Cache-aware scheduling of scientific workflows in a multisite cloud

Gaëtan Heidsieck[a,*], Daniel de Oliveira[b], Esther Pacitti[a], Christophe Pradal[a,c], François Tardieu[d], Patrick Valduriez[a]

[a]*Inria & LIRMM,University of Montpellier, France*
[b]*Fluminense Federal University, Niterói, Brazil*
[c]*CIRAD, UMR AGAP, F-34398 Montpellier, France*
[d]*INRAE LEPSE, Montpellier SupAgro, France*

Abstract

Many scientific experiments are performed using scientific workflows, which are becoming more and more data-intensive. We consider the efficient execution of such workflows in a multisite cloud, leveraging the heterogeneous resources available at multiple geo-distributed data centers. Since it is common for workflow users to reuse code or data from previous workflows, a promising approach for efficient workflow execution is to cache intermediate data in order to avoid re-executing entire workflows. However, caching intermediate data and scheduling workflows to exploit such caching in a multisite cloud with heterogeneous sites is complex. In particular, workflow scheduling must be cache-aware, in order to decide whether reusing cached data or re-executing workflows entirely. In this paper, we propose a solution for cache-aware scheduling of scientific workflows in a multisite cloud. Our solution is based on a distributed and parallel architecture and includes new algorithms for adaptive caching, cache site selection, and dynamic workflow scheduling. We implemented our solution in the OpenAlea workflow system, together with cache-aware distributed scheduling algorithms. Our experimental evaluation in a three-site cloud with a real application in plant phenotyping shows that our solution can yield major performance gains, reducing total time up to 42% with 60% of the same input data for each new execution.

Keywords: Multisite cloud, Distributed Caching, Scientific Workflow, Workflow System, Workflow Scheduling

1. Introduction

In many scientific domains, *e.g*, bio-science [1], complex numerical experiments typically require many processing or analysis steps over huge datasets. They can be represented as scientific workflows, or workflows for short in this paper (but not to be confused with business workflows). These workflows facilitate the modeling, management, and execution of computational activities linked by data dependencies. As data size and computation complexity keep increasing, these workflows become data-intensive [1], thus requiring high-performance computing resources.

The cloud is a convenient infrastructure for supporting workflow execution, as it allows leasing resources at a very large scale and relatively low cost. In this paper, we consider the execution of a data-intensive workflow in a multisite cloud, *i.e.*, a cloud with geo-distributed data centers (henceforth named sites). Today, all popular public clouds, *e.g*, Microsoft Azure, Amazon EC2, and Google Cloud, provide a multisite option that allows accessing multiple cloud sites, or sites for short, with a single cloud account. The main reason for using multiple sites to execute data-intensive workflows is that they often exceed the capabilities of a single site, either because the site imposes usage limits for fairness and security, or simply because the datasets are too big.

In scientific applications, the storage and computing capabilities of the different sites, *e.g*, on premise servers, HPC platforms from research organizations or federated sites at the national level [2], can be very heterogeneous. For instance, plant phenotyping, greenhouse platforms generate terabytes of raw data from plants. Such data is typically stored at data centers geographically close to the greenhouse to minimize data transfers. However, the computation power of those data centers may be limited and fail to scale when the analyses become complex, such as in plant modeling or 3D reconstruction. In this case, the computing capabilities of other sites are required.

Most scientific workflow management systems, or workflow systems for short, can execute workflows in the cloud [3]. Some examples are Swift/T, Pegasus, SciCumulus, Kepler and OpenAlea [4, 5, 6, 7, 8]. Our work is based on OpenAlea [8], which is widely used in plant science for simulation and analysis. Most existing systems use naive or manual approaches to distribute the tasks across sites. The problem of scheduling a workflow execution over a multisite cloud has started to be addressed in [9], using performance models to predict the execution time on different resources. In [10], the authors proposed a solution based on multi-objective scheduling and a single site virtual machine provisioning approach, assuming homogeneous sites, as in

*Corresponding author
Email address: gaetan.heidsieck@inria.fr (Gaëtan Heidsieck)

a public cloud.

Since it is common for workflow users to reuse code or data from other workflows or previous executions of the same workflow [11], a promising approach for efficient workflow execution is to cache intermediate data in order to avoid entire re-execution. Furthermore, a user may need to re-execute a workflow many times with different values of parameters and input data depending on the previous results. When the same workflow is executed several times with different parameters, some workflow fragments, *i.e.*, subsets of workflow activities and dependencies, can be unchanged, so their intermediate data can be reused. Another important benefit of caching intermediate data is to make it easy for users to share it with other research teams, thus fostering new analyses at low cost.

Caching has been supported by some workflow systems, *e.g*, Kepler [12], VisTrails [13] and OpenAlea [14]. In [15], we proposed an adaptive caching method for OpenAlea that automatically determines the most suited intermediate data to cache, taking into account workflow fragments, but only in the case of a single site. Another interesting single site method, also exploiting workflow fragments, is to compute the ratio between re-computation cost and storage cost to determine what intermediate data should be stored [16]. All these methods are designed for a single site. Distributed caching in a multisite cloud is addressed in [17] to deal with hot metadata (frequently accessed metadata) only, not intermediate data.

Caching data in a multisite cloud with heterogeneous sites is much more complex. In addition to the trade-off between re-computation and storage cost at single sites, there is the problem of site selection for placing the cached data. The problem is more difficult than data allocation in distributed databases [18], which deals only with well-defined base data, not intermediate data. Furthermore, the scheduling of workflow executions must be cache-aware, *i.e.*, exploit the knowledge of cached data to decide whether reusing and transferring cached data or re-executing the workflow fragments.

In this paper, we propose a solution for cache-aware scheduling of scientific workflows in a multisite cloud. Our solution enables users to automatically store, share, and reuse intermediate data to speed up their future workflow executions in a multisite cloud. The solution is based on a distributed and parallel architecture and includes new algorithms for adaptive caching, cache site selection and dynamic workflow scheduling. We implemented our solution in OpenAlea. Based on a real data-intensive application in plant phenotyping (Phenomenal), we provide an extensive experimental evaluation using a cloud with three heterogeneous sites.

This paper is organized as follows. Section 2 presents our real use case in plant phenotyping. Section 3 introduces our workflow system architecture in a multisite cloud. Section 4 describes our caching solution. Section 5 gives our experimental evaluation. Section 6 discusses related work. Finally, Section 7 concludes.

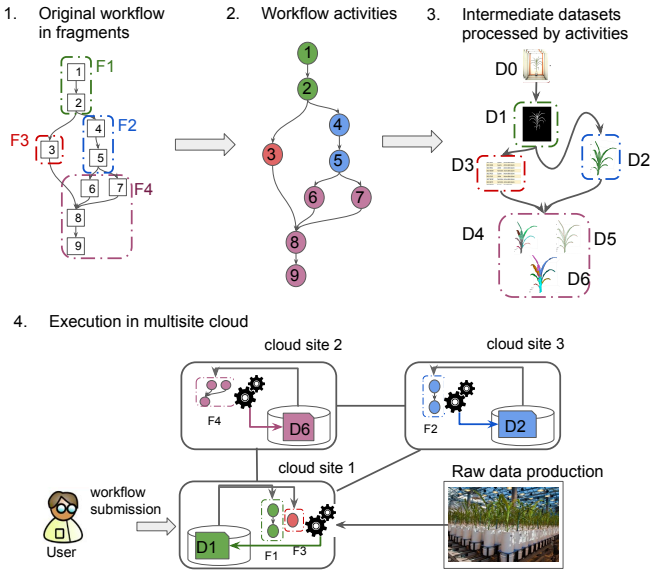


Figure 1: Phenomenal plant analysis workflow.

2. Use Case in Plant Phenotyping

In this section, we introduce a real use case in plant phenotyping that will serve as motivation for the work and basis for the experimental evaluation. In the last decade, high-throughput phenotyping platforms have emerged, allowing for the acquisition of quantitative data on thousands of plants in well-controlled environmental conditions. For instance, the seven facilities of the French Phenome project[1] produce each year 200 Terabytes of data, which are various (images, environmental conditions and sensor outputs), multiscale and coming from different sites. Analyzing such massive datasets is an open, yet important, problem for biologists [19].

The Phenomenal workflow [20], shown in Figure 1, has been developed in OpenAlea to analyze and reconstruct the geometry and topology of thousands of plants through time in various conditions. Phenomenal is continuously evolving with the addition of new state-of-the-art methods, thus yielding new biological insights. The workflow is composed of different fragments, *i.e.*, reusable subworkflows: binarization (circled in green), 3D volume reconstruction (blue), images calibration (red), and organ segmentation (purple). Figure 1.2 gives an abstract representation of the workflow, with the activities grouped by fragments *F1* to *F4*. The intermediate datasets, processed by the activities during execution are shown in Figure 1.3. Dataset D0 is the dataset of raw data that serves as input for the first fragment. Dataset D1 is generated by activity 2 and is the input of fragment 2 as it is processed by activity 4. The datasets are grouped by fragments.

The raw data is produced by the Phenoarch platform, which has a capacity of managing 1,680 plants within a controlled environment (*e.g*, temperature, humidity, irrigation)

[1] `https://www.phenome-emphasis.fr/phenome_eng/`

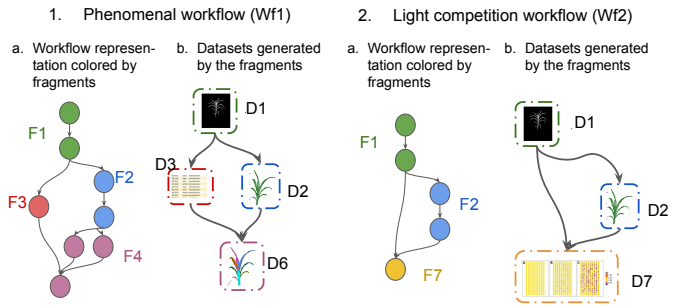


Figure 2: Two workflows in plant analysis and their intermediate data (the shared activities have same color).

and automatic imaging through time. The total size of the raw image dataset for one experiment is 11 Terabytes. The raw data is stored on a server close to the experimental platform. This server is considered as a site and has both data storage and computing resources. However, these resources may not be sufficient to perform a full workflow execution in a relatively short time. Thus, the solution is to use additional resources provided by other sites and execute the workflow in a distributed way on multiple sites.

The multisite cloud architecture (see Figure 1.4) is composed of heterogeneous sites, in terms of computing and storage resources. The site with the raw data is used to execute some Phenomenal fragments that do not require powerful resources. Whenever more computational resources are needed, it is necessary to choose whether transferring the raw data or some intermediate data to a more powerful site, or re-executing some fragments locally before transferring intermediate data.

Figure 1.4 illustrates a case where a user submits the Phenomenal workflow. The workflow fragments are distributed and executed on different sites depending on the site resources. At each site, some intermediate data generated by the fragment execution is stored in a cache to be reused in other fragment executions.

Different users can conduct different analyses by executing some workflow fragments on the same dataset to test different hypotheses [15]. To save both time and resources, it may be useful to reuse the corresponding intermediate data that has already been computed rather than recompute the fragments again. In our use case, we suppose that workflow executions are done in sequential order and that workflows are submitted from the site where the raw data is produced.

Figure 2 shows two workflows used in plant analysis: the Phenomenal workflow (Wf1) and a workflow to simulate light competition for plants in greenhouse (Wf2). Both workflows use fragments F1 (binarization) and F2 (3D reconstruction), so the subsequent execution of Wf2 may benefit from reusing the data generated previously from the corresponding fragments in Wf1. Suppose for instance that Wf1 execution has generated some data that has been cached, as shown in Figure 1.4. Then, a user can reuse datasets D1 and D2 to speed up the execution of Wf2. Thus, the only fragment that requires to be executed is F7.

3. Problem Definition and System Model

In this section, we start by giving an overview of distributed workflow execution. Based on this overview, we formulate the problem of cache-aware scheduling in a multisite cloud. Then, we present our workflow system architecture that integrates caching and reuse of intermediate data in a multisite cloud. We motivate our design decisions and describe our architecture in two ways: in terms of functional layers (see Figure 3), which shows the different functions and components; and in terms of nodes and components (see Figure 4), which are involved in the processing of workflows.

3.1. Problem Definition

We consider a multisite cloud with a set of sites $S=\{s_1, ..., s_n\}$. A workflow $W(A, D)$ is a directed acyclic graph (DAG) of computational activities A and their data dependencies D. A task t is the instantiation of an activity during execution with specific associated input data. A fragment f of an instantiated workflow is a subset of tasks and their dependencies.

We introduce basic cost functions to reflect data transfer and distributed execution. The time to transfer data d from site s_i to site s_j, noted $T_{tr}(d, s_i, s_j)$, is defined by

$$T_{tr}(d, s_i, s_j) = \frac{Size(d)}{TrRate(s_i, s_j)} \tag{1}$$

where $TrRate(s_i, s_j)$ is the transfer rate between s_i and s_j.

The time to transfer input and cached data, $In(f)$ and $Cached(f)$ respectively, to execute a fragment f at site s_i is $T_{input}(f, s_i)$:

$$T_{input}(f, s_i) = \sum_{s_j}^{S}(T_{tr}(In(f), s_j, s_i) + T_{tr}(Cached(f), s_j, s_i)) \tag{2}$$

Note that both the input data $In(f)$ and the cached data $Cached(f)$ used to execute a fragment can be distributed on several sites. The time to transfer the data considers the data transfer from all the different sites.

The time to compute a fragment f at site s, noted $T_{compute}(f, s)$, can be estimated using Amdahl's law [21]:

$$T_{compute}(f, s) = \frac{(\frac{\alpha}{n} + (1 - \alpha)) * W(f)}{P_{perf}(s)} \tag{3}$$

where $W(f)$ is the workload for the execution of f, P_{perf} (s) is the average computing performance of the processors at site s and n is the number of processors at site s. We assume that the local scheduler may parallelize task executions.

Therefore, α represents the percentage of the workload that can be executed in parallel.

The expected waiting time to be able to execute a fragment at site s is noted $T_{wait}(s)$, which is the minimum expected time for s to finish executing the fragments in its queue.

The time to transfer the intermediate data generated by fragment f at site s_i to site s_j, noted $T_{write}(Output(f), s_i, s_j)$, is defined by:

$$T_{write}(Output(f), s_i, s_j) = T_{tr}(Output(f), s_i, s_j) \quad (4)$$

where $Output(f)$ is the data generated by the execution of f.

Based on these different cost functions, we make three assumptions to define our scheduling problem:

- **A1.** The frequency of reusing each fragment is unknown. For each fragment execution, storing data into the cache has a cost (see Equation 4), which gets amortized only if it is reused. Thus, scheduling must take into account cache management.
- **A2.** The sites are heterogeneous and have limited storage and computing resources. For each fragment, the input data and cached data can be distributed on multiple sites. The time to retrieve data before execution (see Equation 2) can be significant. Thus, the scheduling decision should consider data transfers, for both cached and intermediate data.
- **A3.** The workflows are executed in sequential order. Thus, we do not consider concurrency in the data and resources access.

We focus on the problems of workflow scheduling and cache management. The workflow scheduling problem is to map each workflow fragment f for execution to a site in S while minimizing the execution time (from Equations 2 and 3). The cache management problem involves the decision of choosing which intermediate data should be added to the cache dynamically. However, the two problems are not independent. Workflow scheduling depends on cache data management as the cached data can be reused for execution but may require to be transferred to the execution site. On the other hand, cache data management depends on workflow scheduling as the intermediate data is generated on the execution site, which may not be the optimal site where to cache it. However, an efficient solution for one of these problems may not be optimal when considering the overall cost of workflow execution. Thus, our goal is to minimize this cost by managing both the workflow scheduling and the cache.

3.2. Workflow System Architecture

Our architecture capitalizes on the latest advances in distributed and parallel data management to offer performance and scalability [18]. We consider a distributed cloud architecture with on premise servers, where raw data is produced, *e.g*, by a phenotyping experimental platform in our use case, and remote sites, where the workflow is executed. The remote sites are data centers using shared-nothing clusters, *i.e.*, clusters of server machines, each with independent processors, disk and memory. We adopt shared-nothing as it is the most scalable and cost-effective architecture for big data analysis.

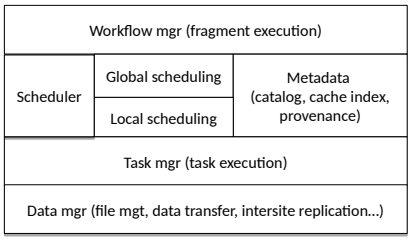


Figure 3: Workflow System Functional Architecture.

In the cloud, metadata is critical for workflow scheduling as it provides a global view of data location, *e.g*, at which nodes some raw data is stored, and enables task tracking during execution [17]. We organize the metadata in three repositories: catalog, provenance database and cache index. The catalog contains all information about users (access rights, etc.), raw data location and workflows (code libraries, application code). The provenance database captures all information about workflow specification and execution. The cache index contains information about tasks and cache data, as well as the location of files that store the cached data. Thus, the cache index itself is small (only file references) and the cached data can be managed using the underlying distributed file system. A good solution for implementing these metadata repositories is a key-value store, such as Cassandra[2], which provides efficient key-based access, scalability and high availability through replication in a shared-nothing cluster.

The raw data files are initially produced and stored at some sites, *e.g*, in our use case, at the phenotyping platform. During workflow execution, the intermediate data is generated and consumed at one site's node in memory. It gets written to disk when it must be transferred to another node (potentially at the same site) or explicitly added to the cache.

Figure 3 extends the workflow system architecture proposed in [22] for single site. It is composed of six modules: workflow manager, global scheduler, local scheduler, task manager, data manager and metadata manager, to support both execution and intermediate data caching in a multisite cloud. The workflow manager provides a user interface for workflow definition and processing. Before workflow execution, the user selects a number of virtual machines (VMs), given a set of possible instance formats, *i.e.*, the technical characteristics of the VMs, deployed on

[2] `https://cassandra.apache.org`

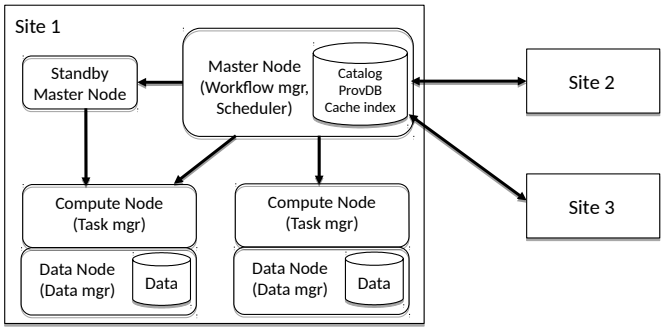


Figure 4: Multisite Workflow System Architecture.

each site's nodes. When a workflow execution is started, the workflow manager simplifies the workflow by removing some workflow fragments and partitions, depending on the raw input data and the cached data (see Section 4). The global scheduler uses the metadata (catalog, provenance database, and cache index) to schedule the workflow fragments of the simplified workflow. The VMs on each site are then initialized, *i.e.*, the programs required for the execution of the tasks are installed and all parameters are configured. The local scheduler schedules the workflow fragments received on its VMs.

The data manager module handles data transfers between sites during execution for both newly generated intermediate data and cached data, and manages cache storage. At a single site, data storage is distributed between cluster nodes. Finally, the task manager manages the execution of fragments on the VMs at each site. It exploits the provenance data to decide whether or not the task's output data should be placed in the cache, based on the cache provisioning algorithm (see Section 4). Local scheduling and execution can be performed as in [15].

Figure 4 shows how these components are organized, using the traditional master-worker model, in a multisite cloud. Each site provides the same functionality, *i.e.*, all the components described in Figure 3. Thus, users can trigger a workflow execution at any site. However, for a given workflow execution, there is one coordinator site, where the execution is started. The coordinator site performs workflow management and global scheduling, and manages the execution with other participant sites. The workflow manager and the global scheduler modules are involved only on the coordinator site while all other modules are involved on all sites.

At each site, there are three kinds of nodes: master, compute and data nodes, which are mapped to cluster nodes at configuration time, *e.g*, using a cluster manager like Yarn[3]. Each site has one active master node and a standby node to deal with master node failure. The master nodes are the only ones to communicate across sites. Each master node supports the top layers of the functional architecture: workflow manager, global scheduler, local scheduler and metadata management.

The master nodes are responsible for transferring data between sites during execution. They are lightly loaded as most of the work of serving clients is done by the compute and data nodes (or worker nodes), which perform local execution and data management, respectively.

4. Cache-aware Workflow Execution

In this section, we present in more details the cache-aware workflow execution in a multisite cloud. In particular, the global scheduler must decide which data to cache (cache data selection) where (cache site selection), and where to execute workflow fragments (execution site selection). Since these decisions are not independent, we propose a cost function to make a global decision, based on the cost components for individual decisions. First, we present the methods and cost functions for cache data selection, cache site selection, execution site selection, and global decision. Then, we introduce our algorithms for cache-aware scheduling.

The execution of a workflow $W(A, D)$ in S starts at a coordinator site s_c and proceeds in three main steps:

1. The global scheduler at s_c simplifies and partitions the workflow into fragments. Simplification uses metadata to decide whether a task can be replaced by corresponding cached data references. Partitioning uses the dependencies in D to produce fragments.
2. For each fragment, the global scheduler at s_c computes a cost function to make a global decision on which data to cache where, and on which site to execute. Then, it triggers fragment execution and caching of data at the selected sites.
3. At each selected site, the local scheduler performs the execution of the received fragments using its task manager (to execute tasks) and data manager (to transfer the required input data). It also applies the decision of the global scheduler on storing new intermediate data into the cache.

4.1. Workflow Simplification

Workflow simplification is performed by the workflow manager before execution, transforming the workflow into an executable workflow and considering the metadata, input, and cache data location. It is based on the workflow simplification method presented in [23].

First, the workflow $W(A, D)$ is transformed into an executable workflow $W_{ex}(A, D, T, Input)$, where T is a DAG of tasks corresponding to the activities in A and $Input$ is the input data. The goal is to transform an executable workflow $W_{ex}(A, D, T, Input)$ into an equivalent, simpler subworkflow $W'_{ex}(A', D', T', Input')$, where A' is a subgraph of A with dependencies D', T' is a subgraph of T corresponding to A' and $Input'$ is a subset of $Input$.

[3] `http://hadoop.apache.org`

The workflow simplification algorithm is recursive and traverses the DAG T starting from the sink tasks to the source tasks. The algorithm marks each task whose output is already in the cache. Then, the subgraphs of T that have each of their sink tasks marked are removed, and replaced by the associated data from the cache. The remaining graph is noted T'. Finally, the algorithm determines the fragments of T', *i.e.*, the subgraphs that still need to be executed.

4.2. Cache Data Selection

To determine what new intermediate data to cache, we consider two different methods: greedy and adaptive. Greedy data selection simply adds all new data to the cache. Adaptive data selection extends our method proposed in [15] to a multisite cloud. It achieves a good trade-off between the cost saved by reusing cached data and the cost incurred to feed the cache.

To determine whether it is worth adding some intermediate data $Output(f)$ at site s_j, we consider the trade-off between the cost of adding $Output(f)$ to the cache and the potential benefit if it was reused. The cost of adding $Output(f)$ to site s_j is the time to transfer it from where it was generated, say site s_i The potential benefit is the time saved from loading $Output(f)$ from s_j to the site of computation instead of re-executing the fragment. We model this trade-off with the ratio between the cost and benefit of the cache, noted $p(f, s_i, s_j)$, which can be computed from Equations 2, 3 and 4,

$$p(f, s_i, s_j) = \frac{T_{write}(Output(f), s_i, s_j)}{\begin{array}{c} T_{input}(f, s_i) + T_{compute}(f, s_i) \\ -T_{tr}(Output(f), s_j, s_i) \end{array}} \quad (5)$$

In the case of multiple users, the probability that $Output(f)$ will be reused or the number of times fragment f will be re-executed is not known when the workflow is executed. Thus, we introduce a threshold $Threshold$ (computed on behalf of the user) as the limit value to decide whether a fragment output will be added to the cache. The decision on whether $Output(f)$ generated at site s_i is stored at site s_j can be expressed by

$$d_{f,i,j} = \begin{cases} 1 & \text{if } p(f, s_i, s_j) < Threshold. \\ 0 & \text{otherwise.} \end{cases} \quad (6)$$

4.3. Cache Site Selection

Cache site selection must take into account the data transfer cost and the heterogeneity of computing and storage resources. We propose two methods to balance either storage load (*bStorage*) or computation load (*bCompute*) between sites. The *bStorage* method prevents bottlenecks when loading cached data. To assess this method at any site s, we use a load indicator, noted $L_{bStorage}(s)$, which represents the relative storage load as the ratio between the storage used for the cached data ($Storage_{used}(s)$) and the total storage ($Storage_{total}(s)$).

$$L_{bStorage}(s) = \frac{Storage_{used}(s)}{Storage_{total}(s)} \quad (7)$$

The *bCompute* method balances the cached data between the most powerful sites, *i.e.*, with more processors, to prevent computing bottlenecks during execution. Using the knowledge on the sites' computing resources and usage, we use a load indicator for each site s, noted $L_{bCompute}(s)$, based on processors idleness ($P_{idle}(s)$) versus total processor capacity ($P_{total}(s)$).

$$L_{bCompute}(s) = \frac{1 - P_{idle}(s)}{P_{total}(s)} \quad (8)$$

The load of a site s, depending on the method used, is represented by $L(s)$, ranging between 0 (empty load) and 1 (full). Given a fragment f executed at site s_i, and a set of sites $\{s_j\}$ with enough storage for $Output(f)$, the best site s^* to add $Output(f)$ to its cache can be obtained using Equation 1 (to include transfer time) and Equation 6 (to consider multiple users),

$$s^*(f)_{s_i} = \underset{s_j}{\text{argmax}}(d_{f,i,j} * \frac{(1 - L(s_j))}{T_{write}(Output(f), s_i, s_j)}) \quad (9)$$

4.4. Execution Site Selection

To select an execution site s for a fragment f, we need to estimate the execution time for f as well as the time to feed the cache with the result of executing f. The execution time of f at site s ($T_{execute}(f, s)$) is the sum of the time to transfer input and cached data to s, the time to get computing resources and the time to compute the fragment. It is obtained using Equations 2 and 3.

$$T_{execute}(f, s) = T_{input}(f, s) + T_{compute}(f, s) + T_{wait}(s) \quad (10)$$

Given a fragment f executed at site s_i and its intermediate data $Output(f)$, the time to write $Output(f)$ to the cache (T_{feed_cache} (f, s_i, s_j)) can be defined as:

$$T_{feed\_cache}(f, s_i, s_j, d_{f,i,j}) = d_{f,i,j} * T_{write}(Output(f), s_i, s_j) \quad (11)$$

where s_j is given by Equation 9.

4.5. Global Decision

At Step 2 of workflow execution, for each fragment f, the global scheduler must decide on the best combination of individual decisions regarding cache data, cache site, and execution site. These individual decisions depend on each other. The decision on cache data depends on the site where the data is generated and the site where it will be stored. The decision on cache site depends on the site

where the data is generated and the decision of whether or not the data will be cached. Finally, the decision on execution site depends on what data will be added to the cache and at which site. Using Equations 10 and 11, we can estimate the total time (T_{total}) for executing a fragment f at site s_i and adding its intermediate data to the cache at another site s_j:

$$T_{total}(f, s_i, s_j, d_{f,i,j}) = T_{execute}(f, s_i) + T_{feed\_cache}(f, s_i, s_j, d_{f,i,j}) \quad (12)$$

Then, the global decision for cache data ($d_{f,i,j}$), cache site (s^*_{cache}) and execution site (s^*_{exec}) implies minimizing the following equation for the n^2 pairs of sites s_i and s_j

$$(s^*_{exec}, s^*_{cache}, d_{f,i,j}) = \underset{s_i,s_j}{\operatorname{argmin}}(T_{total}(f, s_i, s_j, d_{f,i,j})) \quad (13)$$

This decision is performed by the coordinator site before each fragment execution. It only takes into account the site's status at that time. Note that s^*_{exec} and s^*_{cache} can be the same site, including the coordinator site.

4.6. Cache-Aware Scheduling

In this section, we present in details our solution to cache-aware scheduling in our architecture. We propose three algorithms: *GlobalGreedyCache*, *SiteGreedyCache* and *FragGreedyCache*. *GlobalGreedyCache* is a new greedy algorithm that performs cache-aware scheduling. The two other algorithms extend distributed greedy scheduling algorithms [10] to become cache-aware. These three algorithms are dynamic in that they produce scheduling plans that distribute and allocate executable tasks to computing nodes during workflow execution [22]. This kind of scheduling is appropriate for our workflows, where the workload is difficult to estimate, or for environments where the capabilities of the computers varies much during execution.

4.6.1. GlobalGreedyCache.

The *GlobalGreedyCache* algorithm (see Algorithm 1) is based on the global decision (see Equation 13) made by the coordinator site. It takes the simplified workflow graph as input and, starting from the root fragment, computes the global decision for each fragment. Recall that the global decision combines individual decisions regarding cache data, cache site, and execution site, before scheduling each fragment.

Algorithm 1 proceeds as follows. The workflow is partitioned into fragments (line 1), where F represents the set of all fragments of the workflow. Whenever a fragment is ready for execution, it is selected (line 3). Then (line 4), the global decision is computed using Equation 13 to determine the best execution site S_{exec}, cache placement site S_{cache} and cache decision $d_{f,i,j}$. At line 5, the fragment is transferred to the site S_{exec} to be executed. Recall that the cache decision $d_{f,i,j}$ determines whether the intermediate data will be cached. Whenever the intermediate data is to be stored in the cache (lines 6-8), it is transferred at site S_{cache} (line 7). At line 8, the *Cache Index* is updated locally and the update is propagated at all replicas at other sites. Finally (line 11), the fragment is removed from F.

Algorithm 1: GlobalGreedyCache

```
   Input: WF: a workflow,
   Cache index: the index of the placement of the
   data existing in the cache
1  F ← partition WF into fragment;
2  while F not empty do
3      f ← select a fragment of F that is next to be
         computed ;
4      S_exec, S_cache, d_f,i,j ← compute from Equation
         13 ;
5      Schedule f execution on site S_exec ;
6      if d_f,i,j is True then
           /* The intermediate data is cached
              */
7          Place the intermediate data on site S_cache;
8          Update the Cache Index
9      else
           /* The intermediate data will not be
              cached                          */
10     end
11     Remove f from F;
12 end
```

4.6.2. SiteGreedyCache and FragGreedyCache.

SiteGreedyCache (site greedy with caching) extends the *SiteGreedy* algorithm presented in [10]. The scheduling decision of *SiteGreedy* works as follows. Let F be the set of workflow fragments. Whenever a site s is available, it requests the execution of a ready fragment in F to the coordinator site. The selection of the fragment is based on a cost function that takes into account data transfer time and execution time. The idea of this scheduling is to keep sites as busy as possible, scheduling a fragment whenever a site is available. The caching decision is done after the workflow fragment is scheduled for execution by the local scheduler at each site. Unlike *GlobalGreedyCache*, *SiteGreedyCache* does not make a global decision. Instead, the caching decision is done in two steps. First, the choice of the site where the cached data should be stored is determined by Equation 9. Then, the decision on whether or not to store the intermediate data is determined by Equation 5, considering the execution time and the time to transfer the intermediate data. Note that Equation 5 considers both the execution site and the cache site, which are already determined when computed during the execution of *SiteGreedyCache*.

FragGreedyCache (fragment greedy with caching) extends the *ActGreedy* algorithm presented in [10]. *FragGreedyCache* schedules each workflow fragment at the site

that minimizes a cost function based on execution time and input data transfer time. The cost function is the sum of the initialization time, expected execution time, and data transfer time for each fragment at each site. Then, after each fragment execution at the selected site, the local scheduler performs the cache site and cache data decisions based on Equations 9 and 5.

These two greedy algorithms generate a dynamic scheduling plan. After each fragment execution, the decision concerning caching is made by the local scheduler. In contrast, *GlobalGreedyCache* makes a global decision for each fragment. Note that *SiteGreedyCache* and *FragGreedyCache* needs less operations to schedule the fragments.

4.6.3. Analysis.

In this section, we analyze the runtime and storage complexity of the *GlobalGreedyCache* algorithm.

Runtime complexity analysis. Let n be the number of given plant image sets. Our algorithm generates a constant number of tasks for each set of images, *i.e.*, less than 14 tasks per set. Let $|T|$ be the number of generated tasks, then we have $|T| = n * c$, where $c <= 14$ is a constant value. Our algorithm groups tasks into fragments to be scheduled, generating at most $|T|$ fragments. Our algorithm performs one scheduling operation for each fragment, and the time needed for each operation is constant. Thus, the time complexity of the algorithm is $O(|T|)$. Since we have $|T| = n * c$ and c is a constant, the time complexity of the algorithm is $O(n)$, where n is the number of sets of images.

Storage complexity analysis. The largest data structure used by our algorithm is a list containing the fragments. As before, the number of fragment generated by the algorithm is at most $|T|$. Thus, the storage complexity of the algorithm is $O(|T|)$, so $O(n)$, where n is the number of sets of images.

5. Experimental Evaluation

In this section, we first present our experimental setup, which features an heterogeneous multisite cloud with multiple users that re-execute part of the workflow. Then, we compare the performance of our multisite cache scheduling algorithms against two baseline algorithms. We end the section with concluding remarks.

5.1. Experimental Setup

We use a real multisite cloud, with three sites, in France. *Site 1* in Montpellier is the raw data server of the Phenoarch phenotyping platform, with the smallest number of processors and largest amount of storage among the sites. The raw data is stored at this site. *Site 2* is the coordinator site, located in Lille. *Site 3*, located in Lyon, has the largest number of processors and the smallest amount of storage.

To model site heterogeneity in terms of storage and computing resources, we use heterogeneity factor H in three configurations: $H = 0$, $H = 0.3$ and $H = 0.7$. For the three sites altogether, the total number of processors is 96 and the total storage 180 GB. With $H = 0$ (homogeneous configuration), each site has 32 processors and 60 GB. With $H = 0.3$, we have 22 processors and 83 GB for Site 1, 30 processors and 57 GB for Site 2 and 44 processors and 40 GB for Site 3. With $H = 0.7$ (most heterogeneous configuration), we have 6 processors and 135 GB for Site 1, 23 processors and 35 GB for Site 2 and 67 processors and 10 GB for Site 3.

To determine the data transfer rate between the sites, each site sends to the other sites a 10 MB test file and measures the time. This operation is repeated every 30 minutes and the information is updated at the coordinator site.

The workflow we use for testing is the Phenomenal workflow (presented in Section 2). It is composed of 9 main activities. The input dataset for the Phenomenal workflow is produced by the Phenoarch platform (see Section 2). Each execution of the workflow is performed on a subset of the input dataset, *i.e.*, 200 GB of raw data, which represents the execution of 15,000 tasks. The workflow is executed by several users. Each user wants the results produced by the last activity, which requires the execution of all other activities when executed from scratch. For each user, 60% of the raw data is reused from previous executions. Thus, each execution requires only 40% of new raw data. For the first execution, no data is available in the cache.

For the experiments 3, 4 and 5 the workflow executions have been performed three times. Presented values are the average of the three executions.

We implemented our solution in OpenAlea and deployed it at each site using the Conda multi-OS package manager. The metadata database is implemented using the Cassandra NoSQL data store. Communication between the sites is done using the protocol library ZeroMQ. Data transfer between sites is done through SSH. We have also implemented two baseline scheduling methods: 1) *ActGreedy*, a multisite scheduling algorithm [10] that schedules the fragments at multiple sites given a cost function based on execution time and input data transfer time. This algorithm is not cache aware and does not reuse intermediate data for future executions. 2) a centralized version of the three cache-aware scheduling algorithms proposed in this paper(*GlobalGreedyCache*, *SiteGreedyCache* and *FragGreedyCache*). In our experiments, the centralized cache is managed on Site 1.

Table 1 summarizes the different variants of the scheduling algorithms used in our experiments. Prefix "C-" indicates that the cache is centralized at a single site while prefix "D-" that it is distributed. For all algorithms that use a cache, the cache index is fully replicated at all sites.

5.2. Experiments

We compare the three algorithms we proposed (*GlobalGreedyCache*, *SiteGreedyCache*, *FragGreedyCache*) in terms

Table 1: Scheduling algorithms and their main dimensions.

Algorithm	Cost function parameters	Cache decision	Cache placement
ActGreedy	Execution time of Activity & input transfer time	Local after execution	No cache
C-GlobalGreedyCache (C-G)	Execution time of Fragments (Frag.) Input & Cache transfer time	Global per frag. before execution	Single cache site
C-SiteGreedyCache (C-S)	Execution time Frag. Input transfer time	Local after execution	Single cache site
C-FragGreedyCache (C-F)	Execution time Frag. Input transfer time	Local after execution	Single cache site
D-GlobalGreedyCache (D-G)	Execution time Frag. Input & Cache transfer time	Global per frag. before execution	Distributed
D-SiteGreedyCache (D-S)	Execution time Frag. Input transfer time	Local after execution	Distributed
D-FragGreedyCache (D-F)	Execution time Frag. Input transfer time	Local after execution	Distributed

of execution time and amount of data transferred with two baselines. The total time is defined as the workflow execution time plus the transfer time. We consider different workflow executions: with and without caching (Experiment 1); on a monosite or a multisite cloud (Experiment 2); and using a centralized or distributed cache (Experiment 3). Then, we consider multiple users that execute the workflow in the following cases: on the same multisite configuration, where 60% of the data is the same (Experiment 4); on different multisite configurations (Experiment 5); and when adding or removing workflow fragments (Experiment 6).

5.2.1. Experiment 1: with and without caching.

The goal of this experiment is to show that reusing cached data can speed up workflow execution while caching data is also time consuming. Thus, without a minimum amount of cached data reused, the cost of having a cache exceeds the benefits. To do so, we compare two workflow executions: with caching, using the *D-GlobalGreedyCache* scheduling algorithm and the *bStorage* load balancing method; and without caching, using the *ActGreedy* algorithm. We consider one re-execution of the workflow on different input datasets, from 0% to 60% of data reuse. *D-GlobalGreedyCache* outperforms *ActGreedy* from 20% of reused data. Below 20%, the overhead of caching outweighs its benefit. For instance, with no reuse (0%), the total time with *D-GlobalGreedyCache* is 16% higher than with *ActGreedy*. But with 30%, it is 11% lower, and with 60%, it is 42% lower.

5.2.2. Experiment 2: single site versus multisite execution.

The goal of this experiment is to show, in the case of sites with limited resources, that increasing the number of sites reduces the workflow execution total time despite increased data transfers and network latencies. To do so, we compare the total time in four cases with monosite and multisite clouds:

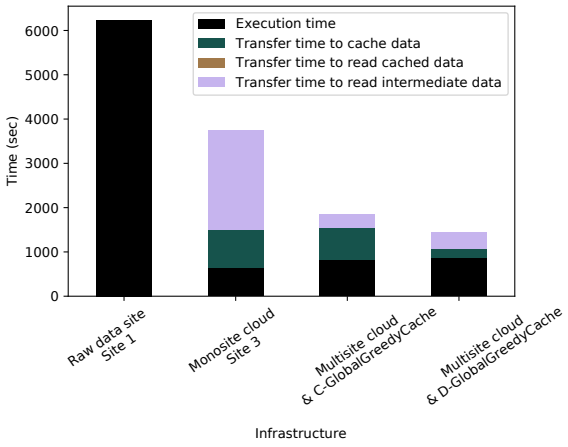


Figure 5: Total time of Phenomenal workflow execution in four cases.

1. a raw data site (Site 1), with only 10 processors, where the raw data is stored;
2. another site (Site 3) with 96 processors, which can perform computation on the raw and cache data (that needs to be transferred from Site 1);
3. a multisite cloud composed of three sites with configuration $H = 0.7$, using *C-FragGreedyCache*;
4. the same multisite cloud but using *D-GlobalGreedyCache*.

Figure 5 shows the total time of the workflow for the different cases. When executing on the raw data site (first chart on Figure 5), all the input data is already stored on Site 1 as well as the cached data, thus there is no data transfer between sites during workflow execution. However, due to the reduced number of available processors, the total time is by far longer than on any other infrastructure (66% longer than the execution on Site 3, 238% longer than the multisite execution with *C-FragGreedyCache* and 334% longer than the multisite execution with *D-GlobalGreedyCache*). An execution of the workflow on the full raw dataset on Site 1 takes more than a week to compute.

In practice, the raw dataset is first sent to a site with more computing resources available before being executed.

The execution on Site 3 yields the shortest execution time, outperforming the multisite execution with *C-FragGreedyCache* and *D-GlobalGreedyCache* in terms of execution time by 21% and 26% respectively. However, the time for transferring the raw data makes its total time much longer, so it is outperformed by the multisite execution using *D-GlobalGreedyCache* by 61%.

The intermediate data transfer time on the multisite cloud is much smaller (83% smaller for the execution with *D-GlobalGreedyCache*) than the raw data transfer time of the execution on Site 3. In a multisite cloud, fragments can be executed on the site of their input data. In this case, the raw data is not transferred between sites, but locally processed on Site 1 by the first workflow fragment. The intermediate data generated by the first fragment is smaller than the raw data and is more easily transferred to other sites, where the other fragments are scheduled.

5.2.3. Experiment 3: centralized versus distributed cache.

The goal of this experiment is to show that in a multisite cloud, distributing the cache enables reducing significantly the data transfer times, as well as the total time. Figure 6 shows the total time of the workflow for the three algorithms *SiteGreedyCache*, *FragGreedyCache* and *GlobalGreedyCache*. The algorithms used with a centralized cache on Site 1 are *C-SiteGreedyCache*, *C-FragGreedyCache* and *C-GlobalGreedyCache*. They are compared with *D-GlobalGreedyCache*, which uses a distributed cache in two configurations: (a) with two users; (b) with different site heterogeneity.

Let us first analyze the results of Figure 6.a, where two users execute the Phenomenal workflow with 60% of common raw data in two configurations: centralized cache on Site 1 and distributed cache with $H = 0.7$. For the first user execution, *D-GlobalGreedyCache* outperforms *C-SiteGreedyCache* in terms of total time by 44%. This is due to *D-GlobalGreedyCache* outperforming *C-SiteGreedyCache* in terms of intermediate and cache data transfer times by 66% and 60% respectively. *D-GlobalGreedyCache* outperforms *C-FragGreedyCache* in terms of total time by 24%, even though *D-GlobalGreedyCache*'s execution time is lower than *C-FragGreedyCache* (5%). This is due to *D-GlobalGreedyCache* outperforming *C-FragGreedyCache* in terms of data transfer time by 44%. *D-GlobalGreedyCache* outperforms *C-GlobalGreedyCache* in terms of total time by 15%. The execution time and intermediate data transfer time are similar (17% shorter and 11% longer). Yet *D-GlobalGreedyCache* outperforms *C-GlobalGreedyCache* in terms of cache data transfer by 32%. For the first execution *D-GlobalGreedyCache* outperforms the three algorithms with centralized cache, mostly due to shorter data transfer times. This is because the distributed cache enables executing the workflow at a site with more computing resources and storing the cached data on that site. For re-execution, *D-GlobalGreedyCache* outperforms the three algorithms with centralized cache, *C-SiteGreedyCache*, *C-FragGreedyCache* and *C-GlobalGreedyCache*, in terms of total time by 63%, 47% and 23%, respectively.

Figure 6.b shows the total time of the workflow for the second user and the four different algorithms: *C-SiteGreedyCache*, *C-FragGreedyCache*, *C-GlobalGreedyCache* and *D-GlobalGreedyCache*. The execution is performed on three values of H in two configurations: centralized cache on Site 1 and distributed cache. In any configuration, *D-GlobalGreedyCache* outperforms the three other algorithms with centralized cache, *C-SiteGreedyCache*, *C-FragGreedyCache* and *C-GlobalGreedyCache*, in terms of total time by 44%, 33% and 17%, respectively for $H = 0$, by 53%, 40% and 25%, respectively for $H = 0.3$, and by 61%, 44% and 22%, respectively for $H = 0.7$. The performance gain is due to less data transfers.

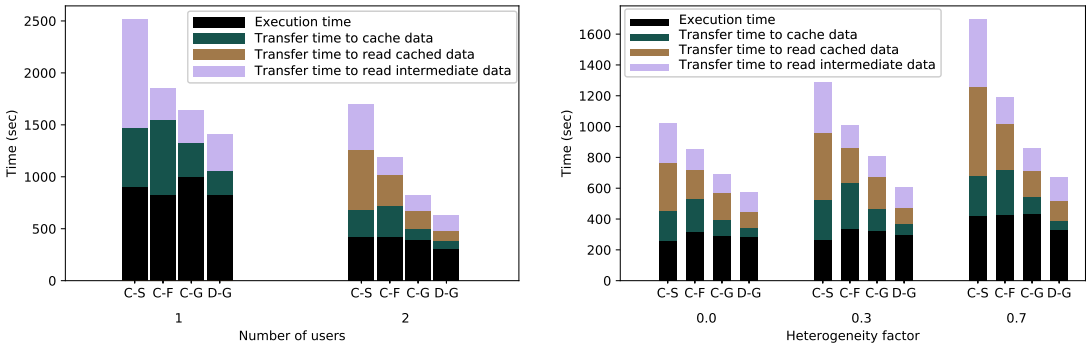
5.2.4. Experiment 4: multiple users.

The goal of this experiment is to show that the proposed algorithms reduce the workflow execution total time in the case of multiple users executing the workflow. Figure 8 shows the total time of the workflow for the three scheduling algorithms, four users, $H = 0.7$, and our two cache site selection methods: (a) *bStorage*, and (b) *bCompute*.

Let us first analyze the results in Figure 8.a (*bStorage* method). For the first user execution, *D-GlobalGreedyCache* outperforms *D-SiteGreedyCache* in terms of execution time by 10% and in terms of data transfer time by 36%. The reason that *D-SiteGreedyCache* is slower is because it schedules some compute-intensive fragments at Site 1, which has the lowest computing resources. Furthermore, it does not consider data placement and transfer time when scheduling fragments.

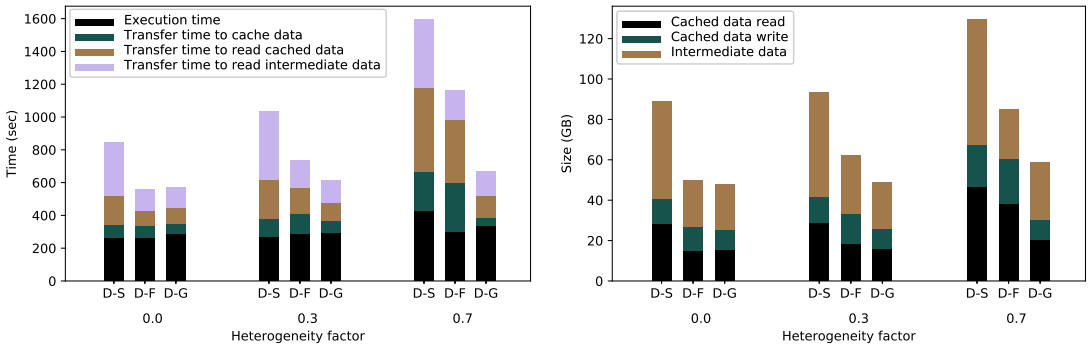
Again for the first user execution, *D-GlobalGreedyCache* outperforms *D-FragGreedyCache* in terms of total time by 20%, when considering the time to transfer data the cache. However, its execution time is a bit slower (by 11%). The reason that *D-FragGreedyCache* is slower is that it does not take into account the placement of the cached data, which leads to larger amounts (by 66%) of cache data to transfer. For other users' executions (when cached data exists), *D-GlobalGreedyCache* outperforms *D-SiteGreedyCache* in terms of execution time by 29%, and for the fourth user execution by 31%. This is because *D-GlobalGreedyCache* better selects the cache site in order to reduce the execution time of the future re-executions. Furthermore, *D-GlobalGreedyCache* balances the cached data and computations. It outperforms *D-SiteGreedyCache* and *D-FragGreedyCache* in terms of intermediate data transfer time (by 63% and 11%, respectively) and cache data transfer time (by 78% and 69%, respectively).

Overall, *D-GlobalGreedyCache* outperforms *D-SiteGreedyCache* and *D-FragGreedyCache* in terms of total time by 61% and 41%, respectively. The workflow fragments are not necessarily scheduled to the site with shortest execution time, but to the site that minimizes overall total time. Considering the multiuser perspective, *D-GlobalGreedyCache*



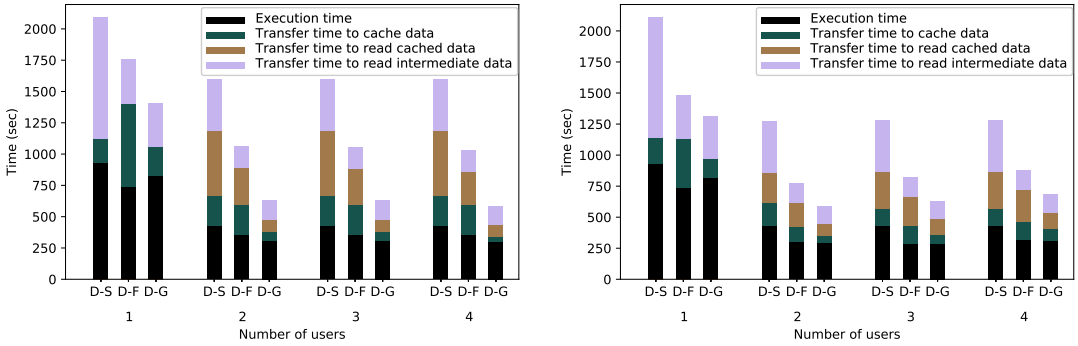
(a) *Two users* (b) *Site heterogeneity*

Figure 6: Centralized versus distributed cache in terms of execution time. Three scheduling algorithms with centralized cache: *C-SiteGreedyCache* (C-S), *C-FragGreedyCache* (C-F) and *C-GlobalGreedyCache* (C-G), and one with distributed cache: *D-GlobalGreedyCache* (D-G).



(a) Total time (b) Amount of data transfer

Figure 7: Execution for one user (60% of same raw data used) on heterogeneous sites with three scheduling algorithms (*D-SiteGreedyCache* (D-S), *D-FragGreedyCache* (D-F) and *D-GlobalGreedyCache* (D-G)).



(a) *bStorage method* (b) *bCompute method*

Figure 8: Total times for multiple users (60% of same raw data per user) for three scheduling algorithms (*D-SiteGreedyCache* (D-S), *D-FragGreedyCache* (D-F) and *D-GlobalGreedyCache* (D-G)).

outperforms *D-SiteGreedyCache* and *D-FragGreedyCache*, reducing the total time for each new user.

Let us now consider Figure 8.b (*bCompute* method). For the first user execution, *D-GlobalGreedyCache* outperforms *D-SiteGreedyCache* and *D-FragGreedyCache* in terms of total time by 38% and 12% respectively. *bCompute* stores the cache data on the site with the most idle processors, which is often the site with the most processors. This leads the cached data to be stored close to where it is generated, thus reducing data transfers when adding data to the cache. For the second user, *D-GlobalGreedyCache* outperforms *D-SiteGreedyCache* and *D-FragGreedyCache* in terms of total time by 54% and 24% respectively. The cached data generated by the first user is stored on the sites with more available processors, which minimizes the transfers of intermediate and cached data. From the third user, the storage at some site gets full, *i.e.*, for the third user's execution, Site 3's storage is full and from the fourth user's execution, Site 2's storage is full. Thus, the performance of the three scheduling algorithms decreases due to higher cache data transfer time. Yet, *D-GlobalGreedyCache* outperforms *D-SiteGreedyCache* and *D-FragGreedyCache* in terms of total time by 47% and 22% respectively.

5.2.5. Experiment 5: site heterogeneity.

We now compare the three algorithms in the case of heterogeneous sites by considering the amount of data transferred and execution time. In this experiment (see Figure 7), we consider one user with the cache already provisioned by previous executions on 60% of the same raw data. We use the *bStorage* method for cache site selection.

Figure 7 shows the execution times and the amount of data transferred using the three scheduling algorithms. With homogeneous sites ($H = 0$), the three algorithms have almost the same execution time. *D-GlobalGreedyCache* outperforms *D-SiteGreedyCache* in terms of amount of data transferred and total time by 47% and 32%, respectively. The execution time of *D-GlobalGreedyCache* is similar to *D-FragGreedyCache* (9% longer). The cached data is balanced as the three sites have same storage capacities. Thus, total times of *D-GlobalGreedyCache* and *D-FragGreedyCache* are almost the same.

With heterogeneous sites ($H > 0$), the sites with more processors have less available storage but can execute more tasks, which leads to a larger amount of intermediate and cached data being transferred between the sites. For $H = 0.3$, *D-GlobalGreedyCache* outperforms *D-SiteGreedyCache* and *D-FragGreedyCache* in terms of total time (by 41% and 17%, respectively) and amount of data transferred (by 48% and 21%, respectively).

With $H = 0.7$, *D-GlobalGreedyCache* outperforms *D-SiteGreedyCache* and *D-FragGreedyCache* in terms of total time (by 58% and 42%, respectively) and in terms of amount of data transferred (by 55% and 31%, respectively). *D-GlobalGreedyCache* is faster because its scheduling leads to a smaller amount of cached data transferred when reused (50% smaller than *D-FragGreedyCache*) and added to the cache (57% smaller than *D-FragGreedyCache*).

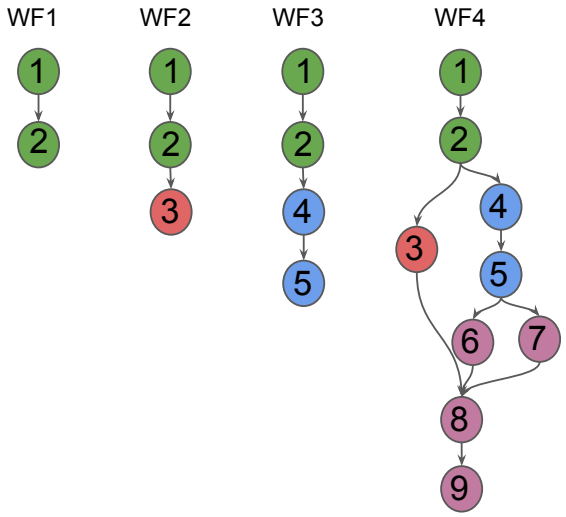
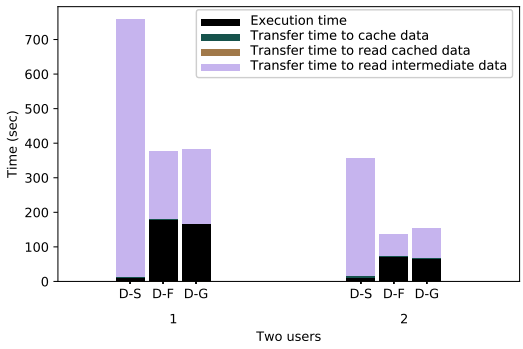


Figure 9: Four subworkflows derived from the Phenomenal workflow.

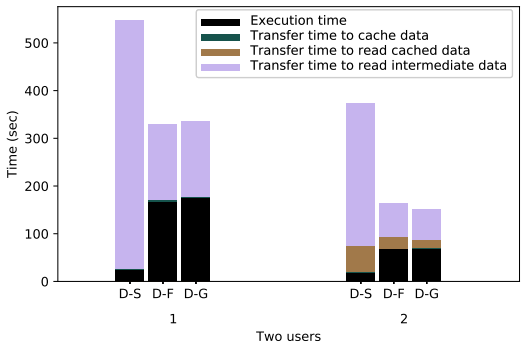
5.2.6. Experiment 6: adding and removing fragments.

In this experiment, we evaluate how our approach performs in terms of total time when subworkflows (with common fragments) derived from the Phenomenal workflow are executed independently. Figure 9 shows four subworkflows, each corresponding to a different analysis required by the user: WF1 performs image binarization, WF2 generates an analysis of the binary images, WF3 generates a 3D reconstruction of the plant and WF4 performs maize analysis. WF1 is mostly data-intensive, the image binarization fragment performing little computation but consuming Terabytes of data. WF2 requires more computational resources but is still mostly data-intensive. Fragment F3 in WF3 (composed of activities 3 and 4 in Figure 9) is mostly computation-intensive. Finally, WF4 is both data- and computation-intensive. The subworkflows are executed two times starting without cached data and 60% of the raw input data is common between the users. Each user wants the output data generated by the last activity of the workflow, *i.e.* activity 2 for WF1, activity 3 for WF2, activity 5 for WF3, and activity for WF9. All executions use method *bStorage*.

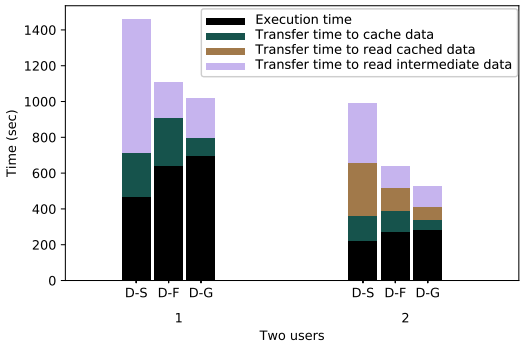
Figure 10 shows the total times for executing WF1, WF2, WF3 and WF4 by two users, one after the other. The first user executes the subworkflow without existing cached data, then the second user executes the subworkflow using 60% of the same raw data. In the case of WF1 (see Figure 10a), *D-SiteGreedyCache* outperforms both *D-FragGreedyCache* and *D-GlobalGreedyCache* in terms of execution times by 92% for the first user and by 83% for the second user. This is because*D-SiteGreedyCache* uses all processors at all sites, whereas *D-FragGreedyCache* and *D-GlobalGreedyCache* almost only use the processors



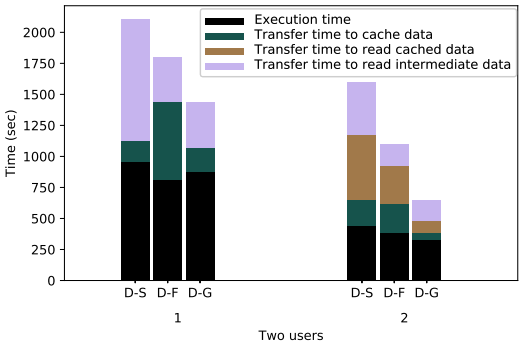
(a) *WF1*



(b) *WF2*



(c) *WF3*



(d) *WF4*

Figure 10: Total times for executing the four subworkflows by two users (with 60% of same raw data for second user) with three algorithms (*D-SiteGreedyCache* (D-S), *D-FragGreedyCache* (D-F) and *D-GlobalGreedyCache* (D-G)).

of Site 1 (where the raw input data is). However, *D-GlobalGreedyCache* transfers less intermediate data (70% less) during execution, which makes *D-GlobalGreedyCache* outperforming *D-Sgreedy* in terms of total time by 49%. *D-GlobalGreedyCache* and *D-FragGreedyCache* have similar total times (*D-GlobalGreedyCache* is outperformed by only *D-FragGreedyCache* by 2%). Since WF1 is mostly data-intensive, both methods *D-GlobalGreedyCache* and *D-FragGreedyCache* try to execute the workflow at the site where the input data is.

In the case of WF2 (see Figure 10b), *D-SiteGreedyCache* also outperforms both *D-FragGreedyCache* and *D-GlobalGreedyCache* in terms of execution times by 91% for the first user and by 79% for the second user. This is because the added fragment can be executed right after the execution of WF1 without delay as it does not required much computational resources. However, *D-GlobalGreedyCache* outperforms *D-SiteGreedyCache* in terms of total time by 39% due to longer data transfer times with *D-SiteGreedyCache*. *D-GlobalGreedyCache* and *D-FragGreedyCache* also have similar total times, *D-GlobalGreedyCache* outperforms *D-FragGreedyCache* by 4% for the second user. WF1 and WF2 are both data-intensive, not compute-intensive. Since the raw data is stored on Site 1, the site with the most storage capacities, it is the most likely to be used as cache site. For these subworkflows, the selection of the execution site by both algorithms *D-FragGreedyCache* and *D-GlobalGreedyCache* depends mostly on the intermediate data location. In this case, they make similar decisions, and thus have similar performance.

In the case of WF3 (see Figure 10c), for the first user, *D-GlobalGreedyCache* outperforms both *D-SiteGreedyCache* and *D-FragGreedyCache* in terms of total time by 30% and 8% respectively. Then, for the second user, *D-GlobalGreedyCache* also outperforms both *D-SiteGreedyCache* and *D-FragGreedyCache* by 47% and 18% respectively. This is due to *D-GlobalGreedyCache* outperforming *D-SiteGreedyCache* and *D-FragGreedyCache* in terms of data transfers by 69% and 35% respectively. *D-FragGreedyCache* selects the best sites that minimize execution times and intermediate data transfer times. In the case of WF3, which has a compute-intensive fragment, most of the computation will be scheduled on the site with the most computational resources. *D-GlobalGreedyCache*, however, will schedule some of the computation to the sites where the intermediate data will be cached and these sites may have less computational resources. This is why *D-FragGreedyCache* has shorter execution time, but is outperformed by *D-GlobalGreedyCache* in terms of total time.

In the case of WF4 (see Figure 10d), *D-GlobalGreedyCache* outperforms both *D-SiteGreedyCache* and *D-FragGreedyCache* by 32% and 24% for the first user and 60% and 40% for the second one respectively. In the case of WF4, the fragments are both data- and compute-intensive, thus the scheduling decision becomes more complex. *D-FragGreedyCache* is scheduling fragments to minimize the execution and intermediate data transfer times, which leads *D-FragGreedyCache* to outperform *D-GlobalGreedyCache* in terms of execution time and intermediate data transfer by 8%. However, the intermediate data that will be cached is bigger than in WF3, and the time to transfer the cached data becomes a major element of the total time. This is why *D-GlobalGreedyCache* outperforms *D-SiteGreedyCache* and *D-FragGreedyCache* in terms of total time.

5.3. Concluding Remarks

The main result of this experimental evaluation is that *GlobalGreedyCache* always outperforms the two greedy algorithms *SiteGreedyCache* and *FragGreedyCache*, both in the case of multiple users and heterogeneous sites.

The first experiment (with or without caching) shows that storing and reusing cached data becomes beneficial when 20% or more of the input data is reused. The fourth experiment (multiple users) shows that *D-GlobalGreedyCache* outperforms *D-SiteGreedyCache* and *D-FragGreedyCache* in terms of total time by up to 61% and 41%, respectively. It also shows that, with increasing numbers of users, the performance of the three scheduling algorithms decreases due to higher cache data transfer times. The fifth experiment (heterogeneous sites) shows that *D-GlobalGreedyCache* adapts well to site heterogeneity, minimizing the amount of cached data transferred and thus reducing total time. It outperforms *D-SiteGreedyCache* and *D-FragGreedyCache* in terms of total time by up to 58% and 42% respectively.

Both cache site selection methods *bCompute* and *bStorage* have their own advantages. *bCompute* outperforms *bStorage* in terms of data transfer time by 13% for the first user and up to 17% for the second user. However, it does not scale with the number of users, and the limited storage capacities of Site 2 and 3 lead to a bottleneck. On the other hand, *bStorage* balances the cached data among sites and prevents the bottleneck when accessing the cached data, thus reducing re-execution times In summary, *bCompute* is best suited for compute-intensive workflows that generate smaller intermediate datasets while *bStorage* is best suited for data-intensive workflows where executions can be performed at the site where the data is stored.

6. Related Work

Several workflow systems support caching and reuse of intermediate data during workflow execution [13, 8, 24]. However, no solution we are aware of takes into account the geo-distributed aspect of workflow execution when using cached data. There is no definitive solution for two important problems: 1) how to determine what intermediate data should be cached, taking into account data transfers; 2) how the workflow should be scheduled when the intermediate data and the cached data processed are distributed over a multisite cloud. The related work provides two kinds of methods, either to reuse intermediate data in a monosite cloud or to optimize workflow execution in a multisite cloud, but without any reuse of intermediate data.

Several workflow systems, such as Kepler, VisTrails, OpenAlea, exploit intermediate data for more efficient workflow execution. Each system has its unique way of addressing data reuse. VisTrails provides visual analysis of workflow results and provenance, *i.e.*, captures the graph of execution and the intermediate data generated [13]. The intermediate data stored is reused when tasks are re-executed on a local computer. The user can then change some activities and parameters in the workflow and efficiently re-execute each workflow activity to analyze the different results. The intermediate data cache is used to enhance reproducibitily when associated with provenance metadata [25]. Caching and reuse of intermediate data is done whenever possible, but does not scale up as the data size increases, *i.e.*, the data cannot be stored locally. Finally, VisTrails does not take distribution into account when storing and using the cache. Our approach is different as it works in a distributed environment where data transfer costs may be significant.

When storing intermediate data in the cloud, the trade-off between the cost of re-executing tasks and the costs of storing intermediate data is not easy to estimate [26]. Yuan et al. [16] propose an algorithm based on the ratio between re-computation cost and storage cost at the fragment level. The algorithm uses the provenance data to generate a graph of the intermediate datasets dependencies. Then, the cost of storing each intermediate data set is weighted by the number of dependencies in the graph. The algorithm determines the optimized set of intermediate datasets to store. Cases et al. [27] propose a scheduling algorithm based on the trade-off between the cost of re-executing tasks and the costs of storing intermediate data in a cloud. The algorithm splits scientific workflows into multiple sub-workflows to balance system utilization via parallelization. It also exploits data reuse and replication techniques to optimize the amount of data that needs to be transferred among tasks at run-time. However, both of these approaches require global knowledge of executions, such as the execution time of each task, the size of each dataset and the number of incoming re-executions, which is hard to monitor in practice. Furthermore, it does not take data transfer into account.

Kepler [28] provides intermediate data caching that can be used by workflows executed on a monosite cloud. The cache data is stored on a remote server. When a workflow is re-executed, the workflow is modified to access the cached data. Then, all the cached data that will be reused by the workflow is sent to the cloud where the workflow will be executed. This solution is improved in [24] to store the cache data on the same site than the execution. This method selects which intermediate data will be cached using an algorithm based on Ant Colony System optimization to find a near optimum data caching policy. Owsiak et al. [12] propose a Kepler architecture to enable multiple users to execute workflows and reuse intermediate data through a cache system. The approach encapsulates all Kepler instances into Docker containers, which can easily be deployed in the shared environment. During workflow execution, each user can generate cached data. However, the cache is not shared between the users. Moreover, these solutions do not take data transfer into account and only work on monosite cloud. Our approach is different as it manages cache and workflow execution in a multisite cloud.

OpenAlea [8] uses caching both in memory and on disk. In memory caching is used on a local computer for smaller workflow execution. Disk caching is based on an adaptive cache decision that automatically determines which intermediate data is to be stored [15]. This solution only works on monosite cloud and the cached data is always reused.

Multisite cloud scheduling algorithms have been proposed to allow distributed workflow execution on multiple sites. Liu et al. [29] propose a scheduling algorithm based on data location, that minimizes data transfer during workflow executions. The algorithm is further improved in [10] with a multi-objective cost function, which includes the time and monetary cost of workflow execution in a dynamic environment. Zhang et al. [30] propose another interesting approach based on a specialized hybrid genetic algorithm, that optimizes the transfer data between the sites. These distributed scheduling approaches focus on optimizing workflow execution, but do not consider caching and reusing intermediate data.

7. Conclusion

In this paper, we considered the efficient execution of data-intensive scientific workflows in a multisite cloud, using caching of intermediate data produced by previous workflows. However, caching intermediate data and scheduling workflows to exploit such caching is complex, because of the heterogeneity of cloud data centers. In particular, workflow scheduling must be cache-aware, in order to decide whether reusing cached data or re-executing workflows.

We proposed a solution for cache-aware scheduling of scientific workflows in a multisite cloud. Our solution is based on a distributed and parallel architecture and includes new algorithms for adaptive caching, cache site selection and dynamic workflow scheduling. Our solution has been implemented in the OpenAlea workflow system. An extensive experimental evaluation is performed in a three-site cloud with a real application in plant phenotyping. We compared our solution with two baselines: 1) a multisite workflow scheduling algorithm that does not consider intermediate data cache, 2) and a centralized cache architecture for workflow execution.

For further comparisons, we extended two multisite scheduling algorithms to exploit our caching architecture. First, we showed that our solution for caching and reusing intermediate data can reduce the total workflow execution up to 42% with 60% of same input data for each new execution. Second, we showed that our solution efficiently distributes the fragments to the sites, which reduces the data transfer times, and thus the total time up by to 61%

compared with a single remote site. Third, we showed that our distributed cache architecture enables reducing the total time by 22% compared with our algorithm *GlobalGreedyCache* with a centralized cache architecture. We showed that the performance gain gets higher with heterogeneous sites. Fourth, we showed that the two methods *bStorage* and *bCompute* efficiently distribute the cache data to reduce the total time in the case of multiple users executing the workflow. Each method provides different benefits. *bStorage* distributes the cache data so that each site still has available storage for future intermediate data caching. *bCompute* enables faster workflow re-executions but the cache of the most powerful sites is rapidly full, which reduces the gain for future workflow executions. Fifth, we showed that our solution provides similar results as the adapted baseline algorithm in the case of homogeneous sites. And as site heterogeneity increases, it reduces total time up to 42%. Finally, we showed that our solution reduces the total time of several data-intensive subworkflows from Phenomenal. For compute-intensive subworkflows only, our solution has a small overhead of up to 4% compared with the adapted baseline algorithm.

The cost model of our solution focuses on minimizing the makespan. Some other objectives, such as minimizing financial costs, meeting deadline constraints, or following security constraints would change the decisions on scheduling and data caching. Furthermore, the objective of minimizing environmental cost becomes essential and could be integrated with the cache decision. The work proposed in this paper presents a solution for workflow caching in the environment of a multisite cloud with multiple users. A possible improvement would be to consider other objectives in the cost model.

Acknowledgement

This work was supported by the #DigitAg French initiative, the SciDISC and HPDaSc Inria associated teams with Brazil, the Phenome-Emphasis project (ANR-11-INBS-0012) and IFB (ANR-11-INBS-0013) from the Agence Nationale de la Recherche and the France Grille Scientific Interest Group.

References

[1] S. Kelling, W. M. Hochachka, D. Fink, M. Riedewald, R. Caruana, G. Ballard, G. Hooker, Data-intensive science: a new paradigm for biodiversity studies, BioScience 59 (7) (2009) 613–620.

[2] S. Crago, K. Dunn, P. Eads, L. Hochstein, D.-I. Kang, M. Kang, D. Modium, K. Singh, J. Suh, J. P. Walters, Heterogeneous cloud computing, in: 2011 IEEE International Conference on Cluster Computing, IEEE, 2011, pp. 378–385.

[3] D. de Oliveira, F. A. Baião, M. Mattoso, Towards a taxonomy for cloud computing from an e-science perspective, in: Cloud Computing. Computer Communications and Networks., Springer, 2010, pp. 47–62.

[4] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, I. T. Foster, Swift/t: Large-scale application composition via distributed-memory dataflow processing, in: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, IEEE, 2013, pp. 95–102.

[5] E. Deelman, K. Vahi, M. Rynge, G. Juve, R. Mayani, R. F. Da Silva, Pegasus in the cloud: Science automation through workflow technologies, IEEE Internet Computing 20 (1) (2016) 70–76.

[6] D. de Oliveira, E. Ogasawara, F. Baião, M. Mattoso, Scicumulus: A lightweight cloud middleware to explore many task computing paradigm in scientific workflows, in: 2010 IEEE 3rd International Conference on Cloud Computing, IEEE, 2010, pp. 378–385.

[7] P. Korambath, J. Wang, A. Kumar, L. Hochstein, B. Schott, R. Graybill, M. Baldea, J. Davis, Deploying kepler workflows as services on a cloud infrastructure for smart manufacturing, Procedia Computer Science 29 (2014) 2254–2259.

[8] C. Pradal, C. Fournier, P. Valduriez, S. Cohen-Boulakia, Openalea: scientific workflows combining data analysis and simulation, in: Int. Conf. on Scientific and Statistical Database Management (SSDBM), 2015, pp. 11:1–11:6.

[9] K. Maheshwari, E. Jung, J. Meng, V. Vishwanath, R. Kettimuthu, Improving multisite workflow performance using model-based scheduling, in: IEEE nt. Conf. on Parallel Processing (ICPP), 2014, pp. 131–140.

[10] J. Liu, E. Pacitti, P. Valduriez, D. de Oliveira, M. Mattoso, Multi-objective scheduling of scientific workflows in multisite clouds, Future Generation Computer Systems(FGCS) 63 (2016) 76–95.

[11] D. Garijo, P. Alper, K. Belhajjame, O. Corcho, Y. Gil, C. Goble, Common motifs in scientific workflows: An empirical analysis, Future Generation Computer Systems (FGCS) 36 (2014) 338–351.

[12] M. Owsiak, M. Plociennik, B. Palak, T. Zok, C. Reux, L. Di Gallo, D. Kalupin, T. Johnson, M. Schneider, Running simultaneous kepler sessions for the parallelization of parametric scans and optimization studies applied to complex workflows, Journal of Computational Science 20 (2017) 103–111.

[13] J. Freire, D. Koop, F. S. Chirigati, C. T. Silva, Reproducibility using vistrails, Implementing Reproducible Research 33.

[14] C. Pradal, S. Artzet, J. Chopard, D. Dupuis, C. Fournier, M. Mielewczik, V. Negre, P. Neveu, D. Parigot, P. Valduriez, et al., Infraphenogrid: a scientific workflow infrastructure for plant phenomics on the grid, Future Generation Computer Systems (FGCS) 67 (2017) 341–353.

[15] G. Heidsieck, D. de Oliveira, E. Pacitti, C. Pradal, F. Tardieu, P. Valduriez, Adaptive caching for data-intensive scientific workflows in the cloud, in: Int. Conf. on Database and Expert Systems Applications (DEXA), 2019, pp. 452–466.

[16] D. Yuan, Y. Yang, X. Liu, W. Li, L. Cui, M. Xu, J. Chen, A highly practical approach toward achieving minimum data sets storage cost in the cloud, IEEE Trans. on Parallel and Distributed Systems 24 (6) (2013) 1234–1244.

[17] J. Liu, L. P. Morales, E. Pacitti, A. Costan, P. Valduriez, G. Antoniu, M. Mattoso, Efficient scheduling of scientific workflows using hot metadata in a multisite cloud, IEEE Trans. on Knowledge and Data Engineering (2018) 1–20.

[18] M. T. Özsu, P. Valduriez, Principles of Distributed Database Systems, Fourth Edition, Springer, 2020.

[19] F. Tardieu, L. Cabrera-Bosquet, T. Pridmore, M. Bennett, Plant phenomics, from sensors to knowledge, Current Biology 27 (15) (2017) R770–R783.

[20] S. Artzet, N. Brichet, J. Chopard, M. Mielewczik, C. Fournier, C. Pradal, Openalea.phenomenal: A workflow for plant phenotyping (Sep. 2018). doi:10.5281/zenodo.1436634.

[21] J. Zhang, J. Luo, F. Dong, Scheduling of scientific workflow in non-dedicated heterogeneous multicluster platform, Journal of Systems and Software 86 (7) (2013) 1806–1818.

[22] J. Liu, E. Pacitti, P. Valduriez, M. Mattoso, A survey of data-intensive scientific workflow management, Journal of Grid Computing 13 (4) (2015) 457–493.

[23] G. Heidsieck, D. de Oliveira, E. Pacitti, C. Pradal, F. Tardieu, P. Valduriez, Efficient execution of scientific workflows in the cloud through adaptive caching, in: Transactions on Large-Scale

Data-and Knowledge-Centered Systems XLIV, Springer, 2020, pp. 41–66.
[24] W. Chen, I. Altintas, J. Wang, J. Li, Enhancing smart re-run of kepler scientific workflows based on near optimum provenance caching in cloud, in: IEEE World Congress on Services (SERVICES), 2014, pp. 378–384.
[25] S. C. Dey, K. Belhajjame, D. Koop, T. Song, P. Missier, B. Ludäscher, Up & down: Improving provenance precision by combining workflow-and trace-level information, in: USENIX Workshop on the Theory and Practice of Provenance (TAPP), 2014.
[26] I. F. Adams, D. D. Long, E. L. Miller, S. Pasupathy, M. W. Storer, Maximizing efficiency by trading storage for computation., in: HotCloud, 2009.
[27] I. Casas, J. Taheri, R. Ranjan, L. Wang, A. Y. Zomaya, A balanced scheduler with data reuse and replication for scientific workflows in cloud computing systems, Future Generation Computer Systems 74 (2017) 168–178.
[28] I. Altintas, O. Barney, E. Jaeger-Frank, Provenance collection support in the kepler scientific workflow system, in: International Provenance and Annotation Workshop, 2006, pp. 118–132.
[29] J. Liu, V. Silva, E. Pacitti, P. Valduriez, M. Mattoso, Scientific workflow partitioning in multisite cloud, in: European Conf. on Parallel Processing (Euro-Par), 2014, pp. 105–116.
[30] J. Zhang, J. Chen, J. Zhan, J. Jin, A. Song, Graph partition–based data and task co-scheduling of scientific workflow in geo-distributed datacenters, Concurrency and Computation: Practice and Experience 31 (24) (2019) e5245.