Parallel Computing 61 (2017) 52–67

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Experimental evaluation of a flexible I/O architecture for accelerating workflow engines in ultrascale environments

Francisco Rodrigo Duro[a,*], Javier Garcia Blas[a], Florin Isaila[a], Jesus Carretero[a], Justin M. Wozniak[b], Rob Ross[b]

[a] Computer Architecture and Communication Area, University Carlos III, Madrid, Spain
[b] Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA

ARTICLE INFO

Article history:
Received 5 February 2016
Revised 30 September 2016
Accepted 5 October 2016
Available online 6 October 2016

MSC:
68M20
68M14

Keywords:
Workflow
I/O acceleration
High-performance computing
Cloud computing

ABSTRACT

The increasing volume of scientific data and the limited scalability and performance of storage systems are currently presenting a significant limitation for the productivity of the scientific workflows running on both high-performance computing (HPC) and cloud platforms. Clearly needed is better integration of storage systems and workflow engines to address this problem. This paper presents and evaluates a novel solution that leverages codesign principles for integrating Hercules—an in-memory data store—with a workflow management system. We consider four main aspects: workflow representation, task scheduling, task placement, and task termination. The experimental evaluation on both cloud and HPC systems demonstrates significant performance and scalability improvements over existing state-of-the-art approaches.

1. Introduction

The high-performance computing (HPC) and data analysis paradigms have evolved as separate fields, with their own methodologies, tools, and techniques. The tools and cultures of HPC and big data analytics have diverged, to the detriment of both [1]. On the one hand, HPC focuses on the data generated by scientific applications. In this scenario, data is stored in high-performance parallel file systems (such as Lustre [2] and GPFS [3]) for future processing and verification. On the other hand, the analysis of large datasets greatly benefits from infrastructures where storage and computation resources are not completely decoupled, such as data centers and clouds using HDFS [4].

With the increasing availability of data generated by high-fidelity simulations and high-resolution scientific instruments in domains as diverse as climate [5], experimental physics [6], bioinformatics [7], and astronomy [8], many synergies between extreme-scale computing and data analytics are arising [9,10]. There is a need to recognize the close relationship between HPC and data analysis in the scientific computing area, and advances in both are necessary for next-generation scientific breakthroughs. Moreover, in order to achieve the desired unification, the solutions adopted should be portable and extensible to future ultra-scale systems. These systems are envisioned as parallel and distributed computing systems, reaching two to three orders of magnitude larger than today's systems [11].

* Corresponding author.
E-mail addresses: frodrigo@inf.uc3m.es, frodrigo@arcos.inf.uc3m.es (F. Rodrigo Duro), fjblas@inf.uc3m.es (J. Garcia Blas), fisaila@inf.uc3m.es (F. Isaila), jcarrete@inf.uc3m.es (J. Carretero), wozniak@mcs.anl.gov (J.M. Wozniak), rross@mcs.anl.gov (R. Ross).

http://dx.doi.org/10.1016/j.parco.2016.10.003
0167-8191/

From the point of view of application developers the convergence between HPC and big data analytics requires the integration of currently dedicated techniques into scalable workflows, with the final goal of achieving the full automatization of complex task ensembles. One of the main challenge in achieving this goal is the lack of scalability of solutions combining workflow engines and data stores. The principal hurdle is represented by the fact that existing storage systems have not been designed for the requirements of scalable workflows.

A further challenge is posed by the divergence of the software stacks in the HPC and big data analytics ecosystems. New approaches are required for bringing together the best from each domain into various types of infrastructures including HPC platforms, clouds, and data centers.

This paper presents and evaluates a novel workflow-aware and platform-agnostic data management solution for data-intensive workflows. The main contributions of this work are as follows:

- We provide a novel workflow-aware data store solution that leverages codesign principles for integrating Hercules—an in-memory data store—with a workflow management system.
- Our solution provides a common ground for accelerating the I/O of workflows on both cloud and HPC systems.
- We demonstrate that the codesign of task scheduling in a workflow engine with data locality exploitation mechanisms in Hercules can open up a range of novel mechanisms that can be used for I/O acceleration.
- Extensive evaluation on both cloud and HPC systems demonstrates the superiority of our solution over existing state-of-the-art approaches.

The remainder of this paper is organized as follows. Section 2 presents the background for this work. Section 3 descibes the design and implementation of our solution. Section 4 discusses the results of our experimental evaluation. Section 5 presents related work. Section 6 summarizes our conclusions.

2. Background

Workflow management systems are becoming increasingly important for scientific computing as a way of providing high-throughput and analysis capabilities that are essential when processing large volumes of data generated at high speed. A workflow consists of many (ranging from tens to millions) interdependent tasks that communicate through intermediate storage abstractions, typically files. Workflow engines such as Swift/T [12], DMCF [13], or Pegasus [14] permit the execution of data-intensive scientific applications on different types of platforms including clouds, HPC clusters or supercomputers, and grids.

In many cases, however, the performance of workflow engines is limited by the I/O performance and scalability of the underlying storage systems. Typical HPC infrastructures use monolithic parallel file systems such as GPFS and Lustre. An alternative approach for providing portability and scalable I/O in cloud platforms is the use of remote shared storage systems. Usually, the aim of this approach is to provide a unified interface and a scalable storage solution for cloud-based applications through storage services such as Amazon S3. Currently, in addition to web services, Amazon offers access to S3 though S3FS [15], a FUSE-based file system. Yet, the storage systems from both HPC and cloud domains do not efficiently support data-intensive workflows, especially because their design was conceived for individual applications and not for ensembles of cooperating applications.

Recently, considerable research has been devoted to distributed key-value stores, which are not fundamentally different from tuple spaces but tend to emphasize simple data-storage, rather than data-driven coordination, and support simpler query methods, such as exact key lookups or range queries. These stores can be used to accelerate scientific workflows. As we describe later in the paper, however, a high-performance solution requires careful codesign with the workflow framework, in order to use the specific workflow patterns efficiently. In the remainder of this section we briefly examine two key-value stores that have been used in our solution and evaluation.

Memcached [16] is a distributed shared-memory system. It offers the available RAM on multiple servers as a single memory space. Memcached stores binary objects leveraging a distributed hash table. The replacement policy is least recently used (LRU), and no mechanism is available to write the data dropped from cache to storage devices. Thus, if an item expires or is discarded by LRU policy, it cannot be recovered. By default, data is not replicated, and Memcached has no mechanisms for managing failures. Currently, however, consistent hashing algorithms do exist, such as libketama [17] and vBuckets [18], which offer item relocation when the number of servers is modified.

Redis [19] is an open source in-memory key-value store that promises high performance and flexibility. Redis stores objects as data structures, providing access to mutable data structures. Compared with Memcached, Redis has several advantages. First, it allows persistent storage (e.g., disk). Two persistence techniques are available by default: point-in-time snapshots of a dataset at specified intervals and a log-structure approach for write operations. Second, Redis applies memory reduction techniques. Third, it offers features such as replication and high-availability clustering.

Either Memcached or Redis can provide a better option for maximizing performance depending on the use case.

3. Hercules: a workflow-oriented store

Hercules is a key-value distributed in-memory store based on existing distributed shared-memory back-end solutions, such as Memcached or Redis.

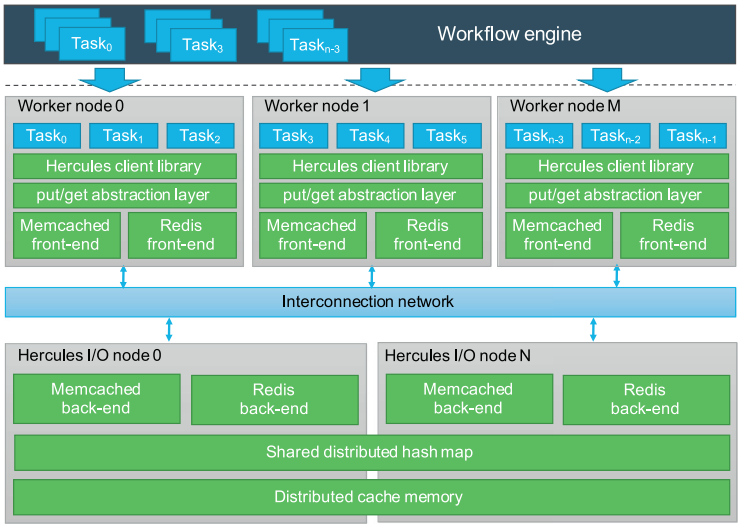

Fig. 1. Hercules layered software architecture. Worker nodes and I/O nodes may execute on the same physical node in practice as depicted in Figure reffig:deployment (b).

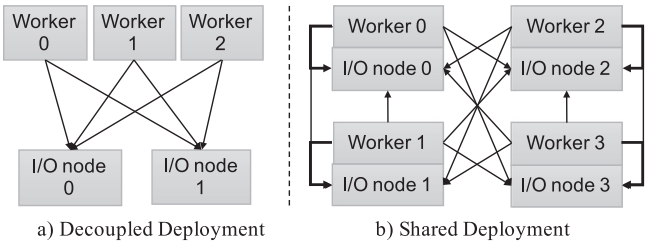

Fig. 2. Potential deployments in Hercules. Strong lines represent data transfers inside nodes.

Hercules [20] was initially designed for large-scale HPC systems as a stackable in-memory store based on Memcached servers with persistence support. In this section we focus on novel features that significantly extend the previous design: symbiosis with workflow systems, support for both cloud and HPC systems, locality-aware data placement, and support for various key-value stores.

3.1. Architecture

The Hercules architecture is shown in Fig. 1. A workflow engine schedules tasks on worker nodes and provides information to Hercules about tasks and data, as discussed below.

The client library leverages a NoSQL database client library for accessing data and metadata from Hercules I/O nodes. The client library offers a file abstraction on top of a put/get interface and manages both data and metadata.

A Hercules file is mapped to a set of blocks that can be distributed to the servers of an abstract put/get store. By default the block distribution of a file relies on the custom hashing of the concrete put/get store. In a new scheme file data may be distributed based on a location value placed on metadata. This approach allows one to control the data placement in order to improve the load balance or move data to the code. The file metadata is distributed over the servers by consistently hashing a key and contain information about data placement. Both data and metadata are stored at the servers of the key-value store. Hercules currently supports a flat namespace. This approach is not likely to cause a bottleneck, given that Hercules is designed to be used as an intermediary store for one application (workflow).

Depending on the deployment, the I/O nodes can be remote or local. Fig. 2 shows two potential deployments. In Fig. 2(a) worker nodes and I/O nodes do not share the same physical node, and all accesses result in a network transfer. In Fig. 2(b) when worker nodes and I/O nodes share physical nodes, locality-aware task placement can avoid the need for network transfer.

Since the initial implementation of Hercules, we have added the ability to leverage multiple back-end data stores. In this paper we compare Hercules atop Memcached and Redis (Section 4.2).

Workflow tasks access data through the Hercules client library. Hercules offers three APIs layered on top of each other as shown in Fig. 3: put/get, POSIX-like, and MPI-IO. The put/get API offers transparent access to items stored on remote servers. It is similar to established APIs, such as the one of Memcached.

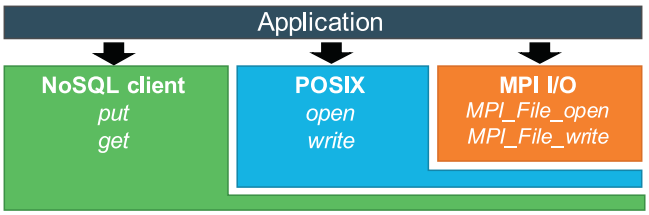

Fig. 3. Three APIs in Hercules layered on top of each other.

The POSIX-like interface offers POSIX syntax (open, read, write, lseek, close) but with different semantics. In single-process operations it is POSIX-compliant (partial read/writes, random accesses, etc.); however, in concurrent operations performed over the same file it does not support Unix consistency. However, this interface does efficiently support common workflow operations such as sequentially writing a whole file and sequentially reading a whole file, and it supports futures based on external information from workflow engines about the termination of a task. A *future* [21] is a construct used for synchronization in some concurrent programming languages. In our solution this concept has been adapted for synchronization to file accesses. A read of a partially or fully non-existing file region will block until the whole region is written. This semantics is not supported by POSIX, despite the fact that it is the most common requirement of workflows communicating through files. Our implementation of futures leverages mechanisms from both Hercules and a workflow management system as discussed in the next section.

The MPI-IO interface is implemented on top of the POSIX-like interface. It allows parallel applications to share the parallel I/O access to a file distributed over Hercules servers. The evaluations in this paper are based on the POSIX-like API.

3.2. Workflow awareness

Our new design is intended to make Hercules work in symbiosis with a workflow framework. Thus, our approach assumes that an external workflow framework provides (1) a directed acyclic graph (DAG) representing tasks and their dependencies; (2) a scheduler of workload tasks; (3) enforcement of and information about task placement; and (4) information about individual task termination.

In this section we discuss how Hercules can be used in both HPC and cloud environments by providing the four requirements listed above.

3.2.1. Swift/T

In HPC environments Hercules has been integrated with Swift/T [22], which is both a workflow definition language and a runtime system. Swift/T provides all the four ingredients listed above.

First, Swift/T offers a programming language that can be used for defining a DAG representing tasks and their dependencies. Each task either can be written in Swift language or can be an external task written in a different programming (e.g., Java) or scripting language (e.g., Python).

Second, each Swift/T task is scheduled through a runtime on workers running on nodes of a cluster or supercomputer. The default scheduling policy targets optimal load balance and is agnostic to data locality. By leveraging mechanisms for task placement awareness and task placement enforcement (as discussed below), the symbiosis between Swift/T and Hercules provides novel scheduling policies that allow the tasks to be moved to the data or the data to the tasks.

Third, Swift/T provides information about the task placement: the whole Swift program runs as an MPI executable, and each worker runs in a process uniquely identified through an MPI rank. This rank can be used for identifying the exact location of the task in the cluster. This mechanism allows for the implementation of a policy that moves the data to the node where the code runs, provided that Hercules offers mechanisms for data movement, as described in the preceding section. Swift/T provides mechanisms for task placement: a task can be placed on a given node (NODE) or a given rank (RANK). The task placement is either enforced (HARD) or advisory (SOFT), resulting in a combination of four policies. This mechanism can be used for various policies that move the code where the data is, provided that Hercules returns the information about data placement, as discussed above.

Fourth, task termination is notified through a pub/sub mechanism that is used in combination with Hercules for implementing futures. A Swift/T data-dependent task whose data is not available blocks, waiting for the data. The task can be placed based on a task placement policy described above, while the data is made available locally or remotely by Hercules. Whenever the producer of the file closes the file, the data is moved to the dependent task (if necessary, i.e. if the dependent task is not running locally), and notifies it about the availability.

3.2.2. Cloud workflow emulation

In cloud environments we have integrated Hercules with an emulation of a workflow manager that addresses the four requirements enumerated above in the following way. First, each task is implemented in C, and the workflow is simply represented as a list in a text file. Second, the scheduling of tasks is done in a shell script that schedules available tasks over

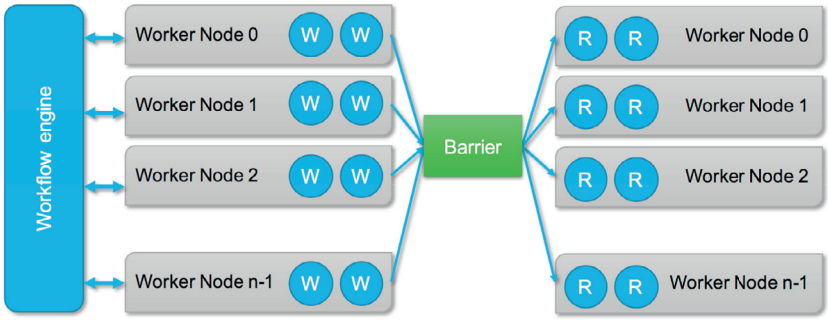

Fig. 4. Filecopy benchmark for cloud platforms. Scheme of the workflow benchmark developed for the evaluation of Hercules.

the available nodes through a remote terminal (ssh). The available nodes are obtained through virtual machine provisioning over a cloud API (e.g., Amazon). Third, the information about task placement is internally kept by the shell script and can be provided on demand over the available nodes. Fourth, task termination is detected by waiting for termination of tasks. The termination is followed by notification of the dependent tasks.

By default each task of the workflow reads the writes and reads input, intermediary data, and output data from a shared file system (such as S3FS in Amazon). Hercules is interposed as a store for intermediary data.

4. Experimental evaluation

In this section we present an experimental evaluation of the feasibility of the proposed Hercules architecture on both cloud and HPC platforms. For both platforms we perform an in-depth performance analysis based on a filecopy workflow. The tasks of the benchmark are exactly the same but are deployed with a different workflow engine depending on the platform. Additionally, we evaluate the data locality exploitation in an HPC environment based on a MapReduce workflow.

4.1. Filecopy benchmark

To evaluate Hercules, we have used a filecopy benchmark that performs a data-parallel file copy. In the experiments performed on cloud infrastructures, we emulate a workflow runtime by deploying concurrent tasks among all the worker nodes. The emulated access pattern aims to mimic those patterns of real data-intensive workflows where tasks consume data produced by other previous tasks. Fig. 4 shows the different stages of the evaluated workflow. The workflow runtime generates multiple writing (W) and reading (R) tasks. The evaluation consists of generating n tasks that are concurrently launched on as many worker nodes as desired. One also can deploy more than one task per worker node in order to take advantage of multicore resources. Each task writes a file, and then the same number of tasks read the files previously created. Both writing and reading tasks access sequentially to a different file (one per task). On cloud infrastructures, reading tasks are not launched until every writing operation is completed.

The HPC version of the filecopy benchmark uses Swift as the language for describing the workflow and Swift/T as the framework for executing the workflow tasks. Workflow external applications `write_task` and `read_task` remain written in C. Data dependencies are solved per task, launching the read operation immediately after each file is completely written (if there are resources available for this task), avoiding the barrier depicted in Fig. 4.

The filecopy benchmark is fully configurable by defining several parameters, such as number of I/O servers, number of computing nodes performing tasks (worker nodes), number of tasks running on each worker node, total number of tasks, total file size, and the size of each input and output operation. The extensive configuration options of this evaluation will be useful for studying the performance of our solution compared with the storage systems on the cloud and HPC platforms for different I/O scenarios.

4.2. Evaluation on cloud platforms

To evaluate Hercules on cloud platforms, we deployed it on m3.xlarge and m3.2xlarge virtual instances of Amazon EC2. We ran the filecopy benchmark on different configurations consisting of a maximum of 64 worker nodes (m3.xlarge instances with 15 GB RAM and 4 vCPUs) and 32 I/O server nodes (m3.2xlarge instances with 30 GB RAM and 8 vCPUs). The network was partitioned for both instance types. The network throughput measured with *iperf* was around 1 Gbps. All experiments were executed using S3FS and our solution with different configurations. In all cases, we show the average results of five consecutive iterations.

For deploying our solution, we created a custom image (AMI) based on Ubuntu Server LTS 14.04 containing all the necessary libraries: hiredis 0.13.1, libmemcached 1.0.18, s3fs 1.74, and ec2-api-tools 1.7.4.0. Memcached 1.4.24 and Redis 3.0.1

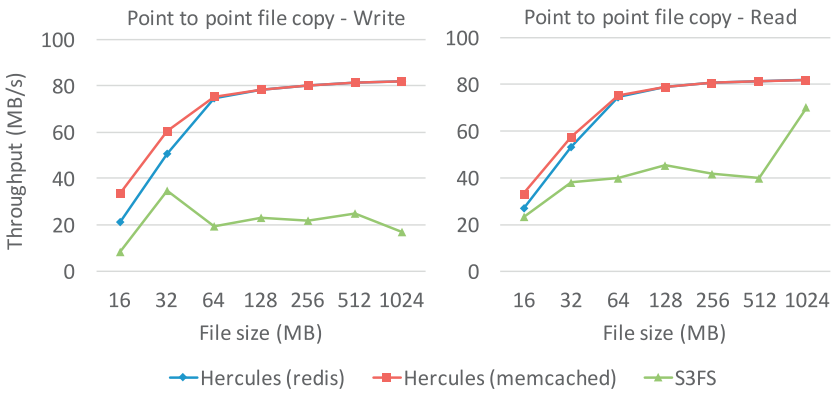

Fig. 5. Performance evolution for various file sizes, one worker node, and one I/O node.

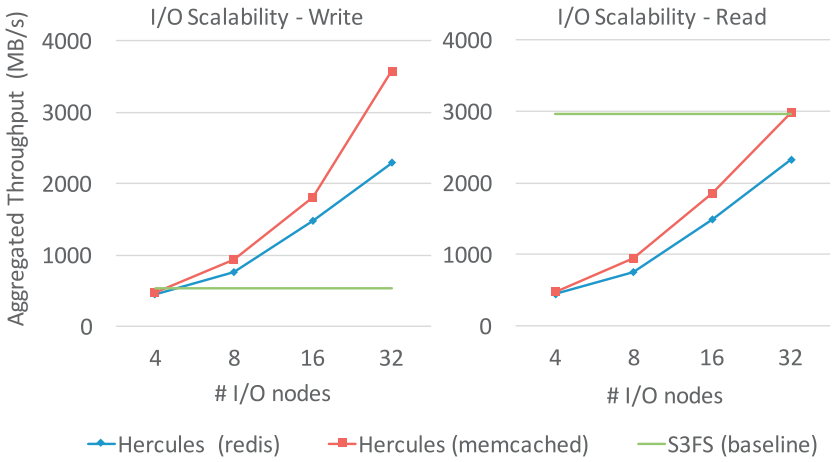

Fig. 6. I/O scalability with the number of I/O nodes. 32 worker nodes perform write and read operations. Each worker writes/read a 2,048 MB file.

were used as Hercules back-ends, and both were launched on each evaluation using 80% of the available total memory in each I/O node, without any persistence enabled and using pure LRU as replacement policy.

4.2.1. Point-to-point file copy performance

In the first experiment, we set up a basic configuration that uses a single worker node, a single I/O node, and different file sizes ranging from 16 to 1024 MB. The goal of this evaluation was to study the behavior of each solution depending on both the size of the file accessed and the operation executed (read or write).

As shown in Fig. 5, the performance of Hercules (for both Memcached and Redis) is limited by the virtual Gigabit Ethernet interface (1 Gbps) even for relatively small file sizes (64 MB). S3FS performance remains at about 20 MB/s in all the write cases and grows smoothly for read operations, reaching a performance peak when the file size surpasses 1 GB. We conclude from this experiment that Hercules, for both Redis and Memcached versions, takes full advantage of the available network capacity and outperforms S3FS for small files. For large files, S3FS approaches the performance of Hercules only in the case of read accesses.

4.2.2. I/O scalability

After evaluating the behavior of Hercules with a single I/O node available, we then focused on I/O scalability as a function of the number of I/O nodes. For this purpose, the number of Hercules I/O node servers is scaled for the same workload: 32 worker nodes running a writing task of 2048 MB per node.

The results in Fig. 6 indicate that the bottleneck of our solution is the available network bandwidth. With 4 I/O nodes the maximum theoretical bandwidth available is proportional to the number of Gigabit Ethernet connections (one on each I/O node). Both the write performance and read performance of Hercules are around 500 MB/s for 4 I/O nodes and scale with the number of I/O nodes. Moreover, unlike the case of S3FS, the performance of writes and reads is similar. We also note that Memcached throughput evolves better than Redis when the number of available I/O nodes increases, demonstrating better scalability. S3FS performance remains constant. Given that the configuration of S3FS is not exposed to the user, we assume that this is due to the fact that S3FS uses a fixed number of I/O servers.

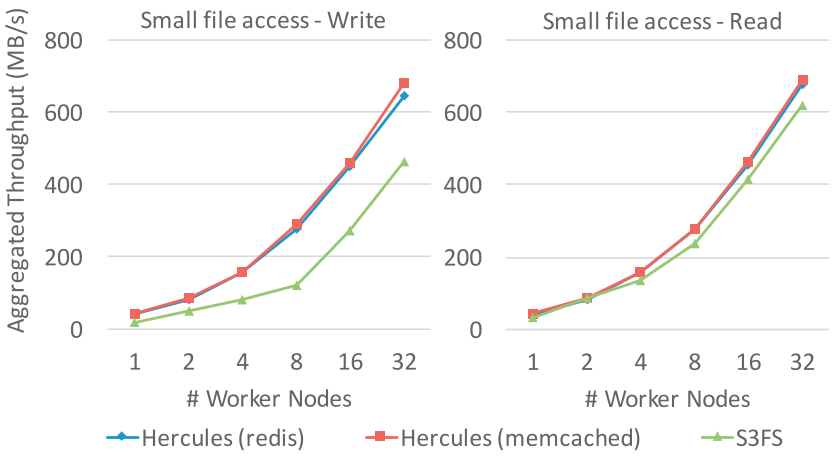

Fig. 7. Performance for small file access. 1 to 32 worker nodes, running 8 concurrent processes each, perform write and read operations. Each task accesses a 16 MB file. Hercules uses 32 I/O nodes for both Memcached and Redis.

Compared with the performance of S3FS, the write performance of Hercules is much higher in all cases, whereas read operations compete only in the case of 32 I/O server nodes. This situation arises because big files are the best case for S3FS. S3FS is able to always parallelize accesses over the 32 nodes performing 32 parallel tasks. In turn, Hercules can serve 32 workers with the same performance only when 32 I/O nodes are available to meet the demands of compute nodes at maximum capacity. In addition, as pointed out previously, we are not aware of how many I/O nodes S3FS uses to serve requests. However, this experiment demonstrates that our solution is scalable and efficiently uses the available network bandwidth. We expect even better performance of our Hercules-based solution with a dedicated high-throughput low-latency network.

4.2.3. Performance of small file access

In our next experiment, we evaluated Hercules for writing and reading small files. This type of access is common in many-task computing applications. We set the number of I/O nodes to 32 for both Memcached and Redis. The number of worker nodes varied from 1 to 32. Every worker node performed exactly the same tasks: 8 concurrent tasks wrote a file of 16 MB each, and 8 tasks read it after write completion.

Fig. 7 shows that the performance of Hercules scales with the number of compute nodes working in parallel and is always superior to the S3FS performance. No significant differences are seen between using Memcached or Redis as the back-end.

4.2.4. Performance of large file access

The objective of our next evaluation was to understand the limitations of our system compared with S3FS by studying an unfavorable case: tasks accessing large files. The configuration consisted of 16 I/O nodes running Memcached or Redis. The number of worker nodes ranged from 4 to 64, and each one ran a single task. All tasks concurrently accessed the shared storage systems. Each task read/wrote a 2048 MB file.

As shown in Fig. 8 (left), as the number of worker nodes concurrently accessing Hercules increases, the performance scales and reaches a peak at 16 nodes, close to the maximum theoretical throughput of 2 GB/s provided by the 16 Gigabit Ethernet connections. Memcached behaves slightly better than Redis, especially for write operations.

We conclude from this evaluation that Hercules behaves better than S3FS in the case of write operations and in the majority of read scenarios (as shown in Fig. 8 (right)). Furthermore, the main advantage of Hercules compared with S3FS is that its performance is stable and predictable, since it is isolated from other users during peak I/O loads.

4.3. Evaluation on HPC clusters

In addition to the cloud evaluation, we evaluated our solution on an HPC environment, namely, the Fusion cluster at Argonne National Laboratory. This cluster has 320 nodes, composed of dual-socket boards with quad-core 2.53 GHz processors and 36 GB of main memory. There are two interconnect networks: InfiniBand QDR with 4 Gb/s per link and Ethernet Gigabit. GPFS is configured with native InfiniBand support, using 4 I/O nodes, with a page pool of 8 GB and 1 GB cache per node at client side. Fusion's GPFS theoretical peak performance is 8 GB/s. Our best recorded result is 3200 MB/s using *dd*. Hercules uses Gigabit Ethernet and TCP/IP over InfiniBand (IB), with a maximum recorded throughput of 4 Gb/s measured with *iperf*, and uses 80% of the available main memory of the computing node. In this evaluation, we focused on the evaluation of Hercules using Memcached servers. The scenarios evaluated are equivalent to those of the cloud evaluation.

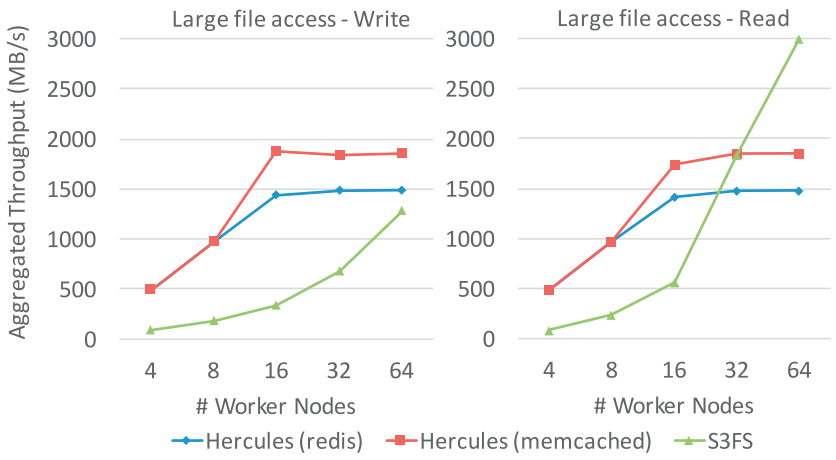

Fig. 8. Performance for large files. Hercules uses 16 I/O nodes, running Memcached or Redis. The number of worker nodes varies between 4 and 64. Each task accesses a 2048 MB file.

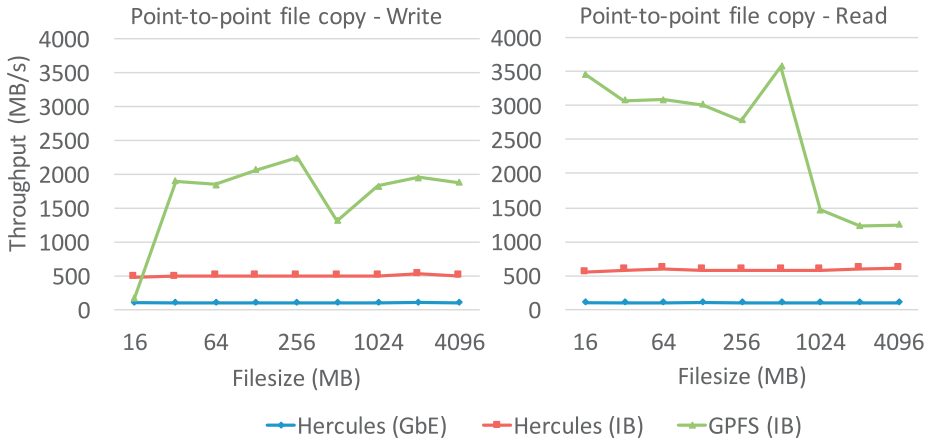

Fig. 9. Performance for various file sizes for Hercules-based and GPFS-based solutions.

4.3.1. Point-to-point file copy performance

We compared our solution with GPFS for a case with a single worker process. In this evaluation, Hercules was configured with one I/O server, while GPFS used the default configuration described above. The main difference from the equivalent cloud experiment was the maximum file size used in the experiments. Specifically, in this case we increased the file size to 4096 MB in order to study what happens when the GPFS client-side cache is filled (it is configured as 1024 MB).

Fig. 9 clearly shows how Hercules performance is limited by the available bandwidth over both Gigabit Ethernet and InfiniBand. In contrast, GPFS shows a much better performance. The reason is that the client-side cache makes possible a throughput close to the maximum theoretical performance of the DDR3 memory. When the cache is full and starts evicting elements, the read performance greatly decreases, to the 1500 MB/s mark. Even in the worst case of this scenario GPFS offers a better performance than does Hercules. The reason is that GPFS communication protocol is optimized for running over InfiniBand and that Hercules is using only one I/O node. In the following sections, we show some scenarios where our Hercules-based solution can address this issue.

4.3.2. I/O scalability

Next, we evaluated the scalability of our solution compared with GPFS. In this evaluation, 32 worker nodes accessed data through a shared data store. The number of Hercules I/O nodes was varied from 4 to 32. Each worker node wrote a dataset containing 2048 MB of data. In contrast with the cloud evaluation, two experiments were performed. In the first, only one worker process ran on each worker node (1 process per node, or ppn), writing/reading a 2048 MB file. In the second experiment, 8 worker processes (8 ppn, same as available cores) wrote/read 256 MB files. The resulting workload was similar to the first experiment: a total of 64 GB of data were written/read. Both experiments were evaluated with Hercules over Gigabit Ethernet (equivalent to the cloud scenario) and InfiniBand (best-performing network on the Fusion cluster).

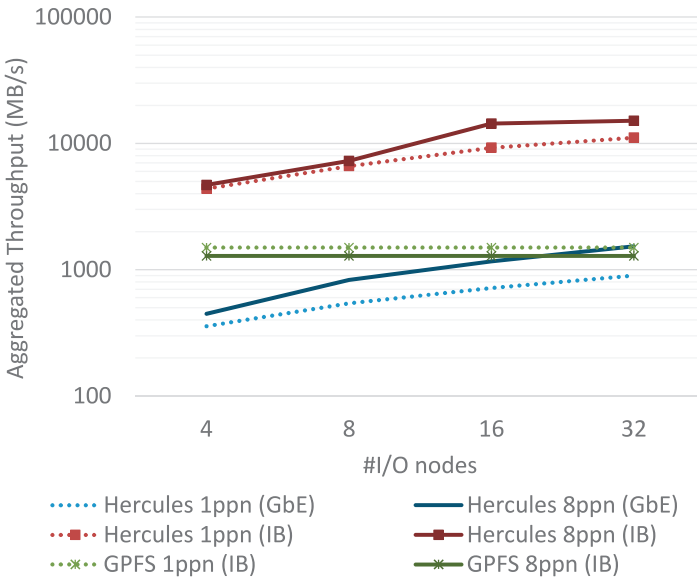

Fig. 10. Scalability evaluation depending on the number of I/O nodes. Here 32 worker nodes perform write and read operations on datasets of 2048 MB per worker node.

The behavior depicted in Fig. 10 confirms our expectations. The aggregated throughput achieved by GPFS remains almost constant, since it always uses the same number of I/O nodes. The only unexpected behavior is how GPFS performance slightly degrades when the number of workers increases, probably due to contention.

We note that the performance increases when 8 processes per node are used but that it degrades in contrast with the cloud scenario. In the cloud scenario, the experiment was designed for fully load balancing the I/O operations over all available I/O nodes. In this case, the tasks are scheduled by Swift/T, and data/metadata of the files are distributed over all nodes. This distribution can be uneven, leading to performance degradation when multiple workers are accessing the same I/O nodes while the rest are idle.

For a larger number of workers running on the same node, the load per worker becomes smaller (given that we are performing a strong-scalability evaluation). The smaller granularity makes more probable that the load is evenly distributed over the I/O nodes. This case can be improved with the locality-aware data-placement and task-placement techniques discussed in Section 3.2.

4.3.3. Performance of small file access

The objective of our next experiments was to evaluate the performance of our solution and GPFS for accesses to small files, a typical workflow data access pattern. At the same time, we wanted to measure the capabilities of both systems for dealing with increasing data accesses. GPFS used the default Fusion configuration, and Hercules was configured with 32 I/O nodes in all experiments. The number of worker nodes concurrently accessing the shared file system (Hercules or GPFS) increased from 1 to 32; and 8 worker processes run on each node, resulting in 8 to 256 workers. The workflow consisted of writing one 16 MB file per worker and then reading the files previously created. For Hercules the worker nodes and I/O nodes ran on non-overlapping compute nodes as shown in Fig. 2(b).

GPFS performance as shown in Fig. 11 does not scale with the number of worker nodes. This behavior is explained by the data-locality unawareness of both GPFS and Swift/T. Write tasks and read tasks sharing access to the same file are commonly scheduled on different worker nodes, exposing the inadequacy of GPFS client-side caching for this access pattern. For a small number of nodes a reader is likely to be scheduled on the same node as is a writer. Thus, for a small number of nodes the performance of GPFS can be boosted by the locality exploitation. However, as the number of nodes increases, this phenomenon becomes less likely and, in conjunction with the larger contention, causes the performance to drop.

For Hercules, the fact that worker nodes and I/O nodes run on different physical nodes means that there is no locality to exploit. However, the dynamic deployment of Hercules makes the performance scale with the number of nodes. For a number workers less than or equal to the number of I/O nodes, the load is well distributed over the I/O nodes, and the performance scales neatly. When the number of workers becomes larger than the number of I/O nodes (64 versus 32), the performance drops, since the contention is not compensated for by a large utilization. When the number of nodes further increases, however, the better utilization of I/O server significantly compensates for the contention effects, and the Hercules-based solution scales. Indeed, for 256 workers it outperforms the GPFS-based solution by more than 100%.

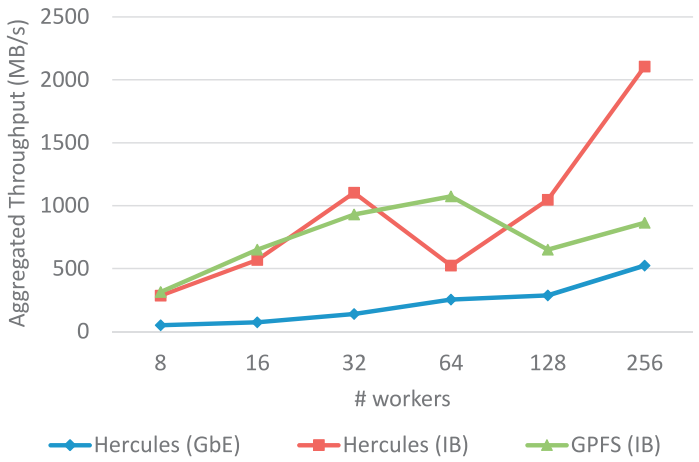

Fig. 11. Performance evaluation for small file access. Concurrent workers perform write and read operations of 16 MB files. The number of nodes are varied from 1 to 64, and there are 8 workers/node. Every Hercules deployment uses 32 I/O nodes running over Gigabit Ethernet and InfiniBand.

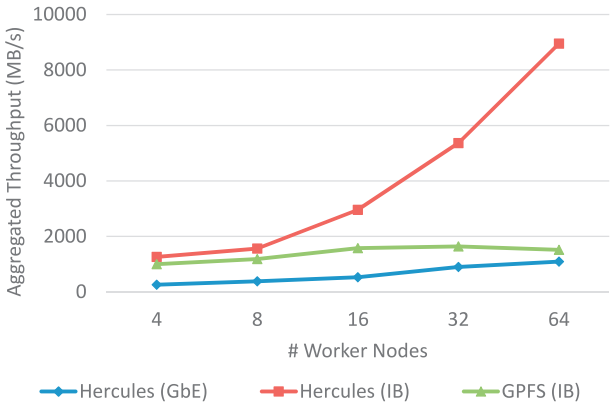

Fig. 12. Performance for large data sets. Hercules uses 16 I/O nodes, running both over Gigabit Ethernet and InfiniBand. The number of worker nodes is varied between 4 and 64. Each worker writes/reads 2048 MB to a file.

4.3.4. Performance of large file access

We next compared the performance of our solution and the GPFS-based solution when the number of worker nodes outnumbered the number of I/O nodes accessing big files. The Hercules I/O nodes were fixed to 16 for every experiment, and the number of worker nodes scaled from 4 to 64, writing and then reading one 2048 MB file per worker. Only one worker process ran on each worker node.

In this case, when the number of I/O nodes exceeds the number of worker nodes, the maximum bandwidth available is determined by the number of worker nodes multiplied by the bandwidth per node (around 100 MB/s per node using Gigabit Ethernet and around 500 MB/s per node using TCP over InfiniBand). On the other hand, when the Hercules I/O nodes outnumber the worker nodes, the maximum bandwidth available for Hercules cases are 1600 MB/s for Gigabit Ethernet and around 8000 MB/s for InfiniBand.

The results from Fig. 12 show a pattern similar to that of the other scenarios. GPFS performance is almost constant, and the available bandwidth is evenly divided among all the worker nodes concurrently accessing the shared file system, whereas our solution flexibly adapts to the load. The throughput is initially constrained by the bandwidth available at client side, but it scales with the number of worker nodes up to the maximum of the bandwidth offered by the back-end. From a worker point of view, the throughput achieved accessing GPFS is strongly penalized when the number of nodes accessing concurrently increases (I/O contention). The advantage of our solution is that the number of I/O nodes can be dynamically chosen when the application starts, whereas in a system such as GPFS the number of I/O nodes is statically provisioned.

4.3.5. Data-locality and load-balance analysis

Our next experiments were designed to evaluate how our data-locality techniques, both data placement and task placement, can improve the Swift/T and Hercules proposed solution. The evaluation is based on a MapReduce-like workflow that performs both computation and I/O. The MapReduce job, as shown in Fig. 13, is divided into three tasks. First, a *map task* reads the file from the shared file system (GPFS), tokenizes it, and writes the results to an intermediate file. Second, a *count*

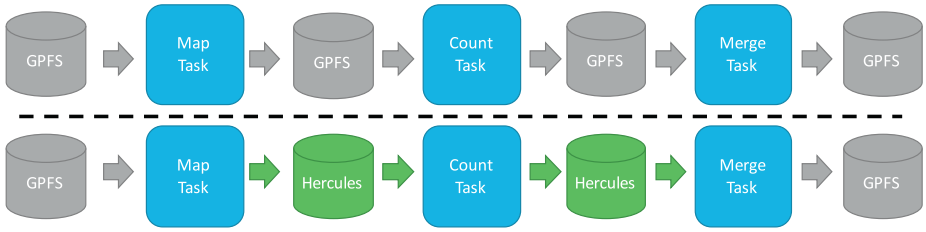

Fig. 13. MapReduce-like workflow. Source data and results are directly accessed through the persistent storage (GPFS). Top part of the figure shows the classical approach where I/O operations are performed over GPFS. In contrast, the bottom part of the figure shows our approach where temporary data are accessed through Hercules.

Table 1
Data-locality configurations.

Data placement	Task placement locality	Tag
default (DP-def)	loc. agnostic (TP-ag)	Her1
informed (DP-inf)	loc. agnostic (TP-ag)	Her2
default (DP-def)	loc. aware HARD (TP-aw)	Her3
default (DP-def)	loc. aware SOFT (TP-aw)	Her4
informed (DP-inf)	loc. aware HARD (TP-aw)	Her5
informed (DP-inf)	loc. aware SOFT (TP-aw)	Her6

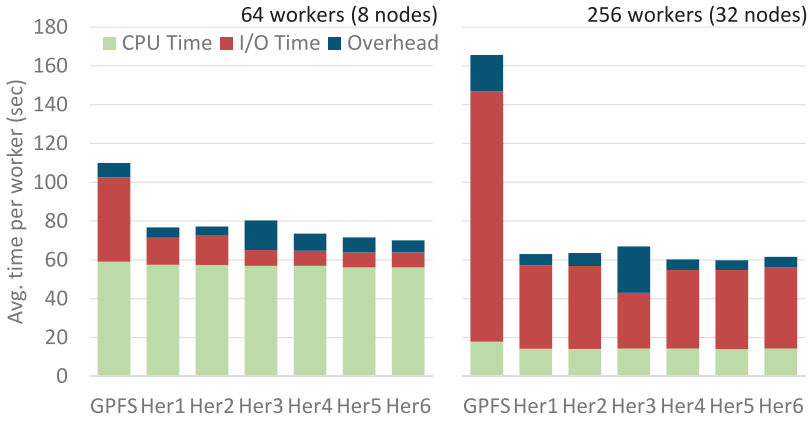

Fig. 14. Breakdown of the average CPU, I/O, and overhead times in a MapReduce-like workflow composed of 256 jobs (767 tasks) for 256 MB text files.

task reads the results of a previous task, process them, and writes the results to a new file. Third, a *merge task* combines two files resulting from the previous task. The final results are written to the shared storage system.

In the Swift/T and Hercules cases, the initial input files were read from GPFS, and the final output file was written to GPFS; any other intermediary I/O operations were performed over Hercules. This approach emulates the behavior of a real application, where input data is stored on the default shared file system and the results should be saved to the persistent default storage.

The application is fully configurable in the number of files, file size, worker nodes, and workers running on each node. For n input files, n map tasks, n count tasks, and n - *1* merge tasks were executed.

For this workflow we compared the default GPFS-based solution with different configurations of our solution detailed in Table 1 and previously described in Section 3.2.1. The default data placement is based on algorithmic hashing, whereas informed data placement is a custom data distribution in which the files are evenly distributed among all the I/O nodes. In the first case, the hash placement can cause an uneven data distribution, which can further result in an unbalanced task placement. The task scheduling policy is denoted HARD for “NODE, HARD” and SOFT for “NODE, SOFT”.

The total work consisted of processing 256 text files containing 256 MB of text data each. The selected chunk size was 32 MB, and the evaluation used 8 workers per node (one per core).

Fig. 14 shows the breakdown of the average time of each worker into compute, I/O, and overheads (e.g., scheduling, load imbalance). The left-hand side of the figure depicts the execution times using 64 workers (8 nodes), and the right-hand side shows the results for 256 workers (32 nodes).

Fig. 14 (left) shows that our solution can speed the overall execution times by up to 1.57x, in the case of informed data placement and best-effort data locality (SOFT) configurations. This time is especially substantial given that I/O operations

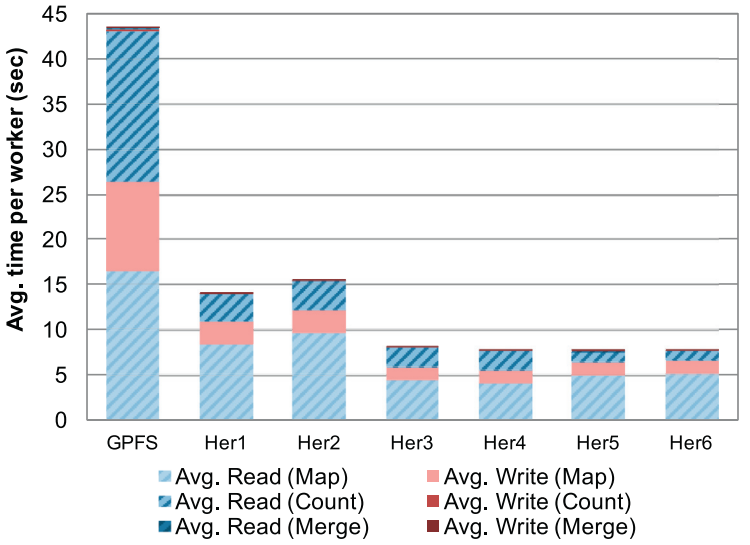

Fig. 15. I/O breakdown of the MapReduce-like workflow task times including only the I/O phases for 64 workers (8 nodes).

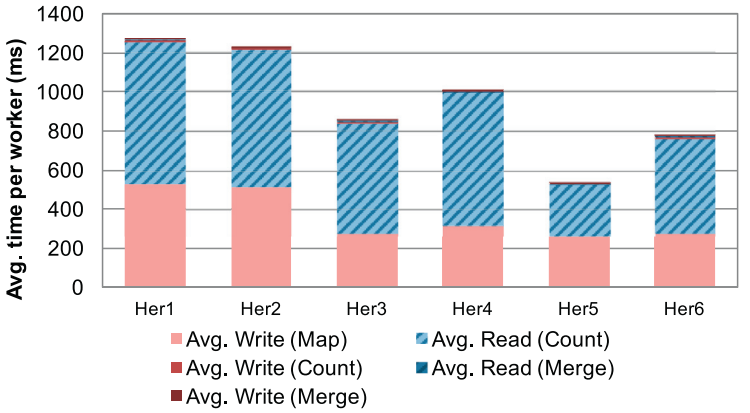

Fig. 16. Hercules I/O breakdown of the MapReduce-like workflow task times for 256 workers (32 nodes). This figure shows the I/O phases directly performed over Hercules: map-write, count-read, count-write, and merge-read.

represent less than 45% of the execution time per worker. If we focus on I/O operations, the times are reduced from 43 s to under 8 s, more than a 5x improvement. Furthermore, if we focus only on the intermediary I/O operations, the improvement is over 10x, thanks to the local data accesses.

Fig. 14 (right) depicts the strong scalability of all configurations. For the same problem size as before (processing 256 files with 256 MB of text data) we used 32 nodes (256 workers) instead of the previous 8 nodes (32 workers). The results are similar to the previous case, but the solutions based on Hercules and Swift can take advantage of the added computing resources for both compute phases and I/O operations directly performed over Hercules. In contrast, the contention on GPFS worsens, resulting in an increase in the total execution time, despite the reduction of the compute time. The improvement in total execution time achieved by Hercules is 2.77x in the best case, and the speedup of I/O operations is up to 4.48x.

The compute operations represent an important share of the execution time, and this phase is not affected by the use of Hercules instead of GPFS. The I/O operations are clearly improved over the default GPFS-based scenario, as shown in Fig. 15 detailing only the I/O phases. Even the initial read phase, where files are read from GPFS, benefits from our solution, since Hercules significantly reduces the contention in the later stages of the workflow, which can overlap with the first phase because of the combination of data and functional parallelism offered by Swift.

For the Hercules-related cases, the worst performance is obtained for the default configuration. When data locality is exploited, as shown in Fig. 16 (which details I/O operations performed over Hercules), the time needed for I/O operations decreases, especially where the data locality is strictly enforced (HARD). In contrast, for best-effort data locality (SOFT) the I/O performance slightly worsens because some tasks involve network transfers instead of fully local accesses. The best I/O performance is achieved by combining HARD data locality and informed data placement. However, the overall execution time is best for DP-inf TP-loc-aw SOFT, because this configuration trades off locality for load balance. Hence, an improvement

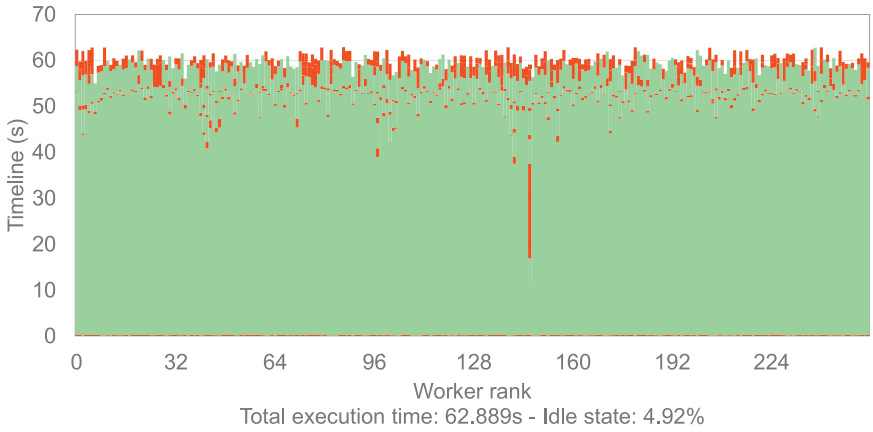

Fig. 17. Timeline in seconds for each worker, using default data placement and locality-agnostic Swift/T scheduling. Execution of the MapReduce-like application with 256 jobs (767 tasks) running on 32 nodes (256 workers). Light green represents busy status; dark red represents idle/wait status. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

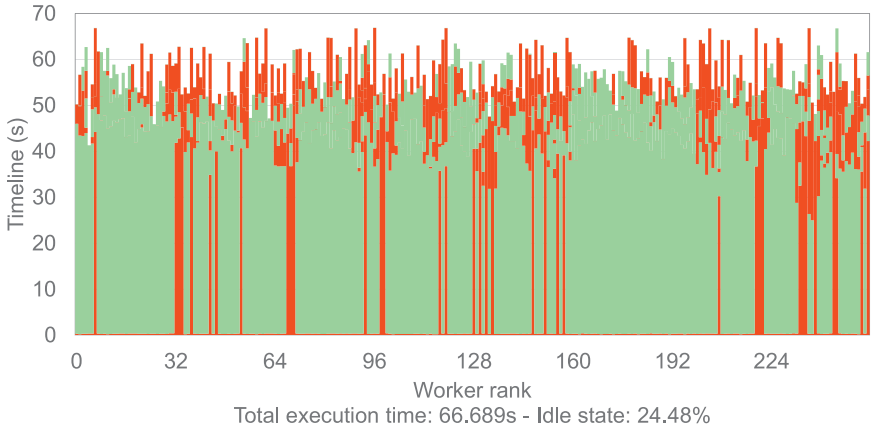

Fig. 18. Timeline in seconds for each worker, using default data placement and locality-aware scheduling (HARD, NODE). Execution of the MapReduce-like application with 256 jobs (767 tasks) running on 32 nodes (256 workers). Light green represents busy status; dark red represents idle/wait status. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

in the I/O performance does not necessary translate into an improvement in the total workflow time. This is caused mainly by the classical trade-off between data locality and load balance; that is, improving data locality may hurt load balance and vice versa.

To deeply analyze this trade-off, we plotted the behavior of each worker for four Hercules configurations from the last studied case (32 nodes in the right-hand side of Fig. 14). Figs. 17–20 show the time spent by each worker computing or performing I/O (in light green) and in idle states (in dark red).

Fig. 17 displays the behavior for the default configuration. It shows a good load balance, with some idleness appearing in the final stages of the workflow. In this case the I/O performance could be improved by exploiting the data locality.

Fig. 18 shows the behavior for strictly enforcing data locality and using the default data placement. As Fig. 14 shows, the time needed by each worker to complete its assigned tasks is substantially reduced, resulting in the best I/O performance. Nevertheless, the total execution time is the worst of the improved data-locality cases. As Fig. 18 shows, forcing data locality without proper data placement means that some workers remain underutilized (24.48% of idle time), while some tasks are still pending because the data are in another node, thus increasing the total execution time.

The behavior of best-effort data-locality task placement with default data-placement is depicted in Fig. 19. In this configuration the data locality is not fully exploited. However, the flexibility of the best-effort strategy allows the scheduler to reduce the idleness (5.18% idle time) and to achieve a reasonable load balance.

The strictly enforced data locality task placement and balanced data distribution shown in Fig. 20 achieves the best total execution time, mainly because of two factors: best I/O performance, due to maximum locality exploitation, and best CPU utilization (95.31%), due to the good load balance achieved by the scheduler.

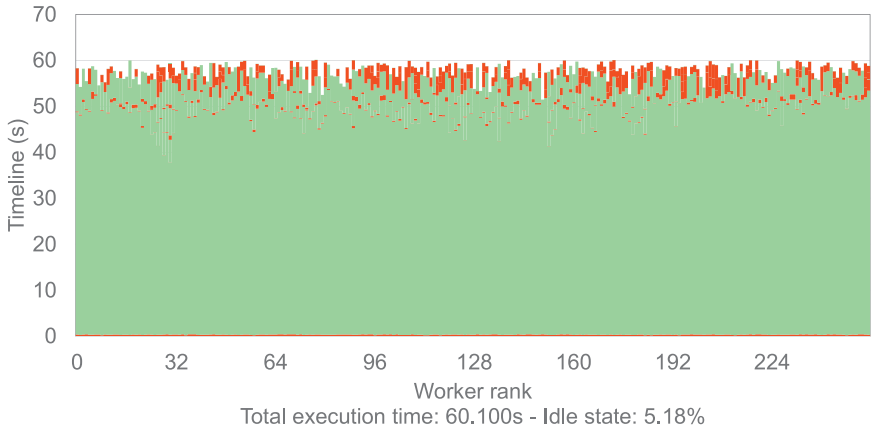

Fig. 19. Timeline in seconds for each worker, using default data-placement and best-effort locality-aware scheduling (SOFT, NODE). Execution of the MapReduce-like application with 256 jobs (767 tasks) running on 32 nodes (256 workers). Light green represents busy status; dark red represents idle/wait status. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

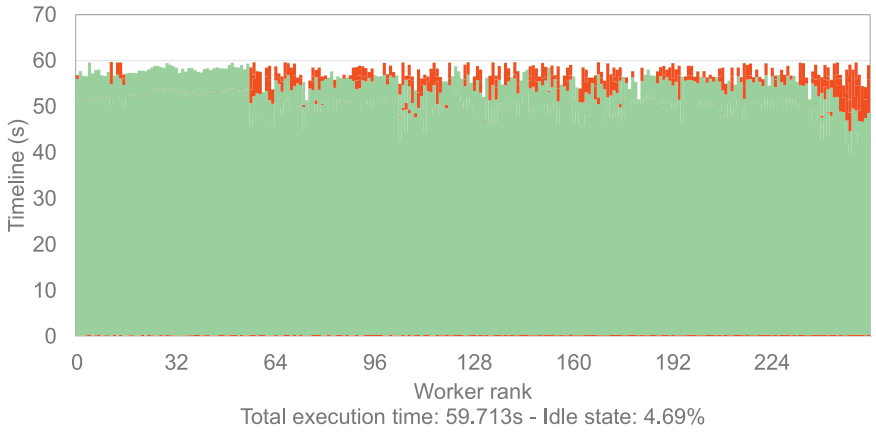

Fig. 20. Timeline in seconds for each worker, using informed data placement and locality-aware scheduling (HARD, NODE). Execution of the MapReduce-like application with 256 jobs (767 tasks) running on 32 nodes (256 workers). Light green represents busy status; dark red represents idle/wait status. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

5. Related work

This section presents related work from three areas: scientific workflow systems, in-memory distributed stores for scaling up storage systems, and workflow-aware storage systems.

5.1. Scientific workflow systems

The Pegasus framework can be used to map complex scientific workflows onto distributed resources. In a similar approach to Swift/T, Pegasus relies on scripting languages (Perl and Python) for defining workflows as a graph of data-dependent tasks. Pegasus can be deployed in multiple execution environments, such as clouds, HPC clusters, and grids. However, the workflow engine relies on a shared filesystem for storing intermediate files; and this approach can limit the processing rate of workflows at large scale, since the existing file systems currently represent a bottleneck for large-scale concurrent files given that workflow structure and locality awareness are not properly exploited.

Another approach for deploying scientific workflow is OmpSs [23]. This solution offers asynchronous parallelism in the form of tasks. OmpSs allows users to annotate function declarations with a task directive. Any call to an OmpSs function creates a new task that will execute the function body. The COMP Superscalar programming framework (COMPs) [24] permits the programming of scientific applications and their execution on a variety of distributed infrastructures such as clouds.

5.2. In-memory distributed stores for scaling up storage systems

Besides Memcached, Redis, and Hercules systems described earlier in this paper, other works target scaling up storage system through in-memory distributed stores.

Parrot [25] is a tool for attaching existing programs to remote I/O systems through the POSIX file system interface, and Chirp [26] is a user-level file system for collaboration across distributed systems such as clusters, clouds, and grids. They are usually combined in order to easily deploy a distributed file system ready to use with current applications through a POSIX API. Many characteristics are shared with Hercules: user-level deployment without any special privileges, transparency through the use of a widely used interface, and easy deployment using a simple command to start a new server. Hercules, however, is designed to achieve high scalability and performance by taking advantage of as many compute nodes as possible for I/O operations. Hercules uses main memory for storage improving performance in data-locality-aware accesses.

AHPIOS [27] is a fully scalable ad hoc I/O parallel I/O system for MPI applications. AHPIOS relies on dynamic partitions and elastic on-demand partitions for the deployment of distributed applications. It provides several memory cache levels. Hercules shares some of its features: (1) user-level deployment without special privileges; (2) transparency, providing wide and easy deployment by using simple commands; (3) leveraging of as many compute nodes as possible for I/O nodes in order achieve high scalability and performance; and (4) use of main memory for temporal storage in order to improve performance in accesses.

HyCache+ [28] is a distributed storage middleware system for HPC systems. It acts as main storage of recently accessed data (metadata, intermediate results for the analysis of large-scale data, etcthe and exchanges data asynchronously with the remote file system. Similarities between HyCache+ and Hercules include the fully distributed metadata approach, the use of the high-performance computing network for data accesses rather than the dedicated shared storage network, and the high scalability objective. HyCache+ presents a POSIX interface to its service, however, whereas Hercules offers a put/get API, a POSIX-like API, and an MPI-IO API. HyCache+ also focuses on improving parallel file systems, whereas Hercules is designed to accelerate workflow execution engines, facilitating the exploitation of data locality in current many-task applications deployed on cloud infrastructures.

Another related project is ElastiCache from Amazon [29]. This service allows deployment of Amazons virtual machines on which either Memcached or Redis is automatically instantiated. Thus, ElastiCache can be used as back-end for Hercules. ElastiCache is accessible only by a put/get API, however, so it does not offer a file abstraction, data locality management, and workflow support.

5.3. Workflow-oriented storage systems

Costa et al. [30,31] propose MosaStore, a versatile storage system that uses attributes on files as a method of passing hints between the workflow engine and the file system. Alternatively, the file system can infer patterns of access by analyzing access patterns over time. Hercules, in contrast, relies on codesigning with the workflow framework in order to expose and exploit data locality. Another important difference is that MosaStore is less scalable than is Hercules, because it uses a centralized metadata server that could become a bottleneck in large-scale systems.

The Data Mining Cloud Framework (DMCF) [13] is a software system 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 cloud platforms, following a Software-as-a-Service approach. DMCF relies on the default storage of the public cloud provider. This implies that the I/O performance of DMCF is limited by the performance of the default storage. In a previous work, we demonstrated that DMCF could be accelerated by Hercules [32].

6. Conclusions

This paper presents and evaluates a novel solution for improving I/O performance of workflows running on both HPC and cloud platforms. We make the case for the codesign of the storage system and workflow management system. To demonstrate this idea, we propose a novel design that integrates a data store and a workflow management system in four main aspects: workflow representation, task scheduling, task placement, and task termination. Extensive evaluation on both cloud and HPC systems demonstrates that our solution makes better use of resources than do existing state-of-the-art approaches and thus provides better performance and scalability in most cases.

Future work includes cost analysis of our solution on both cloud and HPC environments. Furthermore, we plan to evaluate more complex data access patterns, including real workflow applications.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. This work also has been partially funded by the grant TIN2013-41350-P from the Spanish Ministry of Economy and Competitiveness. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 328582. We gratefully acknowledge the computing resources provided on Fusion, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

References

[1] M. Cannataro, D. Talia, P.K. Srimani, Parallel data intensive computing in scientific and commercial applications, Parallel Comput. 28 (5) (2002) 673–704.
[2] P. J. Braam., The lustre storage architecture, Cluster File Systems, Inc., 2004. URL http://www.lustre.org/documentation.html.
[3] F.B. Schmuck, R.L. Haskin, GPFS: a shared-disk file system for large computing clusters.
[4] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop distributed file system, in: IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST 2010), IEEE, 2010, pp. 1–10.
[5] M. Woitaszek, J.M. Dennis, T.R. Sines, Parallel high-resolution climate data analysis using Swift, in: Proc. MTAGS at SC, 2011.
[6] J.M. Wozniak, K. Chard, B. Blaiszik, R. Osborn, M. Wilde, I. Foster, Big data remote access interfaces for light source science, in: Proc. Big Data Computing, 2015.
[7] A. O'Driscoll, J. Daugelaite, R.D. Sleator, Big data, Hadoop and cloud computing in genomics, J. Biomed. Inf. 46 (5) (2013) 774–781.
[8] D.L. Jones, K. Wagstaff, D.R. Thompson, L.D. Addario, R. Navarro, C. Mattmann, W. Majid, J. Lazio, R. Preston, U. Rebbapragada, Big data challenges for large radio arrays, in: Proc. IEEE Aerospace Conference, 2012.
[9] J. Chen, A. Choudhary, S. Feldman, B. Hendrickson, C. Johnson, R. Mount, V. Sarkar, V. White, D. Williams, Synergistic challenges in data-intensive science and exascale computing, in: DOE ASCAC Data Subcommittee Report, Department of Energy Office of Science, 2013, p. 64. URL http://science.energy.gov/~/media/ascr/ascac/pdf/reports/2013/ASCAC_Data_Intensive_Computing_report_final.pdf.
[10] D.A. Reed, J. Dongarra, Exascale computing and big data, Commun. ACM 58 (7) (2015) 56–68, doi:10.1145/2699414.
[11] J. C., et al., Memorandum of understanding, in: Network for Sustainable Ultrascale Computing (NESUS), 2014, p. 30. URL http://w3.cost.eu/fileadmin/domain_files/ICT/Action_IC1305/mou/IC1305-e.pdf.
[12] J. Wozniak, T. Armstrong, M. Wilde, D. Katz, E. Lusk, I. Foster, Swift/t: large-scale application composition via distributed-memory dataflow processing, in: 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2013), 2013, pp. 95–102, doi:10.1109/CCGrid.2013.99.
[13] F. Marozzo, D. Talia, P. Trunfio, JS4cloud: script-based workflow programming for scalable data analysis on cloud platforms, Concurrency Comput. doi:10.1002/cpe.3563.
[14] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P.J. Maechling, R. Mayani, W. Chen, R.F.d. Silva, M. Livny, K. Wenger, Pegasus, a workflow management system for science automation, Future Gener. Comput. Syst. 46 (C) (2015) 17–35, doi:10.1016/j.future.2014.10.008.
[15] FUSE-based file system backed by amazon s3 github, 2014. https://github.com/s3fs-fuse/s3fs-fuse.
[16] B. Fitzpatrick, Distributed caching with Memcached, Linux J. 2004 (124) (2004) 5. URL http://dl.acm.org/citation.cfm?id=1012889.1012894.
[17] libKetama: ketama consistent hashing library git repository, 2009. https://github.com/RJ/ketama.
[18] vBuckets: the core enabling mechanism for couchbase server data distribution.
[19] T. Macedo, F. Oliveira, Redis Cookbook, O'Reilly Media, Sebastopol, 2011.
[20] F.R. Duro, J.G. Blas, J. 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, ACM, New York, NY, USA, 2013, pp. 139–140, doi:10.1145/2488551.2488598.
[21] H.C. Baker Jr., C. Hewitt, The incremental garbage collection of processes, 12, ACM, New York, NY, USA, 1977, pp. 55–59, doi:10.1145/872734.806932.
[22] M. Wilde, M. Hategan, J.M. Wozniak, B. Clifford, D.S. Katz, I. Foster, Swift: a language for distributed parallel scripting, Parallel Comput. 37 (9) (2011) 633–652.
[23] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R.M. Badia, E. Ayguade, J. Labarta, Euro-par 2011, bordeaux, france, august 29 - september 2, 2011, proceedings, part i, springer berlin heidelberg, berlin, heidelberg, in: Ch. Productive Cluster Programming with OmpSs, 2011, pp. 555–566.
[24] E. Tejedor, R.M. Badia, COMP superscalar: bringing GRID superscalar and GCM together, in: 8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'08)., IEEE, 2008, pp. 185–193.
[25] D. Thain, M. Livny, Parrot: transparent user-level middleware for data-intensive computing, Scalable Comput. 6 (3) (2005) 9–18.
[26] D. Thain, C. Moretti, J. Hemmes, Chirp: a practical global filesystem for cluster and grid computing, J. Grid Comput. 7 (1) (2009) 51–72, doi:10.1007/s10723-008-9100-5.
[27] F. Isaila, F.J.G. Blas, J. Carretero, W.-K. Liao, A. Choudhary, A scalable message passing interface implementation of an ad-hoc parallel I/O system, Int. J. High Perform. Comput. Appl. 24 (2) (2010) 164–184, doi:10.1177/1094342009347890.
[28] D. Zhao, K. Qiao, I. Raicu, HyCache+: towards scalable high-performance caching middleware for parallel file systems, 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2014), 2014.
[29] A. Raghavan, A. Chandra, J.B. Weissman, Tiera: towards flexible multi-tiered cloud storage instances, in: Proceedings of the 15th International Middleware Conference, Middleware '14, ACM, New York, NY, USA, 2014, pp. 1–12.
[30] S. Al-Kiswany, A. Gharaibeh, M. Ripeanu, The case for a versatile storage system, Oper. Syst. Rev. 44 (1) (2010) 10–14.
[31] L. Costa, H. Yang, E. Vairavanathan, A. Barros, K. Maheshwari, G. Fedak, D. Katz, M. Wilde, M. Ripeanu, S. Al-Kiswany, The case for workflow-aware storage: an opportunity study, J. Grid Comput. (2014) 1–19, doi:10.1007/s10723-014-9307-6.
[32] F.J.R. Duro, F. Marozzo, J.G. Blas, J.C. Pérez, D. Talia, P. Trunfio, Evaluating data caching techniques in DMCF workflows using Hercules, in: Proceedings of the Second International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2015), 2015.