Data-Centric Iteration in Dynamic Workflows

Jonas Dias[a], Gabriel Guerra[a], Fernando Rochinha[a], Alvaro L.G.A. Coutinho[a], Patrick Valduriez[b], Marta Mattoso [a§]

[a]COPPE - Federal University of Rio de Janeiro, Rio de Janeiro, Brazil.
[b]INRIA and LIRMM, Montpellier, France.

{jonasdias, marta}@cos.ufrj.br, alvaro@nacad.ufrj.br
gguerra@mecsol.ufrj.br, faro@mecanica.coppe.ufrj.br
Patrick.Valduriez@inria.fr

Summary

Dynamic workflows are scientific workflows to support computational science simulations, typically using dynamic processes based on runtime scientific data analyses. They require the ability of adapting the workflow, at runtime, based on user input and dynamic steering. Supporting data-centric iteration is an important step towards dynamic workflows because user interaction with workflows is iterative. However, current support for iteration in scientific workflows is static and does not allow for changing data at runtime. In this paper, we propose a solution based on algebraic operators and a dynamic execution model to enable workflow adaptation based on user input and dynamic steering. We introduce the concept of iteration lineage that makes provenance data management consistent with dynamic iterative workflow changes. Lineage enables scientists to interact with workflow data and configuration at runtime through an API that triggers steering. We evaluate our approach using a novel and real large-scale workflow for uncertainty quantification on a 640-core cluster. The results show impressive execution time savings from 2.5 to 24 days, compared to non-iterative workflow execution. We verify that the maximum overhead introduced by our iterative model is less than 5% of execution time. Also, our proposed steering algorithms are very efficient and run in less than 1 millisecond, in the worst-case scenario.

Keywords

Scientific Workflows; Dynamic Workflows; Iteration; Steering

[§]Corresponding author
Contact: marta@cos.ufrj.br

Contact information for the author responsible for correspondence:
Marta Mattoso
D.Sc. Associate Professor, COPPE, Federal University of Rio de Janeiro
P.O. Box 68511, 21941-972 Rio de Janeiro, RJ, Brazil
Phone: +55-21-2562-8694
Fax: +55-21-2562-8080
Email: marta@cos.ufrj.br

1. Introduction

Large-scale scientific experiments tend to explore big datasets, searching for confirmation (or not) of given hypotheses. The process typically involves multiple computing steps supported by programs and chained with scripts. Scientific Workflow Management Systems (SWfMS) [29] have been successful in helping scientists modeling this process as a workflow of activities that operate on the datasets. SWfMS implement a workflow specification model and its execution model. The specification model is supported by a workflow programming language that allows for workflow composition. The execution model is supported by an execution engine, which also provides for result analysis, based on provenance [12]. SWfMS should support specific needs of large-scale workflows. For instance, to reduce execution time, data intensive workflows need to run in parallel in high performance computing (HPC) environments. They may also need specialized features such as iteration.

Iteration is a basic concept that is supported by most programming languages. Because it is intuitive, it has been used extensively in the development of algorithms and computational models. Scientists usually program their iterative methods directly within compiled applications or scripts. However, when modeling an iterative application as a scientific workflow, the workflow language has to support iterative constructs. Current support for iteration in scientific workflows is designed for simpler loop behaviors [1,7,18,21,30]. They typically are control-flow-based and rely on three iteration types [10]: (1)counting loops without dependencies, (2) counting loops with dependencies and (3) conditional loops. However, these iteration types are static and scientists cannot change the loop specification at runtime.

Workflow configuration is likely to change during its life cycle, as the scientist may need to refine a numerical model, try other algorithms or explore different slices of the parameter space to find better results. This exploratory nature of large-scale scientific experimentation makes workflows iterative. The workflow specification needs to be evaluated and explored several times by scientists until a solution is found. Thus, scientists evaluate the results they get, and as adjustments are needed to achieve better results, they must resubmit the workflow execution after doing the necessary changes. Waiting for these results may take a very long time. Scientists usually do not wait until the end of the workflow execution. They try to analyze partial results and, if they can infer that the execution is not on the right track, they abort it, refine data parameters on the specification, and restart. Such manual support of iteration makes it hard to manage the evolution of scientific data analysis. For example, scientists may find it hard to analyze the converged error at runtime or discover which parameter combination they are exploring has produced the best outcome.

To tackle the exploratory nature of science and the dynamic process involved in scientific analysis, dynamic workflows have been identified as an open challenge [13]. Dynamic workflows are scientific workflows that are subject to rapid reuse and exploration accompanied by continuous adaptation and improvement [13]. They need the ability of adapting the workflow based on external events such as human interaction and runtime steering. A typical user interaction would be to adjust data such as filters or domain-specific parameters based on runtime data and partial results. Similar to an iterative optimization approach, scientists may keep the workflow running in a loop, doing adjustments and refinements searching for the best solution as the workflow execution progresses. This runtime adaptation based on user input is more efficient than pre-programing all possible data combinations in a loop, which may not be viable in a scientific workflow due to inherent complexity and unexpected data transformation behavior. In fact, the expression *human-in-the-loop* has gained a lot of interest in dataflow analytics. According to [2] "there remain many patterns that humans can easily detect but computer algorithms have a hard time finding".

In a previous work [8], we showed the benefits of dynamic workflows in reduced order models for heat conduction and genetic algorithms. We then started working with bigger problems such as Uncertainty Quantification (UQ) [14]. Adaptations of the workflow at runtime in [13] were still hand-made through provenance database queries, but the performance improvements obtained, motivated us to develop specific constructs to support dynamic workflows. We identified scenarios that can make good use of dynamic workflows, such as: bioinformatics workflows involving multiple sequence alignment [24] that need to refine the similarity predefined *e-value* for a given input dataset running the same workflow several times; and UQ workflows, which can be used in several scientific domains. One may foresee a similar scenario in other application areas, e.g. in clustering algorithms, where scientists need to refine the quantity of data clusters, or in genetic algorithms, where scientists refine parameters related to crossover and mutation rates. All these examples require runtime data analysis and data adjustments to refine initial configurations. Otherwise, they may increase the experiment data size and complexity by running all possible alternatives to pick the best after the whole execution. This manual exploratory approach cannot be done even at a terabyte scale.

Resubmitting the workflow execution after manually changing its configuration, as done by scientists today, is labor-intensive and error-prone. Because of the lack of provenance data along the trials, they may lose track of what has already been explored and how the workflow has evolved. To improve this iterative experimental process, the user should be able to analyze partial results during execution and to interfere dynamically accordingly, in the next steps of the workflow. There are obvious advantages if the whole iterative process is modeled as a dynamic workflow. However, current iterative approaches do not allow for workflow configuration data to be adapted and fine-tuned during the execution. They typically have an iterative specification model that is submitted for execution as a static plan. Scientists wait until execution ends to analyze results and provenance data and only then decide if it is necessary to interfere in the workflow data configuration to be further executed.

In this paper, we characterize a fourth iteration type – dynamic loops – as a particular type of conditional loops where the condition of the loop and the data being processed in the loop may be adapted during the execution. As done today, programming the dynamic iteration with big data management tools [28] may not be viable due to legacy scientific code complexity and the requirement for sophisticated provenance querying support. Dynamic loops have a data-centric iterative specification because they need to naturally respond to data-driven events. These events may be data inserts and updates at runtime, which change the condition of the loop in its specification. In other words, conditional loops need to be dynamic to adapt the workflow based on the action of the users when they are steering the execution. To support the execution of workflows with dynamic loops, we propose a dynamic execution model guided by runtime data lineage, a concept inspired in provenance that keeps track of the dataflow changes in the loop. Adaptations made by users when they steer the workflow become part of the dataflow by two algorithms that access the data lineage structure and triggers the data-driven events. Thus, dynamic loops are iterations subjected to adaptations based on user input and we support it by means of a data-centric, provenance oriented approach.

Our dynamic loops support follows the algebraic approach for data-centric scientific workflows [25]. Nevertheless, the original algebra, defined in [25], only supports counting loops (1), which is also known as parameter sweeps. It does not support the other types of iterations such as (2) or (3) neither or adaptations based on user input in the algebraic expression specification at runtime. In order to support dynamic loops, we have defined new algebraic operators and the corresponding formalism to keep the algebraic properties. Since dynamic loops are the most general iteration type among the four, when we support dynamic loops in the algebra, we are now currently supporting all the four types of loops.

We introduce the dynamic loops support without increasing workflow complexity by following the intuitive concepts of iterative data-centric programming languages. For instance, FAD [6], a functional database programming language designed for easing optimization and parallel execution, provides a *whiledo* second-order construct for iterating over first-order function calls. Our approach is also tightly-coupled with a relational data provenance model that encompasses W3C PROV [20] and stores additional information such as execution performance and domain data altogether. In order to store domain data, our approach requires the workflow to be instrumented so that the execution engine can capture data and store it in the provenance database during runtime. Consequently, users can query real-time provenance [14,24], which is essential to support decision making during the dynamic steering. In [14], for example, a UQ workflow is improved using real-time provenance queries even without dynamic loops support. The scientists can monitor several execution parameters, like converged error, average and variance of kinetic energy, by accessing partial workflow result analysis through provenance queries. Querying this information, they are able to evaluate whether the simulation converged before the predefined limit or stopped after a different threshold.

In this paper, we propose a data-centric iteration approach for dynamic workflows, with iterative constructs, an algebraic iterative conditional loop operator that is designed to support runtime adaptations provided by a dynamic execution model. The main contributions of this paper are:

- Algebraic operators to support data-centric iteration in dynamic workflows, with iterative constructs.
- A dynamic execution model guided by runtime data lineage, a concept inspired by provenance, that allows for dynamic loops and two algorithms that support runtime adaptation of the workflow based on user input.
- A thorough experimental evaluation using a real-life experiment for UQ analysis in the Oil & Gas domain. We developed a novel iterative workflow for UQ, the iterative execution module and the steering algorithms inside Chiron, a data-centric scientific workflow engine. We executed the workflow on a 640-core cluster, showing the benefits of our approach from the user perspective.

This paper is organized as follows. In Section 2, we introduce our motivating example – a UQ workflow. Section 3 presents our constructs to enable iterations in the specification model of workflow algebra. Section 4 discusses our dynamic execution model, and the dynamic steering algorithms. Section 5 presents the experimental evaluation. Section 6 discusses related work. Section 7 concludes.

2. Uncertainty Quantification Workflow

Throughout this paper, we propose a novel UQ computational dynamic workflow that establishes a rational framework for exploring the potential of computational simulations within the realm of actual applications, like, for instance those related to the Oil & Gas domain [3]. Usually, numerical simulations, aiming at providing to decision makers reliable information, are hard to manage and subject to different sources of uncertainty. UQ [14] can be used to measure the reliability of experiments that run simulations involving complex numerical models. UQ enables the estimation of confidence intervals in predictions by stressing the numerical model taking into consideration the variability on the uncertain inputs, leading tovery large data exploration. The choice of which slice of the input parameter space to explore in a UQ workflow impacts the amount of data produced, which can be very large. A single data exploration in a workflow execution may generate TBytes of data and can easily take more than a week to run in a parallel computer, e.g. a cluster with hundreds of multicore execution nodes. In order to avoid wasting time and storage, scientists usually start the UQ workflow with a modest configuration. If the outcome is below a given quality criteria, they change the data and run the workflow again. These quality criteria are complex to evaluate and are sensitive to the dataflow runtime characteristics, being impossible to precise in advance. Thus, scientists resubmit the execution of a UQ workflow by changing the input configuration until the result meets their expectations.

The general idea of UQ methods is to explore a model using stochastically varied inputs and then combine an average result and standard deviation. The size of the exploration is associated with a pre-defined accuracy of outputs set by scientists. For the Sparse Grid Stochastic Collocation (SGSC) method [31] we use, exploration size grows with increasing interpolation levels. The higher the level, the bigger is the exploration, and, consequently, more simulation instances to be executed.

The amount of data produced in a single experiment grows exponentially with the increase of interpolation levels as shown in Figure 1(a) (the vertical axis is in log scale). The figure shows four scenarios with the size of the input model varying from one million tetrahedra (1M) to seventeen million (17M). In all scenarios, the simulation is configured to store only 1% of the calculated results. These scenarios were executed on a 640-

core cluster with dedicated high-speed shared-disk storage. The results in Figure 1(a) consider only two variables with uncertainty (two stochastic dimensions) while Figure 1(b) is fixed at interpolation level five for increasing stochastic dimensions. Thus, it is not practical to start the experiment with a high value for interpolation level. Typically, scientists start the experiment with a low level of interpolation. If the reliability of results does not meet a given threshold established by the scientist, he/she runs the experiment again with a higher level of interpolation. This change may be done at runtime if the experiment is modeled as a dynamic workflow.

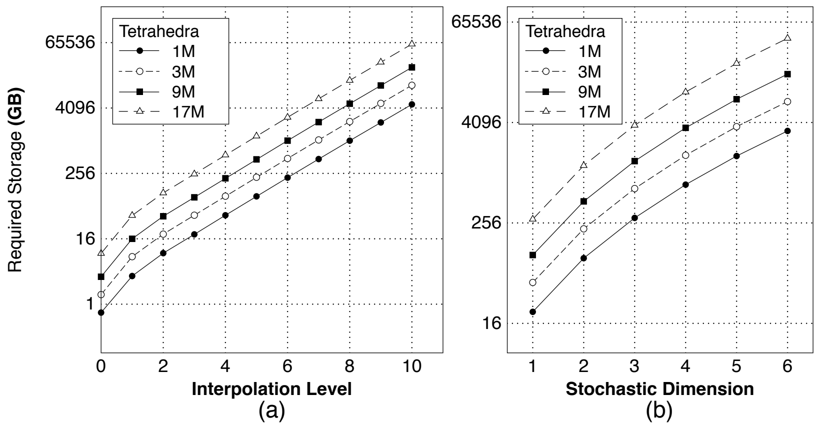

Figure 1. How data grows with the interpolation level increase (a) and problem dimension (b)

Figure 2 shows how scientists typically perform an UQ experiment modeled as a high-level dynamic workflow. They start creating a mesh based on geometry data that represent a physical domain (Activity 1). Activity 2 creates the set of trials that are explored. It proceeds by replicating the mesh, but initializes each replica with a different set of initial conditions. The amount of replicas is related to the initial interpolation level. For example, considering two variables with uncertainty (*i.e.* two dimensions), five interpolation levels require 145 replicas (called *collocation points*) while six interpolation levels require 321 collocation points. The initial conditions vary following a stochastic distribution. Activity 3 partitions each initialized mesh to be consumed by the parallel solver in Activity 4. There is one solver execution for each input mesh. The solver simulates a given phenomena and produces several time-snapshots outputs.

Activity 5 merges the output of the solver, thus creating a single result file for the output. Activity 6 extracts from each collocation point the kinetic energy for all time steps (arranged as a vector). Thus, the output of activity 6 contains a vector for each collocation point. Activity 7 calculates the weighted average, considering all the obtained vectors. Activity 8 evaluates the convergence criteria given by the total kinetic energy using the difference between the norms from the current average (expected value) vector and the vector from the last interpolation level (if there is one). If the difference of the norms is smaller than the threshold, the iteration stops because the current interpolation level is enough for reliability. Otherwise, Activity 9 increases the interpolation level, creating additional collocation points. Based on the new collocation points, Activity 2 initializes new mesh replicas and the iteration continues.

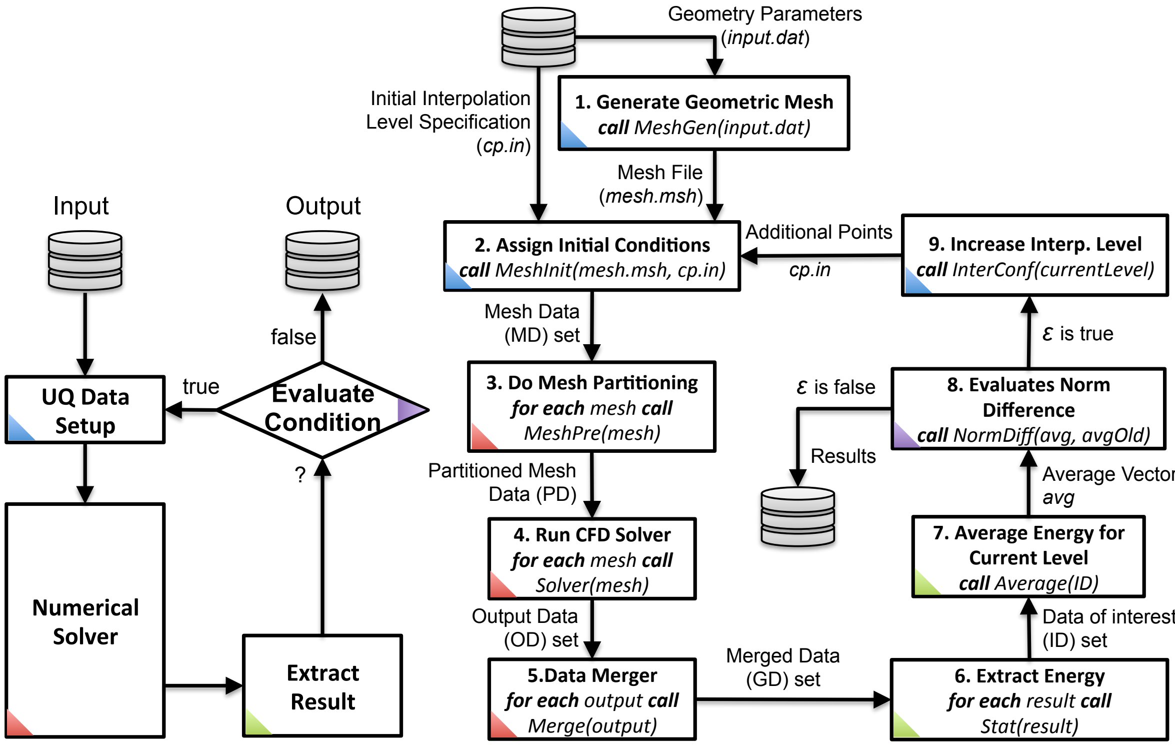

Figure 2. Iterative UQ workflow

As test case for our iterative UQ workflow, we present a stochastic analysis of turbidity currents, a phenomenon of high interest in subsea geology [19]. Turbidity currents are gravity flows, where suspended sediments are transported in a turbulence driven process. This is an important mechanism for the formation of oil-reservoirs over geological time. Turbidity current experiments may be improved considering the stochasticity of fluid properties and boundary and initial conditions. The numerical methods used in the experiments of the present work are described in detail in [15].

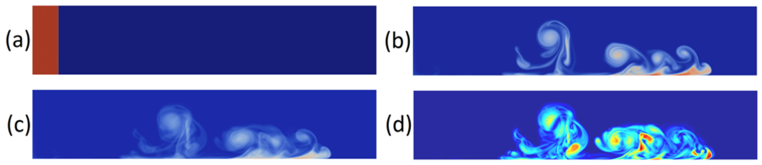

Figure 3. Dataflow steering stages

With this aim, a typical benchmark model used by the scientist is the lock-exchange configuration. In this model, two immiscible fluids with different densities are confined in a rectangular three-dimensional box. At the beginning, they are at rest and separated by a solid wall as shown in Figure 3(a). In this case, the initial condition of the heavy fluid is considered heterogeneous and represented through a distribution that depends on two stochastic parameters. In Figure 3(b), the user visualizes partial results and sees a sediment concentration distribution at the domain midplane for one realization. Figure 3(c) and (d) show the statistical moments, mean and standard deviation respectively, obtained using the SGSC method.

3. Data-Centric Iteration Algebra

In order to support dynamic workflows, we discuss the minimum set of characteristics that allows for iterations on the workflow algebra specification.

3.1 Workflow Algebra Without Iteration

Our data-centric algebra is based on [25] where data is uniformly represented as relations. The relations may store parameters (64bit floating point numbers or strings) and references to files. The definition of a relation includes a schema that describes the type of data that is stored in the relation. An activity Y of the workflow is associated with an algebraic operator ϕ, additional attributes ω of the operator, the set of input relations R and the set of output relations T. We represent an algebraic workflow expression as $T \leftarrow \phi(Y, \omega, R)$. The definition of activity Y includes the relation schemas and the program that is actually instanced during workflow execution. The operator ϕ associated with the activity Y indicates how Y consumes and produces data. For example, if ϕ is a *Map* operator, Y produces a single output tuple in T for each input tuple from R. If it is a *Reduce*, it produces a single output tuple for each group of input tuples grouped by a set g_A of attributes. A *SplitMap* produces several tuples for each input and is typically associated with fragmentation activities. An activity associated with *Filter* decides whether each input tuple should produce an output. In [25], a *Filter* can produce an output tuple or not. However, in this work, we use a functional semantics, so that, if the input tuple does not match the *Filter* criteria, a *null* tuple is produced. To use relational algebra expressions within the workflow, the queries should be associated with the operator *SRQuery*, for queries over a single relation, or with the operator *MRQuery*, for queries with multiple relations (like joins). Again, if the query produces no output, it returns a *null* tuple. The original algebra on [25] presents all the aforementioned operators. However, it does not support iteration or runtime adaptation in workflow configuration based on user input.

3.2 Iterative Constructs for the Workflow Algebra

Iteration support has been proposed in programming languages, data-centric languages [6] and workflows [18,30]. According to [10], there are three kinds of iteration in scientific workflows: (i) *counting loops without dependencies* between iterations, also known as parameter sweep, where a given computation is applied for each input data of the loop; (ii) *counting loops with dependencies* between iterations where the data produced at iteration k is used at k+1 but the number of iterations is known prior to the loop execution; and (iii) *conditional loops*, also known as non-counting iteration, or sequential iteration, where the loop stops after a given condition is met (*e.g.*, the "while…do" loop structure). We propose a fourth kind of iteration for workflows, *dynamic loops*, a particular type of conditional loops where the condition of the loop and the data being processed in the loop may be adapted during the execution.

We follow the iterative constructs of [22] to build a data-centric iterative approach, inspired by functional database programming languages [6]. In order to support dynamic loops, we combine the data-centric iterative approach with a dynamic execution model, which we will discuss in Section 4. To support loops, the workflow algebra must satisfy the following conditions: (1) a loop can start, (2) the loop can iterate and (3) the loop can stop. We allow the workflow to be a directed graph, which may have cycles, by means of a new operator called *Evaluate*. We also introduce the concept of *atom* and define the concepts of *activation* and *workflow* using *atoms*. Finally we define the concept of iteration *lineage,* which is key to support condition (3) and the adaptation based on user input.

The workflow algebra is data-driven [17], i.e. an activity of the workflow fires as soon as its inputs are available. However, since the input of an activity is a relation (a set of tuples), the required input to fire an activity may be unclear. Depending on the operator associated with an activity, the way in which the activity consumes input data differs. For example, if *Map* is associated with the activity, it consumes one tuple independently. Differently, if the *Reduce* is associated with the activity, several tuples are consumed at once. The input relation may also contain several tuples, so multiple instances of the activity (activations) can fire in parallel. In order to formalize the finest unit of data that is necessary to run an instance of activity, we define the concept of data *atom*.

Definition 1. A data *atom* a_i of an input relation R is a horizontal fragment of R such that $a_i \leftarrow \sigma_{F_i}(R)$ for $1 < i < w$, where F is the fragmentation predicate or a minterm predicate [26] and w is the amount of fragments. The *Atom* has the following properties: (i) it cannot be further decomposed in the context of an activity; (ii) for the operators *Map*, *SplitMap* and *Filter*, each atom a_i is a single tuple of the relation R; (iii) for the operator *Reduce*, each a_i may contain a set of tuples according to the set of aggregation attributes in G_A; (iv) for the operators *SRQuery* and *MRQuery,* the *atom* depends on the query associated with the operators; (v) the set of input atoms of a relation R is denoted as *Atoms*(R); (vi) The union of the w atoms $a_i \in$ *Atoms*(R) is R itself, i.e., $\bigcup_{i=1}^{w} a_i = R$.

Property (iv) of Definition 1 does not give the details about the atom for input relations of activities associated with *SRQuery* and *MRQuery* operators. It is hard to predict the atom in this case because the queries of the activities may contain aggregation and joins in their relational algebra expressions. In order to define the atom, it is necessary to decompose the query of the *SRQuery* and *MRQuery* activities to understand how the relation is read.

Definition 2. An activity *activation*, or activation for short, is a self-contained object that holds all information needed, *i.e.*, which program to invoke and which data to access to execute an activity at any core of a computing node in a parallel computer. Activations contain the finest unit of data needed by an activity to execute [4], *i.e.*, an *atom* a_i of the input relation R that enables activity execution. An activation has the following properties: (i) as soon as it is created, an activation is ready to fire; (ii) it consumes an *atom* a_i of tuples; (iii) an activation, once fired, always produces tuples; (iv) its ratio of tuple consumption and production varies according to the operator that is associated with their activities: *Map* (1:1), *SplitMap* (1:m), *Reduce* (n:1), *Filter* (1:1), *SRQuery* and *MRQuery* (n:m); (v) the output relation of an activity is composed by the set of tuples produced by all its activations.

Definition 3. A data-centric scientific workflow is a data-driven dataflow [22], which is made up only of activities associated with algebraic operators, additional attributes of the operator and their respective input and output relations. An activity receives an input relation, which is fragmented into atoms according to the algebraic operator associated to that activity. The activity generates one activation to process each atom. The union of the results of all activations becomes the output relation of the activity. Furthermore, a data-centric scientific workflow is characterized by the following properties: (i) data flows through the relations of the workflow and is transformed by the activities of the workflow; (ii) the input relation of an activity can be the output relation of another activity or another relation defined prior to the execution of the workflow; (iii) an activity can have one or more inputs relations, into which its inputs are inserted; (iv) an activity can have one or more output relations, to which its outputs are inserted; (v) an activity fires, *i.e.*, generates activations, as long as there is at least one *atom* available in its input relations that was not consumed before; (vi) once an activity has fired, it is necessary to receive new *atoms* to fire again; (vii) the output relation of an activity is the union of the output of all its activations; (viii) an activity, once fired, always produces some data through its output relation; (ix) if two or more activities receive the same output relation of another activity, then each activity receives a separate identical instance of such relation as input, and produces their *atoms* according to their associated algebraic operator.

Based on property (ii) of Definition 3, we say there is a data dependency $S \overset{dep}{\longleftarrow} R$ between two relations R and S, *i.e.*, S depends on R, if and only if there is an algebraic expression $S \leftarrow \alpha(R)$ that consumes R to produce S. Data dependency is transitive, thus $\{S \overset{dep}{\longleftarrow} R, T \overset{dep}{\longleftarrow} S\} \vDash T \overset{dep}{\longleftarrow} R$. Figure 4 illustrates the transitiveness example. To illustrate, consider there is an algebraic expression $T \leftarrow \beta(S)$. Since $S \leftarrow \rho(R)$, we can substitute it in the former $T \leftarrow \beta(\rho(R))$. Thus, considering γ as the composition of β and ρ, we can say that $T \leftarrow \gamma(R)$ and consequently $T \overset{dep}{\longleftarrow} R$.

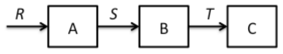

Figure 4. Data dependency example: *T* depends on *S* that depends on *R*

Definition 4. A data-centric scientific workflow subgraph is said to be iterative if at least one activity A exists in the subgraph such that the input of A depends on the output of an activity B and the input of B depends on the output of A.

The definition is straightforward and assumes a simple scenario like in Figure 5.

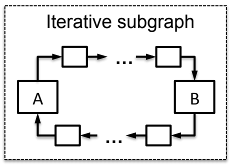

Figure 5. A simple iterative subgraph

Based on Definitions 1-4, we can now discuss how we can guarantee the three conditions ((1) the loop can start, (2) the loop can iterate and (3) the loop can stop) in the workflow algebra.

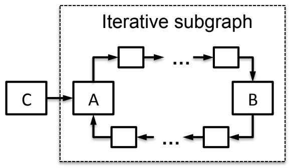

Figure 6. Iterative subgraph starting with activity C.

3.2.1 A Loop can Start

Theorem 1. In a data-centric scientific workflow, it is only possible to start a loop if at least one activity in the iterative subgraph is allowed to receive more than one relation for the same input.

Theorem 1 can be trivially proved by contradiction. Intuitively, in Figure 5, no activity is allowed to receive more than one input. Since the input of *A* depends on the output of *B* and the input of *B* depends on the output of *A*, if *A* is not able to receive an initial input, it cannot ever fire, and this is true for any pair of activities in the subgraph. In the modified example of Figure 6, *A* is able to receive an initial input from C. This causes activity *A* to fire, as well as, eventually, activity *B*. Since the input of *A* derives from the output of *B*, *A* fires again and the iteration process can go on.

Theorem 1 says that two input relations are connected to the same input for a given activity. It is different, for example, from an *MRQuery* activity that can receive more than one relation but each relation is a different input. In the case pointed by Theorem 1, an activity expects to receive a single input, but this input can come as an initial input relation or an input relation from the iterative subgraph. This means that both relations must have a schema that satisfies the activity's needs. Theorem 1 guarantees condition (1) that iterations can start on an algebraic workflow, as long as an activity of the iterative subgraph receives two relations for the same input. In the UQ workflow, for example, Activity 2 receives two equivalent inputs (*cp.in*) with the specification of the interpolation level. One input comes from the initial input and another comes from the iterative subgraph.

3.2.2 A Loop can Iterate

Theorem 2. In a data-centric scientific workflow, iterations are always endless.

Theorem 2 guarantees condition (2) that, once started, a loop iterates. Its proof is intuitive if we consider Figure 6, for instance. At the beginning, Activity *A* fires because it receives initial input data from Activity *C*. Since an activity always emits some data when it fires (property (viii) of Definition 3), activity *B*, which may have received input data, fires and produces output data. Since the input of activity *A* depends on the output of activity *B*, activity *A* fires again and the process never ends.

According to Mosconi *et al.* [22], an iteration can have an end if: (a) there is a mechanism that prevents at least one activity in the iterated subgraph from firing if some conditions are satisfied, or (b) at least one activity in the iterated subgraph is prevented to produce all its outputs; if some conditions are satisfied when firing. We believe the second option best fits scientific workflows. Case (a) prevents an activity *Y* of the workflow from firing. Thus, we would need an enabling signal (token) dataflow [22] to fire the remaining activities of which input data depends on the output of *Y*. In our relational approach, we only use data *atoms* as signals to fire an activity. Therefore, we believe case (b) is better for the iterative workflow algebra because if an activity of the workflow has two output relations and decides whether to insert into one or another, it is just conducting the dataflow based on an internal evaluation that is done when the activity fires. In the UQ workflow, for example, Activity 8 has two outputs and decides, based on a threshold if the loop should continue.

3.2.3 The Evaluate Operator

Current algebraic operators, *e.g.* Map, Reduce, SplitMap and Filter, do not allow for the definition of either two different output relations for the same activity or two relations for the same input. Thus, we define a new algebraic operator to support Theorems 1 and 2.

Definition 5. *Evaluate* is an algebraic operator that is associated with activities that have a Boolean behavior, *i.e.*, the results of an activation of an activity associated with *Evaluate* indicates whether a condition is *true* or *false*. *Evaluate* has the following properties: (i) An activity associated with *Evaluate* accepts two relations R_{init} and R_{loop} for the same input $R_E = R_{init} \cup R_{loop}$ where $schema(R_{loop}) \supseteq schema(R_{init})$; (ii) the default *atom* for an activity associated with *Evaluate* is a single tuple from relation R_E; (iii) if a fragmentation predicate or minterm predicate F is given as an operand to Evaluate, the atom $a_i \leftarrow \sigma_{F_i}(R_E)$ for $1 < i < w$; (iv) an activity associated with *Evaluate* produces two output relations T_{true} and T_{false} such that $schema(T_{true}) \cup schema(T_{false}) \subseteq schema(R_{loop})$; (v) Evaluate requires the definition of an evaluation function ε to decide whether the output of an activation is *true* or *false*; (vi) If ε evaluates to *true*, the output is written in relation T_{true}. Otherwise it is written in T_{false}; (vii) $T_{true} \cap T_{false} = \emptyset$, which means that only one output relation is actually written by *Evaluate* per activation.

We represent an activity Y associated with Evaluate as:

$$\{T_{true}, T_{false}\} \leftarrow Evaluate(Y, \varepsilon, R_E)$$

The additional and optional operand F may also be necessary because an evaluation can consider a single tuple but might also need to evaluate a set of tuples altogether. Thus, the ratio of data consumed per data produced in Evaluate is *n*:*m*. Figure 7 shows an example where Activity E is associated with *Evaluate*. Input data from input relation R_{init} are evaluated by Activity E and, if it evaluates to true, output data is emitted to T_{true} and fires the iterative subgraph. Otherwise, the output data is emitted to T_{false} and fires the exit subgraph. By analogy with the *whiledo*(⟨*loop_fcn*⟩, ⟨*exit_fcn*⟩, ⟨*start_action*⟩) functional operator, in Figure 7, the iterative subgrah would be the *loop_fcn*, the exit subgraph would represent the *exit_fcn* and the activity itself associated with the *Evaluate* operator would be the *start_action*. The *Evaluate* operator can support activities such as Activity 8 of the UQ workflow in Figure 2.

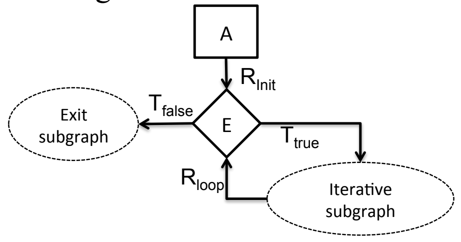

Figure 7. An iterative workflow

3.2.4 Iteration Lineage

The iterative subgraph may contains different types of activities and the aspect of data that return to Activity E through R_{loop} may be different after each evaluation. New atoms could have been produced in the iterative subgraph while others are changed on the dataflow. The evaluation activity itself removes atoms from the iteration when it evaluates to false, directing the output to T_{false}. The only thing all atoms share while they are inside the looped subgraph in the workflow is that they are all derived from the initial input atom $a_i \in Atoms(R_{init})$. This allows keeping track of the atoms that are still flowing in the iterative subgraph following the atom that originated their iteration. Thus, we now define that all atoms that originate from a given ancestor are marked to belong to a given *Lineage* as explained below. Although the concept of lineage is inspired by provenance, in this work we use lineage as a more precise workflow concept regarding the data atoms and their ancestors inside an iterative subgraph.

Definition 6. A *lineage* $\lambda_k(a_i) \mid a_i \in Atoms(R_{init})$ is a possibly empty set of atoms originated from atom a_i after k positive evaluations during an iteration. Lineage has the following properties: (i) there is always one lineage for each atom of R_{init}; (ii) if an atom is evaluated to false, it leaves the lineage, because, based on property (vi) of Definition 5, the output obtained with such atom yields T_{false} and gets out of the looped subgraph; (iii)$\bigcup_{i=1}^{w} \boldsymbol{\lambda_k(a_i)} = \boldsymbol{R_E^k}$, which means that the input for the k-th evaluation, denoted by $\boldsymbol{R_E^k}$ is always the union of all current lineages; (iv) $\lambda_0(a_i)=a_i$; which means that prior to the first evaluation, the lineage of an atom is the atom itself, thus $\boldsymbol{R_E^0} = \boldsymbol{R_{init}}$;(v) if $\lambda_k(a_i) = \varnothing$ then $\lambda_{k+1}(a_i) = \varnothing$, which means that if a lineage is empty on a given evaluation, on the next one it will remain empty.

3.2.5 A Loop can Stop

Theorem 3. If an activity associated with *Evaluate* heads a looped subgraph that receives a finite number of *atoms* on its input relation R_{init} and can evaluate to false for each lineage $\lambda_k(a_i) \mid a_i \in Atoms(R_{init})$, the iteration always has an end.

Proof. The theorem says that after a certain number of k evaluations, the activity evaluates to false for a lineage $\lambda_k(a_i) \mid a_i \in Atoms(R_{init})$. Thus, based on property (ii) of Definition 6, the theorem assumes that after k evaluations a given $\lambda_k(a_i)$ must be empty, *i.e.*, all the atoms that once belonged to $\lambda_k(a_i)$ have been already evaluated as false. Based on this assumption, we can say that there is a function $f(a_i) = b_i$, $a_i \in Atoms(R_{init})$ that for each *atom* a_i, returns a finite and minimum number b_i of evaluations so that $\lambda_{b_i}(a_i) = \emptyset$. We can also say that there is always a maximum value $b_{max} = \max(f(a_1), \ldots, f(a_w))$ such that $\lambda_{b_{max}}(a_i) = \emptyset \; \forall \; a_i \in Atoms(R_{init})$. Using property (iii) of Definition 6, $\bigcup_{i=1}^{w} \lambda_{b_{max}}(a_i) = R_E^{b_{max}} = \emptyset$. Thus, there is always a finite number of evaluations $k=b_{max}$ after which the input relation R_E of the activity associated with Evaluate is empty. Thus, according to property (vi) of Definition 3 and secured by property (v) of Definition 6, after $k=b_{max}$ evaluations, the iteration has an end. ■

Theorem 3 guarantees condition (3) that the loop can stop, which completes the constructs needed for dynamic loops in the workflow algebra. Thus, based on Definitions 5 and 6 and supported by Theorems 1, 2 and 3, we define the concept of dynamic loops.

3.2.6 Dynamic Loops

Definition 7. *Dynamic loops* are iterative subgraphs headed by an activity associated with *Evaluate*, in which, for each input data atom $a_i \in Atoms(R_{init})$, there is a lineage $\lambda_k(a_i)$ for k iterations. Dynamic loops have the following properties: (i) new atoms can be inserted in $Atoms(R_{init})$ during the execution of the loop; (ii) atoms in $Atoms(R_{init})$ can also be updated or deleted during the execution; (iii) additional operands of the activities inside the iterative subgraph can be updated during the execution of the loop, for instance the evaluation function ε of an activity associated to *Evaluate*; (iv) Data atoms in $\lambda_k(a_i) \; \forall \; a_i \in Atoms(R_{init})$, for every k, are subject to querying and visualization.

Dynamic loops require an execution model that supports both the *Evaluate* operator and the lineage concept. Besides, there should be means to adapt the workflow based on user input to enable properties (i), (ii) and (ii) of Definition 7. Section 4 describe our execution model for dynamic loops.

4. Execution Model

Our execution model for dynamic loops is designed for parallel execution on clusters with lots of multicore execution nodes. Data-centric iteration is a good basis for dynamic loops because each new data inserted in the input relation generates new lineages in the loop. The data remains in the loop until the condition evaluation returns false. Thus, one aspect of the execution model is related to loop condition evaluation during activity activation. We extend the activation concept to accommodate the needs of the evaluation step of the

Evaluate operator. Dynamic loops are guided by lineage, when users adapt the workflow and change data, they are affecting lineage and, consequently, the dataflow. Thus, another important aspect of our execution model is how to handle the runtime adaptations based on scientists input. We propose two algorithms to handle adaptation during workflow execution. The workflow adaptation impact on activation scheduling, thus we also discuss the loop response to change and how activations are dynamically scheduled to the execution nodes.

4.1 Dynamic Loop Condition Evaluation

Activations associated with the *Evaluate* operator need to evaluate an output to true or false. Thus, we define a specialization of activation to support the evaluation step using the evaluation function ε. Activations follow a three-step execution procedure: *input instrumentation*, *program invocation* and *output extraction* [25]. Input instrumentation extracts the values of the input *atom* and prepares them for program invocation, setting the values for input parameters according to the expected data type. Program invocation dispatches and monitors the actual program execution. Output extraction gathers values from the output of the invoked program and builds the activation's output. Based on properties (ii) and (iii) of Definition 2, each activation x_i consumes an input atom a_i and produces an output. Based on property (v) of Definition 2, the output is inserted in the output relation of the activity Y_i associated with x_i. However, according to properties (v) and (vi) of Definition 5, if the activation x_i belongs to an activity associated with *Evaluate*, there are two possible outputs (T_{True} and T_{False}) for the output and the evaluation function ε is used to do the evaluation and place the result at the correct output.

The *Evaluate activation* inherits all properties of the original activation. Additionally, it has a fourth step called *evaluation*, which marks the output atom with a Boolean flag obtained with ε. The evaluation function ε should be provided prior to the workflow execution. The function can be a logic expression or a custom program or script developed by the scientist. For example, consider that the output relations T_{True} and T_{False} have schemas $\mathcal{T}_{True}$ and $\mathcal{T}_{False}$, respectively. The output has a schema $\mathcal{Q} = \{f_j, \dots, f_m\} \subseteq \mathcal{T}_{True} \cup \mathcal{T}_{False}$. Then:

$$\varepsilon = \bigwedge_{p_i \in P} p_i \ , P = \{p_i \mid p_i = f_j \ \theta \ Value\} \ 1 \le i \le n$$

where $\theta \in \{ =, <, \neq, \leq, >, \geq \}$ and $Value$ is comparable with f_j. Since the result of ε is true or false, the evaluation step marks the output with the correct flag. If ε is a user program or script, it must be able to read the output of the activation and produce a Boolean result. Based on the Boolean mark of the output, it is possible to insert it in the correct output relation. For example, in Activity 8 of the UQ workflow, considering that its output relation contains a field named *normDiff* that stores the norm difference between the levels, we may say that $\varepsilon = (normDiff \geq 0.001)$.

4.2 User Input Adapting the Loop

Dynamic steering of workflows is strongly related to how scientists interact with their data. This data may be the input datasets they are analyzing, like the specification of the colocation points in the UQ analysis. There are also parameters configured in the workflow that typically needs to be adjusted, like fluid viscosity and time step size in the UQ workflow. Provenance data is also subject to analysis to help scientists to take decisions regarding the workflow. To steer the workflow execution, scientists monitor and analyze data as they are made available and decide to interfere or not with the workflow at runtime. We consider that the dynamic steering happens inside a dynamic loop. This does not restrict our model since any workflow can be enclosed by a dynamic loop to enable dynamic steering. The dynamic loop is headed by an activity associated with the *Evaluate* operator, which has two input relations, so it generates activations when there is a new atom available in R_{init} or R_{loop}. If the atom a_i is from R_{init}, we say $a_i = \alpha_i$ is an α-atom, and it originates a new lineage $\lambda_0(\alpha_i)$. If the atom a_i is from R_{loop}, it already belongs to a lineage $\lambda_k(\alpha_i)$ where α_i is the k-th ancestor that originated the lineage at first place. We consider two types of runtime adaptation in the workflow: (s^{α}) changes made in an α-atom; and (s^{ω}) changes made in the set ω of attributes of the activities. The s^{α} type happens when scientists change the input dataset or modify its content, like changing the parameter combinations they are exploring. In the UQ workflow, for example, the scientist may want to change a parameter such as the simulation time step or the fluid viscosity. The s^{ω} type occurs when scientists want to adjust an attribute of the operator associated with the activity, like the grouping attribute of an activity associated with *Reduce* or the definition of the evaluation function (ε) of an activity associated with *Evaluate*. In the UQ workflow, scientists may want to change the evaluation function ε to increase the threshold regarding the difference of vector norms. Adaptations s^{α} affect only the updated data while adaptations s^{ω} affect the entire dataflow.

We denote by $s^{\alpha} < \alpha_{old}, \alpha_{new}, k_s, \delta >$ a runtime adaptation that changes an α-atom α_{old} to α_{new} on iteration k_s. The optional attribute *depth* ($\delta \mid \delta > 0$) indicates the number of iterations the change should be maintained for. The default value for δ is infinite ($\delta = +\infty$), which means that the change is permanent. All adaptations are asynchronous events kept in a list *S*. Algorithm 1 shows how s^{α} events are handled in the execution model. We consider that the iteration *k* is never incremented before all $s^{\alpha} \in S$ that happened on that iteration have been treated. If s^{α} happens on current iteration ($k = k_s$), the algorithm suspends the lineage of α_{old} (line 4). When a lineage is suspended, all running activations with atoms corresponding to that lineage must finish their executions. However, new activations from this lineage are no longer created. If the lineage of the α-atom α_{new} does not already exist, the algorithm creates a new lineage (line 9). The algorithm also checks if the depth (δ) is using the default infinity value. If so, it removes the event from the list because it is considered permanent (line 11). For provenance matters, a list of all occurred adaptations is kept. If the change is not permanent, after δ iterations the algorithm reverses the adaptation, suspending the α_{new} lineage (line 14) and restarting the α_{old} lineage (line 16). This feature is important because scientists may not be certain if a given change improves the results. Thus, they try the change for several iterations and compare the results. If they like the change, they can make it permanent. If the process had already been reversed, they just need to do a new adaptation and the algorithm will restart the α_{new} lineage. The s^{α} adaptations can also be used to add new input data in the workflow if a *null* value for α_{old} is used.

Algorithm 1 **Alfa Steering**

```
1   for each $s^{\alpha}$ in $S$
2       if $k = k_s$ then
3           call getLineage with $\alpha_{old}$, $k$ returning $\lambda$
4           set status of $\lambda$ to suspended
5           if lineage ($\alpha_{new}$, $k$) exists then
6               call getLineage with $\alpha_{new}$, $k$ returning $\lambda$
7               set status of $\lambda$ to running
8           else
9               call addLineage with $\alpha_{new}$, $k$
10          if $\delta = +\infty$ then
11              call removeFromS with $s^{\alpha}$
12      if $k = k_s + |\delta|$ then
13          call getLineage with $\alpha_{new}$, $k$ returning $\lambda$
14          set status of $\lambda$ to suspended
15          call getLineage with $\alpha_{old}$, $k_s$ returning $\lambda$
16          set status of $\lambda$ to running
17          call removeFromS with $s^{\alpha}$
```

We denote by $s^{\omega} < \omega_{old}, \omega_{new}, k_s, \delta >$ a runtime adaptation that changes the set of attributes ω_{old} of the workflow W to a new set ω_{new} during iteration k_s. A ω set may contain several attributes of the activities of the workflow and a change in any attribute of ω produces a s^{ω} adaptation with a new ω. By default, the value of the depth δ is $+\infty$, which means that the s^{ω} adaptation is permanent. However, differently from s^{α} adaptations, in s^{ω} δ can have negative values (backward adaptation) or positive values (forward adaptation). Since s^{ω} adaptations affect the configuration of the workflow, scientists may want to explore the new configuration starting from a past iteration. Thus, when $\delta < 0$, the adaptation does a rollback in the workflow execution. Because it is important to keep the record of all the explorations of the workflow, rollbacks actually create branches in lineages. Algorithm 2 describes how s^{ω} events are treated. When s^{ω} happens ($k = k_s$), the algorithm checks if there is already a suspended branch with the given configuration ω_{new} for the workflow W (line 4). If a branch with the new required ω_{new} configuration already exists but is suspended, it becomes the active branch (line 6) by the *switchBranch* subroutine (line 25). Otherwise, it changes the configuration of the workflow (line 8) and creates a new branch. If s^{ω} is a backward adaptation (line 9), it creates the new branch based on lineages of the $k_{rollback}$ past iteration (line 10). If s^{ω} is a forward adaptation, it creates a branch based on current lineages. Forward adaptations with non-infinite δ must be reversed after $k_s + \delta$ (line 15). The algorithm does this by calling the subroutine *switchBranch* using ω_{new} as the current configuration and ω_{old} as the new one.

Algorithm 2 **Omega Steering**

```
1   for each $s^{\omega}$ in $S$
2       if $\delta = 0$ set $\delta$ to $+\infty$ end if
3       if $k = k_s$ then
4           if $\lambda_{suspended}[\omega_{new}]$ of $W$ exists then
5               obtain $\omega$ from $W$
6               call switchBranch with $\omega$, $\omega_{new}$, $k$
7           else
8               set $\omega$ of $W$ to $\omega_{new}$
9               if $\delta < 0$ then
10                  set $k_{rollback}$ to $k + \delta$
11                  call branch with $\omega_{old}$, $k_{rollback}$
12                  call removeFromS with $s^{\omega}$
13              else
14                  call branch with $\omega_{old}$, $k$
15      if $k = k_s + \delta$ then
16          call switchBranch with $\omega_{new}$, $\omega_{old}$, $k$
17          call removeFromS with $s^{\omega}$
18  subroutine branch ($\omega_{old}, k$)
19      call getLineages with $k$ returning lineages
20      set $\lambda_{suspended}[\omega_{old}]$ of $W$ to lineages
21      for each $\lambda$ in lineages
22          set status of $\lambda$ to suspended
23          call copyLineage with $\lambda$ returning $\lambda_{new}$
24          set status of $\lambda_{new}$ to running
25  subroutine switchBranch ($\omega_{current}, \omega_{new}, k$)
26      call getLineages with $k$ returning lineages
27      set $\lambda_{suspended}[\omega_{current}]$ of $W$ to lineages
28      for each $\lambda$ in lineages
29          set status of $\lambda$ to suspended
30      obtain lineages from $\lambda_{suspended}[\omega_{new}]$ of $W$
31      for each $\lambda$ in lineages
32          set status of $\lambda$ to running
```

```
33      set $\lambda_{suspended}[\omega_{new}]$ to null
```
These two algorithms are designed to support dynamic steering. They are also able to work together so both types of adaptations can be done during execution. Scientists can branch the workflow execution to a new configuration, then add new data to be explored, and later, do new branches or go back to a previous configuration. The combination of the data-centric iterations with our execution model provides support for dynamic loops, which we believe are essential for dynamic workflows.

4.3 Loop Response to Change

Our execution model is designed to execute an instance of an activity for each of its input data atoms using the multicore execution nodes of the parallel computer. Based on property (v) of Definition 3, as long as there are atoms available in the input relation of an activity, new activations are produced. An activation represents an execution instance of an activity, and several activations can be executed in parallel. Since both the conditional loop evaluation and the adaptations impact on the activation level, we changed the execution model to associate the data dependency relationships available in the lineage with the activations. For each activation there is a data atom, and each atom is associated to a lineage. As activations are executed, they produce their output data, which are inserted in the lineage as new atoms. The new atoms become available for the execution of the next activity and this process goes on until lineage is empty or the workflow ends. If the user adapts the workflow, affecting any lineage, the execution model knows exactly which activations need to be executed again. The execution model has two scheduling policies, which dictates how the activations x_i are scheduled to the execution nodes. The synchronous scheduling policy schedules the activations of a given activity if and only if all the activations of the precedent activities have already finished, i.e., there is synchronization between the activities. For instance, the operator *Reduce* requires the use of synchronous scheduling because it aggregates data on a customized manner. The asynchronous policy schedules pipelines of activations respecting the data dependency between activations. For instance, the operators *Map* and *Filter* can be part of a pipeline. If scientists adapt the workflow affecting the lineages, the activations that need to be executed again obey the same scheduling policy of previous execution and the data dependency relationships in lineage.

5. Experimental Evaluation

To validate our approach, we implemented the data-centric iteration and the dynamic execution model in Chiron [25]. In order to make Chiron support dynamic loops, we developed the new algebraic operator *Evaluate*, the new *Evaluate activation* and made necessary changes in execution model methods. We also changed the data model of provenance in Chiron to support iteration lineage and we developed a module for the two types of adaptations described in Section 4.3. The adaptations are inserted in the provenance database by an API and are handled by Chiron at runtime using the Alfa and Omega Steering algorithms. Chiron is developed in Java and uses a PostgreSQL database to manage provenance data.

Our experimental evaluation explores the UQ workflow described in Section 2. We execute two versions of the UQ workflow (with and without dynamic loops) and discuss the advantages of the dynamic approach from three different user perspectives: (i) incremental starting from interpolation level 0; (ii) incremental starting from interpolation level 3 (short scenario); and (iii) single-runs at different interpolation levels. Perspectives (i) and (ii) are depicted in Figure 9 while (iii) is presented in Table 1 and Figure 10. We consider that the user performs an adaptation of type s^{ω} during execution. The last analysis evaluates experimentally the performance of the Alfa and Omega algorithms, described in Section 4.2. We designed two versions of the UQ workflow as shown in Figure 8. The complexity that dynamic loops introduce in the workflow design is very small since we just add two activities and reestablish some of the dependency relationships between the activities. The manual version is a non-iterative workflow. The numbers depicted inside the activities are associated with the activities described in Section 2. We also show the algebraic operators associated with each activity. The workflow receives the geometry parameters for the input mesh and a given initial interpolation level. The scientists run the manual workflow for a given level. If the obtained results do not attend the convergence criterion, they run the manual workflow again with a higher interpolation level. The alternative version (dynamic) uses a dynamic loop to explore the dynamic characteristics of the UQ analysis. It uses the *Evaluate* operator associated with an activity to evaluate the convergence criterion. For instance, for the UQ workflow, this criterion is the difference between the vector norms and should be below a given threshold initially defined as 0.001. With the dynamic workflow, scientists start the workflow with a given interpolation level and the workflow automatically increases the level, if necessary. If they need to change the convergence criterion (i.e., the evaluation function ε), they perform an adaptation s^{ω} and the dynamic loop adapts itself.

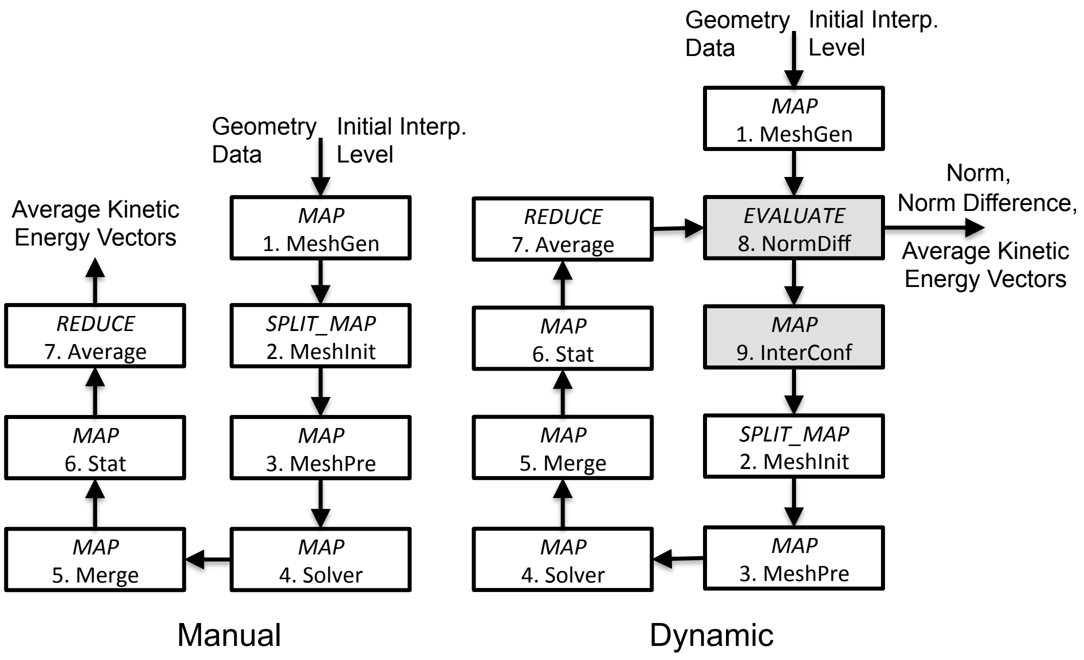

Figure 8. Manual Non-iterative UQ workflow and Dynamic Iterative UQ workflow

Both workflows were executed on a 640-core cluster with Intel Xeon 5640 (2.67GHz) processors, with 1.28 terabytes of distributed memory and 72 terabytes of shared storage. The workflows are explored in two scenarios: complete and short. In the complete scenario, scientists run the UQ analysis starting at interpolation level 0. In the short scenario, prior to the execution of the UQ workflow, the scientist queries provenance data of past executions and concludes that the interpolation level 3 is a better starting level, because the reliability of the results below level 3 is mostly too low for similar execution data. We present the results with the short scenario because in the long term, it is more appropriate. In both cases, the convergence criterion *c* is initially set up as $c = normDiff < 0.001$ (thus $\varepsilon = \neg c$). This threshold may require the experiment to run until interpolation levels 6 or 7. Nevertheless, analyzing the results of interpolation level 4, scientists decide that they need to change the convergence criterion to $c = normDiff < 0.01$. For the manual approach, nothing changes because the scientist is responsible to calculate the vector norms and control the iterative process. For the dynamic approach, however, changing ε causes an adaptation s^{ω}, which is treated by the Omega algorithm in Chiron.

The lock-exchange benchmark we used with the UQ workflow requires interpolation level 5 (6 iterations) to achieve the specified criterion. Thus, the manual workflow in the complete scenario requires that the scientist analyzes and restarts the workflow five times. The restarting points can be seen in Figure 9; we magnify a restarting point at iteration 5. We consider that the time that scientists spend to deploy the workflow is constant. The dynamic approach requires that the scientist starts the workflow only once. An important aspect of UQ workflow is that several explorations at a given interpolation level can be reused at the next levels. However, since it is labor intensive to configure the next execution of the manual workflow to use previously obtained results, scientists usually opt to run the manual workflow step by step. Using the iterative UQ workflow, it is easier to reuse these explorations because they are already in the workflow lineage and are simply referenced by the workflow engine. Figure 9 shows the result for the complete scenario. The dynamic approach saves more than 2.5 days of execution time. This is expected since the dynamic workflow reuses executions from previous interpolation levels and also saves the workflow deployment time. It may not seem fully fair when we say the manual approach do not reuse data from previous executions. Actually, scientists can manually reuse data as long as they setup the new iteration to do so. However they will necessarily need more time to restart the workflow than the time we are considering in our analysis. Besides, managing the reuse manually is error prone and creates gaps in the experiment provenance, making data analysis harder. We also show, in Figure 9, where the dynamic approach stops if the adaptation s^{ω} is not processed. Processing the adaptation at runtime saved almost 3 days of execution time.

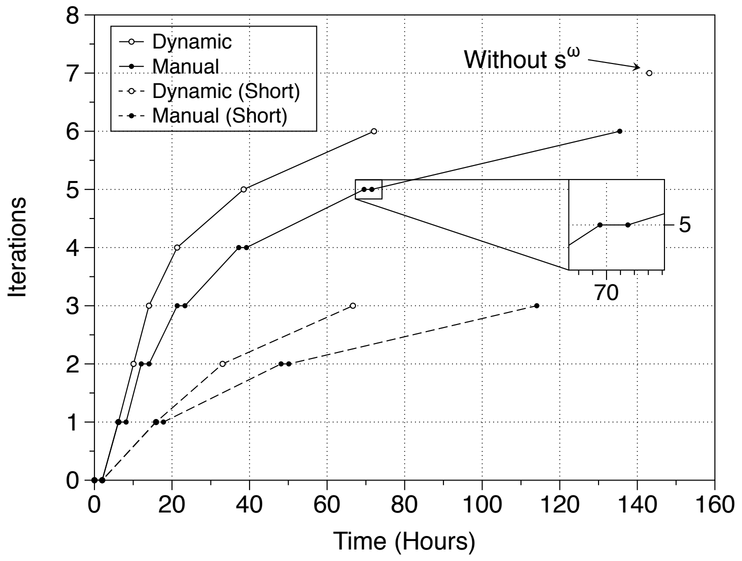

Figure 9. Execution time results for both manual and dynamic workflows

Figure 9 also shows the results for the short scenario. In the short scenario, scientists start both workflows at level 3 and go on until level 5 (3 iterations). This scenario improves execution because it reduces the amount of iterations and, consequently, the number of times that the scientists need to restart the workflow with increasing interpolation levels. Thus, it naturally improves the manual approach. However, the dynamic approach still saves almost 2 days of execution time. The performance differences between the complete and short scenarios are not higher because the collocations points at interpolation levels 0, 1 and 2 are included at level 3 and need to be processed. The difference is that, in the complete scenario, they are processed incrementally, while in the short scenario they are processed as if they belong to level 3. Furthermore, the amount of points at levels 0, 1 and 2 are much lower than at higher levels, since the number of points increases exponentially.

Table 1. Execution time savings of the dynamic approach based on scientist manual choice for the interpolation level.

Manual Choice	**Correct Level**			
	5	*6*	*7*	*8*
5	**-3.63%**	-	-	-
6	51.70%	**-2.05%**	-	-
7	77.67%	52.83%	**-1.11%**	-
8	89.69%	78.22%	53.32%	**-0.59%**

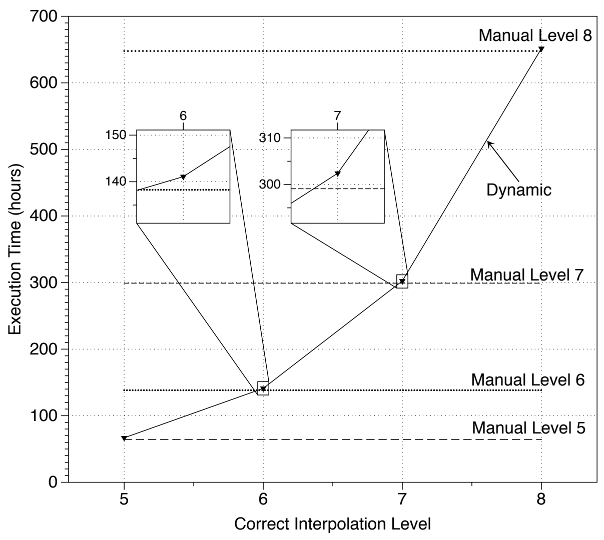

Figure 10. Dynamic approach efficiency compared to the manual approach

The required interpolation level for a problem is unknown prior to the execution of the workflow. Nevertheless, scientists can always choose to start the UQ workflow at higher interpolation levels to improve accuracy in a single run. For example, in the lock-exchange use case, they may have a good guess and start the workflow at level 5. However, choosing higher values for interpolation levels may result in waste of computing resources and storage. For example, instead of starting at level 5, they can choose level 8, which also attends the reliability requirements but may consume much more execution time and storage to finish. In those cases, the dynamic approach should be more efficient. Using results from the UQ workflow, we associate the level scientists choose to start the manual workflow with the correct level that the problem requires and the efficiency of the dynamic workflow starting at level 3. Figure 10 shows the results for the interpolation levels 5 to 8. The horizontal lines are the constant times to run the manual workflow at the levels 5 to 8 in a single run. The dynamic approach adaptively increases execution time according to the correct interpolation level.

If the scientist, in the case he is able to access any prior information, picks an optimal interpolation level to start manually the workflow, that would constitute the best choice. However, the difference in time between the manual and dynamic approach is small and becomes less significant as we increase the interpolation level. In Figure 10, we magnify the results for the levels 6 and 7. At level 6, the dynamic approach is 2.82 hours slower while at level 7 it is 3.32 hours slower. It is important to notice that, although the time difference increases, when we compare it to the total execution time, it becomes less significant as shown in Table 1. The dynamic approach increases execution time by 2.05% at level 6 but increases only by 1.11% at level 7 and 0.59% at level 8. We believe this overhead is acceptable since the dynamic approach may save several hours of execution time if the interpolation level is not correctly chosen for the manual workflow, as shown in Table 1. For example, if the correct interpolation level is 6, but the scientist starts the manual workflow at level 7, the dynamic approach would take 141 hours to execute instead of 300 hours for the manual approach. In this case, the dynamic workflow saves 52.83% of execution time. On the biggest scenario, if the scientist starts the manual workflow at level 8 instead of level 5, the dynamic approach would save 24.2 days (89.69% of execution time). Although this behavior is associated with the uncertainty quantification problem, this situation also happens in other problems that consume data incrementally or problems with adaptivity and self-tuning requirements like non-deterministic optimizations and multiple sequence alignment experiments in bioinformatics.

Another important aspect of dynamic workflows is the performance of the steering algorithms. Scientists interact with the workflow at runtime through an API we developed. The purpose of the API is to configure the steering events and store them properly in the provenance database. Chiron queries the provenance database to select new steering events and treats the event at runtime using either Alfa or Omega algorithms, both described in Section 4.3.To measure the algorithms' performance, we instrumented Chiron to save the time spent to process each steering event. Then, we programmed several batch adaptations to happen at different times of the workflow execution using the API. Figure 11 shows the performance of the Alfa algorithm. Even when Chiron processes 7 events in a row, the time spent with the algorithm is less than 25 microseconds. Since the algorithm runs fast, it is sensitive to small performance fluctuations. This evaluation shows linear performance of the Alfa algorithm.

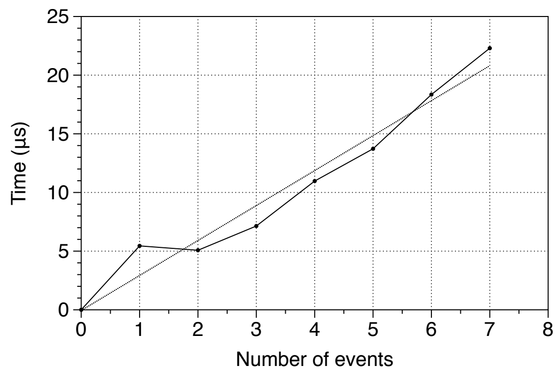

Figure 11. Performance of the Alfa algorithm

The performance of the Omega algorithm is also linear, as shown in Figure 12. Since it takes more time to execute, the performance fluctuations are hard to notice. The time spent to process 7 events in a row is less than a millisecond, which is not significant when we compare to the workflow execution time that may take days or weeks. Furthermore, 7 events s^{ω} in a row is not very realistic because it would require the scientist to change the configuration of the workflow several times in a rush without waiting for the feedback of produced results. Usually, Chiron would process just one s^{ω}event at a time, for each workflow execution context.

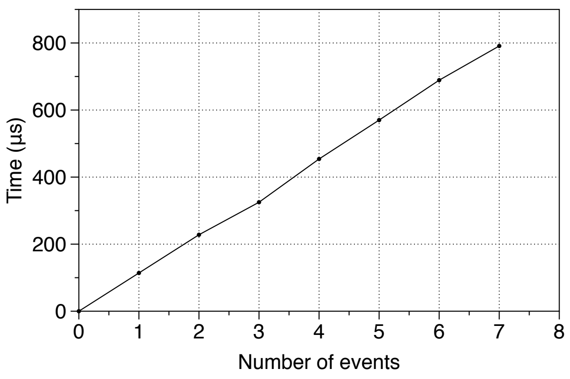

Figure 12. Performance of the Omega Algorithm

Our experimental evaluation shows that dynamic loops improve the execution of experiments that need constant analysis of produced results and adaptation. The dynamic workflow can automatize part of the experimental evaluation and the Alfa and Omega algorithms enable the adaptation of the workflow input data and the workflow configuration based on user input. Both steering algorithms perform well, running in less than one millisecond in the UQ workflow scenario.

6. Related Work

Several SWfMS have long provided support to large-scale scientific experiments. Many of them support counting loops without dependencies in grids and clusters, *e.g.* [1,7,18,21,30], some of which also support counting loops with dependencies [18,21,30]. Counting loops without dependencies can be represented as multiple DAG and counting loops with dependencies can also be linearized as a DAG [18]. For conditional loops, there are dataflow languages [21,30] that supports it out-of-the-box. The main issue of current iterative workflow approaches is that the workflow execution is tightly coupled to the workflow specification model. After the workflow engine has produced the execution plan, scientists cannot change the workflow specification or execution at runtime. This approach limits the continuous adaptation and improvement required by dynamic workflows. Outside the domain of SWfMS, one may establish relationships between our work and recursive or iterative query processing with dynamic query optimization [16]. However, after the query is submitted to execution, a change on data at runtime, e.g. a value on a selection predicate, is not supported.

There are MapReduce approaches that also provide iterative support. Twister [9] and HaLoop [5], for instance, are iterative approaches for MapReduce where users can explore their map and reduce functions iteratively until a given condition is met. Stratosphere [11] has iterative support for dataflows. It explores the content of the programmed functions using the concept of microsteps and optimizes execution. Both Twister, HaLoop and Stratosphere [11] focus on the execution of the iterative process as part of an algorithm implemented in MapReduce, like graph searches, data clustering and page ranking. They change the programmed function to include the microsteps. Differently, our approach is designed for the higher-level experimental process, where the algorithms and other scientific applications are treated as black boxes in the workflow. These applications can be legacy/complex code or very optimized solvers, which are hard to re-code or decompose. Some algorithm classes are hard to program within the MapReduce paradigm [28], thus it may be a good alternative to keep them as black boxes and optimize the whole experimental process rather than the parallel execution of the algorithm internally. In addition, the dynamic loop iterates along several chained executable codes, rather than one map reduce algorithm.

There are some scientific tools with relevant features to interact with workflows. Science gateways such as WorkWays [23] is a Web portal that creates an interactive gateway that allows users to insert and export data from, a continuously running workflow as a service. WorkWays explores the concept of workflows as a service. Thus, a static workflow specification is maintained while the user insert and export data. This approach restricts adaptations of the workflow configuration such as omega adaptations. OpenMole [27] is another workflow system designed for simulation models that need continuous adaptation and improvement. Scientists can exchange software in the workflow during its execution, though the workflow configuration is statically defined on domain specific language. Our approach takes advantage of the benefits provided by the workflow algebra, e.g. workflow optimization, and by data provenance coupled to execution model, which allows for querying the results on a structured manner and at runtime. Additionally, provenance data is kept consistent with the workflow adaptations.

7. Conclusion

Dynamic workflows are subject to continuous adaptation and refinement, thus requiring the ability to deal with user interaction at runtime. Since user interaction with a workflow is iterative, supporting dynamic iteration in SWfMS is a major requirement. In this paper, we proposed algebraic operators to support data-centric iteration in dynamic workflows and a dynamic execution model. We use database foundations for dataflow without forcing the use of a database management system as the execution engine (we used a workflow engine instead). This approach enables the execution of complex scientific workflows using legacy and complex scientific codes and the management of provenance data supporting queries at runtime. We introduced the concept of iteration lineage so that provenance data management is consistent with dynamic iterative workflow changes. Lineage enables scientists to interact with workflow data and configuration at runtime through two steering algorithms. Using an iterative approach to manage the workflow evolution, we allow scientists to query results across the iterations of the experiment. Based on these analyses, scientists may interact with workflow input data or configuration at runtime.

We designed dynamic loops for any experiment that needs to be adapted and refined by scientists during its execution. In this work, we did a thorough experimental evaluation using a real-life experiment for UQ analysis in the Oil & Gas domain. We developed the iterative module inside Chiron, a data-centric scientific workflow engine that supports the algebraic approach. We executed the workflow on a 640-core cluster, showing the benefits of the dynamic loops from the user perspective. The results show impressive execution time savings from 2.5 to 24 days while performing complex queries for result analysis along the iterations. In a situation where we privilege the non-iterative approach, the overhead introduced by the dynamic loops approach is very small (less than 0.6% on a significant scenario) when compared to the total execution time of the experiment. The performance of the steering algorithms is also fast and increases linearly in the UQ scenario. We believe that iterative execution steering still has enormous potential to be explored by scientists supported by dynamic changes.

8. Acknowledgements

Work partially funded by CNPq, CAPES, FAPERJ and INRIA (SwfP2Pcloud and Hoscar projects) and performed (for P. Valduriez) in the context of the Computational Biology Institute (www.ibc-montpellier.fr). We also thank NACAD High Performance Computing Center, where the experiments were executed.

9. References

[1] Abramson, D., Bethwaite, B., Enticott, C., Garic, S., Peachey, T. Parameter Space Exploration Using Scientific Workflows. *Computational Science – ICCS 2009*, , Springer Berlin / Heidelberg, 104–113, 2009.

[2] Bertino, E., Bernstein, P., Agrawal, D., Davidson, S., Dayal, U., Franklin, M., Gehrke, J., Haas, L., Halevy, A., et al. Challenges and Opportunities with Big Data, 2012.

[3] Bickel, E., Bratvold, R. From Uncertainty Quantification to Decision Making in the Oil and Gas Industry. *Energy, Exploration & Exploitation*, 26(5):311–325, 2008.

[4] Bouganim, L., Florescu, D., Valduriez, P. Dynamic load balancing in hierarchical parallel database systems. *Proceedings of the 22nd International Conference on Very Large Databases*, 436–447, 1996.

[5] Bu, Y., Howe, B., Balazinska, M., Ernst, M.D. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, 2010.

[6] Danforth, S., Valduriez, P. A FAD for data intensive applications. *IEEE Transactions on Knowledge and Data Engineering*, 4(1):34–51, 1992.

[7] Deelman, E., Mehta, G., Singh, G., Su, M.-H., Vahi, K. Pegasus: Mapping Large-Scale Workflows to Distributed Resources. *Workflows for e-Science*, , Springer, 376–394, 2007.

[8] Dias, J., Ogasawara, E., Oliveira, D., Porto, F., Coutinho, A., Mattoso, M. Supporting Dynamic Parameter Sweep in Adaptive and User-Steered Workflow. *6th Workshop on Workflows in Support of Large-Scale Science*, 31–36, 2011.

[9] Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., Fox, G. Twister: A Runtime for Iterative MapReduce. *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - HPDC '10*, 810, 2010.

[10] Elmroth, E., Hernández, F., Tordsson, J. Three fundamental dimensions of scientific workflow interoperability: Model of computation, language, and execution environment. *Future Generation Computer Systems*, 26(2):245–256, 2010.

[11] Ewen, S., Tzoumas, K., Kaufmann, M., Markl, V. Spinning fast iterative data flows. *Proc. VLDB Endow.*, 5(11):1268–1279, 2012.

[12] Freire, J., Koop, D., Santos, E., Silva, C.T. Provenance for Computational Tasks: A Survey. *Computing in Science and Engineering, v.10*(3):11–21, 2008.

[13] Gil, Y., Deelman, E., Ellisman, M., Fahringer, T., Fox, G., Gannon, D., Goble, C., Livny, M., Moreau, L., et al. Examining the Challenges of Scientific Workflows. *Computer*, 40(12):24–32, 2007.

[14] Guerra, G., Rochinha, F.A., Elias, R., de Oliveira, D., Ogasawara, E., Dias, J.F., Mattoso, M., Coutinho, A.L.G.A. Uncertainty Quantification in Computational Predictive Models for Fluid Dynamics Using Workflow Management Engine. *International Journal for Uncertainty Quantification*, 2(1):53–71, 2012.

[15] Guerra, G.M., Zio, S., Camata, J.J., Rochinha, F.A., Elias, R.N., Paraizo, P.L.B., Coutinho, A.L.G.A. Numerical simulation of particle-laden flows by the residual-based variational multiscale method. *International Journal for Numerical Methods in Fluids*, 73(8):729–749, 2013.

[16] Jarke, M., Koch, J. Query Optimization in Database Systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[17] Johnston, W.M., Hanna, J.R.P., Millar, R.J. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[18] Laszewski, G., Hategan, M., Kodeboyina, D. Java CoG Kit Workflow. *Workflows for e-Science*, , Springer, 340–356, 2007.

[19] Meiburg, E., Kneller, B. Turbidity Currents and Their Deposits. *Annual Review of Fluid Mechanics*, 42(1):135–156, 2010.

[20] Missier, P., Belhajjame, K., Cheney, J. The W3C PROV family of specifications for modelling provenance metadata. *EDBT/ICDT '13*, 773–776, 2013.

[21] Montagnat, J., Isnard, B., Glatard, T., Maheshwari, K., Fornarino, M.B. A data-driven workflow language for grids based on array programming principles. *4th Workshop on Workflows in Support of Large-Scale Science*, 7:1–7:10, 2009.

[22] Mosconi, M., Porta, M. Iteration constructs in data-flow visual programming languages. *Computer Languages*, 26(2–4):67–104, 2000.

[23] Nguyen, H., Abramson, D. WorkWays: Interactive workflow-based science gateways. *2012 IEEE 8th International Conference on E-Science (e-Science)*, 1–8, 2012.

[24] Ocaña, K.A.C.S., Oliveira, F., Dias, J., Ogasawara, E., Mattoso, M. Designing a parallel cloud based comparative genomics workflow to improve phylogenetic analyses. *Future Generation Computer Systems*, 29(8):2205–2219, 2013.

[25] Ogasawara, E., Dias, J., Oliveira, D., Porto, F., Valduriez, P., Mattoso, M. An Algebraic Approach for Data-Centric Scientific Workflows. *Proc. of VLDB Endowment*, 4(12):1328–1339, 2011.

[26] Özsu, M.T., Valduriez, P. *Principles of Distributed Database Systems*. 3 ed. New York, Springer, 2011.

[27] Reuillon, R., Leclaire, M., Rey-Coyrehourcq, S. OpenMOLE, a workflow engine specifically tailored for the distributed exploration of simulation models. *Future Generation Computer Systems*, 29(8):1981–1990, 2013.

[28] Srirama, S.N., Jakovits, P., Vainikko, E. Adapting scientific computing problems to clouds using MapReduce. *Future Generation Computer Systems*, 28(1):184–192, 2012.

[29] Taylor, I.J., Deelman, E., Gannon, D.B., Shields, M. *Workflows for e-Science: Scientific Workflows for Grids*. 1 ed. Springer, 2007.

[30] Wozniak, J., Armstrong, T., Maheshwari, K., Lusk, E., Katz, D., Wilde, M., Foster, I. Turbine: A distributed-memory dataflow engine for extreme-scale many-task applications. *Proceeding of 1st International workshop on Scalable Workflow Enactment Engines and Technologies*, 2012.

[31] Xiu, D., Hesthaven, J.S. High-Order Collocation Methods for Differential Equations with Random Inputs. *SIAM Journal on Scientific Computing*, 27(3):1118–1139, 2005.