A Data-aware Scheduling Strategy for Workflow Execution in Clouds

Fabrizio Marozzo[§,*], Francisco Rodrigo Duro[¶], Javier Garcia Blas[¶], Jesus Carretero[¶], Domenico Talia[§], Paolo Trunfio[§]

[§]*DIMES, University of Calabria, Rende, Italy*
[¶]*ARCOS, University Carlos III, Madrid, Spain*

SUMMARY

As data intensive scientific computing systems become more widespread, there is a necessity of simplifying the development, deployment, and execution of complex data analysis applications for scientific discovery. The scientific workflow model is the leading approach for designing and executing data-intensive applications in high-performance computing infrastructures. Commonly, scientific workflows are built by a set of connected tasks generally arranged in a directed acyclic graph style, which communicate through storage abstractions. The Data Mining Cloud Framework (DMCF) is a system allowing users to design and execute data analysis workflows on cloud platforms, relying on cloud storage services for every I/O operation. Hercules is an in-memory I/O solution that can be used in DMCF as an alternative to cloud storage services, providing additional performance and flexibility features. This work improves the integration between DMCF and Hercules by using a data-aware scheduling strategy for exploiting data locality in data-intensive workflows. This paper presents experimental results demonstrating the performance improvements achieved using the proposed data-aware scheduling strategy in the Microsoft Azure cloud platform. In particular, with our scheduling strategy, the I/O overhead has been reduced by 55% with respect to the Azure storage, leading to a 20% reduction of the total execution time.

Received ...

KEY WORDS: DMCF, Hercules, workflows, in-memory storage, data-aware scheduling, Microsoft Azure

1. INTRODUCTION

Scientific computing applications and platforms are evolving from CPU-intensive tasks executed over strongly coupled infrastructures, i.e. complex simulations running on supercomputers, to data-intensive systems requiring flexible computing resources depending on the requirements and budget of users. This evolution paves the future of Ultrascale systems, which will blur the differences of the different existing scientific computing infrastructures, such as HPC systems and cloud computing platforms. In current approaches, the interfaces and management solutions of the different infrastructures present notable differences, requiring different programming models, even for the same application. On the other side, the future Ultrascale systems should take advantage of every possible resource available in a transparent way for the user [1].

Workflow engines are the leading approach for executing data-intensive applications in multiple computing infrastructures. Scientific workflows consist of interdependent tasks, connected in a DAG style, which communicate through intermediate storage abstractions, typically files. There is a main

[§]E-mail: {fmarozzo,talia,trunfio}@dimes.unical.it
[¶]E-mail: {frodrigo,fjblas,jcarrete}@inf.uc3m.es
*Correspondence to: Fabrizio Marozzo, DIMES, University of Calabria, Via P.Bucci 41C, 87036 Rende, Italy

tradeoff that should be taken into account when users rely on workflow engines for data-intensive applications. While portability and flexibility offers a broader support of the existing computing resources, the achieved performance is usually limited in contrast with native applications (classical HPC applications running on HPC clusters or supercomputers) [2].

The increasing availability of data generated by high-fidelity simulations and high-resolution scientific instruments in domains as diverse as astronomy [3], experimental physics [4], and bioinformatics [5], has shown the underlying I/O subsystem to be a substantial performance bottleneck. While typical high-performance computing (HPC) systems rely on monolithic parallel file systems, data-intensive workflow implementations must borrow techniques from the Big Data computing (BDC) space, such as exposing data storage locations and scheduling work to reduce data movement. This lack of performance is the result of a sub-optimal exploitation of the available resources, based on two main reasons: task schedulers unable to select the best nodes depending on the characteristics of the task and under-performing I/O solutions.

Our previous work [6] targeted these disadvantages in a real-world scenario by combining two existing solutions: the Data Mining Cloud Framework (DMCF) [7] and an in-memory I/O accelerator, namely Hercules [8]. This work improves the previous one by proposing a data-aware scheduling strategy for exploiting data locality in data-intensive workflows executed over the DMCF and Hercules platform. An experimental evaluation has been carried out to evaluate the benefits of the proposed data-aware scheduling strategy executing data analysis workflows and to demonstrate the effectiveness of the solution. Using the proposed data-aware strategy and Hercules as temporary storage service, the I/O overhead of workflow execution has been reduced by 55% with respect to the standard execution based on the Azure storage, leading to a 20% reduction of the total execution time. This evaluation confirms that our data-aware scheduling approach is effective in improving the performance of data-intensive workflow execution in cloud platforms.

The remainder of the paper is structured as follows. Section 2 discusses other research work in the same field. Section 3 describes the main features of DMCF. Section 4 introduces Hercules architecture and capabilities. Section 5 emphasizes the advantages of integrating DMCF and Hercules and outlines how this integration works. Section 6 details the data-aware scheduling technique proposed in this work. Section 7 presents the experimental results. Finally, Section 8 concludes the paper.

2. RELATED WORK

The popularity of data-intensive techniques applied to scientific computing is rapidly growing in the last few years. The exploitation of these techniques, in combination with the use of cloud computing resources to adapt the size of the resources to the budget of the research institutions, is exposing the bottlenecks of the underlying classical high-performance I/O subsystems and cloud storage services. Data intensive applications, especially where high concurrency in I/O operations appears (many-tasks, workflows, etc.), require novel approaches to address these new challenges.

Most of the problems that have to be addressed in this new data intensive tendency are not completely new, and existing solutions should be explored before addressing the new challenges. That is the case of solutions that rely on in-memory storage as a different approach for solving the bottlenecks in highly concurrent I/O operations, such as Parrot [9], Chirp [10], AHPIOS [11], and MosaStore [12, 13]. Our in-memory solution takes hints from all these solutions, by i) offering popular data access APIs (POSIX, put/get, MPI-IO), ii) focusing on flexible scalability through distributed approaches for both data and metadata, iii) the use of compute nodes (or worker VMs) to enhance the I/O infrastructure, and iv) facilitating the deployment of user-level components. The in-memory approach exploited in our work is supported by the popularity of other solutions, such as Apache Spark [14] and Tachyon [15]. Both solutions are based on two fundamental premises shared with Hercules. First, in-memory storage and data locality are key aspects for improving performance in data intensive applications. Second, I/O operations can be accelerated through the use of high-performance networks when available.

The AMFS framework [16] is a similar approach to our solution. The main goal of AMFS is to provide a simple way for designing and implementing workflows through the use of a simple scripting language. Additionally, the end-user can select in the code which operations should be executed in-memory. AMFS shares similar characteristics with our solution. First, data and metadata accesses are optimized for single-write, multi-read patterns. Second, I/O nodes double as metadata management servers, fully distributing metadata among them. Third, tasks can be executed co-located with the data, exploiting data locality for increased performance. However, DMCF offers simpler ways for designing and implementing workflows, supporting both graphical and scripting definitions. Additionally, we fully support cloud platforms, allowing an easier execution of workflows in cloud-based environments.

Confuga [17] is an active storage cluster file system. Confuga shares its main objective with our solution: the execution of data-intensive DAG-structured workflows, especially focusing on scalability of POSIX applications. As Hercules, it is easy to deploy (based on user-level components) and targets co-location of execution and data in distributed environments. However, there are three main differences with Hercules. First, Confuga targets clusters while the combination of DMCF and Hercules is focused on cloud environments. Second, there is a tight coupling between the storage management and the Confuga scheduler. Third, Confuga relies on a centralized "head node" in contrast with the distributed approach of the Hercules architecture.

SuperGlue [18] is focused on simplifying the design and implementation of workflows by proposing reusable pieces of code in contrast with the more explored scripting approach, which requires coding a new script for each workflow application. DMFC explores this path in a deeper way, bringing the possibility to the user of designing the workflow through a graphical interface.

Juve et al. [19] studied different approaches for data sharing in scientific workflows deployed over public cloud infrastructures. Their work is relevant for comparing different approaches (NFS, GlusterFS, PVFS2 and S3). However it is focused on applying HPC-and grid-oriented solutions in the cloud. These solutions are difficult to configure and offer sub-optimal performance for cloud environments, in contrast with our cloud-centric approach.

Bryk et al. [20] cover the design of storage-aware and locality-aware algorithms for the execution of workflows in cloud infrastructures. They evaluate the quality of such algorithms using a model for minimizing the execution time and execution cost (deadline and budget limits). Our work shares their objective of improving performance through data-aware scheduling algorithms, but we focus our work on real executions with our DMCF and Hercules combination instead of relying on a simulation model for generic workflows and I/O solutions.

In our previous work [6], the performance of DMCF applications has been improved by using Hercules system as an ad-hoc storage system for temporary data. Experiments conducted on a classification workflow, showed that using Hercules as temporary storage in place of the standard Azure storage, there is 36% reduction of I/O overhead that leads to a 13% reduction of the total execution time. In this work, we show how to further improve the integration between DMCF and Hercules by using a data-aware scheduling strategy for exploiting data locality in data-intensive workflows. The experimental evaluation presented in this work show that the data-aware scheduler is able to further reduce the I/O overhead by 55% with respect to Azure storage, thus leading to a 20% reduction of the total execution time.

3. DATA MINING CLOUD FRAMEWORK OVERVIEW

The Data Mining Cloud Framework (DMCF) [21] is a software system designed for executing data analysis workflows on clouds. A Web-based user interface allows users to compose their applications and to submit them for execution to the cloud platform, following a Software-as-a-Service (SaaS) approach.

The architecture of DMCF includes different components that can be grouped into storage and compute modules (see Figure 1). The storage module is divided into three components: i) A *Data Folder* that contains data sources and the results of knowledge discovery processes, and a *Tool Folder* that contains libraries and executable files for data selection, pre-processing, transformation,

data mining, and evaluation of results; ii) *Data Table*, *Tool Table* and *Task Table* contain metadata information associated with data, tools, and tasks; iii) The *Task Queue* contains the tasks that are ready for execution.

The compute components are: i) a pool of *Virtual Compute Servers* (or *Workers*), which are in charge of executing the data analysis tasks; ii) a pool of *Virtual Web Servers* that host the Web-based user interface.

The DMCF architecture has been designed to be implemented on top of different cloud systems. The implementation used in this work is based on Microsoft Azure[†]. However other approaches like Amazon can be considered as well.

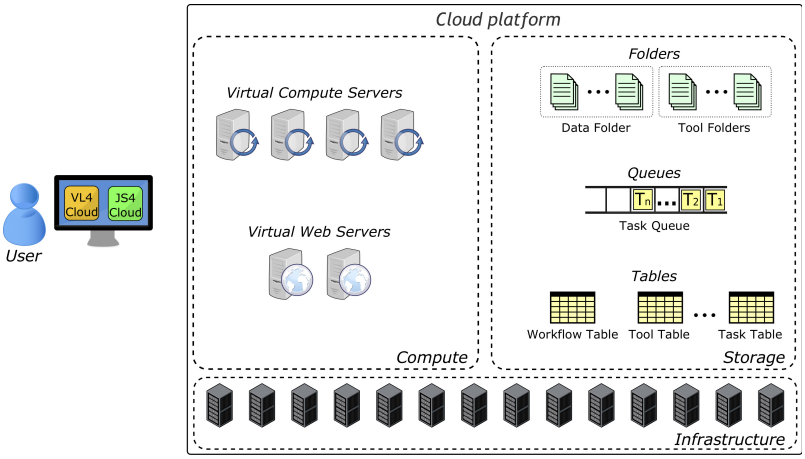


Figure 1. Architecture of Data Mining Cloud Framework.

DMCF allows to program data analysis workflows using *VL4Cloud* and *JS4Cloud*. First, *VL4Cloud* (Visual Language for Cloud) is a visual programming language that lets users develop applications by programming the workflow components graphically [22]. Second, *JS4Cloud* (JavaScript for Cloud) is a scripting language for programming data analysis workflows based on JavaScript [21]. Both languages use two key programming abstractions:

- *Data* elements, denoting input files or storage elements (e.g., a dataset to be analyzed) or output files or stored elements (e.g., a data mining model).
- *Tool* elements, denoting algorithms, software tools or complex applications performing any kind of operation that can be applied to a data element (data mining, filtering, partitioning, etc.).

Another common element is the *Task* concept, which represents the minimum unit of parallelism in our model. A task is a *Tool* invoked in the workflow, which is intended to run in parallel with other tasks on a set of cloud resources. According to this approach, VL4Cloud and JS4Cloud implement a data-driven task parallelism. This means that, as soon as a task does not depend on any other task in the same workflow, the run-time asynchronously spawns it to the first available virtual machine (VM). A task T_j does not depend on a task T_i belonging to the same workflow (with i $\neq$ j), if T_j during its execution does not read any data element created by T_i.

4. HERCULES OVERVIEW

Hercules [8] is a distributed in-memory storage system based on well-known key/value servers. In this work, we have used Memcached [23], as back-end but any other key/value distributed hash table can

[†]http://azure.microsoft.com

be used (i.e. Redis). The approach of Hercules as a distributed in-memory storage is perceived by the user as a single virtual device. Applications deployed over the cloud infrastructure can alternatively use both the default cloud storage service or Hercules as an in-memory storage for data accesses.

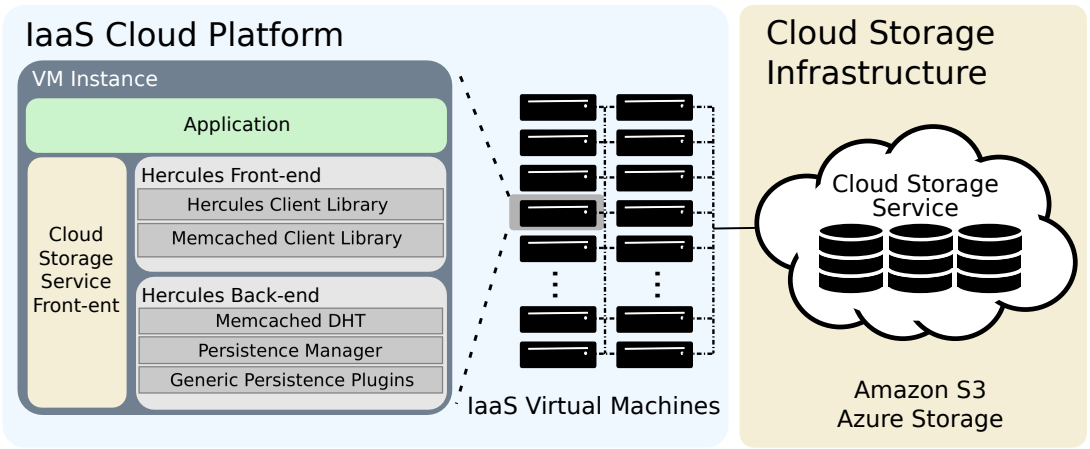


Figure 2. Architecture of Hercules.

In Figure 2, we describe the architecture of Hercules. Our solution is divided into two main components: front-end (client component) and back-end (server component). The front-end is a user-level library used by the applications for any I/O access. The back-end is based on Memcached servers with improved functionality (persistence) and performance tweaks. Hercules is designed for targeting four main objectives: scalability, easy deployment, flexibility, and performance.

- *Scalability.* Hercules distributes data and metadata items among every available node in order to avoid centralized components that tend to expose bottlenecks and result in single points of failure. In addition, the network communications are minimized by calculating data and metadata placement at client side through algorithmic hashing functions.
- *Easy deployment.* Every component of the solution is deployed at user lever, avoiding the necessity of any special privileges for deploying the solution. Servers are completely stateless and unaware of the rest of the infrastructure, enabling the possibility of deploying as many as necessary without complex configurations. This approach also benefits the scalability of the solution.
- *Flexibility.* In order to support legacy applications with minimum changes, our solution offers a POSIX-like interface in addition to the default put/get approach, inherited from Memcached and especially useful for NoSQL-oriented applications. However, thanks to its layered design, it is possible to easily add new services on top of these APIs or rely on other the underlying back-end (i.e., Redis).
- *Performance.* Based on the Hercules scalability capabilities, applications can exploit I/O parallelism by accessing in parallel to the available Hercules I/O nodes. Each node can be accessed independently, multiplying the total throughput peak performance.

5. INTEGRATION BETWEEN DMCF AND HERCULES

DMCF and Hercules can be configured to achieve different levels of integration, as shown in Figure 3.

Figure 3(a) shows the original approach of DMCF, where every I/O operation is carried out against the cloud storage service offered by the cloud provider (Azure Storage). There are, at least, four disadvantages about this approach: usage of proprietary interfaces, I/O contention in the service, lack

of configuration options, and persistence-related costs unnecessary for temporary data. Figure 3(b) presents a second scenario based on the use of Hercules as the default storage for temporary generated data [24]. Hercules I/O nodes can be deployed on as many VM instances as needed by the user depending on the required performance and the characteristics of data. Figure 3(c) shows a third scenario with a tighter integration of DMCF and Hercules infrastructures. In this scenario, initial input and final output are stored on persistent Azure storage, while intermediate data are stored on Hercules in-memory nodes. Hercules I/O nodes share virtual instances with the DMCF workers.

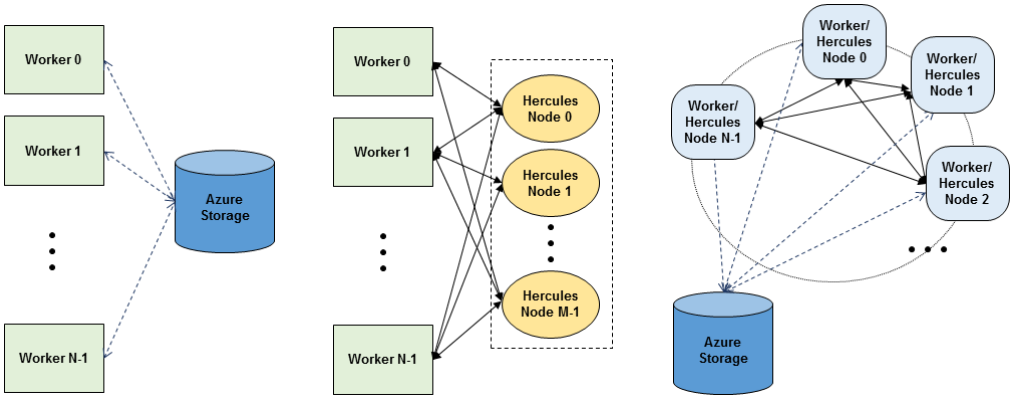
Current trends in data-intensive applications are extensively focusing on the optimization of I/O operations by exploiting the performance offered by in-memory computation, as shown by the popularity of Apache Spark Resilient Distributed Datasets (RDD) [14]. The goal of this work is strengthening the integration between DMCF and Hercules by leveraging the co-location of compute workers and I/O nodes for exposing and exploiting data locality.

A previous work [6] explored the third scenario outlined above. However, in order to simplify the implementation of the solution, some workarounds were used: each time that one worker needed to access data (read/write operations over a file), it copied the whole file from Hercules servers to the worker local storage. This approach may greatly penalize the potential performance gain in I/O operations for two main reasons:

- *Data placement strategy*. The original Hercules data placement policy distributes every partition of a specific file among all the available servers. This strategy has two main benefits: avoids hot spots and improves parallel accesses. In an improved DMCF-Hercules integration, whole files can be stored on the same Hercules server.
- *Data locality agnosticism*. Data-locality will not be fully exploited until the DMCF scheduler is tweaked for running tasks on the node that contains the necessary data and/or the data is placed where the computation will be realized.

Figure 4 describes an improvement to the third scenario of integration between DMCF and Hercules, which is exploited as the base for the proposed data-aware scheduling strategy. Four main components are present: DMCF worker daemon, Hercules daemon, Hercules client library, and Azure client library. The DMCF worker daemons are in charge of executing the workflow tasks; Hercules daemons act as I/O nodes (storing data in-memory and managing data accesses); the Hercules client library is intended to be used by the applications to access to the data stored in Hercules (query Hercules daemons); the Azure client library is used to read/write data from/to the Azure storage.

In this model, we use a RAM disk as generic storage buffer for I/O operations performed by workflow tasks. The objective of this approach is to enable the support of DMCF to any existing tool,



(a) Scenario 1: Azure storage. (b) Scenario 2: Hercules. (c) Scenario 3: Azure storage and Hercules.

Figure 3. Integration scenarios between DMCF and Hercules.

allowing even binaries independently of the language used for their implementation, while offering in-memory performance for local accesses.

The logic used for managing this RAM disk buffer is based on the full information about the workflow possessed by the DMCF workers. When every dependency of an specific task is fulfilled (every input file is ready to be accessed) the DMCF worker brings the necessary data to the node from the storage (Azure Storage in the first scenario or Hercules in the second scenario).

Based on this approach, every existing tool is capable of accessing transparently data, without the need of modifying the code. After the termination of the task, every data written in the RAM disk is transferred to Hercules/Azure by the DMCF worker. Every data transfer to/from Hercules is performed by the DMCF worker through the Hercules client library.

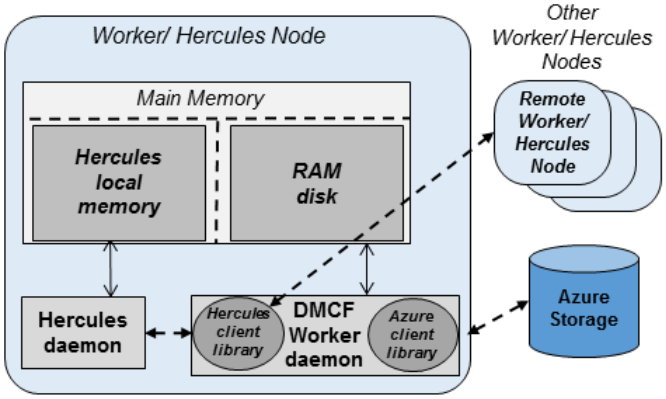


Figure 4. DMCF and Hercules daemons.

6. DATA-AWARE SCHEDULING STRATEGY

This section presents the data-aware scheduling strategy that combines DMCF load-balancing capabilities and Hercules data and metadata distribution functionalities for implementing various locality-aware and load-balancing policies.

Before going into details of the new purposed scheduling mechanism, we recall the high-level steps that are performed for designing and executing a knowledge discovery application in DMCF [7]:

1. The user accesses the website and designs the workflow through a Web-based interface.
2. After workflow submission, the system creates a set of tasks and inserts them into the Task Queue.
3. Each idle worker picks a task from the Task Queue, and concurrently executes it.
4. Each worker gets the input dataset from its location. To this end, a file transfer is performed from the Data Folder where the dataset is located, to the local storage of the worker.
5. After task completion, each worker puts the result on the Data Folder.
6. The Website notifies the user whenever each task has completed, and allows her/him to access the results.

Algorithm 1 describes the new data-aware scheduler employed by each worker. Compared to the original scheduler described in [7], this novel scheduling algorithm relies on a local list within each worker, called *locallyActivatedTasks*. This list contains all the tasks whose status was changed by this worker from 'new' to '$ready$' at the end of the previous iteration. The worker cyclically checks whether there are tasks ready to be executed in *locallyActivatedTasks* or in the Task Queue (lines

```
1  Procedure main()
2      while true do
3          if locallyActivatedTasks.isNotEmpty() or TaskQueue.isNotEmpty() then
4              task ← selectTask(locallyActivatedTasks, TaskQueue);
5              TaskTable.update(task, 'running');
6              foreach input in task.inputList do
7                  transfer(input, DataFolder, localDataFolder);
8              transfer(task.tool.executable, localToolFolder);
9              foreach library in task.tool.libraryList do
10                 transfer(library, ToolFolder, localToolFolder);
11             taskStatus ← execute(task, localDataFolder, localToolFolder);
12             if taskStatus = 'done' then
13                 foreach output in task.outputList do
14                     transfer(output, localDataFolder, DataFolder);
15                 TaskTable.update(task, 'done');
16                 foreach wfTask in TaskTable.getTasks(task.workflowId) do
17                     if wfTask.dependencyList.contains(task) then
18                         wfTask.dependencyList.remove(task);
19                         if wfTask.dependencyList.isEmpty() then
20                             TaskTable.update(wfTask, 'ready');
21                             locallyActivatedTasks.addTask(wfTask);
22             else //Manage the task's failure;
23         else //Wait for new tasks in TaskQueue;
24 SubProcedure selectTask(locallyActivatedTasks, TaskQueue)
25     bestTask ← Hercules.selectByLocality(locallyActivatedTasks, TaskQueue);
26     if bestTask in TaskQueue then
27         TaskQueue.removeTask(bestTask);
28     foreach task in locallyActivatedTasks do
29         if task != bestTask then
30             TaskQueue.addTask(task);
31     empty(locallyActivatedTasks);
32     clean(localDataFolder);
33     clean(localToolFolder);
34     return bestTask;
```

Algorithm 1. Worker operations.

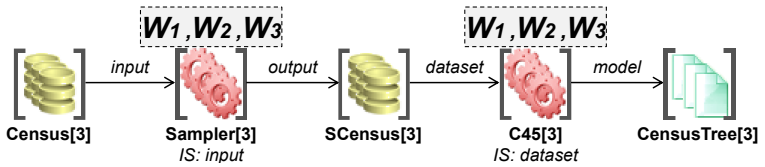
2-3). If so, a task is selected from one of the two sets (line 4) using the *selectTask* subprocedure (lines 24-34), and its status is changed to '$running$' (line 5). Then, the transfer of all the needed input resources (files, executables, and libraries) is carried out from their original location to two local folders, referred to as *localDataFolder* and *localToolFolder* (lines 6-10). At line 11, the worker locally executes the *task* and waits for its completion. If the *task* is '$done$' (line 12), if necessary the output results are copied to a remote data folder (lines 13-14), and the *task* status is changed to '$done$' also in the Task Table (line 15). Then, for each *wfTask* that belongs to the same workflow of *task* (line 16), if *wfTask* has a dependency with *task* (line 17), that dependency is deleted (line 18). If *wfTask* remains without dependencies (line 19), it becomes '$ready$' and is added to the Task Queue (lines 20-21). If the *task* fails (line 22), all the tasks that directly or indirectly depend on it are marked as '$failed$'.

The *selectTask* subprocedure works as follows. First, it invokes the *selectByLocality* function provided by Hercules, which selects the best task that this worker can manage from *locallyActivatedTasks* and the Task Queue (line 25). To take advantage of data locality, the best

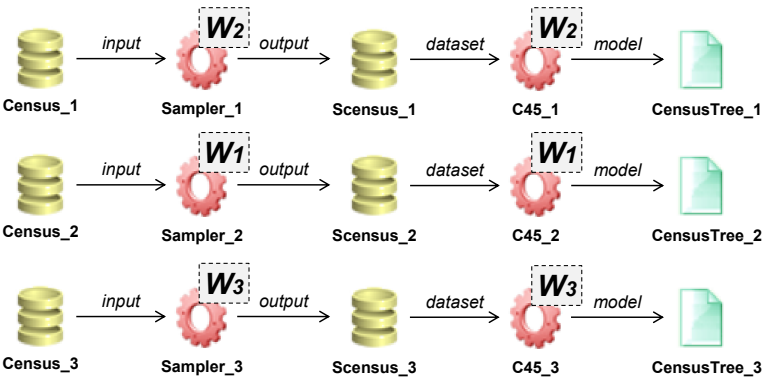
task selected by this function is the one having the highest number of input data that are available on the local storage of the worker, based on the information available to Hercules. This differs from the original data-locality agnostic scheduling policy adopted in DMCF, in which each worker picks and executes the task from the queue following a FIFO policy. If such best task was chosen from the Task Queue, then that task is removed from the Task Queue (line 26-27). All the tasks in *locallyActivatedTasks* that are different from the best task are added to the Task Queue, thus allowing the other workers to execute them (line 28-30). Finally, *locallyActivatedTasks* is emptied, and localDataFolder and localToolFolder are cleaned from the data/tools that are not used by bestTask (lines 31-33).

The improvements of this new scheduler compared to the original one can be described using the two examples reported in Figures 5 and 6.

Figure 5 shows an example of pipeline workflow, in which the output of a task is the input for the subsequent task. Figure 5(a) refers to the workflow executed without using the data-aware scheduler, while Figure 5(b) refers to the same workflow executed using the data-aware scheduler. The workflow processes three datasets ($Census[3]$) in parallel using three instances of $Sampler[3]$ to produce the same number of sampled datasets. Then, the sampled datasets are analyzed in parallel by three instances of C4.5 [25], to produce the same number of data mining models ($CensusTree[3]$). Suppose we have three workers ($W1$, $W2$ and $W3$); using the original FIFO scheduler, the three tasks generated by the execution of $Sampler[3]$ and the other three generated by $C45[3]$ are executed by the three workers in any order. Figure 5(b) presents a possible execution order of the pipeline using the proposed data-aware scheduler. If worker $W2$ executes the first instance of $Sampler$, the same worker will execute the first instance of $C4.5$, having already generated $Scensus_1$ as input for $C4.5$. Same considerations can be done for the other tasks of the same pipeline, executed by $W2$ and $W3$.



(a) Without data-aware scheduling (data and tool arrays used for compactness)

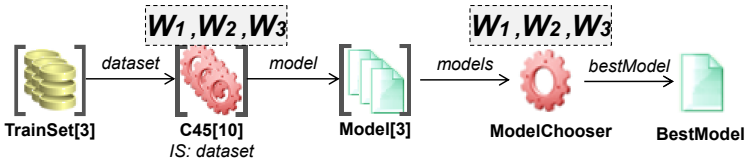


(b) With data-aware scheduling

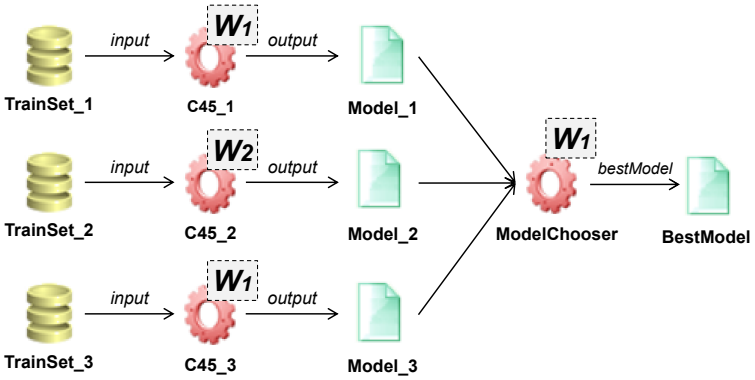
Figure 5. Possible execution orders of a pipeline workflow without and with the data-aware scheduler.

Figure 6 represents an example of data aggregation workflow in which a tool generates one output data from multiple input data. As for the first example, we report a possible execution order obtained without (Figure 6(a)) and with (Figure 6(b)) the use of the data-aware scheduler. The workflow processes three training datasets ($TrainSet[3]$) in parallel using three instances of $C4.5$, to produce the same number of models ($Model[3]$). Then, a $ModelChooser$ tool takes as input three data mining models and chooses the best one based on some evaluation criteria ($BestModel$). Figure 6(b) shows

a possible execution of this workflow using the data-aware scheduler. If worker $W1$ executes the first and the third instance of $C4.5$, the second worker $W2$ executes the second instance of $C4.5$, and the third worker is currently executing some other tasks, the $ModelChooser$ task will be executed by $W1$ since it is the available worker having the highest number of $ModelChooser$'s inputs ($Model_1$ and $Model_3$).



(a) Without data-aware scheduling (data and tool arrays used for compactness)



(b) With data-aware scheduling

Figure 6. Possible execution orders of a data aggregation workflow without and with the data-aware scheduler.

7. EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of the new data-aware scheduling strategy used to improve the integration between DMCF and Hercules. For this evaluation, we have emulated the execution of a data analysis workflow using three alternative scenarios:

- *Azure-only*: every I/O operation of the workflow is performed by DMCF using the Azure storage service.
- *Locality-agnostic*: a full integration between DMCF and Hercules is exploited, where each intermediate data is stored in Hercules, while initial input and final output are stored on Azure. DMCF workers and Hercules I/O nodes share resources (they are deployed in the same VM instance), however, every I/O operation is performed over remote Hercules I/O nodes through the network.
- *Data-aware*: based on the same deployment as in the previous case, this scenario is based on a full knowledge of the data location, and executes every task in the same node where the data are stored, leading to fully local accesses over temporary data. Based on this locality exploitation, most I/O operations are performed in-memory rather than through the network.

The evaluation is based on a data mining workflow that analyzes n partitions of the training set using k classification algorithms so as to generate kn classification models. The kn models generated are then evaluated against a test set by a model selector to identify the best model. Then, n predictors use the best model to produce in parallel n classified datasets. The k classification algorithms used in

Listing 1: Classification JS4Cloud workflow.

```
 1: var TrRef = Data.get("Train");
 2: var STrRef = Data.define("STrain");
 3: Shuffler({dataset:TrRef, sDataset:STrRef});
 4: var n = 20;
 5: var PRef = Data.define("TrainPart", n);
 6: Partitioner({dataset:STrRef, datasetPart:PRef});
 7: var MRef = Data.define("Model", [3,n]);
 8: for(var i = 0; i <n; i++){
 9:   C45({dataset:PRef[i], model:MRef[0][i]});
10:   SVM({dataset:PRef[i], model:MRef[1][i]});
11:   NaiveBayes({dataset:PRef[i], model:MRef[2][i]});
12: }
13: var TeRef = Data.get("Test");
14: var BMRef = Data.define("BestModel");
15: ModelSelector({dataset:TeRef, models:MRef, bestModel:BMRef});
16: var m = 80;
17: var DRef = Data.get("Unlab", m);
18: var FDRef = Data.define("FUnlab", m);
19: for(var i = 0; i <m; i++)
20:   Filter({dataset:DRef[i], fDataset:FDRef[i]});
21: var CRef = Data.define("ClassDataset", m);
22: for(var i = 0; i <m; i++)
23:   Predictor({dataset:FDRef[i], model:BMRef, classDataset:CRef[i]});
```

the workflow are C4.5 [25], Support Vector Machine (SVM) [26] and Naive Bayes [27], which are three of the main classification algorithms [28]. The training set, test set and unlabeled dataset that represent the input of the workflow, have been generated from the *KDD Cup 1999*'s dataset[‡], which contains a wide variety of simulated intrusion records in a military network environment.

Listing 1 shows the JS4Cloud source code of the workflow. At the beginning, we define the training set (*line 1*) and a variable that stores the shuffled training set (*line 2*). At *line 3*, the training set is processed by a shuffling tool. Once defined parameter $n = 20$ at *line 4*, the shuffled training set is partitioned into n parts using a partitioning tool (*line 6*). Then, each part of the shuffled training set is analyzed in parallel by $k = 3$ classification tools ($C4.5$, SVM, $NaiveBayes$). Since the number of tools is k and the number of parts is n, kn instances of classification tools run in parallel to produce kn classification models (*lines 8-12*). The kn classification models generated are then evaluated against a test set by a model selector to identify the best model (*line 15*). Then, $m = 80$ unlabeled datasets are specified as input (*line 15*). Each of the m input datasets is filtered in parallel by m filtering tools (*lines 19-20*). Finally, each of the m filtered datasets is classified by m predictors using the best model (*lines 22-23*). The workflow is composed of $3 + kn + 2m$ tasks. In the specific example, where $n = 20$, $k = 3$, $m = 80$, the number of generated tasks is equal to 223.

Figure 7 depicts the VL4Cloud version of the data mining workflow. The visual formalism clearly highlights the level of parallelism of the workflow, expressed by the number of parallel paths and the cardinality of tool array nodes.

Once the workflow is submitted to DMCF using either JS4Cloud or VL4Cloud, DMCF generates a JSON descriptor of the workflow, specifying which are the tasks to be executed and the dependency relationships among them. Thus, DMCF creates a set of tasks that will be executed by workers. In order to execute a given workflow task, we have provisioned as many D2 VM instances (CPU with 2 cores and 7GB of RAM) in the Azure infrastructure as needed, and configured them by launching both the Hercules daemon and the DMCF worker process on each VM. In order to better

[‡]http://kdd.ics.uci.edu/databases/kddcup99/kddcup99

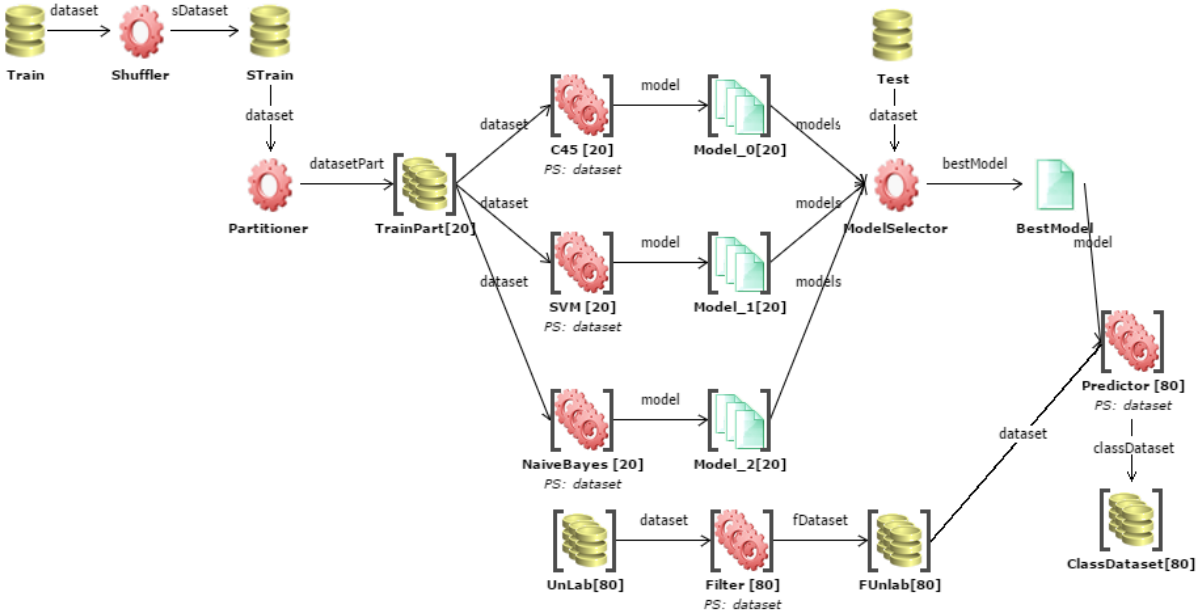


Figure 7. Classification VL4Cloud workflow.

understand the performance results, we have performed a preliminary evaluation of the expected performance of both Azure Storage and Hercules. Table I presents bandwidth performance of an I/O micro-benchmark consisting in writing and reading a 256 MB file, with 4 MB chunk size. Latency results are not relevant for data-intensive applications. Due to the usual large file size, the latency-related time is negligible in comparison with the bandwidth-related time.

Table I. Bandwidth micro-benchmark results (in MB/s).

Operation	Hercules local access	Hercules remote access	Azure Storage data access
Read	800	175	60
Write	1,000	180	30

After the initial setup, DMCF performs a series of preliminary operations (i.e., getting the task from the Task Queue, downloading libraries, and reading input files from the Cloud storage) and final operations (e.g., updating the Task Table, writing the output files to the Cloud storage). Table II lists all the read/write operations performed during the execution of the workflow on each data array. Each row of the table describes: i) the number of files included in the data array node; ii) the total size of the data array; iii) the total number of read operations performed on the files included in the data array; and iv) the total number of write operations performed on the files included in the data array. As can be noted, all the inputs of the workflow (i.e., $Train$, $Test$, $UnLab$) are never written on persistent storage, and the final output of the workflow (i.e., $ClassDataset$) is not used (read) by any other task.

Figure 8(a) shows the turnaround times of the workflow executed in the three scenarios introduced earlier: *Azure-only*, *Locality-agnostic*, *Data-aware*. In all scenarios, the turnaround times have been measured varying the number of workers used to run it on the Cloud from 1 (sequential execution) to 64 (maximum parallelism). In the Azure scenario, the turnaround time decreases from 1 hour and 48 minutes on a single worker, to about 2.6 minutes using 64 workers. In the Locality-agnostic scenario, the turnaround time decreases from 1 hour and 34 minutes on a single worker, to about 2.2 minutes using 64 workers. In the Data-aware scenario, the turnaround time decreases from 1 hour and 26 minutes on a single worker, to about 2 minutes using 64 workers. It is worth noticing that, in all the configurations evaluated, locality-agnostic allowed us to reduce the execution time in about 13%

Table II. Read/write operations performed during the execution of the workflow.

Data node	N. of files	Total size	N. of read operations	N. of write operations
Train	1	100MB	1	-
Strain	1	100MB	1	1
TrainPart	20	100MB	60	20
Model	60	≈20MB	60	60
Test	1	50MB	1	-
BestModel	1	300KB	80	1
UnLab	80	8GB	80	-
FUnLab	80	≈8GB	80	80
ClassDataset	80	≈6GB	-	80

compared to the Azure scenario, while data-aware allowed us to reduce the execution time in about 20% compared to the Azure scenario.

We also evaluated the overhead introduced by DMCF in the three scenarios. We define as overhead the time required by the system to perform a series of preliminary operations (i.e., getting the task from the Task Queue, downloading libraries and reading input files from the Cloud storage) and final operations (e.g., updating the Task Table, writing the output files to the Cloud storage) related to the execution of each workflow task. Table III shows the overhead time of the workflow in the three analyzed scenarios. We observe that the overhead in the Azure-only scenario is 40 minutes, while in the Locality-agnostic scenario is 26 minutes and in the Data-aware is 18 minutes. This means that using Hercules for storing intermediate data we were able to reduce the overhead by 36% using a locality-agnostic approach, and by 55% using a data-aware approach.

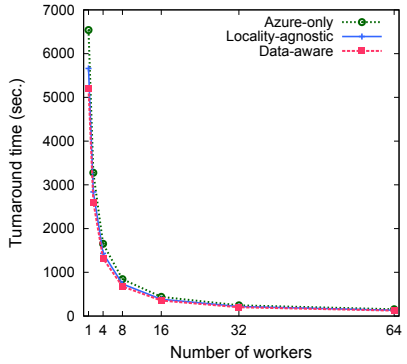
Table III. Overhead in the three scenarios.

	Total time (sec.)	Overhead (sec.)
Azure-only	6,487	2,382
Locality-agnostic	5,624	1,519
Data-aware	5,200	1,086

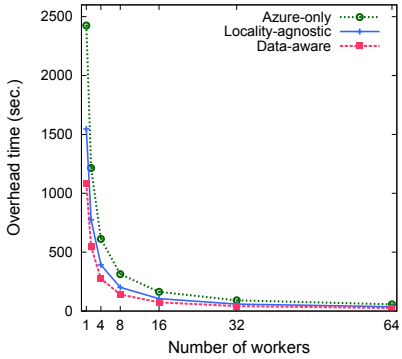
Figure 8(b) presents the time required by the application to perform every I/O operation of the application, and Figure 8(c) increases the level of detail, showing only the operations affected by the use of the Hercules service, i.e., I/O operations that work on temporary data. As shown in the figure, the difference between the three strategies is clear.

In order to better show the impact of the Hercules service, Figure 8(d) presents a breakdown of the total execution time, detailing the time spent on each of the tasks executed by the workflow application: computation tasks (CPU Time), I/O tasks over input/results files stored in Azure Storage, and I/O operations performed over temporary files. This figure clearly shows how the time required for I/O operations over temporary files, the only operations affected by use of Hercules services, are reduced to be almost negligible during the execution of the workflow, showing a great potential for increasing the I/O performance in data-intensive applications with large amounts of temporary data.

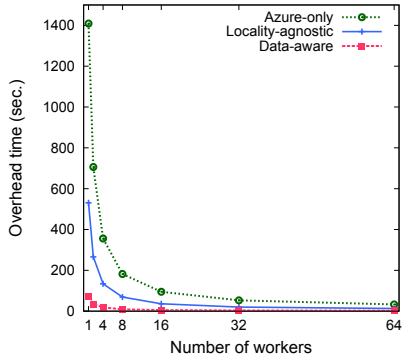
We further evaluated the performance of the data-aware scheduler by considering two more scenarios: 1) the size of the unlabeled input data provided to the classification workflow is increased by a factor two; 2) the number of temporary files generated during classification is increased by a factor two. The new experimental results, reported in Figure 9 and Table IV, show that: 1) with 1x input data, the data-aware scheduler is able to reduce the total execution time by 20% compared to the Azure-only strategy, whereas with 2x input data, the reduction passes to 24%; 2) with 1x number of temporary files, the data-aware scheduler is able to reduce the total execution time by 20% compared to the Azure-only strategy, whereas with 2x temporary files, the reduction passes to 33%. Therefore, in the first case, the new data-aware scheduler leads to an additional 4% of execution time reduction, while, in the second case, there is an additional 13% of execution time reduction.



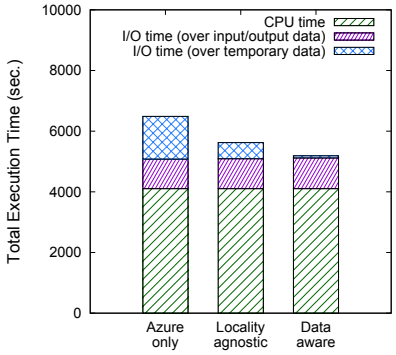
(a) Turnaround time vs number of available workers.



(b) Overhead time vs number of available workers.



(c) Overhead time vs number of available workers of only tasks that read/write temporary data.



(d) Breakdown of the total execution time.

Figure 8. Performance evaluation of the classification workflow using Azure-only, locality-agnostic and data-aware configurations.

This is due to the fact that our scheduler is able to reduce the I/O time over temporary data, which is impacted more by increasing the number of temporary data, than by increasing the size of input data.

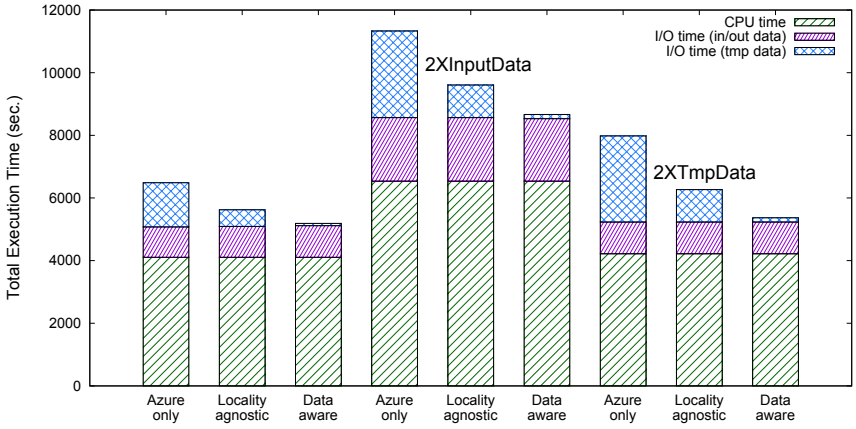


Figure 9. Breakdown of the total execution time by increasing the size of input data (2XInputData) and the number of temporary files (2XTmpData).

Table IV. Total execution time and overhead time of the classification workflow, by increasing the size of input data (2XInputData) and the number of temporary files (2XTmpData).

Configuration	Azure-only		Locality-agnostic		Data-aware	
	Total time (sec.)	*Overhead (sec.)*	*Total time (sec.)*	*Overhead (sec.)*	*Total time (sec.)*	*Overhead (sec.)*
1X	6,487	2,382	5,624	1,519	5,200	1,086
2XInputData	11,330	4,791	9,605	3,066	8,666	2,127
2XTmpData	7,980	3,760	6,266	2,046	5,367	1,147

8. CONCLUSIONS

DMCF allows users to design and execute data analysis workflows on cloud platforms, relying on cloud storage services for every I/O operation. Hercules is an in-memory I/O solution that can be used in DMCF as an alternative to cloud storage services, providing additional performance and flexibility features. The goal of this work is to implement an effective integration between DMCF and Hercules by using a data-aware scheduling strategy for exploiting data locality in data-intensive workflows.

The paper presented an experimental evaluation performed to analyze the performance of the proposed data-aware scheduling strategy executing data analysis workflows and to demonstrate the effectiveness of the solution. The experimental results show that, using the proposed data-aware strategy and Hercules as storage service for temporary data, the I/O overhead of workflow execution is reduced by 55% with respect to the standard execution based on the Azure storage, leading to a 20% reduction of the total execution time. These results demonstrate the effectiveness of our data-aware scheduling approach in improving the performance of data-intensive workflow execution in cloud platforms.

Acknowledgment

This work is partially supported by EU under the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS). This work has been partially supported by the Spanish MINISTERIO DE ECONOMÍA Y COMPETITIVIDAD under the project grant TIN2016-79637-P towards unification of HPC and Big Data paradigms.

REFERENCES

1. Leonid Oliker, Rupak Biswas, Rob Van der Wijngaart, David Bailey, and Allan Snavely. Performance evaluation and modeling of ultra-scale systems. *Parallel Processing for Scientific Computing (Siam 2006)*, pages 77–93, 2006.
2. Domenico Talia, Paolo Trunfio, and Fabrizio Marozzo. *Data Analysis in the Cloud: Models, Techniques and Applications*. Elsevier, 2015.
3. A. Burd et al. Pi of the sky-all-sky, real-time search for fast optical transients. *New Astronomy*, 10(5):409 – 416, 2005.
4. O. Rubel, C.G.R. Geddes, Min Chen, E. Cormier-Michel, and E.W. Bethel. Feature-based analysis of plasma-based particle acceleration data. *Visualization and Computer Graphics, IEEE Transactions on*, 20(2):196–210, 2014.
5. T. Tucker, M. Marra, and JM. Friedman. Massively parallel sequencing: The next big thing in genetic medicine. *The American Journal of Human Genetics*, 85(2):142–154, 2009.
6. Francisco Rodrigo Duro, Fabrizio Marozzo, Javier Garcia Blas, Domenico Talia, and Paolo Trunfio. Exploiting in-memory storage for improving workflow executions in cloud platforms. *The Journal of Supercomputing*, pages 1–20, 2016.
7. Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. A workflow management system for scalable data mining on clouds. *IEEE Transactions on Services Computing*, PP(99):1–1, 2016.
8. Francisco Rodrigo Duro, Javier Garcia Blas, and Jesus Carretero. A hierarchical parallel storage system based on distributed memory for large scale systems. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 139–140, New York, NY, USA, 2013. ACM.
9. Douglas Thain and Miron Livny. Parrot: Transparent user-level middleware for data-intensive computing. *Scalable Computing: Practice and Experience*, 6(3):9–18, 2005.
10. Douglas Thain, Christopher Moretti, and Jeffrey Hemmes. Chirp: a practical global filesystem for cluster and grid computing. *Journal of Grid Computing*, 7(1):51–72, 2009.
11. Florin Isaila, Francisco Javier Garcia Blas, Jesús Carretero, Wei-Keng Liao, and Alok Choudhary. A Scalable Message Passing Interface Implementation of an Ad-Hoc Parallel I/O System. *Int. J. High Perform. Comput. Appl.*, 24(2):164–184, May 2010.

12. Samer Al-Kiswany, Abdullah Gharaibeh, and Matei Ripeanu. The case for a versatile storage system. *Operating Systems Review*, 44(1):10–14, 2010.
13. L.B. Costa, H. Yang, E. Vairavanathan, A. Barros, K. Maheshwari, G. Fedak, D. Katz, M. Wilde, M. Ripeanu, and S. Al-Kiswany. The case for workflow-aware storage:an opportunity study. *Journal of Grid Computing*, pages 1–19, 2014.
14. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI'12*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012.
15. Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Reliable, memory speed storage for cluster computing frameworks. Technical Report UCB/EECS-2014-135, EECS Department, University of California, Berkeley, Jun 2014.
16. Zhao Zhang, Daniel S. Katz, Timothy G. Armstrong, Justin M. Wozniak, and Ian Foster. Parallelizing the execution of sequential scripts. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 31:1–31:12, New York, NY, USA, 2013. ACM.
17. Patrick Donnelly, Nicholas Hazekamp, and Douglas Thain. Confuga: Scalable Data Intensive Computing for POSIX Workflows. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015.
18. J. Lofstead, A. Champsaur, J. Dayal, M. Wolf, and G. Eisenhauer. Superglue: Standardizing glue components for hpc workflows. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 170–171, Sept 2016.
19. Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P. Berman, and Phil Maechling. Data sharing options for scientific workflows on amazon ec2. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–9, 2010.
20. Piotr Bryk, Maciej Malawski, Gideon Juve, and Ewa Deelman. Storage-aware algorithms for scheduling of workflow ensembles in clouds. *Journal of Grid Computing*, 14(2):359–378, 2016.
21. Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. Js4cloud: Script-based workflow programming for scalable data analysis on cloud platforms. *Concurrency and Computation: Practice and Experience*, 27(17):5214–5237, 2015.
22. Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. A cloud framework for big data analytics workflows on azure. *Advances in Parallel Computing*, 23:182–191, 2013. ISBN 978-1-61499-321-6.
23. Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.
24. Francisco Rodrigo Duro, Fabrizio Marozzo, Javier Garcia Blas, Jesus Carretero, Domenico Talia, and Paolo Trunfio. Evaluating data caching techniques in dmcf workflows using hercules. In *In Proceedings of the Second International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2015)*, pages 95–106, Krakow, Poland, 2015.
25. J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
26. S.S. Keerthi, S.K. Shevade, C. Bhattacharyya, and K.R.K. Murthy. Improvements to platt's smo algorithm for svm classifier design. *Neural Computation*, 13(3):637–649, 2001.
27. George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, San Mateo, 1995. Morgan Kaufmann.
28. Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowl. Inf. Syst.*, 14(1):1–37, December 2007.