > 0$, since $\mu(p) > 0$ by definition of slack. Therefore, $\delta(p)(\Theta' - \Theta) \leq \mu(p)(1 - \frac{\Theta'}{\Theta})$ is again a contradiction.

Hence, $\Gamma' \leq \Gamma$. □

7.2 Experimental Test-bed for Algorithm 2

SimGrid simulator was used for the validation of Algorithm 1 in Rodríguez et al. (2012). However, as SimGrid only supports setting up data transfer rates in transmission tasks prior to their execution, our dynamic data-throttling strategy would have no effect if the transmission task to be throttled had already started its execution. To overcome this issue, we developed our own Java-based tool for experimentation, termed as `DT4SW`. Our tool simulates the network model in a way closer to real networks. Furthermore, the source code has been freely released under GPLv3 (see https://gitlab.unizar.es/rrodrigu/data-throttling-workflows) to enable the reproducibility of the experiments and to foster research in the area of data throttling. Currently, `DT4SW` allows us to simulate the execution of a scientific workflow (specified as a DAX file). Our tool assumes that the execution platform has enough nodes to execute each workflow task independently. Although the current version of the tool does not allow to specify heterogeneous nodes, it accepts different execution parameters, as the network topology (two topologies were considered, each node has either a single network link or either as many network links as nodes in the platform), the network bandwidth, and the network latency.

Furthermore, our tool also incorporates a failure model. In particular, the user can simulate a scientific workflow assuming an upper bound to the network bandwidth for every transmission task of the workflow. Hence, when a transmission task begins its execution, its network bandwidth is upper bounded by such a rate. `DT4SW` also provides a rich API allowing the user to simulate a workflow under different data-throttling strategies (none, static, or dynamic).

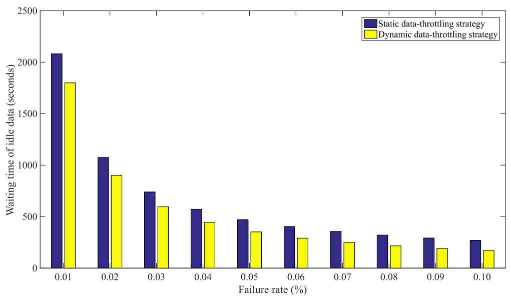
As a workflow for the experimentation, we chose the Montage workflow. The Montage workflow (Berriman et al. 2007) is a scientific workflow used to create image mosaics in the astrophysics domain. The abstract workflow description and performance information were collected from DAX files generated with the Pegasus workflow system (available at https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator). In particular, in this article, we consider three variants of Montage workflow, each one containing a different number of tasks (25, 50, and 100 tasks). We initially considered the Montage variant of 1,000 tasks, but our tool ran out of memory. We discuss these issues more in detail in Section 7.4. Let us remark that these workflows are synthetic workflows generated using the information gathered from actual executions of scientific workflows on the Grid and that the number of tasks were used as bounds to estimate performance. As observed in a Montage workflow in practice, our approach is applicable also for a different number of tasks.

Experiments were performed in a virtualized environment running a GNU/Linux Debian 8.10 AMD64, with 768MB of RAM memory and an Intel Core i7-5650U CPU at 2.20GHz. Regarding software, we have used the Java Virtual Machine version 8.0.151 and GLPK version 3.6 as ILP solver.

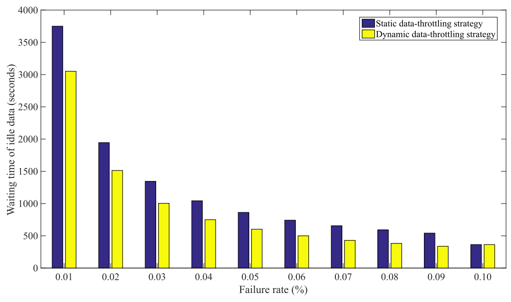
To validate Algorithm 2, we have simulated the synthetic Montage workflows under static and dynamic data-throttling strategies, while upper bounding the transmission rates from 1% to 10%. Under these scenarios, we expect to see the effect of having a dynamic approach that observes deviations from the expected behavior. The workflow makespan for each strategy and experiment is always the same, and, thus, we deliberately omit it from comparison. In this case, data throttling does not influence workflow makespan. This is mainly motivated, because the key metric being considered by our approach is resource usage and thus a more effective use can be made of existing storage/compute resources rather than just decreasing the overall makespan.

7.3 Experimental Validation of Algorithm 2 and Discussion

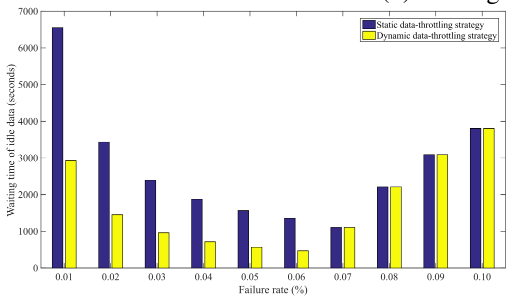
Figure 5(a), (b), and (c) depicts the sum of waiting time (in seconds) of idle data in input buffers of tasks with multiple input dependencies for Montage with 25, 50, and 100 tasks, respectively. For every upper bounded rate (from 1% to 10%), we computed the sum of waiting time simulating the



(a) Montage with 25 tasks



(b) Montage with 50 tasks



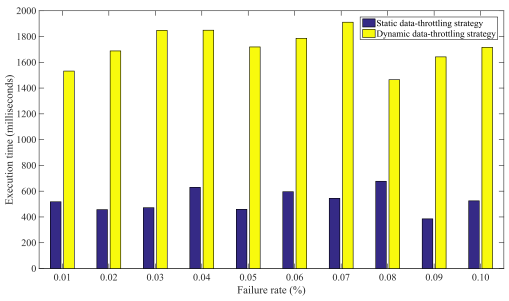
(c) Montage with 100 tasks

Fig. 5. Sum of waiting times of idle data in static and dynamic data-throttling strategies.

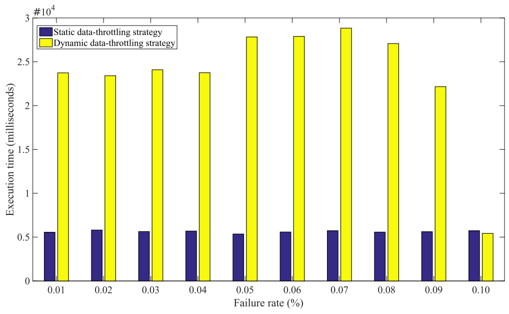
workflow under static and dynamic data throttling. In all cases, a dynamic data-throttling strategy provides a lower waiting time. It is observed that from a certain point (10% for Montage with 50 tasks and 7% for Montage with 100 tasks) both strategies obtain the same results. This happens because the dynamic data-throttling strategy is not taking place, since the upper bounded rate is not producing any observable delays from expected execution.

The execution time of both data-throttling strategies are shown in Figure 6. As shown, the number of tasks clearly has an impact in the execution time of both strategies. The execution time of the dynamic data-throttling strategy is always greater than the static one, since it is executed every time that some deviation is observed while the static data-throttling strategy is executed only once. These graphs also confirm what we claimed before: From a certain point (10% for Montage with 50 tasks and 7% for Montage with 100 tasks), the dynamic data-throttling strategy does not update the data-throttling rates.

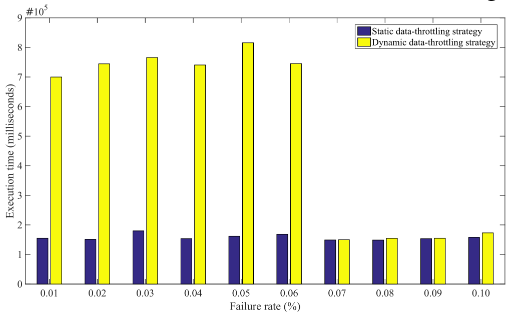
In summary, these experiments allow us to conclude the following. First, a dynamic data-throttling strategy minimizes the waiting time of idle data in tasks with multiple input dependencies. Second, the execution time of the dynamic data-throttling strategy is negligible, since it took less than 1s for the larger workflow used in the experiments (100 tasks). Of course, larger workflow would need longer execution time. However, the usage of input buffers would be minimized as well, and the overhead induced by computation could be overlapped with the actual execution of the workflow. We aim at studying this tradeoff in more detail in the future. However, implementing a data-throttling mechanism will add further technological complexity to the workflow management system. First, network bandwidth can be monitored by any of the available tools. Second, the proposed algorithm for imbalance analysis and data-throttling rates also adds some computations. Finally, data throttling can be accomplished by a traffic-shaping mechanism such as token bucket as discussed by Park and Humphrey (2008).



(a) Montage with 25 tasks



(b) Montage with 50 tasks



(c) Montage with 100 tasks

Fig. 6. Execution times of static and dynamic data-throttling strategies.

7.4 Limitations

Here, first we briefly discuss the main limitations of our tool, and then we discuss the computational complexity of Algorithm 2. The failure model assumed by DT4SW is simplistic. As future work, we aim at extending DT4SW to first support input files describing executional platform (as SimGrid does) and to improve how failures are simulated.

The main limitation of our approach is the computational complexity of the Algorithm 2. Let us recall that the transformation from a workflow $\mathcal{W}$ with n tasks and m transmissions would generate a Petri net having $2m$ places and nm transmissions. Therefore, the LPPs used by Algorithm 2 have $4m + 1$ constraints (at most) and work with matrices of $2m \times n$ dimensions. Hence, although the number of constraints are tractable, the consumption of memory is considerable. Furthermore, to dispose of a tool developed in Java is not helping for this issue. The optimization of Algorithm 2 to save execution time and memory space deserves further study.

8 CONCLUSIONS AND FUTURE WORK

Large datasets analysis and complex scientific simulations are normally represented as workflows. These workflows are deployed into distributed and parallel computing infrastructures to speed up their execution. However, workflow imbalance in runtime and data dependencies may appear when accommodating the workflow to the computing infrastructure, and the subsequent scheduling process might incur in poor network bandwidth and storage usage. Both imbalances can be corrected by enforcing data transfer rates throughout workflow paths, that is, imposing data-throttling transfer mechanisms.

In this article, we have proposed a technique to analyse and quantify workflow imbalance, based on linear programming techniques and Petri nets, and a dynamic data-throttling approach to overcome the imbalance at runtime. We have also introduced a set of rules to transform a data-intensive workflow into a Petri net model. Furthermore, we have explained in detail the theoretical foundations of our approach and provided theorems to prove our claims. In particular, our approach is based on the structural analysis of the Petri net model obtained after transformation and is able to accommodate ratios in presence of unexpected conditions, such as network or host failures. We have validated our approach by applying it to real workflows of different nature. Our technique is therefore complementary to the existing clustering/scheduling techniques and can be used to improve the overall workflow performance. Our findings show that data throttling is not only feasible but also does not introduce a significant overhead. Besides, it minimizes the usage of input buffers and network bandwidth. Our findings also show that dynamic data-throttling approach outperforms much better than other approaches when applied to workflows with large structure imbalance.

As future work, we plan to integrate our dynamic data-throttling approach in a real workflow enactment to experimentally validate our approach. Furthermore, we aim at investigating self-adaptive strategies that continually monitor and adapt to improve workflow computation and data transmissions under unexpected conditions.

REFERENCES

Wil M. P. van der Aalst, Alexander Hirnschall, and H. M. W. (Eric) Verbeek. 2002. An alternative way to analyze workflow graphs. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE'02)*. Springer-Verlag, London, 535–552.

M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. 1995a. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons.

M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. 1995b. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons.

L. Aversano, A. Cimitile, P. Gallucci, and M. L. Villani. 2002. FlowManager: A workflow management system based on Petri nets. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC'02)*. 1054–1059. DOI: https://doi.org/10.1109/CMPSAC.2002.1045148

G. Bruce Berriman, Ewa Deelman, John Good, Joseph C. Jacob, Daniel S. Katz, Anastasia C. Laity, Thomas A. Prince, Gurmeet Singh, and Mei-Hui Su. 2007. Generating complex astronomy workflows. In *Workflows for e-Science*, Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields (Eds.). Springer, London, 19–38. DOI: https://doi.org/10.1007/978-1-84628-757-2_3

J. Campos and M. Silva. 1992. Structural techniques and performance bounds of stochastic Petri net models. In *Proceedings of the Advances in Petri Nets* 1992, G. Rozenberg (Ed.). Lecture Notes in Computer Science, vol 609. Springer, Berlin, Heidelberg. 352–391.

J. Carmona, J. Júlvez, J. Cortadella, and M. Kishinevsky. 2009. Scheduling synchronous elastic designs. In *Proceedings of the 2009 Application of Concurrency to System Design Conference (ACSD'09)*.

Henri Casanova, Arnaud Legrand, and Martin Quinson. 2008. SimGrid: A generic framework for large-scale distributed experiments. In *Proceedings of the 10th IEEE International Conference on Computer Modeling and Simulation*.

Weiwei Chen, Rafael Ferreira da Silva, Ewa Deelman, and Rizos Sakellariou. 2015. Using imbalance metrics to optimize task clustering in scientific workflow executions. *Fut. Gener. Comput. Syst.* 46, C (May 2015), 69–84. DOI: https://doi.org/10.1016/j.future.2014.09.014

G. Chiola, C. Anglano, J. Campos, J. M. Colom, and M. Silva. 1993. Operational analysis of timed Petri nets and application to the computation of performance bounds. In *Proceedings of the 5th International Workshop on Petri Nets and Performance Models (PNPM'93)*. IEEE Computer Society Press, Los Alamitos, CA, 128–137.

E. Deelman, G. Mehta, G. Singh, M. Su, and K. Vahi. 2007. Pegasus: Mapping large-scale workflows to distributed resources. In *Workflows for eScience*. Springer, 376–394.

Rubing Duan, Thomas Fahringer, Radu Prodan, Jun Qin, Alex Villazón, and Marek Wieczorek. 2005. Real world workflow applications in the Askalon grid environment. In *Proceedings of the Advances in Grid Computing European Grid Conference (EGC'05)*. Springer, Berlin, 454–463. DOI: https://doi.org/10.1007/11508380_47

Rubing Duan, Farrukh Nadeem, Jie Wang, Yun Zhang, Radu Prodan, and Thomas Fahringer. 2009. A hybrid intelligent method for performance modeling and prediction of workflow activities in Grids. In *Proceedings of the 9th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'09)*. IEEE Computer Society, Los Alamitos, CA, 339–347. DOI: https://doi.org/10.1109/CCGRID.2009.58

Rubing Duan, Radu Prodan, and Thomas Fahringer. 2006. Run-time optimisation of grid workflow applications. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID'06)*. IEEE Computer Society, Los Alamitos, CA, 33–40. DOI: https://doi.org/10.1109/ICGRID.2006.310995

G. Florin and S. Natkin. 1985. Les réseaux de Petri stochastiques. *Techn. Sci. Inf.* 4 (1985), 143–160.

Zhijie Guan, Francisco Hernandez, Purushotham Bangalore, Jeff Gray, Anthony Skjellum, Vijay Velusamy, and Yin Liu. 2006. Grid-flow: A Grid-enabled scientific workflow system with a Petri-net-based interface: Research articles. *Concurr. Comput. Pract. Exper.* 18 (Aug. 2006), 1115–1140. Issue 10. DOI: https://doi.org/10.1002/cpe.v18:10

Andreas Hoheisel. 2006. User tools and languages for graph-based Grid workflows: Research articles. *Concurr. Comput. Pract. Exper.* 18, 10 (2006), 1101–1113. DOI: https://doi.org/10.1002/cpe.v18:10

Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.

John D. C. Little. 1961. A proof for the queuing formula: L= λ W. *Oper. Res.* 9, 3 (1961), 383–387. DOI: https://doi.org/10.2307/167570

Tadao Murata. 1989. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, Vol. 77. 541–580.

F. Nerieri, R. Prodan, T. Fahringer, and Hong-Linh Truong. 2006. Overhead analysis of Grid workflow applications. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID'06)*. IEEE Computer Society, Los Alamitos, CA, 17–24. DOI: https://doi.org/10.1109/ICGRID.2006.310993

Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. 2006. Taverna: Lessons in creating a workflow environment for the life sciences: Research articles. *Concurr. Comput. Pract. Exper.* 18, 10 (2006), 1067–1100. DOI: https://doi.org/10.1002/cpe.v18:10

Sang-Min Park and M. Humphrey. 2008. Data throttling for data-intensive workflows. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. 1–11. DOI: https://doi.org/10.1109/IPDPS.2008.4536306

Simone Pellegrini, Andreas Hoheisel, Francesco Giacomini, and Antonia Ghiselli. 2008. Using GWorkflowDL for middleware-independent modeling and enactment of workflows. In *Proceedings of the CoreGRID Integration Workshop 2008*.

C. Ramchandani. 1974. *Analysis of Asynchronous Concurrent Systems by Petri Nets*. Ph.D. Dissertation. Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, MA.

Ricardo J. Rodríguez and Jorge Júlvez. 2010. Accurate performance estimation for stochastic marked graphs by bottleneck regrowing. In *Proceedings of the 7th European Performance Engineering Workshop (EPEW'10)*, Lecture Notes in Computer Science, Vol. 6342. Springer, 175–190. DOI: https://doi.org/10.1007/978-3-642-15784-4_12

Ricardo J. Rodríguez, Rafael Tolosana-Calasanz, and Omer F. Rana. 2012. Automating data-throttling analysis for data-intensive workflows. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'12)*. IEEE, 310–317. DOI: https://doi.org/10.1109/CCGrid.2012.27

Ricardo J. Rodríguez, Rafael Tolosana-Calasanz, and Omer F. Rana. 2012. Measuring the effectiveness of throttled data transfers on data-intensive workflows. In *Proceedings of the 6th KES International Conference on Agent and Multi-Agent Systems, Technologies, and Applications (KES-AMSTA'12)*. 144–153. DOI: https://doi.org/10.1007/978-3-642-30947-2_18

Jacek Sroka and Jan Hidders. 2009. Towards a formal semantics for the process model of the Taverna workbench. Part ii. *Fundam. Inf.* 92, 4 (2009), 373–396.

I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields (Eds.). 2007. *Workflows for eScience: Scientific Workflows for Grids*. Springer.

Rafael Tolosana-Calasanz, José A. Bañares, Pedro Álvarez, Joaquín Ezpeleta, and Omer F. Rana. 2010a. An uncoordinated asynchronous checkpointing model for hierarchical scientific workflows. *J. Comput. Syst. Sci.* 76, 6 (Sep. 2010), 403–415.

Rafael Tolosana-Calasanz, José A. Bañares, Omer F. Rana, Pedro Álvarez, Joaquín Ezpeleta, and Andreas Hoheisel. 2010b. Adaptive exception handling for scientific workflows. *Concurr. Comput. Pract. Exper.* 22, 5 (Apr. 2010), 617–642.

Rafael Tolosana-Calasanz, Omer Rana, and José Bañares. 2008. Automating performance analysis from Taverna workflows. In *Component-Based Software Engineering*, Michel Chaudron, Clemens Szyperski, and Ralf Reussner (Eds.). Lecture Notes in Computer Science, Vol. 5282. Springer, Berlin, 1–15. DOI: https://doi.org/10.1007/978-3-540-87891-9_1

Wil van der Aalst and Kees van Hee. 2004. *Workflow Management: Models, Methods, and Systems*. The MIT Press.

Michal Vossberg, Andreas Hoheisel, Thomas Tolxdorff, and Dagmar Krefting. 2008. A reliable DICOM transfer grid service based on Petri net workflows. In *Proceedings of the 2008 8th IEEE International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, Los Alamitos, CA, 441–448. DOI: https://doi.org/10.1109/CCGRID.2008.122

Jia Yu and Rajkumar Buyya. 2005. A taxonomy of workflow management systems for Grid computing. *CoRR* 34, 3 (Sep. 2005), 44–49.