An Algebraic Approach for Data-Centric Scientific Workflows*

Eduardo Ogasawara[1,2] Daniel de Oliveira[1] Patrick Valduriez[4]
Jonas Dias[1] Fábio Porto[3] Marta Mattoso[1]

[1]COPPE/UFRJ
Rio de Janeiro, Brazil

[2]CEFET/RJ
Rio de Janeiro, Brazil

[3]LNCC
Petrópolis, Brazil

[4]INRIA & LIRMM
Montpellier, France

{ogasawara,jonasdias,danielc,marta}@cos.ufrj.br fporto@lncc.br Patrick.Valduriez@inria.fr

ABSTRACT

Scientific workflows have emerged as a basic abstraction for structuring and executing scientific experiments in computational environments. In many situations, these workflows are computationally and data intensive, thus requiring execution in large-scale parallel computers. However, parallelization of scientific workflows remains low-level, ad-hoc and labor-intensive, which makes it hard to exploit optimization opportunities. To address this problem, we propose an algebraic approach (inspired by relational algebra) and a parallel execution model that enable automatic optimization of scientific workflows. We conducted a thorough validation of our approach using both a real oil exploitation application and synthetic data scenarios. The experiments were run in Chiron, a data-centric scientific workflow engine implemented to support our algebraic approach. Our experiments demonstrate performance improvements of up to 226% compared to an ad-hoc workflow implementation.

1. INTRODUCTION

Many scientific experiments are based on complex computer simulations that consume and produce very large datasets and allocate huge amounts of computational resources. As the complexity of the experiments growths, running simulations becomes a challenge. To help scientists in managing resources involved in large-scale in-silico simulations, scientific workflows are gaining much interest. A workflow can be defined as a model of a process, which consists in a series of activities and its dependencies [1]. Workflows have been used primarily in business data processing. A data-centric scientific workflow, for short scientific workflow, structures the processing of a scientific simulation as a graph of activities, in which nodes correspond to data processing activities and edges represent the dataflow between them. Workflow activities are associated to scientific programs that prepare, process and analyze data.

Scientific Workflow Management Systems (SWfMS) [2] are software systems that support the definition, execution and monitoring of scientific workflows. Various SWfMS have been proposed (*e.g.* VisTrails, Kepler, Taverna, Pegasus, Swift and Triana). Each of them has its own language [2] and focuses on different aspects, such as parallel execution, semantic support, domain specific characteristics and management of provenance data.

Although some SWfMS focus on parallel execution, parallelizing large-scale simulations are still hard, ad-hoc and labor-intensive. Workflow developers (and scientists) need to decide on the ordering, dependencies, and the parallelization strategies. These decisions tighten parallelization opportunities, which may yield to miss important optimization opportunities. Let us illustrate the problem with a critical application we are addressing with Petrobras, Brazil's giant oil company. We will use this example consistently in the rest of the paper.

1.1 Motivating Example: RFA application

To illustrate the problem of optimizing data-centric scientific workflows, let us consider the following motivating workflow scenario from oil exploitation. A major function of an ultra-deep water oil exploitation system is pumping oil from thousand meters up to the surface through tubular structures, called risers. Maintaining and repairing risers under deep water is difficult, costly and critical for the environment (*e.g.* to prevent oil spill). Understanding the dynamic behavior of each riser and its life expectancy is critical for Petrobras. Thus, scientists must predict risers fatigue based on complex scientific models and observed data collected from risers. As shown in Figure 1, performing Risers Fatigue Analysis (RFA) requires a complex workflow of data-intensive activities that may take a very long time to compute. A typical riser's fatigue analysis workflow [3] consumes several input files containing riser information, such as finite element meshes, wind, waves, sea currents, case studies, and produces result analysis files to be further studied by the scientists. Such workflow can be modeled as a Directed Acyclic Graph (DAG) of activities with edges for establishing the dataflow. In Figure 1, each rectangle indicates an activity, solid lines represent input and output parameters that are transferred between activities, and dashed lines represent input and output parameters shared through data files. For simplicity, we assume that all data is shared among all activities, *i.e.*, with a shared disk storage. For a single riser's fatigue analysis, the workflow can have as many as 2,000 input files (about 1GB) and produces about 6,500 files (about 22GB) with nine activities including:

Articles from this volume were invited to present their results at The 37th International Conference on Very Large Data Bases, August 29th - September 3rd 2011, Seattle, Washington.
Proceedings of the VLDB Endowment, Vol. 4, No. 12

* Work partially sponsored by CAPES, CNPq and INRIA (Datluge and Sarava projects).

extracting and formatting riser data, static analysis of risers movements, dynamic analysis of risers movements, tension analysis, curvature analysis, merging data from previous activities and compression of results. Some activities, *e.g.* dynamic analysis, are repeated for many different input files, and depending on the mesh refinements and other riser's information, each single activity execution may take some minutes to complete. The sequential execution of the workflow that calls 1,432 activities and consumes a small dataset, on a SGI Altix ICE 8200 cluster, may take as long as 16 hours to complete.

Some SWfMS provide support for parallel execution of activities in scientific workflows. Swift, for example, allows scientists to specify parallel workflows using a scripting language. Similarly, MapReduce implementations, such as Hadoop [4], allow single activities to be parallelized. However, these solutions require scientists to dictate low-level parallelization strategies that limit the opportunities for automatic optimizations.

In order to illustrate possible optimizations, consider the scenario in Figure 1. Would it be better to consume all data in step 2 before passing corresponding output to step 3 or should we immediately forward some output data from step 2 to execution in step 3? This is a difficult optimization decision which, left to workflow developers or scientists, hurts productivity. On the other hand, in order to let SWfMS automatically decide on the best execution alternative, some metadata would be needed. Similarly, if SWfMS could "realize" that activity 7 is selective, i.e. filters out data, and identify that its input data is not modified after activity 3, then activity 7 could be placed between activity 3 and activity 4. This could reduce the size of the input data to time-consuming activities 4 and 5.

In both cases, important semantic information is missing to support automatic optimization in scientific workflow execution.

1.2 Approach and Contributions

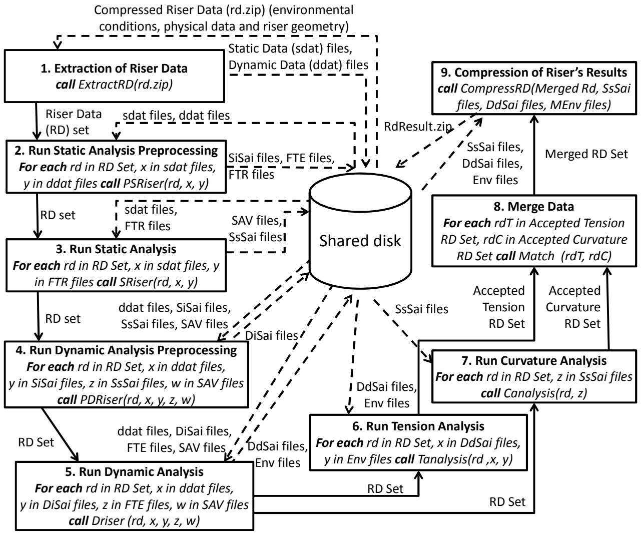

Compressed Riser Data (rd.zip) (environmental conditions, physical data and riser geometry)
Static Data (sdat) files, Dynamic Data (ddat) files
1. Extraction of Riser Data
call ExtractRD(rd.zip)
Riser Data (RD) set
sdat files, ddat files
2. Run Static Analysis Preprocessing
For each rd in RD Set, x in sdat files, y in ddat files call PSRiser(rd, x, y)
SiSai files, FTE files, FTR files
RD set
sdat files, FTR files
SAV files, SsSai files
3. Run Static Analysis
For each rd in RD Set, x in sdat files, y in FTR files call SRiser(rd, x, y)
Shared disk
RD set
ddat files, SiSai files, SsSai files, SAV files
DiSai files
4. Run Dynamic Analysis Preprocessing
For each rd in RD Set, x in ddat files, y in SiSai files, z in SsSai files, w in SAV files call PDRiser(rd, x, y, z, w)
RD Set
ddat files, DiSai files, FTE files, SAV files
DdSai files, Env files
5. Run Dynamic Analysis
For each rd in RD Set, x in ddat files, y in DiSai files, z in FTE files, w in SAV files call Driser (rd, x, y, z, w)
DdSai files, Env files
6. Run Tension Analysis
For each rd in RD Set, x in DdSai files, y in Env files call Tanalysis(rd ,x, y)
RD Set
RD Set
SsSai files
7. Run Curvature Analysis
For each rd in RD Set, z in SsSai files call Canalysis(rd, z)
Accepted Tension RD Set
Accepted Curvature RD Set
8. Merge Data
For each rdT in Accepted Tension RD Set, rdC in Accepted Curvature RD Set call Match (rdT, rdC)
Merged RD Set
SsSai files, DdSai files, Env files
RdResult.zip
9. Compression of Riser's Results
call CompressRD(Merged Rd, SsSai files, DdSai files, MEnv files)

Figure 1. Risers Fatigue Analysis (RFA) scientific workflow.

In this paper, we address the problem of parallelizing activities in scientific workflows by proposing an algebraic approach, which allows for automatic optimization of scientific workflows. Activities in a workflow are ruled by algebraic operators that consume and produce sets of tuples (relations). This yields a declarative workflow representation for scientific workflows that produce and consume relations and eases the generation of a scientific workflow ready for execution. To support such generation, our scientific workflow representation uses the concept of activity activation, inspired by database tuple activations [5]. The concept of activity activation is critical to make a declarative workflow representation able to run in large-scale parallel computers (*e.g.* clusters, grids, or clouds).

The main contributions of this paper are:

- A scientific workflow algebra with operators that provide semantics to activities, which allows automatic transformation of scientific workflows;
- A workflow execution model for this algebra, based on the concept of activity activation, which enables transparent distribution and parallelization of activities;
- A thorough experimental evaluation based on the implementation of our approach in Chiron, a data-centric scientific workflow engine which supports our algebraic approach. We use the Petrobras RFA application workflow on a 256-core parallel computer that shows the benefits of our approach.

1.3 Paper Organization

This paper is organized as follows. In Section 2, we discuss background on workflow management. We describe our workflow algebra in Section 3 and our workflow execution model in Section 4. In Section 5, we provide an experimental evaluation of our approach. Next, Section 6 discusses related work and Section 7 concludes. Appendix A provides additional experiments with synthetic data.

2. WORKFLOW MANAGEMENT

A scientific workflow is composed of activities. An *activity* is a workflow component capable of running programs, with input and output parameters [2]. Parameters are used to represent the dependency between activities within a workflow, *i.e.,* if an input parameter of an activity B is connected to an output parameter of activity A it means that A must execute before B, and the data produced by A is consumed by B.

As an example of a scientific workflow, Figure 2 shows a small part of the RFA workflow, which includes Activity 2 (Static Analysis Preprocessing) and Activity 3 (Static Analysis). Activity 2 consumes <*Rd, 'U-125S.DAT', 'U-125D.DAT'*> as values for input parameters and produces <*Rd, 'U-125.FTE', 'U-125.FTR'*> as values for output parameters. Activity 3 consumes <*Rd, 'U-125S.DAT', 'U-125.FTR'*> as values for input parameters and produces <*Rd, 'U-125.SAV', 'U-125Ss.SAI'*> as values for output parameters.

The scenario depicted in Figure 2 offers very few opportunities for automatic optimization, as the workflow process consumes a single input, <*Rd, 'U-125S.DAT', 'U-125D.DAT'*>. It turns out that during the execution of a scientific experiment, scientists explore the behavior of their model under different inputs. This is commonly known as parameter sweep. In this scenario, with multiple inputs, various optimization opportunities arise. This is particularly relevant considering that the execution of a parameter sweep over the RFA workflow may run for hours/days. Optimizing the RFA workflow under these circumstances require the use of parallelism of activities execution.

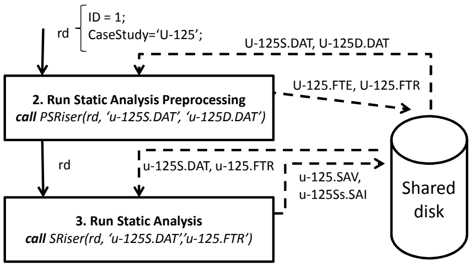
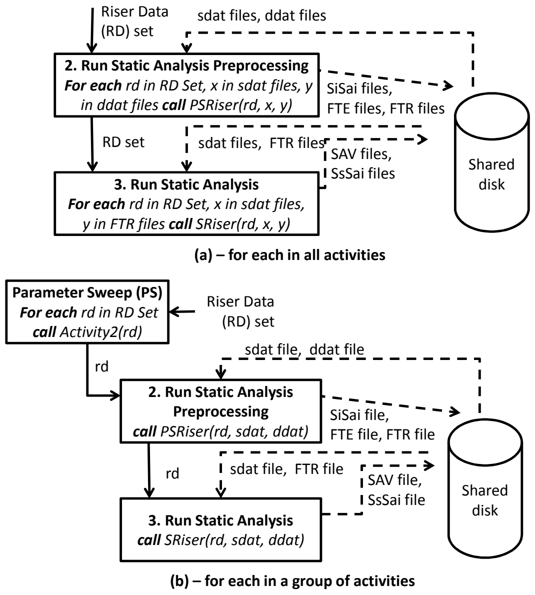


Figure 2. Typical scientific workflow structure.

In support to parameter sweep, a workflow may need to be changed, introducing iteration over the set of values for input parameters (*e.g.* ForEach [6]). Thus, it is up to the workflow developer to decide the ordering, dependency and scheduling of activities on a parallel computer.

For instance, Figure 3 shows two instances of the RFA workflow implementing parameter sweep differently. In the first one (Figure 3.a), a set of inputs is passed to each activity, with a ForEach constructor implementing the iteration. Conversely, Figure 3.b introduces a Parameter Sweep (PS) activity implementing the iteration and embracing workflow activities 2 and 3. These are common strategies used by scientists to introduce parameter sweep in current SWfMS. Nevertheless, the imperative nature of these workflow languages associates to each implementation a, not necessarily optimal, execution strategy.



(a) – for each in all activities

(b) – for each in a group of activities

Figure 3. Different approaches to parallelism.

3. SCIENTIFIC WORKFLOW ALGEBRA

In Section 2 we discuss the restrictions on current SWfMS in producing efficient execution strategies when dealing with multiple inputs. To tackle this problem, our solution is inspired by decades of well-founded query optimization models based on relational algebra. We propose a scientific workflow algebra, where data is uniformly represented by relations and workflow activities are mapped to operators that have data aware semantics. The semantics of these operators allow for workflow rewriting, which enables workflow optimization. We consider values for input and output parameters of workflow activities composing a data unit equivalent to relational tuples. Thus, activities consume and produce relations. Relations are defined as sets of tuples of primitive types (integer, float, string, date etc) or complex data types (*e.g.* references to files).

Each relation R has a schema $\mathtt{R}$, with its typed attributes. Considering the example of Figure 1, we can specify that R_2 is the input relation of the Static Analysis Preprocessing (IPSRiser) activity. The schema for R_2 can be declared as $\mathtt{R}_2$ = *(RID: Integer, CaseStudy: String; SDat: FileRef, DDat: FileRef)*, which can be specified as $R_2(\mathtt{R}_2)$.

Similarly to relational algebra, relations can be manipulated by set operators: union ($\cup$), intersection ($\cap$) and difference (-), as long as their schema are compatible (arity of relations and domain of each attribute are compatible). For example, considering $R(\mathtt{R})$ and $S(\mathtt{R})$ then $(R \cup S)$, $(R \cap S)$, and $(R - S)$ are all valid expressions. It may be also convenient to write a relation in temporary relation variables. The assignment operator ($\leftarrow$) allows this feature. For example, $T \leftarrow R \cup S$ makes a temporary copy of $R \cup S$ in a relation variable T, which can be reused later in algebraic expressions.

In our approach, the definition of an Activity comprehends a computational unit (i.e. a program or a relational algebra expression) and input and output relation schemas. In our notation, Y < $\mathtt{R}$[†], $\mathtt{T}$, CU >, meaning that an activity Y consumes input relations that are compatible with $\mathtt{R}$, and produces output relations that are compatible with $\mathtt{T}$. Table 1 gives an example of an input relation for IPSRiser that conforms to schema $\mathtt{R}_2$.

Activities are ruled by algebraic operators that specify the ratio of consumption and production between input and output tuples, which allows for uniform treatment of data-centric activities and to reason about algebraic transformations. Our algebra includes six operators (see Table 2): *Map*, *SplitMap*, *Reduce*, *Filter*, *SRQuery*, and *JoinQuery*. The first four operators are used to support activities that invoke programs. The last two operators are used to execute activities corresponding to traditional relational algebra expressions, which are useful for data transformation and data filtering. A *SRQuery* consumes a single relation while a *JoinQuery* consumes a set of relations. For these two operators, we assume the existence of an underlying relational algebra execution engine that evaluates the associated relational expressions according to its relational capability.

Table 1. Example of input relation for IPSRiser activity.

RID	CaseStudy	sdat	ddat
1	U-125	U-125S.DAT	U-125D.DAT
1	U-127	U-127S.DAT	U-127D.DAT
2	U-129	U-129S.DAT	U-129D.DAT

[†] Some activities may require more than one input relations. For simplicity, this definition reflects only activities that require single input relation.

We present in more detail these operators in the rest of this section.

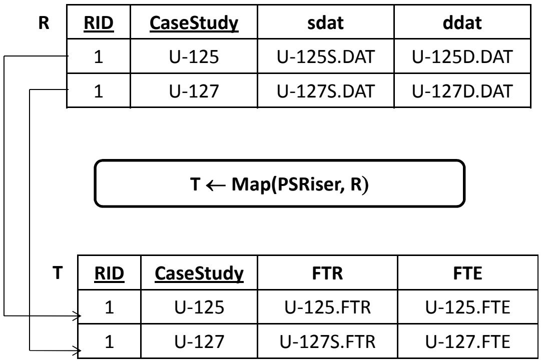
Table 2: Workflow algebraic operators.

Operator	Type of Activity operated	Additional Operands	Result	Ratio of tuples Consumed/ Produced
Map	program	Relation	Relation	1:1
SplitMap	program	FileRef type attribute, Relation	Relation	1:m
Reduce	program	Set of grouping attributes, Relation	Relation	n:1
Filter	program	Relation	Relation	1:(0-1)
SRQuery	rel. algebra expression	Relation	Relation	n:m
JoinQuery	rel. algebra expression	Set of Relations	Relation	n:m

3.1 Map

An activity *Y* is said to be ruled by the *Map* operator whenever a single tuple is produced in the output relation *T* from each tuple consumed in the input relation *R*. The *Map* operator is represented as:

$$T \leftarrow Map(Y, R)$$



R

RID	CaseStudy	sdat	ddat
1	U-125	U-125S.DAT	U-125D.DAT
1	U-127	U-127S.DAT	U-127D.DAT

T

RID	CaseStudy	FTR	FTE
1	U-125	U-125.FTR	U-125.FTE
1	U-127	U-127S.FTR	U-127.FTE

Figure 4. An example of *Map.*

Figure 4 illustrates the *Map* operator using the RFA workflow example. Consider a variable tuple *r* that refers to the first tuple of the input relation *R* for activity *PSRiser*. In this case, *r* ['RID'] = '1' and *r*['CaseStudy'] = 'U-125' are some of the values expressed in that tuple. This tuple is consumed by activity *PSRiser,* which produces the first tuple *t* of the output relation *T* for activity *PSRiser*. In this case, *t*['FTR'] = 'U-125S.FTR' and *t*['FTE'] = 'U-125S.FTE' are some of the values expressed in that tuple.

3.2 SplitMap

An activity *Y* is said to be ruled by the *SplitMap* operator whenever it produces a set of tuples in the output relation *T* for each tuple in the input relation *R*. This decomposition is possible only if the schema *R* for input relation *R* has an attribute *a* of FileRef type to be used for splitting. The file associated to the attribute of FileRef type on each input tuple is decomposed into multiple pieces that are included in the set of output tuples. *SplitMap* is represented as:

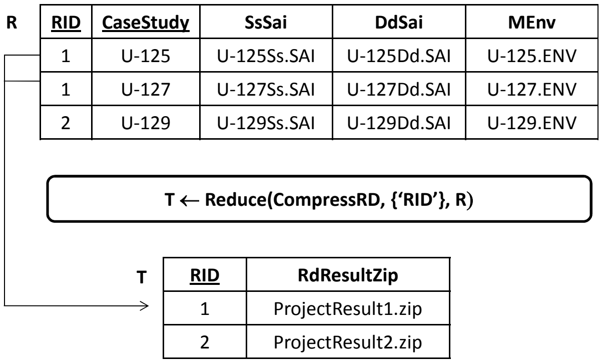
$$T \leftarrow SplitMap(Y, a, R).$$

It is interesting to notice that the splitting criterion is internal to the program represented by the activity.

3.3 Reduce

An activity *Y* is said to be ruled by the *Reduce* operator whenever it produces a single tuple *t* in output relation *T* from each subset of tuples in the input relation *R*. The tuples of *R* are grouped by a set of *n* attributes of the relation *R*, which can be represented as $g_A=\{G_1, ..., G_n\}$. g_A is used to group *R* into subsets (partitions), *i.e.*, g_A establishes criteria for horizontal partitioning over relation *R*. The *Reduce* operator runs activity *Y* consuming each partition at a time and producing an aggregated tuple *t* for each partition. *Reduce* operator is represented as:

$$T \leftarrow Reduce(Y, g_A, R).$$



R

RID	CaseStudy	SsSai	DdSai	MEnv
1	U-125	U-125Ss.SAI	U-125Dd.SAI	U-125.ENV
1	U-127	U-127Ss.SAI	U-127Dd.SAI	U-127.ENV
2	U-129	U-129Ss.SAI	U-129Dd.SAI	U-129.ENV

T

RID	RdResultZip
1	ProjectResult1.zip
2	ProjectResult2.zip

Figure 5. An example of *Reduce.*

Figure 5 shows the reduce operator using the RFA workflow example. It depicts an input relation *R* which contains a grouping attribute *RID* that is the criterion for defining horizontal partitions of *R*. In the example, we have two tuples (r_1 and r_2) for the partition where *RID*=1. r_1['MEnv'] = 'u-125.ENV' and r_2 ['MEnv'] = 'u-127.ENV' as some of the values expressed in these tuples that are used to produce the output tuple *t* of *T*, such that *t*['RID'] = 1, *t*['RdResultZip'] = 'ProjectResult1.zip'.

3.4 Filter Operator (Filter)

An activity *Y* is said to be ruled by the *Filter* operator when each tuple consumed in the input relation (*R*) may be copied to the output relation (*T*), as long as activity *Y* accepts this tuple. Additionally, an important property of the *Filter* operator is that the schema of the input relation is exactly the same as the output relation. The operator executes activity *Y n* times, where *n* is the number of tuples present in *R* and produces *m* accepted output tuples in *T*, where $m \le n$. The *Filter* operator is represented as:

$$T \leftarrow Filter(Y, R).$$

It is interesting to notice that the filtering criteria are internal to the program represented by the activity.

3.5 SRQuery

An activity *Y* is said to be ruled by the *SRQuery* operator whenever it consumes a relation *R* through a relational algebra expression (*qry*) to produce an output relation *T*. The *SRQuery* operator is represented as:

$$T \leftarrow SRQuery(qry, R).$$

This operator is useful for data transformation or data filtering. Additionally, if the schema from the input relation is exactly the same of the output relation, then the *SRQuery* operator is assumed to have also the semantics of a *Filter*.

3.6 JoinQuery

An activity *Y* is said to be ruled by the *JoinQuery* operator whenever it consumes a set of relations to produce a single output relation. The activity is expressed as a relational algebra expression (*qry*) over the input relations $\{R_1, \ldots, R_n\}$. The evaluation of the expression *qry* produces the output relation *T*. This is the only activity that consumes more than one relation. Algebraically, the *JoinQuery* operator is represented as:

$$T \leftarrow JoinQuery(qry, \{ R_1, \ldots, R_n \})$$

This operator is useful for data transformation or data filtering over multiple relations.

3.7 Scientific Workflow Representation

The uniformity obtained by the adoption of the relational data model for data manipulation permits the expression of scientific workflow as composition of algebraic expressions. This can be illustrated by the RFA workflow of Figure 1 restructured in Figure 6 using our algebraic approach.

$T_1 \leftarrow$ SplitMap(ExtractRD, R_1)
$T_2 \leftarrow$ Map(PSRiser, T_1)
$T_3 \leftarrow$ Map(SRiser, T_2)
$T_4 \leftarrow$ Map(PDRiser, T_3)
$T_5 \leftarrow$ Map(DRiser, T_4)

$T_6 \leftarrow$ Filter(Tanalysis, T_5)
$T_7 \leftarrow$ Filter(Canalysis, T_5)
$T_8 \leftarrow$ JoinQuery(Match, $\{T_6, T_7\}$)
$T_9 \leftarrow$ Reduce(CompressRD, T_8)

Figure 6. Algebraic representation of RFA application.

Whenever the output of an activity is entirely consumed by a single activity, it is also possible to rewrite the corresponding algebraic expressions in a more concise way, such as:

$$T_2 \leftarrow Map(PSRiser, SplitMap(ExtractRD, R_1)).$$

4. EXECUTION MODEL

In this section, we present our algebraic parallel execution model. We first introduce the concept of activity activation, as the basic unit of workflow scheduling, whose structure enables any activity to be executed by any core. The flexibility brought by the activation concept allowed us to propose an efficient parallel execution model, which is discussed in the remaining of the section.

4.1 Activity activation

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. Activations contain the finest unit of data needed by an activity to execute [5], *i.e.*, a minimum horizontal partition of the input relation that cannot be further decomposed, yet allowing activity execution. Activations may consume and produce a set of tuples. The ratio of consumption and production of tuples in activations varies according to the operator that rules their activities (see Table 2). The activity output relation is composed of a set of tuples produced by all of its activations. Activations proceed in three steps: input instrumentation, program invocation, and output extraction. Input instrumentation extracts the values of input tuple(s) 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 extracts values from output program parameters and builds activation's output tuple(s) with them.

Figure 7 illustrates the activation of an activity Y consuming input tuples $\{r\}$ of relation *R* and producing output tuples $\{t\}$ of relation *T* for T $\leftarrow \alpha(Y, R)$, where α is the operator that rules activity Y (*Map, SplitMap, Filter or Reduce*).

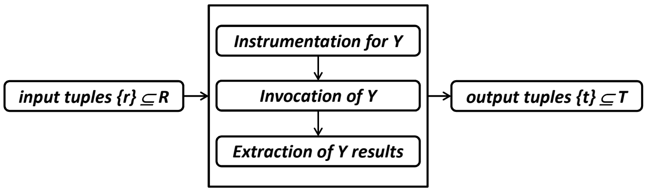


Figure 7. Example of Activation for Map Operator.

Activations may be placed four possible states: *Ready*, *Running*, *Waiting*, and *Finished*. An activation in a *Ready* state is available for execution, but has not been assigned to a particular core. A *Running* activation is allocated to a particular core and is executing one of its three steps (instrumentation, invocation or extraction). An activation is *Finished* when its execution has ended. Activation is *Waiting* when it depends on another activation to finish before it becomes *Ready*.

4.2 Basic Concepts

In this section we discuss some basic concepts required for presenting our execution model. We introduce the concepts of workflow schedule, activity and activation dependency.

A workflow *W* includes a set of activities $Y = \{Y_1, \ldots, Y_n\}$. Given $Y_i \mid (1 \leq i \leq n)$, let $R=\{R_1, \ldots, R_m\}$ be the input relation set for activity Y_i, then $Input(Y_i) \supseteq R$. Also, let *T* be the output relation set produced by activity Y_i, then $Output(Y_i) \supseteq T$. We denote the dependency between two activities as $Dep(Y_j, Y_i) \leftrightarrow \exists\ R_{k'} \in Input(Y_j) \mid R_{k'} \in Output(Y_i)$. Additionally, a fragment of a workflow, *fragment* for short, is a subset *F* of the activities of a workflow *W*, such that either *F* is a unitary set or $\forall Y_j \in F, \exists\ Y_i \in F \mid (Dep(Y_i, Y_j)) \vee (Dep(Y_j, Y_i))$.

Given a workflow W, a set $X=\{x_1, \ldots, x_k\}$ of activations is created for its execution. Each activation x_i belongs to a particular activity Y_j, which is represented as $Act(x_i) = Y_j$. A set of activations of an activity Y_j is denoted as $ActX(Y_j)$. Each activation x_i consumes a set of input tuples $InputX(x_i)$ and produces a set of output tuples $OutputX(x_i)$. We establish the dependency between two activations as $DepX(x_j, x_i) \leftrightarrow \exists\, r \in InputX(x_j) \mid r \in OutputX(x_i) \wedge Dep(Act(x_j), Act(x_i))$.

A workflow schedule is a sequence of activations. Formally, given a workflow W that includes a set of activities $Y=\{Y_1, \ldots, Y_n\}$ and a set $X=\{x_1, \ldots, x_k\}$ of activations created for workflow execution, let $V_p(X)$ be a schedule $<x_1, \ldots, x_k>$ for *X*, we consider ord(x_i) to be the position of x_i in the sequence. We say that $x_i < x_j \leftrightarrow$ ord(x_i) < ord(x_j). $V_p(X)$ is a valid workflow schedule if, for all its activations, either they are independent of each other, or if

$DepX(x_j, x_i) \rightarrow x_i < x_j$, i.e., $V_p(X) \leftrightarrow \neg Dep(Act(x_j), Act(x_i)) \vee \neg(DepX\ (x_j, x_i) \wedge x_j < x_i)$, $\forall x_i\ x_j \mid 1 \leq i, j \leq k, i \neq j$. Possible schedules *$V(X)$* include all valid permutation sequences *$V_p(X)$* for *X*.

Additionally, considering a fragment *F* for the workflow *W*, *X'* is a set of activations in *F*. A strict schedule *$V_p(X')$* for *$X'=\{x_q, ..., x_m\}$* is a valid schedule such that $\forall x_i \in X'\ \exists x_j \in X' \mid (DepX(x_i, x_j) \vee DepX(x_j, x_i))$. Finally, an independent schedule *$V_p(X')$* for *$X'=\{x_q, ..., x_m\}$*, is a valid schedule, such that $\forall x_i, x_j \in X' \mid \neg(DepX(x_i, x_j) \vee DepX(x_j, x_i))$.

4.3 Basic Techniques

In this section we present the basic techniques to support our execution model. We first introduce two possible dataflow strategies: *First Tuple First (FTF)* that partitions a set of activations *X* into a set of strict schedules; and *First Activity First (FAF)* that partitions a set of activations *X* into a set of independent schedules. Then, we associate a dataflow strategy for each workflow fragment as a schedule for its activations. Later, considering an assigned dataflow strategy to a workflow fragment, we also introduce a dispatching strategy as a way to distribute activations. Finally, we describe the execution of dispatched activations by execution processes.

As previously discussed, given a workflow W, an associated workflow activations set $X=\{x_1, \ldots, x_k\}$ is evaluated according to a schedule. The schedule of activations depends on the dataflow strategy assigned to the corresponding workflow fragment. Thus, given a fragment F_i and a dataflow strategy DS_i, a mapping function $DSF(F_i, DS_i)$ assigns a dataflow strategy to a fragment of the workflow. In this context, given a set of activations $X'=\{x_q, \ldots, x_m\}$ associated to a fragment F_i, a dataflow strategy DS_i imposes a partial activation order among activations of *X'*. Furthermore, during scheduling, a set of fragment activation instances (*FAI*) is created as a partition of all activations of that fragment. Each *FAI* is the minimum sequence of activations that obeys the dataflow strategy assigned to the fragment.

A dispatching strategy governs the distribution of *FAIs* to available cores. In the static dispatching strategy, all *FAIs* of a given workflow fragment are pre-allocated to each core before any activation of that fragment starts its execution. Conversely, in a *dynamic* dispatching strategy a subset of *FAIs* of a given workflow fragment is allocated to cores as a response to a request for activations.

Once both scheduling and dispatching strategies have been assigned to a workflow fragment, *i.e.,* an execution strategy has been assigned to the fragment, the scheduler places the fragment in its scheduling queue. Fragments in the scheduling queue with at least one activity having all dependencies satisfied are placed in *Ready* state and may be dispatched to execution processes.

An execution process runs on each available core. It receives a set of *FAIs* to be independently evaluated and notifies the scheduler about its completion. We can interpret the execution process as a map process [7] in the Map-Reduce execution model.

Now we present the scheduler algorithm. The algorithm initiates by receiving a workflow and a set of execution processes. The first step invokes *optimizeWorkflow*(), which splits the workflow into fragments *fragSet*[‡] with a *DS* and a *dispatch strategy* assigned to each of them. While there are still fragments available, a *Ready* fragment *frag* is picked from the *fragSet*. Based on the fragment *frag*, the scheduler generates activations as *FAISet*. Each element in a *FAISet* is dispatched according to dispatch strategy defined for a particular fragment. Whenever all elements in *FAISet* have been completed, the *frag* is removed from the *fragSet*. The scheduling algorithm is shown in Figure 8.

```
SCHEDULER(Workflow W, ExecutionProcessesSet execProc)
    FragmentSet fragSet = optimizeWorkflow(W);
    while (hasElements(fragSet))
        frag = getReadyFragment(fragSet);
        FAISet faiSet = generateActivations(frag);
        DispatchStrategy dispStrat = getDispatchStrategy(frag);
        for each FAI f in faiSet
            dispatch(f, dispStrat, execProc);
        fragSet = removeCompleted(fragSet, frag);
```

Figure 8. Scheduler Algorithm.

5. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate our algebraic approach to optimize and execute data-centric scientific workflows. We use the RFA workflow scenario to measure the performance of different execution strategies under different case studies. In the Appendix, we have conducted a complementary evaluation using synthetic data to analyze broader scientific workflow scenarios.

We performed all experiments on a SGI Altix ICE 8200 cluster (32 nodes, each one with two quad-core processors). We use Chiron, a data-centric scientific workflow engine to support the execution of scientific workflows using our algebraic approach. The system supports four different execution strategies produced by combining the dispatching and dataflow strategies discussed in Section 4.3: static/FTF (*S-FTF*), dynamic/FTF (*D-FTF*), static/FAF (*S-FAF*), and dynamic/FAF (*D-FAF*).

Chiron is implemented in Java 1.6. and MPJ (http://mpj-express.org) for parallel communication. We developed the algebraic operators for *SRQuery* and *JoinQuery* using HyperSQL/HSQLDB (http://hsqldb.org). For storing provenance data, Chiron uses PostgreSQL 8.4.

In this evaluation, we consider three different RFA analyses. The first analysis considers 358 case studies for a single riser. It compares the performance results of the proposed execution strategies, when varying the number of available cores (8, 16, 32, 64, 128), as shown in Figure 9. The differences on the elapsed-time among the 4 execution approaches achieve nearly a factor of nine times, when comparing the average elapsed-time using 8 and 128 cores. We observe that the difference in performance varies significantly, up to 226% between the best and the worst strategies (*i.e. S-FAF* vs *D-FTF*) when running with 128 cores.

The *FTF* strategy yields the best performance gains on this analysis. Even with unbalance among activations execution, all execution processes remain active. Conversely, in *FAF* strategies, the execution processes that finish their work remain idle, waiting

[‡] Due to space restrictions, the workflow fragment split criteria is not discussed in this paper.

for long running execution processes to finish, before receiving activations of the next activity in the workflow.

The second analysis considers 175 case studies for four risers, corresponding to a total of 700 case studies. We measure the efficiency of the execution approaches compared to a serial execution, as the number of cores increases. Efficiency is computed as $T_1/(pT_p)$, where T_1 is the serial elapsed time, and p is the number of cores, and T_p is the parallel elapsed time using p cores. As can be observed in Figure 10, the *D-FTF* maintains good efficiency (85% for 128 cores), while other strategies degrade faster (less than 70% for 128 cores).

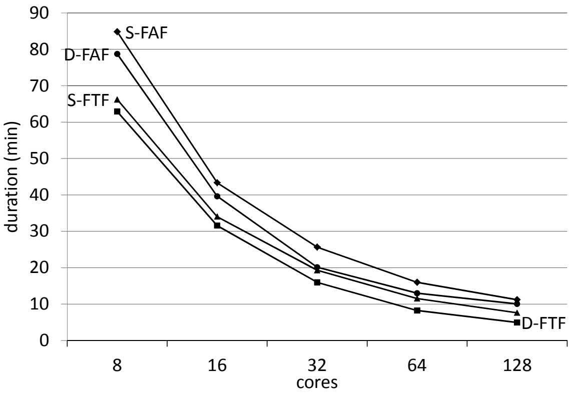


Figure 9. Evaluation of RFA workflow with 358 case studies.

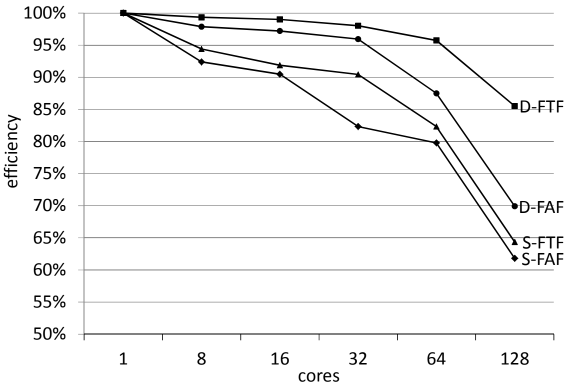


Figure 10. Evaluation of RFA workflow with 700 case studies.

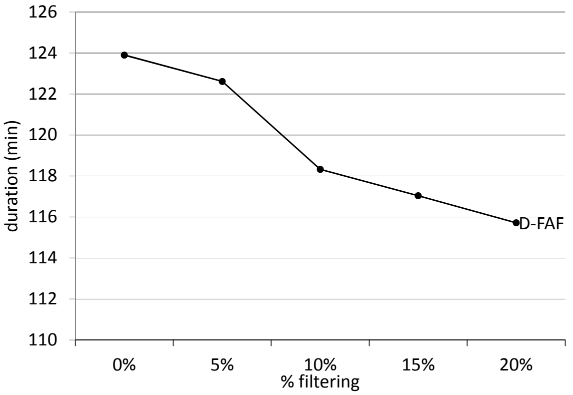


Figure 11. Optimization of RFA workflow.

The third analysis explores the benefits of our algebraic approach, which automatic modifications on the workflow graph may be applied to increase performance. We consider the RFA workflow and swap the order of the *Curvature Analysis* activity with the *Dynamic Analysis* activity. The *Curvature Analysis* is a filtering activity that can reduce the size of the input relation to be consumed by the *Dynamic Analysis* activity, which thus reduces the number of the latter activations to be executed. Figure 11 shows the performance gain with different filtering levels (0%, 5%, 10%, 15% and 20%) using D-FAF. A selective filtering (20%) offers a significant 6% performance gain on the elapsed-time of the workflow execution. In this case, the filtering affects only the *Dynamic Analysis* activity of the RFA workflow. However, if the workflow had other subsequent activities, the benefit would have been propagated and the performance results would have been equally improved.

To summarize, considering both evaluations using RFA workflow and synthetic data, we conclude that in most cases, it is preferable to use *D-FTF*. However, if activity cost is low, it is preferable to use static dispatching strategies. Moreover, whenever a workflow fragment has a constrained activity (CA), it is preferable to use D-FAF. Finally, although in many other SWfMS it may be simpler for workflow developers to implement codes with similar behavior as the *S-FAF*, such as depicted in Figure 3.a, *S-FAF* should be avoided.

6. RELATED WORK

Petri Net and π-calculus are two well-known formalisms to represent workflows. Petri Net provides a formal graph-based representation of discrete distributed systems. π-calculus is a modern process algebra based on the concept of mobility, which includes communication between different π-calculus processes. Several processes together form a pattern of behavior, which represents a workflow pattern. While both Petri Net and π-calculus are powerful representations, they are very general (support many different patterns). Both approaches are mainly used to investigate properties of processes instead of focusing on the data produced and consumed in the course of the workflow. These two characteristics together make it very hard to model data-centric scientific workflow optimization.

In the area of scientific workflow management, the imperative nature of workflow languages makes it hard to model optimization as well. This is due to the absence of semantics in the definition of data-centric activities and the lack of uniform treatment of data processed by workflow activities. Some initiatives started to bring a uniform representation (XML) for pipelined flow [8], but since activities do not have a clear semantic for data production and consumption, it may miss important optimizations, such as executing filtering activities first.

In business process management (BPM), some interesting approaches have been developed for workflow management [9,10]. Pig latin [9] mixes the declarative style of SQL and the low-level procedural style of Map-Reduce, with User Defined Functions (UDF) to implement business routines. Pig latin has a uniform data structure for representing business processes, which enables business workflow optimization. Another important work in BPM is the Process Graph Model (PGM) [10], which uses SQL within the workflow language to support business workflow optimization and execution. Even though these approaches have important concepts, such as uniform data view, they are not directly suitable for supporting scientific workflow applications.

In fact, both Pig latin and PGM are designed and implemented to support business applications which are assumed to suffer little changes over time [11] while scientific experiments demand more flexible approaches that support frequent changes in workflow definitions.

Business data-centric workflows, such as ETL workflows [12] follow a specific workflow template, *i.e.* with Extract, Transform and Load activities that consume and produce data that are already represented as relations for data warehousing. Representation and optimization of ETL workflows are simpler than a generic scientific workflow since all data is structured and only relational algebra operators are used. On the other hand, scientific workflows have activities that consume and produce files in heterogeneous forms (binary, XML) and many of them are related to programs. Although relational algebra is a well-known concept in the database community, we are not aware of any other work that proposes a data-centric algebraic approach to promote parallelization of data-centric workflow activities.

7. CONCLUSION

In this paper, we addressed the problem of workflow optimization through an algebraic approach. Our approach is inspired by decades of well-founded query optimization models based on relational algebra to abstract query execution plans. We proposed a scientific workflow algebra, where data is uniformly represented by relations and workflow activities are mapped to operators that have data aware semantics. Therefore, we are able to combine data and activities in the same algebraic approach.

The representation of activities as data processing algebraic operators enables automatic workflow graph transformation and is the basis for activity parallelization. We proposed a workflow execution model grounded on the concept of activity activation, a fundamental concept for the transparent distribution of activities in parallel computers. Given a set of activations, different execution strategies lead to different workflow execution performance. We investigated four such execution strategies, combining scheduling with dispatching strategies.

We conducted a thorough experimental evaluation of our algebraic approach using Chiron, a data-centric scientific workflow engine that we are building. We performed two complementary sets of experiments. The first set of experiments is based on the Petrobras RFA application, with the objective of comparing the overall performance of the proposed execution strategies versus the number of available cores involved in the execution. The performance results show a variation of up to 226% when we compare the best with the worst performance results. We also evaluated workflow optimization through algebraic manipulation. In this experiment, we swap the order of activities in the workflow graph, which leads to better performance results.

The second set of experiments is based on synthetic data, with the goal of analyzing the proposed execution model in light of selected workflow graphs commonly found in scientific applications. Our experiments show that even for the most simple workflow graph, such as a sequence of Map operators, its performance varies significantly. Furthermore, the best strategy also alternates according to the workflow to be executed. This illustrates how difficult it is for scientists to define an efficient parallel execution strategy for scientific workflows.

We conclude that an algebraic approach for data-centric scientific workflows brings a uniform representation that enables automatic workflow optimization and is the basis for developing efficient strategies for the parallel execution of workflow activities.

8. ACKNOWLEDGMENT

We would like to thank Ismael Santos from Petrobras, for his help with the RFA application and experiments. We are also grateful to the High Performance Computing Center (NACAD-COPPE/UFRJ) where the experiments were performed. We also thank Pedro Cruz, Ricardo Busquet and Vítor Silva for their effort in developing Chiron.

9. REFERENCES

[1] W. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. The MIT Press, 2002.

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

[3] M. Mattoso, C. Werner, G.H. Travassos, V. Braganholo, L. Murta, E. Ogasawara, D. Oliveira, S.M.S. da Cruz, and W. Martinho. Towards Supporting the Life Cycle of Large-scale Scientific Experiments. *Int Journal of Business Process Integration and Management*, 5(1):79–92, 2010.

[4] J. Wang, D. Crawl, and I. Altintas. Kepler + Hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems. *Proc. of 4th Workshop on Workflows in Support of Large-Scale Science*, 1-8, 2009.

[5] L. Bouganim, D. Florescu, and P. Valduriez. Dynamic load balancing in hierarchical parallel database systems. *Proc. of VLDB*, 436-447, 1996.

[6] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73-169, 1993.

[7] F.N. Afrati and J.D. Ullman. Optimizing joins in a map-reduce environment. *Proc. of EDBT*, 99–110, 2010.

[8] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher. Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 25(5):541-551, 2009.

[9] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. *Proc. of SIGMOD*, 1099-1110, 2008.

[10] M. Vrhovnik, H. Schwarz, O. Suhre, B. Mitschang, V. Markl, A. Maier, and T. Kraft. An approach to optimize data processing in business processes. *Proc. of VLDB*, 615-626, 2007.

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

[12] A. Simitsis, P. Vassiliadis, U. Dayal, A. Karagiannis, and V. Tziovara. Benchmarking ETL workflows. *Performance Evaluation and Benchmarking*, R. Nambiar and M. Poess, eds., Springer, 199-220, 2009.

[13] W. van der Aalst, A. Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5-51, 2003.

[14] ProvChallenge. *Provenance Challenge Wiki*, http://twiki.ipaw.info/bin/view/Challenge/WebHome, 2010.

APPENDIX A. SYNTHETIC DATA EVALUATION

We have conducted an evaluation to analyze how complete is the proposed algebraic approach in expressing broader scientific workflow scenarios, in particular, in measuring the performance of each execution strategy (*S-FTF*, *D-FTF*, *S-FAF*, *D-FAF*). Since no performance benchmark is available for scientific workflows, we generated synthetic data for 7 representative workflows scenarios inspired by workflow patterns [13] (see Figure 12). Our scenarios consider only workflow patterns that are relevant for data centric workflows, which correspond to more than half of control patterns (24 of 43). However, because we assume a shared disk architecture, workflow data patterns are not included since they mostly focus on data movement.

We define some workflow scenario configuration variables, configuration variables for short, which influence the performance of a workflow execution. Table 3 shows the set of configuration variables used in the evaluation. The first three variables in Table 3 influence the workflow graph.

The remaining variables set values that influence the workflow execution time for a given workflow graph.

In Figure 12, each activity includes information about its algebraic operator. We use solid lines to represent mandatory workflow elements, while dashed lines represent elements that would vary according to the configuration variable setup.

Table 3. Variables used in evaluation.

Variable	Description
MD	Map Deep: Maximum length of map sequences
FI	Fan In: Maximum in-degree for activities.
FO	Fan Out: maximum out-degree for activities.
AC	Activity Cost: random variable that follows a Gamma distribution with a shape θ = 1 and mean that stands for average time in seconds to complete an activation.
IS	Input Size: number of tuples in the initial input relation.
SF	Split Factor: multiplication factor for the number of output tuples in *SplitMap*.
CA	Constrained Activity: Considering a node containing *n* core, an activation of a constrained activity inhibits the usage of other core to execute other activations.

In general terms, the evaluation of all synthetic data scenarios generated 176 workflows executions, totalizing 1,404 activities and 667,648 activations. The complete evaluation took 46h52min of elapsed-time, which corresponds to a total sequential time of 2,586h12min. We conducted our analysis using 8 nodes with 8 cores each (64 core in total). It is worth mentioning that the scenarios in Figure 12 are representative of many real scientific experiments, such as bioinformatics, computational fluid dynamics, and data mining. For example, workflows described in Provenance Challenge [14] contain combinations of our evaluation scenarios from #1 to #5.

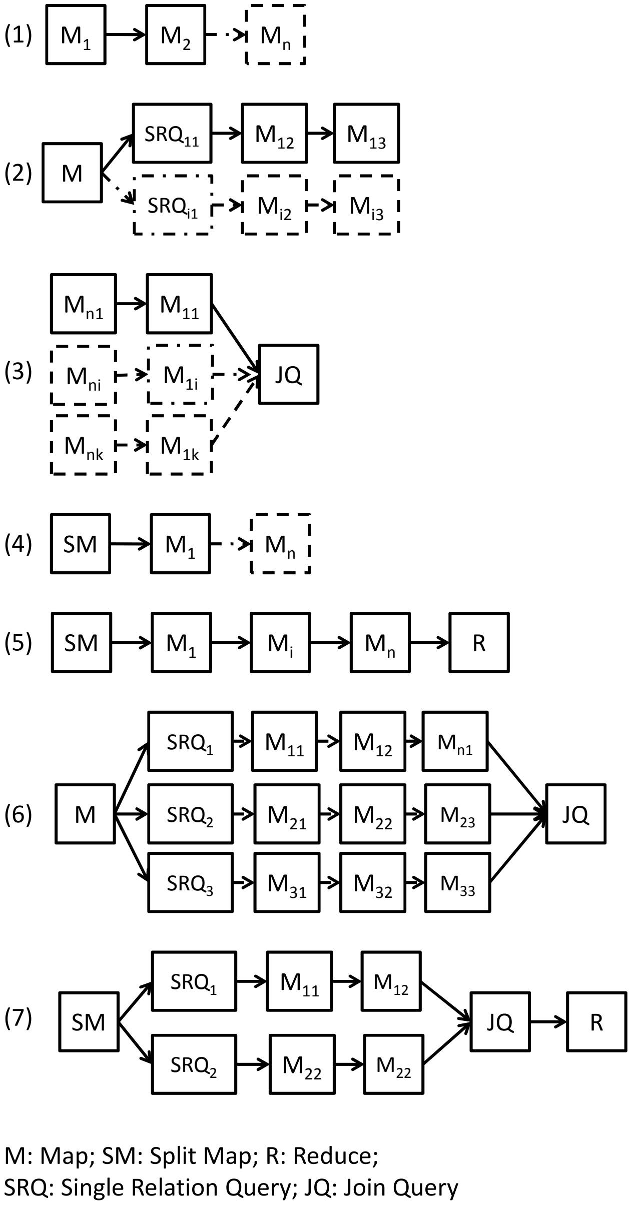


Figure 12. Scenarios for measuring synthetic workflows.

Scenario #1. This scenario shows a workflow graph of a simple sequence of activities. As this scenario is the building block of many workflow graphs, we conduct three different analyses. In the first analysis, we vary MD from 1 to 5 while considering a fixed IS=512 tuples and AC=10. We can observe in Figure 13 that D-FTF has the best performance results.

The second analysis varies AC from 5 to 25, while considering a fixed IS=512 tuples and MD=3. Figure 14 shows that as AC increases the *S-FTF* and *D-FAF* strategies yield similar results, with *D-FTF* having the best performance. The performance difference between the best and worst strategies is 27.6%.

In the third analysis we evaluate workflows with constrained activities (CA). In the scientific workflow domain, it is very common to invoke scientific programs that have a parallel implementation and run using all possible available cores of an allocated node. Due to this characteristic, an activation of a constrained activity blocks all cores, inhibiting further allocations for other activations.

In this analysis, we vary the number of CA from 0 to 4, while considering a fixed IS = 512 tuples, MD=3, and AC=5. Figure 15 shows that whenever a CA is included in a workflow (CA≥1), *FAF* strategies clearly show superior performance. The presence of CAs completely degrades the performance of FTF. A *FAI* containing a CA activation waits for running *FAIs* in that node to complete. At that point all cores are reserved for the *FAI* that blocks them until the end of its execution. The performance difference between the best and worst strategies may reach 169%.

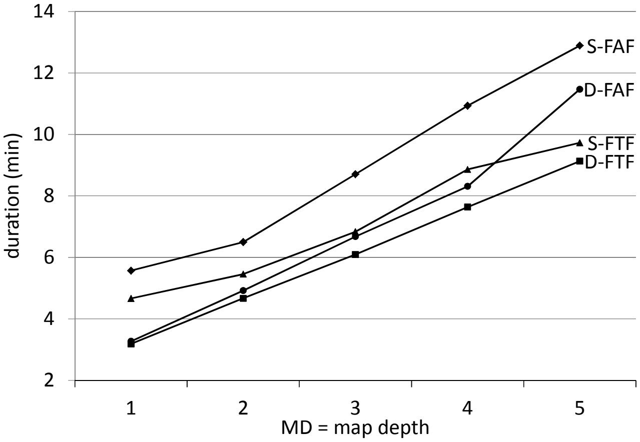


Figure 13. Scenario #1 varying MD.

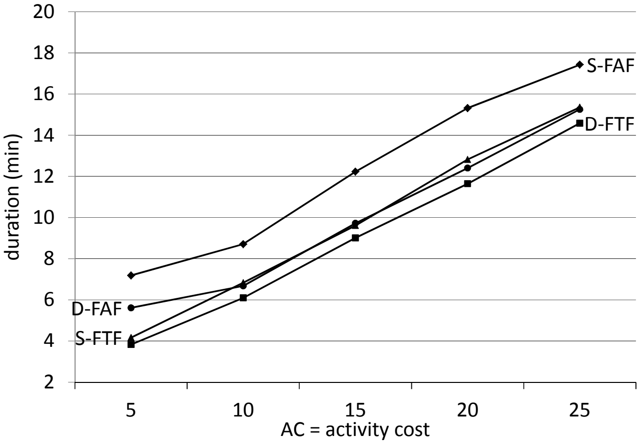


Figure 14. Scenario #1 varying AC.

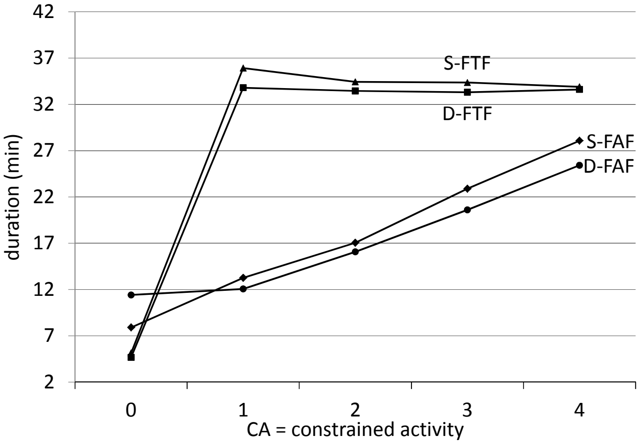


Figure 15. Scenario #1 varying CA.

Scenario #2. This scenario shows a fan-out workflow. The output of the first activity fans out to a sequence of *SRQuery* followed by *Map* activities. In this analysis, we vary FO from 1 to 5 while considering a fixed AC=10, MD=3 and IS=512 (Figure 16). The dynamic strategies yield good performance, with *S-FTF* and *D-FAF* closed to each other.

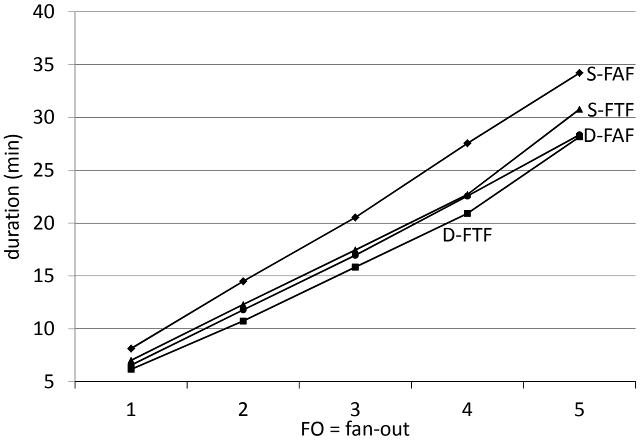


Figure 16. Scenario #2 varying FO.

Scenario #3. This scenario shows a fan-in workflow. The output of multiple sequences of Maps is joined using a *JoinQuery*. In this analysis, we vary FI from 1 to 5 while considering a fixed AC=10, MD=3 and IS=512 (Figure 17). As FI increases, *D-FTF* increases its performance advantage over other strategies.

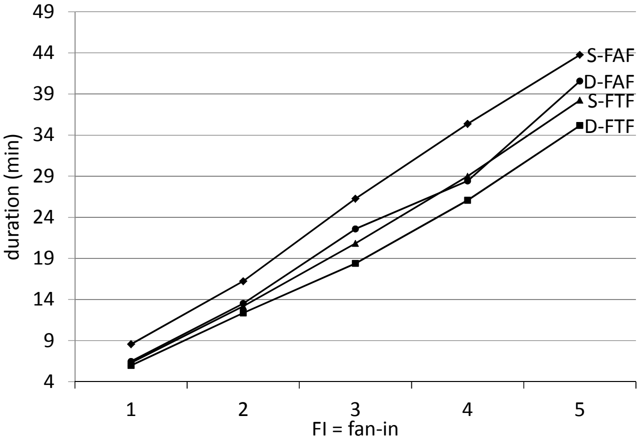


Figure 17. Scenario #3 varying FI.

Scenario #4. This scenario shows a *SplitMap* followed by a sequence of *Map* activities. We vary SF from 1 to 5, while considering a fixed AC=10, MD=3, and IS=256 (Figure 18). Both dynamics strategies have superior performance. Additionally, as SF increases, the performance of *S-FAF* degrades faster than for the other strategies.

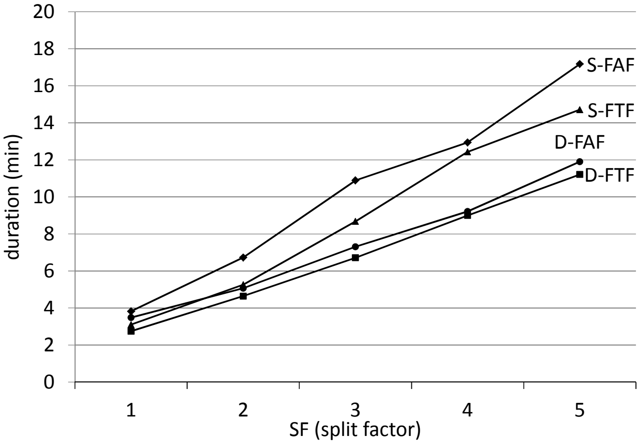


Figure 18. Scenario #4 varying SF.

Scenario #5. This scenario shows a Map-Reduce workflow graph that starts with *SplitMap* activity, then a sequence of *Map* activities and finishes with a *Reduce* activity. In this scenario, we vary IS using the following values (128, 256, 384, 512, 640) while considering a fixed AC=10, SF=3, and MD=3 (Figure 19).

Again both dynamics strategies show superior performance results.

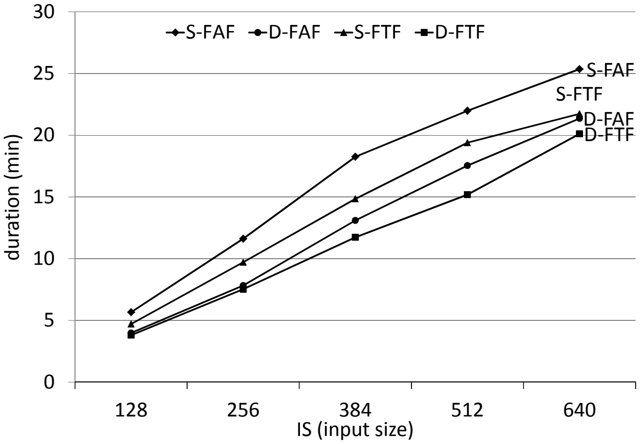


Figure 19. Scenario #5 varying IS.

Scenario #6. This scenario mixes both fan-out and fan-in in a single workflow. The output of the first *Map* is fan-out in different branches. Each branch contains a *SRQuery* followed by a sequence of *Maps*. The output of all branches is fan-in using a *JoinQuery*. In this scenario, we vary AC from 5 to 25 using a fixed MD=3, FO=3, FI=3, and IS=512 (Figure 20). When AC is low (5s), both dynamic strategies exhibit bad performance, indicating a communication overhead.

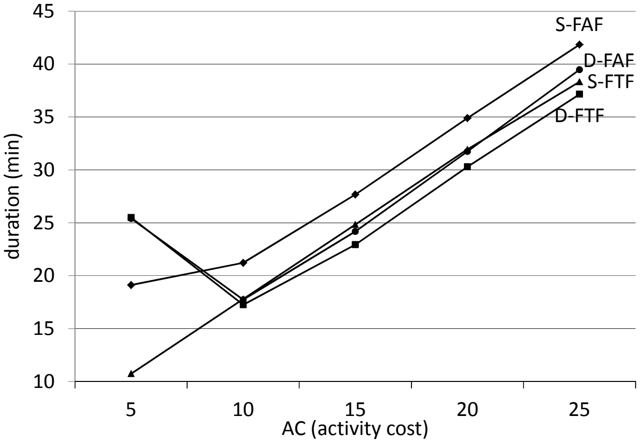


Figure 20. Scenario #6 varying AC.

Scenario #7. This scenario combines most of the operators to generate a complex workflow. The output of the first *SplitMap* is faned-out in different branches. Each branch contains a *SRQuery* followed by sequence of *Map*. The output of all branches is faned-in using a *JoinQuery*. The last activity is a *Reduce*. In this scenario, we varied IS using the following values (128, 256, 384, 512, 640) while considering a fixed AC=10, SF=2, FO=2, and MD=2 (Figure 21). *D-FTF* has the best performance and *S-FTF* and *D-FAF* are close to each other.

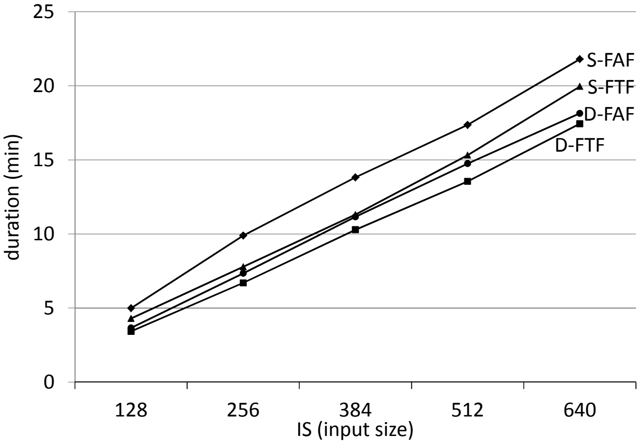


Figure 21. Scenario #7 varying IS.

Let us analyze the performance results obtained from the synthetic data workflow scenarios. Workflows containing a sequence of Maps are common in scientific applications. The behavior of *FAF* in this scenario suffers from the synchronism imposed by execution on a per-activity basis. The occurrence of unforeseen, unbalanced execution profiles leads to inactivity as cores finish their work. This phenomenon is proportionally aggravated as the workflow size increases and idle periods are accumulated through the workflow activities, as depicted in Figure 22. The responsiveness obtained by the dynamic dispatching mode diminishes the synchronism effect as faster execution processes consume more activations, compensating less performing ones. The gain obtained using dynamic dispatching in *FAF* is not sufficient, however, to beat *D-FTF*. In the latter, the consequence of an unbalanced schedule of *FAIs* is pushed to the end of the execution, which, in the worst case, corresponds to the size of the last executing *FAI*. On non-worst case scenarios, the last executing *FAI* would run in parallel with other *FAIs*, beating the synchronous evaluation of *D-FAF*. Figure 23 confirms our observations. It depicts the idleness behavior induced by Scenario #1. We plot the percentage of idle cores versus the total number of allocated cores, during the execution, for each execution strategy.

As shown in Figure 15 and in some other scenarios, the relative performance of *D-FAF* and *S-FTF* sometimes alternate according to skew in execution time of activations. If skew is low, *S-FTF* tends to be better than *D-FAF,* and worse otherwise. Another interesting result concerns the evaluation of workflows having constrained activities (see Figure 15). In our current implementation, a *FAI* having an activation of a CA blocks all available cores during the whole *FAI* execution time. Thus, during the execution of activations of non-CA activities, all cores except the one allocated to the process running the activation of CA remain idle. This execution strategy is in prejudice to *FTF*, proportional to the time taken by the remaining activations in that FAI. Note, however, that if the remaining activations are significantly less costly than that of CA, or the activations in FAI are all of CA type (such as workflow execution with CA ≥ 4 with 6 workflow activities in Figure 15), then *FAF* and *FAI* yield to similar results.

Scenarios like those shown in Figure 16 and Figure 17 explore modifications on the fan-out and on the fan-in in a workflow graph. These scenarios exhibit a behavior comparable to a sequence of Maps. Irrespectively of the execution strategy adopted to run the workflow, the activities implementing the relational query, in the fan-out scenario, and the join query, in the fan-in scenario, are scheduled according to the FAF execution approach. Thus, the comparison among the execution strategies reduces to that of a sequence of Maps. Similarly, Scenario #6 in Figure 20 shows a workflow graph that combines fan-out and fan-in. As expected, it conforms to the sequence of Maps behavior, showing results very close to that of Scenario # 1, Figure 14. An interesting behavior appears, in cases where activity costs are in average low, with respect to the dispatching cost. In cases of this type, static executions overcome dynamic runs. We are evaluating the dispatching implementation to reduce its cost.

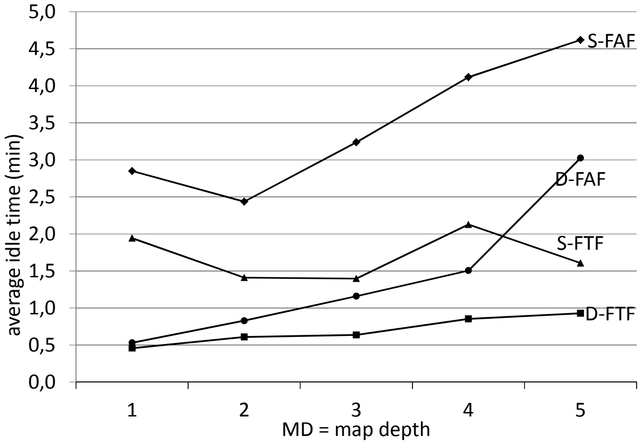


Figure 22. Average idle time during workflow execution relative to scenario #1 (Figure 13).

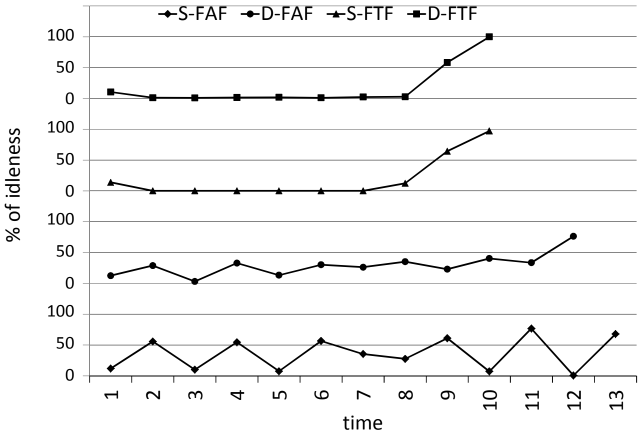


Figure 23. % of idleness during workflow execution of scenario #1 – WD = 5, AC = 10.

The increase of input size was evaluated with both Scenarios #4 and #5, in Figures 18 and 19, respectively. The results show that the increase of the input size does not influence the decisions regarding the choice among execution strategies. Finally, scenario #7 produces a complex workflow using all previous explored workflow variations. The experiment reinforces previous observations that focus the performance behavior in selecting the adequate execution strategy to run sequence of Maps workflow fragments.