


Exploiting in-memory storage for improving workflow executions in cloud platforms

Francisco Rodrigo Duro¹ · Fabrizio Marozzo²  ·
Javier Garcia Blas¹ · Domenico Talia² ·
Paolo Trunfio²

Published online: 27 February 2016
© Springer Science+Business Media New York 2016

Abstract The Data Mining Cloud Framework (DMCF) is an environment for designing and executing data analysis workflows in cloud platforms. Currently, DMCF relies on the default storage of the public cloud provider for any I/O-related operation. This implies that the I/O performance of DMCF is limited by the performance of the default storage. In this work, we propose the usage of the Hercules system within DMCF as an ad hoc storage system for temporary data produced inside workflow-based applications. Hercules is a distributed in-memory storage system highly scalable and easy to deploy. The proposed solution takes advantage of the scalability capabilities of Hercules to avoid the bandwidth limits of the default storage. We evaluated the performance of Hercules compared with the Microsoft Azure Storage solution by using synthetic benchmarks with the objective of demonstrating the viability of the proposed solution. Then, we evaluated the integration of Hercules and DMCF on a real application consisting of a workflow that accesses temporary data using either Azure storage or Hercules. The I/O overhead in this real-life scenario using Hercules has

✉ Fabrizio Marozzo
fmarozzo@dimes.unical.it

Francisco Rodrigo Duro
frodriego@arcos.inf.uc3m.es

Javier Garcia Blas
fjblas@arcos.inf.uc3m.es

Domenico Talia
talia@dimes.unical.it

Paolo Trunfio
trunfio@dimes.unical.it

¹ ARCOS, University Carlos III Madrid, Leganés, Spain

² DIMES, University of Calabria, Rende, Italy

been reduced by 36 % with respect to Azure storage, leading to a 13 % reduction of the total execution time. This confirms that our in-memory approach is effective in improving the performance of data-intensive workflow executions in cloud-based platforms.

Keywords DMCF · Hercules · Workflows · In-memory storage · Data cache · Microsoft Azure

1 Introduction

In the last decade, most of the scientific computing problems are increasing their needs to process large quantities of data. Large simulations, data visualization, and big data problems are some of the application areas leading the trends in scientific computing. This evolution is moving needs from a computing-centric power point of view to a data-centric approach. Current trends in high-performance computing (HPC) also include the use of cloud infrastructures as a flexible approach to virtually limitless computing resources. Given this current scenario, a solution that combines HPC, data analysis, and cloud computing is becoming more and more necessary.

According to their elastic feature, cloud computing infrastructures can serve as effective platforms for addressing the computational and data storage needs of most big data applications that are being developed nowadays. However, coping with and gaining value from cloud-based, big data requires novel software tools and advanced analysis techniques. Indeed, advanced data mining techniques and innovative tools can help users to understand and extract what is useful in large and complex datasets for making informed decisions in many business and scientific applications.

The Data Mining Cloud Framework (DMCF) [12] is an environment for designing and executing data analysis workflows in cloud platforms. Currently, DMCF uses the storage supplied by the cloud provider for any I/O related job. This implies that the I/O performance of DMCF is limited by the performance of the default storage system. Moreover, it is influenced by the contention that occurs when other I/O tasks are concurrently executed in the same region. Finally, the cost of using a persistent storage service to store temporary data should be also taken into account. The solution proposed here consists in using Hercules as storage system for temporary data produced in workflows. Hercules is a distributed in-memory storage system, easy to deploy and highly scalable [4].

This novel approach has three main objectives. The first one is taking advantage of the scalability of Hercules to avoid the bandwidth limits of the default storage. When the number of Hercules I/O nodes increases, the total available aggregated bandwidth usable by worker nodes is enhanced. The second objective is to allow the deployment, thanks to the easy deployment of Hercules, of an ad hoc and independent in-memory storage system to avoid the contention produced during peak-loads in the cloud storage service. The last objective is the independence from the cloud platform used. While each cloud infrastructure has different APIs to access their storage services, Hercules has interfaces for commonly used APIs (like POSIX-like, put/get, MPI-IO) in order to imply minor modifications to existing code.

An extensive evaluation has been performed to evaluate the performance of Hercules compared with Microsoft Azure storage using synthetic benchmarks to demonstrate the viability of the solution [14]. Furthermore, we evaluated the integration of Hercules and DMCF on a real application consisting of a workflow that can access to temporary data using either Azure storage or Hercules. Using Hercules, the I/O overhead in this real-life scenario has been reduced by 36 % with respect to Azure storage, leading to a 13 % reduction of the total execution time. This confirms that our in-memory approach is effective in improving the performance of data-intensive workflow execution in cloud platforms.

The remainder of this paper is organized as follows. Section 2 describes the main features of DMCF. Section 3 introduces Hercules architecture and capabilities. Section 4 emphasizes the advantages of integrating DMCF and Hercules and outlines how this integration will work. Section 5 evaluates the performance of Hercules compared with Microsoft Azure storage. Section 6 evaluates the integration between DMCF and Hercules. Section 7 discusses related work. Finally, Sect. 8 concludes the paper.

2 Data Mining Cloud Framework

The Data Mining Cloud Framework (DMCF) [12] is a software system designed for designing and 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 components (see Fig. 1).

The storage components include:

- A *Data Folder* that contains data sources and the results of knowledge discovery processes. Similarly, a *Tool Folder* contains libraries and executable files for data selection, pre-processing, transformation, data mining, and evaluation of results.

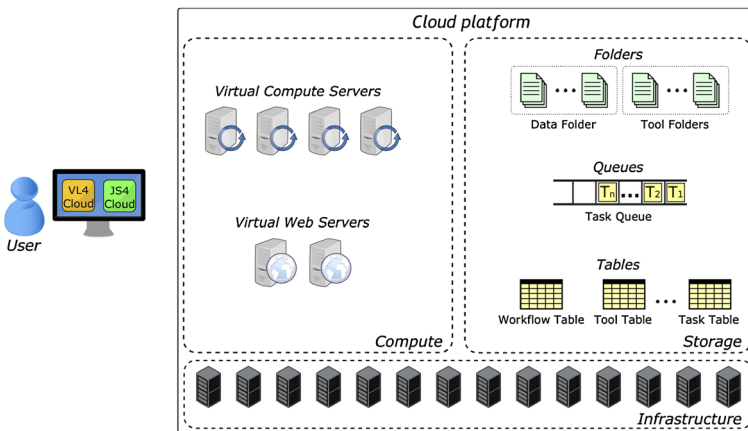


Fig. 1 Architecture of Data Mining Cloud Framework

- *Data Table*, *Tool Table* and *Task Table* contain metadata information associated with data, tools, and tasks.
- The *Task Queue* contains the tasks that are ready for execution.

The compute components are:

- A pool of *Virtual Compute Servers* (or *Workers*), which are in charge of executing the data analysis tasks.
- 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.¹

A user interacts with the system to perform the following steps for designing and executing a knowledge discovery application [10]:

1. The user accesses the Website and designs the workflow through a Web-based interface.
2. After submission, the system creates a set of tasks and inserts them into the Task Queue on the basis of the workflow.
3. Each idle Virtual Compute Server picks a task from the Task Queue, and concurrently executes it.
4. Each Virtual Compute Server gets the input dataset from the location specified by the workflow. To this end, a file transfer is performed from the Data Folder where the dataset is located to the local storage of the Virtual Compute Server.
5. After task completion, each Virtual Compute Server puts the results on the Data Folder.
6. The Website notifies the user as soon as her/his task(s) have been completed, and allows her/him to access the results.

The set of tasks created on the second step depends on how many data analysis tools are invoked within the workflow. Initially, only the workflow tasks without dependencies are inserted into the Task Queue. All the potential parallelism of the workflow is exploited by using all the needed Virtual Compute Servers.

DMCF allows to program data analysis workflows using two languages:

- VL4Cloud (Visual Language for Cloud), a visual programming language that lets users develop applications by programming the workflow components graphically [11].
- JS4Cloud (JavaScript for Cloud), a scripting language for programming data analysis workflows based on JavaScript [12].

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.).

¹ <http://azure.microsoft.com>.

Another common element is the Task concept, which represents the 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 runtime 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 .

3 Hercules

Hercules [4] is a distributed in-memory storage system based on the key/value Memcached database [5]. The distributed memory space can be used by the applications as a virtual storage device for I/O operations and has been especially adapted in this work for being used as an in-memory shared storage for cloud infrastructures. Our solution relies on an improved version of Memcached servers, for offering an alternative storage solution to the default cloud storage service provided by Azure.

As can be seen in the Fig. 2, Hercules architecture has two main layers: front-end (worker library) and back-end (server layer). On top is the worker user-level library

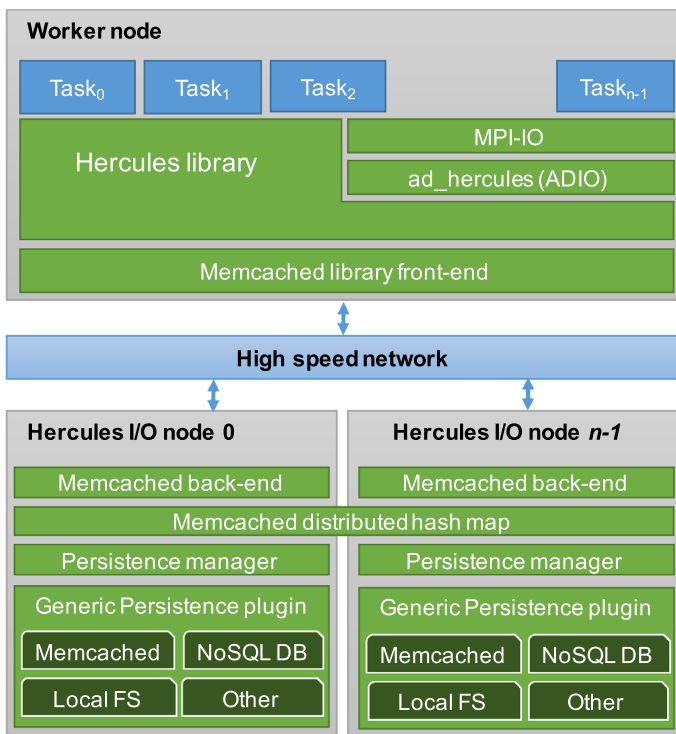


Fig. 2 Hercules architecture. On the *top* the worker-side, a user-level library. At the *bottom* the server-side with the Hercules I/O nodes divided in modules

based on a layered design. Back-ends are based on the Memcached server, extending its functionality with persistence and tweaks. Main advantages offered by Hercules are: scalability, easy deployment, flexibility, and performance.

Scalability is achieved by fully distributing data and metadata information among all the nodes, avoiding the bottlenecks produced by centralized metadata servers. Data and metadata placement is completely calculated in the worker-side by a hash algorithm. The servers, on the other hand, are completely stateless.

Easy deployment and flexibility at worker-side are tackled using a POSIX-like user-level interface (open, read, write, close, etc.) in addition to classic put/get approach existing in current NoSQL databases. Existing software requires minimum changes to run using Hercules. The layered design offers the possibility of performing any future change with the minimum required effort. Servers can be deployed in any kind of Linux systems at user level, without requiring any special privileges. Persistence can be easily configured using the existing plugins or developing new ones. An MPI-IO interface is also available for legacy software relying on MPI as communication system.

Finally, performance and flexibility at server-side are targeted by exploiting the parallel I/O capabilities of Memcached servers. Flexibility is achieved by Hercules due to its ease of deployment dynamically on as many nodes as necessary. Each node can be accessed independently, multiplying the total throughput peak performance. Furthermore, each node can serve requests in a concurrent way thanks to a multi-threading implementation. The combination of these two factors results in full scalability: both when the number of nodes increases and when the number of workers running on the same node increases.

4 Integration between DMCF and Hercules

The final objective of this joint research work is the integration of DMCF and Hercules. As can be seen in Fig. 3, Hercules and DMCF can be configured in more than one deployment scenarios to achieve different levels of integration.

The first scenario shows the original approach of DMCF, where every I/O operation is done against the cloud storage service offered by the cloud provider, which is Azure

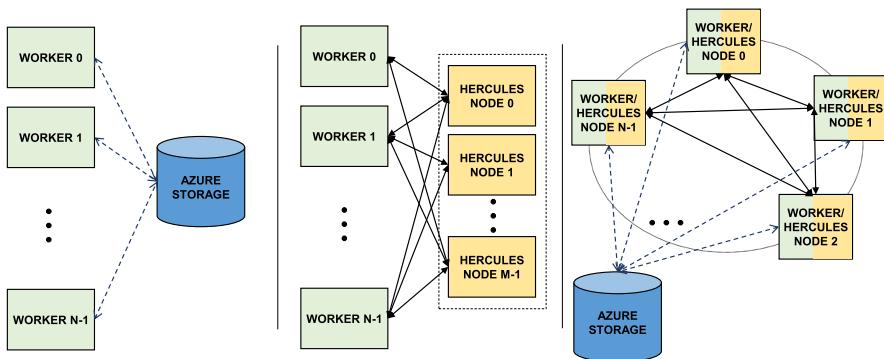


Fig. 3 Deployment scenarios for the combination of Hercules and DMCF infrastructures

Storage in this work. While this storage service is suitable for persistent data, it could be inefficient for temporary data. The main benefits of a cloud storage service are the convenience of using every tool offered by the same provider and the persistence options offered, even in different geographical regions. Nevertheless, there are, at least, four disadvantages about this approach. First, proprietary interfaces and tools to access the storage service offered by different providers. Second, the performance offered by this services could have limitations that cannot be avoided and performance could not be stable when there are peaks of use by other users. Third, the storage services are offered in a closed configuration, and cannot be customized for the necessities of users at any time. Fourth, the cloud philosophy is tightly related with the pay-per-use concept. However, it does not make sense to pay for temporary data as if it was persistent data.

The second scenario, and the first contribution of this paper, is to use Hercules as the default storage for temporary generated data. Temporary data are becoming more and more popular in data analysis and many-task based applications. Most of these applications are developed as a sequence of tasks that communicate by using temporary files. 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. Even the instance type can be configured according to the necessities of each different application. As stated in Sect. 3, Hercules offers different user-level interfaces such as POSIX-like, put/get, and MPI-IO, allowing a more flexible deployment of legacy applications than the default cloud storage service. Cost-wise a better study of the competition between using a persistence-focused service against launching Hercules I/O node instances as temporary storage is needed.

The third scenario shows an even 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. If the data needed by a DMCF worker is stored inside the Hercules I/O node running in the same instance, it will not be necessary to use the network for accessing data, thus every I/O operation will be completely local.

5 Evaluating in-memory versus persistent storage operations

To demonstrate the capabilities of Hercules in accelerating the I/O operations of DMCF workers, we evaluated the performance of the Azure Storage service against our proposed solution. For this purpose, we have designed and implemented a simple benchmark, referred from now on as *Filecopy Benchmark*. In this benchmark, a configurable number of workers perform two simple tasks per worker: the first one is writing files to the configured storage (Azure Storage or Hercules) and, after the write task is complete, a read task starts over the data written previously. The benchmark is fully configurable in terms of:

- *Number of worker nodes* Each worker node is a VM deployed in Azure.
 - *Number of workers per node* Worker processes running in the same node in parallel.
- This parameter is important to evaluate how the storage solutions will behave in

Table 1 Azure instance type characteristics

| Type | Cores | RAM (GB) | Bandwidth (Mbps) ^a | Price (€/h) |
|------|-------|----------|-------------------------------|-------------|
| A0 | 1 | 0.75 | <100 | 0.017 |
| A1 | 1 | 1.75 | ~240 | 0.050 |
| A2 | 2 | 3.50 | ~480 | 0.101 |
| A3 | 4 | 7.00 | ~960 | 0.202 |
| A4 | 8 | 14.00 | ~1700 | 0.408 |
| D1 | 1 | 3.50 | ~480 | 0.097 |
| D2 | 2 | 7.00 | ~900 | 0.194 |
| D3 | 4 | 14.00 | ~1600 | 0.388 |
| D4 | 8 | 28.00 | ~2000 | 0.776 |

^a Bandwidth measured experimentally using *iperf* tool between two VMs of the same instance type in the same region

multi-core architectures and how they perform when different worker processes share the same network interface.

- *File size* The total size in MegaBytes (MB) of the file can be configured to simulate different problem sizes.
- *Chunk size* In Azure storage, a BLOB object is divided into blocks (maximum block size of Azure Storage is 4 MB, not enough for large files). The Java library used for accessing Azure Storage, automatically divides a block object in the required number of block objects. In addition to this behavior, our implementation divides a file into different BLOB objects. Chunk size parameter is the size of each of the block objects that are part of a complete file. In Hercules, it corresponds to the buffer size of the POSIX write operation. Internally, Hercules divides the files in blocks adapted to the key-value hashmap of Memcached.

The computing resources used during the evaluation are completely based on Microsoft Azure. Table 1 shows the characteristics of the different instance types used during our evaluation. All the resources used were located on the “Western Europe” region and the OS installed on the VMs was Ubuntu 14.04 LTS. It is also worth noting that as the objective of the research work is to use Hercules as temporary storage, persistence features are disabled.

5.1 Chunk size evaluation

For the first evaluation case, we have fixed the file size to 128 MB, to have a file size that is big enough to show the performance with different chunk sizes. The chunk size will vary during the evaluation and we have used the five standard (A0–A4) instance types. Figure 4a shows the performance achieved during the write operations, and Fig. 4b the read operations performance. As it can be seen in these figures, Azure Storage performs much better for read (up to 72 MB/s) than for write operations (up to 38 MB/s). Also, the performance increases with the chunk size, achieving the best performance around the 32 MB mark. Finally, it is interesting to note how the performance varies with the instance type used: as expected, the most expensive instances have the better performance.

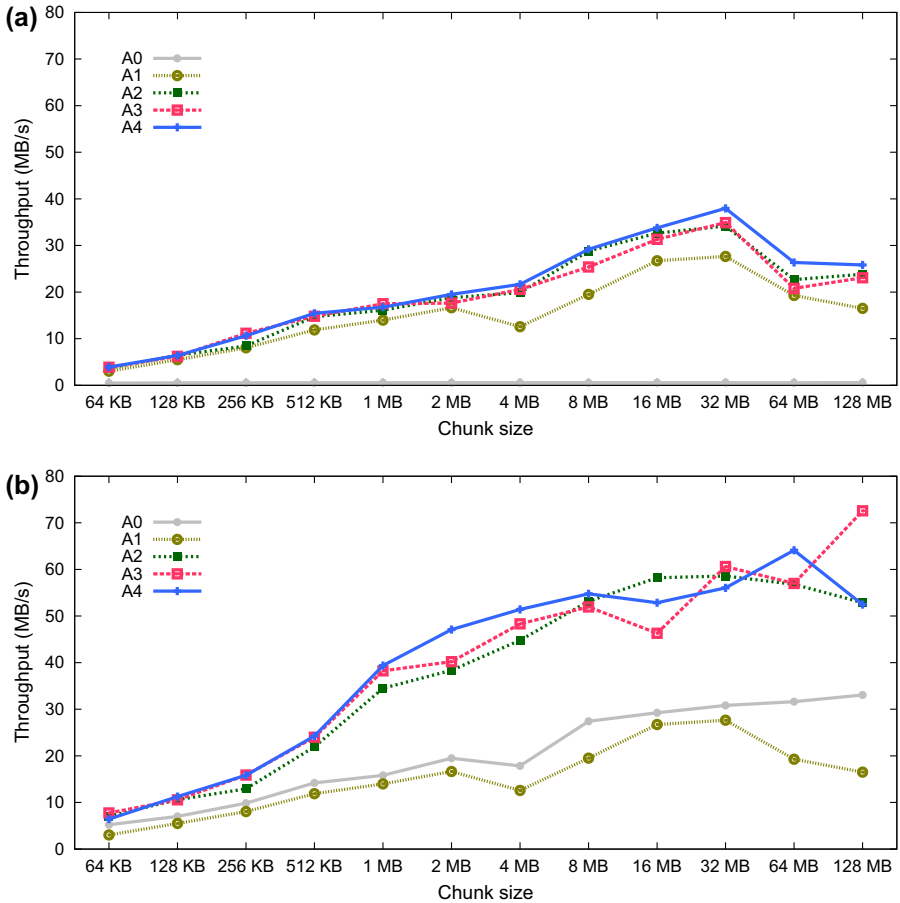


Fig. 4 File copy benchmark configured for evaluating the Azure Storage performance depending on the block object size. **a** Throughput of Azure storage by using the Filecopy Benchmark (128 MBytes) for evaluating the block size for writes. **b** Throughput of Azure storage by using the Filecopy Benchmark (128 MBytes) for evaluating the block size for reads

5.2 Hercules I/O nodes scalability

The next phase in the evaluation process is the measurement of the performance difference between Azure Storage and Hercules using different configurations. Also, we evaluate how Hercules scales its performance as the number of deployed I/O nodes increases. After some quick bandwidth evaluation cases (results shown in Table 1), we selected D1 and D2 instances as the best performers in network bandwidth per core ratio. D1 instances achieve a peak performance of 60 MB/s using one core while D2 tops at around 115 MB/s with two cores, managing to reach almost the best possible performance of the available Gigabit virtual network interface. This is $2\times$ the bandwidth available per core compared with standard 'AX' instance types. In the future, it would be interesting to evaluate the performance achieved by Hercules

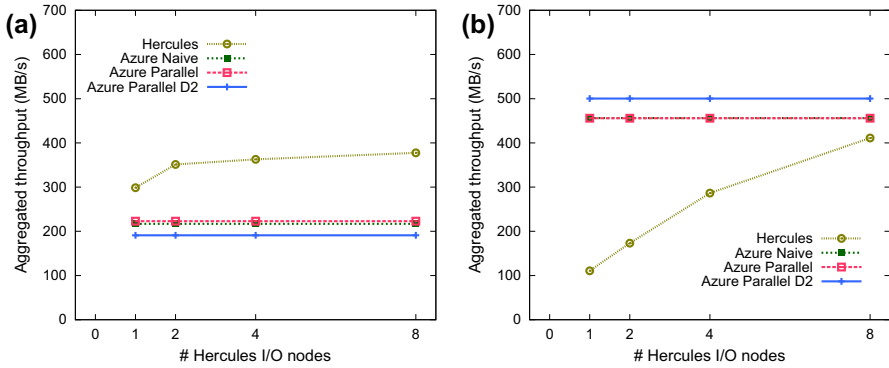


Fig. 5 Filecopy Benchmark configured for evaluating the Hercules I/O nodes scalability. 8 worker processes running on 8 worker nodes access 4 Gigabyte of data. Hercules performance is up to 2× better than Azure Storage in write operations while performing nearly as good as Azure Storage in the best read cases. **a** Throughput of Hercules by using the Filecopy Benchmark for evaluating the scalability of I/O nodes for writes. We set up the experiment with 8 worker nodes, writing 512 MBytes each one (4 GBytes in total). **b** Throughput of Hercules by using the Filecopy Benchmark for evaluating the scalability of I/O nodes for reads. We set up the experiment with 8 worker nodes, reading 512 MBytes each one (4 GBytes in total)

running in the A8 and A9 network-optimized instances with Infiniband network, and 56 and 112 Gigabytes of RAM, respectively. This network-optimized instances should be the optimal option for running Hercules I/O nodes.

The final selection for this test is 8 VMs (D1 instances) as worker nodes and up to 8 VMs (D2 instances) as Hercules I/O nodes. Figure 5 plots the Filecopy Benchmark results, configuring the experiment with a file size of 512 MB, with 32 MB of chunk size and executing one read/write operation per worker node (one worker process per node) which implies a 4096 MB problem size (512 MB × 8 worker nodes). We have compared four different cases. The first one is the performance obtained by Hercules using between 1 and 8 I/O nodes. The second case is Azure Storage baseline approach, using the default access pattern offered by the Java API, without any optimizations. Third case is Azure Storage applying some optimizations to the code, specially important is setting up the *BlobRequestOptions* object property *setConcurrentRequestCount* with 8 threads per process, using 8 concurrent threads to parallel access to Azure Storage. The last case cannot be directly compared with the performance achieved by Hercules, because it uses the reserved D2 instances as worker nodes, instead of using them as I/O nodes, to show the peak performance achievable by Azure Storage with fully working Gigabit interface, hence the dotted line. In the Hercules case, the peak performance is limited by the aggregated bandwidth available worker-side (8×60 MB/s ~ 480 MB/s), not by the server-side 8×115 MB/s (~ 920 MB/s).

Figure 5a shows the performance evolution as the number of Hercules I/O nodes increase compared to the different Azure Storage approaches. The figure clearly demonstrates how Hercules performance tops near the 400 MB/s mark, which is near the maximum theoretical peak performance of 8×60 MB/s (~ 480 MB/s). This peak performance achieved using 8 I/O nodes for parallel access is nearly 2× the performance achieved by Azure Storage in any of the configurations. Some interesting sights in the Azure Storage side are how both the baseline and the parallel approach perfor-

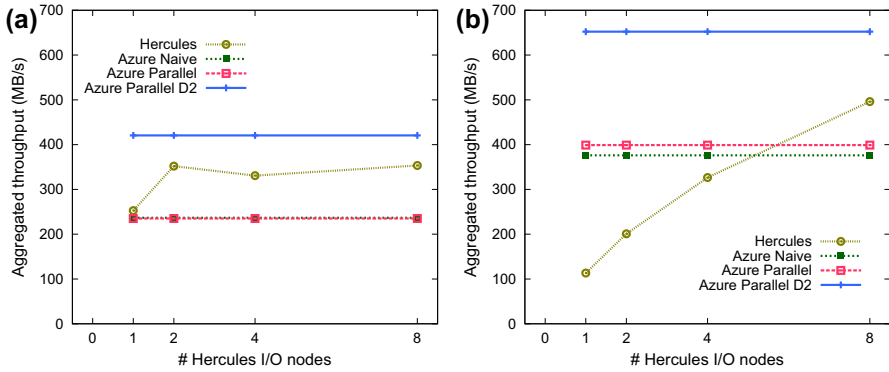


Fig. 6 File copy benchmark configured for evaluating the Hercules I/O nodes scalability. 32 worker processes running on eight worker nodes (4 processes per node) access 4 Gigabyte of data. Hercules performance is up to 2× better than Azure Storage in write operations while performing nearly as good as Azure Storage in the best read cases. **a** Throughput of Hercules by using the Filecopy Benchmark for evaluating the scalability of I/O nodes for writes. We set up the experiment with 8 clients and 4 process per node, writing 128 MBytes each process (4 GBytes in total). **b** Throughput of Hercules by using the Filecopy Benchmark for evaluating the scalability of I/O nodes for reads. We set up the experiment with 8 clients and 4 process per node, reading 128 MBytes each process (4 GBytes in total)

mance is nearly identical caused by only being one core available in D1 instances. Also, it is interesting how the D2 instances performance using parallel accesses is even lower, exposing the deficiencies of Azure Storage performance in write operations.

In Fig. 5b, which depicts the read operations performance, it can be clearly seen how the Hercules performance evolves as the number of I/O nodes available increases. With only one I/O node available, the performance is ~100 MB/s, the maximum offered by the network interface of the I/O node (D2 instance). As the number of I/O nodes increases, the performance evolves, reaching a peak performance of ~400 MB/s, again near the theoretical up mark of 480 MB/s and near the performance of Azure Storage that slightly outperforms Hercules in this case. Azure Storage performs at the peak performance of the available network, with same performance in naive and parallel approaches using D1 instances while performing marginally better when D2 instances are used as worker nodes.

Third evaluation case is an evolution of the previous test for a scenario with higher congestion using the same infrastructure (8 D1 instances as worker nodes and 8 D2 instances as Hercules I/O nodes). In this case, instead of having 1 worker running on each node, we launched 4 workers running in parallel on each of the worker nodes, keeping the problem size in 4096 MB. For this purpose, each worker process writes, and then reads, a 128-MB file, with the same chunk size of 32 MB.

Figure 6a, showing the performance in write operations, reports a very similar behavior of Hercules compared to the previous test case, but achieving a lower peak performance. At the same time, Azure Storage performance with D1 instances increases and the difference between Hercules and Azure Storage is narrowed to a 50 % difference in favor of Hercules. Furthermore, using more than one process per node in the dual-core D2 instances, doubles the performance obtained by Azure Storage than Hercules in this special case.

On the other hand, on Fig. 6b, related with read operations, the peak performance of Hercules is even higher than the previous case, fully utilizing the ~ 480 MB/s of the available aggregated bandwidth at client-side and surpassing the peak throughput performance of Azure Storage accessed from D1 instances. When Azure Storage is accessed by D2 instances with more than one process reading in parallel from different files, the performance is almost doubled, in a similar way seen in the write operations.

As conclusions of the last two cases, we can emphasize how the aggregated throughput of the workers accessing the Hercules storage system approaches the theoretical maximum bandwidth available in every studied case, showing the scalability capabilities of our proposed solution. The performance in write operations is between $1.5\times$ and $2\times$ the performance achieved by Azure Storage with a similar architecture, while the performance in read operations in first case is marginally in favor of Azure and in the second case is comparable.

5.3 Worker nodes strong scalability

The last test cases focus on evaluating the behavior of our solution with an increasing number of worker nodes accessing the Hercules storage system. The objective is to evaluate the impact of the congestion against Azure Storage. The test cases are equivalent to the previous test cases, with Hercules using always 8 I/O nodes, while Azure Storage is evaluated using the native approach and the optimized parallel implementation. The aim of this test is to study a strong scalability scenario, where an increasing number of worker nodes perform the same total work: writing 8×512 MB files, a total problem size of 4096 MB, and then reading them. As expected, as the number of worker nodes increases, the total available bandwidth increases at the same pace, leading to better peak throughput performance, but the bottleneck continues at client-side.

Figure 7 shows the same trends already explained in the previous test cases. In Fig. 7a, which represents the aggregated throughput in write operations, can be seen how Hercules is always reaching the theoretical peak performance of each configuration, and how its performance is better than Azure Storage in every case, even doubling the performance in the most favorable one.

In the read operations performance case, Fig. 7b, again Hercules takes advantage of the available bandwidth in every case and competes really well with Azure Storage but the case of 8 clients where the Azure Parallel performance is better.

From the results of the evaluation, we can conclude that Hercules is capable of fully utilizing the available bandwidth of every infrastructure where it has deployed. Furthermore, the scalability is assured in any case, on one hand when the number of I/O nodes deployed increases and, on the other hand, when the number of concurrent worker nodes scales and the congestion is higher. Compared to Azure Storage, our proposed solution is up to $2\times$ better in performance during write operations and competes on equal conditions on read operations. Furthermore, it should be noted that every test case evaluated in this work uses the best possible configuration for Azure Storage, as explained at the beginning of this section; the same performance could be predicted for Hercules in other scenarios while Azure Storage is expected to be penalized.

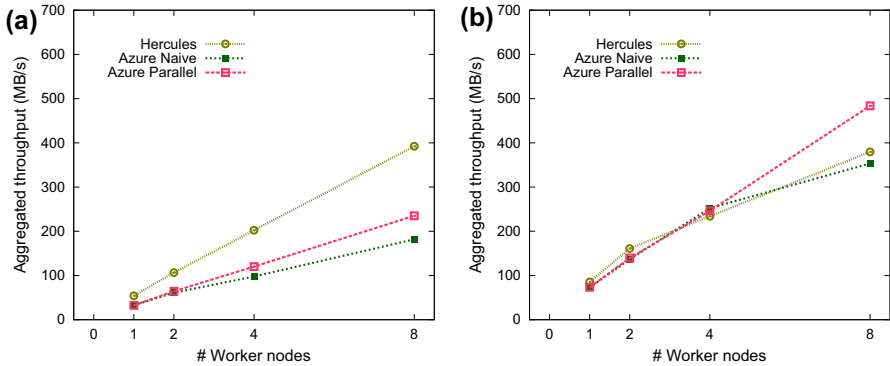


Fig. 7 File copy benchmark configured for comparing Azure Storage and Hercules performance with an increasing number of worker nodes accessing the storage concurrently. Hercules is configured with 8 I/O nodes and from 1 to 8 worker nodes access to the storage systems concurrently. Hercules performance is up to 2× better than Azure Storage in write operations while performing nearly as good as Azure Storage in most cases. **a** Throughput varying the worker nodes from 1 to 8, writing 8 files (512 MBytes) per node. We set up the experiment with 8 I/O nodes in case of Hercules. **b** Throughput varying the worker nodes from 1 to 8, reading 8 files (512 MBytes) per node. We set up the experiment with 8 I/O nodes in case of Hercules

6 Evaluating the integration between DMCF and Hercules

In this section, we show the evaluation results of the integration between DMCF and Hercules through the execution of a data analysis workflow using two alternative configurations. In the first configuration, every I/O operation of the workflow is done by DMCF using the Azure storage service, without exploiting the Hercules functionalities (Fig. 3, Scenario 1). In the second configuration, 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 (Fig. 3, Scenario 3). The goal is to evaluate the increase of performance obtained in the second configuration as compared to the first one.

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 the workflow are C4.5 [13], support vector machine (SVM) [8] and Naive Bayes [7], that are three of the main classification algorithms [16]. The training set, test set and unlabeled dataset, which represent the input of the workflow, have been generated from the *KDD Cup 1999*'s dataset,² which contains a wide variety of simulated intrusion records in a military network environment.

Figure 8 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

² <http://kdd.ics.uci.edu/databases/kddcup99/kddcup99>.

```

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 BMLRef = Data.define("BestModel");
15: ModelSelector({dataset:TeRef, models:MRef, bestModel:BMLRef});
16: var m = 80;
17: var DRef = Data.get("Unlab", m);
18: var FDFRef = Data.define("FUnlab", m);
19: for(var i = 0; i <m; i++)
20:   Filter({dataset:DRef[i], fDataset:FDFRef[i]});
21: var CRef = Data.define("ClassDataset", m);
22: for(var i = 0; i <m; i++)
23:   Predictor({dataset:FDFRef[i], model:BMLRef, classDataset:CRef[i]});

```

Fig. 8 Classification JS4Cloud workflow

$n = 20$ 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*, *Naive Bayes*). 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 9 shows the VL4Cloud version of the data mining workflow. The visual formalism clearly highlight 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 in the Azure infrastructure as needed (see Table 1), and configured them by

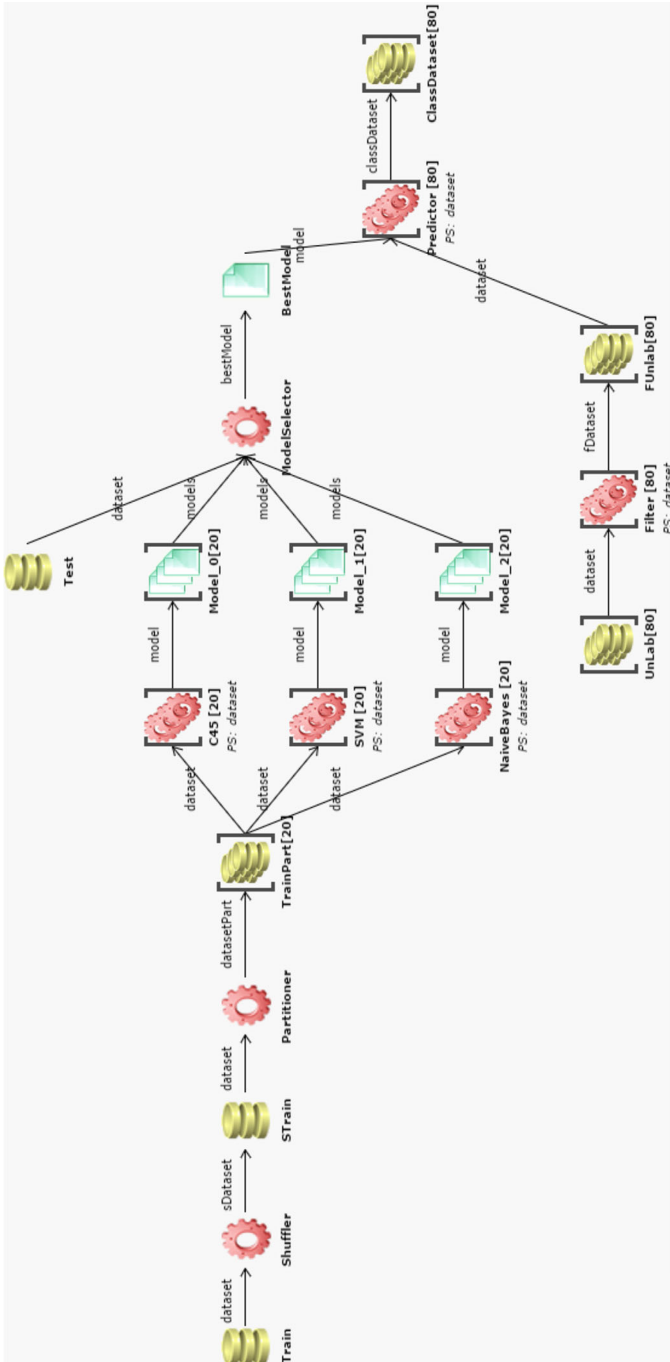


Fig. 9 Classification VL4Cloud workflow

Table 2 Read/write operations performed during the execution of the workflow

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

launching both the Hercules server process and the DMCF worker process on each VM. Then, 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 2 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 output of the workflow (i.e., *ClassDataset*) is never read.

Figure 10a shows the turnaround times of the workflow executed in the two scenarios introduced earlier: (1) DMCF relying on Azure Storage for every I/O operation (labeled as Azure); (2) DMCF relying on Azure Storage and Hercules for temporary data (labeled as Azure+Hercules). In both scenarios, the turnaround times have been measured varying the number of virtual servers used to run it on the cloud from 1 (sequential execution) to 8 (maximum parallelism). In the Azure scenario, the turnaround time decreases from 1 h and 48 min on a single server, to about 15 min using 8 servers. In the Hercules scenario, the turnaround time decreases from 1 h and 33 min on a single server, to about 12 min using 8 servers. It is worth noticing that, in all the configurations evaluated, Hercules allowed us to reduce the execution time of about 13 % compared to the Azure scenario.

The scalability in both scenarios can be further evaluated through Fig. 10b, which illustrates the relative speedup obtained by using up to 8 servers. In the Azure scenario, the speedup passes from 3.6 using 4 servers to 7.3 using 8 servers. In the Hercules scenario, the speedup passes from 3.7 using 4 servers to 7.5 using 8 servers.

We also evaluated the overhead introduced by DMCF in the two scenarios (Azure vs Azure+Hercules). 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, down-

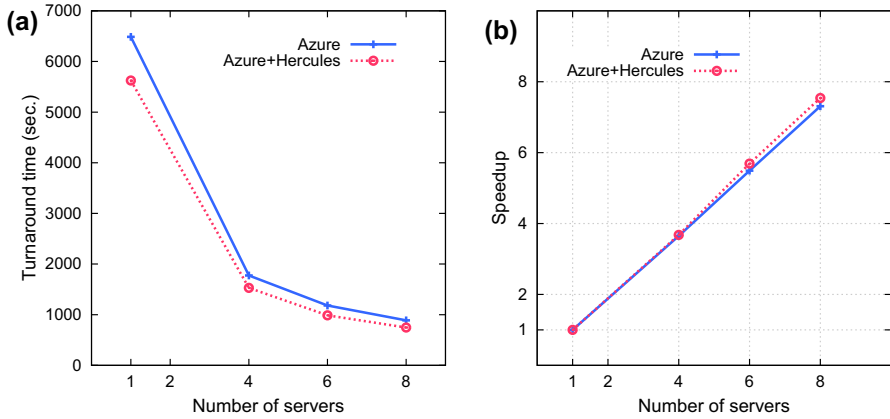


Fig. 10 Classification workflow evaluation using Azure Storage or Hercules as temporary storage service. **a** Turnaround time vs. number of available servers. **b** Speedup vs. number of available servers

Table 3 Overhead of DMCF in execution of the classification workflow using Azure Storage or Hercules as temporary storage service

| | Total time (s) | Overhead (s) |
|----------|----------------|--------------|
| Azure | 6487 | 2382 |
| Hercules | 5624 | 1519 |

loading 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 3 shows the overhead time of the workflow in the two analyzed scenarios. We observe that the overhead in the Azure scenario is 40 min, while in the Hercules scenario is 25 min. This means that using Hercules to store intermediate data we were able to reduce the overhead of a 36 %.

7 Related work

The current trends in scientific computing are greatly influenced by the new possibilities brought by some of the most popular new technologies. The use of cloud computing resources to solve existing scientific problems, and the exploitation of data-intensive techniques as novel approach for scientific research, have uncovered new weak points and bottlenecks of the classical high-performance I/O approaches. Data-intensive applications, especially where high concurrency in I/O operations appears (many-tasks, workflows, etc.), requires novel approaches to solve new challenges.

Some of the problems that have to be addressed in this new data-intensive tendency, are not completely new, and should be taken into account before addressing the new problems. Solutions like Parrot, Chirp, and AHPIOS focused on in-memory storage as an alternative for solving I/O bottlenecks in parallel I/O accesses. Parrot [15] is a middleware focused on adapting existing I/O interfaces, such as POSIX, to new distributed systems and services. Parrot is usually deployed in combination with Chirp, a lightweight user-level I/O protocol and filesystem for collaboration in wide distrib-

uted environments such as clusters, clouds, and grids. Our proposed solution shares with this combination of solutions the user-level approach for easy deployment, the compatibility with legacy software through commonly used interfaces like POSIX, but diverges in the focus of Hercules for scalability.

AHPIOS (Ad Hoc Parallel I/O system for MPI applications) [6] is a fully scalable system for I/O parallel MPI applications. AHPIOS relies on dynamic partitions and elastic demand partitions for distributed deployment applications. AHPIOS provides different memory caches levels. Most of the AHPIOS features are shared with Hercules: (i) easy-deployment user-level components avoid the necessity of any special privileges in the used nodes, transparency using widely and easily deployable simple commands; (ii) scalability and high performance are achieved by the use of as many nodes as possible as I/O nodes; (iii) the use of main memory as high-performance storage for temporary data.

Costa et al. [1,2] propose in their MosaStore solution the use of file attributes as hints for communicating information about data access patterns between the workflow engine and file system. This information can be directly communicated by the engine to the file system, or the file system can infer patterns by analyzing the data accesses. The centralized metadata server used by MosaStore to provide this functionality can be a bottleneck in large-scale systems, in contrast with our fully distributed data and metadata approach.

The AMFS framework [18] shares with our solution the focus on providing a simple scripting language for scripting execution of parallel applications with in-memory accesses. AMFS shares several characteristics with Hercules: (i) I/O and metadata servers are managed by the same nodes and fully distributed among them; (ii) data and metadata accesses are focused on single-write, multi-read patterns; (iii) targets data locality by running compute tasks co-located with I/O nodes. The integration of Hercules with DMCF provides simpler options for developing complex workflows (both graphical and scripting definitions) and an easier framework for execution in cloud environments.

HyCache+ [19] is a distributed storage middleware that allows effective use the network bandwidth of the high-end massively parallel systems. This solution acts as a main storage for recently accessed and asynchronously flush data with the remote file system when needed. Both HyCache+ and Hercules share the fully distributed metadata approach, the lack of necessity of a dedicated I/O network and the utilization of the high-performance compute network available in compute nodes, and the high scalability focus. Hercules offers more flexibility in the use of interfaces, offering the possibility of using *get/set* and MPI-IO in addition to the default POSIX interface provided by HyCache+. HyCache targets the improvement of performance of existing parallel file systems, while Hercules is designed to accelerate workflow execution engines, and opens the possibility of facilitating the exploitation of data locality for temporary data in data-intensive applications.

Confuga [3] is an active storage cluster file system designed for executing DAG-structured workflows. This solution targets scalability of data-intensive scientific applications in POSIX environments. Confuga shares with Hercules its easy deployment base on user-level components and the target of co-locating data and execution in a distributed environment. The differences reside in: (i) the tight coupling of the

Confuga scheduler responsible of job scheduling and the storage. Our proposed solution is the combination of DMCF and Hercules, which can work independently in any case; (ii) our approach targets cloud environments instead of clusters, and (iii) Confuga used a centralized “head node” as opposed to the Hercules architecture.

Other studies have focused on the study of performance of storage services provided on clouds environments. Zhao et al. [20] compares the I/O performance of S3FS, HDFS, and FusionFS [21], while our focus is on the less studied performance of Azure Storage. As demonstrated in the experimental evaluation conducted in this paper, the performance obtained by Hercules equals or exceeds Azure Storage.

Other popular solutions, such as Spark [17] or Tachyon [9], have recently shown two fundamental basis: first, the importance of in-memory storage and data locality for improving performance in data-intensive applications, and second, the necessity of taking advantage of the new high-speed network technologies in I/O operations. However, both technologies target different objectives and different environments than our proposed solution, the enumeration of these distinctions is out of the scope of this paper.

8 Conclusions

In this work, we have presented the integration of the Hercules in-memory storage system and the Data Mining Cloud Framework in order to design and evaluate an ad hoc highly scalable in-memory storage system for temporary data produced in data analysis workflow applications.

We evaluated the performance of Hercules for the management of temporary data compared with Microsoft Azure storage using synthetic benchmarks to demonstrate the effectiveness of the solution. Then, we evaluated the integration of Hercules and DMCF on a real application consisting of a workflow with access to temporary data using either Azure storage or Hercules. The I/O overhead in this scenario using Hercules has been reduced by 36 % with respect to Azure storage, leading to a 13 % reduction of the total execution time. This confirms that our in-memory approach is effective in improving the performance of data-intensive workflow executions in cloud platforms.

As future work, we will continue the integration of DMCF and Hercules by leveraging the co-location of compute workers and I/O nodes for exposing and exploiting data locality. We plan to elaborate a detailed cost-performance analysis of our proposed solution in contrast with the default Azure Storage.

Acknowledgments This work is partially supported by EU under the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS). This work is partially supported by the grant TIN2013-41350-P, *Scalable Data Management Techniques for High-End Computing Systems* from the Spanish Ministry of Economy and Competitiveness.

References

1. Al-Kiswany S, Gharaibeh A, Ripeanu M (2010) The case for a versatile storage system. *Oper Syst Rev* 44(1):10–14

2. Costa LB, Yang H, Vairavanathan E, Barros A, Maheshwari K, Fedak G, Katz D, Wilde M, Ripeanu M, Al-Kiswany S (2014) The case for workflow-aware storage: an opportunity study. *J Grid Comput* 1–19
3. Donnelly P, Hazekamp N, Thain D (2015) Confuga: scalable data intensive computing for POSIX Workflows. In: *IEEE/ACM international symposium on cluster, cloud and grid computing*
4. Duro FR, Blas JG, Carretero J (2013) 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*, New York. ACM, pp 139–140
5. Fitzpatrick B (2004) Distributed caching with memcached. *Linux J* 2004(124):5
6. Florin I, Javier GBF, Jesús C, Wei-Keng L, Alok C (2010) A scalable message passing interface implementation of an ad-hoc parallel I/O system. *Int J High Perform Comput Appl* 24(2):164–184
7. John GH, Langley P (1995) Estimating continuous distributions in bayesian classifiers. In: *Eleventh conference on uncertainty in artificial intelligence, San Mateo. Morgan Kaufmann*, pp 338–345
8. Keerthi SS, Shevade SK, Bhattacharyya C, Murthy KRK (2001) Improvements to Platt's SVM algorithm for SVM classifier design. *Neural Comput* 13(3):637–649
9. Li H, Ghodsi A, Zaharia M, Shenker S, Stoica I (2014) Reliable, memory speed storage for cluster computing frameworks. Technical Report UCB/EECS-2014-135, EECS Department, University of California, Berkeley, Jun
10. Marozzo F, Talia D, Trunfio P (2011) A cloud framework for parameter sweeping data mining applications. In: *Proc. of the 3rd IEEE international conference on cloud computing technology and science (CloudCom 2011)*, Athens, Greece, 1 December. IEEE Computer Society Press. ISBN 978-0-7695-4622-3, pp 367–374
11. Marozzo F, Talia D, Trunfio P (2013) A cloud framework for big data analytics workflows on azure. In: *Charlie C, Wolfgang G, Lucio G, Gerhard J, Jos Luis V-P (eds) Post-Proc. of the high performance computing workshop 2012, volume 23 of advances in parallel computing*, Cetraro, Italy, IOS Press. ISBN 978-1-61499-321-6, pp 182–191
12. Marozzo F, Talia D, Trunfio P (2015) JS4Cloud: script-based workflow programming for scalable data analysis on cloud platforms. *Concurr Comput Pract Exp* 27(17):5214–5237
13. Ross Quinlan J (1993) *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA
14. Duro FR, Marozzo F, García BJ, Pérez JC, Talia D, Trunfio P (2015) Evaluating data caching techniques in DMCF workflows using Hercules. In: *Proceedings of the second international workshop on sustainable ultrascale computing systems (NESUS 2015)*, Krakow, Poland, pp 95–106
15. Thain D, Livny M (2005) Parrot: Transparent user-level middleware for data-intensive computing. *Scalable Comput Pract Exp* 6(3):9–18
16. Xindong W, Vipin Kumar J, Quinlan R, Ghosh J, Yang Q, Motoda H, McLachlan GJ, Ng A, Liu B, Yu PS, Zhou Z-H, Steinbach M, Hand DJ, Steinberg D (2007) Top 10 algorithms in data mining. *Knowl Inf Syst* 14(1):1–37
17. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I (2012) Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: *Proceedings of the 9th USENIX conference on networked systems design and implementation, NSDI'12*, Berkeley, CA. USENIX Association, pp 2–2
18. Zhang Z, Katz DS, Armstrong TG, Wozniak JM, Foster I (2013) Parallelizing the execution of sequential scripts. In: *Proceedings of the international conference on high performance computing, networking, storage and analysis, SC '13*, New York. ACM, pp 31:1–31:12
19. Zhao D, Qiao K, Raicu I (2014) Hycache+: Towards scalable high-performance caching middleware for parallel file systems. In: *IEEE/ACM CCGrid*
20. Zhao D, Yang X, Sadooghi I, Garzoglio G, Timm S, Raicu I (2015) High-performance storage support for scientific applications on the cloud. In: *Proceedings of the 6th workshop on scientific cloud computing, ScienceCloud '15*. ACM, New York, pp 33–36
21. Zhao D, Zhang Z, Zhou X, Li T, Wang K, Kimpe D, Carns P, Ross R, Raicu I (2014) FusionFS: toward supporting data-intensive scientific applications on extreme-scale high performance computing systems. In: *2014 IEEE international conference on big data (Big Data)*, pp 61–70